Digital signal processor fundamentals and system design

M.E. Angoletta
CERN, Geneva, Switzerland

Abstract
Digital Signal Processors (DSPs) have been used in accelerator systems for more than fifteen years and have largely contributed to the evolution towards digital technology of many accelerator systems, such as machine protection, diagnostics and control of beams, power supply and motors. This paper aims at familiarising the reader with DSP fundamentals, namely DSP characteristics and processing development. Several DSP examples are given, in particular on Texas Instruments DSPs, as they are used in the DSP laboratory companion of the lectures this paper is based upon. The typical system design flow is described; common difficulties, problems and choices faced by DSP developers are outlined; and hints are given on the best solution.

1 Introduction

1.1 Overview

Digital Signal Processors (DSPs) are microprocessors with the following characteristics:

a) Real-time digital signal processing capabilities. DSPs typically have to process data in real time, i.e., the correctness of the operation depends heavily on the time when the data processing is completed.

b) High throughput. DSPs can sustain processing of high-speed streaming data, such as audio and multimedia data processing.

c) Deterministic operation. The execution time of DSP programs can be foreseen accurately, thus guaranteeing a repeatable, desired performance.

d) Re-programmability by software. Different system behaviour might be obtained by re-coding the algorithm executed by the DSP instead of by hardware modifications.

DSPs appeared on the market in the early 1980s. Over the last 15 years they have been the key enabling technology for many electronics products in fields such as communication systems, multimedia, automotive, instrumentation and military. Table 1 gives an overview of some of these fields and of the corresponding typical DSP applications.

Figure 1 shows a real-life DSP application, namely the use of a Texas Instruments (TI) DSP in a MP3 voice recorder–player. The DSP implements the audio and encode functions. Additional tasks carried out are file management, user interface control, and post-processing algorithms such as equalization and bass management.

Table 1: A short selection of DSP fields of use and specific applications

Field		Application
Communication	Broadband	Video conferencing / phone
		Voice / multimedia over IP
		Digital media gateways (VOD)
	Wireless	Satellite phone
		Base station
Consumer	Security	Biometrics
		Video surveillance
	Entertainment	Digital still /video camera
		Digital radio
		Portable media player / entertainment console
	Toys	Interactive toys
		Video game console
Industrial and entertainment	Medical	MRI
		Ultrasound
		X-ray
	Point of sale	Scanner
		Vending machine
	Industrial	Factory automation
		Industrial / machine / motor control
		Vision system
Military and aerospace		Guidance (radar, sonar)
		Avionics
		Digital radio
		Smart munitions, target detection

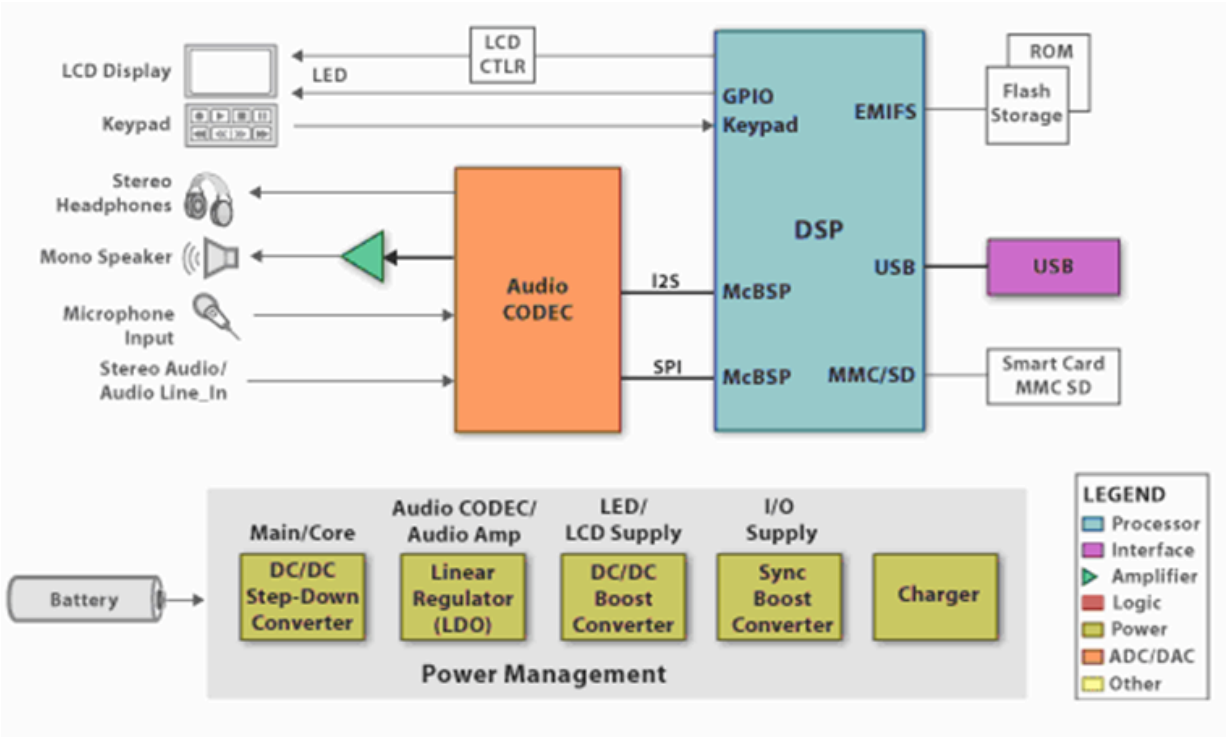


Fig. 1: Use of Texas Instruments DSP in a MP3 player/recorder system. Picture courtesy of Texas Instruments from www.ti.com.

1.2 Use in accelerators

DSPs have been used in accelerators since the mid-1980s. Typical uses include diagnostics, machine protection and feedforward/feedback control. In diagnostics, DSPs implement beam tune, intensity, emittance and position measurement systems. For machine protection, DSPs are used in beam current and beam loss monitors. For control, DSPs often implement beam controls, a complex task where beam dynamics plays an important factor for the control requirements and implementations. Other types of control include motor control, such as collimation or power converter control and regulation. The reader can find more information on DSP applications to accelerators in Refs. [1–3].

DSPs are located in the system front-end. Figure 2 shows CERN's hierarchical controls infrastructure, a three-tier distributed model providing a clear separation between Graphical User Interface (GUI), server, and device (front-end) tiers.

DSPs are typically hosted on VME boards which can include one or more programmable devices such as Complex Programmable Logic Devices (CPLDs) or Field Programmable Gate Arrays (FPGAs). Daughtercards, indicated in Fig. 2 as dashed boxes, are often used; their aim is to construct a system from building blocks and to customize it by different FPGA/DSP codes and by the daughtercards type. DSPs and FPGAs are often connected to other parts of the system via low-latency data links. Digital input/output, timing, and reference signals are also typically available. Data are exchanged between the front-end computer and the DSP over the VME bus via a driver.

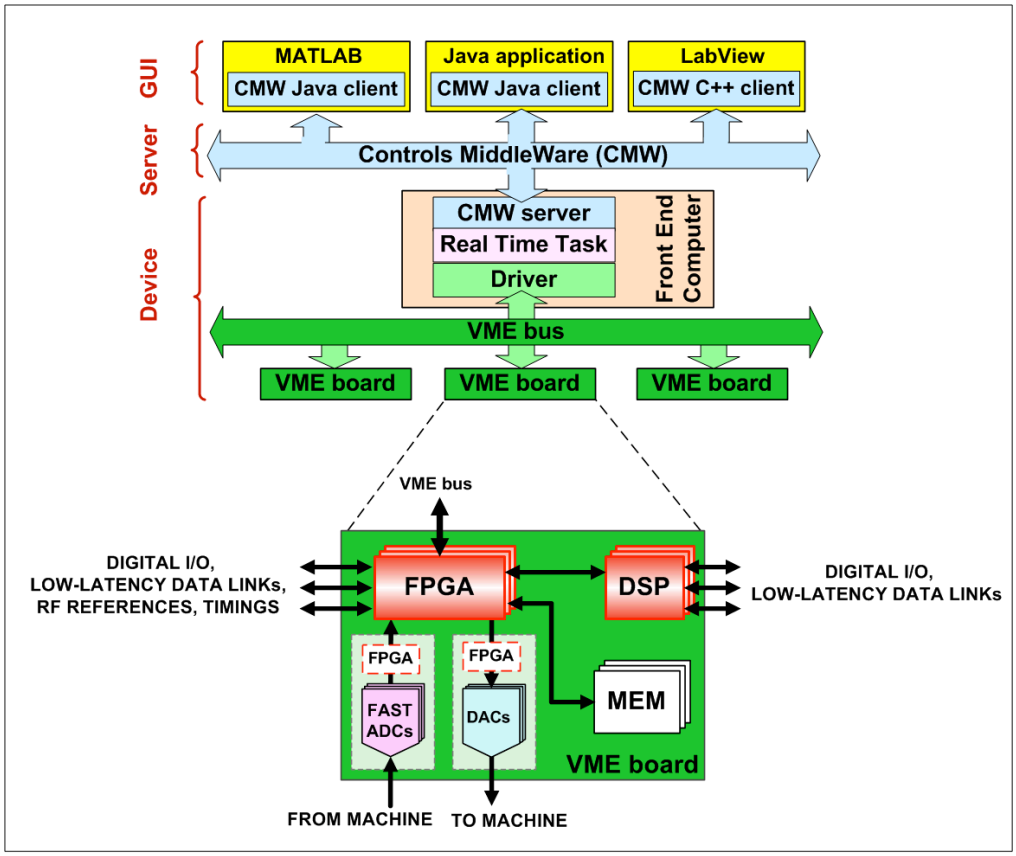


Fig. 2: Typical controls infrastructure used at CERN and DSP characteristics location

2 DSP evolution and current scenery

DSPs appeared on the market in the early 1980s. Since then, they have undergone an intense evolution in terms of hardware features, integration, and software development tools. DSPs are now a mature technology. This section gives an overview of the evolution of the DSP over their 25-year life span; specialized terms such as 'Harvard architecture', 'pipelining', 'instruction set' or 'JTAG' are used. The reader is referred to the following paragraphs for explanations of their meaning. More detailed information on DSP evolution can be found in Refs. [4], [5].

2.1 DSP evolution: hardware features

In the late 1970s there were many chips aimed at digital signal processing; however, they are not considered to be digital signal processing owing to either their limited programmability or their lack of hardware features such as hardware multipliers. The first marketed chip to qualify as a programmable DSP was NEC's MPD7720, in 1981: it had a hardware multiplier and adopted the Harvard architecture (more information on this architecture is given in Section 3.1). Another early DSP was the TMS320C10, marketed by TI in 1982. Figure 3 shows a selective chronological list of DSPs that have been marketed from the early 1980s until now.

From a market evolution viewpoint, we can divide the two and a half decades of DSP life span into two phases: a development phase, which lasted until the early 1990s, and a consolidation phase, lasting until now. Figure 3 gives an overview of the evolution of DSP features together with the first year of marketing for some DSP families.

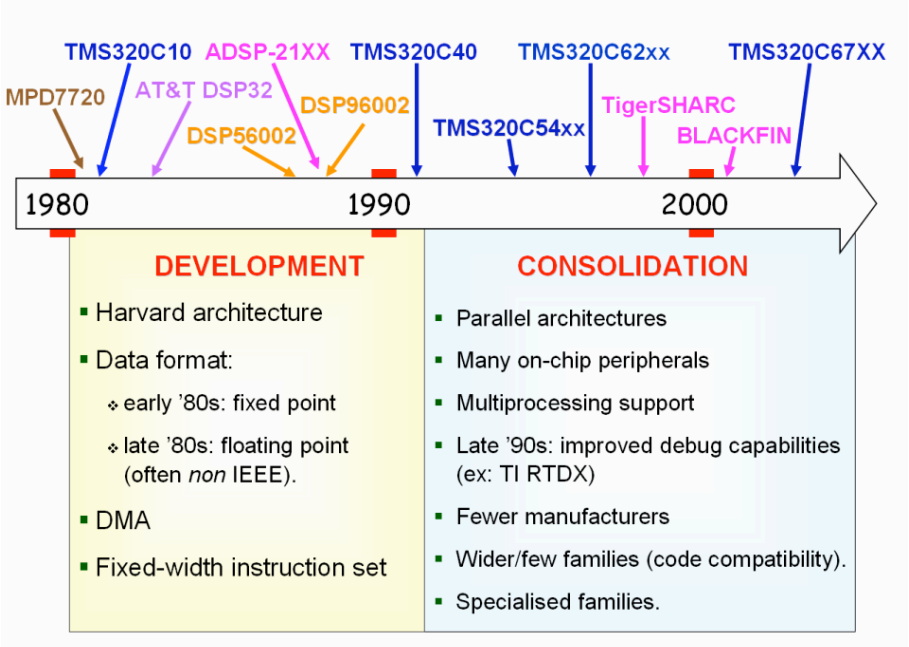


Fig. 3: Evolution of DSP features from their early days until now. The first year of marketing is indicated at the top for some DSP families.

During the market development phase, DSPs were typically based upon the Harvard architecture. The first generation of DSPs included multiply, add, and accumulator units. Examples are TI's TMS320C10 and Analog Devices' (ADI) ADSP-2101. The second generation of DSPs retained the architectural structure of the first generation but added features such as pipelining, multiple arithmetic units, special address generator units, and Direct Memory Access (DMA). Examples include TI's TMS320C20 and Motorola's DSP56002. While the first DSPs were capable of fixed-point operations only, towards the end of the 1980s DSPs with floating point capabilities started to appear. Examples are Motorola's DSP96001 and TI's TMS320C30. It should be noted that the floating-point format was not always IEEE-compatible. For instance, the TMS320C30 internal calculations were carried out in a proprietary format; a hardware chip converter [6] was available to convert to the standard IEEE format. DSPs belonging to the development phase were characterized by fixed-width instruction sets, where one of each instruction was executed per clock cycle. These instructions could be complex, and encompassing several operations. The width of the instruction was typically quite short and did not overcome the DSP native word width. As for DSP producers, the market was nearly equally shared between many manufacturers such as AT&T, Fujitsu, Hitachi, IBM, NEC, Toshiba, Texas Instruments and, towards the end of the 1980s, Motorola, Analog Devices and Zoran.

During the market consolidation phase, enhanced DSP architectures such as Very Long Instruction Word (VLIW) and Single Instruction Multiple Data (SIMD) emerged. These architectures increase the DSP performance through parallelism. Examples of DSPs with enhanced architectures are TI's TMS320C6xxx DSPs, which was the first DSP to implement the VLIW architecture, and ADI's TigerSHARC, that includes both VLIW and SIMD features. The number of on-chip peripherals increased greatly during this phase, as well as the hardware features that allow many processors to work together. Technologies that allow real-time data exchange between host processor and DSP started to appear towards the end of the 1990s. This constituted a real sea change in DSP system debugging and helped the developers enormously. Another phenomenon observed during this phase was the reduction of the number of DSP manufacturers. The number of DSP families was also greatly reduced, in favour of wider families that granted increased code compatibility between DSPs of

different generations belonging to the same family. Additionally, many DSP families are not 'general-purpose' but are focused on specific digital signal processing applications, such as audio equipment or control loops.

2.2 DSP evolution: device integration

Table 2 shows the evolution over the last 25 years of some key device characteristics and their expected values after the year 2010.

Table 2: Overview of DSP device characteristics as a function of time. The last column refers to expected values.

Characteristic \ Year	1980	1990	2000	> 2010
Wafer size [inches]	3	6	12	18
Die size [mm]	50	50	50	5
Feature [µm]	3	0.8	0.1	0.02
RAM [Bytes]	256	2000	32000	1 million
Clock frequency [MHz]	20	80	1000	10000
Power [mW/MIPS]	250	12.5	0.1	0.001
Price [USD]	150	15	5	0.15

Wafer, die, and feature sizes are the basic key factors that define a chip technology. The wafer size is the diameter of the wafer used in the semiconductor manufacturing process. The die size is the size of the actual chips carved up in a wafer. The feature size is the size of the smallest circuit component (typically a transistor) that can be etched on a wafer; this is used as an overall indicator of the density of an Integrated Circuit (IC) fabrication process. The trend in industry is to go towards larger wafers and chip dies, so as to increase the number of working chips that can be obtained from the same wafer; also called yield. For instance, the current typical wafer size is 12 inches (300 mm), and some leading chip maker companies plan to move to 18 inches (450 mm) within the first half of the next decade. (It should be added that the issue is somewhat controversial, as many equipment manufacturers fear that the 18 inches wafer size will lead to scale problems even worse than for the 12 inches.) Feature size is decreasing, allowing one to either have more functionality on a die or to reduce the die size while keeping the same functionality. Transistors with smaller sizes require less voltage to drive them; this results in a decrease of the core voltage from 5 V to 1.5 V. The I/O voltage has been lowered as well, with the caveat that it remains compatible with the external devices used and their standard. A lower core voltage has been one of the key factors enabling higher clock frequencies: in fact, the gap between high and low state thresholds is tightened thus allowing a faster logic level transition. Additionally, the reduced die size and lowered core voltage allow lower power consumption, an important factor for portable or mobile system. Finally, the global cost of a chip has decreased by at least a factor 30 over the last 25 years.

The trend towards a faster switching hardware (including chip over-clocking) and smaller feature size carries the benefit of increased processing power and throughput. There is a downside to it, however, represented by the electromigration phenomenon. Electromigration occurs when some of the momentum of a moving electron is transferred to a nearby activated ion, hence causing the ion to move from its original position. Gaps or, on the contrary, unintended electrical connections can develop with time in the conducting material if a significant number of atoms are moved far from their

original position. The consequence is the electrical failure of the electronic interconnects and the consequent shortened chip lifetime.

2.3 DSP evolution: software tools

The improvement of DSP software tools from the early days until now has been spectacular.

Code compilers have evolved greatly to be able to deal with the underlying hardware complexity and the enhanced DSP architectures. At the same time, they allow the developer to program more and more efficiently in high-level languages as opposed to assembly coding. This speeds up considerably the code development time and makes the code itself more portable across different platforms.

Advanced tools now allow the programming of DSPs graphically, i.e., by interconnecting pre-defined blocks that are then converted to DSP code. Examples of these tools are MATLAB Code Generation and embedded target products and National Instruments' LabVIEW DSP Module.

High-performance simulators, emulator and debugging facilities allow the developer to have a high visibility into the DSP with little or no interference on the program execution. Additionally, multiple DSPs can be accessed in the same JTAG chain for both code development and debugging.

2.4 DSP current scenery

The number of DSP vendors is currently somewhat limited: Analog Devices (ADI), Freescale (formerly Motorola), Texas Instruments (TI), Renesas, Microchip and VeriSilicon are the basic players. Amongst them, the biggest share of the market is taken by only three vendors, namely ADI, TI and Freescale. In the accelerator sector one can find mostly ADI and TI DSPs, hence most of the examples in this document will be focused on them. Table 3 lists the main DSP families for ADI and TI DSPs, together with their typical use and performance.

Table 3: Main ADI and TI DSP families, together with their typical use and performance

Manufacturer	Family	Typical use and performance
TI	TMS320C2x	Digital signal controllers
	TMS320C5x	Power efficient
	TMS320C6x	High performance
ADI	SHARC	Medium performance. First ADI family (now three generations)
	TigerSHARC	High performance for multi-processor systems
	Blackfin	High performance and low power

3 DSP core architecture

3.1 Introduction

DSP architecture has been shaped by the requirements of predictable and accurate real-time digital signal processing. An example is the Finite Impulse Response (FIR) filter, with the corresponding mathematical equation (1), where y is the filter output, x is the input data and a is a vector of filter coefficients. Depending on the application, there might be just a few filter coefficients or many hundreds or more.

$$y(n) = \sum_{k=0}^{M} a_k \cdot x(n-k) \,. \tag{1}$$

As shown in Eq. (1), the main component of a filter algorithm is the 'multiply and accumulate' operation, typically referred to as MAC. Coefficients data have to be retrieved from the memory and the whole operation must be executed in a predictable and fast way, so as to sustain a high throughput rate. Finally, high accuracy should typically be guaranteed. These requirements are common to many other algorithms performed in digital signal processing, such as Infinite Impulse Response (IIR) filters and Fourier Transforms. Table 4 shows a selection of processing requirements together with the main DSP hardware features satisfying them. These hardware features are discussed in more detail in Sections 3.2 to 3.5 and a full overview of a typical DSP core will be built step by step (see Figs. 4, 7, 10, 13). More detailed information on DSP architectural features can be found in Refs. [7] –[14].

Table 4: Main requirements and corresponding DSP hardware implementations for predictable and accurate real-time digital signal processing. The numbers in the first column refer to the section treating the topic.

Processing requirements	Hardware implementations satisfying the requirement
3.2 Fast data access	• High-bandwidth memory architectures • Specialized addressing modes • Direct Memory Access (DMA)
3.3 Fast computation	• MAC-centred • Pipelining • Parallel architectures (VLIW, SIMD)
3.4 Numerical fidelity	• Wide accumulator registers, guard bits, etc.
3.5 Fast execution control	• Hardware-assisted, zero-overhead loops, shadow registers, etc.

3.2 Fast data access

Fast data access refers to the need of transferring data to / from memory or DSP peripherals, as well as retrieving instructions from memory. The hardware implementations considered for this are three, namely a) high-bandwidth memory architectures, discussed in Sub-section 3.2.1; b) specialized addressing modes, discussed in Sub-section 3.2.2; c) direct memory access discussed in Sub-section 3.2.3.

3.2.1 High-bandwidth memory architectures

Traditional general-purpose microprocessors are based upon the Von Neumann architecture, shown in Fig. 4(a). This consists of a single block of memory, containing both data and program instructions, and of a single bus (called data bus) to transfer data and instructions from/to the CPU. The disadvantage of this architecture is that only one memory access per instruction cycle is possible, thus constituting a bottleneck in the algorithm execution.

DSPs are typically based upon the Harvard architecture, shown in Fig. 4(b), or upon modified versions of it, such as the Super-Harvard architecture shown in Fig. 4(c). In the Harvard architecture there are separate memories for data and program instructions, and two separate buses connect them to the DSP core. This allows fetching program instructions and data at the same time, thus providing better performance at the price of an increased hardware complexity and cost. The Harvard

architecture can be improved by adding to the DSP core a small bank of fast memory, called 'instruction cache', and allowing data to be stored in the program memory. The last-executed program instructions are relocated at run time in the instruction cache. This is advantageous for instance if the DSP is executing a loop small enough so that all its instructions can fit inside the instruction cache: in this case, the instructions are copied to the instruction cache the first time the DSP executes the loop. Further loop iterations are executed directly from the instruction cache, thus allowing data retrieval from program and data memories at the same time.

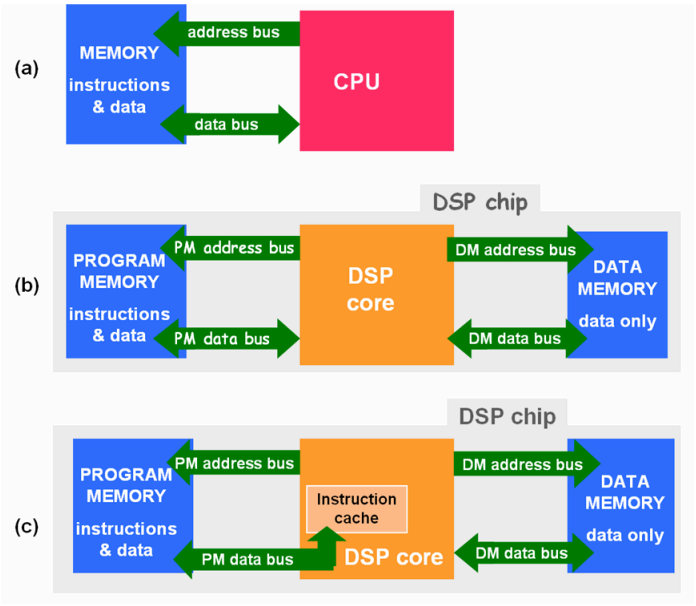


Fig. 4: (a) Von Neumann architecture, typical of traditional general-purpose microprocessors. (b) Harvard and (c) Super-Harvard architectures, typical of DSPs.

Another more recent improvement of the Harvard architecture is the presence of a 'data cache', namely a fast memory located close to the DSP core which is dynamically loaded with data. Of course, the fact of having the cache memory very close to the DSP allows clocking it at high speed, as routing wire delays are short. Figure 5 shows the cache architecture for TI TMS320C67xx DSP, including both program and data cache. There are two levels of cache, called Level 1 (L1) and Level 2 (L2). The L1 cache comprises 8 kbyte of memory divided into 4 kbyte of program cache and 4 kbyte of data cache. The L2 cache comprises 256 kbyte of memory divided into 192 kbyte mapped-SRAM memory and 64 kbyte dual cache memory. The latter can be configured as mapped memory, cache or a combination of the two. The reader can find more information on TI TMS320C67xx DSP two-level memory architecture and configuration possibilities in Ref. [12].

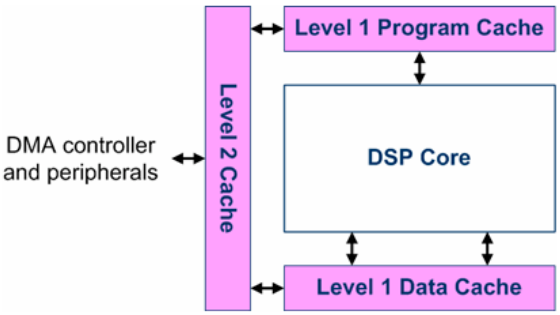


Fig. 5: TI DSP TMS320C67xx family two-level cache architecture

Figure 6 shows the hierarchical memory architecture to be found in a modern DSP [13]. Typical levels of memory and corresponding access time, hardware implementation, and size are also shown. As remarked above, a hierarchical memory allows one to take advantage of both the speed and the capacity of different memory types. Registers are banks of very fast internal memory, typically with single-cycle access time. They are a precious DSP resource used for temporary storage of coefficients and intermediate processing values. The L1 cache is typically high-speed static RAM made of five or six transistors. The amount of L1 cache available thus depends directly on the available chip space. A L2 cache needs typically a smaller number of transistors hence can be present in higher quantities inside the DSPs. Recent years have also seen the integration of DRAM memory blocks into the DSP chip [14], thus guaranteeing larger internal memories with relatively short access times. The Level 3 (L3) memory shown in Fig. 6 is rarely present in DSPs while the external memory is typically available. This is often a large memory with long access times.

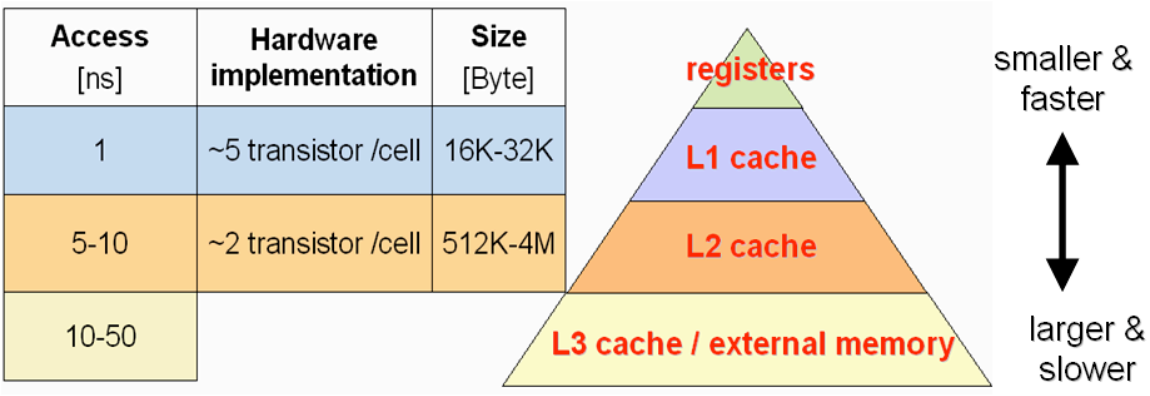


Fig. 6: DSP hierarchical memory architecture and typical number of access clock cycles, hardware implementation, and size for different memory types

As shown above, cache memories improve the average system performance. However, there are drawbacks to the presence of a cache in DSP-based systems, owing to the lack of full predictability for cache hits. A missing cache hit happens when the data or the instructions needed by the DSP are not stored in cache memory, hence they have to be fetched from a slower memory with an execution speed penalty. A situation causing a missing cache hit is, for instance, the flow change due to branch instructions. The consequence is a difficult worst-case-scenario prediction, which is particularly negative for DSP-based systems where it is important to be able to calculate and predict the system time response. There may, however, be methods used to limit these effects, such as the possibility for the user to lock the cache so as to execute time-critical sections in a deterministic way. Advanced cache organizations characterized by a uniform memory addressing are also under study [15].

3.2.2 Specialized addressing modes

DSPs include specialized addressing modes and corresponding hardware support to allow a rapid access to instruction operands through rapid generation of their location in memory. DSPs typically support a wide range of specialized addressing modes, tailored for an efficient implementation of digital signal processing algorithms.

Figure 7 adds the address generator units to the basic DSP architecture shown in Fig. 4(c). As in general-purpose processors, DSPs include a Program Sequencer block, which manages program structure and program flow by supplying addresses to memory for instruction fetches. Unlike general-purpose processors, DSPs include address generator blocks, which control the address generation for specialized addressing modes such as indexing addressing, circular buffers, and bit-reversal addressing. The two last addressing modes are discussed below.

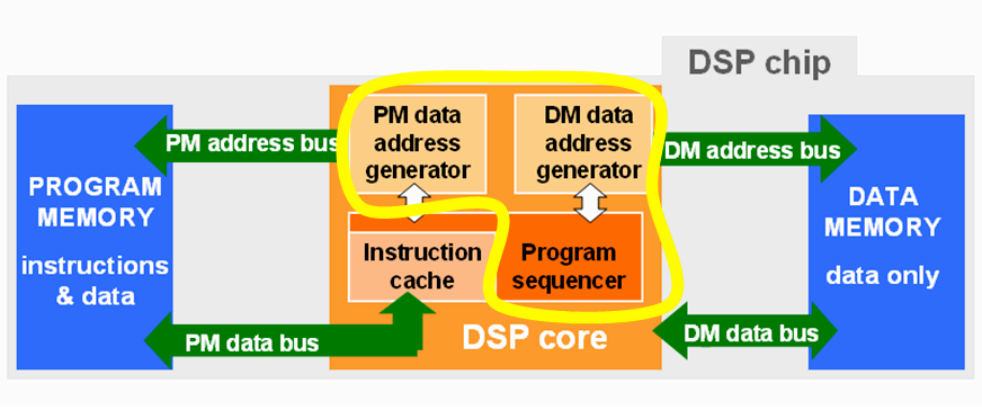


Fig. 7: Program sequencer and address generator units location within a generic DSP core architecture

Circular buffers are limited memory regions where data are stored in a First-In First-Out (FIFO) way; these memory regions are managed in a 'wrap-around' way, i.e., the last memory location is followed by the first memory location. Two sets of pointers are used, one for reading and one for writing; the length of the step at which successive memory locations are accessed is called 'stride'. Address generator units allow striding through the circular buffers without requiring dedicated instructions to determine where to access the following memory location, error detection and so on. Circular buffers allow storing bursts or continuous streams of data and processing them in the order in which they have arrived. Circular buffers are used for instance in the implementation of digital filters; strides higher than one are useful in case of multi-rate signal processing. Figure 8 shows the order in which data are accessed for a read operation in case of an eleven-element circular buffer and with a stride equal to four.

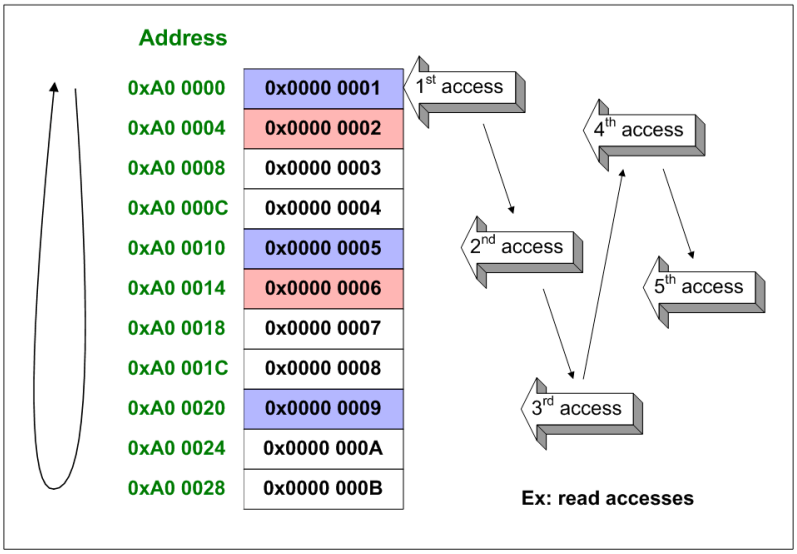


Fig. 8: Example of read data access order in a circular buffer composed of 11 elements and with stride equal to 4 elements

Bit-reversal addressing, shown in Fig. 9, is an essential step in the discrete Fourier transforms calculation. In fact, many implementations of the Fourier transforms require a re-ordering of either the input or the output data that corresponds to reversing the order of the bits in the array index. Figure 9 gives an example of the bit-reversal mechanism. Carrying it out by software is very demanding and

would result in using many CPU cycles, which are saved thanks to the hardware bit-reversal functionality.

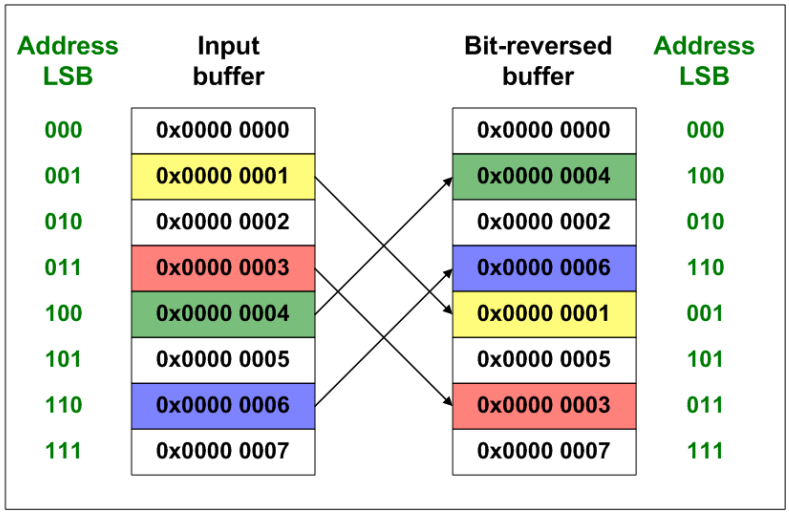


Fig. 9: Bit-reversal mechanism

3.2.3 Direct Memory Access (DMA) controller

The DMA controller is a second processor working in parallel with the DSP core and dedicated to transferring information between two memory areas or between peripherals and memory. In doing so the DMA controller frees the DSP core for other processing tasks. Figure 10 shows an example of the DMA location within a general DSP core architecture.

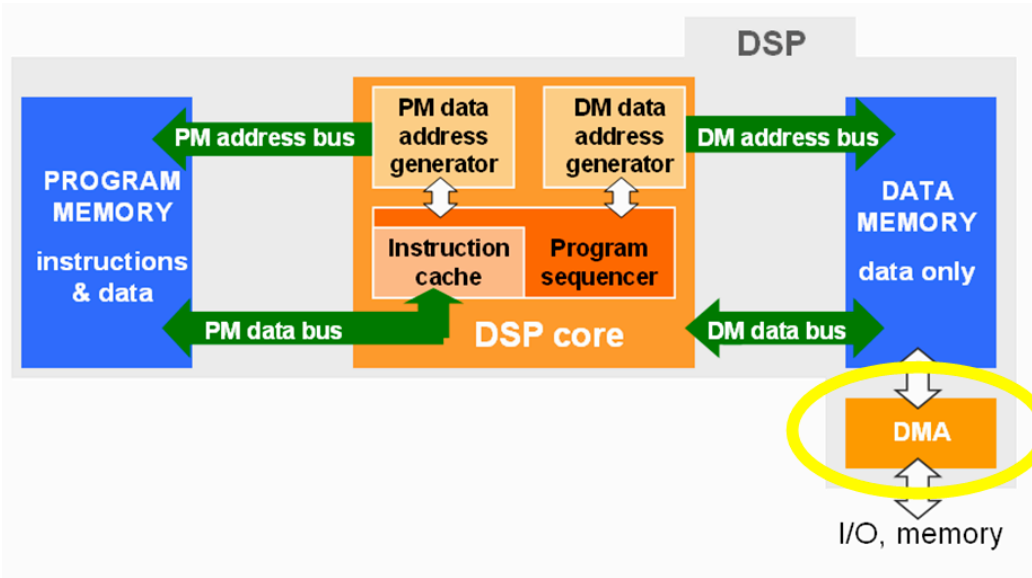


Fig. 10: An example of DMA controller location within a generic DSP core architecture

A DMA coprocessor can transfer data as well as program instructions, the latter transfer corresponding typically to the case of code overlay, i.e., of code stored in an external memory and moved to an internal memory (for instance L1) when needed. Multiple and independent DMA channels are also available for greater flexibility. Bus arbitration between the DMA and the DSP core is needed to avoid colliding memory accesses when the DMA and the DSP core share the same bus to access peripherals and/or memories. To prevent bottlenecks, recent DSPs typically fit DMA controllers with dedicated buses.

Figure 11 shows the advantages of DMA for the DSP core efficient use: the DSP core must set up the DMA but still there is a net gain in the DSP core availability for other processing activities. Nowadays there are two classes of DMA transfer configurations: register-based and RAM-based, the latter one also called descriptor-based. In register-based DMA controllers the transfer set-up is done by the DSP core via the registers set-up. This method is very efficient but allows mainly simple DMA operations. In RAM-based DMA controllers the set-up parameters are stored in memory. This method is preferred by powerful and recent DSPs as it allows great DMA transfer flexibility.

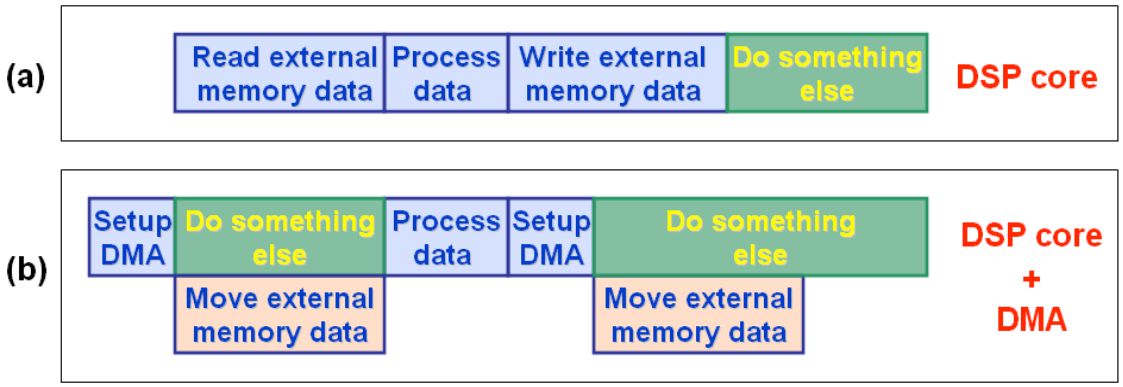


Fig. 11: (a) Read–process–write data when the DSP core only is present; (b) same activity when the DMA takes care of data transfers

Figure 12 provides two examples of transfer configurations. Plot (a) shows a chained DMA transfer, where the completion of a data transfer triggers a new transfer. This type of data transfer is particularly suited to applications that require a continuous data stream in input. Plot (b) shows a multi-dimensional data transfer, obtained by changing the stride of the DMA transfer. This type of data transfer is particularly useful for video applications.

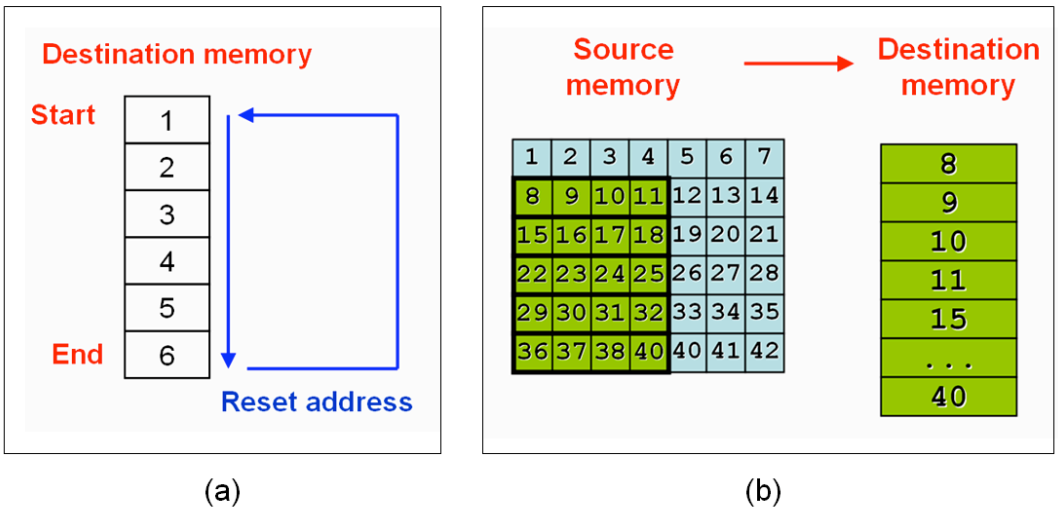


Fig. 12: Examples of DMA transfer configurations. (a): chained DMA transfer; (b): Multi-dimensional data transfer.

DSP external events and interrupts can be used to trigger a DMA data transfer. DMA controllers can also generate interrupts to communicate with the DSP core, for instance to inform it that a data transfer has been completed. An example of a powerful and highly flexible DMA controller is that implemented for TI's TMS320C6000 family [16].

3.3 Fast computation

Here we discuss techniques and architectures used in DSPs for a fast computation. The MAC-centred architecture described in Sub-section 3.3.1 has been common to all DSPs since their early days. The techniques and architectures described in Sub-sections 3.3.2 and 3.3.3 were introduced from the 1990s onwards.

3.3.1 MAC-centred

The MAC operation is used by many digital processing algorithms, as discussed at the beginning of Section 3; consequently its execution must be optimized so as to improve the DSP overall performance. The basic DSP arithmetic processing blocks are a) many registers; b) one or more multipliers; c) one or more Arithmetic Logic Units (ALUs); d) one or more shifters. These blocks work in parallel during the same clock cycle thus optimizing MAC as well as other arithmetic operations. The blocks are shown in Fig. 13 and are briefly described below.

a) Registers: these are banks of very fast memory used to store intermediate data processing. Very often they are wider than the DSP normal word width, so as to provide a higher resolution during the processing.

b) Multiplier: it can carry out single-cycle multiplications and very often it includes very wide accumulator registers to reduce round-off or truncation errors. As a consequence, truncation and round-off errors will happen only at the end of the data processing, when the data is stored onto memory. Sometimes an adder is integrated in the multiplier unit.

c) ALU: it carries out arithmetic and logical operations.

d) Shifters: it shifts the input value by one or more bits, left or right. In the latter case, the shifter is called a barrel shifter and is especially useful in the implementation of floating point add and subtract operations.

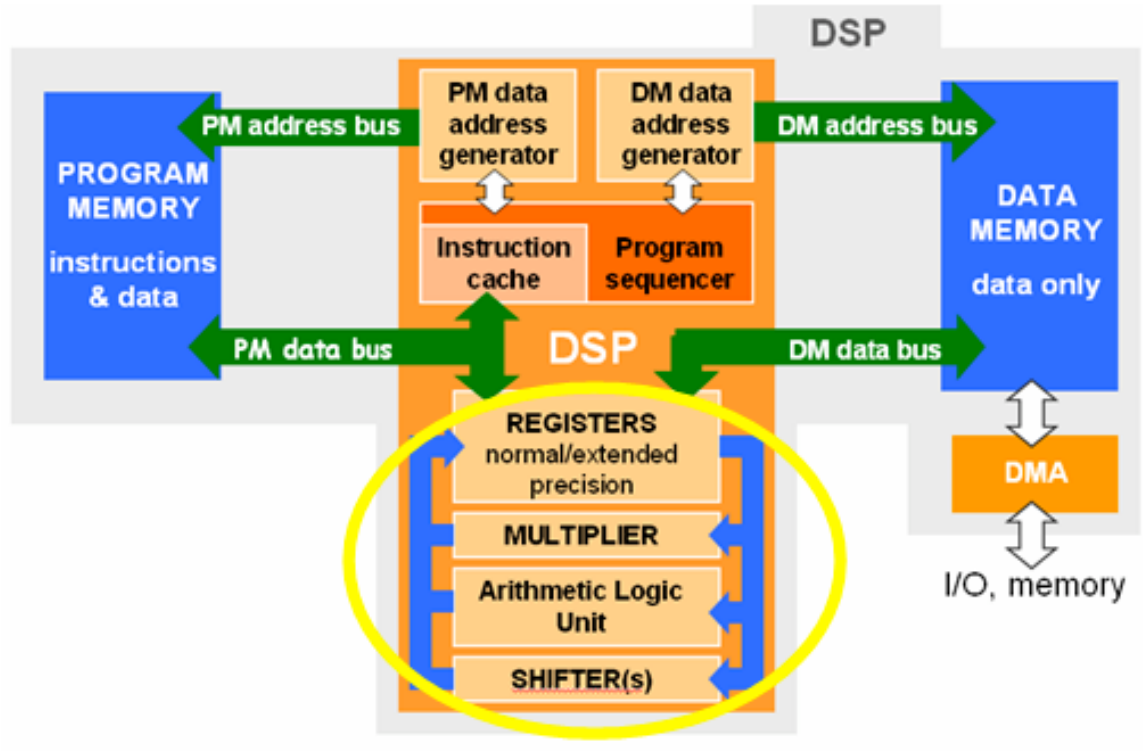


Fig. 13: Basic DSP arithmetic processing blocks. The structure shown is that of ADI SHARC.

3.3.2 Instruction pipelining

Instruction pipelining has become an important element to achieve high DSP performance. It consists of dividing the execution of instructions into different stages and executing the different instructions in parallel stages. The net result is an increased throughput of the instruction execution. The whole process can be compared to a factory assembly line, which produces cars for instance: more than one car is in the assembly line at the same moment, at different stages of assembly. This provides a production higher than the case where only one car at a time is produced, where many specialized crews are idle waiting for the next car to require their work.

Table 5 shows the basic pipelining stage into which each instruction is divided:

1. Fetch. The DSP calculates the address of the next instruction to execute and retrieve the op-code, i.e., the binary word containing the operands and the operation to be carried out on them.
2. Decode. The op-code is interpreted and sent to the corresponding functional unit. The instruction is interpreted and the operands are retrieved.
3. Execute. The instruction is executed and the results are written onto the registers.

Table 5: The three basic pipelining stages and corresponding actions

Basic pipelining stages	Action
Fetch	• Generate program fetch address • Read op-code
Decode	• Route op-code to functional unit • Decode instruction • Read operands
Execute	• Execute instruction • Write results back to registers

Figure 14 shows the advantage of a pipelined CPU with respect to a non-pipelined CPU, in terms of processing time gain. In a non-pipelined CPU the different instructions are executed serially, while in a pipelined CPU only the same type of stages (e.g. Fetch, Decode and Execute) are serialized and different instructions are executed in parallel. A pipeline is called fully-loaded if all stages are executed at the same time; this corresponds to the maximum possible instruction throughput. The depth of the pipeline, i.e., the number of stages into which an instruction is divided, can vary from one processor to another. Generally speaking a deeper pipeline allows the processor to execute faster, hence many processors sub-divide pipeline stages into smaller steps, each one executed at each clock cycle. The smaller the step, the faster the processor clock speed can be. An example of deep pipeline is the TI TMS320C6713 DSP, which includes four fetch stages, two decode stages, and up to ten execution stages.

There are drawbacks and limitations to the pipelining technique. One drawback is the hardware and programming complexity required by it, for instance in terms of capabilities needed in the compiler and the scheduler. This is especially true in the case of deep pipelines. A limitation in the effective instruction execution throughput is given by situations that prevent the pipeline from being fully-loaded. These situations include pipeline flushes due to changes in the program flow, such as code branches or interrupts. In this case, the DSP does not know which instructions it should execute next until the branch instruction is executed. Other situations are data hazards, namely when one instruction needs the result of a previous instruction to be executed. Apart from a reduced throughput,

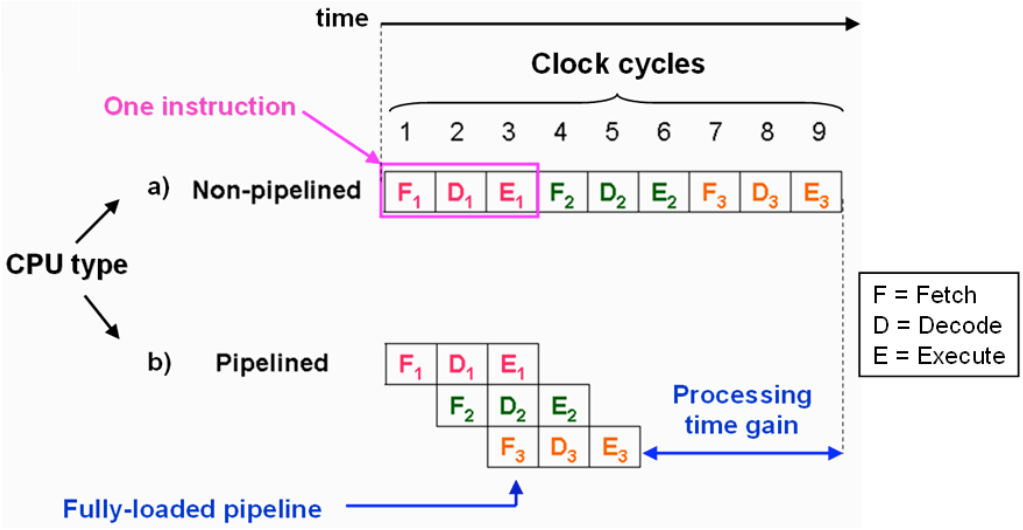


Fig. 14: Instruction execution and processing time gain of a pipelined CPU (plot b) with respect to a non-pipelined one (plot a)

these pipeline limitations cause a more difficult prediction of the worst-case scenario. Techniques not described here are available to provide the DSP programmer with a pipeline control; they include time-stationary pipeline control, data-stationary control, and interlocked pipeline.

3.3.3 Parallel architectures

The DSP performance can be increased by an increased parallelism in the instructions execution. Parallel-enhanced DSP architectures started to appear on the market in the mid 1990s and were based on instruction-level parallelism, data-level parallelism, or a combination of both. These two approaches are called Very Long Instruction Word (VLIW) and Single-Input Multiple-Data (SIMD), respectively and are discussed below. The reader is referred to Refs. [17] and [18] for more information on the subject.

VLIW architectures are based upon instruction level parallelism, i.e., many instructions are issued at the same time and are executed in parallel by multiple execution units. As a consequence, DSPs based on this architecture are also called 'multi-issue' DSP. This is an innovative architecture that was first used in the TI TMS320C62xx DSP family. Figure 15 shows an example of the VLIW architecture: eight, 32-bit instructions are packed together in a 256-bit wide instruction which is fed to eight separate execution units. Characteristics of VLIW architectures include simple and regular instruction sets. Instruction scheduling is done at compile-time and not at run-time so as to guarantee a deterministic behaviour. This means that the decision on which instructions have to be executed in parallel is done when the program is compiled, hence the order does not change during the program execution. A run-time scheduling would instead make the scheduling dependent on data and resources availability, which could change for different program executions. An important advantage of the VLIW architecture is that it can increase the DSP performance for a wide range of algorithms. Additionally, the architecture is potentially scalable, i.e., more execution units could be added to allow a higher number of instructions to be executed in parallel. There are disadvantages as well, such as the high memory use and power consumption required by this architecture. From a programmer's viewpoint, writing assembly code for VLIW architecture is very complex and the optimization is often better left to the compiler.

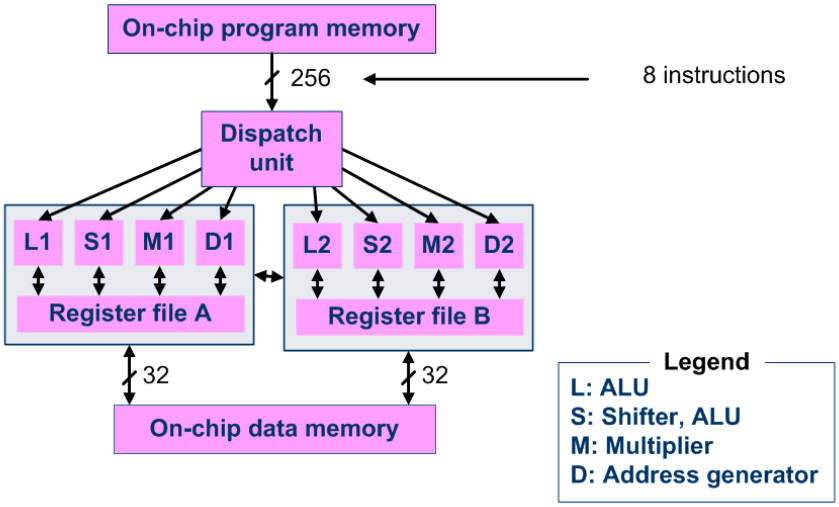


Fig. 15: TI TMS320C6xxx family VLIW architecture

SIMD architectures are based on data-level parallelism, i.e., only one instruction is issued at a time but the same operation specified by the instruction is performed on multiple data sets. Figure 16 shows the example of a DSP based upon the SIMD architecture: two 32-bit input registers provide four, 16-bit each, data inputs. They are processed in parallel by two separate execution units that carry out the same operation. The two, 16-bit data outputs are packed into a 32-bit register. Typical SIMD architecture can support multiple data width and is most effective on algorithms that require the processing of large data chunks. The SIMD operation mode can be switched ON or OFF, for instance in the ADI SHARC DSP. An advantage of the SIMD architecture is that it is applicable to other architectures; an example is the ADI TigerSHARC DSP that comprises both VLIW and SIMD characteristics. SIMD drawbacks include the fact that SIMD architectures are not useful for algorithms that process data serially or that contain tight feedback loops. It is sometimes possible to convert serial algorithms to parallel ones; however, the cost is in reorganization penalties and in a higher program-memory usage, owing to the need to re-arrange the instructions.

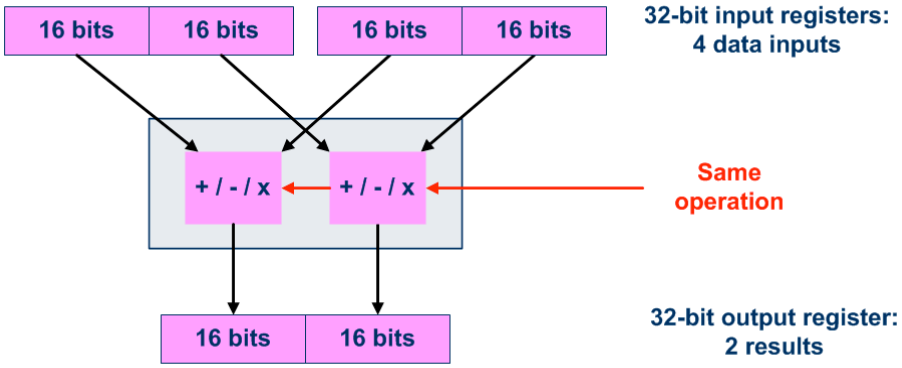


Fig. 16: Simplified schematics for ADI SHARC DSP as an example of SIMD architecture

3.4 Numerical fidelity

Arithmetic operations such as additions and multiplications are the heart of DSP systems. It is thus essential that the numerical fidelity be maximized, i.e., that errors due to the finite number of bits used in the number representation and in the arithmetic operations be minimized. DSPs have many ways to obtain this, ranging from the numeric representation to dedicated hardware features.

As far as the number representation is concerned, DSPs can be divided into two categories: fixed point and floating point.

Fixed-point DSPs perform integer as well as fractional arithmetic, and can support data widths of 16, 24 or 32 bits. A fixed-point format can represent both signed and unsigned integers and fractions. Fractional numbers can take values in the [−1.0, 1.0] range and are often indicated as Qx.y, where 'x' indicates the number of bits located before the binary point and 'y' the number of bits after it. Figure 17(a) shows how 16-bit signed fractional point numbers are coded. Signed fractional numbers with 24-bit and 32-bit data width are coded in an equivalent way as Q1.23 and Q1.31, respectively. They can take values in the same [−1.0, 1.0] range, however, their resolution is higher than the 16-bit implementation. An example of extended precision fixed-point can be found in Ref. [19].

Floating-point DSPs represent numbers with a mantissa and an exponent, nowadays following the IEEE 754 [20] standard shown in Fig. 17(b) for a 32-bit number. The mantissa dictates the number precision and the exponent controls its dynamic range. Numbers are scaled so as to use the full word-length available, hence maximizing the attainable precision. The reader is referred to Ref. [21] for more information on the subject.

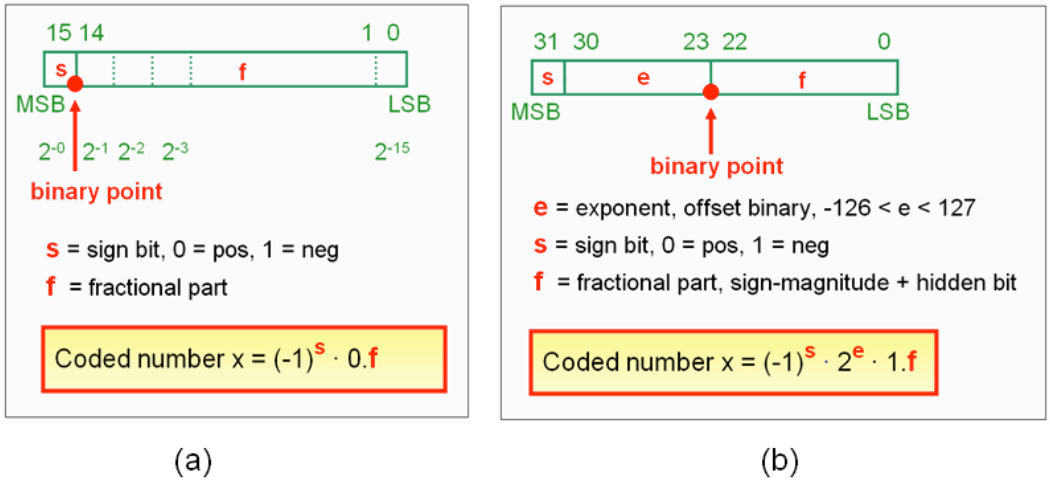


Fig. 17: (a): 16-bit signed fractional point, often indicated as Q1.15. (b): IEEE 754 normalized representation of a single precision floating point number.

Floating-point numbers provide a higher dynamic range, which can be essential when dealing with large data sets and with data sets whose range cannot be easily predicted. The dynamic range for a 32-bit number represented as fixed-point and as floating-point is shown in Fig. 18.

$$\text{Dynamic range}_{dB} = 20 \log_{10} \left[\frac{\text{largest value}}{\text{smallest value}} \right] \begin{cases} \text{Fixed point} \sim 180 \text{ dB} \\ \text{Floating point} \sim 1500 \text{ dB} \end{cases}$$

Fig. 18: Dynamic range for 32-bit data, represented as 32-bit signed fractional point and IEEE 754 normalized number

In addition to the different number formats available, DSPs provide hardware ways to improve numerical fidelity. One example is represented by the large accumulator registers, used to hold intermediate and final results of arithmetic operations. These registers are several bits (at least four) wider than the normal registers in order to prevent overflow as much as possible during accumulation operations. The extra bits are called guard bits and allow one to retain a higher precision in

intermediate computation steps. Flags to indicate that an overflow/underflow has happened are also available. These flags are often connected to interrupts, thus allowing exception-handling routines to be called. Another means DSPs have to improve numerical fidelity is saturated arithmetic. This means that a number is saturated to the maximum value that can be represented, so as to avoid wrap-around phenomena.

3.5 Fast-execution control

Here we show two important examples of how DSP can fast-execute control instructions. The first example is the zero-overhead hardware loop and refers to the program flow control in loops. The second example refers to how DSPs react to interrupts.

Looping is a critical feature in many digital signal processing algorithms. An important DSP feature is the implementation by hardware of looping constructs, referred to as 'zero-overhead hardware loop'. This allows DSP programmers to initialize loops by setting a counter and defining the loop bounds, without spending any software overhead to update and test loop counters or branching back to the beginning of the loop.

The capability to service interrupts very quickly and in a deterministic way is an important DSP characteristic. Interrupts are internal (for instance generated by internal timers) or external (brought to the DSP code via pins) events that change the DSP execution flow when they are serviced. The latency is the time elapsed from when the interrupt event is triggered and when the DSP starts to execute the first instruction of the corresponding Interrupt Service Routine (ISR). When an interrupt is received and if the interrupt has a sufficiently-high priority, the DSP must carry out the following actions:

a) stop its current activity;

b) save the information related to the interrupted activity (called context) into the DSP stack;

c) start servicing the interrupt.

The context corresponding to the interrupted activity can be restored when the ISR has been executed and the previous activity is continued.

Table 6: Interrupt dispatchers available on the ADI ADSP21160M DSP. The instruction cycle is 12.5 μs, hence the number of cycles can easily be converted to time.

Interrupt dispatcher	Cycles before ISR	Cycles after ISR
Normal	183	109
Fast	40	26
Super-fast (with alternate registers set)	34	10
Final	24	15

More than one interrupt dispatcher is typically available in a DSP; this means that the user can select the amount of context to be saved, knowing that a higher number of saved registers implies a longer context switching time. An interesting feature available in some DSPs, such as the ADI SHARC AD21160 [22], is the presence of two register sets, called 'primary' and 'alternate' for all the CPU's key registers. When an interrupt occurs, the alternate register set can be used, thus allowing a very fast context switch. Table 6 shows the four interrupt dispatchers available on the ADSP21160M DSP and their corresponding latency ('Cycles before ISR') and context restore time ('Cycles after ISR'. The 'Final' dispatcher is intended for use with user-written assembly functions or C functions that have been compiled using '*#pragma interrupt*'. In particular, this dispatcher relies on the compiler (or assembly routine) to save and restore all appropriate registers.

3.6 DSP core example: TI TMS320C67x

Figure 19 shows TI's TMS320C6713 DSP [23] core architecture, as an example of modern VLIW architecture implementing many of the characteristics described in Section 3. This DSP is that used in the laboratory companion of the lectures upon which this paper is based.

Boxes inside the yellow square belong to the DSP core architecture, which here is considered to include the cache memory as well as the DMA controller. The white boxes are components common to all C6000 devices; grey boxes are additional features on the TMS320C6713 DSP.

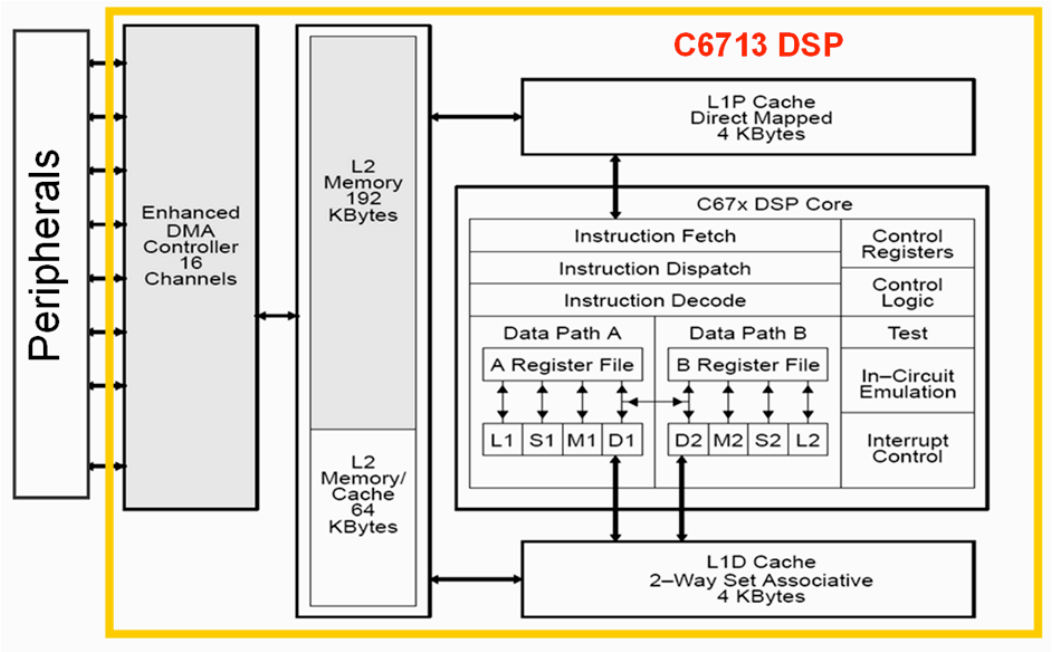


Fig. 19: TI TMS320C6713 DSP core architecture. Picture courtesy of TI [23].

The TMS320C6713 DSP is a floating point DSP with VLIW architecture. The internal program memory is structured so that a total of eight instructions can be fetched at every cycle. To give a numerical example, with a clock rate of 225 MHz the C6713 DSP can fetch eight, 32-bit instructions every 4.4 ns. Features of the C6713 include 264 kBytes of internal memory: 8 kB as L1 cache and 256 kB as L2 memory shared between program and data space. The processing of instructions occurs in each of the two data paths (A and B), each of which contains four functional units (.L, .S, .M, .D). An Enhanced DMA (EDMA) controller supports up to 16 EDMA channels. Four of the sixteen channels (channels 8−11) are reserved for EDMA chaining, leaving twelve EDMA channels available to service peripheral devices.

4 DSP peripherals

4.1 Introduction

The available peripherals are an important factor for the DSP choice. Peripherals are here considered as belonging to two categories:

a) interconnect, discussed in Section 4.2;

b) services, such as timers, PLL and power management, discussed in Section 4.3.

DSP developers must in fact carefully evaluate the needs of their system in terms of interconnect and services required, to avoid bottlenecks and reduced system performance.

Modern DSPs often have several peripherals integrated on-chip, such as UARTs, serial, USB and video ports. There are benefits in using embedded peripherals, such as fast performance and reduced power consumption. There are, however, drawbacks, in that embedded peripherals can be less flexible across applications and their unit cost might be higher.

The evolution of DSP-supported peripherals has been terrific over the last 20 years. From the original few parallel and serial ports, DSP can now support a wide peripherals range, including those needed by audio/video streaming applications. Often the DSP chip does not have pins to allow using all supported peripherals at the same time. To overcome this limitation, the pins are multiplexed, i.e., the DSP developer must select at boot time which peripherals he/she needs to have available. An example of pin multiplexing referred to TI's TMS320C6713 DSP is given in Section 4.4.

An overview of interconnect and DSP services is given in Sections 4.2 and 4.3, respectively. Hints on different interfacing possibilities to external memories and data converter memories are provided in Sections 4.5 and 4.6, respectively. Finally, a brief outline of the DSP booting process is given in Section 4.7.

4.2 Interconnect

The amount of supported interconnect and data I/O is huge, so only a few examples are given below, divided per interconnect type.

Serial interfaces

a) Serial Peripheral Interface (SPI): this is an industry-standard synchronous serial link that supports communication with multiple SPI compatible devices. The SPI peripheral is a synchronous, four-wire interface consisting of two data pins, one device select pin, and a gated clock pin. With the two data pins, it allows for full-duplex operation to other SPI compatible devices. An example of DSP fitted with a SPI port is ADI's Blackfin ADSP-BF533 [24].

b) Multichannel Buffered Serial Ports (McBSP) [25] on TI's DSPs: this serial interface is based upon the standard serial port found in TMS320C2x and TMS320C5x DSPs.

c) Multichannel Audio Serial Port (McASP) [26] on TI's DSPs: this is a serial port optimized for the needs of multichannel audio applications. Each McASP includes transmit and receive sections that can operate synchronized as well as completely independent, i.e., with separate master clocks, bit clocks, and data stream formats.

Parallel interfaces

a) ADI's linkports [27] are parallel interfaces that allow DSP–DSP as well as DSP–peripheral connection. An example of their use for inter-DSP communication to build multi-DSP systems is given in Sub-section 9.3.1.

b) Parallel Peripheral Interface (PPI) [28] on ADI's Blackfin DSP: this is a multifunction parallel interface, configurable between 8 and 16 bits in width. It supports bidirectional data flow and it includes three synchronization lines and a clock pin for connection to an externally-supplied clock. The PPI can receive data at clock speeds of up to 65 MHz, while transmit rates can approach 60 MHz.

Other interfaces commonly found, for instance in TI DSPs, are Peripheral Component Interconnect (PCI) [29], Inter-Integrated Circuit (I2C) [30], Host-Port Interface (HPI) [31] and General-Purpose Input/Output (GPIO) [32].

4.3 Services

System services provide functionality that is common to embedded systems; the on-chip hardware is generally accompanied by an API that allows one to easily interface to them. A few examples of services are given below.

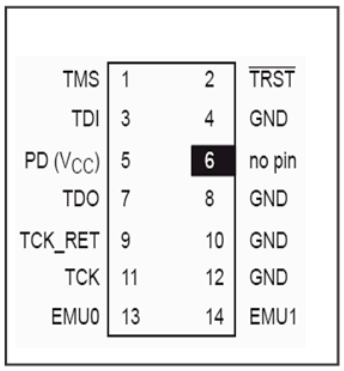
a) Timers: DSPs are typically fitted with one or more general-purpose timers that are used to time or count events, generate interrupts to the CPU, or send synchronization events to a DMA/EDMA controller. More information on timers for TI's TMS320C6000 DSPs can be found in Ref. [33].

b) PLL controller: it generates clock pulses for the DSP code and the peripherals from internal or external clock signals. More information on PLL controllers for TI's TMS320C6000 DSPs can be found in Ref. [34].

c) Power Management: the power-down logic allows the reduction of clocking so as to reduce power consumption. In fact, most of the operating power of CMOS logic dissipates during circuit switching from one logic state to the other. Significant power can be saved by preventing some of these level switches. More information on power management logic of TI's TMS320C6000 DSPs can be found in Ref. [35].

d) Boot configuration: a variety of boot configurations are often available in DSPs. They are user-programmable and determine what actions the DSP performs after it has been reset to prepare for the initialization. These actions include loading the DSP code load from external memory or from an external host. Some boot modes are outlined in Section 4.7. More information on boot modes, device configuration, and available boot processes for TI TMS320C62x/67x is available in Ref. [36].

e) JTAG: this interface implements the IEEE standard 1149.1 and allows emulation and debugging. A detailed description of its use can be found in Section 7.2. Figure 20 shows a typical JTAG connector and corresponding signals [37].



Signal	Description	Emulator state	Target state
TMS	Test mode select	OUT	IN
TDI	Test data input	OUT	IN
TDO	Test data output	IN	OUT
TCK	Test clock: 10.368 MHz clock source from emulation cable pod, that can be used to drive the system test clock.	OUT	IN
$\overline{\text{TRST}}$	Test reset	OUT	IN
EMU0	Emulation pin 0	IN	IN/OUT
EMU1	Emulation pin 1	IN	IN/OUT
PD(Vcc)	Presence detect: it indicates that the emulation cable is connected and that the power is powered up	IN	OUT
TCK_RET	Test clock return, input to the emulator	IN	OUT
GND	Ground		

Fig. 20: Fourteen-pin JTAG header and corresponding signals. Picture courtesy of TI [37].

4.4 TI C6713 DSP example

The peripherals available on TI's TMS320C6713 DSP are shown in Fig. 21 as boxes encircled by a yellow shape. The white boxes are components common to all C6000 devices, while grey boxes are additional features on the TMS320C6713 DSP.

Many peripherals are available on this DSP; however, there are pins that are shared by more than one peripheral and are internally multiplexed. Most of these pins are configured by software via a configuration register, hence they can be programmed to switch functionality at any time. Others (such as the HPI pins) are configured by external pullup/pulldown resistors at DSP chip reset; as a consequence, only one peripheral has primary control of the function of these pins after reset.

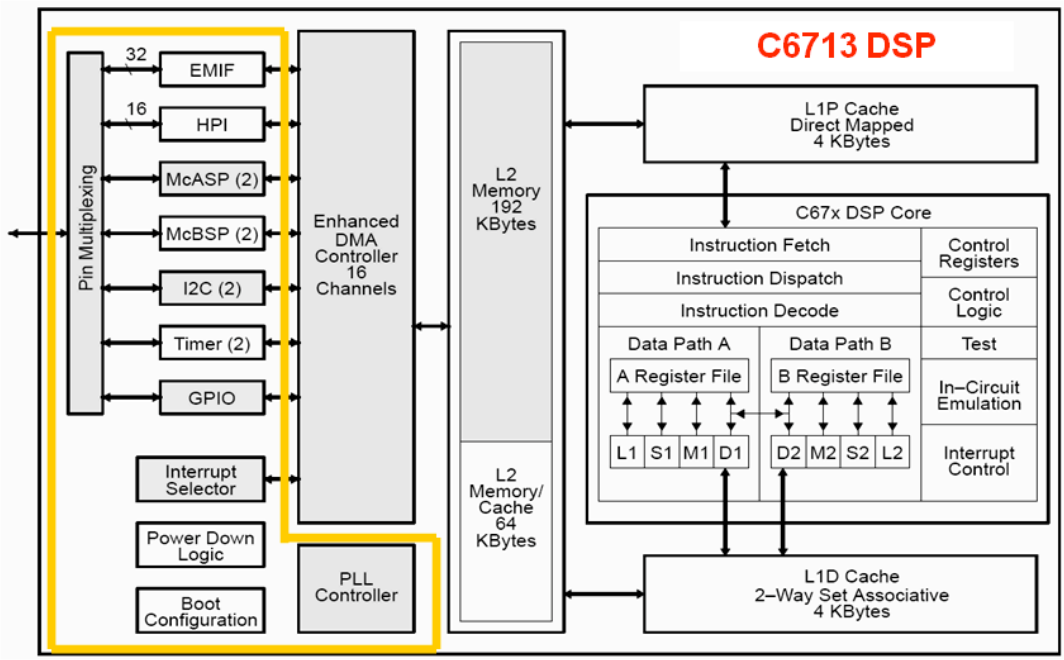


Fig. 21: TI TMS320C6713 DSP available peripherals. Picture courtesy of TI [23].

4.5 Memory interfacing

DSPs often have to interface with external memory, typically shared with host processors or with other DSPs. The two main mechanisms available to implement the memory interfacing are to use hardware interfaces already existing on the DSP chip or to provide external hardware that carries out the memory interfacing. These two methods are briefly mentioned below.

Hardware interfaces are often available on TI as well as on ADI DSPs. An example is TI External Memory Interface (EMIF) [38], which is a glueless interface to memories such as SRAM, EPROM, Flash, Synchronous Burst SRAM (SBSRAM) and Synchronous DRAM (SDRAM). On the TMS320C6713 DSP, for instance, the EMIF provides 512 Mbytes of addressable external memory space. Additionally, the EMIF supports memory width of 8 bits, 12 bits and 32 bits, including read/write of both big- and little-endian devices.

When no dedicated on-chip hardware is available, the most common solution for interfacing a DSP to an external memory is to add external hardware between memory and DSP, as shown in Fig. 22. Typically this is done by using a CPLD or an FPGA which implements address decoding and access arbitration. Care must be taken when programming the access priority and/or interleaved memory access in the CPLD/FPGA. This is essential to preserve the data integrity. Synchronous mechanisms should be preferred over asynchronous ones to carry out the data interfacing.

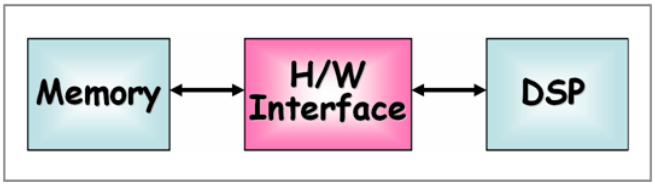


Fig. 22: Generic DSP–external memory interfacing scheme. Very often the h/w interface consists of a CPLD or an FPGA.

4.6 Data converter interfacing

DSPs provide a variety of methods to interface with data converters such as ADCs. On-chip peripherals are a very convenient data transfer mechanism, since data converters are typically much slower than the DSPs they are interfaced with, hence asking the DSP core to directly retrieve data from the converters is a waste of valuable processing time.

Serial interfaces are often available in TI's DSPs: peripherals such as McBSP and McASP plus the powerful DMA allow an easy interface to many data converter types [39, 40]. Another possible solution for TI DSPs is to use the EMIF in asynchronous mode together with the DMA.

In addition to serial interfaces, ADI Blackfin DSP provides a parallel interface, namely the PPI interface mentioned in Section 4.2, as a convenient way to interact with many converters. This interface typically allows higher sampling rates than the serial interfaces.

A general solution for implementing the DSP–data converter interface is to use an FPGA between DSP and converter, so as to re-buffer the data. An example of this hardware implementation for ADI Blackfin DSPs used in wireless portable terminals is given in Ref. [41]. Additional pre-processing, such as filtering or down-conversion, can also be carried out in the FPGA. This is the case for instance in CERN's LEIR LLRF system [42], where converters such as ADCs and DACs are hosted on daughtercards. Powerful FPGAs located on the same daughtercards carry out pre-processing and diagnostics actions under full DSP control.

Finally, mixed-signal DSPs, i.e., DSPs with embedded ADCs and/or DACs, are also available. An example of mixed-signal DSP is ADI's ADSP-21990, containing a pipeline flash converter with eight independent analog inputs and sampling frequency of up to 20 MHz.

4.7 DSP booting

The actions executed by the DSP immediately after a power-down or a reset are called DSP boot and are defined by a certain number of configurable input pins. This paragraph will focus on how the executable file(s) is uploaded to the DSP after a power-down or reset. Two methods are available, which typically correspond to differently built executables. More information on the code building process and on the many file extensions can be found in Section 6.4.

The first method is to use the JTAG connector to directly upload to the executable in the DSP. Upon a DSP power-down the code will typically not be retained in the DSP and another code upload will be necessary. This method is used during the system debugging phase, when additional useful information can be gathered via the JTAG.

On operational systems the DSP loads the executable code without a JTAG cable. Many methods are available for doing this, depending on the DSP family and manufacturer; some general ways are described below.

a) No-boot. The DSP fetches instructions directly from a pre-determined memory address, corresponding to EPROM or Flash memory and executes them. On SHARC DSPs, for instance, the pre-defined start address is typically 0x80 0004.

b) Host-boot. The DSP is stalled until the host configures the DSP memory. For TI TMS320C6xxx DSPs, for instance, this is done via the HPI interface. When all necessary memory is initialized, the host processor takes the DSP out of the reset state by writing in a HPI register.

c) ROM-boot. A boot kernel is uploaded from ROM to DSP at boot time and starts executing itself. The kernel copies data from an external ROM to the DSP by using the DMA controller and overwrites itself with the last DMA transfer. After the transfer is completed the DSP begins its program execution. Figure 23 visualizes the TI DSP process of booting from ROM memory: the program (shown in green) has been moved from ROM to L2 and L1 Program (L1P) cache via EMIF and DMA.

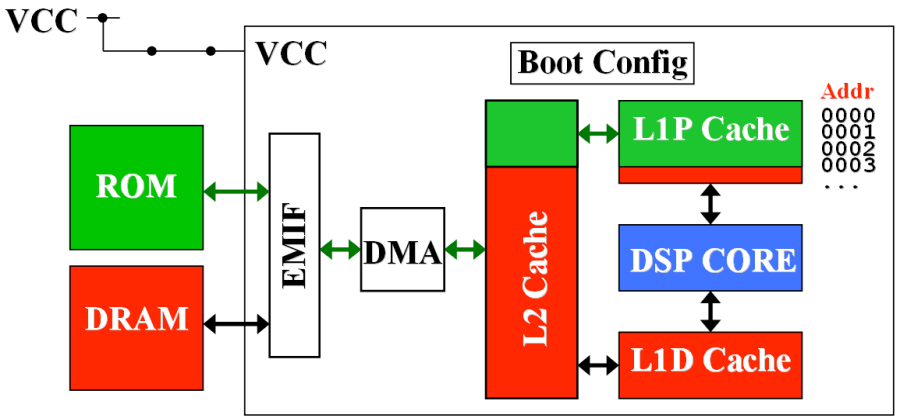


Fig. 23: Example of TI TMS320C6x DSP booting from ROM memory. The picture is courtesy of TI [43].

5 Real-time design flow: introduction

Figure 24 shows a time-ordered view of the various activities or phases that a real-time system developer may be required to carry out during a new system development. These activities will be treated in this document in a didactic rather than in a time-related order, to allow even the un-experienced reader to build up the knowledge needed at each step. It should be underlined that the real-time design flow may be not totally forward-directed, and at each step the developer may have to go back to a previous phase to make modifications or carry out additional tests.

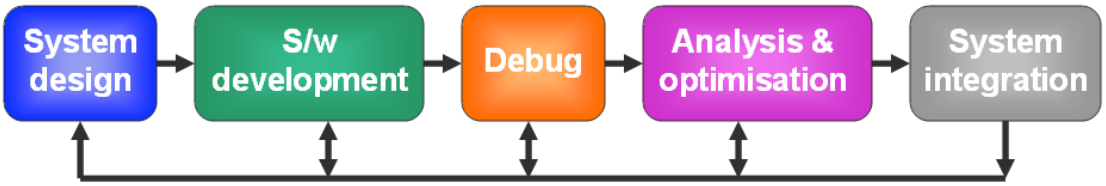


Fig. 24: Activities typically required to develop a new, DSP-based system

The 'system design' phase may include both hardware and software design. For hardware design, the developer must make choices such as the DSP type to use, the hardware architecture/interfaces, and so on. For software design, choices such as the code structure, the data flow and data exchange interfaces must be made. This phase is treated in Section 9.

The 'software development' phase includes creating the DSP project and writing the actual DSP code. Basic and essential information for this phase is given in Section 6.

The ‘debug’ phase is a very critical one, where the developer must verify that the code executes what it was meant to. Some debugging techniques as well as different methodologies available (such as simulation and emulation) are described in Section 7.

The ‘analysis and optimization’ phase allows the developer to optimize the system for different goals, such as speed, memory, input/output bandwidth, or power consumption. Analysis and optimization tools are described in Section 8, together with some optimization guidelines.

Finally, the ‘system integration’ is the essential phase where the system is integrated within the existing infrastructure and is therefore made fully operational. It is not possible to give precise details on this phase owing to the many existing control infrastructures. However, general guidelines and good practices are discussed in Section 10.

6 Real-time design flow: software development

DSPs are programmed by software via a cross-compilation. This means that the executable is created in a platform (such as a Windows- or a SUN-based machine) different from the one that it runs on, i.e., the DSP itself. One reason for this is that DSPs have limited and dedicated resources, hence it would not be convenient or even possible to run a file system with a user-friendly development environment.

The choice of programming languages is vast, including native assembly language as well as high-level languages such as C, C++, C extensions and dialects, Ada and so on. High-level software tools such as MATLAB and National Instruments allow one to automatically generate code files from graphical interfaces, thus providing rapid prototyping methods.

The code-building tools are very often provided by the DSP manufacturers themselves. Compilers and Integrated Development Environments (IDEs) are also available from other sources, such as Green Hills Software. The trend is now towards more powerful and user-friendly tools, capable of taming and using in the best possible way the underlying hardware and software complexity.

6.1 Development set-up and environment

DSP executables are developed by using Integrated Development Environments (IDEs) provided by DSP manufacturers; they integrate many functions, such as editing, debugging, project files management, and profiling. Very often the licences are bought on a ‘per-project’ basis, even if ADI provides also floating (i.e., networked) licences. The development environment for TI and ADI DSPs are called ‘Code Composer Studio’ and ‘VisualDSP++’, respectively; they provide very similar functionalities. It should be underlined that TI has recently made available free of charge the compiler, assembler, optimizer and linker to non-commercial users. However, neither the IDE nor a debugger were included, thus the developer must still use the proprietary tools.

Figure 25 gives an example of a typical Code Composer screen. On the left-hand side there is the list of all files included in the software project. At the centre of the screen two windows show the code, as a C file (*process.c*) and as assembly code (Dis-assembly window). A breakpoint has been set and the execution is stopped there. Below the code windows, two memory windows are also visible, detailing the data present at addresses 0x80000000 and following, and at addresses 0x40000030 and following. Data at address 0x80000002 is of a different colour because its value changed recently. At the bottom of the IDE screen the following item are displayed: a) the Compile/Link window, which details the results from the last code compilation; b) the Watch window, which displays the value assumed by two C-language variables and c) the Register window, which details the contents of all DSP registers. On the right-hand side there are three graphs: the yellow ones show memory regions, while the green one shows the Fast Fourier Transform of data stored in memory as calculated by the

IDE. The reader can find more details about Code Composer Studio and its capabilities in Refs. [44]–[46].

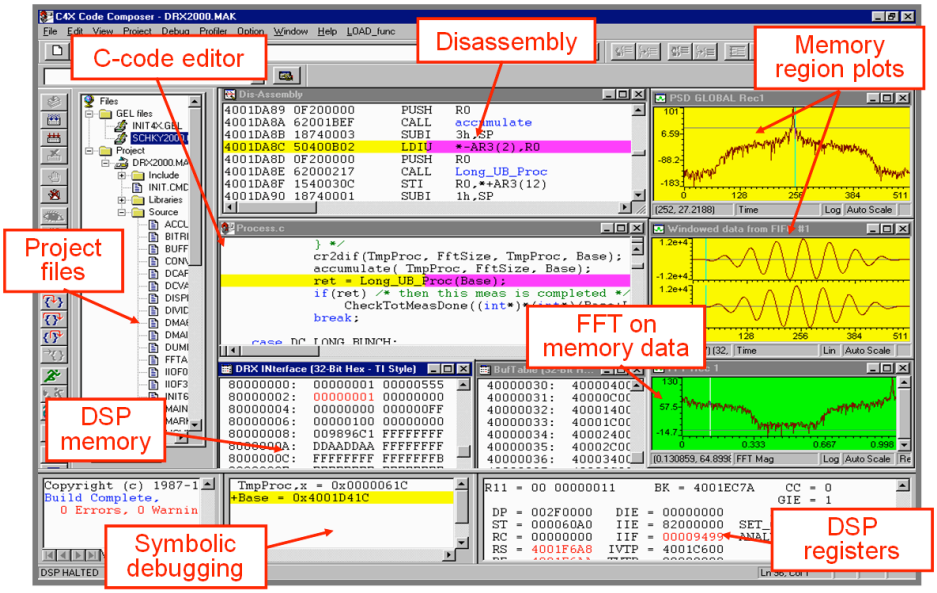


Fig. 25: Screenshot from Code Composer, i.e., the TI DSP IDE. The picture was taken in 1998 from the development of CERN's AD Schottky system.

Figure 26 shows a typical DSP-based system set-up. On the left-hand side the DSP IDE runs on a PC, which is connected to the DSP via a JTAG emulator and pod. This allows one to edit the code, compile it, download it to the hardware and retrieve debug information. On the right-hand side the system exploitation is shown whereby the DSP runs its program and a PowerPC board, running LynxOS and acting as master VME, controls the DSP actions, downloads the control parameters, and retrieves the resulting data. The example shown is that of CERN's AD Schottky measurement system [47].

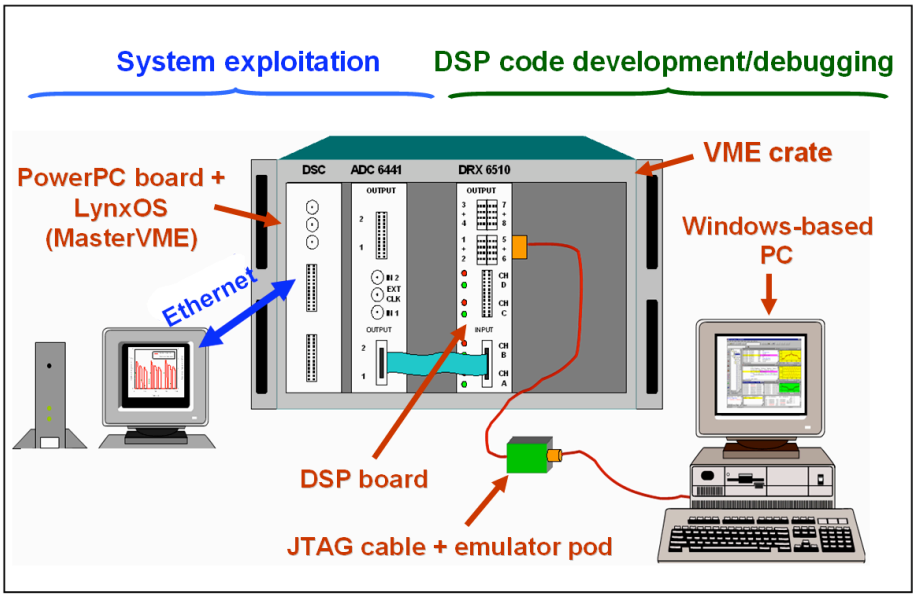


Fig. 26: Typical system exploitation (on the left-hand side) and code development (on the right-hand side) set-ups

6.2 Languages: assembly, C, C++, graphical

The choice of the language(s) to be used for the DSP development is very important and depends mainly on the selected DSP, as different DSPs may support different languages. Often a DSP system will include both assembly and high-level languages; the language choice or the chosen balance between the languages depends also on the required processor workload, i.e., on how much the code should be optimized to satisfy the requirements. The language choice is nowadays much larger than in the past, mainly thanks to the improvements of compilers. Additionally, the increased complexity of DSP hardware (see Section 3), such as deep pipelining, makes the hand-optimization much more difficult. The main language choices include: a) assembly language; b) high-level languages such as C, C dialects/extensions and C++; c) graphical languages such as Matlab. These three choices are discussed below.

6.2.1 Assembly language

The assembly language is very close to the hardware, as it explicitly works with registers and it requires a detailed knowledge of the inner DSP architecture. To write assembly code typically takes longer than to write high-level languages; additionally, it is often more difficult to understand other people's assembly programs than to understand programs written in high-level languages. The assembly grammar/style and the available instruction set/peripherals depend not only on the DSP manufacture, but also on the DSP family and on the targeted DSP. As a consequence, it might be difficult or even impossible to port assembly programs from one DSP to another. For instance, for DSPs belonging to the TI C6xxx family there is about an 85% assembly code compatibility, i.e., when going from a C62x to a C64x DSP there are no issues but if moving from a C64x to a C62x one might have to introduce some changes in the code owing to the different instruction set.

DSP applications have typically very demanding processing requirements. The need to obtain the maximum processing performance has often led DSP programmers to use assembly programming extensively. Nowadays the improvements in code compilers and the increasing difficulty in hand-optimizing assembly code have prompted DSP developers to use high-level languages more often. However, in some DSPs there are still features available only in assembly, such as the super-fast interrupt dispatcher for ADI's ADSP21160M DSP shown in Table 6. Very often, the bulk of the DSP code is written in high-level languages and the parts needing a better performance may be written in assembly.

Different manufacturers adopt different assembly styles, which have also evolved over the years. Table 7 shows a comparison between a traditional assembly style, adopted for instance by TI C40 DSPs, and the algebraic assembly, adopted by ADI SHARC DSPs.

Table 7: Comparison of assembly code styles. The traditional assembly style was adopted for instance by TI TMC320C4x DSPs, while the algebraic assembly is used in ADI.

Operation	Traditional assembly	Algebraic assembly
Move registers contents	mov R7, R0	R7 = R0
Addition	add R0, R1, R2	R0 = R1 + R2
Conditional jump	beq R1, R2, _loc	comp (R1,R2); if eq jump _loc;

Figure 27 gives an example of how one line of C code is converted to the corresponding assembly code for the TI C6317 DSP. The upper window shows part of the 'SIN_to_output_RTDX.c' file, which was included in the DSP laboratory companion of the lectures described in this document; the lower 'Disassembly' window shows the resulting assembly code.

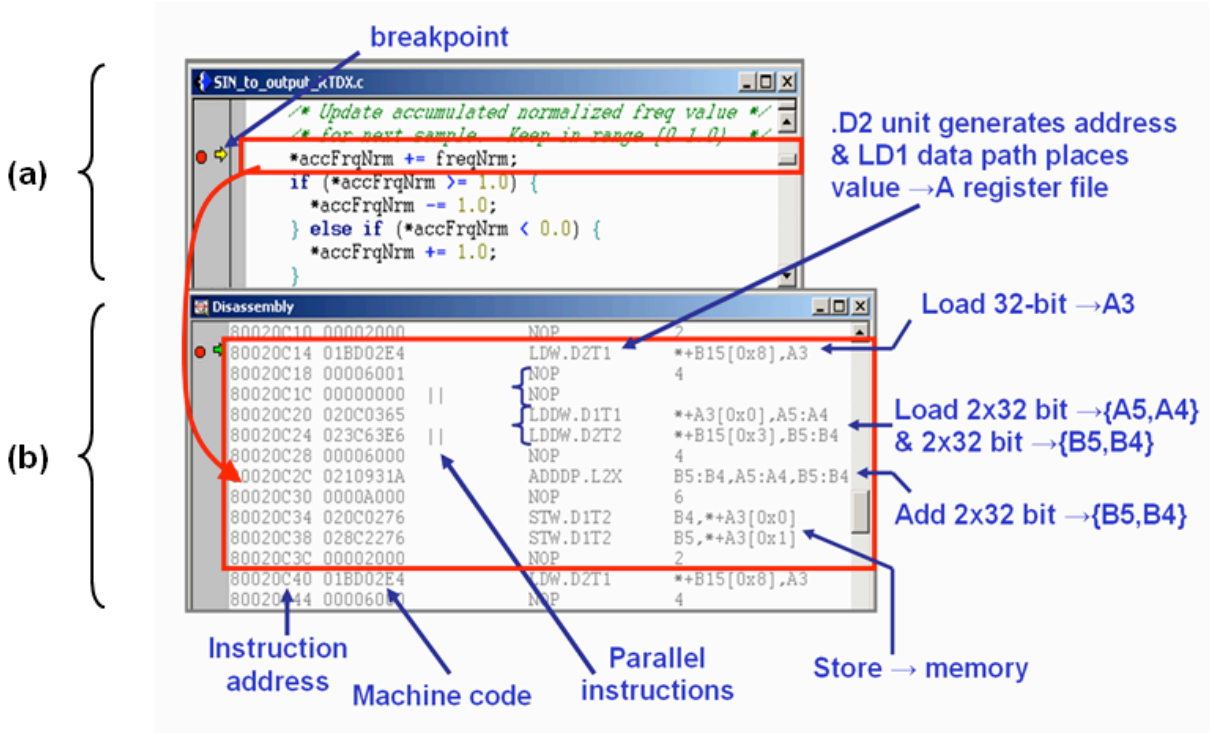


Fig. 27: C and assembly language examples for the TI C6713 DSP. Window (a): C source code. Window (b): assembly code resulting from the first C-code line in window (a).

6.2.2 High-level languages: C

The C language was developed in the early 1970s; three main standards exist, referred to as ANSI, ISO, and C99 respectively. There are many reasons why it is convenient to use the C language in DSP-based systems. The C language is very popular and known by engineers and software developers alike; it is typically easier to understand and faster to develop than assembly code. It supports features useful for embedded systems such as control structures and low-level bit manipulations. All DSPs are provided with a C compiler, hence it may be possible to port the C code from one DSP to another.

There are, however, drawbacks to the use of standard C languages in DSP-based systems. First, the executable resulting from a C-language source code is typically slower than that derived from optimized assembly code and has a larger size. The ANSI/ISO C language does not have support for typical DSP hardware features such as circular buffers or non-flat memory spaces. Additionally, the address at which data must be aligned can vary between different DSP architectures: on some DSPs a 4-byte integer can start at any address, but on other DSPs it could start for instance at even addresses only. As a consequence, the data alignment obtained with ANSI/ISO C compilers may be incompatible with the data alignment required by the DSP, thus leading to deadly bus errors. In the standard C language there is no native support for fixed-point fractional variables, a serious drawback for many DSPs and signal processing algorithms. Finally, the standard C compiler data-type sizes are not standardized and may not fit the DSP native data size, leading for instance to the replacement of fast hardware implementations with slower software emulations. For instance, 64-bit double operations are available in ADI's TigerSHARC as software emulations only; hence the declaration of

variables as double and not as float will result in slower execution. Table 8 shows how data-type sizes can vary for different DSPs.

Table 8: Examples of data-type size for different DSPs

char size	Processor
8	ADI Blackfin
16	ADI '21xx, TI 'C54, 'C55
24	Freescale 56x
32	ADI Blackfin, TI 'C6x
32	ADI SHARC, TigerSHARC

(a)

int size	Processor
16	ADI '21xx, TI 'C54, C55
24	Freescale 56x
32	ADI Blackfin, TI 'C6x
32	ADI SHARC, TigerSHARC

(b)

Table 9 shows the data-type sizes and number format for the TI C6713 DSP. The 2's complement and binary formats are used for signed and unsigned numbers, respectively.

Table 9: Data-type sizes and number format for the TI C6713 DSP

TI 'C6713 DSP		
Data type	# bits	Representation
char	8	ASCII
short	16	2's complement / binary
int	32	2's complement / binary
long	40	2's complement / binary
float	32	IEEE 32-bit
double	64	IEEE 64-bit

There are two main approaches to adapting the C language to specific DSPs hardware and to the needs of signal processing applications. The first approach is the definition of 'intrinsic' functions, i.e., of functions that map directly to optimized DSP instructions. Table 10 shows some examples of intrinsic functions available in TI C6713 DSPs. The second approach is to 'extend' the C-language so as to include specialized data types and constructs. Of course, the drawback of the latter approach is a reduced portability of the resulting C language.

Table 10: TI C6713 intrinsic functions – some examples.

Intrinsic	Description
double _rsqrdp(double src);	Returns approximate 64-bit double square root reciprocal
double _fabs(double src);	Returns absolute value of src
unit _enable_interrupts(void);	Returns previous interrupt state & enables interrupts

6.2.3 High-level languages: C++

The C++ programming language supports object-oriented programming and is the language of choice for many business computer applications. C++ compilers are often available for DSPs; some advantages of using it are the ability to provide a higher abstraction layer and the upwards compatibility with the C language. There are, however, several disadvantages, for instance the increased memory requirements due to the more general constructs. Additionally, many application programs and libraries rely on functions such as *malloc()* and *free()*, which need a heap.

While the way to adapt the C-language to DSPs is to add features, the C++ language is adapted by trimming its features. C++ characteristics typically removed are multiple inheritance and exception handling; the resulting code is more efficient and the executable is smaller.

6.2.4 Graphical languages

A trend which has developed over the last five to ten years is to use graphical programming to generate DSP code. Examples of programs and tools aimed at this are the MATLAB, Hypersignal RIDE (now acquired by National Instruments) and the LabVIEW DSP Module. These methodologies generate DSP executables that often are not highly optimized, therefore not suitable for the implementation of demanding DSP-based systems. However, they allow one to quickly move from the design to the implementation phase, thus providing a rapid prototyping methodology.

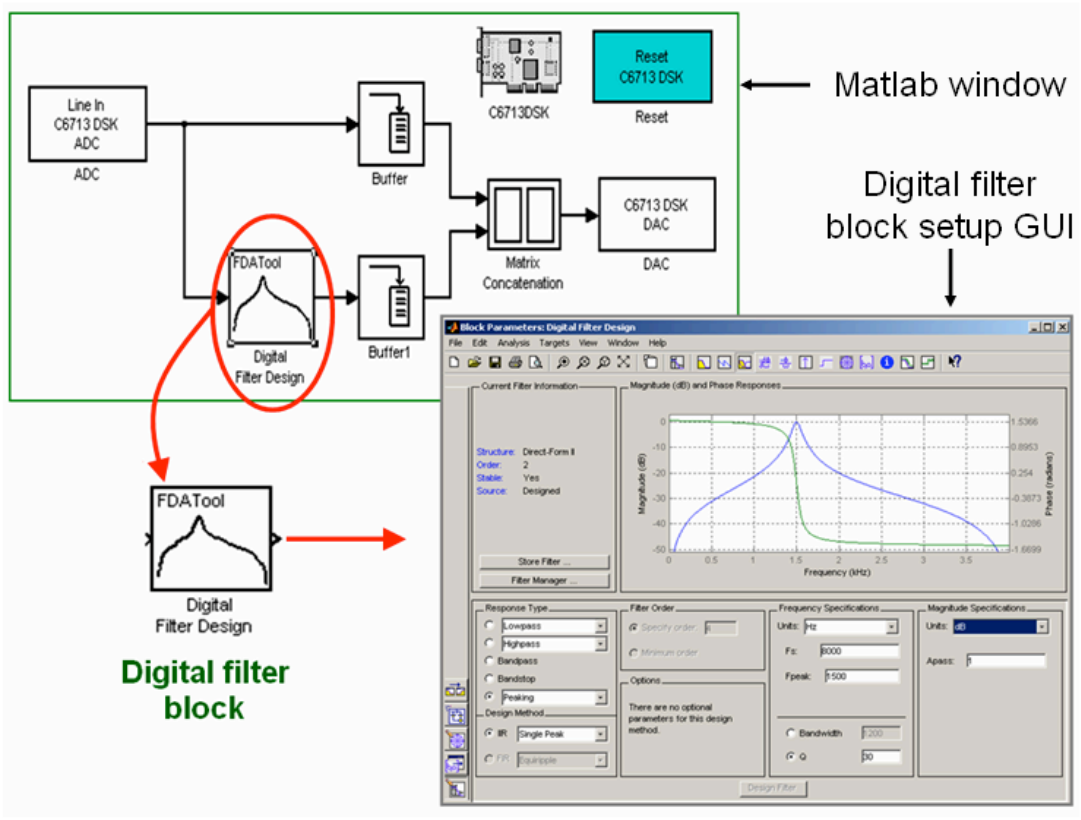


Fig. 28: MATLAB graphical programming used in the DSP laboratory companion in these notes. The digital filter block can easily be set up by using a user-friendly set-up GUI.

As an example, MATLAB provides tools such as Simulink, Real-Time Workshop, Real-Time Workshop Embedded Coder, Embedded Target for TI C6xxx DSPs and Link for Code Composer that allow generating embedded code for TI DSPs and downloading it directly into a DSP evaluation

board. These tools provide interfaces for the DSP peripherals, too. The DSP laboratory companion on these notes was based upon TI C6713 DSK and MATLAB tools. Figure 28 shows the MATLAB graphical program that constituted one of the laboratory exercises. MATLAB allows not only to interface immediately with the on-board CODEC by using the ADC and DAC blocks, but also to set up through a user-friendly GUI the digital filter to be implemented.

6.3 Real-time operating system

A Real-Time Operating System (RTOS) is a program that has real-time capabilities, is downloaded to the DSP at boot time, and manages all DSP programs, typically referred to as tasks. The RTOS interfaces tasks with peripherals such as DMA, I/O and memory, via an Application Program Interface (API), as shown in Fig. 29.

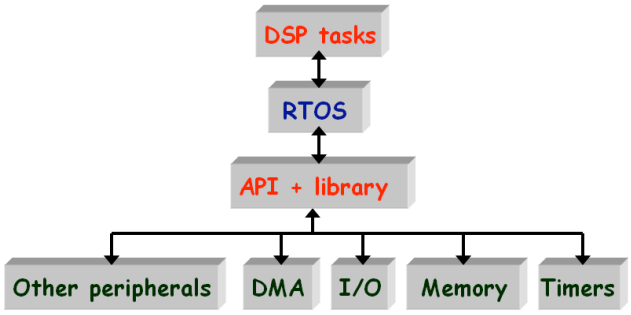


Fig. 29: Embedded DSP software components

A RTOS is typically task-based and supports multiple tasks (often referred to as threads) by time-sharing, i.e., by multiplexing the processor time over the active tasks set. Each task has a priority associated to it and the RTOS schedules which task should run depending on the priority. Very often this is done in a pre-emptive way, meaning that when a high-priority task becomes ready for execution, it pre-empts the execution of a lower-priority task, without having to wait for its turn in the regular re-scheduling. Finally, RTOS have a small memory footprint, so as not to have too negative an impact on the DSP executable size.

There are many advantages when using a RTOS to develop a DSP-based system. For instance, the API and library shown in Fig. 29 provide a device abstraction level between DSP hardware features and task implementation, thus allowing a DSP developer to focus on the task rather than the hardware interface's design and coding. The DSP developer may have to just call different interfacing functions in case the code should be ported to a different DSP, hence easing code portability. A RTOS manages the task's execution hence the developer can cleanly structure the code, define appropriate priority levels for each task, and insure that their execution meets critical real-time deadlines. System debug and optimization can be improved, and memory protection can often be provided. There are, however, drawbacks to the use of RTOS. As an example, a RTOS uses DSP resources, such as processing time and DSP timers, for its own functioning. Additionally, the RTOS turnover is typically quite high and royalties are often required from developers.

Many RTOS are available at any time, typically targeted to a precise DSP family or processor. Examples are TALON RTOS from Blackhawk, targeted at TI DSPs, and INTEGRITY RTOS from Green Hills Software or NUCLEUS RTOS from Accelerated Technology, targeted at ADI Blackfin DSPs. It is worth mentioning Linux-based OS, such as RT-Linux, RTAI and uLinux. Both RT-Linux and RTAI use a small real-time kernel that runs Linux as a real-time task with lower priority. The last RTOS listed above, uLinux, is a soft-time OS adapted to ADI Blackfin DSPs. uLinux cannot always guarantee RTOS capabilities such as a deterministic interrupt latency; however, it can typically satisfy the needs of commercial products, where time constraints are often on the millisecond order as dictated by the ability of the user to recognise glitches in audio and video signals.

Other RTOS worth mentioning are those provided and maintained by DSP manufacturers. Both TI and ADI provide royalties-free RTOS with similar characteristics, such as a small memory footprint, multi-tasks and multi-priority levels support. They are called DSP/BIOS [48] for TI and VisualDSP++ Kernel (VDK) for ADI, and can optionally be included in the DSP code. In particular, TI DSP/BIOS provides thirty priority levels and four classes of execution threads. The thread classes, listed in order of decreasing priority, are Hardware Interrupts (HWI), Software Interrupts (SWI), Tasks (TSK) and Background (IDL). Figure 30 shows how the processing time is shared between different threads in TI DSP/BIOS. In the vertical scale the different threads are ordered by priority, the higher up having more priority; in the horizontal scale the time is shown. Software interrupts can be pre-empted by a higher-priority software interrupt or by a hardware interrupt. Same-level interrupts are executed in a first-come, first-served way. Tasks are capable of suspension (see Task TSK2 in Fig. 30) as well as of pre-emption.

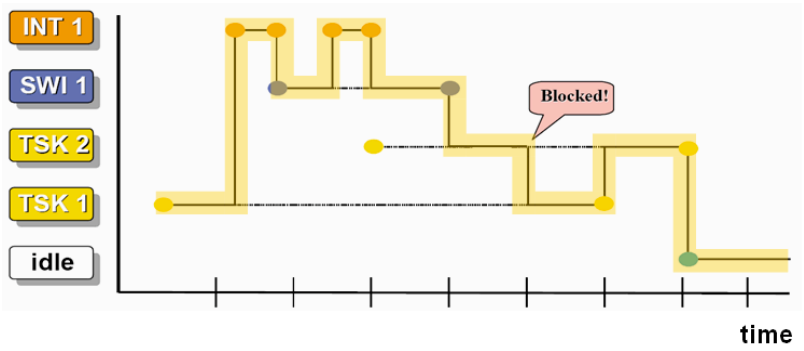


Fig. 30: DSP/BIOS prioritized thread execution example. Image courtesy of Texas Instruments [48].

6.4 Code-building process

The DSP code-building process relies on a set of software development tools, typically provided by DSP manufacturers.

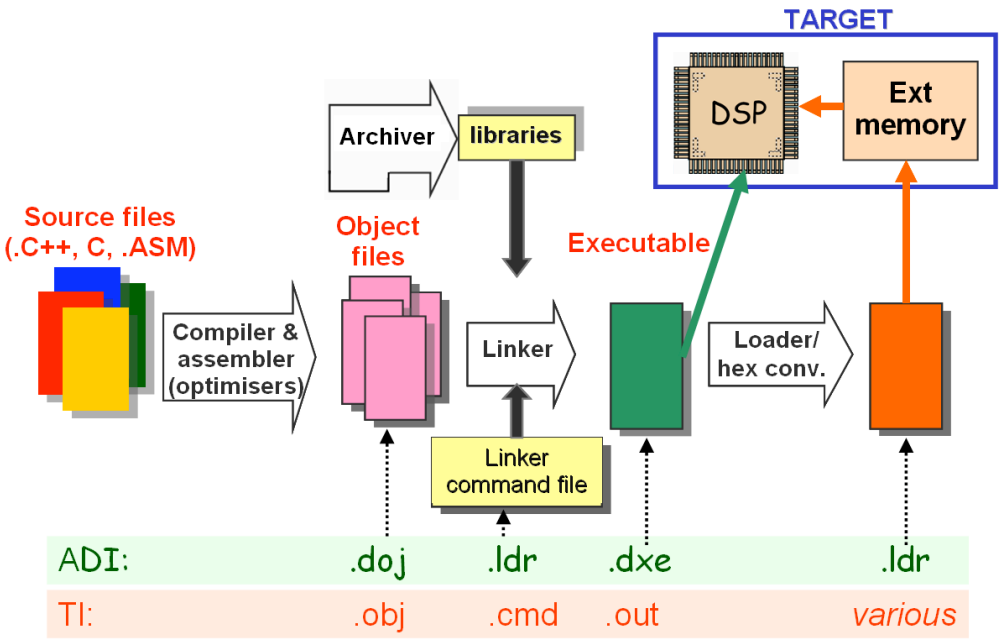


Fig. 31: Main elements of the code building process. Typical file extensions for ADI and TI DSPs are shown at the bottom of the picture.

Figure 31 shows the main elements and tools needed for the code-building process. Source files are converted to object files by the compiler and the assembler. Archiver tools allow the creation of libraries from object files; these libraries can then be linked to object files to create an executable. The executable can be directly downloaded from the IDE to the target DSP via a JTAG interface; as an alternative, the executable can be converted to a special form and loaded to a memory external to the DSP, from which the DSP itself will boot. The first approach is typically used during the DSP development phase, while the second approach is more convenient during system exploitation. Finally, the file extensions used at the different code-building process steps for ADI and TI DSPs are shown at the bottom of Fig. 31.

Three tools, namely compiler, assembler, and linker, are used to generate executable code from C/C++ or assembly source code. Figure 32 shows their use in the code-building process on TI DSPs. The tools' main characteristics are summarized in Sub-sections 6.4.1 to 6.4.3.

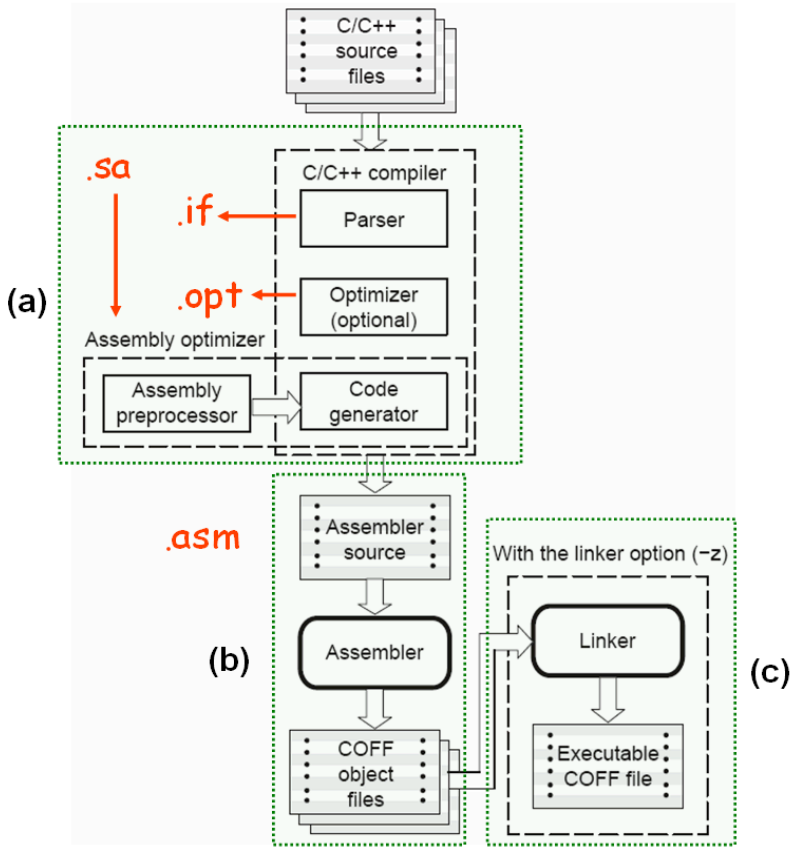


Fig. 32: Generic code-building processing: (a) compiler; (b) assembler; (c) linker. The picture is courtesy of TI [49].

6.4.1 C / C++ compiler for TI C6xxx DSPs [49]

The C/C++ compiler generates C6xxx assembler code (.asm extension) from C, C++ or linear assembly source files. The compiler can perform various levels of optimization: high-level optimization is carried out by the optimizer, while low-level, target-specific optimization occurs in the code generator. Finally, the compiler includes a real-time library which is non-target-specific.

6.4.2 Assembler for TI'C6xxx DSPs [50]

The assembler generates machine language object files from assembly files; the object files format is the Common Object File Format (COFF). The assembler supports macros both as inline functions and

taken from a library; it also allows segmenting the code into sections, a section being the smaller unit of an object file. The COFF basic sections are

a) text for the executable code;

b) data for the initialized data;

c) bss for the un-initialized variables.

6.4.3 Linker for TI C6xxx DSPs [50]

The linker generates executable modules from COFF files as input. It resolves undefined external references and assigns the final addresses to symbols and to the various sections. A DSP system typically includes many types of memory and it is often up to the programmer to place the most critical program code and data into the on-chip memory. The linker allows allocating sections separately in different memory regions, so as to guarantee an efficient memory access. An example of this is shown in Fig. 33.

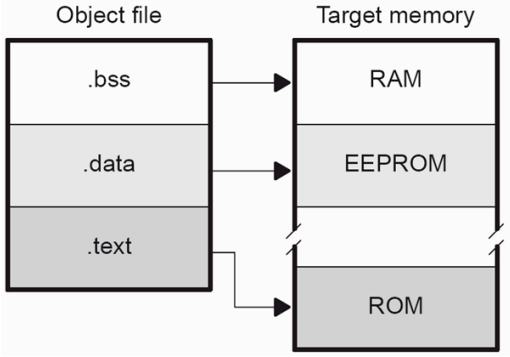


Fig. 33: Example of sections allocation into different types of target memory

The linker also allows one to clearly implement a memory map shared between DSP and host processor; this is essential for instance to exchange data between them.

7 Real-time design flow: debugging

The debugging phase is the most critical and least predictable phase in the real-time design flow, especially for large systems. The debugging capabilities of the development environment tools can make the difference between creating a successful system and spiralling into an endless search for elusive bugs.

The starting point of this phase is an executable code, i.e., a code without compilation and linker errors; the goal is to ascertain that the code behaves as expected. The debugging tools and techniques have a strong impact on the amount of time and effort needed to validate a DSP code.

There are many types of bugs: they can be repeatable or intermittent, the latter being much tougher to track down than the first ones. Bugs can be due to the code implementation, such as logical errors in the source code, or can derive from external problems, i.e., hardware misbehaviours. The approaches and the tools to debug a DSP code include simulation, emulation, and real-time debugging techniques. Simulation tools allow running the DSP code on a software simulator fitted with full visibility into DSP internal registers. Emulation tools embed debug components into the target to allow an information flow between target and host computer. Real-time debugging techniques allow a real-time data exchange between host and target without stopping the DSP. These techniques are described in detail in Sections 7.1. to 7.3.

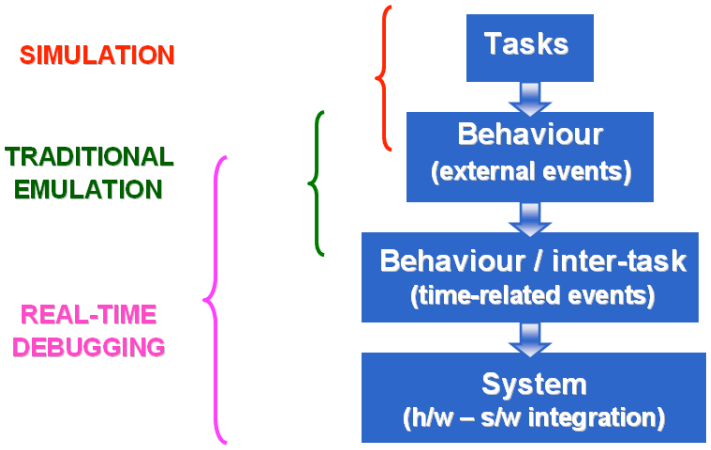


Fig. 34: Debug steps and their suggested sequencing. The debug tools suited to different steps are also shown.

The developer should not attempt to debug the DSP code as a whole, unless the code itself is relatively short and simple. He is instead recommended to debug the code in several steps: Fig. 34 shows an example of steps and of their sequencing, together with the appropriate debug tools and techniques. First, single tasks such as functions and routines should be validated; this step can be carried out via simulation only. Second, the behaviour of sub-systems or specific parts of the code can be tested with respect to external events, such as ISR triggering. This part can be carried out with the help of traditional emulation techniques. Third, the behaviour of many tasks can be validated with respect to real-time constraints, such as the proper frequency of ISR triggering. Once all system components have been validated, the whole system can be tested. These last two steps profit particularly from real-time debugging techniques.

7.1 Simulation

DSP software simulators have been available for more than fifteen years. They can simulate CPU instruction sets as well as peripherals and interrupts, thus allowing DSP code validation at a reduced cost and even before the hardware the code should run on is available. Simulators provide a high visibility into the simulated target, in that the user can execute the code step by step and look at the intermediate values taken by internal DSP registers. Large amount of data can be collected and analysed; resource usage can be evaluated and used for an optimized hardware design.

Simulators are highly repeatable, since the same algorithm can be run in exactly the same way over and over. The reader should note that this kind of repeatability is difficult to obtain with other techniques, such as emulation, as external events (for instance interrupts) are almost impossible to be precisely repeated with hardware. Simulators may also allow measurement of the code execution time, with limitations due to the type of simulator chosen. A useful feature available with the TI C5x and C6x simulators is the ‘rewind’ [51], which allows viewing the past history of the application being executed on the simulator.

The main limitation common to DSP simulators is their execution speed, several orders of magnitude slower than the target they simulate; in particular, the more accurate the modelling of the DSP chip and corresponding peripherals, the slower the simulation. DSP tool vendors have overcome this problem by providing different simulators for the same DSP, providing a different level of chip and peripherals modelling. Figure 35 shows some simulators available for TI DSPs. The reader should notice that TI provides up to three simulators for each DSP, namely:

a) CPU Cycle Accurate Simulator: This simulator models the instruction set, timers, and external interrupts, allowing the debugging and optimization of the program for code size and CPU cycles.

b) Device Functional Simulator: This simulator not only models instruction set, timers, and external interrupts, but also allows features such as DMA, Interrupt Selector, caches and McBSP to be programmed and used. However, the true cycles of a DMA data transfer are not simulated.

c) Device Cycle Accurate Simulator: This simulator models all peripherals and caches in a cycle-accurate manner, thus allowing the user to measure the total device and stall cycles used by the application.

More information on TI simulators can be found in Ref. [52].

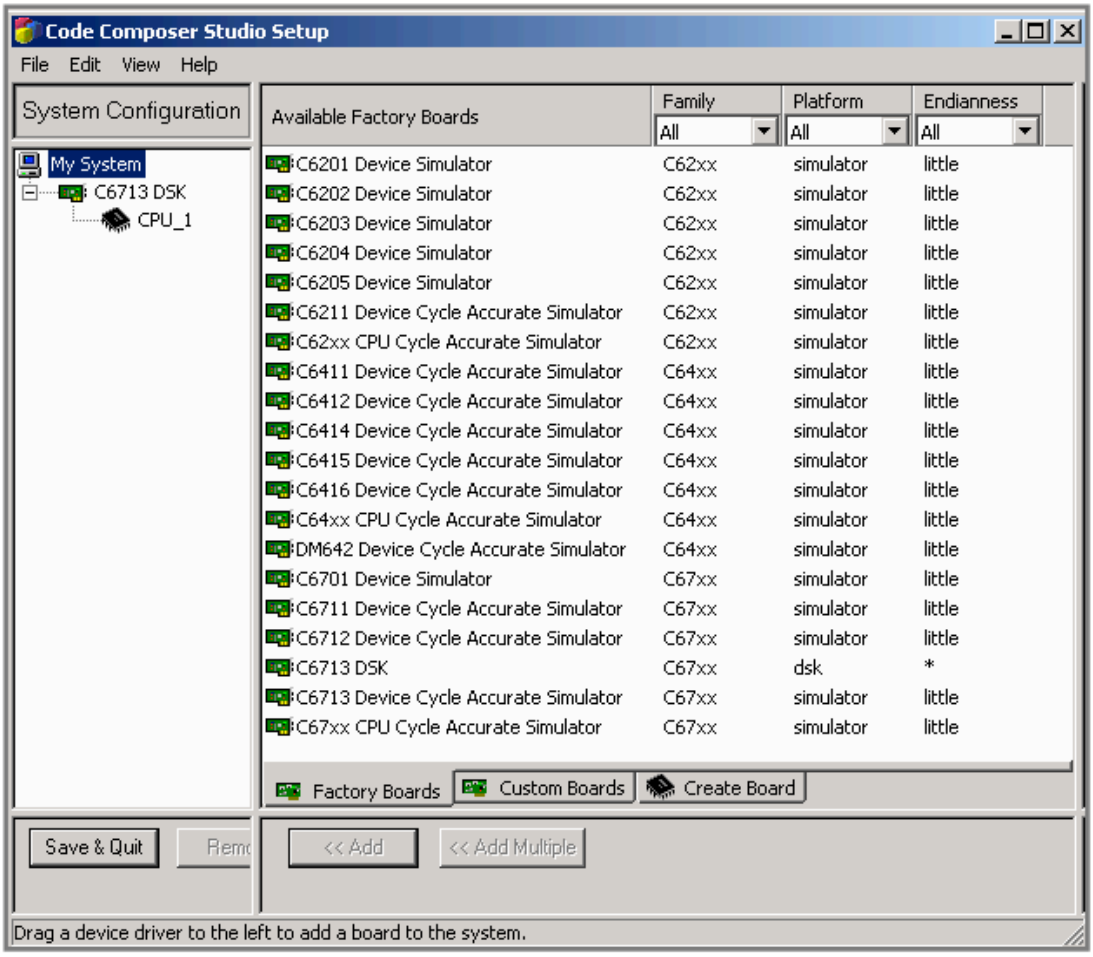


Fig. 35: Example of DSP simulators available with TI's Code Composer Studio development environment

7.2 Emulation

The integration of processor, memory, and peripherals in a single silicon chip is commonly referred to as System-On-a-Chip (SOC). This approach allows reducing the physical distance between components, hence devices become smaller in size, run faster, cost less to manufacture, and are typically more reliable. From a DSP code developer's viewpoint, the main disadvantage of this approach is the lack of access to embedded signals, often referred to as vanishing visibility. In fact,

many chip packages (e.g., ball grid array) do not allow probing the chip pins; additionally, internal chip busses are often not even available at the chip pins. Emulation techniques [53] restore the visibility needed for code debugging by embedding debug components into the chip itself.

There are three main kinds of emulation, namely:

a) Monitor-based emulation: A supervisor program (called monitor) runs on the DSP and uses one of the processor's input–output interfaces to communicate with the debugger program running on the host. The debugging capabilities of this approach are more limited than those provided by the two other approaches; additionally, the monitor presence changes the state of the processor, for instance regarding the instruction pipeline. The advantage is that it does not require emulation hardware, hence its cost is lower.

b) Pod-based In Circuit Emulation (ICE): The target processor is replaced by a device that acts like the original device, but is provided with additional pins to make accessible and visible internal structures such as internal busses. This emulation approach has the advantage of providing real-time traces of the program execution. However, replacing the target processor with a different and more complex device may create electrical loading problems. Additionally, this solution is quite costly, the hardware is different from the commercialized product and becomes quite difficult to implement at high processor speed.

c) Scan-based emulation: Dedicated interfaces and debugging logic are incorporated into commercially-available DSP chips. This on-chip logic is responsible for monitoring the chip's real-time operations, for stopping the processor when for instance a breakpoint is reached, and for passing debugging information to the host computer. An emulation controller controls the flow of information to /from the target and can be located either on the DSP board or on an external pod. Many types of target–host interface exist. On the DSP board one can typically find a JTAG (IEEE standard 1149.1) connector. On the host computer, parallel or USB ports are often available.

The scan-based emulation technique has been widely preferred over the other two since the late 1980s and is nowadays available on the vast majority of DSPs. Figure 36 shows the TI XDS560 emulator, composed of a PC card, a cable with JTAG interface to the target, and an emulation controller pod. Many emulators are available on the market, with different interfaces and characteristics. As an example, it is worth mentioning Spectrum Digital's XDS510 USB galvanic JTAG emulator, which provides voltage isolation.

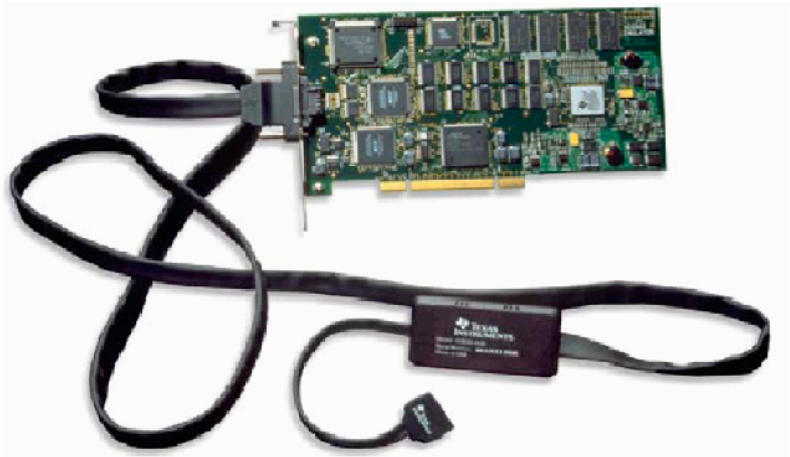


Fig. 36: TI XDS560 emulator, composed of a card to install on the host computer (PCI interface), a JTAG cable and an emulation controller pod

Capabilities of scan-based emulators include source-level debugging, i.e., the possibility to see the assembly instructions being executed and to access variables and memory locations either by name or by address.

Capabilities such as writing to the standard output are available. As an example, the *printf()* function allows printing DSP information on the debugger GUI; the reader should, however, be aware that this operation can be extremely time-consuming, and optimized functions (such as *LOG_printf()* for TI DSPs) should be preferred.

Another common capability supported by emulation technology is the breakpoint. A breakpoint freezes the DSP and allows the developer to examine DSP registers, to plot the content of memory regions, and to dump data to files. Two main forms of breakpoint exist, namely software and hardware. A software breakpoint replaces the instruction at the breakpoint location with one creating an exception condition that transfers the DSP control to the emulation controller. An hardware breakpoint is implemented by using custom hardware on the target device. The hardware logic can for instance monitor a set of addresses on the DSP and stop the DSP code execution when a code fetch is performed at a specific location. Breakpoints can be triggered also by a combination of addresses, data, and system status. This allows DSP developers to analyse the system when for instance it hangs, i.e., when the DSP program counter branches into an invalid memory address. Intermittent bugs can also be tracked down.

It is important to underline that the debugging capabilities provided by emulators allow mostly 'stop-mode debugging', in that the DSP is halted and information is sent to the host computer at that moment. This debugging technique is invasive and allows the developer to get isolated, although very useful, snapshots of the halted application. To improve the situation, DSP tool vendors have developed a more advanced debugging technology that allows real-time data exchange between target and host. This technique is described next.

7.3 Real-time techniques

Over the last ten years, DSP vendors have developed techniques for a real-time data exchange between target and host without stopping the DSP and with minimal interference on the DSP run. This provides a continuous visibility into the way the target operates. Additionally, it allows the simulation of data input to the target.

ADI's real-time communication technology is called Background Telemetry Channel (BTC) [54]. This is based upon a shared group of registers accessible by the DSP and by the host for reading and writing. It is currently supported on Blackfin and ADSP-219s DSPs only.

TI's real-time communication technology is called Real Time Data eXchange (RTDX) [55, 56]. Its main software and hardware components are shown in Fig. 37. A collection of channels, through which data is exchanged, are created between target and host. These channels are unidirectional and data can be sent across them asynchronously. TI provides two libraries, the RTDX target library and the RTDX host library, that have to be linked to target and host applications, respectively. As an example, the target application sends data to the host by calling functions in the RTDX target library. These functions buffer the data to be sent and then give the program flow control back to the calling program; after this, the RTDX target library transmits the buffered data to the host without interfering in the target application. RTDX is also supported when running inside a DSP simulator; to that end, the DSP developer should link the target application with the RTDX simulator target library corresponding to the chosen target. On the host side, data can be visualized and treated from applications interfacing with the RTDX host library. On Windows platforms a Microsoft Component Object Module (COM) interface is available, allowing clients such as VisualBasic, VisualC++, Excel, LabView, MATLAB and others.

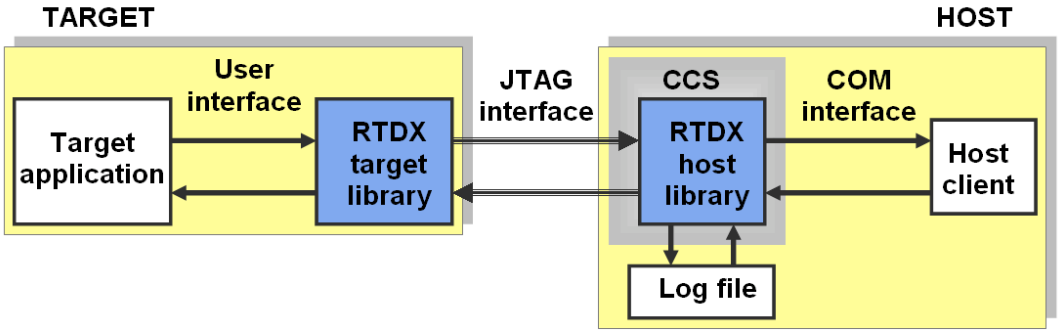


Fig. 37: TI's RTDX main components. The picture is courtesy of TI [56].

In 1998 TI implemented the original RTDX technology, which runs on XDS510-class emulators. A high-speed RTDX version was developed later that relies on additional DSP chip hardware features and on improved emulators, namely the XDS560 class. These emulators make use of two non-JTAG pins in the standard TI JTAG connector to increase RTDX bandwidth. They are also backwards compatible and can support standard RTDX, thus allowing higher data transfer speed. The high-speed RTDX is supported in TI's highest performance DSPs, such as the TMS320C55x, TMS320C621x, TMS320C671x and TMS320C64x families. Table 11 shows the data transfer speeds available with different combinations of RTDX and emulators. RTDX offers a bandwidth of 10 to 20 kbytes/s, thus enabling real-time debugging of applications such as CD audio and audio telephony. The high-speed RTDX with XDS560-class emulators provides a data transfer speed higher than 2 Mbytes/s, thus allowing real-time visibility into applications such as ADSL, hard-disk drives and videoconferencing [57].

Table 11: Data transfer speed as a function of the emulator type for TI's RTDX

Emulation type	Speed
RTDX + XDS510	10–20 kbytes/s
RTDX + USB (ex: 'C6713 DSK board)	10–20 kbytes/s
RTDX + XDS560	≤ 130 kbytes/s
High speed RTDX + XDS560	> 2 Mbytes/s

8 Code analysis and optimization

Most DSP applications are subject to real-time constraints and stress the available CPU and memory resources. As a consequence, code optimization might be required to satisfy the application requirements.

DSP code can be optimized according to one or more parameters such as execution speed, memory usage, input/output bandwidth, or power consumption. Different parts of the code can be optimized according to different parameters. A trade-off between code size and higher performance exists, hence some functions can be optimized for execution speed and others for code size.

Code development environments typically allow defining several code configuration releases, each characterized by different optimization levels. Figure 38 shows the project configurations available in TI Code Composer Studio. The 'Release' configuration comprises the higher optimization

level, while the 'Debug' configuration enables debug features, which typically increase code size. Finally, the user can specify a 'Custom' configuration where user-selectable debug and optimization features are enabled.

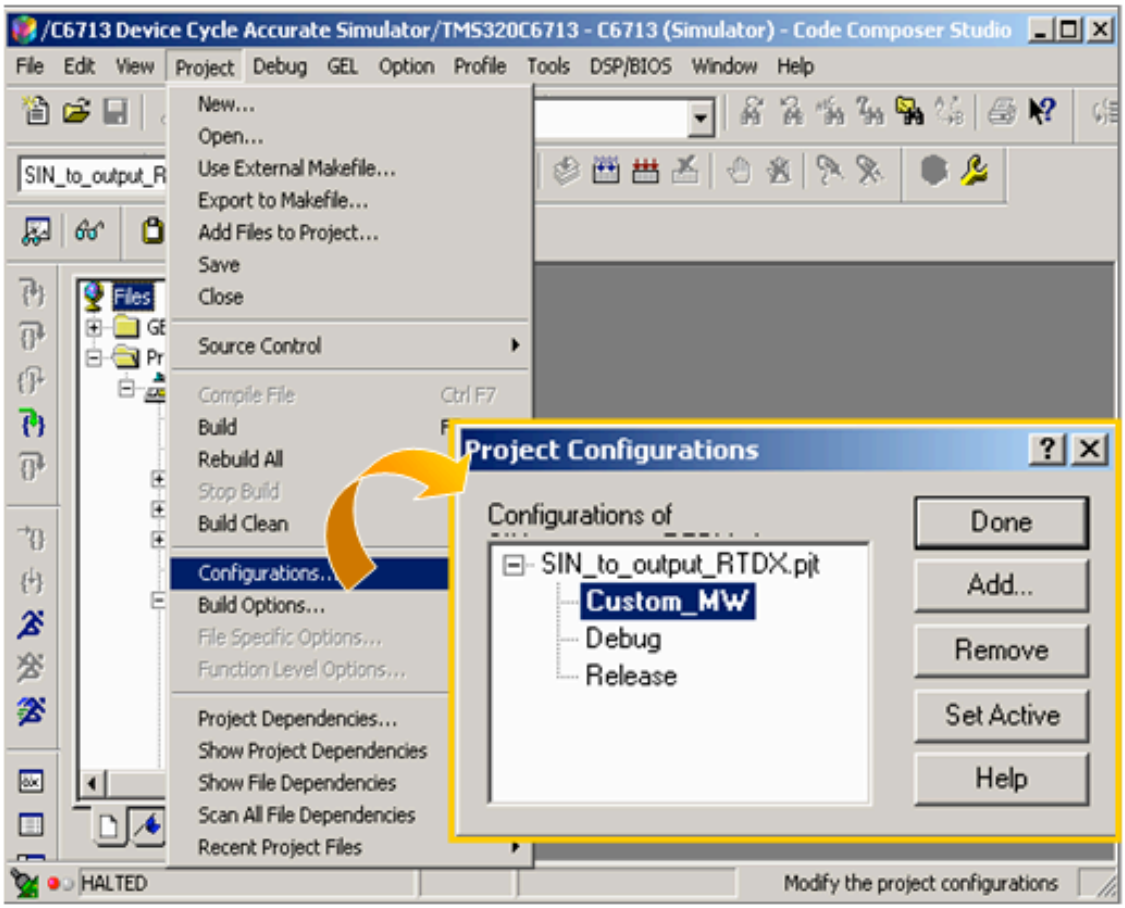


Fig. 38: Choice of the DSP code project configurations in TI Code Composer

It is important to underline that debug and optimization phases are different and often conflicting. In fact, an optimized code does not include any debug information; additionally, the optimizer can re-arrange the code so that the assembly code is not an immediate transposition of the original source code. The reader is strongly encouraged to avoid using the debug and the optimize options together; it is recommended instead to first debug the code and only then to enable the optimization.

8.1 Switching the code optimizer ON

Compilers are nowadays very efficient at code optimization, allowing DSP developers to write higher level code instead of assembly. To do this, compiles must be highly customized, i.e., tightly targeted to the hardware architecture the code will be running upon. However, current trends in software engineering include retargeting compilers to DSP specialized architectures [58].

As previously mentioned, many kinds of optimization can be required. An example is execution speed vs. executable size. Figure 39 shows how the user can select one or the other in the Code Composer Studio development environment.

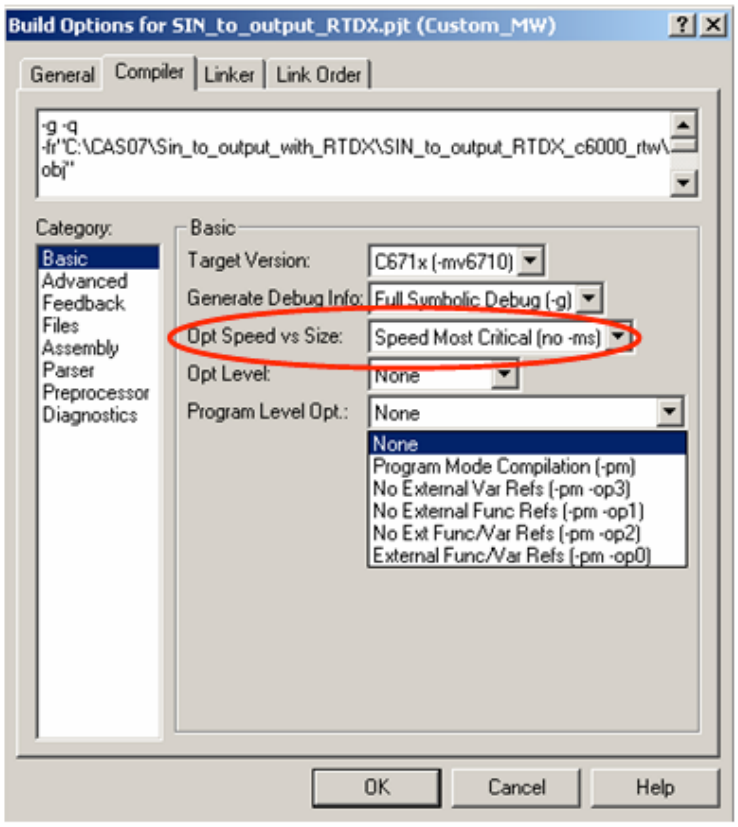


Fig. 39: Choice of optimization levels in TI Code Composer. The plot highlights execution speed vs. executable code size.

The reader should be aware that the optimizer can rearrange the code, hence the code must be written in a proper way. Failing this, the actions generated by the optimized code might be different from those desired and implemented by a non-optimized code. Figure 40 shows two code snippets where the value assumed by the memory location pointed to by *ctrl* determines the *while()* loop behaviour. In particular, the DSP exits the *while()* loop if the *ctrl* content takes the value 0xFF; the *ctrl* content can be modified by another processor or execution thread. Both code snippets will perform equally in case of non-optimization. However, in case of optimization the left-hand side code will not evaluate the *ctrl* content at every *while()* iteration, hence the DSP will remain forever in the loop. On the right-hand side snippet, the *volatile* keyword disables memory optimization locally, thus forcing the DSP to re-evaluate the *ctrl* content value at every *while()* loop iteration. This guarantees the desired behaviour even when the code is optimized. The number of *volatile* variables should be restricted to situations where they are strictly needed, as they limit the compiler's optimization.

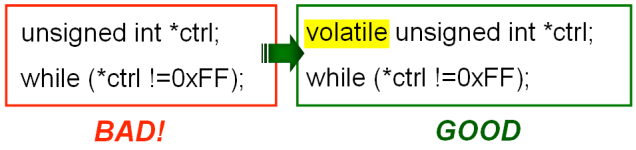


Fig. 40: Example of good and bad programming techniques. The left-hand side code would likely result in a programming misbehaviour.

The recommended code development flow is to first write high-level code, such as C or C++. This code can then be debugged and optimized, to comply with the specified performance. In case the code runs still slower than desired, the time-critical areas can be re-coded in linear assembly. If the

code is still too slow, then the DSP developer should turn to hand-optimized assembly code. Figure 41 shows a comparison of the different programming techniques, with corresponding execution efficiency and development effort.

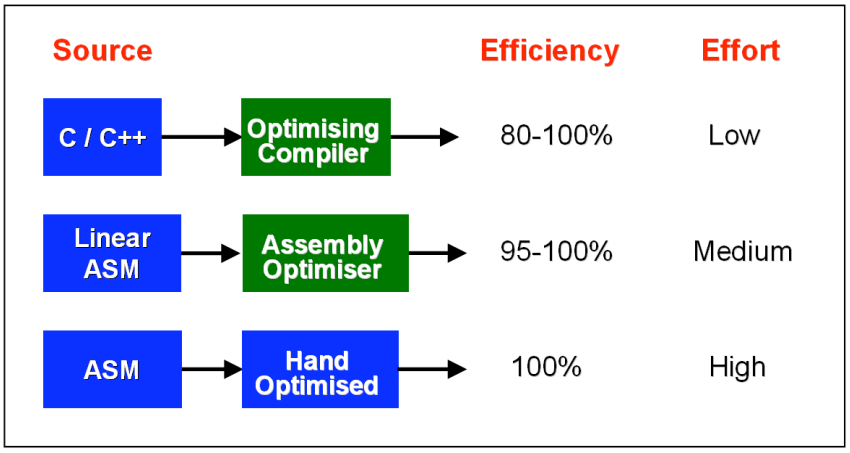


Fig. 41: Comparison of programming techniques with corresponding execution efficiency and estimated development effort. The picture is courtesy of TI [59].

8.2 Analysis tools

DSP code often follows the 20/80 rule, which states that 20% of the software in an application uses 80% of the processing time. As a consequence, the DSP developer should first concentrate efforts on determining where to optimize, i.e., on understanding where the execution cycles are mostly spent.

The best way to determine the parts of the code to optimize is to profile the application. Over the last ten years DSP development environments have considerably enlarged their offer of analysis tools. Some examples of TI's CCS analysis and tuning tools are:

a) Compiler consultant. It analyses the DSP code and provides recommendations on how to optimize its performance. This includes compiler optimization switches and programs, thus allowing a quick improvement in performance. Figure 42 shows how to enable the compiler consultant in CCS.

b) Cache tune. It provides a graphical visualization of memory reference patterns and memory accesses, thus allowing the identification of problem areas related for instance to memory access conflicts.

c) Code size tune. It profiles the application, collects data on individual functions and determines the best combinations of compiler options to optimize the trade-off between code size and execution speed.

d) Analysis ToolKit (ATK). It runs with DSP simulators only and allows one to analyse the DSP code robustness and efficiency. The reader can find more information on the ATK setup and use in Refs. [60, 61].

The DSP developer should not only know when to optimize, as described previously: he/she should also know when to stop. In fact, there is a law of diminishing returns in the code analysis and optimization process. It is thus important to take advantage of the improvements that come with relatively little effort, and leave as a last resort those that are difficult to implement and provide low-yield.

Finally, it is strongly recommended to make only one optimization change at the same time; this will allow the developer to exactly map the optimization to its result.

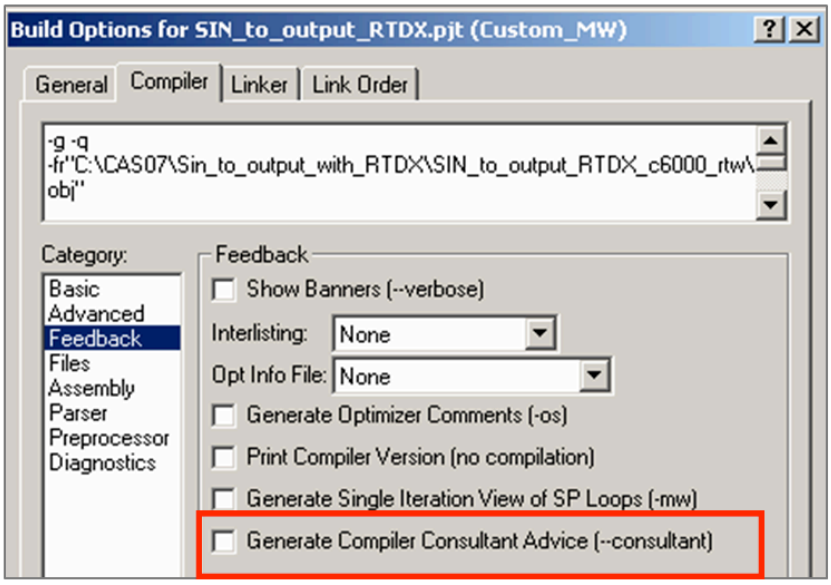


Fig. 42: How the 'Compiler Consultant Advice' can be enabled in TI's CCS Development Environment

8.3 Programming optimization guidelines

This Section includes some general programming guidelines for writing efficient code; these guidelines are applicable to the vast majority of DSP compilers. DSP developers should, however, refer to the manuals of the development tools they are using for more precise information on how to write efficient code. The reference manual for TI TMS320C6xxx DSP can be found in Ref. [62].

Finally, it is strongly recommended to make only one optimization change at the same time; this will allow the developer to exactly map the optimization to its result.

– Guideline 1: Use the DMA when possible and allocate data in memory wisely

DMA controllers (see Sub-section 3.2.3) must be used whenever possible so as to free the DSP core for other tasks. The linker (see Sub-section 6.4.3) should be used for allocating data in memory so as to guarantee an efficient memory access. Additionally, DSP developers should avoid placing arrays at the beginning or at the very end of memory blocks, as this creates problems for software pipelining. Software pipelining is a technique that optimizes tight loops by fetching a data set while the DSP is processing the previous one. However, the last iteration of a loop would attempt to fetch data outside the memory space, in case an array is placed on the memory edge. Compilers must then execute the last iteration in a slower way ('loop epilogue') to prevent this address error from happening. Some compilers, such as the ADI Blackfin one, make available compiler options to specify that it is safe to load additional elements at the end of the array.

– Guideline 2: Choose variable data types carefully

DSP developers should know the internal architecture of the DSP they are working on, so as to be able to use native data type DSPs as opposed to emulated ones, whenever possible. In fact, operations on native data types are implemented by hardware, hence are typically very fast. On the contrary, operations on emulated data types are carried out by software functions, hence are slower and use more resources. An example of emulated data type is the double floating point format on ADI's TigerSHARC floating point DSPs. Another example is the floating point format on ADI's

Blackfin family of fixed-point processors [63]. In these DSPs the floating point format is implemented by software functions that use fixed-point multiply and ALU logic. In this last case a faster version of the same functions is available with non-IEEE-compliant data formats, i.e., formats implementing a 'relaxed' IEEE version so as to reduce the computational complexity. Table 12 shows a, execution times comparison of IEEE-compliant and non-IEEE-compliant functions in ADI's Blackfin BF533.

Table 12: Execution time of IEEE-compliant vs. non-IEEE-compliant library functions for ADI's Blackfin BF533

operation	fast-ft [cycles]	IEEE-ft [cycles]	ratio
multiply	93	241	0.4
add	127	264	0.5
subtract	161	329	0.5
divide	256	945	0.3
pow	8158	17037	0.5

– Guideline 3: Functions and function calls

Functions such as *max()*, *min()* and *abs()*are often single-cycle instruction and should be used whenever possible instead of manually coding them. Figure 43 shows on the right-hand side the *max()* function and on the left-hand side a manual implementation of the same function. The advantage in terms of code efficiency of using a single-cycle *max()* function is evident. Often more complex functions such as FFT, IIR, or FIR filters are available in vendor-provided libraries. The reader is strongly encouraged to use them, as their optimization is carried out at algorithm level.

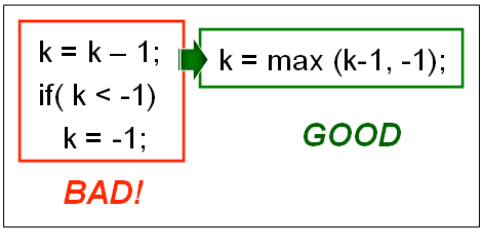


Fig. 43: Example of good and bad programming techniques

As few parameters as possible should be passed to a function. In fact, parameters are typically passed to functions by using registers. However, the stack is used when no more registers are available, thus slowing down the code execution considerably.

– Guideline 4: Avoid data aliasing

Aliasing occurs when multiple variables point to the same data. For example, two buffers overlap, two pointers point to the same software object or global variables used in a loop. This situation can disrupt optimization, as the compiler will analyse the code to determine when aliasing could occur. If it cannot work out if two or more pointers point to independent addresses or not, the compiler will typically behave conservatively, hence avoid optimization so as to preserve the program correctness.

– Guideline 5: Write loops code carefully

Loops are found very often in DSP algorithms, hence their coding can strongly influence the program execution performance. Function calls and control statements should be avoided inside a loop, so as to prevent pipeline flushes (see Sub-section 3.3.2). Figure 44 shows an example of good

and bad programming techniques referred to control statements inside a *for()* loop: by moving the conditional expression *if...else* outside the loop, as shown in the right-hand side code snippet, one can reduce the number of times the conditional expression is executed.

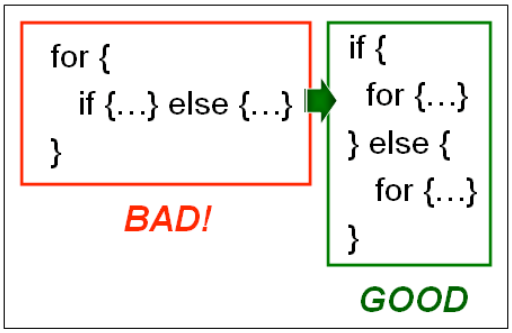


Fig. 44: Example of good and bad programming techniques

Loop code should be kept small, so as to fit entirely into the DSP cache memory and to allow a local repeat optimization. In case of many nested loops, the reader should be aware that compilers typically focus their optimization efforts on the inner loop. As a consequence, pulling operations from the outer to the inner loop can improve performance. Finally, it is recommended to use *int* or *unsigned int* data types for loop counters instead of the larger-sized data type *long*.

– Guideline 6: Be aware of time-consuming operations

There are operations, such as the division, that do not have hardware support for a single-cycle implementation. They are instead implemented by functions implementing iterative approximations algorithms, such as the Newton–Raphson. The DSP developer should be aware of that and try to avoid them when possible. For example, the division by a power-of-two operation can be converted to the easier right shift on unsigned variables. DSP manufacturers often provide indications on techniques to implement the division instruction more efficiently [64].

Other operations are available from library functions. Examples are *sine*, *cosine* and *atan* functions, very often needed in the accelerator sector for the implementation of rotation matrixes and for rectangular to polar coordinates conversion. If needed, custom implementations can be developed to obtain a favourable ratio between precision and execution time. Table 13 shows the comparison of different implementations of the same functions; in particular, the second column shows a custom implementation used in CERN's LEIR accelerator. In this implementation, the *sine, cosine* and *atan* calculation algorithm has been implemented by a polynomial expansion of the seventh order instead of the usual Taylor series expansion.

Table 13: Execution times vs. different implementations of the same functions

	Execution time [µs]		
Function	**CERN single-precision implementation**	**VisualDSP++ single-precision implementation**	**VisualDSP++ double-precision implementation**
cosine	0.25	0.59	5.5
sine	(for a sine/cosine couple)	0.59	5.3
atan	0.4125	1.4	5.6

– Guideline 7: Be aware that DSP software heavily influences power optimization

DSP software can have a significant impact on power consumption: a software-efficient in terms of the required processor cycles to carry out a task is often also energy efficient. Software should be written so as to minimize the number of accesses to off-chip memory; in fact, the power required to access off-chip memory is usually much higher than that used for accessing on-chip memory. Power consumption can be further optimized in DSPs that support selective disabling of unused functional blocks (e.g., on-chip memories, peripherals, clocks, etc.). These 'power down modes' are available in ADI DSPs (such as Blackfin) as well as in TI DSPs (such as the TMS320C6xxx family [35]). Making a good use of these modes and features can be difficult; however, APIs and specific software modules are available to help. An example is TI's DSP/BIOS Power Manager (PWRM) module [65], providing a kernel-level API that interfaces directly to the DSP hardware by writing and reading configuration registers. Figure 45 shows how this module is integrated in a generic application architecture for DSPs belonging to TI's TMS320C55x family.

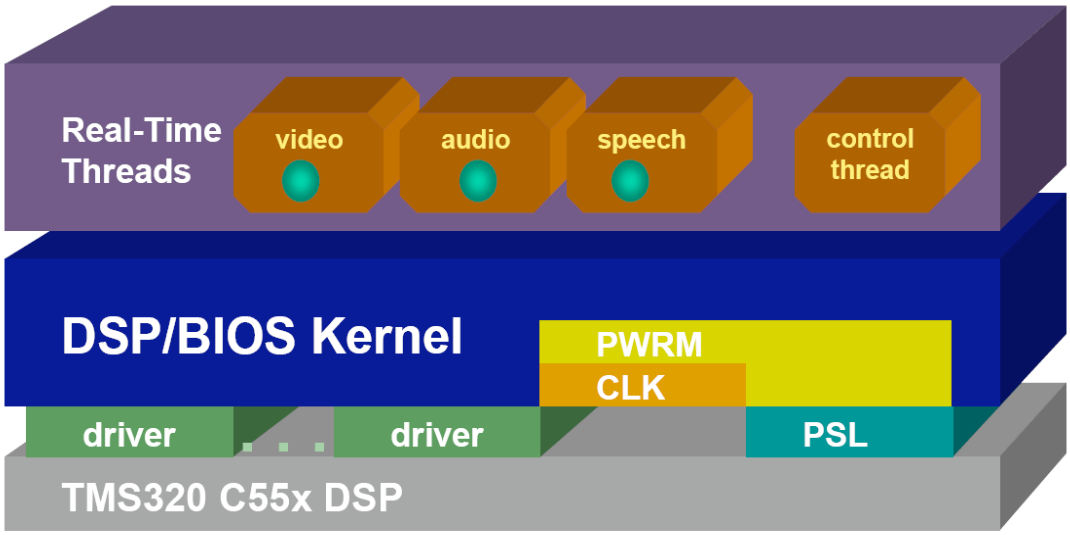


Fig. 45: TI's DSP/BIOS Power Manager (PWRM) module in a general system architecture. Picture courtesy of Texas Instruments [65].

9 Real-time design flow: system design

This section deals with some aspects of digital systems design, particularly with software and hardware architectures. Here the assumption is that the system to be designed is based upon one or more DSPs. The reader should, however, be aware that in the accelerator sector there are currently three main real-time digital signal processing actors: DSPs, FPGAs and front-end computers. The front-end computers are typically implemented by embedded General Purpose Processors (GPPs) running a RTOS. Nowadays, the increase in clock speed allows GPPs to carry out real-time data processing and slow control actions; in addition, there is a tendency to integrate DSP hardware features and specialized instructions into GPPs, yielding GPP hybrids. One example of such processors is given in Fig. 46, showing the PowerPC with Motorola's Altivec extension. The Altivec 128-bit SIMD unit adds up to 16 operations per clock cycle, in parallel to the Integer and Floating Point units, and 162 instructions to the existing RISC architecture.

Fundamental choices to make when designing a new digital system are which digital signal processing actors should be used and how tasks should be shared between them. This choice requires detailed and up-to-date knowledge of the different possibilities.

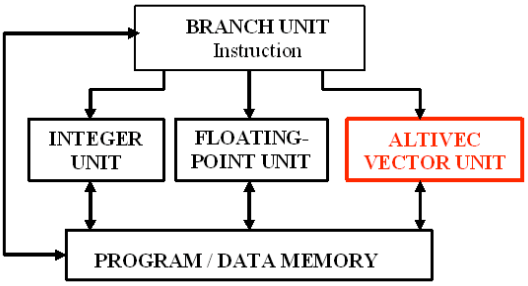


Fig. 46: Altivec technology: SIMD expansion to Motorola PowerPC (G4 family)

In industry the choice of the DSP to use is often based on the '4P' law: Performance, Power consumption, Price and Peripherals. In the accelerator sector, the power consumption factor is typically negligible. Other factors are instead decisive, such as standardization in the laboratory, synergies with existing systems, and possibilities of evolution to cover different machines. Last but not least, one should consider the existing know-how in terms of tools and of hardware, which can be directly translated to a shorter development time.

In this section three design aspects are considered and briefly discussed, namely:

a) DSP choice in Sections 9.1 and 9.2.

b) System architecture in Sections 9.3 to 9.6.

c) DSP code design in Sections 9.7 and 9.8.

9.1 DSP choice: fixed vs. floating-point DSPs

The reader can find a basic description of fixed- and floating-point number formats in Section 3.4.

Fixed-point formats can typically be implemented in hardware in a cheaper way, with better energy efficiency and less silicon than floating-point formats. Very often fixed-point DSPs support a clock faster than floating-point DSPs; as an example, TI fixed-point DSPs can currently be clocked up to 1.2 GHz, while TI floating-point DSPs are clocked up to 300 MHz.

Floating-point formats are easier to use since the DSP programmer can mostly avoid carrying out number scaling prior to each arithmetic operation. In addition, floating-point numbers provide a higher dynamic range, which can be essential when dealing with large data sets and with data sets whose range cannot be easily predicted. The reader should be aware that floating-point numbers are not equispaced, i.e., the gap between adjacent numbers depends on their magnitude: large numbers have large gaps between them, and small numbers have small gaps. As an example, the gap between adjacent numbers is higher than 10 for numbers of the order of $2 \cdot 10^8$. Additionally, the error due to truncation and rounding during the floating-point number scaling inside the DSP depends on the number magnitude, too. This introduces a noise floor modulation that can be detrimental for high-quality audio signal processing. For this reason, high-quality audio has been traditionally implemented by using fixed-point numbers. However, a migration of high-fidelity audio from fixed- to floating-point implementation is currently taking place, so as to benefit from the greater accuracy provided by floating point numbers.

The choice between fixed- and floating-point DSP is not always easy and depends on factors such as power consumption, price, and application type. As an example, military radars need floating-point implementations as they rely in finding the maximal absolute value of the cross-correlation between the sent signal and the received echo. This is expressed as the integral of a function against an exponential; the integral can be calculated by using FFT techniques that benefit from the floating point dynamic range and resolution. For radar systems, the power consumption is not a major issue. The

floating-point DSP additional cost is not an issue either, as the processor represents only a fraction of the global system cost. Another example is the mobile TV. The core of this application is the decoder, which can be MPEG-2, MPEG-4 or JPEG-2000. The decoding algorithms are designed to be performed in fixed-point; the greater precision of floating-point numbers is not useful as the algorithms are in general bit-exact.

It should be underlined that many digital signal processing algorithms are often specified and designed with floating-point numbers, but are subsequently implemented in fixed-point architectures so as to satisfy cost and power efficiency requirements. This requires an algorithm conversion from floating-point to fixed-point and different methodologies are available [66].

Finally, as mentioned in Section 8.3, some fixed-point DSPs make available floating-point numbers and operations by emulating them in software (hence they are slower than in a native floating-point DSP). An example is ADI's Blackfin [63].

The fact that floating-point numbers are not equispaced has already been mentioned. The reader might be interested in looking at some consequences of this with an example from the LHC beam control implementation. Figure 47 shows a zoom onto the beam loops part of the LHC beam control. The 'Low-level Loops Processor' is a board including a TigerSHARC DSP and an FPGA. The FPGA carries out some simple pre-processing and data interfacing, while the DSP implements the low-level loops. In particular, the DSP calculates the frequency to be sent to the cavities from the beam phase, radial position, synchrotron frequency, and programmed frequency; these calculations are carried out in floating-point format. The frequency to be sent to the cavities, referred to as *F_out* in Fig. 47, must be expressed as an unsigned, 16-bit integer. The desired frequency range to represent is 10 kHz, hence the needed resolution is 0.15 Hz. The LHC cavities work at a frequency of about 400.78 MHz but the spacing of a single-precision, floating-point number with magnitude of approximately $400 \cdot 10^6$ is higher than one. To avoid the use of slower, double-precision, floating-point format, the beam loop calculations are carried out as offset from 400.7819 MHz.

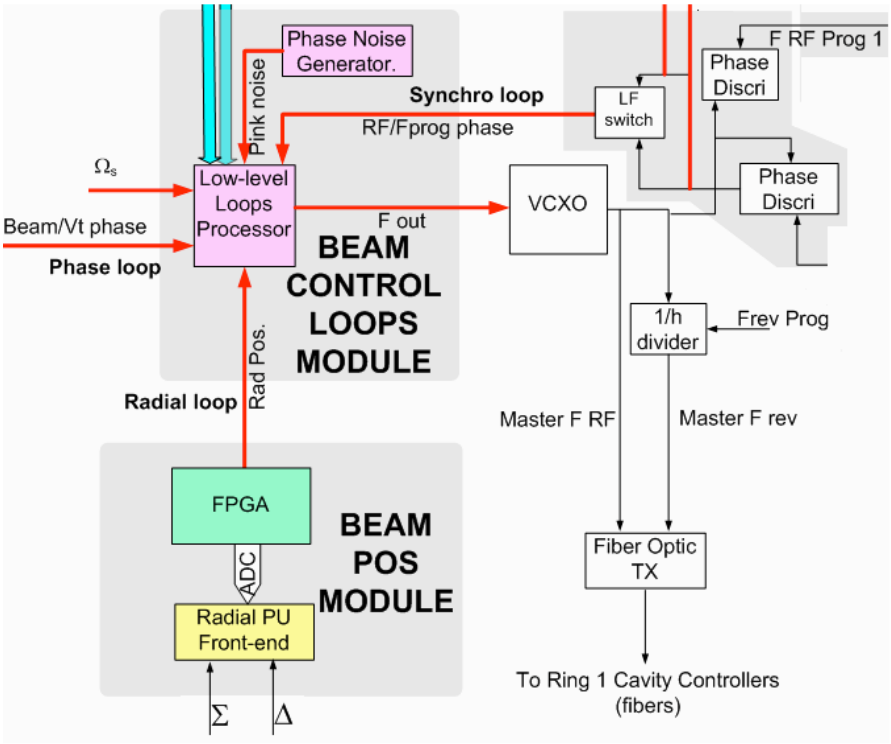


Fig. 47: LHC beam control – zoom onto the beam loops part

9.2 DSP choice: benchmarking

Benchmarking a DSP means evaluating it on a number of different metrics. Table 14 gives an example of some common metrics and corresponding units.

Table 14: Examples of DSP performance metric sets and corresponding units

Metric	Unit
Maximum clock frequency	MHz
Execution speed	Millions of Instructions Per Second (MIPS) Millions of Operations Per Second (MOPS) Number of Multiply-and-Accumulate operations per second
Memory bandwidth	Mbytes/s
Memory latency	Number of clock cycles
Power consumption	W or W/MIPS

Good benchmarks are important for comparing DSPs and allow critical business or technical decisions to be made. It should be underlined that benchmarks can be misleading, thus should be considered in a critical way. As an example, the maximum clock frequency of a DSP can be different from the instruction rates; hence this parameter might not be indicative of the real DSP processing power. Another example is the execution speed measured in MIPS: this metric is easy to measure but it is often too simple to provide useful information about how a processor would perform in a real application. VLIW architectures issue and execute multiple instructions per instruction cycle. These processors usually use simpler instructions that perform less work than the instructions typical of conventional DSPs. As a consequence, MIPS comparison between VLIW-based DSP and conventional ones is misleading.

More complex benchmarks are available; examples are the execution of application tasks (typically called kernel functions) such as IIR filters, FIR filters, or FFTs. Kernel function benchmarking is typically more reliable and is available from DSP manufactures as well as from independent companies.

It is difficult to provide general guidelines to measure the efficacy of DSP benchmarks for DSP selection. Two general rules should be followed: first, the benchmark should perform the type of work the DSP will be expected to carry out in the targeted application. Second, the benchmark should implement the work in a way similar to what will be used in the targeted application.

9.3 System architecture: multiprocessor architectures

Multiprocessor architectures are those where two or more processors interact in real-time to carry out a task. Right from their early days, many DSP families have been designed to be compatible with multiprocessing operation; an example is the TI TMS320C40 family. Multiprocessing architectures are particularly suited for applications with a high degree of parallelism, such as voice processing. In fact, processing ten voice channels can be carried out by implementing a one-voice channel, then repeating the process ten times in parallel. Applications requiring multiprocessing computing to support processing of greater data flow include high-end audio treatment, 3D graphics acceleration, and wireless communication infrastructure, just to mention a few of them.

There is another reason to move to multiprocessing systems. For many years developers have been taking advantage of the steady progress in DSP performance. New and faster processors would be available, allowing more powerful applications to be implemented sometimes only for the price of porting existing code to the new DSP. This favourable situation was driven by the steady progress of the semiconductor industry that managed to pack more transistors into smaller packages and at higher clock frequencies. The increased performance was enabled by architectural innovations, such as VLIW, as well as added resources, such as on-chip memories. In recent years, however, progress in single-chip performance has been slowing down. The semiconductor industry has turned to parallelism to increase performance. This is true not only for the DSP sector, but in general for business computing. One example is the Intel Core Duo processors, including two execution cores in a single processor, now the established platform for personal computers and laptops.

Finally, the reader should be aware that development environments have evolved to provide support for debugging multiple processor cores connected in the same JTAG path [67]. An example is TI's Parallel Debug Manager [68], which is integrated within the Code Composer Studio IDE.

Of the many possible multiprocessing forms, the multi-DSP and multi-core approaches are considered and discussed in Sub-sections 9.3.1 and 9.3.2, respectively. Examples of embedded multi-processors and different approaches can be found in Ref. [69].

9.3.1 Multi-DSP architecture

Many separate DSP chips can co-operate to carry out a task providing an increased system performance. One advantage of this approach is the scalability, i.e., the ability to tune the system performance and cost to the required functionality and processing performance by varying the number of DSP chips used.

The reader should, however, be aware that multi-DSP designs involve different constraints than single-processing systems. Three key aspects must be taken into account.

a) Tasks must be partitioned between processors. As an example, a single processor can handle a task from start to end; as an alternative, a processor can perform only a portion of the task, then pass the intermediate results to another processor.

b) Resources such as memory and bus access must be shared between processors so as to avoid bottlenecks. As an example, additional memory may be added to store intermediate results. Organizing memory into segments or banks allows simultaneous memory accesses without contentions if different banks are accessed.

c) A robust and fast inter-DSP communication means must be established. If the communication is too complex or takes too much time, the advantage of a multiprocessing can be lost.

Two examples of multi-DSP architectures based on ADI DSPs are shown in Fig. 48. The reader can find more detailed information in Refs. [70] and [71].

On the left-hand side (plot a) the point-to-point architecture is depicted, based upon ADI linkport interconnect cable standard [27]. Point-to-point interconnect provides a direct connection between processor elements. This is particularly useful when large blocks of intermediate results must be passed between two DSPs without involving the others. Read/write transactions to external memory are saved by passing data directly between two DSPs, thus allowing the use of slower memory devices. Additionally, the point-to-point interconnect can be used to scale a design: additional links can be added to have more DSPs interacting. This can be done either directly or by bridging across several links.

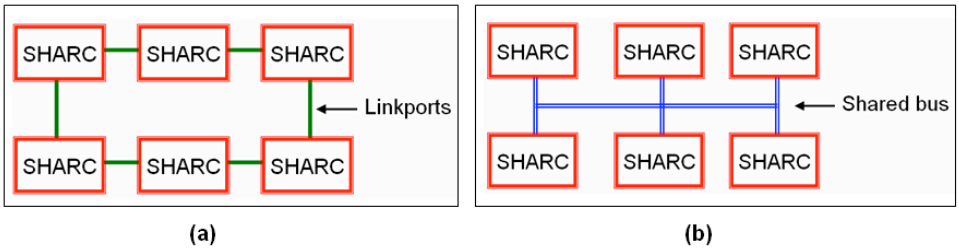


Fig. 48: Examples of multi-DSP configurations. (a) point-to-point, linkport-based and (b) cluster bus

On the right-hand side (plot b) the cluster bus architecture is depicted. A cluster bus maps internal memory resources, such as registers and processor memory addresses, directly onto the bus. This allows DSP code developers to exchange data between DSPs using addresses as if each processor possessed the memory for storing the data. Memory arbitration is managed by the bus master; this avoids the need for complex memory or data sharing schemes managed by software or by RTOS. The map includes also a common broadcast space for messages that need to reach all DSPs. As an example, Fig. 49 shows the TigerSHARC global memory map. The multiprocessing space maps the internal memory space of each TigerSHARC processor in the cluster into any other TigerSHARC processor. Each TigerSHARC processor in the cluster is identified by its ID; valid processor ID values are 0 to 7.

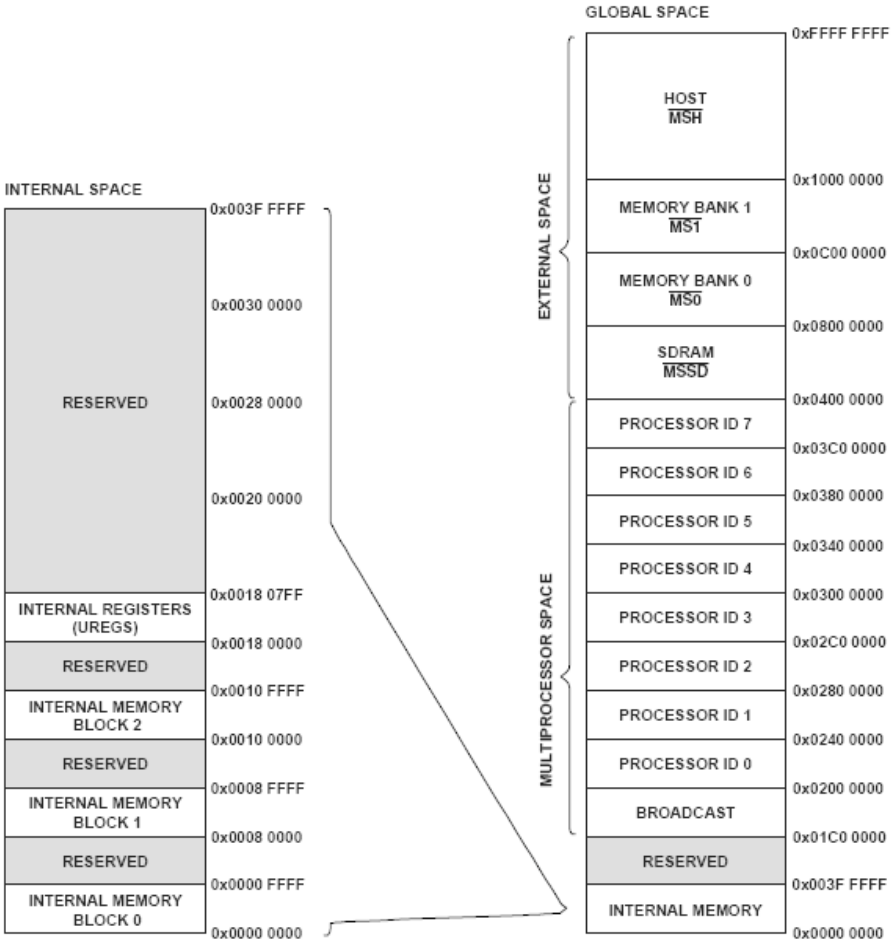


Fig. 49: ADI TigerSHARC TS101 global memory map. Picture courtesy of Analog Devices [72].

The reader should be aware that the two above-mentioned architectures, namely point-to-point and cluster bus, are not mutually exclusive; on the contrary, they can both be used in the same application as complementary solutions.

9.3.2 Multi-core architecture

In a multi-core architecture, multiple cores are integrated into the same chip. This provides a considerable increase of the performance per chip, even if the performance per core only increases slowly. Additionally, the power efficiency of multi-core implementations is much better than in traditional single-core implementations. This approach is a convenient alternative to DSP farms.

As the performance required by DSP systems keeps increasing, it is nowadays essential for DSP developers to devise a processing extension strategy. Multi-core architectures can provide it, in that the DSP performance is boosted without switching to a different core architecture. This has the advantage that applications can be based upon multiple instances of an already-proven core, rather than be adapted to new architectures.

DSP multi-core architectures have been commercialized only recently; however, the DSP market has relied for many years on co-processor technology (also called on-chip accelerators) to boost performance. Figure 50 shows the evolution of DSP architecture. From the initial single-core architecture (a), the single-core plus co-processor architecture soon emerged. The co-processor often runs at the same frequency as the DSP, therefore 'doubling' the performance for the targeted application. Co-processor examples are Turbo and Viterbi decoders for communication applications. Example of decoder coprocessors for TI's TMS320C64x can be found in Refs. [73] and [74]. Finally, over the last few years the multi-core architecture shown in plot (c) has emerged, which still includes co-processors.

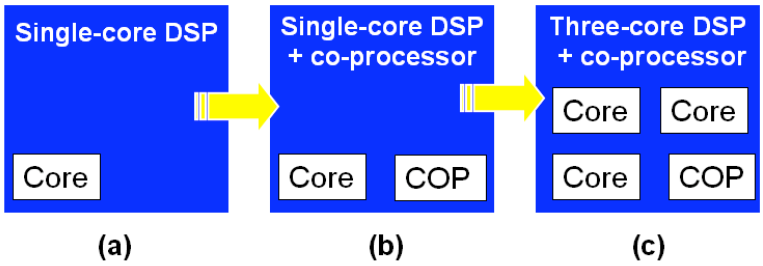


Fig. 50: Multi-core and co-processor DSP architectures evolution. Single-core DSP (a), single-core DSP plus coprocessor (b) and multi-core DSP plus coprocessor (c).

Multi-core architectures are available in two different flavours, namely Symmetric Multi-Processing (SMP) and Asymmetric Multi-Processing (AMP). SMP architectures include two or more processors which are similar (or identical), connected thorough a high-speed path and sharing some peripherals as well as memory space. AMP architectures combine two different processors, typically a microcontroller and a DSP, into a hybrid architecture.

It is possible to use a multi-core device in different ways. The different cores can operate independently or they can cooperate for task completion. An efficient inter-core communication may be needed in both cases, but it is particularly important when two or more cores work together to complete a task. As for the multi-DSP case discussed in Sub-section 9.3.1, it is important to decide how to share resources to avoid bottlenecks and deadlocks, and to ensure that one core does not corrupt the operation of another core. The resources must be partitioned not only at board level, like in the single-core case, but at device level, too, thus adding increase complexity. Figure 51 shows an example of multi-core bus and memory hierarchy architecture. L1 memories are typically dedicated to their own core as non-partitioned between cores, as it may be inefficient to access them from other

cores. The L2 memory is an internal memory shared between the different cores, as opposed to the single-core case where the L2 memory can be either internal or external. The multi-core architecture must make sure that each core can access the L2 memory and the arbitration must be such that cores are not locked out from accessing this resource.

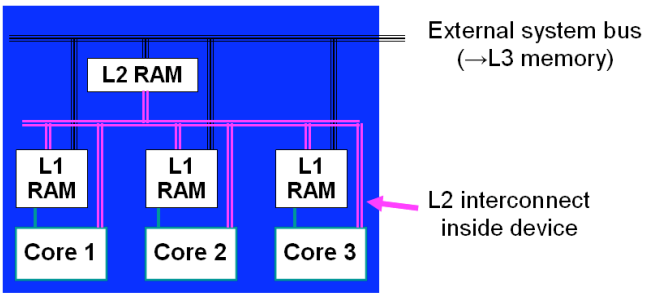


Fig. 51: Multi-core bus and memory hierarchy example

Figure 52 shows the TMS320C5421 DSP as an example of a multi-core, SMP DSP. The TMS320C5421 DSP is composed of two C54x DSP cores and is targeted at carrier-class voice and video end equipment. The cores are 16-bit fixed-point and the chip is provided with an integrated VITERBI accelerator. Four internal buses and dual address generators enable multiple program and data fetches and reduce memory bottlenecks.

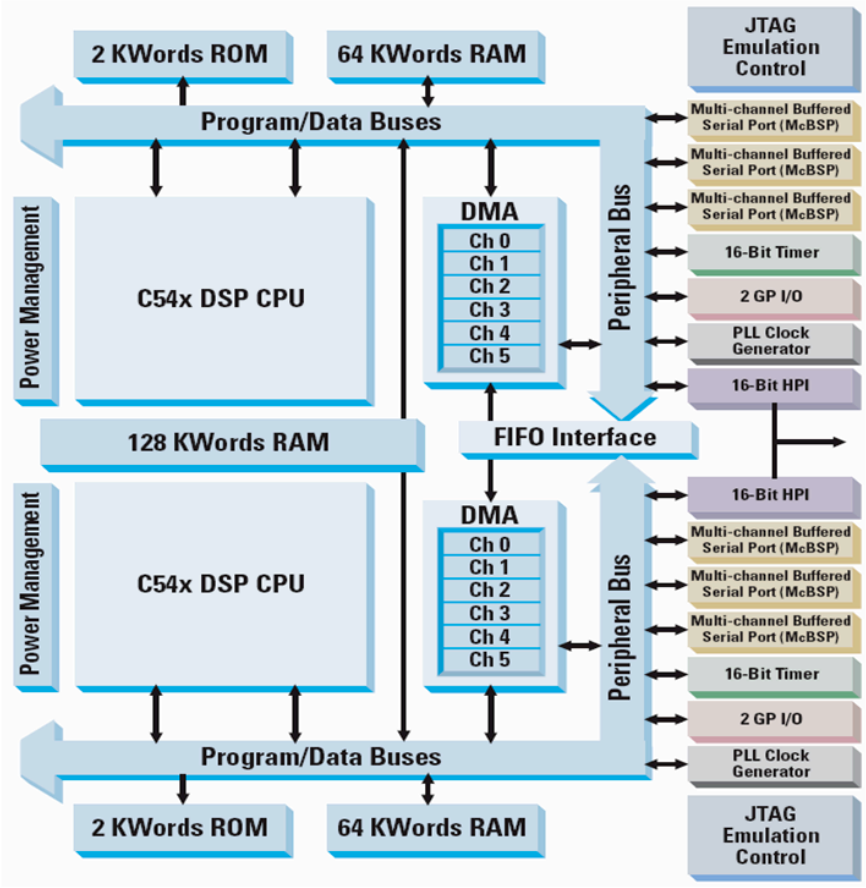


Fig. 52 TMS320C5421 multi-core DSP as an SMP example. Picture courtesy of Texas Instruments, DSP selection guide 2007, p. 48.

The programming of multi-core system is generally more complex than in the single-core case. In particular, the reader should be aware that multi-core code must follow the re-entrance rules, to make sure that one core's processing does not corrupt the data used by another core's processing. This approach is followed by single-core processors, too, when implementing multi-tasking operations.

An example of advantages and challenges a developer is dealing with when moving an audio application from single- to double-core architecture is given in Ref. [75].

9.4 System architecture: radiation effects

Single-Event Upset (SEU) events are alterations in the behaviour of electronic circuits induced by radiation. These alterations can be transient disruptions, such as changes of logic states, or permanent IC alterations. The reader is referred to Ref. [76] for more information on the subject.

Techniques to mitigate these effects in ICs can be carried out at different levels, namely:

a) At device level, for instance by adding extra-doping layers to limit the substrate charge collection.

b) At circuit level, for instance by adding decoupling resistors, diodes, or transistors in the SRAM hardening.

c) At system level, with Error Detection And Correction (EDAC) circuitry or with algorithm-based fault tolerance [77]. An example of the latter approach is the Triple Module Redundancy (TMR) algorithm or the newer Weighted Checksum Code (WCC). The reader should, however, be aware that there are limitations to what these algorithms can achieve. For instance, the WCC method applied to floating-point systems may fail, as roundoff errors may not be distinguished from functional errors caused by radiation.

Neither ADI nor TI currently provide any radiation-hard DSP. Third-party companies have developed and marketed radiation-hard versions of ADI and TI DSPs. An example is Space Micro Inc., based in San Diego, California. This company devised the Proton 200k single-board computer based upon a TI C67xx DSP, fitted with EDAC circuitry and with a total dose tolerance higher than 100 krad.

The LHC power supply controllers [78, 79] are examples of mitigation techniques applied to DSP. They are based upon non-radiation-hard TI C32 DSPs and micro controllers. The memory is protected with EDAC circuitry and by avoiding the use of DSP internal memory, which cannot be protected. A watchdog system restarts the power supply controller in the event of a crash. Radiation tests [80] have been carried out to check that the devised protection strategy is sufficient for normal operation.

9.5 System architecture: interfaces

An essential step in the digital system design is to clearly define the interfaces between the different parts of the system. Figure 53 shows some typical building blocks that can be found in a digital system, namely DSP(s), FPGA(s), daughtercards, Master VME, machine timings, and signals.

The DSP system designer must define the interfaces between DSP(s) and the other building blocks. It is strongly recommended to avoid hard-coding in the DSP code the address of memory regions shared with other processing elements. On the contrary, the linker should be used to allocate appropriately the software structures in the DSP memory, as mentioned in Sub-section 6.4.3. Additionally, the DSP developer should created data access libraries, so as to obtain a modular hence more easily upgradeable approach.

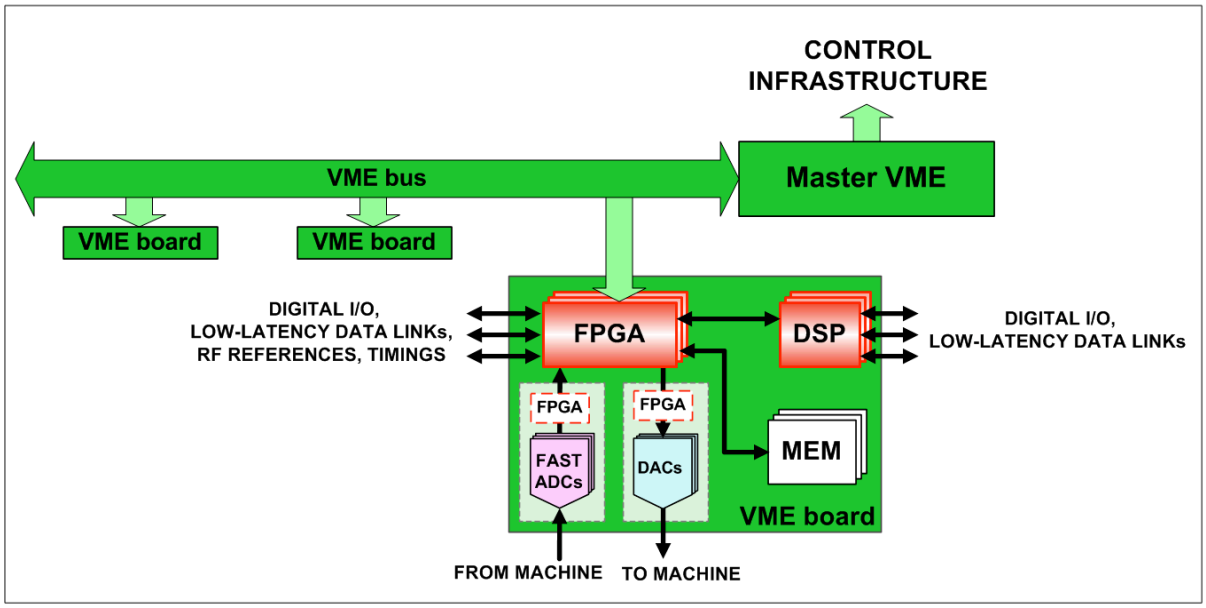


Fig. 53: Typical digital system building blocks and corresponding interfaces

9.6 System architecture: general recommendations

Basically all DSP chips present some anomalies on their expected behaviour. This is especially true for the first release of DSP chips, as discovered anomalies are typically solved on later releases. A list of all anomalies for a certain DSP release, which includes also workarounds when possible, is normally available on the manufacturer's website. The reader is strongly encouraged to look at those lists, so as to avoid being delayed by already-known problems.

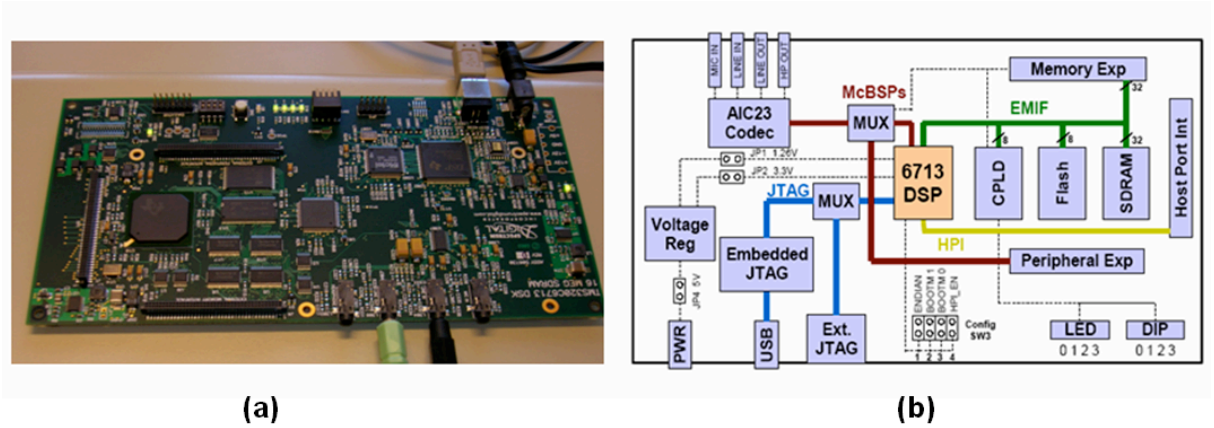


Fig. 54: TI C6713 DSK evaluation board – picture (a) and board layout (b)

A DSP system designer can gain useful software and hardware experience by using evaluation boards in the early stages of system design. Evaluation boards are typically provided by manufacturers for the most representative DSPs. They are relatively inexpensive and are typically fitted with ADCs and DACs; they come with the standard development environment and JTAG interface, too. The DSP designer can use them to solve technical uncertainties and sometimes can even modify them to quickly build a system prototype [81]. Figure 54 shows TI's C6713 DSK evaluation board (a) and corresponding board layout (b); this evaluation board was that used in the DSP laboratory companion of the lectures summarized in this paper.

9.7 DSP code design: interrupt-driven vs. RTOS-based systems

A fundamental choice that the DSP code developer must make is how to trigger the different DSP actions. The two main possibilities are via a RTOS or via interrupts.

An overview of RTOS is given in Section 6.3. RTOS can define different threads, each one performing a specific action, as well as the corresponding threads' priorities and triggers. RTOS-based systems have typically a clean design and many built-in checks. The disadvantage of using RTOS is a potentially slower response to external events (interrupts) and the use of DSP resources (such as some hardware timings and interrupts) for the internal RTOS functioning.

Interrupt-driven systems associate actions directly to interrupts. The resource use is therefore optimized. An example of interrupt-driven system is CERN's LEIR LLRF [42]. Figure 55 shows some of its software components: a background task triggered every millisecond carries out housekeeping actions, while a control task triggered every 12.5 μs implements the beam control actions. Driving a system through interrupts is very efficient with a limited number of interrupts. For a high number of interrupts, the system can become very complex and its behaviour not easily predictable.

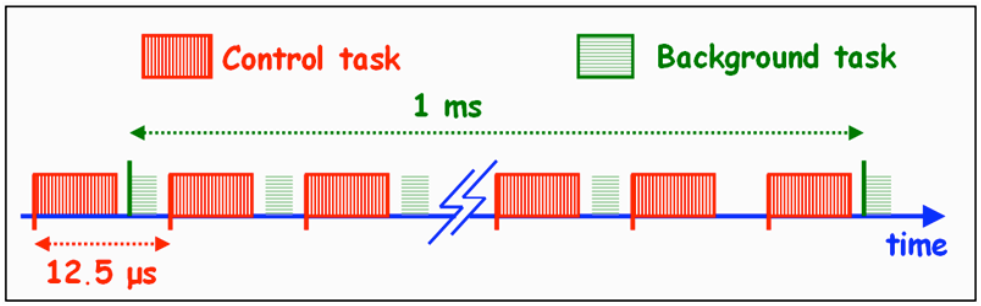


Fig. 55: Example of an interrupt-driven system. Control and background tasks are triggered by interrupts and are shown in red and green, respectively.

9.8 DSP code design: good practice

A vast amount of literature is available on code design good practice. Here just a few points are underlined, which are particularly relevant to embedded systems.

First, digital systems must not turn into tightly sealed black boxes. It is essential that designers embed many diagnostics buffers in the DSP code, so as to prevent this from happening. The diagnostics buffers could take many forms, such as post-mortem, circular or linear buffers. They might be user-configurable and must be visible from the application program. An example of a digital system including extensive diagnostics capabilities can be found in Ref. [42].

Second, every new DSP code release should be characterized by a version number, visible from the application level. The functionality and interface map corresponding to a certain version number should be clearly documented, so as to avoid painful misunderstandings between the many system layers. Source code control is essential for managing complex software development projects, as large projects require more than one DSP code developer working on many source files. Source code control tools make it possible to keep track of the changes made to individual source files and prevent files from being accessed by more than one person at a time. DSP software development environments can often support many source control providers. Code Composer Studio, for example, supports any source control provider that implements the Microsoft SCC Interface.

Finally, DSP developers should also add checks on the execution duration, to make sure the code does not overrun. This is particularly important for interrupt-driven systems (mentioned in Section 9.7), where one or more interrupts may be missed if the actions corresponding to an interrupt are not finished by the time the next interrupt occurs. As an example, the minimum and maximum

number of clock cycles needed for executing a piece of code can be constantly measured and monitored by the user at high level. All DSPs provide means to measure the number of clock cycles required to execute a certain amount of code; the number of clock cycles can then be easily converted into absolute time. Figure 56 shows a possible implementation on ADI SHARC DSPS of the execution duration of a code called 'critical action'. SHARC processors have a set of registers called *emuclk* and *emuclk2* which make up a 64-bit counter. This counter is unconditionally incremented during every instruction cycle on the DSP and is not affected by factors such as cache-misses or wait-states. Every time *emuclk* wraps to zero, *emuclk2* is incremented by one. By determining the difference in the *emuclk* value between before and after the critical action, the DSP developer can determine the number of clock cycles — hence the time — to execute the code.

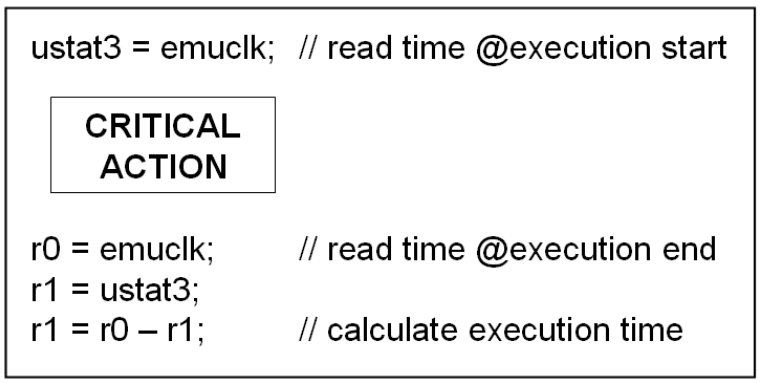


Fig. 56: Execution duration measurement with *emuclk* registers in the ADI SHARC DSP

10 Real-time design flow: system integration

The system integration is one of the final parts in the system development process. This phase is extremely important as it can determine the success or the failure of a whole system. In fact, a system which is well integrated can become operational, while a system only partially integrated will often remain a 'machine development' tool, easily forgotten.

During the system integration phase, the system is commissioned with respect to data exchange with the control infrastructure and the application program(s). Two or more groups, such as Instrumentation, Controls and Operation, can be involved in this effort, depending on the laboratory's organization. As a consequence, a coordination and specification work is required.

Good system integration practices will depend on the laboratory's organization as well as on the system architecture. There are, however, some guidelines that can be applied to most cases.

– Guideline 1: Work in parallel

All software layers needed in a system should be planned in parallel. Waiting until the low-level part is completed before starting with the specification and/or with the development of the other layers may result in unacceptable delays.

– Guideline 2: About interfaces

Section 9.5 summarized the many interfaces that can exist in a system. For a successful system integration it is essential that the interfaces are specified clearly, are agreed upon with all different parties and are fully documented. Recipes on how to set up different software components of the system or on how to interact with them can be really useful and speed up considerably system development as well as debugging. It is recommended that all documents be kept updated and stored on servers accessible by all parties involved. Remember: good fences make good neighbours!

– Guideline 3: Always include checks on the DSP inputs validity

The validity of all control inputs to the DSP should be checked. Alarms or warnings should be raised if a control value falls outside the allowed range. This mechanism will help the system integration part and could even prevent serious malfunctioning from happening.

– Guideline 4: Add spare parameters

It is strongly recommended to map spare parameters between the DSP and application program; they should have different formats for maximum flexibility. These spare parameters allow adding debugging features or making some small update without modifications to the intermediate software layers.

– Guideline 5: Code release and validation

The source code (and if possible the corresponding executable, too) should be saved together with a description of its features and implemented interfaces. This will allow going back to previous working releases in case of problems. Procedure and data sets should also be defined for code validation.

11 Summary and conclusions

This paper aimed at providing an overview of DSP fundamentals and DSP-based system design. The DSP hardware and software evolution was discussed in Sections 1 and 2, together with typical DSP applications to the accelerator sector. Section 3 showed the main features of DSP core architectures and Section 4 gave an overview of DSP peripherals. The real-time design flow was introduced in Section 5 and its steps were discussed in detail in Section 6 (software development), Section 7 (debugging), Section 8 (analysis and optimization), Section 9 (system design) and Section 10 (system integration).

Existing chip examples were often given and referenced to technical manuals or application notes. Examples of DSP use in existing accelerator systems were also given whenever possible.

The DSP field is of course very large and more information, as well as hands-on practice, is required to become proficient in it. However, the author hopes that this document and the references herein can be useful starting points for anyone wishing to work with DSPs.

References

[1] T. Shea, Types of Accelerators and Specific Needs, these CAS proceedings.

[2] M. E. Angoletta, Digital Signal Processing In Beam Instrumentation: Latest Trends And Typical Applications, DIPAC'03, Mainz, Germany, 2003.

[3] M. E. Angoletta, Digital Low-Level RF, EPAC'06, Edinburgh, Scotland, 2006.

[4] J. Eyre and J. Bier, The Evolution of DSP Processors, *IEEE Signal Proc. Mag.*, vol. 17, Issue 2, March 2000, pp. 44–51.

[5] J. Glossner et al., Trends In Compilable DSP Architecture, Proceedings of IEEE Workshop on Signal Processing Systems (SiPS) 2000, November 2000, Lafayette, LA, USA, pp. 181–199, ISBN 0-7803-6488-0.

[6] R. Restle and A. Cron, TMS320Cc30-IEEE Floating-Point Format Converter, Texas Instruments Application Report SPRA400, 1997.

[7] E.A. Lee, Programmable DSP Architectures: Part I, *IEEE ASSP Mag.*, October 1988, pp. 4–19.

[8] E.A. Lee, Programmable DSP Architectures: Part II, *IEEE ASSP Mag*., January 1989, pp. 4–14.

[9] J. Eyre, The Digital Signal Processor Derby, *IEEE Spectrum*, June 2001, pp. 62–68.

[10] L. Geppert, High-Flying DSP Architectures, *IEEE Spectrum*, Nov. 1998, pp. 53–56.

[11] P. Lapsley, J. Bier, A. Shoham and E. A. Lee, DSP Processor Fundamentals: Architectures and Features, IEEE Press, ISBN 0-7803-3405-1, 1997.

[12] TMS320C621x/C671x DSP Two-Level Internal Memory Reference Guide, Texas Instruments Literature Number SPRU609A, November 2003.

[13] M. Anderson, Advanced Processor Features And Why You Should Care, Part 1 And 2, talks ESC-404 and ESC-424, Embedded Systems Conference, Silicon Valley 2006.

[14] Analog Devices Team, The Memory Inside: TigerSHARC Swallows Its DRAM, Special Feature SHARC Bites Back, COTS Journal, December 2003.

[15] S. Srinivasan, V. Cuppu and B. Jacob, Transparent Data-Memory Organisations For Digital Signal Processors, Proceedings of CASES'01, International Conference on Compilers, Architecture and Synthesis for Embedded Systems, Caches and Memory Systems Session, Atlanta, Georgia, USA, 2001, pp. 44–48.

[16] TMS320C620x/C670x DSP Program and Data Memory Controller/Direct Memory Access (DMA) Controller - Reference Guide, Texas Instruments Literature Number SPRU234, July 2003.

[17] D. Talla, L.K. John, V. Lapinskii and B.L. Evans, Evaluating Signal Processing And Multimedia Applications On SIMD, VLIW And Superscalar Architectures, Proceedings of the International Conference on Computer Design, September 2000, Austin TX, USA, ISBN 0-7695-0801-4.

[18] J. A. Fisher, P. Farabosci and C. Young, A VLIW Approach To Architecture, Compilers And Tools, Morgan Kaufmann Publisher, December 2004, ISBN-13 978-1558607668.

[19] Extended-Precision Fixed-Point Arithmetic On The Blackfin Processor Platform, Analog Devices Engineer-to-Engineer Note EE-186, May 2003.

[20] IEEE Standard For Radix-Independent Floating-Point Arithmetic, ANSI/IEEE Std 854–1987.

[21] D. Goldber, What Every Computer Scientist Should Know About Floating-Point Arithmetic, *Comput. Surv*., March 1991.

[22] ADSP-21160 SHARC DSP – Hardware Reference, Revision 3.0, November 2003, Analog Devices Part Number 82-001966-01.

[23] TMS320C6713, TMS320C6713B Floating-Point Digital Signal Processors, Texas Instruments Manual SPRS186I, December 2001, Revised May 2004.

[24] ADSP-BF533 Blackfin Processor – Hardware Reference, Revision 3.1, June 2005, Analog Devices Part Number 82-002005-01.

[25] TMS320C6000 DSP – Multichannel Buffered Serial Port (McBSP) – Reference Guide, Texas Instruments Literature Number SPRU580C, May 2004.

[26] TMS320C6000 DSP – Multichannel Audio Serial Port (McASP) – Reference Guide, Texas Instruments Literature Number SPRU041C, August 2003.

[27] R. Kilgore, Link Port Open Systems Interconnect Cable Standard, Analog Devices Engineer-to-Engineer Note EE-106, October 1999.

[28] J. Kent and J. Sondermeyer, Interfacing ADSP-BF533/BF561 Blackfin Processors to High-Speed Parallel ADCs, Analog Devices Application Note AN-813.

[29] TMS320C6000 DSP Inter-Integrated Circuit (I2C) Module – Reference Guide, Texas Instruments Literature Number SPRU581A, October 2003.

[30] TMS320C6000 DSP Peripheral Component Interconnect (PCI) – Reference Guide, Texas Instruments Literature Number SPRUA75A, October 2002.

[31] TMS320C6000 DSP Host Port Interface (HPI) – Reference Guide, Texas Instruments Literature Number SPRU578A, September 2003.

[32] TMS320C6000 DSP General-Purpose Input/Output (GPIO) – Reference Guide, Texas Instruments Literature Number SPRU584A, March 2004.

[33] TMS320C6000 DSP 32-Bit Timer – Reference Guide, Texas Instruments Literature Number SPRU582A, March 2004.

[34] TMS320C6000 DSP Software-Programmable Phase-Locked Loop (PLL) Controller – Reference Guide, Texas Instruments Literature Number SPRU233B, March 2004.

[35] TMS320C6000 DSP Power-Down Logic and Modes – Reference Guide, Texas Instruments Literature Number SPRU728, October 2003.

[36] TMS320C620x/C670x DSP Boot Modes and Configuration – Reference Guide, Texas Instruments Literature Number SPRU642, July 2003.

[37] TMS320C6000 Peripherals – Reference Guide, Texas Instruments Literature Number SPRU109D, February 2001.

[38] TMS320C6000 DSP External Memory Interface (EMIF) – Reference Guide, Texas Instruments Literature Number SPRU266A, September 2003.

[39] T. Kugelstadt, A Methodology Of Interfacing Serial A-to-D converters to DSPs, Analog Application Journal, February 2000, pp. 1–10.

[40] Efficiently Interfacing Serial Data Converters To High-Speed DSPs, Analog Application Journal, August 2000, pp. 10–15.

[41] J. Sondermeyer, J. Kent, M. Kessler and R. Gentile, Interfacing The ADSP-BF535 Blackfin Processor To High-Speed Converters (Like Those On The AD9860/2) Over The External Memory Bus, Analog Devices Engineer-to-Engineer Note EE-162, June 2003.

[42] M. E. Angoletta et al., Beam Tests Of A New Digital Beam Control System For The CERN LEIR Accelerator, PAC '05, Knoxville, Tennessee, 2005.

[43] D. Dahnoun, Bootloader, Texas Instruments University Program, Chapter 9, 2004.

[44] Code Composer Studio IDE Getting Started Guide – Users Guide, Texas Instruments Literature Number SPRU509F, May 2005.

[45] V. Wan and K-S. Lee, Automated Regression Tests And Measurements With The CCStudio Scripting Utility, Texas Instruments Application Report SPRAAB7, October 2005.

[46] A. Campbell, K-S. Lee and D. Sale, Creating Device Initialization GEL Files, Texas Instruments Application Report SPRAA74A, December 2004.

[47] M.E. Angoletta et al., The New Digital-Receiver-Based System for Antiproton Beam Diagnostics, PAC 2001, Chicago, Illinois, 2001.

[48] D. Dart, DSP/BIOS Technical Overview, Texas Instruments Application Report SPRA780, August 2001.

[49] TMS320C6000 Optimizing Compiler – User's Guide, Texas Instruments Literature Number SPRU187L, May 2004.

[50] TMS320C6000 Assembly Language Tools – User's Guide, Texas Instruments Literature Number SPRU186N, April 2004.

[51] Rewind User's Guide, Texas Instruments Literature Number SPRU713A, April 2005.

[52] TMS320C6000 Instruction Set Simulator – Technical Reference, Texas Instruments Literature Number SPRU600F, April 2005.

[53] C. Brokish, Emulation Fundamentals for TIs DSP Solutions, Texas Instruments Application Report SPRA439C, October 2005.

[54] VisualDSP++ 4.5 – User's Guide, Revision 2.0, April 2006, Analog Devices Part Number 82-000420-02.

[55] B. Novak, XDS560 Emulation Technology Brings Real-time Debugging Visibility to Next Generation High-Speed Systems, Texas Instruments Application Report SPRA823A, June 2002.

[56] H. Thampi, J. Govindarajan, DSP/BIOS, RTDX and Host-Target Communications, Texas Instruments Application report SPRA895, February 2003.

[57] X. Fu, Real-Time Digital Video Transfer Via High-Speed RTDX, Texas Instruments Application report SPRA398, May 2002.

[58] S. Jung, Y. Paek, The Very Portable Optimizer For Digital Signal Processors, Proceedings of CASES'01, International Conference on Compilers, Architecture and Synthesis for Embedded Systems, Compilers and Optimization Session, Atlanta, Georgia, USA, 2001, pp. 84–92.

[59] D. Dahnoun, Linear Assembly, Texas Instruments University Program, Chapter 7, 2004.

[60] Analysis Toolkit v1.3 for Code Composer Studio – User's Guide, Texas Instruments Literature Number SPRU623D, April 2005.

[61] V. Wan and P. Lal, Simulating RF3 to Leverage Code Tuning Capabilities, Texas Instruments Application Report SPRAA73, December 2004.

[62] TMS320C6000 Optimizing Compiler – User's Guide, Texas Instruments Literature Number SPRU197L, May 2004.

[63] Analog Devices Team, Fast Floating-Point Arithmetic Emulation on Backfin Processors, Analog Devices Engineer-to-Engineer Note EE-185, August 2007.

[64] Y-T. Cheng, TMS320C6000 Integer Division, Texas Instruments Application Report SPRA707, October 2000.

[65] V. Wan and E. Young, Power Management in an RF5 Audio Streaming Application Using DSP/BIOS, Texas Instruments Application Report SPRAA19A, August 2005.

[66] D. Menard, D. Chillet and O. Sentieys, Floating-To-Fixed-Point Conversion For Digital Signal Processors, EURASIP *J. Appl. Signal Proc.*, vol. 2006, Article ID 96421.

[67] F. Culloch, Speeding the Development of Multi-DSP Applications, *Embedded Edge*, June 2001, pp. 22–29.

[68] G. Cooper and J. Hunter, Configuring Code Composer Studio For Heterogeneous Debugging, Texas Instruments Application Report SPRA752, May 2001.

[69] R. F. Hobson, A. R. Dyck, K. L. Cheung and B. Ressi, Signal Processing With Teams Of Embedded Workhorse Processors, *EURASIP J. Embedded Syst.*, vol. 2006, Article ID 69484.

[70] M. Kokaly-Bannourah, Introduction To TigerSHARC Multiprocessor Systems Using VisualDSP++, Analog Devices Engineer-to-Engineer Note EE-167, April 2003.

[71] M. Kokaly-Bannourah, Using The Expert Linker For Multiprocessor LDFs, Analog Devices Engineer-to-Engineer Note EE-202, May 2005.

[72] ADSP-TS101 TigerSHARC Processor – Hardware Reference, Revision 1.1, May 2004, Analog Devices Part Number 82-001996-01.

[73] TMS320C64x DSP Turbo-Decoder Coprocessor (TCP) – Reference Guide, Texas Instruments Literature Number SPRU534A, November 2003.

[74] TMS320C64x DSP Viterbi-Decoder Coprocessor (VCP) – Reference Guide, Texas Instruments Literature Number SPRU533C, November 2003.

[75] P. Cohrs, W. Powell and E. Williams, Creating a Dual-Processor Architectures for Digital Audio, *Embedded Edge*, June 2002, pp. 14–19.

[76] P.E. Dodd and W.L. Massegill, Basic Mechanism of Single-Event Upset in Digital Microelectronics, *IEEE Trans. Nucl. Sci.*, vol. 50, No. 3, June 2003, pp. 583–602.

[77] M. Vijay and R. Mittal, Algorithm-Based Fault Tolerance: A Review, Microproc. Microsyst., vol. 21, No. 3, Dec. 1997, pp. 151–161.

[78] Q. King et al., The All-Digital Approach To LHC Power Converter Current Control, CERN SL-2002-002 PO.

[79] H. Schmickler, Usage Of DSP And In Large Scale Power Converter Installations (LHC), these CAS proceedings.

[80] Q. King et al., Radiation Tests On The LHC Power Converter Control Electronics, Université Catholique De Louvain-La Neuve (UCL), CERN AB Note 2003-041 PO.

[81] J. Weber et al., PEP-II Transverse Feedback Electronics Upgrade PAC05, Knoxville, 2005, p. 3928.