

# Computer Architecture and Organization: From Software to Hardware

Manoj Franklin  
University of Maryland, College Park

©Manoj Franklin 2007

# Preface

## Introduction

*Welcome! Bienvenidos! Bienvenue! Benvenuto!* This book provides a fresh introduction to computer architecture and organization. The subject of computer architecture dates back to the early periods of computer development, although the term was coined more recently. Over the years many introductory books have been written on this fascinating subject, as the subject underwent many changes due to technological (hardware) and application (software) changes. Today computer architecture is a topic of great importance to computer science, computer engineering, and electrical engineering. It bridges the yawning gap between high-level language programming in computer science and VLSI design in electrical engineering. The spheres of influence exercised by computer architecture has expanded significantly in recent years. A fresh introduction of the subject is therefore essential for modern computer users, programmers, and designers.

This book is for students of computer science, computer engineering, electrical engineering, and any others who are interested in learning the fundamentals of computer architecture in a structured manner. It contains core material that is essential to students in all of these disciplines. It is designed for use in a computer architecture or computer organization course typically offered at the undergraduate level by computer science, computer engineering, electrical engineering, or information systems departments. On successful completion of this book you will have a clear understanding of the foundational principles of computer architecture. Many of you may have taken a course in high-level language programming and in digital logic before using this book. We assume most readers will have some familiarity with computers, and perhaps have even done some programming in a high-level language. We also assume that readers have had exposure to preliminary digital logic design. This book will extend that knowledge to the core areas of computer architecture, namely assembly-level architecture, instruction set architecture, and microarchitecture.

The WordReference dictionary defines computer architecture as “the structure and organization of a computer’s hardware or system software.” Dictionary.com defines it as “the art of assembling logical elements into a computing device; the specification of the relation between parts of a computer system.” Computer architecture deals with the way in which the elements of a computer relate to each other. It is concerned with all aspects of the design and operation of the computer system. It extends upward into software as a system’s architecture is intimately tied to the operating system and other system software. It is almost impossible to design a good operating system without knowing the underlying architecture of the systems where the operating system will run. Similarly, the compiler requires an even more intimate knowledge of the architecture.

It is important to understand the general principles behind the design of computers, and to see how those principles are put into practice in real computers. The goal of this book is

to provide a complete discussion of the fundamental concepts, along with an extensive set of examples that reinforce these concepts. A few detailed examples are also given for the students to have a better appreciation of real-life intricacies. These examples are presented in a manner that does not distract the student from the fundamental concepts. Clearly, we cannot cover every single aspect of computer architecture in an introductory book. Our goal is to cover the fundamentals and to lay a good foundation upon which motivated students can easily build later. For each topic, we use the following test to decide if it should get included in the text: is the topic foundational? If the answer is positive, we include the topic.

Almost every aspect of computer architecture is replete with trade-offs, involving characteristics such as programmability, software compatibility, portability, speed, cost, power consumption, die size, and reliability. For general-purpose computers, one trade-off drives the most important choices the computer architect must make: speed versus cost. For laptops and embedded systems, the important considerations are size and power consumption. For space applications and other mission-critical applications, reliability and power consumption are of primary concern. Among these considerations, we highlight programmability, performance, cost, and power consumption throughout the text, as they are fundamental factors affecting how a computer is designed. However, this coverage is somewhat qualitative, and not intended to be quantitative in nature. Extensive coverage of quantitative analysis is traded off in favor of qualitative explanation of issues. Students will have plenty of opportunity to study quantitative analysis in a graduate-level computer architecture course. Additional emphasis is also placed on how various parts of the system are related to real-world demand and technology constraints.

Performance and functionality are key to the utility of a computer system. Perhaps one of the most important reasons for studying computer architecture is to learn how to extract the best performance from a computer. As an assembly language programmer, for instance, you need to understand how to use the system's functionality most effectively. Specifically, you must understand its architecture so that you will be able to exploit that architecture during programming.

## **Coverage of Software and Hardware**

Computer architecture/organization is a discipline with many facets, ranging from translation of high-level language programs through design of instruction set architecture and microarchitecture to the logic-level design of computers. Some of these facets have more of a software luster whereas others have more of a hardware luster. We believe that a good introduction to the discipline should give a broad overview of all the facets and their interrelationships, leaving a non-specialist with a decent perspective on computer architecture, and providing an undergraduate student with a solid foundation upon which related and advanced subjects can be built. Traditional introductory textbooks focussing only on software topics or on hardware topics do not fulfill these objectives.

Our presentation is unique in that we cover both software and hardware concepts. These include high-level language, assembly language programming, systems programming, instruction set architecture design, microarchitecture design, system design, and digital logic design.

There are four legs that form the foundation of computer architecture: assembly-level architecture, instruction set architecture, microarchitecture, and logic-level architecture. This book is uniquely concerned about all four legs. Starting from the assembly-level architecture, we carry out the design of the important portions of a computer system all the way to the lower hardware levels, considering plausible alternatives at each level.

## **Structured Approach**

In an effort to systematically cover all of these fundamental topics, the material has been organized in a structured manner, from the high-level architecture to the logic-level architecture. Our coverage begins with a high-level language programmer's view—expressing algorithms in an HLL such as C—and moves towards the less abstract levels. Although there are a few textbooks that start from the digital logic level and work their way towards the more abstract levels, in our view the fundamental issues of computer architecture/organization are best learned starting with the software levels, with which most of the students are already familiar. Moreover, it is easier to appreciate why a level is designed in a particular manner if the student knows what the design is supposed to implement. This structured approach—from abstract software levels to less abstract software levels to abstract hardware levels to less abstract hardware levels—is faithfully followed throughout the book. We make exceptions only in a few places where such a deviation tends to improve clarity. For example, while discussing ISA (instruction set architecture) design options in Chapter 5, we allude to hardware issues such as pipelining and multiple-issue, which influence ISA design.

For each architecture level we answer the following fundamental questions: What is the nature of the machine at this level? What are the ways in which its building blocks interact? How does the machine interact with the outside world? How is programming done at this level? How is a higher-level program translated/interpreted for controlling the machine at this level? We are confident that after you have mastered these fundamental concepts, building upon them will be quite straightforward.

## **Example Instruction Set**

As an important goal of this book is to lay a good foundation for the general subject of computer architecture, we have refrained from focusing on a single architecture in our discussion of the fundamental concepts. Thus, when presenting concepts at each architecture level, great care is taken to keep the discussion general, without tailoring to a specific architecture. For instance, when discussing the assembly language architecture, we discuss register-based

approach as well as a stack-based approach. When discussing virtual memory, we discuss a paging-based approach as well as a segmentation-based approach. In other words, at each stage of the design, we discuss alternative approaches, and the associated trade-offs. While one alternative may seem better today, technological innovations may tip the scale towards another in the future.

For ease of learning, the discussion of concepts is peppered with suitable examples. We have found that students learn the different levels and their inter-relationships better when there is a continuity among many of the examples used in different parts of the book. For this purpose, we have used the standard MIPS assembly language [ref] and the standard MIPS Lite instruction set architecture [ref], a subset of the MIPS-I ISA [ref]. We use MIPS because it is very simple and has had commercial success, both in general-purpose computing and in embedded systems. The MIPS architecture had its beginnings in 1984, and was first implemented in 1985. By the late 1980s, the architecture had been adopted by several workstation and server companies, including Digital Equipment Corporation and Silicon Graphics. Now MIPS processors are widely used in Sony and Nintendo game machines, palmtops, laser printers, Cisco routers, and SGI high-performance graphics engines. More importantly, some popular texts on Advanced Computer Architecture use the MIPS architecture. The use of the MIPS instruction set in this introductory book will therefore provide good continuity for those students wishing to pursue higher studies in Computer Science or Engineering.

In rare occasions, I have changed some terminology, not to protect the innocent but simply to make it clearer to understand.

## Organization and Usage of the Book

This book is organized to meet the needs of several potential audiences. It can serve as an undergraduate text, as well as a professional reference for engineers and members of the technical community who find themselves frequently dealing with computing. The book uses a structured approach, and is intended to be read sequentially. Each chapter builds upon the previous ones. Certain sections contain somewhat advanced technical material, and can be skipped by the reader without loss in continuity. These sections are marked with an asterisk. We recommend, however, that even those sections be skimmed, at least to get a superficial idea of their contents.

Each chapter is followed by a “Concluding Remarks” section and an “Exercises” section. The exercises are particularly important. They help master the material by integrating a number of different concepts. The book also includes many real-world examples, both historical and current, in each chapter. Instead of presenting real-world examples in isolation, such examples are included while presenting the major concepts.

This book is organized into 9 chapters, which are grouped into 3 parts. The first part provides an overview of the subject. The second part covers the software levels, and the third part covers the hardware levels. The coverage of the software levels is not intended

to make the readers proficient in programming in these levels, but rather to help them understand what each level does, how programs at immediately higher level are converted to this level, and how to design this level in a better way.

A layered approach is used to cover the topics. Each new layer builds upon the previous material to add depth and understanding to the reader's knowledge.

Chapter 1 provides an overview of .... It opens with a discussion of the expanding role of computers, and the trends in technology and software applications. It briefly introduces ..... Chapter 2 ... Chapter 3 ..... Most of the material in Chapter 3 should be familiar to readers with a background in computer programming, and they can probably browse through this chapter. Starting with Chapter 4, the material deals with the core issues in computer architecture. Chapter 4 ..... Chapter 5 ..... Chapter 6 .....

The book can be tailored for use in software-centric as well as hardware-centric courses. For instance, skipping the last chapter (or the last 3 chapters) makes the book becomes suitable for a software-centric course, and skipping chapter 2 makes it suitable for a hardware-centric course.

|  |
|--|
| <p><i>“If you are planning for a year, sow rice;<br/>if you are planning for a decade, plant trees;<br/>if you are planning for a lifetime, educate people.”</i><br/>— Chinese Proverb</p> |
|--|

|  |
|--|
| <p><i>“Therefore, since brevity is the soul of wit, And tediousness the limbs and outward flourishes, I will be brief”</i><br/>— William Shakespeare, Hamlet</p> |
|--|

*Soli Deo Gloria*

# Contents

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>Introduction</b>   | <b>1</b> |
| 1.1      | Computing and Computers . . . . .   | 3        |
| 1.1.1    | The Problem-Solving Process . . . . .   | 3        |
| 1.1.2    | Automating Algorithm Execution with Computers . . . . .                           | 5        |
| 1.2      | The Digital Computer . . . . .  | 9        |
| 1.2.1    | Representing Programs in a Digital Computer: The Stored Program Concept . . . . . | 10       |
| 1.2.2    | Basic Software Organization . . . . .   | 12       |
| 1.2.3    | Basic Hardware Organization . . . . .   | 13       |
| 1.2.4    | Software versus Hardware . . . . .  | 15       |
| 1.2.5    | Computer Platforms . . . . .  | 16       |
| 1.3      | A Modern Computer System . . . . .  | 17       |
| 1.3.1    | Hardware . . . . .  | 17       |
| 1.3.2    | Software . . . . .  | 19       |
| 1.3.3    | Starting the Computer System: The Boot Process . . . . .                          | 20       |
| 1.3.4    | Computer Network . . . . .  | 21       |
| 1.4      | Trends in Computing . . . . .   | 22       |
| 1.4.1    | Hardware Technology Trends . . . . .  | 22       |
| 1.4.2    | Software Technology Trends . . . . .  | 23       |
| 1.5      | Software Design Issues . . . . .  | 25       |
| 1.6      | Hardware Design Issues . . . . .  | 25       |
| 1.6.1    | Performance . . . . .   | 25       |
| 1.6.2    | Power Consumption . . . . .   | 26       |
| 1.6.3    | Price . . . . .   | 27       |



|          |   |           |
|----------|---|-----------|
| 1.6.4    | Size . . . . .  | 27        |
| 1.6.5    | Summary . . . . .   | 27        |
| 1.7      | Theoretical Underpinnings . . . . .                             | 27        |
| 1.7.1    | Computability and the Turing Machine . . . . .                  | 27        |
| 1.7.2    | Limitations of Computers . . . . .                              | 28        |
| 1.8      | Virtual Machines: The Abstraction Tower . . . . .               | 30        |
| 1.8.1    | Problem Definition and Modeling Level Architecture . . . . .    | 33        |
| 1.8.2    | Algorithm-Level Architecture . . . . .                          | 33        |
| 1.8.3    | High-Level Architecture . . . . .                               | 37        |
| 1.8.4    | Assembly-Level Architecture . . . . .                           | 38        |
| 1.8.5    | Instruction Set Architecture (ISA) . . . . .                    | 38        |
| 1.8.6    | Microarchitecture . . . . .                                     | 39        |
| 1.8.7    | Logic-Level Architecture . . . . .                              | 39        |
| 1.8.8    | Device-Level Architecture . . . . .                             | 40        |
| 1.9      | Concluding Remarks . . . . .                                    | 41        |
| 1.10     | Exercises . . . . .   | 41        |
| <b>I</b> | <b>PROGRAM DEVELOPMENT — SOFTWARE LEVELS</b>                    | <b>43</b> |
| <b>2</b> | <b>Program Development Basics</b>                               | <b>45</b> |
| 2.1      | Overview of Program Development . . . . .                       | 46        |
| 2.1.1    | Programming Languages . . . . .                                 | 47        |
| 2.1.2    | Application Programming Interface Provided by Library . . . . . | 50        |
| 2.1.3    | Application Programming Interface Provided by OS . . . . .      | 50        |
| 2.1.4    | Compilation . . . . .   | 50        |
| 2.1.5    | Debugging . . . . .   | 50        |
| 2.2      | Programming Language Specification . . . . .                    | 50        |
| 2.2.1    | Syntax . . . . .  | 50        |
| 2.2.2    | Semantics . . . . .   | 50        |
| 2.3      | Data Abstraction . . . . .                                      | 50        |
| 2.3.1    | Constants . . . . .   | 51        |
| 2.3.2    | Variables . . . . .   | 52        |
| 2.3.3    | IO Streams and Files . . . . .                                  | 58        |

|          |   |           |
|----------|---|-----------|
| 2.3.4    | Data Structures . . . . .                                   | 60        |
| 2.3.5    | Modeling Real-World Data . . . . .                          | 60        |
| 2.4      | Operators and Assignments . . . . .                         | 64        |
| 2.5      | Control Abstraction . . . . .                               | 65        |
| 2.5.1    | Conditional Statements . . . . .                            | 65        |
| 2.5.2    | Loops . . . . .   | 66        |
| 2.5.3    | Subroutines . . . . .                                       | 67        |
| 2.5.4    | Subroutine Nesting and Recursion . . . . .                  | 68        |
| 2.5.5    | Re-entrant Subroutine . . . . .                             | 68        |
| 2.5.6    | Program Modules . . . . .                                   | 69        |
| 2.5.7    | Software Interfaces: API and ABI . . . . .                  | 69        |
| 2.6      | Library API . . . . .                                       | 69        |
| 2.7      | Operating System API . . . . .                              | 70        |
| 2.7.1    | What Should be Done by the OS? . . . . .                    | 72        |
| 2.7.2    | Input/Output Management . . . . .                           | 72        |
| 2.7.3    | Memory Management . . . . .                                 | 73        |
| 2.7.4    | Process Management . . . . .                                | 74        |
| 2.8      | Operating System Organization . . . . .                     | 74        |
| 2.8.1    | System Call Interface . . . . .                             | 76        |
| 2.8.2    | File System . . . . .                                       | 76        |
| 2.8.3    | Device Management: Device Drivers . . . . .                 | 77        |
| 2.8.4    | Hardware Abstraction Layer (HAL) . . . . .                  | 78        |
| 2.8.5    | Process Control System . . . . .                            | 78        |
| 2.9      | Major Issues in Program Development . . . . .               | 80        |
| 2.9.1    | Portability . . . . .                                       | 80        |
| 2.9.2    | Reusability . . . . .                                       | 80        |
| 2.9.3    | Concurrency . . . . .                                       | 80        |
| 2.10     | Concluding Remarks . . . . .                                | 80        |
| 2.11     | Exercises . . . . .   | 80        |
| <b>3</b> | <b>Assembly-Level Architecture — User Mode</b>              | <b>81</b> |
| 3.1      | Overview of User Mode Assembly-Level Architecture . . . . . | 82        |
| 3.1.1    | Assembly Language Alphabet and Syntax . . . . .             | 83        |

|       |   |     |
|-------|---|-----|
| 3.1.2 | Memory Model . . . . .                                    | 83  |
| 3.1.3 | Register Model . . . . .                                  | 85  |
| 3.1.4 | Data Types . . . . .                                      | 87  |
| 3.1.5 | Assembler Directives . . . . .                            | 89  |
| 3.1.6 | Instruction Types and Instruction Set . . . . .           | 90  |
| 3.1.7 | Program Execution . . . . .                               | 93  |
| 3.1.8 | Challenges of Assembly Language Programming . . . . .     | 94  |
| 3.1.9 | The Rationale for Assembly Language Programming . . . . . | 95  |
| 3.2   | Assembly-Level Interfaces . . . . .                       | 96  |
| 3.2.1 | Assembly-Level Interface Provided by Library . . . . .    | 97  |
| 3.2.2 | Assembly-Level Interface Provided by OS . . . . .         | 97  |
| 3.3   | Example Assembly-Level Architecture: MIPS-I . . . . .     | 97  |
| 3.3.1 | Assembly Language Alphabet and Syntax . . . . .           | 97  |
| 3.3.2 | Register Model . . . . .                                  | 98  |
| 3.3.3 | Memory Model . . . . .                                    | 101 |
| 3.3.4 | Assembler Directives . . . . .                            | 102 |
| 3.3.5 | Assembly-Level Instructions . . . . .                     | 103 |
| 3.3.6 | An Example MIPS-I AL Program . . . . .                    | 104 |
| 3.3.7 | SPIM: A Simulator for the MIPS-I Architecture . . . . .   | 107 |
| 3.4   | Translating HLL Programs to AL Programs . . . . .         | 107 |
| 3.4.1 | Translating Constant Declarations . . . . .               | 108 |
| 3.4.2 | Translating Variable Declarations . . . . .               | 110 |
| 3.4.3 | Translating Variable References . . . . .                 | 118 |
| 3.4.4 | Translating Conditional Statements . . . . .              | 119 |
| 3.4.5 | Translating Loops . . . . .                               | 122 |
| 3.4.6 | Translating Subroutine Calls and Returns . . . . .        | 123 |
| 3.4.7 | Translating System Calls . . . . .                        | 127 |
| 3.4.8 | Overview of a Compiler . . . . .                          | 128 |
| 3.5   | Memory Models: Design Choices . . . . .                   | 129 |
| 3.5.1 | Address Space: Linear vs Segmented . . . . .              | 129 |
| 3.5.2 | Word Alignment: Aligned vs Unaligned . . . . .            | 130 |
| 3.5.3 | Byte Ordering: Little Endian vs Big Endian . . . . .      | 131 |
| 3.6   | Operand Locations: Design Choices . . . . .               | 131 |

|          |  |            |
|----------|--|------------|
| 3.6.1    | Instruction . . . . .  | 131        |
| 3.6.2    | Main Memory . . . . .  | 131        |
| 3.6.3    | General-Purpose Registers . . . . .                                  | 132        |
| 3.6.4    | Accumulator . . . . .  | 132        |
| 3.6.5    | Operand Stack . . . . .  | 133        |
| 3.7      | Operand Addressing Modes: Design Choices . . . . .                   | 136        |
| 3.7.1    | Instruction-Residing Operands: Immediate Operands . . . . .          | 137        |
| 3.7.2    | Register Operands . . . . .  | 137        |
| 3.7.3    | Memory Operands . . . . .  | 138        |
| 3.7.4    | Stack Operands . . . . .   | 140        |
| 3.8      | Subroutine Implementation . . . . .                                  | 141        |
| 3.8.1    | Register Saving and Restoring . . . . .                              | 142        |
| 3.8.2    | Return Address Storing . . . . .                                     | 143        |
| 3.8.3    | Parameter Passing and Return Value Passing . . . . .                 | 145        |
| 3.9      | Defining Assembly Languages for Programmability . . . . .            | 146        |
| 3.9.1    | Labels . . . . .   | 146        |
| 3.9.2    | Pseudoinstructions . . . . .   | 146        |
| 3.9.3    | Macros . . . . .   | 146        |
| 3.10     | Concluding Remarks . . . . .   | 147        |
| 3.11     | Exercises . . . . .  | 147        |
| <b>4</b> | <b>Assembly-Level Architecture — Kernel Mode</b>                     | <b>149</b> |
| 4.1      | Overview of Kernel Mode Assembly-Level Architecture . . . . .        | 150        |
| 4.1.1    | Privileged Registers . . . . .                                       | 151        |
| 4.1.2    | Privileged Memory Address Space . . . . .                            | 152        |
| 4.1.3    | IO Addresses . . . . .   | 152        |
| 4.1.4    | Privileged Instructions . . . . .                                    | 152        |
| 4.2      | Switching from User Mode to Kernel Mode . . . . .                    | 153        |
| 4.2.1    | Syscall Instructions: Switching Initiated by User Programs . . . . . | 154        |
| 4.2.2    | Device Interrupts: Switching Initiated by IO Interfaces . . . . .    | 156        |
| 4.2.3    | Exceptions: Switching Initiated by Rare Events . . . . .             | 157        |
| 4.3      | IO Registers . . . . .   | 158        |
| 4.3.1    | Memory Mapped IO Address Space . . . . .                             | 158        |

|          |   |            |
|----------|---|------------|
| 4.3.2    | Independent IO Address Space . . . . .                              | 158        |
| 4.3.3    | Operating System's Use of IO Addresses . . . . .                    | 160        |
| 4.4      | Operating System Organization . . . . .                             | 162        |
| 4.4.1    | System Call Layer . . . . .   | 164        |
| 4.4.2    | File System . . . . .   | 164        |
| 4.4.3    | Device Management: Device Drivers . . . . .                         | 165        |
| 4.4.4    | Process Control System . . . . .                                    | 167        |
| 4.5      | System Call Layer for a MIPS-I OS . . . . .                         | 168        |
| 4.5.1    | MIPS-I Machine Specifications for Exceptions . . . . .              | 168        |
| 4.5.2    | OS Usage of MIPS-I Architecture Specifications . . . . .            | 170        |
| 4.6      | IO Schemes Employed by Device Management System . . . . .           | 173        |
| 4.6.1    | Sampling-Based IO . . . . .   | 173        |
| 4.6.2    | Program-Controlled IO . . . . .                                     | 174        |
| 4.6.3    | Interrupt-Driven IO . . . . .                                       | 177        |
| 4.6.4    | Direct Memory Access (DMA) . . . . .                                | 181        |
| 4.6.5    | IO Co-processing . . . . .  | 182        |
| 4.6.6    | Wrap Up . . . . .   | 183        |
| 4.7      | Concluding Remarks . . . . .  | 183        |
| 4.8      | Exercises . . . . .   | 183        |
| <b>5</b> | <b>Instruction Set Architecture (ISA)</b>                           | <b>185</b> |
| 5.1      | Overview of Instruction Set Architecture . . . . .                  | 186        |
| 5.1.1    | Machine Language . . . . .  | 186        |
| 5.1.2    | Register, Memory, and IO Models . . . . .                           | 189        |
| 5.1.3    | Data Types and Formats . . . . .                                    | 189        |
| 5.1.4    | Instruction Types and Formats . . . . .                             | 189        |
| 5.2      | Example Instruction Set Architecture: MIPS-I . . . . .              | 190        |
| 5.2.1    | Register, Memory, and IO Models . . . . .                           | 190        |
| 5.2.2    | Data Types and Formats . . . . .                                    | 190        |
| 5.2.3    | Instruction Types and Formats . . . . .                             | 190        |
| 5.2.4    | An Example MIPS-I ML Program . . . . .                              | 191        |
| 5.3      | Translating Assembly Language Programs to Machine Language Programs | 191        |
| 5.3.1    | MIPS-I Assembler Conventions . . . . .                              | 192        |

|       |   |     |
|-------|---|-----|
| 5.3.2 | Translating Decimal Numbers . . . . .                               | 192 |
| 5.3.3 | Translating AL-specific Instructions and Macros . . . . .           | 192 |
| 5.3.4 | Translating Labels . . . . .  | 193 |
| 5.3.5 | Code Generation . . . . .   | 195 |
| 5.3.6 | Overview of an Assembler . . . . .                                  | 196 |
| 5.3.7 | Cross Assemblers . . . . .  | 196 |
| 5.4   | Linking . . . . .   | 197 |
| 5.4.1 | Resolving External References . . . . .                             | 198 |
| 5.4.2 | Relocating the Memory Addresses . . . . .                           | 198 |
| 5.4.3 | Program Start-Up Routine . . . . .                                  | 198 |
| 5.5   | Instruction Formats: Design Choices . . . . .                       | 199 |
| 5.5.1 | Fixed Length Instruction Encoding . . . . .                         | 201 |
| 5.5.2 | Variable Length Instruction Encoding . . . . .                      | 203 |
| 5.6   | Data Formats: Design Choices and Standards . . . . .                | 203 |
| 5.6.1 | Unsigned Integers: Binary Number System . . . . .                   | 204 |
| 5.6.2 | Signed Integers: 2's Complement Number System . . . . .             | 205 |
| 5.6.3 | Floating Point Numbers: ANSI/IEEE Floating Point Standard . . . . . | 206 |
| 5.6.4 | Characters: ASCII and Unicode . . . . .                             | 212 |
| 5.7   | Designing ISAs for Better Performance . . . . .                     | 213 |
| 5.7.1 | Technological Improvements and Their Effects . . . . .              | 214 |
| 5.7.2 | CISC Design Philosophy . . . . .                                    | 215 |
| 5.7.3 | RISC Design Philosophy . . . . .                                    | 215 |
| 5.7.4 | Recent Trends . . . . .   | 217 |
| 5.8   | Concluding Remarks . . . . .  | 217 |
| 5.9   | Exercises . . . . .   | 218 |

## II PROGRAM EXECUTION — HARDWARE LEVELS 219

|          |   |            |
|----------|---|------------|
| <b>6</b> | <b>Program Execution Basics</b>                 | <b>221</b> |
| 6.1      | Overview of Program Execution . . . . .         | 221        |
| 6.2      | Selecting the Program: User Interface . . . . . | 222        |
| 6.2.1    | CLI Shells . . . . .                            | 223        |
| 6.2.2    | GUI Shells . . . . .                            | 224        |

|          |   |            |
|----------|---|------------|
| 6.2.3    | VUI Shells . . . . .  | 225        |
| 6.3      | Creating the Process . . . . .                                    | 227        |
| 6.4      | Loading the Program . . . . .                                     | 227        |
| 6.4.1    | Dynamic Linking of Libraries . . . . .                            | 228        |
| 6.5      | Executing the Program . . . . .                                   | 230        |
| 6.6      | Halting the Program . . . . .                                     | 231        |
| 6.7      | Instruction Set Simulator . . . . .                               | 231        |
| 6.7.1    | Implementing the Register Space . . . . .                         | 233        |
| 6.7.2    | Implementing the Memory Address Space . . . . .                   | 234        |
| 6.7.3    | Program Loading . . . . .   | 235        |
| 6.7.4    | Instruction Fetch Phase . . . . .                                 | 235        |
| 6.7.5    | Executing the ML Instructions . . . . .                           | 235        |
| 6.7.6    | Executing the Syscall Instruction . . . . .                       | 236        |
| 6.7.7    | Comparison with Hardware Microarchitecture . . . . .              | 237        |
| 6.8      | Hardware Design . . . . .   | 237        |
| 6.8.1    | Clock . . . . .   | 237        |
| 6.8.2    | Hardware Description Language (HDL) . . . . .                     | 237        |
| 6.8.3    | Design Specification in HDL . . . . .                             | 238        |
| 6.8.4    | Design Verification using Simulation . . . . .                    | 238        |
| 6.8.5    | Hardware Design Metrics . . . . .                                 | 238        |
| 6.9      | Concluding Remarks . . . . .                                      | 239        |
| 6.10     | Exercises . . . . .   | 239        |
| <b>7</b> | <b>Microarchitecture — User Mode</b>                              | <b>241</b> |
| 7.1      | Overview of User Mode Microarchitecture . . . . .                 | 243        |
| 7.1.1    | Dichotomy: Data Path and Control Unit . . . . .                   | 243        |
| 7.1.2    | Register File and Individual Registers . . . . .                  | 244        |
| 7.1.3    | Memory Structures . . . . .                                       | 245        |
| 7.1.4    | ALUs and Other Functional Units . . . . .                         | 246        |
| 7.1.5    | Interconnects . . . . .   | 247        |
| 7.1.6    | Processor and Memory Subsystems . . . . .                         | 249        |
| 7.1.7    | Micro-Assembly Language (MAL) . . . . .                           | 249        |
| 7.2      | Example Microarchitecture for Executing MIPS-0 Programs . . . . . | 251        |

|       |   |     |
|-------|---|-----|
| 7.2.1 | MAL Commands . . . . .  | 253 |
| 7.2.2 | MAL Operation Set . . . . .   | 253 |
| 7.2.3 | An Example MAL Routine . . . . .  | 253 |
| 7.3   | Interpreting ML Programs by MAL Routines . . . . .                                | 254 |
| 7.3.1 | Interpreting an Instruction — the Fetch Phase . . . . .                           | 256 |
| 7.3.2 | Interpreting Arithmetic/Logical Instructions . . . . .                            | 257 |
| 7.3.3 | Interpreting Memory-Referencing Instructions . . . . .                            | 258 |
| 7.3.4 | Interpreting Control-Changing Instructions . . . . .                              | 259 |
| 7.3.5 | Interpreting Trap Instructions . . . . .  | 260 |
| 7.4   | Memory System Organization . . . . .  | 260 |
| 7.4.1 | Memory Hierarchy: Achieving Low Latency and Cost . . . . .                        | 260 |
| 7.4.2 | Cache Memory: Basic Organization . . . . .  | 263 |
| 7.4.3 | MIPS-0 Data Path with Cache Memories . . . . .                                    | 264 |
| 7.4.4 | Cache Performance . . . . .   | 264 |
| 7.4.5 | Address Mapping Functions . . . . .   | 264 |
| 7.4.6 | Finding a Word in the Cache . . . . .   | 267 |
| 7.4.7 | Block Replacement Policy . . . . .  | 268 |
| 7.4.8 | Multi-Level Cache Memories . . . . .  | 269 |
| 7.5   | Processor-Memory Bus . . . . .  | 269 |
| 7.5.1 | Bus Width . . . . .   | 270 |
| 7.5.2 | Bus Operations . . . . .  | 270 |
| 7.6   | Processor Data Path Interconnects: Design Choices . . . . .                       | 272 |
| 7.6.1 | Multiple-Bus based Data Paths . . . . .   | 272 |
| 7.6.2 | Direct Path-based Data Path . . . . .   | 273 |
| 7.7   | Pipelined Data Path: Overlapping the Execution of Multiple Instructions . . . . . | 274 |
| 7.7.1 | Defining a Pipelined Data Path . . . . .  | 275 |
| 7.7.2 | Interpreting ML Instructions in a Pipelined Data Path . . . . .                   | 279 |
| 7.7.3 | Control Unit for a Pipelined Data Path . . . . .                                  | 279 |
| 7.7.4 | Dealing with Control Flow . . . . .   | 280 |
| 7.7.5 | Dealing with Data Flow . . . . .  | 284 |
| 7.7.6 | Pipelines in Commercial Processors . . . . .                                      | 286 |
| 7.8   | Wide Data Paths: Superscalar and VLIW Processing . . . . .                        | 287 |
| 7.9   | Co-Processors . . . . .   | 288 |



|          |  |            |
|----------|--|------------|
| 7.10     | Processor Data Paths for Low Power . . . . .   | 288        |
| 7.11     | Concluding Remarks . . . . .   | 290        |
| 7.12     | Exercises . . . . .  | 291        |
| <b>8</b> | <b>Microarchitecture — Kernel Mode</b>   | <b>293</b> |
| 8.1      | Processor Management . . . . .   | 293        |
| 8.1.1    | Interpreting a System Call Instruction . . . . .   | 294        |
| 8.1.2    | Recognizing Exceptions and Hardware Interrupts . . . . .                                 | 295        |
| 8.1.3    | Interpreting an RFE Instruction . . . . .  | 297        |
| 8.2      | Memory Management: Implementing Virtual Memory . . . . .                                 | 297        |
| 8.2.1    | Virtual Memory: Implementing a Large Address Space . . . . .                             | 297        |
| 8.2.2    | Paging and Address Translation . . . . .   | 301        |
| 8.2.3    | Page Table Organization . . . . .  | 304        |
| 8.2.4    | Translation Lookaside Buffer (TLB) . . . . .   | 306        |
| 8.2.5    | Software-Managed TLB and the Role of the Operating System in<br>Virtual Memory . . . . . | 309        |
| 8.2.6    | Sharing in a Paging System . . . . .   | 312        |
| 8.2.7    | A Real-Life Example: a MIPS-I Virtual Memory System . . . . .                            | 312        |
| 8.2.8    | Interpreting a MIPS-I Memory-Referencing Instruction . . . . .                           | 319        |
| 8.2.9    | Combining Cache Memory and Virtual Memory . . . . .                                      | 320        |
| 8.3      | IO System Organization . . . . .   | 321        |
| 8.3.1    | Implementing the IO Address Space: IO Data Path . . . . .                                | 322        |
| 8.3.2    | Implementing the IO Interface Protocols: IO Controllers . . . . .                        | 323        |
| 8.3.3    | Example IO Controllers . . . . .   | 324        |
| 8.3.4    | Frame Buffer: . . . . .  | 325        |
| 8.3.5    | IO Configuration: Assigning IO Addresses to IO Controllers . . . . .                     | 329        |
| 8.4      | System Architecture . . . . .  | 332        |
| 8.4.1    | Single System Bus . . . . .  | 332        |
| 8.4.2    | Hierarchical Bus Systems . . . . .   | 333        |
| 8.4.3    | Standard Buses and Interconnects . . . . .   | 341        |
| 8.4.4    | Expansion Bus and Expansion Slots . . . . .  | 352        |
| 8.4.5    | IO System in Modern Desktops . . . . .   | 354        |
| 8.4.6    | Circa 2006 . . . . .   | 355        |
| 8.4.7    | RAID . . . . .   | 356        |

|          |  |            |
|----------|--|------------|
| 8.5      | Network Architecture . . . . .   | 357        |
| 8.5.1    | Network Interface Card (NIC) . . . . .                                 | 358        |
| 8.5.2    | Protocol Stacks . . . . .  | 359        |
| 8.6      | Interpreting an IO Instruction . . . . .                               | 359        |
| 8.7      | System-Level Design . . . . .  | 359        |
| 8.8      | Concluding Remarks . . . . .   | 360        |
| 8.9      | Exercises . . . . .  | 360        |
| <b>9</b> | <b>Register Transfer Level Architecture</b>                            | <b>361</b> |
| 9.1      | Overview of RTL Architecture . . . . .                                 | 362        |
| 9.1.1    | Register File and Individual Registers . . . . .                       | 362        |
| 9.1.2    | ALUs and Other Functional Units . . . . .                              | 363        |
| 9.1.3    | Register Transfer Language . . . . .                                   | 363        |
| 9.2      | Example RTL Data Path for Executing MIPS-0 ML Programs . . . . .       | 364        |
| 9.2.1    | RTL Instruction Set . . . . .  | 366        |
| 9.2.2    | RTL Operation Types . . . . .  | 367        |
| 9.2.3    | An Example RTL Routine . . . . .                                       | 368        |
| 9.3      | Interpreting ML Programs by RTL Routines . . . . .                     | 369        |
| 9.3.1    | Interpreting the Fetch and PC Update Commands for Each Instruction     | 369        |
| 9.3.2    | Interpreting Arithmetic/Logical Instructions . . . . .                 | 371        |
| 9.3.3    | Interpreting Memory-Referencing Instructions . . . . .                 | 372        |
| 9.3.4    | Interpreting Control-Changing Instructions . . . . .                   | 373        |
| 9.3.5    | Interpreting Trap Instructions . . . . .                               | 374        |
| 9.4      | RTL Control Unit: An Interpreter for ML Programs . . . . .             | 375        |
| 9.4.1    | Developing an Algorithm for RTL Instruction Generation . . . . .       | 375        |
| 9.4.2    | Designing the Control Unit as a Finite State Machine . . . . .         | 377        |
| 9.4.3    | Incorporating Sequencing Information in the Microinstruction . . . . . | 380        |
| 9.4.4    | State Reduction . . . . .  | 381        |
| 9.5      | Memory System Design . . . . .   | 382        |
| 9.5.1    | A Simple Memory Data Path . . . . .                                    | 383        |
| 9.5.2    | Memory Interface Unit . . . . .  | 383        |
| 9.5.3    | Memory Controller . . . . .  | 383        |
| 9.5.4    | DRAM Controller . . . . .  | 383        |

|           |   |            |
|-----------|---|------------|
| 9.5.5     | Cache Memory Design . . . . .   | 383        |
| 9.5.6     | Cache Controller: Interpreting a Read/Write Command . . . . .             | 384        |
| 9.6       | Processor Data Path Interconnects: Design Choices . . . . .               | 384        |
| 9.6.1     | Multiple-Bus based Data Paths . . . . .                                   | 385        |
| 9.6.2     | Direct Path-based Data Path . . . . .                                     | 387        |
| 9.7       | Pipelined Data Path: Overlapping the Execution of Multiple Instructions . | 390        |
| 9.7.1     | Defining a Pipelined Data Path . . . . .                                  | 390        |
| 9.7.2     | Interpreting ML Instructions in a Pipelined Data Path . . . . .           | 393        |
| 9.7.3     | Control Unit for a Pipelined Data Path . . . . .                          | 393        |
| 9.8       | Concluding Remarks . . . . .  | 394        |
| 9.9       | Exercises . . . . .   | 395        |
| <b>10</b> | <b>Logic-Level Architecture</b>   | <b>397</b> |
| 10.1      | Overview . . . . .  | 397        |
| 10.1.1    | Multiplexers . . . . .  | 399        |
| 10.1.2    | Decoders . . . . .  | 400        |
| 10.1.3    | Flip-Flops . . . . .  | 402        |
| 10.1.4    | Static RAM . . . . .  | 403        |
| 10.1.5    | Dynamic RAM . . . . .   | 404        |
| 10.1.6    | Tri-State Buffers . . . . .   | 405        |
| 10.2      | Implementing ALU and Functional Units of Data Path . . . . .              | 405        |
| 10.2.1    | Implementing an Integer Adder . . . . .                                   | 406        |
| 10.2.2    | Implementing an Integer Subtractor . . . . .                              | 415        |
| 10.2.3    | Implementing an Arithmetic Overflow Detector . . . . .                    | 416        |
| 10.2.4    | Implementing Logical Operations . . . . .                                 | 419        |
| 10.2.5    | Implementing a Shifter . . . . .  | 419        |
| 10.2.6    | Putting It All Together: ALU . . . . .                                    | 419        |
| 10.2.7    | Implementing an Integer Multiplier . . . . .                              | 420        |
| 10.2.8    | Implementing a Floating-Point Adder . . . . .                             | 425        |
| 10.2.9    | Implementing a Floating-Point Multiplier . . . . .                        | 425        |
| 10.3      | Implementing a Register File . . . . .                                    | 425        |
| 10.3.1    | Logic-level Design . . . . .  | 426        |
| 10.3.2    | Transistor-level Design . . . . .   | 429        |

|          |  |            |
|----------|--|------------|
| 10.4     | Implementing a Memory System using RAM Cells . . . . .                 | 431        |
| 10.4.1   | Implementing a Memory Chip using RAM Cells . . . . .                   | 431        |
| 10.4.2   | Implementing a Memory System using RAM Chips . . . . .                 | 431        |
| 10.4.3   | Commercial Memory Modules . . . . .                                    | 432        |
| 10.5     | Implementing a Bus . . . . .   | 433        |
| 10.5.1   | Bus Design . . . . .   | 434        |
| 10.5.2   | Bus Arbitration . . . . .  | 435        |
| 10.5.3   | Bus Protocol: Synchronous versus Asynchronous . . . . .                | 435        |
| 10.6     | Interpreting Microinstructions using Control Signals . . . . .         | 439        |
| 10.6.1   | Control Signals . . . . .  | 439        |
| 10.6.2   | Control Signal Timing . . . . .  | 442        |
| 10.6.3   | Asserting Control Signals in a Timely Fashion . . . . .                | 442        |
| 10.7     | Implementing the Control Unit . . . . .                                | 443        |
| 10.7.1   | Programmed Control Unit: A Regular Control Structure . . . . .         | 443        |
| 10.7.2   | Hardwired Control Signal Generator: A Fast Control Mechanism . . . . . | 446        |
| 10.7.3   | Hardwired versus Programmed Control Units . . . . .                    | 449        |
| 10.8     | Concluding Remarks . . . . .   | 449        |
| 10.9     | Exercises . . . . .  | 450        |
| <b>A</b> | <b>MIPS Instruction Set</b>  | <b>451</b> |
| <b>B</b> | <b>Peripheral Devices</b>  | <b>453</b> |
| B.1      | Types and Characteristics of IO Devices . . . . .                      | 453        |
| B.2      | Video Terminal . . . . .   | 454        |
| B.2.1    | Keyboard . . . . .   | 455        |
| B.2.2    | Mouse . . . . .  | 456        |
| B.2.3    | Video Display . . . . .  | 457        |
| B.3      | Printer . . . . .  | 458        |
| B.4      | Magnetic Disk . . . . .  | 459        |
| B.5      | Modem . . . . .  | 460        |



# Chapter 1

## Introduction

*Let the wise listen and add to their learning, and let the discerning get guidance*

**Proverbs 1: 5**

We begin this book with a broad overview of digital computers. This chapter serves as a context for the remainder of this book. It begins by examining the nature of the computing process. It then discusses the fundamental aspects of digital computers, and moves on to recent trends in desktop computer systems. Finally, it introduces the concept of the computer as a hierarchical system. The major levels of this hierarchical view are introduced. The remainder of the book is organized in terms of these levels.

*“The computer is by all odds the most extraordinary of the technological clothing ever devised by man, since it is an extension of our central nervous system. Beside it the wheel is a mere hula hoop....”*

— Marshall McLuhan. *War and Peace in the Global Village*

Born a few years back, digital computer technology, in cohort with telecommunication technology, has ushered us into the information age and is exerting a profound influence on almost every facet of our daily lives<sup>1</sup>. Most of us spend a substantial time every day in front of a computer (most of it on the internet or on some games!). Rest of the time, we are on the cell phone or some other electronic device with one or more computers embedded within. On a more serious note, we are well aware of the critical role played by computers in flying modern aircraft and spacecraft; in keeping track of large databases such as airline reservations and bank accounts; in telecommunications applications such as routing and controlling millions of telephone calls over the entire world; and in controlling power stations and hazardous chemical plants. Companies and governmental agencies are virtually crippled

---

<sup>1</sup>This too shall pass .....

when their computer systems go down, and a growing number of sophisticated medical procedures are completely dependent on computers. Biologists are using computers for performing extremely complex computations and simulations. Computer designers are using them extensively for developing tomorrow's faster and denser computer chips. Publishers use them for typesetting, graphical picture processing, and desktop publishing. The writing of this book itself has benefitted substantially from desktop publishing software, especially LaTeX. Thus, computers have taken away many of our boring chores, and have replaced them with addictions such as chatting, browsing, and computerized music.

What exactly is a *computer*? A *computer science* definition would be as follows: a computer is a programmable symbol-processing machine that accepts input symbols, processes it according to a sequence of instructions called a computer *program*, and produces the resulting output symbols. The input symbols as well as the output symbols can represent numbers, characters, pictures, sound, or other kinds of data such as chess pieces. The most striking property of the computer is that it is *programmable*, making it a truly *general-purpose machine*. The user can change the program or the input data according to specific requirements. Depending on the software run, the end user “sees” a different machine; the computer user's view thus depends on the program being run on the computer at any given instant. Suppose a computer is executing a chess program. As far as the computer user is concerned, at that instant the computer is a chess player because it behaves exactly as if it were an electronic chess player<sup>2</sup>. Because of the ability to execute different programs, a computer is a truly general-purpose machine. The same computer can thus perform a variety of information-processing tasks that range over a wide spectrum of applications—for example, as a word processor, a calculator, or a video game—by executing different programs on it; a multitasking computer can even simultaneously perform different tasks. The computer's ability to perform a wide variety of tasks at very high speeds and with high degrees of accuracy is what makes it so ubiquitous.

*“The computer is only a fast idiot, it has no imagination; it cannot originate action. It is, and will remain, only a tool to man.”*  
 — American Library Association's reaction to the UNIVAC computer exhibit at the 1964 New York World's Fair

---

<sup>2</sup>In the late 1990s, a computer made by IBM called *Deep Thought* even defeated the previous World Chess Champion Gary Kasparov. It is interesting to note, however, that if the rules of chess are changed even slightly (for example, by allowing the king to move two steps at a time), then current computers will have a difficult time, unless they are reprogrammed or reconstructed by humans. In contrast, even an amateur human player will be able to comprehend the new rules in a short time and play a reasonably good game under the new rules!

## 1.1 Computing and Computers

The notion of computing (or problem solving) is much more fundamental than the notion of a computer, and predates the invention of computers by thousands of years. In fact, computing has been an integral aspect of human life and civilization throughout history. Over the centuries, mathematicians developed algorithms for solving a wide variety of mathematical problems. Scientists and engineers used these algorithms to obtain solutions for specific problems, both practical and recreational. And, we have been computing ever since we entered kindergarten, using fingers, followed later by paper and pencil. We have been adding, subtracting, multiplying, dividing, computing lengths, areas, volumes and many many other things. In all these computations, we follow some definite, unambiguous set of rules that have been established. For instance, once the rules for calculating the area of a complex shape have been established—divide it into non-overlapping basic shapes and add up the areas of the shapes—we can calculate the area of any complex shape.

A typical modern-day *computing* problem is much more complex, but works on the same fundamental principles. Consider a metropolitan traffic control center where traffic video images from multiple cameras are being fed, and a human operator looks at the images and takes various traffic control decisions. Imagine automating this process, and letting a computer do the merging of the images and taking various decisions! How should we go about designing such a computer system?

### 1.1.1 The Problem-Solving Process

Finding a solution to a problem, irrespective of whether or not we use a computer, involves two important phases, as illustrated in Figure 1.1:

- Algorithm development
- Algorithm execution

We shall take a detailed look at these two phases.

#### 1.1.1.1 Algorithm Development

The first phase of computing involves the development of a solution *algorithm* or a step-by-step procedure that describes how to solve the problem. When we explicitly write down the rules (or instructions) for solving a given computation problem, we call it an algorithm. An example algorithm is the procedure for finding the solution of a quadratic equation. Informally speaking, many of the recipes, procedures, and methods in everyday life are algorithms.

What should be the granularity of the steps in an algorithm? This depends on the sophistication of the person or machine who will execute it, and can vary significantly from



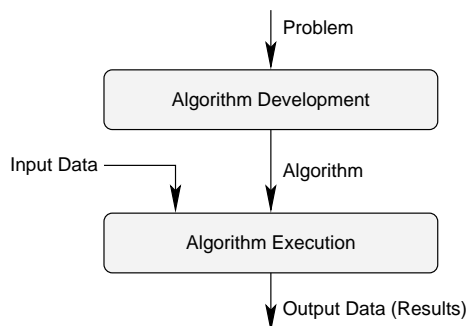


Figure 1.1: The Problem Solving Process

one algorithm to another; a step can be as complex as finding the solution of a sub-problem, or it can be as simple as an addition/subtraction operation. Interestingly, an addition step itself can be viewed as a problem to be solved, for which a solution algorithm can be developed in terms of 1-bit addition with carry-ins and carry-outs. It should also be noted that one may occasionally tailor an algorithm to a specific set of input data, in which case it is not very general.

Algorithm development has always been done with human brain power, and in all likelihood will continue like that for years to come! Algorithm development has been recorded as early as 1800 B.C., when Babylonian mathematicians at the time of Hammurabi developed rules for solving many types of equations [4]. The word “algorithm” itself was derived from the last name of al-Khwārizmī, a 9th century Persian mathematician whose textbook on arithmetic had a significant influence for more than 500 years.

### 1.1.1.2 Algorithm Execution

Algorithm execution—the second phase of the problem-solving process—means applying a solution algorithm on a particular set of *input values*, so as to obtain the solution of the problem for that set of input values. Algorithm development and execution phases are generally done one after the other; once an algorithm has been developed, it may be executed any number of times with different sets of data without further modifications. However, it is possible to do both these phases concurrently, in a lock-step manner! This typically happens when the same person performs both phases, and is attempting to solve a problem for the first time.

The actions involved in algorithm execution can be broken down into two parts, as illustrated in Figure 1.2.

- *Sequencing* through the algorithm steps: This part involves selecting from the algorithm the next step to be executed.

- *Executing* the next step of the algorithm, as determined by the sequencing part.

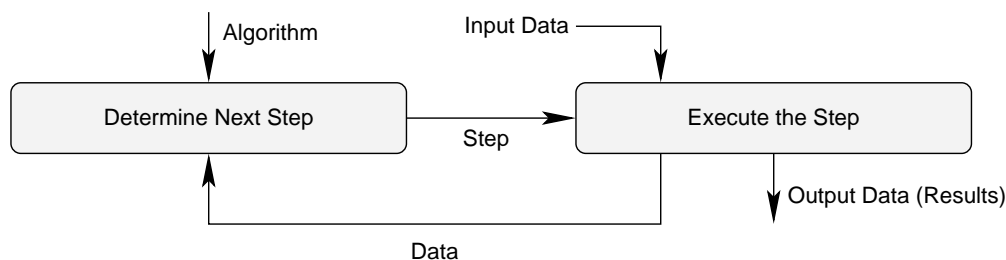


Figure 1.2: The Algorithm Execution Process

For hundreds of years, people relied mainly on human brain power for performing both of these parts. As centuries went by (and the gene pool deteriorated), a variety of computing aids were invented to aid human brains in executing the individual steps of solution algorithms. The Chinese **abacus** and the Japanese **soroban** were two of the earliest documented aids used for doing the arithmetic calculations specified in algorithm steps. The **slide rule** was a more sophisticated computing aid invented in the early 1600s by William Oughtred, an English clergyman; it helped to perform a variety of computation operations including multiplication and division. Later examples of computing aids included **Pascaline**, the mechanical adder built in 1642 by the French mathematician Blaise Pascal (to assist his father in adding long columns of numbers in the tax accounting office) and the stepped-wheel machine of Gottfried Wilhelm Leibniz in 1672 (which could perform multiplication and division in addition to addition and subtraction).

As problems increased in complexity, the number of steps required to solve them also increased accordingly. Several mechanical and electrical calculators were commercially produced in the 19th century to speed up specific computation steps. The time taken by a calculator to perform a computation step was in the order of a few milliseconds, in contrast to the several seconds or minutes taken by a person to perform the same step. It is important to note that even after the introduction of calculators, the sequencing part of algorithm execution was still done by people, who punched in the numbers and the operations. It is also important to note that the granularity of the steps in an algorithm is related to the capabilities and sophistication of the calculating aids used. Thus, a typical calculator lets us specify algorithm steps such as multiplication and square root, for instance, whereas an abacus can perform only more primitive computation steps.

### 1.1.2 Automating Algorithm Execution with Computers

We saw that calculators and other computing aids allowed an algorithm's computation steps to be executed much faster than what was possible without any computing aides. However, the algorithm execution phase still consumed a significant amount of time for

the following reasons: (i) the sequencing process was still manual, and (ii) the execution of each computation step involved manual inputting of data into the calculating aid. Both of these limitations can be overcome if the sequencing process is automated by means of an appropriate machine, and the data to be processed is stored in the machine itself. This is the basic idea behind computers.

*“Stripped of its interfaces, a bare computer boils down to little more than a pocket calculator that can push its own buttons and remember what it has done.”*  
 – Arnold Penzias. *Ideas and Information*.

One of the earliest attempts to automate algorithm execution was that of Charles Babbage, a 19th century mathematics professor. He developed a mechanical computing machine called **Difference Engine**. This computer was designed to execute only a single algorithm—the *method of (finite) differences using polynomials*. Although this algorithm used only addition and subtraction operations, it permitted many complex and useful functions to be calculated. (Chapter 1 of [2] provides a good description of the use of this algorithm in calculating different functions.) The **Difference Engine** performed the sequencing process automatically, in addition to performing the operation specified in each computation step. This is a major advantage because it allows the algorithm execution phase to be performed at machine speeds, rather than at the speed with which it can be done manually. One limitation of executing a single algorithm, however, is that only a few problems can be solved by a single algorithm; such a computing machine is therefore not useful for general-purpose computing.

After a few years, Babbage envisioned the **Analytical Engine**, another massive brass, steam-powered, mechanical (digital) computing machine. The radical shift that it introduced was to have the machine accept an arbitrary solution algorithm (in punched card format), and execute the algorithm by itself. This approach allows arbitrary algorithms to be executed at the speed of the machine, making the machine a general-purpose computer. The radical idea embodied in the Analytical Engine was the recognition that a machine could be “programmed” to perform a long sequence of arithmetic and decision operations without human intervention.

*“What if a calculating engine could not only foresee but could act on that foresight?”*  
 – Charles Babbage. *November 1834*.

The Analytical Engine served as a blueprint for the first *real* programmable computer, which came into existence a century later<sup>3</sup>. The basic organization proposed by Babbage is given in Figure 1.3. The main parts are the mill, the store, the printer and card punch, the operation cards, and the variable cards. The instructions were given to the machine on punch cards, and the input data was supplied through the variable cards. Punched cards had

---

<sup>3</sup>Primitive forms of “programmable” machines had existed centuries ago, dating back to Al-Jazari’s *musical automata* in the 13th century and even to Heron’s *mobile automaton* in the 1st century.

been recently invented by Jacquard for controlling weaving looms. Augusta Ada, Countess of Lovelace as well as a mathematician, was one of the few people who fully understood Babbage's vision. She helped Babbage in designing the Analytical Engine's instruction set, and in describing, analyzing, and publicizing his ideas. In an article published in 1843, she predicted that such a machine might be used to compose complex music, to produce graphics, and would be used for both practical and scientific use. She also created a plan for how the engine might calculate a mathematical sequence known as Bernoulli numbers. This plan is now regarded as the first "computer program," and Ada is credited as the first computer programmer.

*"The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform."*

*"Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent."*

— Countess Ada Lovelace

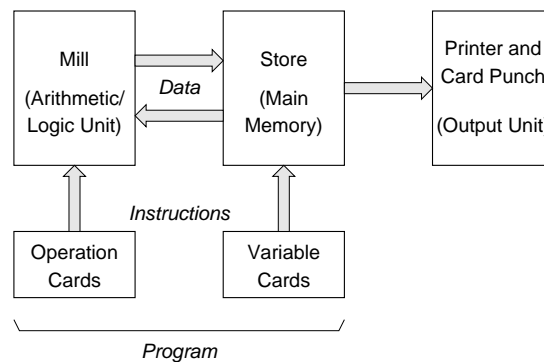


Figure 1.3: Basic Organization of Babbage's **Analytical Engine**

Automated algorithm execution has two side-effects that we need to keep in mind. First, it forces the algorithm development and algorithm execution phases to happen one after the other. It also implies that the algorithm must allow for the occurrence of all possible inputs. Hence computer algorithms are seldom developed to solve just a single instance of a problem; rather they are developed to handle different sets of input values. Thus, in moving from the manual approach to the automated approach, we are forced to sacrifice the versatility inherent in the concurrent development and execution of an algorithm. The big gain, however, is in the speed and storage capabilities offered by the computer machine.

Another side-effect of automated algorithm execution is that for a machine to follow an algorithm, the algorithm must be represented in a formal and detailed manner: the less

sophisticated the follower, the more detailed the algorithm needs to be! Detailed algorithms written for computers are called *computer programs*. By definition, a computer program is an expression of an algorithm in a *computer programming language*, which is a precise language that can be made understandable to a computer. Because of the extensive efforts involved in developing a computer program to make it suitable for execution in a computer, the program itself is often developed with a view to solve a range of related problems rather than a single problem. For instance, it may not be profitable to develop a computer program to process a single type of bank transaction; instead, it is profitable to develop the program with the ability to process different types of transactions.

In spite of these minor side-effects, the idea of using computers to perform automated algorithm execution has been found to have tremendous potential. First of all, once an algorithm has been manually developed to solve a particular problem, computers can be used to execute the algorithm at very high speeds. This makes it possible to execute long-running algorithms that require billions of operations, which previously could never be executed in a reasonable period of time<sup>4</sup>. In fact, a lion's share of computer development took place in the 1930s and 1940s, mostly in response to computation problems that arose in the WW II effort, such as ballistic computations and code-breaking. The ability to execute complex algorithms in real-time is the leading reason for the acceptance of computers in many embedded applications, such as automobiles and aircraft. Secondly, the same computing machine can be used to execute different algorithms at different times, thus having a truly general-purpose computing machine. Thirdly, computers are immune to emotional and physical factors such as distraction and fatigue, and can provide accurate and reliable results almost all the time<sup>5</sup>. Finally, embedded applications often involve working in hazardous environments where humans cannot go, and computers are good candidates for use in such environments.

At this stage, it is instructive to contrast the computing machine against other types of machines such as clocks, which predate the computer by hundreds of years. Such machines are constructed to perform a specific sequence of internal actions to solve a specific problem. For instance, the hands of a clock go around at fixed speeds; this is in fact a mechanical implementation of an algorithm to keep track of time. A digital clock keeps track of time using a quartz crystal and digital circuitry. Such machines can only do the one thing they are constructed to do. A computing machine, on the other hand, is general-purpose in that it can perform a large variety of widely differing functions, based on the algorithm that it is operating upon at any given time. Because the algorithm can be changed, different functions can be implemented by acquiring a single hardware system and then developing different algorithms to perform different functions in the hardware. Thus, by executing a

---

<sup>4</sup>Interestingly, even now, at a time when computers have become faster by several orders of magnitude, there are prodigies like Sakuntala Devi [] who have demonstrated superiority over computers in performing certain kind of complex calculations!

<sup>5</sup>We should mention that computers are indeed susceptible to some environmental factors such as electrical noise and high temperatures. Modern computers use error-correcting codes and other fault tolerance measures to combat the effect of electrical noise and other environmental effects.

computer program for keeping track of time, a computer can implement a clock! This feature is the crucial difference between general-purpose computing machines and special-purpose machines that are geared to perform specific functions.

We have described some of the landmark computers in history. Besides the few computers mentioned here, there are many other precursors to the modern computer. Extensive coverage of these computers can be found in the *IEEE Annals of the History of Computing*, now in its 28th volume [ref].

|  |
|--|
| <p><i>“Computers in the future will weigh no more than 0.5 tons.”</i><br/>— <i>Popular Mechanics: Forecasting Advance of Science, 1949</i></p> |
|--|

## 1.2 The Digital Computer

We saw how computers play a major role in executing algorithms or programs to obtain solutions for problems. Solving a problem involves manipulating information of one kind or other. In order to process information, any computer—mechanical or electrical—should internally represent information by some means. Some of the early computers were *analog computers*, in that they represented information by physical quantities that can take values from a continuum, rather than by numbers or bit patterns that represent such quantities. Physical quantities can change their values by an arbitrarily small amount; examples are the rotational positions of gears in mechanical computers, and voltages in electrical computers. Analog quantities represent data in a continuous form that closely resemble the information they represent. The electrical signals on a telephone line, for instance, are analog-data representations of the original voices. Instead of doing arithmetic or logical operations, an analog computer uses the physical characteristics of its data to determine solutions. For instance, addition could be done just by using a circuit whose output voltage is the sum of its input voltages.

Analog computers were a natural outcome of the desire to directly model the smoothly varying properties of physical systems. By making use of different properties of physical quantities, analog computers can often avoid time-consuming arithmetic and logical operations. Although analog computers can nicely represent smoothly changing values and make use of their properties, they suffer from the difficulty in measuring physical quantities precisely, and the difficulty in storing them precisely due to changes in temperature, humidity, and so on. The subtle errors introduced to the stored values due to such noise are difficult to detect, let alone correct.

The 20th century saw the emergence of *digital computers*, which eventually replaced analog computers in the general-purpose computing domain. Digital computers represent and manipulate information using *discrete* elements called *symbols*. A major advantage of using symbols to represent information is resilience to error. Even if a symbol gets distorted, it can still be recognized, as long as the distortion does not cause it to appear like another

symbol. This is the basis behind error-correcting features used to combat the effects of electrical noise in digital systems. Representing information in digital format has a side-effect, however. As we can only have a limited number of bits, only a finite number of values can be uniquely represented. This means that some of the values can be represented with high degree of precision, whereas the remaining ones will need to be approximated.

Electronic versions of the digital computer are typically built out of a large collection of electronic switches, and use distinct voltage states (or current states) to represent different symbols. Each switch can be in one of two positions, *on* or *off*; designing a digital computer will therefore be a lot simpler if it is restricted to handling just two symbols. So most of the digital computers use only two symbols in their alphabet and are *binary* systems, although we can design computers and other digital circuits that handle multiple symbols with *multiple-valued logic*. The two symbols of the computer alphabet are usually represented as 0 and 1; each symbol is called a binary digit or a *bit*. Computers often need to represent different kinds of information, such as instructions, integers, floating-point numbers, and characters. Whatever be the type of information, digital computers represent them by concatenations of bits called *bit patterns*, just like representing information in English by concatenating English alphabets and punctuation marks. The finite number of English alphabets and punctuation marks do not impose an inherent limit on how much we can communicate in English; similarly the two symbols of the computer alphabet do not place any inherent limits on what can be communicated to the digital computer. Notice, however, that information in the computer language won't be as cryptic as in English, just like information in English is not as cryptic as in Chinese (which has far more symbols).

*“Even the most sophisticated computer is really only a large, well-organized volume of bits.”*  
 — David Harel. *Algorithmics: The Spirit of Computing*

By virtue of their speed and other nice properties, these electronic versions completely replaced mechanical and electromechanical versions. At present, the default meaning of the term “computer” is a *general-purpose automatic electronic digital computer*.

### 1.2.1 Representing Programs in a Digital Computer: The Stored Program Concept

We saw that a computer solves a problem by executing a program with the appropriate set of input data. How is this program conveyed to the computer from the external world? And, how is it represented within the computer? In the ENIAC system developed at University of Pennsylvania in early 1940s, for instance, the program was a function of how its electrical circuits were wired, i.e., the program was a function of the physical arrangement of the cables in the system. The steps to be executed were specified by the connections within the hardware unit. Every time a different program needed to be executed, the system had to be rewired. Conveying a new program to the hardware sometimes took several weeks! Other

early computers used plug boards, punched paper tape, or some other external means to represent programs. Developing a new program involved re-wiring a plugboard, for instance. And, loading a program meant physically plugging in a patch board or running a paper tape through a reader.

A marked change occurred in the mid-1940s when it was found that programs could be represented inside computers in the same manner as data, i.e., by symbols or bit patterns. This permits programs to be stored and transferred like data. This concept is called the *stored program* concept, and was first described in a landmark paper by Burks, Goldstein, and von Neumann in 1946 [1]. In a digital computer implementing the stored program concept, a program will be a collection of bit patterns. When programs are represented and stored as bit patterns, a new program can be conveyed to the hardware very easily. Moreover, several programs can be simultaneously stored in the computer's memory. This makes it easy not only to execute different programs one after the other, but also to switch from one program to another and then back to the first, without any hardware modification. Stored program computers are truly "general-purpose," as they can be easily adapted to do different types of computational and information storage tasks. For instance, a computer can instantaneously switch from being a word processor to a telecommunications terminal, a game machine, or a musical instrument! Right from its inception, the stored program concept was found to be such a good idea that it has been the basis for virtually every general-purpose computer designed since then. In fact it has become so much a part of the modern computer's functioning that it is not even mentioned as a feature!

In a stored program computer, the program being executed can even manipulate another program as if it were data—for example, load it into the computer's memory from a storage device, copy it from one part of memory to another, and store it back on a storage device. Altering a program becomes as easy as modifying the contents of a portion of the computer's memory. The ability to manipulate stored programs as data gave rise to compilers and assemblers that take programs as input and translate them into other languages.

The advent of compilers and assemblers have introduced several additional steps in solving problems using modern digital computers. Figure 1.3 depicts the steps involved in solving a problem using today's computers. First, an algorithm, or step-by-step procedure, is developed to solve the problem. Then this algorithm is expressed as a program in a high-level programming language by considering the syntax rules and semantic rules of the programming language. Some of the common high-level languages are C, FORTRAN, C++, Java, and VisualBasic.

The source program in the high-level language is translated into an executable program (in the language of the machine) using programs such as compilers, assemblers, and linkers. During this compilation process, syntax errors are detected, which are then corrected by the programmer. Once the syntax errors are corrected, the program is re-compiled. Once all syntax errors are corrected, the compiler produces the executable program. The executable program can be executed with a set of input values on the computer to obtain the results. Semantic errors manifest as run-time errors, and are corrected by the programmer.



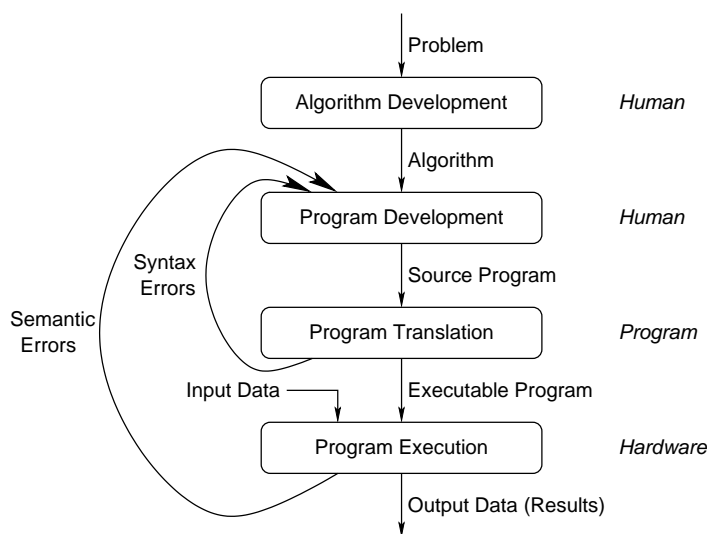


Figure 1.4: Basic Steps in Solving a Problem using a Computer

### 1.2.2 Basic Software Organization

As discussed in the previous section, today's computers use the stored program concept. Accordingly the software consists of symbols or bit patterns that can be stored in storage devices such as CD-ROMs, hard disks, and floppy disks. A program consists of two parts—*instructions* and *data*—both of which are represented by bit patterns. The instructions indicate specific operations to be performed on individual data items. The data items can be numeric or non-numeric.

It is possible to write stand-alone programs that can utilize and manage all of the system resources, so as to perform the required task. This is commonly done in controllers and embedded computers, which typically store a single program in a ROM (read-only memory), and run the same program forever. In the mid and higher end of the computer spectrum, starting with some embedded computers, a dichotomy is usually practiced, however, for a variety of reasons. Instead of writing stand-alone programs that have the ability to access and control all of the hardware resources, the access and control of many of the hardware resources (typically IO devices) are regulated through a supervisory program called the *operating system*. When a program needs to access a regulated hardware resource, it requests the operating system, which then provides the requested service if it is legitimate request. This dichotomy has led to the development of two major kinds of software—*user programs* and *kernel programs*—as shown in Figure 1.4.

The operating system is one of the most important pieces of software to go into a modern computer system. It provides other programs a uniform software interface to the hardware

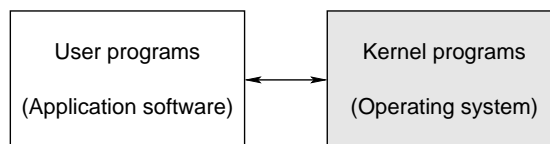


Figure 1.5: Basic Software Organization in a Digital Computer

resources. In addition to providing a standard interface to system resources, in multitasking environments, the operating system enables multiple user programs to *share* the hardware resources in an orderly fashion<sup>6</sup>. This sharing increases overall system performance, and ensures security and privacy for the individual programs. To do this sharing in a safe and efficient manner, the operating system is the software that is “closest to the hardware”. All other programs use the OS as an interface to shared resources and as a means to support sharing among concurrently executing programs.

A hardware timer periodically interrupts the running program, allowing the processor to run the operating system. The operating system then decides which of the simultaneously active application programs should be run next on the processor; it takes this decision with a view to minimize processor waste time. Peripheral devices also interrupt the running program, at which times the operating system intervenes and services the devices.

For similar reasons, memory management and exception handling functions are also typically included in the operating system. Memory management involves supporting a large virtual memory address space with a much smaller physical memory, and also sharing the available physical memory among the simultaneously active programs. Exception handling involves dealing with situations that cause unexpected events such as arithmetic overflow and divide by zero.

### 1.2.3 Basic Hardware Organization

Even the most complex software, with its excellent abstraction and generality features, is only like the mental picture an artist has before creating a masterpiece. By itself it does not solve any problem. For it to be productive, it must be eventually executed on suitable hardware with proper data, just like the artist executing his/her mental picture on a suitable canvas with proper paint. The hardware is thus an integral part of any computer system.

*“You’ll never plow a field by turning it over in your mind.”*  
 — *An Irish Proverb*

While nearly every class of computer hardware has its own unique features, from a func-

---

<sup>6</sup>Even in multitasking computers, hardware diagnostic programs are often run entirely by themselves, with no intervention from the operating system.

tional point of view (i.e, from the point of view of what the major parts are supposed to do), the basic organization of modern computers—given in Figure 1.5—is still very similar to that of the **Analytical Engine** proposed in the 19th century. This organization consists of three functionally independent parts: the CPU (central processing unit), the memory unit, and the input/output unit. The actions performed by the computer are controlled and co-ordinated by the program that is currently being executed by the CPU. The input/output unit is a collection of diverse devices that enable the computer to communicate with the outside world. Standard input/output devices include the keyboard, the mouse, the monitor, and so on. Programs and data are brought into the computer from the external world using the input devices and their controllers. The input unit's function is to accept information from human users or other machines, through devices such as the keyboard, the mouse, the modem, and the actuators. The results of computations are sent to the outside world through the output unit. Some of the input/output devices are *storage devices*, such as hard disks, CD-ROMs, and tapes, which can store information for an indefinite period of time.

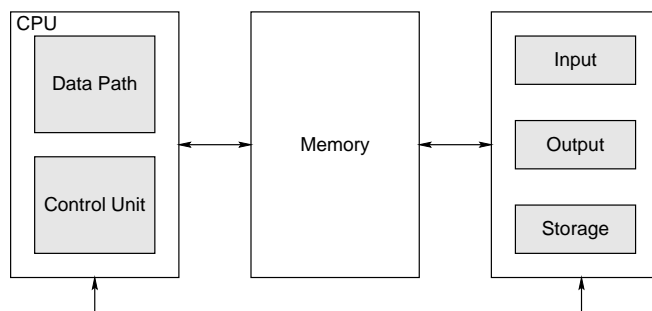


Figure 1.6: Basic Hardware Organization of a Digital Computer

When the program and data are ready to be used, they are copied into the memory unit from either the external environment or a storage device. The memory unit stores two types of information: (i) the instructions of the program being executed, and (ii) the data for the program being executed. The CPU executes a memory-resident program by reading the program instructions and data from the memory. The execution of the program is carried out by the CPU's control unit, which reads each instruction in the program, decodes the instruction, and causes it to be executed in the data path. The control unit is the brain of the system, and behaves like a puppeteer who pulls the right strings to make the puppets behave exactly as needed.

It is interesting to compare and contrast this organization with that of a human being's information processing system, which among other things, involves the brain. The main similarity lies in the way information is input and output. Like the digital computer, the human information processing system obtains its inputs through its input unit (the sense organs), and provides its outputs through its output unit by way of speech and various

motions. The dissimilarity lies both in the organization of the remaining parts and in the way information is stored and processed. All of the information storage and processing happens in a single unit, the brain. Again, the brain stores information not as 0s and 1s in memory elements, but instead by means of its internal connectivity. Information processing is done in the brain on a massively parallel manner. This is in contrast to how information processing is done in a digital computer, where the information is stored in different memory units from where small pieces are brought into the CPU and processed<sup>7</sup>.

### 1.2.4 Software versus Hardware

Software consists of abstract ideas, algorithms, and their computer representations, namely programs. Hardware, in contrast, consists of tangible objects such as integrated circuits, printed circuit boards, cables, power supplies, memories, and printers. Software and hardware aspects are intimately tied together, and to achieve a good understanding of computer systems, it is important to study both, especially how they integrate with each other. Therefore, the initial portions of this book deal with software and programming, and the latter portions deal with the hardware components. This introductory chapter introduces a number of software and hardware concepts, and gives a broad overview of the fundamental aspects of both topics. More detailed discussions follow in subsequent chapters.

The boundary between the software and the hardware is of particular interest to systems programmers and compiler developers. In the very first computers, this boundary—the instruction set architecture—was quite clear; the hardware presented the programmer with an abstract model that took instructions from a serial program one at a time and executed them in the order in which they appear in the program. Over time, however, this boundary blurred considerably, as more and more hardware features are exposed to the software, and hardware design itself involves software programming techniques. Nowadays, it is often difficult to tell software and hardware apart, especially at the boundary between them. In fact, a central theme of this book is:

*Hardware and software are logically equivalent.*

Any operation performed by software can also be built directly into the hardware. Embedded systems, which are more specialized than their general-purpose counterpart, tend to do more through hardware than through software. In general, new functionality is first introduced in software, as it is likely to undergo many changes. As the functionality becomes more standard and is less likely to change, it is migrated to hardware.

*“Hardware is petrified software.”*

— ????

Of course, the reverse is also true: Any instruction executed by the hardware can also

---

<sup>7</sup>Research is under way to develop computers made from quantum circuits, and even biological circuits. In the next decade, we may very well have computers made with such “hardware”, and working with different computation models!

be simulated in software. Suppose an end user is using a computer to play a video game. It is possible to construct an electronic circuit to *directly* handle video games, but this is seldom done. Instead a video game program is executed to *simulate* a video game. The decision to put certain functions in hardware and others in software is based on such factors as cost, speed, reliability, and frequency of expected changes. These decisions change with trends in technology and computer usage.

### 1.2.5 Computer Platforms

Classification is fundamental to understanding anything that comes in a wide variety. Automobiles can be classified according to manufacturer, body style, pickup, and size. Students often classify university faculty based on their teaching style, sense of humor, and strictness of grading. They classify textbooks according to price, contents, and ease of understanding. Likewise, computers come in various sizes, speeds, and prices, from small-scale to large-scale. Table 1.1 gives a rough categorization of today's computers. This categorization is somewhat idealized. Within each category, there is wide variability in features and cost; in practice the boundary between two adjacent categories is also somewhat blurred. The approximate price figures in the table are only intended to show order of magnitude differences between different categories. All computers are functionally similar, irrespective of where they line up in the spectrum. The general principles of computer architecture and organization are the same for the entire computer spectrum, from workstations to multiprocessors and distributed computer systems.

| Category                   | Price        | Typical applications                |
|----------------------------|--------------|-------------------------------------|
| Disposable computer        | \$1          | Greeting cards, watches             |
| Embedded computer          | \$10         | Home appliances, cars               |
| Entertainment PC           | \$100        | Home video games                    |
| Desktop or laptop PC       | \$1000       | Word processing, CAD design         |
| Server                     | \$10,000     | Network server                      |
| Collection of workstations | \$100,000    | LAN                                 |
| Mainframe                  | \$1,000,000  | Bank accounts, airline reservations |
| Supercomputer              | \$10,000,000 | Weather forecast, oil exploration   |

Table 1.1: Different categories of currently available computers

At one end of the spectrum we have disposable computers like the ones used in greeting cards, inexpensive watches, and other similar applications. These are quite inexpensive because they use a single chip with small amounts of memory, and are produced in large quantities. Then there are a wide variety of embedded computers, used in applications such as automobiles and home appliances. The entertainment PCs are computer systems that are optimized for games, personal communications, and video playback. They typically

have high-quality graphics, video, and audio so as to support high clarity and realism. Desktop computers and laptop computers are typically intended for a single user to run applications such as word processing, web browsing, and receiving/sending email. These computers come with different features and costs. In the immediately higher category, we have *servers*. A server is a high-performance computer that serves as a gateway in a computer network. At the other end of the spectrum, we have the supercomputers, which are used for applications involving very complex calculations, such as weather prediction and nuclear explosion modeling. The lower end of the spectrum often provides the best price/performance ratio, and the decision on which system to purchase is often dictated by such issues as software and object code compatibility.

## 1.3 A Modern Computer System

As discussed earlier, computers come in various sizes and kinds. Among these, perhaps the most commonly seen one and one that comes to mind vividly when one thinks of a computer, is a desktop computer. Desktop computers are designed to be truly general-purpose. For these reasons, we provide a detailed description of a typical desktop computer system in this section. In fact, many of the issues discussed here are applicable to all members of the computer spectrum.

### 1.3.1 Hardware

Figure 1.6 shows a typical desktop computer. It has a system unit which is the case or box that houses the motherboard, other printed circuit boards, the storage devices, and the power supply. The system unit is generally designed in such a way that it can be easily opened to add or replace modules. The different components in the system unit are typically connected together using a *bus*, which is a set of wires for transferring electrical signals. Each printed circuit board houses a number of chips, some of which are soldered and the rest are plugged into the board. The latter permits the user to upgrade the computer components. Circuits etched into the boards act like wires, providing a path for transporting data from one chip to another.

Figure 1.7: Photograph of a Typical Desktop Computer System

**Processor:** The processor, also called the *central processing unit (CPU)*, is perhaps the most important part of a computer. It carries out the execution of the instructions of a program.

**Chip Sets:** The chipsets provide hardware interfaces for the processor to interact with other devices, such as DRAM and graphics cards.

**Motherboard:** The motherboard is the main printed circuit board, and holds the computer's processor chip(s), ROM chips, RAM chips, and several other key electronic components. The processor is an important part of a computer, and can be a single chip or a collection of chips. ROM chips typically contain a small set of programs that start the computer, run system diagnostics, and control low-level input and output activities. These programs are collectively called *BIOS* (basic input output system) in PCs. The instructions in the ROM chips are permanent, and the only way to modify them is to reprogram the ROM chips. RAM chips are volatile and hold program and data that is temporary in nature. A battery powered *real-time clock* chip keeps track of the current date and time. The motherboard also typically contains *expansion slots*, which are sockets into which *expansion cards* such as video card, sound card, and internal modem, can be plugged in. An expansion card has a card edge connector with metal contacts, which when plugged into an expansion slot socket, connect the circuitry on the card to the circuitry on the motherboard. The number of expansion slots in the motherboard determines its expandability.

Figure 1.8: Photograph of a Motherboard

**Storage Devices:** The commonly used storage devices are floppy disk drives, hard disk drives, CD-ROM drives, and ZIP drives. A floppy disk drive is a device that reads and writes data on floppy disks. A typical floppy disk drive uses  $3\frac{1}{2}$ -inch floppy disks each of which can store up to 1.44 MB. A hard disk drive can store billions of bytes on a non-removable disk platter. A CD-ROM drive is a storage device that uses laser technology to read data from a CD-ROM. The storage devices are typically mounted in the system unit. The ones involving removable media such as the floppy disk drive, the CD-ROM drive, and the ZIP drive are mounted on the front side of the system unit, and the hard disk drives are typically mounted inside the system unit.

**Input/Output Devices:** Two of the commonly used input devices in a desktop computer are the keyboard and the mouse. A computer keyboard looks similar to that of a typewriter, with the addition of number keys, as well as several additional keys that control computer-specific tasks. The mouse is useful in manipulating objects depicted on the screen. Other commonly used input device is the microphone. The primary output device in a desktop computer is the monitor, a display device that forms an image by converting electrical signals from the computer into points of colored light on the screen. Its functioning is very to a television picture tube, but has a much higher resolution so that a user sitting at close quarters can clearly see computer-generated data such as text and images. Other frequently used output devices are the printer and the speakers.

**Device Controllers:** Each device—keyboard, mouse, printer, monitor, etc—requires special *controller* circuitry for transferring data from the processor and memory to the device, and vice versa. A device controller is designed either as a chip which is placed in the motherboard or as a printed circuit board which is plugged into an expansion slot of the motherboard. The peripheral devices are connected to their respective controllers in the system unit using special cables to sockets called *expansion ports*. The ports are located on the backside of the system unit and provide connections through holes in the back of the system unit. Parallel ports transfer several bits simultaneously and are commonly used to connect printers to the computer. Serial ports transfer a single bit at a time, and are commonly used to connect mice and communication equipment to the computer. Device controllers are very complex. Each logical command from the processor must typically be decomposed into long sequences of low-level commands to trigger the actions to be performed by the device and to supervise the progress of the operation by testing the device's status. For instance, to read a word from a disk, the disk controller generates a sequence of commands to move the read/write arm of the disk to the correct track, await the rotational delay until the correct sector passes under the read/write arm, transfer the word, and check for a number of possible error conditions. A *sound card* contains circuitry to convert digital signals from the computer to sounds that play through speakers or headphones that are connected to the expansion ports of the card. A *modem card* connects the computer to the telephone system so as to transport data from one computer to another over phone lines. A *network card*, on the other hand, provides the circuitry to connect a computer to other computers on a local area network.

### 1.3.2 Software

A desktop computer typically comes with pre-installed software. This software can be categorized into two categories—*application software* and *systems software*.

**Application Software:** Application programs are designed to satisfy end-user needs by operating on input data to perform a given job, for example, to prepare a report, update a master payroll file, or print customer bills. Application software may be *packaged* or *custom*. Packaged software includes programs pre-written by professional programmers, and are typically offered for sale in a floppy disk or CD-ROM. Custom software includes programs written for a highly specialized task.

**Systems Software:** Systems software enables the application software to interact with the computer, and helps the computer manage its internal and external resources. Systems software is required to run applications software; however, the converse is not true. Systems software can be classified into three types—utility programs, language translators, and the operating system. Utility programs are generally used to support, enhance, or expand the development of application programs. Examples consist of editors and programs for merging



files. A language translator or compiler is a software program that translates a program written in a high-level language such as C into machine language, which the hardware can directly execute. Thus a compiler provides the end user with the capability to write programs in a high-level language.

**Operating System:** The operating system is a major component of the systems software. Desktop operating systems allocate and control the use of all hardware resources: the processor, the main memory, and the peripheral devices. They also add a variety of new features, above and beyond what the hardware provides. Running the shell provides the end user with a more “capable” machine, in that the computer system provides direct capability to specify commands by typing them on a keyboard. The GUI (graphical user interface) goes one step further by providing the user with a graphical view of the desktop, and letting the user enter commands by *clicking* on icons. The multitasking feature of the OS provides the user with the capability to run multiple tasks “concurrently”. The file system of the OS provides the user with a structured way of storing and accessing “permanent” information. The operating system is thus an important part of most computer systems because it exerts a major influence on the overall function and performance of the entire computer. Normally, the OS is implemented in software, but there is no theoretical reason why it could not be implemented in hardware!

**Device Driver (Software Driver):** Most application programs need to access input/output devices and storage devices such as disks, terminals, and printers. Allowing these programs to perform the low-level IO activity required to directly control an input/output device is not desirable for a variety of reasons. First, most application programmers would find it extremely difficult to do the intricate actions required to directly control an IO device. Second, inappropriate accesses of the IO devices by amateur or malicious programmers can wreck plenty of havoc. The standard solution adopted in computer systems is therefore to provide a more abstract interface to the application programmer, and let an interface program perform the required low-level IO activity. This interface program is called a device driver or **software driver**. Each device requires specific device driver software, because each device has its own specific commands whereas an application program uses generic commands. The device driver receives generic commands from the application program and converts them into the specialized commands for the device, and vice versa.

### 1.3.3 Starting the Computer System: The Boot Process

Now that you have a good understanding of the role of an operating system in a modern computer, it would be interesting to learn how the operating system is activated each time a computer is turned on. When a computer is turned off, the data in the registers and memory are lost. Thus when the computer is turned on, the OS program is not residing in the main memory, and needs to be brought into main memory from a storage device such as

a diskette or hard disk. In modern computers, this copying is done by executing a program called the *bootstrap program* or *boot program* for short. How can the computer execute this copy program if the memory contains no useful contents? To solve this dilemma, a portion of the memory is implemented using non-volatile memory devices such as a read-only memory (ROM). This memory contains the boot program. When the computer is turned on, it starts executing instructions from the starting address of the boot program. The boot program contains code to perform diagnostic tests of crucial system components and load the operating system from a disk to the main memory. This bootstrap loader may be comprehensive enough to copy the nucleus of the operating system into memory. Or it may first store a more comprehensive loader that, in turn, installs the nucleus in memory. Once loaded, the OS remains in main memory until the computer is turned off.

For copying the OS from a disk drive to the RAM, the computer needs to know how the disk has been formatted, i.e., the number of tracks and sectors and the size of each sector. If information about the hard disk were stored in the ROM, then replacing the hard disk becomes a difficult proposition, because the computer will not be able to access the new hard disk with information about the old disk. Therefore, a computer must have a semi-permanent medium for storing boot information, such as the number of hard disk drive cylinders and sectors. For this purpose, it uses CMOS (complementary metal oxide semiconductor) memory, which requires very little power to retain its contents and can therefore be powered by battery. The battery helps the CMOS memory to retain vital information about the computer system configuration, even when the computer is turned off. When changing the computer system configuration, the information stored in the CMOS memory must be updated, either by the user or by the *plug-and-play* feature.??

### 1.3.4 Computer Network

Till now we were mostly discussing *stand-alone computers*, which are not connected to any computer network. Most of today's desktop computers are instead connected to a network, and therefore it is useful for us to have a brief introduction to this topic. A *computer network* is a collection of computers and other devices that communicate to share data, hardware, and software. Each device on a network is called a *node*. A network that is located within a relatively limited area such as a building or campus is called a *local area network* or *LAN*, and a network that covers a large geographical area is called a *wide area network* or *WAN*. The former is typically found in medium-sized and large businesses, educational institutions, and government offices. Different types of networks provide different services, use different technology, and require users to follow different procedures. Popular network types include Ethernet, Token Ring, ARCnet, FDDI, and ATM.

#### Give a figure here

A computer connected to a network can still use all of its *local resources*, such as hard drive, software, data files, and printer. In addition, it has access to *network resources*, which typically include *network servers* and *network printers*. Network servers can serve as

a *file server*, *application server*, or both. A file server serves as a common repository for storing program files and data files that need to be accessible from multiple workstations—*client* nodes—on the network. When an individual client node sends a request to the file server, it supplies the stored information to the client node. Thus, when the user of a client workstation attempts to execute a program, the client's OS sends a request to the file server to get a copy of the executable program. Once the server sends the program, it is copied into the memory of the client workstation, and the program is executed in the client. The file server can also supply data files to clients in a similar manner. An application server, on the other hand, runs application software on request from other computers, and forwards the results to the requesting client.

In order to connect a computer to a network, a *network interface card (NIC)* is required. This interface card sends data from the workstation out over the network and collects incoming data for the workstation. The NIC for a desktop computer can be plugged into one of the expansion slots in the motherboard. The NIC for a laptop computer is usually a PCMCIA card. Different types of networks require different types of NICs.

A computer network requires a *network operating system* to control the flow of data, maintain security, and keep track of user accounts. Commonly used operating systems such as UNIX, Windows XP, and Windows Vista already include networking capability. There are also software packages such as *Novell Netware* that are designed exclusively for use as network operating system. A network operating system usually has two components: server software and client software. The server software is installed in the server workstation; it has features to control file access from the server hard drive, manage the print queue, and track network user data such as userids and passwords. The client software is installed on the local hard drive of each client workstation; it is essentially a device driver for the NIC. When the client workstation boots up, the network client software is activated and establishes the connection between the client and the other devices on the network.

## 1.4 Trends in Computing

Computer systems have undergone dramatic changes since their inception a few decades ago. It is difficult to say whether it is the hardware that drives the software or if it is the other way around. Both are intimately tied to each other; trends in one do affect the other and vice versa. We shall first discuss trends in hardware technology.

### 1.4.1 Hardware Technology Trends

Ever since transistors began to be integrated in a large scale, producing LSI and VLSI (Very Large Scale Integration) circuits, there have been non-stop efforts to continually reduce the transistor size. Over the last three decades, the feature size has decreased nearly by a factor of 100, resulting in smaller and smaller transistors. This steady decrease in transistor sizes,

coupled with occasional increases in die sizes, have resulted in more and more transistors being integrated in a single chip. This has translated .....

In 2008, Intel® released the first processor chip that integrates more than 2 billion transistors—the quad-core Tukwila. As of 2008, it is also the biggest microprocessor made, with a die size of  $21.5 \times 32.5\text{mm}^2$ .

Below we highlight some of the main trends we see in hardware technology today:

- **Clock speed:** Clock speed had been steadily increasing over the last several decades; however, the current trend hints more of a saturation. In 2007, IBM released the dual-core Power6 processor, which operates at an astonishing 4.7 GHz clock.
- **Low-power systems:** In the late 1990s, as the number of transistors as well as the clock speed steadily increased, power consumption—especially power density—began to increase at an alarming rate. High power densities translated to higher temperatures, necessitating expensive cooling technologies. Today, power consumption has become a first-order design constraint, and power-aware hardware designs are commonplace.
- **Large memory systems:** Memory size has always been increasing steadily, mirroring the downward trend in price per bit. However, memory access time increases with size, necessitating the use of cache memories to reduce average memory latencies. Nowadays, it is commonplace to see multi-level cache organizations in general-purpose computers.
- **Multi-core processors:** The current trend is to incorporate multiple processor cores on the same die. These cores parallelly execute multiple threads of execution.
- **Embedded systems:** Although embedded systems have been around for a while, their popularity has never been as high as it is today. Cell phones, automobile controls, computer game machines — you name it! — all have become so sophisticated, thanks to advances in embedded system technology.

Some of these trends become clear when we look at the microprocessors developed over the last four decades for desktop systems by one of the major processor manufacturers, Intel®. Table 1.2 succinctly provides various features of these processors.

### 1.4.2 Software Technology Trends

As processors and memory became smaller and faster — providing the potential for significant boosts in performance — application programs and operating systems strived to offset that benefit by becoming bigger and sluggish. The time to boot up a computer, for instance, has remained steady—if not increased—over the years. This does not mean that software

| Processor Name         | Word size | Intro Year | Feature size ( $\mu m$ ) | Die area ( $mm^2$ ) | Number of Transistors | Clock Freq. |
|------------------------|-----------|------------|--------------------------|---------------------|-----------------------|-------------|
| 4004                   | 4         | 1971       | 10                       | 13.5                | 2300                  | 108 KHz     |
| 8008                   | 8         | 1972       | 10                       |                     | 3500                  | 200 KHz     |
| 8080                   | 8         | 1974       | 6                        |                     | 6000                  | 2 MHz       |
| 8085                   | 8         | 1976       | 3                        |                     | 6500                  | 2 MHz       |
| 8086                   | 16        | 1978       | 3                        |                     | 29K                   | 4.77 MHz    |
| 80286                  | 16        | 1982       | 1.5                      |                     | 134K                  | 6 MHz       |
| Intel386 <sup>TM</sup> | 32        | 1985       | 1.5                      |                     | 275K                  | 16 MHz      |
| Intel486 <sup>TM</sup> | 32        | 1989       | 1                        | 81                  | 1.2M                  | 25 MHz      |
| Pentium®               | 32        | 1993       | 0.8                      | 294                 | 3.1M                  | 66 MHz      |
| Pentium®Pro            | 32        | 1995       | 0.35                     | 195                 | 5.5M                  | 200 MHz     |
| Pentium®II             | 32        | 1997       | 0.35                     | 131                 | 7.5M                  | 300 MHz     |
| Pentium®III            | 32        | 1999       | 0.18                     | 106                 | 28M                   | 733 MHz     |
| Pentium®4              | 32        | 2000       | 0.18                     | 217                 | 42M                   | 2 GHz       |
| Pentium®D              | 64        | 2005       | 0.09                     | 206                 | 230M                  | 3.2 GHz     |

Table 1.2: Progression of Intel®Microprocessors Designed for Desktop Systems

technology has made no progress. On the contrary, there have been tremendous improvements in software application development. The driving philosophy in software technology development has also been performance and efficiency, *but of the programmer!*.

Below we highlight a few of the current trends in software applications:

- **Application Nature:**

- Multimedia
- Graphics
- Bioinformatics
- Web-based

- **Programming Methodology and Interface:**

- Object-oriented programming: Java, C#
- Visual programming: Scripting, HTML
- Multi-threading
- Application Programming Interface (API): POSIX, Windows API

- **Operating System:**

- Linux

- Windows Vista
- Mac OS X
- **Security:**

## 1.5 Software Design Issues

A good software piece is not one that just works correctly. Modularity, simplicity,

## 1.6 Hardware Design Issues

Among the two phases in the life of a program — its development and execution — hardware designers are concerned with the execution phase. As we will see in Section 1.8, hardware design is carried out in various stages, at different levels of abstraction. When designing hardware, the factors that stand at the forefront are performance, power consumption, size, and price; well designed hardware structures are those that have adequate performance and long battery life (if running off a battery), and are compact and affordable. Other issues that become important, depending on the computing environment, are binary compatibility, reliability, and security.

### 1.6.1 Performance

The speed with which computer systems execute programs has always been a key design parameter in hardware design. We can think of two different metrics when measuring the speed of a computer system: *response time* and *throughput*. Whereas response time refers to how long the computer takes to do an activity, throughput refers to how much the computer does in unit time. The response time is measured by time elapsed from the initiation of some activity until its completion. A frequently used response time metric is *program execution time*, which specifies the time the computer takes for executing the program once. The execution time of a program,  $ET$ , can be expressed as the product of three quantities: (i) the number of instructions executed or instruction count ( $IC$ ), (ii) the average number of clock cycles required to execute an instruction or cycles per instruction ( $CPI$ ), and (iii) the duration of a clock cycle or cycle time ( $CT$ ). Thus,

$$ET = IC \times CPI \times CT$$

Although this simple formula seems to provide a good handle on program execution time, and therefore on computer performance, the reality is not so simple! The instruction count of a program may vary depending on the data values supplied as input to the program. And, the cycles per instruction obtained may vary depending on what other programs are simultaneously active in the system<sup>8</sup>. Finally, we have computer systems that dynamically adjust

---

<sup>8</sup>Nowadays, almost all computer systems execute multiple programs at the same time.

the clock cycle time—*dynamic frequency scaling*—in order to reduce power consumption.

While reporting program execution time, the standard practice used to deal with the first problem is to measure the execution time with a standard set of input data. The second and third problems are avoided by not running only one benchmark program at a time and by not exercising dynamic frequency scaling.

Throughput, the other metric of performance, specifies the number of programs, jobs, or transactions the computer system completes per unit time. If the system completes  $C$  programs during an observation period of  $T$  seconds, its throughput  $X$  is measured as  $C/T$  programs/seconds. For processors, a more commonly used throughput measure is the number of instructions executed in a clock cycle, referred to as its *IPC* (instructions per cycle).

Although throughput and response time are related, improving the throughput of a computer system does not necessarily result in reduced response time. For instance, the throughput of a computer system improves when we incorporate additional processing cores and use these cores for executing independent tasks, but that does not decrease the execution time of any single program. On the other hand, replacing a processor with a faster one would invariably decrease program execution time as well as improve throughput.

### 1.6.2 Power Consumption

After performance, power consumption is perhaps the biggest design issue to occupy the hearts and minds of computer designers. In fact, in some application domains, power consumption has edged out performance as the most important design factor. Why is power such an important issue? This is because it directly translates to heat production. Most of the integrated circuits will fail to work correctly if the temperature rises beyond a few degrees.

Again, the designer's goal is to reduce the power consumption occurring during program execution, as program development can be carried out in a different system where power consumption may not be a burning issue.

Power consumption has two components: *dynamic* and *static*. Dynamic power relates to power consumed when there is switching activity (or change of state) in a system, whereas static power relates to power consumed even when there is no switching activity in the system. Dynamic power is directly proportional to the extent of switching activity in the system and the clock frequency of operation. It is also proportional to the capacitance in the circuits and wires, and to the square of the supply voltage of the circuits.

Static power consumption occurs due to leakage currents in the system. With continued scaling in transistor technology—reduction in transistor sizes—static power consumption is becoming comparable to dynamic power consumption. Static power consumption is also related to the supply voltage. Therefore, to develop low-power systems, computer hardware designers strive to reduce the supply voltage of the circuits as well as reduce the amount of

hardware used to perform the required functionality.

### 1.6.3 Price

Price is an important factor that makes or breaks the success of any computer system. Between two comparable computer systems, all things being equal, price will be an important factor. The major factors affecting the price are design cost, manufacturing cost, and profit margin, all of which may be impacted by the sales volume. In general, price increases exponentially with the complexity of the system. Therefore, it is imperative to reduce hardware complexity at all costs.

### 1.6.4 Size

Size is an important design consideration, especially for laptops and embedded systems.

### 1.6.5 Summary

From the above discussion, it is apparent that design the computer hardware is a complex process, where one has to focus on several factors at the same time. Often, focusing on one factor comes at the expense of others. For instance, attempting to improve performance by using substantial amounts of hardware generally results in high power consumption as well as size and price. A good design will attempt to achieve good performance without increase in hardware complexity, thereby conserving power, and reducing the size and price as well.

## 1.7 Theoretical Underpinnings

### 1.7.1 Computability and the Turing Machine

The fledgling days of computers saw them only solving problems of a numerical nature; soon they began to process various kinds of information. A question that begs an answer is: What kinds of problems can a computer solve? The answer, as per computer science theory, is that given enough memory and time, a computer can solve all problems for which a finite solution algorithm exists. One of the computer pioneers who defined and formalized computation was the British mathematician Alan Turing. While a graduate student at Princeton University in 1936, Turing published a seminal paper titled “On Computable Numbers with an Application to the Entscheidungsproblem,” which laid a theoretical foundation for modern computer science. In that paper he envisioned a theoretical machine, which later became known as a *Turing Machine*, that could read instructions from a punched paper tape and perform all the critical operations of a computer. One of Turing’s remarkable achievements was to prove that a *universal Turing machine*—a simulator of Turing machines—can



perform every reasonable computation [?]. If given a description of a particular Turing machine  $TM$ , the universal Turing machine simulates all the operations performed by  $TM$ . It can do anything that any real computer can do, and therefore serves as an abstract model of all general-purpose computers. Turing's paper also established the limits of computer science by mathematically demonstrating that some problems do not lend themselves to algorithmic representations, and therefore cannot be solved by any kind of computer.

### 1.7.2 Limitations of Computers

*“Computers are useless. They can only give you answers.”*  
 — Pablo Picasso (1881 - 1973).

For the computer to solve a problem, it is imperative to first develop a solution algorithm, or step-by-step procedure, for the problem. Although a general-purpose computer can be used to solve a wide variety of problems by executing appropriate algorithms, there are certain classes of problems that cannot be solved using a computer, either in principle or in practice! Such problems may be grouped into three categories:

- Undecidable or non-computable problems
- Unsolvable problems
- Intractable problems

#### Undecidable or Non-computable

**Problems:** This category includes problems that have been proven not to have finite solution algorithms. Kurt Gödel, a famous mathematician, proved in the 1930s his famous *incompleteness theorem* [ref]. An important consequence of Gödel's theorem is that there is a limit on our ability to answer questions about mathematics. If we have a mathematical model as complex as the set of integers, then there is no algorithmic way by which true statements can be distinguished from false ones. In practical terms, this means that not all problems have an algorithmic solution, and therefore a computer cannot be used to solve any arbitrary problem. In particular, a computer cannot find proofs in sufficiently complex systems. A. M.

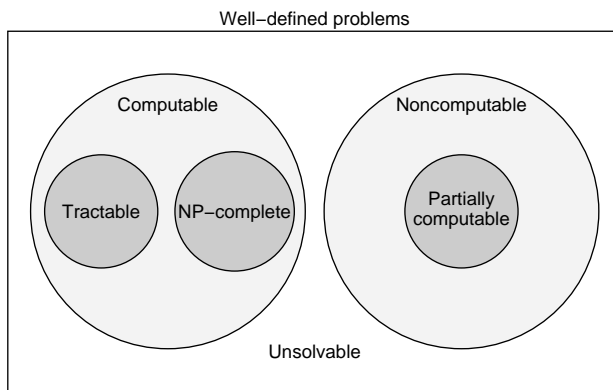


Figure 1.9: Categorization of Well-Defined Problems based on Computability

Turing and Alonzo Church demonstrated a set of undecidable problems in 1936. One of these is what has become known as the *Turing machine halting problem*, which states that no algorithm exists to determine if an arbitrary Turing machine with arbitrary input data will ever halt once it has started working. A practical implication of this result is that given a sufficiently complex computer program with loops, it is impossible to determine if under certain inputs the program will ever halt. Since the 1930s, a number of other problems have been proven to be undecidable. It will never be possible in a logically consistent system to build a computer, however powerful, that by manipulating symbols can solve these undecidable problems in a finite number of steps!

**Unsolvable Problems:** This category includes well-defined problems that have not been proved to be undecidable, but for which no finite algorithm has yet been developed. An example is *Goldbach's conjecture*, formulated by the 18th century mathematician Christian Goldbach. The conjecture states that every even integer greater than 2 is the sum of exactly two prime numbers. Although this conjecture has been verified for a large number of even integers, it has not yet been proved to be true for *every* even integer, nor has any finite algorithm been developed to prove this conjecture. An algorithm that examines all even integers is not finite, and therefore will not terminate. An unsolved problem may eventually be solved or proved to be undecidable.

**Intractable Problems:** This category includes problems that have a finite solution algorithm, but executing the best available algorithm requires unreasonable amounts of time, computer memory, and/or cost. In general, this is the case when the complexity of the best available algorithm grows exponentially with the size of the problem. An example of an intractable problem is the *traveling salesman problem*. The objective in this problem is to find a minimum-distance tour through a given set of cities. The best available solution algorithms for this problem are exponential in  $n$ , where  $n$  is the number of cities in the tour. This means that the execution time of the algorithm increases exponentially as  $n$  increases. For reasonably large values of  $n$ , executing such an algorithm becomes infeasible. Many problems that occur in real life are closely related to the traveling salesman problem; two common examples are the scheduling of airline flights, and the routing of wires in a VLSI chip. An intractable problem becomes more tractable with technological advances that make it feasible to design faster computers. Algorithm developers often tackle intractable problems by devising approximate or inexact solution algorithms. These approximate algorithms often involve the use of various heuristics, and are near-optimal most of the time. *Simulated annealing* is an example of such an algorithm. Recent research seems to indicate that *quantum computing* has the potential to solve many of the intractable problems more efficiently.

## 1.8 Virtual Machines: The Abstraction Tower

If we look at the computer as a physicist would do, we will see that a digital computer executes an algorithm by controlled movement of electrons through silicon substrates and metal wires. A complete description of a computer could be given in terms of all of its silicon substrates, impurity dopings, wire connections, and their properties. Such a view, although very precise, is too detailed even for computer hardware designers, let alone the programmers. Hardly any programs would have been written in all these years if programmers were given such a specification!

Like many other machines built today, computers are incredibly complex. The functions involved in developing programs and in designing the hardware to execute them are so diverse and complex that it is difficult for a user/programmer/designer to have mastery of all of the functions. A practical technique for dealing with complexity in everyday life is *abstraction*<sup>9</sup>. An automobile driver, for instance, need not be concerned with the details of how exactly the automobile engine works. This is possible because the driver works with a high-level abstract view of the car that encapsulates the essentials of what is required for driving the vehicle. The car mechanic, on the other hand, has a more detailed view of the machine. In a similar manner, abstraction is used to deal with the complexity of computers. That is, computer software and hardware can be viewed as a series of *architectural abstractions* or *virtual machines*. Different users see different (virtual) machines depending on the level at which they use the computer. For instance, a high-level language programmer sees a virtual machine that is capable of executing statements specified in a high-level language. An assembly language programmer, on the other hand, sees a different machine with registers and memory locations that can execute instructions specified in an assembly language. Thus, the study of a computer system is filled with abstractions. There is yet another advantage to viewing the computer at several abstraction levels. Programs developed for a particular abstraction level can be executed in different platforms—which differ in speed, cost, and power consumption—that implement the same abstract machine.

The user who interacts with a computer system at a particular abstraction level has a view of what its capabilities are at this level, and this view results directly from the functions that the computer can perform at this level. Conceptually, each architectural abstraction is a set of rules that describes the logical function of a computer as observable by a program running on that abstract machine. The architecture does not specify the details of exactly how its functions will be performed; it only specifies the architecture's functionality. Implementation issues related to the functionality are left to the lower-level machines. The

---

<sup>9</sup>An abstraction is a representation that hides details so that one can focus on a few concepts at a time. Formal abstractions have a well-defined syntax and semantics. Hence, they provide a way of conveying the information about a system in a consistent way that can be interpreted unambiguously. This abstraction (or specification) is like a contract: it defines how the system behaves. The abstraction defines how the outside world interacts with the system. The *implementation*, on the other hand, defines how the system is built, as seen from the inside.

entire point of defining each architectural abstraction is to insulate programmers of that level from those details. The instruction set architecture, for instance, provides a level of abstraction that allows the same (machine language) program to be run on a family of computers having different implementations (i.e., microarchitectures).

*“There are many paths to the top of the mountain,  
but the view is always the same.”  
— Chinese Proverb*

In order to make it easier for comprehension purposes, we have organized the computer virtual machines along a single dimension as an abstraction tower, with one machine “above” the other. Each virtual machine except the one at the lowest level is implemented by the virtual machine below it. This approach is called *hierarchical abstraction*. By viewing the computer as a hierarchy of abstractions, it becomes easier to master the complexity of computers and to design computer systems in a systematic, organized way.

Appropriate interfaces are used to specify the interaction between different abstractions. This implementation is done by translating or interpreting the steps or instructions of one level using instructions or facilities from the lower level. A particular computer designer or user needs to be familiar only with the level at which he/she is using the computer. For instance, a programmer writing a C program can assume that the program will be executed on a virtual machine that directly executes C programs. The programmer need not be concerned about how the virtual machine implements C’s semantics. Similarly, in a multitasked computer system, each active program sees a separate virtual machine, although physically there may be only a single computer! Some of the machine levels themselves can be viewed as a collection of multiple abstractions. One such breakdown occurs in the assembly-level language machine where we further break it into User mode and Kernel mode.

Figure 1.10 depicts the principal abstraction levels present in modern computers. In the figure the planes depict the abstract machines. For each machine level, we can write programs specific to that level to control that machine. The solid blocks depict the people/software/hardware who transform a program for one machine level to a program for the machine level below it. For the sake of clarity and to put things in proper perspective, the figure also includes a few levels (at the top) that are currently implemented by humans. It is important to note that these abstract machines are somewhat different from the virtual machines seen by end users when they run different programs on a computer system. For instance, when you run a MIPS assembler program on a host computer system, you do not “see” that host as a MIPS assembly-level machine or a MIPS ISA-level machine. Instead, your view of the host is simply that of a machine capable of taking a MIPS assembly language program as input, and churning out an equivalent MIPS machine language program as output! You can even run that assembler program without knowing anything about the MIPS assembly language or machine language. The person who wrote the MIPS assembler, on the other hand, does see the MIPS assembly-level architecture as well as the MIPS

ISA. Finally, the MIPS assembler program may have been originally written in an HLL, for example C, in which case its developer also sees an abstract C machine that takes as commands C statements!

**The Use of a Language for an Abstraction:** Each virtual machine level provides an abstraction that is suitable for the computer user/designer working at that level. In order to make use of a particular abstraction, it must be possible to specify commands/instructions to that virtual machine. Without a well-defined language, it becomes difficult to specify a program of reasonable size and complexity. Most of the abstraction levels therefore provide a separate language to enable the user at that level to specify the actions to be performed by the virtual machine. The language specifies what data can be *named* by a program at that level, what operations can be performed on the named data, and what ordering exists among the operations. The language must be rich enough to capture the intricacies of the corresponding virtual machine. When describing each of the abstraction levels, we will show how the language for that level captures the essence of that level, and how it serves as a vehicle to represent the commands specified by the user of that level.

*“The limits of my language mean the limits of my world.”*  
 — L. Wittgenstein. *Tractatus Logico-Philosophicus*

**Translators, Interpreters, Emulators, and Simulators:** An important aspect of the layered treatment of computers is that, as already mentioned, the commands specified at each abstraction level need to be converted to commands specific to the immediately lower level. Such a transformation makes the computer behave as a different machine than the one for which the original program was written. This transformation process can be done by translation or interpretation. In computer parlance, the term translation indicates taking a *static* program or routine, and producing a functionally equivalent static program, usually at a lower level. A static program is one that is not being executed currently. Thus, translation of a loop involves translating each command of the loop exactly once. The term interpretation, on the other hand, indicates taking individual steps of a *dynamic* program and producing an equivalent sequence of steps, usually at a lower level. A dynamic program is one that is in the process of being executed. Thus, interpretation of a loop involves interpreting each command of the loop multiple times, depending on the number of times the loop gets executed. Because of this dynamic nature, an interpreter essentially makes one machine (the host machine) appear as another (the target machine). A translator is almost always implemented in software, whereas an interpreter is implemented in software or hardware. A software interpreter is often called a *simulator*, and a hardware interpreter is often called an *emulator*. Of course, it is possible to build interpreters that use a combination of software and hardware techniques. A simulator is often used to illustrate the working of a virtual machine. It is also used to allow programs compiled for one machine to execute on another machine. For instance, a simulator can execute programs written for an older

machine on a newer machine.

Advantages of using interpretation include (i) the ability to execute the (source) program on different platforms, without additional compilation steps, and (ii) the ease of carrying out interactive debugging. The main disadvantage is performance.

### 1.8.1 Problem Definition and Modeling Level Architecture

*“At the highest level, the description is greatly chunked, and takes on a completely different feel, despite the fact that many of the same concepts appear on the lowest and highest levels.”*

— Douglas R. Hofstadter, in *Gödel, Escher, Bach: An Eternal Golden Band*

The highest abstraction level that we can think of is the level at which problem definition and modeling are done. At this level, we can view the computer system as a machine that can solve well-defined computer problems. We loosely define a well-defined computer problem as one that can be represented and manipulated inside a computer.

The problem modeling person, therefore, takes complex real-life problems and precisely formulates them so as to be solved on the computer. This process involves representing the real-life problem’s data by some form of data that can be manipulated by a computer. This process is called *abstraction* or *modeling*—creating the right model for the problem so as to make it possible to eventually develop an appropriate algorithm to solve it. Notice that in this context, modeling often implies simplification, the replacement of a complex and detailed real-world situation by a comprehensible model within which we can solve a problem. That is, the model captures the essence of the problem, “abstracting away” the details whose effect on the problem’s solution is nil or minimal.

Almost any branch of mathematics or science may be utilized in the modeling process. Problems that are numerical in nature are typically modeled by mathematical concepts such as simultaneous linear equations (e.g., finding currents in electrical circuits, or finding stresses in frames made of connected beams) and differential equations (e.g., predicting population growth, or predicting the rate at which chemicals will react). Several problems can be modeled as graph theoretical problems. Symbol and text processing problems can be modeled by character strings and formal grammars. Once a problem is formalized, the algorithm developer at the lower level can look for solutions in terms of a precise model and determine if an algorithm already exists to solve that problem. Even if there is no known solution, knowledge of the model properties might aid in developing a solution algorithm.

### 1.8.2 Algorithm-Level Architecture

The architecture abstraction below the problem definition level is the algorithm-level architecture. At this level, we see the computer system as a machine that is capable of executing

algorithms. An algorithm, as we saw earlier, is a step-by-step procedure that can be carried out mechanically so as to get a specific output from a specific input. A key feature of computer algorithms is that the steps are precisely defined so as to be executed by a machine. In other words, it describes a process so unambiguously that it becomes mechanical, in the sense that it does not require much intelligence, and can be performed by rote or a machine. Computer scientists also require that an algorithm be *finite*, meaning that (i) the number of steps must be finite so that it *terminates* eventually, and (ii) each step must require only finite time and computational resources.

The basic actions involved in a computer algorithm are:

- Specify data values (using abstract data types)
- Perform calculations and assign data values
- Test data values and select alternate courses of actions including repetitions
- Terminate the algorithm

The algorithm-level architecture supports *abstract data types* and *abstract data structures*; the algorithm designer formulates suitable abstract data structures and develops an algorithm that operates on the data structures so as to solve the problem. Providing abstract data types enables the algorithm designer to develop more general algorithms that can be used for different applications involving different data types. This is often called *algorithm abstraction*. For instance, a sorting algorithm that has been developed without specifying the data types being sorted, can be programmed to sort a set of integers or a set of characters. Similarly, when considering a data structure, such as an array, it is often more productive to ignore certain details, such as the exact bounds of its indices. This is often called *data abstraction*.

**Algorithm Efficiency:** Computer theorists are mainly concerned with discovering the most efficient algorithms for a given class of problems. The algorithm's efficiency relates its resource usage, such as execution time or memory consumption, to the size of its input data,  $n$ . The efficiency is stated using the "Big O" notation,  $O(n)$ . For example, if an algorithm takes  $4n - 2n + 2$  steps to solve a problem of size  $n$ , we can say that the number of steps is  $O(n)$ . Programmers use their knowledge of well-established algorithms and their respective complexities to choose algorithms that are best suited to the circumstances. Examples of such algorithms are *quick-sort* for sorting data (which has an  $(n \log n)$  average running time), and *binary search* for searching through sorted data (which has an  $O(\log n)$  time).

Algorithms can be specified in different ways. Two common methods are *pseudocode* descriptions and *flowchart* diagrams. A pseudocode description uses English, mathematical notations, and a limited set of special commands to describe the actions of an algorithm. A flowchart diagram provides the same information graphically, using diagrams with a finite set of symbols in the place of the more elegant features of the pseudocode. A computer

cannot directly understand either pseudocode or flowcharts, and so algorithm descriptions are translated to computer language programs, most often by human programmers. Thus, a computer program is an embodiment of an algorithm; strictly speaking, an algorithm is a mental concept that exists independently of any representation.

### 1.8.2.1 Computation Models

Another important tenet of an algorithm-level architecture is the computational model it supports. A computational model conceptualizes computation in a particular way by specifying the kinds of primitives, relationships, and events that can be described in an algorithm. A computational model will generally have the following features:

- **Primitives:** They represent the simplest objects that can be expressed in the model. Examples of primitives found in most of the computation models are constants and variables.
- **Methods of combination:** They specify how the primitives can be combined with one another to obtain compound expressions.
- **Methods of abstraction:** They specify how compound objects can be named and manipulated as units.

The computational model determines the kind of computations that can be specified by an algorithm. For example, if we consider a geometric computational model that supports only ruler and compass construction primitives, then we can specify algorithms (rules) for bisecting a line segment, bisecting an angle, and other similar problems. We cannot, however, specify an algorithm to trisect an angle. For solving this problem, we require additional primitives such as a protractor. For arithmetic computation we can use models incorporating different primitives such as an abacus, a slide rule, or even a calculator. With each of these computation models, the type of arithmetic problems that can be solved is different. The algorithms for solving a specific problem would also be different.

Algorithm development is always done for a specific algorithm-level architecture having an underlying computational model. Three basic computational models are currently in use, some of them being more popular than the others: *imperative*, *functional*, and *logic*. These models of computation are equivalent in the sense that, in principle, any problem that has a solution in one model is solvable in every one of the other models also.

**Imperative Model:** The imperative model of computation is based on the execution of a sequence of instructions that modify storage called *state*. The basic concept is the notion of a machine state (comprising variables in the high-level architecture, or registers and memory locations in the assembly-level architecture). Program development consists of specifying a sequence of state changes to arrive at the solution. An imperative program



would therefore consist of a sequence of statements or instructions and side-effect-prone functions; the execution of each statement (or instruction) would cause the machine to change the value of one or more elements of its state, thereby taking the machine to a new state. A side-effect-prone function is one whose execution can result in a change in the machine state. Historically, the imperative model has been the most widely used model; most computer programmers start their programming career with this computational model. It is the closest to modeling the computer hardware. This tends to make it the most efficient model in terms of execution speed. Commonly used programming languages such as C, C++, FORTRAN, and COBOL are based on this computational model.

**Applicative (Functional) Model:** The functional model has its foundation in mathematical logic. In this model, computing is based on recursive function theory (RFT), an alternative (and equivalent) model of effective computability. As with the Turing machine model, RFT can express anything that is computable. Two of the prominent computer scientists who pioneered this computational model are Stephen Kleene and Alonso Church. The functional model consists of a set of values, functions, and the application of side-effect-free functions. A side-effect-free function is one in which the entire result of computation is produced by the return value(s) of the function. Side-effect-free functions can only access explicit input parameters; there are no global variables in a fully functional model. And, in the purest functional models, there are no assignment statements either. Functions may be named and may be composed with other functions. Functions can take other functions as arguments and return functions as results. Programs consist of definitions of functions, and computations are application of functions to values. A classic example of a programming language that is built on this model is LISP.

**Rule-based (Logic) Model:** The logic model of computation is a formalization of the logical reasoning process. It is based on relations and logical inference. An algorithm in this model involves a collection of *rules*, in no particular order. Each rule consists of an *enabling condition* and an *action*. The execution order is determined by the order in which the enabling conditions are satisfied. The logic model is related to relational data bases and expert systems. A programming language designed with this model in mind is Prolog.

**Computational Model Extensions:** Apart from the three popular computational models described above, many other computational models have been proposed. Many extensions have also been proposed to computational models to improve programmer efficiency or hardware efficiency. Two important such extensions are the object-oriented programming model and the concurrent programming model.

- **Object-Oriented Model:** In this model, an algorithm consists of a set of objects that compute by exchanging messages. Each object is bound up with a value and a set of operations that determine the messages to which it can respond. Functions

are thus designed to operate on objects. Objects are organized hierarchically. That is, complex objects are designed as extensions of simple objects; the complex object will “inherit” the properties of the simple object. The object-oriented model may be implemented within any of the other computational models. Imperative programming languages that use the object-oriented approach are C++ and Java.

- **Concurrent Model:** In this model, an algorithm consists of multiple processes or tasks that may exchange information. The computations may occur concurrently or in any order. The model is primarily concerned with methods for synchronization and communication between processes. The concurrent model may also be implemented within any of the other computational models. Concurrency in the imperative model can be viewed as a generalization of control. Concurrency is particularly attractive within the functional and logic models, as subexpression evaluation and inferences may then be performed concurrently. Hardware description languages (HDLs) such as Verilog and VHDL use the concurrency model, as they model hardware components, which tend to operate concurrently.

### 1.8.3 High-Level Architecture

The abstraction level below the algorithm-level architecture is the high-level architecture. This is the highest level that we study in this book, and is defined by different high-level languages, such as C, C++, FORTRAN, Java, LISP, Prolog, and Visual Basic. This level is used by application programmers and systems programmers who take algorithms and formally express them in a high-level language. HLL programmers who develop their own algorithms often perform both these steps concurrently. That is, the algorithm development is done side by side with HLL program development.

To the HLL programmer the computer is a machine that can directly accept programs written in a high-level language that uses alphabets as well as symbols like +, −, etc. It is definitely possible to construct a computer hardware that directly executes a high-level language; several *LISP machines* were developed in the 1970s and 1980s by different vendors to directly execute LISP programs. Directly running high-level programs on hardware is not commonplace, however, as the hardware can run only programs written in one specific high-level language. More commonly, programs written in high-level languages are translated to a lower level by translators known as *compilers*. We shall see more details of the high-level architecture in Chapter 2.

For a computer to solve a problem, the algorithm must be expressed in an unambiguous manner, so that computers can faithfully follow it. This implies expressing the algorithm as a *program* as per the syntax and semantics of a *programming language*.

#### 1.8.4 Assembly-Level Architecture

The next lower level, called the assembly-level architecture, implements the high-level architecture. The architecture at this level has a notion of storage locations such as registers and memory. Its instructions are also more primitive than HLL statements. An instruction may, for instance, add two registers, move data from one memory location to another, or determine if a data value is greater than zero. Primitive instructions such as these are sufficient to implement high-level language programs. The language used to write programs at this level is called an *assembly language*. In reality, an assembly language is a symbolic form for the language used in the immediately lower level, namely the *instruction set architecture*. Often, the assembly-level architecture also includes some instructions that are not present in the instruction set architecture.

The assembly-level architecture and the instruction set architecture are usually hybrid levels in that each of these architectures typically includes at least two modes—the User mode and the Kernel mode. Both modes have many common instructions; however, each mode also has a few instructions of its own. The extra instructions in the Kernel mode include, for instance, those for reading or writing to IO addresses, managing memory allocation, and creating multiple processes. The extra instructions in the User mode are called *system call* instructions. In the microarchitecture, these instructions are interpreted by executing an interpreter program in the Kernel mode at the ISA level. This interpreter program is called the *operating system kernel*. Notice that the operating system itself may have been originally written in a high-level language, and later translated to the lower levels. The instructions that are common to both modes are interpreted directly by the microarchitecture, and not by the OS. Thus, the system call instructions of the User mode are interpreted by the OS and the rest are interpreted directly by the microarchitecture.

#### 1.8.5 Instruction Set Architecture (ISA)

The next lower level is called *instruction set architecture (ISA)*. The language used to specify programs at this level is called a *machine language*. The memory model, IO model, and register model in the ISA are virtually identical to the ones in the assembly-level architecture. However, when specifying register and memory addresses in machine language, they are specified in binary encoding. The instructions in the ISA are also mainly binary encodings of the instructions present in the assembly-level architecture. There may be a few minor differences in the instruction set. Usually, the assembly-level architecture has more instructions than what is available in the ISA. Moreover, most assembly languages permit programmers to define their own *macros*. These enhancements make it much easier to program in an assembly language, compared to a machine language. Programs written in an assembly language are translated to machine language using a program called *assembler*.

The instruction set architecture is sometimes loosely called *architecture*. Different ISAs

differ in the number of operations, data types, and addressing modes they specify. ISAs that include fewer operations and addressing modes are often called RISC (Reduced Instruction Set Computer) ISAs. Those with a large repertoire of operations and addressing modes are often called CISC (Complex Instruction Set Computer) ISAs. The most commonly found ISA is the IA-32 ISA—more often known by its colloquial name, x86—introduced by Intel Corporation in 1979. Other ISAs that are in use today are IA-64, MIPS, Alpha, PowerPC, SPARC, and PA-RISC.

### 1.8.6 Microarchitecture

The microarchitecture is the abstraction level immediately below the ISA; it serves as a platform for interpreting machine language instructions. A microarchitecture specification includes the resources and techniques used to realize the ISA specification, along with the way the resources are organized to realize the intended cost and performance goals. At this level the viewer sees hardware objects such as *instruction fetch unit*, *register files*, *ALUs*, *latches*, cache memory, memory systems, IO interfaces, and *interconnections*. A register file is a collection of registers from which a single register can be read or written by specifying a register number. An ALU (Arithmetic Logic Unit) is a combinational logic circuit that is capable of performing simple arithmetic and logical operations that are specified in machine language instructions. The register file, the ALU, and the other components are connected together using bus-type or point-to-point interconnections to form a *data path*. The basic operation of the data path consists of fetching an instruction from main memory, decoding its bit pattern to determine what it specifies, and to carry out its execution by fetching the required operands, using the ALU to operate on the operand values, and storing back the result in the specified register or memory location.

The actual interpretation of the machine language instructions is done by a *control unit*, which controls and coordinates activities of the data path. It issues commands to the data path to fetch, decode, and execute machine language instructions one by one. There is a fundamental break at the instruction set architecture. Whereas the architectures above it are usually implemented by translation, the ISA and the architectures below it are always implemented by interpretation.

### 1.8.7 Logic-Level Architecture

Descending one level lower into the hardware, we get to the logic-level architecture. This architecture is an abstraction of the electronic circuitry of a computer, and refers to the actual digital logic and circuit designs used to realize the computer microarchitecture. The designer of this architecture uses *gates*, which accept one or more digital inputs and produce as output some logical function of the inputs. Several gates can be connected to form a multiplexer, decoder, PLA, or other combinational logic circuits such as an adder. The outputs of an adder, subtractor, and other functional units can be passed through a multiplexer

to obtain an ALU. Similarly, a few gates can be connected together with some feedback arrangement to form a flip-flop or 1-bit memory, which can be used to store a 0 or a 1. Several flip-flops can be organized to form a *register*, and several registers can be organized to form a *register file*. Memory systems are built in a similar manner, but on a much larger scale. Thus, we use a hierarchical approach for implementing the individual blocks of the microarchitecture in the logic-level architecture. We will examine gates and the logic-level architecture in detail in Chapter 9 of the book. This is the lowest architecture abstraction that we will study in detail in this book.

**Synchronous vs Asynchronous (self-timed):** Currently digital computers are typically designed as *synchronous* or *clocked* sequential circuits, meaning that they use clock signals to co-ordinate and synchronize the different activities in the computer. Changes in the machine *state* occur only at discrete times that are co-ordinated by the clock signals. Thus, the basic speed of a computer is determined by the time of one clock period. The clock speed of the processor used in the original IBM PC was 4.77 MHz. The clock speeds in current state-of-the-art computers range from 1 to 3.8 GHz. If all other specifications are identical, higher clock speeds mean faster processing. An alternative approach is to use *asynchronous* or *self-timed* sequential circuits.

### 1.8.8 Device-Level Architecture

For the sake of completeness, we mention the existence of a machine level below the logic-level architecture, called *device-level architecture*. The primitive objects at this level are devices and wires. The prevalent devices in today's technologies are *transistors*. The designer of this architecture uses individual transistors and wires to implement the digital logic circuits specified in the logic-level architecture. The designer also specifies how the transistors and wires should be laid out. With today's technology, millions and millions of transistors can be integrated in a single chip; such a design is called VLSI (Very Large Scale Integration). Accordingly, device-level architecture is also called *VLSI architecture*.

One possible way of designing a device-level architecture involves taking the logic-level architecture and implementing it as follows: connect a few transistors together to form device-level circuitry that implement logic gates such as inverter, AND, OR, NAND, and NOR. Then, implement each logic gate in the logic-level architecture by the equivalent device-level circuitry. In current practice, a different approach is taken. Instead of attempting to implement the logic-level architecture, the VLSI designer takes the microarchitecture, and implements the functional blocks in the microarchitecture with device-level circuitry. By bypassing the logic-level architecture, this approach leads to a more efficient design.

Different types of transistor devices are available: BJT (Bipolar Junction Transistor), MOSFET (Metal Oxide Semiconductor Field Effect Transistor), etc. Digital computer applications invariably use MOSFETs. Again, different design styles are available with MOSFETs. The most prevalent style is the CMOS (Complementary Metal Oxide Semicon-

ductor) approach, which uses a PMOS network and a complementary NMOS network.

If we want to study the design of transistors, that leads us into *solid-state physics*, which deals with low-level issues such as electrons, holes, tunneling, and quantum effects. At this low abstraction level the machine looks more analog than digital! This level is clearly outside the scope of this book.

## 1.9 Concluding Remarks

We have barely scratched the surface of computing, but we have laid a solid foundation for computing and computers. This chapter began with the general topic of computing and the role of computers in computing applications. Perhaps now we can answer the question: what exactly is a computer? To a large extent, the answer depends on the level at which we view the computer. At the high-level view, it is a machine that accepts high-level language programs, and directly executes them. At the logic-level view, computers are digital electronic circuits consisting of different types of gates that process 0s and 1s. Viewed in the above light, we arrive at a definition of computer architecture as being concerned with the design and application of a series of virtual machines, starting from high-level architecture to logic-level architecture.

Besides the abstractions described in the previous section, a computer can have additional abstractions, such as user interfaces and data communication facilities. The key thing to remember is that computers are generally designed as a series of architectural abstractions, each one implementing the one immediately above it. Each architecture represents a distinct abstraction, with its own unique objects and operations. By focusing on one architecture at a time, we are able to suppress irrelevant details, thereby making it easier to master this complex subject. All of the architectures are important for mastery of the subject; this textbook studies four architectures in detail: the assembly-level architecture, the instruction set architecture, the microarchitecture, and the logic-level architecture. A synthesis of these four architectures will give a depth and richness of understanding that will serve well, irrespective of whether your main interest is in computer science, computer engineering, or electrical engineering.

## 1.10 Exercises

1. Explain the role played by the operating system in a computer.
2. What is meant by a *virtual machine* in the context of computers?
3. Explain what is meant by the *stored program* concept.

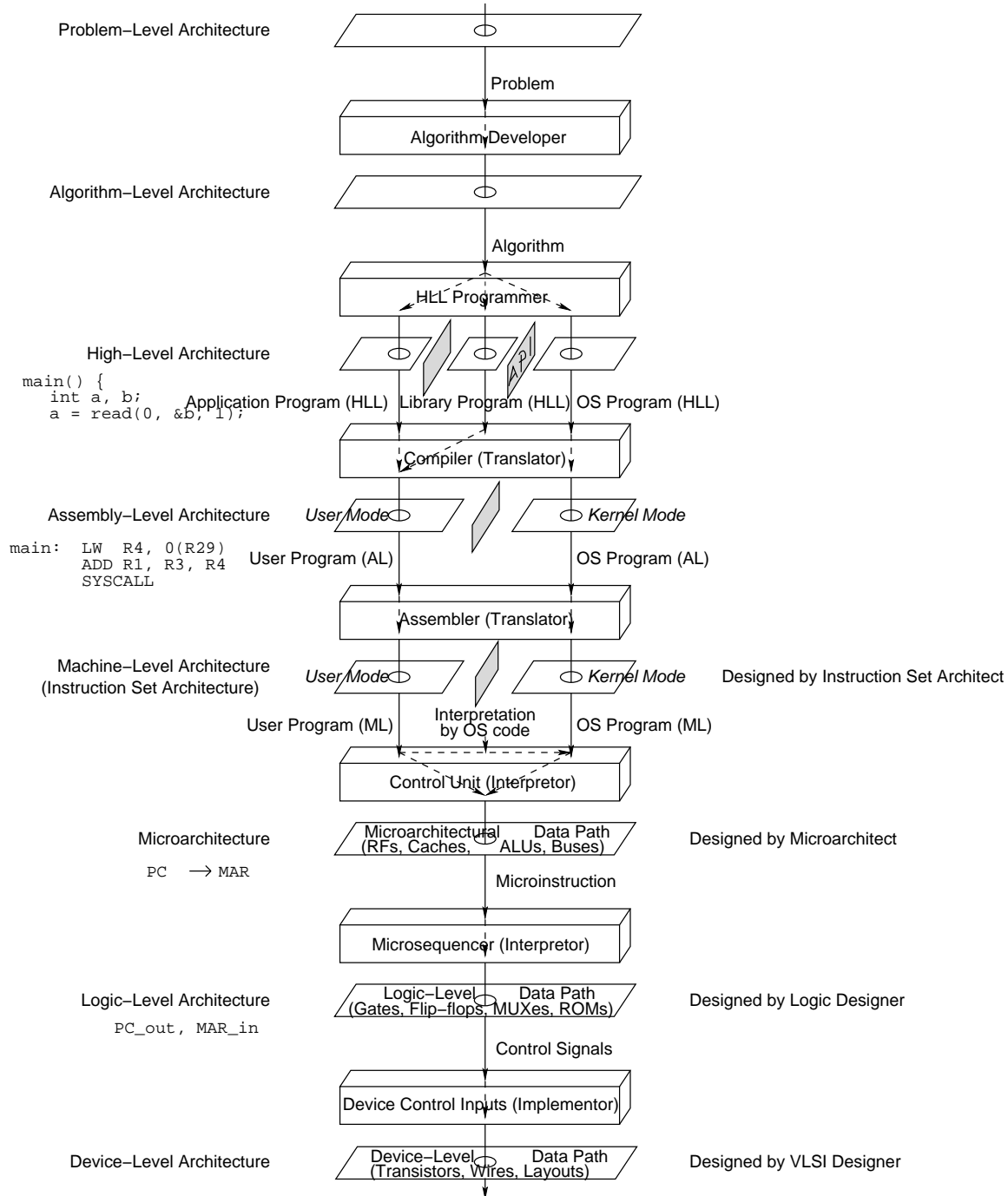


Figure 1.10: Machine Abstractions relevant to Program Development and Execution, along with the Major Components of each Abstract Machine.

# Part I

## PROGRAM DEVELOPMENT — SOFTWARE LEVELS

*Finish your outdoor work and get your fields ready; after that, build your house.*

**Proverbs 24: 27**

This part deals with the software levels of computers. In particular, it discusses the high-level language (HLL)-level architecture, the assembly-level architecture, and the instruction set architecture (ISA). Chapter 2 gives a brief overview of program development. This discussion is focused primarily on the high-level architecture. The detailed discussion of computer architecture begins in this chapter with background information on the high-level architecture, which is usually covered in a pre-requisite course to computer architecture. This material is included in the book for completeness and to highlight some of the software issues that are especially critical to the design of computer systems. Chapter 3 provides a detailed treatment of the assembly-level architecture. In particular, it describes the memory model, the register model, instruction types, and data types. It also discusses programming at this level. Chapter 4 covers the Kernel mode, and different ways of carrying out IO operations. Chapter 5 discusses ISA. It covers instruction encoding, data encoding, translation from assembly language to machine language, and different approaches to instruction set design.





## Chapter 2

# Program Development Basics

*Let the wise listen and add to their learning, and let the discerning get guidance*

**Proverbs 1: 5**

Software development is a fundamental aspect in computing; without software, computing would be limited to a few fixed algorithms that have been hardwired into the hardware system. The phenomenal power of computers is due to their ability to execute different programs at different times or even concurrently. Most of today's program development — *programming* — is done in one of the high-level languages. Therefore, much of the discussion in this chapter is focused on high-level languages. These languages abstract away the hardware details, making it possible to develop portable programs, i.e., programs that are not tied to any specific hardware platform and can therefore be made to execute on different hardware platforms. It is this high degree of abstraction that gives them the name “high-level languages.” Programming at a high level allows programmers not to be concerned with the detailed machine-level specifications, which in turn improves their efficiency (if not the efficiency of the code!).

Many high-level languages are popular today: C, C++, Java, FORTRAN, VisualBASIC, etc. Our objective in this chapter is *not* to teach programming; we assume that you are already familiar with at least one high-level language, and have done some entry-level programming. Our intent is to review important concepts that are common to program development — irrespective of the language — and to lay a foundation for the material presented in the subsequent chapter, which deals with the assembly-level architecture and translation of programs from high-level languages to assembly languages. In that vein, we touch upon basic issues in software engineering as well; however, advanced software engineering concepts are clearly out of the scope of this book.

## 2.1 Overview of Program Development

“Programs should be written for people to read, and only incidentally for machines to execute.”

— Structure and Interpretation of Computer Programs by Harold Abelson and Gerald Jay Sussman

There is no single way to develop computer programs; programmers differ quite a bit when it comes to how they develop programs. A software engineering approach to programming ..... We shall start with an overview of the important aspects in program development. Below we highlight these aspects.

- **Problem modeling:** The model is created by understanding the complete problem to be solved, and making a formal representation of the system being designed.
- **Algorithm development:** Once a formal model of the problem is developed, the next step is to develop an appropriate algorithm for solving the problem. Algorithm development involves defining the following:
  - *Data structures:* the format and type of data the program will represent and manipulate.
  - *Inputs:* the kind of data the program will accept.
  - *Outputs:* the kind of data the program will output.
  - *User interface:* the design of the screen the end user will see and use to enter and view data.
  - *Algorithm:* the methods of manipulating the inputs and determining the outputs.

Algorithm development includes a lion’s share of the problem solving effort. It is a creative process, and has not yet been automated! The reason for this, of course, is that for automating something, an algorithm has to be developed for performing it, which means that we would require an algorithm for writing algorithms! Therein lies the difficulty. An array of guidelines have been developed, however, to make it easier for an algorithm developer to come up with an algorithm for a new problem. Some of these guidelines are given below:

- See if any standard techniques (or “tricks”) can be used to solve the problem.
- See if the problem is a slight variation of a problem for which an algorithm has already been developed. If so, try to adapt that algorithm.
- Divide-and-conquer approach: See if the problem can be broken into subproblems.
- Develop a simplified version of the problem, and develop an algorithm for the simplified problem. Then adapt the algorithm to fit the original problem.

A problem can often be solved by more than one functionally correct algorithm. Choosing between two algorithms often depends on the requirements of a particular application. For instance, one algorithm may be good for small data sets, whereas the other may be good for large data sets.

- **Programming:**
- **Debugging:** Virtually all programs have defects in them called *bugs*, and these need to be eliminated. Bugs can arise from errors in the logic of the program specification or errors in the programming code created by a programmer. Special programming tools assist the programmer in finding and correcting bugs. Some bugs are difficult to locate and fixing them is like solving a complex puzzle.
- **Testing:** Alpha and beta testing. Alpha testing is a small scale trial of the program. The application is given to a few expert users to assess whether it is going to meet their needs and that the user interface is suitable. Bugs and missing features due to the application being unfinished will be found. Any errors in the code and specification will be corrected at this stage. Beta testing is a more wide-ranging trial where the application is given to a selection of users with different levels of experience. This is where the bulk of the remaining bugs are found; some may remain undetected or unfixed.
- **Software delivery:** The completed software is packaged with full documentation and delivered to the end users. When they use the software, bugs that were not found during testing may appear. As these new bugs are reported an updated version of the software with the reported bugs corrected is shipped as a replacement.

Program development, as we saw in Chapter 1, .....

### 2.1.1 Programming Languages

Programming languages are the vehicle we use to express the tasks a computer must perform. It serves as a framework within which we organize our ideas about computer processes.

What makes a programming language powerful? A powerful programming language is more than just a means for instructing a computer to perform various tasks. The power of the language depends on the means it provides for combining simple ideas to form more complex ideas. Every powerful language has three mechanisms for accomplishing this:

- \* primitive expressions, which represent the simplest entities the language is concerned with,
- \* means of combination, by which compound elements are built from simpler ones, and
- \* means of abstraction, by which compound elements can be named and manipulated as units.

In programming, we deal with two kinds of primitives: instructions and data. Informally, data is “stuff” that we want to manipulate, and instructions are descriptions of the rules for manipulating the data. Thus, any powerful programming language should be able to describe primitive data and primitive instructions, and should have methods for combining and abstracting instructions and data.

A computer program is nothing but an algorithm expressed in the syntax of a programming language. For executing an algorithm, it is imperative that it be first expressed in a formal language. The familiar `hello, world!` program given below, when executed on a computer, will display the words “`hello, world!`” on the display. This program uses the syntax and semantics of the C programming language.

---

**Program 1** The Hello World! program in C.

---

```
main() {  
    // Display the string  
    printf("hello, world!");  
}
```

---

The same program, when expressed in a different programming language, will have some differences, but the underlying algorithm will be the same. For instance, when expressed in Java, the same algorithm may look as follows:

---

**Program 2** The Hello World! program in Java.

---

```
class helloworld {  
    public static void main(String[] args) {  
        // Display the string  
        System.out.println("hello, world!");  
    }  
}
```

---

The features supported by a programming language form an important aspect of programming, as the programmer expresses the entire algorithm by means of the programming language chosen. In this section, we discuss features that are common to many high-level programming languages.

- **Alphabet:** High-level languages generally use a rich alphabet, such as the ones used in natural languages, along with many of the symbols used in mathematics. Common languages such as C and Java use the English alphanumerical character set as the alphabet.

- **Syntax:** Syntax specifies the rules for the structure of programs. It specifies the delimiters, keywords, etc, and the possible combinations of symbols. Programming languages can be textual or graphical. Textual languages use sequences of text including words, numbers, and punctuation, much like written natural languages. Graphical languages use symbols and spatial relationships between symbols to specify programs. A languages syntax can be formalized by a grammar or syntax chart.
- **Semantics:** While the syntax of a language refers to the appearance of programs written in that language, semantics refers to the meanings of those programs. The semantics of a language specify the meaning of different constructs in the language, and therefore the behavior of the program when executed. The semantics of a language draw upon linguistics and mathematical logic, and have a connection with the computational model(s) supported by the language.
- **Data types and data abstraction:** All programming languages provide a set of basic data types such as *integers*, *floating-point numbers*, and *characters*. A data type specifies the set of values a variable of that type can have. It also defines how operations such as  $+$  and  $-$  will be carried out on variables of that type. In Pascal, for instance, the expression  $i + j$  indicates integer addition if  $i$  and  $j$  are defined to be **integers**, and floating-point addition if they are defined to be **reals**. Type checking is supported by Pascal to ensure that such operations are applied to data of the same type; more weakly typed languages such as C relax this restriction somewhat. Objects are organized hierarchically

Most of the programming languages allow the programmer to define complex data types out of simpler ones. Examples are the **record** data type in Pascal and the **struct** data type in C. Object-oriented languages such as C++ and Java extend this concept further, by allowing the programmer to define a set of operations for each of the newly defined data type. The data type, along with the associated set of operations, is called an *object*. Program statements are only allowed to manipulate data objects according to the operations defined for that object.

Finally, most modern programming languages allow the programmer to define abstract data types, thereby creating an extended language. An *abstract data type* is a data type that is defined in terms of the operations that it supports and not in terms of its structure or implementation. In the context of programming languages, data abstraction means hiding the details concerning the implementation of data values in computers. Data abstraction thus makes it possible to have a clear separation between the properties of a data type (which are visible to the user interface) and its implementation details. Thus, abstraction forms the basic platform for the creation of user-defined objects.

If a programming language does not directly support data abstraction, the programmer may explicitly design and use abstract data types, by using appropriate coding.

- **Control abstraction:**

- **Library API:**
- **OS API:**

In the discussion that follows, we will provide example code snippets in both C and Java. We selected these two languages because of their popularity. The reader who is not familiar with any of these languages should not be distracted by this choice; the syntax and semantics of these languages are easy to understand and are similar to those of other high-level languages. In any case, we will restrict our discussion to simple constructs in these languages.

### 2.1.2 Application Programming Interface Provided by Library

### 2.1.3 Application Programming Interface Provided by OS

### 2.1.4 Compilation

### 2.1.5 Debugging

*“If builders built houses the way programmers built programs, the first woodpecker to come along would destroy civilization.”*  
— Gerald Weinberg

“Do not look where you fell, but where you slipped.” African proverb

## 2.2 Programming Language Specification

### 2.2.1 Syntax

The syntax of a language refers to .... It affects the readability of the program. It also impacts the ease with which a compiler can parse the program.

### 2.2.2 Semantics

## 2.3 Data Abstraction

Declaration and manipulation of data values is at the heart of computer algorithms. The data types and structures used by algorithms are somewhat abstract in nature. A major part of the programming job involves implementing these abstractions using the more primitive data types and features provided by the programming language. All programming languages provide several primitive data types, and means to combine primitive data types into data structures. Let us look at these primitive data types.

### 2.3.1 Constants

We shall start our discussion of data types with *constants*. Constants are objects whose values do not change during program execution. Many calculations in real-world problems involve the use of constants. Although a constant can be represented by declaring a variable and initializing it to the appropriate value, this may not be the most efficient way from the execution point of view. Most assembly languages do not treat constant and variable data in the same manner. Assembly languages support a special immediate addressing mode that lets a constant value to be directly specified as part of an instruction rather than storing that constant's value in a memory location and accessing it as a variable. By understanding how constants are represented at the assembly language and machine language levels, constants may be appropriately presented in the HLL source code to produce smaller and faster executable programs.

#### 2.3.1.1 Literal Constants and Program Efficiency

High-level programming languages and most modern assembly languages allow you to specify constant values just about anywhere you can legally read the value of a memory variable.

#### 2.3.1.2 Manifest Constants

A manifest constant is a constant value associated with a symbolic name. During program translation, the translator will directly substitute the value everywhere the name appears within the source code. Manifest constants allow programmers to attach meaningful names to constant values so as to create easy-to-read and easily maintained programs.

#### 2.3.1.3 Read-Only Memory Objects

C++ programmers may have noticed that the previous section did not discuss the use of C++ `const` declarations. This is because symbols you declare in a C++ `const` statement aren't necessarily manifest constants. That is, C++ does not always substitute the value for a symbol wherever it appears in a source file. Instead, C++ compilers may store that `const` value in memory and then reference the `const` object as it would a static variable. The only difference, then, between that `const` object and a static variable is that the C++ compiler doesn't allow you to assign a value to the `const` object at runtime.

C++ sometimes treats constants you declare in `const` statements as read-only variables for a very good reason—it allows you to create local constants within a function that can actually have a different value each time the function executes (although while the function is executing, the value remains fixed). Therefore, you cannot always use such "constants" within constant expressions in C++ and expect the C++ compiler to precompute the expression's value.



#### 2.3.1.4 Enumerated Types

Well-written programs often use a set of names to represent real-world quantities that don't have an explicit numeric representation. An example of such a set of names might be various car manufacturers, such as GM, Ford, and Chrysler. Even though the real world does not associate numeric values with these manufacturers, they must be encoded with numerical values if they are to be represented in a computer system. (Of course, it is possible to represent them as "text" by representing each character in the name using ASCII, but that would slow down program execution.) The internal value associated with each symbol can be arbitrary; the important point is that the same unique value is used every time a particular symbol is used. Many programming languages provide a feature known as the enumerated data type that will automatically associate a unique value with each name in a list. The use of enumerated data types helps the programmer to specify the data using meaningful names rather than "magic" numbers such as 0, 1, and 2.

#### 2.3.1.5 Boolean Constants

Many high-level programming languages provide Boolean or logical constants that can represent the values True and False. Because there are only two possible Boolean values, their representation requires only a single bit at the machine language. However, because most machine languages do not permit storage allocation at the granularity of a single bit, most programming languages use a whole byte or even a larger object to represent a Boolean value. The behavior of the unused bits in a Boolean object depends on the programming language. Many languages treat the Boolean data type as an enumerated type.

### 2.3.2 Variables

Irrespective of the specific high-level language used, the programmer sees an abstract machine that supports data structures and operations that can be performed on the data structures. In most high-level languages, the data structures are *declared* a certain *type*. The type indicates both the characteristics of objects that can be represented and the kinds of operations that can be performed on the objects.

As mentioned earlier, manipulation of data values is at the heart of every computer program. It is therefore of utmost importance that high-level languages provide programmers an efficient and easy way of specifying data values that can be modified. Most HLLs allow the programmer to refer to a data value *symbolically* by a name. Variables are used for a variety of data values including input values, output values, loop counts, and intermediate results of computation. Consider a simple program—one that counts the number of words in a text file. This program would need to know the name of the file—information the program end user would need to supply. The program would need a variable to keep track of the number of words counted so far.

Declaring and manipulating variables is a central concept in HLL programming. The HLL variables have some similarity to the variables used in algebra and other branches of mathematics, although there are a few notable differences. In addition to specifying the *name* of a variable, a variable declaration includes specifying the **type**, **scope**, and **storage class** of the variable. The position of a variable declaration statement in a program implicitly specifies the scope of the variable. The declaration is for the benefit of the compiler, which must know how much space to allocate for each variable. Different variable types require different amounts of space. For example, C permits different variable types such as integers, characters, and floats. The declaration of a variable is accomplished by specifying its name and type. For example, in C the declaration

```
int  n;
```

declares an integer variable named **n**.

Most high-level languages allow a variable to be initialized at the time of declaration. In C, the declaration

```
int  n = 5;
```

declares an integer variable named **n**, and calls for initializing its value to 5. Once a variable has been assigned a value, it retains that value until it is modified, either by a direct assignment or an indirect assignment through a pointer.

### 2.3.2.1 Data Type

The data type of a variable defines the set of values that the variable may ever assume, and the semantics of possible arithmetic/logical operations on those values. In other words, a variable type is a formally specified set of values and operations. For example, a variable of type boolean (or logical) can assume either the value **true** or the value **false**, but no other value. Logical operations such as {**and**, **or**, **not**}, and the assignment operation can be performed on it. In addition, the data type indirectly specifies the number of bytes occupied by the variable, and the methodology to be used for carrying out arithmetic operations on the variable. Some of the commonly used data types are discussed next.

**Signed Integers and Unsigned Integers:** These are fundamental data types; virtually every HLL supports them. Most HLLs support different word sizes for integer variables. For instance, C has **short** and **int** variable types for representing 16-bit integers and 32-bit integers, which can take positive as well as negative values. C also lets the programmer declare unsigned integer types by adding the prefix **unsigned** before **short** or **int**.

“Good things, when short, are twice as good” Baltasar Gracian, The Art of Worldly Wisdom

```
double value; /* or your money back! */
short changed; /* so triple your money back! */
```

— Larry Wall (*the Perl C*)

**Floating Point Numbers (for Increased Range):** The range of signed integers representable in a 32-bit fixed-point format is approximately  $-2.15 \times 10^9$  to  $2.15 \times 10^9$ . Many computation problems require the ability to represent numbers that are of much greater or smaller magnitude than the integers in this range. Examples are Avogadro’s number,  $6.02 \times 10^{23}$ ; mass of a proton,  $1.673 \times 10^{-24}$  g; and the US National Debt a few years back, \$17,383,444,386,952.37. In order to represent very large integers and very small fractions, most high-level languages support floating-point (FP) variable types, in which the effective position of the radix point can be changed by adjusting an *exponent*. The radix point is said to float, and the numbers are called floating-point numbers. This distinguishes them from fixed-point numbers, whose radix point is always in the same position. An FP number is written on paper as follows:

$$(\text{Sign})\text{Significand} \times \text{Base}^{\text{Exponent}}$$

The *base* is the radix of the FP number, the *significand* identifies the significant digits of the FP number, and the *sign* identifies the overall sign of the FP number. The exponent, along with the base, determines the *scale factor*, i.e., the factor by which the significand is multiplied to get the actual value. In C, floating-point variables can be declared as follows:

```
float f; /* single precision floating-point */
double d; /* double precision floating-point */
```

**Character:** Textual information has become one of the frequently utilized forms of information for both storage and manipulation. This seems counterintuitive, given that computers have historically been used to “compute,” or perform calculations. However, when we consider the facts that programs are input in text form, that compilers operate on strings of characters, and that computation answers are generally provided via some type of textual information, then the amount of textual information processed by computers begins to be appreciated. Furthermore, the preparation of letters, reports, and other documents has become a major application of computers. The basic unit of textual information is a character. Most high-level languages provide variable type(s) to represent character and/or strings of characters. In C, a character variable can be declared as follows:

```
char c;
```

*If you lost wealth, you lost nothing  
 If you lost health, you lost something  
 If you lost character, you lost everything.*

— An old proverb

**Pointer:** The last data type that we will discuss in this section is the pointer. A pointer is a variable used to hold the (memory) address of another variable. In defining a pointer, the high-level language assumes a limited knowledge of the memory model of the lower level assembly-level architecture that implements it. Only some high-level languages support pointers. Example are Pascal and C. Pointers are helpful for building complex data structures such as linked lists and trees. A pointer variable that points to a character can be declared in C as follows:

```
char *cptr;
```

In this declaration, `cptr` is the pointer variable; the implicit assumption is that whatever `cptr` is pointing to should be interpreted as a data item of type `char`.

**Array:** This is a data structure, i.e., a collection of variables.

**Structure:** This is a data structure, i.e., a collection of variables.

### 2.3.2.2 Scope

Another important piece of information specified in a variable declaration is the variable's scope, which defines where and when it is active and available in the program.

### 2.3.2.3 Static Scoping

With static scoping, a variable always refers to its nearest enclosing binding. Because matching a variable to its binding only requires analysis of the program text, this type of scoping is sometimes also called lexical scoping. Static scope allows the programmer to reason as if variable bindings are carried out by substitution. Static scoping also makes it much easier to make modular code and reason about it, since its binding structure can be understood in isolation. Correct implementation of static scope in languages with first-class nested functions can be subtle, as it requires each function value to carry with it a record of the values of the variables that it depends on. When first-class nested functions are not used or not available (such as in C), this overhead is of course not incurred. Variable lookup is always very efficient with static scope, as the location of each value is known at compile time.

Most high-level languages that have static scoping allow at least the following two scopes for variables:

- global
- local

A global variable can be accessed throughout the program (that is, by all modules or functions in the program). Because of this, declaring too many global variables makes it difficult to debug and track variable values.

A local variable can be accessed only within the block in which it is declared. When a local variable has the same name as that of a global variable, the global variable is not visible in the block where the local variable is visible.

In some high-level languages, the scope of a variable is not explicitly declared; instead, it is implicitly defined by where exactly the variable is declared.

```
int i, j;                /* global variables; static storage class */

main()
{
    int i, *iptr;        /* local variables; automatic storage class */
    static int s;        /* local variable; static storage class */

    iptr = (int *)malloc(40); /* dynamic storage class */
}
```

#### 2.3.2.4 Dynamic Scoping

In dynamic scoping, each identifier has a global stack of bindings. Introducing a local variable with name *x* pushes a binding onto the global *x* stack (which may have been empty), which is popped off when the control flow leaves the scope. Evaluating *x* in any context always yields the top binding. Note that this cannot be done at compile time because the binding stack only exists at runtime, which is why this type of scoping is called dynamic scoping.

Since a section of code can be called from many different locations and situations, it can be difficult to determine at the outset what bindings will apply when a variable is used. This can be beneficial; application of the principle of least knowledge suggests that code avoid depending on the reasons for (or circumstances of) a variable's value, but simply use the value according to the variable's definition. This narrow interpretation of shared data can provide a very flexible system for adapting the behavior of a function to the current state (or policy) of the system. However, this benefit relies on careful documentation of all variables used this way as well as on careful avoidance of assumptions about a variable's behavior, and does not provide any mechanism to detect interference between different parts of a program. As such, dynamic scoping can be dangerous and many modern languages do not use it. Some languages, like Perl and Common Lisp, allow the programmer to choose lexical or dynamic scoping when (re)defining a variable. [edit]

##### Implementation

Dynamic scoping is extremely simple to implement. To find an identifier's value, the

program traverses the runtime stack, checking each activation record (each function's stack frame) for a value for the identifier. This is known as deep binding. An alternate strategy that is usually more efficient is to maintain a stack of bindings for each identifier; the stack is modified whenever the variable is bound or unbound, and a variable's value is simply that of the top binding on the stack. This is called shallow binding. Note that both of these strategies assume a last-in-first-out (LIFO) ordering to bindings for any one variable; in practice all bindings are so ordered. [edit]

Example

```
int x = 0; int f () return x; int g () int x = 1; return f();
```

With static scoping, calling `g` will return 0 since it has been determined at compile time that the expression `x` in any invocation of `f` will yield the global `x` binding which is unaffected by the introduction of a local variable of the same name in `g`.

With dynamic scoping, the binding stack for the `x` identifier will contain two items when `f` is invoked: the global binding to 0, and the binding to 1 introduced in `g` (which is still present on the stack since the control flow hasn't left `g` yet). Since evaluating the identifier expression by definition always yields the top binding, the result is 1.

### 2.3.2.5 Lifetime

A variable's lifetime or *storage class* determines the period during which that variable exists. Some variables exist briefly, some are repeatedly created and destroyed, and others exist for the entire program execution. A variable's storage class determines if the variable loses its value when the block that contains it has completed execution. Most high-level languages support the following three storage classes:

- automatic
- static
- dynamic
- persistent

Automatic variables begin to exist when control reaches their block, and lose their values when execution of their block completes. Examples are the local variables declared within subroutines<sup>1</sup>. Static variables, on the other hand, begin to exist when the program starts running, and continue to retain their values till the termination of the program. Dynamic variables are implicit variables pointed to by pointer variables, and do not exist when the program starts running. They are created during the execution of the program, and continue to exist and retain their values until they are explicitly destroyed by the program. In C, a dynamic variable is created during program execution by calling the library function

---

<sup>1</sup>Some high-level languages permit a local variable to retain its value between invocations of the subroutine by declaring the local variable as a static variable.

`malloc()` or `calloc()`, which returns the starting address of the memory block assigned to the variable. The allotted memory locations are explicitly freed by calling the library routine `free()`. Once some memory locations are freed, those locations should no longer be accessed. Doing so may cause a protection fault or, worse yet, corrupt other data in the program without indicating an error. The following example code illustrates the creation and destruction of a dynamic variable using `malloc()` and `free()`, respectively.

```
char *cptr;

/* allocate a dynamic char array having size elements */
cptr = (char *)malloc(size * sizeof(char));
...
free(cptr);      /* free the block of memory pointed by cptr */
```

A persistent variable is one that keeps its value after program execution, and that has an initial value before program execution. The most common persistent variables are files.

### 2.3.3 IO Streams and Files

The variable types that we saw so far manage data that originate within the program. If a program operates only on internally generated data, then it is geared to solve only a particular instance of a problem (i.e., solving a problem for a particular set of input values only), and is not likely to be very useful. Instead, if the program accepts *external inputs*, then it can solve different instances of a problem. Thus, it is important for programs to have the ability to accept input data. On a similar note, the moment a program completes its execution, its variables cease to exist, and the variable values disappear, without leaving a trace of the computation results. For the purposes of performing input and output, high-level languages provide data types that deal with IO streams and files.

In C, the `stdio` library supports IO streams and files. Each program can access a collection of virtual IO devices (`stdin`, `stdout`, and `stderr`) that may be controlled by a simple set of library functions. The main reason for including IO routines in a library is their complexity: rather than force every application programmer to write these complex routines, simple economics suggest including them in the library software. Example library functions in C that deal with IO streams are `getchar()`, `putchar()`, `scanf()`, and `printf()`. `getchar()` allows application programs to read a single character from standard input, and `putchar()` allows application programs to write a single character to standard output; `scanf()` and `printf()` are useful for performing formatted IO operations with standard input and standard output, respectively<sup>2</sup>.

---

<sup>2</sup>The behavior of these and other similar functions is precisely defined by the ANSI C standard. Standards have been developed for high-level languages by national and international organizations such as ANSI (American National Standards Institute), IEEE (Institute of Electrical and Electronic Engineers), and ISO

For clarification, we present below a simple C program to copy standard input to standard output, one character at a time, using the C library routines `getchar()` and `putchar()`.

```
#include <stdio.h>    /* contains definitions such as EOF */

main()
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

Although IO streams (standard input and standard output) can be used to supply external input to programs and to obtain the results of computation, they are cumbersome to handle large amounts of data. Inputting large amounts of data through a keyboard every time a program is executed is impractical. Moreover, the results sent to the standard output do not leave a “permanent” record. For these reasons, most high-level languages provide a data type called *file*, which provides a permanent way of storing data. Unlike other data structures provided by an HLL, files may be present before the execution of a program, and do not vanish when a program terminates; in other words, they persist.

*“The palest ink is better than the best memory.”*  
— Chinese proverb

The HLL application programmer is provided with an abstraction of a uniform space of named files. Thus, HLL application programmers do not concern themselves with any specific IO devices; instead they can rely on a single set of file-manipulation OS routines for file management (and IO device management in an indirect manner). This is sometimes referred to as **device-independent IO**. For example, a character may be printed on a printer by writing the character to the “printer file”.

Application programs generally read (formatted) data from one or more files, process the data, and then write the result to one or more other files. For example, an accounts payable program reads a file containing invoices and another containing purchase orders, correlates the data, and then prints a check and writes to a file to describe the expenditures. A compiler reads an HLL source program file, translates the program into machine language, and writes the machine language program into an executable file.

The actions that an HLL program is allowed to perform on a file are restricted in certain ways. First of all, before accessing a file, it has to *opened*. Secondly, most of the HLLs permit (International Standards Organization). Adherence to a standard facilitates the development of *portable* programs, which can run in different computer systems with little or no change. Portability is particularly important, given that software is a major investment for many computer users.



only sequential access to the data present in a file. After completing all accesses to a file, the file is *closed*. The activities of opening, closing, reading from, and writing to files are done using special function calls, which are also typically implemented as part of the library routines. Example library functions in C that deal with file accesses are `fopen()`, `fclose()`, `fscanf()`, and `fprintf()`.

Instead of specifying a file's name every time it is accessed, C provides a *file pointer* data type. Unlike other pointer variables, which point to the starting address of a variable, a file pointer does not point to the file, but rather to a data structure that contains information about the file. The following C code illustrates how a file pointer is declared, initialized, and used.

```
#include <stdio.h>

main()
{
    FILE *fp;      /* special data type for accessing a file */
    int c;

    fp = fopen("fname", "r"); /* open file "fname" in read mode */
    c = getc(fp); /* read next character from file pointed by fp */
    fclose(fp); /* close file pointed by fp */
}
```

HLLs such as C provide a uniform interface for both IO streams and files. The generic functions used to interact with the IO streams are similar to those provided for the manipulation of files. In fact, if sequential access to files is assumed, there is practically no difference between a virtual device and a file, and hence virtual devices may be manipulated using the same library routines as those used to access files.

The library routines can be linked to the application program statically at link time or dynamically at run time.

### 2.3.4 Data Structures

### 2.3.5 Modeling Real-World Data

As we saw in Chapter 1, problem solving using a computer involves executing an algorithm with appropriate input data. For a digital computer to solve real-world problems, the real-world data has to be converted to a digital form that can be easily represented and manipulated inside the computer. To begin with, all of the analog data has to be converted to digital data, possibly resulting in some loss of precision depending on the number of bits used to represent the digital data. The digital data itself can be represented in many

different ways. In fact, one of the most important steps in developing a computer algorithm is to carefully consider what real-world data the algorithm needs to process and then choose an appropriate internal representation for that data. For some type of data, an internal representation is fairly obvious; for other types of data, many alternatives exist.

Depending on the architectural level at which programming is done, the details of data representation will differ. In this section, we concern ourselves only with how data is represented at the algorithm development level, arguably the highest level that we can think of.

### 2.3.5.1 Images

Images have become an important type of data these days, especially with the popularity of the internet. Images form an important part of many documents and presentations. Images vary greatly, based on size, color, textures, and shapes of objects. Different formats are used to represent images, depending on these characteristics as well as processing and performance requirements. There are two fundamentally different ways the computer stores and manipulates images.

- Vector image — Draw-type: In this approach, an image is viewed as a collection of lines, shapes, and objects. Lines and curves can be easily defined by mathematical objects called vectors. Geometrically definable shapes can be easily represented as mathematical objects by a small number of parameters. For example, a line can be completely specified by noting the co-ordinates of its end points. A circle can similarly be specified by noting the co-ordinates of its center and the length of its radius.

A draw-type image, often referred to as a vector or scalable image, contains a set of objects whose characteristics are stored mathematically. Each individual object within the image retains its own characteristics, such as the co-ordinates of its vertices (corners), the thickness and colour of its outline, the color of its interior, etc. This makes it possible to target editing actions at specific elements of an image. Vector graphics are resolution independent and can be scaled to any size and printed at any resolution without losing clarity. Vector graphics are best for type and graphics that must retain crisp lines when scaled to various sizes. Examples of commonly used software for producing vector images include: CorelDRAW!, Adobe Illustrator, Aldus Freehand, Microsoft Draw, and AutoDesk AutoCAD.

- Raster image (pixelmap, bitmap) — Paint-type: In this approach, an image is viewed as a rectangular grid of tiny squares called *pixels*. Each pixel has a specific location and color value. When viewed in this manner, the viewer does not “see” objects or shapes! As a result, a raster image can lose detail and appear jagged if viewed at a high magnification or printed at too low a resolution. Raster images are best for representing subtle gradations of shades and color such as in photographs. Raster images can either be created entirely by computer, or can be sampled initially from

other sources, for example, scanning a photograph or capturing a frame of video. When displaying on computer monitors, all images — vector images and raster images — are displayed as pixels on-screen.

## Image Formats

Several different formats are used to represent images. Some of these are lossless in that they retain all of the information that has been captured about an image. Others are lossy and approximate some of the information so as to reduce the amount of data required to store and manipulate the image.

**Bitmap (BMP):** It is the Microsoft Windows standard for image files. The bitmap format stores the image in uncompressed form, and so it is lossless. As the image is not compressed, it renders good images; however, the data file is likely to be quite large. What you store is pretty much what you see! Although they do have the ability to support up to 16.7 million colors, there really is no reason why the average user should need to use this format for image manipulation. Moreover, the bitmap format does not support animation, transparency, and interlacing.

**Graphics Interchange Format (GIF):** The GIF is a lossless image format that is the most common format found on the web. Due to having a 256 maximum color range this format is ideal for making small icons, buttons, or other graphics that have large blocks of the same color, but not for images that are required to be photographic quality. Therefore, if you are working with images from a digital camera, do not use GIF as the file format. This file format supports transparency, interlace, and can be animated, which makes it a excellent format for putting images on a website. To reduce the data size, GIF does use a non-lossy compression algorithm. This means that the compressed image can be converted back to the original image with no loss of detail. The algorithm works by noting sections of the image that are the same color, and works quite well for images that have large areas of solid color with little to no texture. The GIF format is an excellent choice for images that are cartoonlike in appearance such as banner ads, logos, and text on a solid background. It is a poor choice for real-life depictions such as photographs of nature or people which tend to have lots of detailed variations. CompuServe Graphics Interlaced Format. Designed for transmission over modem links, GIF files are compact images that can be stored in an interlaced format. This means that one line of pixels in every four will be decoded first, allowing the user to see what the image will look like before the whole file is downloaded. It has a maximum of 8-bit (256 colours). This type of image is used on the World Wide Web.

**Joint Photographic Experts Group (JPEG):** The JPEG format was developed by the Joint Photographic Experts Group committee. It is a lossy file format that was designed

with photographic images in mind. A JPEG is capable of storing millions of colors, making it a great format for saving digital camera photographs and capturing the proper hues and color that we see in real life. JPEG does support compression, but the more you compress this type of image, the more loss of detail will occur. The predominant uses for JPEGs are for photographic images on websites and for storing pictures from digital cameras. Though the TIFF format is a higher quality format, the size of the resulting TIFF image, makes the JPEG the more practical choice. This format does not support animation or transparency, but can be interlaced. It is a lossy compression meaning the compressed image will not render in as much detail as the original from which it was created. The JPEG format is an excellent choice for photographic images which depict the real world such as nature or people and also for complex backgrounds with lots of texture and detailed variation.

**MPEG:** An MPEG file uses a complex algorithm like a JPEG file does – it tries to eliminate repetition between frames to significantly compress video information. In addition, it allows a soundtrack (which animated GIFs do not). Because a typical sequence has hundreds or thousands of frames, file sizes can still get quite large.

**Shockwave:** Shockwave provides a vector-based animation capability. Instead of specifying the color of every pixel, a Shockwave file specifies the coordinates of shapes (objects like lines, rectangles, circles, etc.) as well as the color of each shape. Shockwave files can be extremely small. They allow animation and sound. The images are also scalable; because they are vector-based, the image can be enlarged and it will still look great.

**Tagged Image File Format (TIFF):** The TIFF format is a lossless image format that is considered the best choice for photographic image quality. Most digital cameras give the option of using TIFF as the format it saves files in. The main problem with this file format is that most applications do not compress the TIFF files, and so they can be quite large. This is not much of a problem for storing the pictures on a computer, but with limited flash memory sizes for cameras it could limit the amount of pictures that can be stored on one card. If you have the storage, then the TIFF format is highly recommended, but if you do not have the space, then go with a JPEG as you most likely will not notice a difference in image quality. This format does not support animation, transparency, and can not be interlaced.

**Encapsulated PostScript (EPS):** Adobe PostScript is the industry standard page description language. EPS files can contain both paint and draw type information, and can often become extremely large.

Name Extension Compressed Loss Animated Max Colors Transparency Interlaced

|   |     |              |     |              |              |        |                                |
|---|-----|--------------|-----|--------------|--------------|--------|--------------------------------|
| Graphics Interchange Format .GIF          | Yes | Lossless     | Yes | 256          | Yes          | Yes    | Joint Photographic             |
| Experts Group .JPG                        | Yes | Lossy        | No  | 16.7 Million | No           | Yes    | Portable Network Graphics .PNG |
| Yes Lossless                              | Yes | 256/16.7 Mil | Yes | Yes          | Bit-Map .BMP | Rarely | Lossless                       |
| No Tagged Image File Format .TIFF or .TIF | Yes | Lossless     | No  | 16.7 Million | No           | No     |                                |

Figure 4: Graphic Formats and their Attributes

### 2.3.5.2 Video

Video data is a natural extension of graphical data. It is a sequence of still images that depict how the scene changes with the passage of time. If these still images are displayed at the rate of 30 or more images per second, then, to the human eye, they appear as a continuous motion picture.

### 2.3.5.3 Audio

Another real-world entity that we often represent and process inside a computer is audio. Although real-world audio is analog in nature, inside the computer it is stored in digital form, called *digital audio*. Digitization is done by electronically sampling the analog waveform at regular time intervals; the time intervals should be small enough to capture every nuance in the analog signal. Each sample is then approximated to one of the allowable discrete values using an analog-to-digital (A/D) converter. These discrete values are then represented in a binary format.

## 2.4 Operators and Assignments

Apart from providing different data types, high-level languages also provide a set of arithmetic operators and logical operators that allow the programmer to specify manipulation of variables. Operators act on the values assigned to variables. Variables, along with the operators, allow the HLL programmer to express the computation specified in the algorithm to be implemented.

The assignment statement in C involves the evaluation of expressions composed of operators, variables, and constants. In C, as in most HLLs, all operators are either monadic or dyadic, i.e., involve one or two operands.

Another operator available in C and other languages that support pointers is the address-of operator, “&”, to take the address of a static variable.

## 2.5 Control Abstraction

What distinguishes a computer from a simple calculator is its ability to make decisions: based on the input data and the values created during the computation, different instructions are executed.

### 2.5.1 Conditional Statements

*I shall be telling this with a sigh  
Somewhere ages and ages hence:  
Two roads diverged in a wood, and I  
I took the one less traveled by,  
And that has made all the difference.*

— *The Road Not Taken* by

Robert Frost

*I have always known  
That at last I would  
Take this road, but yesterday  
I did not know that it would be today.”*

— *Ise Monogatari (The Tales*

*of Ise)* by Ariwara no Narihira (9th century Japan)

**if-else:** Often it is necessary to have the program execute some statements only if a condition is satisfied. All high-level languages provide **if** constructs to program conditional execution. The C **if** statement has the syntax

```
if (expr)
    stmt;
```

The semantics of this statement is: if *expr* evaluates to true, execute *stmt*, otherwise skip *stmt*. Sometimes, a program may need to choose one of two alternative flows of control, depending on the value of a variable or expression. For this purpose, the C language provides an **if-else** construct, which has the syntax

```
if (expr)
    stmt1;
else
    stmt2;
```

The semantics of this statement is: if *expr* evaluates to true, execute *stmt1*, otherwise execute *stmt2*.

**switch:** High-level languages also generally provide a construct for specifying multi-way decisions. The C language provides a **switch** construct, which compares the value of an expression against a list of supplied *constant* integer values, and depending on the match selects a particular set of statements. Its syntax is given below.

```
switch (expr)
{
    case const-expr1:
        stmt1;
        break;
    case const-expr2:
        stmt2;
        break;
    default:
        stmt3;
}
```

The semantics of this **switch** statement is: if *expr* when evaluated matches the evaluated value of *const-expr1* or *const-expr2*, then control branches to *stmt1* or *stmt2*, respectively. If there is no match, then control branches to *stmt3*, which follows the **default** label. After executing the statements associated with the matching **case**, control falls through to the next **case** unless control is explicitly taken out of the **switch** construct using statements such as **break**, **goto**, **return**, or **exit()**.

### 2.5.2 Loops

**while:** Most programs require some action to be repeated a number of times, so long as a particular condition is satisfied. Although it may be possible to include the same code the required number of times, that would be quite tedious and result in longer programs. Moreover, this may not be possible in cases where we do not know at programming time the number of times the action needs to be repeated. High-level languages provide *loop* constructs to express such code in a concise manner. The C **while** loop statement has the syntax

```
while (expr)
    stmt;
```

The semantics of this statement is: if *expr* evaluates to a non-zero value, execute *stmt*, and repeat this process until *expr* evaluates to zero. *stmt* normally modifies a variable that is contained in *expr* so that *expr* will eventually become false and therefore terminate the loop.

**for:** Although a **while** loop is sufficient for implementing any kind of loops, most HLLs provide a **for** loop construct to express some loops in a more elegant manner. The C **for**

statement has the syntax

```
for (initial_expr; expr2; update_expr)
    stmt;
```

The semantics of this **for** loop can be easily understood when considering its equivalent **while** loop, given below:

```
initial_expr;
while (expr2)
{
    stmt;
    update_expr;
}
```

In this loop, *initial\_expr* is the initializing statement; *expr2* calculates the terminating condition; and *update\_expr* implements the update of the loop variables.

### 2.5.3 Subroutines

The next topic that we like to discuss is the *subroutine*, *procedure*, *function*, or *method* concept, an important innovation in the development of programming languages. In a given program, it is often necessary to perform a specific task on different data values. Such a repeated task is normally implemented as a *subroutine*, which can be called from different places in a program. For example, a subroutine may evaluate the mathematical *sine* function or sort a list of values into increasing or decreasing order. At any point in the program the subroutine may be invoked or *called*. That is, at that point, the computer is instructed to suspend the execution of the *caller* and execute the subroutine. Once the subroutine's execution is completed, control returns to the point from where the call was made.

The two principal reasons for the use of subroutines are economy and modularity. A subroutine allows the same piece of code to be called from different places in the program. This is important for economy in programming effort, and for reducing the storage requirements of the program. Subroutines also allow large programming tasks to be subdivided into smaller units. This use of *modularity* greatly eases the programming task. The following C code illustrates the use of subroutines in constructing a modular program.

```
main()
{
    float a, b, c, d;

    b = bigger(a, b);
```



```
        d = bigger(c, d);
    }

    bigger(float p, float q)
    {
        if (p >= q)
            return p;
        else
            return q;
    }
```

In this code, the function `bigger()` is used to determine which of its two input parameters (`p` or `q`) is bigger. This function is called twice from the function `main()`. If the language does not support subroutines/functions, then most of the body of function `bigger()` will need to be repeated twice in `main()`.

The subroutine concept thus permits *control abstraction*; the code fragment within the subroutine can be referred by the subroutine name at the calling place, where it is thought in terms of its function rather than its implementation. In structured programming languages, subroutines (and macros) are the main mechanism for control abstraction.

### 2.5.3.1 Parameter Passing

## 2.5.4 Subroutine Nesting and Recursion

If a subroutine calls a subroutine (either itself or another subroutine), then that is called subroutine nesting. Conceptually, subroutine nesting can be carried out to any depth. The first subroutine to complete will be the one that was called last, causing control to return to the one that called it.

If nesting involves calling the same subroutine itself, either directly or through other subroutines, then we call that recursion. Recursion is an important concept in computer science, analogous to the concept of induction in mathematics. Mathematical problems that can be explained by induction invariably have an elegant recursive solution algorithm.

*A rose is a rose is a rose*

— Gertrude Stein

## 2.5.5 Re-entrant Subroutine

In a multitasked environment, many programs (or processes) can be simultaneously active in a computer system. Many of the concurrently active processes may share a few routines, especially those belonging to libraries and the OS. Consider the scenario where a context

switch happens when a process is in the middle of the execution of a subroutine. Before the control is restored to this process, it is quite possible for another process to execute the same subroutine. In order for a subroutine to be executed by another process in this manner, the subroutine must be *re-entrant*, i.e., it must not update global variables.

### 2.5.6 Program Modules

We can let a subroutine to call subroutines that are physically present in other modules. The caller and callee subroutines may be written by different programmers, at different times! This calls for defining and using specific interfaces for each subroutine that is likely to be called from other modules.

### 2.5.7 Software Interfaces: API and ABI

Application programs do not directly implement all of the functionality required. Instead, they frequently invoke the services of library routines and OS routines.

#### 2.5.7.1 Application Binary Interface (ABI)

An application binary interface (ABI) specifies the machine language interface provided by an execution environment, which is usually a hardware platform along with the operating system running on it (e.g., Linux ABI for the ARM Architecture). Thus, the ABI refers to the specifications to which an executable should conform in order to execute in a specific execution environment. The ABI includes the user mode instruction set architecture (the memory address space, the number, sizes and reserved uses of registers, and the instruction set), the ISA-level system call interface supported by the operating system (including the system call numbers and how an application should make system calls to the operating system), and the binary format of executable files. ABIs also cover other details of system calls, such as the calling convention, which tells how functions' arguments are passed and return values retrieved. ABIs deal with run-time compatibility; a program binary targeted to an ABI can run (without relinking or recompilation) on any system that supports the same ABI. Application binary interfaces are also known as **Abstract Machine Interface**.

## 2.6 Library API

A practical extension of the above modularization concept is the use of library modules. A library is a group of commonly used subroutines or functions bundled into a single module. The basic libraries contain routines that read and write data, allocate and deallocate memory, and perform complex operations such as square root calculation and sorting. Other libraries contain routines to access a database or manipulate terminal windows. Apart from

language-dependent library such as the C `stdio` which provides IO routines, we also have language-independent but OS-dependent libraries such as the Solaris thread library which provides support functions for multithreading, and language- and platform-independent libraries such as the MPI (which supports the message-passing model of parallel processing) and the OpenGL (which supports advanced graphics functions).

## 2.7 Operating System API

A hardware system, augmented by an operating system, cannot do much unless application programs are loaded in it. Application programs are what the end users run when ..... In modern computers, application programs never run in a “vacuum” either. They rely heavily on support from pre-written software such as library routines and the operating system. When the application program wants to carry out a functionality that has already been implemented in a library routine or in the OS, it simply invokes the functionality and does not directly code the functionality.

An application programming interface (API) specifies a language and message format used by an application program to communicate with a systems program (library, OS) that provides services to it. Three commonly used OS APIs are the POSIX API for UNIX, Linux, and MAC OS X systems; the Win32 API for Windows systems; and the Java API for the Java virtual machine. It is the interface by which an application gains access to operating systems.

The application program interface (API) defines the calls that can be made from an application to the operating system. Notice that adherence to an API does not ensure runtime compatibility.

An API specifies a set of calling conventions that defines how a service is invoked through a software package. The calls, subroutines, interrupts, and returns that comprise a documented interface so that a higher-level program such as an application can make use of the services of another application, operating system, network operating system, driver, or other lower-level software program.

In the software field, APIs are structured abstraction layers that conceal the gory details of an individual application, operating system or hardware item and the world outside that software or hardware.

All programs using a common API will have similar interfaces. This makes it easier to port programs across multiple systems and for users to learn new programs.

An application program interface or application programming interface (API) is the specific method prescribed by a computer operating system or by an application program by which a programmer writing an application program can make requests of the operating system or another application.

We just saw the use of library routines in supporting IO streams and files. The library

routines do not directly access the IO devices, however. As we saw in Section 1.2.2, access to and control of many of the hardware resources are regulated through the operating system. This is because the low-level details of the system's operation are often of no interest to the computation at hand, and it is better to free the application programmer and the library developer from dealing with the low-level details. The problem is further aggravated because the program must be able to deal with a multitude of different IO interfaces, not only for different device types but also for different models within the same class of devices. It would not be practical to require each programmer to know the operational details of every device that will be accessed by a program. Moreover, the number and type of devices may change in course of time. It is important that such changes remain transparent to application programs.

To satisfy the preceding requirements of device independence, the operating system provides an *abstract interface* to the application programmer. This interface, called application programming interface (API) of the OS, presents a greatly simplified view of the hardware resources of the computing environment. The OS' API is simply a set of commands that can be issued by the application program to the operating system. If the OS performs these jobs, it simplifies the job of the application programmer, who need not get involved with the details of hardware devices. Further, when an application program uses the API, it is shielded from changes in the computer hardware as new computers are developed. To be specific, the operating system and its device drivers can be changed to support new computer hardware while preserving the API unchanged and allowing application programs to run unchanged on the new computer. The API provided by the OS is formally defined by a set of human readable function call definitions, including call and return parameters.

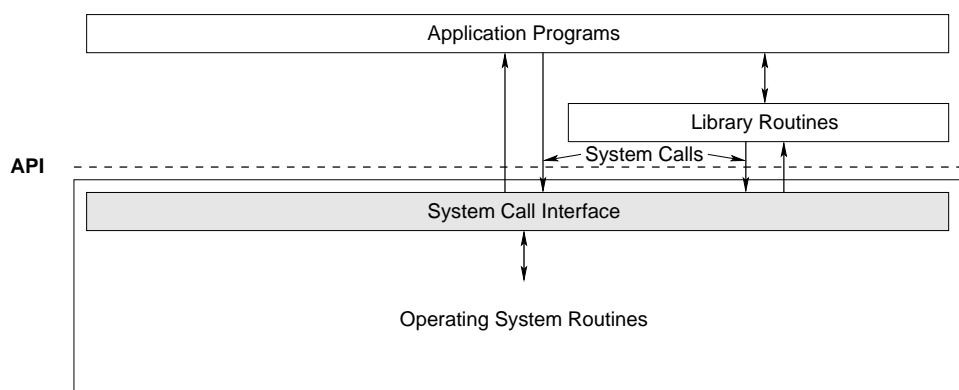


Figure 2.1: Illustration of API

Applications programs written with a particular API in mind can be run only on systems that implement the same API. The APIs defined by most operating systems are very similar, and differ only in some minor aspects. This makes it easy to port application programs developed for one OS to another. Examples of APIs are the Portable Operating System

Interface (POSIX) and the Win32 API. The POSIX standard is based on early UNIX systems and is widely used in UNIX-based operating systems such as FreeBSD and Linux. Many non-UNIX systems also support POSIX. The Win32 API is the one implemented in a Microsoft Windows environment. It supports many more functions (about 2000) than POSIX, as it includes functions dealing with the desktop window system also.

### 2.7.1 What Should be Done by the OS?

- Simple vs Complex Function
- Specific vs Generic Function
- Security

The commands supported by an OS' API can be classified into 3 classes:

- Input/Output management
- Memory management
- Process management

### 2.7.2 Input/Output Management

The most commonly used part of the OS' API is the part that deals with input/output. The basic abstraction provided for application programmers and library programmers to perform input and output operations is called a **file**. The file abstraction supported by the API is more basic than the one supported by library routines in that it is simply a sequence of bytes (with no formatting) in an IO device. This definition of the file as a stream of bytes imposes little structure on a file; any further structure is up to the application programs, which may interpret the byte stream as they wish. The interpretation has no bearing on how the OS stores the data. Thus, the syntax for accessing the data in a file is defined by the API, and is identical for all application programs, but the semantics of the data are imposed by the application program. For instance, the text formatting program **LaTeX** expects to find “new-line” characters at the end of each line of text, and the system accounting program **acctcom** expects to find fixed-length records in the file. Both programs use the same API commands to access the data in the file as a byte stream, and internally, they parse the stream into the appropriate format.

The file abstraction provided by the API is also **device-independent** like the one provided by the standard library. That is, it hides all device-specific aspects of file manipulation from HLL application and library programmers, and provides instead an abstraction of a simple, uniform space of named files.

### 2.7.2.1 Example

For illustration, let us consider the UNIX API. In this API, all input and output operations are done by reading or writing files, because all IO devices—including the user program’s terminal—are considered as files. This means that a single, homogeneous interface handles all communication between a program and peripheral devices.

The lowest level of IO in the UNIX API provides no buffering or any other services. All input and output operations are done by two API functions called `read` and `write`, which specify a file descriptor and the number of bytes to be transferred. The second argument is a buffer in the application program’s memory space where the data is to come from or go to. The third argument is the number of bytes to be transferred. The calls are done as follows:

```
numbytes_read = read(fd, buf, numbytes);
numbytes_written = write(fd, buf, numbytes);
```

Each of these system calls returns a byte count indicating the number of bytes actually transferred. When reading, the number of bytes returned may be less than the number asked for. A return value of zero indicates end of file, and a `-1` indicates an error<sup>3</sup>.

### 2.7.2.2 A Trace of a System Call

When an application program executes a `read()` system call to read data from a file, a set of OS functions are called, which may eventually result in calling an appropriate *device driver*. Figure 2.2 illustrates a situation in which an application program calls the OS twice for reading from standard input. During the first time, it calls the `scanf()` library function, which calls the `read()` system call defined in the API. During the second time, it directly calls the `read()` system call. In both cases, the `read()` routine calls the `keybd.read()` device driver to perform the actual read operation.

If a different keyboard is used in the future, only the keyboard device drivers need to be changed; the application program, the library routines, and the device-independent part of the OS require no change.

### 2.7.3 Memory Management

High-level languages that support pointers naturally allow dynamic allocation (and deallocation) of memory. Applications programmers typically do allocation and deallocation using library functions such as `malloc()` and `free()`, as we already saw in Section 2.1. The `malloc(n)` function, for instance, returns a pointer to an unused block of `n` bytes.

---

<sup>3</sup>More details of `read` and `write` system calls can be obtained in a UNIX/Linux system by typing `man 2 read` and `man 2 write`, respectively.

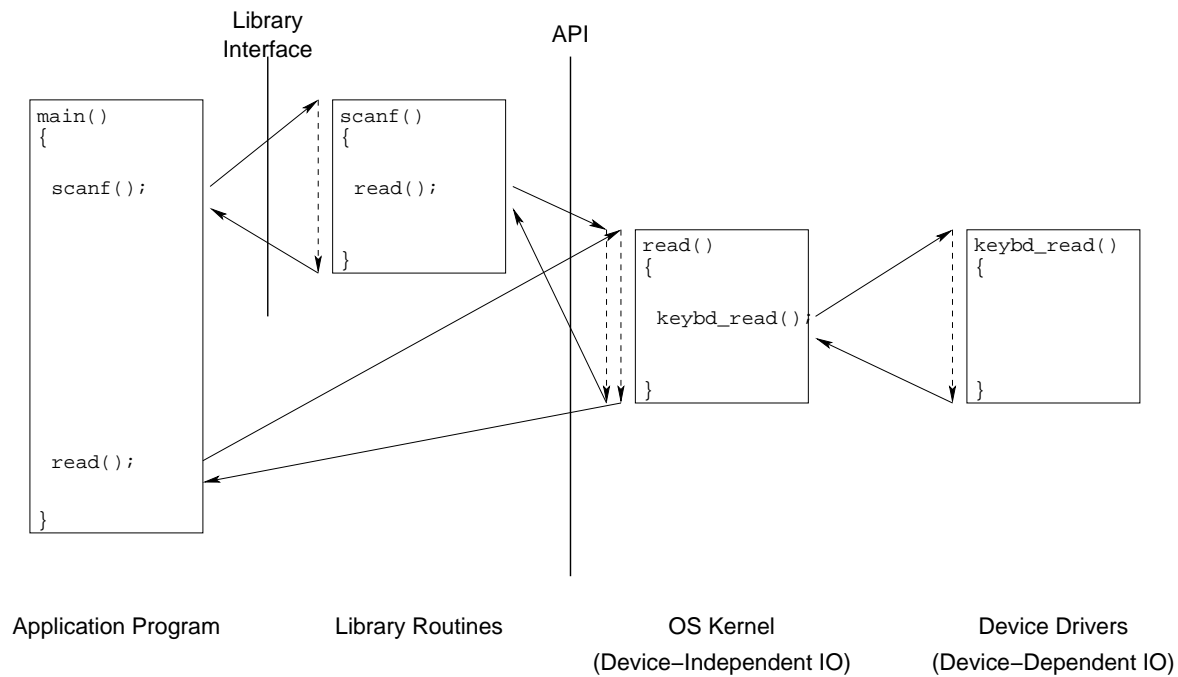


Figure 2.2: A Trace of Routines Executed when Calling API Function `read()`

obtains large chunks of memory address space from the OS using the `sbrk()` system call provided by the API, and manages them. The system call `sbrk(b)` returns a pointer to `b` more bytes of memory.

## 2.7.4 Process Management

process creation

context switch

process control block

## 2.8 Operating System Organization

In modern computers, the operating system is the only software component that runs in the Kernel mode. It behooves us therefore to consider the structure and implementation of an operating system. In particular, it is important to see how the system calls specified in the application programming interface (API) (provided to user programs and library functions)

are implemented by operating systems<sup>4</sup>. The exact internal details of an operating system vary considerably from one system to another. It is beyond the scope of this book to discuss different possibilities. Figure 4.7 gives a possible block diagram, which is somewhat similar to that of a standard UNIX kernel. In the figure, the kernel mode software blocks are shown shaded. The main components of the OS software include a system call interface, file system, process control system, and the device management system (device drivers). This organization uses a layered approach, which makes it easier to develop the OS and to introduce modifications at a later time. This also makes it easier to debug the OS code, because the effect of bugs may be restricted to a single layer. The figure also shows the relationship of the OS to user programs, library routines, and the hardware.

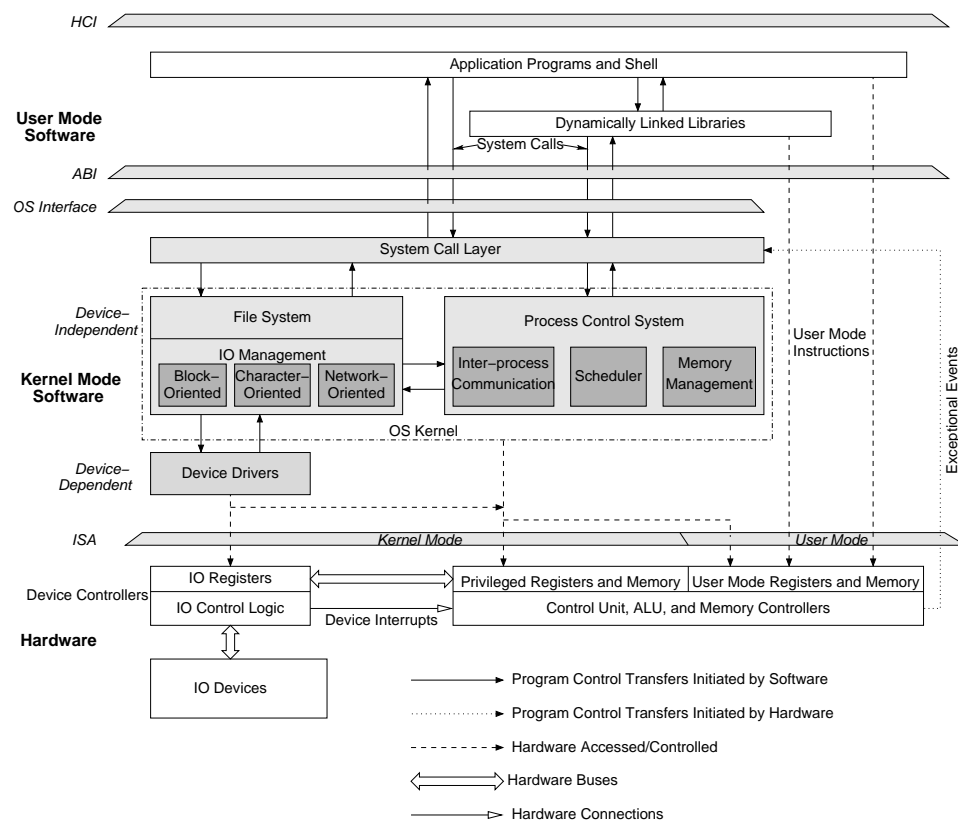


Figure 2.3: Block Diagram Showing How a UNIX-like Kernel Implements the API

<sup>4</sup>Operating systems typically play two roles—controlling the environment provided to the end user and controlling the environment provided to application programs. The former involves tasks such as maintaining a file structure and supporting a graphical user interface. The latter involves tasks such as reading a specified number of bytes from a file on behalf of an application program. In this book, we are concerned only with the latter role, as it is closer to computer architecture.



### 2.8.1 System Call Interface

The system call interface provides one or more entry points for servicing system call instructions and exceptions, and in some cases device interrupts also. The system call interface code copies the arguments of the system call and saves the user process' context. It then uses the system call type to look up a *system call dispatch vector* to determine the kernel function to be called to implement that particular system call, interrupt, or exception. When this kernel function completes, the system call interface restores the user process' context, and switches to User Mode, transferring control back to the user process. It also sends the return values and error status to the user program.

We can summarize the functions performed by the system call interface:

- Determine type of syscall
- Save process context
- Call appropriate handler
- Restore process context
- Return to user program

### 2.8.2 File System

The API provided to application programs by the operating system, as we saw earlier, includes *device-independent IO*. That is, the interface is the same, irrespective of the physical device that is involved in the IO operation. The file abstraction part of the API is supposed to hide all device-specific aspects of file manipulation from application programmers, and provide them with an abstraction of a simple, uniform space of named files. Thus, application programmers can rely on a single set of file-manipulation OS routines for file management (and IO device management in an indirect manner). This is sometimes referred to as **device-independent IO**.

device-  
independent  
IO

As we saw in Section 3.4.7, application programs access IO (i.e., files) through **read** and **write** system call instructions. The **read** and **write** system call instructions (of the User mode) are implemented in the Kernel mode by the *file system* part of the OS, possibly with the help of appropriate *device drivers*.

Files of a computer installation may be stored on a number of physical devices, such as disk drives, CD-ROM drives, and magnetic tapes, each of which can store many files. If the IO device is a storage device, such as a disk, the file can be read back later; if the device is a non-storage device such as a printer or monitor, the file cannot be read back. Different files may store different kinds of data, for example, a picture, a spreadsheet, or the text of a book chapter. As far as the OS is concerned, a file is simply a sequence of bytes written to an IO device.

The OS partitions each file into *blocks* of fixed size. Each block in a file has an address that uniquely tells where within the physical device the block is located. Data is moved between main memory and secondary storage in units of a single block, so as to take advantage of the physical characteristics of storage devices such as magnetic disks and optical disks.

File management related system calls invoked by application programs are interpreted by the file system part of the OS, and transformed into device-specific commands. The process of implementing the **open** system call thus involves locating the file on disk, and bringing into main memory all of the information necessary to access it. The OS also reserves for the file a *buffer* space in its memory space, of size equal to that of a block. When an application program invokes a system call to write some bytes to a file, the file system part of the OS writes the bytes in the buffer allotted for the file. When the buffer becomes full, the file system copies it into a block in a storage device (by invoking the device's device driver); this block becomes the next block of the file. When the application process invokes the **close** system call for closing a file, the file system writes the file's buffer as the final block of the file, irrespective of whether the buffer is full or not, prior to closing the file. Closing a file involves freeing up the table space used to hold information about the file, and reclaiming the buffer space allotted for the file.

### 2.8.3 Device Management: Device Drivers

The device management part of the OS is implemented as a collection of device drivers. Most computers have input/output devices such as terminals and printers, and storage devices such as disks. Each of these devices requires specific *device driver* software, which acts as an interface between the device controller and the file system part of the OS kernel. A device driver is needed because each device has its own specific commands instead of generic commands. A printer device driver, for instance, contains all the software that is specific to a particular type of printer such as a Postscript printer. Thus, the device drivers form the *device-dependent* part of the IO software. By partitioning the kernel mode software into device-independent and device-dependent components, the task of adding a new device to the computer is greatly simplified.

The device drivers form a major portion of the kernel mode software. Each device driver itself is a collection of routines, and can have multiple entry points. The device driver receives generic commands from the OS file system and converts them into the specialized commands for the device, and vice versa. To the maximum extent possible the driver software hides the unique characteristics of a device from the OS file system.

Device drivers can be fairly complex. Many parameters may need to be set prior to starting a device controller, and many status bits may need to be checked after the completion of each device operation. Many device drivers such as the keyboard driver are supplied as part of the pre-installed system software. Device drivers for other devices need to be installed as and when these devices are installed.

The routines in a device driver can be grouped into three kinds, based on functionality:

- Autoconfiguration and initialization routines
- IO initiation routines
- IO continuation routines (interrupt service routines)
- IO initiation routines
- IO continuation routines (interrupt service routines)

The autoconfiguration routines are called at system reboot time, to check if the corresponding device controller is present, and to perform the required initialization. The IO initiation routines are called by the OS file system or process control system in response to system call requests from application programs. These routines check the device status, and initiate IO requests by sending commands to the device controller. If program-controlled IO transfer is used for the device, then the IO initiation routines perform the IO transfers also. By contrast, if interrupt-driven IO transfer is used for the device, then the actual IO transfer is done by the interrupt service routines when the device becomes ready and issues an interrupt.

#### 2.8.4 Hardware Abstraction Layer (HAL)

The hardware abstraction layer provides a slightly abstract view of the hardware to the OS kernel and the device drivers. By hiding the hardware details, it provides a consistent hardware platform for the OS. This makes it easy to port an OS across a family of hardware platforms that have the same user mode ISA, but differ in the kernel mode ISA (such as different MMU architectures).

#### 2.8.5 Process Control System

##### 2.8.5.1 Multi-Tasking

When a computer system supports multi-tasking, each process sees a separate virtual machine, although the concurrent processes are sharing the same physical resources. Therefore, some means must be provided to separate the virtual machines from each other at the physical level. The physical resources that are typically shared by the virtual machines are the processor (including the registers, ALU, etc), the physical memory, and the IO interfaces. Of these, the processor and the IO interfaces are typically time-shared between the processes (*temporal separation*), and the physical memory is partitioned between the processes

(*spatial separation*)<sup>5</sup>. To perform a *context switch* of the virtual machines, the time-shared resources must be switched from one virtual machine to the next. This switching must be managed in such a way that the virtual machines do not interact through any state information that may be present in the physically shared resources. For example, the ISA-visible registers must be saved and restored during a context switch so that the new context cannot access the old context's register state.

Decisions regarding time-sharing and space-sharing are taken in the Kernel mode by the operating system, which is responsible for allocating the physical resources to the virtual machines. If a user process is allowed to make this decision, then it could possibly encroach into another process' resources, and tamper with its execution. The operating system's decisions, however, need to be enforced when the system is in the User mode. This enforcement is done using special hardware (microarchitectural) support so that the enforcement activity does not reduce performance.

### 2.8.5.2 Multi-Programming

Some applications can be most conveniently programmed for two or more cooperating processes running in parallel rather than for a single process. In order for several processes to work together in parallel, certain new Kernel mode instructions are needed. Most modern operating systems allow processes to be created and terminated dynamically. To take full advantage of this feature to achieve parallel processing, a system call to create a new process is needed. This system call may just make a clone of the caller, or it may allow the creating process to specify the initial state of the new process, including its program, data, and starting address. In some cases, the creating (parent) process maintains partial or even complete control over the created (child) processes. To this end, Kernel mode instructions are added for a parent to stop, restart, examine, and terminate its children.

---

<sup>5</sup>Time-sharing the entire physical memory is not feasible, because it necessitates saving the physical memory contents during each context switch.

## **2.9 Major Issues in Program Development**

### **2.9.1 Portability**

### **2.9.2 Reusability**

### **2.9.3 Concurrency**

## **2.10 Concluding Remarks**

## **2.11 Exercises**

## Chapter 3

# Assembly-Level Architecture — User Mode

*He who scorns instruction will pay for it, but he who respects a command is rewarded.*

**Proverbs 13: 13**

This is the first of several chapters that address core issues in computer architecture. The previous chapter discussed high-level architectures. This chapter is concerned with the immediately lower-level architecture that is present in essentially all modern computers: the assembly-level architecture. This architecture deals with the way programs are executed in a computer from the assembly language programmer's viewpoint. We describe ways in which sequences of instructions are executed to implement high-level language statements. We also discuss commonly used techniques for addressing memory locations and registers. A proper understanding of these concepts is an essential part of the study of computer architecture, organization, and design. We introduce new concepts in machine-independent terms to emphasize that they apply to all computers.

The vast majority of today's programs are written in a high-level language such as C, FORTRAN, C++, and Java. Before the introduction of high-level languages, early programmers and computer architects were using languages of a different type, called assembly languages. The main purpose of discussing assembly-level architectures in this book is to provide an adequate link to instruction set architectures and microarchitectures, which provide a closer view on how computers are built and how they operate. To execute any high-level program, it must first be translated into a lower level program (most often by a compiler and occasionally by assembly language programmers). Knowledge of the assembly-level architecture is a must, both for compiler writers and for assembly language programmers. The relationship between high-level, assembly, and machine language features is a key

consideration in computer architecture. Much of the discussion in this chapter is applicable to both the assembly-level architecture and the instruction set architecture, as the former is a symbolic representation of the latter.

The objective of this chapter on assembly-level architecture is not to make you proficient in assembly language programming, but rather to help you understand what this virtual machine does, and how high-level language programs are converted to assembly language programs. We include a number of code fragments that are short and useful for clarification. These code fragments are meant to be conceptual, rather than to be cut and pasted into your application programs. If you like to write intricate assembly language programs, it is better to follow up this material with a good book on assembly language programming.

### 3.1 Overview of User Mode Assembly-Level Architecture

We shall first present an overview of the basic traits of an assembly language machine. You will notice that these basic traits closely resemble those of the generic computer organization described in Section 1.2. An assembly language machine is designed to support a variety of high-level languages, and is not tailored to a particular high-level language. Thus, programs written in different high-level languages can be translated to the same assembly language.

It is also interesting to note that many of the popular assembly-level architectures are quite similar to each other, just like the case of many popular high-level architectures. Therefore once you master one, it is easy to learn others. This similarity occurs because any given assembly-level architecture closely follows an instruction set architecture (ISA), and ISA design is driven by hardware technology and application requirements. Different ISAs have many things in common, because they target similar application domain, and are interpreted by hardware machines built using similar hardware technologies.

*“Real programmers can write assembly code in any language. :-)”*  
– Larry Wall (the Perl guy)

*“All people smile in the same language.”*  
– Author Unknown

As mentioned in Section 1.8, an architecture specifies what data can be *named* by a program written for that architecture, what operations can be performed on the named data, and what ordering exists among the operations. When writing an assembly language program, the locations that can be named are the (virtual) memory address space, and the registers. The value returned by a read to an address is the last value written to that address. In most languages, *sequential order* is implied among the instructions. That is, instructions are to be executed one after the other, in the order in which they are specified in the program.

### 3.1.1 Assembly Language Alphabet and Syntax

The programmer must tell the computer, through the statements/instructions in the programming language used, *everything* that the computer must do. We shall look at the different facets of an assembly language.

#### 3.1.1.1 Alphabet

High-level languages, as we saw in Chapter 2, are somewhat close to natural languages, and are substantially removed from the underlying hardware levels. An assembly language also uses symbols and words from natural languages such as English, but is closer to the underlying hardware<sup>1</sup>. Formally speaking, an assembly language consists of a set of symbolic names and a set of rules for their use. The symbolic names are called *mnemonics* (which mean “aid to memory” in the Greek language).

#### 3.1.1.2 Syntax

The syntax of a language is the set of rules for putting together different tokens to produce a sequence of valid statements or instructions. In the case of assembly languages, this deals with the rules of using the mnemonics in the specification of complete instructions and assembly language programs. The syntax followed by assembly languages is quite different from that followed by high-level languages. Instructions are generally specified in a single line by an *opcode* mnemonic followed by zero or more *operand* mnemonics. The opcode field may be preceded by a *label*, and the instruction may be followed by a *comment*, which starts with a character such as “;” or “#”. Most of the assembly languages require the label to start in the first column of the line, and instructions to start only from the second column or later. You might wonder why the syntax of an assembly language is so restrictive. The reason is to simplify the assembler, which was traditionally written in assembly language to occupy very little space in memory. Apart from instructions, an assembly language program also contains *assembler directives*, which separate data values from instructions and specify information regarding data values.

### 3.1.2 Memory Model

An assembly-level architecture defines a memory model consisting of a large number of *locations*, each of which is a fixed-size group of storage *cells*. Each cell can store a single *bit*

---

<sup>1</sup>Specifically, an assembly language is a symbolic representation of the machine language—which uses only bit patterns (1s and 0s) to specify information. It uses a richer set of symbols (including the English alphabet) instead of bits, and gives symbolic names to commonly occurring bit patterns, such as opcodes and register specifiers, which make it easier for humans to read and comprehend them. For example, in assembly language, we use instructions such as `add $at, $v0, $v1` in place of bit patterns such as `0000000000100010000110000000000000`.



of information—a 0 or a 1. Most assembly languages consider a location to store 8 bits, or a *byte*. Because a single bit represents a very small amount of information, bits are seldom handled individually; the usual approach is to read or write one or more locations at a time. Most assembly languages provide instructions for manipulating data items of different sizes, such as 8 bits, 16 bits, 32 bits, and 64 bits.

For the purpose of reading or writing memory locations, each memory location is given a unique *address*. The collection of all memory locations is often called the **memory address space**. Although assembly languages have provision to deal with numerical addresses, it is customary to use *labels* to refer to memory locations. As labels can be constructed using alphabetical letters (in addition to numerical digits if required), they are easier for the programmer to keep track of.

The *instructions* and *data* of an assembly language program are strongly tied to the memory address space. Each instruction or data item is viewed as occupying one location (or a contiguous set of locations) in the memory address space. Although the assembly language programmer can assign instructions and data items to memory locations in a random manner, for functional reasons it is better to organize the locations into a few sections, much like how the *maître d'hôtel* organizes a restaurant dining area into smoking and non-smoking sections. Each section holds a chunk of code or data that logically belongs together. Some of the sections commonly used by assembly language programs are **text** (code), **data**, **heap**, and **stack**, as illustrated in Figure 3.1.

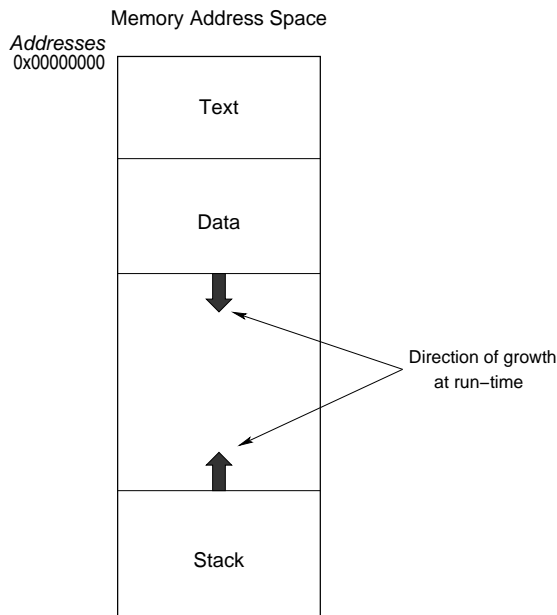


Figure 3.1: Organizing the Memory Address Space as Multiple Sections

The **text** section, as its name implies, is used to allocate instructions, and is read-only from the application program's point of view. The operating system can still write to that section, and uses this ability when loading an application program into memory. Another point to note is that in machines that allow self-modifying code, the text section is read-write.

The **data** section is generally used to store data items that are required throughout the activation of the program. Such items include, for instance, statically allocated global data items and dynamically allocated data items. This section is allowed to grow at run-time as and when allocating new data items dynamically.

The **stack** section is generally used to store data items that are required only during the activation of a subroutine; such items include local variables and parameters to be passed to other subroutines. The stack section is therefore empty at programming time. At program run-time, it starts empty, and then grows and shrinks as subroutines are called and exited. Every time a subroutine is called, the stack section grows by an amount called **stack frame** or **activation record**. The stack frame thus constitutes a private work space for the subroutine, created at the time the subroutine is called and freed up when the subroutine returns. Historically, most machines assume the stack frames to grow in the direction of decreasing memory addresses, although a few machines assume the opposite.

### 3.1.3 Register Model

Most assembly-level architectures include a few registers to store information that needs to be accessed frequently. In the lower-level hardware implementations, these registers are implemented in a manner that permits faster access to them compared to accessing the main memory. This is because of the following reasons, which stem from having only a few registers:

- In a microarchitecture, the decoder or selector used to select a register will be much smaller than the one used to select a memory location.
- In a microarchitecture, the registers are typically implemented inside the processor chip, and so no off-chip access is required to access a register.
- In a device-level architecture, the technology used to implement registers (flip-flops or SRAMs) is faster than the one used to implement memories (DRAMs), which are typically designed for achieving high density.

Apart from these access time advantages, there is yet another advantage at the lower level: a register can be specified with just a few bits in a machine language. This is much less than the 32 or 64 bits required to specify a memory location. For example, in an instruction set architecture that specifies 32 registers, only  $\log_2 32 = 5$  bits are needed to specify a register. Notice that all of these advantages of registers would be lost, if too many registers are specified.

Registers are used for a variety of applications, and generally have names that denote their function. Below, we give the names of commonly used registers and a brief description of their function. Notice that not all machines may have every one of these registers.

- **Program Counter (PC):** This register is specified in virtually every machine, and is used to store the address of the memory location that contains the next instruction to be executed.
- **Accumulator (ACC):** Many of the early machines specified a special register called the accumulator to store the result of all arithmetic and logical operations.
- **Stack Pointer (SP):** This register is used to store the address of the topmost location of the stack section of the memory.
- **Link Register:** This register is used to store the return address when calling a subroutine.
- **General-Purpose Registers (GPRs):** Modern architectures invariably specify a number of GPRs to store key local variables and the intermediate results of computation. Examples are the **AX**, **BX**, **CX**, and **DX** registers in the IA-32 architecture, and registers **\$0** through **\$31** in the MIPS-I architecture. Most architectures specify separate registers for holding integer numbers and floating-point numbers. On some architectures, the GPRs are completely symmetric and interchangeable. That is, to hold a temporary result, the compiler can equally use any of the GPRs; the choice of register does not matter. On other architectures some of the GPRs may have some special functions too. For example, in the IA-32 architecture, there is a register called **EDX**, which can be used as a GPR, but which also receives half the product in a multiplication and half the dividend in a division. Similarly, in the MIPS-I architecture, register **\$31** is a GPR, but is used to store the return address when executing a subroutine call instruction.
- **Flags Register:** If specified, this register stores various miscellaneous bits of information (called *flags* or *condition codes*), which reflect different properties of the result of the most recent arithmetic or logical operation, and are likely to be needed by subsequent instructions. Typical condition code bits include:
  - N** — set if the previous result was negative
  - Z** — set if the previous result was zero
  - V** — set if the previous result caused an overflow<sup>2</sup>

---

<sup>2</sup>Overflow occurs in a computer because of using a fixed number of bits to represent numbers. When the result of an arithmetic operation cannot be represented by the fixed number of bits allotted for the result, then an overflow occurs. The overflow event can be handled in 3 ways: (i) the semantics of the instruction that generated the overflow may include specifications on how the overflow is treated. (ii) the overflow triggers an *exception* event, transferring control to the operating system, which then handles the exception event. (iii) the **V** flag is set to 1 so as to permit subsequent instructions of the application program to monitor the **V** flag and take appropriate action.

C — set if the previous result caused a carry out of the most significant bit (MSB)

A — set if the previous result caused a carry out of bit 3 (auxiliary carry)

P — set when the previous result had even parity.

Flags are set implicitly by certain arithmetic and logical instructions. For example, after a *compare* instruction is executed, the Z flag is used to indicate if the two numbers are equal, and the N flag is used to indicate if the second number is bigger than the first number. A subsequent instruction can test the value of these flags, and take appropriate action. Similarly, the C flag is useful in performing multiple-precision arithmetic. The required multiple-precision addition is done in several steps, with each step doing a single-precision addition. The C flag generated in one step serves as a carry input for the next step. The A flag is useful for performing arithmetic operations on packed decimal numbers.

### 3.1.4 Data Types

In Chapter 2 we saw that declaring and manipulating variables were key concepts in high-level languages; each variable has a *type* associated with it. By contrast, the assembly-level architecture does not have a notion of variables! Instead, the assembly language programmer considers the contents of a register, a memory location, or a contiguous set of memory locations as a data item, and manipulates the contents of these storage locations using instructions.

When a compiler translates an HLL program into an assembly language program, it maps HLL variables to memory locations in the assembly-level architecture. The number of memory locations allocated depends on the variable's type. The memory section or region in which locations are allocated depends on the variable's storage class.

Assembly-level architectures support a variety of data types, such as characters, signed integers, unsigned integers, and floating-point numbers. Support for a particular data type comes primarily in the form of instruction opcodes that interpret a bit pattern as per the definitions of that data type. For example, an assembly language may provide two different ADD instructions—**add** and **fadd**—one that interprets bit patterns as integers, and one that interprets them as floating-point numbers. Remember that in an assembly-level architecture, the data in a particular storage location is not self-identifying. That is, the bits at that storage location do not specify a data type, and therefore have no inherent meaning. The meaning is determined by how an instruction uses them. It is up to the assembly language programmer (or compiler) to use the appropriate opcodes to interpret the bit patterns correctly. Thus, it is the job of the assembly language programmer (or the compiler) to ensure that bit patterns representing integer variables are added together using an integer ADD instruction.

To illustrate this further, we shall use an example. Figure 3.2 shows how two different ADD instructions can produce two different results when adding the same two bit patterns 01010110 and 00010100. In the first case, an *integer add* instruction treats the two patterns

as (binary encoded) integers 86 and 20, and obtains the result pattern 01101010, which has a decimal value 106 when interpreted as an integer. In the second case, a *BCD*<sup>3</sup> *add* instruction treats the two patterns as (BCD encoded) numbers 56 and 14, and obtains a different result pattern of 01110000, which represents the decimal number 70 when interpreted as a BCD number.

| Binary Number System<br>(Unsigned Integers)   | Binary Coded Decimal System<br>(BCD Integers)  |
|---|--|
| $  \begin{array}{r}  01010110 \text{ (86)} \\  + 00010100 \text{ (20)} \\  \hline  01101010 \text{ (106)}  \end{array}  $ | $  \begin{array}{r}  \overset{1}{\curvearrowright} \\  01010110 \text{ (56)} \\  + 00010100 \text{ (14)} \\  \hline  01110000 \text{ (70)}  \end{array}  $ |
| $\uparrow$<br>Decimal   | $\uparrow$<br>Decimal  |

Figure 3.2: An Example Illustrating how the same two Bit Patterns can be added differently to yield Different Results

Below, we highlight the data types that are typically supported in assembly-level architectures.

- **Unsigned Integer and Signed Integer:** Integers are among the most basic data types, and all machines support them. Some machines provide support for unsigned integers as well as signed integers. This support comes primarily in the way of instructions that have a different semantic for recognizing arithmetic overflows.
- **Floating-Point Number:** Most machines support floating-point data type by including specific instructions for performing floating-point arithmetic. Many machines also have separate registers for holding integer values and floating-point values. As mentioned in Chapter 2, an FP number is written on paper as follows:

$$\boxed{(Sign)Significand \times Base^{Exponent}}$$

The *base* in the above equation is the radix of the system, which is a constant for a particular assembly-level architecture, and is usually chosen as 2. The *significand* is used to identify the significant digits of the FP number; the number of bits allotted for the significand determines the precision.

- **Decimal Number:** Some HLLs, notably COBOL, allow decimal numbers as a data type. Assembly-level architectures that were designed to be COBOL-friendly often directly support decimal numbers, typically by encoding a decimal digit in 4 bits and then packing two decimal digits per byte (BCD format). However, instead of

<sup>3</sup>A BCD number is a *binary coded decimal* number. The BCD format is explained later in this section.

providing arithmetic opcodes that work correctly on these packed decimal numbers, they typically provide special decimal-arithmetic-correction opcodes that can be used after an integer addition to obtain the correct BCD answer! These opcodes use the carry out of bit 3, which is available in the A (auxiliary carry) flag.

- **Character:** Most assembly-level architectures support non-numeric data types such as characters. It is not uncommon for an assembly-level architecture to have special opcodes that are intended for handling character strings, that is, consecutive runs of characters. These opcodes can perform copy, search, edit, and other functions on strings.

If an architecture does not support a particular data type, and an arithmetic operation needs to be performed for that data type, the assembly language programmer (or compiler) may have to synthesize that operation using the available instructions. For example, if an architecture does not support the floating-point data type, and there is a need to add two bit patterns as if they are stored in the floating-point format, then the programmer needs to write a routine that separates the exponents and significands, equalizes the exponents by modifying the significands, and then performs the addition and re-normalization operations. Similarly, if the architecture does not support the BCD (binary coded decimal) data type, and if BCD arithmetic needs to be performed on a bit pattern, then the assembly language programmer (or compiler) needs to synthesize that arithmetic operation using sequences of existing instructions.

### 3.1.5 Assembler Directives

An assembly language program instructs the assembly-level machine two things: (i) how to initialize its storage locations (registers and memory), and (ii) how to manipulate the values in its storage locations. The first part is conveyed by means of statements called *assembler directives*, and the second part is conveyed by means of assembly-level *instructions*. (Apart from directives that are used to initialize storage locations, there are some other directives that are *executed* by the assembler and are not assembled. There are also some directives that serve as programming tools, to simplify the process of writing the program.) We shall discuss directives in this section. Instructions will be discussed in the next section.

Without initialization, the values in registers and memory locations would be undefined. The bulk of the initialization done by the directives involves the memory space and those registers that point to different sections in the memory space, such as the program counter, the global pointer, and the stack pointer. Specifically, they indicate which static sections the program will use (`.text`, `.rdata`, `.data`, etc<sup>4</sup>), how big these sections are, where their boundaries are, and what goes into the different data sections.

---

<sup>4</sup>Dynamic sections such as heap and stack are created during execution, and are therefore not specified by directives.

For example, to place the subsequent statements of the program in the `.data` section of memory, we can use a directive such as:

```
.data
```

And, to initialize the next 4 bytes to 58, we can use a directive such as:

```
.word 58
```

When an assembly language program is translated to machine language, the directives are translated to form the machine language program's header, section headers, and various data sections. A directive therefore does not translate to machine language instructions. At the execution time of a machine language program, the initialization task is performed by the *loader* part of the OS, which reads the program's header, section headers, and sections to do so.

### 3.1.6 Instruction Types and Instruction Set

Apart from assembler directives, an assembly language program includes assembly-level instructions also. In fact, the instructions form the crux of the program; a program without instructions would only be initializing the registers and memory! As mentioned before, instructions manipulate the values present in registers and memory locations. For example, to copy the contents of memory location `A` to register `$t0`, we can use an instruction such as:

```
lw    $t0, A
```

Here, the mnemonic `lw` is an abbreviation for **load word**. Similarly, to add the contents of registers `$t1` and `$t2` and to place the result in register `$t3`, we can use an instruction such as:

```
add    $t3, $t1, $t2
```

A typical program involves performing a number of functionally different steps, such as adding two numbers, testing for a particular condition, reading a character from the keyboard, or sending a character to be displayed on a video screen. Each assembly-level architecture has its own set of instructions, called its **instruction set**. In practice, many of the instruction sets are quite similar. The instructions in an instruction set can be functionally classified into four categories:

- Data transfer
- Data manipulation
- Program sequencing and control
- Trap or syscall

**Data Transfer:** Data transfer instructions copy data from one storage location to another, without changing the data stored in the source location. Typical transfers of data

are between registers and main memory locations; between registers and stack locations; or or between registers themselves. Variables declared in a high-level language are generally allocated locations in main memory (some are allocated in general-purpose registers), and so most of the data reside initially in main memory. From the main memory, data is often copied to general-purpose registers or the stack, prior to operating on them. Data transfer instructions are quite useful in this endeavor. Some of the commonly used data transfer instructions in different assembly languages and their semantics are given in Table 3.1.

| Mnemonic | Semantics   |
|----------|---|
| MOV      | Copy data from one register/memory location to another    |
| LOAD     | Copy data from a memory location to a register            |
| STORE    | Copy data from a register to a memory location            |
| PUSH     | Copy data from a register/memory location to top of stack |
| POP      | Copy data from top of stack to a register/memory location |
| XCH      | Exchange the contents of two register/memory locations    |

Table 3.1: Common Data Transfer Instructions in Different Assembly Languages

**Data Manipulation:** Data manipulation instructions perform a variety of operations on data and modify them. These are instrumental for implementing the operators and assignments of HLL programs. There are three types of data manipulation instructions: arithmetic, logical, and shift. Some of the commonly used data manipulation instructions in different assembly languages and their semantics are given in Table 3.2. The input operands for these instructions are specified as part of the instruction, or are available in storage locations such as memory locations, registers, or stack. The result of the instruction is also stored in a storage location. Recently, many assembly languages have included instructions that are geared for speeding up multimedia applications.

**Program Sequencing and Control:** Normally, the instructions of a program are executed one after the other, in a straightline manner. Control-changing instructions are used to deviate from this straightline sequencing. A conditional branch instruction is an instruction that causes a control flow deviation, if and only if a specific condition is satisfied. If the condition is not satisfied, instruction sequencing proceeds in the normal way, and the next instruction in sequential address order is fetched and executed. The condition can be the value stored in a *condition code* flag, or the result of a comparison. Conditional branch instructions are useful for implementing *if* statements and loops. Besides conditional branches, instruction sets also generally provide unconditional *branch* instructions. When an unconditional branch is encountered, the sequencing is changed irrespective of any condition. Finally, assembly languages also include *call* instructions. and *return* instructions



| Mnemonic | Semantics  |
|----------|--|
| ADD      | Add two operand values and store the result value                          |
| SUB      | Subtract one operand value from another and store the result value         |
| MULT     | Multiply two operand values and store the result value                     |
| DIV      | Divide an operand value by another and store the quotient remainder values |
| INC      | Increment an operand value and store the result value in the same location |
| DEC      | Decrement an operand value and store the result value in the same location |
| ABS      | Find the absolute value of the operand value and store the result          |
| AND      | And two operand values and store the result value                          |
| OR       | Or two operand values and store the result value                           |
| XOR      | Exor two operand values and store the result value                         |
| LSHIFT   | Left shift one operand value by another and store the result value         |
| RSHIFT   | Right shift one operand value by another and store the result value        |
| LROT     | Left rotate one operand value by another and store the result value        |
| RROT     | Right rotate one operand value by another and store the result value       |

Table 3.2: Common Data Manipulation Instructions in Different Assembly Languages

to implement subroutine calls and returns. Some of the commonly used program control and sequencing instructions in different assembly languages and their semantics are given in Table 3.3. The first group of instructions in the table are conditional branch instructions; the second group are unconditional branches, and the third group deal with subroutine calls and returns.

**Trap or Syscall Instructions:** The instructions we saw so far cannot do IO operations or terminate a program. For performing such operations, an application program needs to call the services of the operating system (OS). Trap or syscall instructions are used to transfer control to the OS, in order for the OS to perform some task on behalf of the application program. That is, this instruction is used to invoke an OS service. Once the OS completes the requested service, it can return control back to the interrupted application program by executing an appropriate Kernel mode instruction. If a single trap instruction is provided for specifying different types of services, then the required service is specified as an operand of the instruction.

| Mnemonic | Semantics  |
|----------|--|
| JZ       | Jump if Z flag is set                              |
| JNZ      | Jump if Z flag is not set                          |
| JC       | Jump if C flag is set                              |
| JNC      | Jump if C flag is not set                          |
| BEQ      | Branch if both operand values are equal            |
| BNE      | Branch if both operand values are not equal        |
| BLT      | Branch if one operand value is less than the other |
| JMP      | Jump unconditionally                               |
| B        | Branch unconditionally                             |
| JR       | Jump to address in specified register              |
| CALL     | Call specified subroutine                          |
| JAL      | Jump and link to specified subroutine              |
| RETURN   | Return from subroutine                             |

Table 3.3: Some of the Common Program Sequencing and Control Instructions in Different Assembly Languages

### 3.1.7 Program Execution

Consider a simple assembly language program to add the contents of memory locations 1000 and 1004, and store the result in memory location 1008.

```

        .text
__start: lw      $t0, 1000
        lw      $t1, 1004
        add     $t1, $t1, $t0
        sw      $t1, 1008
        sys_halt

```

The programmer's view is that when this program is being executed, it is present in the computer's main memory. The first line in the program, namely `.text`, is a *directive* indicating that the program be placed in the `.text` section of main memory. This line is called an *assembler directive*, and is not part of the executed program. The next line, which contains the first instruction, begins with the *label* `__start`. This label indicates the memory address of the first instruction. Figure 3.3 shows how this small program might be loaded in the `.text` section of the memory space, when it is about to be executed. The five instructions of the program have been placed in successive memory locations, in the same order as that in the program, starting at location `__start`. Notice that the label `__start` does not explicitly appear in the program stored in memory, nor does the directive `.text` in the first line. The comment portions of the statements also do not appear in memory.

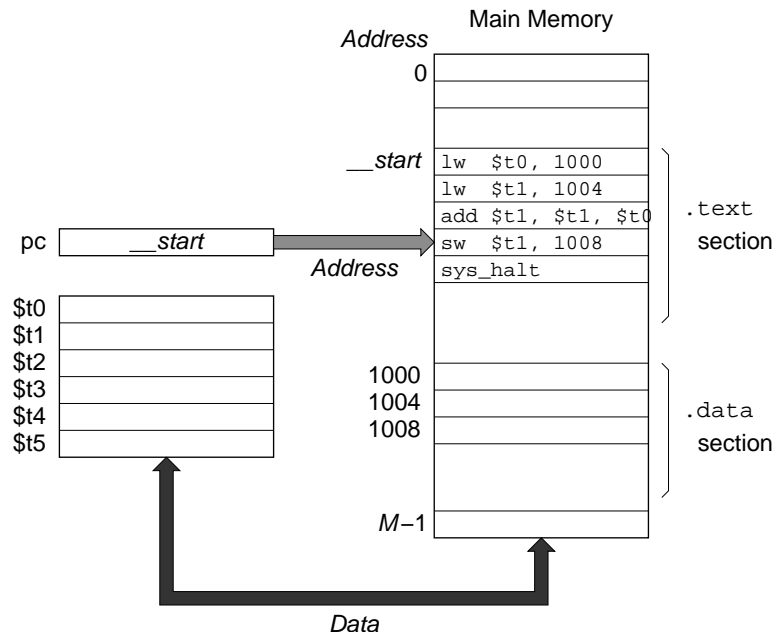


Figure 3.3: A Possible Placement of the Sample Program in Memory

Let us consider how this program is executed. To begin executing the program, the address of its first instruction ( $P$ ) is placed into **pc**. This instruction is executed and the contents of **pc** are advanced to point to the next instruction. Then the second instruction is executed, and the process is continued until the computer encounters and executes the `sys_halt` instruction. The last instruction transfers control to the OS, and tells it to terminate this program. As you can see, instructions are executed in the order of increasing addresses in memory. This type of sequencing is called *straight-line sequencing*.

### 3.1.8 Challenges of Assembly Language Programming

Because an assembly language is less abstract than a high-level language, programming in assembly is considerably more difficult than programming in a high-level language. The lack of abstraction manifests itself in different ways:

- First, the storage resources are more concrete at the assembly level, which has the notion of registers and memory locations, as opposed to variables. The programmer must manage the registers and memory locations at every step of the way. In machines with *condition codes*, the programmer must keep track of the status of condition codes and know what instructions affect them before executing any conditional branches. This sounds tedious, if not difficult.

- Another important difference is that the data items in an assembly language program are not *typed*, meaning they are not inherently specified as belonging to a particular type such as integer or floating-point number. Assembly languages provide different instructions for manipulating a data item as an integer or another data type. Thus, it is the responsibility of the programmer to use the appropriate instructions when manipulating data items. Even then, the number of data types supported in an assembly-level architecture is fewer than that in the high-level architecture; only a few simple data types such as integers, floating-point numbers, and characters are supported. Thus, all of the complex data types and structures supported at the HLL level must be implemented by the assembly language programmer using simple primitives.
- In assembly language programs, most of the control flow changes must be implemented with branch instructions whose semantics are similar to those of the “go to” statements used in HLLs.
- The amount of work specified by an assembly language instruction is generally smaller than that specified by an HLL statement<sup>5</sup>. This means that several assembly language instructions are usually needed to implement the equivalent of a typical HLL statement.
- Assembly language programs take much longer to debug, and are much harder to maintain.
- Finally, it is difficult for assembly language novices, particularly those with high-level language experience, to think at a low enough level.

All of these factors make it difficult to program in assembly language. Apart from these difficulties, there is a practical consideration: an assembly language program is inherently tied to a specific instruction set, and must be completely rewritten to run on a machine having a different instruction set. Because of these reasons, most of the programming done today is in a high-level language.

### 3.1.9 The Rationale for Assembly Language Programming

Under the above circumstances, why would anyone want to program in assembly language today? There are at least two reasons:

1. Speed and code size
2. Access to the hardware

---

<sup>5</sup>Exceptions to this general case are the vector instructions and the MMX<sup>TM</sup> instructions.

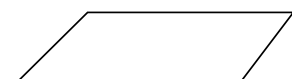
First of all, an expert assembly language programmer can often produce code that is much smaller and much faster than the code obtained by compiling an equivalent HLL program. For some applications, speed and code size are critical. Many embedded applications, such as the code in a smart card, the code in a cellular telephone, the code in an anti-lock brake control, device drivers, and the inner loops of performance-critical applications fall in this category. Second, some functions need complete access to the hardware features, something usually impossible to specify in high-level languages. Some hardware features have no analogues in high-level languages. The low-level interrupt and trap handlers in an operating system, and the device drivers in many embedded real-time systems fall into this category. Similarly, an assembly language programmer may be able to make better use of special instructions, such as string copy instructions, pattern-matching instructions, and multimedia instructions such as the MMX<sup>TM</sup> instructions. Many of these special instructions do not have a direct equivalent in high-level languages, thereby forcing the HLL programmer to use loops. Compilers, in most cases, cannot determine that such a loop can be replaced by a single instruction, whereas the assembly language programmer can easily determine this.

## 3.2 Assembly-Level Interfaces

The assembly-level architecture attributes that we saw so far pertain to the assembly language specification part of the architecture. In many contexts, by assembly-level architecture, we just mean this assembly language specification. For serious programming in assembly, we need to enhance the architecture by the following two additional parts:

- assembly-level interface provided by libraries
- assembly-level interface provided by the OS

*Assembly Language  
Programmer*



Assembly Language  
Specification  
(User Mode)



Assembly-Level  
Interface  
(Library)



Assembly-Level  
Interface  
(OS)

Figure 3.4: Three Different Parts of Assembly-Level Architecture

### 3.2.1 Assembly-Level Interface Provided by Library

### 3.2.2 Assembly-Level Interface Provided by OS

## 3.3 Example Assembly-Level Architecture: MIPS-I

We shall next take a more detailed look at assembly-level architectures. It is easier to do so using an example architecture. The example architecture that we use is the well-known MIPS-I assembly-level architecture [ref]. We use the MIPS-I architecture because it is one of the simplest architectures that has had good commercial success, both in the general-purpose computing world and in the embedded systems world. The MIPS instruction set architecture had its beginnings in 1984, and was first implemented in 1985. By the late 1980s, this architecture had been adopted by several workstation and server companies, including Digital Equipment Corporation and Silicon Graphics. Today MIPS processors are widely used in Sony and Nintendo game machines, palmtops, laser printers, Cisco routers, and SGI high-performance graphics engines. Although it is not popular anymore in the desktop computing world, the availability of sophisticated MIPS-I simulators such as **SPIM** makes it possible for us to develop MIPS-I assembly language programs and simulate their execution. All of these features makes MIPS-I an excellent architecture for use in Computer Architecture courses. Below, we look at some of the important aspects of the MIPS-I architecture; interested readers may refer to [refs] for a more detailed treatment.

### 3.3.1 Assembly Language Alphabet and Syntax

The MIPS-I assembly language format is line oriented; the end of a line delimits an instruction. Each line can consist of up to four fields, as shown in Figure 3.5: a label field, an opcode field, an operand field, and a comment field. The language is free format in the sense that any field can begin in any column, but the relative left-to-right ordering of the fields must be maintained. For the sake of clarity, we will align the fields in each of our code snippets.

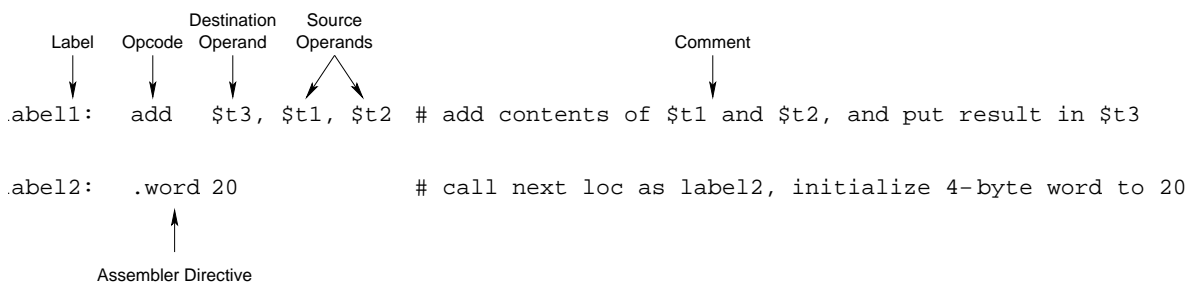


Figure 3.5: Format of a MIPS-I Assembly Language Statement

A *label* is a symbolic name used to identify a memory location that is explicitly referred to in the assembly language program. A label consists of any sequence of alphanumerical characters, underscores (`_`), dollar signs (`$`), or periods (`.`), as long as the first character is not a digit. A label must be followed by a colon. Labels are particularly useful for specifying the target of a control-changing instruction and for specifying the memory location corresponding to a variable.

After the optional label field, the next field specifies an *opcode* or an *assembler directive*. An opcode specifies the operation to be done by the instruction.

The operand field in an assembly language statement specifies the destination operand and source operand(s) of the instruction. Operands are separated by commas, and the destination operand (if present) appears in the leftmost position in the operand field, except for store instructions. For instance, in the assembly language instruction “`add $t3, $t1, $t2,`” the source operands are registers `$t1` and `$t2`, and the destination operand is register `$t3`. Numbers are interpreted as decimal, unless preceded by `0x` or succeeded by `H`, either of which denotes a hexadecimal number. Multiple instructions can be written in a single line, separated by semicolons.

The comment field, which comes last, begins with a sharp sign (`#`), and terminates at the end of the line. Thus, all text from a `#` to the end of the line is a comment<sup>6</sup>. Just as in high-level languages, the comments are only intended for human comprehension, and are ignored by the assembler. A good comment helps explain a non-intuitive aspect of one or more instructions. By providing additional insight, such a comment provides important information to a future programmer who wants to modify the program.

### 3.3.2 Register Model

The MIPS-I assembly-level architecture models 32 general-purpose 32-bit integer registers named `$0` - `$31`, 32 general-purpose 32-bit floating-point registers named `f0` - `f31`, and three special integer registers named `pc`, `hi`, and `lo`. Two of the general-purpose integer registers (`$0` and `$31`) are also somewhat special. Register `$0` always contains the value 0, and can never be changed. If an instruction specifies `$0` as the destination register, the register contents will not change when the instruction is executed. Register `$31` can be used as a general-purpose register, but it has an additional use as a *link register* for storing a subroutine return address, as we will see in Section 3.4.6.

The floating-point arithmetic instructions use the FP registers as the source as well as the destination. However, they can specify only the 16 even-numbered FP registers, `$f0`, `$f2`, `$f4`, ..., `$f30`. When specifying an even-numbered FP register, if the operand is a double-precision FP number, the remaining 32 bits of the number are present in the subsequent odd-numbered FP register. For instance, when `$f0` is specified for indicating a

---

<sup>6</sup>Although a line can contain nothing other than a comment, starting a comment from the very first column of a line is not a good idea. This is because most assemblers invoke the C preprocessor `cpre`, which treat them as preprocessor commands.

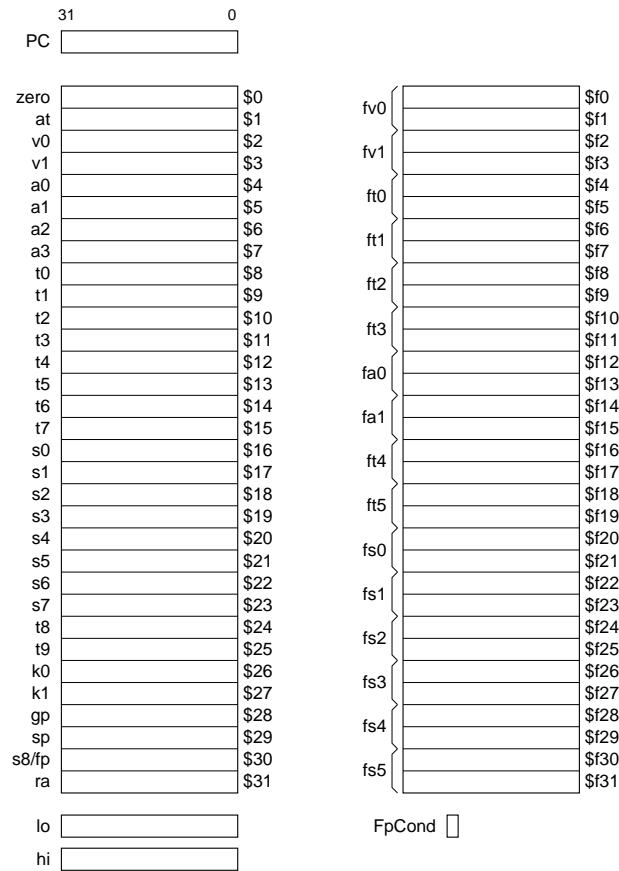


Figure 3.6: MIPS-I User Mode Register Name Space

double-precision operand, the 64-bit number is present in the register pair { $\$f0$ ,  $\$f1$ }.

### Explain HI and LO registers

## MIPS-I Assembly Language Conventions for Registers

The MIPS-I assembly language follows some conventions regarding the usage of registers. These conventions are *not* part of the assembly-level architecture specifications. This means that if you write stand-alone assembly language programs that do not adhere to these conventions, they are still guaranteed to be assembled and executed correctly. However, if you do not adhere to these conventions, you cannot use the standard libraries and the standard operating system, because they have been already compiled with the MIPS-I compiler, which follows these conventions. Some of the important conventions are given below:



- Register **\$at** (\$1) is reserved for use by the assembler for computing certain memory addresses.
- Register **\$sp** (\$29) is reserved for use as the stack pointer.
- The first four integer parameters of a subroutine are passed through registers **\$a0-\$a3** (\$4-\$7). Thus, the subroutines in the standard libraries and operating system have been developed with the assumption that their first 4 parameters will be present in these registers. Similarly, the first two floating-point parameters are passed through FP registers **\$fa0** and **\$fa1** (\$f12 and \$f14). The remaining parameters are passed through the stack frame.
- The integer return values of a subroutine are passed through registers **\$v0** and **\$v1** (\$2 and \$3).
- Register **\$v0** (\$2) is used to specify the exact action required from the operating system (OS) when executing a syscall instruction.
- Registers **\$k0** and **\$k1** (\$26 and \$27) are reserved for use by the operating system.

Table 3.4 gives the names by which the MIPS registers are usually known. These names are based on the conventional uses for the different registers, as explained above.

| Register Number                          | Register Name | Typical Use                                    |
|--|---------------|--|
| \$0                                      | \$zero        | Zero constant, destination of nop instruction  |
| \$1                                      | \$at          | Assembler temporary; reserved for assembler    |
| \$2-\$3                                  | \$v0-\$v1     | Values returned by subroutines                 |
| \$4-\$7                                  | \$a0-\$a3     | Arguments to be passed to subroutines          |
| \$8-\$15                                 | \$t0-\$t7     | Temporaries used by subroutines without saving |
| \$16-\$23                                | \$s0-\$s7     | Saved by subroutines prior to use              |
| \$24-\$25                                | \$t8-\$t9     | Temporaries used by subroutines without saving |
| \$26-\$27                                | \$k0-\$k1     | Kernel uses for interrupt/trap handler         |
| \$28                                     | \$gp          | Global pointer                                 |
| \$29                                     | \$sp          | Stack pointer                                  |
| \$30                                     | \$s8/\$fp     | Saved by subroutines, frame pointer            |
| \$31                                     | \$ra          | Return address for subroutines                 |
| \$f0, \$f2                               | fv0-fv1       | Values returned by subroutines                 |
| \$f4, \$f6, \$f8, \$f10                  | ft0-ft3       | Temporaries used by subroutines without saving |
| \$f12, \$f14                             | fa0-fa1       | Arguments to be passed to subroutines          |
| \$f16, \$f18                             | ft4-ft5       | Temporaries used by subroutines without saving |
| \$f20, \$f22, \$f24, \$f26, \$f28, \$f30 | fs0-fs5       | Saved by subroutines prior to use              |

Table 3.4: Conventional Names and Uses of MIPS-I User Mode Registers

### 3.3.3 Memory Model

The MIPS-I assembly-level architecture models a linear memory address space (i.e., a flat address space) of  $2^{31}$  memory locations that are accessible to user programs. These locations have addresses ranging from 0 to `0x7fffffff`, as indicated in Figure 3.7. Each address refers to a byte, and so a total of 2 Gbytes of memory can be addressed. Although each address refers to a single byte, MIPS-I provides instructions that simultaneously access one, two, three, or four contiguous bytes in memory. The most common access size is 4 bytes, which is equal to the width of the registers. Although a location can be specified by its address, the common form of specification in assembly language is by a *label* or by an *index register* along with an *offset*. In the latter case, the memory address is given by the sum of the register contents and the offset.

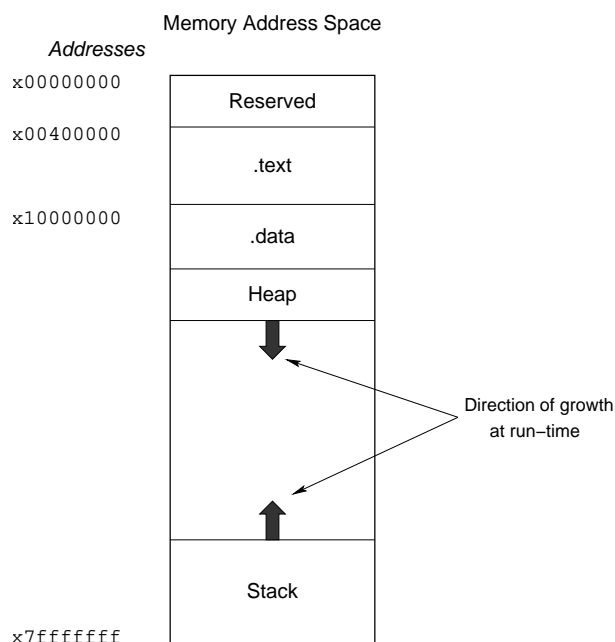


Figure 3.7: Organization of MIPS-I User Memory Address Space

### MIPS-I Assembly Language Conventions for Memory

Like the case with registers, there are MIPS-I assembly language conventions for the memory address space also. Again, these conventions are *not* part of the assembly-level architecture specifications. Figure 3.7 indicates the conventional organization of the MIPS-I memory address space, in terms of where the different sections of the memory start. The conventional starting points for the `.text` and `.data` sections are at addresses `0x400000` and `0x10000000`,

respectively. The user run-time stack starts at address `0x7fffffff`, and grows towards the lower memory addresses. General-purpose register `$sp` is generally used as the *stack pointer*; stack operands are accessed by specifying an *offset* value that is to be added to the stack pointer. Thus, operands that are buried within the stack can also be accessed. Local variables of a subroutine are allocated on the stack. If a subroutine needs the stack to grow, for allocating local variables and temporaries, it decrements `$sp` by the appropriate amount at the beginning, and increments `$sp` by the same amount at the end.

### 3.3.4 Assembler Directives

The MIPS-I assembly-level architecture supports several assembler directives, all of which are written with a dot as their first character. Below, we describe some of the commonly used directives.

- **.rdata**: indicates that the subsequent items are to be stored in the **read-only data** section.
- **.data**: indicates that the subsequent items are to be stored in the **.data** section.
- **.text**: indicates that the subsequent items are to be stored in the **.text** section.
- **.comm**, **.lcomm**: are used to declare uninitialized, global data items. These directives are commonly used when the initial value of a variable is not known at programming time. An item declared with the **.comm** directive can be accessed by all modules that declare it. (The linker allocates memory locations for such an item in the **.bss** or **.sbss** section.) An item declared with the **.lcomm** directive is a global variable that is accessible within a single module. (The assembler allocates memory locations for such an item in the **.bss** section or in the **.sbss** section.)
- **.byte**, **.half**, **.word**: These directives are used to set up data items that are 1, 2, and 4 bytes, respectively. In contrast to the **.comm** directive, these directives provide the ability to initialize the data items. Example declarations using these directives are given below. The last two declarations correspond to arrays, and are extensions of the one used for specifying a single item.

---

```

b:   .byte  5           # Allocate an 1-byte item with initial value 5
                        # at next memory location, and name it b
h:   .half   5          # Label next memory location as h;
                        # allocate next 2 bytes for a 2-byte item with initial value 5
w:   .word   5          # Label next memory location as w;
                        # allocate next 4 bytes for a 4-byte item with initial value 5
ba:  .byte   0:5         # Label next memory location as ba;
                        # allocate next 5 bytes to 5 1-byte items with initial value 0
wa:  .word   1:2, 4      # Label next memory location as wa;
                        # allocate next 12 bytes to 3 4-byte items with initial values 1, 1, 4

```

---

In these directives, if no integer value is specified after `.byte`, `.half`, `.word`, then the item is initialized to zero.

- **.float**: is used to specify a 4-byte data item that is initialized to a single-precision floating-point number.
- **.ascii** and **.asciiz**: are used to declare ASCII strings, without and with a terminating null character, respectively. In the following example, both directives define the same string.

```
a:  .ascii "good\0"    # Place string "good" in memory at location a
z:  .asciiz "good"     # Place string "good" in memory at location z
```

- **.space**: is used to increment the current section's location counter by a stipulated number of bytes. This directive is useful to set aside a specified number of bytes in the current section.

```
s:  .space 40          # Label current section's location counter as s
                        # and increment it by 40 bytes
```

- **.globl**: indicates that the subsequent variable name or label is globally accessible, and can be referenced in other files. For example,

---

```
        .data          # Subsequent items are stored in the data section
        .globl g       # Label g (in this case, a variable) is global
g:      .word 0         # Declare a 4-byte item named g with initial value 0

        .text          # Subsequent items are stored in the text section
        .globl f       # Label f (in this case, the beginning of a function) is global
f:      subu    $sp, 24  # Decrement stack pointer register by 24
```

---

- **.align**: is used to specify an alignment. The alignment is specified as a power of 2.

### 3.3.5 Assembly-Level Instructions

We have already seen some of the instructions and addressing modes of the MIPS-I assembly-level architecture. The MIPS-I architecture supports the following addressing modes: register direct, memory direct, register-relative, immediate, and implicit. Let us now examine the MIPS-I instruction set in a more comprehensive manner<sup>7</sup>.

The MIPS-I architecture is a **load-store architecture**, which means that only load instructions and store instructions can access main memory. The commonly used **lw** (load word) instruction fetches a 32-bit word from memory. The MIPS-I architecture also provides

---

<sup>7</sup>The assembly language examples in this book use only a subset of the MIPS-I assembly-level instruction set. A complete description of the MIPS-I assembly-level instruction set is available in Appendix \*.

separate load instructions for loading a single-byte (**lb** opcode) and a 2-byte half-word (**lh** opcode).

As we saw earlier, conventional MIPS assembly language designates a portion of the memory address space as a *stack* section. Special stack support such as **push** and **pop** instructions are not provided, however. Locations within the stack section of the memory space are accessed just like the remaining parts of memory.

**Arithmetic and Logical Instructions:** MIPS-I provides many instructions for performing arithmetic and logical operations on 32-bit operands. All arithmetic instructions and logical instructions operate on register values or immediate values. While the register operands are always 32 bits wide, the immediate operands can be shorter. These instructions explicitly specify a destination register, and two source operands, of which one is a register and the other is a register or immediate value.

**Control-changing Instructions:** The MIPS-I architecture includes several conditional as well as unconditional branch instructions. The conditional branch instructions base their decisions on the contents of one or two registers.

**System Call Instructions:** The MIPS-I architecture provides a **syscall** instruction for user programs to request a variety of services from the operating system (OS). The specific service requested is indicated by a code value stored in register **\$v0**. The list of services supported and the specific code for each service varies from one OS to another<sup>8</sup>.

**SPIM's ABI — System Calls Supported by SPIM:** The SPIM simulation tool includes a very simple operating system that supports a small set of system calls. These system calls are listed in Table 3.5. If you are familiar with the standard calls supported by different UNIX-style and Windows-style operating systems, you will notice that these system calls are somewhat different. Apart from the much smaller set of calls supported, the functionality provided by the SPIM calls is not very primitive and is more at the level of library routines.

### 3.3.6 An Example MIPS-I AL Program

We are now familiar with the syntax of the MIPS-I assembly language. Next, we shall put together some of the concepts we learned by writing a simple MIPS-I assembly language

---

<sup>8</sup>The differences in the range of services and system call codes between different OSes imply that if a user program uses **syscall** instructions to directly request the OS for services—instead of going through library functions—then the program may not be portable across different OSes. Thus, a **syscall**-laden assembly language program targeted specifically for the SPIM OS will most likely not run correctly on an ULTRIX OS host machine.

| OS Service   | Code in \$v0 | Arguments   | Return Value  |
|--------------|--------------|---|---|
| print_int    | 1            | \$a0: integer to be printed                                     | \$v0: integer read<br>\$fv0: float read<br>\$fv0: double read |
| print_float  | 2            | \$fa0: float to be printed                                      |   |
| print_double | 3            | \$fa0: double to be printed                                     |   |
| print_string | 4            | \$a0: address of string to be printed                           |   |
| read_int     | 5            |   |   |
| read_float   | 6            |   |   |
| read_double  | 7            |   |   |
| read_string  | 8            | \$a0: address for placing read string;<br>\$a1: number of bytes | \$v0: address   |
| sbrk         | 9            | \$a0: amount  |   |
| exit         | 10           |   |   |

Table 3.5: System Calls Supported by SPIM

program. Let us write an assembly language program that prints the familiar “hello, world\n” string, whose C code and Java code are given below.

---

**Program 4** The Hello World! program in Java.

---

```

main() {
    // Display the string
    printf("hello, world!");
}

class helloworld {
    public static void main(String[] args) {
        // Display the string
        System.out.println("hello, world!");
    }
}

```

---

For writing the MIPS AL program, we use the ABI (application binary interface) supported by SPIM. This will help motivated students to ‘execute’ this program using any of the {spim, xspim, PCSpim} tool set. One point to note when using these tools to ‘execute’ assembly language programs is that prior to execution, the tools assemble the program into the equivalent machine language program. Thus, the tools directly simulate the execution of machine language instructions, and not the assembly language instructions. The mem-

ory map displayed by these tools therefore indicates machine language instructions, and not assembly language instructions.

---

```
#####
# data section
#####
        .data                # Store subsequent items in the .data section
string1:                # Label next location as string1
        .asciiz "hello, world\n" # Allocate subsequent bytes and store string "hello, world\n"

#####
# text section
#####
        .text                # Store subsequent items in the .text section
        .globl __start       #
__start:                # Program execution begins here
        la      $a0, string1  # Place the memory address labeled string1 in $a0
        li      $v0, 4        # Place code for print_string system call in $v0
        syscall              # Call OS to print the string

        li      $v0, 10       # Place code for exit system call in $v0
        syscall              # Call OS to terminate the program
```

---

This MIPS-I AL program contains only 2 sections: `.data` and `.text`. The `.data` section has a single declaration—that of the “`hello, world\n`” string, which is declared using the `.asciiz` directive, and given the label `string1`. The string therefore starts at label `string1`.

The `.text` directive tells the assembler to place the subsequent items in the `.text` section. The `__start` label indicates that execution of the program should begin at that point. Although we have placed the `__start` label just before the very first instruction in this program, this is not a rule; the `__start` label can be placed anywhere within the `.text` section. In order to print the string, we have to use the services of the operating system via a `syscall` instruction. In this example code, we have used the `syscall` convention followed by SPIM, a simulator for the MIPS-I architecture. The address of the string (i.e., label `string1`) is placed in register `$a0`, and the code for the `print_string` system call in SPIM (which is 4) is placed in register `$v0`, prior to the `syscall` instruction. Notice that this program will not run on a standard MIPS host, because standard OSes use a different ABI. The standard ABI does not support the `print_string` system call, and instead uses the code 4 for the `write` system call.

Finally, after printing “`hello, world\n`”, the program should terminate. It does so, via another `syscall` instruction, after placing in register `$v0` the value 10, which is the code for the `exit` system call in SPIM. (The standard ABI defines a code value of 1 for the `exit` system call, in contrast to the value of 10 used by SPIM’s ABI.)

### 3.3.7 SPIM: A Simulator for the MIPS-I Architecture

Let us elaborate on the SPIM simulator, as it is a very useful tool for learning MIPS-I assembly language programming. This simulator was developed by Prof. James Larus at University of Wisconsin-Madison, and is available to the public from an `ftp` site at that University. The simulator comes in 3 different flavors: `spim`, `xspim`, and `PCSpim`. Among these, `spim` is the most basic one, providing a terminal-style interface on Unix/Linux hosts, and a DOS interface or console interface on Windows hosts. The `xspim` and `PCSpim` tools are fancier and provide graphical user interfaces (GUI). `xspim` runs on Unix/Linux hosts and provides an X window interface, whereas `PCSpim` runs on Windows hosts and provides a Windows interface.

To execute the above program with `xspim` in a Unix/Linux host, type the command: `xspim -notrap &`. The `-notrap` option tells SPIM not to add its own start-up code, and to begin execution at the `__start` label. An `xspim` window pops up, as shown in Figure 3.8. You can use the `load` button in this window to read and assemble your MIPS-I assembly language program. If any errors are generated during assembly, they are indicated in the bottom portion of the `xspim` window. After fixing the bugs in the AL program file, you can reload the program file into `xspim` by first clearing the memory and register contents—using the `memory & registers` option in the `clear` button—and then using the `load` button to reload the program.

If SPIM has successfully assembled your program, then you can use the `run` or `step` buttons in the `xspim` window to execute your program. In this case, a console window will be automatically opened to display `hello world`. `xspim` provides many other useful features such as breakpoints and a debugger, which make it even more attractive than a real MIPS host for developing MIPS-I assembly language programs. You are encouraged to consult the SPIM manual for learning and using these features.

## 3.4 Translating HLL Programs to AL Programs

We just saw the rudiments of writing a complete program in MIPS-I assembly language, and ‘executing’ it on the SPIM simulator. We shall next take a look at how a high-level language program is translated to assembly language. After all, most of today’s programming is done in a high-level language. We shall revisit direct programming in assembly language when we discuss device drivers and exception handlers in the next chapter.

The translation of a program from a high-level language to the assembly language is typically done by a program called a *compiler*. A detailed treatment of the algorithms used by a compiler is beyond the scope of this book; therefore we restrict our discussion to a sketch of the important ideas. For illustrating the translation process, we again use the MIPS-I assembly language. We illustrate the utility of various instruction types with practical examples of translation from C language to MIPS-I assembly language.



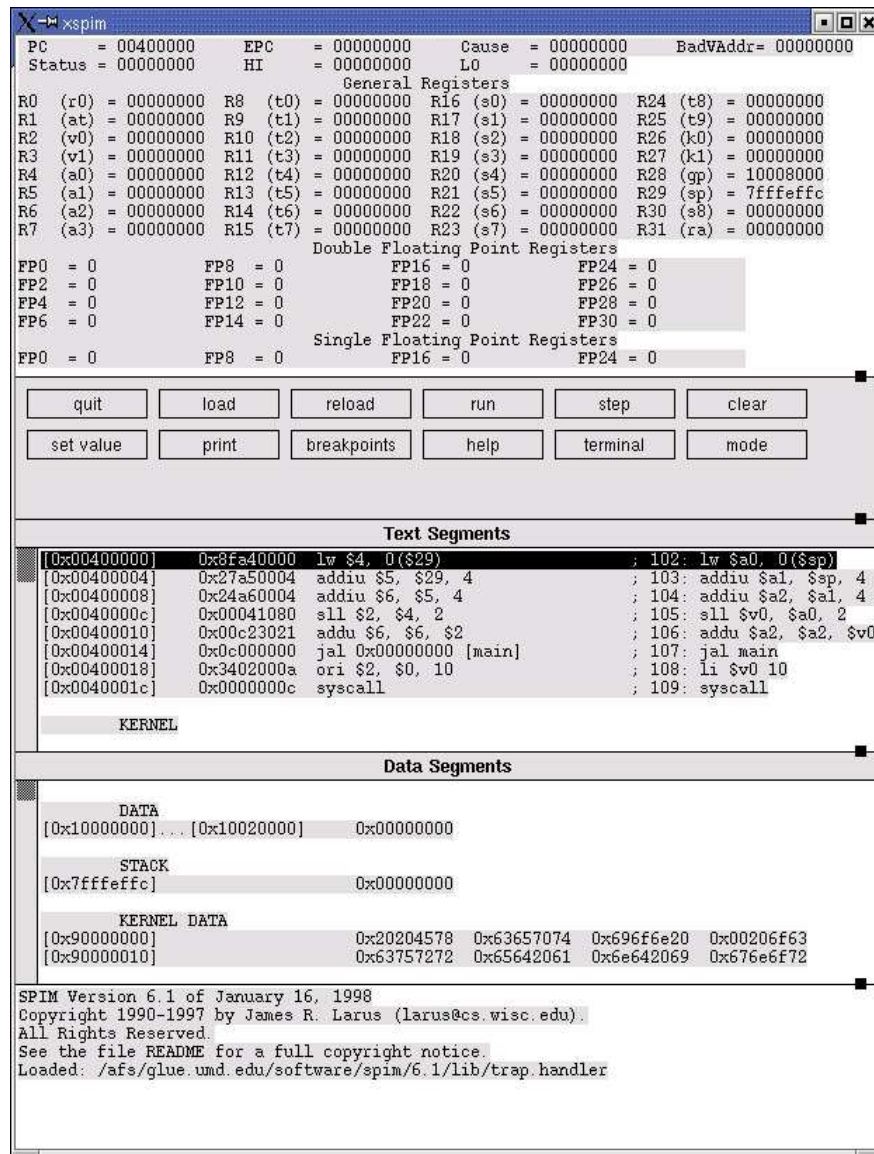


Figure 3.8: A Sample xspim Window

### 3.4.1 Translating Constant Declarations

Constants are objects whose values do not change during program execution. Translating a constant declaration typically involves assigning memory location(s) for the constant. The number of locations required for a constant depends on its size. The memory location(s) will hold the constant's value (a pattern of 0s and 1s) during the execution of the program. Notice that in machines that support only aligned words, integers and floating-point con-

stants need to start on a word boundary. The standard practice is to allocate them in the `.rdata` section of memory.

---

```
#define const 32000
```

---

#### Sample Constant Declaration

---

One way of translating this HLL constant declaration to assembly language is given below. In this translation, we have used a label to specify a memory address that will store the value of the constant. For clarity, we have named this label with the same name as the HLL constant name. The `.rdata` directive tells that the subsequent items need to be placed in the `.rdata` (read-only data) section<sup>9</sup>. The HLL constant `const` is allocated space in memory with the `.word` directive, which allocates 4 bytes. Subsequent instructions that use this constant value will read it from memory.

---

```
# Assign memory locations for the constants and initialize them
.rdata                                # Subsequent items are stored in the .rdata section
const:.word 32000                    # Label next memory location as const;
                                     # allocate next 4 bytes for constant const
```

---

It is important to note here that a compiler may not always translate constants in this manner. Small constants are often not often allocated to memory locations or registers. Instead, the compiler explicitly specifies the value of the constant in instructions that use that constant. Consider the following code snippet.

---

```
#define small 2
main()
{
    var i, j;
    i = small;
    j = small * small;
}
```

---

#### Declaration and Uses of a Small Constant

---

This code can be re-written as follows:

---

```
main()
{
    var i, j;
    i = 2;
    j = 2 * 2;
}
```

---

<sup>9</sup>Certain run-time tables are also allocated in the same section of memory as the constants. Such an example is available in page 122, where we discuss *jump tables* for translating `switch` statements.

---

### Substituting Occurrences of a Small Constant by its Value

---

References to small constants can often be directly specified in an instruction itself as *immediate operands*. This is discussed in Section 3.7.1.

---

**Floating-Point Constants:** Floating-point constants take up many bits even if the value they represent is small. In modern architectures, the minimum number of bits required to represent a floating-point number is 32 bits. Therefore, floating-point constants are seldom explicitly specified within instructions as immediate values. Instead, they are allocated to memory locations just like what is done for large integer constants. Consider, for example, the following C program:

```
#include <stdlib.h>
#include <stdio.h >
int main( int argc, char **argv, char **envp )
{
    static int j;
    static double i = 1.0;
    static double a[8] = {0,1,2,3,4,5,6,7};

    j = 0;
    a[j] = i + 1.0;

}
```

---

### 3.4.2 Translating Variable Declarations

High-level language programs typically have a number of variable declarations, both *global* and *local*. Global variables are visible throughout the program, whereas local variables are visible only when the block in which they are declared are active. Translation of a variable declaration typically involves assigning memory location(s) for the variable; optimizing compilers may allocate some variables to registers in order to speed up the program's execution. The number of locations required for a variable depends on its type. The memory location(s) or register(s) will hold the variable's value (a pattern of 0s and 1s) during the execution of the program. Notice that in machines that support only aligned words, integers and floating-point numbers need to start on a word boundary.

Variables declared in a high-level language are generally allocated locations in one of three sections in memory: the `.data` section, the run-time `stack` section, and the `heap` section. Variables that persist across function invocations, such as *global* variables and *static* variables, are allocated memory locations in the `.data` section. Variables that do not persist across function invocations, such as *local* variables, are generally allocated locations in the `stack` section. Dynamic variables created at execution time and accessed through pointers are allocated memory locations in the `heap` section.

### 3.4.2.1 Global Variables

Consider the following global variable declarations in C.

---

```
int i = 0;
int a = 12;
struct {
    char name[6];
    int length;
} record;
float f = 0.5;
char *cptr;
```

Sample Global Variable Declarations

---

One way of translating these HLL variable declarations to assembly language is given below. Again, we use labels to specify memory addresses that correspond to variables. For clarity, in our examples, the label assigned to the memory location corresponding to an HLL variable generally has the same name as the HLL variable name. The `.data` directive tells that the subsequent items need to be placed in the `.data` section. We have allocated a contiguous space in memory for HLL variables `i`, `a`, `record`, `f`, and `cptr`. Variables `i` and `a` require four bytes each, and are allocated memory using the `.word` directive. The struct variable `record` has 2 fields: `name` and `length`. The field `name`, an array of six characters, requires one byte per character. The field `record.length` is an integer, and therefore starts at the next word boundary. Thus, a total of 12 bytes are allocated for the HLL variable `record`. The HLL variable `f` of type `float` is allocated with the `.float` directive, and occupies 4 bytes in memory. Finally, the HLL pointer variable `cptr` is allocated space in memory with the `.word` directive. Notice that in an assembly language program, the memory location assigned to an HLL variable does not identify the data type. Here, the same directive is used, for instance, for allocating integers as well as pointers.

---

```
# Assign memory locations for the global variables and initialize them
.data                                # Subsequent items are stored in the .data section
i:  .word  0                         # Label next memory location as i;
                                     # allocate next 4 bytes for int variable i
```

```

a:      .word  12          # Label next memory location as a;
                          # allocate next 4 bytes for int variable a

record:          # Label next memory location as record
      .byte  0:6          # Allocate next 6 bytes for record.name
      .word  0          # Allocate next 4 bytes for record.length

f:      .float 0.5        # Label next memory location as f;
                          # allocate next 4 bytes for float variable f

cptr: .word  NULL        # Label next memory location as cptr;
                          # allocate next 4 bytes for pointer variable cptr

```

---

For the HLL variables, we have allocated memory locations in sequential order in the `.data` section, but the exact addresses are not specified. When this program is translated to machine language by the assembler, specific addresses will be assigned. You can verify this by loading the above code in SPIM, and looking at the `DATA` display of the `xspim` window. Figure 3.9 gives one such memory allocation assuming these data items to start at location `0x10010000`. The assembler has allocated a contiguous space in memory for all of the items declared in the `.data` portion of the assembly code given above. Thus, variables `i` and `a`, which require four bytes each, are mapped to locations `0x10010000-0x10010003` and `0x10010004-0x10010007`, respectively. The locations corresponding to struct variable `record` begin at address `0x10010008`, with the first 6 locations corresponding to the 6 characters in `record.name`. Because of the automatic alignment of the `.word` directive on a word boundary, the memory locations corresponding to `record.length` start at the next word boundary, `0x10010010`. Locations `0x1001000e` and `0x1001000f` are therefore unused. Had we placed the `.align 0` directive after the `.data` directive in the above code, then `record.length` would have been mapped to locations `0x1001000e-0x10010011`. However, all accesses to `record.length` then become complicated, as we will see later.

In the above assignment in a machine language, all of the initialized variables (`i`, `a`, and `f`) and the uninitialized variables (`record` and `cptr`) were allocated together in the same section of memory. When a machine language program is shipped, the `.text` section as well as the initialized `.data` section need to be shipped. The assembly language programmer can help to reduce the size of the shipped program by allocating uninitialized data items using the `.comm` or `.lcomm` directives instead of the `.word` directive. The assembler and linker would then allocate memory locations for such data items in a separate data section, called `.bss`. This uninitialized data section need not be included in the shipped program. Notice that the `.comm` directive is not supported by SPIM; so, you cannot try this with SPIM.

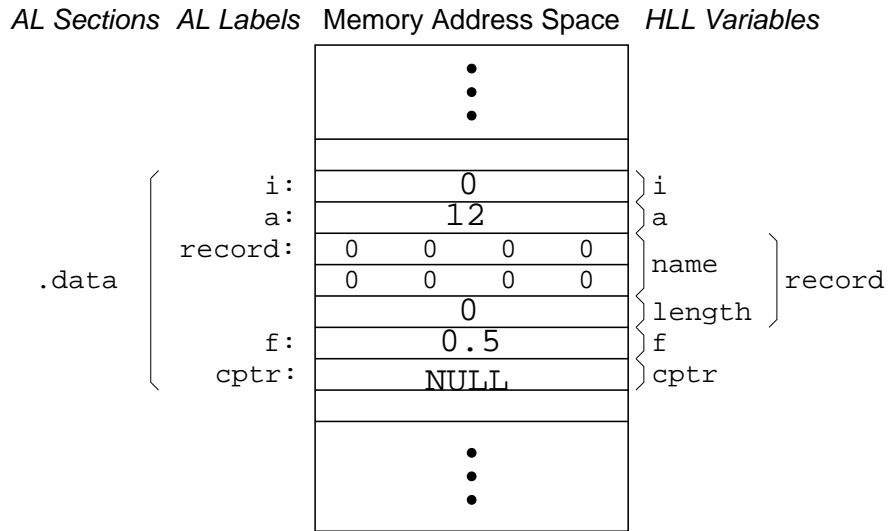


Figure 3.9: A Memory Map View of the Assembly Language Program Snippet Implementing Global Variables

### 3.4.2.2 Local Variables

Next let us turn our attention to local variables (or automatic variables), which are defined inside a subroutine, and are active only when the subroutine is active. Two types of allocation are possible for local variables: static allocation and global allocation.

**Static Allocation:** In this type of allocation, the local variables are assigned memory locations in the `.data` section, just like the globals. This avoids the overhead of creation and destruction of a work space for every subroutine instance. However, this is possible only if the subroutine is non-recursive and non-reentrant, such as in Fortran 77 (an earlier version of Fortran). In these languages only one instance of a given subroutine can be active at a time.

**Dynamic Allocation:** Although we can assign them memory locations along with the globals, such an approach has some drawbacks:

- The local variables of all subroutines will be occupying memory space throughout the entire execution of the program, irrespective of whether they are active or not.
- More importantly, if a subroutine is recursive, then multiple instances of a local variable defined in that subroutine will map to the same memory location. The newer

instances of the subroutine may therefore overwrite the values stored by the earlier, but still active, instances of the subroutine.

In the ensuing discussion, we only consider dynamic allocation. Conceptually, the local variables of a subroutine instance should be “created” only when the subroutine instance comes into existence, and should be “destroyed” when the instance finishes. To do this, the allocation of these variables should happen at *run-time* as opposed to during translation. Thus, when a subroutine calls another, a new set of local variables are created for the callee, although the caller’s local variables are still active. The first subroutine to complete will be the one that is called last, and the local variables to be destroyed first are the ones that were created last in the nested call sequence. That is, the local variables are created and destroyed in a last-in first-out (LIFO) order. This suggests that the local variables could be allocated at run-time in a stack-like structure; the LIFO nature of stack-like structures fits naturally with the LIFO nature of subroutine calls and returns.

A natural place to allocate the local variables, then, is in the **stack** section discussed in Section 3.1.2. A convenient way of doing this allocation is to designate for each active subroutine a contiguous set of locations in the **stack** section called a **stack frame** or **activation record**. The stack frame constitutes a private work space for the subroutine, created when entering the subroutine and freed up when returning from the subroutine. If the subroutine requires more space for its local variables, it can obtain the space it needs by raising the top of stack.

We shall illustrate the use of stack frames using an example code. Consider the following C code. We have included only the `main()` function for simplicity. This function declares two local variables, `x` and `y`.

---

```
int i = 0;
int a = 12;

main()
{
    int x = 5;
    int y;

    ...
}
```

---

#### Example Global Variable and Local Variable Declarations

---

One way of translating these variable declarations to assembly language is given below. A stack frame of 12 bytes is created for the `main` subroutine by the `subu $sp, 12` instruction, which forms the subroutine *prologue code*. The local variables `x` and `y` are allocated 4 bytes

each in the `stack` section. Unlike the global variables, the exact addresses assigned to these local variables are not determined when generating the equivalent machine language program. These addresses will be determined only at run-time, based on where the stack frame for `main()` gets allocated. Figure 3.10 illustrates this memory allocation. Part (a) of the figure shows the memory map prior to entering `main()`, and part (b) shows the same after entering `main()` and executing the `subu $sp, 12` instruction. The newly created stack frame for `main()` is deleted prior to leaving `main()` by executing the `addu $sp, 12` instruction, which forms the subroutine *epilogue code*. You can load this assembly language program in `xspim` and ‘execute’ it to see the allocation of local variables. When running programs having a `main` label, it is better to run `xspim` without the `-notrap` option, allowing SPIM to append a start-up code at the beginning of the program. The start-up code performs some initialization, and then calls the `main` subroutine.

---

```

    # Assign memory locations for the global variables
    # Initialize memory locations if necessary
    .data                # Store subsequent items in the data section
i:    .word    0          # Allocate a 4-byte item with initial value 0
                        # at next memory location, and label it i
a:    .word    12         # Allocate a 4-byte item
                        # at next memory location, and label it a

    .text
    .align 2
    .globl main
    .ent    main          # Entry point of subroutine main (optional)
main:
    # Assign memory locations for the local variables of main()
    subu    $sp, 12        # Decrement $sp to allocate a 12-byte stack frame
    .frame  $sp, 12, $ra   # Stack frame is accessed with $sp; frame size is 12 bytes;
                        # return address is in $ra
    li      $t1, 5
    sw      $t1, 8($sp)    # Initialize local variable x to 5
    ...
    ...
    addu    $sp, 12        # Increment $sp to delete the current stack frame
    jr      $ra            # Return from subroutine main
    .end    main          # End point of subroutine main (optional)

```

---

### 3.4.2.3 Dynamic Variables

A global variable is assigned a fixed memory location (or set of neighboring locations) at compile time, and remains fixed throughout the execution of the program. Such variables are assigned memory locations within the `.data` section. A local variable, on the other hand, begins to exist from the time the corresponding subroutine is called. It is disposed



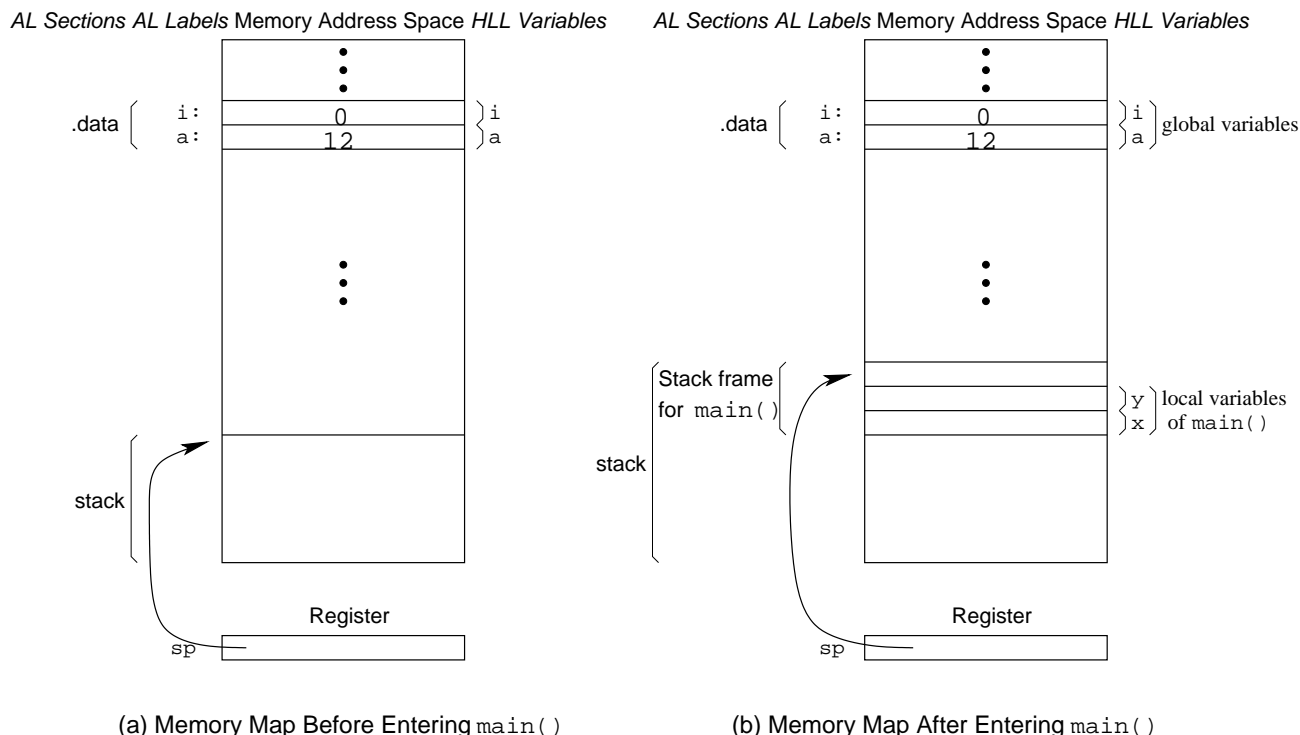


Figure 3.10: A Memory Map View of the Assembly Language Program Snippet Implementing Local Variables

of when control is passed back to the calling routine. A convenient place to allocate such a variable is in the stack frame that will be created when the subroutine is called.

Finally, a dynamic variable (pointed to by a pointer) is created during program execution (by calling a library function such as `malloc()` or `calloc()` in C, which returns the memory address assigned to the variable). Such a variable continues to exist until the allotted memory locations are explicitly freed (by calling a library routine such as `free()`). Like local variables, dynamic variables are also not assigned memory locations at compile time, and are instead assigned memory locations at run-time as and when they are created. Unlike local variables, however, the run-time allocation is done not in the `stack` section, but in the `heap` section. This is because these data items need to be active even after the subroutines in which they were created finish execution.

We shall illustrate the allocation of dynamic variables using an example code. Consider the following C code. We have included only the `main()` function for simplicity. This function declares two local variables, `x` and `y`.

---

```
int i = 0;
```

```

main()
{
    int x, *y;

    y = (int *)malloc(8);
    ...
}

```

---

Example Global Variable, Local Variable, and Dynamic Variable Declarations

---

```

# Assign memory locations for the global variables
# Initialize memory locations if necessary
.data                                # Store subsequent items in the data section
i:  .word  0                        # Allocate a 4-byte item with initial value 0
                                     # at next memory location, and label it i
a:  .word  12                       # Allocate a 4-byte item
                                     # at next memory location, and label it a

.text
.align 2
.globl main
.ent  main                          # Entry point of subroutine main
main:
# Assign memory locations for the local variables of main()
subu  $sp, 12                       # Decrement $sp to allocate a 12-byte stack frame
                                     # for storing $ra and for variables x and y
.frame $sp, 12, $ra                # Stack frame is accessed with $sp; frame size is 12 bytes
                                     # return address is in $ra
sw    $ra, 0($sp)                  # Save register $ra's contents on stack
li    $a0, 8
jal   malloc                       # Call subroutine malloc to allocate a 4-byte item
sw    $v0, 4($sp)                  # Update variable y with the address of dynamic variable
...
...
lw    $ra, 0($sp)                  # Load register $ra with previously saved value
addu  $sp, 12                      # Increment $sp to delete the current stack frame
jr    $ra
.end  main                         # End point of subroutine main

```

---

### 3.4.2.4 Symbol Table

Keeping track of the mapping between the HLL variables and memory locations can be quite tedious for an assembly language programmer, but not for the compiler. Most assembly

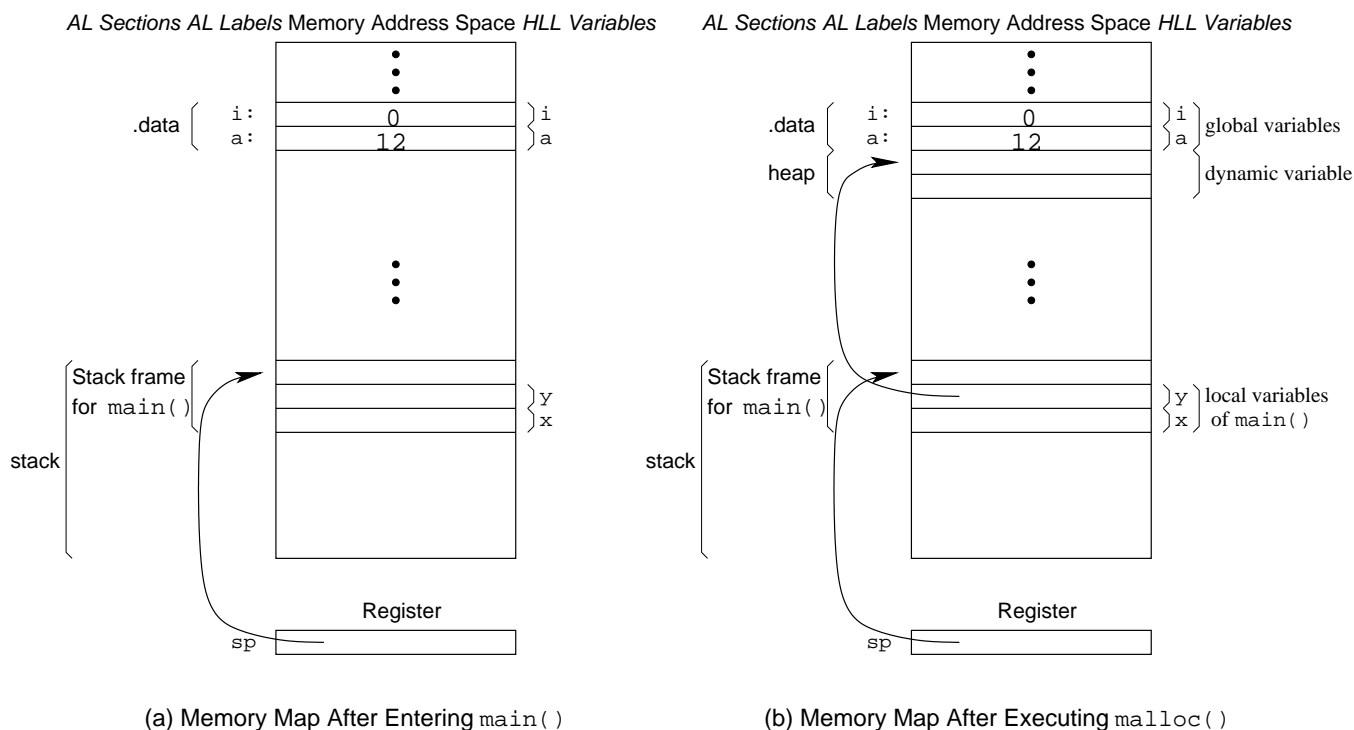


Figure 3.11: A Memory Map View of the Assembly Language Program Snippet Implementing a Dynamic Variable

languages provide the ability to symbolically specify memory locations by labels. Each symbolic name in the assembly language program is eventually replaced by the appropriate memory address during the assembly process. The compiler typically keeps track of variable allocations through a data structure called **symbol table**. Each entry of the symbol table corresponds to one variable, and has enough information for the compiler to remember the memory locations or registers allocated for the variable. As and when the compiler encounters a variable declaration, it creates a new entry for the variable in the symbol table. The symbol table information is useful for displaying the addresses of symbols during debugging or analyzing a program.

### 3.4.3 Translating Variable References

The compiler (or assembly language programmer) needs to generate the proper sequence of data movement operations whenever an HLL variable is referenced. For doing this, it has to remember where in memory (or register) the variable has been allocated. For variables encompassed in complex data structures such as array elements and structure elements, accessing the variable also involves *calculating* the correct address of the variable. Consider

the following assignment statements involving the previously declared global variables.

```
record.length = i;
cptr = &(record.name[3]);
record.name[5] = *cptr;
```

Some machines provide instructions to perform memory-to-memory arithmetic. In such machines we may be able to translate arithmetic expressions without copying the variable values to registers. However, in load-store machines such as the MIPS-I, we must first load the variable values into registers from their corresponding memory locations. Let us translate the above assignment statements to MIPS-I assembly code. Notice that these statements cannot be executed in SPIM without including the appropriate `.data` declarations.

---

|                    |                            |   |
|--------------------|----------------------------|---|
| <code>.text</code> |                            | # Store subsequent items in the text section  |
| <code>lw</code>    | <code>\$t1, i</code>       | # Copy the value in memory location named <code>i</code> into <code>\$t1</code>         |
| <code>la</code>    | <code>\$t2, record</code>  | # Copy the memory address named <code>record</code> into <code>\$t2</code>              |
| <code>sw</code>    | <code>\$t1, 8(\$t2)</code> | # Copy <code>\$t1</code> into memory location allocated to <code>record.length</code>   |
| <code>addu</code>  | <code>\$t3, \$t2, 3</code> | # Calculate address of <code>record.name[3]</code>                                      |
| <code>sw</code>    | <code>\$t3, cptr</code>    | # Store the address into memory location named <code>cptr</code>                        |
| <code>lw</code>    | <code>\$t3, cptr</code>    | # Copy the value in memory location <code>cptr</code> to <code>\$t3</code>              |
| <code>lw</code>    | <code>\$t4, 0(\$t3)</code> | # Copy the value in mem location pointed to by <code>cptr</code> to <code>\$t4</code>   |
| <code>sw</code>    | <code>\$t4, 5(\$t2)</code> | # Store <code>\$t4</code> into memory location allocated to <code>record.name[5]</code> |

---

#### 3.4.4 Translating Conditional Statements

All high-level languages support *if* statements, which cause some statements to be executed only if a condition is satisfied. Consider the following C code:

```
if (i < a)
    cptr = record.name;
else
    cptr = &(record.name[1]);
```

To translate this *if* statement to assembly language, it is easier to first rewrite this code using *goto* statements, because assembly languages usually do not provide *if-else* constructs. The above code can be rewritten as follows:

```
if (i >= a)
    goto else1;
cptr = record.name;
goto done;
else1: cptr = &(record.name[1]);
done: ...
```

We can implement this modified C code in assembly language with the use of *conditional branch* (or conditional jump) instructions, which permit the skipping of one or more instructions. The exact manner in which a conditional branch checks conditions depends on the machine. Some machines provide condition codes that are set by arithmetic instructions and can be tested by conditional branch instructions. Some others like MIPS-I do not provide condition codes, and instead let conditional branches check the value in a register.

A translation for the above C code to MIPS-I assembly code is given below. In this code, the *if* condition evaluation is done using a **bge** instruction. Notice that in the high-level language, when an *if condition* is satisfied, the statement(s) following the **if** statement is (are) executed. In the assembly language, by contrast, when a *branch condition* is satisfied, the instruction(s) following the branch instruction is (are) skipped. Therefore, we need to use the complement of the *if condition* as the branch condition. In this example, the C code checks for the condition **if (i < a)**; the assembly code checks for the branch condition **bge** (branch if greater or equal). Also notice that in the high-level language, once execution goes to the *then* portion, the *else portion* is automatically skipped. However, the assembly language does not provide such a support, and so an unconditional branch instruction is used just before the *else* portion to skip the *else* portion whenever control goes to the *then* portion.

---

```

.text
lw      $t1, i          # Copy the value in memory location i into $t1
lw      $t2, a          # Copy the value in memory location a into $t2
bge     $t1, $t2, else  # Branch to label else if $t1 (i) ≥ $t2 (a)
la      $t1, record     # Copy the memory address named record into $t1
sw      $t1, cptr       # Copy the value in $t1 into memory location named cptr
b       done           # Branch to label done (to skip the else portion)
else: la $t1, record     # Copy the memory address named record into $t1
      addu $t1, 1        # Increment $t1 so as to obtain address of record.name[1]
      sw   $t1, cptr     # Copy the value in $t1 into memory location named cptr
done: ...

```

---

A series of **if-else** statements based on a single variable can be expressed as a multi-way branching statement using the C **switch** statement. Consider the following C code. It contains a **switch** statement, with 5 different cases including the **default** case.

```

switch (record.length)
{
case 0:
case 1:
    *record.name = 'L';
    break;
case 2:
case 3:

```

```

        *record.name = 'M';
        break;
default:
        *record.name = 'H';
    }

```

A trivial way of translating this `switch` statement is to first express it as a series of `if-else` statements, and then translate them as we just did. However, such an approach may be very inefficient, especially if there are many cases to consider. A more efficient translation can be performed by incorporating a software structure called **jump table**. A jump table is an array that stores the starting addresses of the code segments corresponding to the different cases of a `switch` statement. It is indexed by the value of the `switch` statement's control variable.

In the next page we show an assembly language translation of the `switch` statement given above. This code begins with the jump table declaration, which is stored in the read-only data section. The jump table starts at label `JT`, and contains 4 entries, corresponding to values 0-3 for `record.length`. These 4 entries are initialized to 4 labels that are declared in the `.text` section.

The `.text` portion first reads the value of `record.length` from memory. Then the `default` case is handled by checking if the value of this variable is greater than or equal to 4. If this condition is satisfied, then control is transferred to label `default`. If the condition is not satisfied, then we need to index into the jump table. The jump table index is calculated by scaling the value of `record.length` by 4, as each table entry occupies 4 bytes. For instance, if `record.length` has a value of 3, then the `mul` instruction scales it by 4 to obtain an index of 12. The subsequent `lw` instruction adds this offset to the jump table starting address `JT`, and loads the target address into `$t6`. The next instruction performs an unconditional jump to the target address stored in `$t6`.

---

```

        .rdata                # Store subsequent items in the read only data section
#####
JT:
        .word  case0
        .word  case1
        .word  case2
        .word  case3
#####

        .text                # Store subsequent items in the text section
        .align 2             # Align next item on a 22 byte (32-bit word) boundary
        la     $t1, record    # Load starting address of variable record in $t1
        lw     $t6, 8($t1)     # Copy contents of memory location record.length to $t6
        bge    $t6, 4, default # Branch to label default if record.length (in $t6) ≥ 4

```

```

        mul    $t6, $t6, 4    # Scale by 4 to obtain the jump table index
        lw     $t6, JT($t6)   # Copy jump target address from jump table to $t6
        j      $t6

case0:                                # case 0
case1:                                # case 1
        li     $t15, 'L'
        sb     $t15, 0($t1)    # *record.name = 'L'
        b      done           # break

case2:                                # case 2
case3:                                # case 3
        li     $t15, 'M'
        sb     $t15, 0($t1)    # *record.name = 'M'
        b      done           # break

default:                              # default
        li     $t15, 'H'
        sb     $t15, 0($t1)    # *record.name = 'H'

done:

```

---

### 3.4.5 Translating Loops

Loops form a major portion of most programs. Therefore, it is important to translate them in an efficient manner. Consider the following C code that adds the elements of `record.name`, and stores the result in the memory location pointed by variable `cptr`. This code uses the `for` loop construct.

```

*cptr = 0;
for (i = 0; i < record.length; i++)
    *cptr += record.name[i];

```

Most assembly languages do not provide a single instruction that can directly implement complex HLL constructs such as `for` loops. We can, however, rewrite the above loop in terms of an `if` statement with an embedded `goto`, as follows:

```

*cptr = 0;
i = 0;
loop:  if (i < record.length)
    {
        *cptr += record.name[i];
        i++;
    }

```

```

        goto loop;
    }

```

One possible translation for this rewritten loop is given below. In this assembly language program, register `$t2` is used to store the current value of memory word pointed by `cptr`, and register `$t3` is used to store the latest value of `i`.

---

|   |   |
|---|---|
| <code>.text</code>                      | <code># Store subsequent items in the text section</code>                     |
| <code>.align 2</code>                   | <code># Align next item on a 2<sup>2</sup> byte (32-bit word) boundary</code> |
| <code>la \$t1, record</code>            | <code># Load starting address of variable record in \$t1</code>               |
| <code>li \$t2, 0</code>                 | <code># Initialize copy of *cptr in \$t2</code>                               |
| <code>li \$t3, 0</code>                 | <code># Initialize copy of i in \$t3</code>                                   |
| <code>lw \$t4, 8(\$t1)</code>           | <code># Copy contents of memory location record.length to \$t4</code>         |
| <code>loop: bge \$t3, \$t4, done</code> | <code># Branch to label done if i (in \$t3) ≥ copy of record.length</code>    |
| <code>lw \$t5, 0(\$t1)</code>           | <code># Copy record.name[i] (pointed by \$t1) from memory to \$t5</code>      |
| <code>add \$t2, \$t2, \$t5</code>       | <code># Add copy of record.name[i] in \$t5 to running total in \$t2</code>    |
| <code>addu \$t1, 4</code>               | <code># Increment \$t1 to point to next element of record.name[]</code>       |
| <code>add \$t3, 1</code>                | <code># i++; do this on copy of i in \$t3</code>                              |
| <code>b loop</code>                     | <code># Branch to label loop</code>   |
| <code>done: sw \$t3, i</code>           | <code># Store copy of i in \$t3 to memory location named i</code>             |
| <code>lw \$t4, cptr</code>              | <code># Copy pointer value stored in mem addr cptr to \$t4</code>             |
| <code>sw \$t2, 0(\$t4)</code>           | <code># Store calculated sum (\$t2) to memory location pointed by cptr</code> |

---

This loop causes a straight-line sequence of instructions to be executed repeatedly, as many times as needed (`record.length` in this case). The loop starts at location `loop` and ends at the instruction `b loop`. During each pass through this loop, `record.name[i]`'s address is first determined by adding 4 to `$t1`, and then `record.name[i]` is fetched and added to the running total present in `$t2`. Thus, conditional branch instructions are very useful in implementing the `if` statements and loops present in high-level programs.

In the above loop, a form of indirect addressing (indexed addressing, to be precise) is used in the loop to access `record.name[i]`. Indirect addressing permits a different memory address to be specified during each iteration of the loop, by varying the contents of the index register (`$t1` in this case). If an assembly language does not support any form of indirect addressing, then the only way to change the memory address during each iteration is to use *self-modifying code*, i.e., the program considers portions of itself as data, and modifies itself during execution.

### 3.4.6 Translating Subroutine Calls and Returns

We just saw how control flow changes introduced by `if` statements and loops can be implemented in assembly language. Next we discuss a more complicated type of control flow change, the one involving subroutine calls and returns. To implement the special type of branching required to implement subroutines, two basic instructions are used: a *call* instruction that transfers control from the calling program to the subroutine's starting location,



and a *return* instruction that returns control from the subroutine to the place from which it was called, as illustrated in Figure 3.12. In MIPS-I AL, these two instructions are called `jal` and `jr $ra`, respectively. Implementing subroutines in an assembly language is more complicated because of the following reasons:

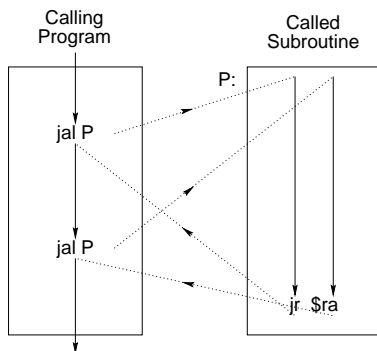


Figure 3.12: Transfer of Control during Subroutine Linkage

- The HLL subroutine may have local variables declared within it, for which storage space needs to be allocated by the assembly language program. Because of the possibility of recursion, each instance of a subroutine requires a new set of storage locations for its local variables.
- Temporary values created within an assembly language subroutine may need to be stored in memory locations and registers. We cannot specify a fixed set of memory locations for a subroutine, again because of recursion. Conceptually, each run-time instance of a subroutine requires a few “fresh” memory locations and registers.
- A subroutine may be called from different places in a program. When the subroutine finishes execution, control must return to the instruction that immediately follows the call instruction that passed control to the subroutine.
- Most subroutines need the calling program to pass parameters to them at the time they are called. Also, often, a subroutine may need to pass a return value to the calling program.

We shall take a detailed look at each of these issues and the solutions developed to handle them. One important aspect that guides these solutions is that the development of an assembly language subroutine—be it by a programmer or a compiler—is often done in isolation to the development of its calling routine(s). This means that the calling routines as well as the subroutines must adhere to a set of well-defined specifications or conventions. If the calling routine was developed with a particular convention in mind, and the subroutine was developed with another, then the program as a whole may not guarantee correct results.

The crux of the first two problems mentioned above is that each invocation of a subroutine needs a separate *working environment*. This environment consists of a set of registers and memory locations that can be used for allocating local variables, and storing temporary values. To solve the last two problems, we need to provide a well-defined communication mechanism between the working environments of the caller and the callee.

We already saw in Section 3.4.2 how local variables of a HLL program are allocated storage space in the corresponding assembly language program. The conventional method is to build a new **stack frame** every time a subroutine is called, and to allocate specific locations within the stack frame for each local variable. The stack frames are created within the **stack** section of the memory address space, and are organized as a LIFO structure. The provision of an independent stack frame for each active subroutine enables each subroutine instance to have its own set of storage locations for allocating its local variables. In the following subsections, we will see how the stack frame concept has become the backbone for solving all of the above mentioned problems associated with implementing subroutines. Frame layouts vary between assembly languages and compilers. If a frame pointer is used, then the frame pointer of the previous frame is stored in the current frame, and restored when returning from a subroutine.

## Return Address Storing

Because a subroutine may be called from different places in a program, provision must be made for returning to the appropriate location in the program after the subroutine completes execution. To do this, the return address must be saved somewhere before transferring control to the subroutine. The MIPS-I assembly-level architecture has earmarked register **\$ra** for storing the return address. The semantics of a call instruction (named **jal** for jump and link) specify that the return address is automatically stored in general-purpose register **\$ra**. If a subroutine needs to call another, then the assembly language programmer (or compiler) writing the caller routine should include instructions to save the contents of **\$ra** on the stack frame (before performing the call) and to restore the contents of **\$ra** (after returning from the callee). Typically, these instructions are placed at the beginning and end, respectively, of the calling subroutine.

Consider the MIPS-I AL program that we saw in page 115. In this program, the routine **main()** calls subroutine **malloc()**. Prior to the **jal malloc** instruction, the return address of **main()** is saved in **main()**'s stack frame using the instruction **sw \$ra, 8(\$sp)**. When executing the **jal malloc** instruction, the contents of **\$ra** will be overwritten with the return address of **malloc()**. Therefore, prior to returning from **main()**, the correct return address of **main()** is restored into **\$ra** using the instruction **lw \$ra, 8(\$sp)**.

## Parameter Passing and Return Value Passing

When calling a subroutine, the calling routine must provide to the callee the parameters, that is, the operands or their addresses, to be used in the computation. Later, the subroutine may return the results of the computation to the calling routine. This exchange of information between a calling routine and a subroutine is referred to as *parameter passing*. The MIPS-I AL convention stipulates the programmer to place the parameters in registers \$a0 through \$a3, where they can be accessed by the instructions of the subroutine. Similarly, the programmer should place the return values in registers \$v0 and \$v1. If more than 4 parameters are present, the rest of the parameters are passed through the stack frame. Figure 3.18 shows a parameter being passed to subroutine P through register \$a0. The return value is passed by the subroutine back to the calling program through register \$v0.

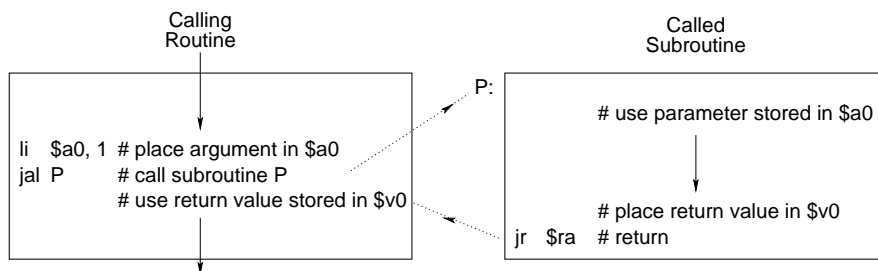


Figure 3.13: Passing Parameters and Return Value through Registers

## Register Saving and Restoring

As discussed, the development of an assembly language subroutine—whether by a programmer or a compiler—is often done in isolation to the development of the calling program. This means that at the time of subroutine development, it is difficult to identify the registers that are not used in the calling environment, and are therefore available for its use. If the subroutine developer blindly uses an arbitrary register for storing a temporary value, there is a possibility of overwriting useful information belonging to the calling environment.

The approach followed in MIPS-I is to let the programmer temporarily save the values pertaining to the caller, prior to the callee using them. Once the callee has completed its usage of a register set, the programmer restores their original values. Again, a convenient place to temporarily save the register values is the stack frame. Figure 3.17 shows the layout of a stack frame in which space has been set apart for saving general-purpose registers as well as floating-point registers. This is the approach followed in the MIPS-I ALA.

Performing the saving (and restoring) of the register values at the proper times is the responsibility of the assembly language programmer. This saving (and restoring) can be done by the caller (*caller save*), by the callee (*callee save*), or by a combination of the two.

It is important to note that it is not necessary to save the entire set of registers in the name space. The MIPS-I AL earmarks some of the registers for caller save, and some others for callee save.

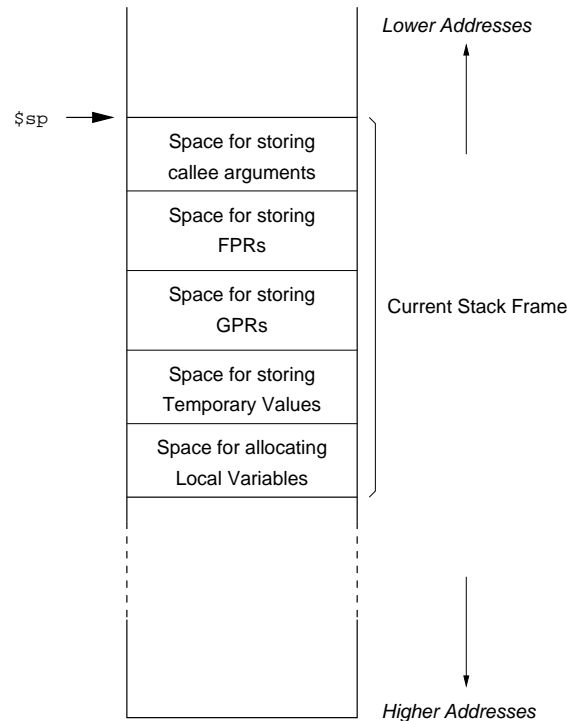


Figure 3.14: Layout of a Typical Stack Frame

Notice that the need for saving and restoring .... or memory locations, the MIPS-I compiler avoids this problem by utilizing a separate stack frame for each active subroutine.

### 3.4.7 Translating System Calls

Consider the C program given in page 53 (and reproduced below), for copying characters from standard input to standard output. This program directly calls the operating system services using the `read()` and `write()` system calls, instead of going through library routines. These system calls can be implemented in an assembly language with the use of trap or syscall instructions. The syscall instruction is quite different from a library function call, although from the assembly language programmer's point of view there may not be a big difference. Consider the following C code which uses the system calls `read()` and `write()` to read a sequence of characters from `stdin` and write them to `stdout`.

```
main()
```

```

{
    char c;

    while (read(0, &c, 1) > 0) /* read one char from stdin */
        write(1, &c, 1);    /* write one char to stdout */
}

```

A MIPS-I AL translation of the above C code is given below. This code uses the `syscall` instruction to request the OS to perform the `read()` and `write()` system calls. Most commercial operating systems use the same set of values for `read_code` and `write_code` (3 and 4, respectively). Notice that this AL program cannot be ‘executed’ on a SPIM simulator, because SPIM does not support these system calls.

---

```

.text
.globl main

main: add    $a1, $sp, 4    # Place the address of c in $a1
      li     $a2, 1        # Place the number of bytes to be read (i.e., 1) in $a2
loop: li     $a0, 0        # Place the file descriptor (0) in $a0
      li     $v0, 3        # Place the code for read() system call in $v0
      syscall                               # Call OS routine to perform the read

      blez   $v0, done     # Break out of while loop if syscall returned zero
      li     $a0, 1        # Place the file descriptor (1) in $a0
      li     $v0, 4        # Place the code for write() system call in $v0
      syscall                               # Call OS routine to perform the write
      b      loop         # Go back to while loop

done: jr     $ra           # Return from main() function

```

---

### 3.4.8 Overview of a Compiler

*[This section needs to be written.]*

The compilation process involves a series of phases.

- Lexical analyzer (or lexer)
- Syntax analyzer (or parser)
- Intermediate code generator
- Code optimizer: Code optimization is an important phase because for most of the applications, once a program is developed, the same program is executed thousands or millions of times.

- Code generator

#### 3.4.8.1 Just-In-Time (JIT) Compiler

The oldest implementation of Java is the interpreter. Every Java command is interpreted to an equivalent sequence of host machine instructions, and is run in the order in which it arrives. This is a really slow way of doing things.

Then came the JIT (just in time) compiler. Every time the Java execution runtime environment would run into a new class—classes are functional groups within the Java program—the JIT compiler would compile it right there. Once something is compiled, it runs with native commands, and it is fast. Spending a little bit of time up front can save a lot of execution time later. That did improve matters, but it still did not get the top performance, because some things that would only run once could take longer to compile than it would take to run them with the interpreter. This means you could wind up with a net loss.

With that observation came the dynamic compiler, which compiles only those things that matter and leaves the rest alone. The dynamic compiler decides whether to compile each class. It has two weapons in its arsenal: an interpreter and a JIT, and it makes an intelligent decision on a class-by-class basis whether to use the one weapon or the other. The dynamic compiler makes that decision by “profiling,” or letting the code run a few internal loops before deciding whether or not to compile that section of code. The decision may be wrong, but statistically the dynamic compiler is right much more often than not; in fact, the longer you run the code, the more likely it is to get it right.

### 3.5 Memory Models: Design Choices

We have seen the basics of translating high-level language programs to assembly language programs. Much of this discussion was centered around the MIPS-I ALA, a somewhat simple architecture. A wide variety of architectures have been in use over the last several decades. They differ in terms of the register model, the memory model, and/or the instruction set. In this section, we shall focus on the memory model. In particular, we shall investigate different options for various facets of the memory model, such as the address space, the endian-ness, and the support for word alignment.

#### 3.5.1 Address Space: Linear vs Segmented

We have already seen that the memory address space is a collection of unique *addresses*, each of which corresponds to a location that can store a single word of information. The address space can be organized in more than one way. The organization we saw so far is the *linear* (one-dimensional array) organization, in which consecutive numbers ranging

from 0 to  $M - 1$  (where  $M$  is the number of memory locations) form the addresses of successive memory locations. The  $M$  addresses ( $0 \rightarrow M - 1$ ) thus constitute a *linear address space* (a.k.a. *flat address space*) for the computer. Thus, the memory model adopted by conventional assembly language (and machine language) architectures is a *linear memory model*. To specify a memory address in this model,  $\log_2 M$  bits are required. For example, if the address space of an assembly-level architecture includes  $2^{32}$  memory locations, then 32 bits are required to specify a memory address in that machine. For this model, we will use the notation  $(X)$  to denote the contents of memory location  $X$ .

In some architectures, this memory is organized in a 2-dimensional manner as a collection of **segments**, each of which is a linear address space for storing logically related words. The segments are typically of different sizes. One segment may hold the program, another may hold the data, and so on. In this memory model, a memory location is referred to by specifying a segment address and a displacement within the segment.

**Expand, saying the adv and disadv of segmented memory model**

### 3.5.2 Word Alignment: Aligned vs Unaligned

Many assembly-level architectures require words to be aligned to their natural boundaries; that is, an  $N$ -bit word may fit within a memory location, but not across two locations. A sketch of the linear memory model, along with an illustration of aligned words and unaligned words is given in Figure 3.15.

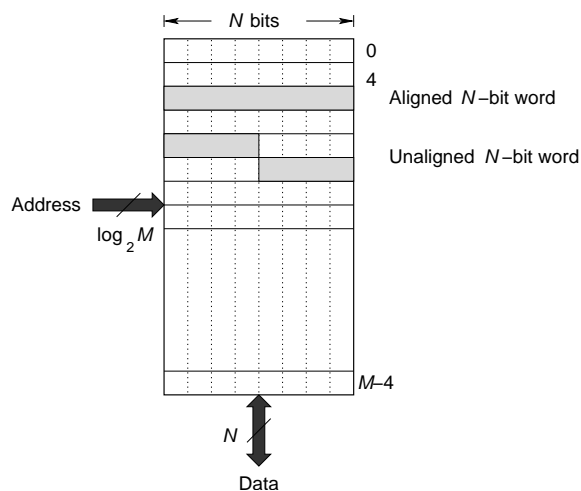


Figure 3.15: Linear Memory Model and Alignment of Words in Memory. Some Assembly-Level Architectures specify words to be aligned in Memory

### 3.5.3 Byte Ordering: Little Endian vs Big Endian

In an assembly-level architecture specifying a byte-addressable memory address space, a word typically spans multiple (adjacent) locations in memory. Each byte within such a word will then have a distinct memory address, although the word is usually referenced by a single address (either the lowest address or the highest address in the range). Two options are available for placing the bytes within the word in memory. In the first option, called *little-endian* organization, the least significant byte of the word is assigned the lowest (byte) address. In the second option, called *big-endian* organization, the most significant byte is assigned the lowest address.

## 3.6 Operand Locations: Design Choices

In this section we will study the different options that exist for holding the operands of an instruction. The MIPS-I architecture limited itself to using the main memory, the general-purpose registers, a few special registers, and even part of the instruction for holding the operands; for data manipulation instructions, operand storage was limited to general-purpose registers. Many other machines specify additional locations for operand storage, such as accumulator and operand stack. We will see each of these options in detail.

### 3.6.1 Instruction

Small constants are frequently used as instruction operands. A convenient place for storing these operands is within the instruction itself. Such operands are called *immediate operands* because they are available for use immediately after the instruction has been fetched from memory. Many small constants such as 0, 1,  $-1$ , 2, and 4 occur frequently in programs, and the immediate operand approach offers a convenient way to store such an operand within the instruction itself. A MIPS-I instruction using an immediate operand is given below.

```
li      $t1, 5      # 5 is an immediate operand
```

### 3.6.2 Main Memory

Main memory is the default location for maintaining the values of the variables declared in a program. For example, when a C compiler encounters the variable declaration

```
int c;
```

it assigns a memory location for variable `c`. The values of the variables are initialized/updated by writing values to the relevant memory locations. Before manipulating a data item, it is often copied to other storage locations such as registers and operand stack.



### 3.6.3 General-Purpose Registers

Besides memory locations, registers are another common place for storing operands. In load-store architectures such as MIPS-I, operands have to be copied to registers before they can be used in arithmetic and logical operations. Registers are also commonly used for storing temporary values and frequently used values. The more the number of registers defined in the architecture, the easier it is to keep the frequently used values in registers.

Modern computers invariably support a number of general-purpose registers at the assembly language and ISA levels—typically 8 to 64—which can be used to temporarily store frequently used operands. When operands are present in registers, it is possible to specify multiple operands explicitly in an instruction, because a register can be specified by just a few bits. Therefore, we can have instructions such as

```
add    BX, CX      # an x86 AL add instruction
add    $t0, $t1, $t2# a MIPS-I AL add instruction
```

Because of these multiple operand or address instructions, machines that use general-purpose registers are often called **multiple address machines**. Among the multiple address machines, some permit data manipulation only on register operands (as well as immediate operands). In such machines, all memory operands must be copied to registers using explicit **LOAD** instructions prior to using them for data manipulation. Similarly, the results of data manipulation instructions are also stored only in registers, from where they are copied to memory locations (if needed) using explicit **STORE** instructions. Machines of this kind are called **load-store architectures** (to reflect the fact that only **LOAD** and **STORE** instructions can access memory) or **register-register architectures** (to reflect the fact that all data manipulation instructions use only register operands (immediate operands as well) and store the result in a register). Examples of load-store architectures are MIPS-I and Alpha.

Machines that permit data manipulation instructions to access register operands as well as memory operands are called **register-memory architectures**. An example is the IA-32 machine. Notice that a register-memory machine can have instructions that use only register operands.

### 3.6.4 Accumulator

The early computers made very judicious use of registers, because registers required a non-trivial amount of hardware. In these machines, one of the registers, called the **Accumulator**, was designated as the one that would be utilized in all arithmetic and logic operations. On machines that operate in this manner, instructions requiring a single operand, such as **COMPLEMENT** and **INCR**, find the operand in the **Accumulator**. The result is also written to the **Accumulator**. Instructions requiring two operands also use the value in the **Accumulator** as one of the operands. The other operand is identified by a single address in

the instruction; hence machines that always use the **Accumulator** are often called **single-address machines**. For example, the single-address instruction

```
add    X;      # Add the contents of mem loc X to ACC
```

means: “Add the contents of memory location **X** to the contents of **accumulator** and place the sum into **accumulator**.” To move the contents of memory location **X** into **accumulator**, we can specify a single address instruction as follows:

```
lw     X;      # Copy the contents of mem loc X to ACC
```

Similarly, to move the contents of **Accumulator** to memory location **Y**, we can specify a single address instruction as follows:

```
sw     Y;      # Copy the contents of ACC to mem loc Y
```

Using only single address instructions, we can add the contents of memory locations **X** and **Y**, and place the result in memory location **Z** by executing the following sequence of instructions:

```
lw     X
add    Y
sw     Z
```

Note that the operand specified in the operand field may be a *source* or a *destination*, depending on the opcode of the instruction. The **lw** instruction explicitly specifies the source operand, memory address **X**, and implicitly specifies the destination, the **Accumulator**. On the other hand, the **sw** instruction explicitly specifies the destination, memory location **Z**, and implicitly specifies the source, the **Accumulator**. A single address machine built in the mid-1960s that enjoyed wide popularity was the PDP-8, made by Digital Equipment Corporation.

### 3.6.5 Operand Stack

The approaches we saw so far use some type of registers—general-purpose registers or an accumulator—in addition to memory locations to hold instruction operands. A radically different approach is to use an *operand stack* to hold the operands. An operand stack is different from the stack frames that we saw earlier. It is a storage structure in which accesses are allowed only to the top location of the stack (sometimes to the topmost two locations)<sup>10</sup>. Only two types of accesses are permitted to the top of the stack: *push* and

---

<sup>10</sup>In computer science and engineering fields, this type of storage mechanisms are also known by the term **last-in first-out (LIFO)**; at any time the first data item that will be taken out of the stack will be the last one that was placed in the stack.

*pop*. These operations are analogous, respectively, to the store and load operations defined on the memory address space. The *push* operation places a new data item to the top of stack, causing the stack to “grow”. The *pop* operation removes the top item from the stack, causing the stack to “shrink”. Thus, the push and pop operations cause a change in the size of the operand stack structure. This is unlike the memory address space, which permits the usual load and store operations, which cause no change to the size of the structure. This peculiar nature of the operand stack can be clarified with a real-world example. Consider a pile of trays in a cafeteria: clean trays are added (pushed) to the pile at the top, causing it to grow; and customers pick up (pop) trays from the top of the pile, causing it to shrink.

Example **push** and **pop** instructions are given below:

|             |          |  |
|-------------|----------|--|
| <b>push</b> | <b>X</b> | # Copy contents of mem loc X to top of operand stack |
| <b>pop</b>  | <b>X</b> | # Copy contents of top of operand stack to mem loc X |

Apart from these *push* and *pop* operations, a machine that supports an operand stack typically provides many arithmetic and logical instructions that operate on operands stored in the operand stack. One such instruction could be

**add**

which means pop the contents of the two topmost locations of the operand stack, add them together, and push the result to the new top of the operand stack. This instruction is interesting in that it does not explicitly specify any operands, but the operands are implied to be on the top of the operand stack. Because only the topmost item(s) of the operand stack are accessible, it is important to push data items to the operand stack in the proper order; otherwise it becomes difficult to use instructions that manipulate data stored in the operand stack.

Only a few architectures support an operand stack. By contrast, almost all architectures support stack frames for allocating the local variables of subroutines. Some of the architectures that support an operand stack define it as a separate address space (as illustrated in Figure 3.16(i)), whereas others define it as part of the memory address space. In the latter case, the operand stack of a subroutine is usually implemented on top of the subroutine’s stack frame, as illustrated in Figure 3.16(ii). The **FP** register points to the bottom of the current stack frame, and the **SP** register points to the top of the operand stack, which is implemented on top of the current stack frame. The Java Virtual Machine (JVM) defines an operand stack in this manner.

A *pure stack machine* takes the operand stack approach to the extreme. It does not define any general-purpose registers, and performs arithmetic and logical operations only with operands present in (the top one or two locations of) the operand stack. It does specify a memory model, along with **push** and **pop** instructions to move data between the operand stack and memory locations. None of the data manipulation instructions in such a machine specifies operands in an explicit manner. Because of these “zero address instructions”, machines that perform data manipulation solely on the operand stack are often called **zero**

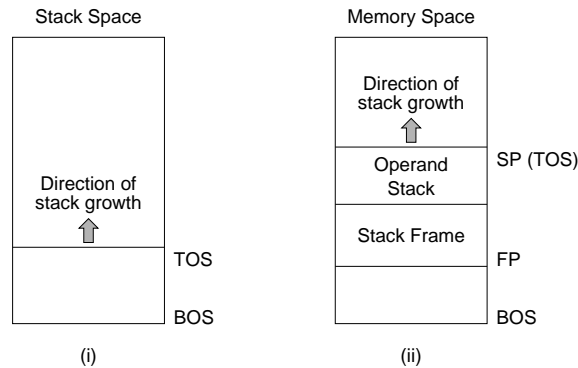


Figure 3.16: Different Ways of Implementing an Operand Stack: (i) Separate Stack Address Space; (ii) Within Memory Address Space, on Top of Current Stack Frame

**address machines.**

*Example:* Write an assembly language program to evaluate the following C language expression in a zero address machine.

$$D = (A + B + C) \times (A + B)$$

Assume that variables  $A$ ,  $B$ ,  $C$ , and  $D$  have been declared as `int`.

```

        .data
A:      .word  5
B:      .word 10
C:      .word  8
D:      .word  0
        .text
push    A      # Copy contents of mem location A to TOS
push    B      # Copy contents of mem location B to TOS
add
push    C      # Copy contents of mem location C to TOS
add
push    A      # Copy contents of mem location A to TOS
push    B      # Copy contents of mem location B to TOS
add
mult
pop     D

```

The pure stack approach does have some drawbacks:

- If the operand stack is implemented in a memory structure outside the processor, all instructions that access the operand stack require off-chip access to fetch/store

operands, and this results in poor performance. The solution often adopted in hardware implementations of stack-based instruction sets is to incorporate the top portion of the operand stack as *microarchitectural registers* inside the processor; the rest of the operand stack is incorporated in memory outside the processor chip. The hardware registers that incorporate the top of the stack can be accessed in a single cycle. As more and more data items are pushed onto the operand stack, these hardware registers get filled up, necessitating the transfer of its bottom portion to the stack in memory. The microarchitecture (and not the assembly language programmer) is responsible for performing this transfer. The assembly language programmer is not even aware of these hardware registers.

- A second drawback of using an operand stack is the inability to reuse temporary values created during computations. Consider the expression that you evaluated just now:

$$F = (A + B + C) \times (A + B)$$

The  $A + B$  portion of this expression needs to be computed only once if the result is stored in a general-purpose register, whereas it needs to be computed twice if operands are stored only on the operand stack, unless the calculated value of  $A + B$  is **popped** into a memory location, and pushed back to the top of stack when needed again.

- Lastly, when all instructions use a common resource such as the top of the operand stack, it becomes difficult to execute multiple instructions in parallel in a high-performance processor implementation.

The proponents of pure stack machines counter these arguments. Their main claims are that the stack machine approach is clean, simple, elegant! These features make it an easy target for compilers. Thus, in the last decade, Sun Microsystems introduced a stack-based machine called Java Virtual Machine (JVM), which has become popular in the world wide web and embedded computing applications.

### 3.7 Operand Addressing Modes: Design Choices

An assembly language instruction specifies the assembly-level machine to do a specific operation on one or more operands. The operands can be present in general-purpose registers, memory, accumulator, stack, or the instruction itself. The exact location of the operands depends on the addressing modes specified in the instruction. An addressing mode specifies a rule for interpreting or modifying an operand address field to obtain the operand. For instance, the operand or its location can be assumed, as in the **CLA** (clear accumulator) instruction, or the operand location can be identified in the instruction itself, as in the “**add \$t1, \$t2, \$t3**” instruction.

We have already used several addressing modes when writing MIPS-I assembly-level programs. Some instruction sets are equipped with even more powerful addressing modes,

which give more flexibility to the assembly language programmer. These addressing modes include capabilities such as pointers to memory, counters for loop control, indexing of data, and relocation of programs. In this section we will examine the common addressing modes for specifying operands in assembly languages. It is important to note that many of these addressing modes are not present in the instruction set architecture. Additional addressing modes are provided in the assembly-level architecture to make it easier to program at the assembly level. Fewer addressing modes are supported by the instruction set architecture so as to make it easier to implement the architecture in hardware. Addressing modes that are unsupported by the instruction set architecture are synthesized using the supported addressing modes.

### 3.7.1 Instruction-Residing Operands: Immediate Operands

We shall start with addressing mode(s) for operands specified within the instruction (immediate operands). In the MIPS-I instruction

```
li      $t1, 5      # 5 is an immediate operand
```

the operand 5 is specified using the immediate addressing mode. Almost all machines provide the immediate addressing mode, for specifying small integer constants. If an instruction set does not support the immediate addressing mode, it would be difficult to specify such constants. One option would be to hardwire a few memory locations or registers with frequently required constants. The rest of the constants will then need to be synthesized from the hardwired constants. Many of the newer machines use such a hardwired register for storing the constant zero, and use the immediate addressing mode for specifying the remaining constants. We have already seen that the MIPS-I architecture has such a register, called `$zero`.

### 3.7.2 Register Operands

In register-based architectures such as MIPS-I, many of the operands reside in registers. The common method for specifying register operands is to employ the **register addressing mode**, which is perhaps the most frequently used addressing mode in a register-based machine. In this addressing mode, the name or address of the register is specified in the instruction. We have already used this addressing mode several times in this chapter. For example, the following MIPS-I instruction uses the register addressing mode to specify 2 operands—a source operand and a destination operand.

```
move    $t1, $t2    # The operand addresses are registers $t1 and $t2
```

In some architectures such as the IA-32, some of the registers are special. Often, when using such as Apart from the register addressing mode,

### 3.7.3 Memory Operands

Memory operands have the largest variety of addressing modes, including ones with indirection. We shall look at the commonly used ones here.

**Memory Direct Addressing:** In this addressing mode, the entire address of the memory operand is given explicitly as part of the instruction. Example instructions that use the memory direct addressing mode to fetch a memory operand are given below:

```
lw      $t1, label1      # Source operand is in memory location label1
lw      $t1, 0x10000000 # Source operand is in memory location 0x10000000
```

In the second instruction, the source operand is present in memory location whose address is 0x10000000. This address is explicitly specified in the instruction. Memory direct addressing has two limitations: (i) At the ISA level, the entire address of the memory operand has to be encoded in the instruction, which makes the instruction long. (ii) The address must be determined and fixed at the time of programming or the assembly process. Once fixed, this address cannot be changed when the program is being run, unless the architecture permits *self-modifying code*<sup>11</sup>. Therefore, memory direct addressing is limited to accessing global variables whose addresses are known at the time of programming or the assembly process.

**Register Indirect Addressing:** In this addressing mode, the instruction specifies a register as in register direct addressing, but the specified register contains the address of the memory location where the operand is present. Thus, the effective address of the memory operand is in the register whose name or number appears in the instruction. This addressing mode is useful for implementing the *pointer* data type of high-level languages. For example,

```
lw      $t1, ($t2)      # Memory address of source operand is in register $t2
```

The big advantage of register indirect addressing is that it can reference memory without paying the price of specifying a full memory address in the instruction. Second, by modifying the contents of the specified register, it is possible to access different memory words on different executions of the instruction. The utility of indirect addressing was demonstrated in the example loop code in Section 3.4.5, which involved finding the sum of the elements of an array. In that program, the `add $t1, 4` instruction (which uses register addressing) causes the address specified by the `lw` instruction (which uses a form of indirect addressing)

---

<sup>11</sup>In an architecture that permits self-modifying code, the program is allowed to use the `.text` section as data as well. The program can thus modify some of its instructions at run time, by writing to the memory locations allotted to those instructions. Self-modifying code was common in the early days of computer programming, but is not in vogue any more because of debugging difficulties.

to point to the next element of the array. What is interesting to note is that the loop body does not explicitly specify any memory addresses.

**Autoincrement Addressing:** This is an extension of register indirect addressing. When this mode is specified, the specified register is incremented by a fixed amount (usually 1), *after* using the current value of the register for determining the effective address of the operand. For example,

```
lw      $t1, ($t2)+    # lw  $t1, ($t2)
                        # addu $t2, 1
```

When this instruction is executed, the current value of `$t2` is used for determining the memory address of the operand, and then `$t2` is incremented. This addressing mode is not specified in the MIPS-I assembly language. It was common in earlier assembly languages, particularly for use inside loops.

**Indexed or Register Relative Addressing:** This is an extension of register indirect addressing. The effective address of the memory operand is given by the sum of an index register value and an *offset* value specified in the instruction. For example,

```
add     $t1, 100($t2) # Mem addr of operand is 100 + contents of $t2
add     $t1, label($t2) # Mem addr of operand is addr of label + contents of $t2
add     $t1, $t3($t2) # Mem addr is contents of $t2 + contents of $t3
```

The first instruction uses a single index register (`$t2`) and an offset, which are added together to obtain the memory address of the operand. Notice that the value of the index register does not change. The second instruction specifies a label as the offset. This mode is particularly useful for accessing arrays; the index register serves as an index into the array starting at `label1`. The third instruction uses two index registers, the contents of which are added together to obtain the memory address of the operand; this indexed addressing mode is not supported in the MIPS-I assembly language. Some assembly languages even permit the index register to be a special register, such as the PC. When register PC is used as an index register, it is often called **PC-relative addressing**. Some machines dedicate one register to function solely as an index register. This register is addressed implicitly when an index mode is specified. In other machines, other special registers or GPRs can be used as an index register. In such a situation, the index register must be explicitly specified in the instruction.

**Memory Indirect Addressing:** In this addressing mode, the instruction specifies the memory address where the effective address of the memory operand is present. This addressing mode is not supported in the MIPS-I assembly language. An example MIPS-like instruction that uses the memory indirect addressing mode is given below.



```
lw      $t1, (label1) # Mem addr of operand is in mem location label1
```

### 3.7.4 Stack Operands

Finally, let us look at stack operands. In the “pure” stack machines that we saw in Section 3.6.5, a stack operand can be accessed only if it is currently at the top of the stack. In such machines, the location of stack operands need not be specified explicitly, and instead can be specified implicitly. This type of addressing is called *implicit addressing mode*. The operand or its effective address is *implicitly* specified by the opcode. An example of such an instruction is given below.

```
add                                # Both operands are on top of stack
```

Most of the register-based machines provide a stack model to the programmer, but use a less strict access mechanism for stack operands. They have a program-managed stack pointer register, which can be used as an index register to access stack operands that are not necessarily at the top of the stack.

| Registers |     | Addr | Main Memory |
|-----------|-----|------|-------------|
| \$t0      |     | 200  | 50          |
| \$t1      |     |      |             |
| \$t2      | 200 | 300  | 500         |
| \$t3      | 200 |      |             |
| \$sp      | 300 | 400  | 1000        |
| pc        | 500 |      |             |
|           |     | 500  | 200         |
|           |     |      |             |

Example: Consider the register map and memory gap given. Each of the memory locations explicitly shown represents 4 bytes. What is the value of the *operand* that is written to **\$t1** in each of the following MIPS-like instructions?

1. `move $t1, $t2`  
This is register direct addressing, and the operand value is 200.
2. `lw $t1, 300`  
This is memory direct addressing, and the operand value is 500.
3. `li $t1, 300`  
This is immediate addressing, and the operand value is 300.

4. `lw $t1, ($t2)`

This is register indirect addressing, and the operand value is the contents of memory location 200, which is 50.

5. `lw $t1, $t2($t3)`

This is register indexed addressing, and the operand value is the contents of memory location 400 (obtained by adding the contents of `$t2` and `$t3`), which is 1000.

6. `lw $t1, (500)`

This is memory indirect addressing; the effective address of the operand is the contents of memory location 500, and the operand value is 50.

## 3.8 Subroutine Implementation

In structured programming languages, subroutines (and macros) are the main mechanism for *control abstraction*, which permits associating a name with a potentially complex code fragment that can be thought in terms of its function rather than its implementation. In Section \*\*\*, we saw how a subroutine is implemented in the MIPS-I assembly language. In this section, we take a broader look at this topic, which is at the core of structured programming. The two things that (i) a subroutine can be called from different places in the program, and (ii) after the completion of the subroutine, control returns to the calling place. For the proper functioning of a subroutine, at the assembly language level, a subroutine requires its own storage space for storing the following: its return address of the subroutine, its local variables, links to variables in non-local scopes<sup>12</sup>, and temporary values it produces. In addition, it may require register space to store frequently used values.

Specifically, we discussed three sub-topics there: return address saving, parameter passing, and saving (and restoring) of registers. In this section, we shall look at some design choices in these areas. Important issues to consider in implementing subroutines in an assembly language are:

- The HLL subroutine may have local variables declared within it, for which storage space needs to be allocated by the assembly language program. Because of the possibility of recursion, each instance of a subroutine requires a new set of storage locations for its local variables.
- Temporary values created within an assembly language subroutine may need to be stored in memory locations and registers. We cannot specify a fixed set of memory locations for a subroutine, again because of recursion. Conceptually, each run-time instance of a subroutine requires a few “fresh” memory locations and registers.

---

<sup>12</sup>Some high-level languages such as

- A subroutine may be called from different places in a program. When the subroutine finishes execution, control must return to the instruction that immediately follows the call instruction that passed control to the subroutine.
- Most subroutines need the calling program to pass parameters to them at the time they are called. Also, often, a subroutine may need to pass a return value to the calling program.

We shall take a detailed look at each of these issues and the solutions developed to handle them. One important aspect that guides these solutions is that the development of an assembly language subroutine—be it by a programmer or a compiler—is often done in isolation to the development of its calling routine(s). This means that the calling routines as well as the subroutines must adhere to a set of well-defined specifications or conventions. If the calling routine was developed with a particular convention in mind, and the subroutine was developed with another, then the program as a whole may not guarantee correct results.

The crux of the first two problems mentioned above is that each invocation of a subroutine needs a separate *working environment*. This environment consists of a set of registers and memory locations that can be used for allocating local variables, and storing temporary values. To solve the last two problems, we need to provide a well-defined communication mechanism between the working environments of the caller and the callee.

We already saw in Section 3.4.2 how local variables of a HLL program are allocated storage space in the corresponding assembly language program. The conventional method is to build a new **stack frame** every time a subroutine is called, and to allocate specific locations within the stack frame for each local variable. The stack frames are created within the **stack** section of the memory address space, and are organized as a LIFO structure. The provision of an independent stack frame for each active subroutine enables each subroutine instance to have its own set of storage locations for allocating its local variables. In the following subsections, we will see how the stack frame concept has become the backbone for solving all of the above mentioned problems associated with implementing subroutines.

### 3.8.1 Register Saving and Restoring

As discussed, the development of an assembly language subroutine—whether by a programmer or a compiler—is often done in isolation to the development of the calling program. This means that at the time of subroutine development, it is difficult to identify the registers that are not used in the calling environment, and are therefore available for its use. If the subroutine developer blindly uses an arbitrary register for storing a temporary value, there is a possibility of overwriting useful information belonging to the calling environment. Notice that a similar problem is avoided for memory values by utilizing a separate stack frame for each active subroutine. One possibility is to provide a similar arrangement for registers. Sun Microsystems' SPARC architecture does precisely that. It uses the concept of multiple **register windows**. Each active subroutine has its own private register name

space called register window. Every time a subroutine is called, a new register window is made available to the newly activated subroutine. When the subroutine finishes execution, its register window ceases to exist.

A more conventional approach for furnishing registers to a subroutine is to let both the caller and the callee use the same register set, but provide a means to temporarily save the values pertaining to the caller, prior to the callee using them. Once the callee has completed its usage of a register set, their original values are restored. Again, a convenient place to temporarily save the register values is the stack frame. Figure 3.17 shows the layout of a stack frame in which space has been set apart for saving general-purpose registers as well as floating-point registers. This is the approach followed in the MIPS-I ALA.

Performing the saving (and restoring) of the register values at the proper times is the responsibility of the assembly language programmer. This saving (and restoring) can be done by the caller (*caller save*), by the callee (*callee save*), or by a combination of the two.

It is important to note that it is not necessary to save the entire set of registers in the name space. Strictly speaking, in order to ensure program correctness, we need to save only those registers that contain *useful* values (such registers are called *live registers*) and are about to be overwritten. However, it is difficult for either the caller or the callee to verify both of these requirements: liveness of a register as well as the definiteness of overwriting it. The caller knows about the liveness of a register, but not its probability to be overwritten by the callee. The callee, on the other hand, knows when the register will be overwritten, but it does not know if the register is live! Thus, whether we use caller save or callee save, some inefficiency is bound to occur. Assembly languages like MIPS-I AL incorporate some conventions about register usage to trim this inefficiency: some of the registers are earmarked for caller save, and some others are earmarked for callee save.

### 3.8.2 Return Address Storing

Because a subroutine may be called from different places in a program, provision must be made for returning to the appropriate location in the program after the subroutine completes execution. To do this, the return address must be saved somewhere before transferring control to the subroutine. This saving can be done either by inserting extra instruction(s) before the call instruction, or by specifying it as part of the call instruction's semantics. Most of the machines follow the latter approach. The way in which a computer supports control flow changes to and from subroutines is referred to as its **subroutine linkage** method.

What would be a good place to store the return address? A commonly used method is to store it in a special register, called a *link register*. This provides a fast mechanism to store and retrieve the return address. However, it does not allow *subroutine nesting*; i.e., one subroutine calling another. When a nested subroutine call is made, the return address of the second call also gets stored in the link register, overwriting its previous contents (the return address of the first call). Hence it is essential to save the link register contents

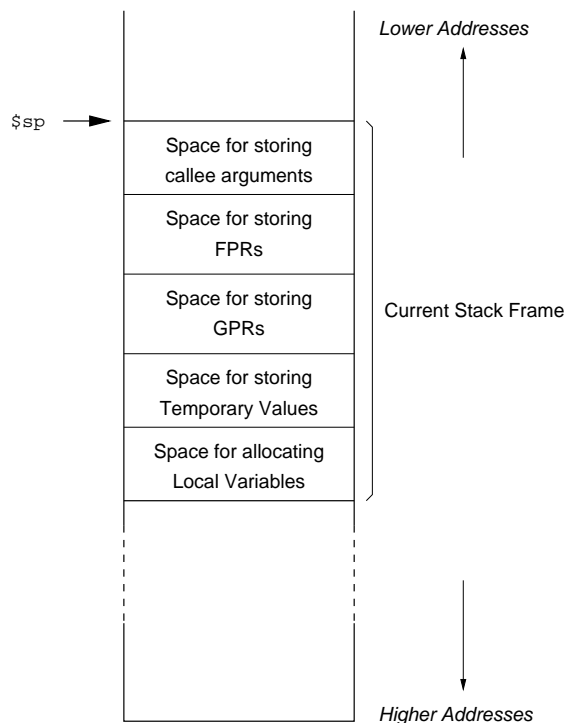


Figure 3.17: Layout of a Typical Stack Frame

somewhere else before calling another subroutine.

Conceptually, subroutine nesting can be carried out to any depth. At any point in time, the first subroutine to complete will be the last one to be called. Its return address is the last one generated by the nested call sequence. That is, the return addresses are generated and used in a last-in first-out (LIFO) order. This suggests that it would be ideal to save the return addresses in a stack-like structure; the LIFO nature of stack pushes and pops fits naturally with the LIFO nature of subroutine calls and returns. Instead of defining a separate stack structure for storing the return addresses, however, we can conveniently store a subroutine's return address in its stack frame itself, as we saw for the MIPS-I assembly language.

If a subroutine needs to call another, then the assembly language programmer (or compiler) writing the caller routine includes instructions to save the contents of `$ra` on the stack frame (before performing the call) and to restore the contents of `$ra` (after returning from the callee). Typically, these instructions are placed at the beginning and end, respectively, of the calling subroutine.

### 3.8.3 Parameter Passing and Return Value Passing

Finally, when calling a subroutine, the calling routine must provide to the callee the parameters, that is, the operands or their addresses, to be used in the computation. Later, the subroutine may return the results of the computation to the calling routine. This exchange of information between a calling routine and a subroutine is referred to as *parameter passing*. Parameter passing may occur in several ways. The parameters may be placed in registers or in the stack, where they can be accessed by the subroutine. Figure 3.18 shows a parameter being passed to subroutine P through register \$a0. The return value is passed by the subroutine back to the calling program through register \$v0.

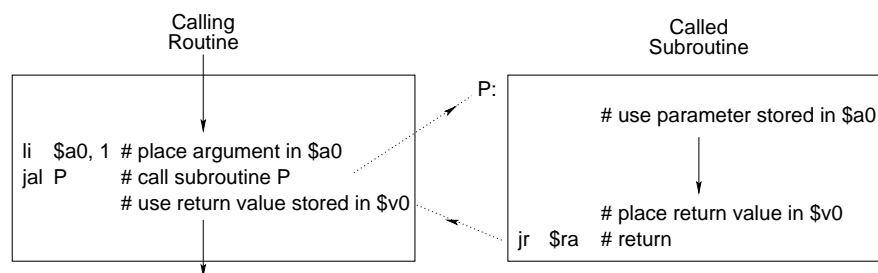


Figure 3.18: Passing Parameters and Return Value through Registers

Many assembly languages have conventions about which registers are used for passing parameters and return values. For instance, the MIPS-I AL, as we saw, designates 4 registers, \$a0-\$a3 for passing parameters and 2 registers, \$v0 and \$v1, for passing return values. When one subroutine wants to call another, it may need to save the incoming parameters (present in registers \$a0-\$a3) in the stack frame, and copy the outgoing parameters (for the callee) onto the same registers. Assembly languages with *register windows* avoid this overhead by overlapping a portion of two adjacent register windows.

Passing parameters through general-purpose registers is straightforward and efficient. However, if many parameters are involved, there may not be enough general-purpose registers available for this purpose. In such a situation, the parameters may be placed on the caller subroutine's stack frame, from where the callee subroutine can access them. This is depicted in the stack frame layout given in Figure 3.17. The stack frame provides a very flexible alternative, because it can handle a large number of parameters. Before calling the subroutine, the calling program copies all of the parameters to the stack frame. The called subroutine can access the parameters from the stack frame. Before returning to the calling program, the return values can also be placed on the stack frame.

## 3.9 Defining Assembly Languages for Programmability

When assembly languages were first introduced, they were very similar to the lower-level machine language (ML) they corresponded to, except that they used alphanumeric symbols instead of binary codes. Thus, instead of coding in a machine language the bit pattern 10001100010000010101010110000010, the programmer could code the same instruction in an assembly language as `lw R1, 0x5582(R2)`. The assembler would translate each AL instruction into precisely one ML instruction. With improvements in assembler technology, this strict correspondence to machine language became relaxed. We now have powerful assembly languages that provide several additional features. We shall discuss some of these features below.

### 3.9.1 Labels

In order to do this, the assembly language programmer has to keep track of the memory locations that correspond to different variables. Keeping track of the memory locations assigned to variables can be quite tedious for an assembly language programmer. Most assembly languages therefore provide the ability to symbolically specify the memory location corresponding to an HLL variable. Each symbolic name in the assembly language program is eventually replaced by the appropriate memory address during the assembly process.

### 3.9.2 Pseudoinstructions

Pseudoinstructions are instructions that are present in the assembly-level architecture, but not in the instruction set architecture. Such instructions are not implemented in the hardware, but are synthesized by the assembler using sequences of instructions present in the machine language. Modern assembly languages often support different types of pseudoinstructions. For example, the assembly-level architecture may support addressing modes that are not really present at the ISA level. By extending the instruction set in this manner, the assembly language makes it easier to program at the assembly level, without adding any complexity to the hardware. During assembly, each pseudoinstruction is synthesized using one or more ML instructions.

### 3.9.3 Macros

Macros go one step beyond pseudoinstructions by allowing the programmer to define (or use predefined) parameterized sequences of instructions that will be expanded during assembly. Macros are very helpful in reducing the source code size as well as in making it more readable, without incurring the overhead of subroutine calls and returns. It is often the case that a sequence of instructions is repeated several times within a program. An example of a sequence of instructions might be the operation that pushes data onto the stack, or

the operation that pops data off the stack. A mechanism that lets the assembly language programmer define sequences of instructions, and associate these instructions with a key word or phrase is called a macro. Macros allow the assembly language programmer to define a level of abstraction.

Simple text-substitution macros can be easily incorporated into an assembly language by using the C language's `#define` construct. The assembler can invoke the C preprocessor `cpp` to do the required text substitution prior to carrying out the assembly process. An example for such a macro definition and use is given below.

```
#define LEAF(fname) \  
    .text; \  
    .globl  fname; \  
    .ent    fname; \  
fname:\  
  
LEAF(foo)
```

## 3.10 Concluding Remarks

### 3.11 Exercises

1. Explain the major differences between high-level languages and assembly languages.
2. Explain why local variables (the ones declared inside subroutines) are typically assigned memory locations in the *stack*, and not in the `.data` section of memory.
3. Explain why the memory direct addressing mode cannot be used for accessing data from the `stack` and `heap` sections of memory.
4. An assembly-level architecture defines directives, non-branch instructions, branch instructions, subroutine calls and returns, registers, memory address space, macros, operand stack, and AL-specific instructions. Which of these are non-essential from a strictly functional point of view? Explain.
5. Explain why, during compilation, dynamically allocated variables of a HLL program are typically assigned memory locations in the *heap* section of memory and not in the *stack*, even if the dynamic allocation takes place inside a subroutine?
6. Consider the following variable declaration and assignment involving pointers in C. Translate this C code into MIPS-I assembly language code.



```
int **ppn, n;
```

```
*ppn = &n;
```

Example Assignment Statement Involving Pointers

7. Consider the following C code snippet:

```
.data
i:    .word 24
j:    .word 22
k:    .word 0

.text
__start:la    $t0, i
        lw     $a0, 0($t0)
        jal    foo
        sw     $v0, k

foo:    lw     $t1, j
        li     $v0, 0
loop:   addi   $v0, $v0, 2
        addi   $t1, $t1, 1
        bne    $t1, $a0, loop
        jr     $ra

        li     $v0, 10          # Code for exit system call
        syscall                # Call OS to exit
```

(a) Trace the execution of this MIPS-I program for 12 instructions. Tracing of an instruction involves showing what value that instruction writes to a register or memory location. The 12 instructions that you trace must be written in the order in which they are executed.

(b) How many memory data references will be made during the execution of these 12 instructions? That is, how many values will be transferred from the memory to the processor?

(c) (2 points) Will this program exit by calling the OS, or will it stay in an infinite loop? Explain.

## Chapter 4

# Assembly-Level Architecture — Kernel Mode

*Give instruction to a wise man, and he will be still wiser;  
Teach a just man, and he will increase in learning.*

**Proverbs 9: 9**

The previous chapter discussed at length the assembly-level architecture machine seen by application programmers. We also saw the basics of translating high-level language application programs to equivalent assembly language programs. As discussed in the last part of chapter 2, high-level languages provide application programmers with an application programming interface (API) for specifying system-specific functions such as input/output and memory management. The API is implemented by the operating system kernel that resides in computer systems. When an application program invokes one of the system call functions specified in the API, the control of the computer system is transferred from the application program to the operating system (OS), which performs the function, and transfers control back to the application program.

In order to perform the system functions in an adequate and efficient manner, the machine needs to include special resources to support the OS. Such resources are generally restricted to the OS, and typically include a few registers, a notable portion of the memory address space, a few instructions, and direct access to all of the IO device controllers. Application programs are not allowed to access these restricted resources. In order to enforce the distinction between application programs (which can only access limited resources) and OS programs (which can access all resources), computers can operate in at least two different execution modes—the *User mode* and the *Kernel mode*, which is also called *Supervisor mode* or *Privileged mode*; in this book we use the term Kernel mode. The Kernel mode is

intended to execute instructions belonging to the OS, and the User mode is intended to execute instructions belonging to application programs.

*“Every mode of life has its conveniences.”*  
— Samuel Johnson. *The Idler*

It is important to note that many of the routines in the OS kernel are executed on behalf of user programs. For example, the shell program executes in the User mode and invokes a system call (syscall) instruction to obtain the characters entered by the computer user on the terminal keyboard. This syscall instruction is implemented by the OS kernel, which executes in the Kernel mode on behalf of the shell program, reads the characters typed on the keyboard, and returns the characters to the shell. The shell then executes in User mode, interprets the character stream typed by the user, and performs the set of actions specified by the user, which might involve invoking other syscall instructions.

Thus, Computer operating systems are another classic example of event-driven programs on at least two levels. At the lowest level, interrupt handlers act as direct event handlers for hardware events, with the CPU hardware performing the role of the dispatcher. Operating systems also typically act as dispatchers for software processes, passing data and software interrupts to user processes that in many cases are programmed as event handlers themselves.

The discussion so far might suggest that systems spend very little time in the kernel mode, and that most of the time is spent in the user mode. The truth is just the opposite! Many embedded systems never leave the kernel mode. A significant amount of code is therefore developed for the kernel mode.

## 4.1 Overview of Kernel Mode Assembly-Level Architecture

In this chapter, we will study the Kernel mode aspects of the assembly-level architecture of modern computers. The Kernel mode part is similar in many ways to the User mode part that we saw in detail in Chapter 3. In particular, the register model, the memory model, the data types, and the instruction types available to the Kernel mode assembly language programmer are all quite similar to those available to the User mode assembly language programmer. Therefore, it is more instructive to consider the differences between the two modes, the details of which, unfortunately, vary somewhat from one machine to another. The main differences are given below:

- In addition to the register set available in the User mode, an additional set of registers called *privileged registers* is available in the Kernel mode. One such register is the **processor status register**, for instance. The register set available in the Kernel mode is a superset of what is available in the User mode.
- Like the case with the register set, an extended memory address space is usually

available in the Kernel mode. The overall Kernel mode memory address space may be divided between addresses that are accessible only in the Kernel mode and addresses that are accessible in both modes. User mode programs can access only User mode addresses. Kernel mode programs can access both Kernel mode and User mode addresses. For instance, in the MIPS-I assembly-level architecture, memory addresses from 0x80000000 to 0xffffffff can be accessed only by the OS.

- In the Kernel mode, the assembly language programmer is provided a set of *IO ports*. These ports are either provided as a set of *IO registers* or as part of the privileged memory address space.
- In the Kernel mode, the program has access to special hardware structures for performing resource management functions. One such hardware structure is TLB (Translation Lookaside Buffer), which is used to implement the *virtual memory* concept.
- In addition to the instruction set available in the User mode, an additional set of instructions called *privileged instructions* is available in the Kernel mode. The privileged instructions cannot be executed while in User mode. For example, a machine may have an instruction that manipulates the processor status register. If a User program uses this instruction, the computer will not execute it, and instead will signal an error.

#### 4.1.1 Privileged Registers

In addition to the register set available in the User mode, an additional set of registers called *privileged registers* is available in the Kernel mode. Table 4.1 lists the privileged registers defined in the MIPS-I kernel mode architecture, along with their names and uses. The first eight registers in the table are used for implementing memory management, and are explained in Chapter 7. The next 3 privileged registers—**sr**, **cause**, and **epc**—are used for processor management. **sr** contains bits that specify the current operating mode and conditions of the processor, such as the current interrupt priority level. **epc** is used to store the memory address of the interrupted instruction, and is useful for returning to the interrupted program after handling the interrupt/exception. Its contents can be copied to a general-purpose register **rt** by executing the privileged instruction “**mfc0 rt, epc**”. Finally, the **PRId** register is used for storing the processor’s generic type number.

Some architectures include a privileged register to point to the current process’ *process control block*, the block of memory in the privileged address space where the OS stores information about the process. Some architectures provide a *page table pointer* for speeding up translations of virtual memory addresses to physical memory addresses. In machines using vectored interrupts, an *interrupt vector register* may be provided.

| Register Number | Register Name | Use   |
|-----------------|---------------|---|
| 0               | Index         |   |
| 1               | Random        |   |
| 2               | EntryLo       |   |
| 4               | Context       |   |
| 5               | PageMask      |   |
| 6               | Wired         |   |
| 8               | BadVaddr      |   |
| 10              | EntryHi       |   |
| 12              | SR            | Status register   |
| 13              | Cause         | Store the cause of the most recent exceptional event              |
| 14              | EPC           | Exception PC; store PC value of interrupted instruction           |
| 15              | PRId          | Processor ID register; store this processor's generic type number |

Table 4.1: Names and Uses of MIPS-I Privileged Registers

### 4.1.2 Privileged Memory Address Space

### 4.1.3 IO Addresses

The IO models supported by the User mode assembly-level architecture and the Kernel mode assembly-level architecture are quite different. As discussed in Chapters 2 and 3, the IO model presented to application program developers is at the level of *files*, and is somewhat abstract. Any operation on a file is accomplished by calling the operating system. The IO model presented to operating system developers is more concrete, and involves a collection of *IO addresses*, which are accessed by IO instructions. In other words, the IO primitives provided by the kernel mode machine consist of an IO address space and a set of (privileged) IO instructions. The exact nature of the IO address space depends on the type of IO addressing used, and is discussed in Section 4.3.

### 4.1.4 Privileged Instructions

The Kernel mode architecture includes additional address spaces and registers, as we just saw. In order to provide exclusive access to these, an additional set of instructions are also included in the Kernel mode ISA. These instructions are called *privileged instructions*. Examples include instructions to access IO addresses, instructions to manage memory, and instructions to manage processes. The Kernel mode instruction set is thus a superset of the User mode instruction set, as pictorially depicted in Figure 4.1. The User mode instruction set contains a set of syscall instructions, as well as non-syscall instructions for performing

data transfer operations between registers and memory, arithmetic/logic operations, and control flow change operations. At the microarchitecture level (which is two levels below the assembly level), a non-syscall instruction is directly interpreted for execution, whereas a syscall instruction is interpreted by invoking a predefined OS service. That is, a syscall instruction is interpreted by executing a sequence of instructions in the Kernel mode (some of which will be privileged instructions), which are then directly interpreted for execution in the underlying kernel mode microarchitecture.

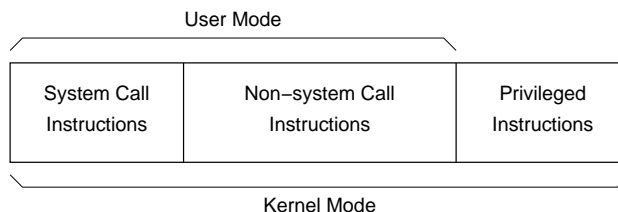


Figure 4.1: Relation between the User Mode and Kernel Mode Instruction Sets

The privileged instructions—which are not available in the User mode—consist of IO instructions, inter-process synchronization instructions, memory management instructions, and instructions to enable and disable interrupts. IO instructions include instructions that read from or write to IO registers. The reason for keeping the IO instructions privileged is straightforward: if an application program is permitted to execute a privileged instruction, then it could read confidential data stored anywhere in the system, write on other users' data, erase all of the information on a disk, and, in general, become a threat to the security of the system itself. So, what will happen if a programmer includes a privileged instruction in an application program? When the program is being executed, an attempt to execute that instruction will generate an *exception*<sup>1</sup>, which causes control to be transferred to the OS. The OS will most likely terminate that application program.

## 4.2 Switching from User Mode to Kernel Mode

We can think of three events that cause the execution mode to switch from User mode to Kernel mode, causing control to transfer from user code to kernel code. They are: syscall instructions, device interrupts, and exceptions. These three events are illustrated in Figure 4.2, and are discussed in detail in this section. When any of these events happen, the kernel gets the control and performs the required action. Among these three events, only the action to be done for syscall instructions is defined in the API (Application Programming Interface).

The first thing the kernel does after getting the control is to disable all interrupts (i.e., set the processor state to not accept any more interrupts), and save the essential state of

<sup>1</sup>Most assemblers will flag this as an error during the assembly process.

the interrupted process on the kernel stack<sup>2</sup>. This state includes the contents of the process' general-purpose registers, program counter, and process status word. Afterwards, the kernel determines the cause of the interrupt, and calls the appropriate low-level handler routine by looking up a *dispatch table* containing the addresses of these routines. This low-level routine performs the functions for which the OS was specifically called at that time. When this low-level routine completes, the kernel restores the state of the process, and sets the execution mode back to the previous value.

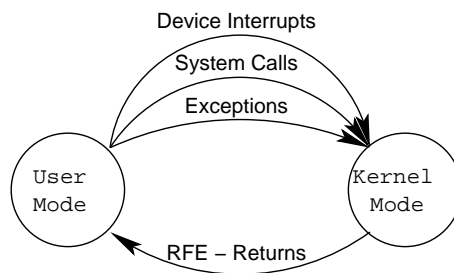


Figure 4.2: Events Causing Switching Between User Mode and Kernel Mode

#### 4.2.1 Syscall Instructions: Switching Initiated by User Programs

When an application program is being executed, the program can voluntarily hand over the machine's control to the OS by executing a syscall instruction (sometimes called *software interrupt* or *programmed interrupt*). Execution of a syscall instruction is a *synchronous* event, because it occurs at the same point of execution when a program is run multiple times. To the application programmer, a syscall instruction seems very much like a function call; however, this control flow change causes the processor to switch to the Kernel mode and to begin executing kernel code. The exact semantics of each system call are defined in the API, and can be different, at least theoretically, in different APIs. For producing portable code, however, the semantics of each system call are kept more or less the same across APIs. To be on the safe side, it is prudent for application programs not to directly call the OS, but instead call an appropriate library routine. When porting to a platform with a different API, all that is required then is to use a different set of library routines that suit the new API.

When a syscall instruction is executed, the computer temporarily stops execution of the current program, switches to Kernel mode, and transfers control to a special OS routine called **system call layer**. Figure 4.3 illustrates how transfer of control takes place when a syscall instruction is executed. As shown, the syscall instruction is treated very similar

<sup>2</sup>Some operating systems save the state in the interrupted process' user stack. Some others save the state in a global *interrupt stack* that stores the frames for those interrupt handlers that are guaranteed to return without switching context.

to a `jal S` (jump and link) instruction, where `S` is the address of the first instruction of the system call layer routine. Thereafter the machine executes the instructions of this part of the OS. After executing this routine, the `eret` instruction at the end of the routine causes control to return to the instruction that immediately follows the `syscall` instruction in the application program. The system call layer routine is very much like a subroutine; an important difference, however, is that it executes in Kernel mode.

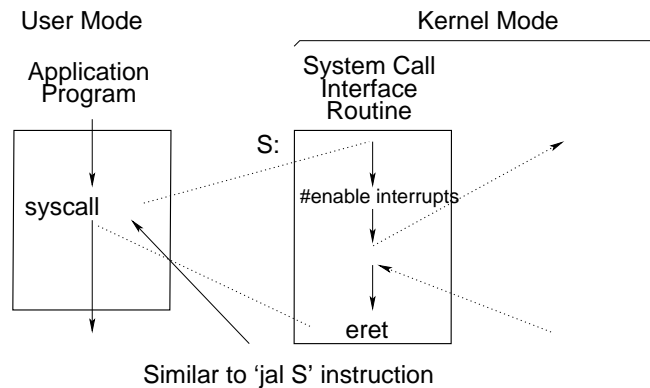


Figure 4.3: Transfer of Control while Executing a System Call Instruction

It is important to see the major actions specified by a `syscall` instruction. These functions are described below:

- Switch to kernel mode
- Disable interrupts: One of the first actions to be performed by the system call layer when control transfers to it is to save the current register state and perform other book-keeping functions. If another interrupt is accepted during this period, there is a potential to lose useful data. Therefore, it is prudent to temporarily disable interrupts. After performing the book-keeping functions, the handler may enable interrupts of higher priority.
- Save return address: The `syscall` instruction is similar to a subroutine call in many ways. One of the striking similarities is in the manner of control flow return. When control returns to the program that contains the `syscall` instruction, execution continues from the next instruction onwards. In order to effect such a control flow transfer, the return address (the address of the instruction immediately after the `syscall` instruction in the static program) needs to be recorded. The MIPS-I architecture, for instance, specifies a privileged register called `epc` (exception program counter) for storing this return address.
- Record the cause for this exceptional event: Once the `syscall` instruction is executed and control is transferred to a handler, the system has no way of remembering the



reason for activating the handler. This is especially the case if multiple exceptional events transfer control to the same entry point, i.e., the same handler. For instance, the MIPS-I architecture specifies the same entry point (0x80000080) for all but two of the exceptional events. To identify the reason for transferring control to this memory location, the MIPS-I architecture provides a privileged register called **cause**.

- Update **pc** to point to the entry point associated with syscall instructions. For the MIPS-I architecture, this entry point is 0x80000080.

### 4.2.2 Device Interrupts: Switching Initiated by IO Interfaces

The syscall instruction is useful when an application program needs some service from the OS. Sometimes, the currently executing application program may not need any service from the OS, but an IO device needs attention, requiring the OS to be executed. This requirement has led to the provision of *device interrupts* (also called *hardware interrupts*) by which an IO device can notify the computer when it requires attention. Unlike a system call, a device interrupt is an *asynchronous* event, because it may not occur at the same point of execution when a program is run multiple times.

In order to run the OS, the currently running program has to be temporarily stopped. Therefore, when an interrupt is received, the computer temporarily stops execution of the current program and transfers control to a special OS routine called **interrupt service routine (ISR)** or **interrupt handler**. If the machine was in the User mode at the time of the interrupt, then it is switched to the Kernel mode, giving the operating system privileged access to the machine's resources. An ISR is very much like the subroutines that we saw earlier; an important difference, however, is that a subroutine performs a function required by the program from which it is called, whereas the ISR may have nothing in common with the program being executed at the time the interrupt request is received. The exact manner in which the interrupting device is identified and the appropriate ISR is called varies from one machine to another.

Figure 4.4 illustrates one way of transferring control to the ISR when an interrupt is raised. Assume that an interrupt request arrives during the execution of instruction  $i$ . The computer first completes the execution of  $i$ . Then it transfers control to an OS routine called *interrupt handler interface*. This routine identifies the interrupting device, and determines the starting address of the appropriate ISR, namely P in the figure. It then transfers control to the ISR by means of a CALL instruction. Thereafter the machine executes the instructions of the ISR. After executing the ISR, control is returned to the interrupt handler interface. The interface routine is terminated by an ERET instruction, which causes control to return to instruction  $i + 1$  in the interrupted program. Along with this, the computer also switches back to the User mode.

Because an interrupt is an unscheduled event, the ISR must save and restore any registers that it modifies. A convenient place to save the registers is the kernel stack.

interrupt  
service  
routine,  
interrupt  
handler

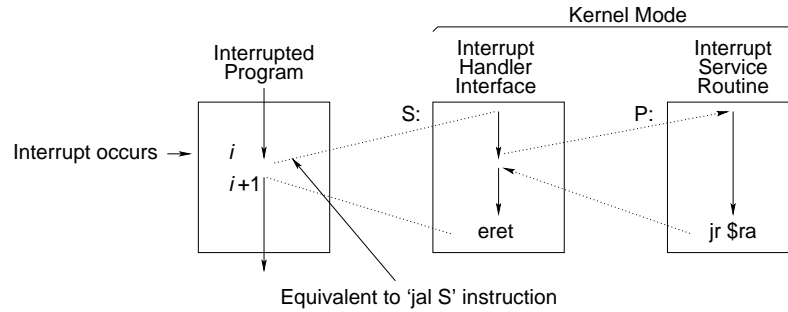


Figure 4.4: Transfer of Control while Servicing an Interrupt

In a multi-tasking environment, whenever the machine switches to the Kernel mode, handing control over to the OS, it also performs *process scheduling*. That is, the OS decides which application process should run next. For the OS to do this process scheduling, it must run periodically. However, extended periods of time may elapse with the computer being in the User mode and no syscall instructions or device interrupts. In order to perform process scheduling in an adequate manner, the OS needs to gain control of the machine on a periodic basis. Multi-tasking computers typically implement this by including as an IO device a hardware *timer*, which issues a hardware interrupt at regular intervals fixed by the OS.

### 4.2.3 Exceptions: Switching Initiated by Rare Events

*“The young man knows the rules, but the old man knows the exceptions”.*  
 — Oliver Wendell Holmes, Sr (American Physician, Poet, Writer, Humorist and Professor at Harvard, 1809-1894) in *The Young Practitioner*

An exception is an unexpected event generated from the program being executed. Examples are attempt to execute an undefined instruction, arithmetic overflow, and divide by zero. When an exception occurs, the machine switches to Kernel mode and generates an *exception vector* depending on the type of exception. The exception vector indicates the memory address from which the machine should start execution (in Kernel mode) after it detected the exceptional event. In other words, after an exceptional event occurs, the machine starts executing in Kernel mode from the address specified in the exception vector. The machine may also record the cause of the exception, usually in a privileged register. The rest of exception handling is similar to that of interrupt handling. That is, the return address is saved, and control is transferred to an exception handler routine in the OS.

The exception handler routine at the exception vector performs the appropriate actions. If that vector is used for different exception types, then the routine inspects the recorded cause of the exception, and other relevant state information, and branches to an appropri-

ate exception handler routine to handle the exception. After taking the necessary steps, control may be returned to the application program that caused the exception, switching the mode back to the User mode. Sometimes, exception handling may involve terminating the application program that generated the exception, in which case the OS gives control to another application program.

## 4.3 IO Registers

We saw in Section 4.1 that the IO model supported in the kernel mode consists of a set of IO registers and a set of instructions to access them. The nature of these IO registers vary considerably, depending on the addressing method used. Two types of addressing methods are used for IO registers: memory mapped IO and independent IO (or IO mapped IO). We shall discuss these two schemes in detail.

### 4.3.1 Memory Mapped IO Address Space

In memory mapped IO, each IO address refers to an *IO register*, an entity similar to a memory location that can be read/written. Because of this similarity, the IO registers are assigned locations within the memory address space itself. Thus, there is a single address space for both memory locations and IO registers. Some portions of the memory address space are assigned to IO registers; loads and stores to those addresses are interpreted as reads and writes to IO registers. The main advantage with this approach is that no extensions are required to the instruction set to support IO operations<sup>3</sup>. Another advantage is that it allows for wider compatibility among IO specifications across different computer families. The MIPS-I architecture uses this approach; memory addresses from 0xa0000000 to 0xbfffffff are available to the OS for use as IO registers. An example IO read instruction for the MIPS-I architecture is

```
lw    $t1, keyboard_status
```

which copies the contents of IO register labeled `keyboard_status` to general-purpose register `$t1`. At execution time, this label will refer to an address in the range 0xa0000000 - 0xbfffffff. This mapping is done either at assembly time by the assembler or at IO port configuration time by the OS.

### 4.3.2 Independent IO Address Space

In this type of IO addressing, a separate IO address space is provided independent of the memory address space. Thus, like the register space and the memory address space, there is an IO address space also. When accessing an address in the IO address space, an IO

---

<sup>3</sup>At the microarchitectural level, the hardware has to distinguish IO operations from memory operations, and treat them accordingly.

address can be specified either by a new addressing mode, or by a new set of opcodes. It is customary to use the latter approach — providing a separate set of opcodes specifically for manipulating IO addresses. An example IO read instruction for a MIPS-I-like architecture would be

```
in    $t1, keyboard_status
```

What do we gain by providing a separate address space for the IO (and a separate set of opcodes as well)? On first glance, there seems to be no apparent gain. Now consider this scenario. When we have a separate IO address space, instead of organizing it as a linear address space, we have the flexibility of organizing it as a set of *IO ports* or *IO programming interfaces*, entities that are more complex than IO registers and memory locations. That is, each IO address refers to an IO port. Several instruction opcodes can be provided to perform complex operations on an IO port. An example IO instruction that tests the status of a port used for connecting a keyboard is

```
test  keyboard_port
```

Examples of machines with independent IO are the Intel x86 and the IBM 370 computers. Figure 4.5 illustrates the two types of IO address mapping. It is important to note that the two types of IO addressing are not mutually exclusive. In a machine that supports independent IO, some of the memory addresses can still be mapped to IO registers. For instance, a graphics display port is usually memory-mapped, to permit device drivers to easily modify bit patterns in memory, which are then displayed on the screen.

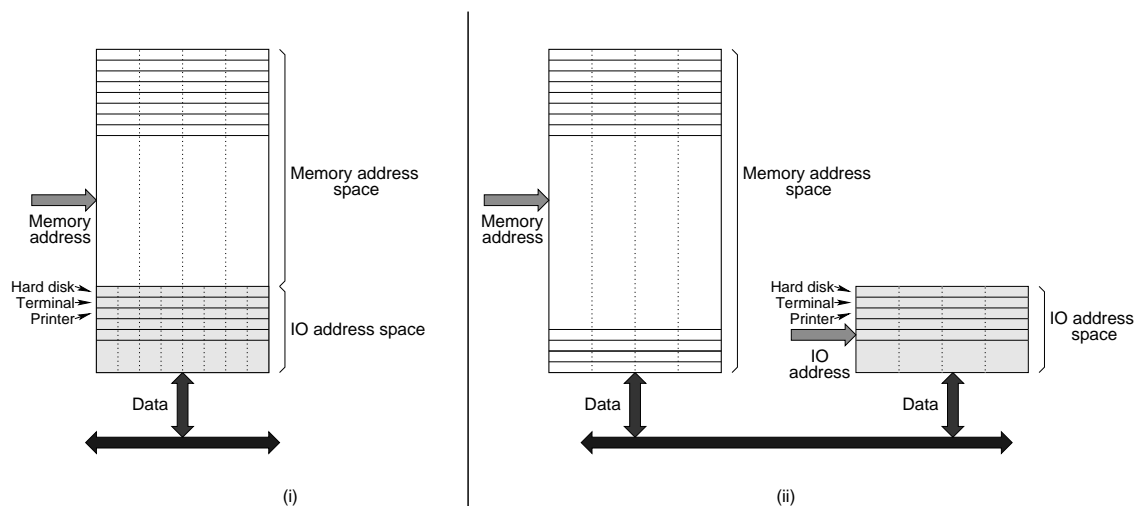


Figure 4.5: Different IO Address Mappings: (i) Memory Mapped IO; (ii) Independent IO

### 4.3.3 Operating System's Use of IO Addresses

Irrespective of the type of IO addressing used, the OS views the IO address space as a collection of **IO ports** or **IO programming interfaces**. The specifications of different IO ports can be different, and are determined based on the type of IO devices that are meant to be connected to them. Most of the ports adhere to one standard or other, as we will see in Chapter 8.

In an ISA that specifies IO registers, each IO port then encompasses several consecutive IO registers. Depending on the characteristics of the port, some of its registers are used to record status information related to the port (*IO status registers*); individual bits of a status register may correspond to a different attribute of the port for example, the least significant bit may specify whether a new character has been typed on the keyboard, the next bit may specify whether an error has occurred, etc.). Other registers may be used to store data to be transferred between IO ports and general-purpose registers or main memory. These registers are called *IO data registers*. Some of the registers in a port may be viewed as read-only, some may be write-only, and the rest may be read-write. The OS decides which IO port needs to be accessed to access a particular file.

Notice that all that the OS program “sees” of an IO device is its port, which includes a set of IO registers and specifications for the operation of these registers. When an IO write instruction writes a value to a status/control register in an IO port, the device controller interprets it as a command to a particular IO device. When an OS routine wants to know the status of an IO device, it uses an IO read instruction to read the status register of its IO port. Similarly, an IO write instruction can be executed by an OS routine to send data to the data register or status register of the IO port pertaining to an IO device. An example IO write instruction for the MIPS-I architecture is

```
sw    $t1, keyboard_status
```

where `keyboard_status` is the address assigned to the status register of the IO port pertaining to the keyboard.

The astute reader would have realized by now that the IO model presented to OS programmers, although more concrete than the one presented to applications programmers, still does not provide *direct* access to the IO devices. That is, the instructions in the OS program do not directly access the IO device hardware. There are a variety of reasons for this:

- The IO devices are incredibly diverse. It would be impractical for the OS device drivers to incorporate the necessary functionality to directly control a wide range of IO devices. For example, if the OS directly accesses a hard disk, to get a data item from the disk, the device driver would need to execute many instructions that deal with the intricate details of how exactly that disk works. In the future, if this disk is replaced by a slightly different hard disk, then the device driver (to access the disk) may also need to be changed.

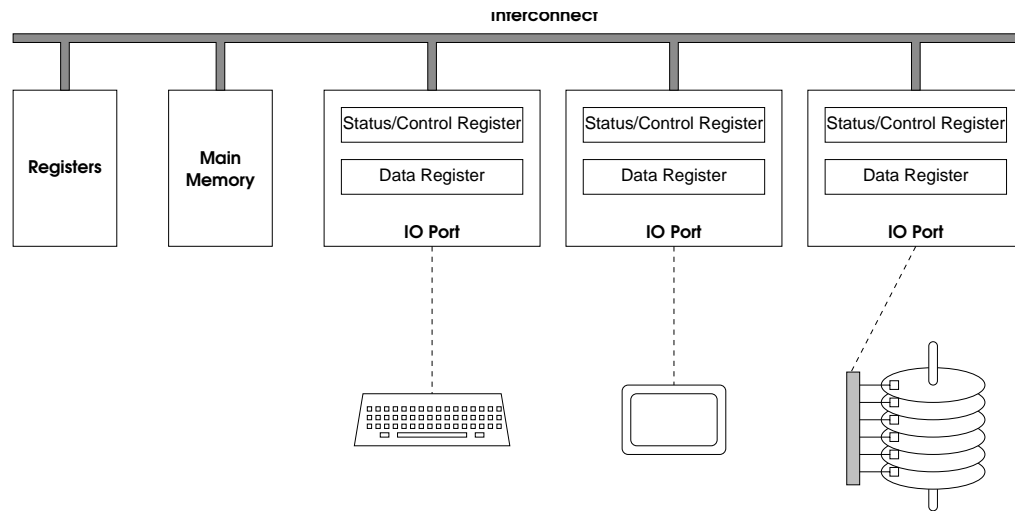


Figure 4.6: Abstraction of IO Devices Presented to the OS

- IO devices are often electromechanical devices whose manner of operation is different from that of the rest of the machine, which are implemented using electronic devices. Therefore, a conversion of signal values may be required.
- The data transfer rate of IO devices is often much lower than that of the rest of the system, necessitating special synchronization operations for correct transfer of data.
- IO devices often use data formats and word lengths that are different from those specified in the kernel mode.

Because IO ports are modeling electromechanical IO devices, the behavior of IO registers can be quite different from that of ordinary registers and memory locations. An assembly language systems programmer needs to be aware of these differences. The following differences may come as surprises:

- IO registers may be active elements, and not just passive storage elements. A write to an IO register often has side effects, such as initiating an activity by the IO device connected to the port. A read to an IO register may also have side effects, such as clearing an interrupt request or clearing an error condition.
- IO registers may have timing characteristics that are different from ordinary memory. If a write to an IO register is expected to produce a visible change in that or some other register, the device driver programmer may need to introduce a pause, for example by executing `nops`, to give the device time to respond.

- A read to an IO register may not necessarily return the value that was last written to it. This is because some bits do not exist (always zero or meaningless), and some others do not store a value, but are only sensitive to the value conveyed to them during a write operation. Sometimes the contents or meaning of a bit varies, based on the contents of other IO registers. In some cases, a single IO register may serve the dual purpose of being a *command* register as well as a *status register*. The device driver writes to this IO register to send commands to the interface, and reads from the same register to obtain status information. Depending on whether the register is being read or written, the contents associated with the register are different<sup>4</sup>! Finally, an IO register may be updated by the IO device connected to the port.

The behavior of an IO register depends on the specifics of the IO port to which it belongs. The specifications of an IO port include the characteristics of its IO registers. Often, it is possible to program the behavior of IO registers by writing specific commands in the control/status registers of the same IO port. All of this depends on the port specifications. It is therefore important to study the device's manual, and learn how it functions, before writing device drivers that control that IO device.

Although the above discussion seems to imply that the grouping of IO registers into IO ports is done at the time of OS development, that is rarely the case. The behavior of a port is very much dependent on the IO device it models. At the time of writing an OS, it is difficult to know which devices will be eventually connected to a particular computer system that uses that OS. Therefore, in practice, the OS is split into two parts—the kernel and the device drivers (or device handlers, or IO handlers, or software drivers). The kernel consists of the parts that do not change from system to system. The device drivers are specific to each computer system, and are usually added on when each IO device is hooked to the system.

**Standard IO Ports:** We will see later how this IO model is used for writing assembly language device driver routines that can do specific IO operations, such as reading a character from a keyboard.

## 4.4 Operating System Organization

In modern computers, the operating system is the only software component that runs in the Kernel mode. It behooves us therefore to consider the structure and implementation of an operating system. In particular, it is important to see how the system calls specified in the application programming interface (API) (provided to user programs and library functions)

---

<sup>4</sup>At the microarchitectural level, the IO interface module typically implements such a register by two separate registers. It is reasonable to ask why the assembly-level architecture and the ISA define a single IO register for dual functions. The reason is to conserve the IO address space, which was once a scarce resource.

are implemented by operating systems. The exact internal details of an operating system vary considerably from one system to another. It is beyond the scope of this book to discuss different possibilities. Figure 4.7 gives a possible block diagram, which is somewhat similar to that of a standard UNIX kernel. In the figure, the kernel mode software blocks are shown shaded. The main components of the OS software include a system call layer, file system, process control system, and the device management system (device drivers). This organization uses a layered approach, which makes it easier to develop the OS and to introduce modifications at a later time. This also makes it easier to debug the OS code, because the effect of bugs may be restricted to a single layer. The figure also shows the relationship of the OS to user programs, library routines, and the hardware.

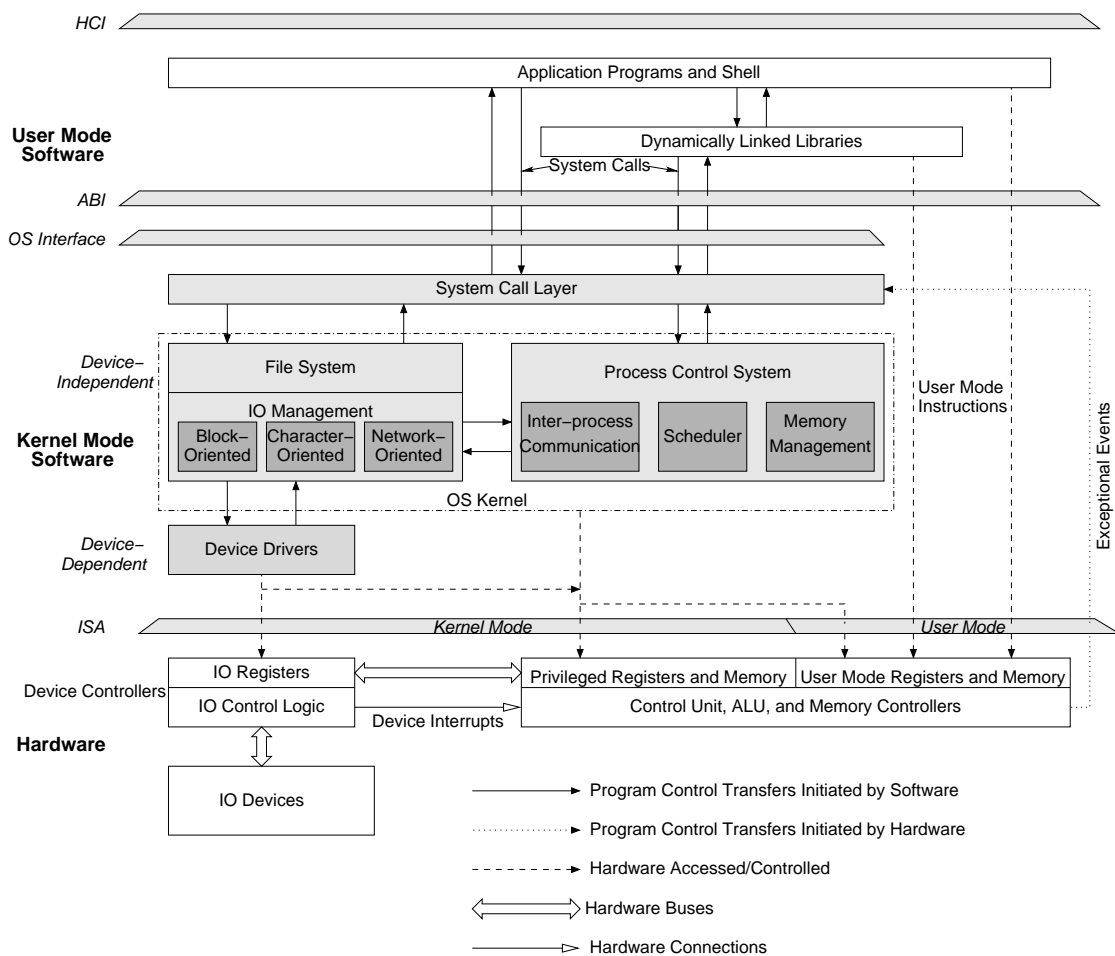


Figure 4.7: Block Diagram Showing the Structure of a UNIX-like Operating System



### 4.4.1 System Call Layer

The system call layer provides one or more entry points for servicing syscall instructions and exceptions, and in some cases device interrupts also. The user program conveys the system call type by placing the system call number on the user stack or in a register; the MIPS-I assembly language convention is to use register \$2 for this purpose. The system call layer copies the arguments of the system call from the user stack (or registers) and saves the user process' context, possibly on the kernel stack. It then uses the system call number to look up a *system call dispatch vector* to determine the kernel function to be called to implement that particular system call, interrupt, or exception. It then calls that kernel function, sometimes mapping or converting the arguments. When this kernel function completes, the system call layer sets the return values and error status in the user stack (or registers), restores the user process' context, and switches to User Mode, transferring control back to the user process.

We can summarize the functions performed by the system call layer:

- Determine type of syscall
- Save registers
- Call appropriate handler
- Restore registers
- Return to user program

### 4.4.2 File System

The API provided to application programs by the operating system, as we saw earlier, includes *device-independent IO*. That is, the interface is the same, irrespective of the physical device that is involved in the IO operation. The file abstraction part of the API is supposed to hide all device-specific aspects of file manipulation from HLL application programmers, and provide them with an abstraction of a simple, uniform space of named files. Thus, HLL application programmers can rely on a single set of file-manipulation OS routines for file management (and IO device management in an indirect manner). This is sometimes referred to as **device-independent IO**.

As we saw in Section 3.4.7, application programs access IO (i.e., files) through **read** and **write** system calls. The **read** and **write** system calls (of the User mode) are implemented in the Kernel mode by the *file system* part of the OS, possibly with the help of appropriate *device drivers*.

Files of a computer installation may be stored on a number of physical devices, such as disk drives, CD-ROM drives, and magnetic tapes, each of which can store many files. If the IO device is a storage device, such as a disk, the file can be read back later; if the device is

device-  
independent  
IO

a non-storage device such as a printer or monitor, the file cannot be read back. Different files may store different kinds of data, for example, a picture, a spreadsheet, or the text of a book chapter. As far as the OS is concerned, a file is simply a sequence of bytes written to an IO device.

The OS partitions each file into *blocks* of fixed size. Each block in a file has an address that uniquely tells where within the physical device the block is located. Data is moved between main memory and secondary storage in units of a single block, so as to take advantage of the physical characteristics of storage devices such as magnetic disks and optical disks.

File management related system calls invoked by application programs are interpreted by the file system part of the OS, and transformed into device-specific commands. The process of implementing the `open` system call thus involves locating the file on disk, and bringing into main memory all of the information necessary to access it. The OS also reserves for the file a *buffer* space in its memory space, of size equal to that of a block. When an application program invokes a system call to write some bytes to a file, the file system part of the OS writes the bytes in the buffer allotted for the file. When the buffer becomes full, the file system copies it into a block in a storage device (by invoking the device's device driver); this block becomes the next block of the file. When the application process invokes the `close` system call for closing a file, the file system writes the file's buffer as the final block of the file, irrespective of whether the buffer is full or not, prior to closing the file. Closing a file involves freeing up the table space used to hold information about the file, and reclaiming the buffer space allotted for the file.

#### 4.4.3 Device Management: Device Drivers

The device management part of the OS is usually implemented separate from the kernel, to facilitate easy adding or removal of devices. It is actually a collection of device drivers. Most computers have input/output devices and storage devices such as disks, terminals, and printers. Each of these devices requires specific *device driver* software, which acts as an interface between the device controller and the file system part of the OS kernel. A specific device driver is important, because each device has its own specific commands instead of generic commands. Each device driver itself is a collection of routines, and can have multiple entry points. The device driver receives generic commands from the OS file system and converts them into the specialized commands for the device, and vice versa. To the extent possible, the driver software hides the unique characteristics of a device from OS file system.

A device driver, or a software driver is a specific type of computer software, developed to interact with hardware devices. This usually constitutes an interface for communicating with the device, through the specific computer bus or communications subsystem that the hardware is connected to, providing commands to and/or receiving data from the device, and on the other end, the requisite interfaces to the operating system and software applications.

Because of its interfacing nature, it is specific to the hardware device as well as to the operating system.

The key design goal of device drivers is abstraction. Every model of hardware (even within the same class of device) is different. Newer models also are released by manufacturers that provide more reliable or better performance and these newer models are often controlled differently.

The operating system cannot be expected to know how to control every device, both now and in the future. To solve this problem, operating systems essentially dictate how every type of device should be controlled. The function of the device driver is then to translate these OS mandated function calls into device specific calls. In theory a new device, which is controlled in a new manner, should function correctly if a suitable driver is available. This new driver will ensure that the device appears to operate as usual from the operating systems' point of view.

Device drivers can be fairly complex. Many parameters may need to be set prior to starting a device controller, and many status bits may need to be checked after the completion of each device operation. Many device drivers such as the keyboard driver are supplied as part of the pre-installed system software. Device drivers for other devices need to be installed as and when these devices are installed.

The routines in a device driver can be grouped into three kinds, based on functionality:

- Autoconfiguration and initialization routines
- IO initiation routines
- IO continuation routines (interrupt service routines)
- IO initiation routines
- IO continuation routines (interrupt service routines)

The autoconfiguration routines are called at system reboot time, to check if the corresponding device controller is present, and to perform the required initialization. The IO initiation routines are called by the OS file system or process control system in response to system call requests from application programs. These routines check the device status, and initiate IO requests by sending commands to the device controller. If program-controlled IO transfer is used for the device, then the IO initiation routines perform the IO transfers also. By contrast, if interrupt-driven IO transfer is used for the device, then the actual IO transfer is done by the interrupt service routines when the device becomes ready and issues an interrupt.

#### 4.4.4 Process Control System

##### 4.4.4.1 Multi-Tasking

When a computer system supports multi-tasking, each process sees a separate virtual machine, although the concurrent processes are sharing the same physical resources. Therefore, some means must be provided to separate the virtual machines from each other at the physical level. The physical resources that are typically shared by the virtual machines are the processor (including the registers, ALU, etc), the physical memory, and the IO interfaces. Of these, the processor and the IO interfaces are typically time-shared between the processes (*temporal separation*), and the physical memory is partitioned between the processes (*spatial separation*)<sup>5</sup>. To perform a *context switch* of the virtual machines, the time-shared resources must be switched from one virtual machine to the next. This switching must be managed in such a way that the virtual machines do not interact through any state information that may be present in the physically shared resources. For example, the ISA-visible registers must be saved and restored during a context switch so that the new context cannot access the old context's register state.

Decisions regarding time-sharing and space-sharing are taken in the Kernel mode by the operating system, which is responsible for allocating the physical resources to the virtual machines. If a user process is allowed to make this decision, then it could possibly encroach into another process' resources, and tamper with its execution. The operating system's decisions, however, need to be enforced when the system is in the User mode. This enforcement is done using special hardware (microarchitectural) support so that the enforcement activity does not reduce performance.

##### 4.4.4.2 Multi-Programming

Some applications can be most conveniently programmed for two or more cooperating processes running in parallel rather than for a single process. In order for several processes to work together in parallel, certain new Kernel mode instructions are needed. Most modern operating systems allow processes to be created and terminated dynamically. To take full advantage of this feature to achieve parallel processing, a system call to create a new process is needed. This system call may just make a clone of the caller, or it may allow the creating process to specify the initial state of the new process, including its program, data, and starting address. In some cases, the creating (parent) process maintains partial or even complete control over the created (child) processes. To this end, Kernel mode instructions are added for a parent to stop, restart, examine, and terminate its children.

---

<sup>5</sup>Time-sharing the entire physical memory is not feasible, because it necessitates saving the physical memory contents during each context switch.

## 4.5 System Call Layer for a MIPS-I OS

We just saw a functional organization of an operating system, and the important functions performed by each major block. To get a better appreciation of what the OS code looks like, let us get our feet wet with a detailed real-life example. In this section, we discuss the barebones of the system call layer of an OS for a MIPS-I machine. The system call layer implements the interface for system calls, device interrupts, and exceptions. We restrict ourselves to the system call layer for two reasons: (i) It is perhaps the smallest major block in an OS (and therefore manageable to be discussed in a book of this scope), but is detailed enough to reflect many of the idiosyncrasies that make OS routines different from application programs. (ii) It is the block that directly interacts with application programs, which is what many of the programmers care about.

### 4.5.1 MIPS-I Machine Specifications for Exceptions

Software is always written for a specific (abstract) machine specification. Before writing assembly-level systems software for the system call layer of a MIPS-I OS, we need to know how the MIPS-I machine specifies information regarding the occurrence of system calls, device interrupts, and exceptions. Interestingly, MIPS-I does not make a big distinction between the 3 categories—system calls, device interrupts, and exceptions—when reporting their occurrence. It treats them all as exceptions! To be specific, it does 3 things when an exceptional event occurs:

- It modifies privileged register **sr** (**status register**) to disable device interrupts and to reflect Kernel mode of operation.
- It stores the restart instruction's address in privileged register **epc**.
- Generates an *exception vector* depending on the type of exception.

The MIPS-I architecture provides only 3 exception vectors, in contrast to many others that provide a much larger set of exception vectors. These 3 vectors are stated below:

- **0xbfc00000**: This exception vector is specified at computer *reset*. Thus, after a reset, the computer starts executing in the Kernel mode the program starting at memory address **0xbfc00000**.
- **0x80000000**: This exception vector is specified when a *User TLB miss* exception occurs; Chapter 7 discusses this case in detail.
- **0x80000080**: This exception vector is specified when any other exceptional event occurs.

| ExcCode | Mnemonic | Expansion                       | Description  |
|---------|----------|---------------------------------|--|
| 0       | Int      | Interrupt                       | Device interrupt                                   |
| 1       | Mod      | TLB Modification Exception      | Attempt to write to an address marked as read-only |
| 2       | TLBL     | TLB Load Exception              | TLB miss for load or instruction fetch             |
| 3       | TLBS     | TLB Store Exception             | TLB miss for store                                 |
| 4       | AdEL     | Address Error Load Exception    | Address error for load or instruction fetch        |
| 5       | AdES     | Address Error Store Exception   | Address error for store                            |
| 6       | IBE      | Instruction Bus Error Exception | Bus error for instruction fetch                    |
| 7       | DBE      | Data Bus Error Exception        | Bus error for data reference                       |
| 8       | Sys      | Syscall                         | <b>syscall</b> instruction                         |
| 9       | Bp       | Breakpoint                      | <b>break</b> instruction                           |
| 10      | RI       | Reserved Instruction Exception  | Illegal instruction                                |
| 11      | CpU      | Co-processor Unusable Exception | Software can emulate the offending instruction     |
| 12      | Ovf      | Overflow Exception              | Arithmetic overflow                                |
| 13-15   | —        | Reserved                        |  |

Table 4.2: Exceptions Corresponding to Different Values of **ExcCode** Field of **Cause** register in a MIPS-I Architecture

The third exception vector (0x80000080) corresponds to different types of exceptional events, and so there must be some provision for the OS to know the exact cause of the exceptional event whenever this exception vector is generated. The MIPS-I architecture defines a privileged register called **Cause** for recording the cause of the most recent exceptional event. This register has a 4-bit **ExcCode** field, which holds an encoded bit pattern corresponding to the exception cause. Thus, a maximum of 16 different exception types can be uniquely specified by the machine in this field. Table 4.2 shows the **ExcCode** values for different types of exceptional events. Interestingly, a single value (0) corresponds to all types of device interrupts; however, there is an **IP** (interrupt pending) field in **Cause** which helps to do some differentiation between the interrupt sources. The OS can thus perform selective polling (of appropriate IO registers) to determine the exact cause for the interrupt. Similarly, a single value (8) corresponds to all types of **syscall** instructions; the OS can determine the type of **syscall** by inspecting the contents of register \$2.

### 4.5.2 OS Usage of MIPS-I Architecture Specifications

With the above background on the features provided in the MIPS-I architecture to support exceptional events, let us turn our attention to the barebones of a typical system call layer used in MIPS-I OSes. This interface code provides 3 entry points, corresponding to the 3 exception vectors: `0xbfc00000`, `0x80000000`, and `0x80000080`. Figure 4.8 shows these 3 entry points and the placement of the exception handler code in the MIPS-I kernel address space. The third entry point is common for a number of exceptional events, and so the routine at that entry point checks the **Cause** register to determine what caused the event. Depending on the contents of the **ExcCode** field of **Cause**, an appropriate handler is called. This checking is similar to that of implementing the C `switch` statement, and the standard way for making this selection is by means of a jump table that contains the starting addresses of the handler routines.

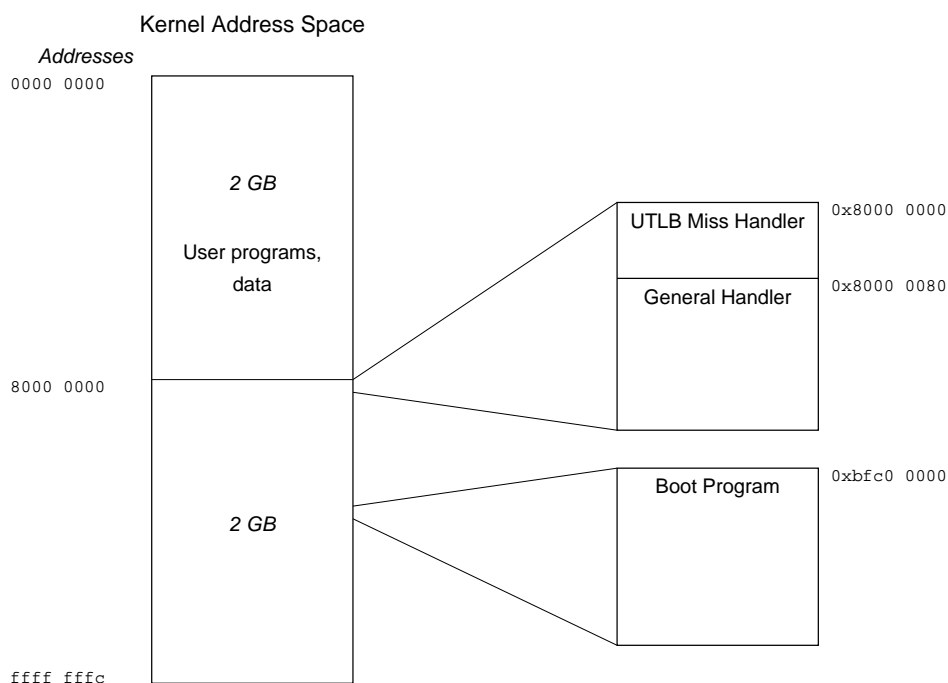


Figure 4.8: Placement of the Exception Handler Code in the MIPS-I Kernel Address Space

In the skeleton code given below, the jump table containing handler addresses is called `handler_table`. It is placed in the kernel's data section, and is initialized statically (i.e., at assembly time as opposed to run time).

Let us take a closer look at the handlers themselves, which are placed in the kernel's text section. The first handler given here—the one starting at address `0x80000000`—deals

with user TLB miss exceptions, and is discussed in detail in Chapter 8. It is included here just for the sake of completeness. The second handler, which starts at address 0x80000080, deals with general exceptional events. The first thing it does is to save the register values of the interrupted process. In particular, registers `s0-s7`, `k0`, `k1`, `sp`, and `epc` need to be saved. A good place to save them is the kernel stack<sup>6</sup>.

At the end of the handler code, we have a pair of instructions, `jr $k1` and `rfe`, which merit further discussion. The `jr $k1` instruction specifies the transfer of control back to the interrupted program. The `rfe` (restore from exception) instruction tells the machine to restore the system's mode and interrupt status to what it was prior to taking this exception; thus, it puts the system back in the condition in which the interrupted program was running. What is a good place to include the `rfe` instruction in the exception handler? If the `rfe` instruction is placed before the `jr` instruction, then the machine may try to execute in user mode the `jr` instruction, which is in the kernel code space. If, on the other hand, the `rfe` instruction is placed after the `jr` instruction, then control is transferred to the interrupted program, preventing the execution of the `rfe` instruction. What we really require is that these two instructions must be executed *atomically*, meaning this two-instruction sequence should be done together, without any interruptions<sup>7</sup>.

---

```
#####
# Handler Table
#####
        .kdata                # Store subsequent items in kernel data section
        .align 2
handler_table:
        .word  IntHandler      # Initialize to interrupt handler address
        .word  ModHandler      # Initialize to modification exception handler address
        .word  TLBLHandler     # Initialize to TLB load miss exception handler address
        .word  TLBSHandler     # Initialize to TLB store miss exception handler address
        .word  AdELHandler     # Initialize to load address error exception handler address
        .word  AdESHandler     # Initialize to store address error exception handler address
        .word  IBEHandler      # Initialize to instruction bus error exception handler address
        .word  DBEHandler      # Initialize to data bus error exception handler address
        .word  SysHandler      # Initialize to syscall handler address
```

---

<sup>6</sup>As with the user stack, most of the RISC machines do not provide direct support for a kernel stack either. The kernel stack is therefore implemented within part of the memory address space. Like the user stack, it is accessed by using the register-displacement addressing along with one of the general-purpose registers (which serves as the kernel stack pointer). Therefore, the kernel cannot save onto the kernel stack the value stored in this register by the interrupted program. If the interrupted program was using this register for its purposes, this presents a problem. The MIPS-I assembly language convention to solve this dilemma is to reserve registers `$k0` and `$k1` (i.e., `$26` and `$27`) for use by the OS.

<sup>7</sup>At the ISA level, the MIPS-I `jr` instruction uses the concept of *delayed branching*; i.e., the transfer of control induced by the `jr` instruction takes effect only after executing the immediately following instruction, which in this case is the `rfe` instruction. The later versions of MIPS have a kernel mode `eret` (exception return) instruction, which performs both actions in a single instruction.



```

        .word  BpHandler      # Initialize to breakpoint handler address
        .word  RIHandler      # Initialize to reserved instruction exception handler address
        .word  CpUHandler     # Initialize to co-processor unusable exception handler address
        .word  OvHandler      # Initialize to arithmetic overflow exception handler address

#####
# User TLB Miss Handler
#####
        .ktext 0x80000000      # Store subsequent items in kernel text section
UTLBMiss_Handler:             # starting at address 0x80000000
        mfc0    $k0, $context   # Copy context register contents (i.e., kseg2 virtual
                                # address of required user PTE) into GPR k0
        mfc0    $k1, $epc       # Copy epc contents (address of faulting instruction)
                                # into GPR k1
        lw      $k0, 0($k0)     # Load user PTE from kseg2 addr to GPR k0
                                # This load can cause a TLBMISS exception!
        mtc0    $k0, $EntryLo    # Copy the loaded PTE into EntryLo register
        tlbwr                                # Write the PTE in EntryLo register into TLB
                                # at slot number specified in Random register
        jr      $k1             # Jump to address of faulting instruction
        rfe                                # Switch to user mode

#####
# General Handler
#####
        .ktext 0x80000080      # Store subsequent items in kernel text section
General_Handler:              # starting at address 0x80000080

#####
# Save interrupted process' s0 - s7 registers, k0, k1, sp, and epc on kernel stack
...

#####
# Determine cause for coming here, and jump to appropriate handler
        mfc0    $k0, $cause     # Copy Cause register to R26
        andi    $k0, $k0, 0x3c  # Take out the ExcCode value
        lw      $k0, handler_table($k0) # Get starting address of appropriate handler
        jalr    $k0             # Call appropriate handler

#####
# Returned from handler
# Restore interrupted process' s0 - s7 registers, k0, k1, sp, and epc
# from kernel stack
...

#####
# Return to interrupted program
        mfc0    $k0, $epc       # Copy epc register to R26

```

```
jr      $k0          # Return to interrupted program
rfe                      # Restore from exception
```

---

## 4.6 IO Schemes Employed by Device Management System

This section gives an overview of the device management part of a typical OS. A complete treatment of the internal operation of the device management system of even a single OS is beyond the scope of this book.

As can be imagined, there is a wide disparity in the nature of the various abstract IO devices. Different schemes are available to accommodate this disparaging differences in speed and behavior. They are:

- *sampling*
- *program-controlled IO*
- *interrupt-driven IO*
- *direct memory access (DMA)*
- *IO co-processing*

As we go from the first scheme to the last, more functionality is shifted from the device driver to the IO interface. The scheme used by a device driver to perform IO transfers is, of course, transparent to the user program, because it is not specified in the API. We shall discuss each of these schemes in detail.

Central to all these schemes is the need to do appropriate *synchronization* between the device driver and the device. IO devices tend to be significantly slower than the processor. Moreover, their response times tend to have a wide variance. Because of these reasons, no assumptions can be made about the response times of IO devices. The different IO transfer schemes also differ in how synchronization is done between the device driver and the device.

### 4.6.1 Sampling-Based IO

This is the simplest of the IO data transfer methods. In this scheme, the IO device is treated like main memory; that is, the device is always ready to accept or to provide data, as appropriate. No checking of device status is required. The programming interface is extremely simple, consisting of just one or more data registers. Example IO devices that can be communicated in this manner are simple devices such as digital input port and motor port.

### 4.6.2 Program-Controlled IO

With program-controlled IO, the device driver being executed on the machine has direct control of the IO operation, including sensing the device status, sending a read or write command, and transferring the data. Sensing the device status may involve executing an IO read instruction to read the device controller's status register into a general purpose register (GPR), and then checking the appropriate bits of the GPR. This process of checking the status by continuous interrogation of the status register associated with the device is performed until the IO device becomes ready<sup>8</sup>. For example, the status register of the DEC LP11 line printer interface contains a *done bit*, which is set by the printer controller when it has printed a character, and an *error bit*, which indicates if the printer is jammed or out of paper. The bytes to be printed are copied from a general-purpose register to the data register of the printer interface (by means of IO write instructions), one at a time. Each byte is copied only after ensuring that the done bit has been set. The error bit is also checked every time to determine if a problem has occurred in the printer.

A flowchart of the device driver that carries out program-controlled IO is shown in Figure 4.9. The flowchart assumes that a sequence of words has to be read from an IO device and stored in main memory. The device driver continually examines the status of the device interface until the appropriate flag becomes 1. A word is then brought into a GPR, and transferred to main memory. The entire process is repeated until all of the data have been transferred.

*Example:* To review the basic concepts of program-controlled IO, consider the IO operations involved in reading a character from the keyboard and copying it to a memory location. For an application program, the convenient way to read a character is by invoking the operating system (by means of a syscall instruction). A MIPS-I assembly language application program for doing this IO operation is given below. Notice that the execution of the `syscall` instruction causes the system to switch to kernel mode, and execute many instructions in the kernel mode before returning to the user program.

---

```

.text
li    $a0, 0          # Place the file descriptor for keyboard (0) in $a0
la    $a1, buffer     # Place the starting address of buffer in $a1
li    $a2, 1          # Place the number of bytes to be read (i.e., 1) in $a2
li    $v0, read_code  # Place the code for read in $v0
syscall                # Call OS routine to perform the read

```

---

Now, assume that the keyboard's interface sets the LSB of its 8-bit status register to 1 whenever a new character is typed by the user. The device driver resets it to 0 after reading that character. The display's interface resets the next-to-LSB of its 8-bit status register to

---

<sup>8</sup>Because IO devices are slow, it is important to check their status prior to issuing a new command. For instance, when a new command is sent to a device, the device may not be ready to accept it because it is still working on the previous command. This situation can be recognized by checking its status.

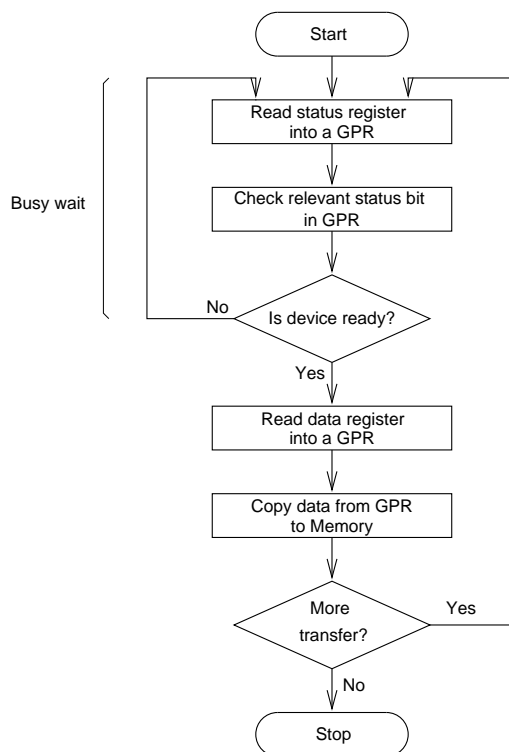


Figure 4.9: A Flowchart Depicting the Working of a Device Driver that Reads Data from an IO Device using Program-Controlled IO Transfer

0 whenever the display is ready to accept a new character. The device driver sets it to 1 when it writes a character to the display interface's data register. Figure 4.10 illustrates the interface provided by the IO devices, along with the specific addresses given to the IO registers.

Let us write an assembly language device driver routine that will be called when the above `syscall` instruction is executed. First of all, when the `syscall` instruction is executed, the execution mode switches to Kernel mode, and control passes over to the `system call layer` part of the operating system. This routine saves the User process' registers and other relevant state on the Kernel stack. It then calls the `SysHandler` routine, which subsequently calls the keyboard device driver. Below, we present an example code for a keyboard device driver that reads from the keyboard up to as many characters as specified in register `$a2`. If `$a2` contains zero or a negative number, then no character is read. Similarly, reading stops when the end of line is reached, which is detected by checking the read character against `'\'`. The number of characters actually read is placed in register `$v0`. The read characters are placed in consecutive memory locations, starting from the address

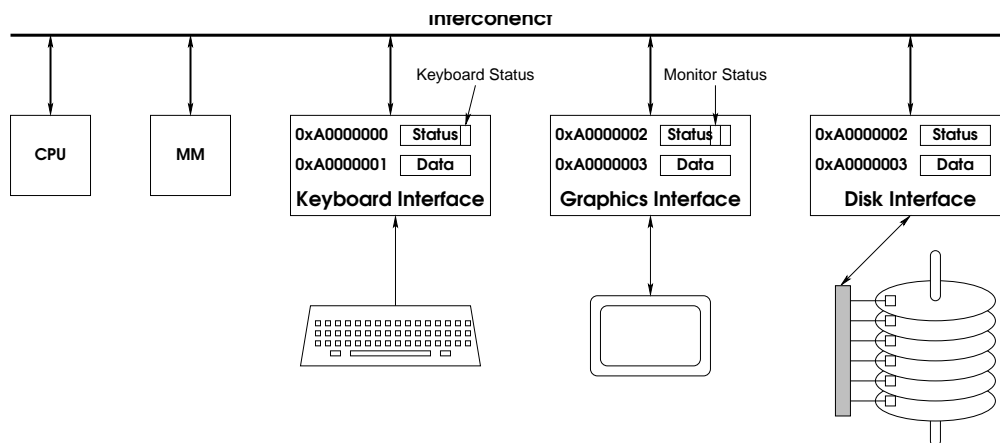


Figure 4.10: A Flowchart for an Assembly Language Device Driver to Read Data from an IO Device using Program-Controlled IO Transfer

originally specified in register `$a1`. Notice that this simple device driver does not check for error conditions, one of the tasks a real device driver must do.

---

```
#####
# Keyboard Read Device Driver: Called by OS File System
# $a1 contains address of buffer; $a2 contains number of bytes to read
#####
```

```
        .ktext                # Store subsequent items in kernel text section
keybd_read:
    li    $v0, 0               # Initialize number of bytes read ($v0) to 0
    blez  $a2, read_done      # Go to label read_done if no byte is to be read
read_loop:
    lb    $9, keybd_status     # Read keyboard status register into R9
    andi  $10, $9, 1           # Isolate the status bit for keyboard input
    beqz  $10, read_loop       # Branch back to label read_loop if status bit is not set
    lb    $11, keybd_data      # Read the byte from keyboard data register
    sb    $11, 0($a1)          # Store the byte in the buffer
    andi  $12, $9, 0xFE        # Reset keyboard input status bit (LSB) to 0
    sb    $12, keybd_status     # Update keyboard status register
    addu  $v0, 1               # Increment number of bytes read
    beq   $11, '\n', read_done # Go to label read_done if end of line
    addu  $a1, 1               # Increment address of buffer to store next byte
    subu  $a2, 1               # Decrement number of bytes to be read
    bnez  $a2, read_loop       # Go to read_loop if there are more bytes to be read
read_done:
    jr    $ra                  # Return control to OS file system
```

---

The `read_loop` in the above device driver constantly checks the keyboard status until the next character has been typed. This type of continuous checking of the device status bits to see if it is ready for the next IO operation is a hallmark of program-controlled IO. The device controller places the information in one of its status registers, and the device driver gets this information by reading this register. The device driver is in complete control of the IO transfer.

The disadvantage of program-controlled IO is that it can waste a lot of time executing the driver because the IO devices are generally slower compared to the other parts of the system. The driver code may read the device status register millions of times only to find that the IO device is still working on a previous command, or that no new character has been typed on the keyboard since the last time it was polled. For output devices, it has to do this status checking until the output operation is completed to ensure that the operation was successful. Ideally, it is preferable to execute a device driver only when a device is ready for a new transfer and a transfer is required. For instance, the keyboard driver needs to be executed only when a character is available in the input buffer of the keyboard interface. Similarly, the printer driver needs to be executed only when the printer has completed a previous print command, or is ready to accept a new command and a print request is pending.

### 4.6.3 Interrupt-Driven IO

The overhead in program-controlled IO transfer was recognized long ago, leading to the use of interrupt-driven IO for at least some of the peripheral devices. An interrupt is generated by an IO port to signal that a status change has occurred. With interrupt-driven IO, device driver routines need not be continuously executed to check the status of IO ports; instead other useful programs can be executed until the IO device becomes ready. Indeed, by using interrupts, such waiting periods can almost be eliminated, provided the system has a sufficient degree of multiprogramming.

When an interrupt is generated, the system acknowledges it when it is ready to process the interrupt. At that time, the system enters the Kernel execution mode, and the appropriate interrupt service routine (ISR) or interrupt handler is called with one or more parameters that uniquely identify the interrupting device. The parameters are important, because a single ISR may handle multiple devices of the same type. If the identity of the interrupting device is not supplied, the ISR may need to poll all potential IO ports to identify the interrupting one.

Data transfer by means of interrupts is illustrated in Figure 4.11 as a time line. Time is depicted along the horizontal direction. Process A executes a `read` system call at time T1. Control is then transferred to the `syscall interface` routine of the OS, from where control is transferred to the file system, and then to the appropriate device driver. After the device driver updates the relevant status information, control is transferred to the `process control system`, which then allows process B to run (at time T2). The device becomes

ready at time T3, and generates an interrupt. Control is transferred to the OS, which then calls the ISR for the device. After the ISR completes, the **process control system** allows process A to continue. Notice that during the time taken by the device to become ready (T2 to T3), the system is utilized by executing process B.

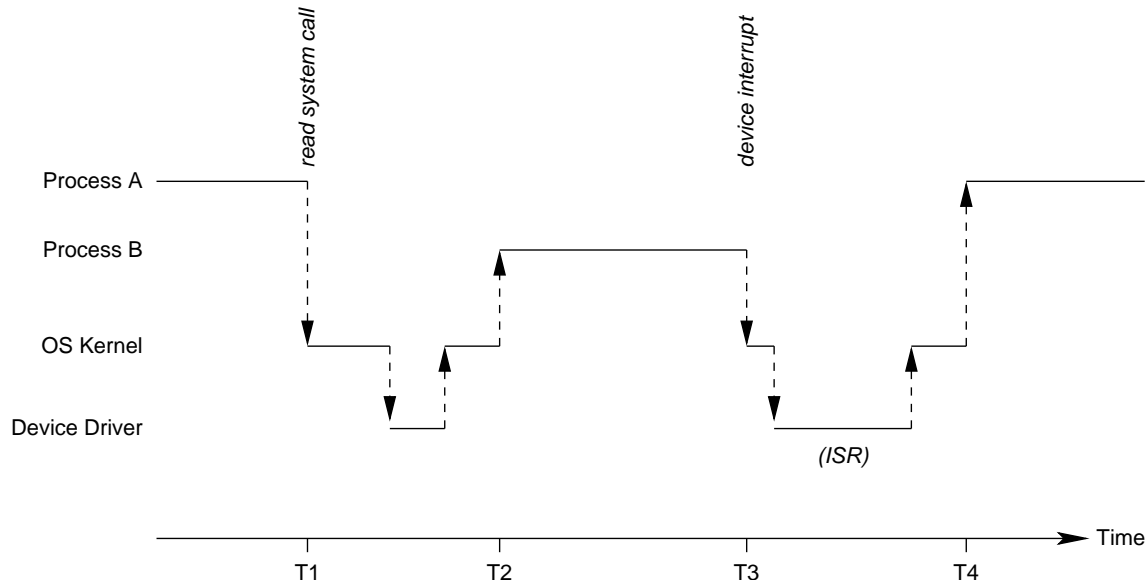


Figure 4.11: A Timeline Depicting an Interrupt-driven IO Transfer on behalf of Process A

*Example:* The keyboard interface raises an interrupt when a character is typed on the keyboard. When the interrupt is accepted, control goes to the system call layer of the OS, which saves the interrupted process' state and registers. It then calls the interrupt handler routine, which subsequently calls the ISR part of the keyboard device driver. Below, we present the ISR part of the keyboard device driver to read in a character from the keyboard interface to memory location `buffer`. Again, we assume that the status register address is named `keybd_status` and the data register address is named `keybd_data`.

```
.ktext                                # Store subsequent items in kernel text section
keybd_isr: subu $sp, 12                # Update stack pointer ($sp)
          sw  $8, 0($sp)                # Save value of R8 in stack before overwriting it
          mfc0 $8, C12                  # Copy processor status register to R8
          and $8, $8, 0xFFFFXXFF      # Code to disable lower priority interrupts
          mtc0 C12, $8                  # Copy R8 to processor status register
          sw  $9, 4($sp)                # Save value of R9 in stack before overwriting it
          sw  $10, 8($sp)               # Save value of R10 in stack
          lb  $9, keybd_data            # Read keyboard interface's data register
          sb  $9, buffer                # Store the byte in the buffer
```

```

lb    $9, keybd_status    # Read keyboard interface's status register
and   $9, $9, 0xFE        # Reset keyboard's status bit (LSB) to 0
sb    $9, keybd_status    # Update keyboard interface's status register
lw    $9, 4($sp)          # Restore value of R9 from stack
lw    $10, 8($sp)         # Restore value of R10 from stack
mfc0  $8, C12             # Copy processor status register to R8
or    $8, $8, 0xFF00      # Code to enable lower priority interrupts
mtc0  C12, $8             # Copy R8 to processor status register
lw    $8, 0($sp)          # Restore value of R8 from stack
addu  $sp, 12             # Restore stack pointer ($sp)
jr    $ra                 # Return control to calling routine

```

A number of actions are involved in processing an interrupt. Figure 4.12 clearly depicts these actions.

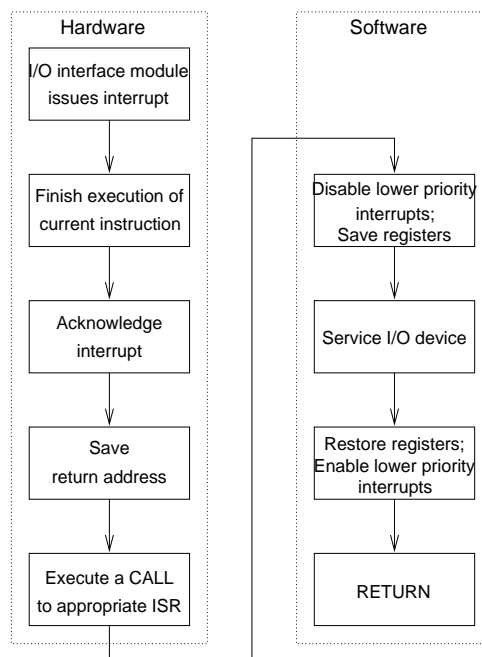


Figure 4.12: A Flowchart Depicting the Steps Involved in Processing an Interrupt

### Handling Multiple Interrupts

We now consider a computer system in which several interfaces are capable of raising interrupts. Because these interfaces operate independently, there may not be a specific order in which they generate interrupts. For example, the keyboard interface may generate an interrupt while an interrupt caused by a printer interface is being serviced, or all device interfaces may generate interrupts at exactly the same time, in a pathological situation!



The presence of multiple interrupt-raising device interfaces raises several questions:

1. How can the system identify the IO port that generated an interrupt?
2. Because different devices probably require different ISRs, how to obtain the starting address of the appropriate ISR?
3. Should an ISR for one device be allowed to be interrupted (by another IO port)?
4. What should be done if multiple device interfaces generate interrupts at the same time?

The ways in which these problems are resolved vary considerably from one machine to another. The approach taken in any machine is an important consideration in determining the machine's suitability for specific real-time applications. We shall discuss some of the common techniques here.

**Device Identification:** Consider the case in which a device interface requests an interrupt by activating an **Interrupt Request** line that is common to all device interfaces. When a request is received over the common **Interrupt Request** line, additional information is needed to identify the particular device that needs attention. One option is to set appropriate bits in the status register of the interrupting device interface, so that the interrupting device can be identified by *polling*. A better option is to use a scheme called *vectored interrupts*, in which the interrupting device's identification is sent as a special code along with the interrupt.

**Prioritizing Interrupts:** Now let us consider the situation in which several interrupts occur simultaneously. In this case, the system must decide which device to service first. A priority interrupt system establishes a priority over the various interrupt sources to determine which interrupt request to service first when two or more arrive simultaneously. The system may also determine which interrupts are permitted to interrupt an ISR. Higher levels of priority are assigned to requests that, if delayed or interrupted, could have serious consequences. Devices with high-speed transfers such as magnetic disks are given high priority, and slow devices such as keyboards receive the lowest priority. When two devices raise interrupt at the same time, the computer services the device with the higher priority first. Establishing the priority of simultaneous interrupts can be done by software or hardware. Software uses a polling procedure to identify the interrupt source of higher priority. In this method, there is one common routine to which the control is transferred upon receiving an interrupt. This routine performs the polling of the interrupt sources in the order of their priority. In the hardware priority scheme, the priority is implemented by some kind of hardware.

#### 4.6.4 Direct Memory Access (DMA)

Both program-controlled IO and interrupt-driven IO are particularly suited for interacting with low-bandwidth IO devices. Both schemes place the burden of moving data and managing the transfer on a device driver or OS ISR, which does so with the help of IO read/write instructions such as

```
lw      R1, data_register
```

Of course, an instruction to transfer input or output data is executed only after executing other instructions that confirm that the IO interface is indeed ready for the transfer. In either case, considerable overhead is incurred, because several instructions must be executed to transfer each data word. The reason is that instructions are needed for performing tasks such as incrementing the memory address and keeping track of the word count. This overhead becomes intolerable for performing high-bandwidth transfers such as disk IO. For such high-bandwidth devices, most of the transfers involve large blocks of data (hundreds to thousands of bytes). So computer architects invented a mechanism for off-loading the device driver, and letting the IO interface directly transfer data to or from main memory without involving the device driver. This kind of transfer is called direct memory access (DMA).

##### Steps Involved in Setting Up a DMA Transfer:

Although the DMA mode of data transfer happens without interrupting the program being executed, its operation is still initiated by a device driver routine. To initiate the transfer of a block of words, instructions are executed by the device driver routine to convey the following information to the *DMA interface* (IO interface capable of performing DMA transfers):

- the address or identity of the IO device
- the starting address in the main memory
- the number of words to be transferred
- the direction of transfer (read/write)
- the command to initiate the DMA operation

Once this information is conveyed to the DMA interface, the device driver stops execution, and another program is executed. When the entire block has been transferred, the DMA interface raises an interrupt. An interrupt service routine is then executed to interrogate the DMA interface to verify that the entire operation indeed completed successfully. Figure 4.13 depicts the actions involved in a DMA transfer as a flowchart.

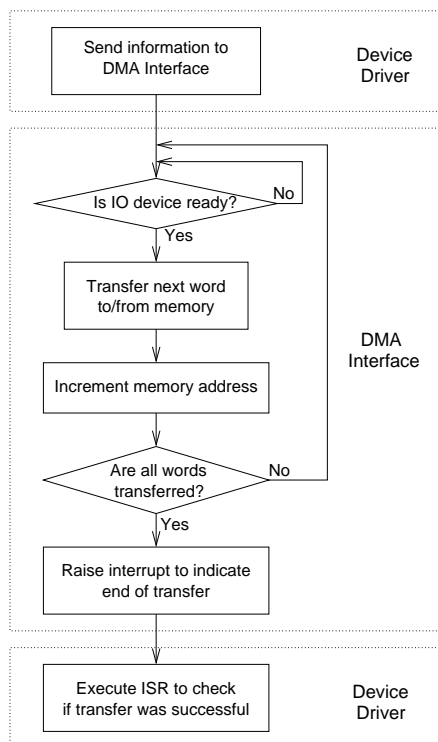


Figure 4.13: A Flowchart Depicting the Steps Involved in a DMA Transfer

### 4.6.5 IO Co-processing

The DMA scheme offers significant performance improvements over interrupt-driven IO whenever large blocks of data are to be transferred. Is it possible to do even better? To answer this question, we must first look at the limitations of the DMA scheme. First, each DMA operation has to be initiated by a device driver, by executing several instructions. Second, a DMA interface can handle only a single DMA operation at a time. To overcome these limitations, we can use a scheme called IO co-processing, in which (complex) IO functions are implemented by IO routines, which are executed (by IO processors or IO channels or peripheral processing units (PPUs)<sup>9</sup> in parallel with normal execution of programs (by the main processor). The IO routines are also typically placed in the memory address space. The execution of an IO routine is triggered by the device driver as in the case of a DMA

<sup>9</sup>An IO channel is itself a computer, albeit simple, which is capable of executing a small set of general-purpose instructions along with many special-purpose IO instructions such as “read a track from the disk” or “skip forward two blocks on the tape.” More sophisticated IO processors such as a peripheral processing unit (PPU) can do even more sophisticated IO functions. Similarly, a modern graphics processing unit (GPU) does complex graphics functions such as rendering, and hidden surface calculation.

operation.

#### 4.6.5.1 Graphics Co-Processing

#### 4.6.6 Wrap Up

As we can see, there is a trade-off between IO interface complexity and performance. When we use a simple, inexpensive device interface, most of the functionality is implemented in software (its device driver), and hence its performance will be low. A more expensive interface, on the other hand, performs more functions in hardware, and is therefore faster.

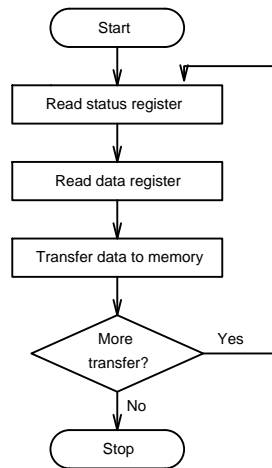
### 4.7 Concluding Remarks

*“Anybody who really knows computers spends a certain amount of time helping out friends who have loaded up their computers with trash, viruses, and spyware.”*  
— Howard Gilbert in *Introduction to PC Hardware*

### 4.8 Exercises

1. Consider a situation in which an OS routine calls a user mode library routine. What mode — user mode or kernel mode — will the library routine execute?
2. Explain the role played by the *system call layer* in a MIPS-I operating system.
3. Explain why interrupts are disabled when handling an exception.
4. Consider a simple embedded computer with two IO devices—a transmitter and a receiver. **Describe the protocol** Write a MIPS-I assembly language program (kernel mode) to accept data from the receiver and send them to the transmitter.
5. Consider a computer system that supports only a single mode of operation, i.e., it does not provide a separate user mode and a kernel mode. What would be the shortcomings of such a computer system?
6. For the computer system mentioned in the previous problem, explain how the hardware can provide some minimal functions traditionally carried out by the operating system, such as resource allocation among the processes.
7. What does a computer system do when two IO devices simultaneously raise interrupts? How does the computer recognize the identity of the devices?

8. Explain the differences between hardware interrupts, software interrupts, and exceptions. What do they have in common? Give an example of each.
9. A device driver programmer uses the following flow-chart for performing program-controlled reads from a keyboard. Explain what could go wrong with this scheme. Give the correct flow-chart.



## Chapter 5

# Instruction Set Architecture (ISA)

*Whoever loves instruction loves knowledge.*

**Proverbs 12: 1**

The central theme of this book is to view a modern computer as a series of architectural abstractions, each one implementing the one above it. We have already seen the high-level architecture and the two distinct modes of the assembly-level architecture. This chapter discusses the architectural abstraction immediately below the assembly-level architecture, called the instruction set architecture (ISA). In principle, this architecture is defined by how the computer appears to a machine language programmer. We can also view it as a set of rules that describe the logical function of a computer as observable by a (machine language) program running on that machine. Historically, this architecture came into existence before any of the other architectures we saw so far. The ISA has a special significance that makes it important for system architects: it is the interface between the software and the hardware. This architecture serves as the boundary between the hardware and the software in modern computer systems, and thus provides a functional specification for the hardware part of a computer system.

In the early days of computers, all of the programming used to be done in machine language! This was found to be so tedious that computer scientists quickly invented assembly languages, which used symbols and notations that were closer to the way people think and communicate. Once assembly languages were introduced as a symbolic representation for machine languages, programmers virtually stopped using machine languages to program. And, in theory, we can build hardware units that directly interpret assembly language programs. It is reasonable to ask why we still maintain a separate virtual machine level at the ISA level. The reason is that hardware that directly interprets assembly language programs will be significantly more complex than one that directly interprets machine language programs. The main difficulty arises from the use of labels and macros in assembly languages.

By defining a lower-level abstraction at the ISA level, software in the form of assemblers can be used to translate assembly language programs to machine language programs, which can be more easily interpreted by hardware circuits.

The ISA does not specify the details of exactly how the hardware performs its functions; it only specifies the hardware's functionality. Thus, an ISA specifies general-purpose registers, special registers, a memory address space, and an instruction set<sup>1</sup>. The ISA provides a level of abstraction that allows the same (machine language) program to be run on a family of computers having different implementations (i.e., microarchitectures). An example is the x86 ISA which is implemented in the 80386, 80486, Pentium, Pentium II, Pentium III, and Pentium 4 processors by Intel Corporation, and in the K6 and Athlon processors by AMD.

## 5.1 Overview of Instruction Set Architecture

We shall begin this chapter with an overview of the basic traits of an instruction set architecture. The assembly-level architecture that we covered in the last two chapters is nothing but a symbolic representation of the ISA. Many aspects of the ISA are therefore very similar to that of the assembly-level architecture. In fact, both have very similar memory models, register models, and IO models. The main differences between the two abstraction levels are in the two areas of language and instruction set. Let us take a detailed look into both of these issues.

### 5.1.1 Machine Language

The language used to specify programs that are targeted to an ISA has been historically called the machine language. The most notable feature of machine languages is their alphabet, which is restricted to just two symbols (0 and 1). Apart from allowing only an extremely frugal set of symbols for the language (as opposed to the richer set of alphanumeric characters available for an assembly language), the ISA also bans many of the luxuries permitted by the assembly-level architecture, such as *labels*, macros, AL-specific instructions, and AL-specific addressing modes.

#### 5.1.1.1 Language Alphabet

*“There are 10 types of people in this world —  
those who understand binary, and those who don’t.”  
— J. Kincaid and P. Lewis*

---

<sup>1</sup>Issues related to implementation of the ISA (such as cache memory, ALUs, and the type of internal connections) do not form part of the ISA. The entire point of defining an ISA is to insulate the machine-level programmer (and the assembler) from those details.

A machine language alphabet has only two symbols:  $\{0, 1\}$ . This means that all of the information in a machine language program—instructions as well as data—have to be expressed using sequences of just 0s and 1s! The earliest programmers, who programmed computers at the ISA level, did precisely this. Thus, opcodes (such as `add` and `and`) and operands (such as `$t0` and `$sp`) are all specified in bit patterns rather than in a natural language-oriented symbolic form. A machine language program is therefore far less readable than an equivalent assembly language program. In addition, assembly languages permit programmers to use *labels* to identify specific memory addresses that hold data or form the target of branch instructions. Assembly languages also provide a variety of other convenience features that make programs at this level shorter and easier to write. For example, *data layout directives* allow a programmer to describe data in a more concise and natural manner than its binary representation. Similarly, *macros* and *pseudoinstructions* enable a sequence of instructions to be represented concisely by a single macro or pseudoinstruction.

Under these circumstances, why would anyone other than ISA designers want to study machine language? Certainly, not for writing application programs! The compelling reason is that besides application programs, there are a lot of support programs such as compilers, assemblers, disassemblers, and debuggers. The writers of these programs need to know the ISA details. Microarchitecture designers also need to know the machine language to design the hardware for executing machine language programs.

**Information Representation with Bit Patterns:** A machine language needs to represent two kinds of information: instructions and data. An assembly language had a large set of symbols—alphanumeric characters and punctuation marks—to represent information. By contrast, the alphabet used by a machine language for representing information has only 2 symbols—0 and 1. Thus, any kind of information can be represented at this abstraction level only by bit patterns obtained by concatenating bits. The same sequence of bits can have different meanings; the correct meaning depends on:

- the instruction being executed and
- the number systems and coding schemes adopted by the ISA designers

For example, consider the bit pattern 0100 1110 0111 0010 0000 0000 0000 0000 (i.e., 4E320000H). If this is interpreted as an integer, the value is  $2.106 \times 10^{10}$ . If it is interpreted as an ANSI/IEEE 754 floating-point number, the value is  $1.015 \times 10^9$ . If it is interpreted as part of an ASCII character string, then it indicates the 4 characters N, r, Null, Null. If it is interpreted as a Motorola 68000 instruction, then it indicates the STOP instruction if the system is in the Kernel mode, and a syscall instruction, otherwise. What this shows is that a bit pattern does not have an inherent meaning; we must know the context as well as the rules concerning the interpretation of those bits. The rules for interpreting a bit pattern are established at ISA design time, and will be effective throughout the life of the system. The assumptions about bit meaning will impact the design of the lower-level



hardware that manipulates the bits. The choice of a number system is based on available hardware technology, desired range of values, and other system goals.

The ISA specifies exactly how different instruction types and data types are encoded as bit patterns<sup>2</sup>.

*“A word aptly spoken is like apples of gold in settings of silver”*  
*— Proverbs 25: 11*

### 5.1.1.2 Syntax

A machine language program starts with a **header**, which provides a description of the various *sections* that are present in the program and are mapped to various portions of the memory address space. This description includes the name of the section, its size, its location within the program file, the portion of memory address space it maps to, etc. Table 5.1 lists the commonly found sections in machine language programs. Some of these sections—`.rdata`, `.text`, `.sdata`, and `.data`—are defined in the assembly language also, as we already saw in the previous two chapters. The rest—`.bss`, `.sbss`, `.lit8`, `.lit4`, and `.comment`—are limited to machine languages, and are created by the assemblers.

| Section Name          | Explanation                       |
|-----------------------|-----------------------------------|
| <code>.rdata</code>   | Read-only data                    |
| <code>.text</code>    | Code                              |
| <code>.bss</code>     | Uninitialized writable data       |
| <code>.sbss</code>    | Uninitialized writable small data |
| <code>.lit8</code>    | 64-bit floating-point constants   |
| <code>.lit4</code>    | 32-bit floating-point constants   |
| <code>.sdata</code>   | Writable small data               |
| <code>.data</code>    | Writable data                     |
| <code>.comment</code> | Comments                          |

Table 5.1: Typical Sections in a Machine Language Program

Unlike the case with assembly languages, each of the sections that are present appears exactly once. Within each section, the bit patterns corresponding to contiguous memory locations are stored contiguously. This is pictorially depicted in Figure 5.1. The order of the sections within the machine language program is not critical.

---

<sup>2</sup>Some of this encoding information—in particular, the encoding of data types—is known to the assembly language programmer too.

|        |        |       |       |       |       |        |       |          |
|--------|--------|-------|-------|-------|-------|--------|-------|----------|
| Header | .rdata | .text | .data | .lit8 | .lit4 | .sdata | .sbss | .comment |
|--------|--------|-------|-------|-------|-------|--------|-------|----------|

Figure 5.1: Organization of Header and Various Sections in a Machine Language Program

### 5.1.1.3 ELF Format

### 5.1.1.4 Portable Executable File Format

## 5.1.2 Register, Memory, and IO Models

To produce machine language code, machine language programmers and the developers of assemblers and disassemblers need to know the specifications of the instruction set architecture. These attributes include the memory model, the register model, the IO model, the data types, and the instruction types. The collection of all this information is what defines the instruction set architecture.

The register model supported by an ISA is the same as that supported by the assembly-level architecture for that computer. On a similar note, the ISA's memory model is also virtually the same as that supported by the assembly-level architecture. The only difference is that the ISA does not permit labels to be used to specify memory locations.

## 5.1.3 Data Types and Formats

A picture is worth 1,000 words, especially in memory requirements. And a video is worth a million words!

## 5.1.4 Instruction Types and Formats

Perhaps the biggest difference in the two architectures is in the area of the instruction set. The instruction set supported by the ISA is generally a subset of that supported by the assembly-level architecture. While the assembly-level instruction set is designed to facilitate assembly language programming, the machine-level instruction set is designed to facilitate lower-level implementations. Thus, there is a noticeable semantic gap between an assembly-level architecture and its corresponding ISA. This is especially the case for RISC (reduced instruction set computer) ISAs, as we will see later in this chapter.

In the case of the MIPS-I, the assembly-level instruction set has several opcodes that permit 32-bit immediate operands. Examples are `li` and `lw`. In order to encode these instructions using bit patterns, we need more than 32 bits. However, the MIPS-I ISA designers wanted fixed length instruction formats in which all instructions are encoded with 32 bits. Therefore, `li` is not included in the MIPS-I ISA and `lw` is allowed to have

only a 16-bit immediate operand. This begs the question of how we get a 32-bit data value into a register, at the ISA level. The standard method is to split the 32-bit data value into 2 parts—a 16-bit upper half and a 16-bit lower half—and use a `lui-ori` sequence to load the two halves.

Like the load instructions, the branch instructions are also limited to specifying a 16-bit immediate value at the ISA level. As 16 bits are not sufficient by themselves to specify the branch target address, the solution adopted is to consider the 16-bit immediate value as the instruction offset from the branch instruction. Thus, when executing the ISA-level instruction represented symbolically as

`beq $1, $2, 8`

8 instructions will be skipped if the contents of registers \$1 and \$2 are equal.

## 5.2 Example Instruction Set Architecture: MIPS-I

### 5.2.1 Register, Memory, and IO Models

### 5.2.2 Data Types and Formats

### 5.2.3 Instruction Types and Formats

The designers of the MIPS-I ISA decided to use a fixed length instruction format, so as to make it easier for the lower level machine (microarchitecture) to quickly determine instruction boundaries to facilitate the overlapped execution of multiple instructions. Every instruction is encoded in exactly 32 bits. In the MIPS-I ISA, the opcode implicitly specifies the addressing modes for each of the operands, and therefore no addressing mode information is explicitly recorded in the instruction bits. Considering all of the instructions in the MIPS-I ISA, we have the following fields: `opcode`, `rs`, `rt`, `rd`, `offset/immediate`, and `target`. Based on the fields used, we can classify the MIPS-I ISA instructions into 3 categories:

- Instructions that specify an `immediate/offset`: Examples are: (i) `addi rt, rs, immediate`, (ii) `lw rt, offset(rs)`, and (iii) `beq rs, rt, offset`. These instructions are called *I format* instructions.
- Instructions that specify 3 registers: An example is `add rd, rs, rt`. These instructions are called *R format* instructions.
- Instructions that specify a `target`. An example is `jal target`. These instructions are called *J format* instructions.

Among the different fields, the `rs`, `rt`, and `rd` fields require 5 bits each, in order to specify one of 32 general-purpose registers. Based on an analysis of existing programs, the MIPS-I

ISA designers decided to use 16 bits to specify the **offset/immediate** field. For the first category of instructions, this leaves 6 bits for the **opcode**. With 6-bit opcodes, the maximum number of opcodes that can be supported by the ISA is 64. Incidentally, the MIPS-I ISA has more than 64 opcodes. To accommodate more than 64 opcodes with a fixed opcode field, we need to either reduce the number of bits for the **offset/immediate** field, or use longer instruction formats, both of which have undesired consequences. The solution adopted by the MIPS-I ISA designers was to use *variable length opcodes* or *expanding opcodes* within a fixed-length instruction format. Instructions that cannot afford to support a longer **opcode** field—the ones in the first and third categories—are given a 6-bit **opcode** field. Instructions in the second category have to specify only 3 register addresses, requiring a total of 15 bits, besides the opcode. Therefore, these instructions are given a longer opcode field. For these instructions, the standard **opcode** field is set to the bit pattern 000000, and a separate 6-bit **func** field is used to specify the desired operation. The three main formats of the MIPS-I ISA are given in Figure 5.2.

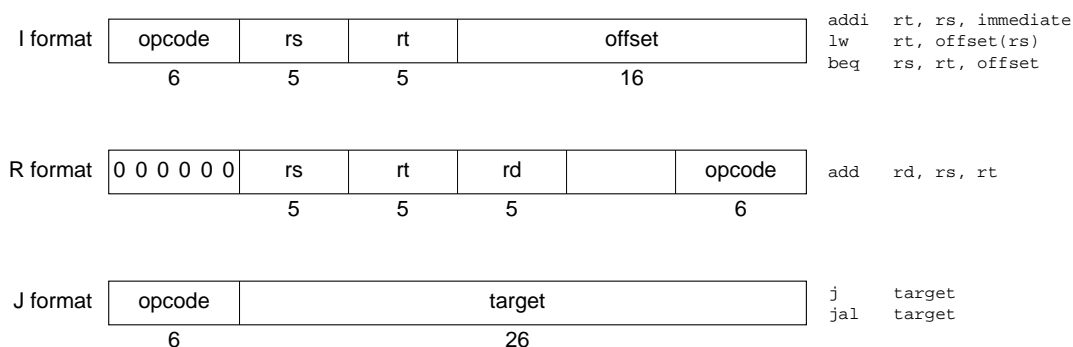


Figure 5.2: Instruction Formats Used in the MIPS-I ISA

#### 5.2.4 An Example MIPS-I ML Program

### 5.3 Translating Assembly Language Programs to Machine Language Programs

Programs written in an assembly language are translated into machine language programs by a special program called an **assembler**. An assembly language program is usually entered into the computer through a terminal and stored either in the main memory or on a magnetic disk. At this point, the program is simply a set of lines of alphanumeric characters, which are stored as patterns of 0s and 1s, most likely as per the ASCII chart. When the assembler program is executed, it reads this assembly language program, and generates the desired machine language program (*object code* file), which also consists of patterns of 0s and 1s (but in a tightly encoded form). The object code is a combination

of machine language instructions, data, and information related to placing the instructions properly in memory. It cannot be executed directly. Prior to execution, it undergoes the steps of linking and relocation, which transform it into a binary executable file.

Because the assembly language program is just a symbolic representation of a machine language program, on first appearance it might seem that an assembler is a simple software program that reads one statement at a time, and translates it to machine language. In fact, for the most part, there is a one-to-one correspondence between instructions in an assembly language program and the corresponding machine language program. The assembler's complexity arises from the additional features that are supported in the assembly-level architecture. For instance, most assembly languages support AL-specific instructions and macros. In addition, all assembly languages support the use of *labels*.

### 5.3.1 MIPS-I Assembler Conventions

Like the MIPS-I compiler, the MIPS-I assembler also follows some conventions. These conventions deal with the addresses allocated for different sections in memory, for instance. Again, these conventions are *not* part of the machine language specifications. Some of the important conventions are given below:

- The `.text` section starts at memory address `0x400000`, and grows in the direction of increasing memory addresses.
- The `.data` section starts at memory address `0x10000000`; it also grows in the direction of increasing memory addresses.
- The `stack` section starts at memory address `0x7fffffff`, but grows in the direction of decreasing memory addresses.

### 5.3.2 Translating Decimal Numbers

### 5.3.3 Translating AL-specific Instructions and Macros

Many assembly languages provide AL-specific instructions, which are not present in the corresponding ISA level. The availability of these additional instructions makes it eas-

ier to program for the assembly-level architecture. The assembler translates AL-specific instructions by using short sequences of instructions that are present in the ISA. For instance, the MIPS-I assembly-level architecture provides the opcode `li`, which is not present in the MIPS-I ISA. A MIPS-I sequence for translating the AL-specific instruction `li R1, 0x11001010` is given below.

```
lui    R1, 0x1100    # Enter upper 16 bits of address in R1; clear lower 16 bits
ori    R1, R1, 0x1010 # Bitwise OR the lower 16 bits of address to R1
```

A few other examples are given in Table 5.2. In the table, the AL-specific instruction `li` is synthesized with an `addiu` instruction or `lui-ori` instruction sequence, depending on whether the *immediate* value lies between  $\pm 32K$  or not. The AL-specific instruction `la` is also synthesized with a `lui-ori` pair. The AL-specific instruction `lw` has a slightly different format than the ISA-level `lw` instruction in that it uses memory direct addressing whereas the latter uses register-relative addressing. This conversion is also done by using the `lui` instruction to load the upper 16 bits of the memory address to a register, and then using the ML `lw` instruction with the lower 16 bits as the displacement. Notice that when the displacement value is negative, a 1 needs to be added to the upper 16 bits.

| MIPS-I AL Pseudoinstruction    | Equivalent MIPS-I AL Sequence                                  |
|--------------------------------|--|
| <code>li R2, -10</code>        | <code>addiu R2, R0, -10</code>                                 |
| <code>li R2, 0x8000</code>     | <code>ori R2, R0, 0x8000</code>                                |
| <code>li R2, 0x55ffff</code>   | <code>lui R2, 0x55</code><br><code>ori R2, R2, 0xffff</code>   |
| <code>la R2, 0x1000ffff</code> | <code>lui R2, 0x1000</code><br><code>ori R2, R2, 0xffff</code> |
| <code>lw R4, 0x10001010</code> | <code>lui R4, 0x1000</code><br><code>lw R4, 0x1010(R4)</code>  |
| <code>lw R5, 0x1000ffff</code> | <code>lui R5, 0x1001</code><br><code>lw R5, 0xffff(R5)</code>  |

Table 5.2: Some MIPS-I AL-specific Instructions and Their Equivalent Sequence

Translation of the macros can be done using text substitution in a similar manner. For example, the assembler can invoke the C preprocessor `cpp` to do the required text substitution prior to doing the assembly process.

### 5.3.4 Translating Labels

Another feature that assembly languages have incorporated for improving programmability is the use of labels. A label is a symbolic representation for a memory location that

corresponds to an instruction or data. For example, in the following code, the label `var1` corresponds to a memory address allocated for data, and the label `L1` corresponds to a memory address allocated for an instruction.

```

        .data
var1: .word 6           # Allocate a 4-byte item at memory location named var1
        .text
        beq    R1, R2, L1    # Branch to label L1 if the contents of R1 and R2 are same
        ori    R1, R1, 0x1010
L1:     lw      R2, var1      # Load from memory location named var1 into R2

```

Translating a label involves substituting all occurrences of the label with the corresponding memory address. Although this is a straightforward substitution process, the situation is complicated by the fact that most assembly languages permit a label to be used prior to declaring it. This is illustrated in the above example code, where label `L1` is used by the `beq` instruction before its declaration two instructions later. Thus, the assembler cannot perform the substitution for `var1` when it first encounters the `beq` instruction. In order to translate such labels, the assembler needs to go through a program in two passes. The first pass scans the assembly language program for symbol declarations, and builds a *symbol table*, which contains the addresses associated with each symbol. The second pass scans the program, and substitutes each occurrence of a symbol with the corresponding address. This pass also translates each instruction to machine language, and generates the machine language program, possibly along with the assembly listing.

#### 5.3.4.1 First Pass: Creating the Symbol Table

The purpose of the symbol table is to store the mapping between the symbolic names (labels) and their memory addresses. As noted earlier, the objective of the first pass is to scan the assembly language program, and expand the AL-specific instructions as well as build the symbol table. For each symbol defined in a module, a memory address is assigned, and an entry is created in the symbol table. The symbol table is a data structure, on which two operations can be performed: insertion and search. The first pass of the assembler performs only insertions. An efficient organization of the symbol table is important for fast assembly, as the symbol table may be referenced a large number of times during the second pass.

#### 5.3.4.2 Second Pass: Substituting Symbols with Addresses

In the second pass through the program, the assembler substitutes symbols with the respective addresses stored in the symbol table created in the first pass. For each symbol encountered, it looks up in the symbol table to obtain the address. Undefined symbols, if any, are marked as external references. A list of unresolved references is made available at the end of this pass.

### 5.3.5 Code Generation

Code generation involves translating each instruction to its corresponding bit pattern as per the ISA definition. This step can be done in the second pass, and mostly involves table lookups.

#### 5.3.5.1 Example Object File Format

To clarify the assembly process, we shall use a detailed example. For convenience, we shall use the assembly language code that we saw earlier in pages 112 and 123. This code is reproduced below. For the global variables, we shall use the same memory allocation given in Figure 3.5. In the code given below, the left hand side shows the assembly language code. The right hand side shows the memory map of the translated machine language code, along with deassembled code (in assembly language format). This example shows how AL-specific instructions (such as `la` and `bge`) are translated using 2-instruction sequences. The example also shows how various symbols (variable names as well as labels) are translated. After the translation, references to variables are specified using the appropriate memory addresses. Notice that in the MIPS-I ISA, a branch instruction specifies a branch offset instead of an absolute target address.

---

|         |                  |  |                      |
|---------|------------------|--|----------------------|
|         |                  | # Assign memory locations for the global variables |                      |
|         |                  | # Initialize memory locations if necessary         |                      |
|         | .data            |  |                      |
| a:      | .word 0          | 0x10001000:  | 0x0                  |
| b:      | .word 12         | 0x10001004:  | 0xc                  |
| record: | .word 0:3        | 0x10001008:  | 0x0                  |
|         |                  | 0x1000100c:  | 0x0                  |
|         |                  | 0x10001010:  | 0x0                  |
| f:      | .word            | 0x10001014:  |                      |
| cptr:   | .word            | 0x10001018:  |                      |
|         | .text            |  |                      |
|         | .align 2         |  |                      |
| la      | R1, record       | 0x400000:  | # lui R1, 0x1000     |
|         |                  | 0x400004:  | # ori R1, R1, 0x1008 |
| li      | R2, 0            | 0x400008:  | # add R2, R0, R0     |
| li      | R3, 0            | 0x40000c:  | # add R3, R0, R0     |
| lw      | R4, 8(R1)        | 0x400010:  | # lw R4, 8(R1)       |
| loop:   | bge R3, R4, done | 0x400014:  | # slt R6, R3, R4     |
|         |                  | 0x400018:  | # beq R6, R0, 6      |
|         |                  | 0x40001c:  | # lw R5, 0(R1)       |
|         | lw R5, 0(R1)     | 0x400020:  | # add R2, R2, R5     |
|         | add R2, R2, R5   | 0x400024:  | # addi R1, R1, 4     |
|         | addi R1, R1, 4   | 0x400028:  | # addi R3, R3, 1     |
|         | addi R3, R3, 1   | 0x40002c:  | # beq R0, R0, -6     |
|         | b loop           |  |                      |



```

done:  sw      R3, a           0x400030:      # lui   R6, 0x1000
      lw      R4, cptr       0x400034:      # sw    R3, 0x1000(R6)
      sw      R2, 0(R4)      0x400038:      # lw    R4, 0x1018(R6)
      sw      R2, 0(R4)      0x40003c:      # sw    R2, 0(R4)

```

---

One important aspect to note is that the target of the assembled code is the machine language seen by the linker.

### 5.3.6 Overview of an Assembler

#### Summary

| AL Program        | ML Program  |
|-------------------|---|
| Macro             | Expanded (by a pre-processor)                               |
| Directive         | Do not appear explicitly                                    |
| Comment           | Ignored, or placed in the <code>.comments</code> section    |
| Label declaration | Assigned memory address(es) in the appropriate section      |
| Label reference   | Substituted by the address assigned to the label            |
| Instruction       | Synthesized by an instruction or a sequence of instructions |

Table 5.3: Translation of AL features to ML

### 5.3.7 Cross Assemblers

We just saw how an assembler translates assembly language programs to machine language. As the assembler is implemented in software, it has to be executed on a host computer for it to perform the translation. Normally when we execute an assembler on a particular host computer, the assembler translates the program to the machine language of the host computer itself. We can say that the assembler's *target ML* is the same as that of the *host ML*. Sometimes we may want to translate a program to an ML that is different from the host ML. A *cross assembler* is used to do such a translation. There are several reasons why such a cross assembly is required: (i) The target computer may not have enough memory to run the assembler. This is often the case in embedded systems. (ii) When a new ML is being developed, no host computer exists for that ML. To analyze different trade-offs, the designer often builds a software simulator for the target computer. In order to “execute” programs on the simulator, programs need to be translated to the new ML on a different host computer.

## 5.4 Linking

Many computer programs are very large, and may even have millions of lines of instructions and data. In order to make it easier to develop such large programs, a single program is often developed as a collection of *modules* (i.e., a logically related collection of subroutines). Different modules may even be developed by different programmers, possibly at different times. For instance, the library routines are written as a common set of routines before the development of application programs, and rarely undergo changes after development. Several programs may even share modules other than the libraries.

When an assembly language program is developed as a collection of several modules, how should it be assembled? If all of the source modules are combined into a single AL program, and then the assembly is performed, that entails re-assembling the entire AL program every time a change is made to one of the modules. This might require a long time to perform the assembly, besides wasting computing resources. An alternative is to assemble each module independently of other modules, and store the translated output as an object module file (on the disk). A change in a single source module would then require re-assembling only that module. On Microsoft systems such as DOS, Windows 95/98, and Windows NT, an object module is named with a `.obj` extension, and an executable ML program is given the extension `.exe`. On UNIX systems, object files have extension `.o`, whereas executable ML programs have no extension.

An object module cannot be directly executed as a ML program, because it is likely to contain *unresolved references*. An object module contains an unresolved reference to a label, if its source module does not contain a definition for that label. For instance, the module may have a call instruction to label `foo`, and `foo` may be defined in another module. In order to generate an executable ML program, we need to combine all of the component object modules into a single program. This process of combining — called *linking* — is customarily done using a program called *link editor* or *linker*. The linking process has to be repeated whenever one or more of the component modules is re-assembled.

The linker performs four important tasks:

- Resolve references among multiple object modules, i.e., patch the external references.
- Search the program libraries to find library routines used by the program.
- Determine the memory locations that will be occupied by the instructions and data belonging to each module, and relocate its instructions by adjusting absolute references.
- Append a *start-up routine* at the beginning of the program.

### 5.4.1 Resolving External References

A linker's first task is to resolve external references in the object modules. The linker does this by matching the unresolved references of each object module and the external symbols of every other object module. Information about external symbols and unresolved references is generated in the second pass of the assembly process, and stored as part of the object module. An external symbol in one object module resolves a reference from another object module if both have the same name. An unmatched reference means that a symbol was used, but not defined in any of the object modules. Unresolved references at this stage in the linking process do not necessarily mean that there is a bug in the program. The program could have referenced a library routine (such as `printf`) whose code was not in the object modules passed to the linker.

After attempting to resolve external references in the object modules, the linker searches the system library archives such as `/lib/libc.a` to find predefined subroutines referenced by the object modules. When the program uses a library routine, the linker extracts the routine's code from the library and incorporates it into the program text section. This new routine, in turn, may call other library routines, so the linker continues to fetch other library routines until no external references remain to be resolved, or a routine cannot be found. A program that references an unresolved symbol that is not in any library is erroneous and cannot be linked.

### 5.4.2 Relocating the Memory Addresses

If all external references are resolved, then the linker determines the memory locations that each module will occupy. Because the modules were assembled in isolation, the assembler did not know where a module's instructions and data will be placed relative to other modules. When the linker places a module in memory, all absolute references must be *relocated* to reflect its true location. Because the linker is provided with relocation information that identifies all relocatable references, it can efficiently find and backpatch these references.

### 5.4.3 Program Start-Up Routine

Finally, the linker appends a *start-up routine* at the beginning of the program. The start-up routine performs several book-keeping functions. One of the book-keeping functions performed by MIPS-I start-up routines involves copying from the stack to the argument registers (`$a0-$a3`) the first four parameters (to be passed to the function `main`). After performing the required initializing jobs, the start-up code calls the main routine of the program. When the main routine completes execution, control returns to the start-up routine, which then terminates the program with an `exit` system call.

---

`.text`

```

        .globl  _start
_start:
        lw      $a0, 0($sp)      # Program execution begins here
        addiu   $a1, $sp, 4      # Copy argc from stack to R4
        addiu   $a2, $a1, 4      # Place stack address corresponding to argv in R5
        addiu   $a2, $a1, 4      # Start calculating stack address of envp
        mul     $v0, $a0, 2      # Stack addr of envp = stack addr of argv + argc × 4 + 4
        addu    $a2, $a2, $v0    # Place stack address corresponding to envp in R6
        jal     main             # Call subroutine main
        ori     $v0, $0, exit_code # Place code for exit system call in R2
        syscall                               # Call OS to terminate the program

main: add     $a1, $sp, 4      # Place the address of c in $a1
      li      $a2, 1          # Place the number of bytes to be read (i.e., 1) in $a2
loop: li      $a0, 0          # Place the file descriptor (0) in $a0
      li      $v0, read_code   # Place the code for read in $v0
      syscall                               # Call OS routine to perform the read

      blez    $v0, done       # break out of while loop if syscall returned zero
      li      $a0, 1          # Place the file descriptor (1) in R4
      li      $v0, write_code  # Place the code for write in $v0
      syscall                               # Call OS routine to perform the write
      b       loop            # go back to while loop

done: jr      $ra              # return from subroutine main to start-up code

```

---

The linker produces an executable file that can directly run on a computer system (that has an appropriate *loader*). Typically, this file has the same format as an object file, except that it contains no unresolved references. Its target is the instruction set architecture seen by the loader. Notice that, unlike the assembly process, the linking process does not represent a change of level in the virtual machine abstraction tower, because the linker's input and output are programs for the same virtual machine, and use the same language. Some linkers also produce an ASCII representation of the symbol table. This can be used during debugging for displaying the addresses of symbols used in both object and executable files.

## 5.5 Instruction Formats: Design Choices

Like an assembly language program, a machine language program also consists of a sequence of instructions, each one instructing the machine to do a specific operation. However, the machine language alphabet has only two symbols, {0, 1}. Therefore *bit patterns* are the sole means of specifying each instruction. For instance, the MIPS-I AL instruction

add \$v0, \$a1, \$a2

is represented in machine language as:

00000000100001010001000000100000

Never mind how we get this bit pattern—we will show that a little later. This bit pattern encodes all of the information needed to correctly interpret the instruction, including the opcode and all 3 explicit operands. In the general case, the encoded information includes:

- opcode — operation to be performed
- address/value of each explicit operand
- addressing mode for each operand (optional)

The addressing mode part is treated as optional because the opcode often implies the addressing mode; i.e., the addressing mode of the operand is unambiguous, and need not be explicitly specified. How about labels and comments? Are they also encoded in the machine language? Consider the same `add` instruction, with a label and comment added as shown below:

```
done:    add $v0, $a1, $a2      # calculate return value
```

It turns out that the encoding for this AL instruction is same as that given above for the instruction without the label and the comment! The label field is indeed translated to a memory address, but this address is not encoded along with the instruction; instead, arrangements are made to ensure that the encoded ML instruction will be placed in that memory address when the program is loaded. The comment field is ignored altogether, and is not represented in machine language programs.

A key issue to consider in deciding an encoding for the instructions is the number of bits to be used for encoding each instruction type. If too many bits are used to encode an instruction, then the code size increases unnecessarily, resulting in increased memory storage requirement<sup>3</sup> as well as increased instruction fetch bandwidth requirements. If too few bits are used, it might become difficult to add new instructions to the ISA in the future. Some ISAs use a fixed number of bits to encode all of the instructions in the ISA, and the rest use a variable number of bits to encode all of the instructions. Newer ISAs such as MIPS-I, SPARC, Alpha, and PowerPC use fixed length instruction encoding, whereas older ISAs such as IA-32 and VAX use variable length instruction encoding. There are good reasons why the older ISAs went for variable length instructions. When computers were first introduced, memory address space was at a premium, and it was important to reduce the size of programs. The use of variable length instructions, coupled with an encoding technique called Huffman encoding<sup>4</sup>, enabled these ISAs to promote the development of machine language programs that required the minimum amount of memory.

When an ISA design team has to choose instruction formats for a new ISA, a number of factors must be considered. If a particular computer becomes commercially successful,

---

<sup>3</sup>Memory storage requirement was a major concern when memory was expensive. With memory prices falling steadily, this has become less of a concern, especially for general-purpose computers. In the embedded computing world, it is still a concern.

<sup>4</sup>In this encoding, the most frequent instructions are encoded with the fewest bits and the least frequent ones are encoded with the maximum number of bits. The words in the English language follow such an encoding to a large extent.

its ISA may survive for 20 years or more, like the Intel IA-32. The ability to add new instructions and exploit other opportunities that arise over an extended period is of great importance, if the ISA survives long enough for it to be a success.

**Field-based Encoding:** Apart from the number of bits used for instruction encoding, there is another major difference in how information in the different fields are encoded. One approach starts by listing all of the unique combinations of opcodes, operand specifiers, and addressing modes. Once this listing is done, a unique binary value is assigned to each unique combination. Such an encoding is called **total encoding**. An alternate approach is to encode each field—the opcodes, the operand specifiers, and the addressing modes—separately. This approach is called *field-based encoding*. Such an encoding makes it easier for the lower level machine to decode instructions during execution. A related feature that makes it even easier to decode instructions is **fixed field encoding**. In such an encoding, a particular set of bits are devoted for specifying a particular part of an instruction, such as its opcode or operands. This type of encoding helps to determine the operands or their addresses even before figuring out the opcode of the instruction.

### 5.5.1 Fixed Length Instruction Encoding

In this type of encoding, all instructions are encoded with the same number of bits, and therefore have the same length. Using a fixed number of bits for instruction encoding makes it easier to determine where each instruction starts and ends. Although this might waste some space because all instructions are as long as the longest one, it makes instruction decoding easier for the lower-level microarchitecture, which interprets ML instructions. The customary method is to divide the bits in an instruction word into groups called *fields*. Consider the instruction `add rd, rs, rt`. When the assembler converts this assembly-language instruction into a machine-language instruction, it produces a bit pattern that encodes the information contained in `add rd, rs, rt`. Consider the format given in Figure 5.3.

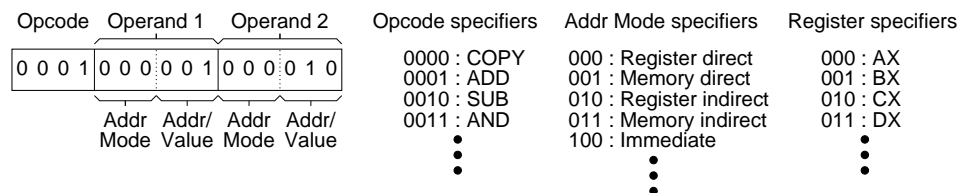


Figure 5.3: A Sample 16-bit Format to Represent the Instruction `add rd, rs, rt`

In this format, 4 bits are allocated to specify the opcode, thereby allowing up to 16 unique opcodes in the instruction set. The opcode `ADD` is encoded by the bit pattern 0001. The register names are also given specific 3-bit patterns. For instance, the bit pattern 001

has been used to represent **BX**, and the bit pattern 010 has been used to represent **CX**. Three bits are used to specify an addressing mode. For instance, the bit pattern 000 is used to denote register direct addressing.

Example: What is the bit pattern for **ADD AX, BX** in the sample encoding of Figure 5.3?

The bit pattern for the opcode, **ADD**, is 0001. The bit patterns for registers **AX** and **BX** are 000 and 001, respectively. The bit pattern for register direct addressing is 000. Therefore, the bit pattern corresponding to the instruction **ADD AX, BX** is 0001000000000001.

What would be the bit pattern for **COPY AX, M[32]**? We can see that it is not possible to represent the bit pattern for memory address 32 in the 3 bits allotted for specifying an address. To accommodate long addresses in a fixed length format, instruction sets use a technique called *expanding opcodes* or variable length opcodes.

Let us illustrate the idea of expanding opcodes using the instruction encoding done in PDP-8, a popular single-address machine of the 1960s. It was a 12-bit machine made by DEC. Of the 12 bits allotted to an instruction, 3 bits were used for specifying the *opcode*. The PDP-8 used 3 different instruction formats, as shown in Figure 5.4. Let us calculate the maximum number of unique opcodes that PDP-8 can have, and the maximum number of memory locations that it can specify directly.

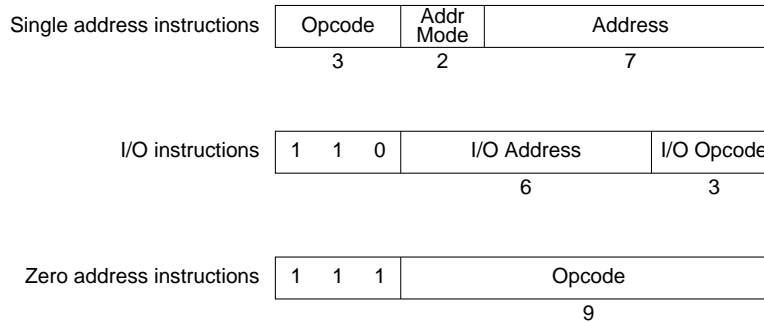


Figure 5.4: Instruction Formats Used in the PDP-8 ISA

Consider the first format, namely the one for single-address instructions. 3 bits have been allocated for encoding the opcode, and therefore,  $2^3 = 8$  unique bit patterns are possible. Out of the 8, two are used for identifying IO instructions and zero-address instructions. Thus, a maximum of 6 single-address instruction opcodes are possible. In the IO instruction format, 3 bits are allocated for encoding the opcode, and therefore, a maximum of 8 IO instruction opcodes are possible. In the zero-address format, 9 bits are allocated for the opcode, allowing a maximum of  $2^9 = 512$  unique opcodes. Thus, a total of  $6 + 8 + 512 = 526$  unique opcodes are possible for the PDP-8.

The maximum number of bits allocated for directly encoding memory addresses in any of the three formats is 7. Thus, a maximum of  $2^7 = 128$  memory locations can be specified.

### 5.5.2 Variable Length Instruction Encoding

Some instruction sets have a large number of instructions, with many different addressing modes. With such instruction sets, it may be impractical to encode all of the instructions within a fixed length of reasonable size. To encode such an instruction set, many instruction sets use *variable length instructions*.

In the IA-32 ISA, instruction lengths may vary from 1 byte up to 17 bytes. Figure 5.\* shows the instruction formats for some of the commonly used IA-32 instructions. The 1 byte format is typically used for instructions that have no opcodes and those that involve specifying a single register operand. The opcode byte usually contains a bit indicating if the operands are 8 bits or 32 bits. For some opcodes, the addressing mode and the register are fixed, and therefore need not be explicitly specified. This is especially the case for instructions of the form `opcode register, immediate`. Other instructions use a *post-byte* or extra opcode byte, labeled `mod, reg, r/m`, which contains the addressing mode information. This is the format for many of the instructions that specify a memory operand. The base plus scaled index mode uses a second post-byte labeled `sc, index, base`.

## 5.6 Data Formats: Design Choices and Standards

We have seen how instructions are represented in a machine language using bit patterns. Next, we will see how data values are represented in an ISA. The encoding used to represent data values has a significant effect on the complexity of the lower-level hardware that performs various arithmetic/logical operations on the values. The hardware for performing an arithmetic operation on two numbers can be simple or complex, depending on the representation chosen for the numbers! Therefore, it is important to choose representations that enable commonly used arithmetic/logical operations to be performed in a speedy manner.

When using  $N$  bits to represent numbers,  $2^N$  distinct bit patterns are possible. The following table summarizes the number of representable values for popular word sizes.

Machine language programs often need to represent different kinds of information—instructions, integers, floating-point numbers, and characters. All of these are represented by bit patterns. An important aspect regarding the typical usage of bit patterns is that:

**Bit patterns have no inherent meaning**

Stating this differently, a particular bit pattern may represent different information in different contexts. Thus, the same bit pattern may be the representation for an integer as well as a floating-point number. We shall illustrate this concept with an example from the English language. The word *pen* has several meanings in English: (i) a small enclosure for animals, (ii) an instrument used for writing, or (iii) to write. Depending on the context,



| Word Size<br>(in bits) | Number of<br>Representable Values | ISAs/Machines                            |
|------------------------|-----------------------------------|--|
| 4                      | 16                                | Intel 4004                               |
| 8                      | 256                               | Intel 8080, Motorola 6800                |
| 16                     | 65,536                            | DEC PDP 11, Intel 8086, Motorola?? 32020 |
| 32                     | $4.29 \times 10^9$                | IBM 370, Motorola 68020, VAX 11/780      |
| 48                     | $1.41 \times 10^{14}$             | Unisys                                   |
| 64                     | $1.84 \times 10^{19}$             | Cray, DEC Alpha                          |

Table 5.4: Number of Representable Values for Different Word Sizes

we are able to figure out the intended meaning of the word *pen*. Similarly, based on the context, the computer correctly figures out the information type and the intended meaning of a bit pattern. It is important to note that the information type could also be encoded in the bit pattern using additional bits, but that is rarely done.

*“Time flies like an arrow, but fruit flies like an orange.”*

### 5.6.1 Unsigned Integers: Binary Number System

First, consider the encoding of unsigned integers, which only have a magnitude and no sign. The standard encoding used to represent unsigned integers at the ISA level is to use the *binary number system*. This is a positional number system, and the value of an  $N$ -bit pattern,  $b_{N-1}b_{N-2}...b_1b_0$ , if interpreted as an unsigned integer, is given by

$$V_{unsigned} = b_{N-1} \times 2^{N-1} + b_{N-2} \times 2^{N-2} + b_{N-3} \times 2^{N-3} + ..... + b_0 \times 2^0$$

Example: The value of bit pattern 10001101, if interpreted as an unsigned binary integer, is given by

$$2^7 + 2^3 + 2^2 + 2^0 = 128 + 8 + 4 + 1 = 141.$$

To convert the representation of an  $N$ -bit unsigned integer to a  $2N$ -bit number system, all that needs to be done is to append  $N$  zeroes to the left of the most significant bit (MSB).

Range of an  $N$ -bit Unsigned Number System:

$$0 \rightarrow 2^N - 1$$

Figure 5.5 depicts this range for a 32-bit number system.



Figure 5.5: Range of Unsigned Integers Represented in a 32-bit Binary Number System

### 5.6.2 Signed Integers: 2's Complement Number System

The most widely used number system for representing signed integers at the ISA level is the *2's complement number system*. In the 2's complement number system, positive numbers are represented in their binary form as before, but negative numbers are represented in 2's complement form. The 2's complement number system is also a positional number system, and the value of a bit pattern,  $b_{N-1}b_{N-2}...b_1b_0$ , if interpreted as a signed integer, is given by

$$V_{signed} = -b_{N-1} \times 2^{N-1} + b_{N-2} \times 2^{N-2} + b_{N-3} \times 2^{N-3} + ..... + b_0 \times 2^0$$

This expression is similar to the one for unsigned integers, except for the negative sign in front of the  $b_{N-1}$  term.

To convert the representation of an  $N$ -bit signed integer to a  $2N$ -bit number system, all that needs to be done is to append  $N$  copies of the sign bit to the left of the MSB.

*Example:* The integer  $-12$  is represented in the 8-bit 2's complement number system as 11110100. Its representation in the 16-bit 2's complement number system is 1111111111110100, obtained by appending 8 copies of the sign bit (1) to the left of the most significant bit.

*Range of an  $N$ -bit 2's Complement Number System:*

$$\boxed{-2^{N-1} \rightarrow 2^{N-1} - 1}$$

Figure 5.6 depicts this range for a 32-bit number system. Roughly half of the range is on the positive side and the other half is on the negative side. On comparing this range with the one given earlier for unsigned integers, we can see that the upper half of the range of unsigned integers have been replaced by negative integers.

Table 5.5 gives the bit pattern for some positive and negative numbers in the 8-bit 2's complement number system.

The main advantages of using the 2's complement number system are:

- Only one representation for zero—allows an extra (negative) number to be represented.

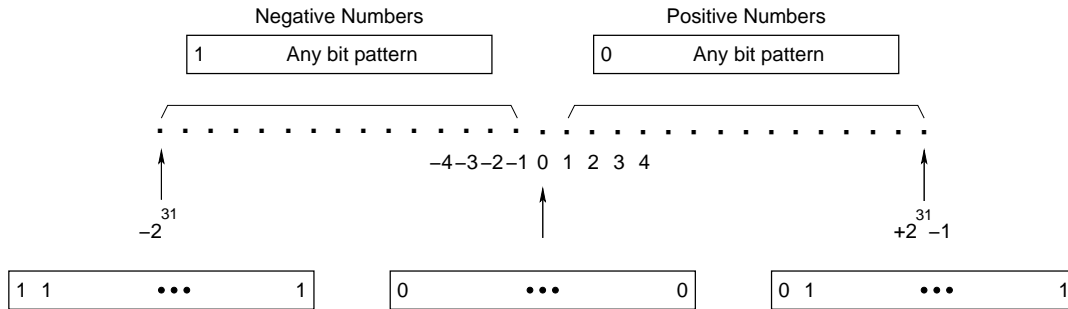


Figure 5.6: Range of Signed Integers Represented in a 32-bit 2's Complement Number System

| Bit pattern | Decimal value | Remarks                        |
|-------------|---------------|--------------------------------|
| 10000000    | -128          | Unique representation for zero |
| 10000001    | -127          |                                |
| 11111111    | -1            |                                |
| 00000000    | 0             |                                |
| 00000001    | 1             |                                |
| 00000010    | 2             | Largest representable integer  |
| 01111110    | 126           |                                |
| 01111111    | 127           |                                |

Table 5.5: Decimal Equivalents of 8-bit Patterns in 2's Complement Number System

- The same adder can be used for adding unsigned integers as well as signed integers, thereby simplifying the design of the ALU at the digital logic level.

For instance, if two signed integers (e.g., 45 and -62), expressed in the Sign-Magnitude number system, are added as if they are unsigned integers, then the result will be incorrect (-107), as shown below. On the other hand, if the same two numbers are expressed in the 2's complement number system and added as unsigned integers, the result (-17) will be correct.

### 5.6.3 Floating Point Numbers: ANSI/IEEE Floating Point Standard

Our next objective is to come up with suitable representations for floating-point numbers. We already saw in Section 3.1.4 that a floating-point number has 4 parts: the sign, the significand, the base, and the exponent. Among these, the base is fixed for a given ISA,

| Sign-Magnitude System        | 2's Complement System       |
|------------------------------|-----------------------------|
| 0 0 1 0 1 1 0 1 (45)         | 0 0 1 0 1 1 0 1 (45)        |
| 1 0 1 1 1 1 1 0 (-62)        | 1 1 0 0 0 0 1 0 (-62)       |
| <hr/> 1 1 1 0 1 0 1 1 (-107) | <hr/> 1 1 1 0 1 1 1 1 (-17) |
| ↑                            | ↑                           |
| Incorrect                    | Correct                     |

Figure 5.7: Adding Two Signed Integers in Sign-Magnitude Number System and 2's Complement Number System

and need not be explicitly encoded or represented in the bit pattern for a floating-point number. The remaining parts (the sign, the significand, and the exponent) need to be explicitly stored in the bit pattern. In designing a format for floating-point numbers, the obvious choice is to use *field-based encoding*, i.e., partition the available bit positions into fields, and pack the different parts of the FP number into different fields. Thus the items to be decided at ISA design time are:

- the base
- the signing convention for the significand and the exponent
- the number of bits for specifying the significand and the exponent
- the ordering of the sign, significand, and exponent fields

Until about 1980, almost every ISA used a unique FP format. There was no consensus for even the base; different powers of 2 such as 2, 4, 8, or 16 have been used as the base. The lack of a standard made it difficult to exchange FP data among different computers. Worse yet, some computers occasionally did floating-point arithmetic incorrectly because of some subtleties in floating-point arithmetic. To rectify this situation, the IEEE set up a committee in the late 1970s to standardize FP representation and arithmetic. The goal was not only to permit FP data to be exchanged among different computers but also to provide hardware designers with a model known to be correct. The resulting work led to IEEE Standard 754. Almost all of the present day ISAs (including the IA-32, Alpha, SPARC, and JVM) use the IEEE FP standard and have FP instructions that conform to this standard.

The IEEE standard defines three formats: single precision (32 bits), double precision (64 bits), and extended precision (80 bits). The extended precision format is intended to reduce roundoff errors while performing arithmetic operations. It is primarily used inside FP arithmetic units, and so we will not discuss it further. Both the single- and double-precision formats use base 2, and excess code for exponents. These two formats are shown below.

“The most valuable of all talents is that of never using two words when one will do”  
Thomas Jefferson

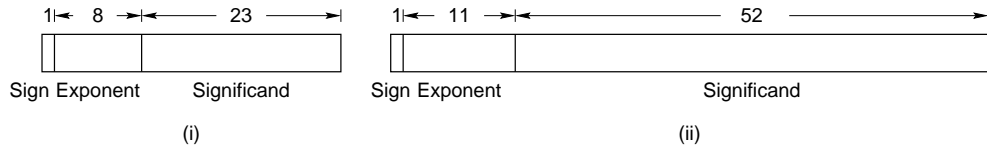


Figure 5.8: IEEE Floating-Point Formats: (i) Single Precision (ii) Double Precision

#### Single-Precision:

- Base of the FP number system is 2
- Number of bits allotted for representing the significand is 23
- Significand's precision,  $m$  is 24 (because the MSB is not explicitly stored)
- Number of bits allotted for representing the exponent,  $e$  is 8
- Format of the exponent: excess 127 code

#### Double-Precision:

- Base of the FP number system is 2
- Number of bits allotted for representing the significand is 52
- Significand's precision,  $m$  is 53 (because the MSB is not explicitly stored)
- Number of bits allotted for representing the exponent,  $e$  is 11
- Format of the exponent: excess 1023 code

Notice that the value used for the base is not encoded in the bit pattern, but forms part of the definition of the number system. In the single-precision format, the exponent is represented in 8 bits. With 8 bits, the exponent can take values ranging from  $-128$  to  $127$ . Of these, the most negative value ( $-128$ ) is treated as special; when the exponent is  $-128$ , the significand part is used to represent special numbers such as  $+\infty$ ,  $-\infty$ , and NaN (Not a Number). Because of this arrangement, the most negative exponent allowed in this FP format is  $-127$ .

### Normalization of Significand

When using the floating-point notation, a number can be written in a myriad ways. For example,  $0.00101_2 \times 2^3$ ,  $1.01_2 \times 2^0$ , and  $10100_2 \times 2^{-4}$  all denote  $1.01_2$ . To reduce this profusion of equivalent forms, a specific form is defined to give every FP number a unique representation. This process is called normalization. Normalization fixes the position of the radix point such that it immediately follows the *most significant non-zero bit* of the significand. For example, the normalized form of  $0.00101_2 \times 2^3$  is  $1.01_2 \times 2^0$ . The normalized significand is a fraction with value between 1 and almost 2.

Specifically,  $\text{Significand}_{\text{Max}} = 1.111111\dots_2 = 2_{10}$  (approx)  
 $\text{Significand}_{\text{Min}} = 1.000000\dots_2 = 1_{10}$

With non-zero binary numbers, the most significant non-zero bit of the significand is always a 1, and therefore need not be explicitly stored. By not explicitly storing this bit, an extra bit of precision can be provided for the significand. Thus, the single-precision format effectively represents 24-bit significands, which provides approximately the same precision as a 7-digit decimal value. This technique of not explicitly representing the most significant bit is often called the **hidden bit technique**. With this technique, the significand 1.011, for instance, will be represented as follows, with the leading 1 and the radix point omitted.

|  |                          |
|--|--------------------------|
|  | 011000000000000000000000 |
|--|--------------------------|

What about the number zero? Can it be expressed in normalized form? It turns out that the number zero cannot be represented in the normalized form, because its significand does not contain any 1s! The closest representable number is  $1.0_2 \times 2^{-127}$ , which has the smallest normalized significand and the smallest exponent. A similar representation problem arises when attempting to represent FP numbers that have magnitudes smaller than  $1.0_2 \times 2^{-127}$ , the smallest normalized FP number that can be represented with the allowed precision. To provide good precision for these small numbers and to represent zero, IEEE 754 allows these numbers to be represented in *unnormalized* form. Thus, when the exponent value is  $-127$ , the significand part is considered to have a hidden 0 (instead of a hidden 1) along with a radix point. This makes it possible to represent numbers with very small magnitudes and the number zero, at the expense of not representing some numbers such as  $2^{-127}$ . For example, the number zero will be represented as follows:

|                        |                          |
|------------------------|--------------------------|
| 0 Bit pattern for -127 | 000000000000000000000000 |
|------------------------|--------------------------|

## Biasing of Exponent

The designers of the IEEE 754 standard also considered several hardware issues when coming up with this standard. In particular, they wanted an integer comparator to be able to compare two floating-point numbers also. This is one of the motivating reasons for placing the sign bit in the most significant position, and the exponent before the significand. Placing the sign bit at the most significant position ensures that the bit pattern of a negative FP number will seem smaller than that of a positive FP number, if both patterns are interpreted as signed integers (in the 2's complement number system). When using normalized significands, for FP numbers with positive exponents, the bit patterns of numbers with smaller exponents are guaranteed to appear smaller than numbers with bigger exponents.

A problem occurs, however, with negative exponents when exponents are represented in the 2's complement number system or any other number system in which negative integers

have a 1 in the MSB. This is because an FP number with a negative exponent will look like a very big binary number. For example, the number  $1.01_2 \times 2^{-1}$ , which has a negative exponent, would be represented as

0 11111111 010000000000000000000000

The number  $1.01_2 \times 2^1$ , which has a positive exponent, would be represented as

0 00000001 010000000000000000000000

If these two bit patterns are compared using an integer comparator, the latter pattern's value will be considered to be smaller than that of the former!

The desired notation for FP numbers must therefore represent the most negative exponent as  $00000000_2$  and the most positive exponent as  $11111111_2$ . This can be achieved by using the *excess 127 code* for representing the exponents so that the most negative exponent ( $-127$ ) can be represented as the bit pattern  $00000000$ . In an excess  $E$  code system, before representing an integer  $I$ , the excess value of  $E$  is purposely added to  $I$  such that the integer value of the bit pattern stored is given by

$$S = I + E$$

With the use of the excess 127 code for the exponents, the representations for the above two numbers ( $1.01_2 \times 2^{-1}$  and  $1.01_2 \times 2^1$ ) become

0 01111110 010000000000000000000000

and

0 10000000 010000000000000000000000

respectively, in which case an integer comparator is sufficient to compare them. Expressing the exponent in excess code thus allows simpler hardware to compare two floating-point numbers. At the same time, the excess code system retains the benefits of the 2's complement number system in that it can handle negative exponents as gracefully as positive exponents. Notice that the range of representable exponents is still  $-127 \rightarrow 127$ . Thus, the excess code system neither allows more exponents to be represented nor changes the range of representable exponents. It only changes the bit patterns of the exponents that were representable before the excess was applied.

In order to place the different features of the IEEE 754 standard in the proper context and to see how they all fit together, let us look at an example.

Example: What is the decimal equivalent of the following bit pattern if interpreted as an IEEE single precision floating point number?

0 10000110 001100000000000000000000

- Step 1: Extract the 8-bit exponent field (10000110) and subtract 127 from it. One way to do is to add the bit pattern 10000001, which is the 2's complement of the bit pattern for 127. The resulting bit pattern gives the exponent in 8-bit 2's complement number system. Find the decimal equivalent of this exponent.
- Step 2: Extract the 23-bit significand (001100000000000000000000). If the exponent's actual value is  $-127$ , include a '0.' before the significand; otherwise include a '1.'

- Step 3: Extract the sign bit. If it is a 1, then attach a negative sign before the significand.
- Step 4: Find the decimal equivalent of the significand.

Interestingly, when representing the exponents with the excess 127 code, the representation of floating-point number 0.0 becomes the all-zero pattern 00000000000000000000000000000000. Because zero is a frequently encountered floating-point number, the ability to quickly identify zero from its bit pattern is a clear advantage.

Figure 5.6 succinctly shows the range of numbers representable by the IEEE 754 single-precision format, and how it represents different regions within this range. The figure shows the real number line; the dots in the line denote the real numbers that are representable in the IEEE format. One major difference between the set of real numbers and the set of representable floating-point numbers is their density. Real numbers form a continuum, whereas the representable floating-point numbers do not form one. If a real number cannot be represented in floating-point format, then the obvious thing to do is to round that number to the nearest expressible floating-point number.

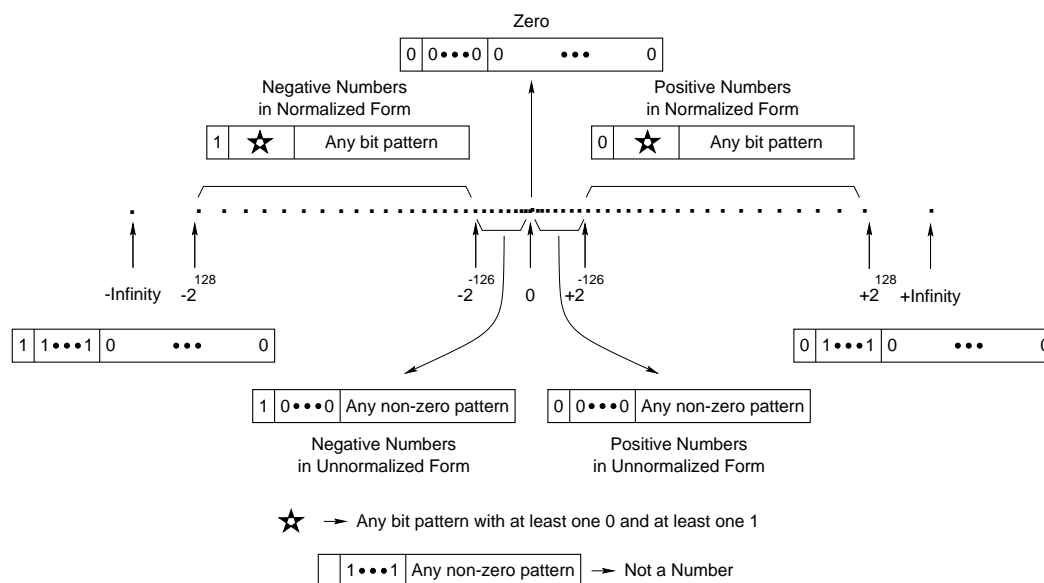


Figure 5.9: Range of Numbers Representable by IEEE 754 Single Precision Format

As we can see from the figure, the spacing between adjacent expressible numbers is not constant throughout. However, when the spacing is expressed as a percentage of the expressed numbers, there is no systematic variation throughout the expressible range. Therefore, the **relative error** introduced by rounding is approximately the same for small numbers and large numbers.



**Rounding:** As discussed earlier, because of using a finite number of bits, only a finite number of distinct real numbers can be represented in the computer. This means that most of the infinitely many real numbers must be rounded off to the nearest representable floating-point number, producing *roundoff errors*. In order to reduce roundoff errors during computation, IEEE provides the 80-bit extended precision format. **Expand.**

#### 5.6.4 Characters: ASCII and Unicode

The last type of data type we look at is the character. At the ISA level, characters, like numeric information, can only be represented by bit patterns. And in a manner similar to numeric data, the assumptions made about the format of characters is made at ISA design time.

The characters that are usually encountered in computer applications are upper-case alphabet (A-Z), lower-case alphabet (a-z), decimal digits (0-9), punctuation and formatting symbols (full stop, comma, quotation marks, space, tab, parenthesis, etc), arithmetic symbols (+, −, etc), and control characters (CR, LF, etc). Altogether, we have more than 64 characters, but fewer than 128. Therefore, we need a minimum of 7 bits for encoding these characters.

A standard code has been developed for representing characters. This code has received almost universal acceptance, and is called ASCII (American Standard Code for Information Interchange). ASCII uses a 7-bit representation for each character, allowing up to 128 different characters to be represented. The normal method for storing a character is to place the 7-bit pattern in an 8-bit field called a *byte*. The 8th bit can be used as the parity bit if required. Table 5.6 gives a sample of the bit patterns assigned to characters in the ASCII code. The first 32 codes (0x0 - 0x1F) are assigned to control characters: codes originally intended not to carry printable information, but rather to control devices (such as printers) that make use of ASCII, or to provide meta-information about data streams such as those stored on magnetic tape. For example, the bit pattern 0x08 represents “backspace”, and the bit pattern 0x0A represents the “line feed” function (which causes a printer to advance its paper).

The character digits “0”-“9” are represented with their values in binary prefixed with 0x3 (this means that converting a BCD number to ASCII code involves just taking each BCD nibble separately and prefixing it with 0x3. The uppercase alphabets A-Z are assigned consecutive patterns from 0x41 - 0x5A. The lowercase alphabets a-z are also assigned consecutive patterns. The bit patterns for the lowercase and uppercase letters differ only in the most significant bit. This trivializes case conversion to a range test (to avoid converting characters that are not letters) followed by a single bitwise operation.

A character string is represented by a concatenation of the ASCII codes for its component characters. For example, the 11-character string “God is good”, is encoded in ASCII by the following bit pattern sequence:

G        o        d                    i        s                    g        o        o        d

1000111 1101111 1100100 0100000 1101001 1110011 0100000 1100111 1101111 1101111 1100100

| ASCII Code | Character       | ASCII Code | Character |
|------------|-----------------|------------|-----------|
| 00H        | NULL            |            |           |
| 08H        | BACKSPACE       |            |           |
| 0AH        | Line Feed       |            |           |
| 0DH        | Carriage Return |            |           |
| 20H        | SPACE           |            |           |
| 21H        | !               |            |           |
| 30H        | 0               |            |           |
| 31H        | 1               |            |           |
| .          | .               |            |           |
| .          | .               |            |           |
| 39H        | 9               |            |           |
| 41H        | A               | 61H        | a         |
| 42H        | B               | 62H        | b         |
| .          | .               | .          | .         |
| .          | .               | .          | .         |
| 5AH        | Z               | 7AH        | z         |

Table 5.6: ASCII Codes for a Sample of Characters

In terms of mere adoption, the ASCII standard is perhaps one of the most successful software standards ever introduced. Clearly, the 7-bit ASCII representation is not sufficient for representing the characters of languages such as Chinese and Japanese, which have a very large number of characters. For representing the characters of such languages, more bits are required. Recently, a 16-bit character code known as **Unicode** has been developed as an international standard. The characters of most of the natural languages are encoded in Unicode. In fact, the ASCII standard has become embedded in Unicode, as the first 128 characters.

## 5.7 Designing ISAs for Better Performance

*“We still have judgment here, that we but teach bloody instructions,  
which, being taught, return to plague th inventor”*  
— Act I, scene 7 of *Macbeth*, William Shakespeare

The last major topic we discuss in this chapter is ISA design, that is, coming up with the specifications for a new ISA. Unlike the case with microarchitectures and other lower-level architectures, ISAs are not changed frequently, because of the need to maintain binary compatibility for existing machine language programs.

### 5.7.1 Technological Improvements and Their Effects

Instruction set architectures have evolved greatly since the late 1940s, when the first ISAs were implemented. Much of this evolution has been influenced by continued improvements in hardware technology and compiler technology, and also by changes in the application domain. For instance, the physical dimensions of the devices used to implement switching functions at the lower hardware levels have been shrinking at a rapid pace, ever since semiconductor transistors were invented. Today's technology allows millions of transistors to be integrated in a single chip. How does this affect ISA design? First of all, more functionality can be incorporated in the processor, thereby facilitating complex instructions in the ISA. Secondly, large memories can be built more easily, thereby facilitating large memory address spaces in the ISA.

Along with improvements in hardware technology came advances in language translation techniques.

There are many aspects of an ISA that are common to all ISAs. Common arithmetic and logical instructions, control-changing instructions, .... An ISA designer must consider several aspects of a potential feature to decide if it supports or conflicts with the design goals.

During these fifty years of ISA development, a consensus has emerged about the importance of some ISA features, for example, support of arithmetic and logical instructions, conditional branch instructions, subroutine call and return instructions, indirect addressing mode(s), constant addressing mode, etc. Almost all of the current ISAs include these; decisions to exclude any of these features must be made very carefully. On other issues, there has been and remains fundamental disagreement, for instance over the question of whether the support or lack of complex instructions is better. No single set of value judgments has yet emerged, because different ISAs have different goals and intended uses. DSP processors, for instance, include features such as .....

**Competing Design Goals:** In the previous two chapters we had identified different types of instruction opcodes and addressing mechanisms in the assembly-level architecture. One of the questions that must be addressed by a computer architect concerns the number and complexity of instructions to be included at the ISA level. One option is to include a large number of opcodes and addressing modes so as to reduce the total number of instructions in a program. An ISA that uses this method is called a *complex instruction set computer (CISC)*. An alternative method is to reduce the complexity of the instruction set, and thereby reduce the complexity of the hardware required for interpreting the instruction set. An ISA of this type is called a *reduced instruction set computer (RISC)*. In this section, we will examine some of the issues involved in this decision process.

The earliest computers were very simple in their ISA and implementation, both because of lack of experience with computers and because the implementation technology of those days mandated a simple machine. However, computer users wanted to solve rela-

tively complex problems, and high-level languages were developed that treated variables and arithmetic at a level higher than that of the machine language. Compilers were used to translate programs written in high-level languages to machine language programs. This resulted in what has become known as the *semantic gap*, which is the gap between the language of the programmer and the language of the hardware.

### 5.7.2 CISC Design Philosophy

One of the objectives of CISC designs is to reduce this semantic gap between the high-level language statements and the machine language instructions. It was felt that narrowing the gap by means of complex instructions and addressing modes would lead to better performance. Thus, computer architects began to make the ISAs more complex, with correspondingly complex hardware implementations, made possible by advances in hardware technology. CISC ISAs generally seek to reduce the compiler's complexity by making the ML instructions more closely conform to the operations of the high-level languages. Some computer systems (e.g. SYMBOL) have carried this to the extreme by completely abolishing the semantic gap between the high-level language and the machine language. That is, the high-level language itself is used as the native machine language. However, this is a rare practice.

Another reason for the popularity of complex instructions was that it reduced the size of machine language programs. Program and data storage were at a premium in those days. For instance, when Motorola introduced the M6800, 16 KB RAM chips cost \$500, and 40 MB hard disk drives cost \$55,000. When the MC68000 was introduced, 64 KB RAM chips still cost several hundred dollars, and 10 MB hard drives cost \$5,000.

If there is an overriding characteristic of the CISC concept, it is an approach to ISA design that emphasizes *doing more with each instruction*. As a result, CISC machines have a wide variety of addressing modes, 14 in the case of the MC68000, and 25 in its more complex successor, the MC68020. Furthermore, CISC machines allow an instruction to have variable number of operands, which can be present in registers and/or memory. For instance, the VAX `ADD` instruction can have two or three operands, and any one of them can be in a register or in memory. Another example of a CISC ISA is Intel's IA-32 ISA (more commonly known as the x86 ISA).

### 5.7.3 RISC Design Philosophy

By the early 1980s, compilers had advanced to such a level that almost all of the programming began to be done in high-level languages. At that time, studies were conducted to analyze the instruction set usage in contemporary CISC machines. These studies indicated that compiler-generated ML programs, for the most part, were utilizing only a small subset of all available instructions. An indication of this can be seen when inspecting the assembly language programs we wrote in the last two chapters; instructions such as `li`, `lw`, `sw`, `add`,

`sub`, `beq`, and `bne` were used most of the time. Carrying this observation to the next logical step, computer architects figured that computer performance could be enhanced by including in the ISA only the frequently used instructions, and by making them execute as fast as possible. The RISC approach, therefore, is to use a *simpler ISA* so as to permit a *faster clock* than that possible in a CISC processor. With many commercial implementations this premise is fulfilled. Examples are the MIPS-I, Sparc, PowerPC, and Alpha ISAs, which were implemented in several commercial processors. In practice, RISC-type ISAs permit a number of strategies to be employed by the microarchitects so as to make use of a variety of implementation features, such as pipelining and multiple instruction issue. In addition, they help to free up space within the processor chip that can be usefully employed to incorporate on-chip cache memory.

The complex statements of a high-level language would be translated to longer instruction sequences than corresponding CISC instruction sequences. The result is that a machine language program for a RISC ISA is likely to have more instructions, each of which may execute faster. The name RISC signifies the focus on reducing the number and complexity of the instructions in the ISA. There are several other tenets common to RISC ISAs, and these are described in the following paragraphs. You should be aware, however, that a particular RISC ISA may not have all of these tenets.

**Minimal Number of Opcodes and Addressing Modes:** RISC ISAs usually limit themselves to about three addressing modes: register addressing, register indexed addressing, and constant addressing. Other, more-complex addressing modes are synthesized in software from the simple ones. Only the instructions that are frequently executed, RISC machines exclude infrequently used instructions. Simpler instructions permit shorter clock cycles, because less work has to be done in a clock. The result is a smaller, faster processor, that is capable of executing more instructions in a given amount of time than a CISC processor. Complicated addressing modes mean longer clock periods, because there is more address calculation to perform.

**Fixed Instruction Length and Simple Instruction Formats:** Another common characteristic of RISC ISAs is the use of a few, simple instruction formats. Instructions have fixed length and are aligned on word boundaries. Instructions are encoded in such a way that field locations, especially the opcode, are fixed. This type of encoding has a number of benefits. First, with fixed fields, register operand accesses can proceed prior to the completion of instruction decoding. Second, simplified instruction formats simplify the instruction decoder circuitry, making it faster.

**Only Load and Store Instructions Access Main Memory:** Another important characteristic of RISC ISAs is that arithmetic/logic instructions deal only with register operands. Thus, only data transfer instructions, such as `LOAD` and `STORE` instructions, deal with the memory address space. Restricting the operands of arithmetic/logic instructions

to be in registers makes it easier to meet the above stated objective of single cycle execution. The RISC approach relies on the observation that values stored in registers are likely to be used several times before they are written to main memory.

The proponents of the RISC philosophy cite several reasons why the RISC approach to ISA design can potentially lead to faster processors. Some of these reasons are given below.

1. Instruction decoding can be faster because of simpler instruction decoder (due to fewer instructions in the ISA).
2. Memory access is faster because of the use of on-chip cache memory (cf. chapter 7), which is possible because of less hardware in the chip to do instruction decoding, complex addressing modes, and complex instructions.
3. The use of simpler instructions makes it easier to use a hardwired control unit, which is faster (cf. chapter 9).
4. Clock cycle time is potentially smaller because of less hardware in time-critical path(s).

#### **5.7.4 Recent Trends**

“The amount of money you make is directly proportional to your vocabulary.”

### **Multimedia Extensions**

#### **Vector Instructions**

#### **VLIW and EPIC Instructions**

## **5.8 Concluding Remarks**

From a functional point of view, the ISA implements the assembly-level architecture. Although our discussion was based primarily on this view, historically, the ISA is defined before defining the assembly-level architecture. In other words, the assembly-level architecture being a symbolic form of the ISA, is built on top of the ISA. Considering the historical development of computers, the ISA came into existence before the assembly-level architecture as well as the high-level architecture; in the early computers, programs were directly written in machine language, in 0s and 1s!

## 5.9 Exercises

1. What is the decimal equivalent of 0xDD580000 if interpreted as an IEEE single precision floating-point number?
2. What is the decimal equivalent of the following bit pattern if interpreted as an IEEE single precision floating-point number?  
0 00000000 000000 .... 0
3. What are the decimal equivalents of the largest and smallest positive numbers that can be represented in the IEEE single precision floating-point format?
4. Consider a floating-point number format that is similar to the IEEE format in all respects, except that it does not use the *hidden bit* technique. That is, it explicitly stores the most significant bit of the significand. Show that this number system can represent only fewer floating-point numbers than the IEEE number system.
5. Explain why denormalization is done in the IEEE floating-point format for the smallest exponent?
6. Both the assembly language program and the machine language program are stored as “bit patterns” inside the main memory. Explain how the bit patterns of these two programs are different.
7. An ISA defines 2 different instruction formats. In the first format, 6 bits are used for the opcode, Among these opcode bit patterns, three are used as special patterns. When any of these three patterns appear in the opcode field, the instruction is encoded using the second format, and the actual opcode is present in another 6-bit field. How many unique opcodes can this ISA support?
8. Design a variable-length opcode (but fixed-length instruction) to allow all of the following to be encoded in 36-bit formats:  
7 instructions with two 15-bit addresses and one 3-bit register number  
500 instructions with one 15-bit addresses and one 3-bit register number  
50 instructions with no addresses or registers.

## Part II

# PROGRAM EXECUTION — HARDWARE LEVELS

*By wisdom a house is built, and through understanding it is established; through knowledge its rooms are filled with rare and beautiful treasures.*

**Proverbs 24: 3-4**



The theme of this book is that a modern computer can be viewed as a series of architectural abstractions, each one implementing the one above it. Part II of the book gave a perspective of the computer from the high-level language, assembly language, and machine language programmers' points of view. These views relate to the abstract machine levels that deal with *program development*, typically called the software levels.

We now turn our attention to the design of hardware for carrying out *program execution*—moving electrons through wires and semiconductors to find the results of executing a program with a specific set of inputs. Leaping from the “lofty” programs to the “lowly” electrons is quite a dive, as you can imagine! Naturally, computer hardware design, like computer programming, is carried out at multiple abstraction levels. Although such a multi-layer implementation may incur some performance costs, it does have important engineering advantages. The main advantage of adding extra implementation levels is that it permits the complexity of the hardware to be tackled in a step-by-step manner. Once an appropriate interface has been agreed upon between two adjacent levels, the development of these levels can proceed more or less independently.

We will study three of these levels—the microarchitecture, the RTL architecture, and the logic-level architecture—in this part of the book. Although these machine levels have traditionally been implemented in hardware (for a variety of reasons), their real distinction from the previously seen machine levels is that they relate to program execution, and implement the machine levels immediately above them by *interpretation* rather than by translation. One point will become apparent when you study the hardware abstraction levels presented in this last part of the book: many of the traditional digital circuit design techniques are of limited use while designing the computer hardware! For instance, in theory, the digital computer can be viewed as a *finite state machine*. However, it is not practical to design a digital computer as a single finite state machine, because of the combinatorial explosion in the number of states. In practice, computer hardware design is carried out by partitioning the hardware into different parts, based on functionality.

Program execution begins with copying the executable binary from a storage medium into the computer's memory. This process, called *loading* the program, is typically done by the operating system. Loading, along with related topics such as dynamic linking, are discussed in Chapter 6.

The microarchitecture—the first virtual machine level we study in this part—implements the instruction set architecture (ISA) by interpreting ML programs, as illustrated in Figure 1.10 on page 42. The details of the microarchitecture depend on the ISA being implemented, the hardware technology available, and the cost-performance-power consumption goals for the computer system. Microarchitecture-related issues are covered in two chapters. In Chapter 7, we discuss microarchitectural aspects related to implementing the user mode ISA. The initial objective is to present a microarchitecture that correctly implements the user mode ISA and correctly executes user mode ML programs. Then we move on to more advanced microarchitectures for the processor system and the memory system. In this part of the chapter, we give special importance to achieving high performance and low power. One of the important microarchitectural techniques discussed for improving processor performance is pipelining. For memory systems, we discuss cache memories in detail. Chapter 8 discusses microarchitectural aspects that are specific to implementing the kernel mode ISA. Topics covered include exceptions and interrupts, virtual memory, and the IO system.

Chapter 9 discusses register transfer level architecture.

The logic-level architecture implements the microarchitecture using logic gates, flip-flops, and other building blocks. This virtual machine level is covered in detail in Chapter 10.

## Chapter 6

# Program Execution Basics

*Apply your heart to instruction and your ears to words of knowledge.*

**Proverbs 23: 12**

A successfully developed program becomes useful only when it is executed, most likely with a set of inputs. Program execution involves executing in the proper sequence the instructions specified in the program. A typical computer system has ....

*“The execution of the laws is more important than the making of them.”*  
— Thomas Jefferson

*“I know no method to secure the repeal of bad or obnoxious laws so effective as their stringent execution.”*  
— Ulysses S. Grant

### 6.1 Overview of Program Execution

In a small embedded computer system or a microcontroller that executes only a single program, the program to be executed may be permanently stored in some kind of ROM (read-only memory) that implements the memory address space of the computer. This stand-alone program utilizes and manages all of the system resources by itself. Upon reset, the program counter points to the beginning of the program. Thereafter, the system continues to execute the program until the next reset or power down. “Loading” the program into the ROM — a very rare event — is usually done using a ROM programmer device that has direct access to the ROM.

The situation is more complex for general-purpose systems such as desktops and laptops, which are controlled by an OS and can execute multiple programs. The majority of the memory address space in such systems is implemented by RAM (random access memory) to make it easy for end-users to change the program to be executed. Several steps are involved in the execution of a program in such a system:

- Select the program
- Create the process
- Load the program
- Dynamically link libraries (optional)
- Execute the program
- Stop and restart the program (optional)
- Halt the program

This chapter discusses each of these steps in detail. All of the steps, except the interpretation step, are typically done by the OS. The interpretation step, which forms the bulk of program execution, is usually done directly by the hardware, for efficiency reasons. We can, however, do this step also in software, and for pedagogic reasons, we include a discussion on software interpreters at the end of this chapter.

## 6.2 Selecting the Program: User Interface

In computer systems with an installed operating system, the operating system decides which program should be executed and when. The end-user can specify the program to be executed by giving the appropriate command to the operating system (OS). The part of the OS that receives commands from the end-user is called a *shell*. From the end-user's perspective, the shell is thus an important part of a computer system; it is the interface between the user and the system.

Functionally, the shell behaves like an interpreter, and operates in a simple loop: accept a command, interpret the command, execute the command, and then wait for another command. Executing the command often involves requesting the kernel to create a new process (called a *child* process) that performs the command. The child process is overlaid with the program to be executed; once the program is completed, the child process is terminated. The program to be executed is typically stored on a file system maintained by the OS.

Over the years, many shells have been used and improved. Historically, they have a flavor heavily dependent on the OS they are attached to. We can classify shells into one of three categories based on the primary input device used to enter commands:

- Text based — *command line interface* (CLI)
- Graphics based — *graphical user interface* (GUI)
- Voice based — *voice user interface* (VUI)

In a CLI environment, a command can be given, for instance, by typing the command in a terminal via a keyboard. In a GUI environment, a command can be given, for instance, by pointing the mouse at the program's *icon* and clicking it. In a VUI environment, a command can be given, for instance, by speaking into a microphone.

### 6.2.1 CLI Shells

In a CLI environment, the user specifies a program to be executed by typing the command in a terminal via a keyboard. A CLI shell displays a *prompt* whenever it is ready to accept a new command. A command consists of a command name, followed by command options (optional) and command arguments (optional). The command name, options, and arguments, are separated by blank space. The basic form of a CLI command is:

`commandname [-options] [arguments]`

`commandname` is the name of the program the end-user wants the computer to execute. `options`, usually indicated by a dash, alter the behavior of the command. `arguments` are the names of files, directories, or programs that the program needs to access. The square brackets ([ and ]) signify optional parts of the command, which may be omitted.

**Example:** The Unix command `ls -l /tmp` gives a long listing of the contents of the `/tmp` directory. In this example, `ls` is the command name, `-l` is an option that tells `ls` to create a long, detailed output, and `/tmp` is an argument naming the directory that `ls` should list.

A lot of typing on a keyboard can lead to repetitive strain injury (RSI)<sup>1</sup>. Newer CLI shells reduce the amount of typing, with features such as auto-completion, showing the files it would complete, and showing the current directory in the prompt. In addition, they provide many facilities such as command aliasing and job control. Furthermore, a collection of commands may be stored in a file, and the shell can be invoked to execute the commands in that file. Such a file is known as a *shell script* file. The language used in that file is called *shell script language*. Like other programming languages, it has variables and flow control statements (e.g. if-then-else, while, for, goto). Most shells also permit the user to abort a command being executed, by typing Control-C on the terminal.

---

<sup>1</sup>Repetitive Stress Injury, sometimes referred to as overuse syndrome, is pain and swelling that results from performing a repetitive task, like text messaging and typing. The traditional input devices that computer professionals use—the keyboard and the mouse—force the user to adapt their posture in order to adequately operate the input devices. The static tension of the muscles that occurs while operating a standard keyboard and mouse is thought to be one of the main causes of RSI for computer professionals.

Examples of CLI shells for Unix/Linux operating systems are Bourne shell (sh), Bourne-Again shell (bash), C shell (csh), TENEX C shell (tcsh), and Korn shell (ksh). Examples of non-Unix CLI shells are command.com (for DOS), cmd.exe (for OS/2 in text mode and for Windows NT).

### 6.2.2 GUI Shells

In a GUI environment, the end-user specifies a program to be executed by pointing the mouse at the program's icon and clicking it. The system then responds by loading the program into the physical memory. It also creates a new *window* on the display for the user to interact with the program. The end user can also change the size, shape, and position of the window.

---

---

Figure 6.1: Photo/Screen Shot of a GUI Desktop

---

The visual approach and the ease of use offered by GUIs have made them much more popular, particularly among the masses. The relative merits of CLI- and GUI-based shells are often debated. From the performance perspective, a GUI would probably be more appropriate for a computer used for image or video editing. On the other hand, for certain other operations such as moving files, a CLI tends to be more efficient than a GUI, making it a better choice for servers used for data transfers and processing with expert administration. The best choice is really dependent on the way in which a computer will be used.

Graphical user interfaces do have some disadvantages. They are harder to develop and require a lot more memory to run. The system may also need to have powerful graphic video capability. Moreover, they do not free the end user from repetitive strain injury (RSI). Although the amount of muscle movement required is small with a mouse (compared to a keyboard), all of the work done with a mouse is done with one hand and mostly with the

index finger, whereas the keyboard lets the user distribute the work between two hands and multiple fingers. In laptops, the mouse is usually located as a pointer in the middle of the keyboard or as a touch pad. These require excellent coordination and precision of the fingers, making the muscles extra tense. It is therefore important to use an external mouse whenever possible.

*“In baiting a mousetrap with cheese, always leave room for the mouse.”  
— Saki, in The Infernal Parliament*

Examples of GUI shells for Unix/Linux operating systems (X Windows based) are KDE, GNOME, Blackbox, and CDE. Examples of GUI shells for Microsoft Windows environments are Windows Explorer, Litestep, Geoshell, BB4Win, and Emerge Desktop. Modern versions of Microsoft’s Windows operating system officially support only Windows Explorer as their shell. Explorer provides the familiar desktop environment, start menu, and task bar, as well as the file management functions of the operating system. Older versions also include Program Manager, which was the Shell for the 3.x series of Microsoft Windows. Macintosh Finder is a GUI for Apple. The newer versions of GUI shells improve end-user performance by providing advanced menu options.

### 6.2.3 VUI Shells

The VUI environment provides a new dimension for the end-user to select programs to run and to interact with the computer. It permits the user to speak command(s) directly into a noise-cancelling microphone, with no intermediate keying or mouse clicks. The voice is converted to digital form by the sound card, and is then processed by speech recognition software. Voice recognition can be viewed as a problem defined by three axes: vocabulary size, degree of speaker independence, and continuous speech capability. Some of the important issues in a VUI environment are speaker independence, continuous speech capability, and flexible vocabulary.

- **Vocabulary size:** A large vocabulary size is beneficial, as it lets the end-user specify a large number of unique commands. Some systems also permit customization of the vocabulary.
- **Degree of speaker independence:** Speaker independence allows a VUI to accept and recognize (with high accuracy) commands spoken by many users, including voices that were not part of its training set. Speaker-independent VUIs require no prior training for an individual user. In contrast, a speaker-dependent system requires samples of speech for each individual user prior to system use (i.e., the user has to train the system). Speaker-independence is desirable, as it permits the same computer to be used by multiple end-users, without training sessions in between.

- **Degree of connectedness:** This parameter deals with the extent to which words can be slurred together. If a good deal of connectedness is allowed, the user can talk naturally without pauses between words. The other extreme would sound like a teacher in a locution class, mouthing each word. Depending on the degree of connectedness, VUIs can be categorized as *isolated word*, *connected word*, and *continuous speech*. Isolated word recognisers are the simplest, but require a short pause of approximately 1/5 second between each word. Connected word recognizers recognize words spoken continuously, so long as the words do not vary as they run together, i.e., they require clear pronunciation. Continuous speech allows the user to speak words as normally spoken in fluent speech, and is the most desirable one to have.

The simplest of the VUIs support a small vocabulary of speaker-dependent words, that must be uttered with distinct pauses between each. On the other extreme would be a VUI that supports a large vocabulary, and caters to a large number of speakers, who tend to have slurred speech.

A typical office environment, with a high amplitude of background speech, turns out to be one of the most adverse environments for current speech recognition technologies. Large-vocabulary systems with speaker-independence that are designed to operate within these adverse environments have been observed to have significantly lower recognition accuracy. The typical recognition rate achievable as of 2005 for large-vocabulary speaker-independent systems is about 80%-90% for a clear environment, and about 50% for a noisy environment such as with a cellular phone.

Examples of voice recognition software are Apple Speech Recognition, DragonDictate (a large vocabulary, speaker-adaptive voice recognition dictation system capable of recognizing up to 120,000 words. It allows free-form dictation into most text-based applications. It allows full mouse movement as well as text and numerical dictation and complete formatting, all by voice, completely hands free. DragonDictate uses industry standard third-party sound cards.), NaturallySpeaking software, Voice Xpress package, and ViaVoice Pro.

In the past decade, tremendous advances in automatic speech recognition have taken place. A reduction in the word error rate by more than a factor of 5 and an increase in recognition speeds by several orders of magnitude (brought about by a combination of faster recognition search algorithms and more powerful computers), have combined to make high-accuracy, speaker-independent, continuous speech recognition for large vocabularies possible in real time, on off-the-shelf workstations. These advances promise to make speech recognition technology readily available to the general public.

Although VUI was introduced to combat the RSI problem inherent with keyboard and mouse, it is not a magic wand either. Intensive use of VUI can subject the vocal cords to overuse, and associated injury (vocal loading).

*“When I tell you in few years it will be possible for you to sketch in the air and have the thing you sketch in the air come to your eye, solid and real, so that you can walk around it, so that you can scrutinize it from any direction and any view point you please. I am telling you the truth.”*

*— Steven A. Coons. in a panel discussion entitled: The past and future of design by computer, 1968*

## 6.3 Creating the Process

## 6.4 Loading the Program

When the OS receives a command to execute a program, it transfers control to a *loader* (a part of the OS), which reads the executable program into memory and initiates its execution. When the program is loaded into memory, it is not just copied into memory and executed; there are a number of steps the loader takes to load the image correctly and set up things in that memory image before it jumps to the code in that image. The UNIX OS uses the following steps to do loading; steps 2 and 3 may make sense only after studying *virtual memory*, described in Chapter 9.

1. Read the executable file’s header and determine the size of the text and data sections.
2. Create a new virtual address space for the program. Allocate enough physical memory to hold the text, data, and stack sections.
3. Copy the instructions and the data from the executable file onto the allotted physical memory. If there is not enough physical memory, then copy at least the portion of the text containing the entry point of the program, i.e., the first instruction to be executed.
4. Link and load libraries if load-time dynamic linking is to be done.
5. Copy onto the stack any parameters that are to be passed to the main function of the program. Such parameters include command line arguments supplied by the user to the OS shell.
6. Initialize the general-purpose registers. In general, most registers are cleared, but the SP register is assigned the address of the top of stack.
7. Jump to the start-up routine at the beginning of the program.



### 6.4.1 Dynamic Linking of Libraries

In Chapter 6, we saw that libraries are often statically linked to the application program while forming the executable binary. While this produces a “self-contained” executable program, static linking of libraries does have some drawbacks. First and foremost, libraries are generally quite large, and linking them to the static executable can result in very large executables. Another drawback is that the executable cannot take advantage of newer versions of the library when they become available, without re-doing the static linking. Examples of libraries that are traditionally designed to be statically linked include the ANSI C standard library and the ALIB assembler library.

Dynamic linking is a solution adopted to deal with these drawbacks<sup>2</sup>. With this scheme, the library code is not stitched into the program executable by the static linker. Instead, the static linker only records which libraries the program needs and their index names or numbers. The libraries themselves remain in a separate file on disk. The majority of the work of linking is done at program execution time. Such library routines are called *dynamically linked library* or *dynamic link library* (DLL) routines. In Microsoft Windows environments, dynamic libraries use the filename extension `.dll`. Two different options exist for the time at which dynamic linking is done: load-time dynamic linking and run-time dynamic linking.

#### 6.4.1.1 Load-time dynamic linking

In this case, dynamic linking is done by the dynamic linker at the time the application is loaded. The dynamic linker finds the relevant libraries on disk and links them to the executable; these libraries are loaded at the same time to the process’ memory space. The dynamic linker itself may be part of the library (as in Linux), or may be part of the OS kernel (as in Windows ..). In the former case, the OS maps the dynamic linker to a part of the process’ address space, and execution begins with the bootstrap code in the linker. Some operating systems support dynamic linking only at load time, before the process starts executing.

**Lazy Procedure Linkage:** Executables that call dynamic libraries generally contain calls to a lot of functions in the library. During a single execution, many of these functions may never be called. Furthermore, each dynamic library may also contain calls to functions in other libraries, even fewer of which will be called during a given execution. In order to reduce the overhead during program loading, dynamically linked ELF programs use lazy binding of procedure addresses. That is, the address of a procedure is not bound until the first time the procedure is called.

---

<sup>2</sup>Although dynamic linking libraries have gained popularity only recently, they date back to at least the MTS (Michigan Terminal System), built in the late 1960s [8].

### 6.4.1.2 Run-time dynamic linking

In this case, the linking of a library is done just when it is actually referenced during the execution of the process. This type of dynamic linking is often called **delay loading**. Figure 6.2 illustrates how this type of dynamic linking occurs. When the user program wants to call library routine `S.dll` (from address `0x401000`), it uses a `syscall` to convey the request to the *dynamic loader* part of the OS, as shown in part (a) of the figure. It will also pass the name of the routine (`S`) to the OS. The dynamic loader loads `S.dll` (at address `0x402000`), as it was not previously loaded. This is shown in part (b) of the figure. The loader then transfers control to `S.dll`. After the execution of `S.dll` is over, control passes back to the dynamic loader, which then transfers control back to the user program (at address `0x401004`).

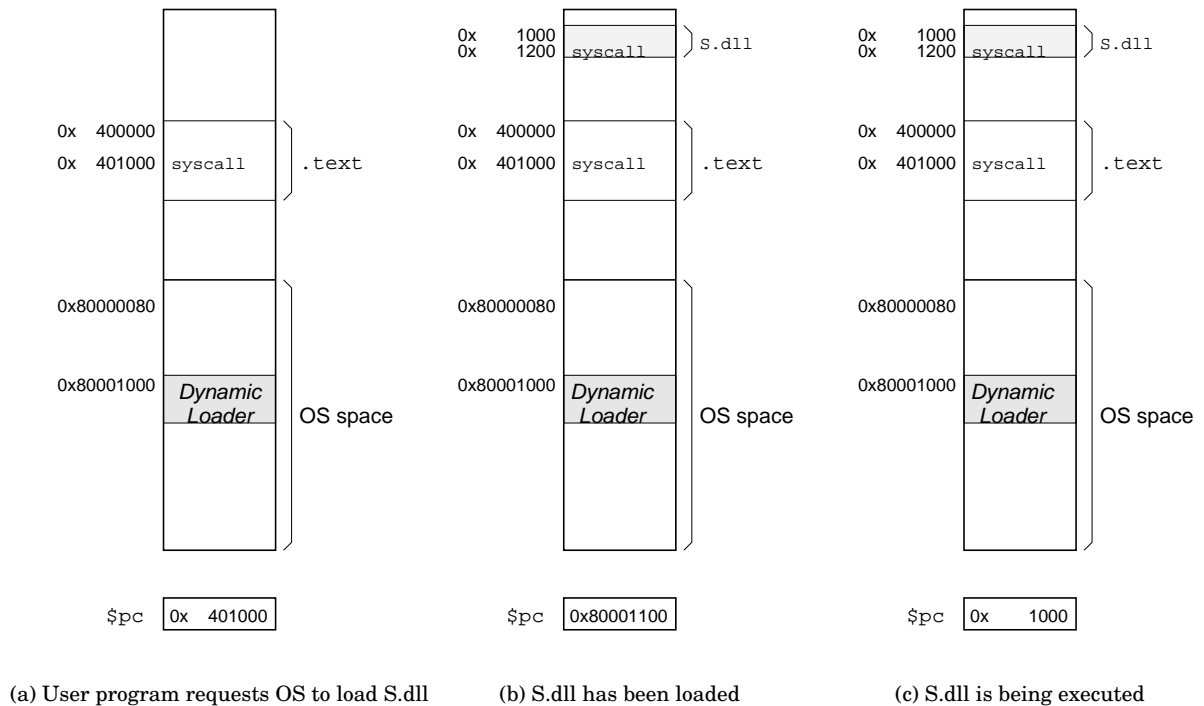


Figure 6.2: Memory Map During Different Stages of Dynamic Linking

One wrinkle that the loader must handle is that the memory location of the actual library code is not known until after the executable and all dynamically linked libraries have been loaded into memory, because the memory locations assigned will depend on which specific DLLs have been loaded. In theory, it is possible to examine the program and replace all references to data in the libraries with pointers to the appropriate memory locations once all DLLs have been loaded. However, this would require a large amount of

time and memory. Therefore, most dynamic library systems take a different approach. At compile time, a symbol table with blank addresses called the *import directory* is linked into the program. At load time, this table is updated with the location of the library code/data by the loader/linker. At run time, all references to library code/data pass through the import directory.

The library itself contains a table of all the methods within it, known as entry points. Calls into the library "jump through" this table, looking up the location of the code in memory, then calling it. This introduces overhead in calling into the library, but the delay is usually so small as to be negligible.

**Specifying the Library Routine:** A dynamic link library routine can be specified in a program executable in two different ways. The first option is to specify the path that locates the library within the OS file system. Any change to the library naming or layout of the file system will cause these systems to fail. In the second option — the more common one — only the name of the library routine (and not the path) is specified in the executable, with the operating system incorporating a mechanism to locate the library in the file system. Unix-based systems have a list of "places to look" in a configuration file, and dynamic library routines are placed in these places. On the downside this can make installation of new libraries problematic, and these "known" locations quickly become home to an increasing number of library files, making management more complex. Microsoft Windows will check the Registry to determine the proper place to find an ActiveX DLL, but for standard DLLs it will check the current working directory; the directory set by `SetDllDirectory()`; the `System32`, `System`, and `Windows` directories; and finally the `PATH` environment variable.

## 6.5 Executing the Program

After the loader loads the selected program into memory and completes all steps of dynamic linking, it transfers control to the program. That is, the **program counter** is set to the *entry point* of the program. Thereafter, the processor executes the loaded program by executing the instructions (in the `.text` portion) of the program. Execution of these instructions involves repeating two actions: fetch and execute. Fetch concerns obtaining the instruction bit pattern from memory and decoding the bit pattern. Execution involves carrying out the action specified in the instruction, including determining the next instruction to be executed. The fetch and execute phases can be subdivided into the following sequence of steps (some of which are optional):

### Executing the ML Program

- **Fetch Phase**

1. *Fetch instruction*: Read the next instruction from the memory.
2. *Decode instruction*: Decode the instruction bit pattern so as to determine the action specified by it.

- **Execution Phase**

1. *Fetch source operand values*: Fetch source operands (if any) from registers, memory locations, or IO registers.
2. *Process data*: Perform arithmetic or logical operation on source operand values, if required.
3. *Write result operand value*: Write the result of an arithmetic or logical operation to a register, memory location, or IO register, if required.
4. *Update program counter*: Normally, the program counter needs to be incremented so as to point to the immediately following instruction. When a control flow change is required because of a branch instruction or the like, the program counter is updated with the new target address determined in the execution phase.

## 6.6 Halting the Program

## 6.7 Instruction Set Simulator

We have seen the fundamental steps involved in executing a (machine language) program. Although the operating system plays a major role in setting up the stage, the bulk of the work — interpreting the program — is usually carried out directly in hardware. A hardware interpreter is more popular because of its superiority in performance and power consumption. As pointed out in Chapter 1, however, the distinction between hardware and software is somewhat blur. We can in fact implement an emulator in software; such an emulator is generally called a *software simulator*. The SPIM simulator that we discussed in Chapter 3 is a good example. It loaded MIPS assembly language programs, translated them to MIPS machine language (ML) programs, and then executed them by interpreting the ML instructions, one by one. Partly to prove this point, and more importantly, to start with a simple emulator, we shall first study a software emulator, and then move on to hardware interpreters in the next chapter.

It is important to note that a software simulator program can perform its emulation

of a *target ISA* only when the simulator is executed on another microarchitecture (called *host machine*), entailing an interpretation by the host machine's control unit. Thus, implementing a microarchitecture in software introduces an extra interpretation step for the ML program. Why in any case, would anyone want to introduce extra interpretation steps, other than for pedagogic reasons? Although software simulation may not make much sense for ordinary users, software simulation is the *de facto* tool of the trade for computer architects who are in the business of developing new ISAs and microarchitectures.

Another common situation that warrants the use of a software simulator is in executing Java applets that are supplied over the web. Java applets are compiled and assembled into **bytecode**, with the **Java Virtual Machine (JVM)** as the target ISA. Java bytecode can be directly executed on hardware only if the hardware implements the JVM ISA. When a web browser running on a different platform downloads a Java applet, it interprets the bytecode using a software interpreter, as illustrated in Figure 6.3. This permits the web server to supply the same bytecode irrespective of the platform from which web browsing is done.

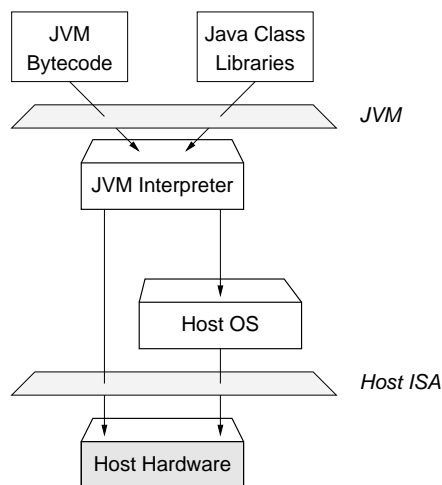


Figure 6.3: Execution of a Java Application by Interpretation

It is important to note that this simulator uses a behavioral approach, and models only the functional aspects of emulation, without doing a hardware microarchitecture design. A functional simulator thus correctly executes machine language programs, without modeling any hardware devices. Such a simulator has no notion of hardware-specific features such as clock cycles and buses. It just takes in as input the program to be executed (along with the executed program's inputs), and outputs (among other things) the outputs generated by the executed program. Examples are the SPIM simulator and the JVM interpreter.

We shall develop a simple functional simulator here. For simplicity and ease of understanding, we restrict ourselves to a subset of the MIPS-I ML instruction set which we call

**MIPS-0.** The encodings of the MIPS-0 instructions and the semantics are same as that in the MIPS-I ISA. The MIPS-0 instruction set, along with the instruction encodings, is given in Figure 6.4.

|         |                |        |    |    |    |        |
|---------|----------------|--------|----|----|----|--------|
| LUI     | rt, immed      | 001111 |    | rt |    | immed  |
| LW      | rt, offset(rs) | 100011 | rs | rt |    | offset |
| SW      | rt, offset(rs) | 101011 | rs | rt |    | offset |
| ADDI    | rt, rs, immed  | 001000 | rs | rt |    | immed  |
| ADDU    | rd, rs, rt     | 000000 | rs | rt | rd | 100001 |
| SUBU    | rd, rs, rt     | 000000 | rs | rt | rd | 100011 |
| ANDI    | rt, rs, immed  | 001100 | rs | rt |    | immed  |
| AND     | rd, rs, rt     | 000000 | rs | rt | rd | 100100 |
| ORI     | rt, rs, immed  | 001101 | rs | rt |    | immed  |
| OR      | rd, rs, rt     | 000000 | rs | rt | rd | 100101 |
| NOR     | rd, rs, rt     | 000000 | rs | rt | rd | 100111 |
| SLLV    | rd, rs, rt     | 000000 | rs | rt | rd | 000100 |
| BEQ     | rs, rt, offset | 000100 | rs | rt |    | offset |
| BNE     | rs, rt, offset | 000101 | rs | rt |    | offset |
| JALR    | rd, rs         | 000000 | rs |    | rd | 001001 |
| JR      | rs             | 000000 | rs |    |    | 001000 |
| SYSCALL |                | 000000 |    |    |    | 001100 |

Figure 6.4: The MIPS-0 Instruction Set, along with Encoding

### 6.7.1 Implementing the Register Space

Let us take a closer look at designing functional simulators. First, we will see how the register model specified in the ISA is implemented in the simulator. This can be accomplished by declaring an appropriate data structure (usually an array) in the simulator.

```
/* Allocate space for modeling the ISA-visible registers */
long R[32];
long PC;
```

Apart from the ISA-visible registers, many new “registers” may need to be defined, for storing temporary values.

### 6.7.2 Implementing the Memory Address Space

Conceptually, the memory address space of the target ISA can be implemented in a manner similar to what we did for the register space. Notice that with this simple approach, either the target’s memory space should be smaller than that of the host, or the entire memory space of the target is not simulated.

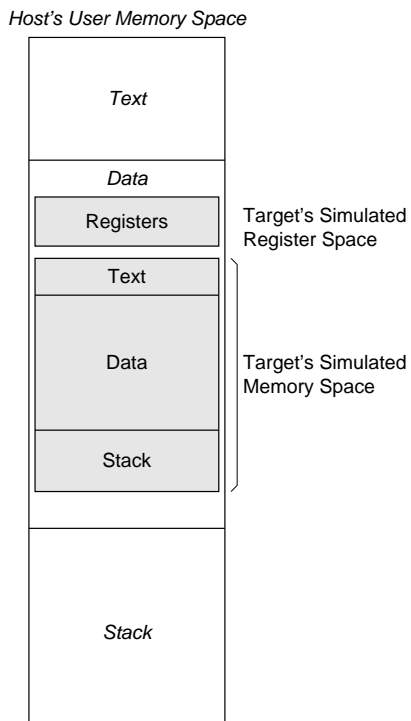


Figure 6.5: Implementing the Register Space and Memory Space in a Functional Simulator

The simulator maintains the mapping between the target addresses and the host addresses. When the simulated program accesses a memory location, the simulator translates that address to a host memory address.

```
/* Allocate space for modeling the 3 sections of the memory address space */
long text[0x800000];
long data[0x800000];
```

```
long stack[0x800000];
```

### 6.7.3 Program Loading

The binary program to be executed (i.e., the ML program whose execution needs to be simulated) is typically stored as a file in the host machine's file system. Prior to execution by the simulator, this program needs to be loaded into the target's instruction memory which is modeled in the simulator. This loading job is quite similar to the job done by the *loader* part of the OS when a user requests a program to be executed directly in a machine. The simulator needs to have the functionality to perform this "loading".

### 6.7.4 Instruction Fetch Phase

```
/* Fetch the instruction from the text section of memory */
IR = text[PC - 0x400000];

/* Separate the different fields of the instruction */
opcode = [IR & 0xfc000000] >> 26;
rs      = [IR & 0x3e00000] >> 21;
rt      = [IR & 0x1f0000] >> 16;
rd      = [IR & 0xf800] >> 11;
func    = IR & 0x3f;
offset  = IR & 0xffff;

/* Decode the instruction */
switch (opcode)
{
    case 000000b:

        case 000100b: /* beq */

        case 000101b: /* bne */

        case 001000b: /* addi */
}
```

### 6.7.5 Executing the ML Instructions

Once the program to be executed is loaded into the simulator's memory, execution of the program (i.e., simulation of the execution) can begin. The PC variable is initialized to the



program starting address. The simulator then goes through a loop, with each iteration of the loop implementing the fetching and execution of an ML instruction.

The simulation of the machine is done by just a big function in the simulator. This function understands the format of MIPS instructions and the expected behavior of those instructions as defined by the MIPS architecture. When the MIPS simulator is executing a “user program”, it simulates the behavior of a real MIPS CPU by executing a tight loop, fetching MIPS instructions from the simulated machine memory and “executing” them by transforming the state of the simulated memory and simulated machine registers according to the defined meaning of the instructions in the MIPS architecture specification. Remember that the simulated machine’s physical memory and registers are data structures in the simulator program.

### 6.7.6 Executing the Syscall Instruction

Execution of syscall instructions in the simulator is somewhat complex due to a variety of reasons. First of all, the binary program being “executed” does not contain the Kernel code that needs to be executed upon encountering a syscall instruction; this code is a part of the operating system of the *target* machine. Therefore, instead of simulating the execution of Kernel code on an instruction-by-instruction basis, the simulator directly implements the functionality specified by the syscall instruction.

Secondly, many of the syscall instructions require intervention of the host machine’s operating system. For instance, if a syscall instruction specifies a file to be opened (`open()` system call), only the host machine’s operating system can open the file; remember that the simulator is just an Application program (User mode program) that runs on the host machine. Similarly, if a syscall instruction specifies another process to be created (`fork()` system call), only the host machine’s operating system can create another process. For such system calls, the simulator should act as an interface to the host machine’s operating system. Some other system calls, such as `brk()` (which requests the operating system to increase the size of the heap memory segment), on the other hand, do not require any intervention from the host operating system. Instead, the simulator itself will act as the operating system for the executed program.

The simulator’s “kernel” controls the simulated machine in the same way that a real OS kernel controls a real machine. Like a real kernel on a real machine, the simulator’s kernel can direct the simulated machine to begin executing code in user mode at a specific memory address. The machine will return control to the kernel if the simulated user program executes a syscall or trap instruction, or if an interrupt or other machine exception occurs.

Like a real kernel, the simulator’s kernel must examine and modify the machine registers and other machine state in order to service exceptions and run user programs. For example, system call arguments and results are passed between a user program and the kernel through the machine’s registers. The kernel will also modify page table data structures that are used by the simulated machine for translating virtual addresses to physical addresses. From the

perspective of the simulator's kernel, all of the machine state – registers, memory, and page tables – are simply data structures (most likely arrays) in the kernel address space??

**Figure needed here.**

### 6.7.7 Comparison with Hardware Microarchitecture

In the next chapter, we will study hardware microarchitectures, which interpret machine language programs using hardware circuitry. It is informative to compare and contrast the functionality performed in traditional microarchitectures and simulators.

| Attribute            | HW Microarchitecture       | SW Microarchitecture        |
|----------------------|----------------------------|-----------------------------|
| Register space       | Hardware registers         | Simulator program variables |
| Memory address space | Hardware memory structures | Simulator program variables |
| Loading              | Loader part of OS          | Loader part of simulator    |
| Interpretation       | Control unit hardware      |                             |

Table 6.1: A Succinct Comparison of Hardware and Software Microarchitectures

## 6.8 Hardware Design

### 6.8.1 Clock

Computer hardware, like other synchronous digital circuits, use a clock signal to coordinate the actions of two or more circuit blocks. A clock signal oscillates between a high and a low state, normally with a 50% duty cycle. In other words, the signal is a square wave. The circuits using the clock signal for synchronization may become active at either the rising or falling edge, or both, of the clock signal. Processor clock speed or clock rate is an indicator of the speed at which the processor executes instructions. Every computer hardware contains an internal clock that regulates the rate at which instructions are executed and synchronizes all the various computer components. The processor requires several clock ticks (or clock cycles) to execute each instruction. The faster the clock, the more instructions the processor can generally execute per second. Clock speeds are expressed in megahertz (MHz) or gigahertz (GHz).

### 6.8.2 Hardware Description Language (HDL)

The two major HDLs are Verilog and VHDL (Very high speed integrated circuits HDL). Both of these are equally popular and have roughly equal market presence.

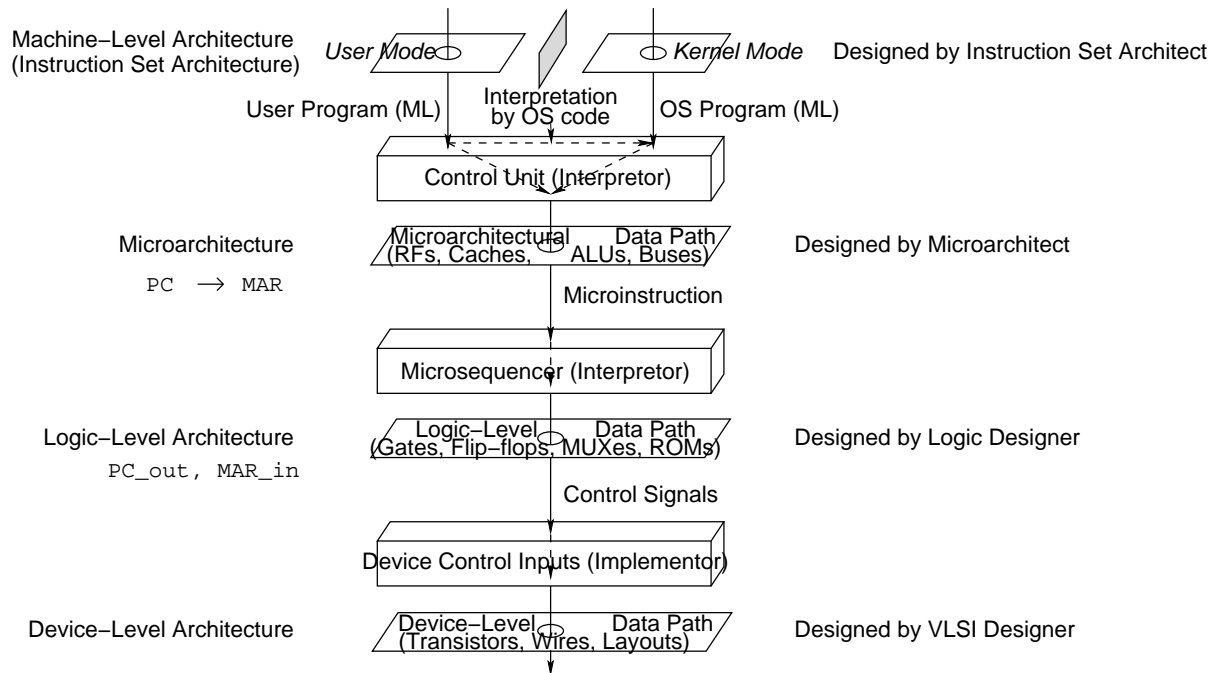


Figure 6.6: Machine Abstractions relevant to Program Execution, along with the Major Components of each Abstract Machine.

### 6.8.3 Design Specification in HDL

### 6.8.4 Design Verification using Simulation

### 6.8.5 Hardware Design Metrics

#### 6.8.5.1 Technology Trends

Moore's Law:

#### 6.8.5.2 Performance

*"A man with a watch knows what time it is.  
A man with two watches is never sure."  
— Segal's Law*

The performance of a computer depends on a number of factors, many of which are related to the design of the processor data path. Three of the most important factors are the strength of the machine language instructions, the number of clock cycles required to fetch

and execute a machine language instruction, and the clock speed. A powerful instruction performs a complex operation and accomplishes more than what a simple instruction accomplishes, although its execution might take several additional clock cycles. The strength of instructions is an issue that is dealt with at the ISA level (taking into consideration microarchitectural issues), and is not under the control of the microarchitect.

Clock speed has a major influence on performance, and depends on the technology used to implement the electronic circuits. The use of densely packed, small transistors to fabricate the digital circuits leads to high clock speeds. Thus, implementing the entire processor on a single VLSI chip allows much higher clock speeds than would be possible if several chips were used. The use of simple logic circuits also makes it easier to clock the circuit faster. Clock speed is an issue that is primarily dealt with at the design of logic-level architectures, although microarchitectural decisions have a strong bearing on the maximum clock speeds attainable.

The third factor in performance, namely the number of clock cycles required to fetch and execute an instruction, is certainly a microarchitectural issue. In the last several years, speed improvements due to better microarchitectures, while less amazing than that due to faster circuits, have nevertheless been impressive. There are two ways to reduce the average number of cycles required to fetch and execute an instruction: (i) reduce the number of cycles needed to fetch and execute each instruction, and (ii) overlap the interpretation of multiple instructions so that the average number of cycles per instruction is reduced. Both involve significant modifications to the processor data path, primarily to add more connectivity and latches.

#### **6.8.5.3 Power Consumption**

#### **6.8.5.4 Price**

### **6.9 Concluding Remarks**

### **6.10 Exercises**

1. Explain the difference between static, load-time, and lazy (fully dynamic) linking. What are the advantages and disadvantages of each type of linking?
2. Write a software simulator program (in any high-level language) that can interpret machine language programs written for the MIPS-0 ISA.



## Chapter 7

# Microarchitecture — User Mode

*Listen to counsel and receive instruction, That you may be wise in your latter days.*

**Proverbs 19: 20**

Executing a program involves several steps, as highlighted in the previous chapter. These steps are carried out by different entities: program selection is typically done by the end user; process creation, program loading, and process halting are typically done by the operating system. Program execution — the bulk of the effort — generally falls on the processor's shoulders<sup>1</sup>. The processor is a hardware entity.

Our objective in this chapter is to implement the machine specification given in the instruction set architecture (ISA). Because of the complexity of this machine, it is impractical to directly design a gate-level circuitry that implements the ISA. Therefore, computer architects have taken a more structured approach by introducing one or more hardware abstraction levels in between<sup>2</sup>. The abstraction level directly below the instruction set architecture is called the *microarchitecture*. This level is responsible for *executing* machine language programs.

In this chapter, we study the organization and operation of the different components that constitute the user mode microarchitecture, the machine that is responsible for implementing the user mode instruction set architecture (ISA)—i.e., carrying out the execution of user mode machine language programs. A microarchitectural view of a computer is restricted to seeing the major building blocks of the computer, such as register files, memory units, ALUs and other functional units, and different buses.

---

<sup>1</sup>Even the steps carried out by the operating system involve work by the processor, as the operating system itself is a process that is executed by the processor, in kernel mode.

<sup>2</sup>Even the design of simple digital circuits involves a somewhat structured approach. When designing a circuit as a finite state machine, we first construct the state transition diagram, and then implement the state transition diagram.

A particular ISA may be implemented in different ways, with different microarchitectures. An executable program developed for this ISA can be executed on any of these microarchitectures without any change, regardless of their differences. For example, the Intel IA-32 ISA has been implemented in different processors such as Pentium III, Pentium 4, and Athlon. Some of these processors are built by Intel, and the others by competitors such as AMD and Cyrix. Even when using the same processor type, computer vendors build many different system microarchitectures by putting together processors, memory modules, and IO interface modules in different ways. One microarchitecture might focus on high performance, whereas another might focus on reducing the cost or the power consumption. The ability to develop different microarchitectures for the same ISA allows processor vendors and memory vendors to take advantage of new IC process technology, while providing upward compatibility to the users for their past investments in software. Although our discussion may seem to hint that the microarchitecture is always implemented in hardware, strictly speaking, the microarchitecture only specifies an abstract model. This abstract machine can be implemented in software if needed. Examples are the SPIM simulator we saw in Chapter 4 (which is a functional simulator) and the SimpleScalar simulator [?] (which is a cycle-accurate simulator).

Like the design of the higher level architectures that we already saw, microarchitecture design is also replete with trade-offs. The trade-offs at this level involve characteristics such as speed, cost, power consumption, die size, and reliability. For general-purpose computers such as desktops, one trade-off drives the most important choices the microarchitect must make: speed versus cost. For laptops and embedded systems, the important considerations are size and power consumption. For space exploration and other critical applications, reliability is of primary concern.

Present day computer microarchitecture is incredibly complex. Fortunately, the two principles that enabled computer scientists to develop high-level architectures that support million-line programs—modularization and partitioning—are also available to computer engineers to design complex microarchitectures that meet various requirements. By breaking up a microarchitecture into multiple blocks, the design becomes much more tractable. For simplifying the discussion, this chapter presents only microarchitectural features that are specific to implementing the user mode features of the ISA. Chapter 8 deals with microarchitectural aspects that are specific to implementing the kernel mode features of the ISA. The current chapter addresses some of the fundamental questions concerning user mode microarchitecture, such as:

- What are the building blocks in a computer microarchitecture, and how are they connected together?
- What steps should the microarchitecture perform to sequence through a machine language program, and to execute (i.e., accomplish the work specified in) each machine language instruction?
- What are some simple organizational techniques that can reduce the number of time-

steps needed to execute each machine language instruction?

- What are some techniques commonly used to reduce the latency of memory accesses?

*Go to the ant, you sluggard; consider its ways and be wise! It has no commander, no overseer or ruler, yet it stores its provisions in summer and gathers its food at harvest.*

**Proverbs 6: 6-8**

## 7.1 Overview of User Mode Microarchitecture

A commonly used approach for designing hardware — especially small to medium size circuits — is the *finite state machine* approach. Finite state machine-based design involves identifying a suitable *state* variable, identifying all possible states, and then developing a state transition diagram. Theoretically, the computer hardware can also be built in this manner as a single finite state machine; it is, after all, a digital circuit. However, such a finite state machine would have far too many states (a single bit change in a single memory location changes the system state), making it extremely difficult to comprehend the complex functionality, let alone design one in an efficient manner.

### 7.1.1 Dichotomy: Data Path and Control Unit

In order to tackle the above state explosion, microarchitects partition the computer hardware into a *data path* and a *control unit*<sup>3</sup>. The data path serves as the platform for executing machine language (ML) programs, whereas the control unit serves as the *interpreter* of ML programs, for execution on the data path. This point needs further clarification. In order to execute a machine language program on the data path, the program has to be interpreted. This interpretation is done by the control unit. The relation between the interpreter and the microarchitecture is pictorially depicted in Figure 7.1.

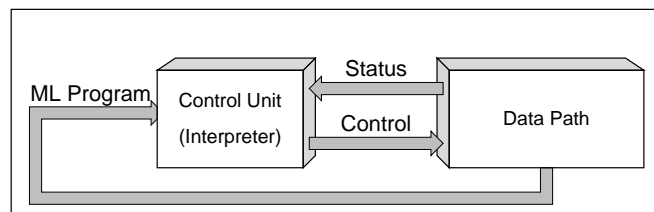


Figure 7.1: Relation between the Data Path and the Control Unit (Interpreter)

<sup>3</sup>The data path itself is partitioned into several smaller data paths (with their own miniature control units) to keep the design even simpler. Thus, the memory unit is designed as a separate data path, with its own memory controller.



The data path incorporates circuitry and interconnections that are required for executing each of the instructions defined in the instruction set architecture, and serves as a platform for executing ML programs. It is a collection of storage components such as registers and memories, functional units such as ALUs, multipliers, and shifters, as well as interconnects that connect these components. The movement of data through the data path is controlled by the control unit. Microarchitects use a language called **micro-assembly language (MAL)** to indicate the control actions specified by the control unit. In order to execute ML programs on the data path, ML programs are interpreted in terms of MAL commands. This interpretation is performed by the control unit, which supplies the MAL commands to the data path. Because a significant portion of the functionality has been shifted to the data path, the control unit can be conveniently built as a finite state machine with a manageable number of states. A change in a main memory value does not cause a state transition in the control unit, for instance.

The sequence of operations performed in the data path is determined by commands generated by the control unit. The control unit thus functions as the interpreter of machine language programs.

The computer data path can be easily divided into major building blocks and subsystems based on how data flow is specified in the ISA. That is, we break the functionality specified in the machine language into different parts, and use separate building blocks to implement each of the parts in the data path. Thus at the system level, we see the computer hardware in terms of major building blocks and their interconnections.

In order to design a data path for implementing a user mode ISA we need to consider what the user mode ISA requires for data to flow through the system. There are two aspects to consider here:

- The storage locations (register name space and memory address space) defined in the user mode ISA
- The operations or functions defined in the user mode ISA.

The data path of a computer is built from building blocks such as registers, memory elements, arithmetic-logic units, other functional units, and interconnections. Assumptions about the behavior of these building blocks become specifications of the data path, which are then conveyed to the logic-level designer of the data path.

### 7.1.2 Register File and Individual Registers

A microarchitecture defines one or more register files, each consisting of several physical registers. The register files are used to implement the integer and floating-point register name spaces defined in the ISA. Figure 7.2 depicts a register file. It has an address input for specifying the register to be read from or written into. Any specific register in the register file can be read or written by placing the corresponding register number or address

in the address input lines. The data that is read or written is transmitted through the **data** lines. The data lines can be bidirectional as shown in the figure, or can be two sets of unidirectional lines.

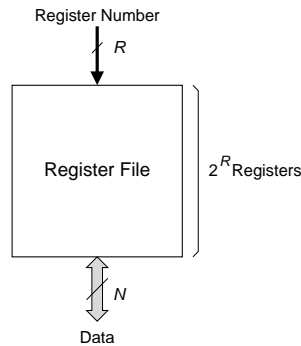


Figure 7.2: An  $N$ -bit wide Register File containing  $2^R$  Registers

Apart from register files, several individual hardware registers are also included in the microarchitecture. Some of these registers can be used to implement other ISA-defined registers such as **pc**, **sp**, and **flags**. Individual registers and register files that implement registers defined in the ISA are often called *ISA-visible registers*.

Register files as well as individual registers that are not ISA-visible are invisible to the machine language programmer, and are called *microarchitectural registers*. These serve as temporary storage for values generated or used in the midst of instruction execution in the microarchitecture. The data path may also need to keep track of special properties of arithmetic and logical operations such as condition codes; most data paths use a register called **flags**<sup>4</sup>. Notice that the total number of registers provided in the microarchitecture is higher than the number of registers defined in the ISA.

### 7.1.3 Memory Structures

A microarchitecture defines one or more memory structures, each containing very large numbers of memory locations. These memory structures are useful for implementing the memory address space and sections defined in the ISA. Each memory structure resembles a very large register file. Like a register file, each memory structure has an address input for specifying the location to be read from or written into. Thus, any specific memory location can be read or written by placing the corresponding memory address in the address input lines. The different storage elements — memory structure(s) as well as register file(s) — are pictorially shown in Figure 7.3.

<sup>4</sup>In some architectures such as the IA-32, the **flags** register is visible at the ISA level (usually as a set of condition codes). In MIPS-I, it is not visible at the ISA level.

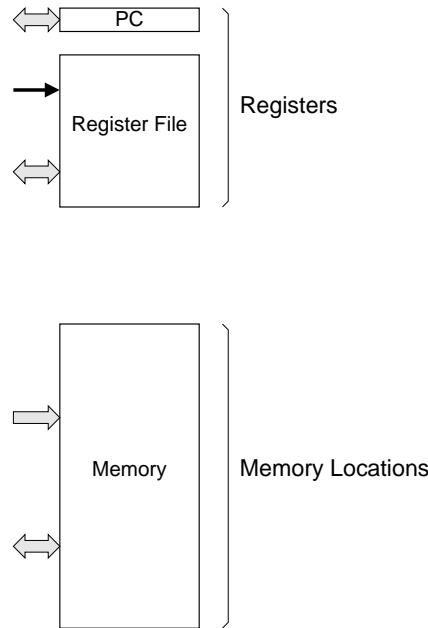


Figure 7.3: Implementing the Storage Locations Defined in the User Mode ISA

#### 7.1.4 ALUs and Other Functional Units

We have just looked at the storage elements present in a microarchitecture. Next, we shall consider the elements that serve to perform the operations and functionalities specified in the ISA. In order to do this, let us review the functions the microarchitecture must perform so as to execute instructions. Some of the important functionalities are listed below.

- Data transfer between memory locations and registers.
- Data transfer between registers
- Arithmetic/logical operations on data present in registers

Next, let us consider equipping the data path to perform arithmetic and logical operations, such as addition, subtraction, AND, OR, and shift. For carrying out operations, the microarchitecture defines hardware circuitry that can perform different operations on bit patterns, and produce result bit patterns in the specific data encodings used. Many microarchitectures consolidate the hardware circuitry for performing arithmetic and logical operations on integer data into a single multi-function block called *arithmetic and logic unit* (ALU). An ALU takes data inputs as well as control inputs, as depicted in Figure 7.4. The two main sets of data inputs are marked as  $X$  and  $Y$ . The control inputs indicate the specific arithmetic/logic function to be performed, and the data type of the inputs. An ALU

typically performs functions such as addition and subtraction (for signed integers as well as unsigned integers), AND, OR, NAND, NOR, left shift, logical right shift, and arithmetic right shift.

Hardware circuitry for performing complex operations such as integer multiplication, integer division, and all types of floating-point operations are usually not integrated into the ALU; instead, these are typically built as separate **functional units**. This is because these operations typically require much longer times than that required for integer addition, integer subtraction, and all logical operations. Integrating all of these operations into a single ALU forces every operation to be as slow as the slowest one.

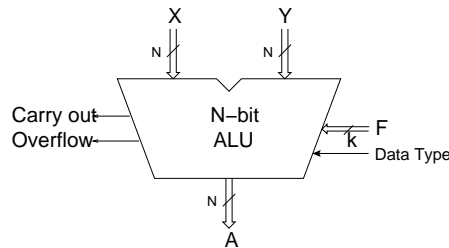


Figure 7.4: An  $N$ -bit ALU Capable of Performing  $2^k$  Functions

### 7.1.5 Interconnects

It is not sufficient to implement the storage locations and the functional units in a microarchitecture. For meaningful operations to be carried out in a data path, it is important that the storage elements and the functional units be connected properly. To achieve a reasonable speed of operation, most of these connections transfer a full word in parallel over multiple wires. The connections can be either *buses* or *direct paths*.

A bus connects together a large number of blocks using a single set of wires, and is a shared communication link. Multiple blocks can communicate over a single bus, at different times. At any particular instant, data can be transferred from a single block to one or more blocks. Thus a bus permits broadcasting of data to multiple destination blocks. In addition to the wires that carry the data, additional wires may be included in a bus for *addressing* and *control* purposes. One point to note here is that a bus allows only a single transfer at a time, and so, if a data path has only a small number of buses, then only very few data transfers can happen in parallel in that data path.

A *direct path*, unlike a bus, is a point-to-point connection that connects exactly two blocks. A direct path-based interconnect therefore consists of a collection of point-to-point connections between different blocks. Direct paths tend to be faster than buses, due to two reasons. First, their wires have only two connections, and this results in less capacitance

than that in buses, which have many connections. Secondly, direct paths tend to be much shorter than buses, and so have lower transmission delays. Direct paths are particularly suited for connecting hardware blocks that are physically close and communicate frequently. The downside is that a direct path based data path tends to have too many paths, requiring a large chip area for the wires.

The different blocks in a microarchitecture can be interconnected in a variety of ways. The type of interconnects and the connectivity they provide determine, to a large extent, the time it takes to execute a machine language instruction. A typical data path has too many hardware blocks for every block to be connected to every other block by point-to-point connections. What would be a good interconnect to use here? The simplest type of connection that we can think of is to use a single bus to connect all of them, as illustrated in Figure 7.5. In this figure, all of the storage elements (including the microarchitectural registers) and the ALU are connected together by a single bus called *system bus*.

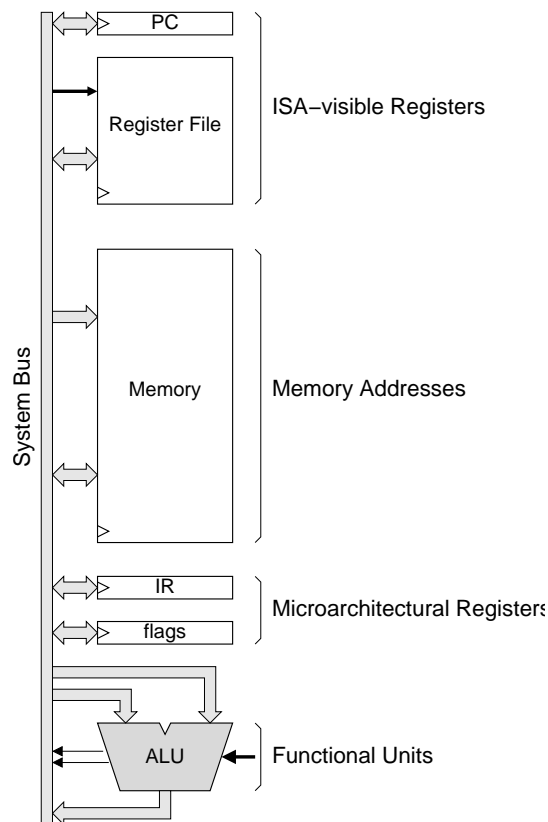


Figure 7.5: Interconnecting the ISA-visible Registers, Microarchitectural Registers, and the ALU using a Single Bus to form a Computer Data Path

### 7.1.6 Processor and Memory Subsystems

In the discussion so far, the registers, the memory structures, and the functional units are all interconnected with a single bus, called the *system bus*. The use of a single bus has several limitations:

- It permits only a single data transfer between the different units at any given instant.
- Some of the transfers may be very fast (those involving only registers), whereas some others (those involving memory) are very slow. The difference in transfer rates necessitates an *asynchronous* protocol for the system bus, which introduces an overhead to the register-only transfers, which are more frequent.
- The bus may have high capacitance due to two reasons: (i) it is very long so as to connect all devices in the system, and (ii) each device that is hooked up to the bus adds to the capacitance. To reduce the capacitance, each device must have expensive low-impedance drivers to drive the bus.

In order to improve performance, we typically organize the user mode microarchitecture as two subsystems: the **processor subsystem** and the **memory subsystem**. This is depicted in Figure 7.6. In such an organization, the registers and functional units are integrated within the processor subsystem, and the memory structures are integrated within the memory subsystem. Because the functional units work with values stored in registers, they are typically included in the processor subsystem where the registers are present. The memory subsystem is traditionally placed outside the processor chip<sup>5</sup>.

Another notable change is that we now have two buses—a *processor bus* and a *system bus*. The processor bus is used to interconnect the blocks within the processor subsystem, and the system bus is used to connect the two subsystems. The processor subsystem also includes a *memory interface* to act as a tie between the two buses. The use of two buses enables multiple transfers to happen in parallel. For instance, data could be transferred from the register file to the ALU, while data transfer is taking place between the memory subsystem and the memory interface.

### 7.1.7 Micro-Assembly Language (MAL)

As mentioned earlier, the data path does not do anything out of its own volition; it merely carries out the elementary instruction that it receives from the control unit. Thus, it is the control unit that decides what function should be carried out by the processor data path in a particular clock cycle. Likewise, the function to be performed by the execution

---

<sup>5</sup>As we will see later in this chapter, the memory subsystem of a typical computer includes multiple levels of cache memories, of which the top one or two levels are often integrated into the processor chip. With continued advancements in VLSI technology, a number of researchers are even working towards the integration of the processor and memory subsystems into a single chip.

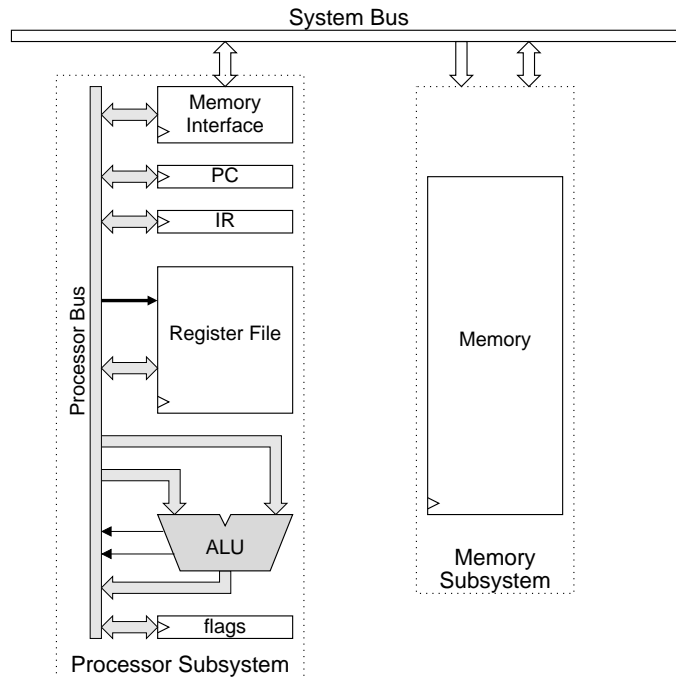


Figure 7.6: Organizing the User Mode Microarchitecture as a Combination of Processor and Memory Subsystems

unit in a clock cycle is also decided by the control unit. For ease of understanding, we shall use an assembly-like language to express these elementary instructions sent by the control unit to the data path. This language uses mnemonics similar to those used in the assembly languages studied in Chapters 3 and 4, and is called a *micro-assembly language* or *MAL*. Accordingly, an elementary instruction represented in this language is called a *micro-assembly language command* or a *MAL command* for short. A MAL command may be composed of one or more elementary operations called *MAL operations*, performed on data stored in registers or in memory. A MAL operation can be as simple as copying data from one physical register to another, or more complex, such as adding the contents of two physical registers and storing the result in a third physical register.

We shall deal with the specifics of the data transfer when we look at RTL architectures in Chapter ??.

## 7.2 Example Microarchitecture for Executing MIPS-0 Programs

Designing a computer microarchitecture is better caught than taught. Accordingly, in this chapter we will design several example microarchitectures, and illustrate the principles involved. Although a microarchitecture can be designed in a generic manner to support a variety of ISAs, such an approach is rarely taken. In practice, each microarchitecture is designed with a specific ISA in mind<sup>6</sup>. The primary reason for this is performance. If we design a generic microarchitecture to support the idiosyncrasies of a number of ISAs (like the XXL size T-shirts distributed during class reunion ceremonies), then that data path is likely to be significantly slower than one that caters to a specific ISA. In accordance with this practice, our microarchitecture designs are also for a specific ISA. For the sake of continuity with the preceding chapters, these microarchitectures are designed for executing MIPS ISA instructions<sup>7</sup>. For simplicity and ease of understanding, we restrict ourselves to a representative subset of the MIPS-I instruction set which we call **MIPS-0**. We will first discuss a simple example microarchitecture in detail, and later move on to more complex microarchitectures. This simple microarchitecture will be used in the discussions of the control unit as well.

In order to design a microarchitecture for the MIPS-0 ISA, we first consider the ISA-defined storage locations that are typically implemented within the processor subsystem. These include the general-purpose registers and the special registers. The MIPS ISA defines 32 general-purpose registers, R0 - R31, and we use a 32-entry register file to implement them. In each clock cycle, this register file can accept a single address input for specifying the register to be read from or written into. A special register called PC is used to store the memory address of the next instruction to be interpreted and executed. Apart from these ISA-visible registers, we shall include the following microarchitectural register: **flags** to store important properties of the result produced by the ALU. The contents of the **flags** register can be used to perform selective updates to PC.

We shall use a single multi-function ALU (Arithmetic and Logic Unit) to perform the arithmetic and logical operations specified in the MIPS-0 ISA. More advanced designs may use a collection of specialized functional units.

We shall design the data path as a combination of a processor subsystem and a memory subsystem, as discussed in Section 7.1.6. The processor subsystem includes the registers, the ALU, and the interface to the memory subsystem. We shall use a single 32-bit processor bus to interconnect the hardware structures inside the processor subsystem, and another

---

<sup>6</sup>This is in contrast to the practice followed in designing an assembly-level architecture, where the design is not tailored for any particular high-level language.

<sup>7</sup>The microarchitecture presented in this section is somewhat generic, and is *not* necessarily close to the ones present in any of the commercial MIPS processors. In Sections 7.6.2 and 7.7, we present microarchitectures that are tailored for the MIPS ISA, and are therefore closer to commercial MIPS processor microarchitectures.



bus — the system bus — to connect the processor subsystem to the memory subsystem.

Figure 7.7 pictorially shows one possible way of interconnecting the main building blocks so as to perform the required functions. This figure suppresses information on how the control unit controls the functioning of the blocks; these details will be discussed later. The use of a bus permits a full range of paths to be established between the blocks connected by the bus. It is quite possible that some of these paths may never have to be used. The direct path based data path that we will see later in this chapter eliminates all such unnecessary paths by providing independent connections only between those blocks that need to communicate.

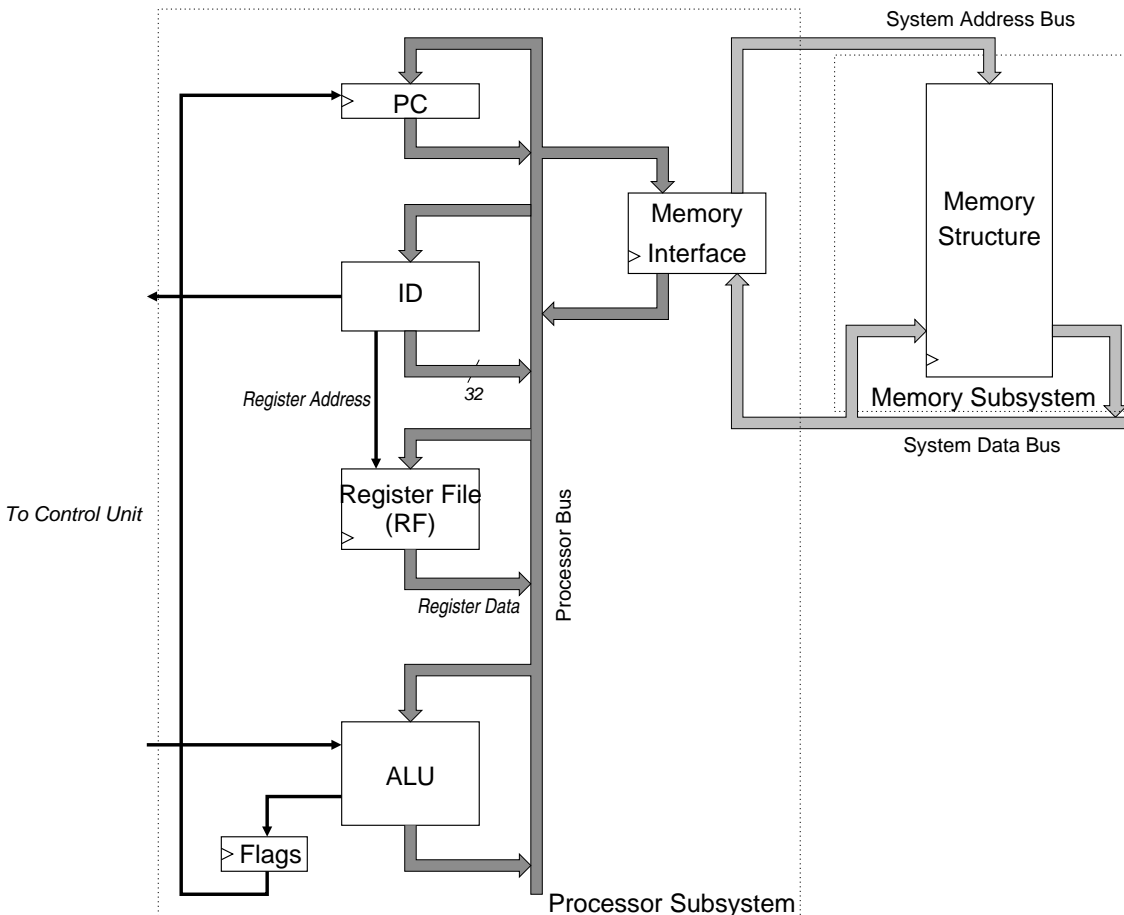


Figure 7.7: A Microarchitecture for Implementing the MIPS-0 User Mode ISA

Finally, the data path includes the memory subsystem, which implements the memory address space defined in the user mode ISA. There is a **memory interface** unit for interfacing the processor bus to the memory subsystem.

### 7.2.1 MAL Commands

The process of executing a machine language (ML) instruction on the data path can be broken down into a sequence of more elementary steps. We saw in Section 7.1 how we can use a micro-assembly language (MAL) to express these elementary steps. We can define a set of MAL operations for the MIPS-0 data path defined in Figure 7.7.

If the interconnections of the data path are rich enough to allow multiple MAL operations in the same time period, these can be denoted by writing them in the same line, as follows:

**Fetch instruction;    Increment PC**

specifies that in the same step, the next instruction is fetched from memory, and the contents of PC are incremented by 4.

Finally, for MAL operations that are conditional in nature, an “if” construct patterned after the C language’s “if” construct is defined.

**if (src1 == src2)    Update PC**

indicates that if the zero flag is equal to 1, the contents of AOR is copied to PC; otherwise no action is taken.

### 7.2.2 MAL Operation Set

Table 7.1 gives a list of useful MAL operations for the microarchitecture of Figure 7.7. The MAL operations done (in parallel) in a single step form a *MAL command*.

### 7.2.3 An Example MAL Routine

Now that we are familiar with the syntax of MAL as well as the useful MAL operations for the microarchitecture of Figure 7.7, we shall look at a simple sequence of MAL operations called a MAL routine. We shall write a MAL routine that adds the contents of registers specified in the **rs** and **rt** fields of the instruction, and writes the result into the register specified in the **rd** field of the instruction. The astute reader may have noticed that executing this MAL routine amounts to executing the MIPS-I machine language instruction **and rd, rs, rt** in the microarchitecture of Figure 7.7, provided the binary pattern of this instruction has been fetched and decoded.

In theory, we can write entire programs in MAL that can perform tasks such as ‘printing “hello, world!” ’ and more; we will, of course, need a special storage for storing the MAL programs and a mechanism for sequencing through the MAL program. Having seen the difficulty of writing programs in assembly language and machine language, one can easily imagine the nightmare of writing entire programs in MAL! However, developing a MAL program may not be as bad as it sounds, given that we can develop translator software such as assemblers that take machine language programs and translate them to MAL programs.

| No. | MAL Command |              | Comments                                   |
|-----|-------------|--------------|--|
| 0   | Fetch       | Instruction  | Instruction = Memory[PC]                   |
| 1   | Decode      | Instruction  | Determine opcode, src1, src2, dest, offset |
| 2   | Add         | src1, src2   | Result = R[rs] + R[rt]                     |
| 3   | Add         | src1, offset | Result = R[rs] + sign-extended offset      |
| 4   | And         | src1, src2   | Result = R[rs] AND R[rt]                   |
| 5   | And         | src1, offset | Result = R[rs] AND sign-extended offset    |
| 6   | Compute     | MemAddress   | Address = R[rs] + sign-extended offset     |
| 7   | Compute     | BranchTarget | Target = PC + 4 + 4 × sign-extended offset |
| 8   | Compute     | JumpTarget   | Target = PC + 4 + 4 × sign-extended offset |
| 9   | Load        |              | R[rt] ← Memory[Address]                    |
| 10  | Store       |              | Memory[Address] ← R[rt]                    |
| 11  | Write       |              | Write result to register dest              |
| 12  | Save        | PC           | R[31] ← PC                                 |
| 13  | Increment   | PC           | PC ← PC + 4                                |
| 14  | Update      | PC           | PC ← Target                                |

Table 7.1: A List of Useful MAL Operations for the Microarchitecture of Figure 7.7

| Step | MAL Command |           | Comments  |
|------|-------------|-----------|---|
| 0    | Read        | registers | Read registers rs and rt                                      |
| 1    | Compute     | result    | AND the operand values  |
| 2    | Write       |           | Write result into destination register (rd) if it is non-zero |

Table 7.2: An Example MAL Routine for Executing an Instruction in the Microarchitecture of Figure 7.7

The real difficulty is that MAL programs will be substantially bigger than the corresponding ML programs, thereby requiring very large amounts of storage. This is where *interpretation* comes in. By generating the required MAL routines on-the-fly at run-time, the storage requirements are drastically reduced, as we will see next.

### 7.3 Interpreting ML Programs by MAL Routines

Having discussed the basics of defining MAL routines, our next step is to investigate how machine language programs can be interpreted by a sequence of MAL commands that are defined for a particular data path. To that end, we will take individual MIPS-0 instructions and how MAL commands can be put together to carry out its execution in the data path. Before developing the MAL command routines, let us briefly review the functions the data path must perform so as to execute instructions one by one. As seen in Section 6.5, the

interpretation process involves two major functions:

- **Fetching the instruction:** This concerns fetching the next instruction in the program.
- **Executing the instruction:** This concerns executing the fetched instruction.

These actions are typically done as a *fetch-execute* sequence. We also saw that these two functions can be further divided into a sequence of steps (some of which are optional), as indicated below.

#### Executing the next instruction

- **Fetch Phase**

1. *Fetch instruction:* Read the next instruction from the main memory into the processor data path.
2. *Decode instruction:* Decode the instruction bit pattern so as to determine the action specified by it.

- **Execute Phase**

1. *Fetch source operand values:* Fetch source operands (if any) from registers, main memory, or IO registers.
2. *Process data:* Perform arithmetic or logical operation on source operand values, if required.
3. *Write result operand value:* Write the result of an arithmetic or logical operation to a register, memory location, or IO register, if required.
4. *Update instruction address:* Determine the memory address of the next instruction to be interpreted.

These 6 steps serve as directives for the data path in its attempt to carry out the execution of each instruction. In order to execute an entire ML program, which is just a sequence of instructions, the interpretation should also implement the *sequencing* among instructions. That is, after the completion of the above 6 steps for a single instruction, it should go back to the fetch phase to begin executing the next instruction in the ML program. Thus, the *fetch-execute* sequence becomes a *fetch-execute* cycle.

All of these steps can be accomplished by a sequence of MAL commands called a **MAL routine**. A MAL routine is allowed to freely use and modify any of the ISA-invisible microarchitectural registers. The ISA-visible registers, however, can be modified only as per the semantics of the instruction being interpreted.

### 7.3.1 Interpreting an Instruction — the Fetch Phase

When implementing ISAs with fixed length instructions, the actions required to perform the fetch phase of the interpretation are the same for all instructions. The MIPS-0 ISA is no exception. We shall consider this phase of instruction interpretation first. In the data path of Figure 7.7, register PC keeps the memory address of the next instruction to be fetched. To fetch this instruction from memory, we have to utilize the memory interface provided in the data path. Table 7.3 gives a MAL routine that implements the fetch phase of executing the instruction. This routine also includes the MAL command for incrementing PC after the fetch phase so as to point to the sequentially following instruction, which is normally the instruction executed after the execution of the current instruction. If the current instruction is a branch instruction, then the PC value may be modified in the *execute* phase.

| Step                    | MAL Command        | Comments                           |
|-------------------------|--------------------|------------------------------------|
| <i>Fetch and Decode</i> |                    |                                    |
| 0                       | Fetch instruction  |                                    |
| 1                       | Decode instruction | Determine opcode, src1, src2, dest |
| <i>PC increment</i>     |                    |                                    |
| 2                       | Increment PC       | Increment PC by 4                  |

Table 7.3: A MAL Routine for the Fetching a MIPS-0 ISA Instruction in the Microarchitecture of Figure 7.7

The first step involves getting the binary pattern corresponding to the instruction from the memory<sup>8</sup>. In this MAL routine, we consider the time taken to perform this MAL command as one unit; the exact number of cycles taken depends on the specifics of the memory system used and the RTL sequence used. In step 1, the fetched instruction bit pattern is decoded by the instruction decoding circuitry. The exact manner in which instruction decoding is performed is not specified here; this will be relevant only to the lower level of design. The **opcode** and **funct** fields of the fetched instruction uniquely identify the instruction. Steps 0-1 thus constitute the MAL routine for the fetch phase of instruction interpretation. Naturally, this portion is the same for every instruction in the MIPS ISA because all instructions have the same size. Had the MIPS ISA used variable length instructions, this fetch process would have to be repeated as many times as the number of words in the instruction.

After interpreting an instruction, the interpreter needs to go back to step 0 to interpret the next instruction in the ML program. However, prior to that, it has to increment PC to point to the next instruction; otherwise the same instruction gets interpreted repeatedly. In this MAL routine, PC update is done immediately after completing the instruction fetch.

---

<sup>8</sup>We have used a single MAL command, namely **Fetch instruction**, to tell the microarchitecture to fetch an instruction. At the register transfer level (RTL), this MAL command may be implemented by a sequence of RTL instructions.

Thus, in step 2, PC is incremented by 4 to point to the next instruction in the executed machine language program. After step 2, the interpreter goes to the *execute phase*.

This MAL routine has 3 steps. We can, in fact, perform the **Increment PC** function in parallel to the instruction fetch or decode functions, and reduce the total number of steps required for the interpretation. Table 7.4 provides the modified MAL routine, which requires only 2 steps. In this routine, in step 0, PC is incremented at the same time the instruction is fetched from memory. It is important to note that to do multiple MAL operations in parallel, the microarchitecture needs to have appropriate connectivity. In general, the more the connectivity provided in a data path, the more the opportunities for performing multiple MAL operations in parallel.

| Step                                   | MAL Command                     | Comments |
|--|---------------------------------|----------|
| <i>Fetch, Decode, and PC increment</i> |                                 |          |
| 0                                      | Fetch instruction; Increment PC |          |
| 1                                      | Decode instruction              |          |

Table 7.4: An Optimized MAL Routine for Fetching a MIPS-0 ISA Instruction in the Microarchitecture of Figure 7.7

### 7.3.2 Interpreting Arithmetic/Logical Instructions

We just saw a MAL routine for the fetch phase of instruction interpretation. Next, let us consider the *execute phase* of instructions. Unlike the actions in the fetch phase, the actions in the *execute phase* are not identical for different instructions. Therefore, the MAL routines for the execute phase are different for the different instructions. We shall consider one instruction each from the four types of instructions: (i) data transfer instruction, (ii) arithmetic/logical instruction, (iii) control flow changing instruction, and (iv) syscall instruction.

Let us start by considering an arithmetic instruction. We shall put together a sequence of MAL commands to interpret an arithmetic/logical instruction.

Example: Consider the MIPS ADDU instruction whose symbolic representation is **addu rd, rs, rt**. Its encoding is given below.

|        |    |    |    |  |     |
|--------|----|----|----|--|-----|
| 000000 | rs | rt | rd |  | ADD |
|--------|----|----|----|--|-----|

The fields **rs**, **rt**, and **rd** specify register numbers; the first two of these contain the data values to be added together as unsigned intergers. The **rd** field indicates the destination register, i.e., the register to which the result should be written to, as long as it is not register \$0. In this data path, the addition of the two register values can be done in the ALU.

Table 7.5 specifies a sequence of MAL commands for carrying out the execute phase of this instruction in the data path of Figure 7.7. Let us go through the working of this MAL routine. We name the first MAL command of the routine as *step 2*, as the execute phase is a continuation of the fetch phase, which ended at step 2.

| Step                 | MAL Command                  | Comments  |
|----------------------|------------------------------|---|
| <i>Execute phase</i> |                              |   |
| 2                    | Read <code>src1, src2</code> | Read operands <code>R[rs]</code> and <code>R[rt]</code>                   |
| 2                    | Add                          | Perform arithmetic operation  |
| 3                    | Write                        | Write result to register <code>rd</code> , if <code>rd</code> is non-zero |

Table 7.5: A MAL Routine for Executing the MIPS-0 Instruction Represented Symbolically as `addu rd, rs, rt`. This MAL Routine is for executing the instruction in the Data Path of Figure 7.7

Step 2 begins the execution phase. First, the register operands must be fetched from the register file. In step 2, the operand values present in general-purpose registers `rs` and `rt` are read, and are added together in the ALU. In step 3, the result of the addition is written to the destination register (`rd`), if `rd` is non-zero. By performing this sequence of MAL commands in the correct order, the instruction `addu rd, rs, rt` is correctly interpreted.

### 7.3.3 Interpreting Memory-Referencing Instructions

Let us next put together a sequence of MAL commands to fetch and execute a memory-referencing instruction. Because all MIPS instructions are of the same length, the MAL routine for the fetch part of the instruction is the same as before; the differences are only in the execution part. Consider the MIPS load instruction whose symbolic representation is `lw rt, offset(rs)`. The semantics of this instruction are to copy to GPR `rt` the contents of memory location whose address is given by the sum of the contents of GPR `rs` and sign-extended `offset`. We need to come up with a sequence of MAL commands that effectively fetch and execute this instruction in the data path of Figure 7.7.

|        |    |    |        |
|--------|----|----|--------|
| 100011 | rs | rt | offset |
|--------|----|----|--------|

The interpretation of a memory-referencing instruction requires the computation of an address. For the MIPS ISA, address calculation involves sign-extending the `offset` field of the instruction to form a 32-bit signed offset, and adding it to the contents of the register specified in the `rs` field of the instruction. In this data path, the address calculation is done using the same ALU, as no separate adder has been provided. With this introduction, let us look at the MAL routine given in Table 7.6 to interpret this `lw` instruction.

In step 2, the memory address is computed by reading the contents of register specified

| Step                 | MAL Command        | Comments                                      |
|----------------------|--------------------|---|
| <i>Execute phase</i> |                    |   |
| 2                    | Compute MemAddress | Memory address = R[rs] + sign-extended offset |
| 3                    | Load               | Load from memory into register rt             |

Table 7.6: A MAL Routine for the Execute Phase of the Interpretation of the MIPS-0 ISA Instruction Represented Symbolically as `lw rt, offset(rs)`. This MAL Routine is for executing the instruction in the Data Path of Figure 7.7

in the `rs` field and adding the sign-extended `offset` value to it. In step 3, the contents of the memory location at the computed address is loaded into register `rt`. Recall that the memory transfer takes place through the memory interface.

### 7.3.4 Interpreting Control-Changing Instructions

The instructions that we interpreted so far — `addu` and `lw` — do not involve control flow changes that cause deviations from straightline sequencing in the ML program. Next let us see how we can interpret control-changing instructions, which involve modifying PC, usually based on a condition.

*Example:* Consider the MIPS-0 conditional branch instruction whose symbolic representation is `beq rs, rt, offset`. The semantics of this instruction state that if the contents of GPRs `rs` and `rt` are equal, then the value `offset`  $\times 4 + 4$  should be added to PC so as to cause a control flow change<sup>9</sup>; otherwise, PC is incremented by 4 as usual. The encoding of this instruction is given below:

|        |    |    |        |
|--------|----|----|--------|
| 000100 | rs | rt | offset |
|--------|----|----|--------|

| Step                 | MAL Instruction      | Comments                         |
|----------------------|----------------------|----------------------------------|
| <i>Execute phase</i> |                      |                                  |
| 2                    | Compute BranchTarget |                                  |
| 3                    | Compare src1, src2   | Compare operands R[rs] and R[rt] |
| 4                    | Update IfEqual       | Update PC if operands are equal  |

Table 7.7: A MAL Routine for Executing the MIPS-0 ISA Instruction Represented Symbolically as `beq rs, rt, offset`. This MAL Routine is for executing the instruction in the Data Path of Figure 7.7

<sup>9</sup>The actual MIPS ISA uses a *delayed branch* scheme; i.e., the control flow change happens only after executing the instruction that follows the branch instruction in the program. We avoid delayed branches to keep the discussion simple.



Table 7.7 presents a MAL routine to execute this instruction in the data path given in Figure 7.7. The execute routine has 3 steps numbered 2-4. The first step of the routine (step 2) calculates the target address of the branch instruction by adding 4 times the sign-extended `offset` value to the incremented PC value. In step 3, the contents of registers `rs` and `rt` are compared, and the result of the comparison is stored in the `flag` register. In the last step, PC is updated with the calculated target address, if the two operands were equal. Thus, if the operands turned out to be not equal, then PC retains the incremented value it obtained in step 1, which is the address of the instruction following the branch in the machine language program being interpreted.

### 7.3.5 Interpreting Trap Instructions

We have seen the *execute phase* for all of the instruction types other than the `syscall` instructions. Let us next look at how the data path performs `syscall` execution, which is somewhat different from the previously seen ones. Like control-changing instructions, `syscall` instructions also involve modifying the contents of `pc`. For the MIPS-I ISA, the `pc` is updated to `0x80000080`. In addition, the machine is placed in the kernel mode. We shall take a detailed look at the corresponding MAL sequence in Section 8.1.1, along with other kernel mode implementation issues.

## 7.4 Memory System Organization

The memory system is an integral component of the microarchitecture of a stored program computer, as it stores the instructions and data of the program being executed. In this section we look at the microarchitecture of this subsystem more closely. This section begins by illustrating the need to organize the physical memory system as a hierarchy, in order to achieve high speed without incurring high cost. Then it describes how different components of the memory hierarchy work. The next section provides an in-depth treatment of cache memories, and demonstrates how they help to increase the apparent speed of the memory system.

*“In a hierarchy every employee tends to rise to his level of incompetence.”*  
— Dr. Laurence Peter, 1919-90, in *The Peter Principle*, 1969

### 7.4.1 Memory Hierarchy: Achieving Low Latency and Cost

The discussion we had so far may seem to indicate that the main memory is implemented as a single structure in a computer microarchitecture. That is, a single memory structure implements the entire address space, provides fast access, and is inexpensive. Unfortunately, this ideal situation is feasible only in some small systems that have a small address space and work with a slow processor. For general-purpose computers, it is impossible to meet

all three of these requirements simultaneously with today's semiconductor technology. As might be expected, there is a trade-off among these three key characteristics, because of the following relationships:

- larger a memory structure, slower its operation.
- larger a memory structure, greater its cost
- faster a memory structure, greater its cost

*“Everyone who comes in here wants three things:  
(1) They want it quick.  
(2) They want it good.  
(3) They want it cheap.  
I tell 'em to pick two and call me back.”  
— A sign on the back wall of a small printing company*

When we implement the entire address space using a single memory structure, a memory access cannot be completed in 1 cycle, contrary to what we had in mind when writing the MAL routines. Instead, a memory access will take anywhere from 50-100 processor clock cycles, because the memory structure is quite large and also requires off-chip access. Furthermore, because the memory address space defined in ISAs is often quite large, it may not be cost-effective to implement the entire address space using semiconductor memory.

Unlike the predicament of the printing company in the above quote, we do have a solution for the memory subsystem to deal with the problems introduced by the three relationships given above. To motivate this solution, let us consider a simple analogy that illustrates the key principles and mechanisms well. Many of you may be using an *Address Book* that stores information about frequently used telephone numbers or email addresses, in order to reduce the time spent in going through a large telephone directory. Once this Address Book has been initialized, chances are high that most of the time you will get the required telephone number or address directly from this list, without going through the telephone directory. Keeping the frequently accessed information as a separate list results in significant time savings, compared to going through the telephone directory each time.

The same principle can be used to create the illusion of a large memory that can be accessed as quickly as a small memory. Just like we do not access every number in the telephone directory with equal probability at any given time, a program does not access all of its program or data locations with equal probability at any given time. The **principle of locality** underlies both the way in which we deal with telephone numbers and the way that programs operate. This principle indicates that programs access a relatively small portion of their address space at any instant of time. There are two different types of locality:

- *Temporal locality* (locality in time): If an item is referenced now, it is likely to be accessed again in the near future.

- *Spatial locality* (locality in space): If an item is referenced now, items whose addresses are close by are likely to be accessed in the near future.

Just as accesses to telephone numbers exhibit locality, locality in programs arises from simple and natural program structures. For example, most programs contain loops. Therefore, instructions and data are likely to be accessed repeatedly, resulting in high amounts of temporal locality. Analyses of program executions show that about 90% of the execution time is generally spent on 10% of the code. This may be due to simple loops, nested loops, or a few subroutines that repeatedly call each other. Also, because instructions are normally accessed sequentially, instruction accesses show high spatial locality. Accesses to data also exhibit a natural spatial locality. For example, accesses to elements of an array or a record will naturally have high degrees of spatial locality.

Much work has gone into developing clever structures that improve the apparent speed of the memory, without terribly increasing the cost, by taking advantage of the principle of locality. The solution that has been widely adopted is not to rely on a single memory component or technology, but to use a *memory hierarchy*. A memory hierarchy consists of multiple levels of memory with different speeds and sizes, as illustrated in Figure 7.8. The fastest memories are more expensive per bit than the slower ones, and are usually smaller. The goal is to achieve a performance close to that of the fastest memory, and a cost per bit close to that of the least expensive memory.

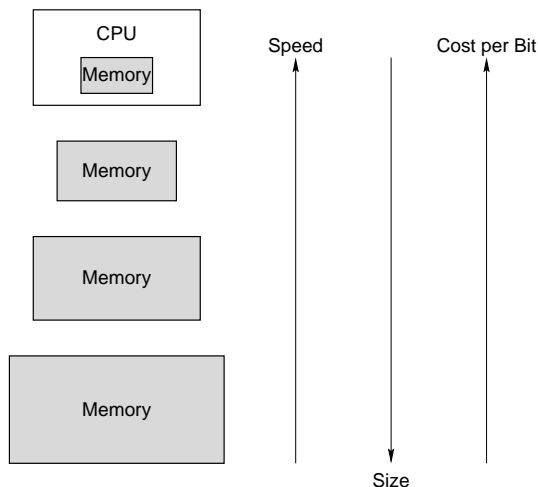


Figure 7.8: The Basic Structure of a Typical Memory Hierarchy

A memory hierarchy can consist of multiple levels, but data is copied only between two adjacent levels at a time. The fastest elements are the ones closest to the processor. The top part of the hierarchy will supply data most of the time because of the principle of locality. Memory hierarchies take advantage of temporal locality by keeping recently accessed data

items closer to the processor. They take advantage of spatial locality by moving blocks consisting of multiple contiguous words to upper levels of the hierarchy, even though only a single word was requested by the processor. If the fraction of times a memory request is satisfied at the upper level is high, the memory hierarchy has an effective access time close to that of the highest (and fastest) level and an effective size equal to that of the lowest (and largest) level.

### 7.4.2 Cache Memory: Basic Organization

Over the years, processor speeds have been increasing at an astronomical rate. Main memory speeds, however, have not been increasing at a comparable rate. Today, it takes anywhere from 50-200 processor cycles to access a word from main memory. A common solution adopted to bridge this gap in speeds is to insert a relatively small, high-speed memory, called *cache memory*, between the processor and the main memory. The cache memory keeps a copy of the frequently accessed memory locations. When the processor issues a request to access a memory location, the cache memory responds if it contains the requested word. If it does not contain the requested word, the request is passed on to main memory.

Conceptually, the operation of cache memory is very simple. The temporal aspect of the locality of reference suggests that whenever the contents of a location is needed for the first time, this item should be brought into the cache so that the cache can supply the value the next time the same location is accessed. The spatial locality suggests that instead of bringing just one item from the main memory to the cache memory, it is wise to bring several nearby items as well. We will use the term **block** to refer to a set of contiguous addresses. Another term that is often used to refer to a block is **line**.

Consider the simple arrangement shown in the previous figure. When the processor issues a Read request to a memory location for the first time, the contents of the specified location and the surrounding block of locations are transferred to the cache, one word after another. Subsequently, when the program references any of the words in this block, the requested word is read directly from the cache, and the request is not forwarded to main memory.

Notice that cache memory is a microarchitectural feature to enhance performance, and is not part of most instruction set architectures (ISAs). Thus, the machine language programmer does not need to know the existence of the cache. The processor simply issues Read and Write requests using main memory addresses. The cache control circuitry determines if the requested word currently exists in the cache. If the cache contains the word, a *cache hit* is said to have occurred. If it does not, then a *cache miss* is said to have occurred.

### 7.4.3 MIPS-0 Data Path with Cache Memories

#### 7.4.4 Cache Performance

A basic question concerning a cache memory system is how to quantify its performance. The figure of merit of interest to us is the *average memory access time*, considering the cache memory and the main memory together as a single system. A simple formula for this metric is given by:

$$T_{EFF} = T_{CA} + m \times T_{Miss}$$

where  $T_{CA}$  is the access time of the cache,  $T_{Miss}$  is the time taken to service a cache miss, and  $m$  is the miss ratio (i.e., the fraction of accesses that resulted in a cache miss).

Example: In order to improve the memory access time from 50 cycles, a system designer decided to incorporate a cache memory, which has an access time of 4 cycles. The cache was able to supply data for 80% of the memory references. Has the cache memory improved the memory system's performance or decreased it?

When the system did not use cache memory, each memory access took 50 cycles. When a cache memory is introduced, the cache miss ratio is 20%. Therefore, the average memory access time becomes  $4 + 0.2 \times 50 = 14$  cycles. Thus, in this case, the introduction of the cache memory improves the average access time from 50 cycles to 14 cycles. If the cache memory has a higher hit ratio and/or a lower access time, then the improvement will be even more marked.

#### 7.4.5 Address Mapping Functions

The cache memory is obviously much smaller than the main memory, which means that the maximum number of blocks it can store at any time is less than the total number of blocks present in the main memory. This has two major implications:

- We cannot use the straightforward *linear addressing* methodology for accessing the cache memory. For instance, if we want to access memory address 1 billion, we cannot just go to the 1 billionth location in the cache. Thus, there has to be some mapping between the main memory blocks and the physical frames of the cache. In an Address Book, for instance, this mapping is typically based on the first letter of the person's name, as illustrated in Figure 7.9; the information pertaining to a person whose name starts with "A" is entered in the page corresponding to letter "A". Below, we will look at address mapping schemes that are typically used in cache memories.
- Multiple main memory blocks can map to the same cache block frame, at different times. This implies that there must be some mechanism to identify the main memory block that is currently mapped to each cache clock frame. Again, in Figure 7.9, we can see that two different names are mapped to the page corresponding to letter "A".

The name currently mapped to the page is identified by explicitly storing the name — which serves as a *tag* — along with the address and phone number — which form the **data**.

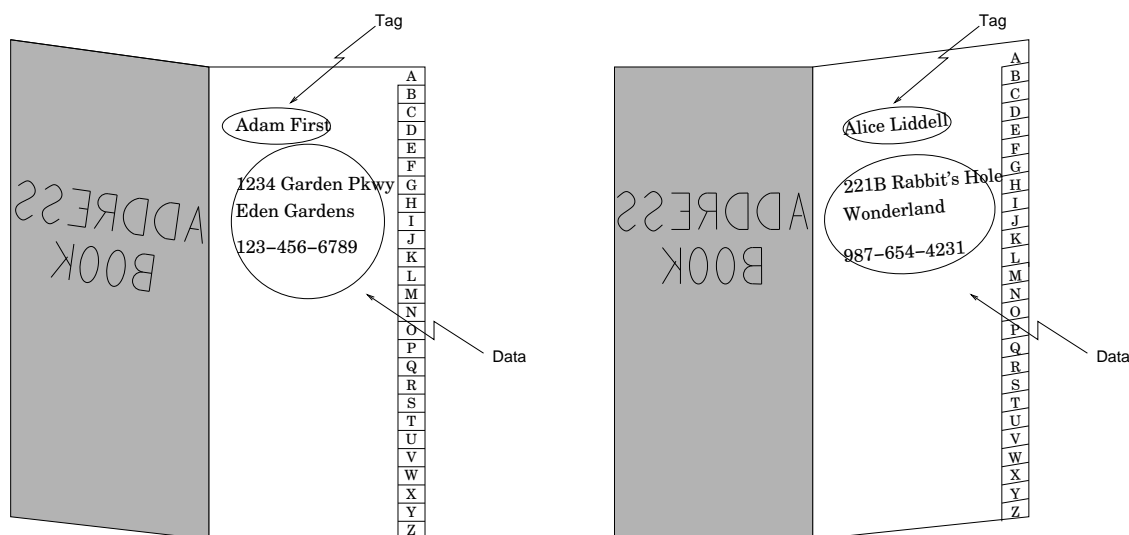


Figure 7.9: An Alphabetically Organized Address Book Containing Two Different Name Entries

Identification of the memory block that is currently mapped to a cache block frame is typically achieved by including a **Tag** field in each cache block frame. When a new memory block is copied to a cache block frame, the **Tag** field is updated appropriately. A **Valid** bit is also included for indicating if the cache block frame currently has a valid main memory block.

The correspondence between the main memory blocks and those present in the cache is specified by a mapping function. When the cache is full and the processor accesses a word that is not present in the cache, the cache control hardware must decide which block should be removed to create space for the new block that contains the referenced word. The collection of rules for making this decision constitutes the **replacement algorithm**.

To discuss possible methods for specifying where main memory blocks are placed in the cache, we use a specific small example. Consider a cache consisting of 8 blocks of 32 words each, for a total of 256 words, and assume that the main memory is addressable by a 10-bit address. The main memory has 1M words, which can be viewed as 32K blocks of 32 words each.

The simplest way to map main memory blocks to cache block frames is the **fully associative mapping** technique. In this mapping, a main memory block can be placed in any

cache block frame, as shown in the first part of Figure 7.10. Therefore, a new memory block will replace an existing block only if the cache is full. However, the cost of an associative cache is high because of the need to search all cache block frames to determine if a given main memory block is in the cache. A search of this kind is called an *associative search*. For performance reasons, the tags must be searched in parallel.

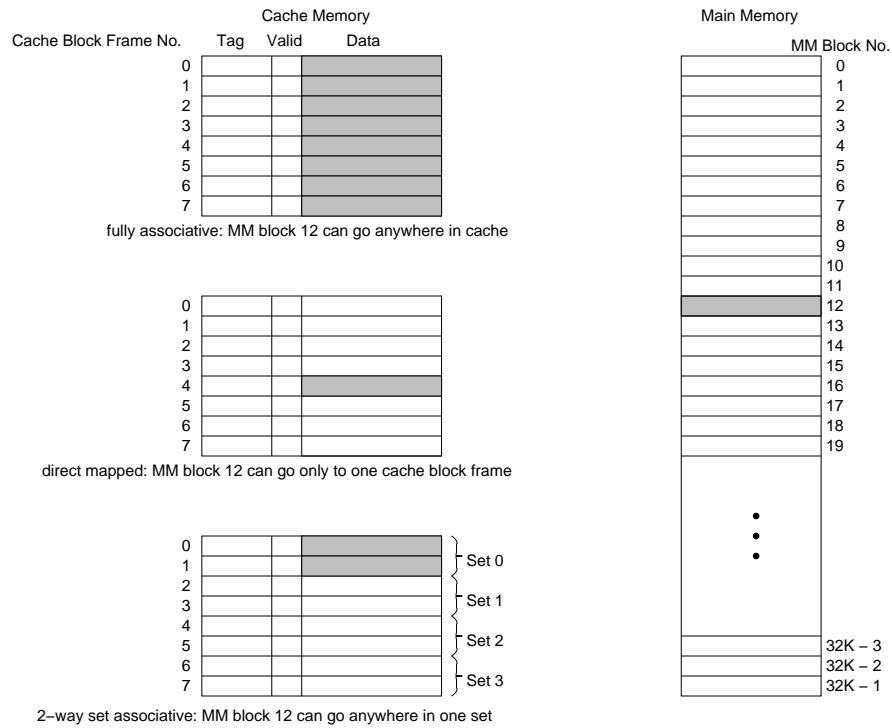


Figure 7.10: Possible Mappings between Main Memory and Cache Memory

An alternative approach is the **direct mapping** technique, depicted in the second part of Figure 7.10. In this technique, a particular main memory block maps to exactly one cache block frame. The commonly used hash function is such that main memory block  $i$  maps onto cache block frame  $i \bmod \text{Num\_Cache\_Blocks}$ . Thus, it is easy to determine if a particular main memory block is present in the cache or not. However, because a main memory block can be placed only in one particular cache block frame, contention may arise for that position even if the cache is not full or if another cache block frame has an unused main memory block.

*“He who always puts things in place, is too lazy to look for them.”*  
— Anonymous

**Set associative mapping** is a compromise that captures the advantages of both the fully associative and the direct mapping approaches. Cache blocks are grouped into **sets**, and the mapping allows a main memory block to reside in any block of a specific cache set. Hence, the contention problem of the direct mapping method is reduced by having some choices for block replacement. At the same time, the hardware cost of fully associative mapping is reduced by decreasing the size of the associative search.

#### 7.4.6 Finding a Word in the Cache

How does the cache memory find a memory word if it is contained in the cache? Given a memory address, the cache first finds the memory block number to which the address belongs. This can be determined by dividing the memory address by the block size. Because it is time-consuming to do an integer division operation, cache designers only use block sizes that are powers of 2. The division, then, becomes taking out the upper bits of the memory address. Figure 7.11 shows how a memory address bit pattern is split. The first split is between the **Main Memory Block Number** and the **Offset within Block**. The **Main Memory Block Number** field is further split into the **Tag** field and the **Cache Set Number** field. The **Offset within Block** field selects the desired word once the block is found, the **Cache Set Number** field selects the cache set, and the **Tag** field is compared against the tag values in the selected set to check if a hit occurs.

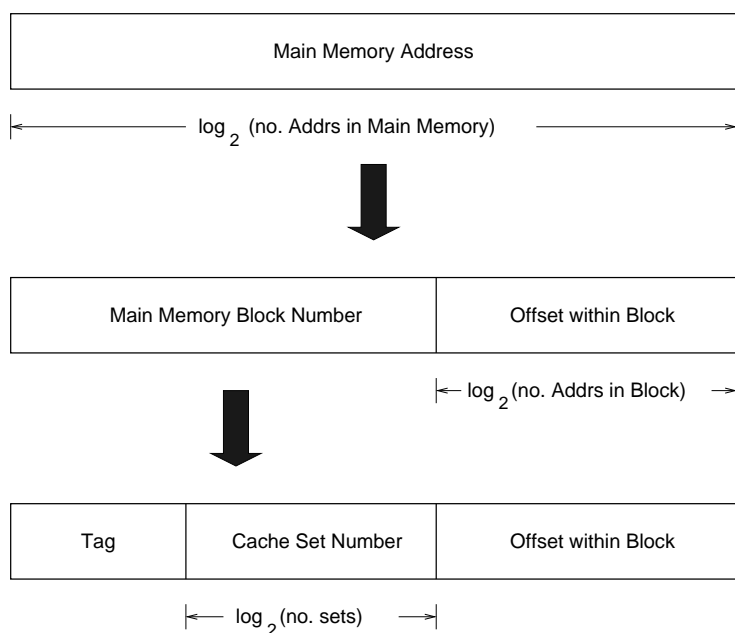


Figure 7.11: Splitting a Memory Address Bit Pattern for Accessing the Cache Memory



For a given total cache size, increasing the associativity reduces the number of cache sets, thereby decreasing the size of the **Cache Set Number** field and increasing the size of the **Tag** field. That is, the boundary between the **Tag** and **Cache Set Number** fields moves to the right with increasing associativity, with the limit case of fully associative caches having no **Cache Set Number** field (because the entire cache is just one set).

*Example:* Consider a computer system that has a 4-way set-associative cache. The cache receives 16-bit addresses, and splits it as follows: 5 bits for finding the **Offset within Block**; 8 bits for finding the **Cache Set Number**.

1. How many sets are present in the cache memory?  
Because 8 bits are decoded to determine the **Cache Set Number**, the cache has  $2^8 = 256$  sets.
2. How many words are present in a block?  
Because 5 bits are decoded to determine the **Offset within Block**, each block has  $2^5 = 32$  words.
3. How many block frames are present in the cache memory?  
The number of block frames in a cache memory is given by the product of the number of sets it has and the number of blocks in each set. Therefore, this cache has  $256 \times 4 = 1024$  blocks.
4. How many words are present in the cache memory?  
The number of words in a cache memory is given by the product of the number of block frames it has and the number of words in each block. Therefore, this cache has  $1024 \times 32 = 32\text{K}$  words.
5. What is the cache set number corresponding to hexadecimal address FBFC?  
The hexadecimal address FBFC corresponds to bit pattern 1111101111111100. The **Cache Set Number** is given by bits 12-5, which is the 8-bit pattern 11011111. In hexadecimal number system, this bit pattern is represented as DF.

### 7.4.7 Block Replacement Policy

When a new memory block is to be brought into a cache set, and all the block frames that it may occupy are occupied by other memory blocks, the cache controller must decide which of the old blocks should be replaced. In a direct mapped cache, there is exactly one cache block frame where a particular main memory block can be placed; hence there is no need for a replacement strategy. In fully associative and set-associative caches, on the other hand, there is a choice in determining the block to be replaced. A good replacement strategy is important for the cache memory to perform well. In general, the objective is to keep in the cache those blocks that are likely to be re-referenced in the immediate future. However, it is not easy to determine which blocks will be re-referenced in the future.

*“The best thing about the future is that it comes to us only one day at a time.”*  
 — Abraham Lincoln

The property of locality of references suggests that blocks that have been referenced recently are very likely to be re-referenced in the near future. This is illustrated in the following pattern of accesses to three memory addresses  $\{X, Y, Z\}$  which belong to three different blocks that map to the same cache block frame.

XXXYYYYYYXXXXZZZZ

In this pattern, except for the few times a transition happens from one address to another, the memory block being referenced is the same as the one referenced during the last access. Therefore, when a block is to be replaced in a set, it is sensible to replace the one that has not been referenced for the longest time in that set. This block is called the *least recently used block (LRU)*, and this replacement policy is called the **LRU replacement policy**.

The LRU replacement policy works well for most scenarios. It works poorly, however, for some scenarios. Consider, for instance, a 2-way set associative cache in which accesses to addresses  $\{X, Y, Z\}$  follow a different pattern:

XYZXYZXYZXYZXYZ

In this case, whenever an access to  $Z$  causes a miss,  $X$ 's block is the LRU block. Interestingly, this is the block to be retained in the cache, for the next access to be a hit. The LRU replacement policy is a poor choice for this kind of access pattern, although it is a repeating pattern.

#### 7.4.8 Multi-Level Cache Memories

### 7.5 Processor-Memory Bus

The next topic that we look at is the interconnection between the processor and the memory system, which includes the cache memories. For a variety of reasons, this connection is often done via a bus called the processor-memory bus. (In Intel x86-based computers, this bus is called the **Front Side Bus** or **FSB**.) A bus, as we saw earlier, is a shared communication link, connecting multiple devices using a single set of wires. In addition to wires, a bus also includes hardware circuitry for controlling access to the bus and the transfer of information over the wires, as per specific protocols. The processor-memory bus consists of 3 sets of wires: *address*, *data*, and *control*. The address lines carry the address of the memory location to be accessed. The data lines carry data information between the source and destination. The control lines are used for transmitting signals that control data transfer as per specific protocols. They specify the times at which the processor and the memory interfaces may place data on the bus or receive data from the bus.

In a desktop environment, the processor-memory bus is almost always contained within the motherboard; several sockets are provided along the bus for inserting the memory

modules. The bus speed is generally matched to the memory system so as to maximize processor-memory bandwidth and minimize memory latency. Processor-memory buses are often ISA-specific, because their structure is closely tied to the ISA.

Bus design and operation are sufficiently complex subjects that entire books have been written about them. The major issues in bus design are bus width, bus clocking, bus arbitration, and bus operations. Each of these issues has a substantial impact on the speed and bandwidth of the bus. We will briefly examine each of these.

### 7.5.1 Bus Width

Bus width is an important microarchitectural parameter. The wider the bus, the more its potential bandwidth. The problem, however, is that wide buses require more wires than narrow ones. They also take up more physical space (e.g., on the motherboard) and need bigger connectors. All of these factors make the wider bus more expensive. To cut costs many system designers tend to be shortsighted, with unfortunate consequences later. There are two ways to increase the data bandwidth of a bus: decrease the bus cycle time (more transfers/sec) or increase the data bus width (more bits/transfer). Speeding up the bus is possible, but difficult because the signals on different lines travel at slightly different speeds, a problem known as *bus skew*. The faster the bus, the more serious bus skew becomes. Another problem with speeding up the bus is that doing this will not be backward-compatible. Old boards designed for the slower bus will not work with the new one. Therefore the usual approach to improving bus performance is to add more data lines.

To get around the problem of very wide buses, sometimes bus designers opt for a **multiplexed bus**. In this design, instead of having separate address and data lines, a common set of lines is used for both address and data. At the start of a bus operation, the lines are used for the address. Later on, they are used for data. Multiplexing the lines in this manner reduces bus width (and cost), but results in a slightly slower system. Bus designers have to carefully weigh all these options when making choices.

### 7.5.2 Bus Operations

The simplest bus operation involves a master and a slave. The master, which in this case can be the processor or the cache controller, initiates a read/write request by placing the address of the location in the address lines of the bus. For a write operation, in addition the data to be written is placed on the data lines. The master also activates the control lines to indicate if it is a read or write operation. When the slave becomes ready, it performs the read/write to the location specified in the address lines.

**Block Transfers or Burst Transfers:** Normally, one word is transferred in one transfer. The throughput of such a bus is limited by the protocol overhead associated with each transaction. Each transfer requires arbitration for bus mastership, transmission of an

address, and slave acknowledgement, in addition to the transfer of the actual data. This type of transfer is particularly wasteful for transferring contiguous blocks of data, such as what happens during cache refill for servicing cache misses. In such situations, only the starting address and word count need be specified by the master. In order to improve the throughput for such situations, processor-memory buses generally support block transfer transactions that effectively amortize the transaction overhead over larger data blocks. When a block read operation is initiated, the bus master tells the slave how many words are to be transferred, for example by initially placing the word count on the data lines. Instead of returning a single word, the slave returns one word at a time until the count has been exhausted.

**Split Transaction:** In the bus operations discussed so far, the bus is tied up for the entire duration of a data transfer, that is, from the time the request is initiated until the time the transfer is accomplished. No other transfers can be done or even initiated during this period. If a memory module has a 50-cycle access time, for example, a read transaction might take a total of about 55 cycles—the majority of which constitute idle time. To improve the bus efficiency under such circumstances, certain buses split conventional transfer operations into two parts: a request transaction and a response transaction. The intent is to relinquish the bus in between these two transactions, allowing it to be used for other transactions in the interim. Such a bus is called a split-transaction bus. A variety of issues must be addressed to make a split-transaction bus work. First, in general, the bus protocols must work even when several pending operations are directed at a single slave module. This typically involves pipelined operation of the slave itself. In addition, if the bus is to support out-of-order responses from slave modules, there must be some provision to tag each response with the identity of the request it corresponds to.

**Read-Modify-Write Operation:** Other kinds of bus cycles also exist. For example, to support synchronization in a multiprocessor system, many ISAs provide an instruction like `TEST_AND_SET (X)`. The semantics of this instruction is: read the value at memory location `X`, and set it to 1, iff the read value is 0. The read and write operations together form an *atomic* operation. Executing this instruction requires a bus read cycle and a bus write cycle. It is important that after the read cycle is performed, and before the write cycle is performed, no other device reads or modifies memory location `X`. To implement this instruction correctly, multiprocessor bus systems often have a special read-modify-write bus cycle that allows a processor to read a word from memory, inspect and modify it, and write it back to memory, all without releasing the bus. This type of bus cycle prevents competing processors from being able to use the bus and thus interfere with the first processor's operation.

## 7.6 Processor Data Path Interconnects: Design Choices

The previous sections discussed the basics of designing a data path for interpreting machine language instructions. We purposely used a very simple data path for ease of understanding. Although such a data path may not provide high performance, it may be quite sufficient for certain embedded applications. The data path of modern general-purpose computers, on the other hand, is far more complex. In this section and the one following it, we will introduce high-performance versions of the processor data path. The processor plays a central role in a computer microarchitecture's function. It communicates with and controls the operation of other subsystems within the computer. The performance of the processor therefore has a major impact on the performance of the computer system as a whole.

### 7.6.1 Multiple-Bus based Data Paths

As discussed in Chapter 6, three important factors that determine the performance of a computer are the strength of the machine language instructions, the number of clock cycles required to fetch and execute a machine language instruction, and the clock speed. Among these three, the number of cycles required to fetch and execute an instruction can be reduced by redesigning the data path so that several micro-operations can be performed in parallel. The more interconnections there are between the different blocks of a data path, the more the data transfers that can happen in parallel in a clock cycle. The number of micro-operations that can be done in parallel in a data path does depend on the connectivity it provides. In a bus-based processor data path, the number of buses provided is probably the most important limiting factor governing the number of cycles required to interpret machine language instructions, as most of the microinstructions would need to use the bus(es) in one way or other.

A processor data path with a single internal bus, such as the one given in Figure 7.7, will take several cycles to fetch the register operands and to write the ALU result to the destination register. This is because it takes one cycle to transfer each register operand to the ALU, and one cycle to transfer the result from the ALU to a register. An obvious way to reduce the number of cycles required to interpret instructions is to include multiple buses in the data path, which makes it possible to parallelly transfer multiple register values to the ALU. Before going into specific multiple bus-based data paths, let us provide a word of caution: buses can consume significant chip area and power. Furthermore, they may need to cross at various points in the chip, which can make it difficult to do the layout at the device level.

### A 2-Bus Processor Data Path

The single-bus processor data path given in Figure 7.7 can be enhanced by adding one more internal bus. The additional bus can be connected to the blocks in different ways. Figure

7.12 pictorially shows one possible way of connecting the second bus. In the figure, the new bus is shown in darker shade, and is used primarily for routing the ALU result back to different registers. A unidirectional path is provided from the first bus to the second. Notice that to perform a register read and a register write in the same clock cycle, the

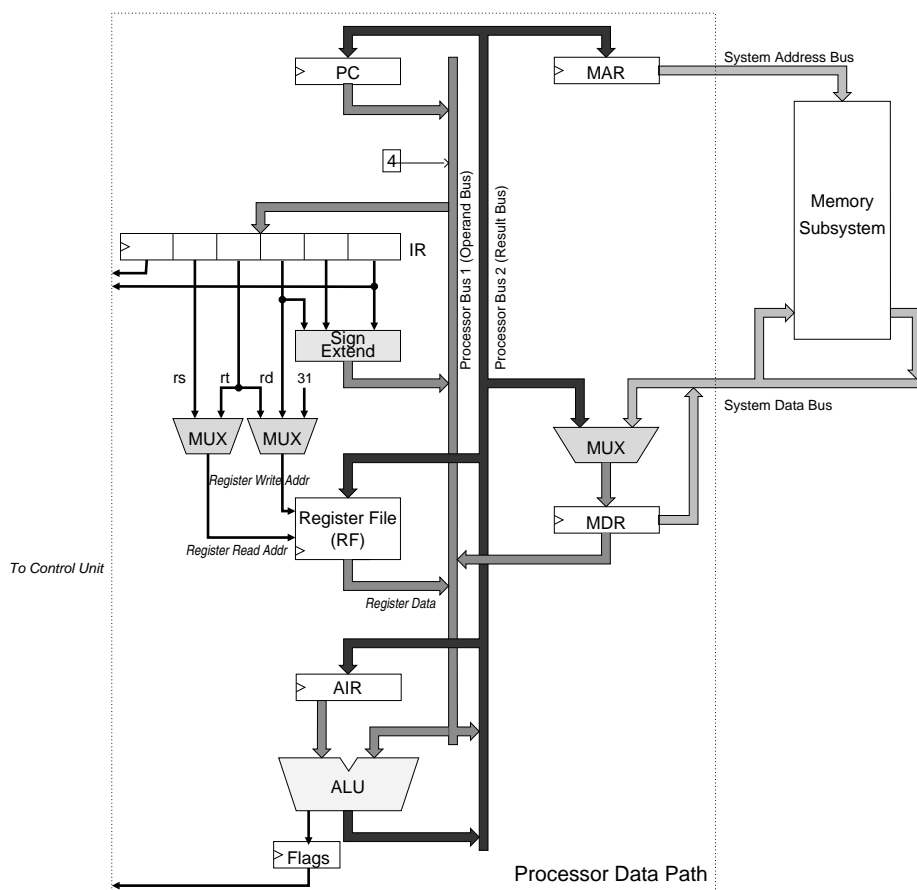


Figure 7.12: A 2-Bus Based Processor Data Path for Implementing the MIPS-0 ISA

register file requires two ports (a read port and a write port).

### 7.6.2 Direct Path-based Data Path

As we keep adding more buses to the processor data path, we eventually get a data path that has many point-to-point connections or direct paths. If we are willing to use a large number of such direct path connections, it is better to redesign the processor data path.

Figure 7.13 presents a direct path-based data path for the MIPS-0 ISA. In this data path, instead of using one or more buses, point-to-point connections or direct paths are provided between each pair of components that transfer data. This approach allows multiple data transfers to take place simultaneously, thereby speeding up the execution of instructions, as we will see in Chapter 9. For instance, the MAL command `ComputeAddress` requires several data transfers, which can only be done sequentially in a single-bus based data path. The use of direct paths probably makes the data path's functioning easier to visualize, because each interconnection is now used for a specific transfer as opposed to the situation in a bus-based data path where a bus transaction changes from clock cycle to clock cycle. Notice, however, that a direct path based data path is even more tailored to a particular ISA.

Notice that there are two paths emanating from the register file. If both paths are to be active at the same time, then the register file needs to have two *read ports*.

## 7.7 Pipelined Data Path: Overlapping the Execution of Multiple Instructions

The use of multiple buses and direct paths in the processor data path enables us to reduce the number of steps required to fetch and execute instructions. Can we further reduce the number of steps by increasing the connectivity? It may be possible; however, it is very difficult, because many of the steps are somewhat sequential in nature. Instead of reducing the number of steps required to fetch and execute each instruction, we can, however, reduce the total number of steps required to fetch and execute a sequence of instructions by overlapping the interpretation of multiple instructions. In order to overlap the execution of multiple instructions, a commonly used technique is **pipelining**, similar in spirit to the assembly-line processing used in factories.

In the data paths that we studied so far, after an instruction is fetched and passed onto the instruction decoder, the fetch part of the processor sits idle until the execution of the instruction is completed. By contrast, in a pipelined data path, the fetch part of the data path utilizes this time to fetch the next several instructions. Similarly, after the instruction decoder has decoded an instruction, it starts decoding the next instruction, without waiting for the previous instruction's execution to be completed.

It is important to understand that pipelining does not result in individual instructions being executed faster; rather, it increases the rate at which instructions are interpreted. With a four-stage pipeline, for instance, instruction execution can be completed up to four times the rate of an unpipelined processor. Based on this discussion, it may appear that the performance obtained with pipelining increases linearly with the number of pipeline stages. Unfortunately, this is not the case. For a variety of reasons, a pipeline stage may not be able to perform useful work during every clock cycle. This causes the pipelined data path's performance to be less than the maximum possible.

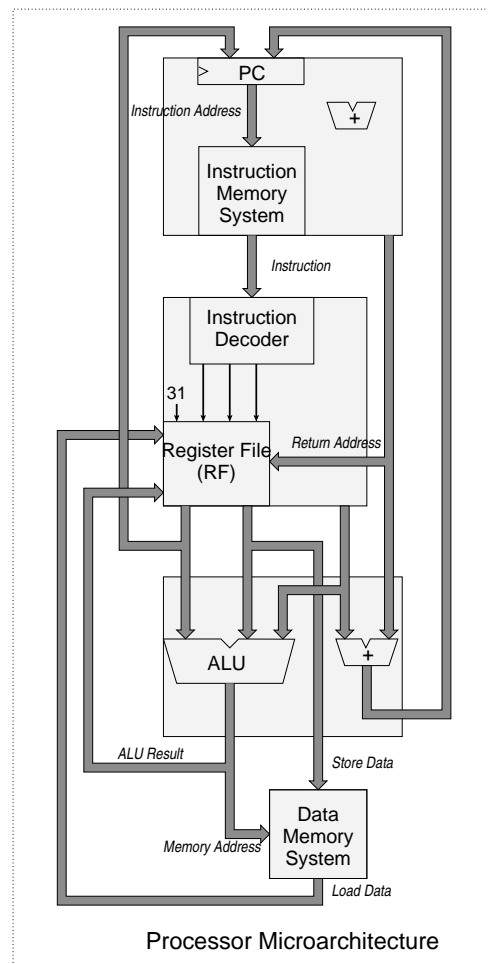


Figure 7.13: A Direct Path-Based Processor Microarchitecture for Implementing the MIPS-0 ISA

### 7.7.1 Defining a Pipelined Data Path

To begin with, we need to determine which hardware blocks in the data path are used during every clock cycle, and make sure that the same hardware block is not used for two different instructions during the same clock cycle. Ensuring this becomes difficult if the same hardware resource is used in multiple cycles during the execution of an instruction. This makes it difficult to use a bus-based data path, as a bus is generally used multiple times during the execution of an instruction. Therefore, our starting point is the direct path-based data path that we saw in Figure 7.13. What would be a natural way to partition this data path into different stages? While we can think of many different ways to partition this data



path, a straightforward way to partition is as follows: place the hardware resources that perform each of the MAL commands of the MAL routines (in Table ??) in a different stage. Thus, we can think of a 5-stage pipeline as depicted in Figure 7.14. The 5 stages of the pipeline are named *fetch* (F), *read source registers* (R), *ALU operation* (A), *memory access* (M), and *end* (E).

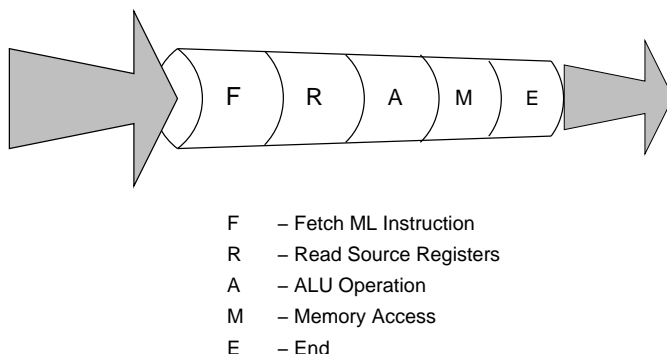


Figure 7.14: Illustration of a 5-Stage Pipelined Data Path

Next, let us partition the direct path-based data path of Figure 7.13 as per this 5-stage framework. Figure 7.15 shows such a partitioning. In this pipelined data path, the instruction fetch portion of the data path—which includes PC and its update logic—has been demarcated as the F stage. IR, which stores the fetched instruction, forms a buffer between the F stage and the R stage. The instruction decoder and the register file have been placed in the R stage. Similarly, the ALU is placed in the A stage. AOR, which stores the output of the ALU, forms a buffer between the A stage and the M stage, which houses the logic for accessing the data memory. Finally, the E stage just consists of the logic for updating the register file. Table 7.8 shows the primary resources used in each stage of the pipeline.

| Stage         | Primary Resources                  |
|---------------|------------------------------------|
| Fetch         | PC, Instruction memory             |
| Read register | Register file, Sign extension unit |
| ALU           | ALU                                |
| Memory access | Data memory, Register file         |

Table 7.8: Primary Resources Used in Each Stage of the Pipelined Data Path of Figure 7.15

In the pipelined data path being discussed, an instruction uses the major hardware blocks in different clock cycles, and hence overlapping the interpretation of multiple instructions introduces relatively fewer conflicts. Some of the hardware blocks in the direct path-based

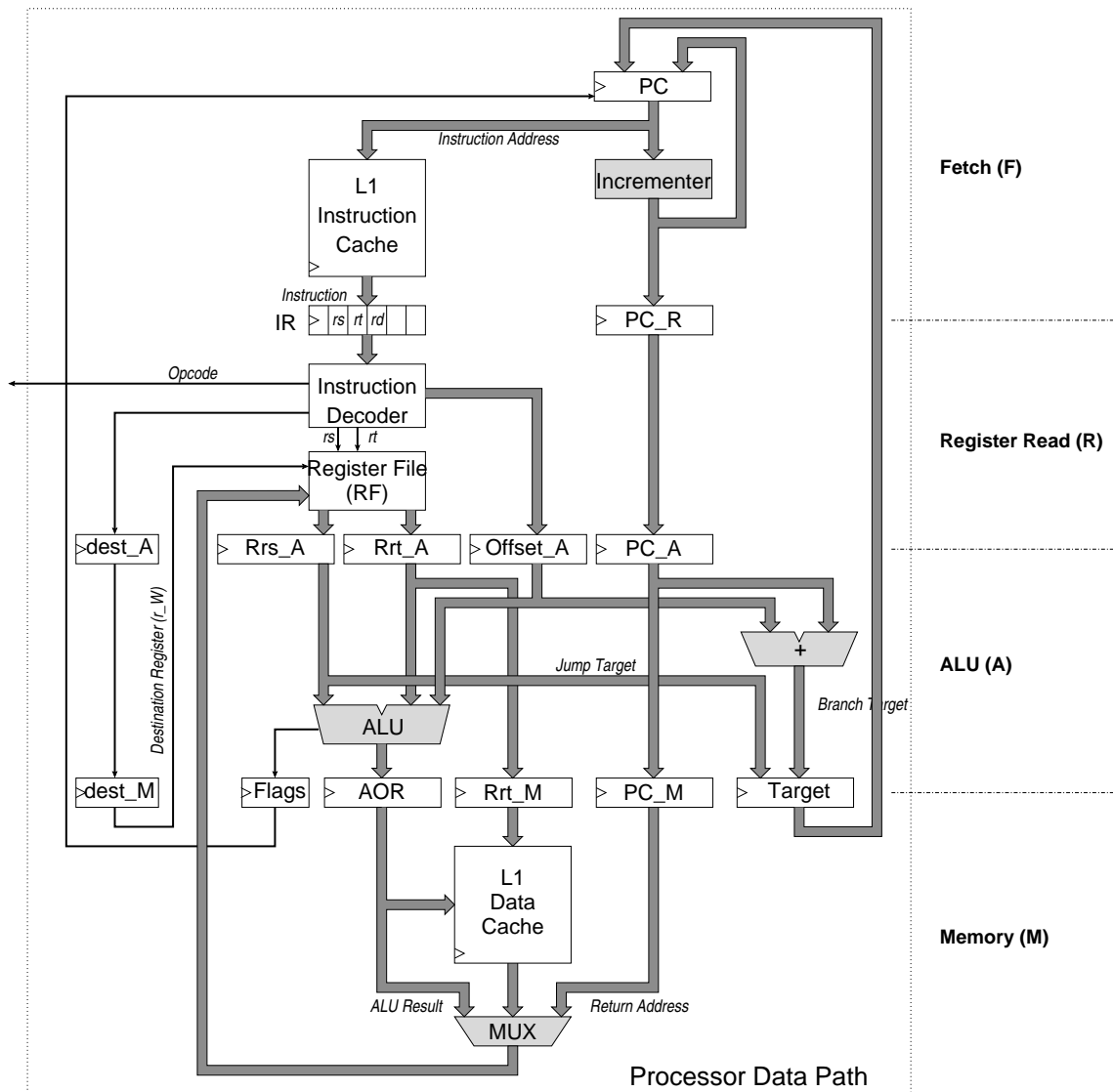


Figure 7.15: A Pipelined Processor Microarchitecture for Implementing the MIPS-0 ISA

data path are used multiple times while interpreting some instructions. In order to avoid conflicts for these hardware blocks, they have been replicated in the pipelined data path. For example, a single PC cannot store the addresses of two different instructions at the same time. Therefore, the pipelined data path must provide multiple copies of PC, one for each instruction that is being executed in the processor.

Pipelining has been used since the 1960s to increase instruction throughput. Since then,

a number of hardware and software features have been developed to enhance the efficiency and effectiveness of pipelining. Not all pipelined data paths have all of these characteristics; the ones we discuss below are quite prevalent.

**Multi-ported Register File:** When multiple instructions are being executed in a pipelined data path, the same hardware block cannot be used for two instructions in the same clock cycle. This requirement is straightforward to meet if each hardware block is accessed at most in one pipeline stage. A prime example of this is the ALU. It is required for an instruction only in the A stage. An inspection of Table 7.8 reveals that the register file, on the other hand, may be used by instructions present in two different stages of the pipeline—stages R and M. Thus, in any given clock cycle, when the instruction in the R stage is trying to read from the register file, a previous instruction in the M stage may be attempting to write to the register file. The solution adopted in our data path is to make the register file *multi-ported*. That is, the register file permits multiple accesses to be made in the same clock cycle<sup>10</sup>.

**Separate Instruction Cache and Data Cache:** Just like the case with register file accesses, memory accesses may also be performed from multiple pipeline stages — instruction fetch from the F stage and data access from the M stage. If a single memory structure is used for storing both the instructions and the data, then the data access done by a memory-referencing instruction (load or store) will happen at the same time as the instruction fetch for a succeeding instruction. The data path of Figure 7.15 used two different memory structures—an L1 instruction cache and an L1 data cache—to avoid this problem. Another option is to use a single dual-ported memory structure.

**Interconnect:** Pipelined data paths typically use the direct path-based approach discussed earlier, which uses direct connections between the various hardware blocks. A bus-based data path is not suitable for pipelining, because only one data transfer can take place in a bus in a clock cycle. The direct path approach allows data transfers to take place simultaneously in each pipeline stage, which indeed they must if the goal of allowing many activities to proceed simultaneously is to be achieved. In fact, we have to add even more interconnections that in a non-pipelined data path.

**Pipeline Registers or Latches:** When the processing of an instruction shifts from one pipeline stage to the next, the vacated stage is utilized for processing the next instruction. Thus, consecutive stages of the pipeline contain information pertaining to different instructions. If adequate buffering is not provided between consecutive stages, information pertaining to different instructions can mix together, leading to incorrect execution.

---

<sup>10</sup>Making a register file multi-ported increases its size as well as the time it takes to read from or write to the register file.

Therefore, it is customary to separate consecutive stages by latches or registers. Because an instruction can “occupy” only one pipeline stage at any given time, whatever information is needed to execute that instruction at a subsequent stage must be carried along until that stage is reached. This is the case even if the information is not required at every stage through which it passes. Each instruction must “carry its own load,” so to speak. Such information might include opcode, data values, etc. A convenient way of propagating this information is to insert extra *pipeline latches* between the pipeline stages. For example, the incremented PC value is carried along, with the help of pipeline registers PC\_R, PC\_A, and PC\_M.

**Additional Hardware Resources:** Pipelined data paths usually need additional hardware resources to avoid resource conflicts between simultaneously executed instructions. A likely candidate is a separate incrementer to increment the PC, freeing the ALU for doing arithmetic/logical operations in every clock cycle.

### 7.7.2 Interpreting ML Instructions in a Pipelined Data Path

The objective of designing a pipelined data path is to permit the interpretation of multiple instructions at the same time, in a pipelined manner. The MAL routines used for interpreting each instruction is similar to those for the direct path-based data path. Table 9.13 illustrates how the MAL routines for multiple instructions are executed in parallel.

| Step No. | MAL Command                     |                                 |  |
|----------|---------------------------------|---------------------------------|--|
|          | lw rt, offset(rs)               | addu rd, rs, rt                 | beq rs, rt, offset   |
| 0        | Fetch instruction               |                                 |  |
| 1        | Read registers                  | Fetch instruction               |  |
| 2        | Rrs_E + Off_E $\rightarrow$ AOR | Read registers                  | Fetch instruction  |
| 3        | DM[AOR] $\rightarrow$ R[r_W]    | Rrs_E + Rrt_E $\rightarrow$ AOR | Read registers   |
| 4        |                                 | AOR $\rightarrow$ R[r_W]        | Rrs_E == Rrt_E $\rightarrow$ Z<br>PC_E + Off_E $\times$ 4 $\rightarrow$ Target |
| 5        |                                 |                                 | if (Z) Target $\rightarrow$ PC   |

Table 7.9: Overlapped Execution of MAL Routines for Fetching and Executing Three MIPS-0 Instructions in the Pipelined Data Path of Figure 7.15

### 7.7.3 Control Unit for a Pipelined Data Path

Converting an unpipelined data path into a pipelined one involves several modifications, as we just saw. The control unit also needs to be modified accordingly. We shall briefly look at how the control unit of a pipelined data path works. With a pipelined data path, in each clock cycle, the control unit has to generate appropriate microinstructions for interpreting

the instructions that are present in each pipeline stage. This calls for some major changes in the control unit, perhaps even more than what was required for the data path. A simple approach is to generate at instruction decode time all of the microinstructions required to interpret that instruction in the subsequent clock cycles (in different pipeline stages). These microinstructions are passed on to the pipelined data path, which carries them along with the corresponding instruction by means of extended pipeline registers. Then, in each pipeline stage, the appropriate microinstruction is executed by pulling it out of its pipeline register.

An alternate approach is for each pipeline stage to have its own control unit, so to speak. Depending on the opcode of the instruction that is currently occupying pipeline stage  $i$ , the  $i^{\text{th}}$  control unit generates the appropriate microinstruction and passes it to stage  $i$ . Designing a control unit for a pipelined data path requires sophisticated techniques which are beyond the scope of this book.

#### 7.7.4 Dealing with Control Flow

The pipelined data path discussed above will not work correctly when the executed program deviates from straightline sequencing. Such deviations happen whenever a branch instruction is taken (i.e., its condition is satisfied and control is transferred to its target address), or a jump instruction is encountered. System call instructions also cause a change in control flow, in addition to switching the system to kernel mode. The pipelined data path that we have been studying does have the ability to perform the control flow deviation dictated by branches, jumps, and syscalls. The **Target** register is updated with the appropriate target address, and this value can be selectively loaded into the PC register, so that instructions can be fetched from the target address thereafter. The problem, however, is that prior to fetching instructions from the target address, the data path would have already fetched the three instructions that immediately follow the branch/jump/syscall instruction. These instructions, if left unhampered, will wreak havoc in the data path by causing unintended changes to the ISA-visible state — the register file, PC, and the memory.

Consider the following MIPS program snippet. It starts with a conditional branch instruction at address `0x400500`. If the branch condition is not satisfied, control flow proceeds in a straightline fashion to fall-through instructions 1, 2, 3, .... By contrast, if the condition is satisfied, control flow is transferred to the `sub` instruction, skipping the 3 `add` instructions.

```
0x400500:  beq  $1, $2, 3      # skip next 3 instructions
                                # if values in $1 and $2 are equal
0x400504:  add  $3, $1, $2     # fall-through instruction 1
0x400508:  add  $5, $1, $4     # fall-through instruction 2
0x40050c:  add  $6, $2, $4     # fall-through instruction 3
0x400510:  sub  $7, $3, $4     # target address instruction
```

Now, let us track the execution of this program snippet in our pipelined data path. When

this snippet is about to be executed, PC contains address 0x400500. Once this address is in place in PC, fetching of the branch instruction automatically happens in the F stage. In addition, the PC is also updated to 0x400504 in the F stage. In the next clock cycle, the fetched branch instruction moves to the R stage of the pipeline, where reading of its source registers \$1 and \$2 occurs. At the same time, the F stage continues its action by fetching fall-through instruction 1 from address 0x400504. In the third clock cycle, evaluation of the branch condition takes place in the A stage. Calculation of the branch target address (0x400510) also happens in the A stage. The outcomes of these actions are recorded in the **flags** register and **Target** register, which sit at the boundary between the A and M stages. If the branch condition is satisfied, updation of the PC register with the target address happens in the 4th clock cycle, when the branch instruction moves into the M stage of the pipeline. By this time, stages A, R, and F of the pipeline would be populated by fall-through instructions 1, 2, and 3, respectively. The pipeline latches present at the boundaries of these stages would accordingly be updated by information pertaining to these instructions. All of this is perfectly fine if the branch condition is not satisfied and program control should continue to these instructions as per straightline sequencing. If the branch condition is satisfied, on the other hand, then allowing these three instructions to proceed would result in incorrect execution of the program. Specifically, the value of register \$3 used by the target address instruction would be the one produced by fall-through instruction 1.

There are several ways to deal with the above **control hazard** problem. The most straightforward among these is to detect the presence of a control hazard, and then *squash* the unwanted instructions from the pipeline, if there is a change in control flow. Squashing an instruction amounts to converting the instruction into a **nop** instruction within the pipeline. Notice that the squashed instructions would not have made any changes to our data path's ISA-visible state such as the register file and the memory locations. Figure \*\*\*\*\* illustrates this dynamic squashing of instructions within a pipeline.

The squashing method discussed above results in a loss of 3 clock cycles, as three instructions were converted to **nop** instructions. We call this 3-cycle loss as the **branch penalty**. In most of the real-life programs, a control flow change is not an infrequent event — roughly one in 6 of the executed instructions can be a control-changing instruction. Loosing 3 clock cycles every 6 cycles is not very appealing. Microarchitects reduce this penalty by incorporating one or more of the following techniques:

- Branch latency reduction
- Branch outcome prediction
- Branch effect delaying

#### 7.7.4.1 Branch Latency Reduction

In our pipelined data path, the outcome of a conditional branch instruction is determined in the A stage; calculation of the target address of control-changing instructions is also done

in the A stage. We can in fact determine the branch outcome in the R stage itself, as the branch conditions in the MIPS-I ISA are quite simple (testing the equality of two register values, or determining if the first value is greater than or less than the second value). An early computation such as this would, of course, require an extra comparator to be included in the R stage.

In a similar manner, calculation of the target address can also be performed in the R stage itself, as the ingredients for this calculation, such as the PC value and the `Offset` value, are available in the R stage itself. An important point is in order here: to complete the target address calculation in a timely manner in the R stage, the calculation may have to be started before knowing for sure if target address calculation is indeed required, i.e., before instruction decoding has been completed. Moreover, in the MIPS-I ISA, the target address calculation is different for the branch instructions and the jump instructions. This means that both types of calculation have to be done in parallel, such that the correct one among them can be selected after instruction decoding is completed. Notice also that such *speculative* actions increase the power consumption within the data path.

Performing both branch condition evaluation and target address calculation in the R stage reduces the branch penalty to just one cycle! Being able to determine the branch outcome as well as the target address in the R stage may seem to be quite an achievement; not so for modern pipelined processor data paths, which tend to have long, multi-stage, fetch engines. In these machines, several pipeline stages may precede the R stage; even if branch resolution is done in the R stage, several clock cycles are lost for every branch instruction. Can branch resolution be done earlier in such pipelines? Yes and no. While complete resolution of a branch instruction can be done only after the source registers have been read, a *speculative resolution* can be done before knowing the source register values. Next, let us look at this widely used scheme for reducing branch penalties in deeply pipelined machines.

#### 7.7.4.2 Branch Outcome Prediction

When a conditional branch instruction is executed, one of two possible outcomes can happen: either the branch is *taken* or it is *not taken*. Once this outcome is known, the data path can take appropriate actions for dealing with the instructions that were fetched following the branch. Can the data path somehow predict the outcome of a branch instruction well before its outcome becomes known? If it can do so, then it can take appropriate actions based on the predicted outcome. If the prediction turns out to be correct, then the data path does not suffer from an inordinate branch penalty. If the prediction turns out to be incorrect, then the data path suffers a **branch misprediction penalty**, which may even be higher than the normal branch penalty. If the branch predictions turn out to be correct the majority of the time, then it is worth the gamble!

Are branch instruction outcomes very predictable? Branch behavior is an artifact of the way programmers write high-level programs. Extensive studies by computer architects

with real-life programs have conclusively shown that branch outcomes are very predictable. There are several reasons for this predictability. First, branches that are used to implement loops tend to be taken many times in a row before they end up being not taken. These branches are highly biased, in that they are taken many more times than they are not taken. Second, many branches serve the function of testing for error conditions, which come into existence very rarely. Such branches are also highly biased. Even if a branch is not highly biased one way or the other, it may still be very predictable. Some of them depict alternating behavior: taken, not taken, taken, not taken, .... Some others have outcomes that have more complex, yet predictable, patterns. Yet others show behavior that is strongly correlated to the outcomes of branches that immediately precede it. A detailed treatment of branch behavior and branch prediction techniques is beyond the scope of this textbook. We shall, however, outline some of the branch behavior and branch prediction schemes that exploit such behavior. Before that, we need to see how the data path decides to make a prediction before knowing that the instruction being fetched is a branch instruction. The key to determining this is, again, temporal locality—branches that are executed now are most likely to have been encountered before.

**Branch Target Buffer (BTB):** The branch target buffer is a microarchitectural structure for storing that target address of previously encountered branch instructions. It is usually indexed by hashing the `pc` value of the instruction. Because multiple `pc` values can hash to the same entry in the BTB, a `pc` field is kept in each BTB entry to identify the address of the branch instruction that is currently mapped to that entry. Figure 7.16 shows the organization of a BTB. This BTB has two entries occupied with information about the branches at addresses `0x400800` and `0x400020`. Their target addresses are `0x400880` and `0x400008`. While fetching an instruction from the instruction cache, the BTB is consulted in parallel to determine if the instruction being fetched is a branch instruction (that has been seen before), and if so, to obtain its target address. A miss in the BTB indicates that the instruction is either not a branch instruction or a branch instruction that is unlikely to have been encountered in the recent past. In either case, if there is a miss in the BTB, pipelined execution proceeds as normal, as in straightline sequencing.

**Loop-terminating branches:** As mentioned earlier, branch instructions that terminate a loop tend to be *taken* many more times than they are *not taken*. A simple way to exploit this behavior is to speculatively start executing from the target address whenever a loop-terminating branch is encountered. How can the microarchitecture identify a loop-terminating branch? A simple heuristic to use is to consider all branches with negative offsets to be loop-terminating branches.

**Dynamic branch prediction:** Branches that are not loop-terminating may not be heavily biased in one direction; however, they may still have predictable behavior. The best way to capture such behavior is to use a microarchitectural structure that keeps track of the



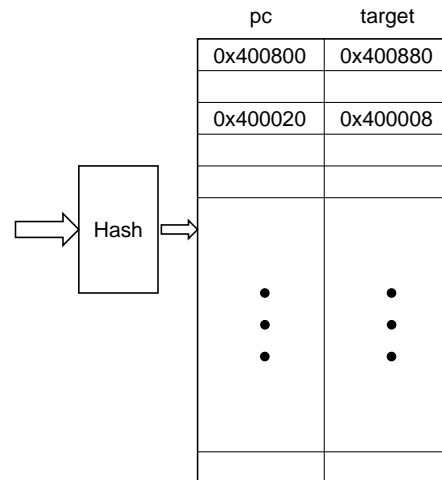


Figure 7.16: Organization of a Branch Target Buffer (BTB)

recent history of branches, and predicts the outcome every time a branch instruction is fetched. The scheme used for prediction could range from the simple *Last Outcome* based prediction to the most complex *Perceptron* based prediction. The Last Outcome predictor .....

### 7.7.4.3 Branch Effect Delaying

Branch prediction is a popular technique for reducing branch penalty in high-performance processors. However, it calls for higher power consumption as well as hardware complexity. The hardware structures used to store the past behavior of branches can be quite large, taking up space in the processor chip. They can also consume a significant amount of electrical power. Both of these factors can make branch prediction less attractive in embedded systems where size and power consumption may be constrained. Moreover, real-time system designers like to have very little uncertainties in the system; with branch prediction, it is difficult to accurately estimate the running time of a program. In such a scenario, we can employ yet another scheme for reducing branch penalty: redefine the semantics of a branch instruction to mean that a fixed number of instructions after the branch instruction should be executed irrespective of the outcome of the branch.

### 7.7.5 Dealing with Data Flow

The pipelined data path discussed above will work correctly as long as the instructions in the pipeline do not need to read registers that are updated by earlier instructions that are still present in the pipeline. There is indeed a problem when an instruction has to read a

register that will be updated by an earlier instruction. Consider the following instruction sequence:

```
add  $3, $1, $2
addi $4, $3, 16
```

The first instruction updates register \$3, which is the source register for the second instruction. When these two instructions are executed in a non-pipelined data path, fetching and execution of the second instruction begins only after the execution of the first instruction is completed and its result is written in \$3. By contrast, in a pipelined data path, the execution of the two instructions overlap in time. Let us trace this overlapped execution and see what happens. Figure 7.17 shows a timeline depicting the major actions happening in the pipeline.

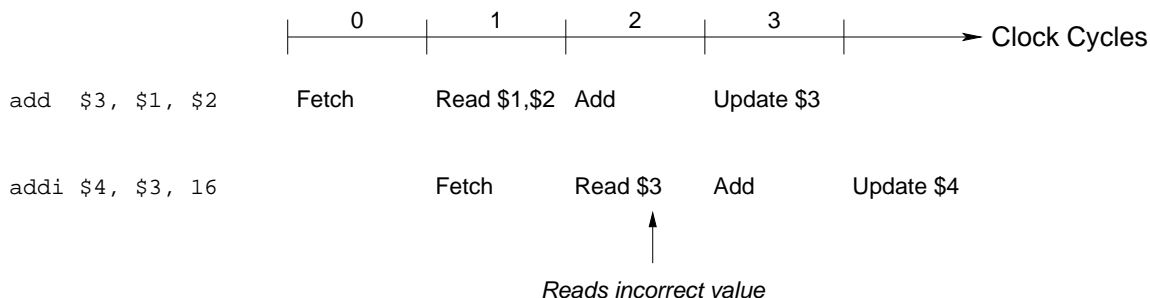


Figure 7.17: An Example Illustrating Data Hazards in a Pipelined Processor

The first instruction starts execution in clock cycle 0, and completes in clock cycle 3 when it writes its result to register \$3. The second instruction starts execution in clock cycle 1, and reads register \$3 in clock cycle 2. As this read operation happens before the write operation of clock cycle 3, it obtains the value that was in register \$3 prior to the execution of the first instruction. This clearly violates the semantic meaning of the program.

There are different ways to prevent such a **data hazard** from causing incorrect execution. The simplest among them is to detect the presence of data hazards, and temporarily switch to unpipelined operation if and when a data hazard is detected. In the above example, for instance, if the second instruction is delayed by 2 cycles as shown in Figure 7.18, then its register read operation happens only after the first instruction performs its register write operation. Of course, we need to introduce additional hardware to detect the presence of data hazards. Whenever a register-reading instruction reaches the R stage of the pipeline, this hardware should check if its source registers match the destination registers of the instructions present in the A and M stages of the pipeline.

The additional hardware for detecting data hazards turns out to be straightforward to design, once we list all data hazard scenarios. For the pipelined data path that is on our plate, we can group the data hazards into the following scenarios:

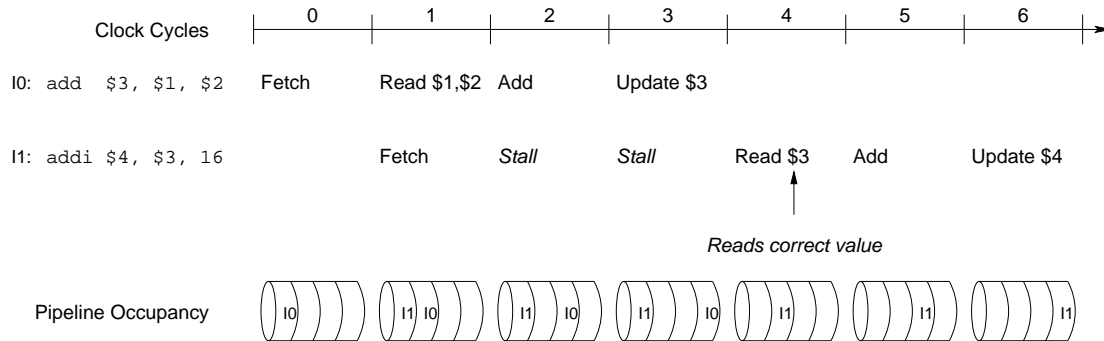


Figure 7.18: An Example Illustrating the use of Pipeline Stalls to deal with Data Hazards in a Pipelined Processor

1. lw    rt, offset(rs)    # any load instruction (register rt is destination)  
   add   rd, rs, rt        # any R-format instruction or branch-type instruction  
                          # (registers rs and rt are sources)

A data hazard exists if the `rt` value of `lw` is equal to the `rs` or `rt` values of `add`.

2. lw    rt, offset(rs)    # any load instruction (register rt is destination)  
   addi rt, rs, immed    # any I-format arithmetic/logic instruction  
                          # (register rs is source)

A data hazard exists if the `rt` value of `lw` is equal to the `rs` value of `addi`.

3. lw    rt, offset(rs)    # any load instruction (register rt is destination)  
   ...                    # any instruction that doesn't have register rt as source  
   add   rd, rs, rt
4. lw    rt, offset(rs)    # any load instruction (register rt is destination)  
   ...                    # any instruction that does not have register rt as source  
   addi rt, rs, immed    # any I-format arithmetic/logic instruction  
                          # (register rs is source)

### 7.7.6 Pipelines in Commercial Processors

Pipelining has been in use for several decades. One of the earliest machines to use pipelining was the CDC 6600. The MIPS R2000 and R3000 processors, which came out in 198\* and 199\*, respectively, used the 5-stage pipeline that we just looked at. Intel's Pentium processor also used a relatively short 5-stage pipeline. The Pentium Pro/II/III series of processors used a deeper pipeline having 12 stages, whereas the recent Pentium IV uses a very deep pipeline having 20 stages.

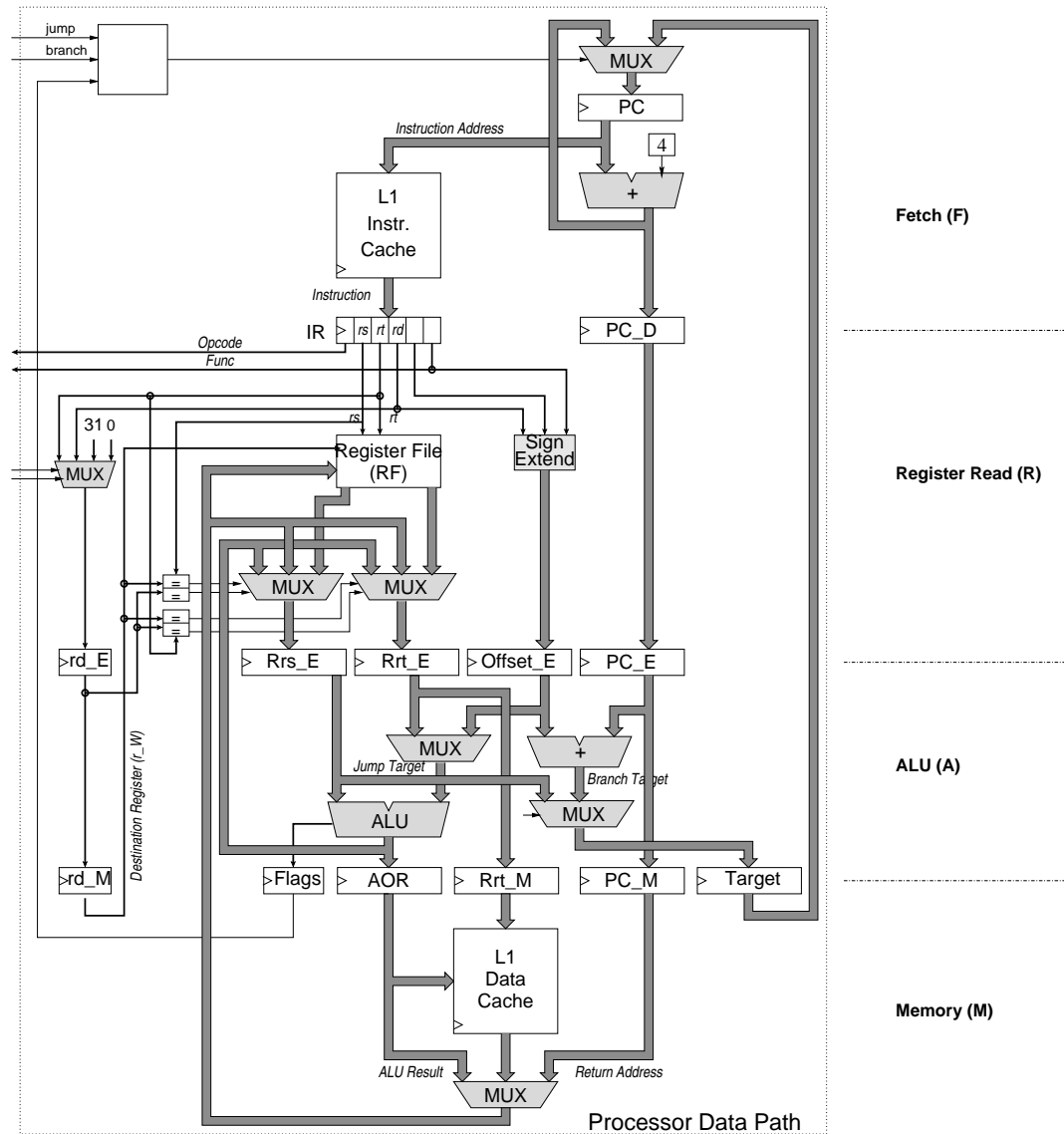


Figure 7.19: A Pipelined Processor Data Path with Data Forwarding

## 7.8 Wide Data Paths: Superscalar and VLIW Processing

We have seen how pipelining the datapath overlaps the execution of multiple instructions, thereby executing more instructions in unit time. The maximum instruction execution rate that can be achieved with pipelining, however, is only one instruction per clock cycle; notice that during every clock cycle, at most one new instruction is entering the pipeline (at

the fetch stage). Modern processors surpass this execution rate by *widening* the pipelined data path. This means that every clock cycle, they fetch multiple instructions in parallel, decode them in parallel, and execute them in parallel. This type of multiple-issue is done in addition to the pipelining technique described above. For multiple-issue to bear fruit, the processor data path should have multiple ALUs, or functional units.

Wide-issue datapaths have multiple instructions in each stage of the pipeline, and therefore overlap the execution of a large number of instructions. This exacerbates the control hazards and data hazards problem.

Compared to a single-wide pipeline, wider pipelines overlap the execution of a larger number of instructions, exacerbating the problem of control hazards and data hazards. For instance, when a control hazard is detected, more instructions are flushed from the pipeline, in general. The complexity of hazard detection and corrective action due to the hazards increases in two ways:

- More instructions are present in the pipelined datapath
- Hazards may be present among the multiple instructions present in the same stage of the pipeline

The first one among these is similar to what happens in a *deep pipeline*, and calls for rigorous application of the techniques discussed in Sections 7.7.4 and ?? for dealing with control hazards and data hazards — techniques such as branch prediction and data forwarding. Thus, we need branch predictors that are highly accurate, for instance. We also need many more paths for forwarding data.

Hazards introduced due to the presence of multiple instructions in the same pipeline stage ....

*[This section needs to be expanded.]*

## 7.9 Co-Processors

### 7.10 Processor Data Paths for Low Power

*“The ultimate “computer,” our own brain, uses only ten watts of power – one-tenth the energy consumed by a hundred-watt bulb.”*  
 — Paul Valery, Poet and Essayist

All of this chapter’s discussion so far focussed primarily on performance. While performance is one of the primary considerations in microarchitecture design, power consumption is becoming an important design consideration. Power consumption in a hardware circuit is the sum of the power consumed by its individual low-level components (primarily the transistors). The dynamic power consumed by a transistor is directly proportional to the

activity in the transistor in unit time, which can be loosely quantified as the product of number of times it switched in unit time, its capacitance, and the square of the supply voltage. The static power consumed by a transistor is directly proportional to its supply voltage and leakage current.

From the above discussion, it is clear that power consumption can be reduced by reducing the supply voltage and the number of transistors. Reducing the supply voltage, although feasible up to a point, does have the drawback of reducing the noise immunity of the circuit. Reducing the number of transistors directly translates to reducing the hardware circuitry in the design. In many situations, this is likely to hurt performance. Thus, there is often a trade-off between performance and power consumption. Intelligent designs attempt to use as little hardware as possible without reducing the performance.

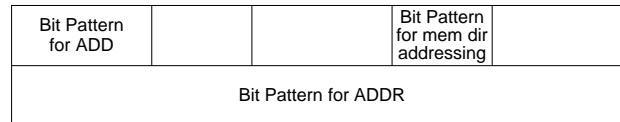
|                                    |
|------------------------------------|
| <b>Recent Processors Intel ***</b> |
|------------------------------------|

**Recent Processors**

## 7.11 Concluding Remarks

## 7.12 Exercises

1. Consider a non-MIPS ISA that has variable length instructions. One of the instructions in this ISA is ADD (ADDR), which means add the contents of memory location ADDR to ACC register. This instruction has the following encoding; notice that it occupies two memory locations.



Specify a MAL routine to carry out the interpretation of this instruction in the data path given in Figure 9.12.

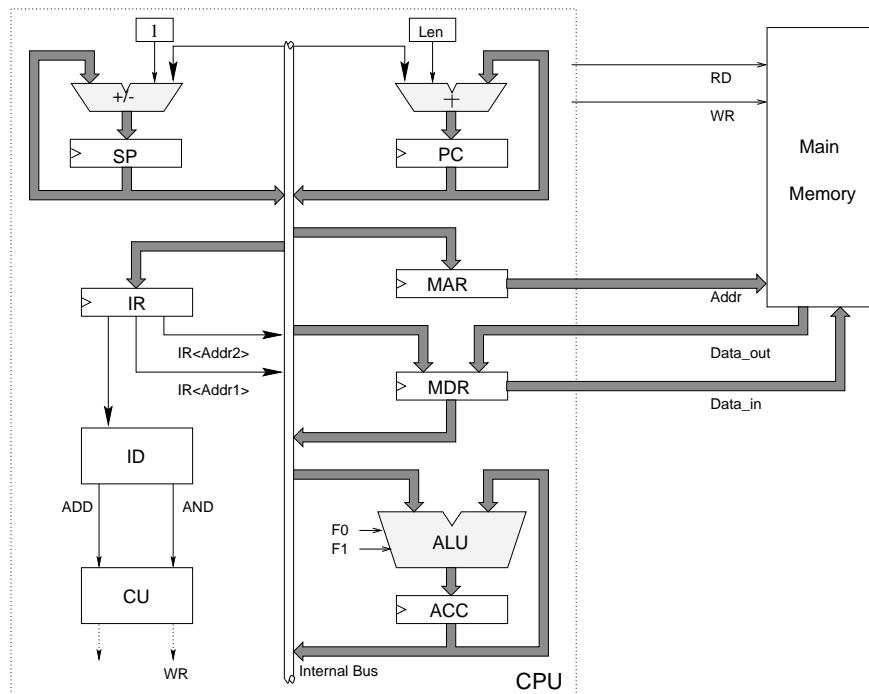


Figure 7.20: A Single-Bus based Processor Data Path for Interpreting an ACC based ISA

2. Consider the non-MIPS instruction ADD (ADDR1), (ADDR2), where ADDR1 and ADDR2 are memory locations whose contents needed to be added. The result is to be stored in memory location ADDR1. This instruction has the following encoding; notice that



addresses ADDR1 and ADDR2 are specified in the words subsequent to the word that stores the ADD opcode.

|                        |  |  |  |  |               |
|------------------------|--|--|--|--|---------------|
| Bit Pattern<br>for ADD | Bit Pattern<br>for mem dir<br>addressing |  | Bit Pattern<br>for mem dir<br>addressing |  | <i>Word 0</i> |
| Bit Pattern for ADDR1  |  |  |  |  | <i>Word 1</i> |
| Bit Pattern for ADDR2  |  |  |  |  | <i>Word 2</i> |

Specify the data transfer operations required to fetch and execute this instruction in the data path given in Figure 9.12. Notice that register ACC is a part of the ISA, and therefore needs to retain the value that it had before interpreting the current instruction. You are allowed to grow the stack in the direction of lower memory addresses.

3. Explain how adding multiple CPU internal buses can help improve performance.
4. Explain with the help of diagram(s) how pipelining the processor data path helps improve performance.
5. Consider a very small direct mapped cache with a total of 4 block frames and a block size of 256 bytes. Assume that the cache is initially empty. The CPU accesses the following memory locations, in that order: c881H, 7742H, 79c3H, c003H, 7842H, c803H, 7181H, 7381H, 7703H, 7745H. All addresses are byte addresses. To get partial credit, show clearly what happens on each access.
  - (a) For each memory reference, indicate the outcome of the reference, either “hit” or “miss”.
  - (b) What are the final contents of the cache? That is, for each cache set and each block frame within a set, indicate if the block frame is empty or occupied, and if occupied, indicate the tag of the memory block that is currently present.

## Chapter 8

# Microarchitecture — Kernel Mode

*A cheerful look brings joy to the heart, and good news gives health to the bones*

**Proverbs 15: 30**

**Look at Proverbs 23: 23**

The previous chapter discussed aspects of the microarchitecture that deal with the implementation of the user mode ISA. The kernel mode ISA includes several additional features, such as privileged registers, privileged memory address space, IO registers, and privileged instructions. This chapter focuses on microarchitecture aspects that deal specifically with the implementation of these additional features present in the kernel mode ISA. The functionality served by these features can be classified under 3 broad categories: processor management, memory management (virtual memory system), and IO system. This chapter is organized precisely along this classification.

The first part of the chapter discusses microarchitectural aspects that are essential for performing context switches at times of system calls, exceptions, interrupts, and return from exception. The second part provides an in-depth treatment of virtual memory. After presenting the major concepts of virtual memory, we take a close look at the virtual memory scheme in a MIPS-I system. Finally, the last part of the chapter address IO subsystem design and other system architecture issues.

### 8.1 Processor Management

Processor management involves allocating the processor to a particular process. As we saw in Chapter 4, the processor is generally time-multiplexed among the active processes in the

system<sup>1</sup>. We can break down the task of processor management into three aspects: recognizing exceptions and interrupts; enabling and disabling of interrupts; and switching back and forth between the user and kernel modes. We shall look at how the microarchitecture recognizes exceptions, and performs switching between the user and kernel modes.

### 8.1.1 Interpreting a System Call Instruction

In Chapter 7, we briefly discussed the execution of `syscall` instructions. However, we did not specifically say what happens after the system is placed in the kernel mode. We shall see here how the data path implements this part of the `syscall` instruction execution. Again, our discussion is based on the MIPS-I ISA, although the underlying principles are applicable to other ISAs as well. It is important to recall from Section 4.2.1 the major functionality specified by an instruction like `syscall`. We highlight these functions below:

- Switch to kernel mode: In the MIPS-I ISA, this is done by modifying the `state` register.
- Disable interrupts: In the MIPS-I ISA, this is also done by modifying the `state` register.
- Save return address: In the MIPS-I ISA, the return address of a `syscall` instruction is saved in the privileged register called `epc` (exception program counter).
- Record the cause for this exceptional event: In MIPS-I ISA, the same entry point (0x80000080) is specified for all but two of the exceptional events. Therefore, to identify the reason for transferring control to this memory location, the `syscall` code is entered into the `ExcCode` field of `cause` register.
- Update PC to point to the entry point associated with `syscall` (0x80000080).

In order to perform these functions, the data paths presented in Chapter 7 need to be enhanced to include the privileged registers `state`, `cause`, and `epc`. Figure 8.1 shows a data path obtained by including these privileged registers in the data path of Figure 7.7.

Table 8.1 presents a MAL routine for executing the MIPS-I `syscall` instruction in the data path of Figure 8.1. The first step of this routine updates the `state` register to place the system in the Kernel mode and to disable device interrupts. The next step saves the address of the `syscall` instruction in the `EPC` register. PC is then updated with the entry point address 0x80000080. Because this entry point is common for many exceptional events, the `ExcCode` field of the `cause` register is set to 8 to indicate that the exceptional event in this case is a `syscall` instruction.

---

<sup>1</sup>This is in contrast to the memory system and the IO system, which are generally space-multiplexed among the active processes.

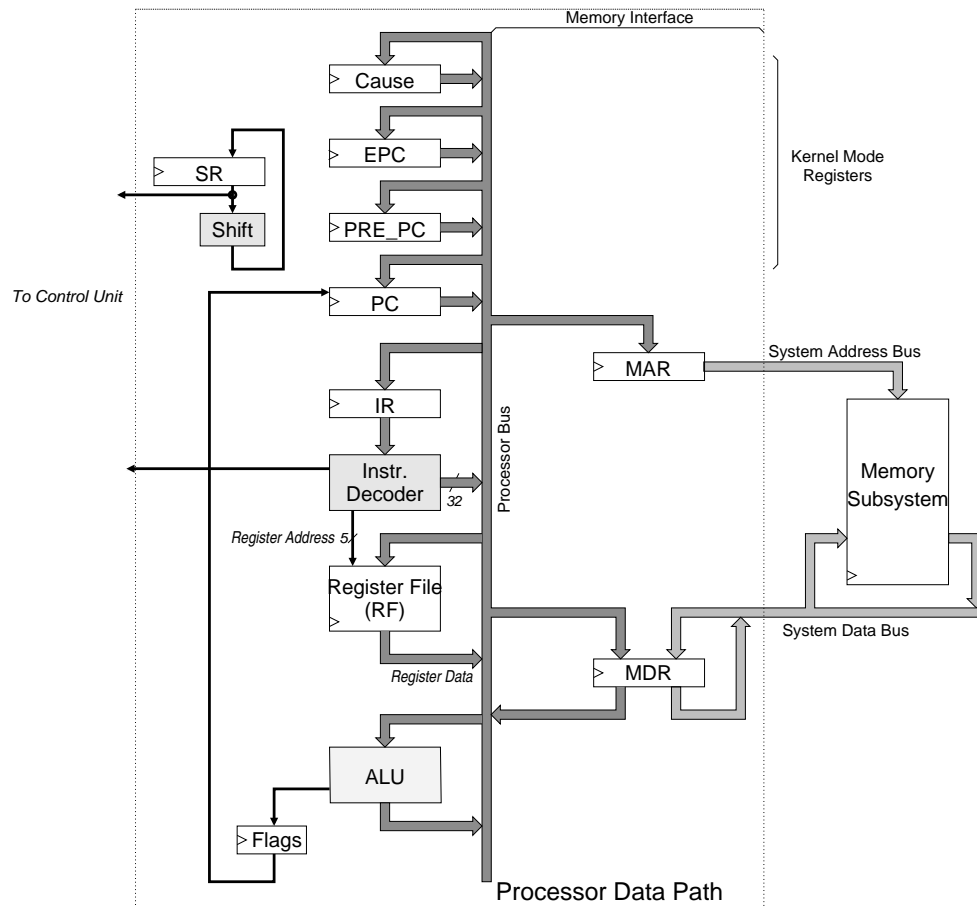


Figure 8.1: A Data Path for Implementing the MIPS-0 Kernel Mode ISA

### 8.1.2 Recognizing Exceptions and Hardware Interrupts

A realistic processor must do more than fetch and execute instructions. It must handle exceptions and also respond to device interrupts, irrespective of whether it is in the user mode or the kernel mode. The processor must explicitly check for exceptions whenever an exception can occur. Similarly, at the end of executing an ML instruction, it must check if there are any pending interrupts from IO devices. The actions taken in both these cases are somewhat similar to those taken when executing a syscall instruction. After all, a syscall instruction is a *software interrupt*.

Consider the **add** instruction of the MIPS-I ISA. This instruction is similar to the **addu** instruction whose execution we saw in detail in Chapter 7. It specifies the contents of registers **rs** and **rt** to be added and the result to be placed in register **rd**. The only difference

| Step No.             | Next Step No. for Control Unit | MAL Instruction to be Generated for Data Path | Comments  |
|----------------------|--------------------------------|---|---|
| <i>Execute phase</i> |                                |   |   |
| 4                    |                                | SR<3:0> << 2 → SR<5:0>                        | <i>Set system in kernel mode and disable interrupts</i> |
| 5                    |                                | Pre_PC → EPC                                  | <i>Save instruction address</i>                         |
| 6                    |                                | 0x80000080 → PC                               | <i>Set PC to 0x80000080</i>                             |
| 7                    | Goto step 0                    | 8 → cause<ExcCode>                            | <i>Update cause register</i>                            |

Table 8.1: A MAL Routine for carrying out the Execute Phase of the MIPS-0 Instruction `syscall`

between `add` and `addu` is that `add` specifies checking for arithmetic overflow (assuming the integers to be in the 2's complement number system). In the event of an overflow, an exception should be generated. Table 8.2 presents a MAL routine for interpreting the `add` instruction.

| Step No.                | Next Step No. for Control Unit | MAL Instruction to be Generated for Data Path | Comments   |
|-------------------------|--------------------------------|---|--|
| Execute phase           |                                |   |  |
| 4                       |                                | R[rs] → AIR                                   | Update Ov flag also                              |
| 5                       |                                | R[rt] + AIR → AOR, Ov                         |  |
| 6                       | if Ov goto step 0              | if (Ov && rd) AOR → R[rd]                     |  |
| Take Overflow Exception |                                |   |  |
| 7                       |                                | SR<3:0> << 2 → SR<5:0>                        | Set system in kernel mode and disable interrupts |
| 8                       |                                | Pre_PC → EPC                                  | Save instruction address                         |
| 9                       |                                | 0x80000080 → PC                               | Set PC to 0x80000080                             |
| 10                      | Goto step 0                    | 12 → cause<ExcCode>                           | Update cause register                            |

Table 8.2: A MAL Routine for the Execute Phase of the Interpretation of the MIPS-0 ISA Instruction Represented Symbolically as `add rd, rs, rt`. This MAL Routine is for executing the ML instruction in the Data Path of Figure 8.18

The first step of this MAL routine is same as before. In the ALU operation step, in addition to writing the addition result in AOR, the Ov (overflow) flag is updated to indicate the occurrence of overflow, if any. The next step explicitly checks for the value of Ov. If this flag is not set, then the addition result is written to register `rd` and control goes back to step 0. Otherwise, the microarchitecture proceeds to step 7 which begins the routine for taking the exception. This part is the same as that of the `syscall` instruction except that the code written in the `ExcCode` field of the `cause` register is 12.

### 8.1.3 Interpreting an RFE Instruction

The kernel mode includes many privileged instructions, especially for manipulating different privileged registers. One of the kernel mode instructions that warrants special treatment is the `rfe` (restore from exception) instruction. This instruction tells the machine to restore the system's mode and interrupt status to what it was prior to taking this exception. Ironically, it does not tell the machine to transfer control to the interrupted program; a standard `jr` instruction is used to achieve this transfer of control, as we saw in Chapter 4. Table 8.3 presents the MAL routine for the execute phase of the `rfe` instruction.

| Step No.             | Next Step No. for Control Unit | MAL Instruction to be Generated for Data Path | Comments  |
|----------------------|--------------------------------|---|---|
| <i>Execute phase</i> |                                |   |   |
| 4                    | Goto step 0                    | SR<5:0> >> 2 → SR<3:0>                        | <i>Restore system mode and interrupt status</i> |

Table 8.3: A MAL Routine for the Execute Phase of the Interpretation of the MIPS-0 kernel mode Instruction `rfe`

## 8.2 Memory Management: Implementing Virtual Memory

### 8.2.1 Virtual Memory: Implementing a Large Address Space

Next, let us turn our attention to the implementation of the memory address space defined in the ISA. The most straightforward approach to is to implement the entire memory address as system memory (i.e., *physical memory*). Modern ISAs, however, specify quite a large address space, spanning  $2^{32}$  (i.e., 4 G) or even  $2^{64}$  locations. Implementing such a large address space as physical memory is not a worthwhile proposition from the economic point of view, despite today's falling memory prices.

The above problem becomes more acute in a *multitasked* system. Recall from Chapter 5 that modern computer systems incorporate multitasking, permitting multiple processes to be simultaneously active in the system. The processor is time-multiplexed among the active processes by the operating system. At the time of a context switch, the current state of the processor (including the register values) is saved in the kernel address space, and the processor state is updated with the previously stored state of the process that is scheduled to run next. It is important to note, however, that a process' state is not limited to its processor state; the state includes the memory values too. Therefore, besides saving and restoring the register values, the memory values also need to be saved and restored. However, time-multiplexing the system memory is quite impractical, as it requires writing and reading huge amounts of data to a slow secondary storage device such as a hard disk. Each context switch will then take seconds or even minutes! Again, a brute-force solution

would be to provide a separate physical memory for each active process. But such a naive approach stretches the physical memory requirement even further, especially considering today's high degrees of multitasking (64 processes and more). Imagine a computer system with  $64 \times 4 \text{ GB} = 256 \text{ GB}$  of physical memory!

The above discussion highlights the difficulty of implementing the entire memory address space of one or more processes by physical memory. Most of today's computer systems overcome this difficulty by using a scheme called *virtual memory*. In this scheme, the ISA-defined memory address space is implemented at the microarchitectural level by a two-tier memory system, consisting of a small amount of physical memory at the upper tier and a large amount of *swap space* at the lower tier, managed by the operating system<sup>2</sup>. The memory addresses generated by the processor, called *virtual addresses*, are translated into physical addresses on the fly by a *Memory Management Unit (MMU)*. If a virtual address refers to a part of the program or data space that is mapped to the physical memory, then the contents of the appropriate location in the physical memory are accessed immediately. On the other hand, if the referenced location is mapped to the swap space, then an *exception* is generated to turn control over to the operating system, which gets the requested word after a much longer time. Finally, if the referenced location is not mapped to either the physical memory or the swap space, then special action needs to be taken by the operating system. The low-level application programmer, as we saw in Chapter 3, is not aware of the limitations imposed by the smaller amount of physical memory available.

While the physical memory can be accessed in a few processor cycles, accessing the swap space takes millions of processor cycles. Because of this huge discrepancy, unless the number of accesses to the swap space is limited to a very tiny fraction of the total references, the performance of the system will be very poor. It is imperative that the operating system does a good job here, and we will see how it does this in the next chapter.

**Protection and Sharing:** Another important issue in memory management in a multi-tasking environment is one of protection. When the physical memory is partitioned among multiple processes, the operating system has to protect itself and the others from accessing each other's memory.

Next, let us turn our attention to the implementing the virtual memory concept that we have been discussing. Recall that the primary purpose of a virtual memory system is to implement the large address space defined in modern ISAs in a cost-effective manner. Let us first highlight the main features of this memory system. In a virtual memory system, the ISA-defined memory address space is implemented at the microarchitectural level by a two-tier memory system. This two-tier consists of a small amount of physical memory at the upper tier and a large amount of *swap space* at the lower tier. The management of the system is done by a combination of hardware (the *memory management unit (MMU)*) and software (the *memory management system* of OS). The memory addresses generated by the

---

<sup>2</sup>The operating system generally stores the swap space in a hard disk.

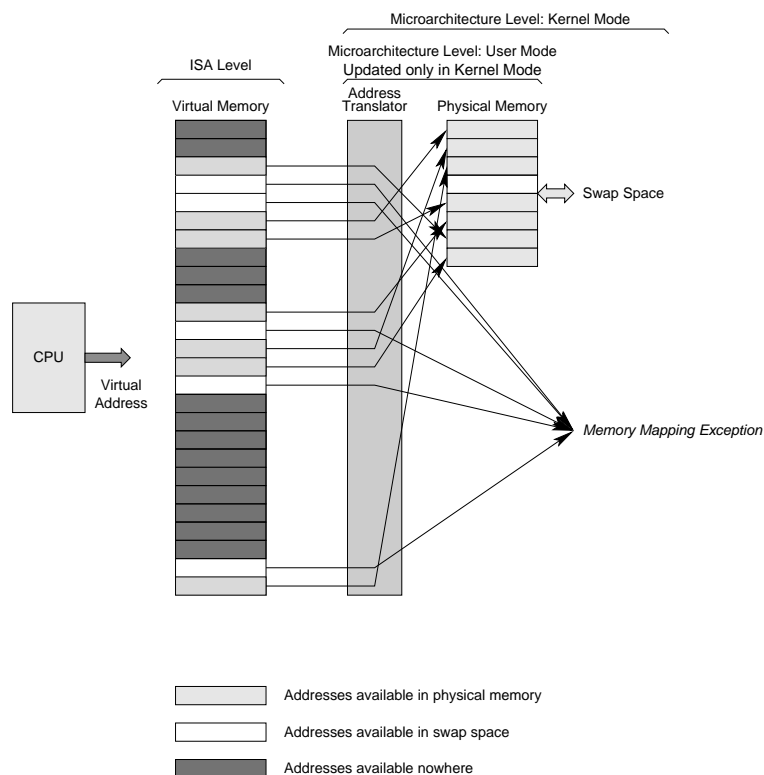


Figure 8.2: Illustrating the Concept of Virtual Memory

processor, called *virtual addresses*, are translated into physical addresses on the fly by the MMU. If a virtual address refers to a part of the program or data space that is mapped to the physical memory, then the contents of the appropriate location in the physical memory are accessed immediately. On the other hand, if the referenced location is mapped to the swap space, then the access time will be much longer. Finally, if the referenced location is not mapped to either the primary memory or the swap space, then special action needs to be taken by the system. Thus, the low-level application programmer sees a single memory address space (*virtual address space*), and is not aware of the limitations imposed by the smaller amount of physical memory available.

Today's general-purpose computer systems have physical memory ranging from about 32 MB to about 512 MB, which is less than the memory address space defined by the ISA for a single process. The operating system partitions the available physical memory among itself and the other active processes in the system, although not necessarily in equal portions. Some portions are common to multiple processes so that they can share common data.

fbox “What belongs to everybody belongs to nobody.”  
— Spanish Proverb



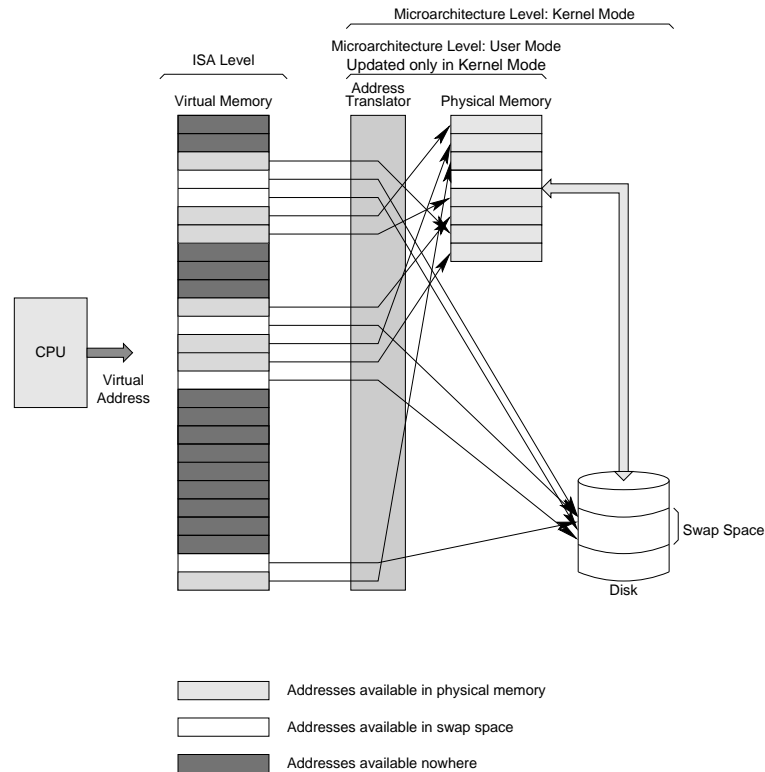


Figure 8.3: Illustrating the Implementation of Virtual Memory

Whereas the access time of the physical memory equals a few processor cycles, the access time of the swap space equals millions of processor cycles. Because of this huge discrepancy, unless the number of accesses to the swap space is a very tiny fraction of the total references, the performance of the system will be really bad. In order to obtain good performance, the mapping (from virtual addresses to physical addresses) is dynamically adapted in such a manner that the locations that were frequently accessed in the recent past (and are therefore expected to be accessed again in the near future) are mapped to the physical memory. The infrequently accessed locations are mapped to the swap space.

With the above arrangement, whenever a memory access gets translated to the swap space, the contents of that location are brought into a suitable location in the physical memory prior to using them. At that time, if there is no empty space in the physical memory, then the displaced contents are automatically *swapped out* to the swap space. Thus, programs and their data are automatically moved between the physical memory and

secondary storage in such a way as to capture temporal locality among memory accesses.

### 8.2.2 Paging and Address Translation

To facilitate the implementation of virtual memory and to effectively capture the spatial locality present among the memory references (which is very important for obtaining good performance), the translation from virtual addresses to physical addresses is done at a granularity much larger than that of individual addresses. Two distinct schemes and their combinations are widely used—*paging* and *segmentation*.

The paging scheme is similar to the cache memory scheme that we saw in the previous chapter. In this scheme, the translation is done at the granularity of equal fixed-size chunks called **pages**. Each page consists of a block of words that occupy contiguous addresses in the address space. It constitutes the basic unit of information that is transferred between the physical memory and the swap space in the disk whenever the translation mechanism determines that a transfer is required. (Pages are similar to *blocks* used in cache memory, but are much bigger.) The virtual address space is partitioned into *virtual pages*, and the physical memory is partitioned into *page frames*. Page sizes commonly range from 2 KB to 16 KB in size. Page size is an important parameter in the performance of a virtual memory system. If the pages are too small, then not much of spatial locality will be exploited, resulting in too many *page faults* (i.e., accesses to the secondary storage). Given that the access time of a magnetic disk is much longer (10 to 20 milliseconds) than the access time of physical memory, it is very important to keep the number of page faults at an extremely small value. On the other hand, if pages are too large, it is possible that a substantial portion of a page may not be used, and temporal locality cannot be exploited to the desired extent.

Information about the mapping from virtual page numbers to page frame numbers is kept in tabular form in a structure called **page table**. Figure 8.4 shows the basic organization of a page table, along with how it is used in address translation. The page table has one entry for each virtual page. Each page table entry (PTE) has at least three fields—**Protection** field, **Valid** bit, and **Page Frame Number** (PFN). The **Protection** (P) field stores the access rights for the virtual page. The **Valid** (V) bit indicates if the virtual page is currently mapped to physical memory, in which case page frame number (PFN) is available the PFN field.

The actions involved in translating a virtual address to a physical address are best expressed using a flow chart. Figure 8.5 presents such a flowchart. This flowchart starts at the top left corner. We can conceptually divide this flowchart into two parts. The actions on the left side of the figure are the frequently encountered ones and are best done by the hardware (MMU). Those on the right are encountered infrequently and are done by the software (MMS). In this flowchart, the page table is assumed to be a hardware structure that can be read in the Kernel mode as well as in the User mode; of course, writes to the page table can only be done in the Kernel mode. Later, we will see more modern organizations in

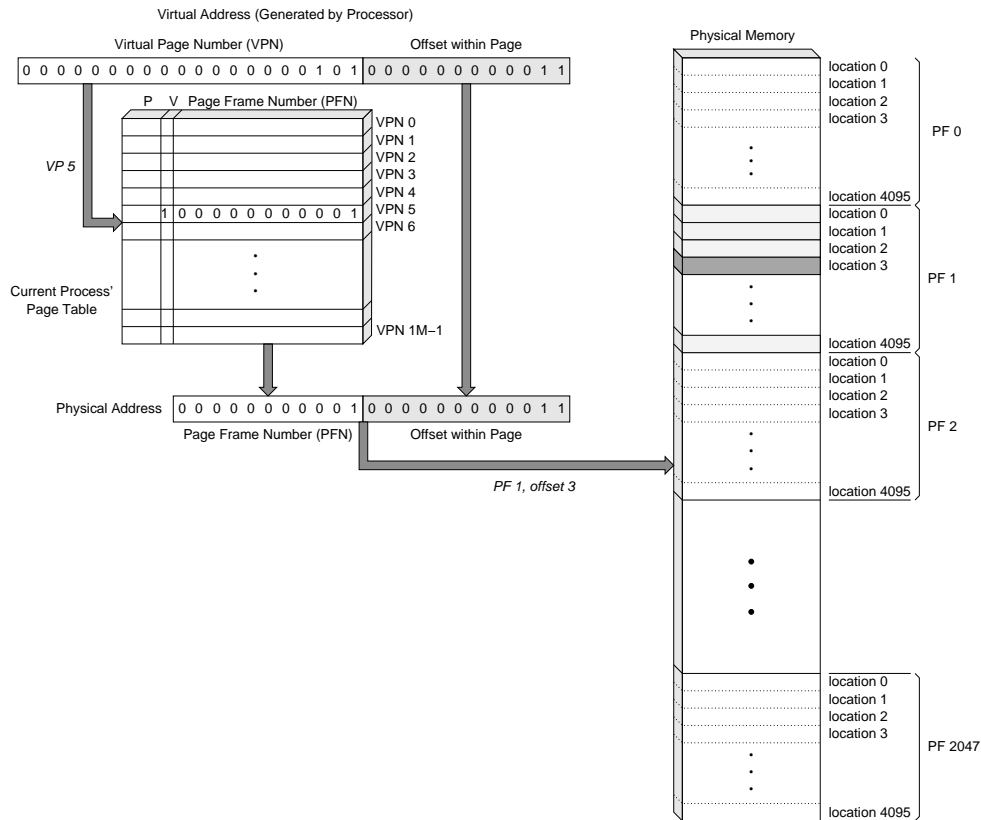


Figure 8.4: Virtual Address to Physical Address Translation Using a Page Table

which the page table is stored within the memory address space itself! While going through the sequence of actions of the flowchart, it is useful to consult Figure 8.4 for a structural understanding of what goes on.

Given a virtual address, the MMU splits it into two parts—a **Virtual Page Number (VPN)** followed by an **Offset within Page** that specifies the location within the page. For instance, in Figure 8.4, the VPN corresponding to virtual address `0x5003` is 5. The MMU uses the VPN value as an index into the page table, and reads the corresponding page table entry (PTE). If the **Valid (V)** bit of the PTE is set, then the virtual page is currently present in the physical memory, and translation continues. This is the frequent case. If the protection bits of the PTE indicate that the access is not permitted, then the MMU generates a *memory access violation exception* to transfer control to the OS, which generates an error message and terminates the offending process. If the access is a permitted one, the MMU calculates the physical address by concatenating PFN and **Offset within Page**. The physical memory access is then performed using the physical address obtained.

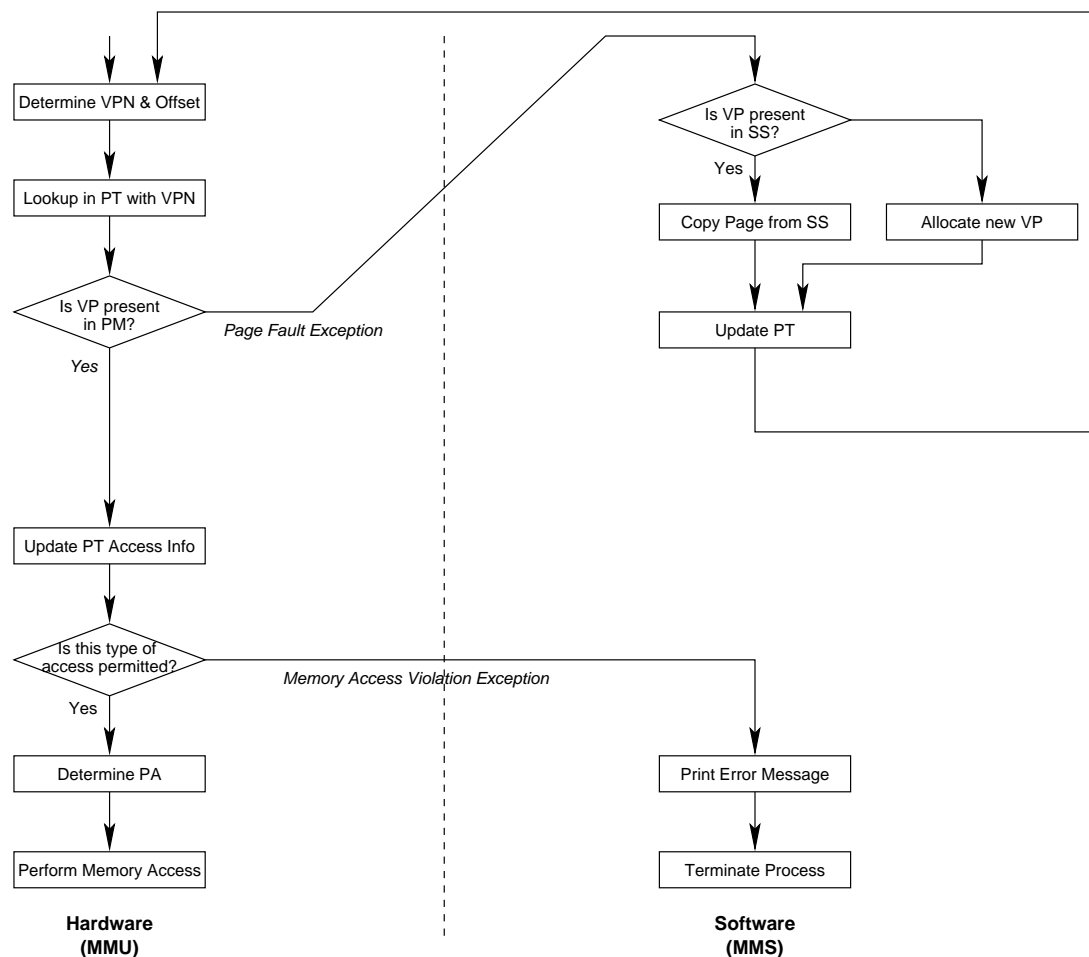


Figure 8.5: A Flowchart of the Steps Involved in Address Translation with a Page Table

In Figure 8.4, the PFN is 1, and the physical address is 0x1003. This location is shaded in the figure.

If the **Valid** bit of the relevant PTE is not set, the requested VPN is not currently mapped to physical memory, and the MMU generates a **page fault** exception. The generation of the exception automatically switches control to the OS, and eventually to the MMS part of the OS. The MMS performs one of two things: (i) copy the required page from the swap space to a page frame (if this involves replacing a dirty page, the old page is copied to the swap space), or (ii) “create” a new page if the virtual page does not yet “exist” (for example, when a process allocates a new stack frame in an uncharted territory). It then updates the corresponding PTE to reflect the new translation, and transfers control back to the interrupted process, which results in re-execution of the instruction that caused the

page fault exception. Thus, page faults and page table updates are typically handled by the OS software.

*Example:* In order to clarify these ideas further, let us go through an example based on Figure 8.4. The example also gives an idea about the exorbitant space requirement of page tables. Assume that the ISA (instruction set architecture) specifies a 32-bit address space in which each address corresponds to a byte. The virtual memory system uses a page size of 4 KB. The computer has 8 MB of physical memory.

1. How many bits of the 32-bit address are used to determine the **Offset within Page**?  
Because the page size is 4 KB, the number of bits needed to determine the **Offset within Page** is  $\log_2 4K = 12$  bits.
2. How many virtual pages exist?  
The virtual memory address space consists of  $2^{32}$  bytes. The number of virtual pages is given by dividing the virtual address space size by the page size, and is therefore equal to  $2^{32} \text{ bytes} \div 4 \text{ KB} = 2^{20} = 1 \text{ M}$ .
3. How many page frames exist?  
The physical memory consists of 8 MB. The number of page frames is obtained by dividing the physical memory size by the page size, and is therefore equal to  $8 \text{ MB} \div 4 \text{ KB} = 2K$ .
4. If each PTE occupies 4 bytes, how many bytes will the page table occupy?  
The page table has as many entries as the number of virtual pages, namely 1 M. If each PTE occupies 4 bytes, the page table occupies  $1 \text{ M} \times 4 \text{ bytes} = 4 \text{ MB}$ .
5. What is the physical address corresponding to virtual address 0x00005003?  
The least significant 12 bits of the virtual address (0x003) give the Offset within Page, and the remaining bits (0x00005) give the VPN. The MMU then identifies the PTE corresponding to VPN 5. The V bit of this PTE is set, indicating that the PFN field is valid. Therefore, the PFN value is 0x001. On concatenating the PFN with the Offset within Page, we get the physical address of 0x001003.

### 8.2.3 Page Table Organization

An issue that we have skirted so far is the physical location of the page table. A few decades ago, when address spaces were much smaller, the page table was small enough to be stored in the MMU hardware structure, permitting fast address translations. As memory address spaces grew, page tables grew along with them, eventually forcing system designers to migrate them to the memory address space itself. Thus, the current practice is to store the page tables of all active processes in the memory address space. The starting address of each page table can be either calculated from the ASID (address space ID) or is recorded

in a separate table. But, this approach of storing the page tables in the memory address space raises three important problems.

First, if the page table is allocated in the virtual address space, accessing the page table requires an address translation, which in turn requires another translation, and so on, resulting in an endless cycle of address translations. This problem is dealt with by specifying a portion of the memory address space to be either *untranslated* (also called *unmapped*) or *direct mapped*. In the former case, if a virtual address is within an untranslated region, then the physical address is the same as the virtual address. In the latter case, if a virtual address is within a direct-mapped region, then the physical address is obtained by applying some trivial hashing function to it. By placing the page table in an untranslated/direct-mapped region, the page table's physical location can be determined without performing a page table-based address translation.

The second problem is the size of the page table. For a 32-bit byte-addressable user address space partitioned into 4 KB pages, the page table has a million ( $2^{32} \text{ B} \div 4 \text{ KB} = 2^{20}$ ) entries, which requires at least 4 MB of storage. Considering the amount of physical memory required to store the page tables of several active processes, storing the entire page tables in the untranslated/direct-mapped address space seems to be a difficult proposition. Ironically, large portions of the page tables may not be required by an application, and only a small subset of the virtual pages may be currently mapped to physical pages. One solution to deal with the size problem mentioned above is to store the user page tables in the translated portion of the kernel address space, and to store only the kernel page table (which is then required to access the user page tables) in the untranslated/direct-mapped address space. Figure 8.6 shows such an organization.

Figure 8.6: Root Page Table

If the root page table is found to be too large, it can be organized in an hierarchical manner, with only the topmost level of the hierarchy stored in the untranslated/direct-

mapped address space. Figure 8.7 shows the organization of an **hierarchical page table**. The page tables in all of the levels except the bottom-most one serve as directories for the level immediately below it. The space occupied by each small page table (at each level) is typically limited to one page. Many recent computer systems implement hierarchical page tables. For instance, computer systems based on the DEC Alpha 21\*64 processor, which supports a 64-bit address space, typically use a 4-tiered hierarchical page table.

Figure 8.7: An Hierarchical Page Table

#### 8.2.4 Translation Lookaside Buffer (TLB)

The third problem is that every memory reference specified in the program requires two or more memory accesses, all of which except the last one are page table accesses done for translating the address; the last access is to the translated memory location. The solution commonly adopted to reduce the number of accesses for translation purposes is to cache the active portion of the page table in a small, special hardware structure called *Translation Lookaside Buffer (TLB)*, which is kept within the MMU, as shown in Figure 8.8. Because each TLB entry covers one page of memory address space, it is possible to get a very high hit rate from a reasonably small TLB.

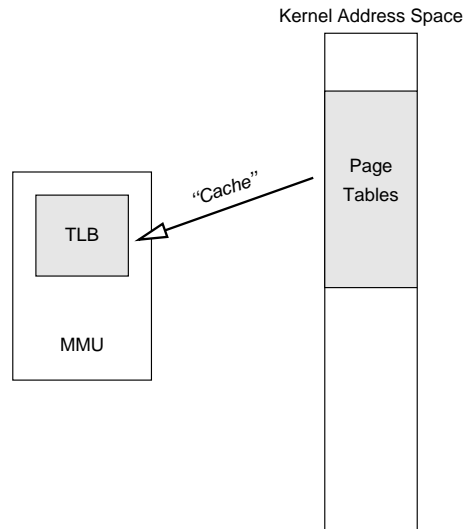


Figure 8.8: Relationship between TLB and Page Tables

With the introduction of a TLB, the address translation procedure involves a small change, as depicted in Figure 8.9. The main change is that the TLB has replaced the page table as the primary structure used for obtaining address translation. The page table is still used, but only if the TLB fails to provide a translation.

The TLB can be managed either by the hardware or by the software. We shall first discuss the operation when using a hardware-managed TLB. Figure 8.10 presents a flowchart of the actions in such a system. This flowchart is a modification of the one shown earlier without a TLB. The main difference is that after determining the VPN, the MMU looks in the TLB for the referenced VPN (instead of directly accessing the page table). If the page table entry for this VPN is found in the TLB, the PFN and protection information are obtained immediately. If the protection bits indicate that the access is not permitted, then a *memory access violation exception* is generated as before. If the access is a permitted one, the physical address is determined by concatenating PFN and Offset within Page, and the main memory access is performed. TLB hit scenario

If there is no entry in the TLB for the specified VPN, a *TLB miss* is said to have occurred. The MMU handles the TLB miss by reading the required PTE from the page table stored in the kernel address space. If the PTE is obtained, then the MMU updates the TLB with the new entry. If the page table does not contain a valid PTE for the virtual page, then a *page fault exception* is generated as before. The MMS part of the OS handles the page fault exception and then transfers control back to the program that caused the exception. TLB miss scenario  
Page fault scenario





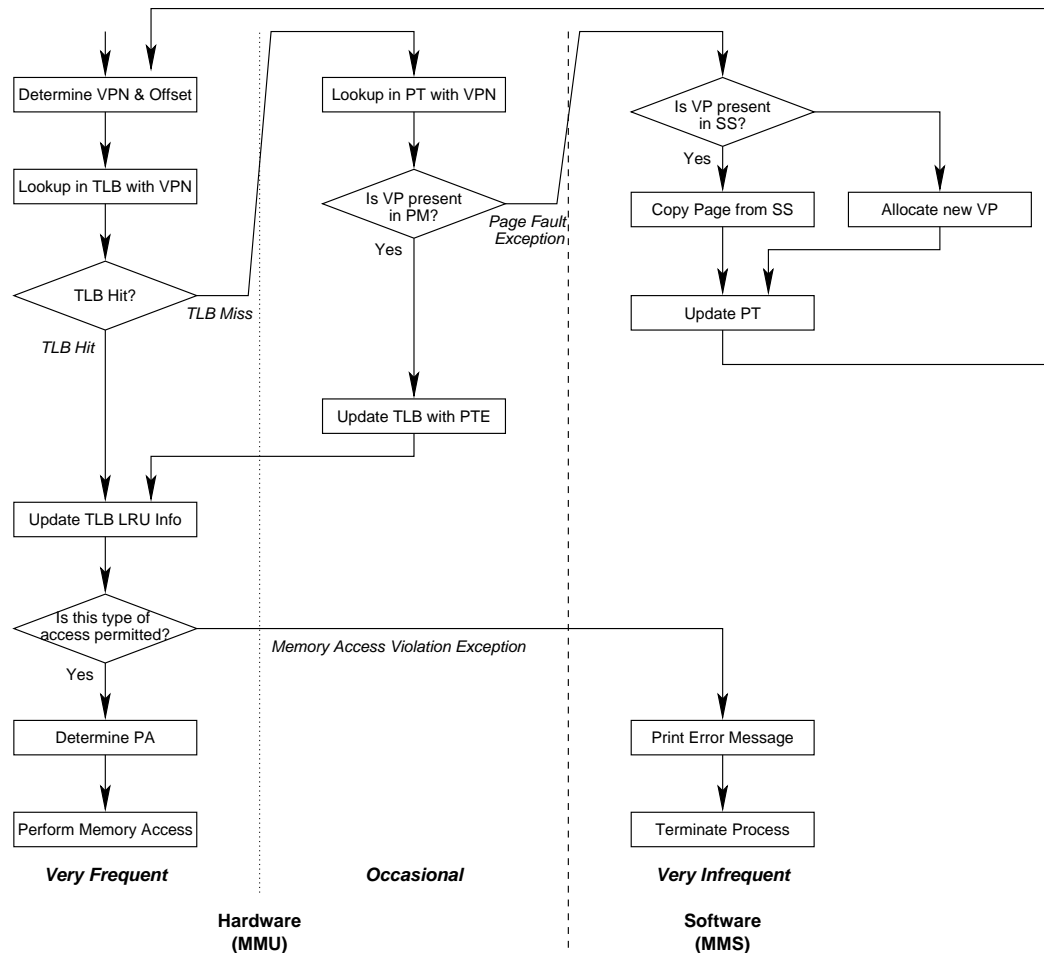


Figure 8.10: A Flowchart of the Address Translation Steps with a Hardware-Managed TLB

look at software-managed TLBs.

### 8.2.5 Software-Managed TLB and the Role of the Operating System in Virtual Memory

One aspect that must be clear by now is that unlike the implementation of cache memories, the implementation of the virtual memory concept in modern computers is rarely done entirely in hardware. Instead, a combination of hardware and software schemes is used, with the software part being implemented as a collection of OS routines called the *memory management system (MMS)*. The page tables are usually placed in the kernel address space.

In a virtual memory system implementing a hardware-managed TLB, TLB misses are

handled in hardware by the MMU. Examples of processors that include a hardware-managed TLB are the IA-32 family and the PowerPC. The organization of such a system is illustrated in Figure 8.11(i). In such a system, the TLB is not defined in the (kernel mode) ISA, and so the OS does not know about this hardware unit. The TLB manager hardware, which handles TLB misses, is also unknown to the OS. The page table (PT) is defined in the kernel mode ISA so that when handling TLB misses, the TLB manager can perform a PT lookup — *hardware page table walking*.

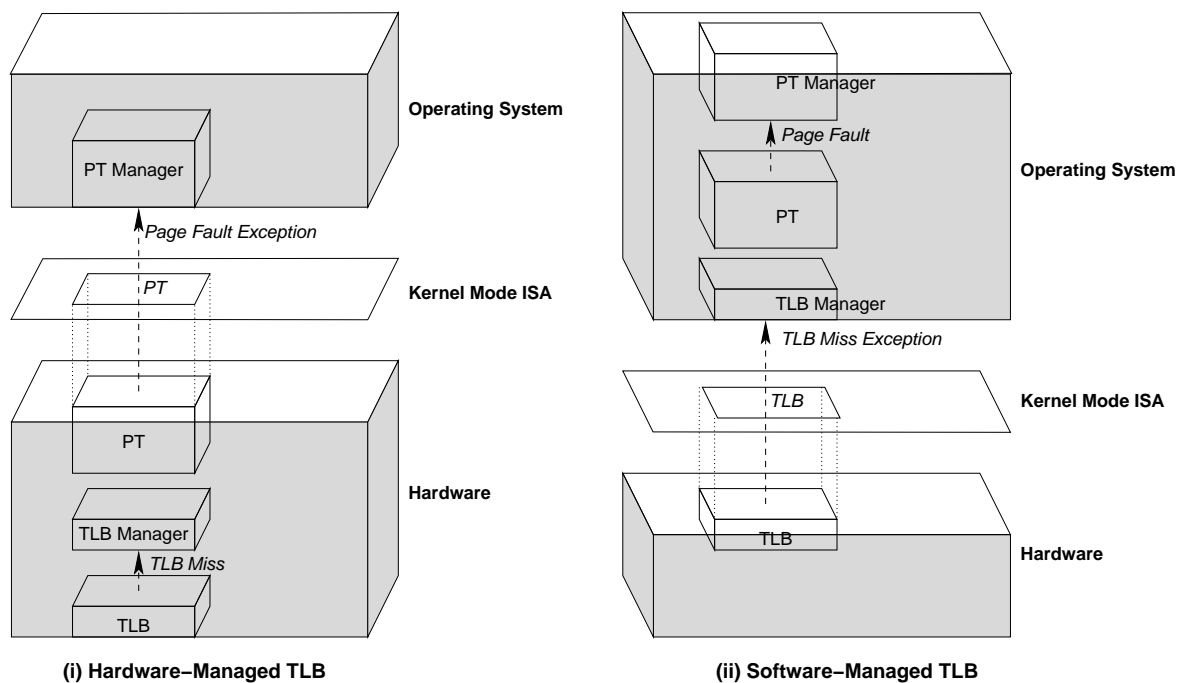


Figure 8.11: Conceptual Organization of a Computer System with (i) a Hardware-Managed TLB; (ii) a Software-Managed TLB

### 8.2.5.1 Software-Managed TLB

In a system implementing a software-managed TLB, TLB misses are handled in software by the MMS part of the OS. Examples of processors that require a software-managed TLB are MIPS, Sparc, Alpha, and PA-RISC. The organization of such a system is illustrated in Figure 8.11(ii). In such systems, the TLB specification is included in the kernel mode ISA. A TLB miss in such a system would cause a **TLB Miss Exception**, which automatically transfers control to the operating system. The operating system then locates and accesses the appropriate PTE, updates the TLB with information from this PTE, and transfers

control back to the program that caused the TLB miss.

Figure 8.12 shows a flowchart that reflects the working of a virtual memory system using a software-managed TLB. On comparing this with the flowchart of Figure 8.10, we can see that there are no differences in the steps themselves, but only in who performs some of the steps. The steps in the middle column are performed here in software by the MMS part of the OS, rather than by the MMU hardware.

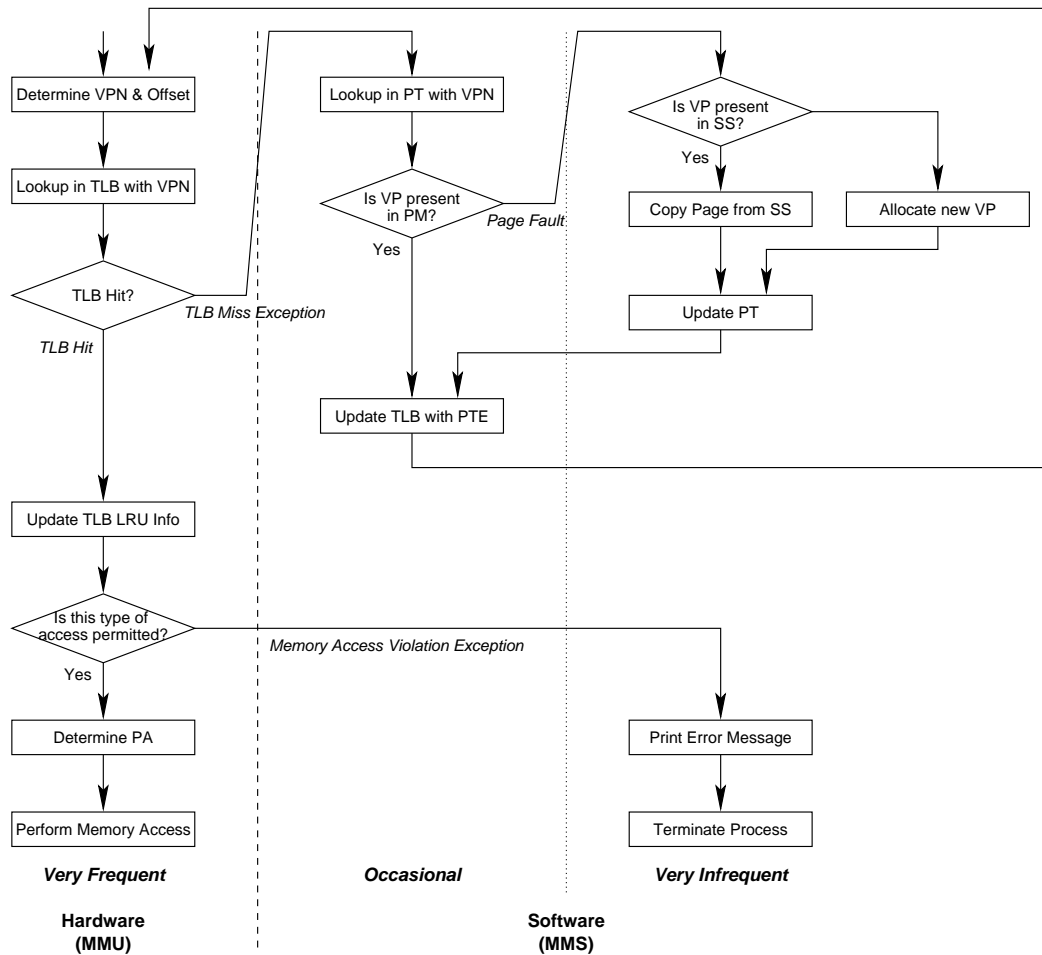


Figure 8.12: A Flowchart of the Address Translation Steps with a Software-Managed TLB

The advantages of using a software-managed TLB are two-fold:

- The OS can implement virtually any TLB replacement policy.
- The algorithm, structure, and format of the page table are not fixed, and the OS can implement any page table organization.

The disadvantage, of course, is that it takes more time to service a TLB miss. If the OS can implement TLB replacement policies that reduce the number of TLB misses significantly, then this strategy has an overall advantage.

### 8.2.6 Sharing in a Paging System

### 8.2.7 A Real-Life Example: a MIPS-I Virtual Memory System

Computer systems have implemented a wide variety of virtual memory organizations. They differ primarily in terms of how the page table is organized, and who manages the TLB—hardware or software. Instead of describing all possible organizations, we have attempted to cover the fundamental principles behind them. For completeness, we will also look at a concrete example—the memory management system in a MIPS-I based system—to see how all of the different aspects work together<sup>3</sup>. The MIPS-I ISA supports one of the simplest memory management organizations among recent microprocessors. It was one of the earliest commercial ISAs to support a software-managed TLB. This feature implies that the page table is not defined in the kernel mode ISA, and that the OS is free to choose a PT organization. Some support is provided, nevertheless, for implementing a simple linear PT organization in the kernel's virtual address space. Similarly, although the OS is free to implement any replacement policy for the TLB, the ISA provides some support for a random replacement policy. In the ensuing discussion, we first describe the MIPS-I kernel address space, and then the support provided in the kernel ISA to implement virtual memory. Finally, we show how these ISA features can be used in a possible virtual memory system organization for MIPS-I based systems.

#### 8.2.7.1 Kernel Mode ISA Support for Virtual Memory

We shall start with the kernel address space. The MIPS-I ISA specifies a 32-bit kernel address space. Part of the address space is *unmapped* so that accesses to this part can proceed without doing a table lookup. The 4 GB address space is divided into 4 segments, as illustrated in Figure 8.13. These segments have different mapping characteristics and serve different functions:

- **kuseg**: This 2 GB segment is cacheable and mapped to physical addresses. In the kernel mode, accesses to this segment are treated just like user mode accesses. This serves as a means for the kernel routines to access the code and data of the user process on whose behalf they are running. Thus, translations for this segment are available in the corresponding user page table (stored in **kseg2**).

---

<sup>3</sup>Our discussion of the MIPS-I virtual memory system is detailed enough to get a good appreciation of the fundamental issues involved. However, it does not cover every nuance that a MIPS OS developer needs to be aware of to develop the MMS part of the OS. Additional details can be obtained from sources such as [?].

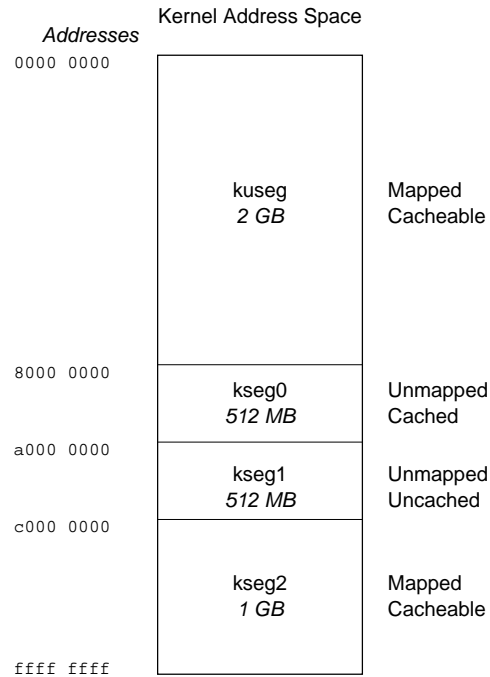


Figure 8.13: Functionality-based Partitioning of the MIPS Kernel Address Space into Segments

- **kseg0**: This 512 MB segment is cached and unmapped. Addresses within this segment are direct-mapped onto the first 512 MB of physical address space without using the TLB. This segment is typically used for storing frequently executed parts of the OS code—such as the exception handlers—and some kernel data such as the kernel page table (which stores the mapping for the **kseg2** segment). The TLB miss handler is stored in this segment.
- **kseg1**: This 512 MB segment is uncached and unmapped. It is also direct-mapped to the first 512 MB of physical address space without using the TLB. Unlike **kseg0**, however, this segment is uncacheable. It is used for the boot-up code, the IO registers, and disk buffers, all of which must be unmapped and uncached.
- **kseg2**: This 1 GB segment is cacheable and mapped to arbitrary physical addresses, like **kuseg**. Translations for this segment are stored in the kernel's page table (*root page table*). This segment is typically used to store the kernel stack, “U-area”, user page tables, and some dynamically allocated data areas.

The MIPS-I kernel mode ISA supports a paged virtual memory with 4 KB sized pages. A 32-bit address can thus be split into a 20-bit VPN part and a 12-bit page offset part. The

ISA also specifies a unified 64-entry, fully-associative TLB. The OS loads page table entries (PTEs) into the TLB, using either random replacement or specified placement. Among the 64 TLB entries, eight are *wired* entries; these do not get replaced when doing a random replacement. This wiring feature allows the OS to store a few important PTEs, such as the frequently required root PTEs. Figure 8.14 shows the format of a TLB entry. It has the following fields/bits:

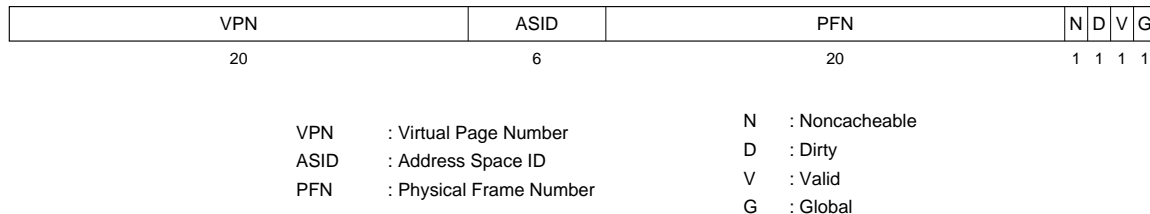


Figure 8.14: A MIPS-I TLB Entry

- **VPN** (*Virtual Page Number*):
- **ASID** (*Address Space ID*):
- **PFN** (*Physical Frame Number*):
- **N** (*Noncacheable*): This bit indicates if the mapped page is noncacheable. If it is set, the processor sends the address to main memory, bypassing the cache.
- **D** (*Dirty*): This bit indicates if the mapped page has been updated by the process.
- **V** (*Valid*): This bit indicates if the entry contains a valid mapping.
- **G** (*Global*): This bit indicates if the mapped page is shared. If it is set, the MMU does not perform an ASID match while doing a TLB lookup.

**TLB-Related Exceptions:** TLB misses in a MIPS-I system are conveyed to the OS as an exception. We can group the TLB-related exceptions into 3 categories:

- **UTLBMISS** (User mode TLB Miss): This exception category includes cases where no matching entry has been found in the TLB for an address in the **kuseg** portion of the address space. Such exceptions can occur for instruction fetch, data read, or data write. This is the exception that is likely to occur the most frequently.
- **TLBMISS** (Kernel mode TLB Miss): This category cases where either a TLB miss has occurred to the **kseg2** portion of the address space, or a matching TLB entry was found with the Valid bit not set.

- **TLBMOD:** This exception indicates that a TLB hit has occurred for a store instruction to an entry whose Dirty bit is not set. The purpose is to let the OS set the Dirty bit of the TLB entry.

Among these 3 categories, the first one has its own exception vector (0x80000000) because it is the one that occurs most frequently among the TLB-related exceptions. The last 2 categories are grouped into a single exception vector (0x80000080).

**Privileged Registers:** When a TLB miss exception occurs, information about the exception event (such as the faulting virtual address) needs to be conveyed to the OS. It is customary to use a privileged register to store this information. Similarly, the OS might require additional special registers for manipulating the TLB. The MIPS kernel mode ISA provides 6 privileged registers—**EntryHi**, **EntryLo**, **Index**, **Random**, **BadVAddr**, and **Context**—for conveying additional information regarding TLB miss exceptions to the OS and for the OS to modify the TLB. These additional registers are depicted in Figure 8.15, and are described below:

- **EntryHi:** This register is used to store a VPN and an ASID (Address Space Identifier).
- **EntryLo:** This register is used to store a PFN and status information. The information in a TLB entry is equivalent to the concatenation of **EntryHi** and **EntryLo** values.
- **Index:** This register is used by the OS to store the index value to be used for accessing a specific TLB entry. The 6-bit **Index** field of this register can store values between 0-63.
- **Random:** This register holds a pseudo-random value, which is used as the TLB index when the OS performs random replacement of TLB entries. The MMU hardware automatically increments it every clock cycle so that it serves as a pseudo-random generator.
- **BadVAddr:** The MMU hardware uses this register to save the bad virtual address that caused the latest addressing exception.
- **Context:** This register is used to store some information that facilitates the handling of certain kinds of exceptions such as TLB misses. The **PTEBase** field is set by the OS, normally to the high-order address bits of the current user ASID's page table, located in the **kseg2** segment of the kernel address space. The **BadVPN** field is set by the MMU hardware when the addressing exception occurs.

**Privileged Instructions:** The kernel mode ISA also provides the following privileged instructions for the OS to refill the TLB:



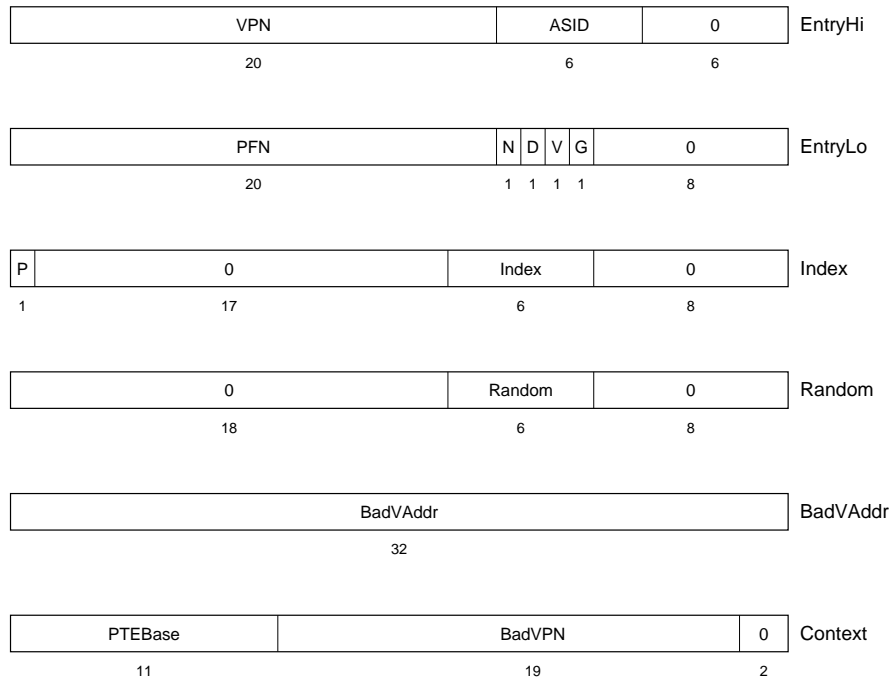


Figure 8.15: Privileged Registers Present in the MIPS-I Kernel Mode ISA for Supporting Virtual Memory

- **tlbwr:** Write the PTE present in **EntryHi-EntryLo** pair onto the TLB at index provided in register **Random**. This instruction is used by the OS to insert a PTE randomly into the TLB.
- **tlbwi:** Write the PTE present in **EntryHi-EntryLo** pair onto the TLB entry indexed by register **Index**. This instruction is used by the OS to insert a PTE at a specified index in the TLB.
- **tlbr:** Read TLB entry indexed by register **Index**, and place the information into **EntryHi** and **EntryLo** registers.

**TLB Operation:** Let us put in a nutshell the operation of the TLB hardware. If the system is in the Kernel mode, all accesses to the **kseg0** or **kseg1** segments bypass the TLB, and are direct-mapped as described earlier. If the access is to **kuseg** (in either mode) or to **kseg2**, then the MMU determines the VPN and associatively compares it against the VPN field of all TLB entries. If there is a match, and if either the **ASID** field of **EntryHi** register matches that of the matching TLB entry or if the TLB entry's Global bit is set, then translation can continue. If no such entry is found in the TLB, a **UTLBMISS** exception

is generated if the address is to **kuseg**, and a TLBMISS exception is generated if the address is to **kseg2**. If a matching TLB entry was found, but its Valid bit was not set, then a TLBMISS exception is generated, irrespective of whether the address is to **kuseg** or **kseg2**. In all of these cases, the faulting VPN is placed in the **Context** register.

### 8.2.7.2 Operating System Usage of Machine Features

With the above background on the features provided in the MIPS-I kernel ISA to support virtual memory, let us turn our attention to the barebones of a typical MMS used in MIPS-I based systems. This MMS organizes each user page table as a simple linear array of PTE entries. Each PTE is 4 bytes wide, and its format matches the bitfields of the **EntryLo** register. Each user ASID's PT requires 2 MB of space. All of the user ASID page tables are placed in the **kseg2** part of the kernel address space. The kernel PT stores the mapping for the **kseg2** segment only. The entire kernel PT (part of which contains the root PTEs) is stored in **kseg0**. The placement of the page tables is pictorially shown in Figure 8.16. Each kernel PTE stores the current mapping for 4 KB of **kseg2**. If the kernel PTE happens to be a root PTE, then this 4 KB stores 1 K entries of a user page table, and that root PTE effectively covers 1 K user PTEs, or  $1\text{ K} \times 4\text{ KB} = 4\text{ MB}$  of the user address space.

The OS makes sure that whenever a user process is being run, the ASID field of the **EntryHi** register reflects the user ASID and the **PTEBase** field of the **Context** register reflects the base virtual address of the user page table.

A UNIX-like OS typically uses the 8 *wired* entries of the TLB for storing kernel PTEs as follows: one for a kernel PTE that maps 4 KB of the kernel stack, one for a kernel PTE that maps 4 KB of the U-area, one each for a root PTE that covers 4 MB each of the currently active user {text, data, stack}, and up to 3 more for root PTEs that cover 4 MB each of user data.

We shall focus our discussion on UTLBMISS, the most frequent TLB-related exception. When a UTLBMISS exception occurs, the **Context** register gives the **kseg2** virtual address of the user PTE to be read to service the TLB miss. When the UTLBMISS handler attempts to read the PTE present at this **kseg2** virtual address, an address translation is required because **kseg2** is a mapped segment. If a TLB entry is available for this **kseg2** virtual address (most likely, this should be one of the *wired* entries in the TLB), then the OS can successfully access the required user PTE with just one memory access. In the worst case, this TLB lookup for a **kseg2** virtual address can cause a TLBMISS exception. The TLBMISS handler will read the root PTE stored in the unmapped **kseg0** segment, enter this root PTE in the TLB, and return control back to the UTLBMISS handler.

After obtaining the required user PTE from **kseg2**, the UTLBMISS handler updates the TLB with this PTE. Assembly language code for an UTLBMISS handler is given below. Notice that the **rfe** instruction (at the end), which puts the system back in the user mode is in the delay slot of the **jr \$k1** instruction, which specifies the transfer of control back to the user program that caused the TLB miss.



## An Assembly Language UTLBMISS Handler for a MIPS-I System

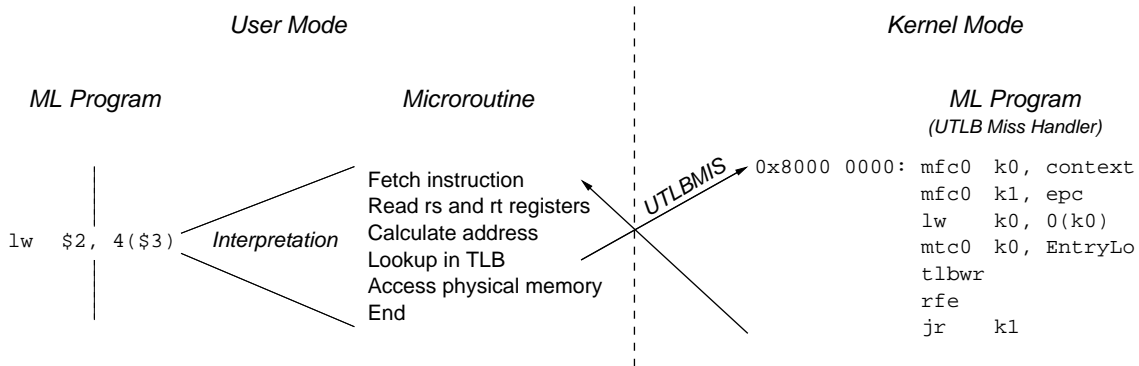


Figure 8.17: A Program Execution View of a UTLBMISS Exception

## 8.2.8 Interpreting a MIPS-I Memory-Referencing Instruction

In the previous chapter, we had looked at the interpretation of a memory-referencing instruction, without considering the address translation process. The calculated (virtual) address was directly copied to **MAR**, from where it was supplied to the memory subsystem. In this section, we shall briefly revisit the interpretation of memory-referencing instructions, by considering the address translation process also. Again, we tailor the discussion to the MIPS-I ISA. For ease of understanding, we modify the familiar data path of Figure 8.1 to include a *memory management unit*. The modified data path is given in Figure 8.18. It includes a microarchitectural register called **VAR** for holding the virtual address to be translated. It also includes a software-managed TLB. The microarchitectural register **MAR** now consists of 2 fields. The **Offset** field is directly updated from the **Offset** field of **VAR**. The **PFN** field can be updated either from the TLB or directly from the **VPN** field of **VAR**.

Table 8.4 gives a modified MAL routine for interpreting a MIPS **lw** instruction in this data path. In step 6, the calculated virtual address is copied from **AOR** to **VAR** (instead of **MAR**). In step 7, the **VPN** field of this address is supplied to the TLB. If it results in a TLB miss, then a TLB miss exception is activated by transferring control to MAL step 10. On the other hand, if there is a TLB hit—which will be the frequent case—the **PFN** field of the concerned TLB entry is copied to **MAR**'s **PFN** field. The **Offset** field of **VAR** is also copied to the corresponding field of **MAR**.

The TLB miss exception is initiated in steps 10-15. This part of the routine is similar to the initiation of the overflow exception that we saw earlier. The main points to note are: (i)

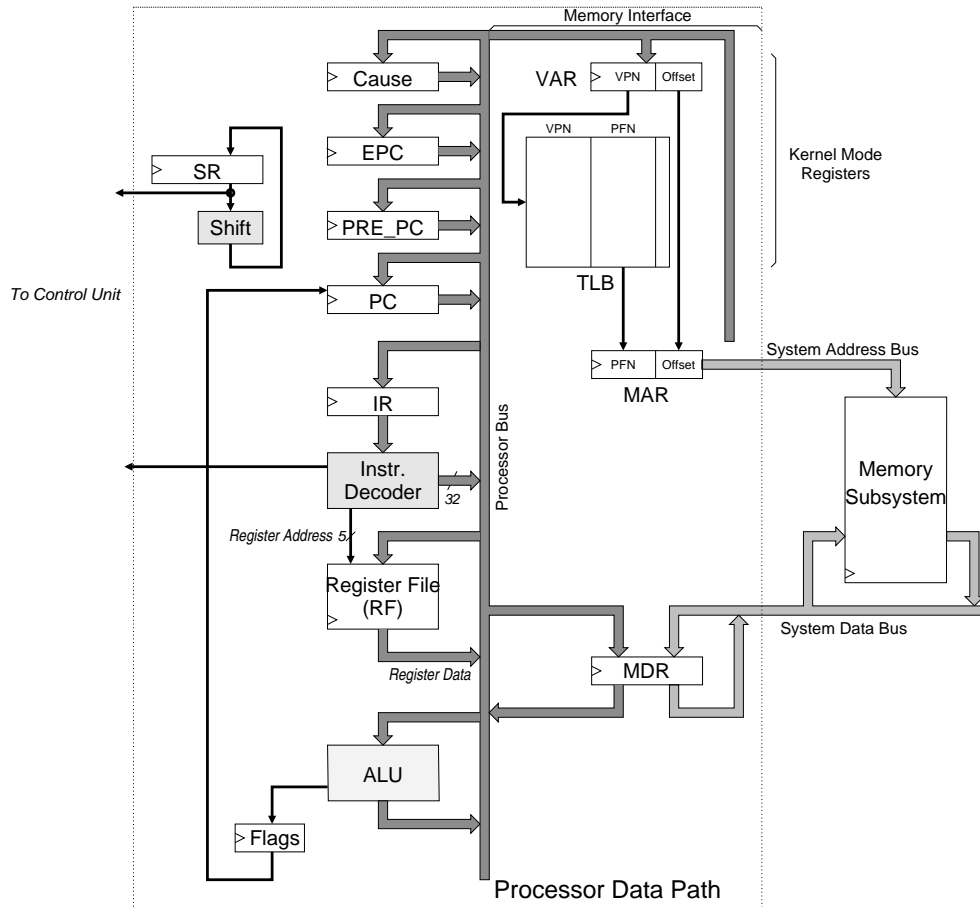


Figure 8.18: A Data Path for Implementing the MIPS-0 Kernel Mode ISA Including Virtual Memory

the **BadVPN** field of **Context** register is updated with the virtual page number that missed in the TLB; (ii) **PC** is updated with **0x80000000** or **0x80000080** depending on whether the TLB miss occurred in the user mode or the kernel mode, respectively.

### 8.2.9 Combining Cache Memory and Virtual Memory

The discussion of virtual memory clearly parallels the cache memory concepts discussed in the previous chapter. There are many similarities and some differences. The cache memory bridges the *speed gap* between the processor and the main memory and is implemented entirely in hardware. The virtual memory mechanism bridges the *size gap* between the main memory and the virtual memory and is usually implemented by a combination of

| Step No.                | MAL Instruction for Data Path  | Comments  |
|-------------------------|--|---|
| Execute phase           |  |   |
| 4                       | R[rs] → AIR  |   |
| 5                       | SE(offset) + AIR → AOR   |   |
| 6                       | AOR → VAR  |   |
| 7                       | if (VAR.VPN misses in TLB) goto step 10<br>else TLB[hit_index].PFN → MAR.PFN;<br>VAR.Offset → MAR.Offset |   |
| 8                       | M[MAR] → MDR   |   |
| 9                       | if (rt) MDR → R[rt]  |   |
| Goto step 0             |  |   |
| Take TLB Miss Exception |  |   |
| 10                      | SR<3:0> << 2 → SR<5:0>   | Set system in kernel mode and disable interrupts<br>Save instruction's address<br>Save bad virtual address<br>Save bad VPN<br>Set PC to 0x80000000 or 0x80000080<br>Place TLBL code in cause<br>Goto step 0 |
| 11                      | Prev_PC → EPC  |   |
| 12                      | VAR → BadVAddr   |   |
| 13                      | VAR.VPN → Context.BadVPN   |   |
| 14                      | 0x80000000   (SR<3> << 6) → PC   |   |
| 15                      | 2 → Cause.ExcCode  |   |

Table 8.4: A MAL Routine for the Execute Phase of the Interpretation of the MIPS-0 ISA Instruction Represented Symbolically as `lw rt, offset(rs)`. This MAL Routine is for executing the ML instruction in the Data Path of Figure 8.18

hardware and software techniques. Table 8.5 provides a succinct comparison of the two concepts and their typical implementations.

The descriptions of cache memory and virtual memory were given in isolation.

## 8.3 IO System Organization

Another major component of the kernel mode microarchitecture is the IO system, which houses the IO devices, their interfaces, and their interconnections. In terms of size, the IO system is usually the largest part of any computer. This is because the IO devices are electromechanical devices, which cannot be easily miniaturized. The IO interfaces also tend to be bulky, as they are built out of several integrated circuits (IC chips). Moreover, most of the IO interconenctions are made out of cables that are external to the IC chips.

The IO system performs the following functions:

| No. | Cache Memory   | Virtual Memory   |
|-----|--|--|
| 1.  | Cache hit time is small (1-2 cycles)   | Physical memory hit time is large (10-100 cycles)                |
| 2.  | Cache miss time is large (10-100 cycles)   | Page fault handling time is very large                           |
| 3.  | Block size is small (4-32 words)   | Page size is large (1K-1M words)                                 |
| 4.  | Usually direct mapping or low level of associativity                                 | Usually fully-associative mapping or high level of associativity |
| 5.  | On a cache miss, the cache hardware loads the block from main memory to cache memory | On a page fault, the OS loads the page from disk to main memory  |
| 6.  | The cache memory is usually not visible in the ISA                                   | The virtual memory is visible in the kernel mode ISA             |

Table 8.5: A Succinct Comparison of Cache Memory and Virtual Memory

- Implement the IO address space defined in the kernel mode ISA
- Implement the IO interface protocols of the IO devices
- Implement the IO devices themselves

The first two functions are definitely related, and we shall explain them in more detail. Implementation of various IO devices is discussed in Appendix \*\*.

### 8.3.1 Implementing the IO Address Space: IO Data Path

We shall begin our discussion of the IO system with the topic of implementing the IO address space defined in the kernel mode ISA, as we saw in Section 4.3. This address space can be part of the memory address space (as in memory-mapped IO) or a separate address space (as in independent IO). Irrespective of whether the IO register space is memory-mapped or independently mapped, the hardware registers that implement this name space are built separately from the memory system. When the processor executes an IO instruction—an instruction that accesses the IO address space—it issues a signal indicating that the address on the system bus is an IO address.

Unlike monolithic structures such as the register file used to implement the general-purpose registers, the structures used to implement the IO address space are distributed. That is, the set of hardware registers implementing the IO address space is distributed over different hardware structures, depending on their function. For instance, the IO registers that relate to the keyboard device will be physically located within the keyboard interface module, and those that relate to the mouse will be physically located within the mouse

interface module. This type of distribution is required because the behavior of IO registers vary, depending on the IO device they relate to, as discussed in Section 4.3. The number of IO addresses allotted for an IO device is generally small, of the order of a few tens or less; an exception is video memory.

Design of the IO data path is a very complex subject, perhaps even more complex than that of the processor data path and the memory data path. Therefore, we shall present this topic in a step by step manner, beginning with very simple data paths and steadily increasing the complexity. The simplest IO data path we present consists of a single system bus that connects several IO registers as shown in Figure \*\*\*\*.

GIVE MAL ROUTINE FOR an `lw` instruction that is an IO instruction. Let this MAL routine be related to the one given in virtual memory.

GIVE MAL ROUTINE for a MIPS-like `in` instruction.

When comparing the register file and the IO hardware registers, besides physical distribution, there is another major difference: a register in a register file has a fixed address whereas an IO hardware register may not have a fixed address. Like a physical memory location in a virtual memory system, an IO hardware register also can map to different logical IO register addresses at different times.

In this section, we give an overview of selected aspects of computer IO and communication between the processor and IO devices. Because of the wide variety of IO devices and the quest for faster handling of programs and data, IO is one of the most complex areas of computer design. As a result, we present only selected pieces of the IO puzzle. We start with a discussion of IO interface modules, which are pivotal in implementing the IO registers specified in the kernel mode ISA. These interface modules come in various kinds, and provide different types of ports for connecting various IO devices. We consider the interface module for connecting a keyboard as an illustration. We then introduce the system bus and other types of buses to connect the IO interface modules to the processor-memory system.

#### 8.3.1.1 System Bus

The IO hardware registers must be connected to the rest of the data path — the processor and memory system — for them to be accessible by IO instructions. The most convenient way to carry out this connection is to extend the processor-memory bus that we used earlier to connect the processor and memory system. When a single bus is used in this manner to connect all of the devices—the processor, the memory, and the IO registers—the bus is usually called a *system bus*. Such a connection was illustrated in Figure 8.19.

#### 8.3.2 Implementing the IO Interface Protocols: IO Controllers

Unlike the general-purpose registers and the memory locations, the IO registers are generally implemented in a distributed manner, using special register elements inside the different IO



interface modules. To perform the functionality specified in an IO port specification, an *IO interface module* is used. The interface module that implements the port interacts with the appropriate IO devices. From an IO device's viewpoint, its interface module serves as a conduit for connecting it to the rest of the computer system. An IO interface module performs one or more of the following functions:

- It interacts with IO devices to carry out the functionality specified for the IO port that it implements. For example, a keyboard port may specify that whenever its status bit is set, its data register will have a code value that corresponds to the last key depressed or released in a keyboard.
- It performs conversion of signal values and data formats between electromechanical IO devices and the processor-memory system.
- It performs necessary synchronization operations for correct transfer of data from slow IO devices.
- It performs necessary buffering of data and commands for interfacing with slow IO devices.
- It performs direct transfer of data between IO devices and memory.

One side of an IO interface module connects to a system bus, and the other side connects to one or more IO devices via tailored data links, as shown in Figure 8.19. The IO interface module can be thought of as a small processor that has its own register set and logic circuitry.

It is worthwhile to point out that most of the IO devices are electromechanical devices, with some electronic circuitry associated with it. This circuitry carries out functions that are very specific to that device. For example, in a printer, this circuitry controls the motion of the paper, the print timing, and the selection of the characters to be printed. The printer interface module does not carry out these functions. Thus, an interface module directly interacts only with the electronic circuitry, and not with the mechanical parts of the device.

When an IO write instruction writes a value to an interface module's status/control register, it is interpreted by the module as a command to a particular IO device attached to it; and the module sends the appropriate command to the IO device.

### 8.3.3 Example IO Controllers

The IO controllers consist of the circuitry required to transfer data between the processor-memory system and an IO device. Therefore, on the processor side of the module we have the IO registers that are part of the kernel mode ISA, along with circuitry to interact with the bus to which it is connected on the processor side. On the device side we have a data path with its associated controls, which enables transfer of data between the module

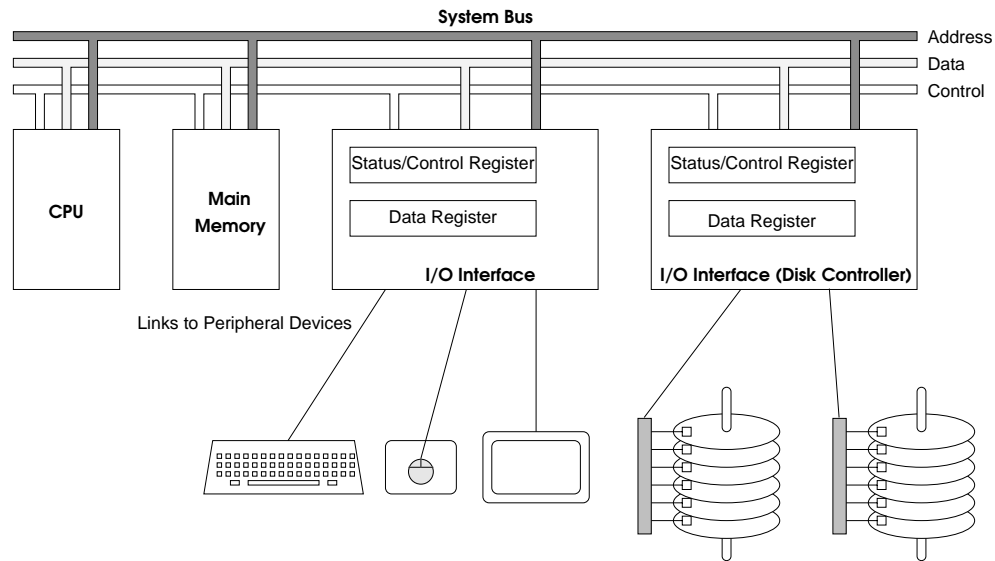


Figure 8.19: Use of IO Interface Modules to Implement IO Ports to Connect IO Devices

and an IO device. This side may be either designed or programmed at run-time to be device-dependent.

#### 8.3.4 Frame Buffer:

aka **Display memory** and **Video memory**.

A framebuffer is a video output device that drives a video display from a memory buffer containing a complete frame of data. The information in the buffer typically consists of color values for every pixel (point that can be displayed) on the screen. Color values are commonly stored in 1-bit monochrome, 4-bit palettized, 8-bit palettized, 16-bit highcolor and 24-bit truecolor formats. An additional alpha channel is sometimes used to retain information about pixel transparency. The total amount of the memory required to drive the framebuffer depends on the resolution of the output signal, and on the color depth and palette size.

Before an image can be sent to a display monitor, it is first represented as a bit map in an area of video memory called the frame buffer. The amount of video memory, therefore, dictates the maximum resolution and color depth available.

With a conventional video adapter, the bit map to be displayed is first generated by the computer's microprocessor and then sent to the frame buffer. Most modern video adapters, however, are actually graphics accelerators. This means that they have their own microprocessor that is capable of manipulating bit maps and graphics objects. A small

amount of memory is reserved for these operations as well.

Because of the demands of video systems, video memory needs to be faster than main memory. For this reason, most video memory is dual-ported, which means that one set of data can be transferred between video memory and the video processor at the same time that another set of data is being transferred to the monitor. There are many different types of video memory, including VRAM, WRAM, RDRAM, and SGRAM. The standard VGA hardware contains up to 256K of onboard display memory. While it would seem logical that this memory would be directly available to the processor, this is not the case. The host CPU accesses the display memory through a window of up to 128K located in the high memory area. (Note that many SVGA chipsets provide an alternate method of accessing video memory directly, called a Linear Frame Buffer.) Thus in order to be able to access display memory you must deal with registers that control the mapping into host address space. To further complicate things, the VGA hardware provides support for memory models similar to that used by the monochrome, CGA, EGA, and MCGA adapters. In addition, due to the way the VGA handles 16 color modes, additional hardware is included that can speed access immensely. Also, hardware is present that allows the programmer to rapidly copy data from one area of display memory to another. While it is quite complicated to understand, learning to utilize the VGA's hardware at a low level can vastly improve performance. Many game programmers utilize the BIOS mode 13h, simply because it offers the simplest memory model and doesn't require having to deal with the VGA's registers to draw pixels. However, this same decision limits them from being able to use the infamous X modes, or higher resolution modes.

**Host Address to Display Address Translation** The most complicated part of accessing display memory involves the translation between a host address and a display memory address. Internally, the VGA has a 64K 32-bit memory locations. These are divided into four 64K bit planes. Because the VGA was designed for 8 and 16 bit bus systems, and due to the way the Intel chips handle memory accesses, it is impossible for the host CPU to access the bit planes directly, instead relying on I/O registers to make part of the memory accessible. The most straightforward display translation is where a host access translates directly to a display memory address. What part of the particular 32-bit memory location is dependent on certain registers.

#### 8.3.4.1 Universal Asynchronous Receiver/Transmitter (UART)

The transfer of data between two blocks may be performed in parallel or serial. In parallel data transfer, each bit of the word has its own wire, and all the bits of an entire word are transmitted at the same time. This means that an  $N$ -bit word is transmitted in parallel through  $N$  separate conductor wires. In serial data transmission, bits in a word are sent in sequence, one at a time. This type of transmission requires only one or two signal lines. The key feature of a serial interface module is a circuit capable of communicating in bit-serial fashion on the device side (providing a serial port) and in bit-parallel fashion on the bus side

(processor side). Transformation between parallel and serial formats is achieved with shift registers that have parallel access capability. Parallel transmission is faster, but requires many wires. It is used for short distances and when speed is important. Serial transmission is slower, but less expensive, because it requires only one conductor wire.

The UART is a transceiver (**transmitter/receiver**) that translates data between parallel and serial interfaces.

The UART sends a *start bit*, five to eight data bits with the least-significant-bit first, an optional *parity bit*, and then one, one and a half, or two *stop bits*. The start bit is the opposite polarity of the data-line's idle state. The stop bit is the data-line's idle state, and provides a delay before the next character can start. (This is called asynchronous start-stop transmission).

#### 8.3.4.2 DMA Controller

DMA transfers are performed by a special device controller called a **DMA controller**. The DMA controller performs the functions that would normally be performed by the processor when accessing the main memory. For each word transferred, it provides the memory address and all the bus signals that control data transfer. The DMA controller also increments the memory address after transferring each word, and keeps track of the number of words transferred. A DMA controller may handle DMA operations for a number of IO devices, or may be dedicated to a single IO device. The latter controller is called a *bus-mastering* DMA controller.

Some computer systems permit DMA operations between two IO devices without involving main memory. For example, a block transfer can be performed directly between two hard disks, or from a video capture device to a display adapter. Such a DMA operation can improve system performance, especially if the system provides multiple buses for simultaneous transfers to happen in different parts of the system.

#### 8.3.4.3 Keyboard Controller

We shall look at a specific example—an interface module for a keyboard—to get a better appreciation of what IO interface modules are and how they work. An interface module that connects a keyboard is one of the simplest and most commonly used interface modules in a general-purpose computer. To study its design and working, we need to know a little bit about how the keyboard device and the associated controller circuitry work.

Figure 8.20 shows a keyboard unit, along with its connections to the interface module. Whenever a key is depressed or released, a corresponding *K-scan code* is generated and sent to the encoder. The encoder converts the K-scan code to a more standard *scan code*, and sends it to the interface module—housed in the system unit—usually through a serial keyboard cable. Notice that if a key is depressed continuously, after a brief period the

keyboard unit repeatedly sends the scan code for the key's *make code* until the key is released.

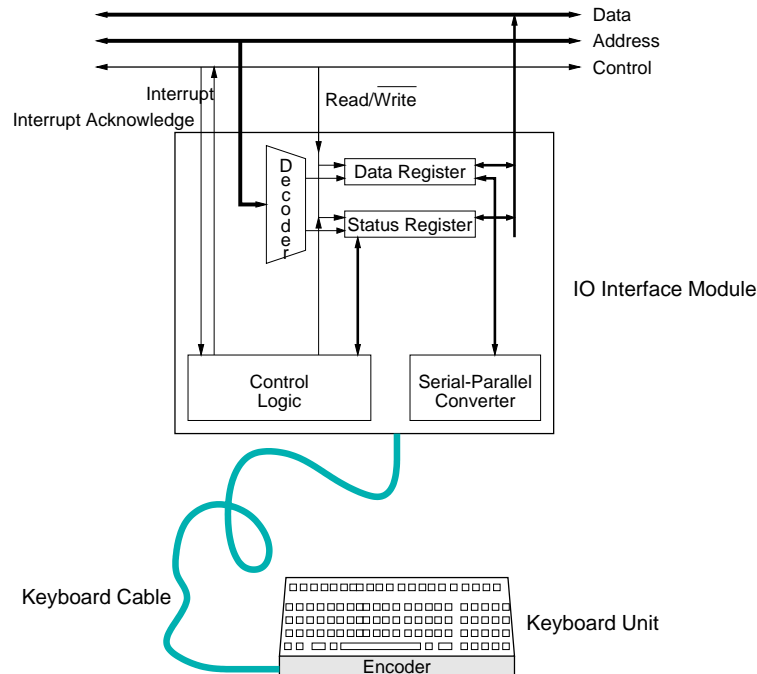


Figure 8.20: Role of IO Interface Module for Connecting a Keyboard Unit

With this background on the working of a keyboard unit, let us summarize the major functions the interface module needs to perform:

- Incorporate the IO registers (a small portion of the IO address space) that are part of the keyboard port.
- Convert the scan code arriving serially from the keyboard unit into parallel form, and place it in an IO register to be read by the processor.
- Maintain the status of the keyboard circuitry—interface module and the keyboard device—in an IO register to be read by the processor.
- Generate an interrupt at the occurrence of keyboard-related events, such as updation of the data register with a scan code.
- Process processor commands, such as varying the keyboard's *typematic rate* and *delay*. This provides a means for the keyboard user to specify to the OS the desired repetition rate.

Figure 8.20 also shows the internals of an interface module that can perform the above functions<sup>4</sup>. It incorporates a **data register**, a **status register**<sup>5</sup>, an address decoder, a serial-parallel converter, and control logic. The module works as follows. Upon receiving a scan code from the keyboard unit, the serial-parallel converter converts it to parallel form, and places the code in the **data register**. The control logic makes appropriate modifications to the **status register**, and raises an interrupt to inform the presence of a valid value in the **data register**.

The interrupt causes the execution of the ISR part of the keyboard device driver, which reads the **input data register**, and copies it to the corresponding process' keystroke buffer. The scan code is then converted into ASCII code or other meaningful data by the systems software, which usually also displays the character on the monitor.

When the keyboard device driver wants to send a command, such as varying the repetition rate, it executes IO write instructions to update the **data register** and the **status register**. The control logic, upon sensing the update, conveys the information to the serial-parallel converter, which converts it to serial form and sends it to the keyboard unit.

### 8.3.5 IO Configuration: Assigning IO Addresses to IO Controllers

Within a computer system, from the point of view of the processor (and the device drivers), an IO controller is uniquely identified by the set of IO addresses assigned to its hardware registers. An IO instruction would specify an IO address, and the corresponding IO controller would respond.

An issue that we have skirted so far is: how are IO addresses assigned to IO controllers? And, how does an IO controller know the addresses assigned to its IO registers? A simple-minded approach is for the kernel mode ISA to specify the addresses assigned to each IO register. For example, kernel mode addresses 0x a0000000 - 0x a0000007 can be earmarked for the mouse controller, addresses 0xa00000008 - 0xa000000ff for the keyboard, and so on. The device drivers would then be written with these addresses hardcoded into them. The designer of an IO controller can also hardcode addresses into the design so that its address decoder recognizes only the IO addresses assigned to it. However, this approach precludes standardization of IO controllers across different computer families, driving up controller costs.

A slightly better option is not to hardcode the addresses in the IO controller, but provide an ability to manually enter the addresses by setting switches or jumper connections on the controller, before installing the controller in a computer system. Once the controller knows the addresses assigned to it, its decoder can specifically look for those addresses in the

---

<sup>4</sup>In XT systems, for example, this function is implemented by an Intel 8255A-5 or compatible keyboard interface chip.

<sup>5</sup>The number of registers incorporated in an IO interface module, in general, is higher than the number of IO registers visible for the port at the kernel mode ISA level. This is similar to the processor microarchitecture which incorporates a number of registers in addition to the ones defined in the ISA.

future. Although this solves the standardization problem, manual entering of addresses is, nevertheless, clumsy.

### 8.3.5.1 Autoconfiguration

Autoconfiguration shifts the burden of deciding the IO addresses to the OS. In the basic autoconfiguration scheme, called **plug-and-play**, the *Plug and Play Manager* part of the OS enters the addresses into the controller at boot-up time. This requires some extra instructions in the boot-up code, but removes the burden of manually entering the addresses. The astute student might wonder how the Plug and Play Manager will communicate to an IO controller that does not know the addresses it should respond to! What address will the Plug and Play Manager use for getting the controller's attention? Consider the following classroom scenario. In the first day of class, the professor wants to assign his students unique ID numbers that should be used throughout the semester. He wants to convey the ID numbers to them without calling out their names. One way he could do this is by calling out the individual seat numbers and telling the corresponding ID number. Once all students know their ID numbers, individual students can be called out using their ID numbers, and the students are free to sit wherever they like.

In a similar manner, before an IO controller in the system gets its addresses, we need a different attribute to identify the controller. In the plug-and-play scheme, a unique number is associated with each IO connector position (analogous to class seat number). These unique connector numbers help to create yet another address space, called the *configuration address space*. Depending on the connector into which an interface module is plugged, the module has a unique range of configuration addresses. Each module also has a *configuration memory*, which stores information about the type and characteristics of the module.

During the interface initialization part of boot-up, the Plug and Play Manager reads the configuration memory of each IO controller that is hooked up to the system; it uses the module's configuration addresses to do this access. The configuration information uniquely identifies the controller, and provides information concerning the device drivers and resources (such as DMA channels) it requires. After obtaining the information present in the configuration memory, the Plug and Play Manager assigns IO addresses to the IO registers present in the controller. Thereafter, the OS uses these IO addresses to communicate with the controller.

In the above scheme, address assignment is done only during boot-up time. This scheme mandates that the computer be reset every time a new IO card is connected to the system. While this scheme may be acceptable for IO controllers that connect to disk drives and CD-ROM drives, which are rarely connected when the system is working (especially in a desktop environment), it is somewhat annoying for IO controllers that connect to keyboards, mice, speakers, etc, which may be connected or disconnected when the system is in operation. Furthermore, in mission-critical computers, downtimes can be extremely costly. In order to avoid resetting the computer every time a new IO controller is plugged in, modern

computer systems enhance the plug-and-play feature with a feature called **hot plug** or **hot swap**. With this feature, if a new controller is connected when the computer system is in operation, the system recognizes the change, and configures the new controller with the help of systems software. This requires more sophisticated software and hardware (such as protection against electrostatic discharge) than does plug-and-play. Recent IO buses such as USB and PCIe are particularly geared for the hot swap feature, as they incorporate a tree topology. In older standards such as the PCI, the hot swap feature can be incorporated by using a dedicated secondary PCI bus at each primary PCI slot, as illustrated in Figure 8.21. Because each secondary PCI bus has a single PCI slot, an IO controller can be added or removed without affecting other controllers.

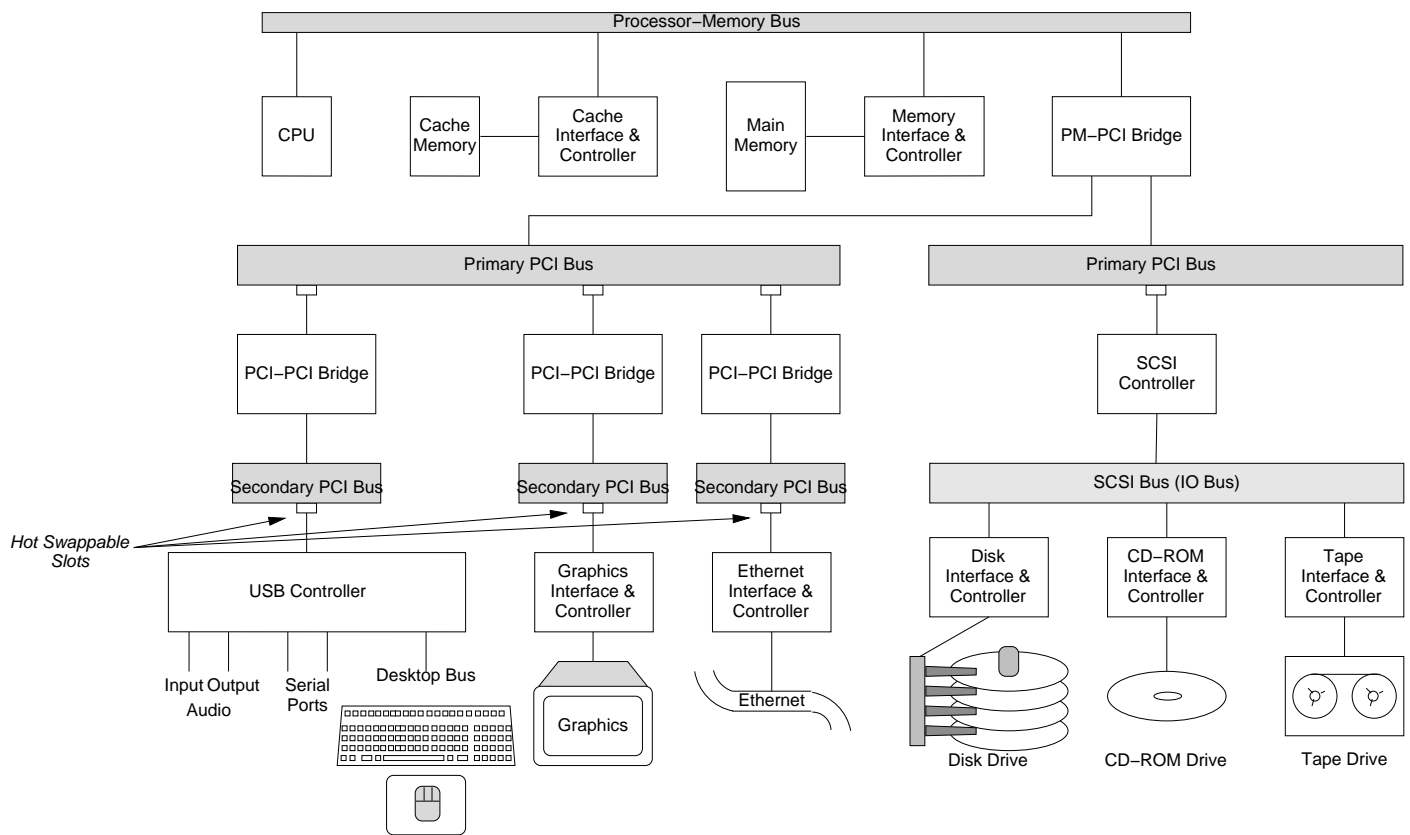


Figure 8.21: Hot Pluggable PCI Slots



## 8.4 System Architecture

As with the processor system and the memory system, many of the characteristics of the IO system are driven by technology. For example, the properties of disk drives affect how the disks are connected to the processor, as well as how the operating system interacts with them. IO systems differ from the processor and memory systems in several important ways. Although processor-memory designers often focus primarily on performance, designers of IO systems must consider issues such as expandability and resilience in the face of failure as much as they consider performance. Second, for an IO system, characterizing the performance is a more complex characteristic than for a processor. For example, with some devices we may care primarily about access latency, whereas with others bandwidth—the number of bits that can be transferred per second—is more important. Furthermore, performance depends on many aspects of the system: the device characteristics, the connection between the device and the rest of the system, the memory hierarchy, and the operating system.

Until recently, computer performance was almost exclusively judged by the performance of the processor-memory subsystem. In the past few years, the situation has begun to change with users giving more importance to overall system performance. This calls for balanced systems with IO subsystems that perform as good as the processor-memory subsystem.

The IO controllers need to be connected to the processor-memory system in order for them to serve as IO ports for hooking up IO devices. There are several ways in which this connection can be made. The manner in which the controllers are connected affect the system performance in a major way. The earliest computers connected each controller directly to the processor using a direct connection. The resulting connection topology is a *star*, as shown in Figure 8.22. In addition to the direct connections to the processor, DMA controllers, if present, are connected to the main memory as well. The disadvantage of a star topology is that it requires several cables, at least one for each controller. Moreover, each controller has to be tailored for a particular processor, and cannot be used in computers having a different processor.

### 8.4.1 Single System Bus

In order to reduce the number of cables required, the simplest approach that we can think of is to hook up all of the controllers to the processor-memory bus, which connects the processor and the main memory. When a single bus is used in this manner to connect all of the major blocks—the processor, the main memory, and the controllers—the bus is usually called a *system bus* (as opposed to a processor-memory bus). Such a connection was illustrated in Figure 8.19. The two major advantages of hooking up the controllers to a bus instead of using tailored connections are versatility and low cost due to fewer cables. Adding new devices and their controllers to the system is very straightforward.

A bus has its own data transfer protocol, and so in a single-bus system, the data transfer

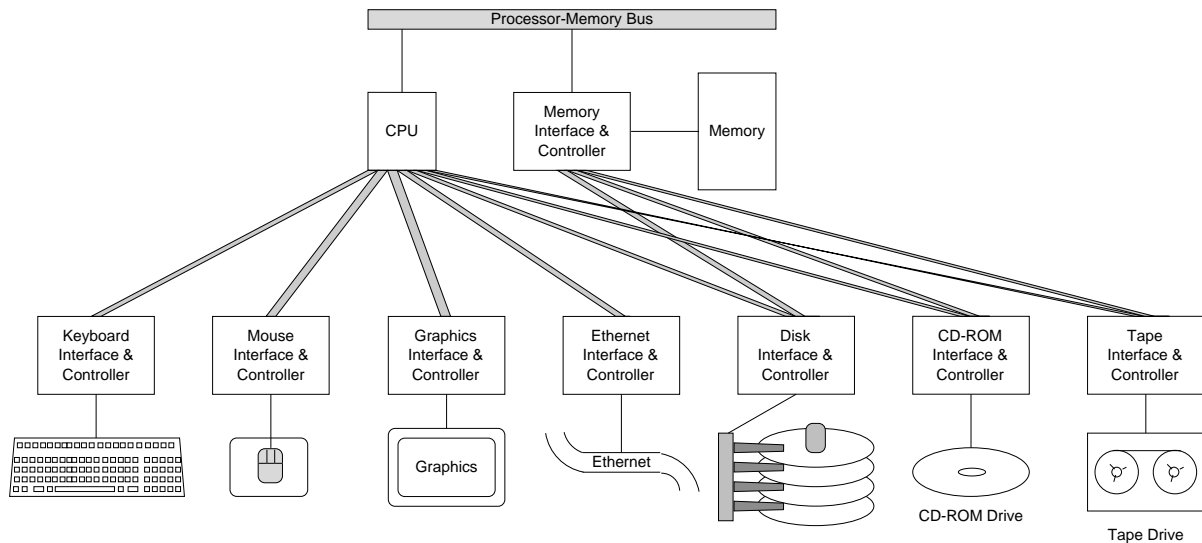


Figure 8.22: Connecting IO Controllers to the Processor via a Star Topology

protocol between any two devices in the system is identical. The operation of the system bus is, in fact, quite straightforward. For each data transfer, the master device (usually the processor or a DMA controller) places the slave address (a memory address or an IO register address) on the address lines of the bus. The data lines are used to transfer the data value to be written or the value that is read from a location. The control lines are used to specify the nature of the transfer, and also to coordinate the actions of the sending and receiving units.

A computer that used a single system bus, and became very popular during its time in the 1970s was PDP-11, a minicomputer manufactured by Digital Equipment Corporation (DEC). Its system bus was called *Unibus*. The earliest PCs also had only a single system bus, as the first and second generation CPUs ran at relatively low clock frequencies, which permitted all system components to keep up with those speeds.

#### 8.4.2 Hierarchical Bus Systems

A single bus configuration that interconnects the processor, the memory system, and the IO controllers has to balance the demands of processor-memory communication with those of IO interface-memory communication. Such a setup has several drawbacks:

- The use of a single system bus permits only a single data transfer between the units at any given instant, potentially becoming a communication bottleneck.
- The use of a single system bus may impose a severe practical restriction on the number

of controllers that can be connected to the bus without increasing the bus capacitance and skew effects. A single bus may have high electrical capacitance due to two reasons: (i) the bus ends up being long as it has to connect all devices in the system, and (ii) each device that is hooked up to the bus adds to the capacitance. To reduce the capacitance, each device must have expensive low-impedance drivers to drive the bus.

- Because of the nature of most of the IO devices, there is a significant mismatch in speed between transfers involving their controllers and the high-speed transfers between the processor and memory. If we use a single high-speed system bus, then we are forced to use expensive, high clock speed, controllers<sup>6</sup>, but without significant improvements in overall performance. By contrast, if we use a low-speed system bus, then the system will have poor performance.
- If we use a single system bus, many of its specifications (such as word size) will be tied to the ISA (instruction set architecture), and therefore tend to differ widely across different computer families, and even between family members. This forces us to design specific controllers for each computer family, which increases the cost of IO controllers.

These issues are a concern, except for small computers in which the processor, the main memory, and the IO controllers are placed on a single printed-circuit board (PCB) or a single chip (SoC), allowing the system bus to be fully contained within the board. Often, such systems do not require high performance either. Larger computers such as main-frames and modern desktops, on the other hand, incorporate a large number of IO controllers, making it difficult to place all components on a single board. These IO devices tend to have widely different speeds, and the system as a whole tends to require high IO throughputs. All of these make it impossible to use a single system bus. Therefore, larger computer systems use multiple buses to connect various devices. These buses are typically arranged in a hierarchical manner, with the fastest bus at the top of the hierarchy and the slower buses towards the bottom of the hierarchy<sup>7</sup>. Devices are connected to the bus hierarchy at the level that matches their speed and bandwidth requirements. Activity on one bus does not necessarily obstruct activity on another.

Figure 8.23 illustrates this concept by depicting a 2-level hierarchical arrangement of buses. The bus at the top of the hierarchy is typically used to connect the processor and the memory, and is called a processor-memory bus as before. Apart from the processor-memory bus, there is a (slower) IO bus that connects all of the controllers together, and forms the backbone of the IO system. The IO system is connected to the processor-memory

---

<sup>6</sup>If the IO controllers are made to operate at the same clock speeds as the processor, they will become very expensive, without substantially enhancing the IO throughput, which depend primarily on the IO devices themselves.

<sup>7</sup>A faster bus does not imply a higher speed of electrical transmission through the wires, but rather a shorter time between meaningful bus events (the “bus clock period”). The controllers connected to a faster bus have to react more quickly, which calls for more rigorous engineering.

system by means of an **IO bus bridge**, which acts as an interface between the IO bus and the processor-memory bus. The interfacing function involves translating the signals and protocols of one bus to those of the other, akin to the function served by the exit ramps that are used to connect the high-speed highway road systems to slow-speed local road systems.

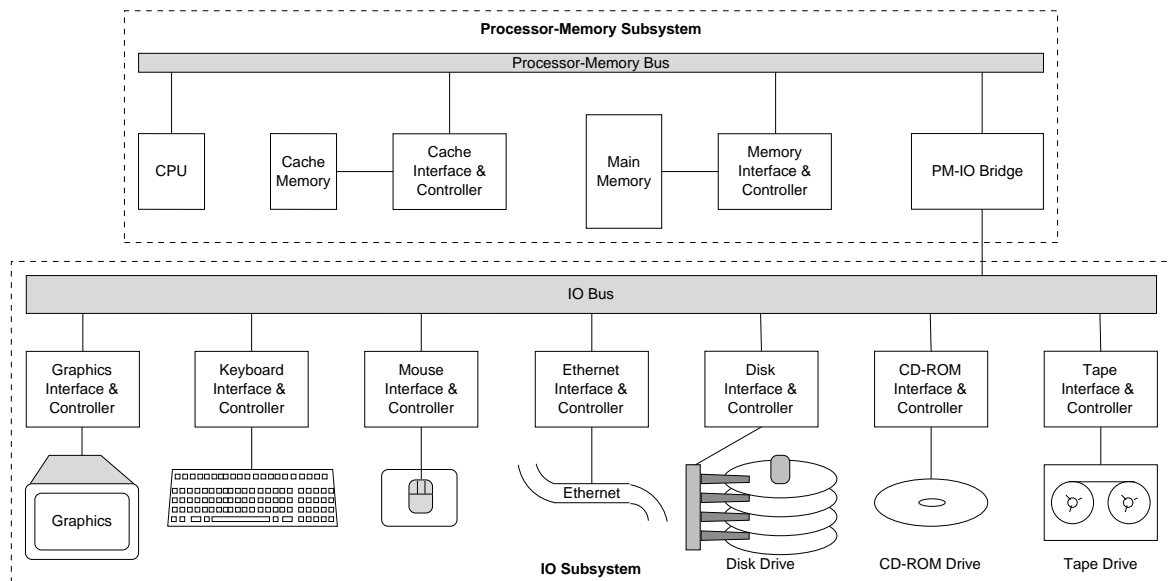


Figure 8.23: Use of an IO Bus to Interconnect IO Interfaces in the IO System

Large computers and modern desktops extend the bus hierarchy further by using multiple buses within the IO subsystem, with a bus bridge serving as an interface whenever two buses are connected together. These buses differ primarily in speed. Depending on the speed requirements of an IO device, it is connected to the appropriate bus via a proper IO controller. Based on speed, the different buses can be loosely classified into the following categories:

- **processor-memory buses**
- **high-speed IO bus or backplane buses**
- **low-speed IO bus or peripheral buses.**

Processor-memory buses are used within the processor-memory system, and are the ones closest to the processor. They are generally short, high speed, and matched to the memory system so as to maximize processor-memory bandwidth. We have already seen processor-memory buses in detail in the previous chapter. We shall look into the other two types of buses in this chapter.

Despite the prevalence of standardization among computer buses, no standardization exists for the terminology used to characterize computer buses! What some mean by a “local bus” is entirely different from what others mean by it. The classification given above is loosely related to other types of classification, such as:

- Transparent versus non-transparent
- Serial versus parallel
- Local (internal) versus expansion (external)

#### 8.4.2.1 High-Speed and Low-Speed IO Buses

The buses used within the IO system can be high-speed or low-speed. Unlike processor-memory buses, these buses can be lengthy, and can be used to hook up controllers having different data bandwidth requirements. Among the two categories of IO buses, the backplane buses are closer (physically as well as logically) to the processor, and serve as the main expressway that handles a large volume of high-speed IO traffic. Aside from the graphics interfaces of game PCs, the processor talks to most of the IO interfaces through the backplane bus, albeit the transfer may eventually go through other buses also. High-speed device interfaces such as video cards and network controllers are most often connected directly to the backplane bus. Backplane buses are generally slower than processor-memory buses, although the difference in speed has been reducing lately. The name *backplane bus* originated in the days when IO interface circuit boards used to be housed in a card cage having a *plane* of connectors at the *back*. Corresponding pins of the backplane connectors were wired together to form a bus, and each IO interface board was plugged into a connector. Examples of standard backplane buses are AGP bus, PCI bus, PCIe, and InfiniBand bus.

Peripheral buses are usually slower than backplane buses, and are therefore not directly connected to the processor-memory system. Instead, they are connected to a backplane bus, and are therefore farther from the processor than the backplane bus. They are typically used to connect controllers that connect to secondary storage devices such as disk drives, CD-ROM drives, and tape drives.

#### 8.4.2.2 Transparent and Non-transparent Buses

When a separate IO bus is used as in Figure 8.23, the new bus can be logically configured in one of two ways: (i) as a logical extension of the processor-memory bus (*transparent bus*), or (ii) logically disjoint to the processor-memory bus (*non-transparent*). In the former case, the IO controllers connected to the IO bus logically appear to the processor as if they are connected directly to the processor-memory bus. The processor can read their IO registers or write to them by executing IO read/write instructions. The bus bridge is

logically transparent to the processor; all it does is to translate the signals and protocols of one bus to those of the other. Examples of expansion buses are ISA, PCI, AGP, and PCI Express. Thus, the PCI bridge is designed in such a manner that IO interfaces connected to the PCI bus logically appear to the processor as if they are connected directly to the processor-memory bus.

In the latter case, called *non-transparent bus*, the processor cannot directly access the registers in the IO controllers connected to the IO bus. Instead, a *bus adapter* (also known as *bus controller*) acts as an IO controller whose registers can be directly accessed by the processor. Logically speaking, the address space seen by the IO controllers connected to such a bus is different from the IO address space seen by the processor (**address domain isolation**). When the processor wants to communicate with an IO controller connected to such a bus, it sends the appropriate command (along with proper IO controller identification) to the bus adapter, which then performs the required address domain translation and interacts with the IO controller by accessing its registers. A non-transparent bus is rarely used as a backplane bus.

Examples of standard non-transparent buses are SCSI, USB, and Firewire. For device drivers to access these buses, they need to be connected to the computer through a bus controller/adaptor such as SCSI controller, USB host, and FireWire host. A PCI bus can also be set up to operate as a non-transparent bus by connecting it to the backplane PCI bus using a non-transparent PCI-to-PCI bridge. Non-transparent bus bridges (or *embedded bridges*) are quite useful when an IO device requires a large address space. They are also useful when configuring intelligent IO subsystems such as RAID controllers, because the entire subsystem appears to the processor as a single virtual PCI device (only a single device driver is needed for the subsystem). They also facilitate the interconnection of multiple processors in communications and embedded PCI applications.

#### 8.4.2.3 Serial and Parallel Buses

One of the important factors affecting the cost of a bus, especially those that are not short, is the number of wires in the bus cable. A serial bus transfers data one bit at a time. Although this impacts the bandwidth, it permits thin cables to be used. Serial buses were initially used within the IO system for hooking up the controllers of low-bandwidth devices such as keyboard, mouse, and audio. Those serial buses were somewhat slow, and served as peripheral buses that connected to a backplane bus. A few standards were also developed for such type of serial buses. Examples are Serial Peripheral Interface Bus (SPI) and I2C.

As bus speeds continued to increase, it became possible to design serial buses that offer higher bandwidth. Serial buses such as USB, Firewire, and SATA have become very popular now. Because a serial bus requires only a single signal wire to transmit the data, it is able to save space in the connector as well as in the cable. This means that many more ports can be physically located at the computer system unit's periphery, and that topology can be more sophisticated than the simple linear topology. USB was designed to allow peripherals to

be connected without the need to plug expansion cards into the computer's ISA, EISA, or PCI bus, and to improve plug-and-play capabilities by allowing devices to be hot-swapped (connected or disconnected without powering down or rebooting the computer). Moreover, unlike parallel buses, serial buses are not confined to a small physical area. Also, it is often the case that serial buses can be clocked considerably faster than parallel buses, because designers do not need to worry about clock skew as well as crosstalk between multiple wires, and better isolation from surroundings achievable due to thinner cables.

Parallel buses provide multiple wires for transmitting addresses and data. Therefore, bits of address or data move through the bus simultaneously rather than one at a time. Expansion slots and ports of parallel buses have a large number of pins and are therefore big. This means that only a small number of expansion slots and ports can be incorporated on a motherboard or backplane.

#### 8.4.2.4 Local and Expansion Buses (Internal and External Buses)

A bus is generally called a local bus (also called internal bus or **host bus**) if it is physically “close” to the processor, and has no expansion slots. Its electrical and functional properties are suited for relatively short wires and relatively simple device controllers with a fast response time. As such, it is usually on the motherboard<sup>8</sup>, and is used for connecting the processor to the memory, off-chip cache memory, or other high-speed structures. These buses tend to be proprietary. Some standard buses such as VLB and PCI may also be used as local buses, especially in small systems.

Larger computer systems often have the need to connect to more “distant” devices such as peripheral devices and other computers. An **expansion bus** (or **external bus**) connects the motherboard to such devices. To this end, it provides expansion slots, thereby allowing users to “expand” the computer. The expansion slots permit plug-in expansion cards to be directly connected to the expansion bus. Standard buses that are commonly used as expansion buses are ISA, PCI, AGP, USB, and PCIe. An expansion bus can be used as a backplane bus or a peripheral bus. Historically, the ISA bus was the predominant expansion bus. Then, the PCI bus became very popular, with the ISA bus being provided for legacy interfaces.

#### 8.4.2.5 Data Transfer Types

A bus may support multiple types of data transfer.

**Control Transfer:** Control transfers are typically used by the processor for sending short, simple commands to IO devices. They are also used by the processor for receiving status

---

<sup>8</sup>In the embedded systems arena, an entire system is often integrated into a single chip (**System on Chip – SoC**) or a single package (**System in Package – SiP**), in which case the local bus is contained within a chip package.

responses from the devices.

**Interrupt Transfer:** Interrupt transfers are used by devices that transfer very little data and need guaranteed quick responses (bounded latency). Examples are keyboards and pointing devices such as mice.

**Bulk Transfer:** Bulk transfers involve large sporadic transfers such as file transfers. Devices such as printers that receive data in one big packet use the bulk transfer mode. These transfers generally do not have guarantees on bandwidth or latency.

**Isochronous Transfer:** Isochronous transfers are *guaranteed* to occur at a predetermined speed. This speed may be less than the required speed, in which case data loss can occur. Streaming devices such as speakers that deal with realtime audio and video generally use the isochronous transfer mode.

#### 8.4.2.6 Bus Characteristics

We have seen some of the features of computer buses. Some other characteristics relevant to buses are:

- Clock speed
- Bus width
- Bandwidth
- Number of slots
- Address spaces supported (memory, IO, configuration, etc.)
- Arbitration
- Communication medium and distance (copper cable, fiber cable, backplane, IC, wireless, etc.)
- Physical topology (backbone, ring, tree, star, etc.)
- Signalling method (single-ended, differential, etc.)
- Plug and Play (PnP) and hot plug features: Plug and play is the ability to automatically discover the configuration of devices attached to a bus at system boot time, without requiring reconfiguration or manual installation of device drivers. This automatic assignment of I/O addresses and interrupts to prevent conflicts and identification of drivers makes it easy to add new peripheral devices, because the user does not need to



“tell” the computer that a device has been added. The system automatically detects the device at reboot time.

Hot plug or hot swap is the ability to remove and replace devices, while the system is operating. Once the appropriate software is installed on the computer, a user can plug and unplug the component without rebooting. A well-known example of this functionality is the Universal Serial Bus (USB) that allows users to add or remove peripheral components such as a mouse, keyboard, or printer.

#### 8.4.2.7 Expansion Slots and Hardware Ports

An IO bus is a medium for devices to communicate. For an IO bus to be useful, it must provide the ability to hook up IO controllers. These controllers are hooked up only at specific places in the bus where a hardware socket is located. We can classify these hardware sockets into two kinds, depending on where they are physically located: internal (*expansion slots*) and external (*hardware ports*).

Expansion slots are located inside the system unit, and serve as receptacles for directly plugging in IO interface cards (or **expansion cards**). They are usually situated on the motherboard, as illustrated in Figure 8.24. The figure also shows a sound card being inserted into one of the expansion slots. Expansion slots are named after the bus to which they are connected; their size and shape are also based on the bus. Some of the common expansion slots are SATA slot, IDE or PATA slot, AGP (for graphics cards), PCI slot, and PCIe slot.

*Hardware ports* are specialized outlets situated on the front or back panel of the computer system unit. These serve as receptacles for plugs or cables that connect to external devices. The ports themselves are mounted on the motherboard or the expansion cards in such a way they show through openings in the system unit panels. Looking at the sound card in Figure 8.24, we can see several hardware ports. When this card is inserted into an expansion slot, these ports will become visible at the back side of the system unit. Hardware ports are generally not made of male connectors, since the protruding pins of a male connector can break easily. Table 8.6 lists some of the commonly found hardware ports, the typical cards that house them, and their typical uses.

The connectors for the above ports cover a wide variety of shapes such as round (e.g., PS/2), rectangular (e.g., FireWire), square (e.g., RJ11), and trapezoidal (e.g., D-Sub). They also cover a variety of colors such as orange, purple, or gray (e.g., PS/2 port for keyboard), green (e.g., PS/2 port for mouse, TSR port for speaker), pink (e.g., TSR port for microphone), amber (serial DB-25 or DB-9), and blue or magenta (e.g., parallel DB-25). While many of the above ports have their own specific connectors, others may use different types of connectors. The RS232 port, for instance, may use a wide variety of connectors such as DB-25, DE-9, RJ45, and RJ50.

| Port Name                  | Housing Controller      | Typical Use                                    |
|----------------------------|-------------------------|--|
| PS/2                       |                         | Keyboard, Mouse                                |
| Serial (tty, COM1:, COM2:) | RS-232                  | Terminal, Printer                              |
| Parallel                   | LPT1:, LPT2:            | Printer  |
| VGA                        | Video card              | VGA monitor                                    |
| DVI                        | Video card              | LCD monitor                                    |
| Ethernet                   | Network interface card  | LAN, ISDN                                      |
| USB                        | USB controller          | Flash drive, Mouse, Camera                     |
| FireWire                   | FireWire controller     | computer network, Camcorder                    |
| MIDI or “game”             | Sound card              | Joystick, other game controllers               |
| HDMI                       | Motherboard, video card | Uncompressed HDTV                              |
| Mini audio                 |                         | Speaker  |
| RCA                        |                         | Microphone, speaker, musical keyboard, video c |

Table 8.6: Commonly Found Hardware Ports

### 8.4.3 Standard Buses and Interconnects

We saw that computers use different types of buses, having different characteristics. An IO interface that is tailored to connect to a particular bus cannot, in general, be connected to a different bus, because of differences in bus characteristics. Two important considerations in designing an IO system, besides performance, are low cost and the ability to interchange IO interfaces across different computer platforms. In order to satisfy these two criteria, it is important to standardize the buses used in the IO systems of different computers, and make them independent of the ISA (instruction set architecture)<sup>9</sup>. If the computer community defines a standard for IO buses, then different vendors can make IO interfaces suitable for that standard bus, increasing the availability of IO interfaces and reducing their cost. These motivations have propelled the development and use of standard buses in computers designed in the last 2-3 decades, even if the computers implement different ISAs and are manufactured by different vendors. This has led to an enormous degree of commonality among the IO systems of different computers.

Although a uniform bus standard may be desirable, in practice, it is difficult to develop a standard bus that covers all peripherals (because of the extremely wide range of transfer speeds and other requirements). Therefore, over the years, several standard buses have emerged. For each standard, several revisions—supporting higher data rates, lower supply voltages, etc.—have also appeared. Some of the notable bus standards are ISA (Industry Standard Architecture), IDE (Integrated Device Electronics), PCI (Peripheral Component

---

<sup>9</sup>The processor-memory bus tends to be *proprietary*. It is difficult to define a standard for this bus, because its signals and protocols are closely tied to the ISA implemented by the computer, and its speed to the processor speed.

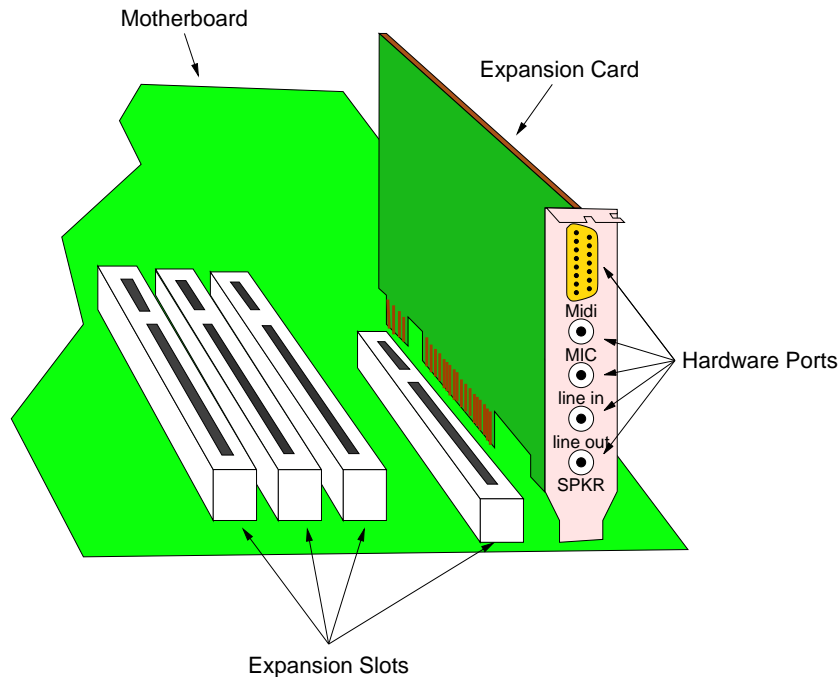


Figure 8.24: Expansion Slots and Hardware Ports

Interconnect), SCSI (Small Computer System Interface), and USB (Universal Serial Bus). Some of these were specifically developed by the industry as standards whereas the others became standards through their popularity. By the year 2000, two dominant standards had emerged in the desktop personal computer market. These are the PCI and SCSI standards. The IBM-compatible and Macintosh platforms being developed today have invariably adopted PCI as the backplane bus and SCSI as the peripheral bus. A larger fraction of workstation vendors are also adhering to these standards. Although systems with older buses (ISA or IDE) continue to ship, such systems have rapidly been replaced on all but the lowest-performance computers. We shall take a detailed look at the PCI, SCSI, and USB standards, which are very popular today in the desktop computing world.

#### 8.4.3.1 Peripheral Component Interconnect (PCI) Bus

This bus standard was developed by Intel in the early 1990s, and became a widely adopted standard for PCs, thanks to Intel's decision to put all of the related patents into the public domain (which allows other companies to build PCI-based peripherals without paying royalties). The PCI bus is typically used as a computer's backplane bus for connecting many high-bandwidth devices and other buses. Structurally, it is almost always set up as

| Bus                   | Internal/<br>External | Clock<br>speed | Data<br>width      | Data<br>Rate | Max<br>devices | Topology           | Application               |
|-----------------------|-----------------------|----------------|--------------------|--------------|----------------|--------------------|---------------------------|
| ISA                   | Internal              | 8 MHz          | 16 bits            |              |                | Backplane          | Specialized industrial us |
| ATA-7                 | External              | 66.6 MHz       |                    | 133 MB/s     |                |                    |                           |
| SCSI-1                | External              | 5 MHz          | 8 bits             | 5 MB/s       | 7              | 50 pin bus         | Mass storage devices      |
| Fast<br>Wide<br>SCSI  | External              | 10 MHz         | 16 bits            | 20 MB/s      | 15             | 68 pin bus         | Mass storage devices      |
| Wide<br>Ultra<br>SCSI | External              |                | 16 bits            | 40 MB/s      | 15             |                    | Mass storage devices      |
| Ultra640<br>SCSI      | External              | 160 MHz        | 16 bits            | 640 MB/s     | 15             |                    | Mass storage devices      |
| SATA/300              | Internal              | 3 GHz          | Serial             | 300 MB/s     |                | 4 pin cable        | Hard drives, CD/DVD c     |
| SAS-2                 |                       | GHz            | Serial             | 6 Gb/s       |                | Tree               | Hard drives, CD/DVD c     |
| PCI 2.2               | Internal              | 66 MHz         | 64 bits            | 533 MB/s     | 5              | Backplane          |                           |
| PCIe 2.0              | Internal              |                | Serial<br>32 lanes | 16 GB/s      | 1              | Point-to<br>-point | Video card                |
| AGP 8x                | Internal              | 533 MHz        | 32 bits            | 2.1 GB/s     | 1              |                    | Video card                |
| USB 2.0               | External              |                | Serial             | 60 MB/s      | 127            | Tree               | Ubiquitous                |
| Firewire 800          | External              |                | Serial             | 800 Mb/s     | 63             | Tree               | Video devices             |

Table 8.7: Characteristics of Standard Computer Buses

an *expansion bus*; that is, expansion slots are attached to the bus. Logically, it is typically set up as a transparent bus. That is, the PCI bridge is designed in such a manner that IO interfaces connected to the PCI bus logically appear to the processor as if they are connected directly to the processor-memory bus. The PCI bus also supports autoconfiguration; thus, it supports three independent address spaces: memory, IO, and configuration. The PCI bus provides good support for multimedia applications by providing high bandwidth, long burst transfers, good support for multiple bus masters, and guaranteed low-latency access. A 32-bit PCI bus operating at 33 MHz supports a peak transfer rate of  $33 \text{ MHz} \times 4 \text{ bytes} = 132 \text{ MB}$  per second. The PCI bus is multiplexed, which means that some signal lines are used for multiple purposes, depending on the operation performed or the step of a particular operation. Although this makes it slightly difficult to design PCI-compatible IO interfaces, these interfaces require only fewer pins, which make them less costly.

The PCI bus is common in today's PCs and other computer types. It is being succeeded by PCI Express (PCI-E), which offers much higher bandwidth. As of 2007 the PCI standard is still used by many legacy and new devices that do not require the higher bandwidth of PCI-E. PCI is expected to be the primary expansion bus in desktops for a few more years.

### 8.4.3.2 Small Computer System Interface (SCSI) Bus

The original SCSI standard (now known as SCSI-1) was defined by the American National Standards Institute (ANSI), under the designation X3.131. The SCSI bus is usually used as a peripheral bus for connecting storage devices such as disk drives, CD-ROM drives, and tape drives. It is also used to connect input-output devices such as scanners and printers. As a peripheral bus, it is slower than the backplane bus, and is connected to the computer through the backplane bus. Unlike the PCI bus, the SCSI bus is not usually set up as a transparent bus. IO interfaces connected to the SCSI bus are therefore not directly manipulated by the device driver's IO instructions. Each SCSI-1 bus can support up to 8 controllers, allowing up to 8 peripherals to be connected.

**Controllers and Devices:** IO controllers are connected to a SCSI bus via cables. The SCSI bus itself is connected to the backplane bus via a *SCSI controller* (also called a *host adapter*), coordinates between all of the device interfaces on the SCSI bus and the computer. The SCSI controller can be a card that is plugged into an available backplane bus slot or it can be built into the motherboard. The controller itself is like a microcontroller in that it has a small ROM or Flash memory chip that stores the SCSI BIOS — the software needed to access and control the devices on the bus. It has the ability to perform DMA operations. The controller can be directly accessed by the device driver running on the CPU. On getting commands from the device driver, the SCSI controller accesses the controller of the appropriate IO device, and performs the required IO operations. The controller can also deliver electrical power to SCSI-enabled devices. The SCSI bus expects its device interfaces to have functionality such as buffering of data and ensuring of data integrity. Each device connected to a SCSI bus, including the SCSI controller, must have a unique identifier (ID) in order for it to work properly. This ID is specified through a hardware or software setting. If the bus can support sixteen devices, for instance, the IDs range from zero to 15, with the SCSI controller typically having the highest ID.

**Cables and Connectors:** Internal devices are connected to a SCSI controller by a ribbon cable. External devices are attached to the controller in a daisy chain using a thick, round cable. In a daisy chain, each device connects to the next one in line. For this reason, external SCSI devices typically have two SCSI connectors — one to connect to the previous device in the chain, and the other to connect to the next device. Different SCSI standards use different connectors, which are often incompatible with one another. These connectors usually use 50, 68 or 80 pins. Serial Attached SCSI (SAS) devices use (smaller) SATA cables and SATA-compatible connectors.

**SCSI Types:** Since the introduction of the first SCSI standard, many upgrades have been introduced, resulting in a proliferation of SCSI standards. They differ in terms of speed, bus width, connectors, and maximum number of devices that can be attached. Examples for

these improved SCSI standards are SCSI-2, Fast SCSI, Ultra SCSI, and Wide SCSI. All of these SCSI buses are parallel. The newest type of SCSI, called *Serial Attached SCSI (SAS)*, however, uses SCSI commands but transmits data serially. SAS uses a point-to-point serial connection to transfer data at 3 gigabits per second, and each SAS port can support up to 128 devices or expanders.

The SCSI bus has several benefits. It is reasonably fast, with transfer rates up to 640 megabytes per second. It has been around for more than 20 years and has been thoroughly tested; so it has a reputation for being reliable. However, the SCSI standards also have some potential problems. They have limited system BIOS support, and have to be configured for each computer. There is also no common SCSI software interface. Finally, the different SCSI types are incompatible, differing in speed, bus width, and connector. The SCSI bus is slowly being replaced by other buses such as serial-ATA (SATA).

#### 8.4.3.3 Universal Serial Bus (USB)

**Overview:** The USB is a popular interconnect for connecting external devices to a computer. Just about any computer that you buy today comes with one or more USB ports, identified by the USB icon shown in Figure 8.25. These USB ports let you attach everything from mice to printers to your computer quickly and easily. This standard was developed by a collaborative effort of several companies.



Figure 8.25: USB Icon

Universal Serial Bus (USB) provides a serial bus standard for connecting IO devices to computers (although it is also becoming commonplace on video game consoles such as Sony's PlayStation 2, Microsoft's Xbox 360, Nintendo's Revolution, and PDAs, and even devices like televisions and home stereo equipment). The USB's low cost led to its adoption as the standard for connecting most peripherals that do not require a high-speed bus.

Because the USB port is serial, it is small. The USB standard allows a thin (serial) cable to connect an IO controller module to a USB port, and this makes it possible to move the IO controller from the systems unit to the IO device itself. In fact, the USB was designed to allow peripherals to be connected without the need to plug expansion cards into the computer's ISA, EISA, or PCI bus, and to improve plug-and-play capabilities by allowing devices to be hot swapped.

USB can connect peripherals such as mice, keyboards, gamepads and joysticks, scanners, digital cameras, printers, external storage, networking components, etc. For many devices such as scanners and digital cameras, USB has become the standard connection method.

USB is also used extensively to connect non-networked printers, replacing the parallel ports which were widely used; USB simplifies connecting several printers to one computer. As of 2005, the only large classes of peripherals that cannot use USB are displays and high-quality digital video components (because they need a higher bandwidth than that supported by USB).

**Cables, Connectors, and Topology:** The USB's connectivity is different from that of a regular bus. It uses a tree-type topology, with a hub at each node of the tree. Each hub has a few connectors for connecting USB-compatible IO interfaces and other USB hubs, subject to a limit of 5 levels of branching. The hub at the root of the tree is called the *root hub*. It contains a USB host controller, whereas the remaining hubs contain a USB repeater. The root hub is hooked to the backplane bus of the computer. Physically, it may be implemented on a USB adapter card that is plugged onto an expansion slot, or it may be directly implemented on the motherboard. Device drivers can directly access only the host controller, which contains the hardware registers that map to the computer's IO address space. Not more than 127 devices, including the bus devices, may be connected to a single host controller. Modern computers often have several host controllers, allowing a very large number of USB devices to be connected. USB endpoints actually reside on the connected device: the channels to the host is referred to as a *pipe*. The devices (and hubs) have associated pipes (logical channels) which are connections from the host controller to a logical entity on the device named an *endpoint*. Individual USB cables can run as long as 5m, and have 4 wires: two are for the power lines (4.5 V and Ground) and the other two form a twisted pair for data. USB cables do not need to be terminated. USB 2.0 uses bursts unlike firewire.

**Power supply:** The USB also has wires for supplying electrical power from the computer to the connected devices. Devices such as mice that require only a small amount of power can thus obtain power from the bus itself, without requiring an external power source. The USB connector provides a single nominally 5 volt wire from which connected USB devices may power themselves. A given segment of the bus is specified to deliver up to 500 mA. Devices that need more than 500 mA must provide their own power. Many hubs include external power supplies which will power devices connected through them without taking power from up-stream.

**Enumeration:** When the host controller powers up, it queries all of the devices connected down-stream. For each device, it assigns a unique 7-bit address and loads the device driver it needs. This process is called enumeration. Devices are also enumerated when they are connected to the bus while the computer is working; recall that USB devices are hot-swappable. At the time of enumeration, the host also finds out from each device the type of data transfer it will perform in the future. The host also keeps track of the total bandwidth requested by all of the isochronous and interrupt devices. They can consume up to 90

percent of the available bandwidth. After 90 percent is used up, the host denies access to any other isochronous or interrupt devices. Control packets and bulk transfer packets use any bandwidth left over (at least 10 percent).

**IO Transaction:** An IO request reaching the root hub from the backplane bus is propagated to all nodes in the tree. All IO interfaces connected to the tree nodes will see the request, just like in a regular bus, although the request may travel through one or more hubs. However, a transaction from an IO interface to the processor-memory system is propagated only towards the root hub, and is not seen by other IO interfaces connected to the tree. This function is logically different from that of a bus.

**USB 2.0 (High-speed USB):** The standard for USB version 2.0 was released in April 2000 and serves as an upgrade for USB 1.1. USB 2.0 provides additional bandwidth for multimedia and storage applications and has a data transmission speed 40 times faster than USB 1.1. To allow a smooth transition for both consumers and manufacturers, USB 2.0 has full forward and backward compatibility with original USB devices and can work with cables and connectors made for the original USB. By supporting the following three speed transfer speeds, USB 2.0 supports low-bandwidth devices such as keyboards and mice, as well as high-bandwidth ones like high-resolution Webcams, scanners, printers and high-capacity storage systems.

- A Low Speed rate of 1.5 Mbit/s that is mostly used for Human Interface Devices (HID) such as keyboards, mice and joysticks.
- A Full Speed rate of 12 Mbit/s that was the fastest rate before the USB 2.0 specification. All USB Hubs support Full Speed.
- A Hi-Speed rate of 480 Mbit/s.

**Storage Devices:** USB implements connections to storage devices using a set of standards called the USB mass-storage device class. This was initially intended for traditional magnetic and optical drives, but has been extended to support a wide variety of devices. As USB makes it possible to install and remove devices without opening the computer case, it is especially useful for external drives. Today, a number of manufacturers offer external, portable USB hard drives that offer performance comparable to internal drives. These external drives usually contain a translating device that interfaces a drive of conventional technology (IDE, ATA, SATA, ATAPI, or even SCSI) to a USB port. Functionally, the drive appears to the user just like another internal drive.



#### 8.4.3.4 FireWire (IEEE 1394)

IEEE 1394 is a serial bus interface standard for high-speed communications and isochronous real-time data transfer. It is more commonly known as *FireWire*, the name given by Apple Inc., its creator (Sony Corp. calls it *i.LINK*).

While the USB was designed for low-to-medium speed peripherals, the FireWire was designed for interfacing with high-speed devices such as digital camcorders and disk drives. It has many similarities with the USB, such as plug-and-play, hot swap, provision of power through the cable, and ability to connect many devices. Apart from speed differences, the big difference between FireWire and USB 2.0 is that USB 2.0 is host-based, meaning that devices must connect to a computer in order to communicate. FireWire, on the other hand, is peer-to-peer, meaning that two FireWire devices can talk to each other without going through the processor and memory subsystems. Implementing FireWire costs a little more than USB.

The main features of FireWire are:

- \* Fast transfer of data
- \* Ability to put lots of devices on the bus
- \* Provision of power through the cable
- \* Plug-and-play: Unlike USB devices, each FireWire node participates in the configuration process without intervention from the host system.
- \* Hot-pluggable ability
- \* Low cabling cost

FireWire is a method of transferring information between digital devices, especially audio and video equipment. FireWire is fast – the latest version achieves speeds up to 800 Mbps. At some time in the future, that number is expected to jump to 3.2 Gbps when optical fiber is introduced.

Up to 63 devices can be connected to a FireWire bus.

When the computer powers up, it queries all of the devices connected to the FireWire bus and assigns each one an address, a process called enumeration. Each time a new device is added to or removed from the bus, the FireWire bus is re-enumerated. FireWire is plug-and-play as well as hot pluggable; so if you connect a new FireWire device to your computer, the operating system auto-detects it and starts talking to it.

FireWire 800 is capable of transfer rates up to 800 Mbps, and permits the cable length to be up to 100 meters. The faster 1394b standard is backward-compatible with 1394a.

**Cables and Connectors:** FireWire devices can be powered or unpowered. FireWire allows devices to draw their power from their connection. Two power conductors in the cable can supply power (8 to 30 volts, 1.5 amps maximum) from the computer to an unpowered device. Two twisted pair sets carry the data in a FireWire 400 cable using a 6-pin configuration. Some smaller FireWire-enabled devices use 4-pin connectors to save space, omitting the two pins used to supply power. The maximum cable length is 4.5m. The length between nodes can be increased by adding repeaters.

**Sending Data via FireWire:** FireWire uses 64-bit fixed addressing, based on the IEEE 1212 standard. There are three parts to each packet of information sent by a device over FireWire:

- \* A 10-bit bus ID that is used to determine which FireWire bus the data came from \*
- A 6-bit physical ID that identifies which device on the bus sent the data \*
- A 48-bit storage area that is capable of addressing 256 terabytes of information for each node

The bus ID and physical ID together comprise the 16-bit node ID, which allows for 64,000 nodes on a system. Data can be sent through up to 16 hops (device to device). Hops occur when devices are daisy-chained together.

**Digital Video:** Now that we've seen how FireWire works, let's take a closer look at one of its most popular applications: streaming digital video. FireWire really shines when it comes to digital video applications. Most digital video cameras or camcorders now have a FireWire plug. When you attach a camcorder to a computer using FireWire, the connection is amazing.

An important element of FireWire is the support of isochronous devices. In isochronous mode, data is streamed between the device and the host in real-time with guaranteed bandwidth and no error correction. Essentially, this means that a device like a digital camcorder can request that the host computer allocate enough bandwidth for the camcorder to send uncompressed video in real-time to the computer. When the computer-to-camera FireWire connection enters isochronous mode, the camera can send the video in a steady flow to the computer without anything disrupting the process.

#### 8.4.3.5 Ethernet

**Topology:** At the most fundamental level, all Ethernet networks are laid out as a bus topology, with the devices tapping into the bus.

**Packet Assembly:** To transmit a message across an Ethernet, a node constructs an Ethernet *frame*, a package of data and control information that travels as a unit across the network. Large messages are split across multiple frames.

The arbitration method followed for the Ethernet bus network is different from those used for the buses inside the computer. Instead of using a centralized bus arbitration unit, it takes a distributed approach. Before a device places a frame in the bus, it has to make sure that the bus is not in use, i.e., no other frames are already present on the bus. The device hardware follows the following protocol: it monitors the bus and waits until no frame is present in the bus. Once it finds no frame to be present, it waits further for a short period of time and checks again. If the bus is found to be idle, then the device begins transmitting the frame.

With the above arrangement, it is possible that two or more devices may simultaneously find the bus to be idle and start transmitting their frames at exactly the same time. In Ethernet parlance, this is termed a *collision*. When the Ethernet NIC detects a collision, it waits for a *random* amount of time, re-checks to see if the bus is idle and then attempts a re-transmission.

The original Ethernet described communication over a single cable shared by all devices on the network. Once a device attached to this cable, it had the ability to communicate with any other attached device. This allows the network to expand to accommodate new devices without requiring any modification to those devices already on the network.

One interesting thing about Ethernet addressing is the implementation of a broadcast address. A frame with a destination address equal to the broadcast address (simply called a broadcast, for short) is intended for every node on the network, and every node will both receive and process this type of frame.

#### 8.4.3.6 Bluetooth

Bluetooth is an industrial specification for wireless personal area networks (PANs). The standard also includes support for more powerful, longer-range devices suitable for constructing wireless LANs. Bluetooth provides a way to connect and exchange information between devices like personal digital assistants (PDAs), mobile phones, laptops, PCs, printers and digital cameras via a secure, low-cost, globally available short range radio frequency. It is a radio standard primarily designed for low power consumption, with a short range (power class dependent: 10 centimeters, 10 meters, 100 meters) and with a low-cost transceiver microchip in each device.

Bluetooth lets these devices talk to each other when they come in range, even if they are not in the same room, as long as they are within up to 100 meters of each other, depending on the power class of the product.

##### Communication and Connection

A Bluetooth device playing the role of the *master* can communicate with up to 7 devices playing the role of the *slave*. This network of up to 8 devices is called a *piconet*. At any given time, data can be transferred between the master and one or more slaves; but the role of the master switches rapidly among the devices in a round-robin fashion. (Simultaneous transmission from the master to multiple slaves, although possible, is not very common). Either device may switch the master/slave role at any time.

Bluetooth specification also allows connecting multiple piconets together to form a *scat-ternet*, with some devices acting as a bridge by simultaneously playing the master role in one piconet and the slave role in another piconet.

Any Bluetooth device will transmit the following sets of information on demand

\* Device Name \* Device Class \* List of services \* Technical information eg: device features, manufacturer, Bluetooth specification, clock offset

Any device may perform an "inquiry" to find other devices to which to connect, and any device can be configured to respond to such inquiries. However, if the device trying to connect knows the address of the device it will always respond to direct connection requests and will transmit the information shown in the list above if requested for it. Use of the device's services however may require pairing or its owner to accept but the connection itself can be started by any device and be held until it goes out of range. Some devices can only be connected to one device at a time and connecting to them will prevent them from connecting to other devices and showing up in inquiries until they disconnect the other device.

Every device has a unique 48-bit address. However these addresses are generally not shown in inquiries and instead friendly "Bluetooth names" are used which can be set by the user, and will appear when another user scans for devices and in lists of paired devices. Most phones have the Bluetooth name set to the manufacturer and model of the phone by default. Most phones and laptops will only show the Bluetooth names and special programs are required to get additional information about remote devices. This can get confusing with activities such as Bluejacking as there could be several phones in range named "T610" for example. On Nokia phones the Bluetooth address may be found by entering "#2820#". On computers running Linux the address and class of a USB Bluetooth dongle may be found by entering "hciconfig hci0 class" as root ("hci0" may need to be replaced by another device name).

Every device also has a 24-bit class identifier. This provides information on what kind of a device it is (Phone, Smartphone, Computer, Headset, etc), which will also be transmitted when other devices perform an inquiry. On some phones this information is translated into a little icon displayed beside the device's name.

Bluetooth devices will also transmit a list of services if requested by another device; this also includes some extra information such as the name of the service and what channel it is on. These channels are virtual and have nothing to do with the frequency of the transmission, much like TCP ports. A device can therefore have multiple identical services.

#### Pairing

Pairs of devices may establish a trusted relationship by learning (by user input) a shared secret known as a *passkey*. A device that wants to communicate only with a trusted device can cryptographically authenticate the identity of the other device. Trusted devices may also encrypt the data that they exchange over the air so that no one can listen in. The encryption can however be turned off and passkeys are stored on the device's file system and not the Bluetooth chip itself. Since the Bluetooth address is permanent a pairing will be preserved even if the Bluetooth name is changed. Pairs can be deleted at any time by either device. Devices will generally require pairing or will prompt the owner before it allows a remote device to use any or most of its services. Some devices such as Sony Ericsson phones will usually accept OBEX business cards and notes without any pairing or prompts. Certain printers and access points will allow any device to use its services by default much like unsecured Wi-Fi networks.

### Air interface

The Bluetooth protocol operates in the license-free ISM band at 2.45 GHz. In order to avoid interfering with other protocols that use this band, the Bluetooth protocol divides the band into 79 channels (each 1 MHz wide) and changes channels up to 1600 times per second. Implementations with versions 1.1 and 1.2 reach speeds of 723.1 kbit/s. Version 2.0 implementations feature Bluetooth Enhanced Data Rate (EDR), and thus reach 2.1 Mbit/s. Technically version 2.0 devices have a higher power consumption, but the three times faster rate reduces the transmission times, effectively reducing consumption to half that of 1.x devices (assuming equal traffic load).

Bluetooth differs from Wi-Fi in that the latter provides higher throughput and covers greater distances but requires more expensive hardware and higher power consumption. They use the same frequency range, but employ different multiplexing schemes. While Bluetooth is a cable replacement for a variety of applications, Wi-Fi is a cable replacement only for local area network access. A glib summary is that Bluetooth is wireless USB whereas Wi-Fi is wireless Ethernet, both operating at much lower bandwidth than the cable systems they are trying to replace.

### Applications:

- Wireless networking between desktops and laptops, or desktops in a confined space and where little bandwidth is required
- Bluetooth peripherals such as printers, mice and keyboards
- Bluetooth cell phones, which are able to connect to other cell phones, computers, personal digital assistants (PDAs), and automobile handsfree systems.
- Bluetooth mp3 players and digital cameras to transfer files to and from computers
- Bluetooth headsets for mobile phones and smartphones
- \* Medical applications Advanced Medical Electronics Corporation is working on several devices
- \* For remote controls where infrared was traditionally used.
- \* Hearing aids Starkey Laboratories have created a device to plug into some hearing aids [2]
- \* Newer model Zoll Defibrillators for the purpose of transmitting Defibrillation Data and Patient Monitoring/ECG data between the unit and a reporting PC using Zoll Rescue Net software.

### Specifications and Features

The Bluetooth specification was first developed by Ericsson

#### 8.4.4 Expansion Bus and Expansion Slots

A **bus** is a set of electronic signal pathways that allows information and signals to travel between components inside or outside of a computer.

An **expansion slot (connector)**: Remember that the expansion bus, or external bus, is made up of the electronic pathways that connect the different external devices to the rest of your computer. These external devices (monitor, telephone line, printer, etc.) connect to ports on the back of the computer. Those ports are actually part of a small circuit board or 'card' that fits into a connector on your motherboard inside the case. The connector is called an expansion slot.

Note: Communication ports (com ports), printer ports, hard drive and floppy connectors, etc., are all devices which used to be installed via adapter cards. These connectors are now integrated onto the motherboard, but they are still accessed via the expansion (external) bus and are allocated the same type of resources as required by expansion cards. As a matter of fact (and unfortunately, in my opinion), other devices like modems, video technology, network and sound cards are now being integrated, or embedded, right onto the motherboard.

Expansion slots are easy to recognize on the motherboard. They make up a row of long plastic connectors at the back of your computer with tiny copper 'finger slots' in a narrow channel that grab the metal fingers or connectors on the expansion cards. In other words, the expansion cards plug into them. The slots attach to tiny copper pathways on the motherboard (the expansion bus), which allows the device to communicate with the rest of the computer. Each pathway has a specific function. Some may provide voltages needed by the new device (+5, +12 and ground), and some will transmit data. Other pathways allow the device to be addressed through a set of I/O (input/output) addresses, so that the rest of the computer knows where to send and retrieve information. Still more pathways are needed to provide clock signals for synchronization and other functions like interrupt requests, DMA channels and bus mastering capability.

As with any other part of the computer, technology has evolved in an effort to increase the speed, capability and performance of expansion slots. Now you'll hear about more busses - PCI bus, ISA bus, VESA bus, etc. Not to worry! These are all just types of expansion (external) busses. They just describe the type of connector and the particular technology or architecture being used. Thus, the adapter card being installed must match the architecture or type of slot that it's going into. An ISA card fits into an ISA slot, a PCI adapter card must be installed into a PCI expansion slot, etc.

Traditionally, PCs have utilized an expansion bus called the ISA bus. In recent years, however, the ISA bus has become a bottleneck, so nearly all new PCs have a PCI bus for performance as well as an ISA bus for backward compatibility.

The current popular expansion bus is the PCI (Peripheral Component Interconnect) bus for all cards except the graphics cards. For graphics cards, the bus of choice is AGP. Most motherboards today have one AGP slot and several PCI slots. Your expansion cards will plug into these card slots. Be sure you get cards that match the available type of slots on your motherboard. Other popular computer buses include NuBus for the Apple Macintosh, VESA local bus, and PCMCIA (PC Card).

### 8.4.5 IO System in Modern Desktops

The IO system is one area where there is a significant difference across different computers. This is despite the use of standard buses and standard ports, which isolate any effects of the kernel mode ISA as well as processor characteristics. The significant differences in IO system organization arise from 3 factors:

- *Differences in the number and types of buses used:* First, systems often differ in the number of backplane buses, serial buses, and peripheral buses used. Second, although standard buses have been proposed, there are just far too many standards available for each type of bus. For instance, for the backplane bus, the widely used standard is the PCI. However, because the ISA bus was popular until recently as a backplane bus, many systems still support an ISA bus that is connected to the PCI bus using a ISA bus controller. On a different note, even within a standard, there are variations such as SCSI, SCSI-2, and SCSI-3.
- *The manner in which the buses are interconnected:* Even if two computer systems used exactly the same buses for their IO system, they can differ in the manner in which these buses are interconnected. Depending on the system requirements, system architects connect the buses in different manners.
- *Differences in the number and types of IO interfaces connected:* Finally, even if two computers have the same bus configuration, they may not have the same set of IO devices hooked to them. Although they may have been shipped with identical system configuration, the end user can add or remove IO devices later. Moreover, one user may connect a particular IO interface, such as a keyboard interface, to one port while another may connect it to a different port.

Figure 8.26 shows one possible organization of the IO system in a mid-range to high-end desktop machine in 2002. In this configuration, PCI is used as the backplane bus, with slower devices sharing the lower-performance SCSI bus. The PCI backplane bus is used to connect all interfaces to the processor-memory system. Several of the slow IO devices (audio IO, serial ports, and the desktop bus) share a single port onto the PCI bus. Serial ports provide for connections such as low-speed Appletalk network. The desktop bus provides support for keyboards and mice. The second interface in the figure is used for graphics output. The third interface is used for connecting to the ethernet network. The fourth interface is the SCSI controller, which is used extensively for connecting bulk storage devices such as disks, tapes, and CD-ROMs. In the figure, a disk drive, a tape drive, and a CD-ROM reader are connected to the SCSI bus by means of their respective controllers (interfaces).

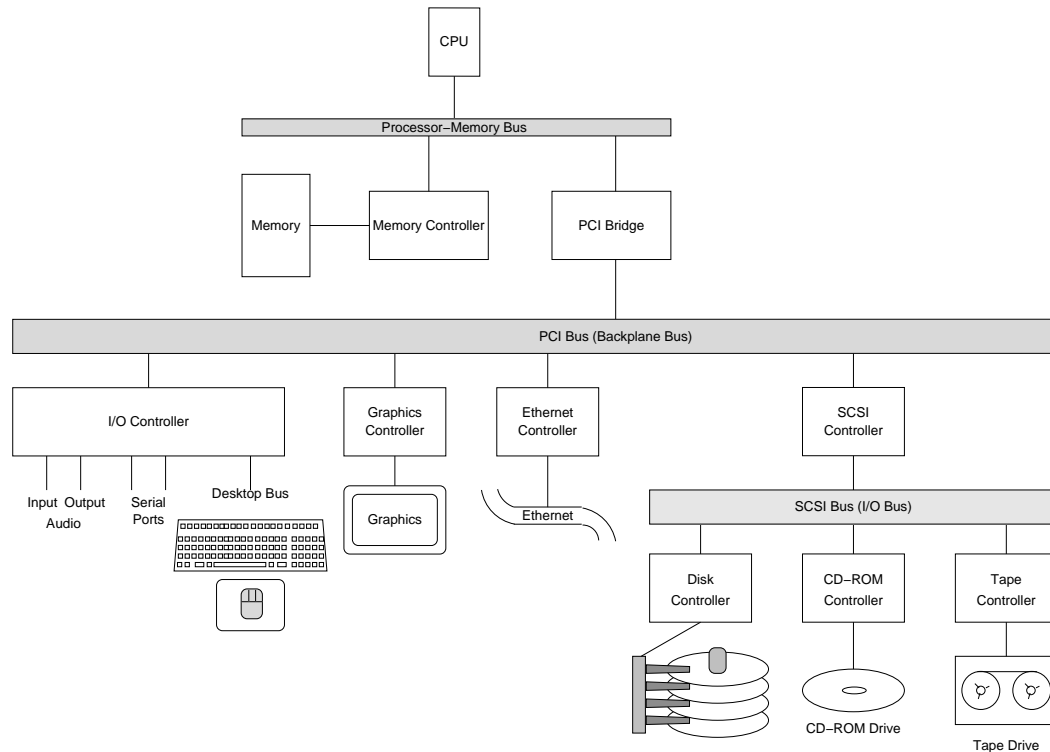


Figure 8.26: A Possible Organization of the IO System in a Modern Desktop Computer

#### 8.4.6 Circa 2006

A circa 2006 PC incorporates many specialized buses of different protocols and bandwidth capabilities, such as the Serial ATA, USB, and Firewire. The PCI bus is too slow to be the "backbone" for hooking up all of these buses. A solution adopted since the late 1990s?? is to expand the functionality of the PMI-PCI bridge and connect some devices directly to this bridge. Examples of such devices include the AGP. This central bridge itself is usually organized as two parts—the **northbridge** and the **southbridge** or **IO bridge**. The northbridge is used to connect the 3 fastest parts of the system—the CPU, the memory, and the video card. In a modern computer system, the video card's GPU (graphics processing unit) is functionally a second (or third) CPU.

The northbridge is connected to the southbridge, which routes the IO traffic between the northbridge and the rest of the IO subsystem. The southbridge provides ports for different types of buses, such as the PCI. With continued advances in bus standards and speeds, the southbridge is steadily evolving to accommodate more complex bus controllers such as Serial ATA and Firewire. So today's southbridge is sort of the Swiss Army Knife of IO



switches, and thanks to Moore's Law it has been able to keep adding functionality in the form of new interfaces that keep bandwidth-hungry devices from starving on the PCI bus.

In an ideal world, there would be one primary type of bus and one bus protocol that connects the different IO devices including the video card/GPU to the CPU and main memory. Although we are unlikely to return to this utopian ideal, PCI Express (PCIe) promises to bring some order to the current chaos. In any case, it is expected to dominate the personal computer in the coming decade. With Intel's recent launch of its 900-series chipsets and NVIDIA and ATI's announcements of PCI Express-compatible cards, PCIe will shortly begin cropping up in consumer systems. Infiniband shows even more technical promise towards returning to the ideal single bus.

### 8.4.7 RAID

#### Future Directions in IO Organization

What does the future hold for IO systems? The rapidly increasing performance of processors strains IO systems, whose mechanical components cannot offer similar improvements in performance. To reduce the growing gap between the speed of processors and the access time to secondary storage (mainly disks), operating systems often cache active parts of the file system in the physical memory. These caches are called *file caches*, and like the cache memories we saw earlier, they attempt to make use of temporal and spatial localities in access to secondary storage. The use of file caches allows many file accesses to be satisfied by physical memory rather than by disk.

Magnetic disks are increasing in capacity quickly, but access time is improving only slowly. In addition to increases in density, transfer rates have grown rapidly as disks increased in rotational speed and disk controllers improved. In addition, virtually every high-performance disk manufactured today includes a track or sector buffer that caches sectors as the read head passes over them.

Many companies are working feverishly to develop next-generation bus standards, as standards such as PCI are fast approaching the upper limits of what they can do. All of these new standards have one thing in common. They do away with the shared-bus topology used in PCI and instead use point-to-point switching connections<sup>10</sup>. By providing multiple direct links, such a bus can allow multiple data transfers to occur simultaneously.

**HyperTransport**, one of the recently proposed standards, is beginning to replace the front side bus. For each session between nodes, it provides two point-to-point links. Each link can be anywhere from 2 bits to 32 bits wide, supporting a maximum transfer rate of

---

<sup>10</sup>This trend is similar to the direct path based processor data paths we saw in Chapter 7. However, there are some differences between the direct paths used in modern processor data paths and the direct links used in modern IO buses. In the case of IO buses, a direct connection between two nodes is established only while they are communicating with each other. While these two nodes are communicating, no other node can access that path.

6.4 GB per second, operating at 2.6 GHz. By using a HyperTransport-PCI bridge, PCI devices can be connected to the HyperTransport bus.

**PCI-Express** is another point-to-point system, allowing for better performance. It is also scalable. A basic PCI-Express slot will be a 1x connection. This will provide enough bandwidth for high-speed Internet connections and other peripherals. The 1x means that there is one lane to carry data. If a component requires more bandwidth, PCI-Express 2x, 4x, 8x, and 16x slots can be built into motherboards, adding more lanes and allowing the system to carry more data through the connection. In fact, PCI-Express 16x slots are already available in place of the AGP graphics card slot on some motherboards. As prices come down and motherboards built to handle the newer cards become more common, AGP could fade into history.

Motherboards can incorporate PCI-Express connectors that attach to special cables. This could allow for completely modular computer system, much like home stereo systems. The basic unit would be a small box with the motherboard having several PCI-Express connection jacks. An external hard drive could be connected via USB 2.0 or PCI-Express. Small modules containing sound cards, video cards, and modems could also be connected as needed. Thus, instead of one large unit, the computer would be only as large as the devices connected.

## 8.5 Network Architecture

Till now we were mostly discussing *stand-alone computers*, which are not connected to any computer network. Computer networks are a natural extension of a computer's IO organization, enabling it to communicate with other computers and remote peripherals such as network printers. Unlike the IO buses which typically use electrical wires, the connections in a computer network could be via some forms of telecommunication media such as telephone wires, ethernet cables, fiber optics, or even microwaves (wireless). Based on transmission technology, we can categorize computer networks into two: *broadcast networks* and *point-to-point networks*.

Broadcast networks have a single communication channel shared by all the devices on the network. Any of these devices can transmit short messages called *packets* on the network, which are then received by all the other devices. An address field within the packet specifies the device for whom it is intended. Upon receiving a packet, a device checks this address field to see if it is intended for it.

By contrast, point-to-point networks consist of connections between individual pairs of machines.

Physical distance is an important metric for classifying computer networks, as different techniques are used at different size scales.

Most of today's desktop computers are instead connected to a network, and therefore it

is useful for us to have a brief introduction to this topic. A *computer network* is a collection of computers and other devices that communicate to share data, hardware, and software. Each device on a network is called a *node*. A network that is located within a relatively limited area such as a building or campus is called a *local area network* or *LAN*, and a network that covers a large geographical area is called a *wide area network* or *WAN*. The former is typically found in medium-sized and large businesses, educational institutions, and government offices. Different types of networks provide different services, use different technology, and require users to follow different procedures. Popular network types include Ethernet, Token Ring, ARCnet, FDDI, and ATM.

#### Give a figure here

A computer connected to a network can still use all of its *local resources*, such as hard drive, software, data files, and printer. In addition, it has access to *network resources*, which typically include *network servers* and *network printers*. Network servers can serve as a *file server*, *application server*, or both. A file server serves as a common repository for storing program files and data files that need to be accessible from multiple workstations—*client nodes*—on the network. When an individual client node sends a request to the file server, it supplies the stored information to the client node. Thus, when the user of a client workstation attempts to execute a program, the client's OS sends a request to the file server to get a copy of the executable program. Once the server sends the program, it is copied into the memory of the client workstation, and the program is executed in the client. The file server can also supply data files to clients in a similar manner. An application server, on the other hand, runs application software on request from other computers, and forwards the results to the requesting client.

### 8.5.1 Network Interface Card (NIC)

In order to connect a computer to a network, a *network interface card (NIC)* is required. This interface card sends data from the computer out over the network and collects incoming data for the computer. The NIC for a desktop computer can be plugged into one of the expansion slots in the motherboard. The NIC for a laptop computer is usually a PCMCIA card. Different types of networks require different types of NICs.

The NIC performs the tasks that are at the lowest levels in the communication protocol. The transmitter in the NIC places the actual bit stream on the communications channel. Most often, it also performs the task of appending additional fields to the bit stream, such as a preamble for timing purposes, or error check information. The receiver in the NIC receives packets addressed to the computer, and most likely strips off preamble bits and performs error checking.

### Modems

### 8.5.2 Protocol Stacks

Computer networks may be implemented using a variety of protocol stack architectures, computer buses or combinations of media and protocol layers, incorporating one or more of ..... such as ATM, Bluetooth, Ethernet, and FDDI.

Computer networks are also making great strides. Both 100 Mbit Ethernet and switched Ethernet solutions are being used in new networks and in upgrading networks that cannot handle the tremendous explosion in bandwidth created by the use of multimedia and the growing importance of the World Wide Web. ATM represents another potential technology for expanding even further. To support the growth in traffic, the Internet backbones are being switched to optical fiber, which allows a significant increase in bandwidth for long-haul networks.

## 8.6 Interpreting an IO Instruction

We have discussed at length the IO system and different ways of organizing it. It behoves us to take a look at how an IO instruction is interpreted for a microarchitecture. As we recall from Chapters 5 and 6, in both the assembly-level architecture and the ISA, all of the IO instructions in a systems program involve reading or writing IO registers. The rest of the IO functions are performed by the IO interfaces in conjunction with the IO controllers and, of course, the IO devices.

As far as the processor is concerned, the interpretation of an IO instruction is quite similar to that of a load/store instruction. In fact, for memory-mapped IO addresses, there is hardly any difference, except that IO addresses are generally unmapped (no address translation is required), and that the IO access step (the one analogous to the memory access step) may take a much longer time than a memory access step.

Let us trace through an IO access microinstruction in a MIPS microarchitecture. This microinstruction is executed after the *ALU operation* step in which the IO address is calculated. The IO request is initially transmitted over the processor-memory bus. The backplane bus bridge, which is hooked to the processor-memory bus, recognizes the address as an IO address. It performs the necessary signal conversions, and transmits the request over the backplane bus. All of the IO interfaces hooked to the backplane bus see the address. The interface that has the matching address responds to the request.

## 8.7 System-Level Design

The exact boundaries of chips and printed circuit boards (PCBs) keep changing. In the early days, the processor was built out of multiple chips. With the current very large scale integration (VLSI) technologies, the processor is invariably built as a single chip. The main memory has traditionally been built out of multiple chips. However, that is changing

now. Currently, parts of the cache memory hierarchy have migrated into the processor chip. Research efforts are on to completely integrate the processor and memory systems into a single chip.

In the domain of embedded systems, the situation can be different. An entire system is often built on a single chip (*system-on-a-chip*).

## 8.8 Concluding Remarks

## 8.9 Exercises

1. Explain why interrupts are disabled when interpreting a `syscall` instruction.
2. What are the fundamental storage components in a paging-based virtual memory system? Explain with figure(s) the basic working of such a virtual memory system.
3. A virtual memory system uses 15-bit virtual addresses. The physical memory consists of 8 Kbytes. The page size is 2 Kbytes. The TLB can hold up to 3 entries. Both the TLB and the page table are replaced using the LRU (least recently used) policy.
  - (a) Indicate using a diagram how the MMU would split a 15-bit address to get the virtual page number (VPN).
  - (b) Consider the following sequence of memory address accesses: 0x6ffc, 0x7ffc, 0x6000, 0x4000, 0x3000, 0x2000, 0x7ffc, 0x2008, 0x74fc, 0x64fc. For each of these accesses, indicate if it would be a TLB hit or miss. Also indicate if it would be a page fault or not.
  - (c) Draw the final state of the TLB and the page table.
4. Explain the advantages of USB devices over PCI devices.
5. Explain what is meant by a hierarchical bus organization in a computer system. Explain why hierarchical bus organizations are used in computer systems.

## Chapter 9

# Register Transfer Level Architecture

*Listen to counsel and receive instruction, That you may be wise in your latter days.*

**Proverbs 19: 20**

Our objective in this chapter is to study ways of implementing various microarchitecture specifications we saw in Chapters 7 and 8. Because of the complexity of hardware design, it is impractical to directly design a gate-level circuitry that implements the microarchitecture. Therefore, computer architects have taken a more structured approach by introducing one or more abstraction levels in between. In this chapter, we study the RTL implementation of the microarchitectures discussed in the last two chapters. An RTL view of a computer is restricted to seeing the major storage elements in the computer, and the transfer of information between them.

A particular microarchitecture may be implemented in different ways, with different RTL architectures. Like the design of the higher level architectures that we already saw, RTL architecture design is also replete with trade-offs. The trade-offs at this level also involve characteristics such as speed, cost, power consumption, die size, and reliability. For general-purpose computers such as desktops, the most important factors are speed and cost. For laptops and embedded systems, the important considerations are size and power consumption. For space exploration and other critical applications, reliability is of primary concern.

This chapter addresses some of the fundamental questions concerning RTL architectures, such as:

- What are the building blocks in an RTL architecture, and how are they connected together?

- What steps should the RTL architecture perform to sequence through a machine language program, and to execute (i.e., accomplish the work specified in) each machine language instruction?
- What are some simple organizational techniques to reduce the number of clock cycles taken to execute each machine language instruction?

## 9.1 Overview of RTL Architecture

The RTL architecture implements the (more abstract) microarchitecture. The building blocks at this level are at a somewhat lower level, such as multiplexers and .....

The RTL architecture follows the data path - control unit dichotomy.

RTL designers often use a language called **register transfer language**<sup>1</sup> to indicate the control actions specified by the control unit. Each major function specified at the microarchitecture level — as a MAL instruction — is expressed as a sequence of RTL instructions.

The sequence of operations performed in the data path is determined by commands generated by the control unit. The control unit thus functions as the data path's interpreter of machine language programs.

At the RTL, the computer data path consists of several building blocks and subsystems connected together based on the microarchitecture-level data path specification. The major building blocks of the microarchitectures we saw were registers, memory elements, arithmetic-logic units, other functional units, and interconnections. Assumptions about the behavior of these building blocks are used by the RTL designer.

### 9.1.1 Register File and Individual Registers

An RTL architecture incorporates all of the register files and individual registers specified in the microarchitecture. Apart from these registers, the RTL architecture may incorporate several additional registers and/or latches. These serve as temporary storage for values generated or used in the midst of instruction execution. One such register/latch, for instance, is generally used to store a copy of the ML instruction that is currently being executed. This register is generally called *instruction register* (IR). Similarly, the result of an arithmetic operation may need to be stored in a temporary hardware register before it can be routed to the appropriate destination register in the register file. The data path may also need to keep track of special properties of arithmetic and logical operations such as condition codes;

---

<sup>1</sup>Modern designs are more frequently specified using a hardware description language (HDL) such as Verilog or VHDL.

most data paths use a register called `flags`<sup>2</sup>.

### 9.1.2 ALUs and Other Functional Units

We have just looked at RTL implementations of the storage elements present in a microarchitecture. Next, we shall consider the elements that serve to perform the operations and functionalities specified in the ISA.

### 9.1.3 Register Transfer Language

By now, it must be amply clear that the data path does not do anything out of its own volition; it merely carries out the elementary instruction that it receives from the control unit. Thus, it is the control unit that decides what register value should be made available in the processor bus in a particular clock cycle. Likewise, the function to be performed by the ALU on the input values that are present at its inputs in a clock cycle is also decided by the control unit. For ease of understanding, we shall use the *register transfer notation* to express these elementary instructions sent by the control unit to the data path. This language is called a *register transfer language*. Accordingly, an elementary instruction represented in this language is called an *RTL instruction*. An RTL instruction may be composed of one or more elementary operations called *RTL operations*, performed on data stored in registers or in memory. An RTL operation can be as simple as copying data from one physical register to another, or more complex, such as adding the contents of two physical registers and storing the result in a third physical register. In RTL, data transfer operations and arithmetic/logical operations are specified using a notation called **register transfer notation (RTN)**. In this notation, a data transfer is designated in symbolic form by means of the replacement operator ( $\rightarrow$ ). An example RTL operation is

$$\text{PC} \rightarrow \text{MAR}$$

The semantics of this RTL instruction is quite straightforward: copy the contents of register `PC` to register `MAR`. By definition, the contents of the source register do not change as a result of the transfer. The RTL operation does not specify *how* the microarchitecture should do this copying; it merely specifies *what* action the microarchitecture needs to do. We shall deal with the specifics of the data transfer when we look at logic-level architectures in Chapter 9.

---

<sup>2</sup>In some architectures such as the IA-32, the `flags` register is visible at the ISA level (usually as a set of condition codes). In MIPS-I, it is not visible at the ISA level.



## 9.2 Example RTL Data Path for Executing MIPS-0 ML Programs

Designing a computer data path is better caught than taught. Accordingly, in this chapter we will design several example data paths and illustrate the principles involved. For simplicity and ease of understanding, we restrict ourselves to implementing the microarchitecture-level data paths that we studied in Chapters 7 and 8 for the MIPS-0 ISA. We will start with the simple data paths, and later move on to the more complex data paths. This simple data path will be used in the discussions of the control unit as well.

In order to design an RTL data path, we consider each block in the microarchitecture, and design it as a collection of combinational logic subcircuits and registers/latches. Thus, we may add some more non-architected registers: **IR** to store a copy of the current ML instruction (bit pattern) being executed, and **flags** to store important properties of the result produced by the ALU. The output of the **flags** register is connected to the control unit (not shown in figure), which can access its bits individually.

We shall augment the multi-function ALU (Arithmetic and Logic Unit) with an ALU Input Register (AIR) and an ALU Output Register (AOR) to temporarily store one of the input values and the output value, respectively.

Figure 9.1 pictorially shows one possible way of interconnecting the combinational subcircuits and the registers/latches so as to perform the required functions. This figure suppresses information on how the control unit controls the functioning of the subcircuits; these details will be discussed later. In addition to the processor bus, some dedicated paths have been provided to perform data transfers that are inefficient to be mapped to the bus. One such path connects the **offset** field of **IR** to the **sign extend** unit.

The RTL register **IR** and its outputs have special properties that warrant further discussion. The data path uses this register to store the bit pattern of the instruction being interpreted and executed. Unlike other registers, the outputs from **IR** are organized as *fields*, in line with the different fields specified in the MIPS instruction formats. These outputs are depicted in Figure 9.2. The 6-bit **opcode** and **func** field outputs are supplied to the **Opcode Decoder (OD)**, which produces a binary pattern that uniquely identifies the opcode of the fetched instruction. This binary pattern is supplied to the processor control unit (not shown in Figure 9.1). The 5-bit **rs**, **rt**, and **rd** outputs are used to select the register that is to be read or written into. The register read addresses can be **rs** or **rt**, whereas the register write address can be **rt**, **rd**, or 31. (Recall that when a subroutine call instruction is executed, the return address is written into R31.) The 16-bit **offset** output includes the least significant 16 bits of **IR**, and is supplied to a **sign extend** unit to convert it into a 32-bit signed integer. This 32-bit output of the **sign extend** unit is connected to the **processor bus** directly. This connection is provided for the purpose of executing the memory-referencing instructions, the register-immediate ALU type instructions, and the branch instructions, all of which specify a 16-bit **immediate** operand.

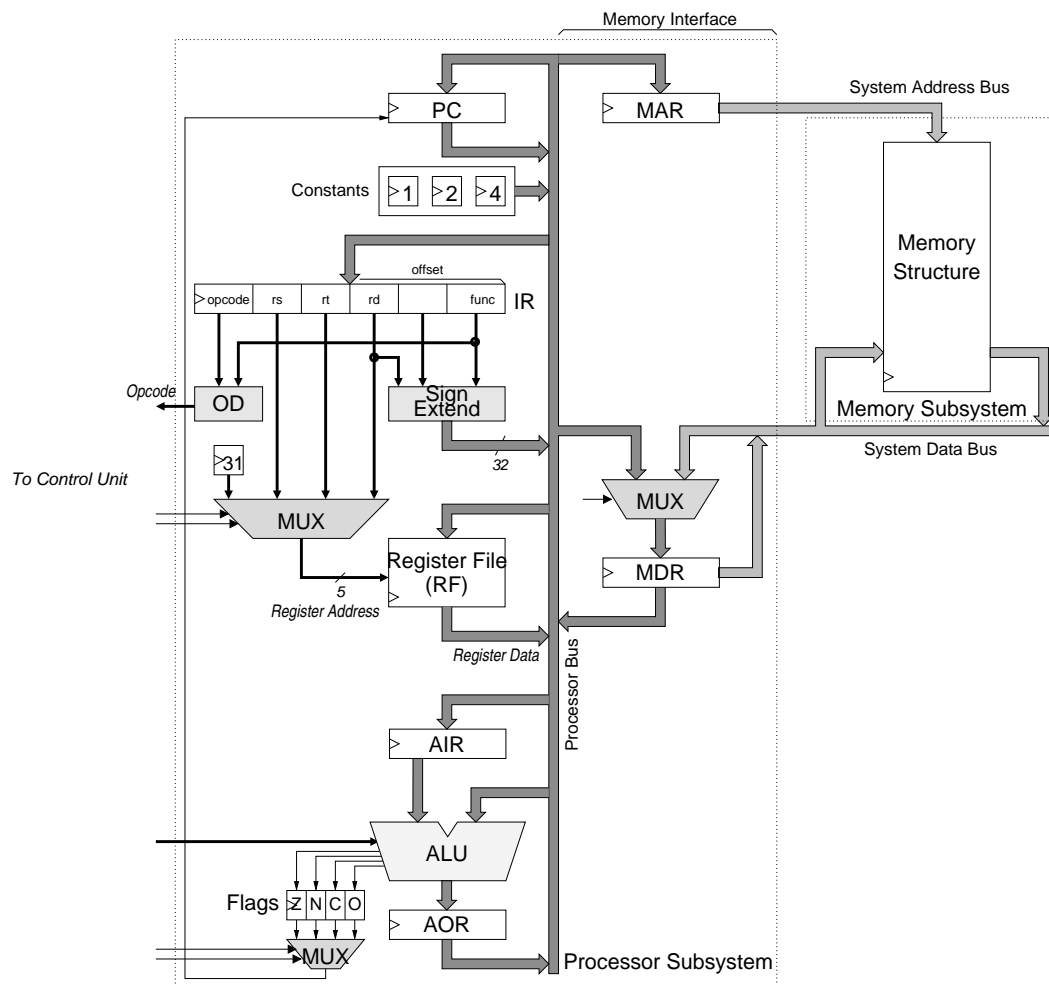


Figure 9.1: An RTL Data Path for Implementing the MIPS-0 User Mode ISA

The ALU has two 32-bit data inputs, one of which is always taken from microarchitectural register **AIR** via a direct path. The other input is taken from the processor bus. The output of the ALU is fed to microarchitectural register **AOR** through a direct path. We need register **AOR** because the ALU output cannot be routed to a storage location in the same step via the processor bus (though which one of the ALU data inputs arrives in the same step). Apart from the normal ALU output, other outputs such as zero, negative, and carry are routed to a **flags** register via direct paths.

Finally, the data path includes the memory subsystem, which implements the memory address space defined in the user mode ISA. For interfacing the processor bus to the memory subsystem, this data path uses two special microarchitectural registers, called *memory*

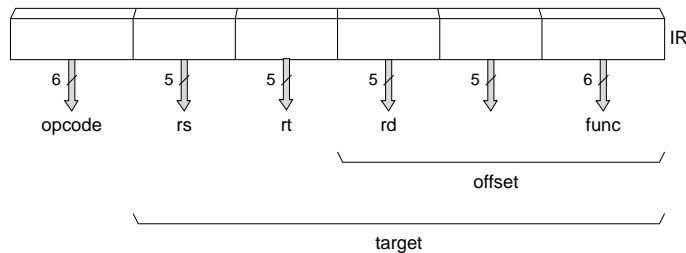


Figure 9.2: Outputs of Microarchitectural Register IR, which holds a copy of the ML Instruction

*address register* (MAR) and *memory data register* (MDR). The former is used to store the address of the memory location to be read from or written to, and the latter is used to store the data that is read or to be written. The address stored in MAR is made available to the address inputs of the memory subsystem through the **system address bus** during a memory read/write operation. When a memory read operation is performed, MDR is updated from the **system data bus**. Similarly, when a memory write operation is performed, the contents of MDR are transmitted to the appropriate memory through the **system data bus**.

### 9.2.1 RTL Instruction Set

The process of executing a machine language (ML) instruction on the data path can be broken down into a sequence of more elementary steps. We saw in Section 7.1 how we can use a register transfer language to express these elementary steps. We can define a set of RTL operations for the MIPS-0 data path defined in Figure 9.1. An RTL operation can specify the flow of data between two or more storage locations only if there is a connection between them. Thus

$$\text{MAR} \rightarrow \text{AOR}$$

cannot be a valid RTL operation in this data path, because there is no direct connection between MAR and AOR. Similarly,

$$\text{MDR} \rightarrow \text{ALU}$$

can not be a valid RTL operation, because ALU is combinational logic, and not a storage device.

Normally, all bits of a register are involved in a transfer. However, if a subset of the bits is to be transferred, then the specific bits are identified by the use of pointed brackets. The RTL operation

$$\text{IR} \langle 15:0 \rangle \rightarrow \text{MDR}$$

specifies that bits 15 to 0 of IR are copied to MDR. Similarly, memory locations or general-purpose registers (GPRs) are specified with square brackets. The RTL operation

$$R[\text{rs}] \rightarrow \text{MDR}$$

indicates that the contents of the GPR whose address is present in the **rs** field (of **IR**) are transferred to **MDR**. The **rs** field specifies a particular GPR. Similarly, the RTL operation

$$\text{MDR} \rightarrow M[\text{MAR}]$$

indicates that the contents of **MDR** are transferred to the memory location whose address is present in **MAR**.

If the interconnections of the data path are rich enough to allow multiple RTL operations in the same time period, these can be denoted by writing them in the same line, as follows:

$$\text{AIR} + 4 \rightarrow \text{AOR}; \quad M[\text{MAR}] \rightarrow \text{MDR}$$

specifies that in the same step, the value in **AIR** is incremented by 4 and written to **AOR**, and the contents of the memory location addressed by **MAR** are copied to **MDR**.

Finally, for RTL operations that are conditional in nature, an “if” construct patterned after the C language’s “if” construct is defined.

$$\text{if (Z)} \quad \text{AOR} \rightarrow \text{PC}$$

indicates that if the zero flag is equal to 1, the contents of **AOR** is copied to **PC**; otherwise no action is taken.

### 9.2.2 RTL Operation Types

The actions performed by RTL operations can be classified into three different types:

1. **Data transfer:** These RTL operations copy the contents of one register/memory location to another. An example data transfer RTL operation is  $\text{PC} \rightarrow \text{MAR}$ .
2. **Arithmetic/Logic:** These RTL operations perform arithmetic or logical operations on data stored in registers. Data transfers can only occur along the interconnections provided in the data path, from one register/memory location to another. An example arithmetic RTL operation is  $\text{AIR} + \text{PC} \rightarrow \text{AOR}$ .
3. **Conditional:** These RTL operations perform a function such as a data transfer if and only if the specified condition is satisfied. An example conditional RTL operation is  $\text{if (Z)} \text{ AOR} \rightarrow \text{PC}$ .

Table 9.1 gives a list of useful RTL operations for the microarchitecture of Figure 9.1. A given RTL operation may specify actions of more than one type. For example, the RTL operation  $\text{if (Z)} \text{ AOR} \rightarrow \text{PC}$  is a conditional as well as a data transfer RTL operation. The RTL operations done (in parallel) in a single step form an *RTL instruction*.

| No.                          | RTL Operation<br>for Data Path | Comments                            |
|------------------------------|--------------------------------|-------------------------------------|
| <i>Data Transfer Type</i>    |                                |                                     |
| 0                            | PC → MAR                       | Read from memory<br>Write to memory |
| 1                            | PC → MDR                       |                                     |
| 2                            | PC → AIR                       |                                     |
| 3                            | PC → R[31]                     |                                     |
| 4                            | M[MAR] → MDR                   |                                     |
| 5                            | MDR → M[MAR]                   |                                     |
| 6                            | MDR → IR                       |                                     |
| 7                            | MDR → AIR                      |                                     |
| 8                            | MDR → R[rt]                    |                                     |
| 9                            | MDR → PC                       |                                     |
| 10                           | R[rs] → AIR                    |                                     |
| 11                           | R[rs] → PC                     |                                     |
| 12                           | R[rt] → AIR                    |                                     |
| 13                           | R[rt] → MDR                    |                                     |
| 14                           | SE(offset) → AIR               |                                     |
| 15                           | AOR → R[rd]                    |                                     |
| 16                           | AOR → R[rt]                    |                                     |
| 17                           | AOR → MAR                      |                                     |
| <i>Arithmetic/Logic Type</i> |                                |                                     |
| 18                           | AIR <i>op</i> R[rt] → AOR      |                                     |
| 19                           | AIR <i>op const</i> → AOR      |                                     |
| 20                           | AIR <i>op</i> SE(offset) → AOR |                                     |
| 21                           | AIR <i>op</i> R[rt] → Z        |                                     |
| <i>Conditional Type</i>      |                                |                                     |
| 22                           | if (Z) AOR → PC                |                                     |

Table 9.1: A List of Useful RTL Operations for the Data Path of Figure 9.1

### 9.2.3 An Example RTL Routine

Now that we are familiar with the syntax of RTL as well as the useful RTL operations for the data path of Figure 9.1, we shall look at a simple sequence of RTL operations called an RTL routine. Let us write an RTL routine that adds the contents of registers specified in the **rs** and **rt** fields of IR, and writes the result into the register specified in the **rd** field of IR. The astute reader may have noticed that executing this RTL routine amounts to executing the MIPS-I machine language instruction **and rd, rs, rt** in the data path of Figure 9.1, provided the binary pattern of this instruction is languishing in IR.

In theory, we can write entire programs in RTL that can perform tasks such as ‘printing

| Step | RTL Instruction     | Comments  |
|------|---------------------|---|
| 0    | R[rs] → AIR         | Copy value of register <b>rs</b> to AIR         |
| 1    | AIR AND R[rt] → AOR | AND this value with value in register <b>rt</b> |
| 2    | AOR → R[rd]         | Write result into register <b>rd</b>            |

Table 9.2: An Example RTL Routine for the Data Path of Figure 9.1

“hello, world!” ’ and more; we will, of course, need a special storage for storing the RTL programs and a mechanism for sequencing through the RTL program. Having seen the difficulty of writing programs in assembly language and machine language, one can easily imagine the nightmare of writing entire programs in RTL! However, developing an RTL program may not be as bad as it sounds, given that we can develop translator software such as assemblers that take machine language programs and translate them to RTL programs. The real difficulty is that RTL programs will be substantially bigger than the corresponding ML programs, thereby requiring very large amounts of storage. This is where *interpretation* comes in. By generating the required RTL routines on-the-fly at run-time, the storage requirements are drastically reduced, as we will see next.

### 9.3 Interpreting ML Programs by RTL Routines

Having discussed the basics of defining RTL routines, our next step is to investigate how machine language programs can be interpreted by a sequence of RTL instructions that are defined for a particular data path. This section discusses how RTL instructions can be put together as an **RTL routine** to carry out this interpretation for each instruction defined in the ISA. An RTL routine is allowed to freely use and modify any of the ISA-invisible microarchitectural registers. The ISA-visible registers, however, can be modified only as per the semantics of the ML instruction being interpreted.

#### 9.3.1 Interpreting the Fetch and PC Update Commands for Each Instruction

When implementing ISAs supporting fixed length instructions, the actions required to perform the fetch command are the same for all instructions. The MIPS-0 ISA is no exception. We shall consider this phase of instruction execution first. In the data path of Figure 9.1, register PC keeps the memory address of the next ML instruction to be fetched. To fetch this instruction from memory, we have to utilize the memory interface provided in the data path. This interface consists of two registers, MAR and MDR, that store the memory address and data, respectively. Table 9.3 gives an RTL routine that implements the fetch phase of executing the instruction. This routine also includes the RTL instructions for updating PC after the fetch phase so as to point to the next ML instruction to be executed.

| Step                      | RTL Instruction           | Comments |
|---------------------------|---------------------------|----------|
| <i>Fetch and Decode</i>   |                           |          |
| 0                         | PC $\rightarrow$ MAR      |          |
| 1                         | M[MAR] $\rightarrow$ MDR  |          |
| 2                         | MDR $\rightarrow$ IR      |          |
| 3                         |                           |          |
| <i>Decode instr</i>       |                           |          |
| <i>PC increment</i>       |                           |          |
| 4                         | PC $\rightarrow$ ATR      |          |
| 5                         | ATR + 4 $\rightarrow$ AOR |          |
| 6                         | AOR $\rightarrow$ PC      |          |
| <i>Goto execute phase</i> |                           |          |

Table 9.3: An RTL Routine for Fetching a MIPS-0 ML Instruction in the Data Path of Figure 9.1

The first step involves copying the contents of PC to MAR so that this address can be supplied through the **system address bus** to the memory system. Thus, at the end of step 0, MAR has the address of the memory location that contains the instruction bit pattern. Step 1 specifies a memory read operation, during which the contents of the memory location specified through the **system address bus** by MAR are read from the memory. The instruction bit pattern so read is placed on the **system data bus**, from where it is loaded into memory interface register MDR in the processor data path. Thus, at the end of step 1, the instruction bit pattern is present in MDR. In this RTL routine, we consider the time taken to perform this step as one clock cycle, although the exact number of cycles taken depends on the specifics of the memory system used. In step 2, the instruction bit pattern is copied from MDR into microarchitectural register IR.

Once the instruction bit pattern is copied into IR, the instruction decoding circuitry (not shown in figure) decodes the bit pattern. The exact manner in which instruction decoding is performed is not specified here; we have earmarked a separate step for carrying out the decode operation. The **opcode** and **funct** fields of IR uniquely identify the instruction. Steps 0-3 thus constitute the RTL routine for the fetch phase of instruction interpretation. Naturally, this portion is the same for every instruction in the MIPS ISA because all instructions have the same size. Had the MIPS ISA used variable length instructions, this fetch process would have to be repeated as many times as the number of words in the instruction.

After executing an ML instruction, the system needs to go back to step 0 so as to execute the next instruction in the ML program. However, prior to that, it has to increment PC to point to the next instruction; otherwise the same ML instruction gets executed repeatedly. In this RTL routine, this update of PC is done immediately after completing the instruction fetch. Thus, in steps 4-6, PC is incremented by 4 to point to the next instruction in the executed machine language program. After step 6, the system goes to the *execute phase*.

This RTL routine takes 7 clock cycles to complete. We can, in fact, perform the PC increment function in parallel to the instruction fetch function, and reduce the total number of clock cycles required for the two activities. Table 9.4 provides the modified RTL routine, which requires only 4 clock cycles. In this routine, in steps 0 and 1, the updated value of PC is calculated in parallel with the transfer of the instruction bit pattern from the main memory to MDR. It is important to note that multiple RTL operations can be done in parallel, only if they do not share the same bus or destination register. In general, the more the connectivity provided in a data path, the more the opportunities for performing multiple RTL operations in parallel.

| Step                                   | RTL Instruction                                     | Comments   |
|--|---|--|
| <i>Fetch, Decode, and PC increment</i> |   |  |
| 0                                      | PC $\rightarrow$ MAR; PC $\rightarrow$ AIR          | Read instr                                       |
| 1                                      | M[MAR] $\rightarrow$ MDR; AIR + 4 $\rightarrow$ AOR |  |
| 2                                      | MDR $\rightarrow$ IR                                | <i>Decode instr</i><br><i>Goto execute phase</i> |
| 3                                      | AOR $\rightarrow$ PC                                |  |

Table 9.4: An Optimized RTL Routine for Fetching a MIPS-0 ML Instruction in the Data Path of Figure 9.1

### 9.3.2 Interpreting Arithmetic/Logical Instructions

We just saw an RTL routine for fetching an instruction. Next, let us consider the *execute phase* of ML instructions. Unlike the fetch phase, this phase is different for different ML instructions. Therefore, the RTL routines for the execute phase will be different for the different instructions. We shall consider one instruction each from the four types of ML instructions: (i) data transfer instruction, (ii) arithmetic/logical instruction, (iii) control flow changing instruction, and (iv) syscall instruction.

Let us start by considering an arithmetic instruction. We shall put together a sequence of RTL instructions to interpret an arithmetic/logical instruction.

Example: Consider the MIPS ADDU instruction whose symbolic representation is `addu rd, rs, rt`. Its encoding is given below.

|        |    |    |    |  |     |
|--------|----|----|----|--|-----|
| 000000 | rs | rt | rd |  | ADD |
|--------|----|----|----|--|-----|

The fields `rs`, `rt`, and `rd` specify register numbers; the first two of these contain the data values to be added together as unsigned intergers. The `rd` field indicates the destination register, i.e., the register to which the result should be written to, as long as it is not



register \$0. In this data path, the addition of the two register values can be done in the ALU. However, there is only a single bus to route the two register values as well as the result of the addition operation. Therefore, we will have to do the routing of data values in multiple steps and use the temporary registers AIR and AOR.

Table 9.5 specifies a sequence of RTL instructions for carrying out the execute phase of this instruction in the data path of Figure 9.1. Let us go through the working of this RTL routine. We name the first RTL instruction of the routine as *step 4*, as the execute phase is a continuation of the fetch phase, which ended at step 3.

| Step                 | RTL Instruction     | Comments                     |
|----------------------|---------------------|------------------------------|
| <i>Execute phase</i> |                     |                              |
| 4                    | R[rs] → AIR         |                              |
| 5                    | AIR + R[rt] → AOR   | Perform arithmetic operation |
| 6                    | if (rd) AOR → R[rd] | Write result                 |

Table 9.5: An RTL Routine for the Execute Phase of the Interpretation of the MIPS-0 ISA Instruction Represented Symbolically as `addu rd, rs, rt`. This RTL Routine is for executing the ML instruction in the Data Path of Figure 9.1

Step 4 begins the execution phase. First, the register operands must be fetched from the register file. In step 4, the value in the `rs` field of IR is used as an address to read the general-purpose register numbered `rs` into microarchitectural register AIR. In step 5, the contents of general-purpose register numbered `rt` are read and supplied to the ALU, which adds it to the contents of AIR, and stores the result in the microarchitectural register AOR. In step 6, the contents of AOR are transferred to the general-purpose register numbered `rd`, if `rd` is non-zero. By performing this sequence of RTL instructions in the correct order, the ML instruction `addu rd, rs, rt` is correctly interpreted.

### 9.3.3 Interpreting Memory-Referencing Instructions

Let us next put together a sequence of RTL instructions to fetch and execute a memory-referencing machine language instruction. Because all MIPS instructions are of the same length, the RTL routine for the fetch part of the instruction is the same as before; the differences are only in the execution part. Consider the MIPS load instruction whose symbolic representation is `lw rt, offset(rs)`. The semantics of this instruction are to copy to GPR `rt` the contents of memory location whose address is given by the sum of the contents of GPR `rs` and sign-extended `offset`. We need to come up with a sequence of RTL instructions that effectively fetch and execute this instruction in the data path of Figure 9.1.

|        |    |    |        |
|--------|----|----|--------|
| 100011 | rs | rt | offset |
|--------|----|----|--------|

The interpretation of a memory-referencing instruction requires the computation of an address. For the MIPS ISA, address calculation involves sign-extending the `offset` field of the instruction to form a 32-bit signed offset, and adding it to the contents of the register specified in the `rs` field of the instruction. In this data path, the address calculation is done using the same ALU, as no separate adder has been provided. With this introduction, let us look at the RTL routine given in Table 9.6 to interpret this `lw` instruction.

| Step                 | RTL Instruction        | Comments               |
|----------------------|------------------------|------------------------|
| <i>Execute phase</i> |                        |                        |
| 4                    | R[rs] → AIR            | Compute memory address |
| 5                    | AIR + SE(offset) → AOR |                        |
| 6                    | AOR → MAR              |                        |
| 7                    | M[MAR] → MDR           | Read from memory       |
| 8                    | if (rt) MDR → R[rt]    | Write loaded value     |

Table 9.6: An RTL Routine for the Execute Phase of the Interpretation of the MIPS-0 ISA Instruction Represented Symbolically as `lw rt, offset(rs)`. This RTL Routine is for executing the ML instruction in the Data Path of Figure 9.1

In step 4, the contents of register specified in the `rs` field is copied to `AIR`. In step 5, the sign-extended `offset` value is added to this value to obtain the memory address. This memory address is stored in `AOR`, and is copied to `MAR` in the next step. Recall that for a memory transfer to take place, the address of the memory location must be present in `MAR`. In step 7, the actual memory read is performed and the value obtained is stored in `MDR`; this step is the same as step 2 of the fetch routine. Finally, in step 8, the loaded value is copied to the register specified in the `rt` field of `IR`.

### 9.3.4 Interpreting Control-Changing Instructions

The instructions that we interpreted so far — `addu` and `lw` — do not involve control flow changes that cause deviations from straightline sequencing in the ML program. Next let us see how we can interpret control-changing instructions, which involve modifying `PC`, usually based on a condition.

*Example:* Consider the MIPS-0 conditional branch instruction whose symbolic representation is `beq rs, rt, offset`. The semantics of this instruction state that if the contents of GPRs `rs` and `rt` are equal, then the value  $\text{offset} \times 4 + 4$  should be added to `PC` so as to cause a control flow change<sup>3</sup>; otherwise, `PC` is incremented by 4 as usual. The encoding of this instruction is given below:

---

<sup>3</sup>The actual MIPS ISA uses a *delayed branch* scheme; i.e., the control flow change happens only after executing the ML instruction that follows the branch instruction in the program. We avoid delayed branches to keep the discussion simple.

|        |    |    |        |
|--------|----|----|--------|
| 000100 | rs | rt | offset |
|--------|----|----|--------|

| Step                 | RTL Instruction                      | Comments                            |
|----------------------|--------------------------------------|-------------------------------------|
| <i>Execute phase</i> |                                      |                                     |
| 4                    | PC $\rightarrow$ AIR                 |                                     |
| 5                    | SE(offset) $\ll$ 2 $\rightarrow$ AOR | Multiply offset by 4                |
| 6                    | AIR + AOR $\rightarrow$ AOR          | Calculate branch target address     |
| 7                    | R[rs] $\rightarrow$ AIR              |                                     |
| 8                    | R[rt] == AIR $\rightarrow$ Z         | Evaluate branch condition           |
| 9                    | if (Z) AOR $\rightarrow$ PC          | Update PC if condition is satisfied |

Table 9.7: An RTL Routine for the Execute Phase of the Interpretation of the MIPS-0 ISA Instruction Represented Symbolically as `beq rs, rt, offset`. This RTL Routine is for executing the ML instruction in the Data Path of Figure 9.1

Table 9.7 presents an RTL routine to execute this instruction in the data path given in Figure 9.1. The execute routine has 6 steps numbered 4-9. The first part of the routine (steps 4-6) calculates the target address of the branch instruction. In step 4, the incremented PC value is copied to AIR. In the next step, the sign-extended `offset` value is multiplied by 4 by shifting it left by 2 bit positions. In the next step, it is added to the copy of PC value present in AIR to obtain the target address of the `beq` instruction. In step 8, the contents of register `rt` are compared against those of AIR (which were copied from register `rs`), and the result of the comparison is stored in flag Z. That is, Z is set to 1 if they are the same, and reset to 0 otherwise. In the last step, PC is updated with the calculated target value present in AOR, if flag Z is 1. Thus, if flag Z is not set, then PC retains the incremented value it obtained in step 3, which is the address of the instruction following the branch in the machine language program being interpreted.

### 9.3.5 Interpreting Trap Instructions

We have seen the execute phase of the interpretation for all of the instruction types except the `syscall` instructions. Let us next look at how the data path performs this interpretation, which is somewhat different from the previously seen ones. Like the control-changing instructions, `syscall` instructions also involve modifying PC. For the MIPS-I ISA, the `pc` is updated to `0x80000080`. In addition, the machine is placed in the kernel mode. We shall take a detailed look at the corresponding RTL sequence in Section 8.1.1, along with other kernel mode implementation issues.

## 9.4 RTL Control Unit: An Interpreter for ML Programs

We have seen how individual machine language instructions can be interpreted using a sequence of RTL instructions for execution in a data path. Stated differently, execution of an RTL routine in the data path is tantamount to executing the corresponding ML instruction. And, the execution of many such RTL routines is tantamount to executing an entire ML program. In other words, when the data path executes an *RTL program* — a sequence of RTL routines — it indirectly executes an ML program.

Who generates the RTL routine for each ML instruction? Where is the routine stored after it is generated? These are questions we attempt to answer in this section. Generation of the RTL instructions, in the proper sequence, is the function of the control unit. The control unit thus serves as the *interpreter* of machine language programs. In other words, it is the unit that converts ML programs to equivalent RTL programs. The algorithm for doing this conversion is hardwired into the control unit hardware.

We can think of two different granularities at which the control unit may supply RTL routines to the data path<sup>4</sup>. The first option is to generate the entire RTL routine for the execution of an instruction (about 3-5 RTL instructions) in one shot, and give it to the data path. The data path will then need a storage structure to store the entire routine. It will also have to do its own sequencing through this RTL routine. Because of shifting some of the sequencing burden to the data path, the control unit will be somewhat simpler.

In the alternate option — the more prevalent one — RTL instructions are supplied one at a time to the data path. The control unit is then responsible for sequencing through the RTL sequence. The data path just needs to carry out the most recent RTL instruction that it has received from the control unit. We will be dealing exclusively with this option in this chapter.

Figure 9.3 shows the general relationship between the overall data path and the processor control unit. The figure explicitly shows only the processor's data path. The processor control unit periodically receives the status of the data path, and specifies the next microinstruction to be performed by the data path. Thus, the processor control unit controls the processing of data within the processor data path, as well as the flow of data within, to, and from the processor data path. In this section we show how to design a processor control unit that generates microinstructions in a timely fashion.

### 9.4.1 Developing an Algorithm for RTL Instruction Generation

Before the design of the control unit can begin, it is imperative to define the processor data path (including the interface to the processor-memory bus) as well as the RTL routines

---

<sup>4</sup>An important point is in order here. RTL instructions are not supplied to the data path in the English-like RTN format that we have been using in this chapter. Rather, they are supplied in an encoded manner called **microinstructions**. In the ensuing discussion, we often use the terms “microinstruction” and “RTL instruction” interchangeably.

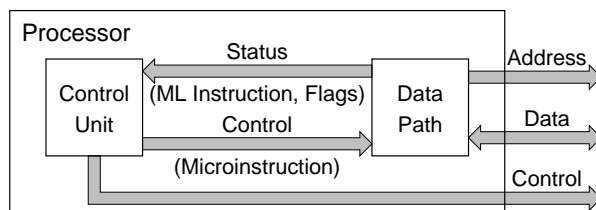


Figure 9.3: A Microarchitectural View of the Interaction between the Processor Control Unit, Processor Data Path, and the Rest of the System

for interpreting each of the machine language instructions. The control unit designer then translates all of the RTL routines into an encoded form called **microroutines** either by hand or by a **micro-assembler** (which is a program similar to an assembler). In implementing the control unit, there are two distinct aspects to deal with: (i) proper sequencing through the steps, and (ii) generating the appropriate microinstruction for each step. We can achieve both of these tasks by first combining the RTL routines for all of the machine language instructions into a single microprogram. Such an algorithm will contain not only intra-routine sequencing, but also inter-routine sequencing.

We shall use the now familiar data path of Figure 9.1 (on page 365) for designing the sample control unit. For this data path, we had already seen the RTL routines to be generated to carry out the fetch phase of the MIPS-0 ISA instructions, and the execute phase of the following three instructions: (i) `addu rd, rs, rt`, (ii) `lw rt, offset(rs)`, and (iii) `beq rs, rt, offset`. These routines are available in Tables 9.4-9.7.

The next step in the design is to put all of these information together to develop a flowchart or algorithm for the functioning of the control unit. Table 9.8 combines the generation of all of these RTL routines into a single algorithm starting at step 0. The first column, **step**, indicates the current step performed by the control unit, and the next column, **next step**, indicates the flow of control through this algorithm. The third field indicates the RTL instruction to be generated for the current step, and the last field indicates comments, if any. The **next step** field facilitates the processor control unit in its sequencing function. As can be seen from the **next step** entries, control flow through the control unit algorithm is mostly straightline in nature. i.e., the **next step** is generally the immediately following step.

In this algorithm, the generation of the instruction fetch routine appears just once (as it is the same for all instructions), and includes steps 0 through 3. The decode process takes place in step 3, and enables the control circuitry to choose the appropriate RTL instructions for the execution phase. The **next step** field for step 3 has an entry of *n*. This indicates that a multi-way branch is required in the control unit's algorithm, based on the opcode of the decoded ML instruction. Notice also that the **next step** field is set to 0 at the end of each *execute* routine to indicate that control has to go back to step 0 after the execution of that RTL instruction.

| Step                                 | Next Step | RTL Instruction Generated for Data Path             | Comments                         |
|--------------------------------------|-----------|---|----------------------------------|
| Fetch phase of every instruction     |           |   |                                  |
| 0                                    | 1         | PC $\rightarrow$ MAR; PC $\rightarrow$ AIR          | Decode<br>Branch based on opcode |
| 1                                    | 2         | M[MAR] $\rightarrow$ MDR; AIR + 4 $\rightarrow$ AOR |                                  |
| 2                                    | 3         | MDR $\rightarrow$ IR                                |                                  |
| 3                                    | n         | AOR $\rightarrow$ PC                                |                                  |
| Execute phase of addu                |           |   |                                  |
| 4                                    | 5         | R[rs] $\rightarrow$ AIR                             | End of routine<br>Go to step 0   |
| 5                                    | 6         | R[rt] + AIR $\rightarrow$ AOR                       |                                  |
| 6                                    | 0         | if (rd) AOR $\rightarrow$ R[rd]                     |                                  |
| Execute phase of lw                  |           |   |                                  |
| 7                                    | 8         | R[rs] $\rightarrow$ AIR                             | End of routine<br>Go to step 0   |
| 8                                    | 9         | AIR + SE(offset) $\rightarrow$ AOR                  |                                  |
| 9                                    | 10        | AOR $\rightarrow$ MAR                               |                                  |
| 10                                   | 11        | M[MAR] $\rightarrow$ MDR                            |                                  |
| 11                                   | 0         | if (rt) MDR $\rightarrow$ R[rt]                     |                                  |
| Execute phase of beq                 |           |   |                                  |
| 12                                   | 13        | PC $\rightarrow$ AIR                                | End of routine<br>Go to step 0   |
| 13                                   | 14        | SE(offset) $\ll$ 2 $\rightarrow$ AOR                |                                  |
| 14                                   | 15        | AIR + AOR $\rightarrow$ AOR                         |                                  |
| 15                                   | 16        | R[rs] $\rightarrow$ AIR                             |                                  |
| 16                                   | 17        | R[rt] == AIR $\rightarrow$ Z                        |                                  |
| 17                                   | 0         | if (Z) AOR $\rightarrow$ PC                         |                                  |
| Execute phase of next ML instruction |           |   |                                  |
| ...                                  |           |   |                                  |

Table 9.8: The Algorithm followed by the Control Unit for Generating RTL Instructions so as to Interpret MIPS-0 ML Programs for Execution in the Single-Bus Data Path of Figure 9.1

On inspecting the algorithm given in Table 9.8, we can see that the processor control unit essentially goes through an infinite loop starting from step 0. This control unit can be designed as a finite state machine (FSM). In order to do this, we can develop a state transition diagram for this algorithm. This state diagram corresponds to a Moore-type FSM, where the output values are strictly a function of the current state. Figure 9.4 presents a state transition diagram for this FSM. In this diagram, each step of Table 9.8 is implemented by a state, which decides the microinstruction to be generated when in that state. The figure also indicates the conditions that cause the control unit FSM to go from one state to another. Most of the transitions are not marked by a condition, which means that those transitions occur unconditionally. The first 4 states, S0 - S3, correspond to the instruction fetch routine; the microinstruction to be generated in each of these states  $s$

corresponds to what is given in Table 9.8 for step  $s$ . At the end of the microroutine for instruction fetch, the fetched ML instruction would have been decoded, and a multi-way branch occurs from state S3 to the appropriate microroutine for executing the instruction. By specifying the states and their transitions, we specify the microinstruction the processor control unit must generate in order for the data path to fetch, decode, and execute every instruction in the ISA.

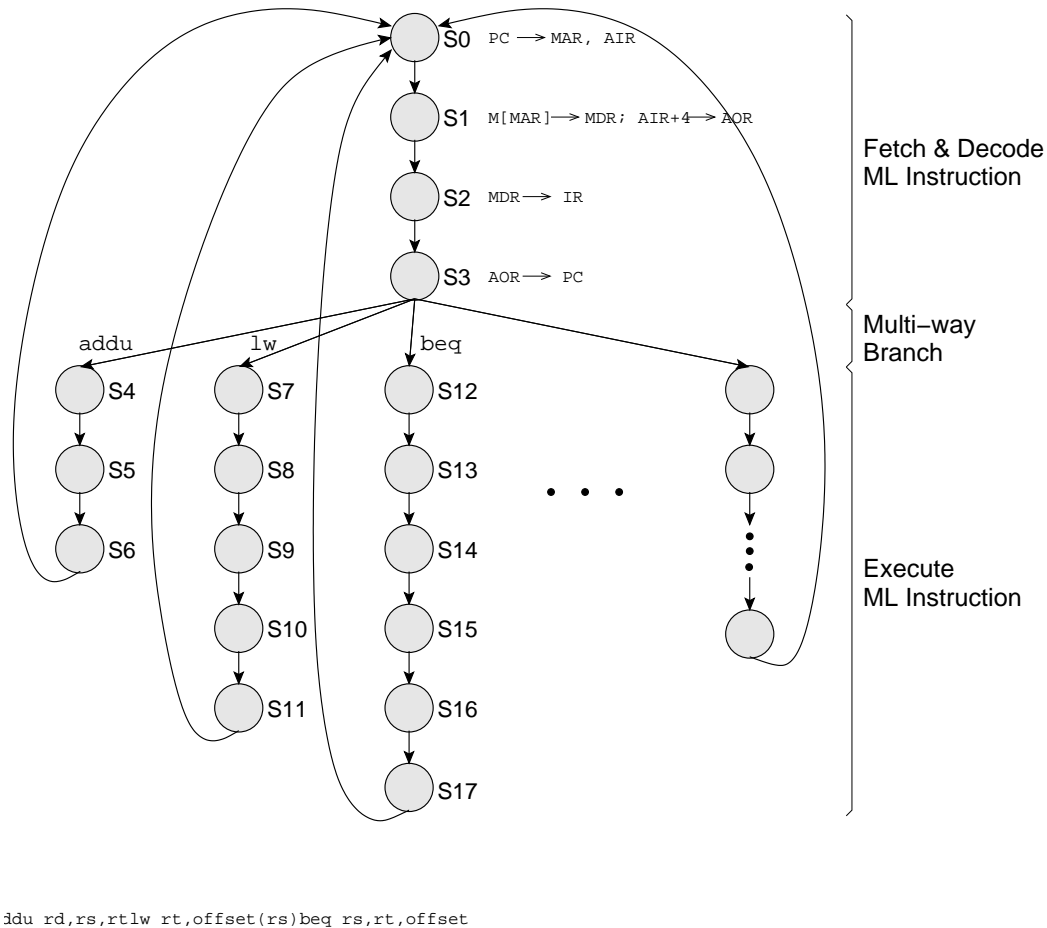


Figure 9.4: A State Transition Diagram for a Processor Control Unit for the Single-Bus Data Path of Figure 9.1

Having developed a state transition diagram to interpret ML instructions, the next step is to design a hardware unit that implements this state transition diagram. Figure 9.5 gives a high-level block diagram of a generic processor control unit FSM. This FSM performs two functions:

- Sequencing through the control unit states
- Generating the microinstruction for the current state

The left side of the figure handles the sequencing part, and the right side of the figure handles the microinstruction generation part.

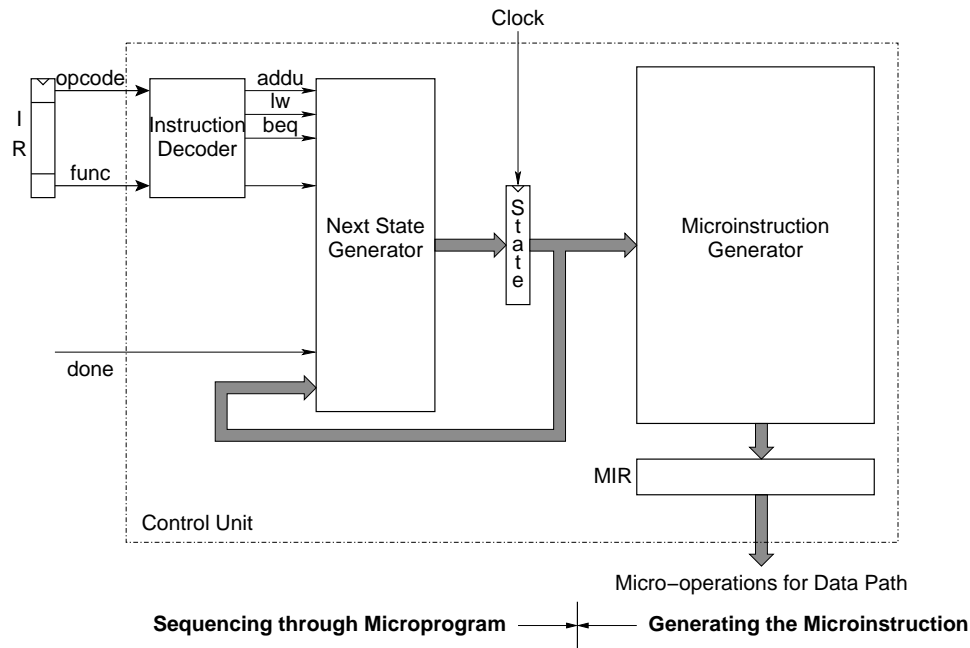


Figure 9.5: A Block Diagram of a Processor Control Unit

The FSM's state is stored in a **state** register. The **microinstruction generator** block takes the contents of **state** as input and generates the corresponding microinstruction, which can be optionally latched onto a **MIR** (microinstruction register). Please be careful not to confuse **state** with **PC**; **MIR** with **IR**; and microinstructions with ML instructions.

The next state value is calculated by a combinational block called **next state generator**, based on the current state and some inputs from the data path. Normally, the state value is incremented at each clock pulse, causing successive microinstructions to be generated. The following 3 events cause a deviation from this straightline sequencing.

- When the current state corresponds to the end of the fetch microroutine (state 3 in our example), a new ML instruction has just been decoded. The **next state generator** has to deviate from straightline sequencing so as to start generating the execute-microroutine corresponding to the newly decoded ML instruction. It determines the



next state based on the output of the instruction decoder. Thus, the control unit is able to generate the microinstructions for executing the newly decoded ML instruction.

- Similarly, when the current state corresponds to the end of an execute-microroutine, the **next state generator** block deviates from straightline sequencing, and initializes **state** to zero so as to start fetching the next ML instruction.
- Finally, some microinstructions require multiple clock cycles to complete execution. For instance, a memory read microinstruction (written symbolically in RTL as  $M[MAR] \rightarrow MDR$ ) may not be completed in a single clock cycle. Worse still, the completion time of this microinstruction may not even be deterministic due to a variety of reasons, as we will see later in this chapter. The easiest approach to handle state updation when executing a multi-cycle microinstruction is to maintain the same state value until the microinstruction is completed. For microinstructions with non-deterministic execution latencies, information regarding their completion can be conveyed by the data path to the control unit using a signal called **done**.

Thus, the updation of the **state** register in the control unit is performed by the **next state generator** block, based on:

1. the contents of the **state** register
2. the outputs of the instruction decoder
3. the **done** signal from the data path.

### 9.4.3 Incorporating Sequencing Information in the Microinstruction

The **next state generator** block can be simplified by including in each microinstruction the necessary sequencing information. That is, the **goto** operations of Table 9.8 can also be included in the microinstruction (in an encoded form). With such an arrangement, when a microinstruction is generated, if it contains a **goto** operation, this information can be fed back to the **next state generator** block, as shown in Figure 9.6. For instance, if the current microinstruction includes the encoded form of **goto  $S_n$** , this information (along with the instruction decoder output) will be used by **next state generator** to deviate from straightline sequencing. Similarly, while sequencing through an execute-microroutine, if the current microinstruction includes the binary equivalent of **goto  $S_0$** , **next state generator** resets the **state** register to zero, so that in the next cycle the processor can begin fetching the next ML instruction. Notice that the **microinstruction generator** block becomes more complex.

At the digital logic level, the sequencing part of the control unit FSM can be implemented using a sequencer plus a decoder or one flip-flop per state. The sequencer can be built in many ways. One possibility is to use a counter or shift register with synchronous reset

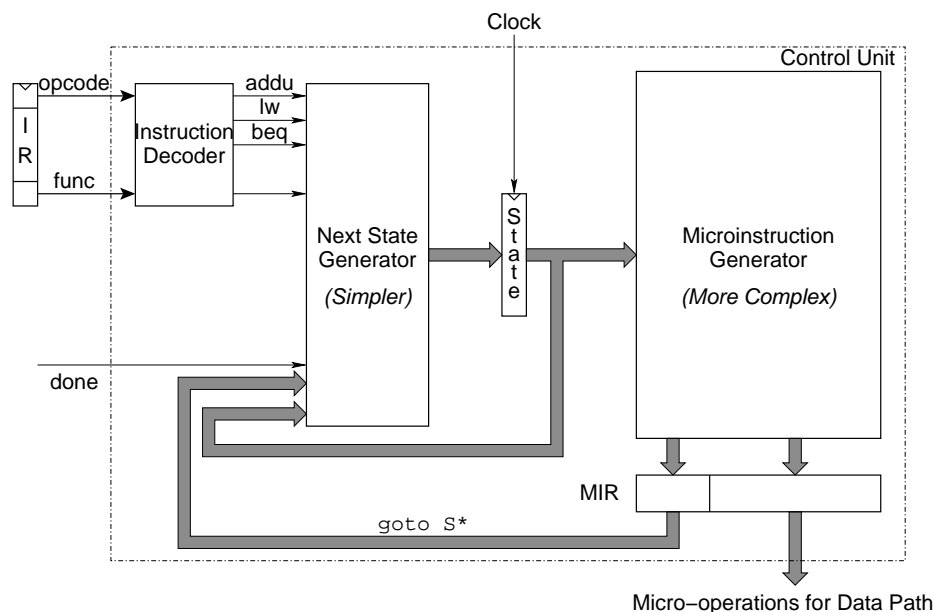


Figure 9.6: A Block Diagram of a Processor Control Unit that encodes Sequencing Information in the Microcode to Simplify the Next State Generator Block

and parallel loading facility. The **microinstruction generator** can also be built in more than one way. Two common methods involve the use of either discrete logic or ROM. The control units so designed are called **hardwired control** and **microprogrammed control**, respectively. We will discuss both of these approaches in Chapter 10.

#### 9.4.4 State Reduction

The state diagram of the control unit we just designed incorporates a separate microroutine for the execution part of each ML instruction. For an ISA that specifies hundreds of instructions, this approach is likely to produce an FSM with thousands of states. Many of the states in such an FSM can actually be combined by considering the fact that the execution microroutines of similar instructions have many equivalent states. For instance, the first 3 states in the execution microroutine of a memory-referencing instruction deal with computing the effective address, and will be the same for all memory-referencing instructions. Just like we used a common microroutine for the fetch phase of all ML instructions, we can use a common *address calculation* microroutine for all memory-referencing instructions in the MIPS ISA. If we take this approach, we can get a reduced FSM as shown in Figure 9.7. This FSM has far fewer states than the one given earlier, allowing a much smaller **Microinstruction Generator** to be used. Notice, however, that we will have more **goto** operations, which

either offset part of the decrease in size of the **Microinstruction Generator** (assuming that the **goto** operations are also generated by the **Microinstruction Generator**).

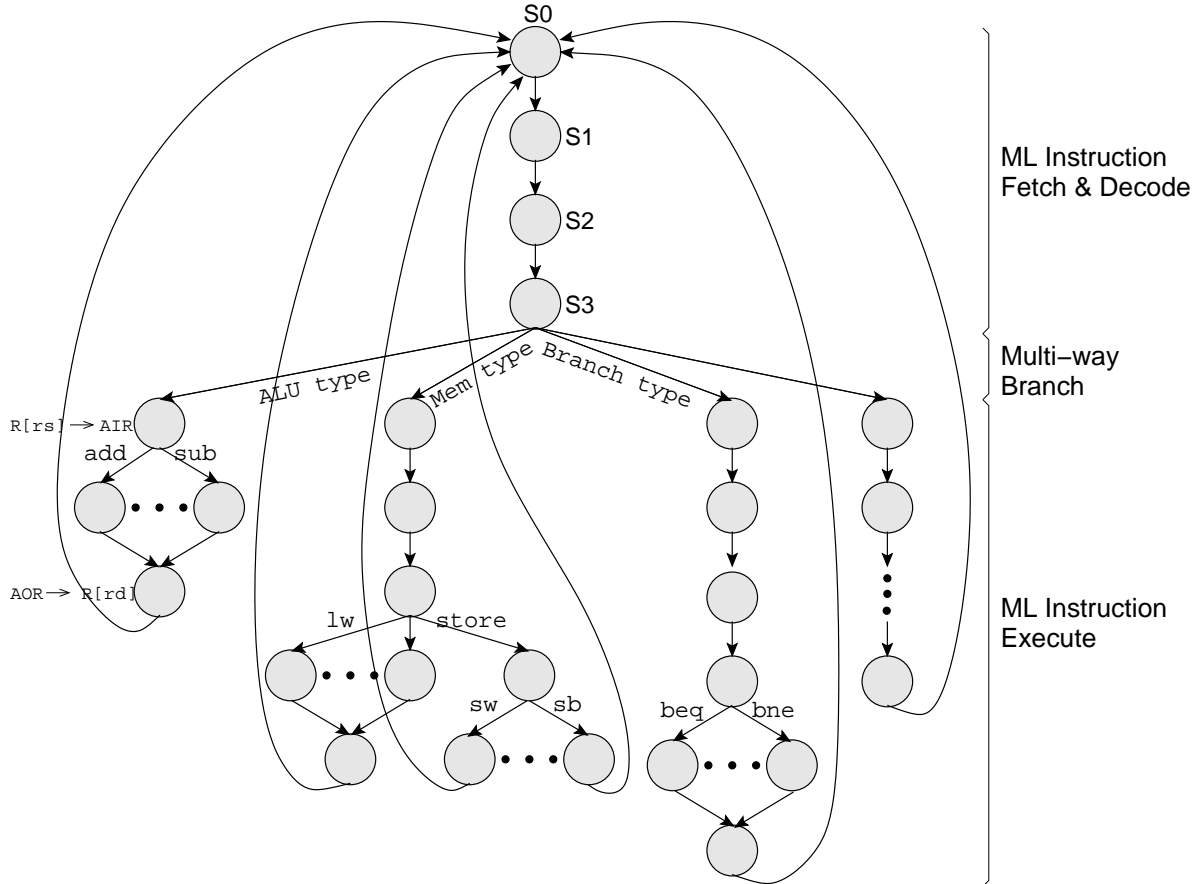


Figure 9.7: A Reduced State Diagram for a Processor Control Unit for the Single-Bus Data Path of Figure 9.1

With a reduced state diagram, we have a smaller **Microinstruction Generator**, although at the expense of a more complex **Next State Generator**.

## 9.5 Memory System Design

The memory system is an integral component of the microarchitecture of a stored program computer, as it stores the instructions and data of the program being executed. In this section we look at the RTL architecture of this subsystem more closely.

### 9.5.1 A Simple Memory Data Path

### 9.5.2 Memory Interface Unit

We now look at how the memory system is interfaced to the rest of the computer system. This interface is much simpler than, say, the one used to interface IO devices. This is because the memory system is made out of similar technology as that of the processor.

### 9.5.3 Memory Controller

The memory controller supports the processor control unit in its interpretation function. In particular, it accepts *read* and *write* requests from the processor control unit (routed via the register bus - memory bus bridge), and provides the appropriate control information for the memory data path to carry out these functions.

### 9.5.4 DRAM Controller

In most memory systems, a DRAM controller performs the task of address multiplexing and the generation of the DRAM control signals. It also contains the refresh counters and the circuitry required to do a periodic refresh of the DRAM cells.

### 9.5.5 Cache Memory Design

In the previous chapter we saw that the memory subsystem of most computers is usually implemented as a hierarchy. A memory hierarchy can consist of multiple levels, but data is copied only between two adjacent levels at a time. The fastest elements are the ones closest to the processor. The top part of the hierarchy will supply data most of the time because of the principle of locality. Memory hierarchies take advantage of temporal locality by keeping recently accessed data items closer to the processor. They take advantage of spatial locality by moving blocks consisting of multiple contiguous words to upper levels of the hierarchy, even though only a single word was requested by the processor.

#### 9.5.5.1 Indexing into the Cache

In Section 7.4.6, we saw that given a memory address, the cache controller first finds the memory block number to which the address belongs. This can be determined by dividing the memory address by the block size. Because it is time-consuming to do an integer division operation, cache designers only use block sizes that are powers of 2. The division, then, becomes taking out the upper bits of the memory address. Figure 9.8 shows how a memory address bit pattern is split. The first split is between the **Main Memory Block Number** and the **Offset within Block**. The **Main Memory Block Number** field is further

split into the **Tag** field and the **Cache Set Number** field. The **Offset within Block** field selects the desired word once the block is found, the **Cache Set Number** field selects the cache set, and the **Tag** field is compared against the tag values in the selected set to check if a hit occurs.

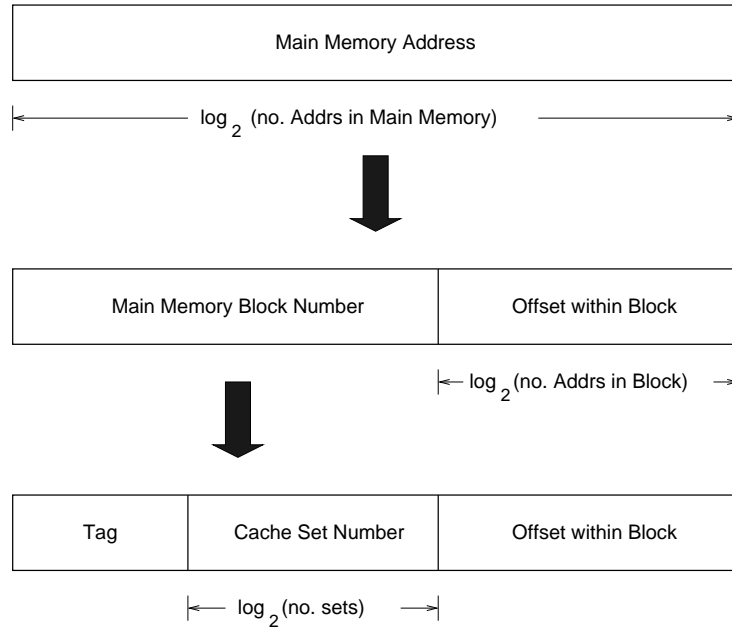


Figure 9.8: How the Cache Memory Controller Splits a Memory Address Bit Pattern

### 9.5.6 Cache Controller: Interpreting a Read/Write Command

The job of a cache controller is to accept requests from the upper level of the memory hierarchy (the level that is immediately closer to the processor), and service them. If the requested word is present in the cache, then the request can be satisfied with a short latency. If this is not the case, then the cache controller will have to send additional requests to the immediately lower level of the memory hierarchy.

## 9.6 Processor Data Path Interconnects: Design Choices

The previous sections discussed the basics of designing a data path for interpreting machine language instructions. We used a very simple data path for ease of understanding. Although such a data path may not provide high performance, it may be quite sufficient for certain embedded applications. The data path of general-purpose computers, on the other hand,

is far more complex. In this section and the one following it, we will introduce high-performance versions of the processor data path. The processor plays a central role in a computer microarchitecture's function. It communicates with and controls the operation of other subsystems within the computer. The performance of the processor therefore has a major impact on the performance of the computer system as a whole.

The performance of a computer depends on a number of factors, many of which are related to the design of the processor data path. Three of the most important factors are the strength of the machine language instructions, the number of clock cycles required to fetch and execute a machine language instruction, and the clock speed. A powerful ML instruction performs a complex operation and accomplishes more than what a simple instruction accomplishes, although its execution might take several additional clock cycles. The strength of instructions is an issue that is dealt with at the ISA level (taking into consideration microarchitectural issues), and is not under the control of the microarchitect.

Clock speed has a major influence on performance, and depends on the technology used to implement the electronic circuits. The use of densely packed, small transistors to fabricate the digital circuits leads to high clock speeds. Thus, implementing the entire processor on a single VLSI chip allows much higher clock speeds than would be possible if several chips were used. The use of simple logic circuits also makes it easier to clock the circuit faster. Clock speed is an issue that is primarily dealt with at the design of logic-level architectures, although microarchitectural decisions have a strong bearing on the maximum clock speeds attainable.

The third factor in performance, namely the number of clock cycles required to fetch and execute an ML instruction, is certainly a microarchitectural issue. In the last several years, speed improvements due to better microarchitectures, while less amazing than that due to faster circuits, have nevertheless been impressive. There are two ways to reduce the average number of cycles required to fetch and execute an instruction: (i) reduce the number of cycles needed to fetch and execute each instruction, and (ii) overlap the interpretation of multiple instructions so that the average number of cycles per instruction is reduced. Both involve significant modifications to the processor data path, primarily to add more connectivity and latches.

### 9.6.1 Multiple-Bus based Data Paths

In order to reduce the time required to execute machine language instructions, we need to redesign the data path so that several micro-operations can be performed in parallel. The more interconnections there are between the different blocks of a data path, the more the data transfers that can happen in parallel in a clock cycle. The number of micro-operations that can be done in parallel in a data path does depend on the connectivity it provides. In a bus-based processor data path, the number of buses provided is probably the most important limiting factor governing the number of cycles required to interpret machine language instructions, as most of the microinstructions would need to use the

bus(es) in one way or other. A processor data path with a single internal bus, such as the one given in Figure 9.1, will take several cycles to fetch the register operands and to write the ALU result to the destination register. This is because it takes one cycle to transfer each register operand to the ALU, and one cycle to transfer the result from the ALU to a register. An obvious way to reduce the number of cycles required to interpret ML instructions is to include multiple buses in the data path, which makes it possible to parallelly transfer multiple register values to the ALU. Before going into specific multiple bus-based data paths, let us provide a word of caution: buses can consume significant chip area and power. Furthermore, they may need to cross at various points in the chip, which can make it difficult to do the layout at the device level.

## A 2-Bus Processor Data Path

The single-bus processor data path given in Figure 9.1 can be enhanced by adding one more internal bus. The additional bus can be connected to the blocks in different ways. Figure 9.9 pictorially shows one possible way of connecting the second bus. In the figure, the new bus is shown in darker shade, and is used primarily for routing the ALU result back to different registers. A unidirectional path is provided from the first bus to the second. Because the ALU result can be routed through a bus that is different from the one used for routing the source operand, there is no need to buffer the ALU result in AOR, and so we do not include AOR in this data path. Notice that to perform a register read and a register write in the same clock cycle, the register file requires two ports (a read port and a write port).

Table 9.9 gives an RTL interpretation of `addu rd, rs, rt` in this 2-bus data path. On comparing this routine against that given in Table 9.5, we can see that steps 2 and 3 have been combined into a single step, as we can use the two buses to do both transfers in parallel. In the new step 3, when the instruction is being decoded, the `rs` field of IR is used to read GPR numbered `rs` into AIR in a *speculative* manner. That is, this transfer is done before completing instruction decoding, with the anticipation that it would be required later. Because most of the MIPS instructions specify `rs` as one of the source operands, this technique results in a saving of one clock cycle most of the time. Notice that the ALU operation (step 4) directly writes the result into GPR `rd`, and therefore the number of clock cycles required to interpret the instruction is reduced by two compared to that given in Table 9.5.

On a similar note, Table 9.10 gives a RTL interpretation of `lw rt, offset(rs)` in this 2-bus data path. This RTL routine has 7 steps, which is 2 steps fewer than the one given in Table 9.6 for the single bus data path. One step is saved because of speculatively copying the contents of register `rs` to register AIR in step 3. Another step is saved because the computed address can be directly stored into MAR without going through an AOR register.

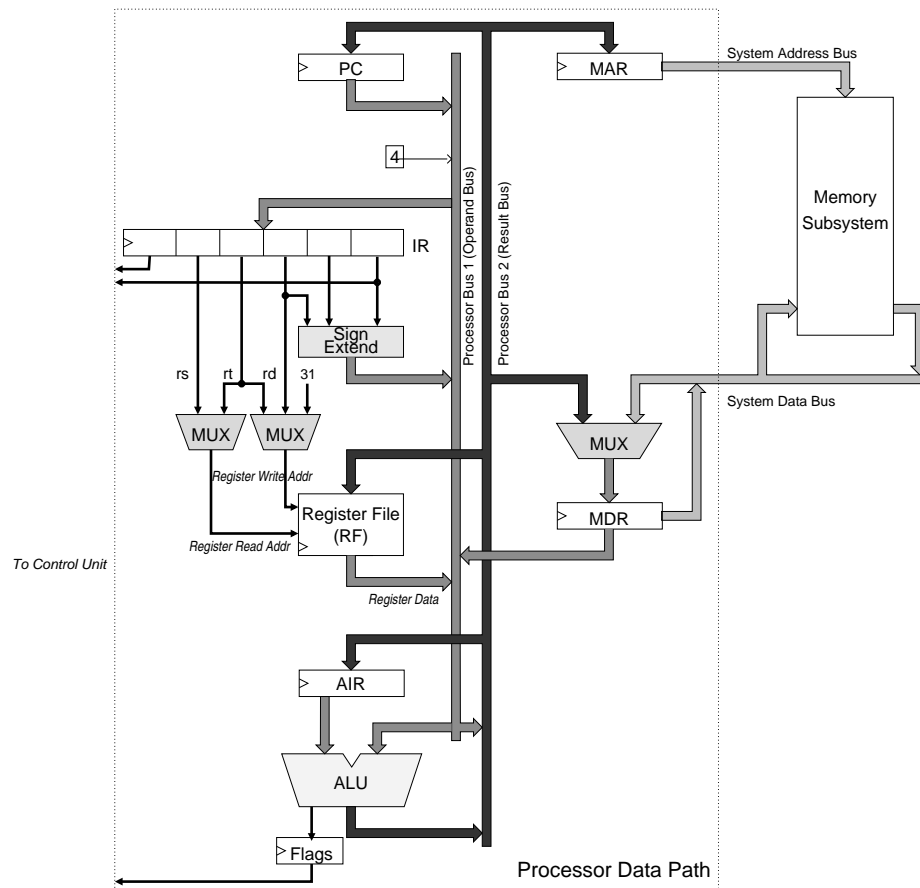


Figure 9.9: A 2-Bus Based Processor Data Path for Implementing the MIPS-0 ISA

### 9.6.2 Direct Path-based Data Path

As we keep adding more buses to the processor data path, we eventually get a data path that has many point-to-point connections or direct paths. If we are willing to use a large number of such direct path connections, it is better to redesign the processor data path. Figure 9.10 presents a direct path-based data path for the MIPS-0 ISA. In this data path, instead of using one or more buses, point-to-point connections or direct paths are provided between each pair of components that transfer data. This approach allows many register transfers to take place simultaneously, thereby permitting multiple RTL operations to be executed in parallel. To facilitate parallel transfers, we have added a few microarchitectural registers, namely, *Rrs*, *Rrt*, and *Offset*. As indicated by their names, these registers are used to store copies of the *rs* register, *rt* register, and sign-extended *offset*, respectively.



| Step No.      | RTL Instruction for Data Path                      | Comments     |
|---------------|--|--------------|
| Fetch phase   |  |              |
| 0             | PC $\rightarrow$ MAR; PC $\rightarrow$ AIR         | Decode instr |
| 1             | M[MAR] $\rightarrow$ MDR; AIR + 4 $\rightarrow$ PC |              |
| 2             | MDR $\rightarrow$ IR                               |              |
| 3             | R[rs] $\rightarrow$ AIR                            |              |
| Execute phase |  |              |
| 4             | R[rt] + AIR $\rightarrow$ R[rd]                    |              |

Table 9.9: An RTL Routine for Interpreting the MIPS-0 ML Instruction Represented Symbolically as `addu rd, rs, rt`, in the 2-Bus Data Path

| Step No.      | RTL Instruction for Data Path                      |  |  |  | Comments     |
|---------------|--|--|--|--|--------------|
| Fetch phase   |  |  |  |  |              |
| 0             | PC $\rightarrow$ MAR; PC $\rightarrow$ AIR         |  |  |  | Decode instr |
| 1             | M[MAR] $\rightarrow$ MDR; AIR + 4 $\rightarrow$ PC |  |  |  |              |
| 2             | MDR $\rightarrow$ IR                               |  |  |  |              |
| 3             | R[rs] $\rightarrow$ AIR                            |  |  |  |              |
| Execute phase |  |  |  |  |              |
| 4             | SE(offset) + AIR $\rightarrow$ MAR                 |  |  |  |              |
| 5             | M[MAR] $\rightarrow$ MDR                           |  |  |  |              |
| 6             | MDR $\rightarrow$ R[rt]                            |  |  |  |              |

Table 9.10: An RTL Routine for Interpreting the MIPS-0 ML Load Instruction Represented Symbolically as `lw rt, offset(rs)`, in the 2-Bus Data Path

The use of direct paths probably makes the data path's functioning easier to visualize, because each interconnection is now used for a specific transfer as opposed to the situation in a bus-based data path where a bus transaction changes from clock cycle to clock cycle. Notice, however, that a direct path based data path is even more tailored to a particular ISA.

If multiple sources feed a block, a multiplexer is used to select the required source in each cycle. For instance, in our data path, the second input of the ALU can receive data from register `Rrt` as well as from register `Offset`. Therefore, we use a multiplexer to select one of these inputs. Another modification is that there are two paths emanating from the register file. If both paths are to be active in the same cycle, then the register file needs to have two *read ports*.

Table 9.11 gives RTL routines for interpreting the 3 MIPS-0 ML instructions (i) `addu`

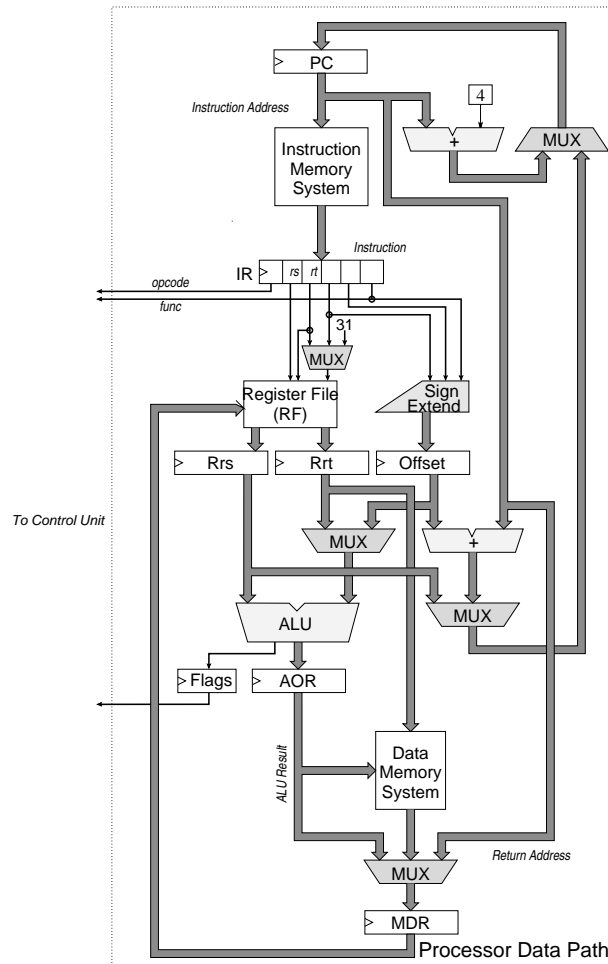


Figure 9.10: An RTL Implementation of the Direct Path-Based Processor Data Path in Figure 7.13

`rd, rs, rt`, (ii) `lw rt, offset(rs)`, and (iii) `beq rs, rt, offset` in this direct path-based data path. You can see that the routine for the `lw` instruction requires only 5 steps to complete, which is 2 steps less than the one given in Table 9.10 for the 2-bus data path. There is no reduction in steps for the `addu` instruction; this is because there is no direct path from the output of the ALU to the register file.

| Step No.                            | RTL Instruction for Data Path                    | Comments     |
|-------------------------------------|--|--------------|
| Fetch phase                         |  |              |
| 0                                   | IM[PC] → IR; PC + 4 → PC                         | Decode instr |
| 1                                   | R[rs] → Rrs; R[rt] → Rrt;<br>SE(offset) → Offset |              |
| Execute phase of addu rd, rs, rt    |  |              |
| 2                                   | Rrs + Rrt → AOR                                  |              |
| 3                                   | AOR → R[rd]                                      |              |
| Execute phase of lw rt, offset(rs)  |  |              |
| 2                                   | Rrs + Offset → AOR                               |              |
| 3                                   | DM[AOR] → R[rt]                                  |              |
| Execute phase of beq rs, rt, offset |  |              |
| 2                                   | Rrs == Rrt → Z                                   |              |
| 3                                   | if (Z) PC + LS(Offset) → PC                      |              |

Table 9.11: RTL Routines for Interpreting 3 MIPS-0 ML Instructions, in the Direct Path based Data Path

## 9.7 Pipelined Data Path: Overlapping the Execution of Multiple Instructions

### 9.7.1 Defining a Pipelined Data Path

To begin with, we need to determine what should happen in the data path during every clock cycle, and make sure that the same hardware resource is not used by two different instructions during the same clock cycle. Ensuring this becomes difficult if the same hardware resource is used in multiple cycles during the execution of an ML instruction. This makes it difficult to use a bus-based data path, as buses are used multiple times during the execution of an instruction. Therefore, our starting point is the direct path-based data path that we saw in Figure 9.10. What would be a natural way to partition this data path into different stages? While we can think of many different ways to partition this data path, a straightforward way to partition is as follows: place the hardware resources that perform each of the RTL instructions of the RTL routines (in Table 9.11) in a different stage. The 5 stages of our pipeline are named *fetch* (F), *read source registers* (R), *ALU operation* (A), *memory access* (M), and *end* (E).

Next, let us partition the direct path-based data path of Figure 9.10 as per this 5-stage framework. Figure 9.11 shows such a partitioning. In this pipelined data path, the instruction fetch portion of the data path—which includes PC and its update logic—has been apportioned as the F stage. IR, which stores the fetched instruction, forms a buffer between the F stage and the R stage. The register file, its access logic, and the sign-extend unit have been placed in the R stage. Similarly, the ALU—along with its input MUXes—

and the left-shift unit are placed together in the A stage. AOR, which stores the output of the ALU, forms a buffer between the A stage and the M stage, which houses the logic for accessing the data memory. Finally, the E stage just consists of the logic for updating the register file. Table 9.12 shows the primary resources used in each stage of the pipeline.

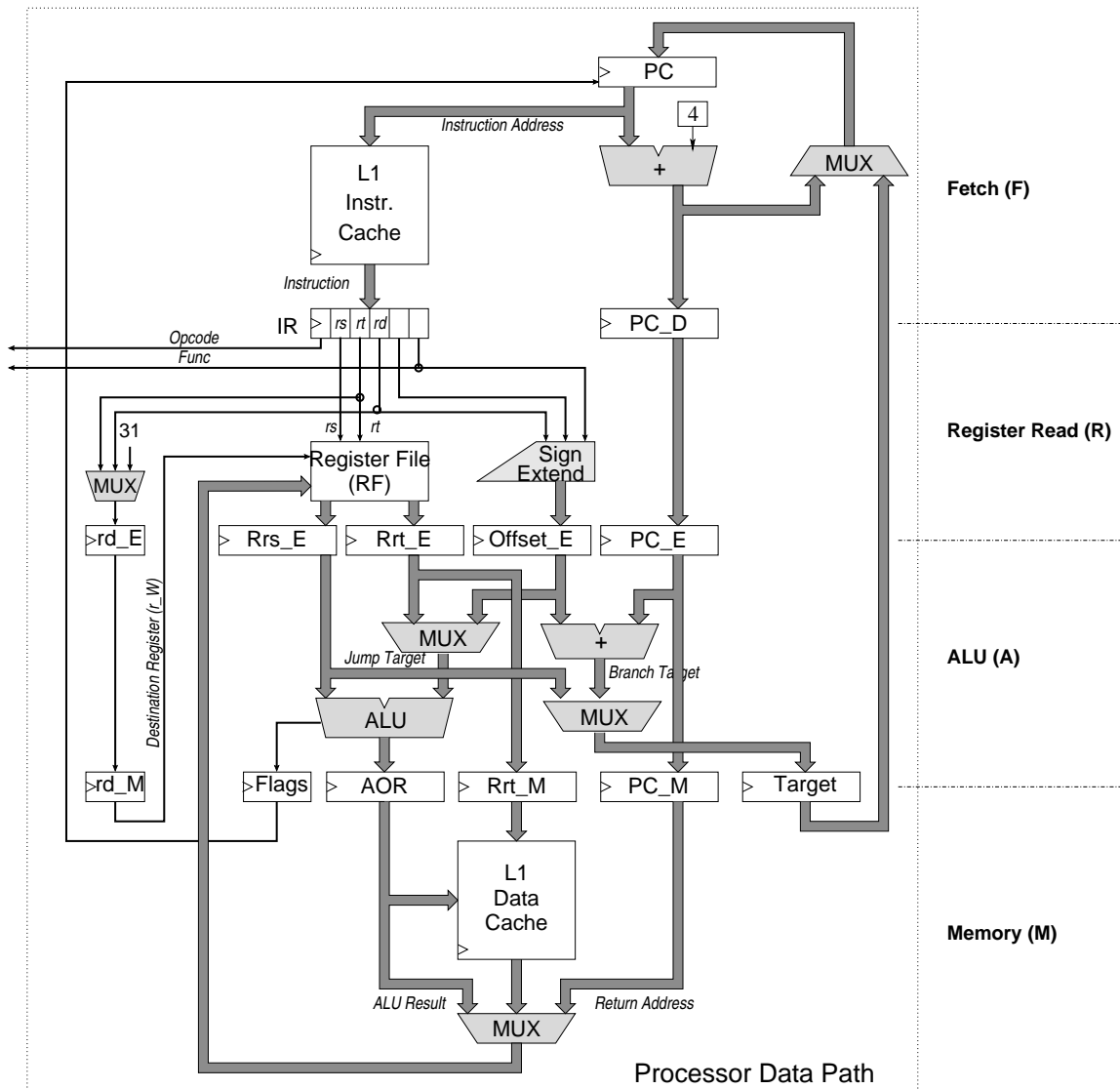


Figure 9.11: A Pipelined Processor Data Path for Implementing the MIPS-0 ISA

In the pipelined data path being discussed, an ML instruction uses the major hard-

| Stage         | Primary Resources                  |
|---------------|------------------------------------|
| Fetch         | PC, Instruction memory             |
| Read register | Register file, Sign extension unit |
| ALU           | ALU                                |
| Memory access | Data memory, Register file         |

Table 9.12: Primary Resources Used in Each Stage of the Pipelined Data Path of Figure 9.11

ware blocks in different clock cycles, and hence overlapping the interpretation of multiple instructions introduces relatively fewer conflicts. Some of the hardware blocks in the direct path-based data path are used multiple times while interpreting some ML instructions. In order to avoid conflicts for these hardware blocks, they have been replicated in the pipelined data path. For example, a single PC cannot store the addresses of two different ML instructions at the same time. Therefore, the pipelined data path must provide multiple copies of PC, one for each instruction that is being executed in the processor.

Pipelining has been used since the 1960s to increase instruction throughput. Since then, a number of hardware and software features have been developed to enhance the efficiency and effectiveness of pipelining. Not all pipelined data paths have all of these characteristics; the ones we discuss below are quite prevalent.

**Multi-ported Register File:** When multiple instructions are present in a pipelined data path, two instructions cannot be attempting to use the same hardware resource in the same clock cycle. This requirement is straightforward to meet if each hardware resource is accessed exactly in one pipeline stage. A prime example of this is the ALU. It is accessed only by the instruction present in the A stage. An inspection of Table 9.12 reveals that the register file, on the other hand, may be used by instructions present in two different stages of the pipeline—stages R and M. Thus, in any given clock cycle, when the instruction in the R stage is trying to read from the register file, a previous instruction in the M stage may be attempting to write to the register file. The solution adopted in our data path is to make the register file *multi-ported*. That is, the register file permits multiple accesses to be made in the same clock cycle<sup>5</sup>.

**Separate Instruction Cache and Data Cache:** Just like the case with register file accesses, memory accesses may also be performed from multiple pipeline stages — instruction fetch from the F stage and data access from the M stage. If a single memory structure is used for storing both the instructions and the data, then the data access done by a memory-referencing instruction (load or store) will happen at the same time as the instruc-

---

<sup>5</sup>Making a register file multi-ported is very likely to increase the time it takes to read from or write to the register file.

tion fetch for a succeeding instruction. The data path of Figure 9.11 used two different memory structures—an L1 instruction cache and an L1 data cache—to avoid this problem. Another option is to use a single dual-ported memory structure.

**Interconnect:** Pipelined data paths typically use the direct path-based approach discussed earlier, which uses direct connections between registers. A bus-based data path is not suitable for pipelining, because it prevents more than one instruction from using the bus in the same clock cycle. The direct path approach allows many register transfers to take place simultaneously in each pipeline stage, which indeed they must if the goal of allowing many activities to proceed simultaneously is to be achieved. In fact, we may have to add more interconnections to allow for more parallel data transfers.

**Pipeline Registers or Latches:** When an instruction moves from one pipeline stage to the next, the vacated stage is occupied by the next instruction. Because of this, whatever an instruction needs to know about itself at each pipeline stage must be carried along until that stage is reached. Such information might include opcode, data values, etc. Each instruction must “carry its own bags,” so to speak. Thus, portions of both the data and the instruction must travel from stage to stage, even if they are not modified by a particular stage. To do this in a convenient fashion, extra registers, called *pipeline latches*, are typically inserted between the pipeline stages. For example, the incremented PC value is carried along, with the help of pipeline registers PC\_D, PC\_E, and PC\_M.

**Additional Hardware Resources:** Pipelined data paths usually need additional hardware resources to avoid resource conflicts between simultaneously interpreted instructions. A likely candidate is a separate incrementer to increment the PC, freeing the ALU for doing arithmetic/logical operations in every clock cycle.

### 9.7.2 Interpreting ML Instructions in a Pipelined Data Path

The objective of designing a pipelined data path is to permit the interpretation of multiple ML instructions at the same time, in a pipelined manner. The RTL routines used for interpreting each ML instruction is similar to those for the direct path-based data path. Table 9.13 illustrates how the RTL routines for multiple ML instructions are executed in parallel.

### 9.7.3 Control Unit for a Pipelined Data Path

Converting an unpipelined data path into a pipelined one involves several modifications, as we just saw. The control unit also needs to be modified accordingly. We shall briefly look at how the control unit of a pipelined data path works. With a pipelined data path, in each

| Step No. | RTL Instruction                |                                 |  |
|----------|--------------------------------|---------------------------------|--|
|          | lw rt, offset(rs)              | addu rd, rs, rt                 | beq rs, rt, offset   |
| 0        | <i>RTL Instr for Fetch</i>     |                                 |  |
| 1        | <i>RTL Instr for Reg Read</i>  | <i>RTL Instr for Fetch</i>      |  |
| 2        | Rrs_E + Off_E $\rightarrow$ AR | <i>RTL Instr for Reg Read</i>   | <i>RTL Instr for Fetch</i>   |
| 3        | DM[AOR] $\rightarrow$ R[r_W]   | Rrs_E + Rrt_E $\rightarrow$ AOR | <i>RTL Instr for Reg Read</i>  |
| 4        |                                | AOR $\rightarrow$ R[r_W]        | Rrs_E == Rrt_E $\rightarrow$ Z<br>PC_E + Off_E $\times$ 4 $\rightarrow$ Target |
| 5        |                                |                                 | if (Z) Target $\rightarrow$ PC   |

Table 9.13: Overlapped Execution of RTL Routines for Interpreting Three MIPS-0 ML Instructions in the Pipelined Data Path of Figure 9.11

clock cycle, the control unit has to generate appropriate microinstructions for interpreting the ML instructions that are present in each pipeline stage. This calls for some major changes in the control unit, perhaps even more than what was required for the data path. A simple approach is to generate at instruction decode time all of the microinstructions required to interpret that instruction in the subsequent clock cycles (in different pipeline stages). These microinstructions are passed on to the pipelined data path, which carries them along with the corresponding instruction by means of extended pipeline registers. Then, in each pipeline stage, the appropriate microinstruction is executed by pulling it out of its pipeline register.

An alternate approach is for each pipeline stage to have its own control unit, so to speak. Depending on the opcode of the instruction that is currently occupying pipeline stage  $i$ , the  $i^{\text{th}}$  control unit generates the appropriate microinstruction and passes it to stage  $i$ . Designing a control unit for a pipelined data path requires sophisticated techniques which are beyond the scope of this book.

## 9.8 Concluding Remarks

## 9.9 Exercises

1. Consider a non-MIPS ISA that has variable length instructions. One of the instructions in this ISA is ADD (ADDR), which means add the contents of memory location ADDR to ACC register. This instruction has the following encoding; notice that it occupies two memory locations.

|                        |  |  |  |  |
|------------------------|--|--|--|--|
| Bit Pattern<br>for ADD |  |  | Bit Pattern<br>for mem dir<br>addressing |  |
| Bit Pattern for ADDR   |  |  |  |  |

Specify an RTL routine to carry out the interpretation of this instruction in the data path given in Figure 9.12.

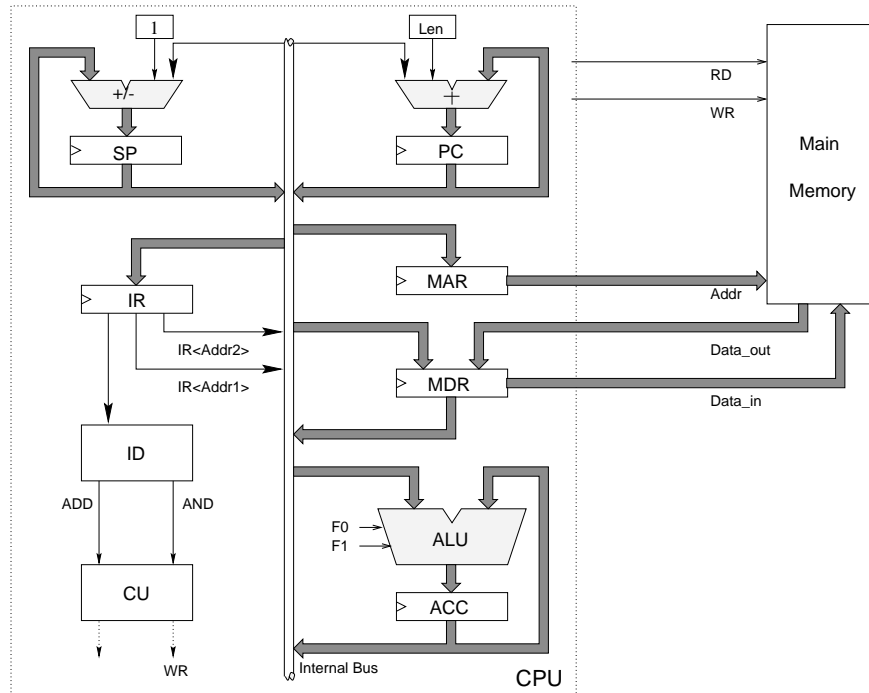


Figure 9.12: A Single-Bus based Processor Data Path for Interpreting an ACC based ISA

2. Consider the non-MIPS instruction ADD (ADDR1), (ADDR2), where ADDR1 and ADDR2 are memory locations whose contents needed to be added. The result is to be stored in memory location ADDR1. This instruction has the following encoding; notice that



addresses ADDR1 and ADDR2 are specified in the words subsequent to the word that stores the ADD opcode.

|                        |  |  |  |  |               |
|------------------------|--|--|--|--|---------------|
| Bit Pattern<br>for ADD | Bit Pattern<br>for mem dir<br>addressing |  | Bit Pattern<br>for mem dir<br>addressing |  | <i>Word 0</i> |
| Bit Pattern for ADDR1  |  |  |  |  | <i>Word 1</i> |
| Bit Pattern for ADDR2  |  |  |  |  | <i>Word 2</i> |

Specify the data transfer operations required to fetch and execute this instruction in the data path given in Figure 9.12. Notice that register ACC is a part of the ISA, and therefore needs to retain the value that it had before interpreting the current instruction. You are allowed to grow the stack in the direction of lower memory addresses.

3. Explain how adding multiple CPU internal buses can help improve performance.
4. Explain with the help of diagram(s) how pipelining the processor data path helps improve performance.
5. Consider a very small direct mapped cache with a total of 4 block frames and a block size of 256 bytes. Assume that the cache is initially empty. The CPU accesses the following memory locations, in that order: c881H, 7742H, 79c3H, c003H, 7842H, c803H, 7181H, 7381H, 7703H, 7745H. All addresses are byte addresses. To get partial credit, show clearly what happens on each access.
  - (a) For each memory reference, indicate the outcome of the reference, either “hit” or “miss”.
  - (b) What are the final contents of the cache? That is, for each cache set and each block frame within a set, indicate if the block frame is empty or occupied, and if occupied, indicate the tag of the memory block that is currently present.

## Chapter 10

# Logic-Level Architecture

*All hard work brings a profit, but mere talk leads only to poverty.*

**Proverbs 14: 23**

This chapter concerns the logic-level architecture of computers. This is the lowest level that we discuss in detail in this book. The objective of the logic-level architecture is to implement the microarchitecture using logic-level circuits. We will go about doing this by taking each major block of the microarchitecture, and implementing it using appropriate logic-level circuitry. The principle of abstraction, which is fundamental to the success of computer science in designing programming languages that support million-line programs, also operates in designing and understanding the logic-level architecture. For example, the logic design of the register file can proceed with little or no concern for the logic design of the ALU. We will give special importance to speed and power considerations when discussing different design alternatives. Finally, whereas the classical logic designer sees NAND gates, NOR gates, and D flip-flops, the computer logic designer sees higher-level blocks such as multiplexers, decoders, and register files. This is because logic-level design is carried out using logic design and minimization tools such as Espresso, MIS, and the like. This has the salutary effect of reducing complexity by abstracting away the gates and flip-flops, and replacing them with black boxes that have only input constraints and output behavior.

### 10.1 Overview

Logic-level circuitry can be classified into combinational circuits and sequential circuits. While the outputs of a combinational circuit are dependent solely on its current set of inputs (assuming that sufficient time has elapsed since the application of the inputs), the

outputs of a sequential circuit depend on its current inputs as well as its past sequence of inputs.

Three methods are commonly used for describing the functionality of a combinational logic circuit. We list them below in decreasing levels of abstraction:

- Truth Table
- Boolean Algebraic Expression
- Logic Diagram

**Truth Table:** A truth table indicates a Boolean function's value for all possible combination of input values. It is an abstract representation because it only specifies *what* the combinational logic does, and not exactly *how* it does it. Table 10.1 gives the truth table for the EXOR function of 3 variables  $\{X, Y, Z\}$ .

| Inputs |     |     | Output |
|--------|-----|-----|--------|
| $X$    | $Y$ | $Z$ | $E$    |
| 0      | 0   | 0   | 0      |
| 0      | 0   | 1   | 1      |
| 0      | 1   | 0   | 1      |
| 0      | 1   | 1   | 0      |
| 1      | 0   | 0   | 1      |
| 1      | 0   | 1   | 0      |
| 1      | 1   | 0   | 0      |
| 1      | 1   | 1   | 1      |

Table 10.1: Truth Table for the EXOR of 3 Variables

**Boolean Algebraic Expression:** Boolean algebra<sup>1</sup> deals with binary variables and logic operations. A Boolean algebraic expression is an implementation of a truth table, and is therefore less abstract than the truth table. It specifies not only *what* the output values of a combinational logic circuit are, but also *how* the outputs are generated. Just like ordinary algebra has four basic operations (addition, subtraction, multiplication, and division), Boolean algebra has three basic operations—AND, OR, and NOT (complement). The standard notation for these operations are “.”, “+”, and “-” (or “”), respectively. As in the case with multiplication, the notation for the AND operation is often omitted, if the context guarantees that there is no ambiguity. The truth table given in Table 10.1 can be implemented by different Boolean expressions, one of which is  $E = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ$ .

---

<sup>1</sup>Boolean algebra was developed by George Boole.

**Logic Diagram:** A Boolean algebraic expression can be directly implemented using a set of logic gates that are interconnected in an appropriate manner. A logic diagram represents the gates and the interconnections in a pictorial manner<sup>2</sup>. This representation corresponds more closely to the hardware because the gate interconnections represent physical wires that connect physical gates in the hardware. Any Boolean function can be implemented with a suitable combination of {AND, OR, NOT} gates. These three gate types therefore form a *universal set*. In addition to these basic gates, other gates such as NAND, NOR, and XOR are also frequently used. NAND gates (or NOR gates) by themselves form a universal set, as any Boolean function can be implemented using these gates alone.

We use all of the above three methods in describing logic-level circuits. In addition, for each method, we often use a hierarchical approach to hide some of the details which are very obvious. Such an approach helps the student to understand the important concepts, without getting bogged down in the irrelevant details.

### 10.1.1 Multiplexers

A multiplexer is a combinational logic block that selects one of several data inputs to be routed to a single output. The selection of the input is done based on a set of control inputs. The control inputs are generally supplied in binary encoded form. Thus, a multiplexer with  $n$  control input lines can support up to  $2^n$  data inputs. Figure 10.1 shows a truth table, Boolean expression, logic diagram, and logic symbol for a 1-bit 4-input multiplexer.  $D0$  through  $D3$  are the data input lines;  $S0$  and  $S1$  form the selection input lines; and  $M$  forms the sole output line. In the rest of the chapter, we always represent decoders by their logic symbol only.

Because this block has a total of 6 inputs, a complete truth table would require  $2^6 = 64$  rows. Therefore, we have drawn the truth table in a condensed manner, by not listing the columns for the data inputs, and listing only the columns for the selection inputs. In addition, the output column does not list a binary value, but instead the data input that is selected for each combination of selection input values. For instance, the first row in the truth table indicates that the output will be the same as that of the input value  $D0$  when the selection inputs are  $S0 = 0$  and  $S1 = 0$ . That is,  $M = D0$  when  $S0 = 0$  and  $S1 = 0$ .

The inverters and AND gates in the logic diagram essentially decode the two selection input lines into four lines. Each of the 4 data input lines is ANDed with a decoded line, and the resulting lines are ORed together to form the final output line.

Most of the multiplexers used in computer data paths are multi-bit multiplexers in which each of the data inputs and the single output are multiple bits wide. A multi-bit multiplexer is quite useful whenever we need to combine mutually exclusive input sets. For instance, in the data path of Figure 7.7, there are two mutually exclusive inputs feeding to the MDR

---

<sup>2</sup>CAD tools usually represent the logic diagram in a non-pictorial form for ease of representation and manipulation.

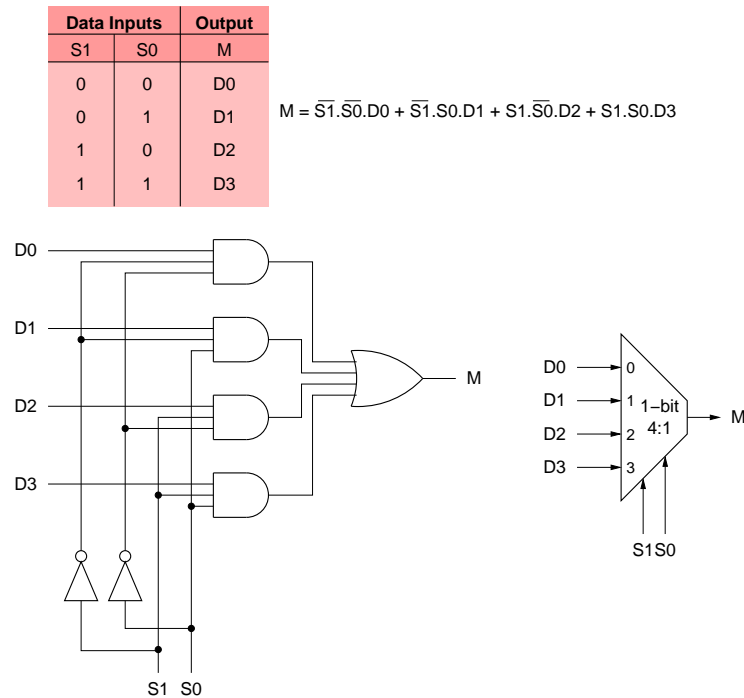


Figure 10.1: Truth Table, Boolean Expression, Logic Diagram, and Logic Symbol of a 1-bit 4-Input Multiplexer

register. At the logic level, we implement this using a multi-bit 2:1 multiplexer, as illustrated in Figure 10.2. This multiplexer is controlled by a *control signal* called *MDR\_select*.

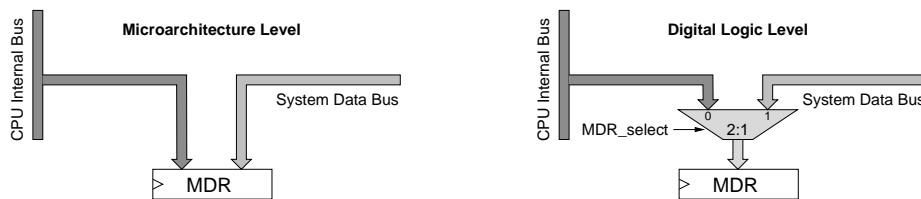


Figure 10.2: Use of a MUX at the Digital Logic Level for Selecting the Input to MDR

### 10.1.2 Decoders

Another commonly used combinational block in a computer data path is a decoder. This block is used because information is frequently stored and transmitted in the data path in binary encoded form. A decoder is a combinational logic block that accepts a binary coded input and activates one of several output lines based on the encoded value that appears

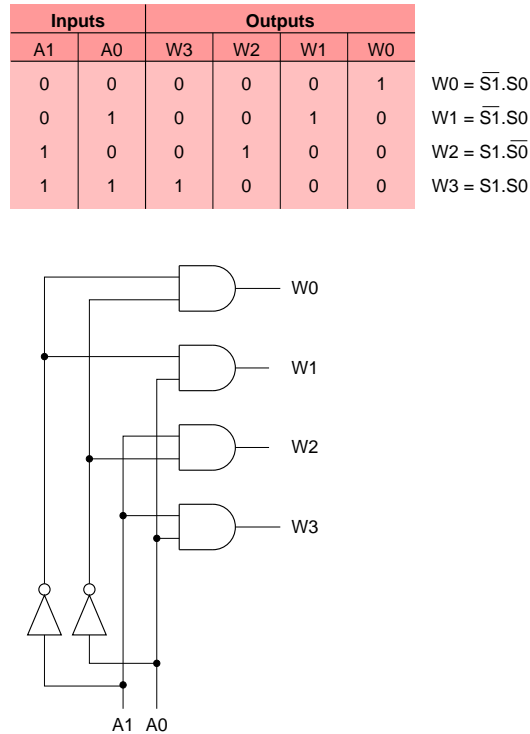


Figure 10.3: Truth Table, Boolean Expression, Logic Diagram, and Logic Symbol of a 2-Input Decoder

at the inputs. Thus, a decoder with  $n$  input lines can have up to  $2^n$  output lines. Figure 10.3 shows a truth table, Boolean expression, logic diagram, and logic symbol for a 2-input decoder.  $A1$  and  $A0$  are the input lines, and  $W0$  through  $W1$  are the output lines. In the rest of the chapter, we represent decoders by their logic symbol only.

One of the places in a data path where a decoder is frequently used is for selecting a specific register in a register file, as illustrated in Figure 10.4. The left hand side of the figure shows a portion of the microarchitecture-level data path. The right hand side shows the same portion of the data path at the digital logic level. In this part of the figure, a 5-bit 4:1 MUX is used to select the register address to be used. We use 2 control signals, collectively named **R\_select**, as the control inputs to this MUX. The register address at the output of the MUX is in 5-bit binary encoded form, and the 5:32 decoder decodes the address so as to access the addressed register in the register file.

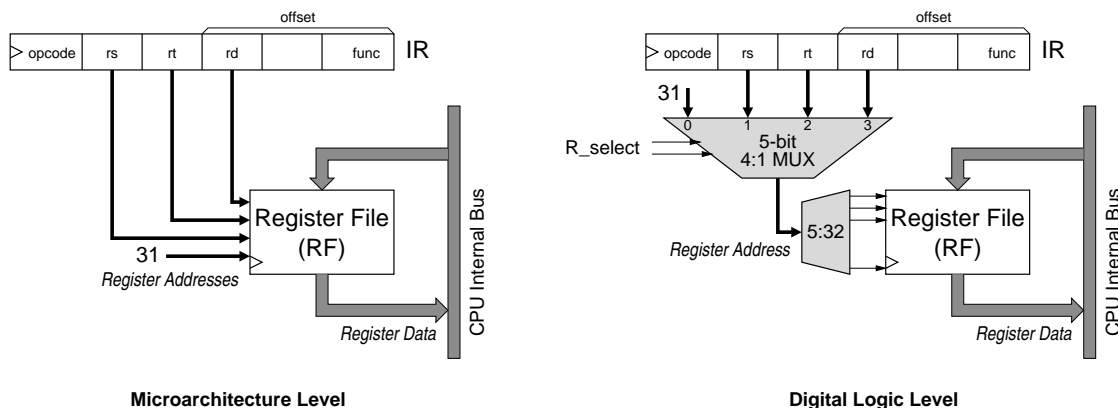


Figure 10.4: Use of a Decoder at the Digital Logic Level for Selecting a Register from RF

### 10.1.3 Flip-Flops

Many of the blocks in a computer microarchitecture deal with memory function, i.e., the storing and retrieval of information. Semiconductor memory elements are used in different blocks, such as register file, cache memory, and main memory. In this section, we discuss the basic building block of semiconductor memories—flip-flops.

A semiconductor memory system is made up of many *cells*—each of which can store a single bit of information—together with associated circuits to transfer information into and out of the cells. Thus, for a physical device to be useful as a memory cell, it must have (i) two stable states, which can be used to represent binary 0 and 1, (ii) a reliable mechanism to write a 0 and a 1, and (iii) a reliable mechanism to read the current state. In the past, memories have been built of a variety of devices that match this characteristic, including relays, vacuum tubes, storage tubes, and delay lines. In each case, information in the form of bits was entered into the memory, and then at some later time extracted for use.

Nowadays, we use semiconductor memory, which is both fast and small in size. Semiconductor memories are available in a wide range of speeds. Their cycle times range from a few hundred nanoseconds to less than 10 nanoseconds. When first introduced in the 1960s, they were much more expensive than the prevalent doughnut-shaped, magnetic-core memories. However, thanks to subsequent advances in VLSI, the cost of semiconductor memories dropped substantially. Currently they are used exclusively for implementing cache memories and main memories.

Two types of semiconductor memories are used in various parts of a computer: *random access memory (RAM)* and *read-only memory (ROM)*. Although their names seem to indicate that only RAM is randomly accessible, this is not true; In reality, both types are randomly accessible! RAM is volatile and loses stored information when power is switched off. ROM, on the other hand, is non-volatile, and retains stored information even when

power is switched off. It is typically used to implement portions of the main memory that contain fixed programs or data. Many monitors and keyboards contain ROMs to hold programs to be run on internal processors that control the display and the keyboard. Appliances such as video game cartridges, microwave ovens, and automobile fuel injection controllers also often use ROMs to store the programs to be executed on their processors. A different application for ROMs in computers is in the design of the control store discussed later in this chapter.

#### 10.1.4 Static RAM

Semiconductor RAMs come in 2 kind—*static RAM (SRAM)* and *dynamic RAM (DRAM)*. RAMs that consist of circuits that are capable of retaining their state as long as power is applied are known as static RAMs. Figure 10.5 illustrates how an SRAM cell may be implemented. This is one of the few instances in this textbook where we descend below the logic gate level in order to explain the structure of a circuit. Two inverters are cross-connected to form a latch. The latch is connected to 2 bit lines by transistors  $T_1$  and  $T_2$ . These transistors act as switches that can be opened or closed under control of the word line. When the word line is at the ground level, the transistors are turned off, and the latch retains its state. For example, let us assume that the cell is in state 1 if the logic value at point  $X$  is 1 and at point  $Y$  is 0. This state is maintained as long as the signal on the word line is at ground level.

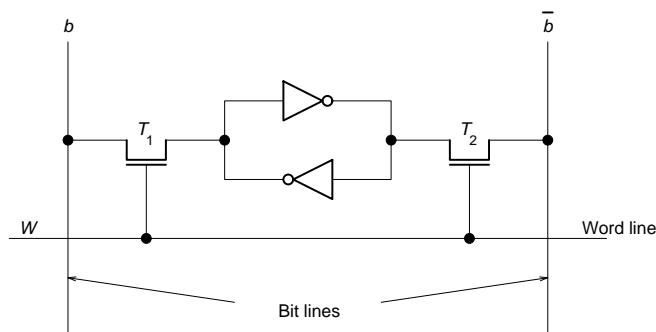


Figure 10.5: A Static RAM Cell

In order to read the state of the SRAM cell, the word line is activated to close switches  $T_1$  and  $T_2$ . If the cell is in state 1, the signal on bit line  $b$  is high and the signal on bit line  $\bar{b}$  is low. The opposite is true if the cell is in state 0. Thus,  $b$  and  $\bar{b}$  are complements of each other. Sense/Write circuits at the end of the bit lines monitor the state of  $b$  and  $\bar{b}$  and set the output accordingly.

A value is written to the SRAM cell by placing the appropriate value on bit line  $b$  and its complement on  $\bar{b}$ , and then activating the word line. This forces the cell into the



corresponding state. The required signals on the bit lines are generated by the Sense/Write circuit.

### 10.1.5 Dynamic RAM

Static RAMs are fast, but they come at a high cost because their cells require several transistors. Less expensive RAMs can be implemented if simpler cells are used. However, such cells do not retain their state indefinitely; hence, they are called *dynamic RAMs* (*DRAMs*). A DRAM cell stores information in the form of a charge on a small capacitor. This capacitor tends to discharge within a few milliseconds, thereby losing the value stored in the cell. Therefore, the contents of a DRAM cell must be periodically refreshed by restoring the capacitor charge to the required value.

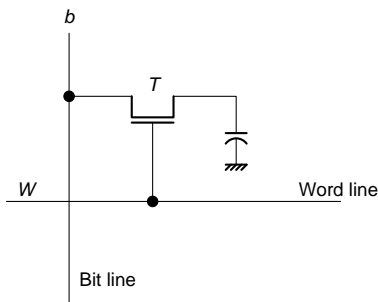


Figure 10.6: A Single-Transistor DRAM Cell

An example of a DRAM cell that consists of a capacitor and a transistor is shown above. In order to store information in this cell, the transistor is turned on by activating the word line, and an appropriate voltage is applied on the bit line. This causes a known amount of charge to be stored on the capacitor. After the transistor is turned off, the capacitor begins to discharge due to its own leakage resistance and due to the transistor continuing to conduct a tiny amount of current, measured in picoamperes, after it is turned off. Hence, the information stored in the cell can be retrieved correctly only if it is read before the charge on the capacitor drops below some threshold value. To ensure that the contents of a DRAM are maintained, the DRAM cells must be accessed periodically, typically once every 2 to 16 milliseconds. A **refresh circuit** is usually included to perform this function.

During a Read operation, the bit line is placed in a high-impedance state, and the transistor is turned on. A sense circuit connected to the bit line determines if the charge on the capacitor is above or below the threshold value. Because this charge is too small, the Read operation discharges the capacitor in the cell that is being accessed. In order to retain the information stored in the cell, DRAM includes special circuitry that writes back the value that has been read.

Because of their high density and low cost, DRAMs are widely used in the main memory

units of computers. Available chips range from 1 Kbits to 16 Mbits, and even larger chips are being developed.

### 10.1.6 Tri-State Buffers

For the logic gates and circuits we saw so far, the output(s) can have only one of two possible logic values: 0 and 1. A tri-state buffer deviates from this binary behavior. Its output can have 3 logic values: {0, 1, Z}. The first two values have the same meaning as before. The third value, called a *high impedance* state, represents a situation in which the output behaves like an open circuit, or electrically disconnected. Figure 10.7 gives the truth table and logic symbol for a tri-state buffer.

| Inputs |   | Output |
|--------|---|--------|
| E      | I | O      |
| 0      | 0 | Z      |
| 0      | 1 | Z      |
| 1      | 0 | 0      |
| 1      | 1 | 1      |

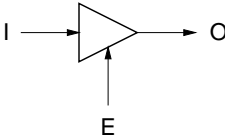


Figure 10.7: Truth Table and Logic Symbol of a 1-bit Tri-State Buffer

Tri-state buffers are particularly useful for implementing the connections to a shared bus. Figure 10.8 illustrates how multi-bit tri-state buffers can be used in a processor data path to connect the outputs of two registers and a sign-extend unit to the **processor bus**. Three control signals, **PC\_out**, **SE\_out**, and **AOR\_out**, control these tri-state buffers. By activating any one of these control signals in a clock cycle, we can transfer the contents/outputs of the appropriate block to the bus. It is important that at most one of these control signals is activated in a cycle.

## 10.2 Implementing ALU and Functional Units of Data Path

We have seen the basic elements that are available to the logic level designer, such as gates, multiplexers, decoders, flip-flops, and memory cells. Our next step is to work towards a logic-level data path, by implementing the major blocks of a microarchitecture-level data path, using these basic elements. We specifically look at the ALU, the register file, and the memory system. This section deals with the ALU, which some call the brawn of a computer. In a machine language program, the basic operations performed on data are arithmetic operations (ADD, SUB, MULT, DIV) and logical operations (AND, OR, NOT, etc). As we saw in Chapter 7, all of these operations are performed in the ALU. The ALU is typically built as a collection of different hardware units that can do various arithmetic/logic

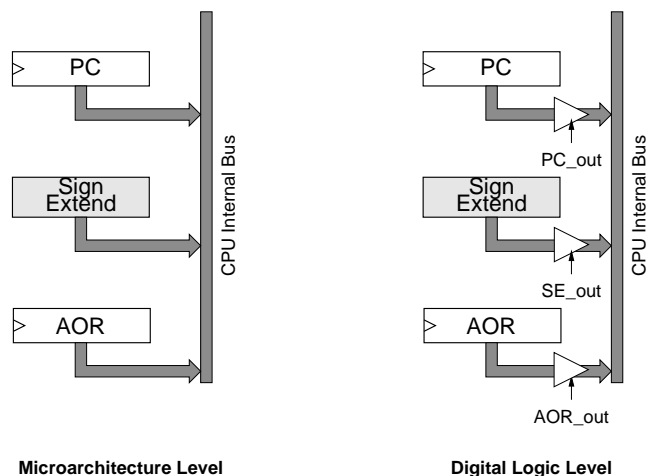


Figure 10.8: Use of Tri-State Buffers at the Digital Logic Level for Connecting Multiple Lines to Processor Bus

operations in the particular number systems selected for use in that particular instruction set architecture (ISA).

At the digital logic level, the ALU is generally designed as a combinational circuit. Its standard logic symbol is as shown below. The output of the ALU is an  $N$ -bit binary number,  $A = (A_{N-1}...A_0)_2$ , which is the result produced by performing some arithmetic or logic operation on two  $N$ -bit binary numbers, or operands,  $X = (X_{N-1}...X_0)_2$  and  $Y = (Y_{N-1}...Y_0)_2$ . Besides data inputs, an ALU must have control inputs to specify the function to be performed. The operation to be performed is determined by a  $k$ -bit selection code  $F = (F_{k-1}...F_0)_2$ , such that the number of possible ALU operations is  $2^k$ .

The precise algorithms to be implemented in an ALU depend on the representation of different data types: the algorithm for adding integers will be different for the sign-magnitude number system and the 2's complement number system. As discussed in Chapter 5, the compelling arguments for choosing one representation over others revolve around the relative efficiency in implementing basic arithmetic operations for each representation.

### 10.2.1 Implementing an Integer Adder

We shall begin ALU design by designing an adder circuit for adding integers. Integers come in two kinds—unsigned and signed. When these two data types are represented using the binary number system and the 2's complement number system, respectively, the algorithm for performing addition is the same for both data types; the only difference is in overflow detection, as discussed in Section 10.2.3. Thus, both types of addition can be performed using the same hardware with some minor modifications. Without loss of generality, we

therefore use in our discussion the term *adder*, which can be applied for unsigned integer addition as well as signed integer addition.

Addition can be considered as a *universal arithmetic operation* because all of the various arithmetic operations — add, subtract, multiply, and divide — can be implemented by appropriate combinations of the add function. An ALU that can perform addition and negation can easily do subtraction, and can be enhanced to do multiplication and division as a series of additions and subtractions. The ALU must implement algorithms to perform these operations, algorithms that are similar to the “paper-and-pencil” methods used to do the operations.

### 1-Bit Full Adder (FA)

In the above examples, we saw that at each bit position we will be adding two data bits and one carry bit. A basic cell that can be used to perform this addition (for a single bit position) is the *full adder*.

Let us find out the *worst-case delay* of this full adder. To find the worst-case delay, we identify a *critical path* in the circuit, and count the number of gates along the *critical path*. (A critical path is one of the longest path from any input to any output in the circuit.) When the full adder is built in this manner, it has a worst-case delay of 2 gate delays.

### Ripple Carry Adder (RCA)

A simple way of building an  $N$ -bit adder is to connect  $N$  full adders in cascade, with the carry output from one full adder connected to the carry input of the next full adder. Figure 10.9 shows the block diagram of a 4-bit integer adder designed in this manner. Bits are numbered 0-3, with 0 being the least significant bit and 3 being the most significant bit. The carry in ( $C_0$ ) input may come from a *condition code* or some other source. The carry out ( $C_4$ ) output from the addition is usually used to set the C flag. Because the carries are connected in a chain, a carry propagating from the least significant bit to the most significant bit will cause a “rippling effect”. Hence this type of adder is called a ripple carry adder.

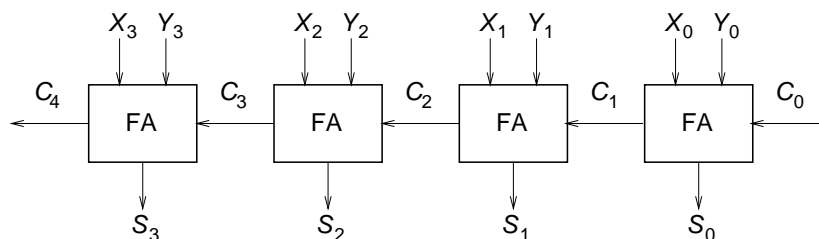


Figure 10.9: Block Diagram of a 4-bit Ripple Carry Adder

A ripple carry adder is *slow*, because in the worst case a carry must propagate from the least significant full adder to the most significant one. This occurs if one of the addends is 11...11 and the other is 00...01. This “rippling” of the carry determines the adder’s worst-case delay. The worst-case time to do an  $N$ -bit addition with the RCA is given by

$$T_{RCA} = 2N \text{ gate delays}$$

assuming that all gates have the same delay. Therefore, the time required is *linear* in the number of bits to be added, and limits the use of ripple-carry addition to small values of  $N$ .

### Look-ahead Carry Adder (LCA)

Fast addition is a very desirable feature of an ALU. Indeed, the clock cycle time of a processor is often determined by the time required to perform a single  $N$ -bit addition. This means that, for a processor operating at a clock rate of 1 GHz, an addition operation must be completed well within  $\frac{1}{1\text{GHz}}$  or 1 ns. Even if each logic gate has only a 0.1 ns delay, an RCA still takes 6.4 ns to do a 32-bit addition!

To do the addition well within 1 ns, the adder must have relatively few levels of logic or, equivalently, have a short critical path. From Boolean algebra theory, we know that any combinational logic circuit can be built with just 2 levels of logic by expressing its outputs

in the sum-of-products form. That is, all the inputs would be applied simultaneously and propagate through two levels of logic to produce the result. However, even for small values of  $N$ , such a design is impractical, because (i) it requires far too many gates, and (ii) it requires excessive fan-in as well as fan-out. Therefore, to design easily realizable fast adders, we must settle for worst-case delays that are higher than the minimum number of gate delays possible theoretically.

Fast addition with reasonable amounts of hardware requires a method of generating carry signals that is intermediate in complexity between the very fast but costly sum-of-products approach and the slow but inexpensive ripple-carry technique. The most widely used scheme of this sort, called *look-ahead carry* technique, employs special carry circuits to generate each  $C_i$  in a small number of logic levels (for reasonable values of  $i$ ), as shown in Figure 10.10. Notice that there is no rippling of carries, and that  $C_i$  is determined directly from  $X_{i-1}$ ,  $Y_{i-1}$ , ...,  $X_0$ ,  $Y_0$ , and  $C_0$ .

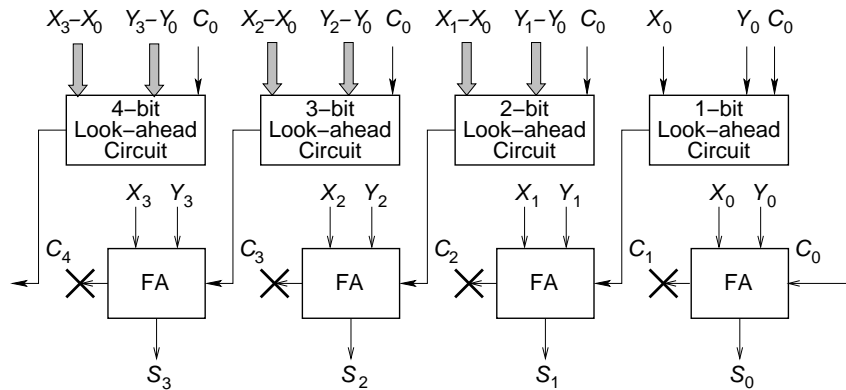


Figure 10.10: Incorporating the Look-ahead Carry Technique in a 4-bit Adder

### Systematic Design of the Look-ahead Carry Generator

Each of the look-ahead carry circuits can be designed in sum-of-products form. But that entails significant amount of logic as well as complexity. Instead, we do a systematic design that shares logic between the different look-ahead circuits. Two definitions are key to a systematic design of the look-ahead carry logic:

- **Carry generate:** This function indicates if a particular adder stage  $i$  generates a carry, irrespective of the carry input  $C_i$  to that stage.
- **Carry propagate:** This function indicates if a particular adder stage  $i$  can pass on a carry to the next stage if supplied with a carry.

Corresponding to these definitions, we can write Boolean equations for the carry generate signal,  $G_i$ , and the carry propagate signal,  $P_i$ , for each stage of a look-ahead carry adder:

$$\begin{aligned} G_i &= X_i Y_i \\ P_i &= X_i + Y_i \end{aligned}$$

Thus, a stage unconditionally generates a carry if both of its addend bits are 1. Similarly, a stage propagates a carry supplied to it if at least one of its addend bits is 1. A stage's carry output can now be written in terms of its generate and propagate signals:

$$C_{i+1} = G_i + P_i C_i$$

To eliminate carry ripple, we recursively expand the  $C_i$  term for each stage to obtain a 2-level AND-OR expression. After expansion, we obtain the following Boolean equations for the carries out of the first 4 stages of the adder:

$$\begin{aligned} C_1 &= G_0 + C_0 P_0 \\ C_2 &= G_1 + G_0 P_1 + C_0 P_0 P_1 \\ C_3 &= G_2 + G_1 P_2 + G_0 P_1 P_2 + C_0 P_0 P_1 P_2 \\ C_4 &= G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + C_0 P_0 P_1 P_2 P_3 \end{aligned}$$

Each carry equation corresponds to a circuit with just 3 levels of gate delays—one for the carry generate and carry propagate signals, and two for the sum-of-products shown. These equations are significantly simpler than the corresponding sum-of-products expression for each  $C_i$  in terms of the primary inputs  $X$  and  $Y$ .

A *look-ahead carry adder (LCA)* implements  $N$ -bit binary addition using the foregoing carry look-ahead technique. It consists of 3 principal subcircuits:

- Circuit to produce the  $N$  carry generate and  $N$  carry propagate signals.
- A two-level carry look-ahead generator that implements the  $N$  carry functions  $C_1, C_2, \dots, C_N$ .
- A summation circuit that produces the  $N$  sum bits from the primary inputs and the intermediate carry signals  $C_0, C_1, \dots, C_{N-1}$ , using the Boolean equation  $S_i = X_i \oplus Y_i \oplus C_i$ .

Realizing these in the most direct fashion gives the logic circuit shown in next page. The total number of gates is 26; the number of logic levels is 4. A comparable 4-bit RCA contains only 20 gates, but has 8 levels of logic. This illustrates a typical trade-off between *hardware complexity* and *operation speed*.

Example: Let us add two 4-bit binary numbers, 1101 and 0011, using the 4-bit LCA technique. First, we calculate the  $G_i$ s and  $P_i$ s. Then we calculate the  $C_i$ s, and finally the  $S_i$ s.

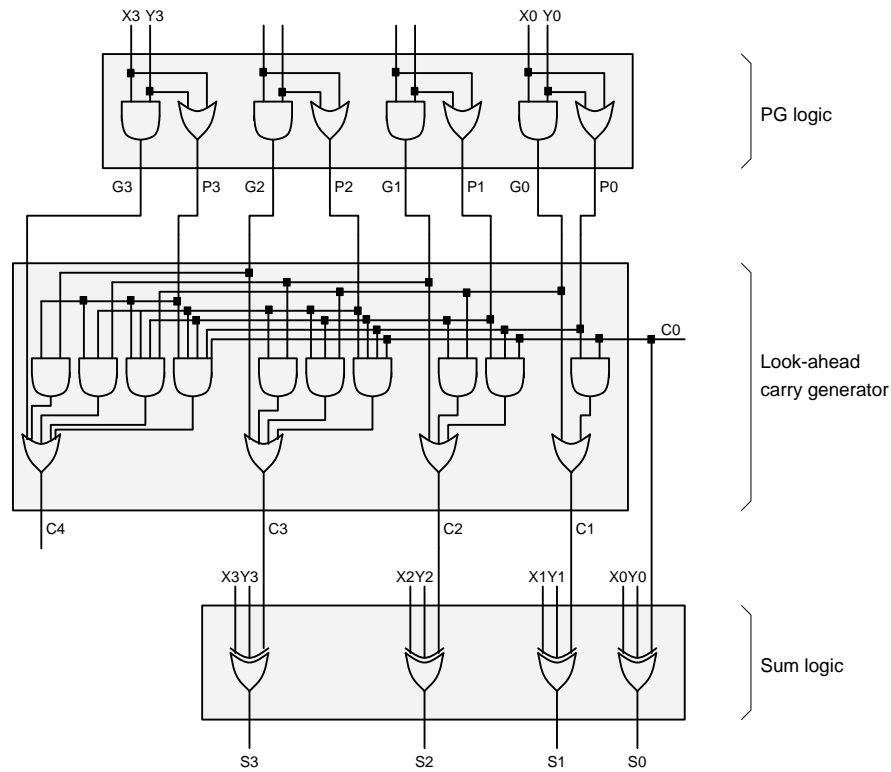


Figure 10.11: Logic Diagram of a 4-bit LCA

| Bit number ( $i$ )                |   | 3 | 2 | 1 | 0 |
|-----------------------------------|---|---|---|---|---|
| $X_i$                             |   | 1 | 1 | 0 | 1 |
| $Y_i$                             |   | 0 | 0 | 1 | 1 |
| $C_0$                             |   |   |   |   | 0 |
| $G_i = X_i Y_i$                   |   | 0 | 0 | 0 | 1 |
| $P_i = X_i + Y_i$                 |   | 1 | 1 | 1 | 1 |
| $C_i$                             | 1 | 1 | 1 | 1 |   |
| $S_i = X_i \oplus Y_i \oplus C_i$ |   | 0 | 0 | 0 | 0 |

Let us find the worst-case delay of this adder. To do this, we need to first identify a critical path (i.e., one of the longest paths from an input to an output). One such path proceeds from input  $X_0$  to output  $S_3$ , and has 4 gates along this path: an AND gate (or an OR gate), an AND gate, an OR gate, and an EXOR gate. Thus, the worst-case delay of this adder is 4 gate delays.



### Group-Ripple Adder

A practical problem with using this approach for large values of  $N$  is the gate fan-in constraints. The  $i^{\text{th}}$  carry signal  $C_i$  is the logical OR of  $i + 1$  product terms, the most complex of which has  $i + 1$  literals. This places a practical limit on the number of bits across which carry look-ahead can be usefully employed (due to fan-in limitations). A typical limit is 4 bits. The natural question to ask is: how can we apply look-ahead carry techniques to 8-bit adders and 16-bit adders?

We can take one of two approaches. The first is to design a *group-ripple adder* by cascading multiple 4-bit LCAs as shown in Figure 10.12. Notice that the carries inside each block are formed by look-ahead circuits. However, the carries from each block still ripple between blocks.

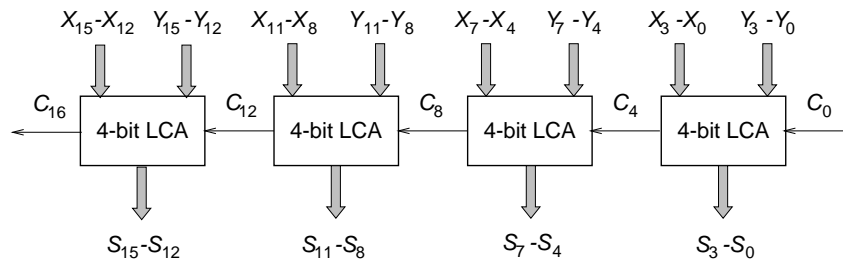


Figure 10.12: Block Diagram of a 16-bit Group Ripple Adder

Let us find the worst-case delay of this adder. One of the critical paths proceeds from input  $X_0$  to output  $S_{15}$ , producing carries  $C_4$ ,  $C_8$ , and  $C_{12}$  along the way. Generation of  $C_4$  takes 3 gate delays. Once  $C_4$  is available, *only two additional gate delays are required to produce  $C_8$* . Similarly, two more gate delays are needed to produce  $C_{12}$ . After generating  $C_{12}$ , two more gate delays are required to produce  $C_{15}$ , and an EXOR operation is needed afterwards to generate  $S_{15}$ . Thus, the worst-case delay is 10 gate delays.

### Hierarchical Look-ahead Carry Adder

For longer word lengths, it may be necessary to speed up the addition operation even further. This can be done by applying the look-ahead technique to the carry signals between the blocks; thus, a second level of look-ahead is employed, as shown in Figure 10.13.

We can design a 16-bit fast adder as follows. Suppose that each of the 4-bit adder blocks provides two new output functions called *group carry generate* and *group carry propagate*, which are defined as follows:

$$G_{i-j} = G_j + G_{j-1}P_j + G_{j-2}P_{j-1}P_j + \dots + G_iP_{i+1}P_{i+2}\dots P_j$$

$$P_{i-j} = P_iP_{i+1}\dots P_{j-1}P_j$$

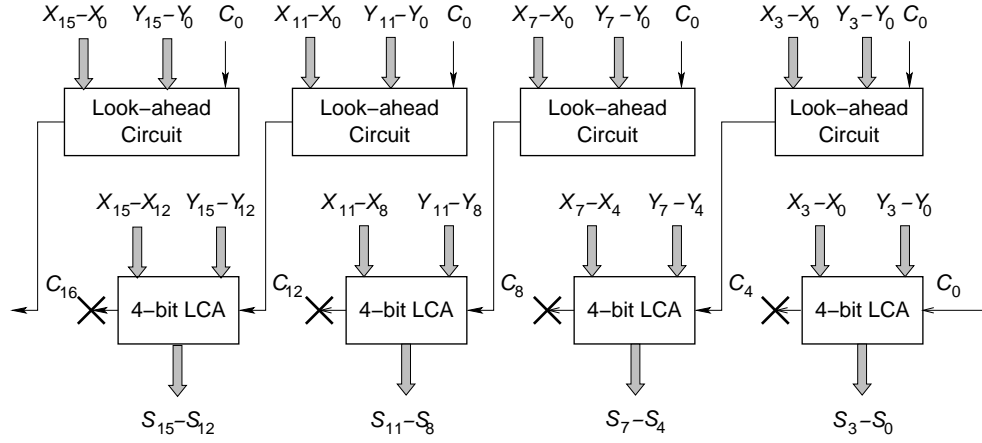


Figure 10.13: Incorporating Look-ahead Carry Logic in a Hierarchical Manner in a 16-bit Adder

In words, if we say that  $G_i$  and  $P_i$  determine if bit stage  $i$  generates or propagates a carry, then  $G_{i-j}$  and  $P_{i-j}$  determine if block  $i-j$  generates or propagates a carry. With these new functions available, it is not necessary to wait for carries to ripple between all of the 4-bit blocks. They can instead be calculated quickly as follows:

$$C_4 = G_{0-3} + C_0 P_{0-3}$$

$$C_8 = G_{4-7} + G_{0-3} P_{4-7} + C_0 P_{0-3} P_{4-7}$$

$$C_{12} = G_{8-11} + G_{4-7} P_{8-11} + G_{0-3} P_{4-7} P_{8-11} + C_0 P_{0-3} P_{4-7} P_{8-11}$$

$$C_{16} = G_{12-15} + G_{8-11} P_{12-15} + G_{4-7} P_{8-11} P_{12-15} + G_{0-3} P_{4-7} P_{8-11} P_{12-15} + C_0 P_{0-3} P_{4-7} P_{8-11} P_{12-15}$$

Realizing these equations in the most direct fashion gives the logic circuit shown in Figure 10.14. The newly added portion, identified in the figure, corresponds to the circuitry for generating the group carry generates and propagates, and that for producing the intermediate carries  $\{C_4, C_8, C_{12}, C_{16}\}$ .

Let us find the worst-case delay of this adder. One of the critical paths proceeds from input  $X_0$  to output  $S_{15}$ . Along this path,  $G_{0-3}$  is produced after 3 gate delays, as shown in the figure. Once  $G_{0-3}$  is produced, it requires two more gate delays to produce the intermediate carry  $C_{12}$ , as shown in the figure. Once  $C_{12}$  is produced, it takes two additional gate delays to produce  $C_{15}$ , and one more gate delay to produce  $S_{15}$ . Thus, the total gate delay is 8.

Example: To get these ideas into concrete form, let us add two 16-bit signed integers, 0111 0110 1110 1101 and 0011 1100 1011 1110, using the 16-bit hierarchical LCA that we have just designed.

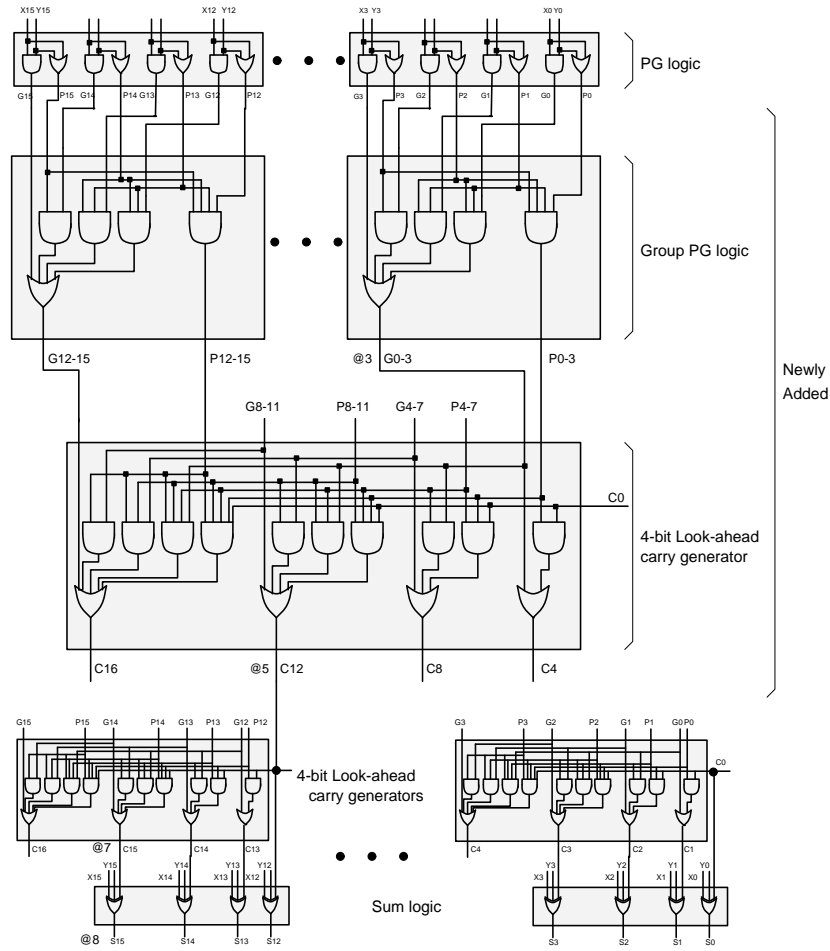


Figure 10.14: Logic Diagram of a 16-bit Hierarchical LCA with 4-bit Look-aheads

| $i$                               | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----------------------------------|---|---|---|---|---|---|---|---|---|---|
| $X_i$                             | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| $Y_i$                             | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| $C_0$                             |   |   |   |   |   |   |   |   |   | 0 |
| $G_i = X_i Y_i$                   | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| $P_i = X_i + Y_i$                 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| $G_{i-j}$                         | 0 |   | 1 |   |   | 1 |   |   | 1 |   |
| $P_{i-j} = P_i P_{i+1} \dots P_j$ | 0 |   | 0 |   |   | 1 |   |   | 1 |   |
| $C_{16}, C_{12}, C_8, C_4$        | 0 |   | 1 |   | 1 |   | 1 |   |   |   |
| $C_i$                             | 1 | 1 | 1 |   | 1 | 0 | 0 |   | 1 | 0 |
| $S_i = X_i \oplus Y_i \oplus C_i$ | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |

### 10.2.2 Implementing an Integer Subtractor

Digital circuits to perform subtraction of binary numbers can be developed in the same manner as that for binary addition. A *full subtractor* module can be designed in a manner similar to the design of a full adder, and these modules can be cascaded to form an  $N$ -bit subtractor. It is also possible to design fast subtractors using the *look-ahead borrow* technique, similar to the look-ahead carry technique that we used for adders.

When designing an ALU that can perform both addition and subtraction, we can save hardware by using the adder to perform subtraction. Recall that  $X - Y$  can be performed by adding  $-Y$  to  $X$ . In the 2's complement number system,  $-Y$  is obtained by taking the 2's complement of  $Y$ , as illustrated in Figure 10.15. This is true irrespective of whether  $Y$  is positive or negative!

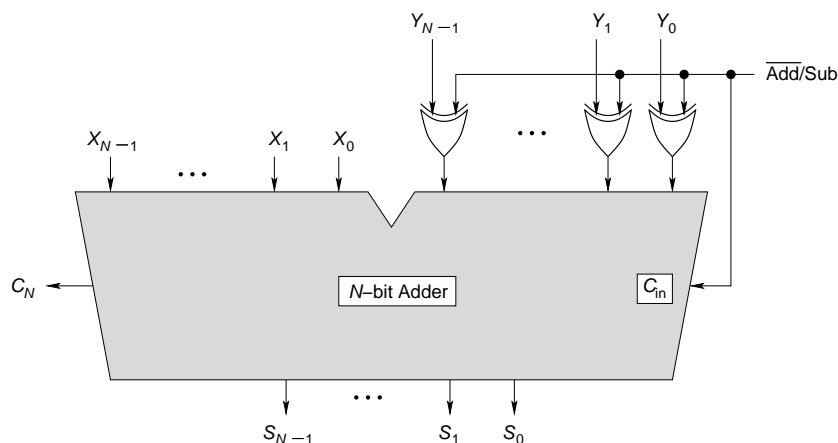
$$\begin{array}{rcl}
 & \text{2's Complement System} & \\
 \begin{array}{r}
 00101101 \quad (45) \\
 - 00111110 \quad (62) \\
 \hline
 \end{array} & \longrightarrow & \begin{array}{r}
 00101101 \quad (45) \\
 + 11000010 \quad (-62) \\
 \hline
 11101111 \quad (-17)
 \end{array}
 \end{array}$$

Figure 10.15: Subtraction of Signed Numbers (2's Complement Number System) Using Addition

We have seen that  $X - Y$  can be performed by doing  $X + (-Y)$ . In the 2's complement number system,  $-Y$  can be formed by taking the 1's complement of  $Y$  and adding 1 to it. When implementing a subtractor in this manner, the 2's complementing can be done along with the addition part, as shown below in Figure 10.16.

The inverters needed for obtaining the complement of the  $Y$  input are realized by  $N$  two-input EXOR gates, with one input of each gate connected to a common control line,  $\overline{\text{Add/Sub}}$ . To perform addition, the  $\overline{\text{Add/Sub}}$  control signal is set to 0 so that the  $Y$  inputs are applied unchanged to one of the adder inputs, along with a carry-in signal ( $C_{\text{in}}$ ) of 0. To perform subtraction, the  $\overline{\text{Add/Sub}}$  control signal is set to 1 so as to complement the  $Y$  input values, i.e., the output of the  $i$ th EXOR is  $Y_i \oplus 1 = \bar{Y}_i$ . The addition of 1 required to complete the 2's complementing process is achieved by supplying a 1 at the  $C_{\text{in}}$  input. Hence this circuit can be switched between the arithmetic operations  $X + Y$  and  $X - Y$ , simply by changing the value of the  $\overline{\text{Add/Sub}}$  control signal.

The hardware simplicity and speed of adding and subtracting signed integers in the 2's complement number system is the reason why this number representation is used in modern computers.

Figure 10.16: Block Diagram of an  $N$ -bit Adder/Subtractor

### 10.2.3 Implementing an Arithmetic Overflow Detector

Computer arithmetic is distinguished from paper-and-pencil arithmetic by the constraints of limited precision. This can cause arithmetic operations to produce numbers that are not representable within the available range. For instance, as discussed in Chapter 5, the range of unsigned integers that can be represented in an  $N$ -bit binary number system is

$$0 \text{ to } 2^N - 1$$

Any arithmetic operation on unsigned integers that attempts to generate a number outside this range is said to generate an *overflow*. Similarly, the range of signed integers that can be represented in an  $N$ -bit binary number system is

$$-2^{N-1} \text{ to } 2^{N-1} - 1$$

Any arithmetic operation on signed integers that attempts to generate a number outside this range generates an overflow. When an overflow occurs, the output bit pattern produced by the arithmetic hardware is not a valid representation of the actual result, in the number system used to specify the input values.

Figure 10.17 shows the addition of three pairs of bit patterns, when they are interpreted as unsigned integers as well as when they are interpreted as signed integers. When the bit patterns are treated as unsigned integers, the first and third pairs produce overflow when added. Consider the first pair of bit patterns, 11000000 and 01000000. When they are treated as unsigned integers, their values are 192 and 64, respectively. The addition of these two bit patterns yields the bit pattern 00000000, which when interpreted as an unsigned integer reads zero, and not 256, the sum of 192 and 64. Thus, an overflow has occurred.

When the same bit patterns are treated as signed integers, the first pair does not produce overflow when added, but the second and third pairs do. It is important to detect overflow occurrences so that invalid results are not used inadvertently. For instance,

| Binary Number System<br>(Unsigned Integers) |  |                |        |                |        |
|---|--|----------------|--------|----------------|--------|
| 11<br>↙ ↘                                   |  |                | 1<br>↙ |                | 1<br>↙ |
| 11000000 (192)                              |  | 01000000 (64)  |        | 10000000 (128) |        |
| 01000000 (64)                               |  | 01000000 (64)  |        | 10000000 (128) |        |
| 1 00000000 (0)                              |  | 10000000 (128) |        | 1 00000000 (0) |        |
| Overflow                                    |  | No overflow    |        | Overflow       |        |

---

| 2's Complement System<br>(Signed Integers) |  |                 |  |                 |  |
|--|--|-----------------|--|-----------------|--|
| 11<br>↙ ↘                                  |  | 1<br>↙          |  | 1<br>↙          |  |
| 11000000 (-64)                             |  | 01000000 (64)   |  | 10000000 (-128) |  |
| 01000000 (64)                              |  | 01000000 (64)   |  | 10000000 (-128) |  |
| 1 00000000 (0)                             |  | 10000000 (-128) |  | 1 00000000 (0)  |  |
| No overflow                                |  | Overflow        |  | Overflow        |  |

Figure 10.17: Illustration of Addition Overflow for Unsigned Integers and Signed Integers

the main cause for the Ariane 5 rocket crash was an undetected arithmetic overflow (cf. <http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html>).

### Overflow Detection Hardware for Unsigned Integers

On inspecting the left hand side of Figure 10.17, we can see that when adding unsigned integers, the presence of an overflow is accompanied by a carry-out from the most significant bit. Thus, it is very straightforward to detect addition overflows for unsigned integers; just checking the carry-out bit will do.

Similarly, when subtracting unsigned integers, an overflow occurs when a larger number is subtracted from a smaller one. If the subtraction is performed by adding the 2's complement of the subtrahend, then the lack of a carry-out indicates this. In other words, arithmetic overflows for unsigned integers can be detected by the presence of carry-out bit for addition, and the absence of carry-out bit for subtraction.

### Overflow Detection Hardware for Signed Integers

The overflow detection for signed integers is a little more involved. Let us inspect the right hand side of Figure 10.17, which deals with signed integers represented in the 2's complement number system. In the first case, a negative number and a positive are added together. The value of the result, in such a case, will be in between the two numbers, and so no overflow can occur.

In the second case, two positive numbers are added together, and the result obtained is apparently a negative number! In the third case, two negative numbers have been added together, and the result obtained is apparently a positive number. In both cases, the result has an incorrect sign bit. Therefore, overflow detection for signed integers can be done by observing the sign bits of the input values and the result. An addition overflow occurs if the sign bits of the addends ( $X$  and  $Y$ ) are same, and the sign bit of the result ( $S$ ) is different.

$$\text{Overflow}_{\text{Addition}} = X_{N-1} Y_{N-1} \overline{S_{N-1}} + \overline{X_{N-1}} \overline{Y_{N-1}} S_{N-1}$$

For a combined addition-subtraction unit, the variable  $Y_{N-1}$  in the **Overflow** expression should be taken from the output of the EXOR gate that complements the primary input signal  $Y_{N-1}$ . This provides the correct indication of overflow from either addition or subtraction.

Arithmetic overflows for signed integers can also be detected by observing the carry-in and carry-out signals pertaining to the sign bit (most significant bit) of the adder/subtractor. The truth table given in Table 10.2 shows the 8 possible situations that can occur in the sign bit of an  $N$ -bit adder. Bits  $X_{N-1}$  and  $Y_{N-1}$  represent the sign bits of the numbers being added and are therefore inputs to the stage, along with carry signal  $C_{N-1}$ . The outputs of the stage are the carry-out ( $C_N$ ) and the sum bit ( $S_{N-1}$ ).

| Inputs to Sign Stage |           |           | Outputs from Sign Stage |       | Overflow |
|----------------------|-----------|-----------|-------------------------|-------|----------|
| $X_{N-1}$            | $Y_{N-1}$ | $C_{N-1}$ | $S_{N-1}$               | $C_N$ |          |
| 0                    | 0         | 0         | 0                       | 0     | 0        |
| 0                    | 0         | 1         | 1                       | 0     | 1        |
| 0                    | 1         | 0         | 1                       | 0     | 0        |
| 0                    | 1         | 1         | 0                       | 1     | 0        |
| 1                    | 0         | 0         | 1                       | 0     | 0        |
| 1                    | 0         | 1         | 0                       | 1     | 0        |
| 1                    | 1         | 0         | 0                       | 1     | 1        |
| 1                    | 1         | 1         | 1                       | 1     | 0        |

Table 10.2: Truth Table of Sign Bit Position of  $N$ -bit Adder

As seen from the table, an overflow occurs if and only if

$$C_{N-1} \neq C_N$$

The corresponding logic circuit requires a single XOR gate, as shown in Figure 10.18. This circuit is simpler than that of the previous method, but requires access to the carry signal  $C_{N-1}$  between the last two stages of the adder.

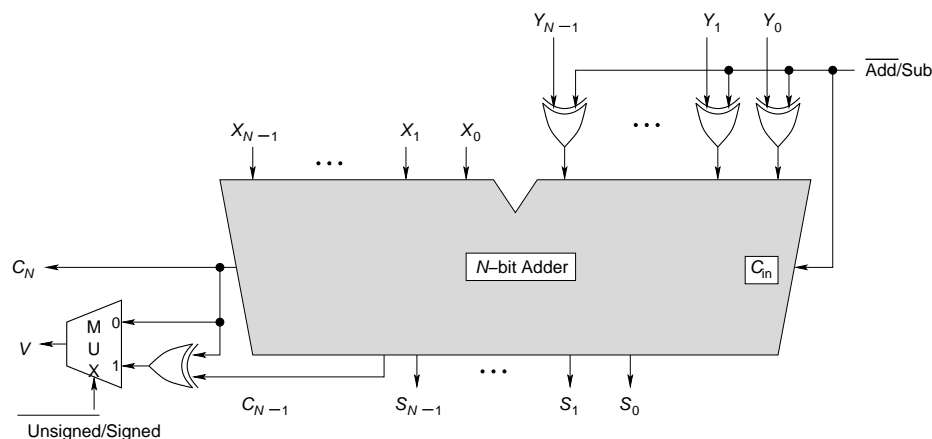


Figure 10.18: Block Diagram of an  $N$ -bit Adder/Subtractor with Overflow Detection for Unsigned/Signed Integers

### 10.2.4 Implementing Logical Operations

The logic functions performed by a computer—such as AND, OR, and XOR—are parallel, bit-wise operations on two bit patterns  $X$  and  $Y$ . This means that bit  $i$  of the result,  $F_i$ , is strictly a logic function of input bits  $X_i$  and  $Y_i$ .

An  $N$ -bit AND operation can be implemented with a string of  $N$  2-input AND gates, as shown below. Other logic operations can be easily implemented in hardware in a similar manner.

### 10.2.5 Implementing a Shifter

### 10.2.6 Putting It All Together: ALU

Let us design a simple  $N$ -bit ALU that can perform 4 functions: the two standard arithmetic operations, ADD and SUB, and the two logical operations, AND and OR. Because there



are a total of 4 functions, the function selection code must contain 2 bits; that is,  $F = F_1F_0$ . Table 10.3 gives a possible set of selection codes for the 4 functions.

| Function | $F_1$ | $F_0$ |
|----------|-------|-------|
| ADD      | 0     | 0     |
| SUB      | 0     | 1     |
| AND      | 1     | 0     |
| OR       | 1     | 1     |

Table 10.3: Function Codes for Example ALU

Figure 10.19 shows an ALU composed by combining an adder/subtractor with the logical units. Each of the logical units are shown by a single large AND or OR gate. The first  $N$ -bit 2:1 multiplexor selects “ $XY$ ” or “ $X + Y$ ,” depending on the value of  $F_0$ . The second  $N$ -bit 2:1 multiplexor selects the output of the logical unit (AND/OR) or the arithmetic unit (adder/subtractor) based on the value of  $F_1$ . The worst-case delay of this ALU will be dictated by the adder/subtractor block, as its delay is more than the combined delay of the AND block (OR block) and its following MUX.

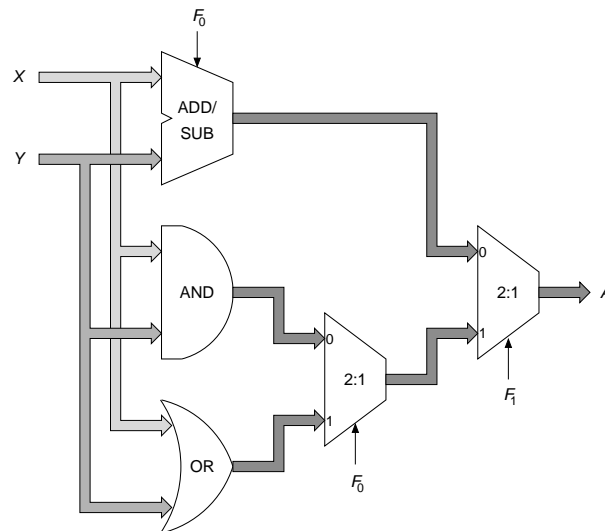


Figure 10.19: A Block Diagram of a 4-Function ALU

### 10.2.7 Implementing an Integer Multiplier

Next, let us consider the logic design of an integer multiplier. Unlike the arithmetic operations we saw so far, multiplication takes a much longer time to do, and therefore a multiplier

is not usually implemented as part of the ALU.

## Unsigned Integers

First, let us look at the paper-and-pencil (or *shift-and-add*) method for multiplying two unsigned integers. Figure 10.20 shows an example with 4-bit unsigned integers. The top number is called the *multiplicand* and the second number is called the *multiplier*. The 4 rows labeled  $PP_0$  to  $PP_3$  are known as *partial products*. The partial products are progressively shifted left by 1 bit to provide them with the appropriate weights. The final product is the sum of these 4 partial products.

|   |               |               |              |
|---|---------------|---------------|--------------|
|   | 0 1 1 0       | ( $6_{10}$ )  | Multiplicand |
| × | 0 1 0 1       | ( $5_{10}$ )  | Multiplier   |
|   | 0 1 1 0       |               | $PP_0$       |
|   | 0 0 0 0       |               | $PP_1$       |
|   | 0 1 1 0       |               | $PP_2$       |
|   | 0 0 0 0       |               | $PP_3$       |
|   | 0 0 1 1 1 1 0 | ( $30_{10}$ ) | Product      |

Figure 10.20: An Example Multiplication of 4-bit Unsigned Integers Using the Paper-and-Pencil Method

Notice that if the product were expressed in 4 bits, an overflow would have occurred. In general, when multiplying two  $N$ -bit numbers, if we provide only  $N$  bits for expressing the product, then overflow will occur very frequently. To avoid frequent overflows, the result is usually expressed with more bits. With  $N$  bits, the range of unsigned integers that can be represented is 0 to  $2^N - 1$ . The biggest product is obtained when  $2^N - 1$  is multiplied by itself to produce  $2^{2N} - 2^{N+1} + 1$ . This product requires  $2N$  bits to be represented in the binary format. Thus  $2N$  bits are required to express the largest product without causing overflow. We shall next consider the algorithm for multiplying signed integers, and then design a single logic-level multiplier circuit to handle both types of integers.

## Signed Integers

We now discuss multiplication of signed integers, which generates a double-length ( $2N$ -bit) product in the 2's complement number system. The general strategy is still to accumulate partial products by adding versions of the multiplicand as selected by the multiplier bits. Let us first look at the number of bits required to express the product without getting

overflows. With  $N$  bits, the range of signed integers that can be represented is  $-2^{N-1}$  to  $+2^{N-1} - 1$ . The biggest product is obtained when  $-2^{N-1}$  is multiplied by itself to produce  $2^{2N-2}$ . If  $2N - 1$  bits are used to represent the product, the largest representable integer is  $2^{2N-2} - 1$ , which is one less than the largest product,  $2^{2N-2}$ . Therefore, for signed integers also, the minimum number of bits required to express the product without ever causing an overflow is  $2N$ .

When multiplying two positive integers, multiplication can proceed as seen in Figure 10.20 for unsigned integers. However, if we use the same method with a negative multiplicand, then we get incorrect result, as shown in the left hand side of Figure 10.21. What went wrong here? The mistake is that the non-zero partial products in this case are negative, and we have added partial products of different lengths without doing sign-extension! The easiest way to handle this case is to sign-extend the multiplicand to the left as far as the product might extend (i.e., up to a total of  $2N$  bits). Thus, to handle negative multiplicands, all that needs to be done is to express the multiplicand in  $2N$  bits with the use of sign-extension, as shown in the right hand side of Figure 10.21. Notice that the sign-extension method produces correct results for positive multiplicands also.

|   |  |   |  |
|---|--|---|--|
| $  \begin{array}{r}  1\ 1\ 1\ 0\ (-2_{10}) \\  \times\ 0\ 1\ 0\ 1\ (5_{10}) \\  \hline  1\ 1\ 1\ 0\ \text{PP}_0 \\  0\ 0\ 0\ 0\ \text{PP}_1 \\  1\ 1\ 1\ 0\ \text{PP}_2 \\  0\ 0\ 0\ 0\ \text{PP}_3 \\  \hline  1\ 0\ 0\ 0\ 1\ 1\ 0\ (-58_{10})  \end{array}  $ |  | $  \begin{array}{r}  1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ (-2_{10}) \\  \times\ 0\ 1\ 0\ 1\ (5_{10}) \\  \hline  1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ \text{PP}_0 \\  0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ \text{PP}_1 \\  1\ 1\ 1\ 1\ 1\ 1\ 0\ \text{PP}_2 \\  0\ 0\ 0\ 0\ 0\ 0\ \text{PP}_3 \\  \hline  1\ 1\ 1\ 1\ 0\ 1\ 1\ 0\ (-10_{10})  \end{array}  $ |  |
| <b>Incorrect Result</b>   |  | <b>Correct Result</b>   |  |

Figure 10.21: Incorrect and Correct Multiplications Involving Negative Multiplicand

Next, consider the situation where the multiplier is negative. A straightforward solution would be to take the 2's complement of both the multiplier and the multiplicand, and proceed as in the case of a positive multiplier. This is possible because complementation of both operands does not change the value or the sign of the product. However, there is a more elegant way to handle negative multipliers. This method makes use of the property that the 2's complement number system is a positional number system. The value of a negative number represented in this number system is given by the following formula; the positional weights in this formula are the same as that for the binary system, except that the most significant bit has a negative weight:

$$V_{2's\text{Complement}} = -b_{N-1} \times 2^{N-1} + b_{N-2} \times 2^{N-2} + b_{N-3} \times 2^{N-3} + \dots b_0 \times 2^0$$

For example, in 2's complement number system, the value of bit pattern 101101 means  $-1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = -19$ . Thus, to handle negative multipliers, all multiplier bits except the MSB can be treated as before when forming partial products. The multiplier's MSB has a negative weight, and this is handled by taking the 2's complement of the multiplicand when considering the partial product corresponding to the multiplier's MSB.

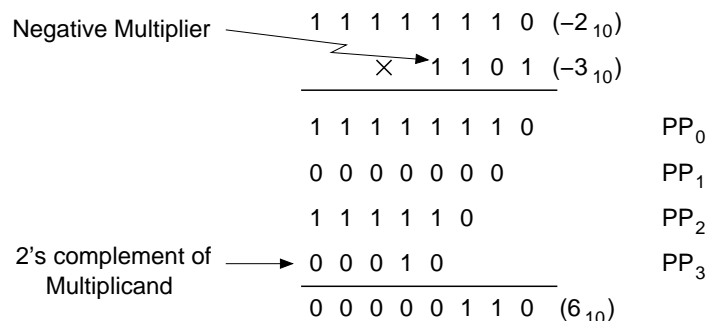


Figure 10.22: Multiplication Involving a Negative Multiplier

## Logic-Level Hardware for Integer Multiplication

Now that we have reviewed the basics of multiplication, let us look at hardware circuits for carrying out multiplication. The algorithm for multiplication, as we just saw, involves adding together a set of partial products. One way of doing this is to perform a series of 2-input additions using a single adder. This can be done as follows: add the first two partial products to form a temporary result. After that, repeatedly add the next partial product to the temporary result. A hardware circuit to perform multiplication in this manner can be developed using one of the adders that we had designed earlier. Figure 10.23 shows such a circuit, drawn specifically for multiplying 4-bit integers. We have drawn the circuit in such a way that the data flow from the top to bottom more closely resembles the paper-and-pencil method that we illustrated. The circuit consists of a  $2N$ -bit shift register (the **Multiplicand Register**), an  $N$ -bit shift register (the **Multiplier Register**), a  $2N$ -bit **Product Register**, and a  $\log_2 N$ -bit **Counter**.

### Register Initializations:

- *Multiplicand register:* The least significant  $N$  bits are initialized to the multiplicand value. The most significant  $N$  bits are initialized to 0s or the sign bit, depending on whether the data type is unsigned or signed.
- *Multiplier register:* This  $N$ -bit register is initialized to the multiplier value.

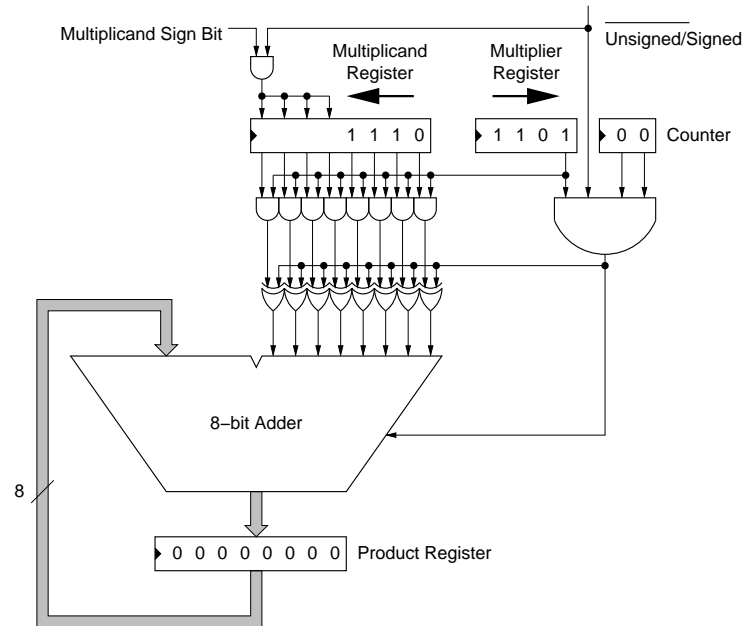


Figure 10.23: A Hardware Circuit for Multiplying 4-bit Unsigned or Signed Integers

- *Product register*: This  $2N$ -bit register is initialized to zero.
- *Counter*: This  $\log_2 N$ -bit register is initialized to zero.

Several observations can be made concerning this circuit. First of all, the adder used is a  $2N$ -bit adder; this is required because the partial products are  $2N$  bits long. The shifting of the partial products is achieved by left-shifting the Multiplicand register. And the partial product is created exactly as shown in the previous binary multiplication examples: AND gates are used to generate the partial product from the multiplicand. The multiplier bit to be used is obtained from the Multiplier shift register. For signed integers, sign extension of the multiplicand is performed. In Figure 10.24, we depict this hardware's working algorithm.

*Example:* Let us see how the registers of this multiplier hardware change when the signed integer 1110 is multiplied by signed integer 1101. Table 10.4 gives a progression of the multiplier hardware register values during the multiplication of these two signed integers.

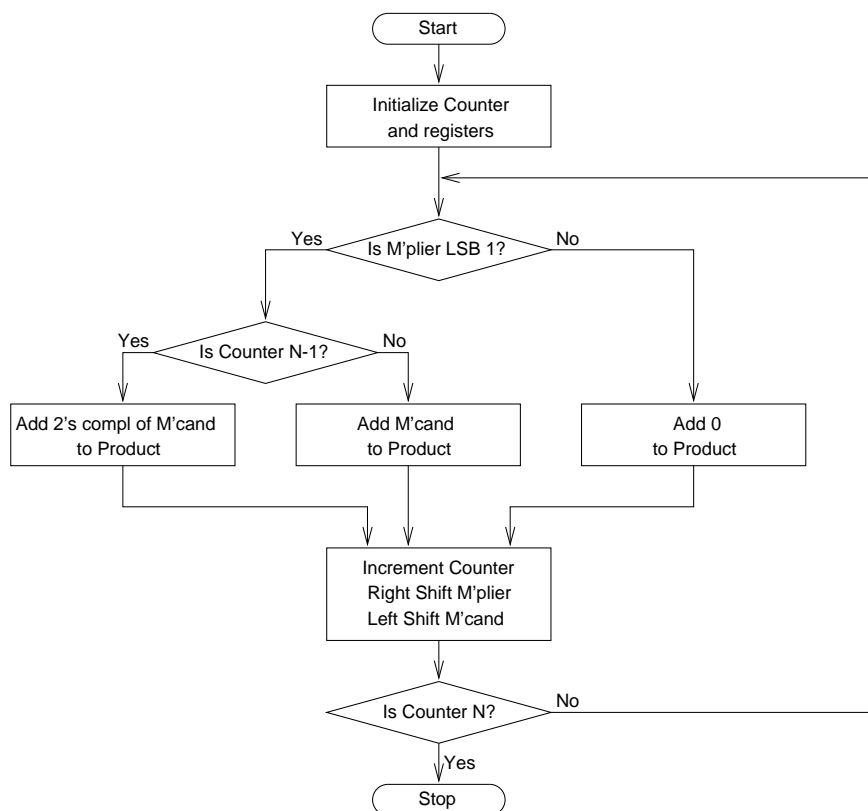


Figure 10.24: A Flowchart Depicting the Working of Multiplier Hardware

### 10.2.8 Implementing a Floating-Point Adder

### 10.2.9 Implementing a Floating-Point Multiplier

## 10.3 Implementing a Register File

An important component in all of the datapaths we saw in Chapter \*\* is a register file, a collection of registers organized as an array. An individual register in this array is read or written by specifying its address. At the microarchitecture level, the specification of a register file includes the following attributes: (i) the number of registers ( $R$ ), (ii) the width of each register ( $W$ ), (iii) the number of read/write ports ( $p$ ), (iv) the number of read-only ports ( $r$ ), and (v) the number of write-only ports ( $w$ ). When designing a register file at the logic level, we adhere to these microarchitectural specifications. Other parameters of interest are the area, access time, and power consumption; we address these after presenting

| Iter. | MAL Operation   | Counter | M'plier      | M'cand       | Product                |
|-------|---|---------|--------------|--------------|------------------------|
| 0     | Initial Values  | 0       | 1101         | 1110         | 0000 0000              |
| 1     | Product $\leftarrow$ Product + 1 $\times$ M'cand<br>M'plier $\gg=$ 1; M'cand $\ll=$ 1; Counter++    | 0<br>1  | 1101<br>0110 | 1110<br>1100 | 1111 1110<br>1111 1110 |
| 2     | Product $\leftarrow$ Product + 0 $\times$ M'cand<br>M'plier $\gg=$ 1; M'cand $\ll=$ 1; Counter++    | 1<br>2  | 0110<br>0011 | 1100<br>1000 | 1111 1110<br>1111 1110 |
| 3     | Product $\leftarrow$ Product + 1 $\times$ M'cand<br>M'plier $\gg=$ 1; M'cand $\ll=$ 1; Counter++    | 2<br>3  | 0011<br>0001 | 1000<br>0000 | 1111 0110<br>1111 0110 |
| 4     | Product $\leftarrow$ Product + (-1) $\times$ M'cand<br>M'plier $\gg=$ 1; M'cand $\ll=$ 1; Counter++ | 3<br>4  | 0001<br>0000 | 0000<br>0000 | 0000 0110<br>0000 0110 |

Table 10.4: Progression of Multiplier Register Values During Multiplication

transistor-level designs.

Single-ported register files are generally present only in the simplest of the datapaths; an example is the single-bus datapath of Figure \*\*. More complex datapaths ranging from multi-bus datapaths to pipelined datapaths to superscalar datapaths all use multi-ported register files, as we saw in Chapter \*\*. Greater the degree of multiple issue, greater the number of read ports and write ports required. For instance, Intel's Itanium 2 processor, which is designed to issue up to six instructions in a clock cycle—each requiring two register reads and a register write—and to accept two late return values from the data cache, has a register file with 12 read ports and 8 write ports!

Associated with each port, there is one set of address input lines and one set of data lines. Each port can therefore accept a separate register address in a clock cycle; a 2-ported register file can thus accept two register addresses in a clock cycle. Each port may be read-only, write-only, or read/write. If two read ports that are active in a clock cycle receive the same register address, both sets of data lines will have the same data. However, if two simultaneously active write ports receive the same address, only one of them can be allowed to proceed. Some register files do not have the provision to prevent multiple write ports from writing the same entry simultaneously. In such cases, the control unit has to ensure that multiple register writes in a clock cycle are to different registers.

Let us consider the 2-read port, 1-write port MIPS register file used in the direct path based datapath as well as the pipelined data paths we studied earlier. Figure 10.25 shows a block diagram of this register file. In any clock cycle, we can simultaneously read from two (possibly same) registers, and write into one register.

### 10.3.1 Logic-level Design

We shall first describe a logic-level design of the MIPS register file discussed above. For clarity, we shall present this design at a higher level with registers, multiplexers, and decoders as the building blocks. The functionality of a register file can be broken down into

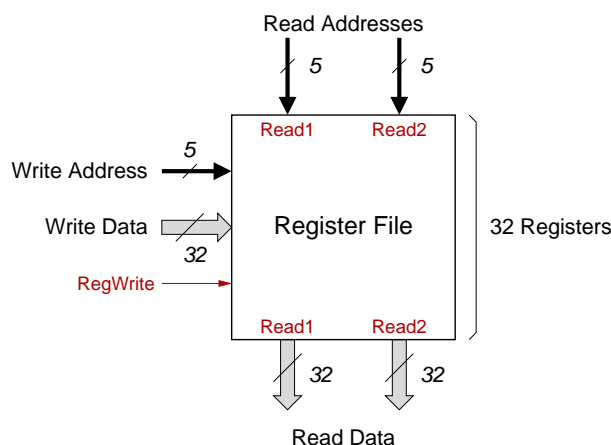


Figure 10.25: Logic Symbol of a 2-read port, 1-write port, 32-entry 32-bit register file

3 parts:

- data storage
- read port(s)
- write port(s)

Each register is just a collection of bits; in this design individual bits are stored in *flip-flops*. Thus, each register is implemented using 32 flip-flops arranged as a one-dimensional array. The clock input of the flip-flops are connected together; the `load` input of the flip-flops are also connected together so that an entire register is updated at the same time. 32 such registers are stacked together as an array of registers to form the data storage portion of the register file.

The most straightforward way to implement a write port is to use a decoder to decode the write address, and connect each output of the decoder to the `load` input of the corresponding register. The data lines of the write port are routed to the flip-flops of each register.

When carrying out a logic-level design, a read port can be implemented in two different ways: (i) based on a multiplexer and (ii) based on a combination of a set of tristate buffers and a decoder. In the first approach, shown in Figure 10.26, the data outputs of all 32 registers are fed to a (32-bit wide) 32:1 multiplexer, which selects the appropriate register value. The 5-bit register address of the read port is fed to the 5-bit `select` input of the multiplexer. The second read port is implemented by using a second MUX.

In the second approach, the outputs of all 32 registers are connected to a register output bus through tristate buffers. The tri-state buffers are controlled by the outputs of a 5:32 decoder that is fed by the 5-bit register address of the read port. This approach is shown



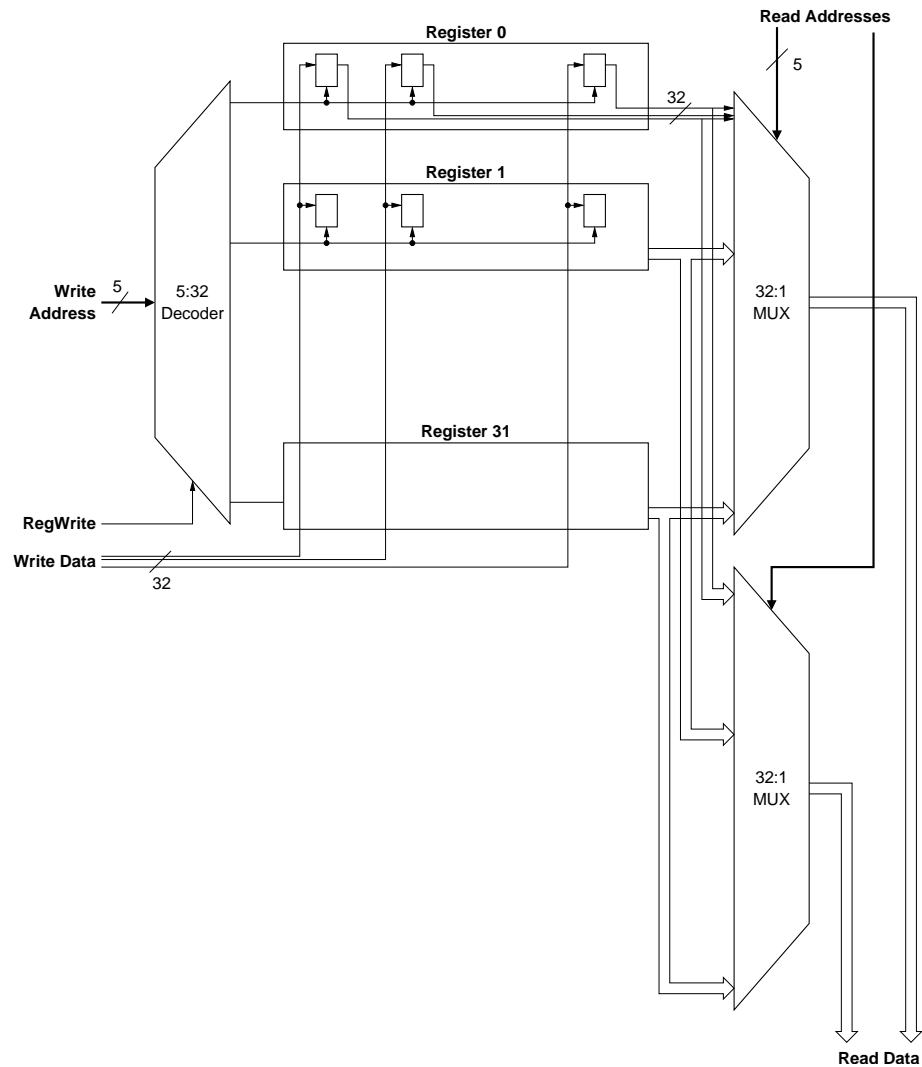


Figure 10.26: Logic-Level Design of a 2-read port, 1-write port, 32-entry 32-bit register file using a 32-bit wide 32:1 MUX for each read port

in Figure 10.27, again for a single read port. If there are multiple read ports, for each read port, we need a separate set of tristate buffers that will be controlled by the outputs of a separate 5:32 address decoder.

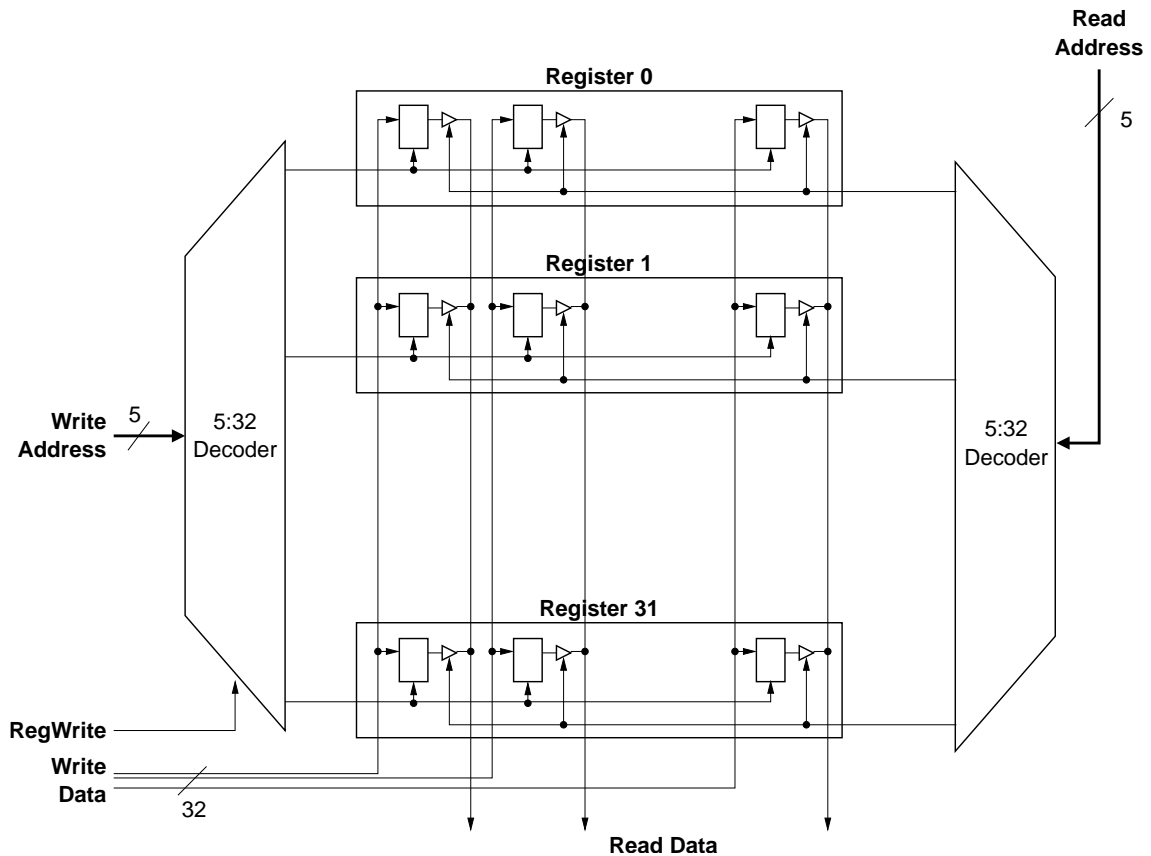


Figure 10.27: Logic-Level Design of a 1-read port, 1-write port, 32-entry 32-bit register file using a 5:32 decoder for the read port

### 10.3.2 Transistor-level Design

Modern integrated circuit-based register files are usually implemented directly at the transistor level, by way of fast static RAM (Random Access Memory) cells. The design of SRAM-based register files is very similar to that of SRAM-based memory structures that we will see in the next section. One difference, however, is that a read/write port in a register file is implemented as a dedicated read port and a write port, whereas they are usually implemented as a single port in SRAM designs meant for memory structures.

In the transistor-level register file design, the basic unit of storage is called an SRAM cell instead of a flip-flop. The SRAM cell is functionally equivalent to a flip-flop in that it has two stable states and can be made to switch between the states with the help of external input. The main difference is that direct implementation of the flip-flop functionality using transistors leads to a more efficient design than taking a gate-level flip-flop circuit and

implementing its gates using transistors.

Multiple SRAM cells are organized in a two-dimensional manner to form the data storage portion of the register file, much like the two-dimensional organization of flip-flops in the logic-level design<sup>3</sup>. Again, the usual layout convention is to line up the cells of a register horizontally, and to line up the registers vertically. Figure 10.28 depicts this arrangement.

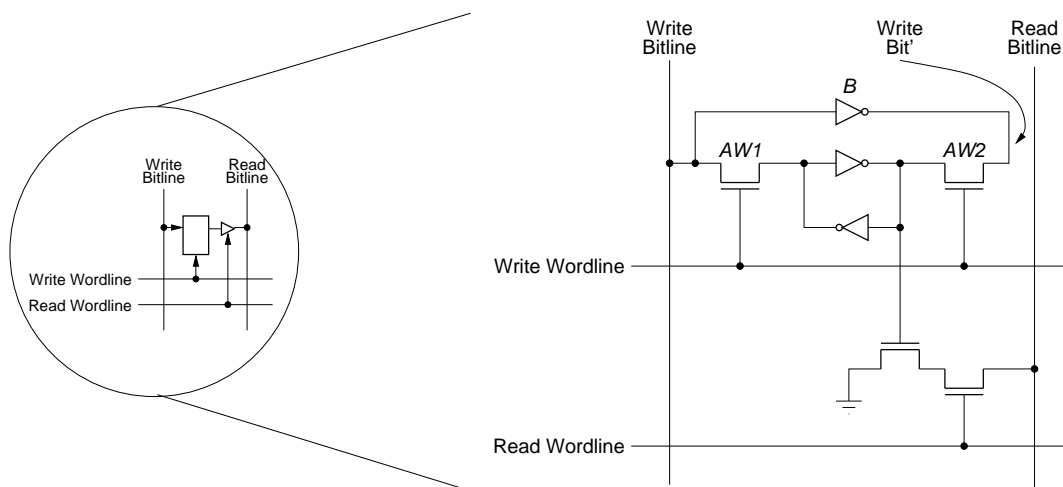


Figure 10.28: Transistor-Level Design of an RF Cell incorporating one read port and one write port

The implementation of write ports in this transistor-level design is somewhat similar to that in the logic-level design: for each write port, there is a decoder whose outputs drive word lines corresponding to individual registers. The same word line connects to all  $W$  cells of a register, and decides if the cells of the register should be connected to the corresponding bit line(s). This controlling action is done with the help of two *access transistors*  $AW1$  and  $AW2$ . The bit values to be written onto the cells of this register are placed on the (vertical) bit lines. For better noise margins, it is customary to provide complementary values at the inputs of the two inverters that are connected back-to-back. We can implement this either by using two bit lines for each bit position, or by using a single bit line and local inverters at each cell. Figure 10.28 shows the latter approach; inverter  $B$  is the local inverter. With the use of local inverters, we are able to reduce the number of bit lines required per write port to  $W$ , the width of each register.

The implementation of the read ports is also similar to the decoder-based read port design that we saw for the logic-level design. For each read port, there is a decoder whose outputs drive the *read word lines*. Activation of a read word line causes that row of cells to place their data on the (vertical) bit lines. Read bit lines often swing only by a small

<sup>3</sup>Larger register files are sometimes constructed by tiling mirrored and rotated simple arrays.

voltage. Sense amplifiers, placed at the bottom of the bit lines, convert these small-swing signals into full logic levels.

A register file altogether has  $R(r + w)$  word lines and  $W(r + w)$  bit lines (or  $W(r + 2w)$  bit lines if local inverters are not used to get the complement of the write value), where  $R$  is the number of registers,  $W$  is the width of registers,  $r$  is the number of read ports, and  $w$  is the number of write ports. Therefore, the pitch area for the wires increases as the square of the number of ports. Multi-ported register files often tend to be widest pitch unit in the processor datapath. If the VLSI layout of the processor uses *pitch matching* so as to avoid having the many busses passing over the datapath turn corners (i.e., all units in the datapath have the same bit pitch), then multi-ported register files often set the pitch of the processor datapath. Ironically, we can reduce the area of highly multi-ported register files (and increase their speed as well) by replicating the register file array and partitioning the read ports among the replica. For instance, a 16-read port, 4-write port register file can be implemented in less area by designing two 8-read port, 4-write port register files that are parallelly updated during every write operation. Because the cells of the replicated register files are considerably smaller, the overall length of the bit lines and word lines also tend to be smaller, resulting in faster operation as well.

## 10.4 Implementing a Memory System using RAM Cells

### 10.4.1 Implementing a Memory Chip using RAM Cells

Memory cells are usually organized in the form of an array. One such organization is shown in Figure 10.29. Each row of cells constitute a memory word, and all cells of a row are connected to a common line referred to as the **word line**, which is driven by an on-chip address decoder. The cells in each column are connected to a Sense/Write circuit by two **bit lines**. The Sense/Write circuits are connected to the data input/output lines of the chip. During a Read operation, these circuits sense (or read) the information stored in the cells selected by a word line and transmit this information to the output data lines. During a Write operation, the Sense/Write circuits receive input information and store it in the cells of the selected word. Two control lines,  $R/\overline{W}$  (Read/ $\overline{\text{Write}}$ ) and  $CS$  (Chip Select), are provided in addition to the address and data lines. The  $R/\overline{W}$  input specifies the required operation, and the  $CS$  input selects a given chip in a multichip memory system. This is discussed in more detail below.

### 10.4.2 Implementing a Memory System using RAM Chips

A big memory system is implemented using several smaller RAM chips. The choice of a particular size RAM chip for a given memory system depends on several factors. Foremost among these factors are the chips' speed, power dissipation, and size. Consider a memory system consisting of 4 Mwords, each of which is 32 bits wide. Figure 10.30 shows how this

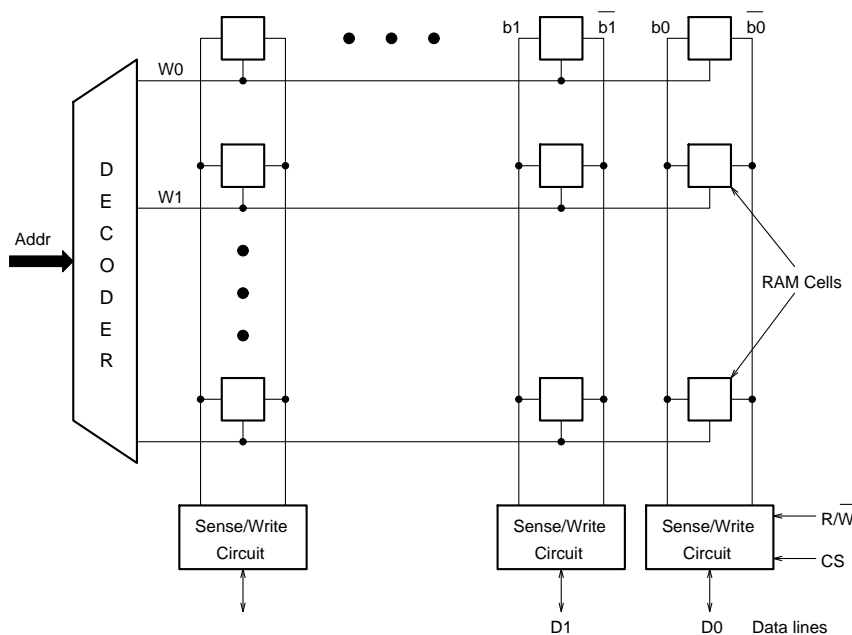


Figure 10.29: Organization of Memory Cells in a RAM Chip

memory system can be implemented using  $1 \text{ M} \times 8$  RAM chips. When the  $\overline{\text{CS}}$  input of a chip is activated, the chip is able to accept data input or to place data on its output lines. On a Read operation, only the selected chip places data on the output line. The address bus for selecting a word in the 4 Mword memory system is 22 bits wide. The two high-order bits of the address are decoded to obtain the four  $\overline{\text{CS}}$  control signals, and the remaining 20 address lines are used to access a specific row of cells inside each selected chip.

### 10.4.3 Commercial Memory Modules

We have considered the key aspects of DRAM chips and the larger memories that can be constructed using these chips. Let us briefly consider the issue of physically assembling a memory system. Modern computers use a large amount of main memory; even a small personal computer is likely to have at least 64 MB of memory. Typical workstations have at least 256 MB of memory. If a large memory system is built by placing DRAM chips directly on the motherboard (i.e., the printed circuit board that contains the processor and the off-chip cache), it will occupy an unacceptably large amount of space on the motherboard. Also, it is difficult to provide for future expansion of the memory. These packaging considerations have led to the development of larger memory units known as **SIMMs** (Single In-line Memory Modules). A SIMM is an assembly of several DRAM chips on a separate small PCB that plugs vertically into a single socket on the motherboard, thereby utilizing vertical

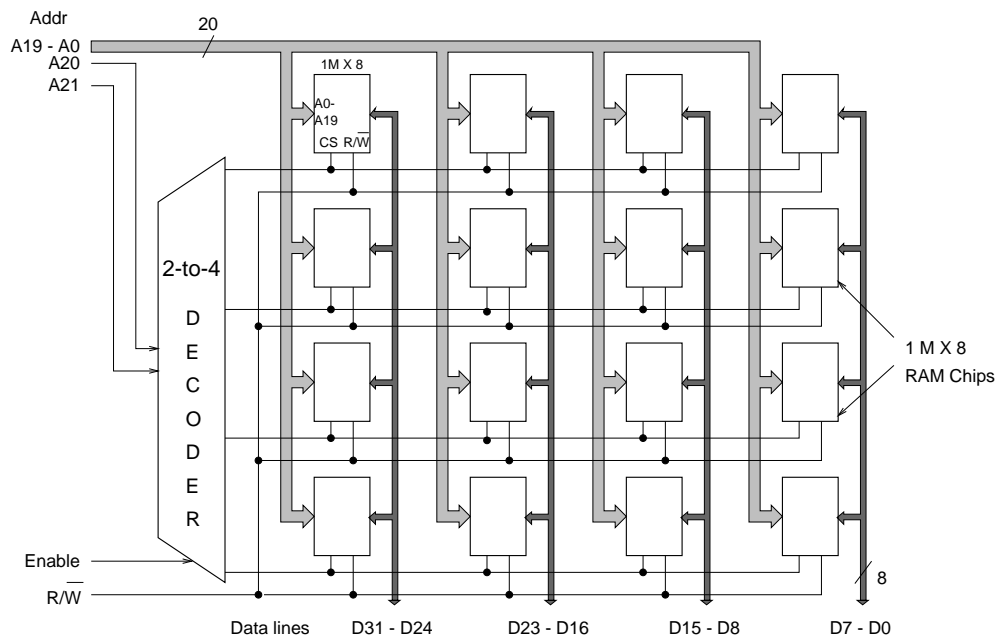


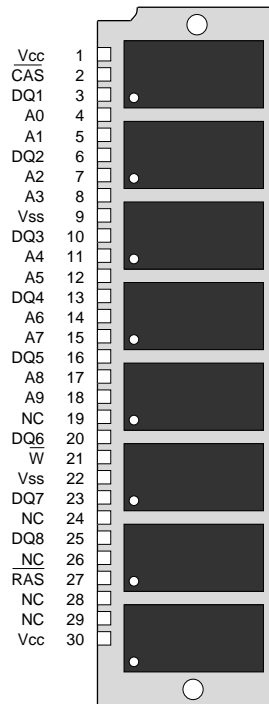
Figure 10.30: Organization of a 4 M × 32 Memory Module using 1 M × 8 RAM Chips

space inside the system unit. SIMMs of different capacities are designed to use the same size socket. For example, 1M × 8, 4M × 8, and 16M × 8 SIMMs all use the same 30-pin socket. And, 1M × 32, 2M × 32, 4M × 32, and 8M × 8 SIMMs use the same 72-pin socket.

Figure 10.31 shows an 1M × 8 SIMM module built from eight 1M × 1 RAM chips. This module has 30 pins, numbered 1-30. Notice that all of the pins lie in a single line. There are 8 pins (DQ1-DQ8) for transferring data in and out. The 20-bit address for selecting a memory location is assumed to be split into a 10-bit row address and a 10-bit column address, which are loaded separately through the 10 address lines A0 – A9. The Column Address Strobe and Row Address Strobe input signals are used by the module to gate in the corresponding portion of the address.

## 10.5 Implementing a Bus

A typical bus consists physically of a number of parallel conductors (or wires) routed to every connected module. Although flexible ribbon cable with appropriate connectors can be used to extend the bus between modules, the most common bus-connector technology is a rigid **backplane** containing a row of multiconductor connectors whose corresponding pins are wired together as sketched in the following figure. System modules (typically each a printed circuit card) plug into the connectors, thereby sharing the backplane wiring.

Figure 10.31: Organization of an  $1\text{M} \times 8$  SIMM Module

The physical aspects of bus communication force us to deal with several electrical issues. The number of interconnected modules and the physical distances involved mean that considerable power must be devoted to driving bus lines, and that signal propagation over bus lines must be considered as time-consuming rather than instantaneous.

### 10.5.1 Bus Design

At the microarchitecture level, the specification of a bus includes the following attributes: (i) the width of the data bus ( $D$ ), (ii) the width of the address bus ( $A$ ), (iii) the modes of transfer supported, and (iv) prioritized access (if any). When designing a bus at the logic level, we adhere to these microarchitectural specifications. Other parameters of interest are the clocking, arbitration, area, data rate, and power consumption; we address these in the following sections.

### 10.5.2 Bus Arbitration

Now that we have reviewed some of the many design options for buses, we can deal with one of the most important issues in bus design: How is the bus reserved by a device that wishes to use it to communicate? It is reasonable to ask why we need a scheme for controlling bus access. The answer is that without any control, multiple devices trying to communicate could each try to assert the control and data lines for different transfers! Deciding which bus master gets to use the bus next is called bus arbitration. There are a wide variety of schemes for bus arbitration; these may involve special hardware or extremely sophisticated bus protocols. The arbiter implements a priority system for granting access to the bus. For instance, the DMA controller is given a higher priority than the processor (because it deals with high-speed peripherals such as disk and tape drives) so that when both the processor and the DMA controller request for the bus, the DMA controller is allowed to use the bus.

### 10.5.3 Bus Protocol: Synchronous versus Asynchronous

When a signal or value is transmitted over a bus to a receiver, the receiver has no inherent ability to determine exactly *when* the signal becomes valid. To prevent a receiver from reading incorrect data before the arrival of the correct value, it is important to institute some means for synchronizing the writes and reads. A variety of schemes have been devised to enforce this synchronization. These can be broadly classified as either synchronous or asynchronous schemes.

- clocked (synchronous)
- handshake-based (asynchronous)

The synchronous scheme is typically used for the processor-memory bus, whereas the asynchronous scheme is typically used for the peripheral bus and the backplane bus.

#### Synchronous Protocol

With a synchronous protocol, all devices connected to the bus derive timing information from a common clock line. Equally spaced pulses on this clock line define equal time intervals; each interval constitutes a *bus cycle*, during which one data transfer can take place. Such a scheme is illustrated in Figure ???. Note that the address and data lines in the figure are shown as high and low at the same time. This indicates that some lines are low and some high, depending on the particular address or data pattern being transmitted. The crossing points indicate the times at which these patterns change. A signal line in a high impedance state is represented by an intermediate level half-way between the low and high signal levels.



Let us consider the sequence of events during a read operation. At time  $t_0$ , the processor places the memory address on the address lines and sets the mode control lines to indicate a read operation. This information travels over the bus at a speed determined by its physical and electrical characteristics. The addressed memory module, recognizing that an input operation is requested, places the requested data on the data lines at time  $t_1$ . At the end of the clock cycle, that is, at time  $t_2$ , the processor *strokes* the data lines and loads the data into its input buffer.

Figure: Timing Diagram of a Read Operation on a Synchronous Bus

The procedure for a write operation is similar to that for the input operation. The processor places the output data on the data lines when it transmits the address and mode information. At time  $t_1$ , the addressed memory device strobes the data lines and loads the data into its input buffer.

**Advantages and Disadvantages of Synchronous Bus:** The synchronous bus protocol is simple and results in a simple design for the device interface. This allows a fast bus, if all modules connected to it are fast. Synchronous buses have two major disadvantages, however. **First**, every module on the bus must run at the same clock rate. The clock speed must therefore be chosen such that it accommodates the longest delays on the bus and the slowest interface. Note that the processor has no way of determining if the addressed module has actually responded. It simply assumes that, at time  $t_2$ , the output data have been received by the module or the input data are available on the data lines. If, because of a malfunction, the module did not respond, this error will not be detected. **Second**, because of clock-skew problems, synchronous buses cannot be long if they are to be fast.

## Asynchronous Protocol

An alternative scheme for controlling data transfers on the bus is based on the use of a *handshake* between the processor and the module being addressed. The common clock is

eliminated; hence, the resulting bus operation is *asynchronous*. The clock line is replaced by 2 control signals, which we refer to as **Request** and **Acknowledge**. In principle, a data transfer controlled by a handshake proceeds as follows: The processor places the address and read/write mode information on the bus. Then it indicates to all modules that it has done so by activating the **Request** line. When the addressed module observes the **Request** signal, it performs the required operation and then informs the processor that it has done so by activating the **Acknowledge** line. The processor waits for the **Acknowledge** signal before it removes its signals from the bus. In the case of a read operation, it also strobes the data into its input buffer.

An example of the timing of a data transfer using the handshake scheme is given in the following figure; the figure depicts the following sequence of events:

Figure: Timing Diagram of a Read Operation on an Asynchronous Bus

- $t_0$ —The processor places the address and read/write mode information on the bus.
- $t_1$ —The processor activates the **Request** line. The delay  $t_1 - t_0$  is intended to allow the address information to propagate to all modules and be decoded by them. The propagation time takes into consideration any *skew* that may occur on the bus. Skew occurs when two signals simultaneously transmitted from one source arrive at the destination at different times. This happens because different lines of the bus may have different propagation delays. Thus, to guarantee that the **Request** signal does not arrive at any module ahead of the address and mode information, the delay  $t_1 - t_0$  should be larger than the maximum possible bus skew.

- $t_2$ —The interface of the addressed module receives the **Request** signal, and having already decoded the address and mode information, it recognizes that it should perform an input operation. Hence, it gates the data from the addressed register to the data lines of the bus. At the same time, it activates the **Acknowledge** signal. The period  $t_2 - t_1$  depends on the distance between the processor and the interface module. It is this variability that gives the bus its asynchronous nature.
- $t_3$ —The **Acknowledge** signal arrives at the processor, indicating that the input data are available on the bus. However, because it was assumed that the interface module transmits the **Acknowledge** signal at the same time that it places the data on the bus, the processor should allow for bus skew. After a delay equivalent to the maximum bus skew, the processor strobes the data into its input buffer. At the same time, it drops the **Request** signal, indicating that it has received the data.
- $t_4$ —The processor removes the address and mode information from the bus. The delay between  $t_3$  and  $t_4$  is again intended to allow for bus skew. Erroneous addressing may take place if the address, as seen by some module on the bus, starts to change while the **Request** signal is still in activation.
- $t_5$ —When the interface module sees the inactivation of the **Request** signal, it removes the data from the bus and deactivates the **Acknowledge** signal. This completes the input transfer. Note that the events corresponding to times  $t_4$  and  $t_5$  may happen in any order.

The timing for an output operation, illustrated in the following figure, is similar to that for an input operation. In this case, the processor places the output data on the data lines at the same time that it transmits the address and mode information. The addressed module strobes the data into the addressed register when it receives the **Request** signal and indicates that it has done so by activating the **Acknowledge** signal. The remainder of the cycle is identical to that of the input operation.

Although a synchronous bus may be faster, the choice between a synchronous bus versus asynchronous bus has implications not only for data bandwidth but also for an IO system's capacity in terms of physical distance and the number of modules that can be connected to the bus. Asynchronous buses scale better with technology changes, and can support a wider variety of response speeds. It is for these reasons that IO buses are often asynchronous, despite the increased time overhead.

Figure: Timing Diagram of a Write Operation on an Asynchronous Bus

## 10.6 Interpreting Microinstructions using Control Signals

We have now considered how the major blocks in the microarchitecture-level data path can be implemented at the logic level. Our next aim is to investigate how a microinstruction (i.e., an encoded MAL instruction) can be interpreted in a logic-level data path. In order to study how this interpretation is done, we first decompose each MAL instruction to indicate sub-steps such as bus operations. For instance, the MAL instruction  $PC \rightarrow MAR$  can be decomposed as:

|               |                             |
|---------------|-----------------------------|
| PC            | $\rightarrow$ Processor Bus |
| Processor Bus | $\rightarrow$ MAR           |

Similarly, the MAL instruction  $M[MAR] \rightarrow MDR$  can be decomposed as:

|                                |                                  |
|--------------------------------|----------------------------------|
| MAR                            | $\rightarrow$ System Address Bus |
| $M[\text{System Address Bus}]$ | $\rightarrow$ System Data Bus    |
| System Data Bus                | $\rightarrow$ MDR                |

The interpretation of each microinstruction is carried out at the digital logic level by recognizing these sub-steps, and activating *control signals* to carry out each sub-step.

### 10.6.1 Control Signals

Control signals are binary signals that activate the various data transfer and data manipulation operations in a data path. To show the use of control signals, we shall introduce control signals in the processor data path that we studied in Figure 7.7 on page 252. That processor data path, augmented with the control signals, is given in Figure 10.32. Five types of control signals are used here:

- Control signals such as `PC_in` and `MAR_in` are used to gate in values to registers.
- Control signals such as `PC_out` and `MDR_out` are used to control the tri-state buffers that feed to the processor bus.
- Control signals such as `MDR_select` and `R_select` are used as the control inputs of multiplexers. Multiplexers are introduced where more than one sets of inputs feed to a module. The inputs to a 2:1 multiplexer are marked as 0 and 1, whereas the inputs of a 4:1 multiplexer are marked as 00, 01, 10, and 11. We follow the convention that when the control input of a 2:1 multiplexer is not asserted, the data input marked as 0 is diverted to its output, and when the control input is asserted, the data input marked as 1 is diverted to its output.



The control signals thus perform different functions, such as multiplexer control, register read enable, and register write enable.

Thus, the MAL instruction  $PC \rightarrow MAR$  can be interpreted by activating in sequence the two control signals  $PC\_out$  and  $MAR\_in$  in the logic-level data path of Figure 10.32. If  $MAR\_in$  is activated before activating  $PC\_out$ , incorrect data may get stored. To store the correct result in  $MAR$ ,  $PC\_out$  is activated near the beginning of the clock cycle and maintained high until the activation of  $AOR\_in$ , which happens near the end of the clock cycle. Notice that the designer of the logic-level architecture has to deal with the complexity and the inter-relations between the three-state buses and the control signals, in order to implement the abstract MAL operations of the microarchitecture level.

*Example:* What are the control signal sequences that need to be asserted to interpret the following two MAL instructions in the logic-level data path of Figure 10.32?

|                           |  |
|---------------------------|--|
| $PC \rightarrow MAR:$     | $PC\_out$<br>$PC\_out, MAR\_in$                  |
| $M[MAR] \rightarrow MDR:$ | $MemRead, MDR\_select$<br>$MDR\_select, MDR\_in$ |

*Example:* Consider the MAL routine given in Table 9.5 (on page 372) to interpret the MIPS ML instruction represented symbolically as `addu rd, rs, rt`. Let us specify the control signals that need to be asserted (in the proper sequence) in order to interpret this MAL routine. Table 10.5 gives the MAL routine as well as the control signal assertions for interpreting each of the MAL instructions.

|                      | Step | MAL Instruction   | Control Signals   |
|----------------------|------|---|---|
| <i>Fetch phase</i>   | 0    | $PC \rightarrow MAR; \quad PC \rightarrow AIR$          | $PC\_out$<br>$MAR\_in, AIR\_in$                           |
|                      | 1    | $M[MAR] \rightarrow MDR; \quad AIR + 4 \rightarrow AOR$ | $MemRead, 4\_out, MDR\_select, ADD$<br>$MDR\_in, AOR\_in$ |
|                      | 2    | $MDR \rightarrow IR$                                    | $MDR\_out$<br>$IR\_in$                                    |
|                      | 3    | $AOR \rightarrow PC$                                    | $AOR\_out$<br>$PC\_in$                                    |
| <i>Execute phase</i> | 4    | $R[rs] \rightarrow AIR$                                 | $R\_select = 01, R\_out$<br>$AIR\_in$                     |
|                      | 5    | $R[rt] + AIR \rightarrow AOR$                           | $R\_select = 10, R\_out, ALU\_select=ADD$<br>$AOR\_in$    |
|                      | 6    | $AOR \rightarrow R[rd]$                                 | $AOR\_out, R\_select = 11$<br>$R\_in$                     |

Table 10.5: A MAL Routine and the Corresponding Control Signal Sequence for Interpreting the MIPS-0 ML Instruction Represented Symbolically as `addu rd, rs, rt`

### 10.6.2 Control Signal Timing

We just saw a methodology for interpreting each microinstruction using a sequence of control signal activations. We have neglected the issue of timing until now. Not all of the sub-steps are of equal duration. For instance, a typical ALU operation consists of setting up its inputs, giving the ALU time to compute the result, and holding the result. These events require unequal durations of time. If the ALU's outputs are gated into AOR before the ALU has completed the computation, incorrect value will get stored. To store the correct result in AOR, the inputs to the ALU are supplied near the beginning of the clock cycle, and AOR\_in is activated near the end of the clock cycle, as shown in Figure 10.33.



Figure 10.33: Control Signal Timing for Interpreting an ALU Microinstruction

### 10.6.3 Asserting Control Signals in a Timely Fashion

We just saw that a set of control signals can be activated at the proper time in a clock cycle to interpret a microinstruction. Next we will look into the hardware unit that is responsible for asserting the control signals in the proper order at the proper time. This function can be performed by a *control signal generator* block, which takes as input a microinstruction and generates the corresponding control signals. Instead of building the control signal generator as a separate block, most processor control units directly generate the control signals. That is, instead of generating a microinstruction each cycle, and then using it to generate the appropriate control signals, the processor control unit directly generates the control signals.

## 10.7 Implementing the Control Unit

In Section 9.4 we saw the design of a processor control unit at the microarchitecture level, as a finite state machine. In this section, we look at the design of the processor control unit at the digital logic level. To activate a sequence of data-processing operations, the control unit sends the proper sequence of control signals to the data path. The control unit, in turn, receives status bits from the data path, and uses the status bits in defining the specific sequence of operations to be performed.

An important note is in order here. A microarchitecture designer (microarchitect) typically uses MAL routines to specify how the control unit should interpret each ML instruction. Each MAL instruction is internally expressed in encoded form as a microinstruction. At the lower level (digital logic level), each of these microinstructions is interpreted by one or more **control signals**. It is more efficient for the control unit to directly generate the control signals, rather than first generate a microinstruction and then interpret it using control signals. The control unit designer therefore translates microroutines into sequences of control signal assertions.

Before the design of the control unit can begin, it is imperative that the *data path* be defined, and the appropriate *control points* and *control signals* identified. Once the set of control signals has been identified, then their *order of assertion* must be determined based on the *microroutine description* of the instructions. This specification takes the form of a state transition diagram for a *state machine-based control unit implementation*. The state machine can be implemented in a variety of ways. The two common methods are programmed control and hardwired control.

### 10.7.1 Programmed Control Unit: A Regular Control Structure

We first discuss a scheme called programmed control (also called *microprogrammed control*), in which the control signal activations are specified by a program (in an appropriate low-level language). In this approach, the control signal generator is implemented by storing the control program in a special ROM called *control store*.

There are many ways to design a programmed control unit. In the design just mentioned, the ROM has a separate row entry for each possible combination of CSC value and instruction decoder outputs. Storing the control program directly in a ROM leads to the implementation given in Figure 10.34. The CPC (control program counter) enables control instructions to be read sequentially from the control memory. When the control memory responds with the contents of that address, this information is optionally latched onto a CIR (control-instruction register). The bits that are set to 1 in the CIR form the set of control signals that need to be asserted at that time. Please be careful not to confuse the CPC with the PC, the CIR with the IR, and control-instructions with MAL instructions and ML instructions.

Normally, the CPC is incremented at the processor clock edge, causing successive control-



| Step No.                                | Control Signals          |  |
|---|--------------------------|--|
|   | for Control Unit         | for Data Path  |
| <i>Fetch phase of every instruction</i> |                          |  |
| 0                                       |                          | PC_out, MAR_in, AIR_in                                     |
| 1                                       |                          | MemRead, 4_out, MDR_select, ALU_select=Add, MDR_in, AOR_in |
| 2                                       |                          | MDR_out, IR_in   |
| 3                                       | goto $S_n$               | AOR_out, PC_in   |
| <i>Execute phase of addu</i>            |                          |  |
| 4                                       |                          | R_select=01, R_out, AIR_in                                 |
| 5                                       |                          | R_select=10, R_out, ALU_select=Add, AOR_in                 |
| 6                                       | goto S0                  | AOR_out, R_select=11, R_in                                 |
| <i>Execute phase of lw</i>              |                          |  |
| 7                                       |                          | R_select=01, R_out, AIR_in                                 |
| 8                                       |                          | SE_out, ALU_select=Add, AOR_in                             |
| 9                                       |                          | AOR_out, MAR_in  |
| 10                                      |                          | MemRead, MDR_select, MDR_in                                |
| 11                                      | goto S0                  | MDR_out, R_select=10, R_in                                 |
| <i>Execute phase of beq</i>             |                          |  |
| 12                                      |                          | R_select=01, R_out, AIR_in                                 |
| 13                                      |                          | R_select=10, R_out, ALU_select=Sub, Z_in                   |
| 14                                      | if ( $\bar{Z}$ ) goto S0 | SE_out, AIR_in   |
| 15                                      |                          | ALU_select=Shift2, AOR_in                                  |
| 16                                      |                          | AOR_out, AIR_in  |
| 17                                      |                          | PC_out, ALU_select=Add, AOR_in                             |
| 18                                      | goto S0                  | AOR_out, PC_in   |

Table 10.6: Control Program to Interpret 3 MIPS ML Instructions

instructions to be read from the control memory. Every time a new ML instruction is loaded into the IR, the combinational logic block called *next address generator* generates a different control program address to be loaded into the CPC. Thus, the control signals are generated in the correct sequence. Notice that the control sequence corresponding to instruction fetch is same for all ML instructions, and is stored only once.

The programmed control unit can be viewed as an interpreter for the nanoprogram. We have again applied the engineering lesson gleaned from the model of Turing universality: To build a complex system  $A$ , build a smaller subsystem  $B$  that interprets a coded representation  $A'$  of  $A$ .

The extra level of interpretation performed by the control ROM provides significant flexibility to implement powerful features in the immediately higher level machine, namely the microarchitecture. Indeed, the interpretive power of a programmable control unit lets the microarchitecture to be quite flexible. Of course, there is a performance cost associated with this extra interpretation; the logic-level machine could be made to execute MAL instructions faster if they were interpreted directly by the hardware (random logic) rather than by control-ROM instructions. For this reason, a programmed control unit is not used

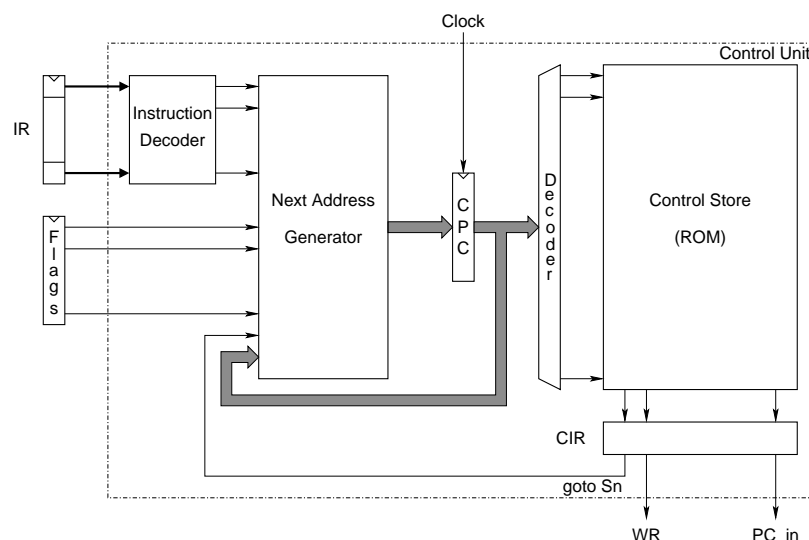


Figure 10.34: A Programmed Control Unit

in modern high-performance processors.

## Horizontal and Vertical Control Codes

The kind of control code discussed so far, in which the control-instructions have a separate bit for each control signal, is called *horizontal microcode*. Complex processors have a large number of control signals. Thus, the horizontal approach requires a wide control memory. Moreover, only a few bits are set to 1 in any given control-instruction, which means that the control memory is very sparse in terms of the number of 1s. Notice that many of these control signals are not needed simultaneously, and some of the signals are mutually exclusive. For example, the contents of only one register can be gated to the bus in a cycle; therefore signals `PC_out`, `MDR_out`, `AOR_out`, `SE_out`, and `R_out` are mutually exclusive. This suggests that control signals can be grouped so that all mutually exclusive signals are placed in the same group. Thus, at most one control signal per group is specified in any control-instruction. Then it is possible to use a binary coding scheme to represent the control signals within a group. For example, the 5 register output control signals `PC_out`, `MDR_out`, `AOR_out`, `SE_out`, and `R_out` can be placed in a 3-bit group. Any one of these can be selected by a unique 3-bit code.

Further natural groupings can be made for the remaining signals. Figure 10.35 shows an example of a partial format for the control-instructions, in which each group occupies a field large enough to contain the required codes. Most fields must include one inactive code for

the case in which no action is required. For instance, the all-zero pattern in the *register-out* field indicates that none of the registers that may be specified in this field should have its contents placed on the bus.

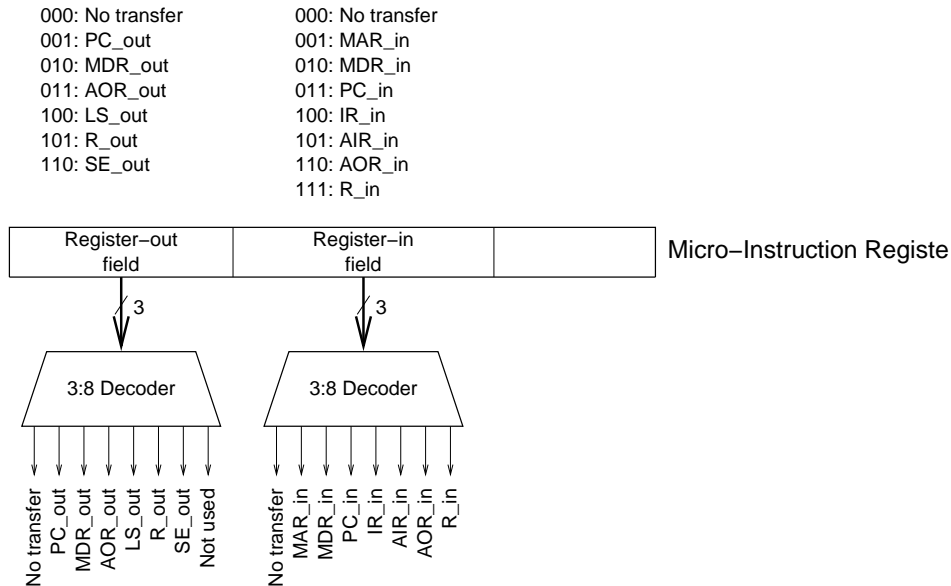


Figure 10.35: A Semi-Vertical Encoding of Control Code

Notice, however, that reducing the width of the control-instruction is at the expense of an additional level of decoding after the CIR. If control unit speed is a concern, then this type of encoding should not be used.

So far, we have considered grouping and encoding only mutually exclusive control signals. We can extend this idea by enumerating the bit patterns corresponding to all meaningful control-instructions. Each meaningful control-instruction can then be assigned a distinct binary code that represents the control-instruction. Such full encoding is called *vertical control code*. This will further reduce the width of the control memory, but will also increase the complexity of the decoder circuits that are placed after the CIR, as shown in Figure 10.36.

### 10.7.2 Hardwired Control Signal Generator: A Fast Control Mechanism

The programmed control unit leads to a very flexible approach, and allows new ML instructions to be introduced with relatively less changes to the control unit. Next, we will look at a faster design for the control unit. In this approach, called hardwired control, the control store (ROM) is replaced by random logic. Thus, each of the control signals is generated by

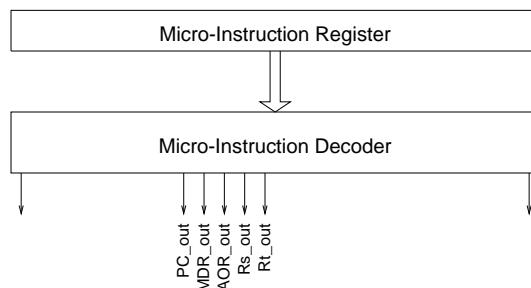


Figure 10.36: Vertical Encoding of Control Code

writing the appropriate Boolean equation, and designing the logic circuitry to generate the signal.

The Boolean equations for each control signal can be determined by the designer by combing through the control sequences of each machine language instruction, looking for every occurrence of that control signal. Let us look at the Boolean equations to produce some of the control signals.

$$\begin{aligned}
 \text{PC\_out} &= S0 + S17 \\
 \text{PC\_in} &= S3 + S18 \\
 \text{MDR\_in} &= S1 + S10 \\
 \text{goto } S0 &= S6 + S11 + \bar{Z}.S14 + S18 \\
 \text{goto } S7 &= \text{lw}.S3
 \end{aligned}$$

There are different ways to design a hardwired control unit. In a slightly different approach, the control sequences are written as in Table 10.7. The advantage of this approach is that the control steps range only from 0 to a small number, 10 in this example.

If we design the hardwired control unit following this approach, we get a control unit as given in the Figure 10.37. The **state** of the finite state machine implementing the control unit is determined by the contents of the *control step counter* (CSC). This counter has provision to do the following two functions: (i) incrementing state (count value), and (ii) resetting the state to zero. Normally, the control step counter is incremented at each clock pulse. When the counter reaches the last step for a particular machine language instruction, it is reset to zero, so that the processor can begin fetching the next machine language instruction.

The outputs of the control unit (which are the various control signals needed to manipulate data and move it from block to another) are produced by the *control signal generation logic* based on (i) the contents of the control step counter, (ii) the outputs of the instruction decoder, and (iii) the contents of the flags register. The Boolean equations for each control

| Step                             | Control Signal Assertions                            |                              |                                      |
|----------------------------------|--|------------------------------|--------------------------------------|
| Fetch phase of every instruction |  |                              |                                      |
| 0                                | PC_out, MAR_in, AIR_in                               |                              |                                      |
| 1                                | MemRead, 4_out, MDR_sel, ALU_sel=Add, MDR_in, AOR_in |                              |                                      |
| 2                                | MDR_out, IR_in                                       |                              |                                      |
| 3                                | AOR_out, PC_in                                       |                              |                                      |
| Execute phase of addu            |  | Execute phase of lw          | Execute phase of beq                 |
| 4                                | R_sel=01, R_out, AIR_in                              | R_sel=01, R_out, AIR_in      | R_sel=01, R_out, AIR_in              |
| 5                                | R_sel=10, R_out, ALU_sel=Add, AOR_in                 | SE_out, ALU_sel=Add, AOR_in  | R_sel=10, R_out, ALU_sel=Sub, Z_in   |
| 6                                | AOR_out, R_sel=11, R_in, End                         | AOR_out, MAR_in              | SE_out, AIR_in, if ( $\bar{Z}$ ) End |
| 7                                |  | MemRead, MDR_sel=1, MDR_in   | ALU_sel=Shift2, AOR_in               |
| 8                                |  | MDR_out, R_sel=10, R_in, End | AOR_out, AIR_in                      |
| 9                                |  |                              | PC_out, ALU_sel=Add, AIR_in          |
| 10                               |  |                              | AOR_out, PC_in, End                  |

Table 10.7: Control Signal Assertions to Interpret 3 MIPS Instructions

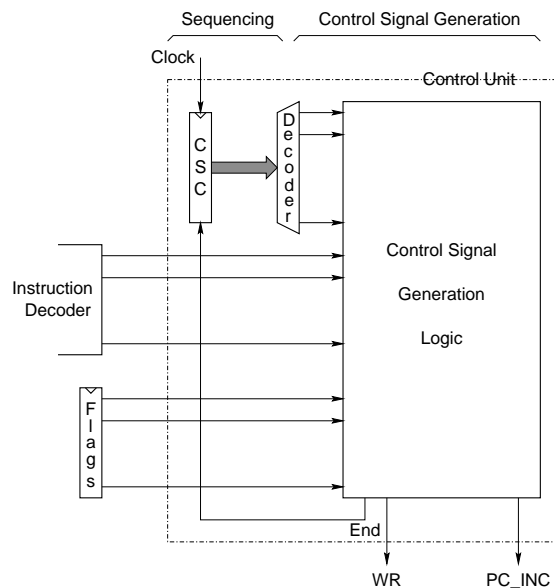


Figure 10.37: Block Diagram of a Hardwired Control Unit

signal can be determined by the designer by combing through the control sequences of each machine language instruction, looking for every occurrence of that control signal. Let us look at the Boolean equations to produce some of the control signals.

$$\begin{aligned}
 \text{PC\_out} &= \text{S0} + \text{beq} \cdot \text{S9} \\
 \text{PC\_in} &= \text{S3} + \text{beq} \cdot \text{S10} \\
 \text{MDR\_in} &= \text{S1} + \text{lw} \cdot \text{S7}
 \end{aligned}$$

```
End           =  add.S6 + lw.S8 + beq.Z.S6 + beq.S10
```

The use of random logic allows hardwired control unit to be fast; however, this approach is not very flexible. If a new machine language instruction has to be added at the later stages of the design, then most of the control unit logic has to be redesigned.

### 10.7.3 Hardwired versus Programmed Control Units

The hardwired and programmed methods form two radically different approaches to control unit design. Each has advantages and disadvantages when performance and cost are compared:

- **Speed:** Hardwired control units are faster than programmed control units. They have a latency of a just a few gate delays, considerably less than that of programmed control units, which must perform a memory fetch for each control step.
- **Ease of prototyping:** The programmed control unit wins here, because it is generally easier to reprogram a ROM than to redesign random logic.
- **Flexibility:** Once again, programmed control unit is superior when the designer wishes to make changes to the instruction set.

The characteristics of the two approaches, as summarized in the preceding list, tend to favor hardwired control units when a processor is to be put into high-volume production, and to favor programmed control units when developing prototypes or when manufacturing small quantities.

**A Little Bit of History:** As with most of the concepts underlying computer design, the concept of programmed control units originated early in the history of computing. Maurice Wilkes proposed the concept in 1951, as a way of simplifying the control logic of the computer. Although Wilkes did construct a machine, the EDSAC2, with a programmed control unit, it was not until the early 1960s that IBM made the concept popular with the 360 line of computers, which used programmed control units in a big way. The technique saw widespread usage from then until the late 1980s, when hardwired control units again became popular, largely because of their faster speed and the greater availability of CAD tools for logic design. Programmed control units may well again become popular, as implementation domains change.

## 10.8 Concluding Remarks

## 10.9 Exercises

1. Design a 5-bit look-ahead carry adder. Write all of the Boolean equations for implementing such an adder, and draw the logic diagram. If the maximum fan-in (i.e., number of inputs) allowed for a gate is 4, what is the worst-case delay of your adder?
2. Design a *full subtractor* module in a manner similar to the design of a full adder, and connect these modules to form an  $N$ -bit subtractor, in a manner similar to the design of an  $N$ -bit ripple carry adder. What is the worst-case delay of your subtractor?
3. Which of the following arithmetic operations will cause an overflow (in an 8-bit 2's complement number system)?
4. Which is faster—hardwired control or microprogrammed control? Explain.
5. Draw the complete timing diagram to do a WRITE operation over an asynchronous bus. Clearly mark all signals, and the times at which different steps take place.
6. Draw the hardware to do multiplication of two 8-bit signed integers, and explain how it works.
7. Design a  $64\text{K} \times 8$  memory system using  $16\text{K} \times 4$  RAM chips.

## Appendix A

# MIPS Instruction Set

*Gold there is, and rubies in abundance, but lips that speak knowledge are a rare jewel.*

**Proverbs 20: 15**





# Appendix B

## Peripheral Devices

*Gold there is, and rubies in abundance, but lips that speak knowledge are a rare jewel.*

**Proverbs 20: 15**

In the preceding chapters, we discussed information representation, information manipulation, instruction set processing, datapath and control unit design, and memory organization. It is not sufficient to do computations; the results of the computation must be communicated to *other systems* and *humans*. This communication capability allows a user, for example, to enter a program and its data via the keyboard of a video terminal and receive results on a display or a printer. A computer may need to communicate with a variety of equipment, such as video terminals, printers, and plotters, as well as magnetic disks. In addition to these standard IO devices, a computer may be connected to other types of equipment. For example, in industrial control applications, input to a computer may be the digital output of a voltmeter, a temperature sensor, or a fire alarm. Similarly, the output of a computer may be a digitally coded command to change the speed of a motor, open a valve, or cause a robot to move in a specified manner. A general-purpose computer should have the ability to deal with a wide range of device characteristics in varying environments.

We illustrate just three IO devices in detail: video terminal, printer, and hard disk.

### B.1 Types and Characteristics of IO Devices

The IO system of a computer provides an efficient mode of communication between the processor and the outside environment. Programs and data must be entered into main memory for processing, and results obtained from computations must be recorded or displayed. Among the IO devices that are commonly found in computer systems are keyboards, monitors, printers, magnetic disks, and CD-ROMs. Other input and output devices frequently

encountered are modems or other communication interfaces, magnetic tape drives, scanners, speakers, and microphones. Significant numbers of computers, such as those used in automobiles, have analog-to-digital converters, digital-to-analog converters, and other data acquisition and control components.

The IO facility of a computer is a function of its intended application. This results in a wide diversity of attached devices and corresponding differences in the needs for interacting with them. Three characteristics are useful in organizing this wide variety:

- Behavior: Input (read once), output (write only, cannot be read), or storage (can be reread and usually rewritten).
- Partner: Either a human or a machine is at the other end of the peripheral device. If the partner is human, then the information format must be human-friendly.
- Data rate: The peak rate at which data can be transferred between the peripheral device and the main memory or processor.

| IO Device        | Behavior        | Partner | Data Rate (KB/sec) |
|------------------|-----------------|---------|--------------------|
| Keyboard         | Input           | Human   | 0.01               |
| Mouse            | Input           | Human   | 0.02               |
| Voice input      | Input           | Human   | 0.02               |
| Scanner          | Input           | Human   | 200.00             |
| Voice output     | Output          | Human   | 0.60               |
| Line printer     | Output          | Human   | 1.00               |
| Floppy disk      | Storage         | Machine | 50.00              |
| Laser printer    | Output          | Human   | 100.00             |
| Graphics display | Output          | Human   | 30,000.00          |
| Network terminal | Input or output | Machine | 0.05               |
| Network LAN      | Input or output | Machine | 200.00             |
| Optical disk     | Storage         | Machine | 500.00             |
| Magnetic tape    | Storage         | Machine | 2,000.00           |
| Magnetic disk    | Storage         | Machine | 2,000.00           |

Table B.1: The Diversity of IO Devices

The above table shows some of the IO devices that are commonly connected to computers, and their characteristics. We will briefly look at three of the common IO devices: video terminal, printer, and magnetic disk.

## B.2 Video Terminal

Video terminals are probably the most commonly used IO devices. They consist of a keyboard as the input device, usually complemented by a trackball or mouse device, and

a video display as the output device. They are used for direct human interaction with

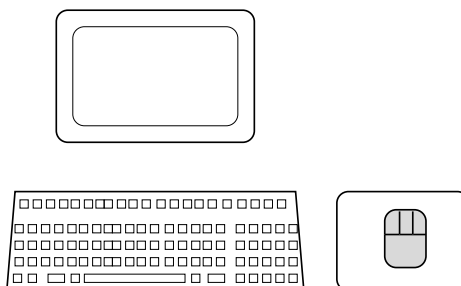


Figure B.1: Video Terminal

the computer—for entering programs and data, and receiving results of computation in an interactive environment. The input entered through the keyboard is usually displayed on the screen, so that the user can see what is being transmitted. The basic unit of exchange is a **character**. Characters are of two types: displayable and control. Displayable characters are the alphabetic, numeric, and special characters that the user can enter at the keyboard and that can be displayed on the screen. Control characters are interpreted to cause a device-specific action, such as carriage return.

### B.2.1 Keyboard

The keyboard is one of the simplest electromechanical devices attached to a general-purpose computer. It predates the computer, having been used in telegraphy since the development of the Teletype in the 1920s. The keyboard consists of a collection of keys that can be depressed by the user. Most keyboards still lay out the keys in the same manner as the traditional layout used in typewriters. Each time the user depresses or releases a key, the keyboard circuitry detects this action; the keyboard makes a distinction between *depressing* a key and *releasing* a key. In order to identify the key being pressed or released, a 2-dimensional *scan matrix* is laid out beneath the keys. Figure 8.20 shows a scan matrix arranged in an  $8 \times 16$  layout.

Figure: Keyboard Interface and Keyboard Scan Matrix

A decoder drives the  $X$  lines of the matrix, and a multiplexer is attached to the  $Y$  lines of the matrix. The decoder and the multiplexer are controlled by a microcontroller, a tiny computer that contains a RAM, a ROM, a timer, and simple IO interface logic. The microcontroller is programmed to periodically scan all intersections in the matrix by placing the appropriate signal values in the decoder inputs and the multiplexer control inputs. When a key is depressed, a signal path is closed from the decoder's output line corresponding to the pressed key to the multiplexer's input line corresponding to the pressed key. In this figure, a 7-bit code (3-bit decoder input and 4-bit multiplexer control input) can uniquely identify the pressed key. To this 7-bit code, the keyboard microcontroller appends a 0 at the most significant position to indicate that the key has been *depressed* (as opposed to not being *released*). The resulting code is called a *K-scan* code. In this case, it is the key's *make code*. When the key is released, the K-scan code produced by the microcontroller is the key's *break code*, which differs from its *make code* only in the most significant bit. The K-scan code produced by the keyboard microcontroller is transferred serially to the keyboard controller/interface.

In and of itself, a make code value or break code value has no inherent meaning. The codes generated by an English keyboard, for example, can be identical to the codes generated by a German, French, or Italian keyboard. It is up to the OS to attach meaning to these codes and convert them, for instance, to ASCII or other meaningful codes.

It is important to note that with the scanning scheme mentioned, the entire keyboard matrix has to be scanned on a frequent basis so as not to miss any characters typed. Keyboard microcontrollers typically perform a complete scanning hundreds of times per second. Next time you are at a keyboard, see if you can beat the keyboard microcontroller by typing fast!

### B.2.2 Mouse

The mouse is a common input device attached to modern computers. It is used to point to objects on the screen. With the advent of window-based operating systems, such as the Macintosh OS, the UNIX Xterm, and Windows '95, the importance of mouse has increased dramatically. The basic unit of exchange from the mouse to the processor is the X-coordinate and Y-coordinate of the mouse's current position. The mouse can keep track of its current position with the help of two counters that can be incremented/decremented as the mouse is moved. As with the case of keyboard, any movement of the mouse is also displayed on the screen so that the user can know the current position of the mouse. Most mice also include one or more buttons, and the system must be able to detect when a button is depressed. By monitoring the status of the button, the system can also differentiate between clicking the button and holding it down.

### B.2.3 Video Display

The video display or monitor is the primary output device for the interactive use of computers. Displays use a number of different technologies, the most prevalent of which is currently the cathode-ray tube (CRT). These displays descended from the television set, which first came into common use nearly 50 years ago. The video monitor used in computers has a higher spatial resolution than the monitor in the TV set. Spatial resolution is usually measured as the number of *dots per inch (dpi)* that can be distinguished on the screen.

The video monitor used for computer applications may be color or black and white. We shall describe color monitors because they are more prevalent. Figure B.2 shows a simplified view of the operation of an RGB video monitor. The CRT display is defined in terms of picture elements called *pixels*. At each pixel location on the screen, there are 3 small phosphor dots, one that emits red light, one that emits green light, and one that emits blue light when excited by a beam of electrons. In order to excite the 3 phosphors simultaneously, 3 electron guns are used, with each gun focused on its own color phosphor dot. The color that results for a given pixel is determined by the intensity of the electron beams striking the phosphors within the pixel.

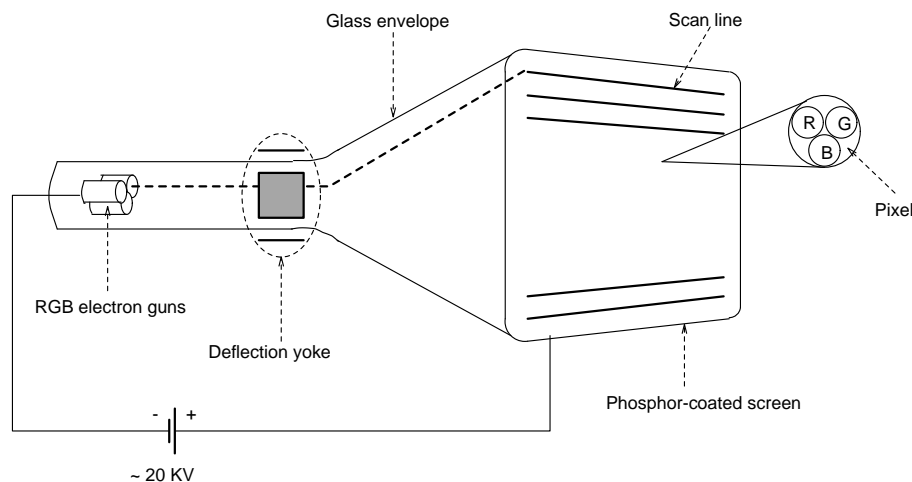


Figure B.2: Schematic View of RGB Video Monitor

The input signals contain pixel color intensities and vertical and horizontal synchronization (sync) information. Circuitry within the monitor extracts the pixel intensity information, and presents it to the RGB electron guns of the CRT. The 3 electron guns emit 3 streams of electrons proportional to the RGB intensities. The electrons are accelerated toward the screen by a high voltage power supply. When the electron beams strike the screen, they cause the respective phosphor coatings to glow for 1-10 ms.

The beams are deflected, or swept, from left to right and top to bottom by the horizontal

and vertical sweep, or deflection, currents applied to the deflection yoke shown at the neck of the tube.

## B.3 Printer

Printers are used to produce hard copy of output data or text. Although people once thought that the use of printers would decline with the development of high-quality displays and the “paper-less” office concept, the long lines of users and the huge stack of wasted paper near printers indicate that the contrary is indeed true! Printers are usually classified as either *impact* or *nonimpact* type, depending on the nature of the printing mechanism used. Impact printers use mechanical printing mechanisms, and non-impact printers rely on optical, ink-jet, or electrostatic techniques. Mechanical printers can be only as fast as their mechanical parts allow, with speeds upwards of 1000 lines per minute being attainable. Considerably higher speeds can be achieved with nonimpact printers, where printing several thousand lines per minute is possible in large, high-speed printers. We shall consider only nonimpact printers.

**Laser Printer:** Laser printers use photocopying technology. The image to be printed is raster-scanned onto a rotating drum coated with positively charged photoconductive material using a low-powered laser beam. The positive charges that are illuminated by the beam are neutralized. The drum then passes through a bath of finely-divided, negatively charged powder known as *toner*. Toner adheres to the drum where the positive charge remains, thus creating a page image that is then transferred to the paper by heat and pressure. As the last step, the drum is cleaned of any excess toner material to prepare it for printing the next page. Laser printers provide high resolution, ranging from 300 dpi to 1200 dpi.

**Ink-Jet Printer:** In an ink-jet printer, the print head contains a cartridge(s) containing viscous inks. An extremely small jet of electrically charged ink droplets from this cartridge is fired through a nozzle onto the paper. The droplets are deflected by an electric field to trace out the desired character patterns on the paper they strike. Several nozzles with different color inks can be used to generate color output.

**Fonts:** Most printers form characters and graphic images in the same way that images are formed on a video screen. In the past, printers formed characters very much like alphanumeric display terminals. They accepted an ASCII character stream, and printed it in the font that was stored in the *character ROM* on board the printer. The quest for letter-quality ( $\geq 300$  dpi) output led to the development of the PostScript page description language by John Warnock, founder of Adobe Systems. PostScript printers have on-board PostScript processors that allow them to accept high-level PostScript print commands.

Fonts can be downloaded from the computer to the printer, and the printer can render the fonts at any specified point size.

## B.4 Magnetic Disk

Magnetic disks are secondary storage devices used for storing large volumes of data permanently. There are two major types of magnetic disks: *floppy disks* and *hard disks*. Both types rely on a rotating platter coated with a magnetic surface and use a movable read/write head to access the disk. The primary differences arise because the floppy disk is made of a mylar substance that is flexible, while the hard disk uses metal (or, recently glass). Floppy disks can thus be removed and carried around, while most hard disks today are not removable. Because the platters in a hard disk are metal, they have several advantages over floppy disks:

- The hard disk can be larger, because it is rigid.
- The hard disk can be controlled more precisely, and so it has higher density.
- The hard disk can be spun faster, and so it has a higher data transfer rate.
- The hard disk can incorporate multiple platters.

A hard disk consists of a collection of platters (1 to 20), each of which has two recordable disk surfaces, as shown in Figure B.3. Each platter is physically similar to a phonograph record. The stack of platters is rotated at 3600 to 7200 RPM and has a diameter of just over an inch to just over 10 inches. Each disk surface is divided into concentric circles, called *tracks*. There are typically 500 to 2000 tracks per surface. Each track is in turn divided into *sectors* that store information; each track may have 32 sectors. The sector is the smallest unit that is read or written, so as to amortize the large amount of time required to get to the required sector. Notice that, unlike the main memory, the disk does not inherently associate an address with the data stored: the identifying information must be stored along with the data or elsewhere.

To enable information to be accessed, a set of read/write heads is mounted on an actuator that can move the heads radially over the disk. The time required to move the heads from the current track to the desired track is called the *seek time*. The time required to rotate the disk from its current position to that having the desired sector under the head is called the *rotational delay*. In addition, a certain amount of time is required by the disk controller to access and output information. This time is called the *controller time*. Once the required sector is located, it takes some time to transfer the sector of data. This time is called the *transfer time*. The time required to locate and read a sector is the *disk access time*, which is the sum of the controller time, the seek time, the rotational delay, and the transfer time. The transfer time for a sector is equal to the proportion of the track occupied by the sector divided by the rotational speed of the disk.



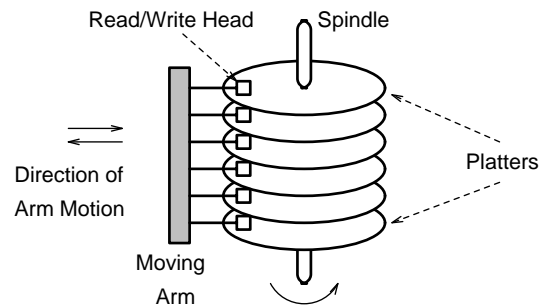


Figure B.3: A Multi-platter Hard Disk Drive

*Example:* What is the average time to read or write a 1024-byte sector from a magnetic disk that has an average seek time of 10 ms, rotates at 7200 RPM, and has a transfer rate of 8 MB/sec. The controller overhead is 1 ms.

As mentioned above, each disk has movable arms that position somewhere on the surface to read or write data. Each disk has its own position. When copying data from one disk to another, the arms on each disk can remain fixed in their respective places. By contrast, when copying data from one file to another on the same disk, the arms of the disk may move back and forth, resulting in much longer transfer times. So if there is room in the system unit, it is better to use a few small disks than one large disk.

## B.5 Modem

# Bibliography

- [1] A. W. Burks, H. H. Goldstine, and J. von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," Report prepared for U.S. Army Ordnance Department, 1946.
- [2] J. P. Hayes, *Computer Architecture and Organization*, 3rd Edition. WCB McGraw-Hill, 1998.
- [3] H. D. Huskey and V. R. Huskey, "Chronology of Computing Devices," *IEEE Transactions on Computers*, Vol. 25, pp. 1190-1199, 1976.
- [4] D. E. Knuth, *The Art of Computer Programming*, Vol. 1 (Fundamental Algorithms), 2nd Edition. Reading, Massachusetts: Addison-Wesley Publishing Co., 1973.
- [5] S. Rosen, "Electronic Computers: A Historical Survey," *Computing Surveys*, Vol. 1, pp. 7-36, 1969.
- [6] E. Squires, *The Computer — An Everyday Machine*. Reading, Massachusetts: Addison-Wesley Publishing Co., 1977.
- [7] M. V. Wilkes, "Computers Then and Now," *Journal of the ACM*, Vol. 15, pp. 1-7, 1968.
- [8] "A History of MTS," *Information Technology Digest*, Vol. 5, No. 5.

“From coupler-flange to spindle-guide I see Thy Hand, O God—  
Predestination in the stride o’ yon connectin’-rod.”

—R. Kipling

# Index

# Index

- Abstract machine interface, 69
- Accumulator, 86, 132
- Activation record, 114
- Address Domain Isolation, 337
- Addressing mode, 136
  - Autoincrement addressing, 139
  - Immediate addressing, 137
  - Implicit addressing, 140
  - Indexed addressing, 139
  - Memory direct addressing, 138
  - Memory indirect addressing, 140
  - PC-relative addressing, 139
  - Register addressing, 137
  - Register indirect addressing, 138
- Application binary interface (ABI), 69, 106
  - Spim, 104
- Application programming interface (API), 70, 72, 73, 76, 149, 153, 154, 164
  - UNIX, 73, 75, 163
- Arithmetic Logic Unit (ALU), 405
- Arithmetic logic unit (ALU), 246, 363
- Assembler, 31, 38, 112, 191
- Assembly language, 38, 83
  - Assembler directive, 89, 98
    - .align, 103
    - .ascii, 103
    - .asciiz, 103
    - .byte, 102
    - .comm, 102
    - .data, 102, 106
    - .float, 103
    - .half, 102
    - .rdata, 102
    - .space, 103
    - .text, 102, 106
    - .word, 102
  - Label, 98, 146
  - Macro, 146
  - Programmability, 146
  - Rationale, 95
- Assembly-level architecture, 38, 87, 97
- Autoconfiguration, 330
- BCD format, 88, 89
- Benchmark program, 26
- Bluetooth
  - Passkey, 351
- Boot up, 20
  - Boot program, 170
  - Bootstrap loader, 21
- Branch misprediction, 282
- Branch misprediction penalty, 282
- Branch penalty, 281
- Branch prediction, 282
- Branch target buffer (BTB), 283
- Bus, 247
  - Adapter, 337
  - Bridge, 335
  - Controller, 337
  - Expansion bus, 338
  - Expansion slot, 338, 340
  - External bus, 338
  - Host bus, 338
  - Local bus, 338
  - Non-transparent bus, 336, 337
  - Parallel bus, 338
  - Serial bus, 337
  - Transparent bus, 336

- Byte ordering, 131
- Cache memory, 263, 383
  - Address mapping, 264
  - Cache block, 263
  - Performance, 264
  - Replacement policy, 268
  - Tag, 265
- calloc(), 116
- Character, 89
- Command line interface (CLI), 223
- Compiler, 11, 107
- Computer network
  - Network architecture, 357
- Condition code, 86
- Conditional branch, 91
- Configuration address space, 330
- Configuration memory, 330
- Control abstraction, 68, 141
- Control hazard, 281
- Control unit, 243, 375
- Critical path, 407
- Cross assembler, 196
- Data hazard, 285
- Data path, 243
- Data type, 87
- Delayed branch, 171, 259, 284, 373
- Design specification, 238
- Device controller, 19
- Device driver, 20, 77, 165
- Device-independent IO, 59, 76, 164
- Device-level architecture, 40
- Direct memory access (DMA), 181
- Direct path, 247
- Dynamic linking, 228
  - Delay loading, 229
  - DLL, 228
  - Load-time, 228
  - Run-time, 229
- Dynamically linked library (DLL), 228
- Endian
  - Big endian, 131
  - Little endian, 131
- Ethernet, 349
  - Collision, 350
- Exception, 157
  - exception vector, 157, 168
- Expanding opcode, 202
- Expansion bus, 338
- Expansion card, 340
- Expansion slot, 338, 340
- External bus, 338
- Field-based encoding, 201, 207
- File, 58
- File system, 76, 164
- Finite state machine (FSM), 377
- FireWire, 348
- Fixed field encoding, 201
- Floating-point, 88
- Frame buffer, 325
- Framepointer, 125
- free(), 116
- Front Side Bus (FSB), 269
- Functional simulator, 232
- Functional unit, 246, 363
- General handler, 170
- Graphical user interface (GUI), 223, 224
- Hardware abstraction layer (HAL), 78
- Hardware description language (HDL), 237
- Hardware port, 340
- Hardware specification, 238
- Header, 188
- Heap, 116
- Hierarchical bus, 333
- High-level architecture, 37
- Host bus, 338
- Host machine, 232
- Hot plug, 331
- Hot swap, 331
- Immediate addressing, 137

- Immediate Operand, 137
- Immediate operand, 131
- Instruction register (IR), 364
- Instruction set, 90
- Instruction set architecture (ISA), 38, 185
- Instruction-Residing Operand, 137
- Interconnect, 247
- Internal bus, 338
- Interpreter, 32, 243, 375
- Interrupt
  - Device identification, 180
  - Device interrupt, 156
  - interrupt handler, 156
  - interrupt service routine (ISR), 156
  - Interrupt-driven IO, 177
  - Priority, 180
- IO
  - Autoconfiguration, 330
  - Configuration address space, 330
  - Configuration memory, 330
  - Expansion card, 340
  - Expansion slot, 340
  - Hardware port, 340
  - Hierarchical bus, 333
  - Plug-and-play, 330
- IO addressing
  - Independent IO, 158
  - Memory mapped IO, 158
- IO channel, 182
- IO data register, 160
- IO port, 159
- IO processor, 182
- IO programming interface, 159
- IO status register, 160
- IO stream, 58
- IO synchronization, 173
- ISA-visible register, 245
- Java
  - applet, 232
  - bytecode, 232
  - JVM, 136, 232
- JIT Compiler, 129
- Jump table, 121, 170
- Kernel mode, 149, 150
  - Switching to kernel mode, 153
- Keyboard interface, 174
- Label, 84
- Library, 58, 69
- Linear memory, 129
- Link register, 86, 143
- Linker, 197
- Linking, 197
- LISP machine, 37
- Live register, 143
- Load-store architecture, 103, 132
- Loading, 227
- Local bus, 338
- Look-ahead carry adder, 408
- Loop, 66
- LRU (least recently used) replacement, 269
- Machine language, 38, 186, 191
- malloc(), 116
- Memory address space, 84, 101, 129
- Memory management system (MMS), 298
- Memory management unit (MMU), 298
- Memory model, 83, 101
- Memory subsystem, 249
- Micro-assembler, 376
- Micro-assembly language (MAL), 244, 249
  - MAL command, 250, 253
  - MAL command set, 253
  - MAL operation, 250
  - MAL routine, 254, 255
- Microarchitectural register, 245
- Microinstruction, 280, 375, 394
- Microroutine, 376
- MIPS
  - UTLBMISS, 314, 317
- MIPS-I
  - Assembly directives, 102
  - Assembly-level instructions, 103

- MIPS-I ALA, 97
  - Assembly language conventions, 99, 101
  - Assembly language syntax, 97
- MIPS-I ISA, 190
- MIPS-I OS, 168, 170
- MIPS-I virtual memory, 312
- Multi-ported, 278, 392
- Multiple address machine, 132
- Network interface card (NIC), 358
- Non-transparent bus, 336, 337
- Object-oriented, 36
- Operand stack, 133
- Operating system
  - Command line interface (CLI), 223
  - Graphical user interface (GUI), 223, 224
  - Voice user interface (VUI), 223, 225
- Operating system (OS), 12, 13, 20, 71, 74, 162
- OS
  - Plug and play manager, 330
- Overflow, 86
- Page, 301
- Page fault, 301, 303, 307
- Page table, 301
  - Page table entry (PTE), 302, 304, 311, 315, 317
  - Root page table, 313
- Paging, 301
- Parallel bus, 338
- Parameter passing, 126
- PCI bus, 342
- Peripheral processing unit (PPU), 182
- Pipeline latch, 278, 393
- Pipeline register, 278, 393
- Pipelining, 274, 390
- Plug and play manager, 330
- Plug-and-play, 330
- pointer, 116
- Port
  - Read port, 274, 388
- Power consumption, 26
- Printer, 458
- Privileged instruction, 152
- Privileged register, 151, 294, 315
- Process scheduling, 157
- Processor subsystem, 249
- Processor-memory bus, 269
- Program counter (PC), 86
- Program sequencing, 91
- Program-controlled IO, 174
- Random access memory (RAM)
  - Dynamic RAM (DRAM), 404
  - Static RAM (SRAM), 403
- Register file, 244, 362
- Register model, 85
- Register transfer language, 362, 363
  - RTL instruction, 363, 367
  - RTL instruction set, 366
  - RTL operation, 363
  - RTL routine, 369
- Register transfer notation (RTN), 363
- Register window, 142, 145
- Register-memory architecture, 132
- Ripple carry adder, 408
- Sampling-based IO, 173
- SCSI bus, 344
- Segmented memory, 130
- Self-modifying code, 85, 123, 138
- Serial attached SCSI (SAS), 345
- Serial bus, 337
- Serial port, 326
- Shell, 222
- Shell script, 223
- Simulation, 231, 238
- Simulator
  - Instruction set simulator, 231
- Single address machine, 133
- Software driver, 20
- Software emulator, 231
- SPIM, 231



- Spim, 104, 105, 107
  - ABI, 104, 105
  - PCSpim, 107
  - Xspim, 107
- Stack frame, 85, 114, 125, 142
- Stack machine, 134
- Stack pointer (SP), 86
- Start-up routine, 198
- Storage device, 18
- Straight-line sequencing, 94
- Subroutine, 67
  - Callee save, 126, 143
  - Caller save, 126, 143
  - Epilogue code, 115
  - Linkage, 143
  - Nesting, 68, 143
  - Parameter passing, 126, 145
  - Prologue code, 114
  - Recursion, 68
- Symbol table, 117, 118, 194
- System call, 92, 154
  - Interpretation, 294
  - Syscall instruction, 92, 127, 154
  - Translation, 127
  - Trap instruction, 92, 127
- System call interface, 76
- System call layer, 154, 164
  - MIPS, 168
- System in package (SiP), 338
- System on chip (SoC), 338
- Timer, 157
- TLB
  - TLB exceptions, 314
  - UTLBMISS, 314, 317
- Total encoding, 201
- Transistor, 40
- Translation lookaside buffer (TLB), 306
  - Hardware-managed TLB, 307
  - Software-managed TLB, 309
- Translator, 32
- Transparent bus, 336
- Tri-state buffer, 405
- UNIX
  - API, 73
  - Kernel, 75, 163
  - Loading, 227
  - Object file, 197
  - TLB, 317
- USB, 345
- User mode, 81
- UTLB miss handler, 170, 317
- UTLBMISS, 314, 317
- Variable, 52, 110
  - Data type, 53
  - Dynamic variable, 115
  - Global variable, 111
  - Lifetime, 57
  - Local variable, 113
  - Scope, 55
  - Storage class, 57
- Verilog, 237
- VHDL, 237
- Video memory, 325
- Virtual machine, 30
- Virtual Memory
  - Implementation, 297
- Virtual memory
  - MIPS-I, 312
  - Page, 301
  - Page fault, 301
  - Paging, 301
  - Software-managed TLB, 309
- VLSI, 40
- VLSI architecture, 40
- Voice user interface (VUI), 223, 225
- Word alignment, 130
- Zero address machine, 135