System Design Methodology and Tools

Daniel Gajski
Junyu Peng
Andreas Gerstlauer
Haobo Yu
Dongwan Shin

Technical Report CECS-03-02
January 12, 2003

Center for Embedded Computer Systems
University of California, Irvine, CA 92697, USA

{gajski, pengj, gersrtl, haoboy, dongwans}@cecs.uci.edu
http://www.cecs.uci.edu

Table of Contents

Chapter 1.	System level design flow	5
Chapter 2.	System level modeling	35
Chapter 3.	Design of a GSM Vocoder	53
Chapter 4.	System level refinement	61
Chapter 5.	Design of a JPEG encoder	87
Chapter 6.	Design of a JBIG encoder	95
References		111

Chapter 1

System Level Design Flow

System Level Design Flow

What is needed and what is not

Daniel D. Gajski
Center for Embedded Computer Systems
University of California, Irvine
http://www.cecs.uci.edu/~gajski

Copyright © 2002 Daniel D. Gajski

System Level Design Flow

What is needed and what is not

Daniel D. Gajski
Center for Embedded Computer Systems
University of California, Irvine
`http://www.cecs.uci.edu/~gajski`

Copyright © 2002 Daniel D. Gajski

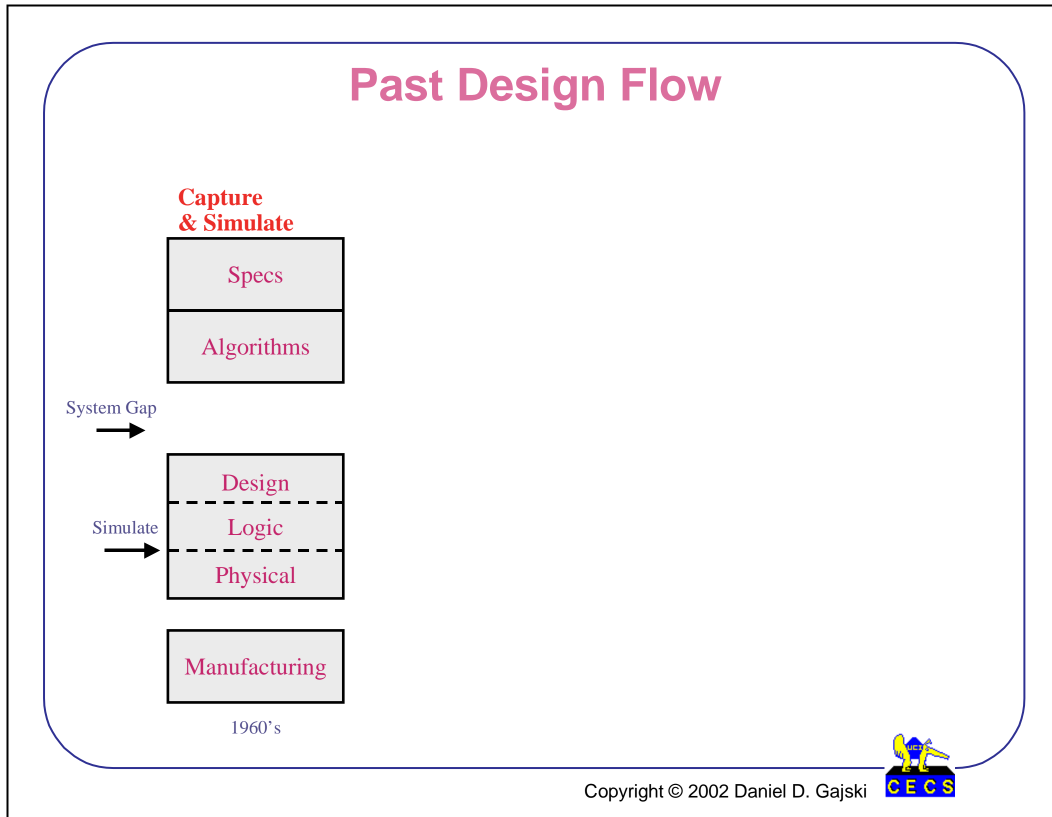
With complexities of Systems-on-Chip rising almost daily, the design community has been searching for new methodology that can handle given complexities with increased productivity and decreased times-to-market. The obvious solution that comes to mind is increasing levels of abstraction, or in other words, increasing the size of the basic building blocks. However, it is not clear what these basic blocks should be beyond the obvious processors and memories. Furthermore, if a design consists of SW and HW the modeling language should be based on C since standard processors come only with C compilers. Unfortunately, C language was developed for describing software and not hardware. It is missing basic constructs for expressing hardware concurrency and communication among components. Therefore, we need a language that can be compiled with standard compilers and that is capable of modeling hardware and software on different levels of abstraction including cycle-level accuracy.

Outline

- System gap
- Semantics, styles and refinements
- RTL Semantics
- System-Level Semantics
- Where are we going?
- Conclusion

Copyright © 2002 Daniel D. Gajski

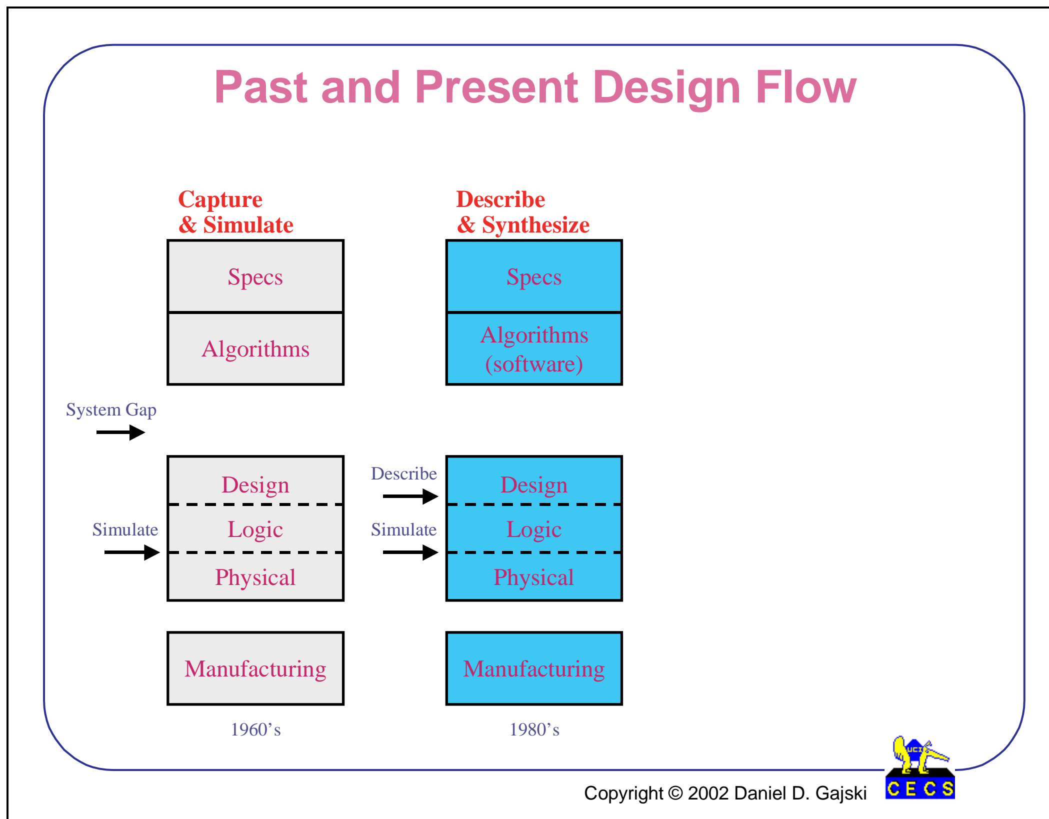
In order to find the solution for system-level design flow, we will look first at the system gap between SW and HW designs and then try to bridge this gap by looking at different levels of abstraction, define different models on each level and propose model refinements that will bring the specification to a cycle-accurate implementation. We will exemplify this by looking at the RTL and SL abstraction levels. From this point of view we will try to analyze the basic approaches in the academia and the industry today, and try to find out where we, as a design community, are going. We will finish with a prediction and a roadmap to achieve the ultimate goal of increasing productivity by more then 1000X and reducing expertise level needed for design of complex systems to the basic principles of design science only.



Design methodology has been changing with increase in complexity**.** We can distinguish three different phases over the last 40 years:

(a) Capture-and-Simulate Methodology (approximately from 1960s to 1980s)

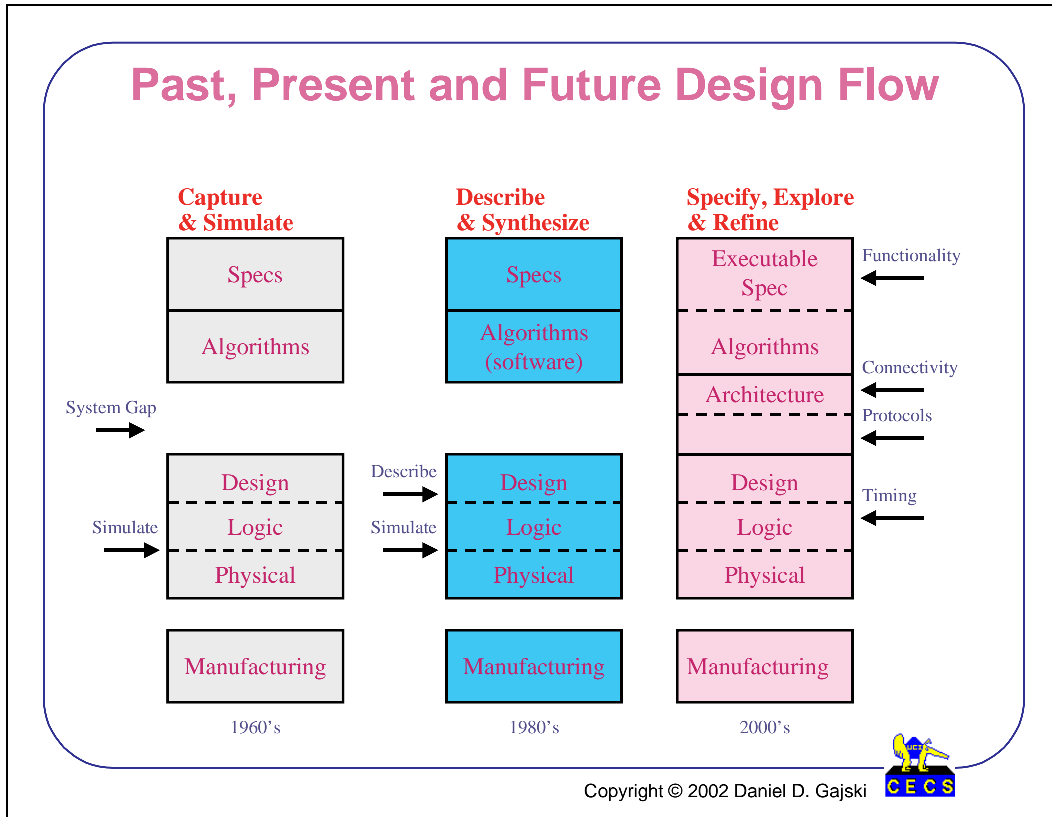
In this methodology software and hardware design was separated by a system gap. SW designers tested some algorithms and possibly wrote the requirements document and initial specification. This specification was given to HW designers who read it and started system design with a block diagram. They did not know whether their design would satisfy the specification until the gate level design was produced. When gate netlist was captured and simulated designers could find whether system really worked as specified. Usually it did not work as specified, and therefore specification was changed. This approach started the myth that specification is never complete. It took many years to realize that specification is independent of its implementation. The main obstacle to close the gap was the design flow is which designers waited until the gate level design was finished to verify the system behavior. Since they captured system design once at the end of design cycle, before simulation this methodology is called capture-and-simulate.



(b) Describe-and-Synthesize methodology (late 1980s to late 1990s)

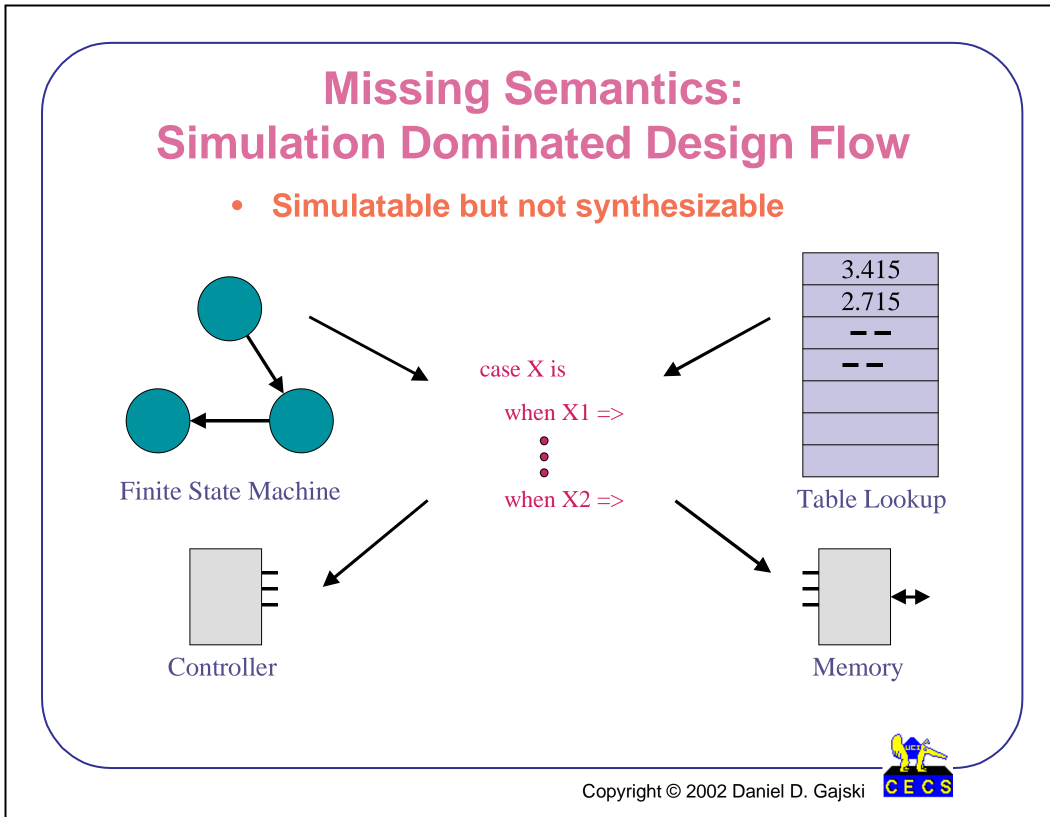
1980's brought us logic synthesis, which has significantly altered the design flow, since designers first specify what they want in terms of Boolean equations or FSM descriptions and the synthesis tools generate the implementation in terms of a gate netlist. Therefore, the behavior or the function comes first and the structure or implementation next. Also, there are two models to simulate: behavior (function) and gate-level structure (netlist). Thus, in this methodology specification comes before implementation and they are both simulatable. Also, it is possible to verify their equivalence since they can be both reduced to a canonical form in principle. In practice, today's designs are too large for this kind of equivalence checking.

By late 1990s the logic level has been abstracted to RTL or cycle-accurate description and synthesis. Therefore, we have two abstraction levels (RTL and gate levels) and two different models on each level (behavioral and structural). However, the system gap still persists.



(c) Specify, Explore-and-Refine Methodology (from early 2000s)

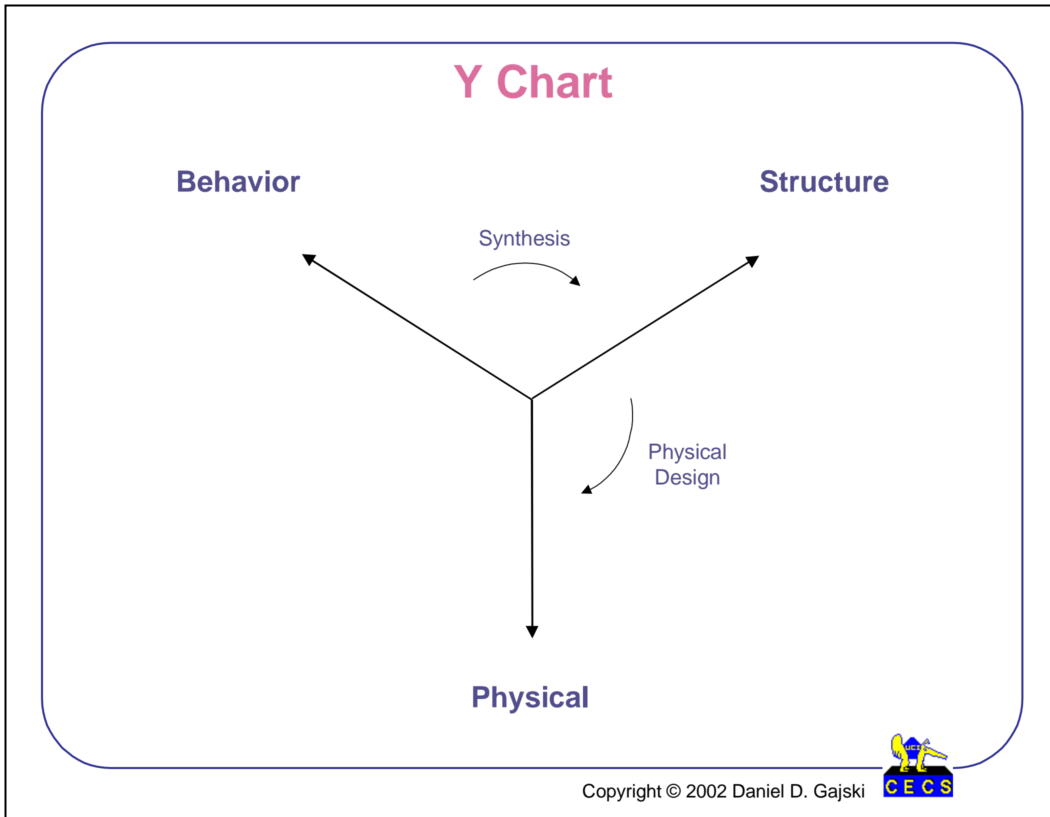
In order to close the gap we must increase level of abstraction from RTL to SL. On SL level we have executable specification that represents the system behavior or function and structural models with emphasis on connectivity or communication protocols. Each model is used to prove some system property such as functionality, connectivity, communication and so on. In any case we have to deal with several models in order to close the gap. Each model can be considered to be a specification for the next level model in which more detail in implementation is added. Therefore specify-explore-refine (SER) methodology represents a sequence of models in which each model is a refinement of the previous one. Thus, SER methodology follows the natural design process where designers specify the intent first, explore possibilities and then refine the model according to their decisions. Thus, SER flow can be viewed as several interactions of the basic describe-and-synthesize methodology.



With introduction of SL abstraction, designers have to generate even more models. One obvious solution is to automatically refine one model into another. However, that requires well defined model semantics, or, in other words, good understanding what is meant by the given model. This is not as simple as it sounds, since design methodologies and EDA industry has been dominated by simulation based methodologies in the past. For example, all HDL (such as Verilog, VHDL, SystemC, and others) are simulatable but not synthetizable or verifiable.

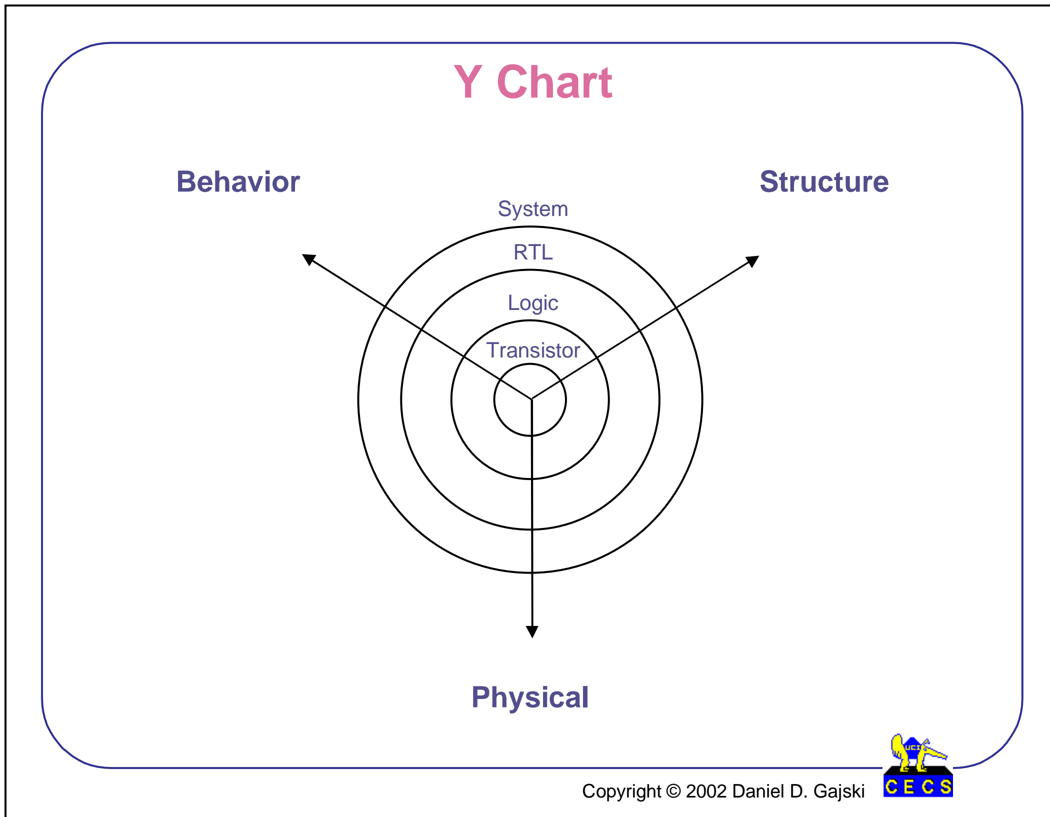
As an example of this problem, we can look at a simple case statement in any of the languages. It can be used to model a FSM or a look-up table, for example. However, FSMs and look-up tables require different implementations: a FSM can be implemented with a controller while a look-up table is implemented with a memory. On the other hand, using memory to implement an FSM or control logic to implement a table is not very efficient and not acceptable by any designer. Therefore, the model which uses case statement to model FSMs and tables is good for simulation but not good for implementation since a designer does not know what was meant by the case statement.

Thus, clean and unambigues semantics is needed for refinement, synthesis and verification. This semantics is missing from most of the simulation-oriented languages.

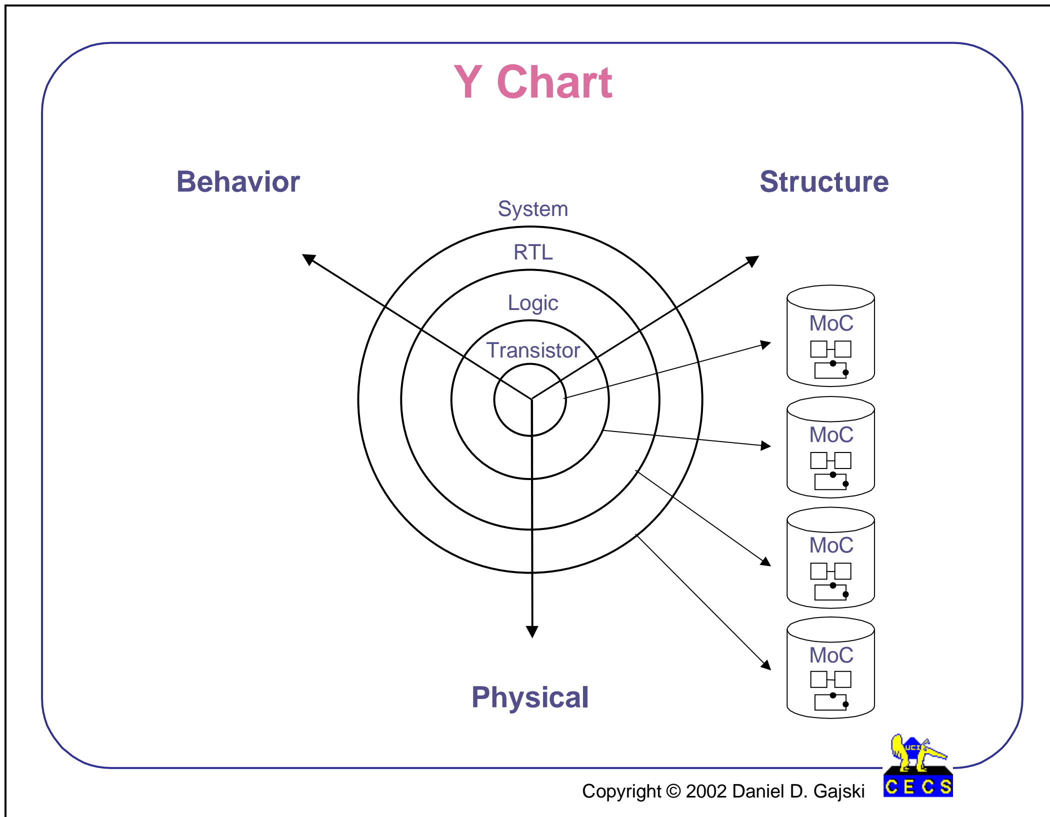


In order to explain the relationship between different abstraction levels, design models and design methodologies or design flow we will use Y-Chart, which was developed in 1983 to explain different design methodologies or design flows. The Y-Chart makes the assumption that each design, no matter how complex, can be modeled in three basic ways emphasizing different properties of the same design.

Therefore, Y-Chart has three axes representing design behavior (function, specification), design structure (connected components, block diagram), and physical design (layout, boards, packages). Behavior represents a design as a black box and describes its outputs in terms of its inputs and time. The black-box behavior does not indicate in anyway how to implement the black box or what its structure is. That is given on the structure axis, where black box is represented as a set of components and connections. Although, behavior of the black box can be derived from its component behaviors such an obtained behavior may be difficult to understand. Physical design adds dimensionality to the structure. It specifies size (height and width) of each component, the position of each component, each port and each connection on the silicon substrate or board or any other container.



Y-Chart can also represent design on different abstraction levels identified by concentric circles around the origin. Usually, four levels are used: Transistor, Logic, Register-transfers and System levels. The name of the abstraction level is derived by the main component used in the structure on this abstraction level. Thus, the main components on Transistor level are N or P-type transistors, while on Logic level they are gates and flip-flops. On the Register-transfers level the main components are registers, register files and functional units such as ALUs. While on the System level they are processors, memories and buses.



Each abstraction level needs also a database of components on this level. Each component in the database has tree models representing three different axes in the Y-Chart: behavior or function (sometimes called Model of Computation (MoC)), structure of components from the lower level of abstraction and the physical layout or implementation of the structure. These components are IPs for each abstraction level.

SoC Algebra

Algebra := < {objects}, {operations} >

SoC Algebra := < {models}, {transformations} >

Ordered set of transformations $< t_m, \ldots , t_2, t_1 >$ is a refinement iff

$$model\ B = t_m(\ldots (t_2(t_1(model\ A))) \ldots)$$

Question: { models } ? ; { transformations } ?

Copyright © 2002 Daniel D. Gajski

In order to define a design methodology or a design flow we must define first a set of different models and a set of transformations that will generate one model from the other. This is similar to an abstract algebra that consists of a set of objects and a set of operations on those objects. Each object and operations may have certain properties. For example, we say that on operations is commutative if the order of objects for this operation is not important. Similarly to an abstract algebra, we can define SoC algebra which consists of a set of models (objects) and a set of transformations (operations) with the following property: for each model in the set we can find a ordered set of transformations that will generate another model in the set. If the models in the set are ordered by the complexity or the level of detail we say tha SoC algebra is ordered. More formally, for each model A we can find a ordered set of transformations $< t_m, \ldots , t_2, t_1 >$ such that the next (more detailed) model B can be derived by applying this set of transformations to A. In other words: $B = t_m(\ldots (t_2(t_1(A))) \ldots)$. Every ordered SoC algebra defines a design methodology or design flow.

Why SoC Algebra?

1. Enabling standards for SL design automation
2. Discover truth behind SL myths
3. Define SL field (abstract semantics)
4. Identify SL methodology
5. Introduce interoperability
6. Support IP trade
7. Define how to SL languages

Copyright © 2002 Daniel D. Gajski

Proper definition of abstraction levels and models for design will enable standards in system-level (SL) design automation. It will also introduce some science into ad hoc methods used in approaches to create tools for SL simulation, synthesis and verification. It will also help define SL field and identify SL methodology. Formal definition of models (model semantics) will introduce interoperability and establishment of IP trading. It will also allow us to properly use SL languages such as SystemC, SpecC and others.

Semantics, Styles & Refinements

- Each model uses well defined semantics
- Each model has simple style
- Each style uniquely expressed
  - no syntactic variance or semantic ambiguity
- Each model needs style checker

- Ordered set of models
- Clear refinement rules
- Ordered set of refinement rules for each model
- Verifiable model refinements

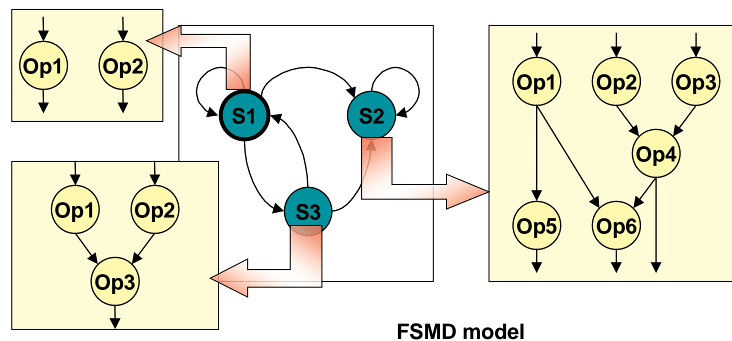
Copyright © 2002 Daniel D. Gajski

The main requirement for success of SoC algebra and SL design automation is formal definition of models and transformations. Thus, each model needs well defined semantics which can be expressed with a very simple syntax and modeling style which is free of syntactic variance or semantic ambiguity. Syntactic variance allows same meaning to be expressed (or described) syntactically in several different ways, which makes all synthesis or verification algorithms extremely complicated. On the other hand, semantic ambiguity allows same syntax to have different meanings as shown previously in the case of missing semantics. Since each language can be use for describing different models and each model may require different style, we need a style checkers to help designers comply with the model style.

In order to automate SoC design methodology we need ordered set of models and a ordered set of transformations or refinement rules for transforming higher-level model into lower-level ones. Also, the model transformations should be verifiable. We will explain these concepts for RTL and SL levels of abstraction.

RTL Computational Models

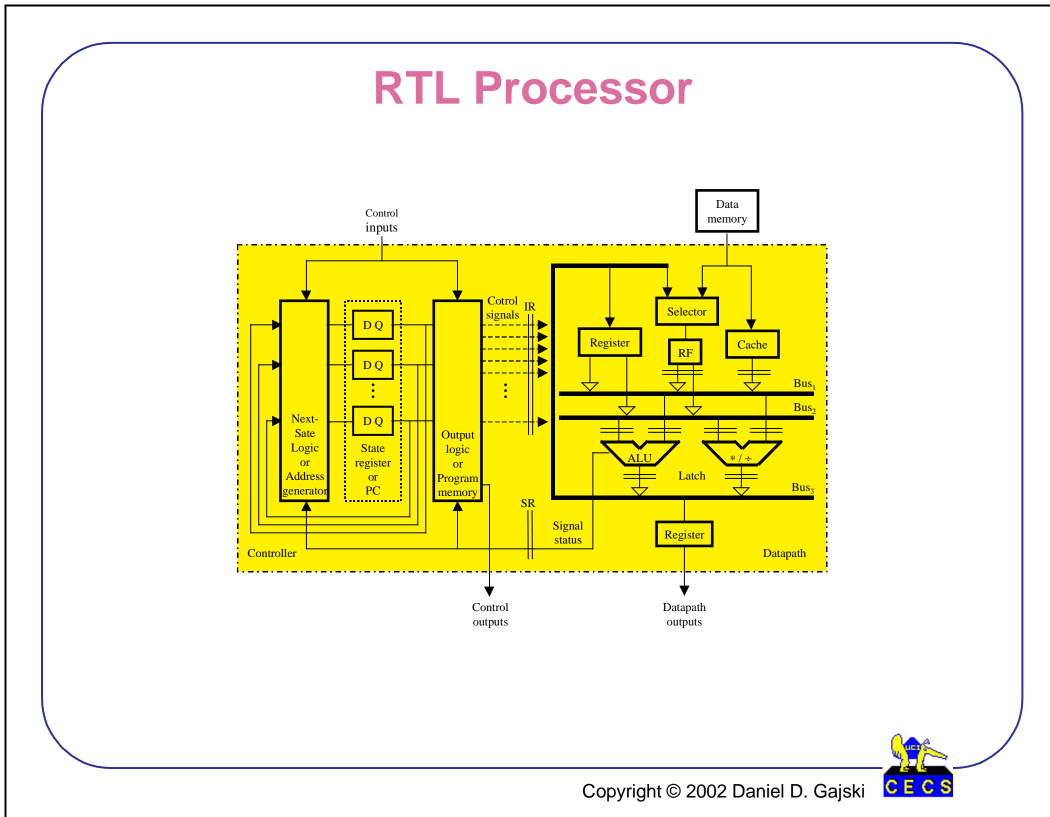
- **Finite State Machine with Data (FSMD)**
  - Combined model for control and computation
    - FSMD = FSM + DFG
  - Implementation: controller plus datapath



Copyright © 2002 Daniel D. Gajski

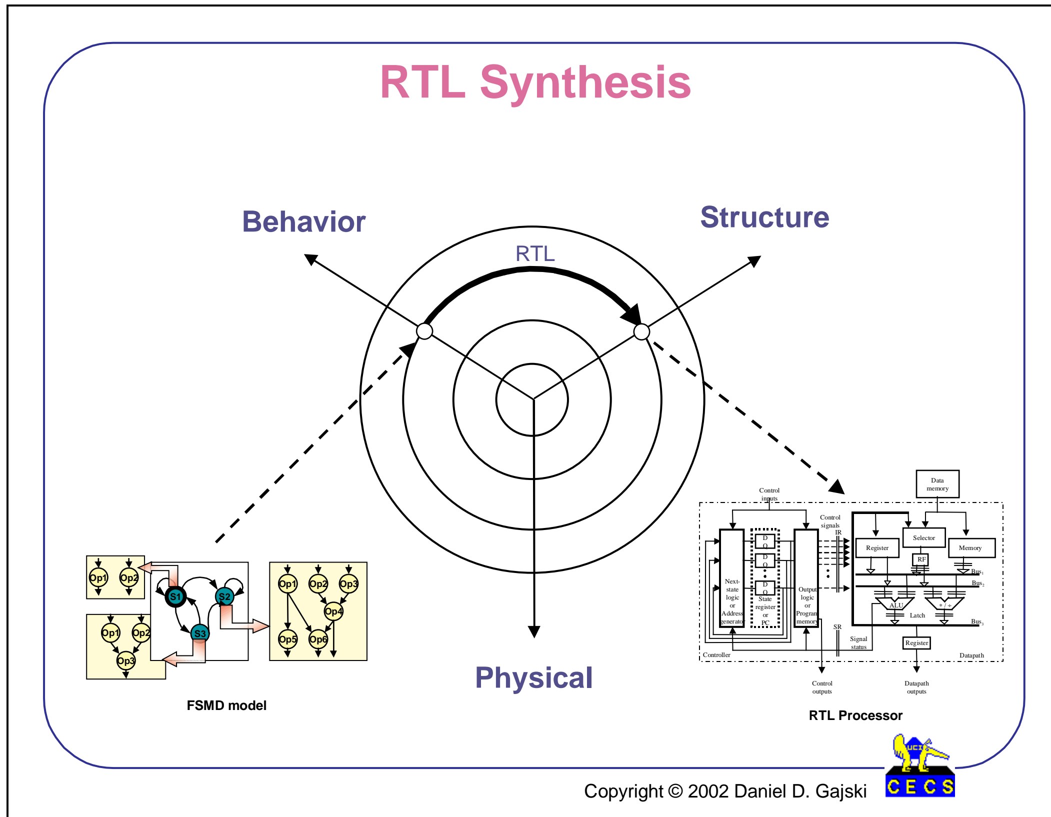
The RTL behavior or computational model is given by a Finite-state-machine with Data (FSMD). It combines finite-state-machine (FSM) model for control and data-flow-graph (DFG) for computation. FSM has a set of states and a set of transitions from one state into others depending on the value of some of the input signals. In each state FSMD executes a set of expressions represented by a DFG. FSMD model is clock-accurate if each state takes a single clock-cycle.

It should be noted that FSMD model encapsulates the definition of the state-based (Moore-type) FSM in which the output is stable during duration of each state. It also encapsulates the definition of the input-based (Mealy-type) FSM with the following interpretation: Input-based FSM transitions to a new state and outputs data conditionally on the value of some of FSM inputs. Similarly, FSMD executes set of expressions depending on the value of some FSMD inputs. However, if the inputs change just before the clock edge there may be not enough time to execute the expressions associated with that particular state. Therefore, designers should avoid this situation by making sure the input values change only early in the clock period or they must insert a state that waits for the input value change. In this case if the input changes too late in the clock cycle, FSMD will stay in the waiting state and proceed with a normal operation in the next clock cycle.

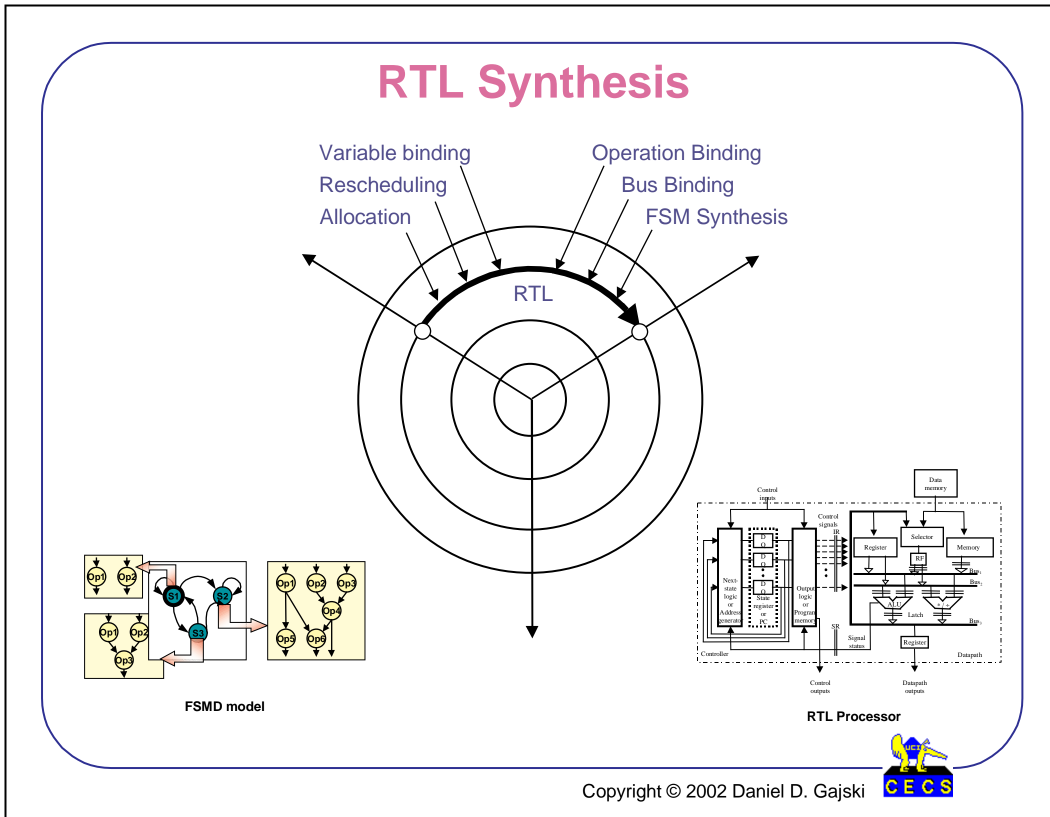


Copyright © 2002 Daniel D. Gajski

The behavior represented by a FSMD model can be implemented by the structure of a RTL processor, which consists of a Controller and the Datapath. Datapath consists of set of storage elements (registers, register files, memories), set of functional units (ALUs, multipliers, shifters, custom functions) and set of busses. All these RTL components may be allocated in different quantities and types and connected arbitrarily through busses. Each component may take one or more clock cycles to execute, each component may be pipelined and each component may have input or output latches or registers. The entire Datapath can be pipelined in several stages in addition to components being pipelined themselves. The Controller defines the state of the RTL processor and issues the control signals for the Datapath.
The RTL processor may represent an implementation of a standard processor (such as Pentium, PowerPC, or a DSP) or a custom processor or IP specifically synthesized for a particular function. In the former case, the Controller is programmable with a Program memory, PC and an Address generator. In the later case, the Controller is hardwired with a State register, Next-state logic and Output logic providing the control signals to the Datapath.



RTL synthesis starts with the FSMD model in the behavior axes of the Y-chart and ends up with a custom RTL processor containing any number and type of components connected as required by the FSMD model. Note, that FSMD model can be obtained easily from a programming language code such as C by grouping all the consecutive statements into basic blocks (BB) and introducing two states for each **if** statement or **loop** statement, where each state executes a BB. Such a FSMD is sometimes called super-state FSMD since each BB may be considered to be executed in one super state. This generation is very simple. Note that each BB will be partitioned into several states during RTL synthesis, where the number of states depends on the resources allocated to the RTL processor.

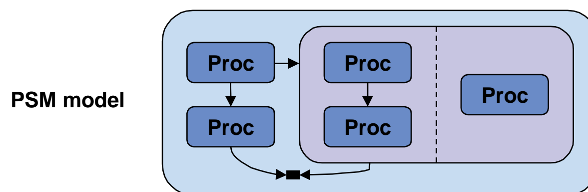


RTL synthesis is the process of converting a FSMD model into a RTL processor that generates the same result. It consists of several tasks:
(a) Allocation of components from the RTL library,
(b) Rescheduling of computation in each state since some components may need more than one clock cycle,
(c) Binding of variables, operations and register transfers to storage elements, functional units and busses,
(d) Synthesis of programmable or hardwired controller.
(e) Generation of refined model representing the RTL processor.
Any of the above tasks can be performed manually or automatically. If all of them are done automatically, we call the above process RTL synthesis. On the other hand, if (a) to (d) are performed by designer and only (e) is done automatically, we call the process model refinement. Obviously, many other strategies are possible as exemplified by available EDA tools that may perform each of the above tasks only partially in automatic fashion and leave the rest to the designer.

System Behavioral Model

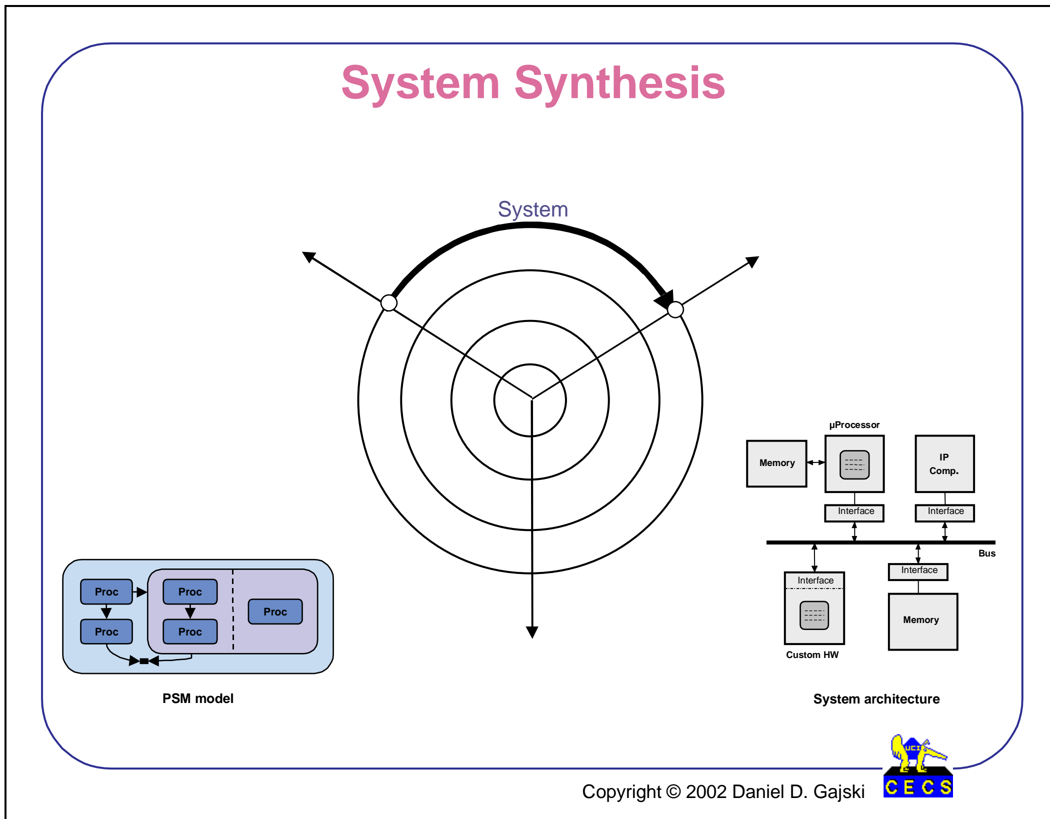
- **Program State Machine**
  - States described by procedures in a programming language
  - Example: SpecC! (SystemC!)

PSM model

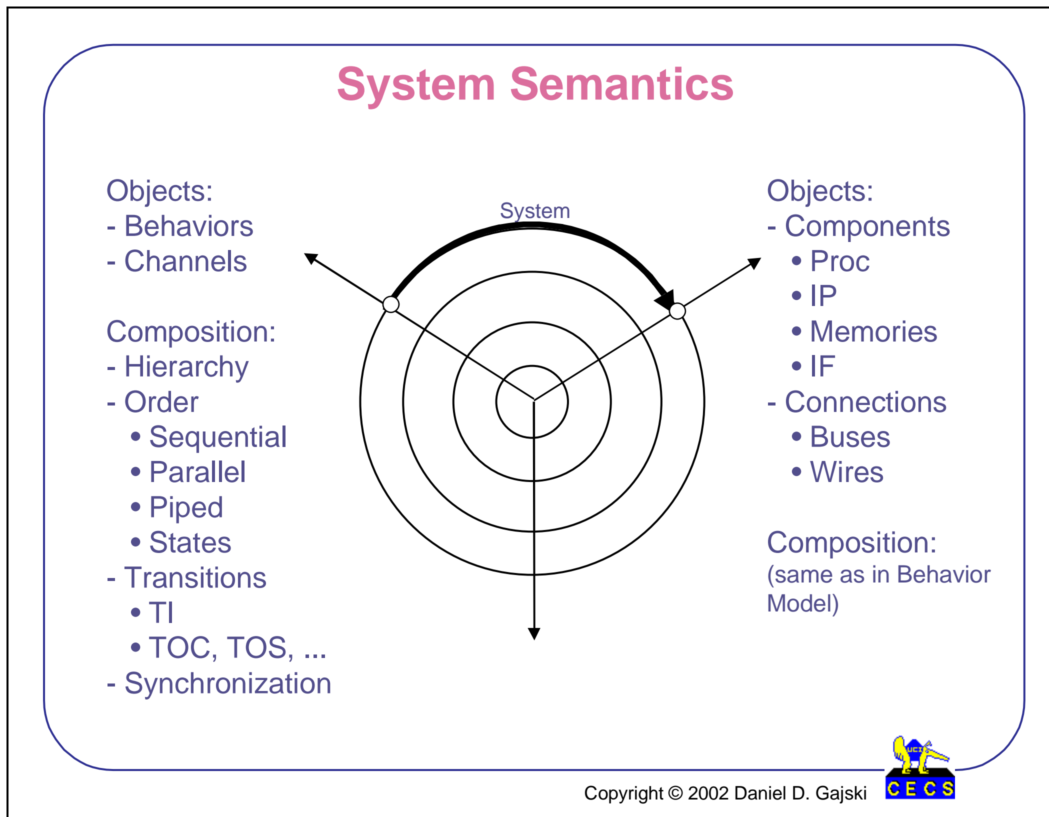


Copyright © 2002 Daniel D. Gajski

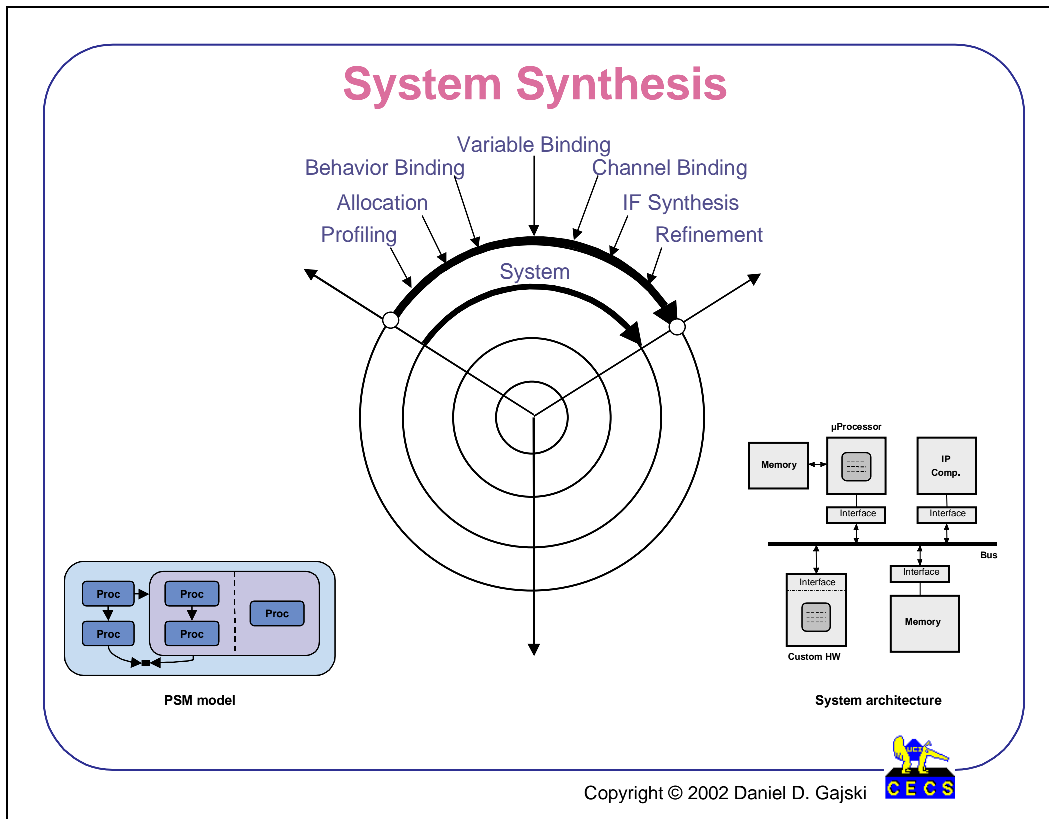
Since systems consist of many communicating RTL processors the FSMD model will not suffice. Furthermore, such model must represent SW and HW. The easiest way is to retain the concept of states and transitions as in a FSM but to extend the computation in each state to include any procedure in a programming language such as C/C++. Furthermore, in order to represent an architecture with several processor working in parallel or in pipelined mode we must introduce concurrency (two states running in parallel) and pipelining (several states running in parallel with data passed between them sequentially). Since states are running concurrently we need a synchronization mechanism for data exchange. Furthermore, we need a concept of channel to encapsulate data communication. Also, we need to support hierarchy to allow humans to write easily complex systems specifications.



System synthesis starts with a behavioral model of the system such as PSM model and generates the system architecture such as bus-functional model, which describes components, their behavior and connectivity among components. Such model describes the operation of each component as a set of functions or procedures with lumped time assigned to each function. The communication is described with channels including time accurate protocols.

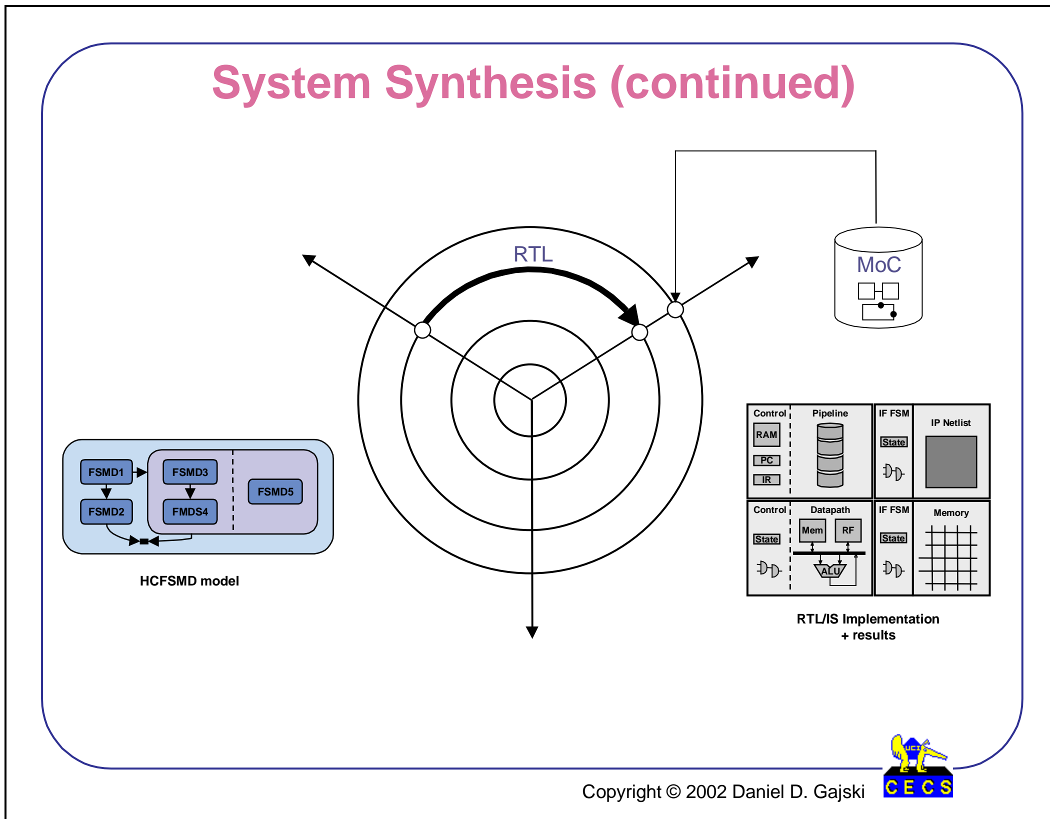


In system semantics we must transform a behavioral model into a structural model. Following the SoC algebra the behavioral model is a composition of two objects: behaviors and channels. They can be composed hierarchically and ordered through sequential, parallel pipelined and state operators. The transition from state to state may be accomplished on interrupt, completion, or specific variable value. For concurrent processing we can use some type of synchronization such as waiting for event generated by another process. The structural model uses different object: behaviors are replaced by components (processors, IPs, etc.) and channels are replaced by buses or wires with well defined protocols. The composition rules stay the same.



PSM model can be synthesized into a arbitrary architecture by the following set of tasks:
(a) Profiling of code in each behavior and collecting statistics about computation, communication, storage,traffic, power consumption, etc.,
(b) Allocating components from the library of processors, memories, IPs and custom RTL processors,
(c) Binding behaviors to processing elements, variables to storage elements (local and global), and channels to busses,
(d) Synthesizing IF between components and busses with incompatible protocols,
(e) Refining the PSM model into a architecture model that reflect allocation and binding decisions.
The above tasks can be performed automatically or manually. Tasks (b)-(d) are usually performed by designers while tasks (a) and (e) are better done automatically since they require lots of mundane effort.

Once the refinement is performed the architecture model can be validated by simulation quite efficiently since all the component behaviors are described by high level functions.



Copyright © 2002 Daniel D. Gajski

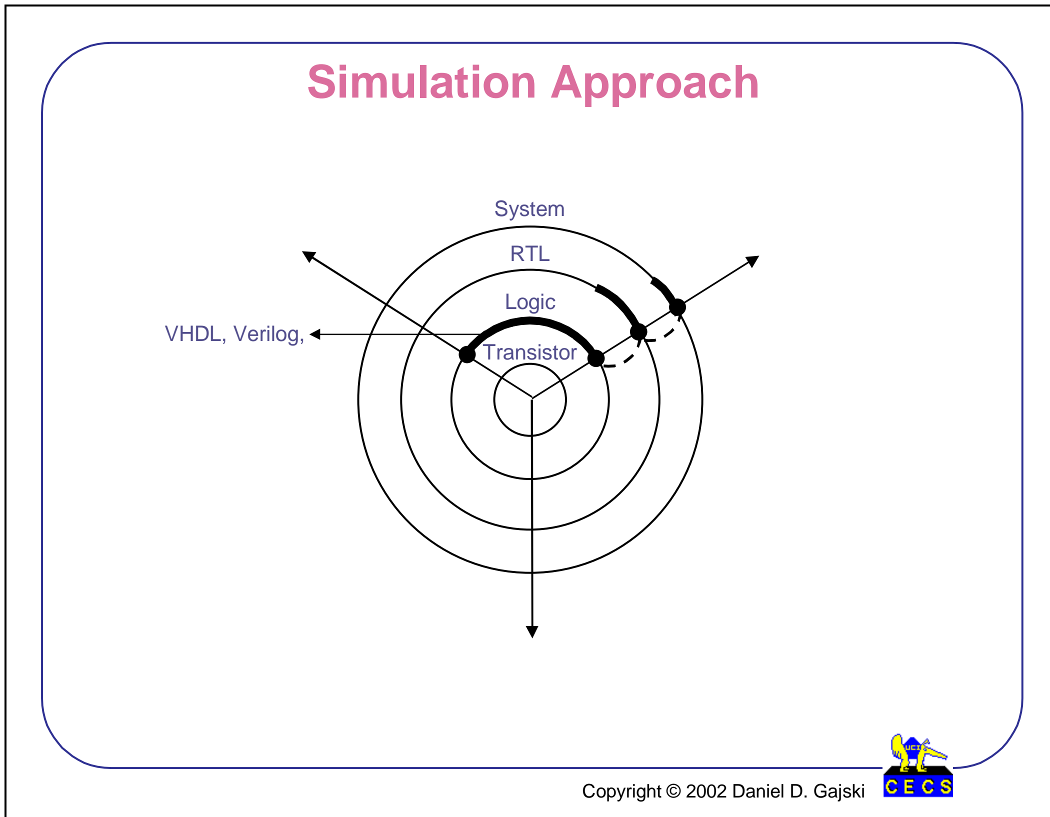
In order to generate cycle –accurate model, we must replace each component functional model with a FSMD model for custom HW or IS model for standard processors executing SW. Once we have bus-functional model, we can refine it further to cycle-accurate model by performing RTL synthesis for custom RTL processors or custom IFs and compiling behaviors assigned to standard processors to instruction-set level and inserting IS simulator to execute the compiled instruction stream. RTL synthesis can start from super-state FSMD that is obtained through two different mechanisms. On one hand, we can replace a behavior assigned to an IP with a super-state FSMD model from IP library. On the other hand, we can perform RTL synthesis on any behavior assigned to RTL processor after refining the behavior to BB super-state FSMD. After RTL/IS refinement we end up with cycle-accurate model of the entire system.

System-Level Trends

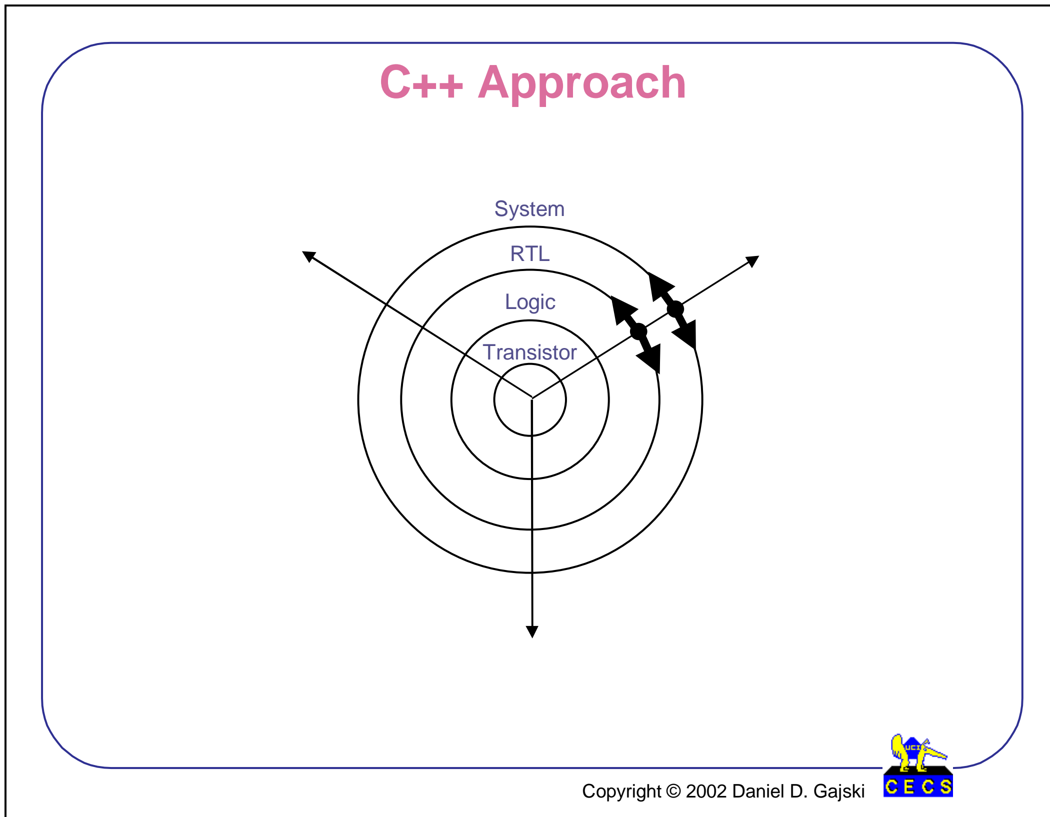
- Simulation
- C++
- MoC
- Syntax first
- Semantic first

Copyright © 2002 Daniel D. Gajski

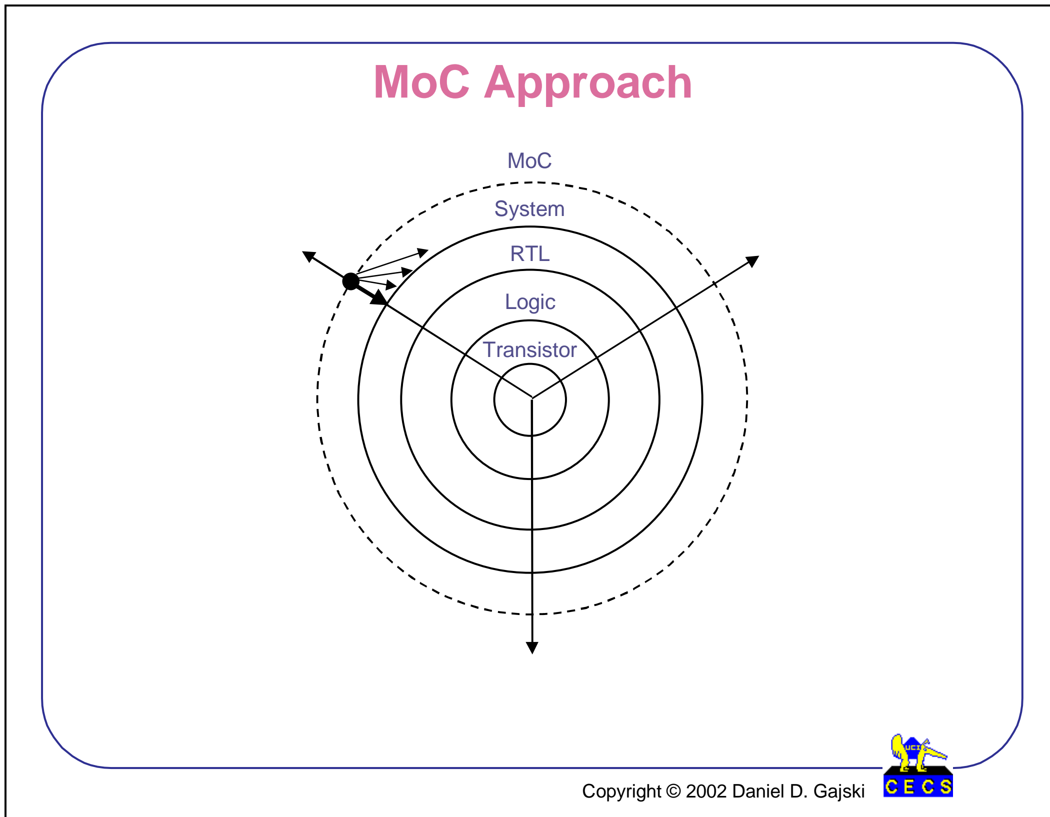
System-level abstraction arrived suddenly as a consequence of large increases in chip complexity. Since the methodology and tools are not available yet, many research groups are experimenting with different approaches to solve the complexity issues. We can identify several more popular approaches. Simulation is the most popular among EDA community. Some people are trying to extend C++ for their particular needs, while others are developing new MoCs for particular applications. Others are trying to develop multi purpose system-level languages. Two trends can be identified in this group depending whether syntax or semantics is developed first. We will look at each trend in detail in the rest of this report.



Simulation approach focuses design methodology on simulation. Designers develop design on different levels and simulate its behavior by writing simulation models. Formal verification or equivalence checking is almost impossible since simulation models are ambiguous and not strict enough for synthesis and verification. Usually, simulation model must be restricted to a language subset or a particular style to be synthesizable and even more restricted to be verifiable. This simulation approach reverses the trend established by logic synthesis where design methodology focuses on synthesis while creating simulatable models automatically.

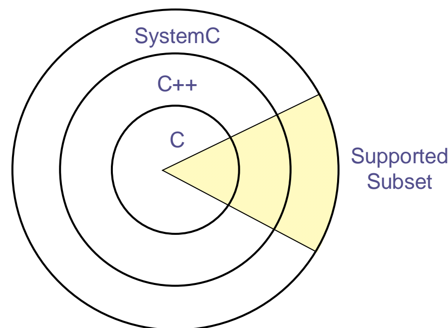


C++ is an extensible language. Providing a standard set of C++ classes and a free-source simulator is very appealing to all faculty ad their students as well as to large number of industrial researchers and managers since they can add some classes and adapt the standard set to their particular application, methodology and modeling style. However, these additional classes make their methodology and models useless for others. Therefore, C++ is good for experimentation but is improbable to become a standard modeling language.



Developing new models of computation (MoCs) that are tuned to different applications simplifies the specification captive for a particular application. However, this approach leaves a huge gap between MoC and working SW and HW on a SoC. This gap is difficult to bridge with today's algorithms and methods for synthesis, verification and test in order to generate an efficient design.

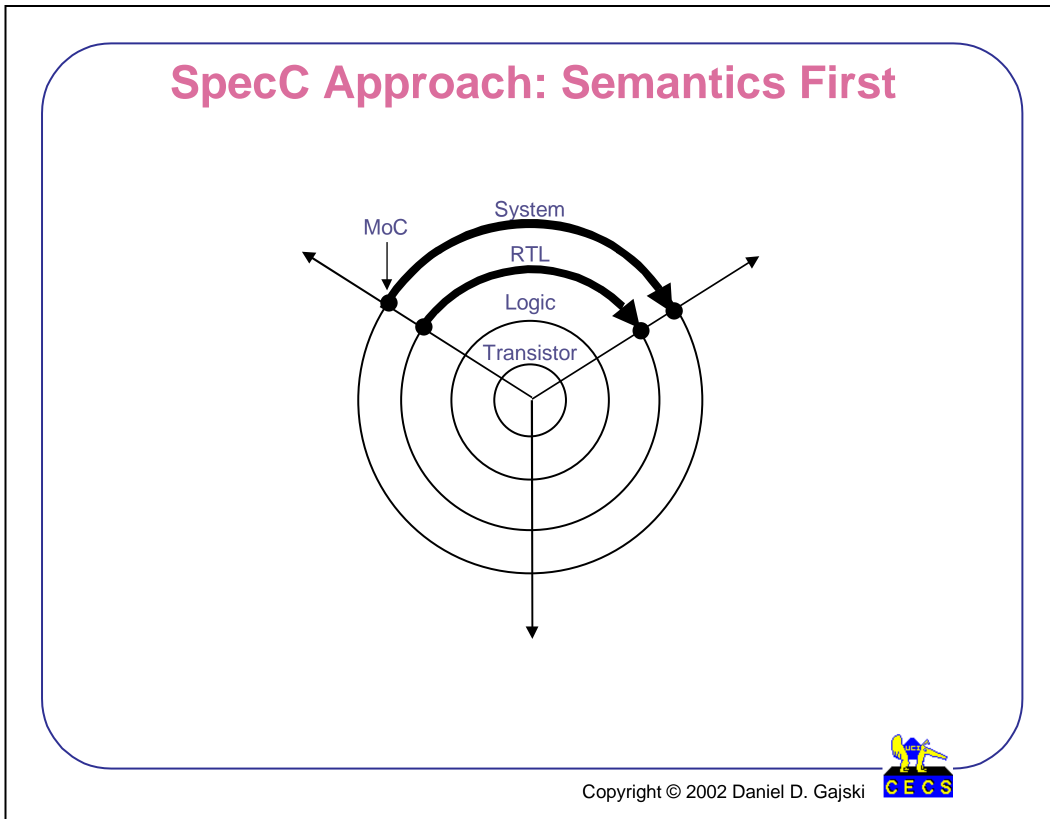
SystemC Approach: Language First



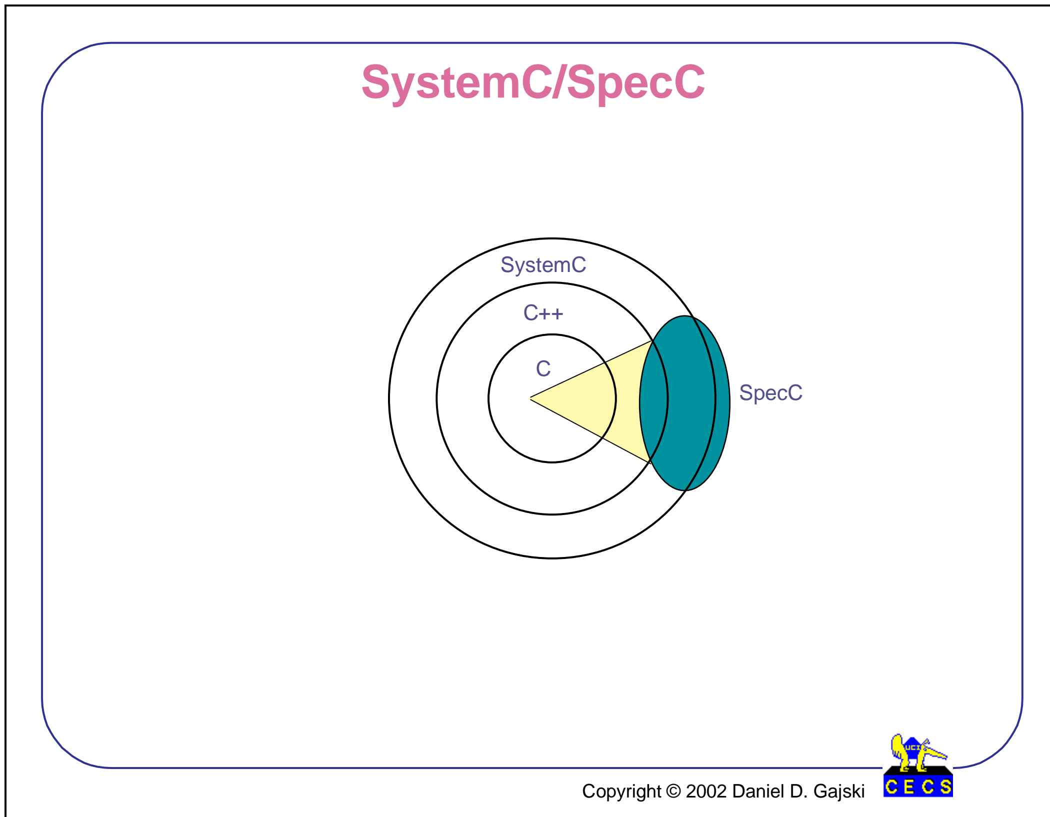
Source: J. Kunkel, VP Synopsis, (CODES, May 2002)

Copyright © 2002 Daniel D. Gajski

There are two approaches in developing system methodology. One approach is to develop a language first and then experiment with it to find how to use it in different applications. Such a language is accompanied with a simulator to support simulation of the models written in the language. SystemC is such a language based on C++. It is appealing to many researchers who can experiment with it by adding new classes that they optimize for their own application. However this extensibility does not help. On the contrary, it creates incompatibility, since everyone has his or her own classes that are not compatible with anyone else. At this moment, SystemC already has too many classes that are not easy too support. That will eventually result in a subset for synthesis and another subset for verification. It is difficult to predict when these subsets will emerge, how they will look like and who will define them. Similar situation occurred with other simulation-oriented languages such as VHDL and Verilog.



The other approach to language definition is to define abstraction levels and define behavioral and structural models and transformation rules for deriving one model from the other. This approach is more limited since there is less flexibility later but it results into interoperability of models and tools. With well-defined semantics the designer education is easier and IP trade can develop sooner. Since SpecC was designed to support minimal and orthogonal set of concepts for modeling SW and HW it is easy to use by designers as well as tool makers and it does not need subsetting since the SpecC models are synthetizable and verifiable. The SpecC version 2.0 supports automatic refinement of a PSM model into architecture/bus-functional model, into FSMD model, and finally into cycle-accurate RTL model ready for synthesis with available standard EDA tools.



In the moment, SystemC looks like a popular simulation language that is looking for synthesis and verification subsets. On the other hand, SpecC is more restricted, synthesizable and verifiable. Therefore, the obvious conclusion is that SpecC may become synthesizable and verifiable subset of SystemC, since it compiles to C++ for simulation anyhow. This outcome may become a real possibility if both group came together and smooth some minor differences in the syntax and semantics of both languages.

Conclusion

Work to be done:

1. Abstraction Levels
2. Model Semantics
3. Refinement Rules
4. Methodology
5. Language
6. Simulation, Synthesis, Verification Tools
7. ESDA Market/Community Emergence

Prediction: No success in 7 without 1-6

Copyright © 2002 Daniel D. Gajski

From the above discussion it is obvious that old strategy of developing a language and subsetting it for different design task is not acceptable for SL design. The strategy for success is to solve the following issues:
(1) Define the abstraction levels for SoC design flow,
(2) Define semantic for each model without synthetic variance or semantic ambiguity,
(3) Define refinement rules for deriving a refined model from a more abstract one,
(4) Define the methodology including models, tasks and necessary tools,
(5) Design the language to support the above models, refinements and methodology,
(6) Develop tools for simulation, synthesis and verification of different models,
(7) Stimulate the organization of SL community, and ESDA market.
It is obvious that issues 1-6 must be resolved before a community or market will emerge. SL academic community has made sizable progress toward this goal in the past, but more work is needed in the future before we can claim that SL design flow is understood.

Chapter 2

System Level Modeling

System Level Modeling

Andreas Gerstlauer
Daniel D. Gajski
Center for Embedded Computer Systems
University of California, Irvine
http://www.cecs.uci.edu

Copyright © 2002 A. Gerstlauer, D. Gajski

Motivation and Goals

- **Well-defined abstraction levels**
  - Focus on critical issues
  - Early feedback
- **Well-defined models**
  - Automated synthesis and refinement
  - Minimal, localized changes
  - IP integration
- ➢ **Rapid design space exploration**
- ➢ **Synthesis-based design flow**

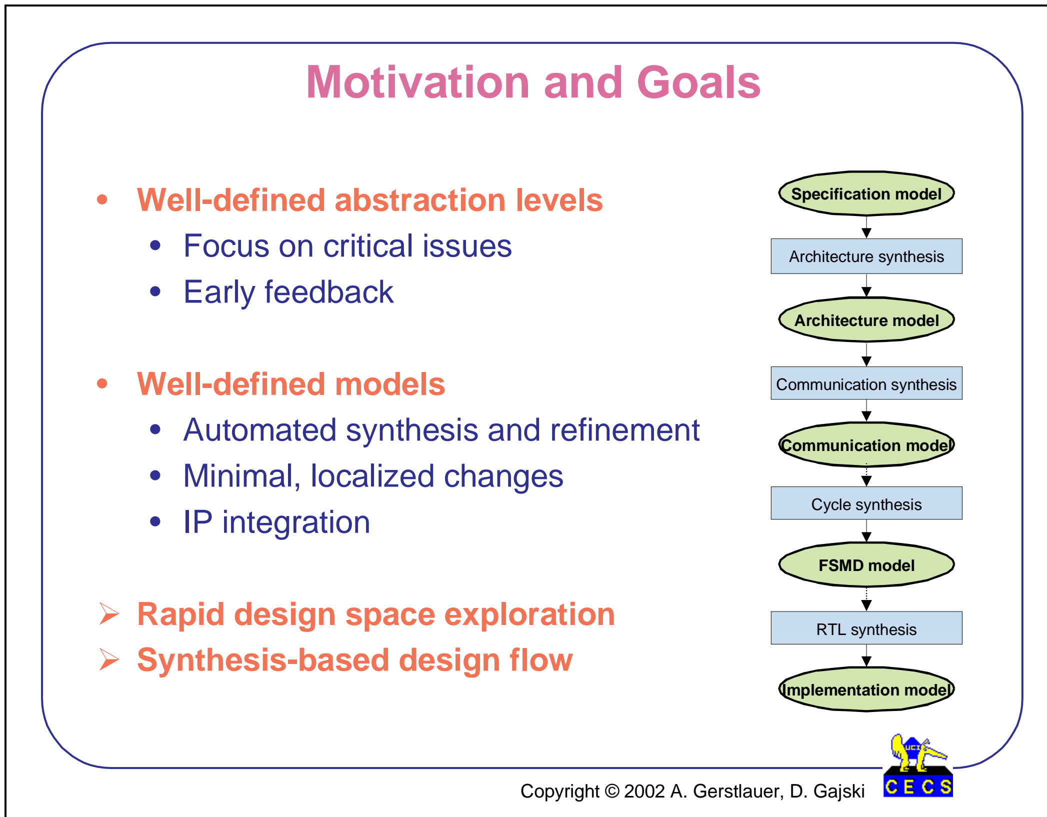
Copyright © 2002 A. Gerstlauer, D. Gajski

A requirement for any design flow is a set of well-defined abstraction levels and models. The number of models and their semantics must be defined in a way that will simplify design decisions and model transformations. Also, the number of objects in the model must be minimized to improve efficiency while providing enough detail to perform exploration at each step. Furthermore, a clear and unambiguous definition of these models is then needed for automation of synthesis and verification. In addition, such a formalized definition is a necessity for interoperability across tools, designers and IP vendors.

Traditionally, abstracted models of a design are used mainly for simulation purposes. In such simulation-centric approaches, the designer is responsible for manually rewriting the model to accommodate the changes in the design. However, none of these approaches attack the vertical integration of models that is needed for a synthesis-centric design flow with automatic refinement of higher-level models into lower-level ones.

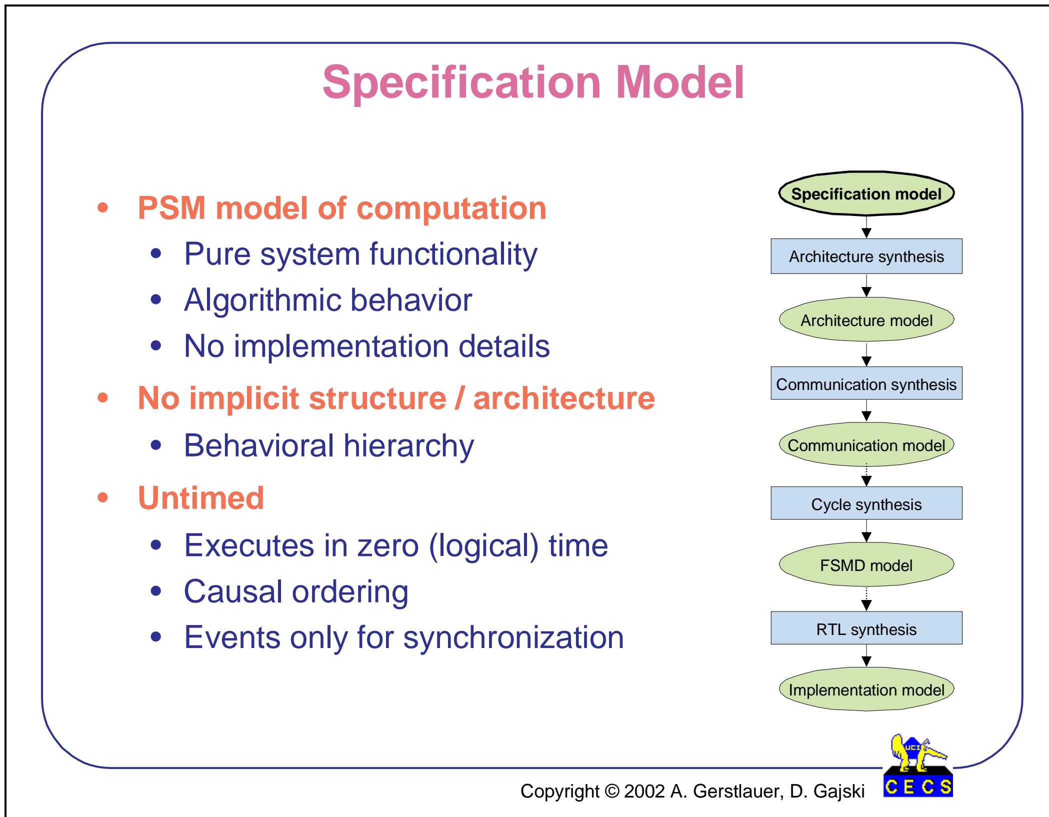
For each design task, the models at the input and output of the flow have to be defined in such a way that the transformation between the models is possible. If the gap between them is too big, the task needs to be performed in several steps, thus creating additional intermediate models. On the other hand, tasks should be as independent as possible in order to perform them separately.

In summary, models on any level trades off accuracy for efficiency. On the other hand models have to be defined with the right amount of detail that will allow rapid and meaningful exploration, synthesis, and validation.



In general, the system design process is too complex to be completed in one single step. The gap between requirements and implementation is impossible to cover using non-exponential algorithms. Hence, we need to divide the process into a sequence of smaller, manageable steps. Since computation and communication refinement are largely orthogonal we can subdivide the design process into the two separate tasks of computation and communication design. However, computation synthesis needs to be performed before communication synthesis since partitioning of computation influences the amount of communication to be performed.

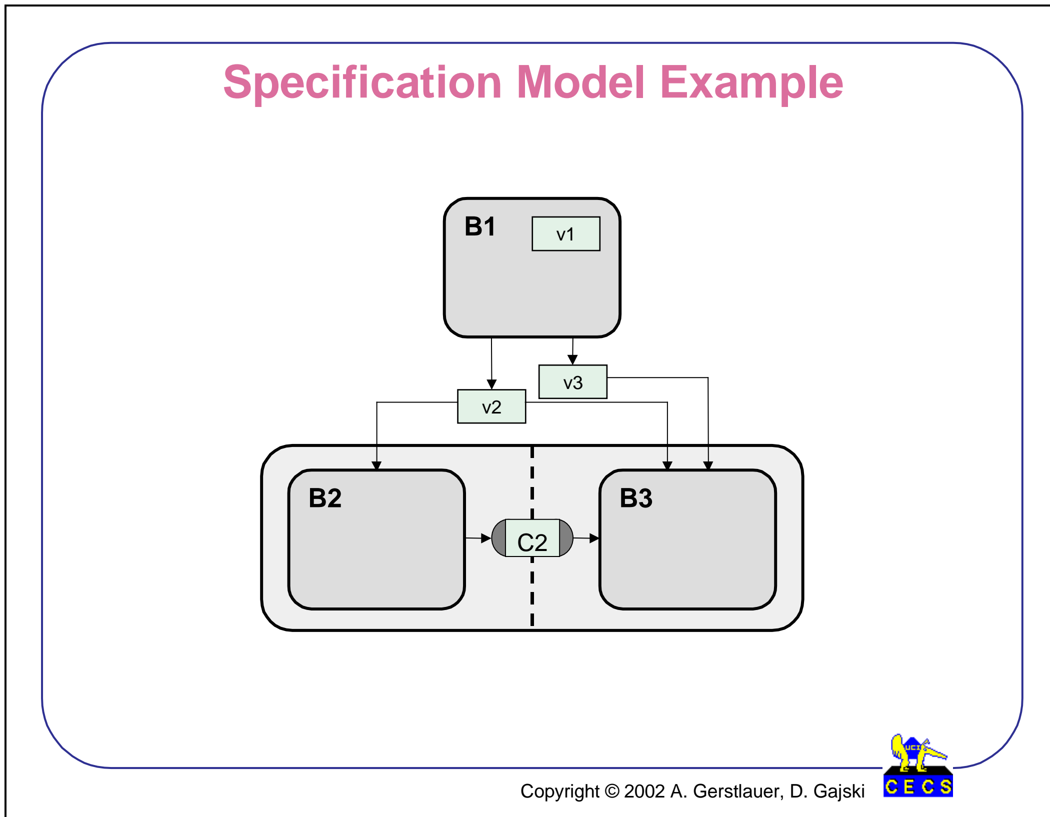
In our design methodology, system design starts with the behavioral specification model. In the first step, computation is implemented on processing elements (PEs), resulting in the intermediate architecture model, which is a mixed behavioral/structural description. It defines the computation structure but leaves communication at a behavioral level. Communication synthesis completes the system design flow and creates the structural system communication model. On the lower level of abstraction, each PE is then implemented separately trough cycle and RTL synthesis. Clock scheduling in cycle synthesis creates the cycle-accurate FSMD model. Finally, the implementation model as the result of RTL synthesis is a structural description of each PE in the form of an RTL netlist.



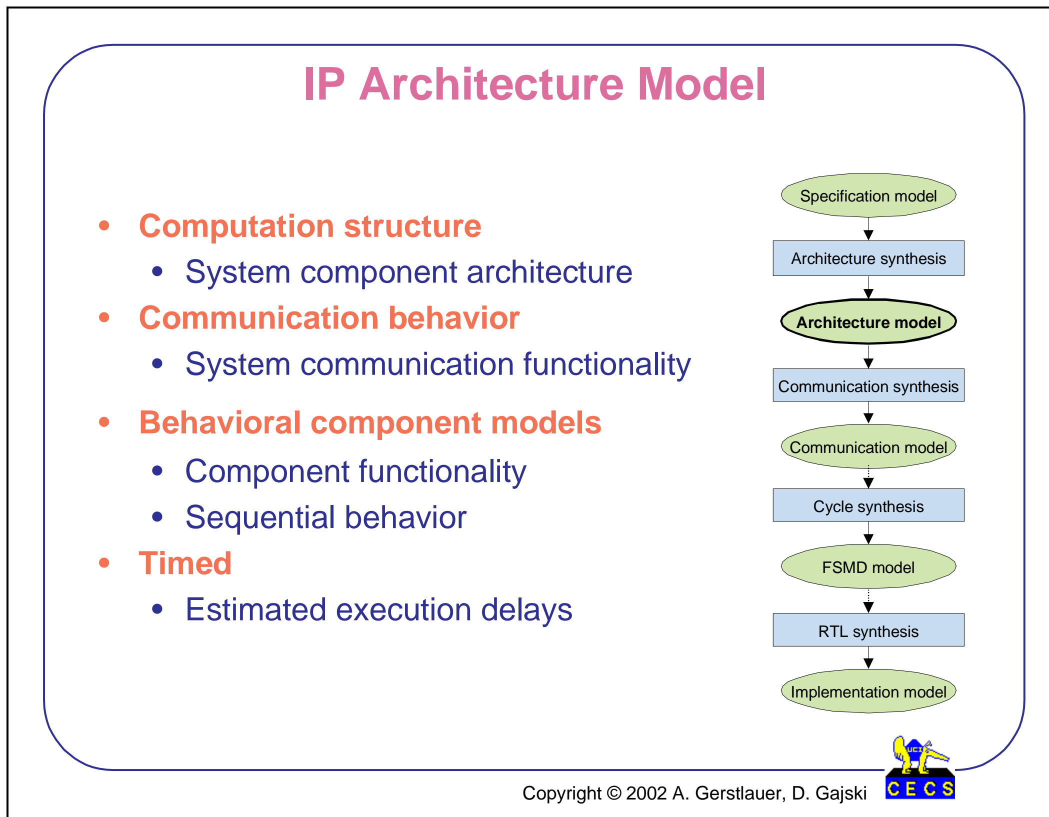
The system design process starts with the specification model written by the user to specify the desired system functionality. It forms the input to architecture exploration, the first step of the system design process. Therefore, it defines the basis for all exploration and synthesis. For example, the specification model defines the granularity for exploration through the size of the leaf behaviors, it exposes the available parallelism, it separates communication from computation, and it uses hierarchy to group related functionality and to manage complexity.

The specification model is a purely functional, abstract model that is free of any implementation details. The hierarchy of behaviors in the specification model solely reflects the system functionality without implying anything about the system architecture to be implemented. For example, parallel behaviors in the specification model describe independent groups of functions that can run concurrently. However, parallelism in the specification does not make any premature assumptions about a concurrent implementation.

The specification model is also free of any notion of time. The model executes in zero logical (simulation) time. Events in the specification model are used for synchronization, which establishes a partial ordering among the behaviors based on desired causality.



This figure shows an example of a simple yet quite typical specification model.
In general, at each level of hierarchy the specification is an arbitrary serial-parallel composition of behaviors. Behaviors communicate through variables and synchronize through events attached to their ports. At the lowest level of hierarchy, leaf behaviors execute the algorithms in the form of C code.
In the example shown here, execution starts with leaf behavior *B1*, followed by the parallel composition of leaf behaviors *B2* and *B3*. *B1* contains a local variable *v1* to store state information. *B1* then produces variable *v2*, which is consumed by both *B2* and *B3*, and variable *v3*, which is read by *B3*. In addition, the concurrent behaviors *B2* and *B3* exchange data and synchronize through channel *c2*.



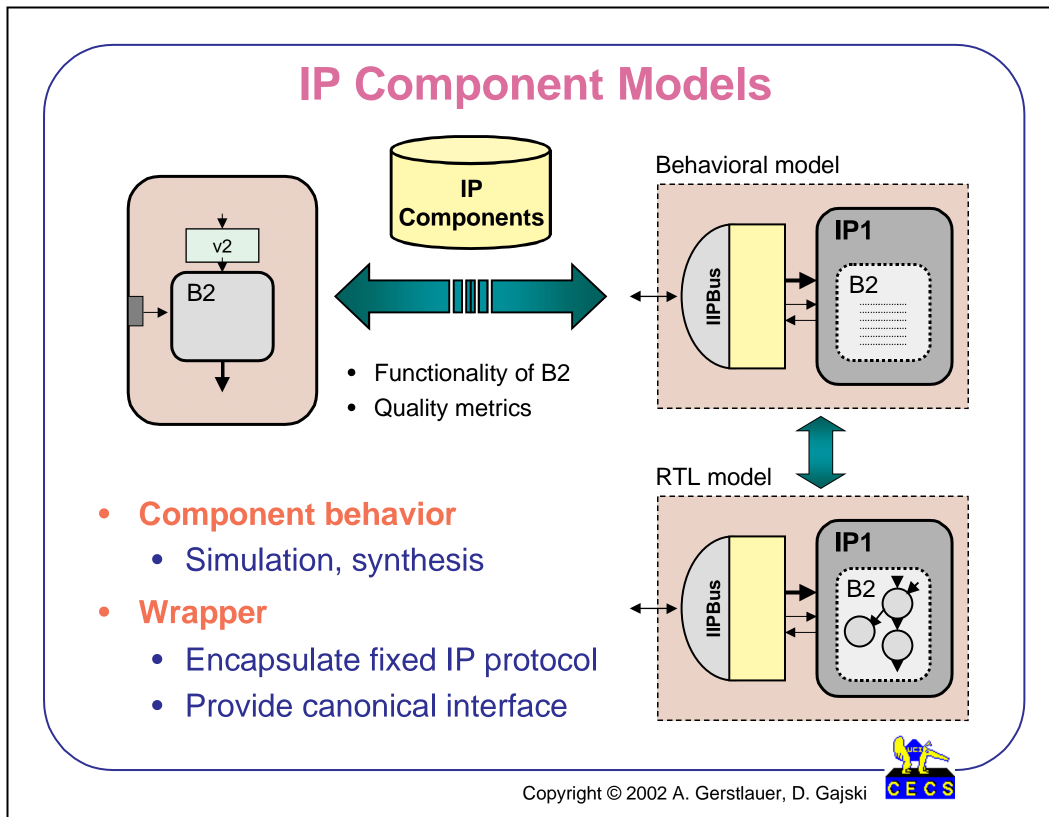
The architecture model is an intermediate model of the system design process. As a result of architecture exploration, computation has been mapped onto processing elements of a system architecture.

The architecture model reflects the component structure of the system architecture. At the top-level of the behavior hierarchy, the design is a set of concurrent, non-terminating component behaviors. However, communication is still on an abstract level and components communicate via message-passing channels. The communication synthesis task that follows will implement the abstract communication over busses with real protocols.

The behaviors grouped under the components according to the selected mapping specify the desired functionality for the implementation of the component during later stages.

Concurrency is limited to the top-level of the design in the architecture model. All the concurrency in the design at this point is captured by the set of components running in parallel. Inside each component, behaviors execute sequentially in a certain order. In the system architecture, true concurrency can only be exploited by executing computation in parallel on different components.

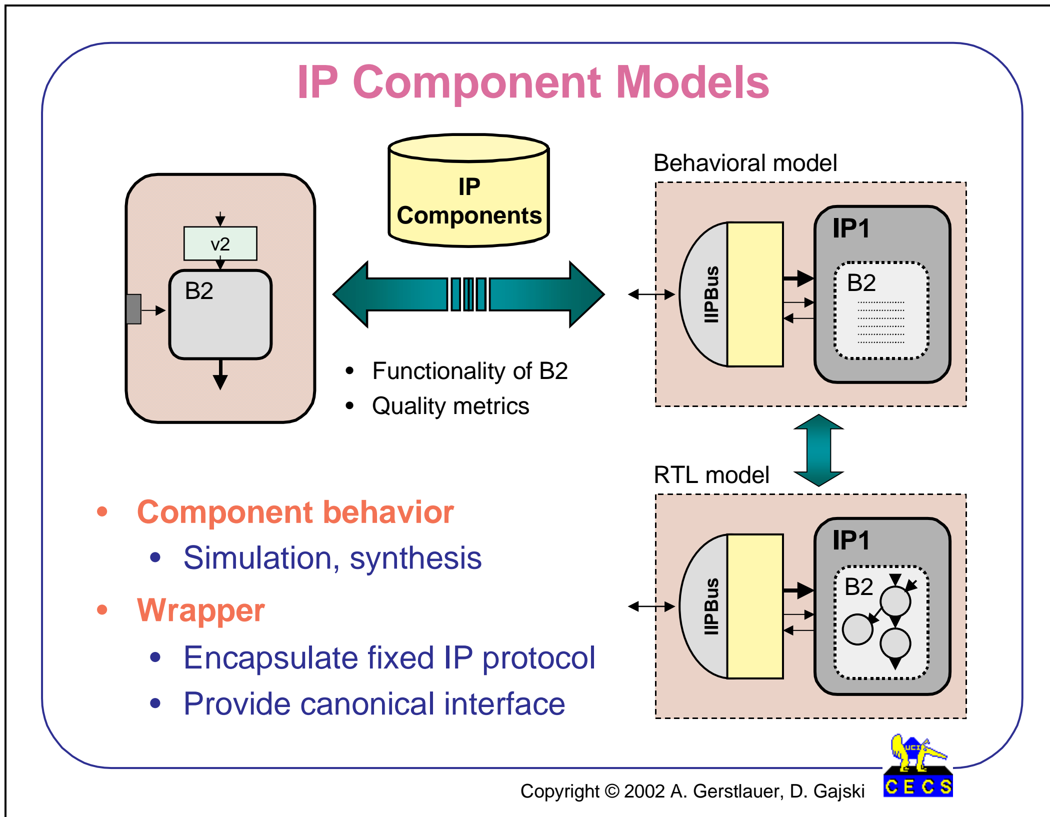
The architecture model is timed in terms of the computational parts of the design. Behaviors are annotated with estimated execution delays for simulation feedback, verification and further synthesis.



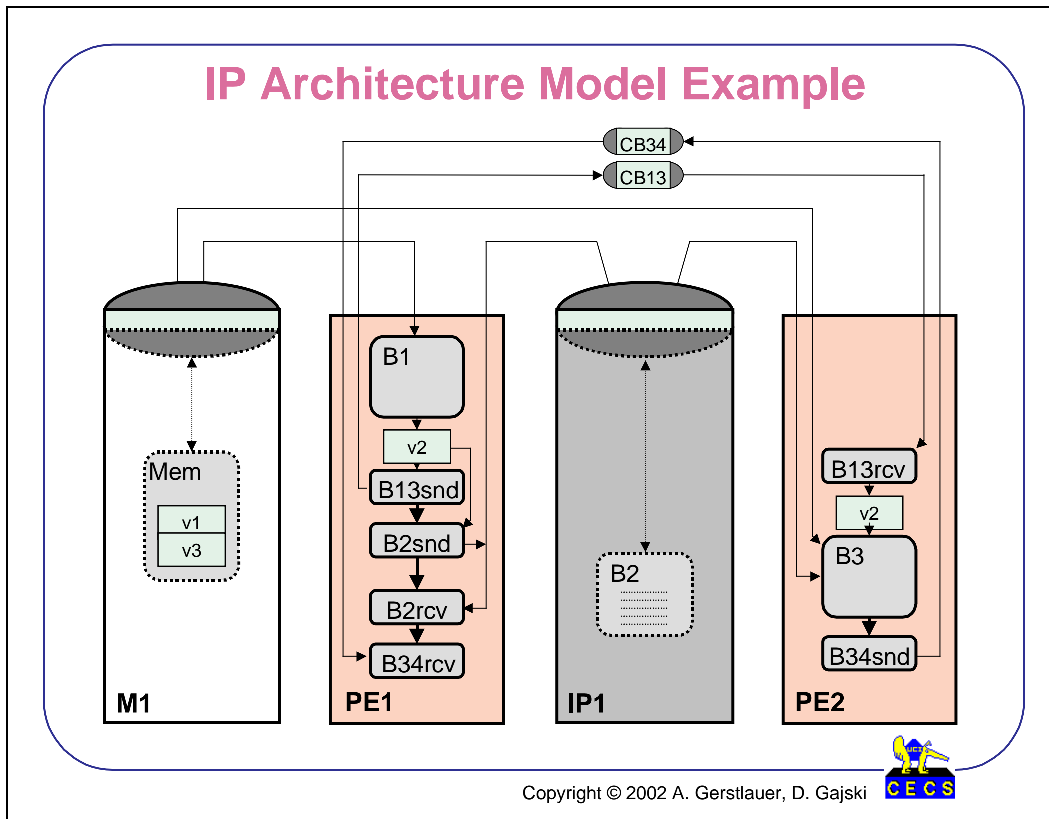
As part of PE allocation during architecture synthesis, Intellectual Property (IP) components can become part of the architecture model. Due to the specific characteristics of IP components, they have to be modeled in a special manner, different from other, synthesizable components.

IP components are predesigned components with predefined functionality. As such, their external interfaces and communication protocols are fixed, too. Note that this may also include microprocessor cores with specific, fixed bus interfaces.

IP components are selected out of the component library during component allocation to implement a specific part of the system specification. Depending on quality and cost metrics like power, performance or area, a certain part of the specification can be mapped onto an IP component during partitioning instead of implementing it on a general-purpose PE. For example, under the assumption that the component library contains an IP that implements the functionality of behavior *B2*, the behavior will be mapped onto and replaced with an instance *IP1* of the corresponding IP in the architecture model.

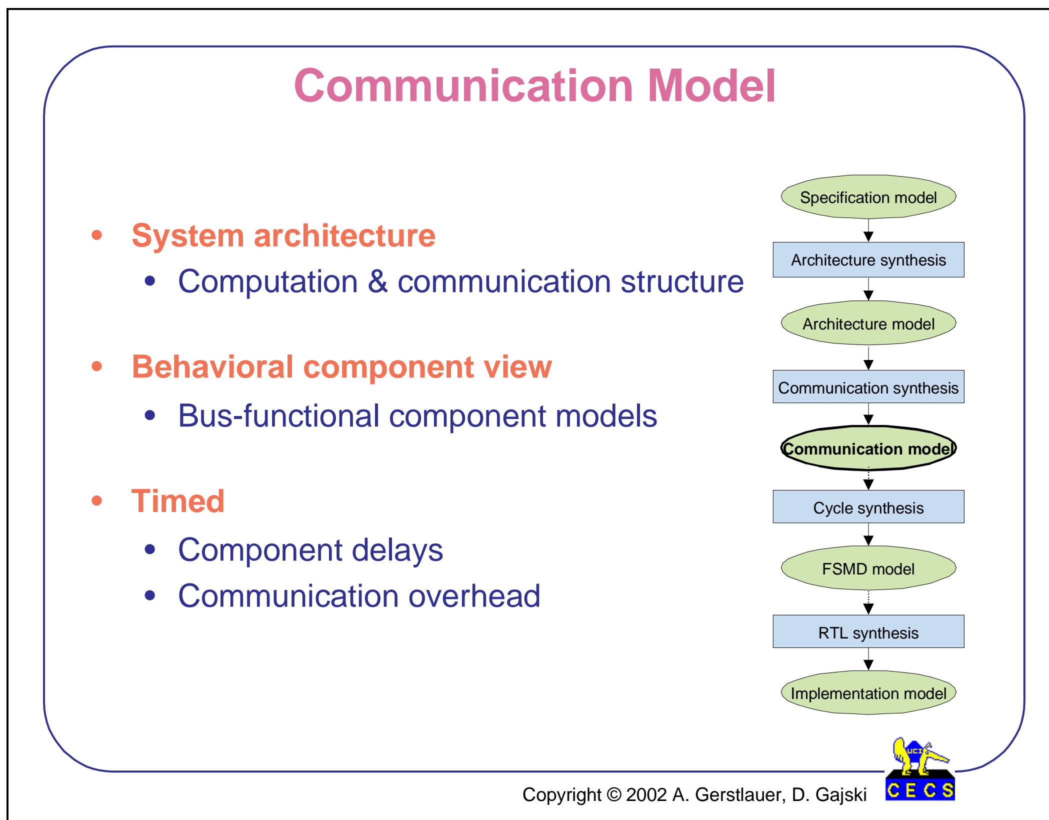


Each IP component model is stored in the component library as a combination of a IP behavior and a protocol wrapper.
The IP component behavior is a bus-functional model of the actual IP for simulation and/or synthesis. It provides a description of the fixed IP functionality at the ports of the component. Internally, it models the behavior of the IP and drives the set of ports according to the fixed IP protocol. The library can contain bus-functional models for IP component behaviors at different levels of abstraction. Depending on the desired levels of accuracy and complexity, models can range from purely functional descriptions of the IP behavior down to cycle-accurate RTL models.
The IP wrapper is a channel that encapsulates a description of the IP protocol and provides a canonical, abstract interface to the IP functionality. A wrapper implements the abstract semantics needed for communication in the architecture model by driving and sampling the IP ports according to the IP protocol.

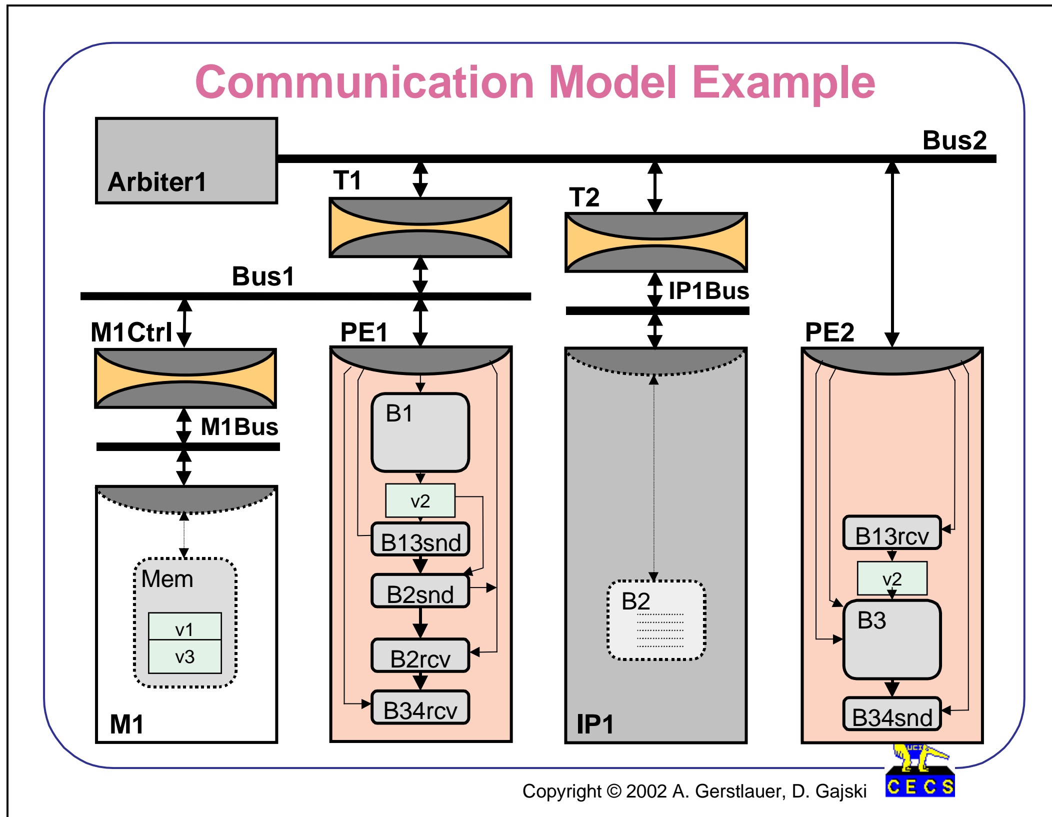


Copyright © 2002 A. Gerstlauer, D. Gajski

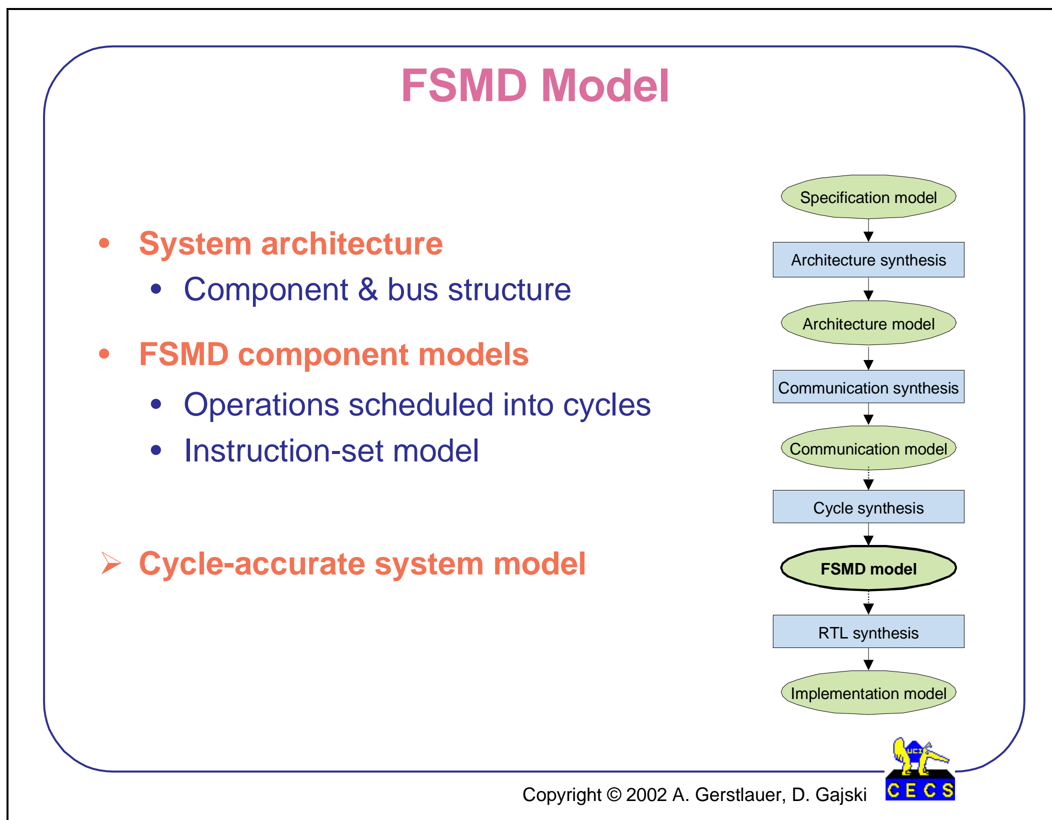
The figure shows the architecture model for our example. It demonstrates the complexity of the architecture model. In this case, behaviors *B1* and *B3* are mapped onto processing elements *PE1* and *PE2*, respectively. *B2* is implemented by an existing IP component that provides the same functionality. A vendor-supplied description of *IP1* encapsulates a model of the IP while allowing integration into the system through a channel interface. A system memory *M1* holds variables *v1* and *v3* and provides read and write access through its channel interface. On the other hand, local copies of the channel *c2*, degenerated to a simple variable *v2* after serialization, have been created in *PE1* and *PE2*.
In addition, communication and synchronization blocks *BXXsnd* and *BXXrcv* have been inserted to preserve the original execution order. Execution of formerly sequential blocks mapped to concurrent PEs is synchronized, and updated variable values are communicated to keep local copies in sync. Finally, behavioral blocks inside *PE1* and *PE2* communicate via global channels *CBxx* or by accessing the channel interfaces of *M1* and *IP1* directly.
The behavior pair *B13snd* and *B13rcv* ensures that *B3* on *PE2* doesn't start executing until *B1* on *PE1* is finished. In addition, variable *v2* produced by *B1* is transferred to *B3* through these behaviors and channel *CB13*. Behavior *B2snd* and *B2rcv* replace the original behavior *B2* with corresponding communication with *IP1*. Finally, the pair *B34snd*/*B34rcv* is added so that the next cycle on *PE1* won't start until the whole execution sequence including *B3* on *PE2* is finished.



The communication model is the final result of the system synthesis process and as such defines the structure of the system architecture in terms of both components and connections. Computation has been mapped onto components and communication onto busses.
At the top-level of the hierarchy, the communication model is a parallel composition of a set of non-terminating components communicating via a set of system busses.
Inside the components, a sequence of behaviors describes their functionality. The behaviors also define the timing of bus transactions as determined by the communication calls executed by the code. The bus adapter channels inside the components, in turn, define the timing-accurate implementation of each transaction over the bus wires.
At their interfaces, the components therefore provide a timing-accurate model of the component functionality down to the level of events on the bus wires. As a result, the communication model is timed in terms of both computation and communication.



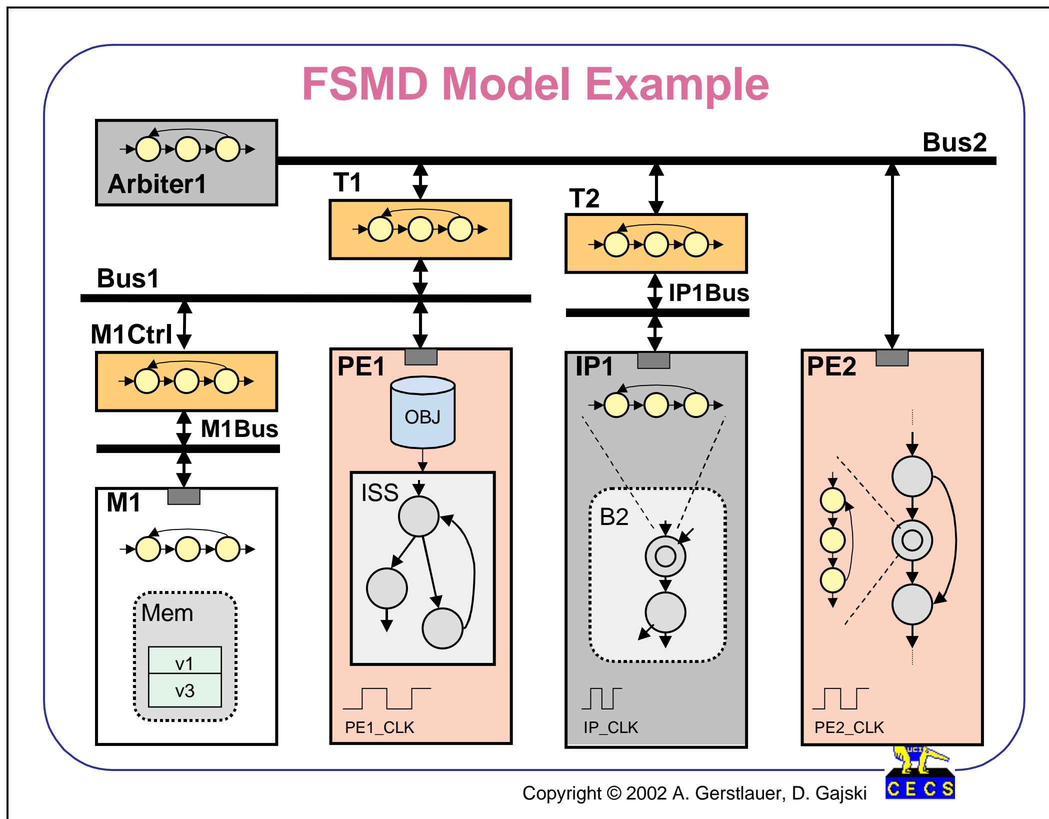
The figure shows the communication model for our example. In the figure, the memory *M1* is connected to processor *PE1* via the processor's bus *Bus1*. A memory controller *M1Ctrl* translates requests on the processor bus into transactions on the memory bus *M1Bus*.
IP1 and co-processor *PE2* communicate via *Bus2*. A bus bridge *T1* connects the two main system busses. *PE2* is a synthesizable component that implements the *Bus2* protocol directly. The IP component, on the other hand, is connected to the bus through a protocol transducer *T2* that translates between the bus protocol and the proprietary IP protocol on the IP bus *IP1Bus*.
Inside *PE1* and *PE2*, behavioral blocks connect to bus driver channels that implement message-passing over the bus wires. The channel interface of *IP1* in the architecture model has been moved into *T2* where it implements communication with *IP1* over the exposed wires of the IP bus.
Finally, an additional PE, the arbiter *Arbiter1*, has been inserted as part of the communication model. The arbiter regulates conflicting bus accesses of bus masters *T1* and *PE2* on *Bus2* according to the bus' arbitration protocol.



The FSMD model is an intermediate model, which contains cycle-accurate descriptions for each PE.

At the top-level, the system architecture is a set of non-terminating, concurrent components communicating via system busses. At the component level, computation and communication functionality is described in the form of cycle-accurate state machines: FSMD models for custom hardware and instruction-set models for software on programmable processors.

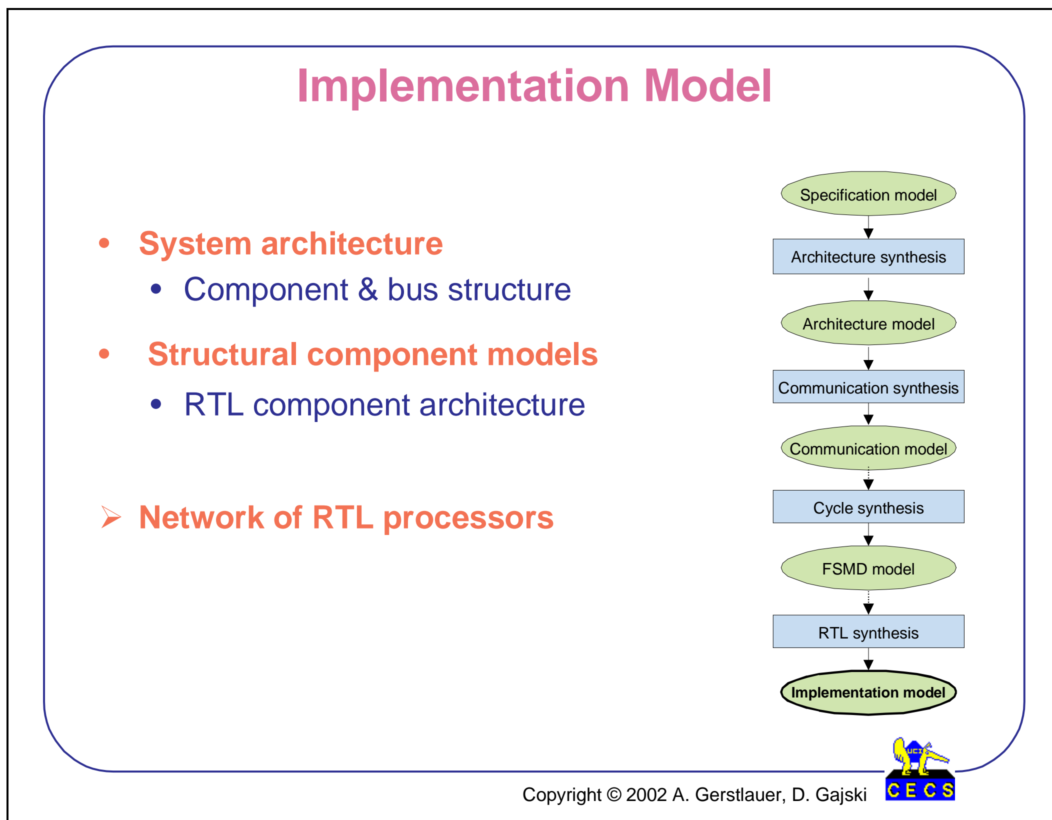
The FSMD model is a cycle-accurate system description. The order and timing of computation and computation in the system is described in terms of component clocks. A global order is imposed among the system's components via the order of events on the common bus wires.



The figure shows the FSMD model for our design example. Compared to the communication model shown previously, the architecture at the system level consisting of PEs connected to busses is unchanged. However, each PE is replaced with a cycle-accurate description based on the PE's individual clock. In each PE, states and transitions in the form of FSMDs model the cycle-based behavior of both computation and communication in the PE.

The programmable processor *PE1* is replaced with an instruction-set simulator (ISS) that executes the compiled object code. For the IP component *IP1*, a cycle-accurate model is pulled out of the library and plugged into the system. Similarly, the system memory *M1* is replaced with a cycle-accurate description. For the custom hardware component *PE2*, an FSMD is synthesized by scheduling the PE behavior into clock cycles. In all cases, both the behaviors and the bus driver channels from the communication model are transformed into cycle-accurate descriptions. Bus driver calls are either implemented as hierarchical (super-) states or through communicating state machines.

Finally, pure communication functionality inside bus bridges, transducers or arbiters is synthesized into bus interface FSMDs similar to the implementation of other custom hardware or IP components.

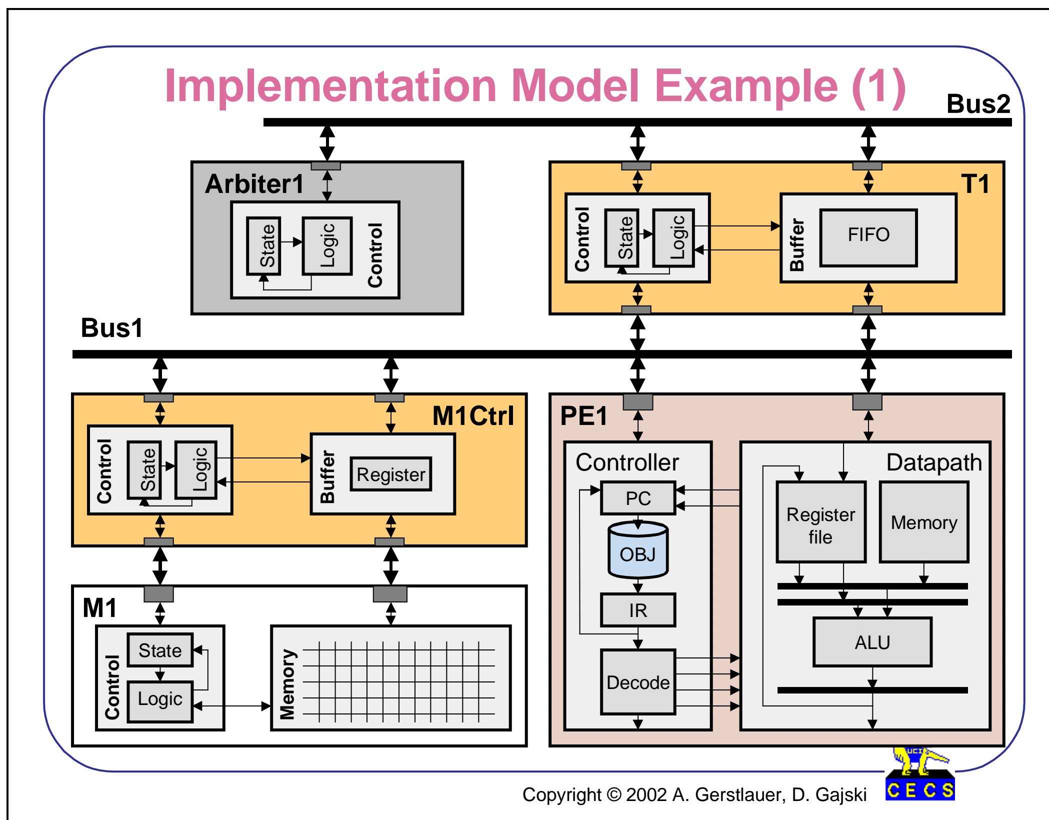


The implementation model is the result of the backend process and as such the final end-result of the whole system design flow. It is a structural description of the system down to the component RTL architectures.

Like the FSMD model, the system architecture at the top level is a set of non-terminating, concurrent components communicating via system busses. At the component level, however, computation and communication functionality in the implementation model is implicitly given as a description of the component RTL architectures. For each PE, a structural description of the PE's controller and datapath in the form of a netlist of RTL components is available.

Therefore, the implementation model is a netlist of RTL processors. As such, the implementation model is accurate down to sub-cycle delays.

Finally, the implementation model is then further processed and refined down to manufacturing through traditional design flows. For example, logic synthesis of custom hardware RTL descriptions is followed by physical design to generate the final chip layout.

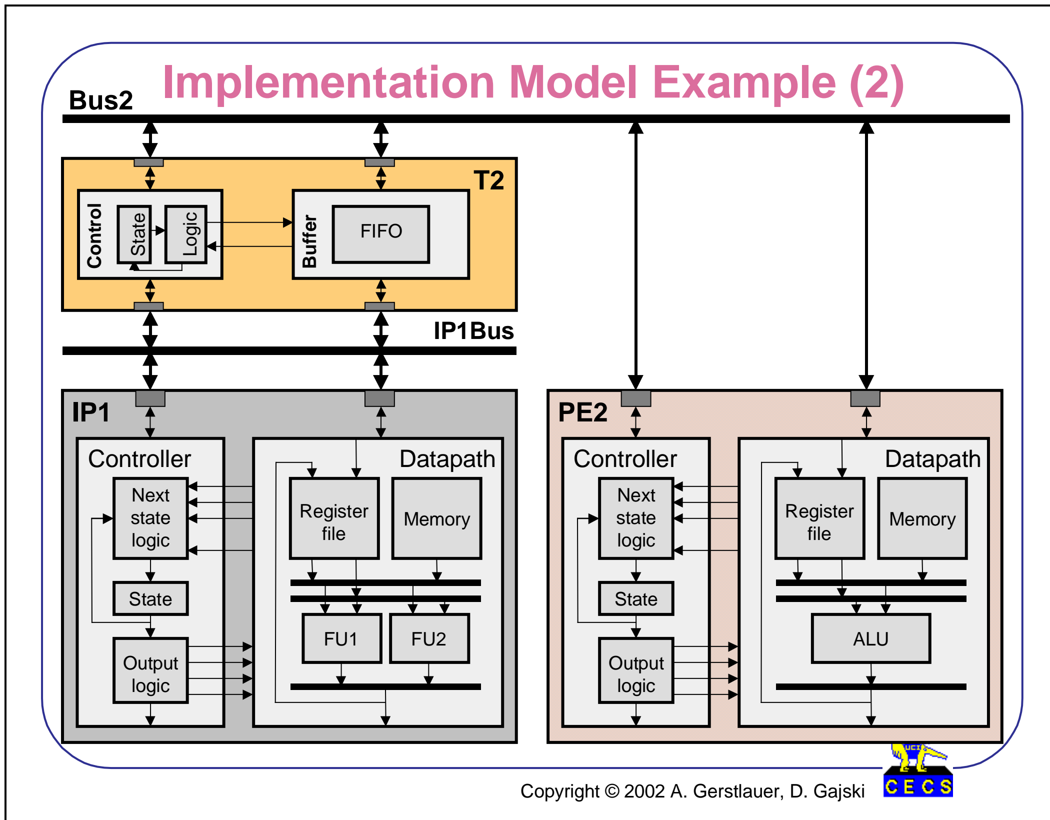


Copyright © 2002 A. Gerstlauer, D. Gajski

The final implementation model for our design example is shown here. In general, each PE from the FSMD model is replaced with an RTL processor that implements the PE's state machine. An RTL processor is a structural description of the PE's controller and datapath architecture in the form of a netlist of RTL components.

For the programmable processor *PE1*, the (possibly pipelined) datapath contains the typical register files, memories, functional units and busses. The controller of the programmable processor, on the other hand, contains the necessary program counters, program memory, instruction register and instruction decoder of the control pipeline.

The memory *M1* is mainly a description of the memory cell array with its simple controller. The bus arbiter *Arbiter1* contains only a controller implementing the bus arbitration protocol. Furthermore, memory controllers *M1Ctrl*, bus bridges *T1* and bus transducers *T2* contain controllers implementing bus protocol translations and datapaths with buffers to temporarily hold data transferred between busses.



In case of the custom hardware component *PE2*, a custom controller and custom datapath is synthesized. In general, the custom datapath will contain registers, memories and functional units connected through busses. The custom controller will contain the state register, next state and output logic required to implement the state machine of the FSMD model.

Finally, for the IP component *IP1*, depending on the type of IP (soft, firm, or hard) such a custom RTL processor netlist for the implementation model is either synthesized from the FSMD IP model or pulled out of the IP library directly.

Summary & Conclusions

- **Accomplishments**
  - Finalized model definitions & model refinements
  - Verified design flow consistency
  - Checked compatibility with existing backend flows
  - Tested on three industrial-strength examples:
    - Vocoder
    - JPEG
    - JBIG
- **Developed SC environment**
- **Future work**
  - Formal verification
  - Model style checker
  - Connection to EDA tools

Copyright © 2002 A. Gerstlauer, D. Gajski

In summary, we have shown a set of well-defined system-level models covering the flow from specification to RTL implementation. The definition of models is based on a separation of concerns that minimizes interactions between levels, simplifies refinement between models, and allows easy exploration with a variety of components and IPs. The resulting design flow supports rapid design space exploration allowing critical decisions at early stages while providing quick feedback.

The models define a framework on top of which system-level languages and design methodologies can be developed. For example, platform based design predefines the sets of PEs and busses supported by architecture and communication models. The formalization of models is the enabler for interoperability and design automation. We verified the feasibility of the models and the design flow through several industrial-strength design examples.

Based on the abstract definitions, we can demonstrate automatic model refinement between levels. We have developed the corresponding refinement tools and integrated them into a design environment.

In the future, we want to extend the formalization to a general algebra on which proof ably correct transformations can be defined. Such a formalized framework of models and transformations based on the definitions presented in this report is the foundation for the vertical integration of models through synthesis and verification.

Chapter 3

Design of a GSM Vocoder

Design of a GSM Vocoder

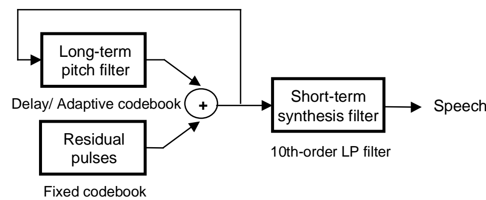
Andreas Gerstlauer

Center for Embedded Computer Systems
University of California, Irvine
http://www.cecs.uci.edu/~specc/

Copyright © 2002 A. Gerstlauer

GSM Vocoder

- Vocoder standard (source: ETSI)
  - Speech synthesis model



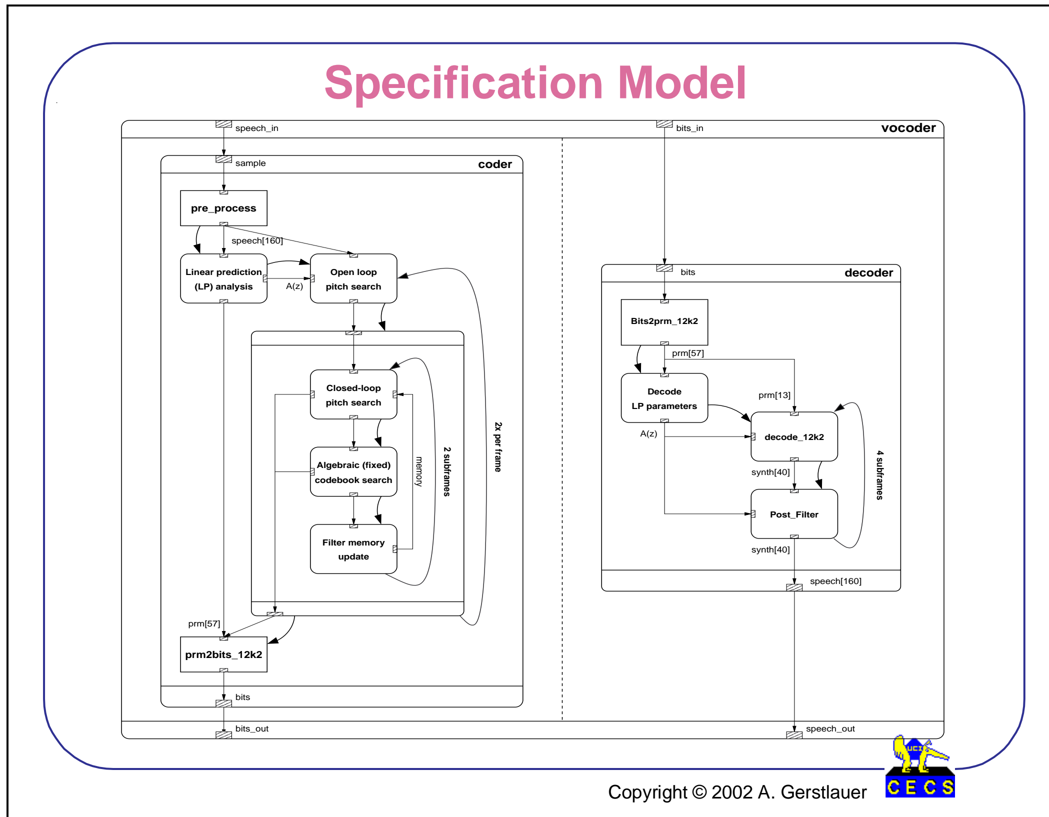
    - Frames of 4 x 40 = 160 samples (4 x 5ms = 20ms of speech)
    - Transmit model parameters (244 bits / frame)
  - Transcoding constraint (back-to-back encoder/decoder)
    - First subframe: < 10ms
    - Whole frame (4 subframes): < 20 ms

Copyright © 2002 A. Gerstlauer

The CELP voice-encoding scheme is based on a speech synthesis model, which tries to emulate the way in which speech is generated in the human vocal tract. The combination of the output of a long term pitch filter (also called adaptive codebook) and a set of residual pulses out of a fixed codebook models the buzz produced by the human vocal chords. This excitation is then fed into a short-term, *linear prediction* (LP) synthesis filter (linear, weighted sum of the past 10 inputs) that models the modulation occurring in the human throat and mouth as a system of lossless tubes.

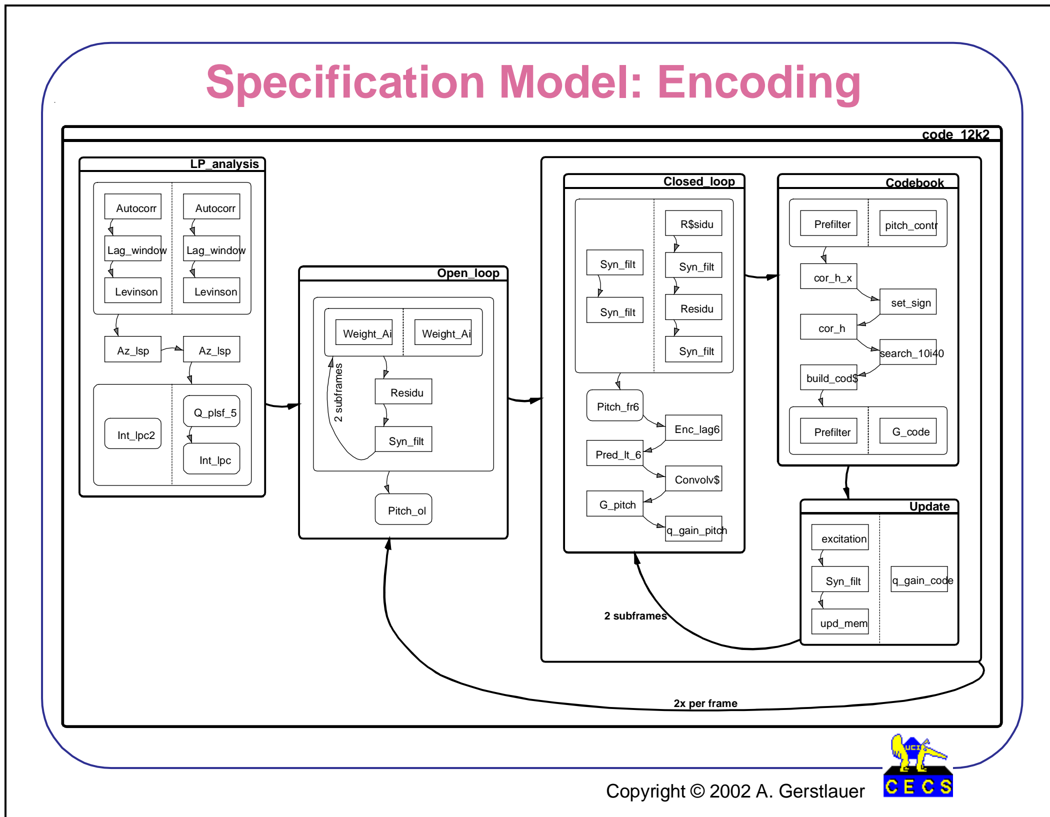
Instead of transmitting compressed speech samples, the filter parameters of the speech synthesis model are extracted in the encoder, transmitted, and used for driving the synthesis of speech in the decoder. In the encoder, parameters are extracted via an analysis-through-synthesis approach such that the mean-square error between synthesized and original speech is minimized. Analysis and synthesis operate on speech frames of 160 samples corresponding to 20 ms of speech. Each frame is subdivided into 5 subframes of 40 samples (5 ms of speech) each, and one set of parameters is transmitted per subframe.

The Vocoder standard specifies a maximal total latency of 10 ms for the first subframe when operating encoder and decoder in back-to-back mode. In addition, a complete frame has to be encoded and decoded within the 20 ms before the next frame arrives.

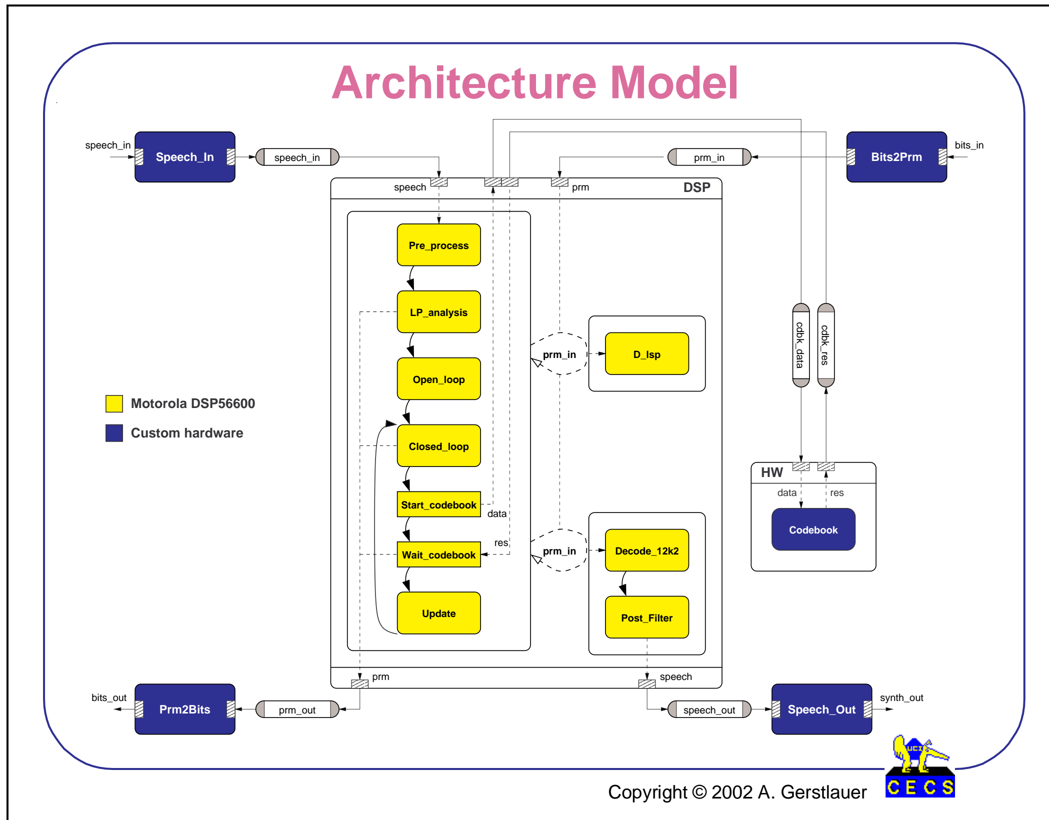


The original vocoder standard published by the European Telecommunication Standards Institute (ETSI) contains a bit-exact reference implementation of the standard in C. This reference code was taken as the basis for developing the SpecC specification model. At the lowest level, the C algorithms were directly reused by encapsulating them in SpecC leaf behaviors. However, the C function hierarchy had to be converted into a clean and efficient SpecC hierarchy by analyzing dependencies, exposing available parallelism, grouping related parts hierarchically, and so on. In contrast to the original C code, the SpecC specification describes the vocoder functionality in a clear and concise manner, which greatly eases understanding for both the user and any automated tools.

At the top level, the specification model runs encoding (left) and decoding (right) behaviors in parallel. The encoder preprocesses and frames incoming speech before analyzing the speech frames in two nested loops. LP and initial, open-loop pitch filter parameters are extracted once for every two subframes. Final, closed-loop pitch parameters and the fixed codebook vector are computed for every subframe. The decoder, on the other hand, decodes the incoming speech synthesis parameters and synthesizes speech frames following the synthesis model in a loop over all subframes.



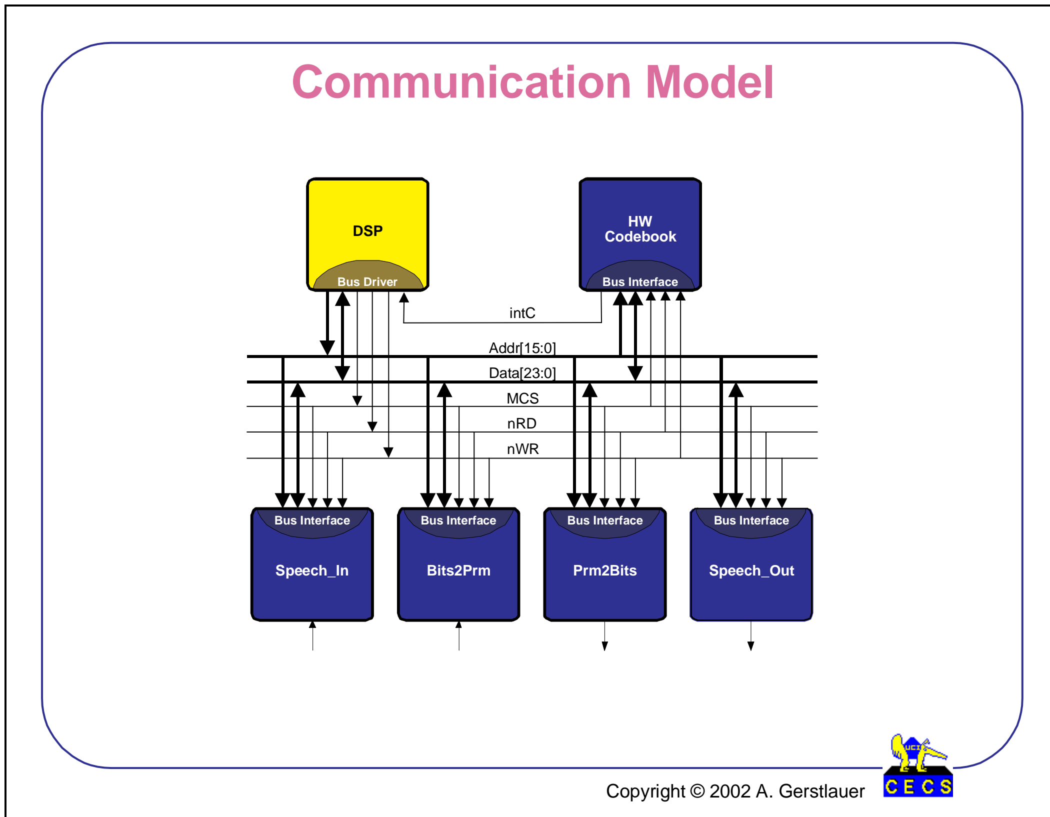
The model shown on the previous page only depicts the top levels of the hierarchy. As an example of the overall complexity of the vocoder specification, this figure shows the complete hierarchy of the encoding part of the vocoder down to the leaf behaviors.
In general, at each level of the hierarchy, the specification model hides unnecessary details while focussing the designer and the tools onto the important aspects at hand.
In total, the specification model of the vocoder contains 43 leaf behaviors and consists of 13,000 lines of code.



The architecture model is shown here. At the center of the architecture is the DSP56600 digital signal processor (yellow). The *DSP* runs encoding (left) and decoding (tasks) concurrently in a dynamic scheduling approach. The main loop of the application is formed by the encoding task reading incoming speech samples, processing them, and producing the encoded bit stream at the output. However, encoding is interrupted whenever a new packet arrives at the decoding side. Depending on the state of the decoding process, the corresponding decoding stage is executed and once the decoder has finished processing the incoming packet control returns to the encoder.

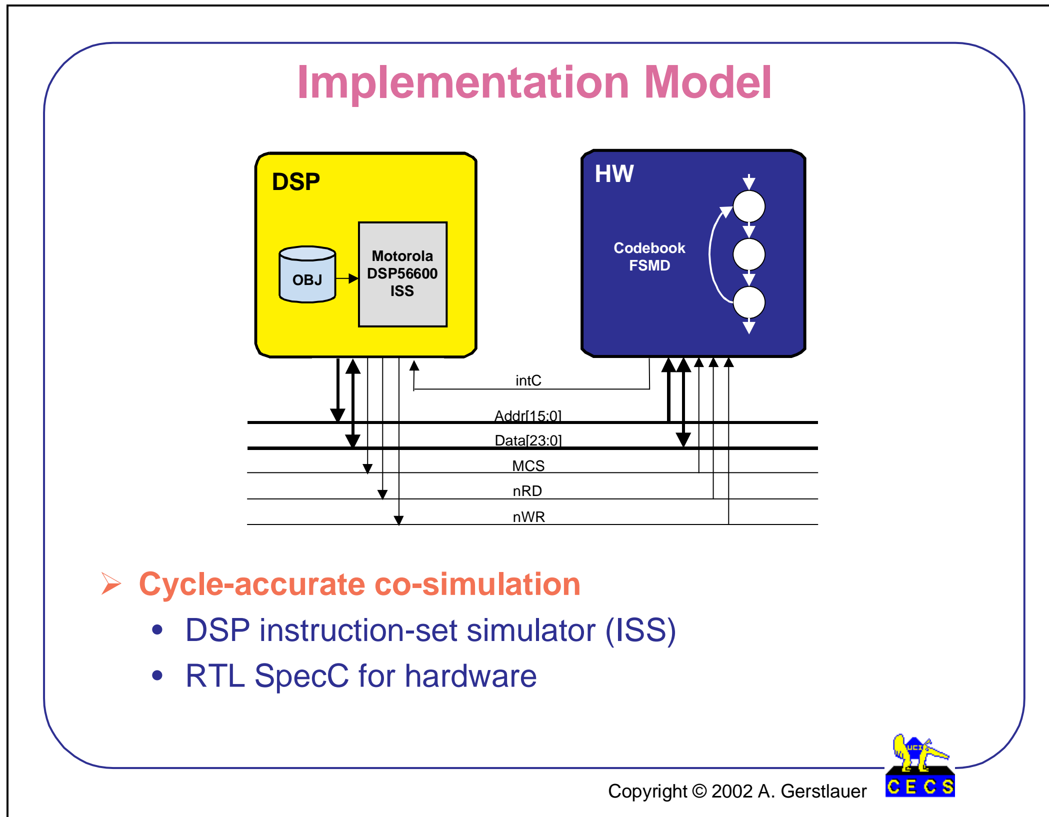
The encoding task on the *DSP* is supported by the codebook search custom *HW* component (blue, middle right). The encoder communicates with the codebook *HW* via message-passing channels in order to send data into the co-processor for processing and to receive the corresponding results.

Finally, the *DSP* is surrounded by four peripheral custom hardware components that handle I/O with the environment, pre-process incoming speech or bit streams, and perform the necessary framing. Similar to the codebook *HW*, the *DSP* communicates with the peripheral HW via message-passing channels, sending and receiving encoded/decoded bit and speech frames.



The communication model as the result of the communication synthesis process is shown here. The five components are connected by and communicate via the *addess*, *data,* chip select (*MCS*), read (*nRD*) and write (*nWR*) wires of the bus. The *DSP56600* processor (yellow) is the master on the bus. The *codebook HW* co-processor (blue, top right) and the four peripheral hardware components (blue, bottom) are bus slaves, listening for transfers with matching addresses on the bus. In addition, hardware components can signal the DSP by raising interrupts in the processor, as exemplified by the connection from the codebook HW to the DSP's *intC* interrupt line.

Internally, the component's behaviors are unmodified from the architecture model. However, the bus drivers and bus interfaces generated during communication synthesis are inserted into the components as adapter channels interfacing the component's behaviors to the bus.



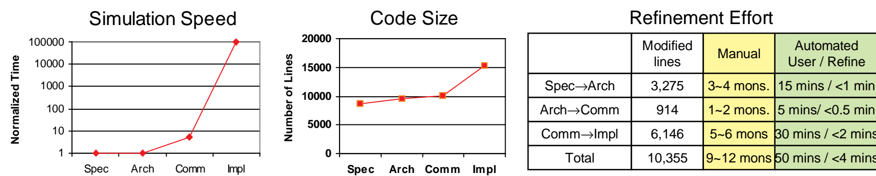
The finalimplementation model of the vocoder design is shown here. The implementation model performs a cycle-accurate co-simulation of hardware and software components communicating via bus wires.

The model for the DSP component runs an instruction-set simulation of the assembly code of the software generated in the backend, driving and sampling the bus wires according the the simulator's outputs and inputs. In case of the vocoder, the instruction-set simulator (ISS) supplied by Motorola for their DSP56600 processor family was hooked into the SpecC simulation via the ISS' C-level API.

The hardware component models are refined into FSMD models of the custom hardware RTL design. The state machines in the hardware execute the functionality and access the bus wires in a cycle-accurate manner. The FSMD descriptions of the scheduled hardware are modeled in the form of SpecC code, which plugs directly into the co-simulation.

Results and Conclusions

- Experiment on GSM Vocoder design



	Modified lines	Manual	Automated User / Refine
Spec→Arch	3,275	3~4 mons.	15 mins / <1 min
Arch→Comm	914	1~2 mons.	5 mins/ <0.5 min
Comm→Impl	6,146	5~6 mons	30 mins / <2 mins
Total	10,355	9~12 mons	50 mins / <4 mins

- Conclusion
  - Productivity gain >2,000X for industrial strength designs
  - Enables extensive design exploration (60/day)

Copyright © 2002 A. Gerstlauer, J. Peng

Results for the different vocoder models are shown here, from system specification model to implementation model. The tables list the time needed for the simulation, the number of lines of code for each model and the refinement effort.
To validate the models, we performed simulations at all levels. The simulation performance at different levels for the vocoder are shown in the graph. As we move down in the level of abstraction, more timing information is added, increasing the accuracy of the simulation results. However, simulation time increases exponentially with lower levels of abstraction. As the results show, moving to higher levels of abstraction enables more rapid design space exploration. Through the intermediate models, valuable feedback about critical computation synthesis aspects can be obtained early and quickly.
As the number of lines of code for different models suggests, more lines of code are added to the model with lower level of abstraction. Reflecting the additional complexity needed to model the implementation detail introduced with each step.
The refinement effort table demonstrates that by using the automated system level refinement process, large productivity gains of 2000x or more can be achieved for the vocoder project.

Chapter 4

System Level Refinement

System Level Refinement

Junyu Peng
HaoBo Yu
Dongwan Shin
Daniel Gajski

Center for Embedded Computer Systems
University of California, Irvine
http://www.cecs.uci.edu/~specc

Copyright © 2002 J. Peng, H. Yu, D. Shin, D. Gajski

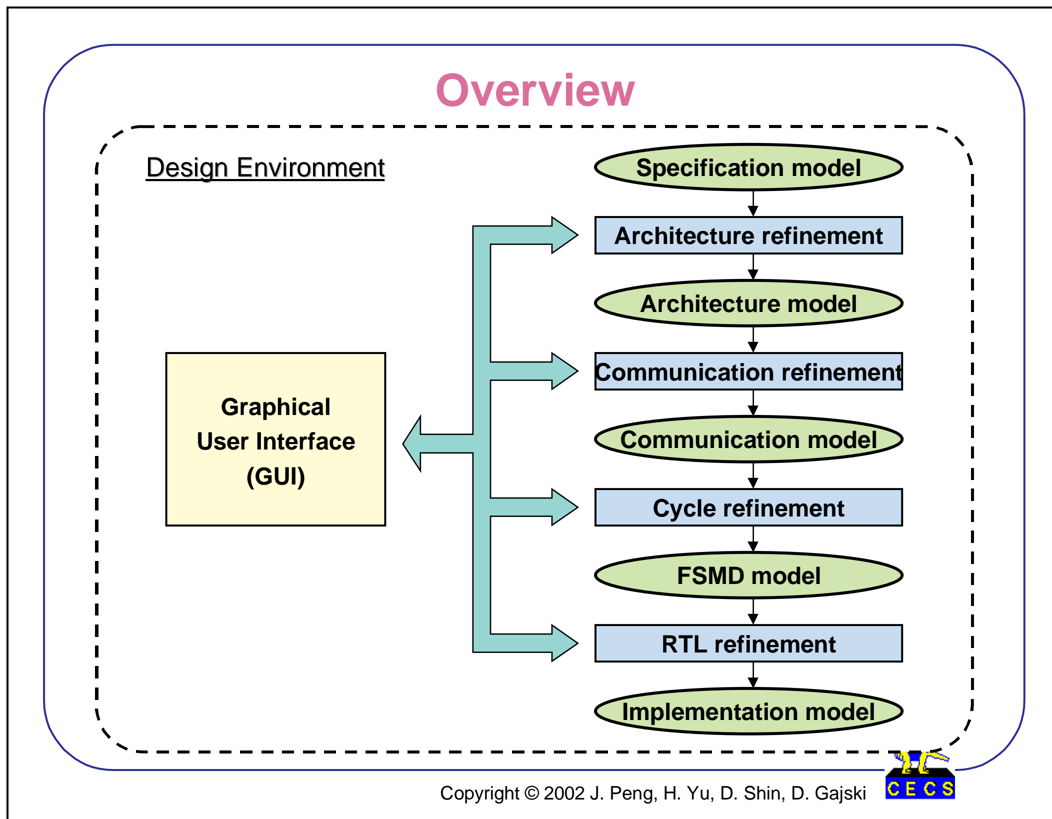
Motivation and Goals

- Model refinement is a tedious, error-prone and time-consuming work
- Frequently, many iterations are needed
- Well-defined refinement saves designer time, thus increasing productivity

Copyright © 2002 J. Peng, H. Yu, D. Shin, D. Gajski

In order to handle the ever-increasing complexity of System-on-Chip (SoC) design, system methodologies usually divide the design process into a number of steps. Accordingly, a number of models are employed to represent the design at each of these steps, in which, designers make decisions by evaluating several options and selecting the best one. Each decision results in a new detail being added to the design. Obviously, each new detail requires modification or refinement of the corresponding model. Refinement means deleting, adding and rewriting portions of the model, which is an elaborate but mundane task that designers try to avoid. To make things worse, model refinement must be repeated for each design option during the design exploration phase.

However, if the models and refinement are well-defined, the tedious and mechanical refinement task can be automated. With this kind of automation, the exploration and the design flow become productive and fun.

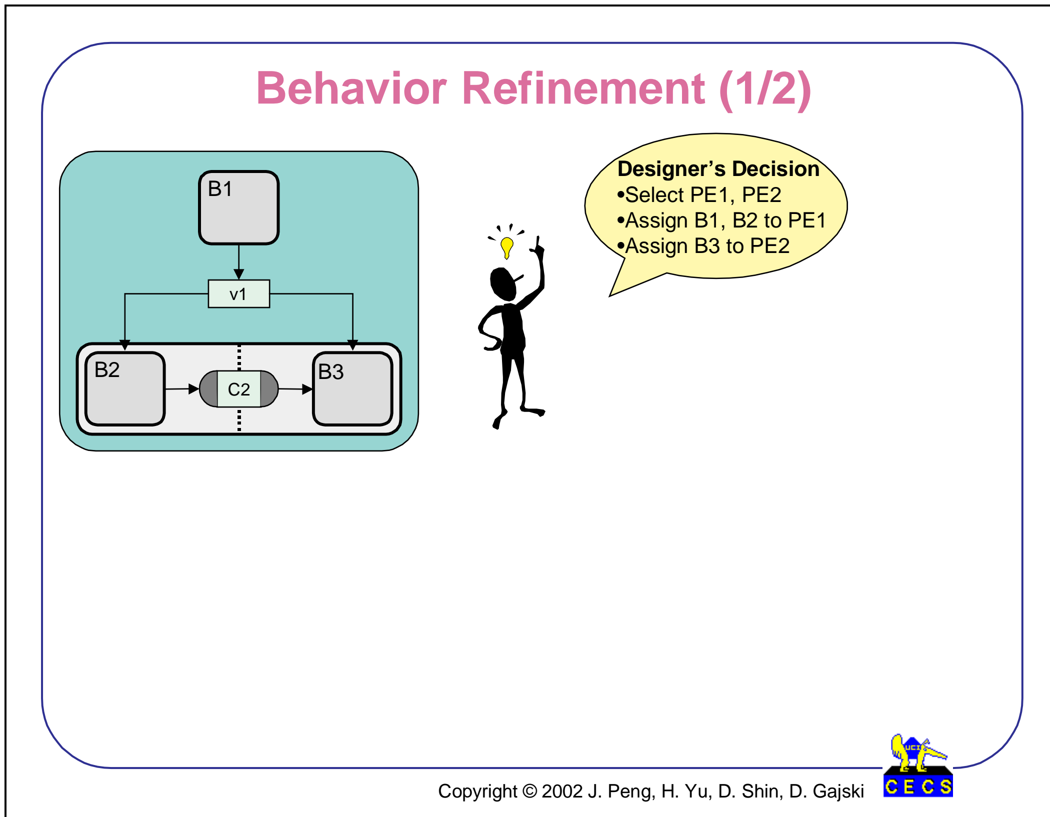


In our Design Environment, the design flow (on the right side) is facilitated with a Graphical User Interface (GUI).

The design flow consists of 5 levels of models (ovals), specification, architecture, communication, FSMD and implementation models, and 4 synthesis steps (boxes), architecture, communication, cycle and RTL syntheses. Note that each synthesis step itself can be iterated, which is not shown in the figure for the sake of clarity.

The GUI provides a bi-directional interface between designers and the design flow. At each design step, the GUI displays relevant information of the design, such as profiling data or estimated metrics, for designers to make design decisions. Then, designers' decisions are passed back into refinement tools through the GUI.

In the following sections, we will explain our design flow, in particular the models and refinements, by going through a simple example. At the end, we will show some experiment results on a couple of industrial designs.

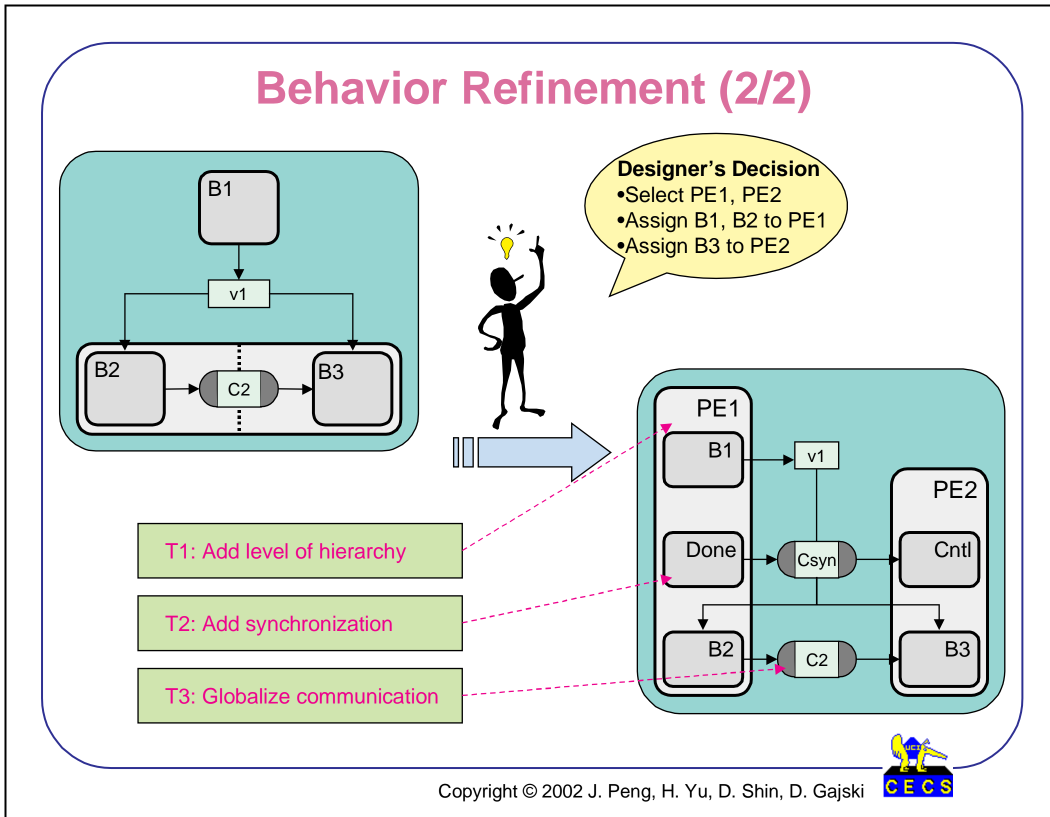


Our example specification model is shown on the left side. The execution starts with leaf behavior *B1*, followed by the parallel composition of leaf behavior *B2* and *B3*. *B1* produces variable *v1*, which is then consumed by both *B2* and *B3*. In addition, the concurrent behaviors *B2* and *B3* communicate through channel *C2*.

As we can see, the specification model describes the overall functionality of the system without any implementation details. It is also untimed because initially we do not know on which processing elements (processors, ASICs, IPs…) each behavior is executed. The specification model is served as the input model in our design flow.

An early design decision in the first synthesis step, architecture synthesis, is to select processing elements and assign behaviors to the processing elements for execution. As we mentioned before, in our design environment, this decision is made by the designers with the help of the GUI.

For our example, since behavior *B2* and *B3* can be executed concurrently, we can exploit the parallelism by running them on two different processing elements. Therefore, we will select components *PE1* and *PE2*. Then behaviors *B1* and *B2* are assigned to *PE1* while *B3* to *PE2*.



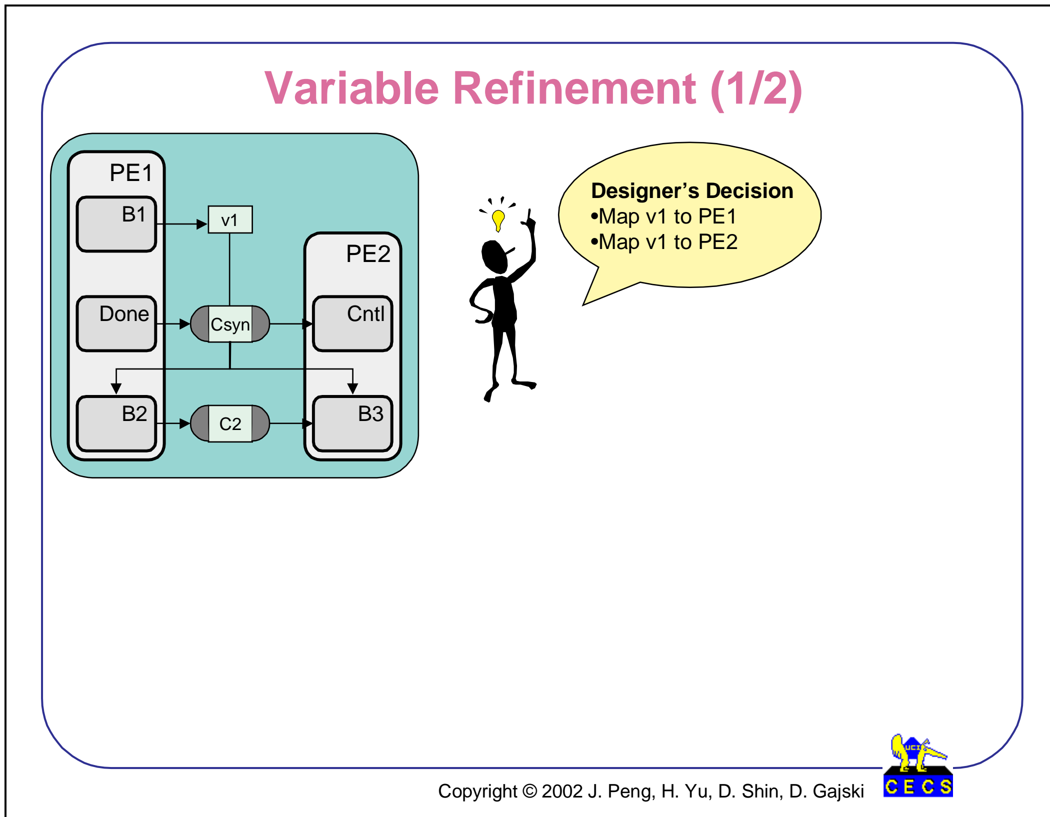
With the decisions made, a new model incorporating these decisions can be derived and shown on the right side.

If we compare these two models, similarities and differences can be easily identified between each other. Obviously, with three changes (*Behavior Refinement*), the specification model (left) can be refined into the desired new model (right) automatically.

First, an additional level of behavior hierarchy, *PE1* and *PE2*, is introduced to represent the selected components, which were not specified in the original specification model. *PE1* is composed of *B1* and *B2* while *PE2* includes *B3* to reflect the behavior assignment decision.

Then synchronization, a pair of new behaviors *Done*, *Cntl* with channel *Csyn*, is inserted at appropriate points inside *PE1* and *PE2* to make sure that *B3* (on *PE2*) will not start until *B1* (on *PE1*) is finished, as implied by the original specification.

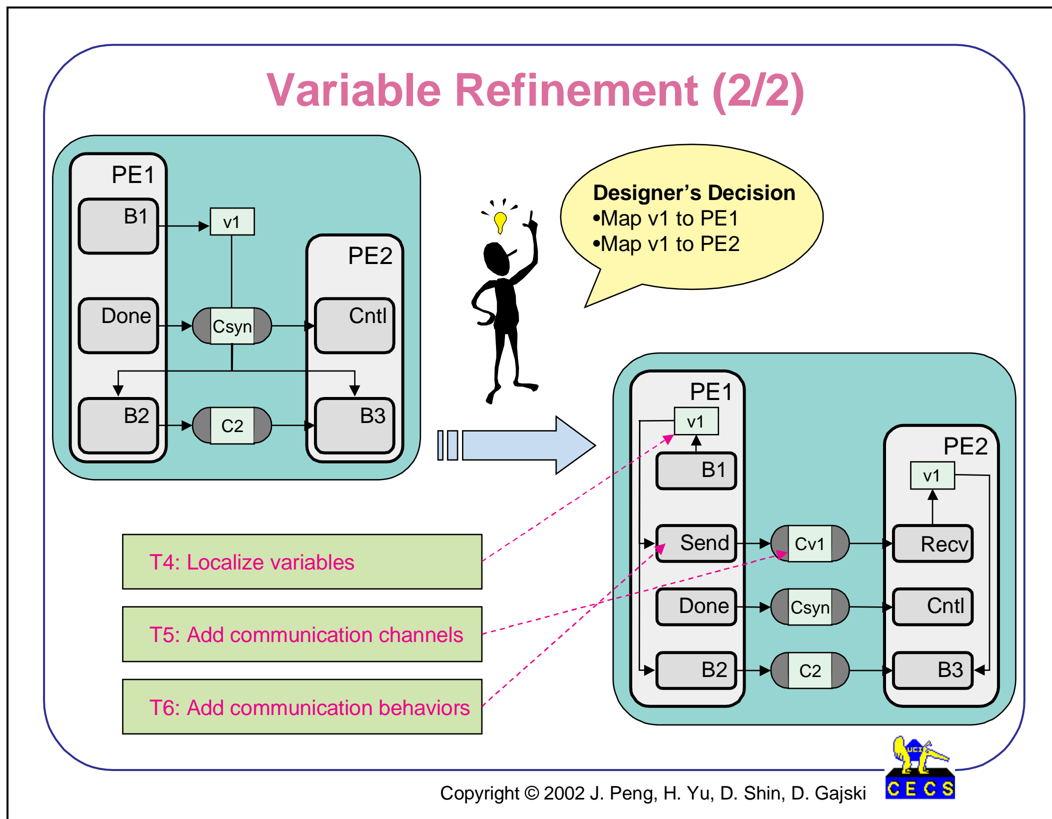
Finally, channel *C2* and variable *v1* are exposed as global (inter-component) communication since they are accessed by behaviors on both *PE1* and *PE2*.



Till this point, the communication among components is realized through global shared variables (*v1*) and global variable channels (*Csyn*, *C2*). However, the shared variables have to be assigned to actual memories in the system architecture and the global channels assigned to physical connections (ie., busses) among components. We will focus on shared variables in this section and discuss variable channels later in communication synthesis.

There are two ways to map a shared variable. First, it can be mapped to a dedicated shared memory component, allocated in addition to the processing elements. The shared memory approach tends to reduce the concurrence of the processing elements because the shared memory becomes the critical component in the system architecture. Alternatively, it can be mapped to the local memories of the processing elements (*PE1* and *PE2*). However, additional message-passing mechanism is needed to make all local copies of the same data consistent.

In our simple example, there is only one global shared variable *v1*. Behavior *B1* produces *v1*, which is consumed by *B2* and *B3*. Let us assume that the design exercises the second option to map *v1* into local memories of both *PE1* and *PE2*.

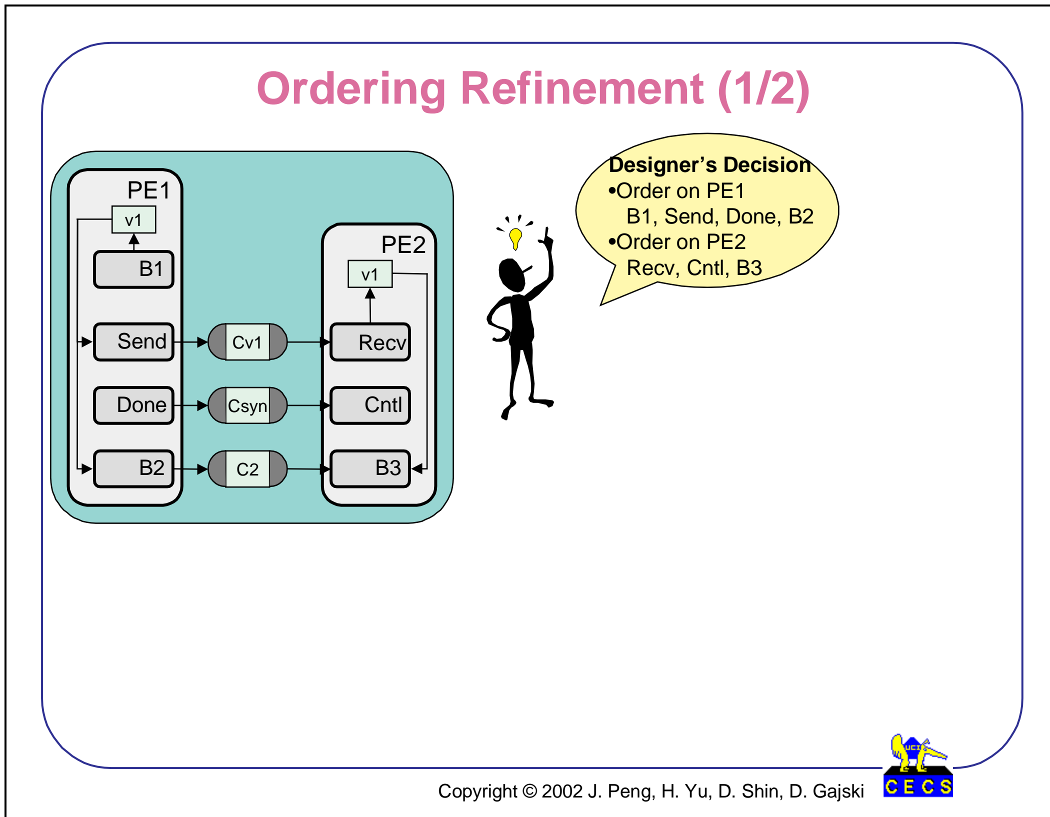


The resulted new model is shown on the right side. Compare with the input model on the left side, the new model has a few differences. These differences can be made by performing three refinements on the input model.

First, a local copy of *v1* is created in each of *PE1* and *PE2*. The behaviors inside each component use the corresponding local copy stored in local memory instead of accessing a global variable.

Then, a message-passing channel *Cv1* is created for transferring *v1* to *PE2*.

Finally, a pair of behaviors, *Send* and *Recv*, are added in components *PE1* and *PE2* respectively to transfer the value of *v1* from *PE1* to *PE2*.



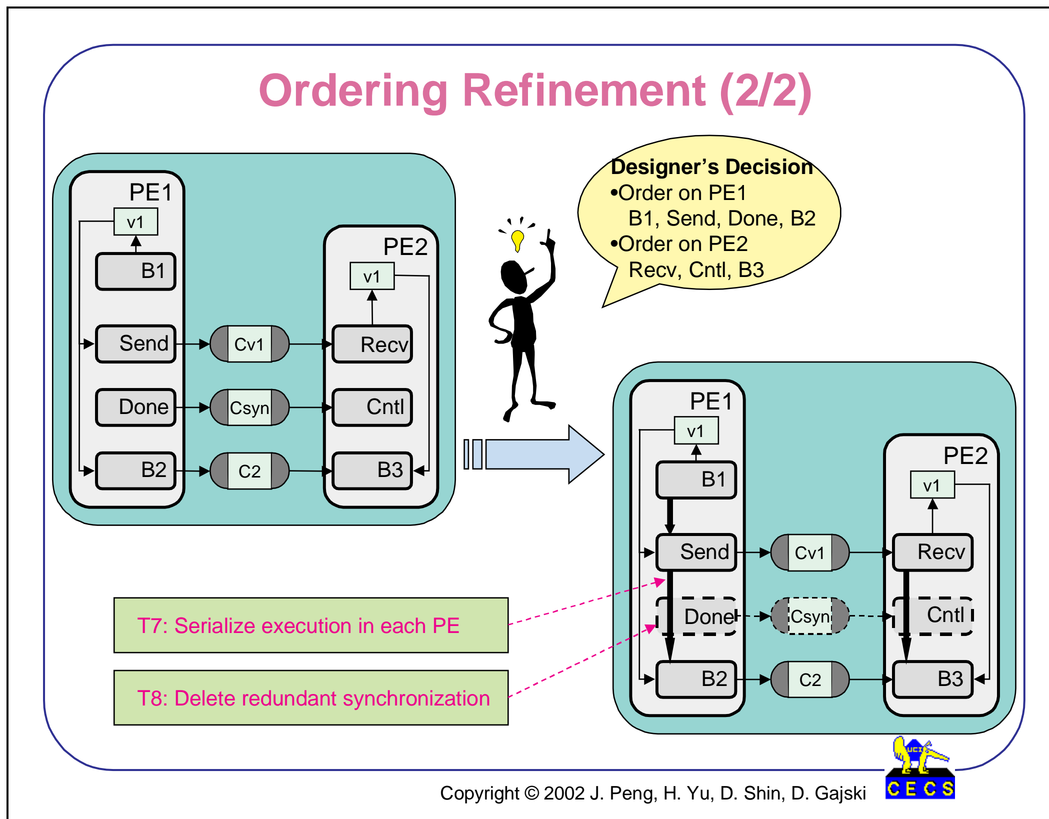
Inherently, each processing element executes sequentially with single thread of control. On the contrary, the original specification may include parallel composition of behaviors. Therefore behaviors assigned to the same component have to be serialized for sequential execution.

In a static scheduling approach, behaviors are executed in a fixed and pre-determined order. As the result, the parallel compositions of behaviors in the model are converted into sequential compositions. In a dynamic scheduling approach, the order of behavior execution is determined dynamically during runtime. In this case, a central scheduler behavior (RTOS) is introduced into the model, which dynamically selects a behavior to execute according to a certain scheduling policy. The insertion of RTOS will be discussed in detail in the later refinement steps.

For now, let us assume that the designer determines a static schedule for each component as follows:

Execution order for *PE1*: *B1*, *Send*, *Done* and *B2*

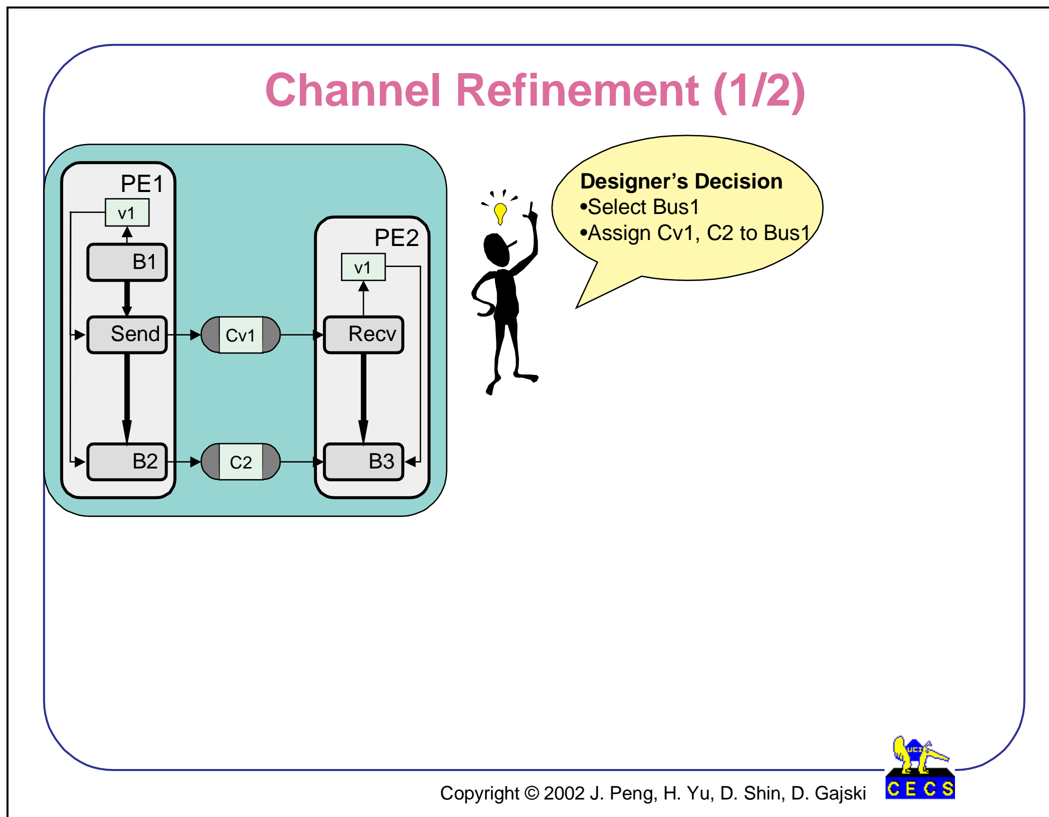
Execution order for *PE2*: *Recv*, *Cntl* and *B3*



According to the given static schedules, the behaviors are serialized for each component (bold arrows in the figure). On *PE1*, behavior *B1* executes and writes to *v1*. Then behavior Send sends value of *v1* to *PE2* through channel *Cv1*. After that, behavior *Done* notifies *PE2* of completion of *B1*. Finally, *B2* starts execution. Meanwhile, on *PE2*, behavior *Recv* receives value of *v1* and stores it in *v1*. Then behavior *Cntl* waits for the completion of *B1* on *PE1*. At the end, *B3* reads *v1* and starts execution.

After the serialization, it is obvious that the pair of synchronization behaviors, *Done* and *Cntl*, become redundant because the pair of communication behavior for *v1*, *Send* and *Recv*, already ensure that *B3* executes after *B1*. Therefore we can optimize them away from the model.

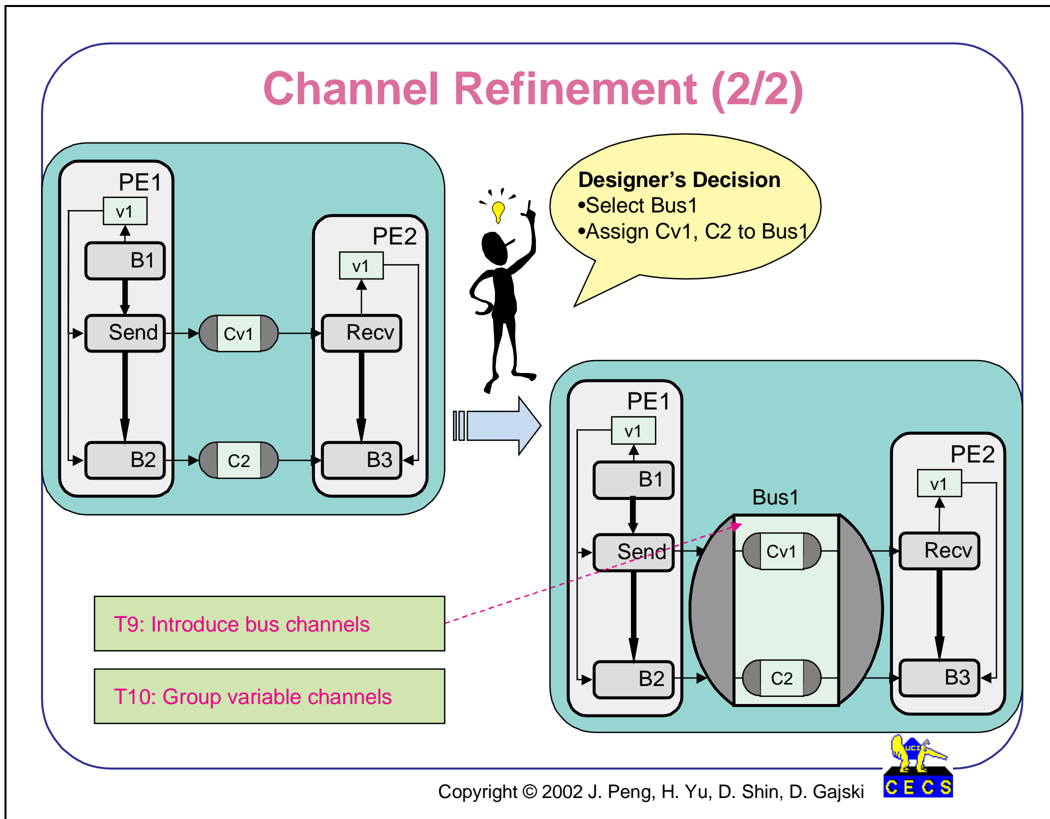
The resulted new model is called the architecture model that reflects the component structure of the system architecture. In the architecture model, all component behaviors run concurrently. However, the connectivity among the components is in the form of point-to-point abstract variable channels. In the next communication refinement step, these variable channels are refined into physical busses.



As we mentioned earlier, at this point, the communication among components is realized through variable channels (*Cv1*, *C2*), which transfer data of any size by calling abstract send/recv methods. In the final implementation, components are usually connected through system busses. Therefore, the variable channels have to be transformed into busses with real protocols.

The first decision is to select a number of busses and connect components to them. Then variable channels are assigned to busses. These decisions are important because they dictate the connection cost and communication latency, which sometimes dominates the overall performance of a design. The two extreme approaches include a single-shared-bus connection and a point-to-point connection. As described by its name, in the single-shared-bus approach, there is one common bus that connects all components. This approach is economic in terms of connection cost, however, it may suffer from competition for the common bus from different components. In the point-to-point approach, there is a dedicated connection between each pair of components. Therefore, it avoids the competition for a common resource, but it is much more costly.

In our example, since there are only two components in the system, the designer can easily decide to use one shared bus, *Bus1*, to connect *PE1* and *PE2*. Obviously, the two variable channels, *Cv1* and *C2* are all mapped onto *Bus1*.

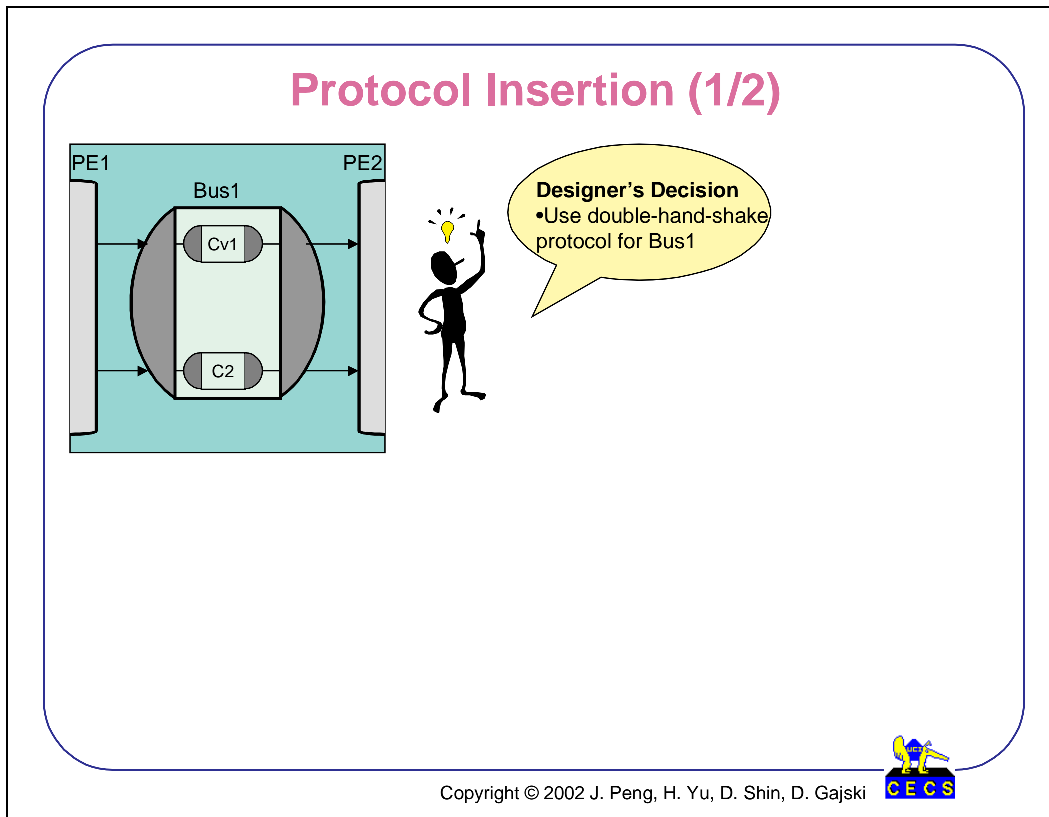


With the decisions made, the input model is modified accordingly and displayed on the right side. A couple of transformations (*Channel Refinement*) were applied to derive the new mode.

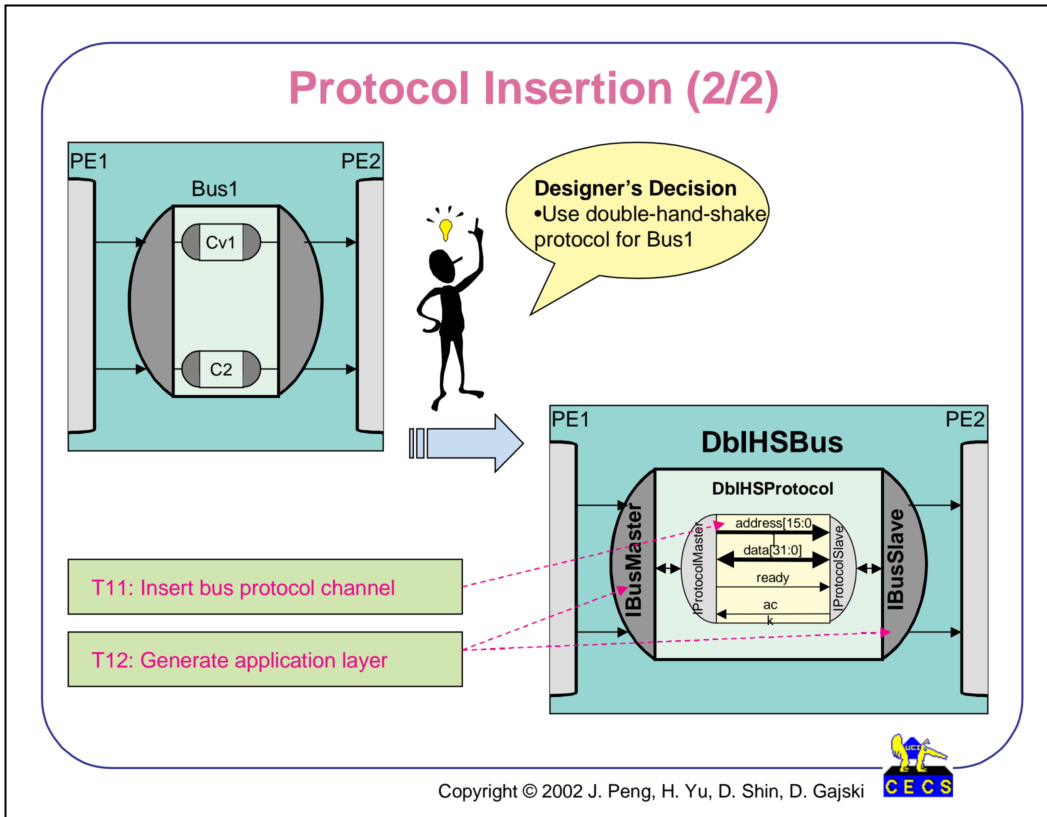
First, a new channel *Bus1* is created to represent the selected bus. *Bus1* is then instantiated at the top level to connect components *PE1* and *PE2*.

Then, the variable channels (*Cv1*, *C2*), which were at the top level in the input model, are moved inside the bus channel *Bus1*. As the result, the behaviors previously connected to the variable channels are now connected to the bus channel. For instance, behaviors *Send* and *Recv*, which were connected to *Cv1*, now are all connected to Bus1.

As we can see in the example, channel Bus1 becomes a hierarchical channel since itself includes other channel instantiations. The idea of having hierarchical channel is very useful in the communication refinement.



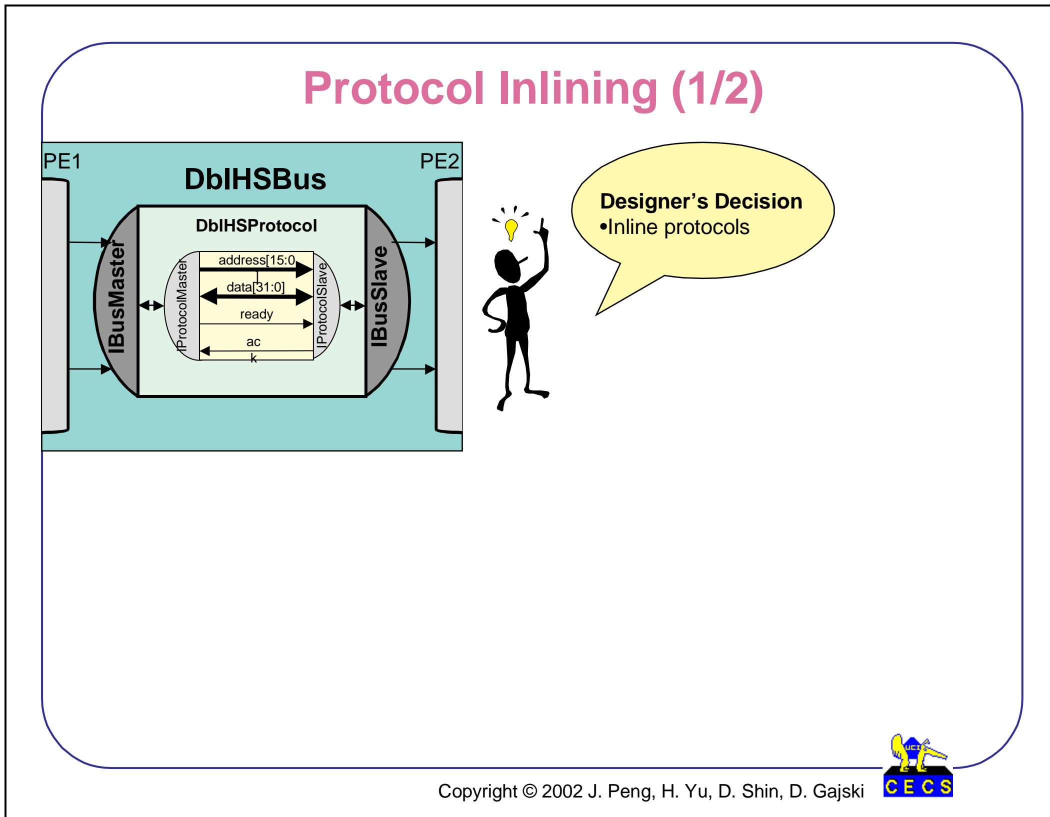
After bus selection and variable channel assignment, the bus protocol mechanisms can be selected and inserted into the model to replace the abstract communication. The bus protocol is described with yet another channel called *protocol channel*. The protocol channel encapsulates the bus wires and implements the bus protocol by driving and sampling bus wires according to the protocol timing.
The protocols can be any standard bus protocol, such as PCI bus protocol. Or it can be a custom-tailored protocol to meet unusual performance requirements.
Let us assume that in our example, a simple double-hand-shake protocol is to be used for Bus1.
The double-hand-shake protocol uses *address*, *data*, *ready* and *ack* lines. The bus transfers, send or receive, can only be initiated by the master of the bus, *PE1* in this example. During each data transfer, the master of the bus (*PE1)* starts by driving the *address* lines, and *data* lines if it is a send transfer. Then the master raises the *ready* line. When the slave of the bus (*PE2*) sees the *ready* raised, it either uploads (send) or latches (receive) the *data* lines. Following that, the *ack* line is raised by the slave. When the master sees the raise of *ack* line, it latches the *data* line and lowers the *ready* line. At last, the slave lowers the *ack* line.



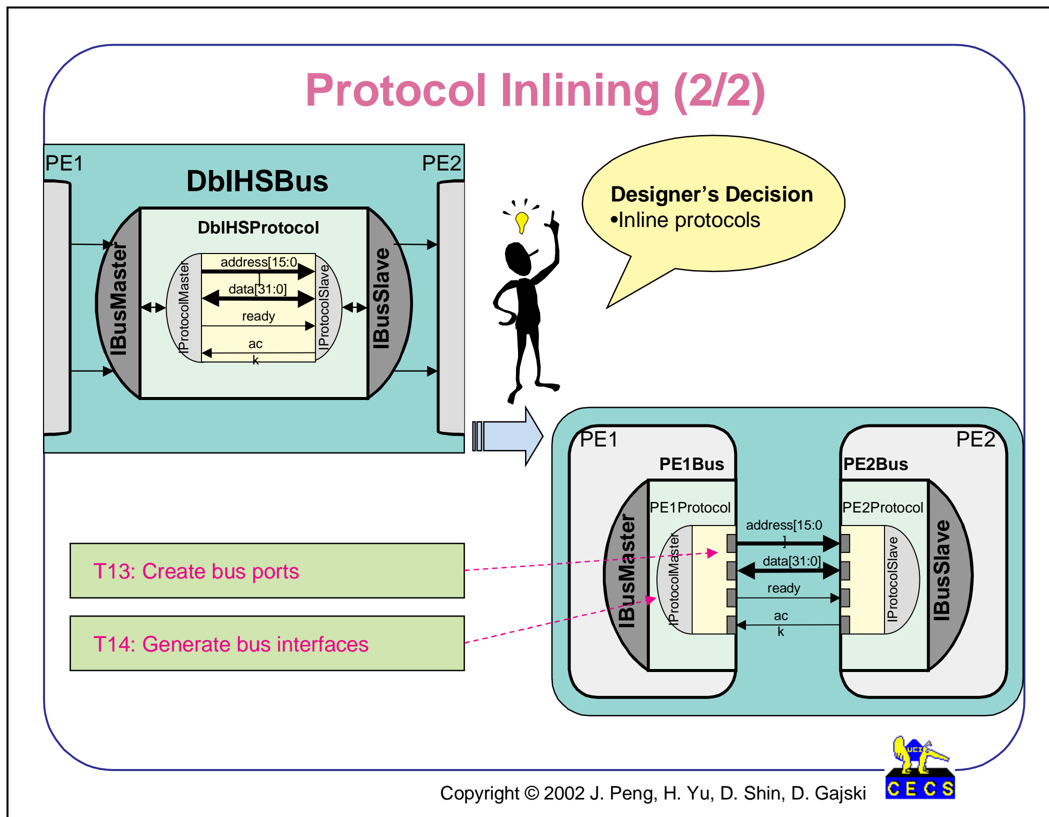
The new model with the inserted double-hand-shake protocol is shown on the right side. By comparing the two models, we can see that all differences are inside the bus channel (now called *DblHSBus* to echo the selected protocol) itself and the rest of the model remains identical. As we zoom into the bus channel, it becomes clear that two refinements are applied to the input model.

First, a protocol channel, *DblHSProtocol*, is taken out from protocol library. This protocol channel is instantiated inside the bus channel *DblHSBus*.

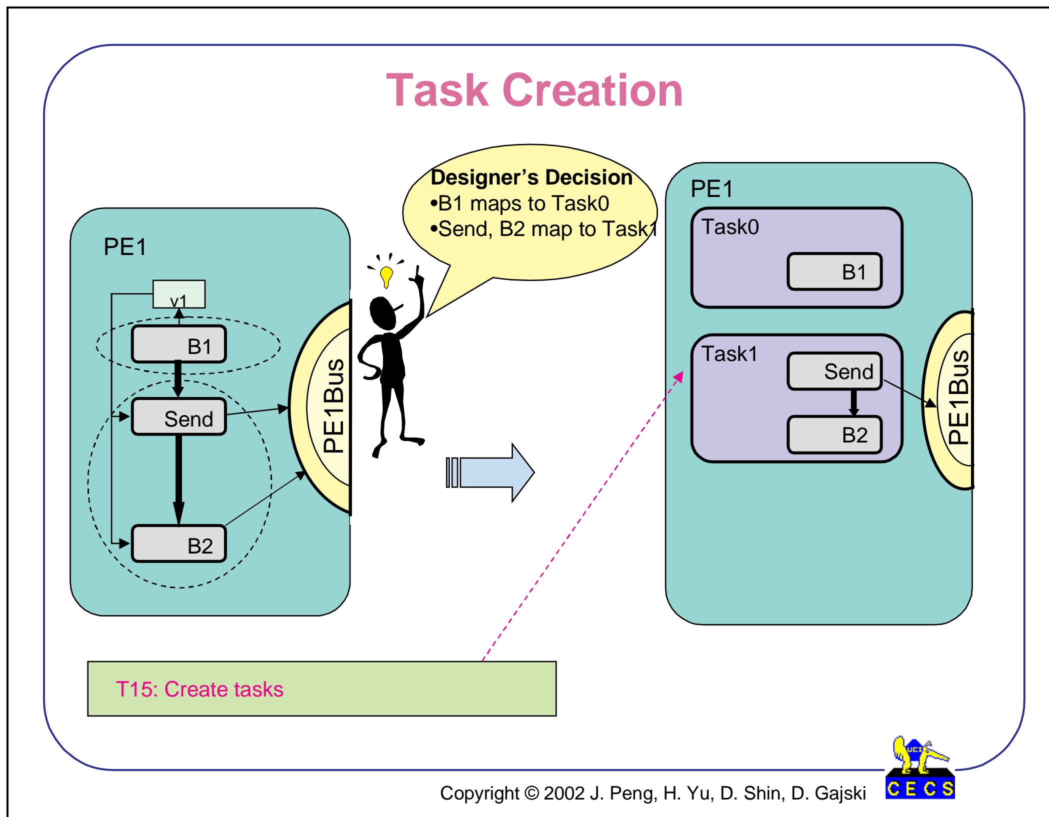
Second, an application layer (*IBusMaster*, *IBusSlave*) is implemented on top of the protocol channel. More specifically, the application layer includes a set of abstract communication methods, which are called directly by the application. For example, a application layer may have a method to send 64-bytes-array. These methods internally are implemented by calling the primitive methods provided by the protocol. For example, the application layer method for sending 64-byte-array can be implemented by calling sendByte primitive 64 times in a loop. As we can see, the internal implementation of the application layer is total transparent to the applications. Therefore, different protocols can be experimented easily.



When the designer is satisfied with the selected protocol, he may decide to inline the bus protocols into components. The protocol lining basically moves methods of channels (bus channels, protocol channels, …) into components connected to them. In other words, before inlining, inter-component communication was implemented by the channels, which are not part of any component behavior. After inlining, communication becomes part of each component behavior. The communication of each component will be synthesized together with the rest of the component's functional (computation) behaviors. After protocol inlining, a bus-functional model is generated, where the connections between components are represented by bus wires.



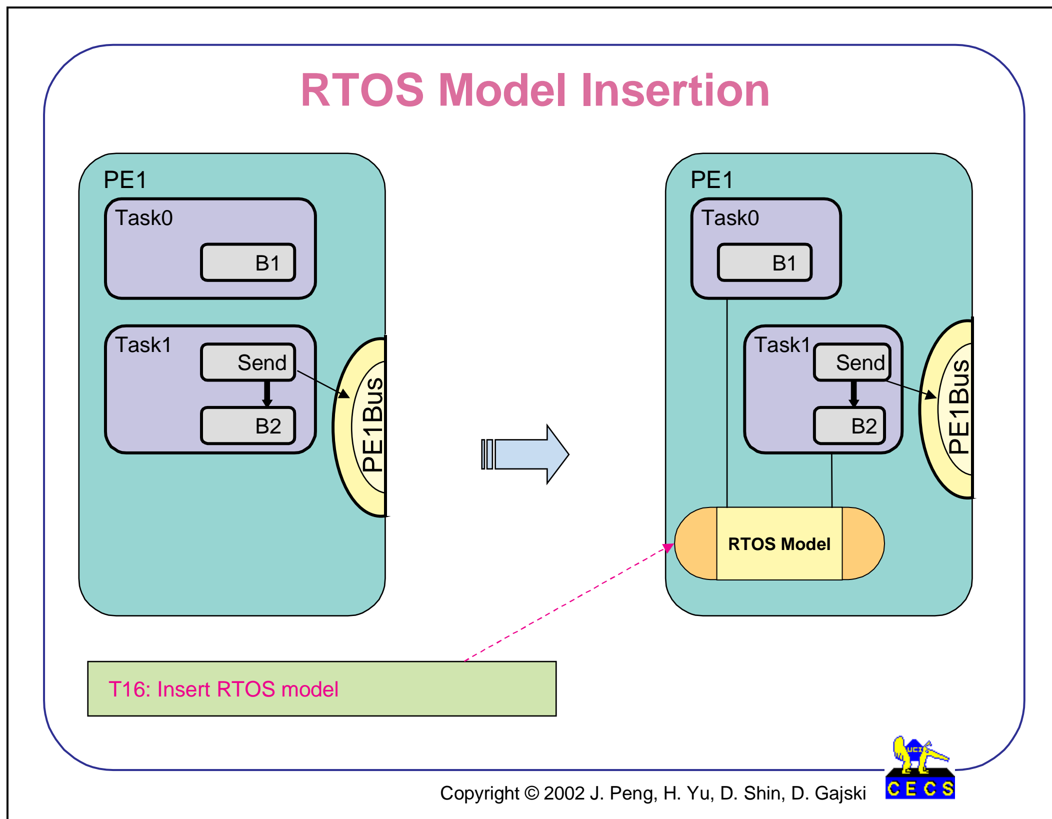
When designers decide to inline the bus protocol, a couple of refinements are performed on the input model to obtain the new model.
First, bus ports (*address*, *data*, *ready*, *ack*) are created for components, *PE1* and *PE2*. These ports are needed for connecting to the bus wires.
Then, the part of the bus channel becomes bus interface channel inside the corresponding component. The interface channel is instantiated and connected to the bus ports. In the example here, interface channel *PE1Bus* and *PE2Bus* are instantiated inside *PE1* and *PE2*, respectively. (The behaviors in *PE1* and *PE2* are omitted intentionally for clarity.)
The resulted communication model is a bus-functional model, where each component is a behavioral description but the communication among them is described with time-accurate bus protocol. In the following sections, the computations (behaviors) as well as the inlined communications (channels) are further refined through software synthesis for software components or hardware synthesis hardware components.



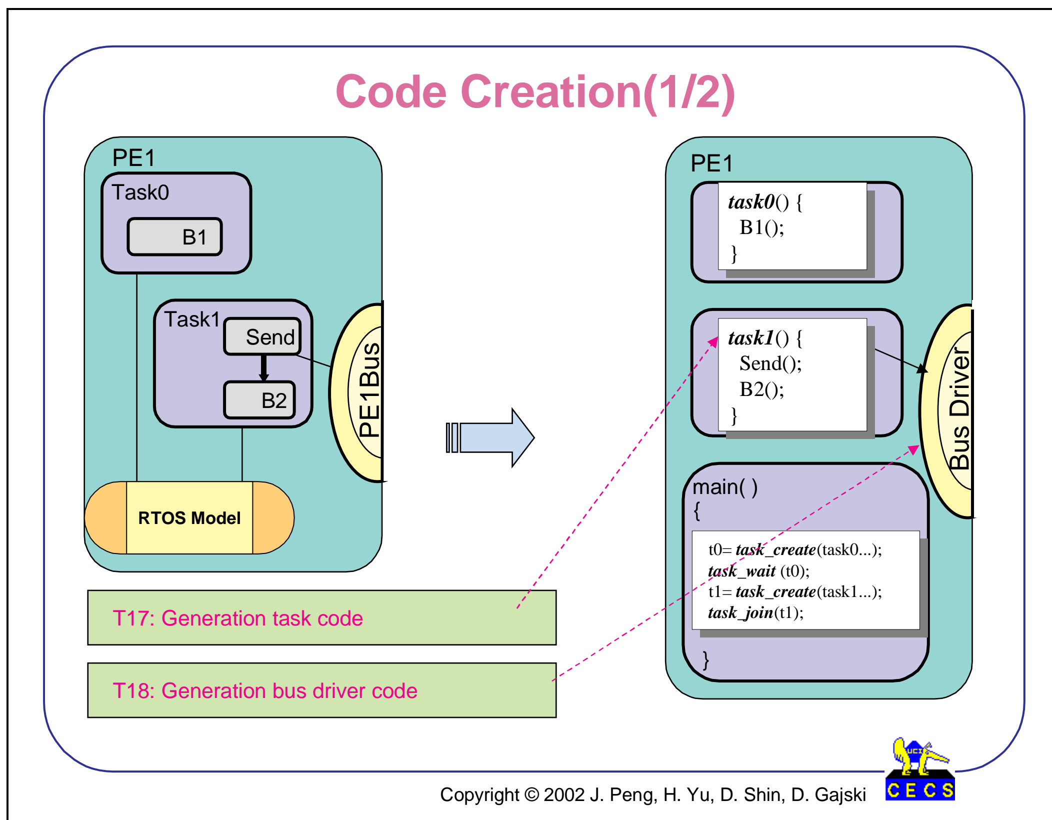
The computation represented by the behaviors executing on the programmable processor component is implemented by the software. Software refinement is the process in which the behaviors mapped onto a programmable processor are converted into C/C++ code, compiled into the processor's instruction set, and possibly linked against an RTOS. Task creation is the first step. Task is a piece of sequentially executed code, which has a single entry and exit point. If the original specification includes parallel composition of behaviors mapped to a single processor, then each concurrently executed behavior will be implemented as a task in software.

During the task creation process, an additional level of behavioral hierarchy is introduced to wrap around the behaviors included in the tasks. Generally, the parallel behaviors in the specification are refined into parallel tasks while the sequential behaviors are grouped in one task.

For our simple design example, we assume that the designer maps behavior B1 to *Task0*, behavior Send and B2 to *Task1*, so after task creation, two additional wrapper behaviors are inserted to represent two tasks: behavior *Task0* represents the first task which wraps around the code of behavior B1 while behavior *Task1* represents the second task which groups the code of behavior Send and B2.



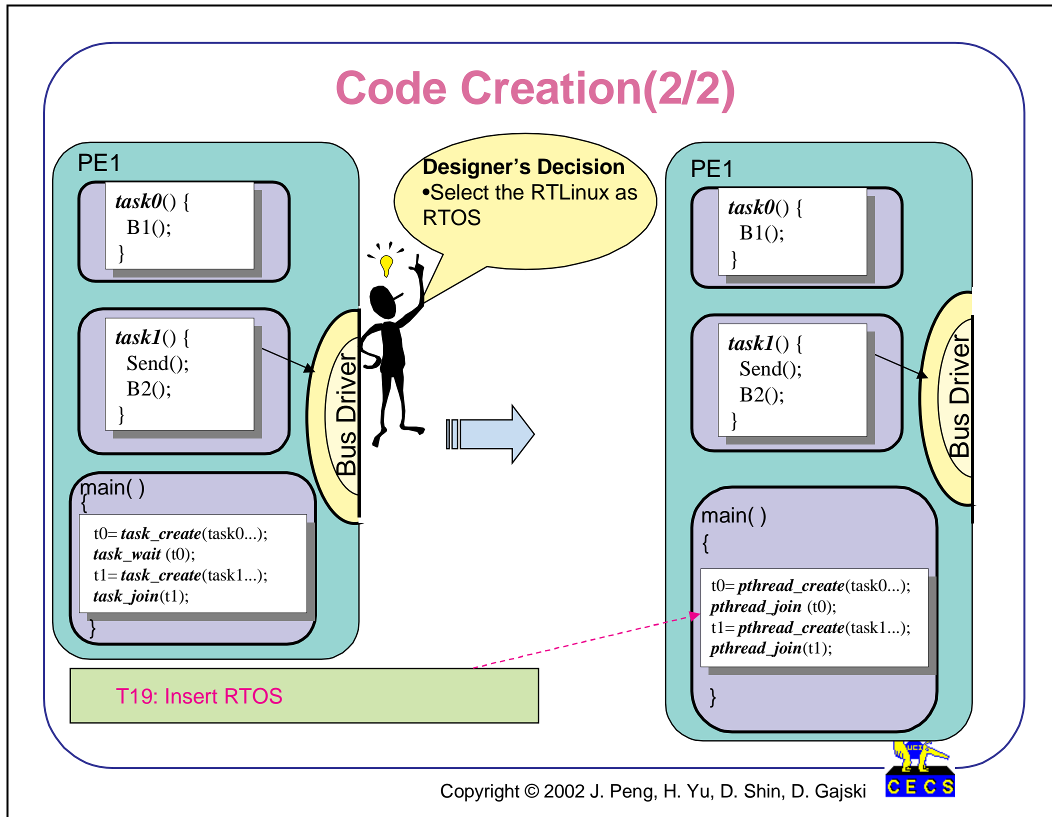
Usually, tasks can be scheduled statically, but there are cases in which data inter-dependence between the tasks requires dynamic scheduling. Besides, we want to explore different dynamic scheduling algorithms and their effects on system performance at an early stage. In the implementation, scheduling service is provided by a real time operating system (RTOS). However, at this stage, using a detailed, real RTOS implementation would negate the purpose of an abstract system model. Furthermore, information may not be available to target a specific RTOS. Therefore, we need techniques to capture the abstracted RTOS behavior in system level models. Thus, a high level model of the RTOS is introduced into the system at this step. The RTOS model provides an abstraction of the key features that define a dynamic scheduling behavior independent of any specific RTOS implementation. It maintains a pool of tasks and dynamically selects a task to execute according to its scheduling algorithm.
In our design example, *Task0* and *Task1* are created using the RTOS model's API and scheduled by the RTOS model.



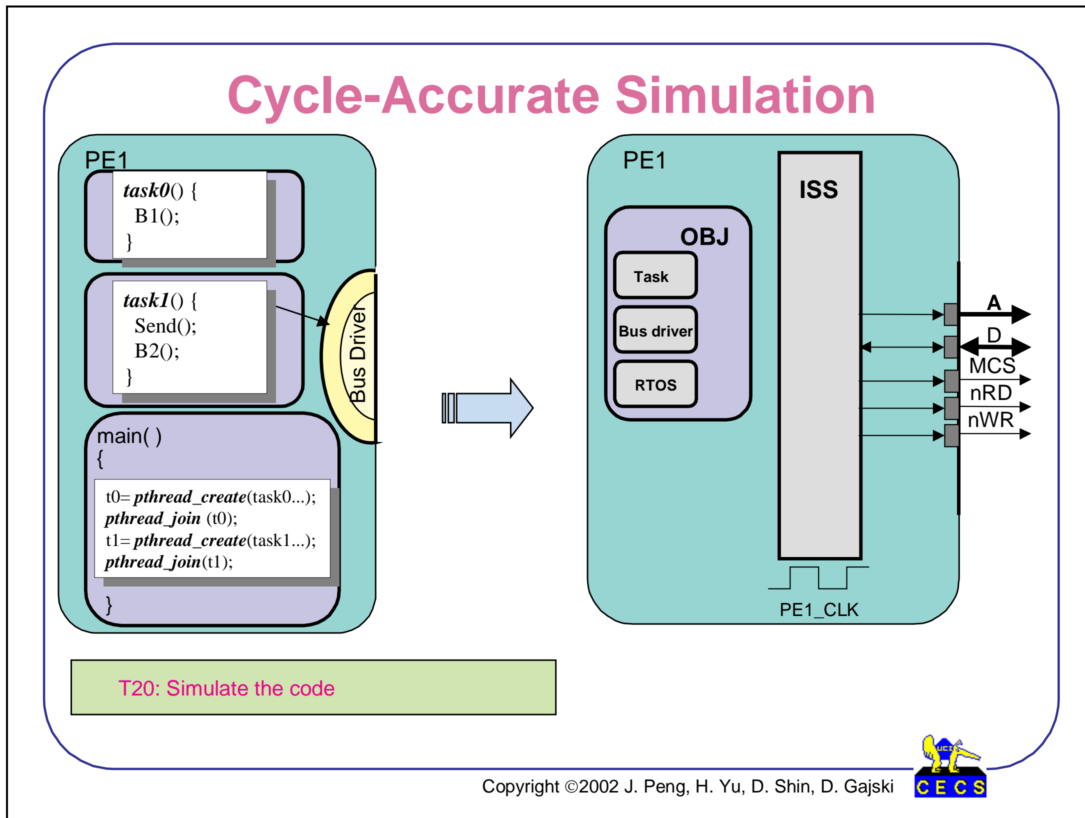
After the tasks are created for a processor and the RTOS model is introduced, the designer can selectively simulate the system software part either at this level or at cycle-accurate level.

If the designer want to simulate at cycle-accurate level for the software part, we need generate binary code for the processor. The first step of code generation is to create the ready-to-compile code for the software tasks. First, all the behaviors inside a task are converted into classes with the behavioral hierarchy converted into class hierarchy and the ports of the behaviors into class parameters. In the next step, the top level behaviors inserted during the task creation process are converted to C functions, with each function implements a task and contains the class code of the behaviors mapped to that task. As the last step of the code creation process, the communication channels are refined into bus drivers and each call to send/receive method are replaced by calls to the send/receive member functions of the corresponding bus driver.

For our design example, first, all the behaviors in the original specification are converted into C++ classes (class B1, class Send, class B2), then the two additional behaviors inserted during the task creation process are converted into two C functions (*Task0* and *Task1*) implementing the two tasks. The channel PE1Bus is converted to a C++ class containing the bus driver send/receive methods of PE1Bus.

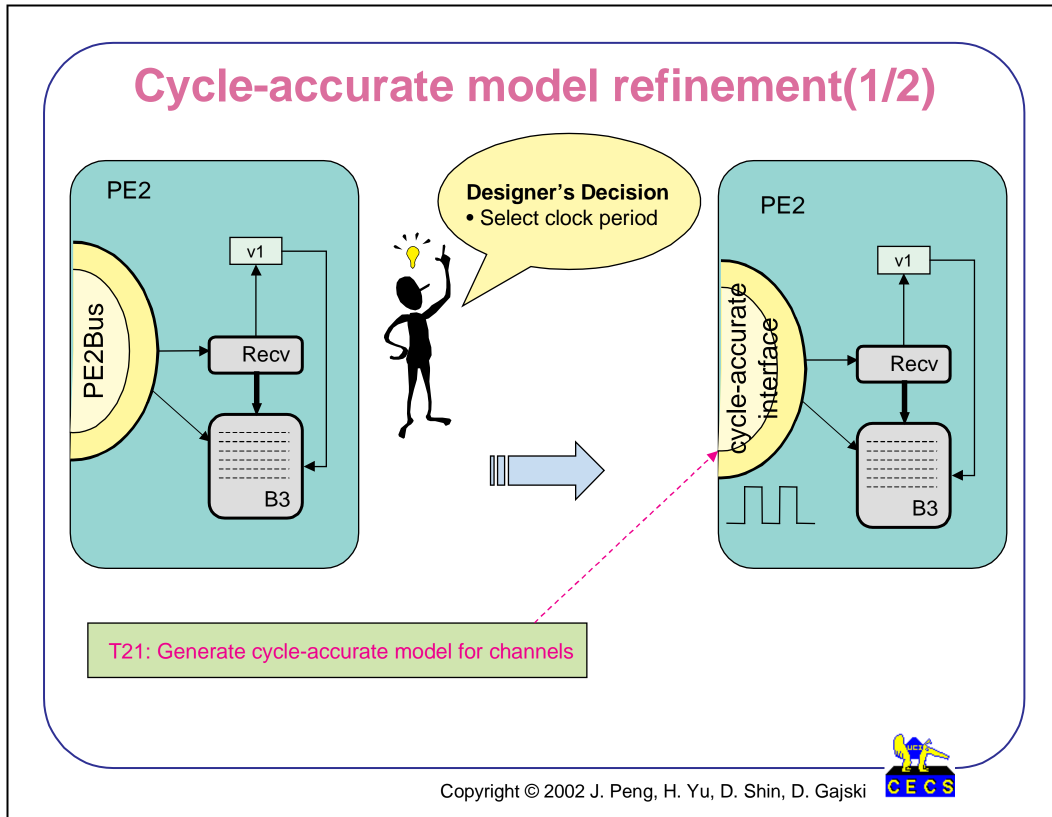


As the next step of code creation, the RTOS model is implemented. The services of the RTOS model are mapped onto the APIs of a specific standard or custom RTOS. In the former case, the designer selects a specific RTOS, then the interface of the RTOS model is implemented either by mapping them to the equivalent services of the selected RTOS or by creating code on top of the actual RTOS primitives if the service is not provided natively. After that, the compiled application object code is linked against the specific RTOS library to produce the executable code for the target PE. If the designer doesn't want to use a standard RTOS, since the number of interface routines of the RTOS model is small, we make a custom RTOS to implement these RTOS model interface routines.
In our example, the designer select RTLinux as the target RTOS and the two interface routines of the RTOS model (task_create, task_wait) are mapped into the corresponding system calls in RTLinux(pthread_create,pthread_join). The code generated in the previous step is then compiled and linked against RTLinux library to create the final executable code.

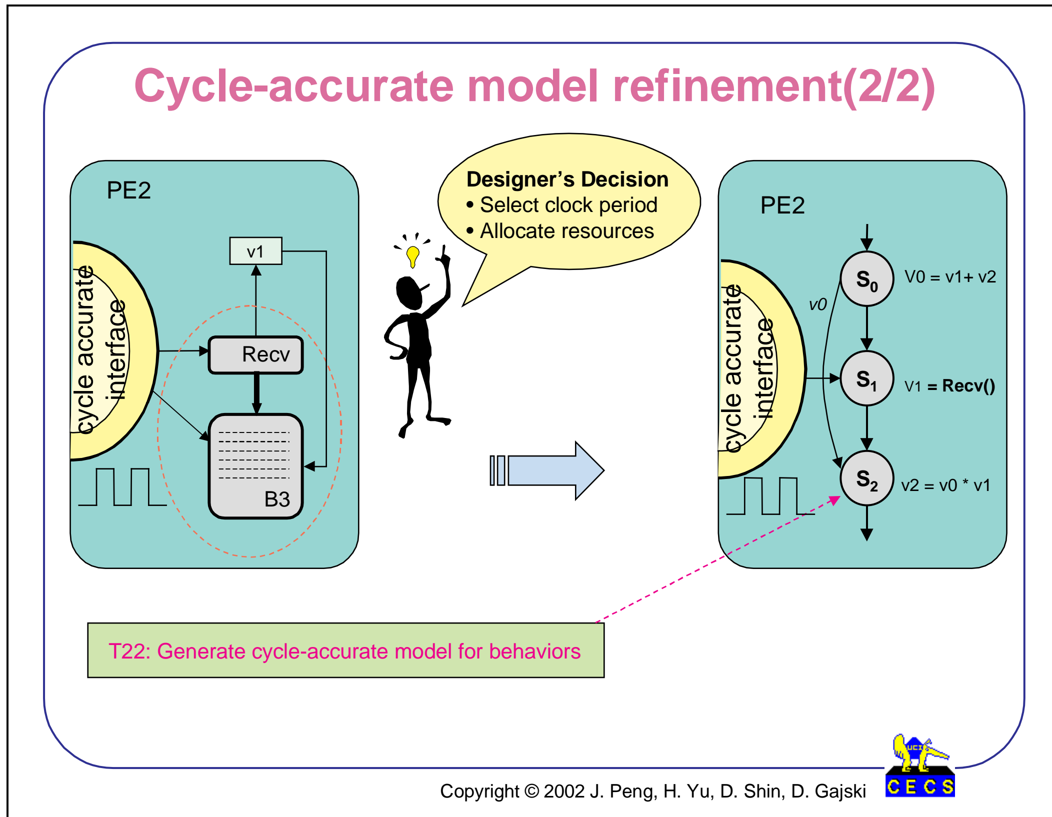


As we mentioned previously, the designer can selectively simulate the software part at different levels. If the designer want to simulate the software code at the cycle-accurate level, the instruction-set simulator (ISS) executes binary code created in the previous steps based on the component's clock and drives the ports of the programmable processor according to the inputs and outputs generated by the simulator.

Due to the fact that the component's interfaces remain unchanged during the software refinement process, a mixed-level co-simulation of communication and implementation models is easily possible. In our example, the instruction-set simulation of the *PE1* implementation model can be combined with a bus-functional simulation of other PEs in communication model. This allows evaluation of the detailed implementation of a single component together with higher-level models of the other components with the result of increased simulation speed.



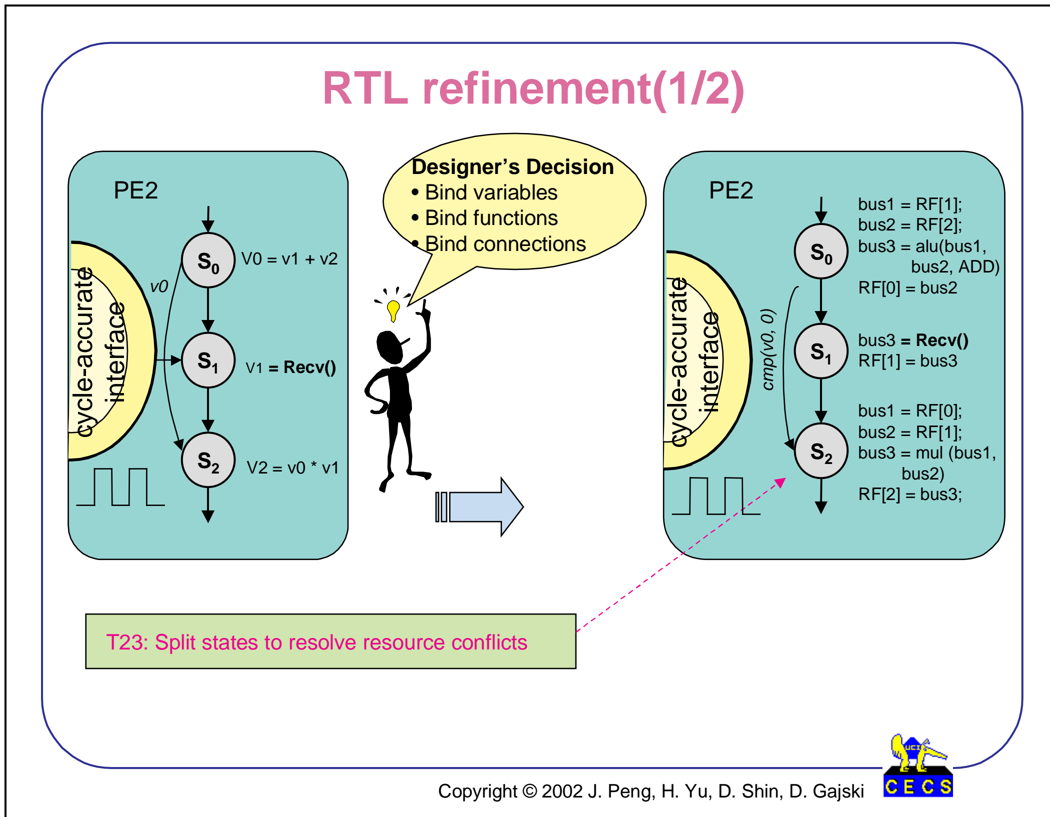
The PEs assigned for custom hardware will be implemented with RTL processors. Thus cycle-accurate refinement converts communication model of PEs to cycle-accurate model that will execute on user defined RTL processor.
The bus function model of the communication protocol in channel describes timing diagram of the protocol as events on the wires and associated timing constraints.
The first task of cycle-accurate model refinement is to obtain cycle-accurate interface model by scheduling bus protocols into the selected clock cycle, while still satisfying timing constraints. We use FSMD model to represent the cycle accurate model. The FSMD model is an intermediate model, which contains cycle-accurate descriptions for each PE.
In this example, bus functional model of *PE2Bus* is refined into *cycle-accurate interface* model.



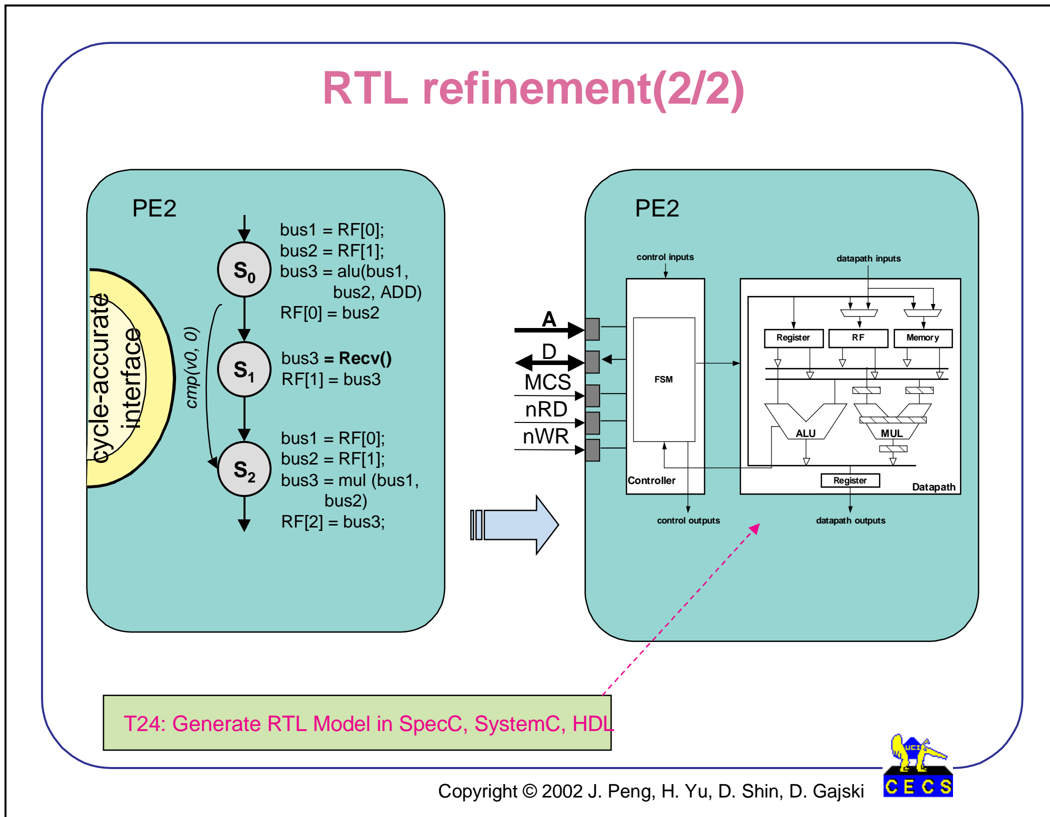
During cycle-accurate model refinement for behaviors, user will select the clock period and allocate resources from RTL component database. RTL components database contains function units and storage units. In this task, the behavioral description of PEs will be split into clock cycles, which can be represented by FSMD model.

The target architecture for custom hardware is an RTL processor. The RTL processor consists of a controller and a datapath. Datapath consists of set of storage units, set of functional units and set of busses. All these RTL components may be connected arbitrarily through busses. Each component may take one or more clock cycles to execute, each component may be pipelined and each component may have input or output latches or registers. The entire datapath can be pipelined in several stages in addition to components being pipelined themselves. The controller defines the state of the RTL processor and issues the control signals for the datapath.

In our example, the hierarchy of behaviors (*Recv* and *B3*) will be flattened into a behavior of *PE2* which will be divided into clock cycles. For example, variable *v0* is obtained by adding variable *v1* and *v2* in S0. Interface function call of *Recv()* will be performed in state S1. In S2, *v2* is the result of multiplication of variable *v0* and *v1*.



In this step, variables in behavioral description will be assigned to storage units such as registers, register files, and memories. The operations will be mapped to function units such as ALUs, multipliers, logic units and others. The data transfers between variables and operations will be assigned to busses. All the assignment can be done by user to maximally utilize user insight. Then behaviors will be split for conflicting resources such as number of function units, number of ports of the storage units and number of busses.
In our example, variable *v0*, *v1* and *v2* are assigned to register file *RF[0], RF[1],* and *RF[2]* respectively. Operation addition (+) and multiplication (*) are mapped to *alu* unit and *mul* units respectively. *Alu* unit is 1-cycle function unit and *mul* unit is 2-stage pipelined unit with input and output latches. Data transfers between variables and operations are assigned to *bus1*, *bus2* and *bus3*. Then after all assignment is done by users, states will be split to resolve the resource conflict. The final FSMD is shown in right side of above figure.



The final step of RTL refinement is generation of implementation model, which implements the PE's state machine and is structural description of RTL processor. For each PE, a structural description of the PE's controller and datapath in the form of a netlist of RTL components is available. As such, the implementation model is accurate down to sub-cycle delays.

Finally, the implementation model is then refined down through traditional design flows. The structural representation of RTL can be written in SpecC, SystemC, Verilog or VHDL. The structural RTL description in HDL represents the input to standard EDA synthesis tools such as Synopsys Design Compiler.

Conclusions

- **Synthesis steps are separated into decision-making and model refinement**
- **Automatic mode refinement made possible**
  - refinements are well-defined
  - steps are simple and small
- **Benefits from automation**
  - Significant productivity gain
  - Extensive design space exploration
  - Formal verification can be applied

Copyright © 2002 J. Peng, H. Yu, D. Shin, D. Gajski

In this chapter, we illustrated the concept of automatic model refinement using our example design flow. This kind of automation is based on the separation of the decision-making task and the model-changing task at each design step. The decision-making usually needs designers' involvement meanwhile the model-changing does not.
As shown in the simple example, we identified the specific design decisions needed for each synthesis step. In addition, the corresponding refinement rules are defined to change models to reflect design decisions. The refinements are simple and well-defined, which are necessary for automation.
The benefit of the automatic refinement is multifold. Obviously, model refinement will significantly shorten the design cycle thus increase productivity. Furthermore, it enables extensive design space exploration at each step to produce a better design.
This kind of automation also sheds light on the verification problem, which is becoming more and more critical in the current design practices. First of all, automation totally eliminates human errors and mistakes, which are hard to avoid when re-writing models manually. Furthermore, the well-defined nature of the models and refinement rules makes it possible to formally verify or even guarantee the equivalence of models at different steps.

Chapter 5

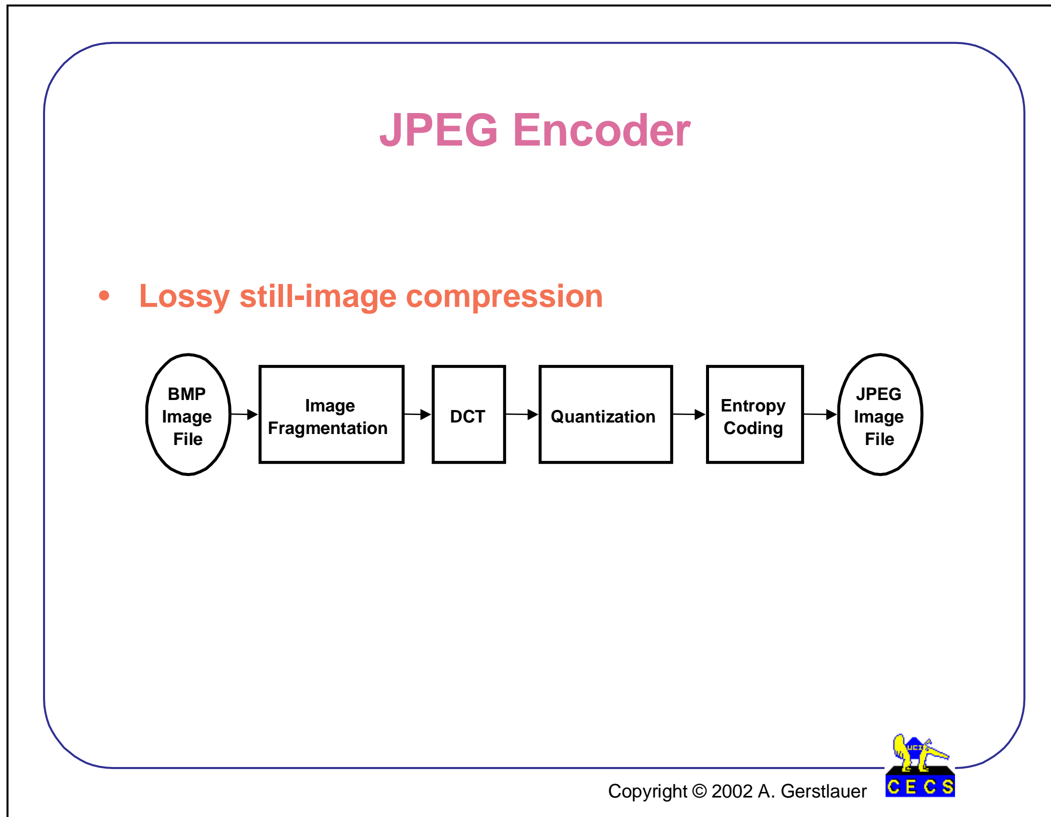
Design of a JPEG Encoder

Design of a JPEG Encoder

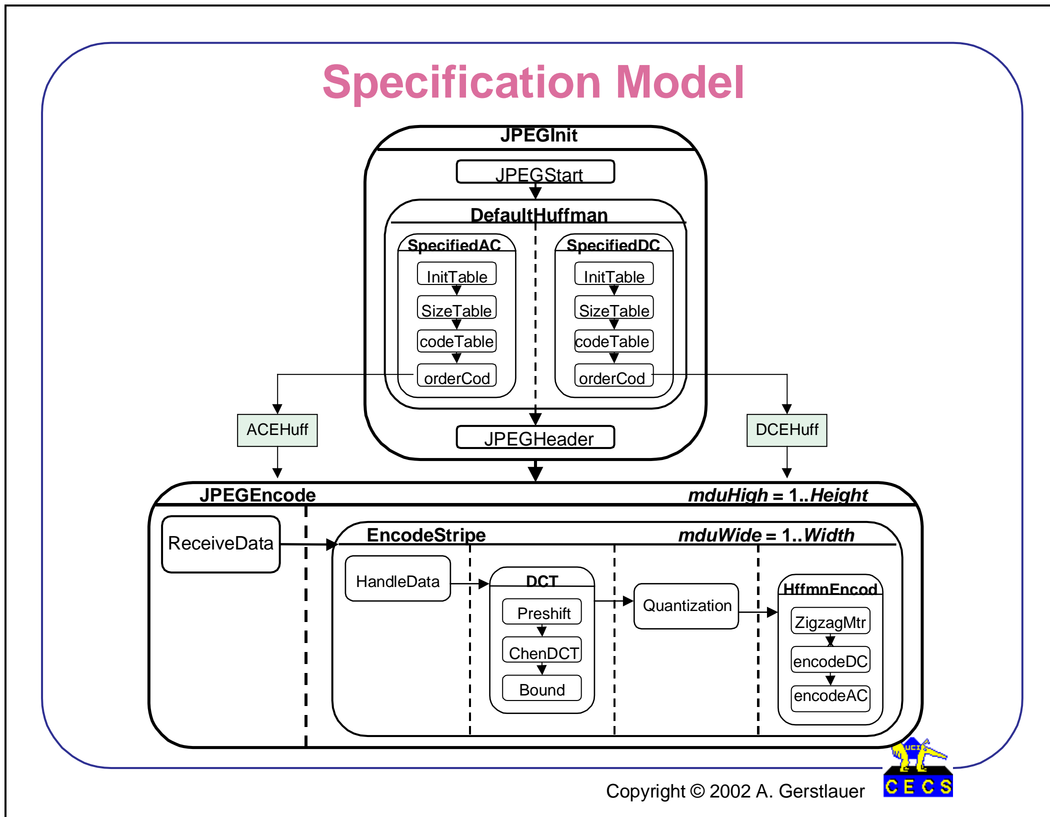
(Source: ICS TR-99-54)

Andreas Gerstlauer
Center for Embedded Computer Systems
University of California, Irvine
http://www.cecs.uci.edu/~SpecC

Copyright © 2002 A. Gerstlauer

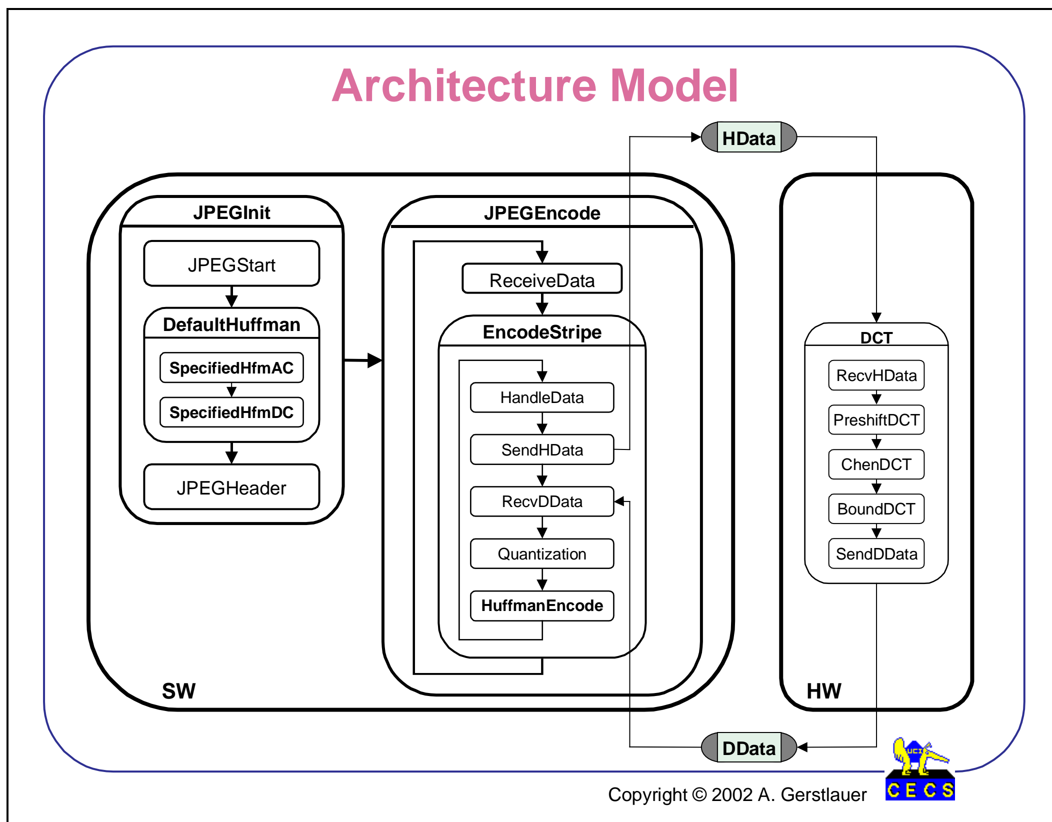


In addition to the JBIG design example, we also applied the system-level modeling concepts to the example of a JPEG encoder. JPEG is a widely used lossy image compression standard. Compression is based on the characteristics and limitations of the human eye and is therefore especially suited, for example, for digital photographs.
There are four modes of the operations in the JPEG standard: the sequential discrete cosine transform (DCT)-based mode, the progressive DCT-based mode, the lossless mode, and hierarchical mode. Our design employs the first mode, the sequential DCT-mode, which is the simplest and the most commonly used mode.
At the core of the JPEG encoding algorithm is a four-stage pipeline. In the first stage, the source image is divided into non-overlapping data blocks, each of which contains an 8x8 blocks. Next, each block is processed through a discrete cosine transform.
In the quantization block, the DCT output coefficients are quantized. Finally, the AC coefficients are encoded by using a predictive coder and the DC coefficients are encoded by using a run-length coder. Then the Huffman coding algorithm is employed to generate JPEG image.



At the top of the JPEG specification model shown here, the encoder consists of two sequential behaviors, *JPEGInit* followed by *JPEGEncode*. *JPEGInit* performs initialization of the two Huffman tables in two parallel subbehaviors, and writes the output header. Then, the actual encoding is done in two nested, pipelined loops. The outer pipeline splits the image into stripes of 8 lines each. The inner pipeline then splits the stripes into 8x8 blocks and processes each block through DCT, quantization and Huffman encoding.

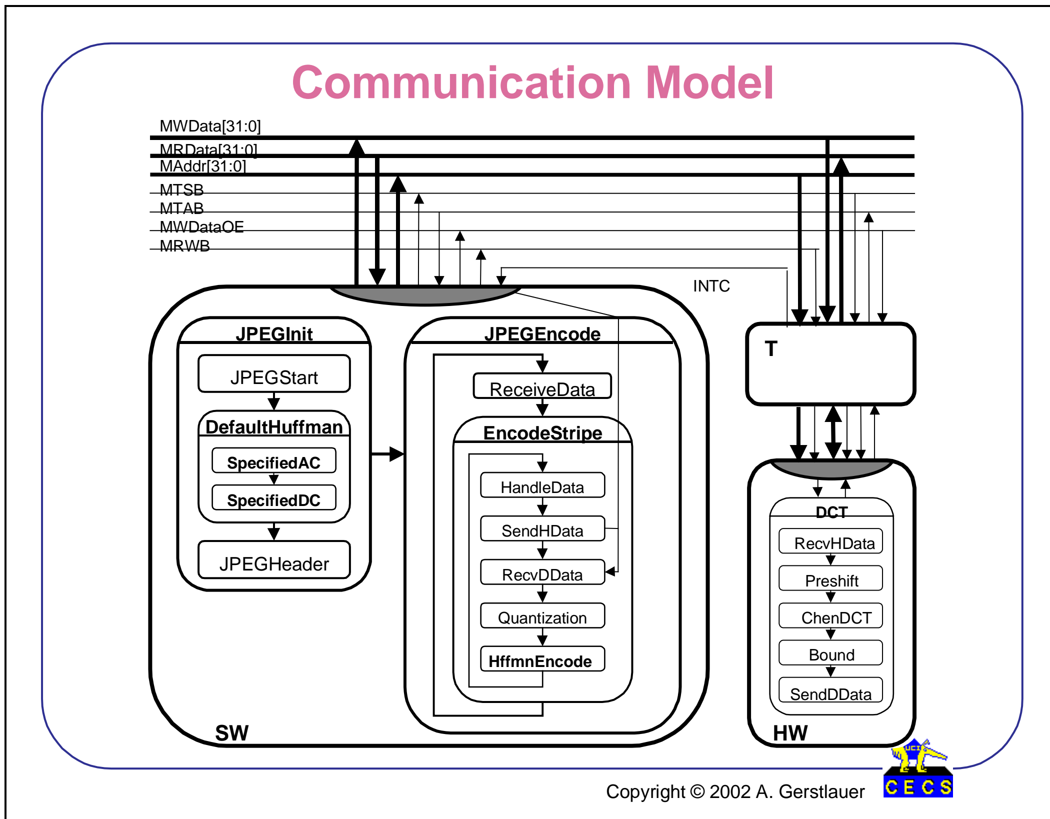
As an example of communication, the figure shows the two Huffman tables *ACEHuff* and *DCEHuff* that are sent from *JPEGInit* to *JPEGEncode*. Note that since the two behaviors are composed sequentially, channels can degenerate to simple variables.



For the purpose of architecture synthesis, we assumed a mapping of the encoder on a Motorola Coldfire processor (*SW*) assisted by a custom hardware co-processor (*HW*) for acceleration of the DCT.

Software and hardware communicate via two message-passing channels, sending and receiving 8x8 blocks from software to the DCT processor and back. Behaviors inside the SW processor are statically scheduled and serialized. The two nested pipelines are converted into two nested, sequential loops.

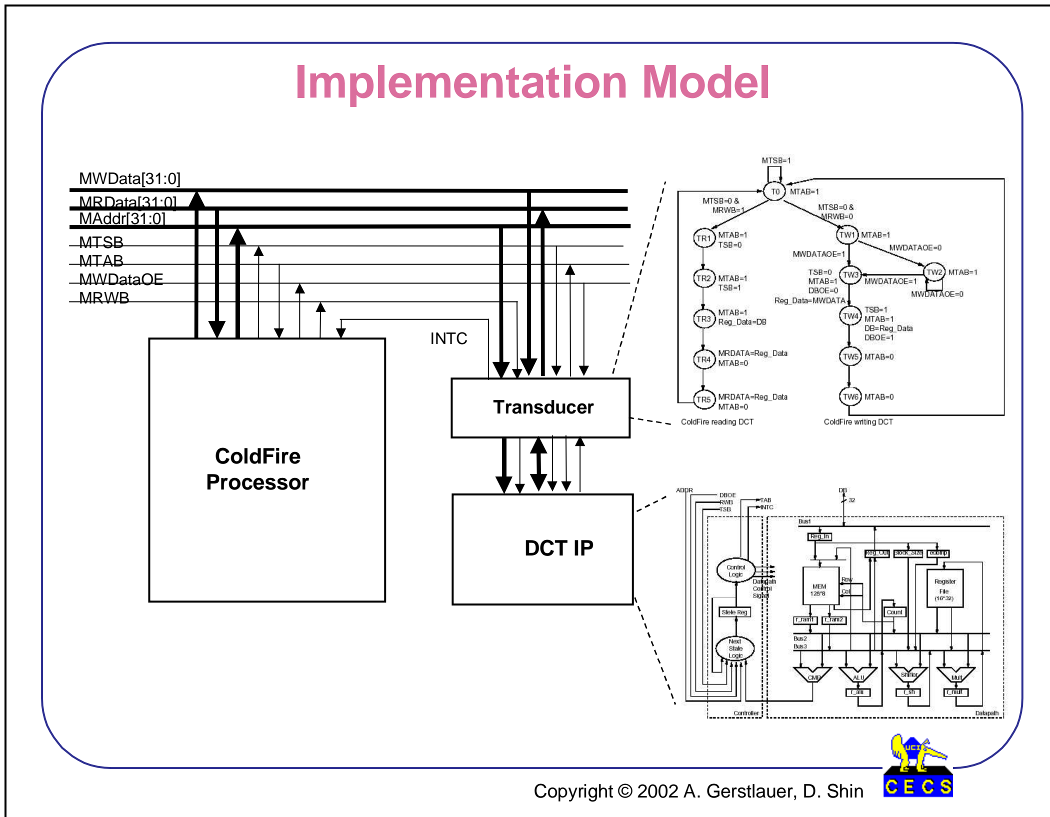
In the JPEG architecture model, the software waits for the result of the DCT before continuing with any processing. By changing only a few lines of code, we were able to modify the architecture such that software and hardware operate in a pipelined fashion (i.e. while the DCT is processing a block the software continues processing of the previous block and prepares the next one), resulting in 100% utilization of the *SW* processor. Similarly, other architectural alternatives can be easily explored in a very short amount of time with minimal changes in the model.



For communication synthesis, we connected the two processors via a single bus using the Coldfire bus protocol. Furthermore, it was assumed that the protocol of the DCT IP is fixed and incompatible with the Coldfire protocol, necessitating the inclusion of a transducer.

The *SW* processor is the master on the bus and drives the address and control lines. The *HW* co-processor listens directly on the address bus and its associated control lines while the transducer translates between data transfer protocols. For synchronization, the hardware signals the software through the processor's interrupt line *INTC*.

Inside the two PEs, bus drivers and interrupt handlers translate the message-passing calls of the behaviors into bus transactions by driving and sampling the PE's bus ports according to the protocol.



The final implementation model after communication synthesis and synthesis of the custom hardware parts is shown above.

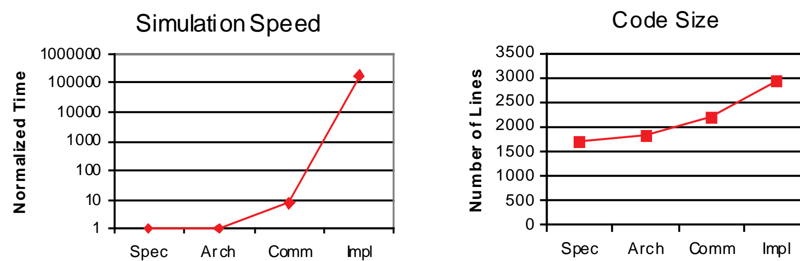
For the software part of the JPEG design(ColdFire processor), the corresponding behaviors will be compiled into binary codes which will be simulated by ISS.

For the custom hardware part of the JPEG design(*DCT IP*), a corresponding RTL processor with custom controller and custom datapath was instantiated from IP library. DCT IP has CMP, ALU, SHIFT and MUL to perform operations and 128x8 internal memory, 18x32 register files and registers to store run-time data.

For the *DCT IP* interface, a transducer was synthesized that performs protocol translation between the *DCT IP* interface and the system bus (ColdFire bus protocol) based on the timing diagrams of the two bus protocols. The behavior of the transducer is represented by FSM.

Results and Conclusions

- Simulation Speed & Code size



- Refinement Effort

Refinement Effort

	Modified lines	Manual	Automated User / Refine
Spec→Arch	751	1~2 mons.	5 mins / <0.5 min
Arch→Comm	492	~1 mons.	3 mins/ <0.5 min
Comm→Impl	1,278	3~4 mons	20 mins / <1 mins
Total	2,521	5~7 mons	28 mins <2mins

Copyright © 2002 A. Gerstlauer, J. Peng

Results for the JPEG encoder models in SpecC are shown here from system specification model to implementation model. The graphs show the time needed for the simulation and the number of lines of code for each model for encoding of a 116x96 black-and-white image.
To validate the models, we performed simulations at all levels. As we move down in the level of abstraction, more timing information is added, increasing the accuracy of the simulation results. However, simulation time increases exponentially with lower levels of abstraction. As the results show, moving to higher levels of abstraction enables more rapid design space exploration. Through the intermediate multiprocessing model, valuable feedback about critical computation synthesis aspects can be obtained early and quickly.
As the number of lines of code suggests, refinement between models is minimal and is limited to the additional complexity needed to model the implementation detail introduced with each step.
The refinement effort table shows that automated system level refinement process can achieve 2000 times productivity gains comparing manual refinement process in JPEG project.

Chapter 6

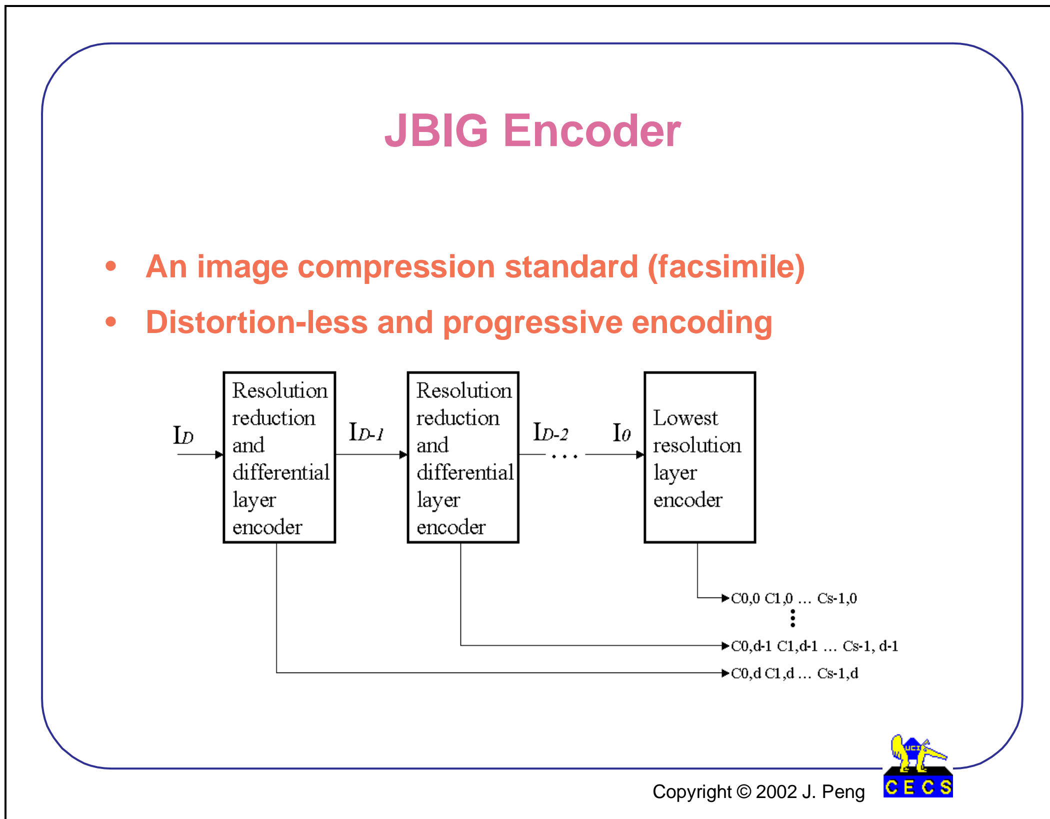
Design of a JBIG Encoder

Design of a JBIG Encoder

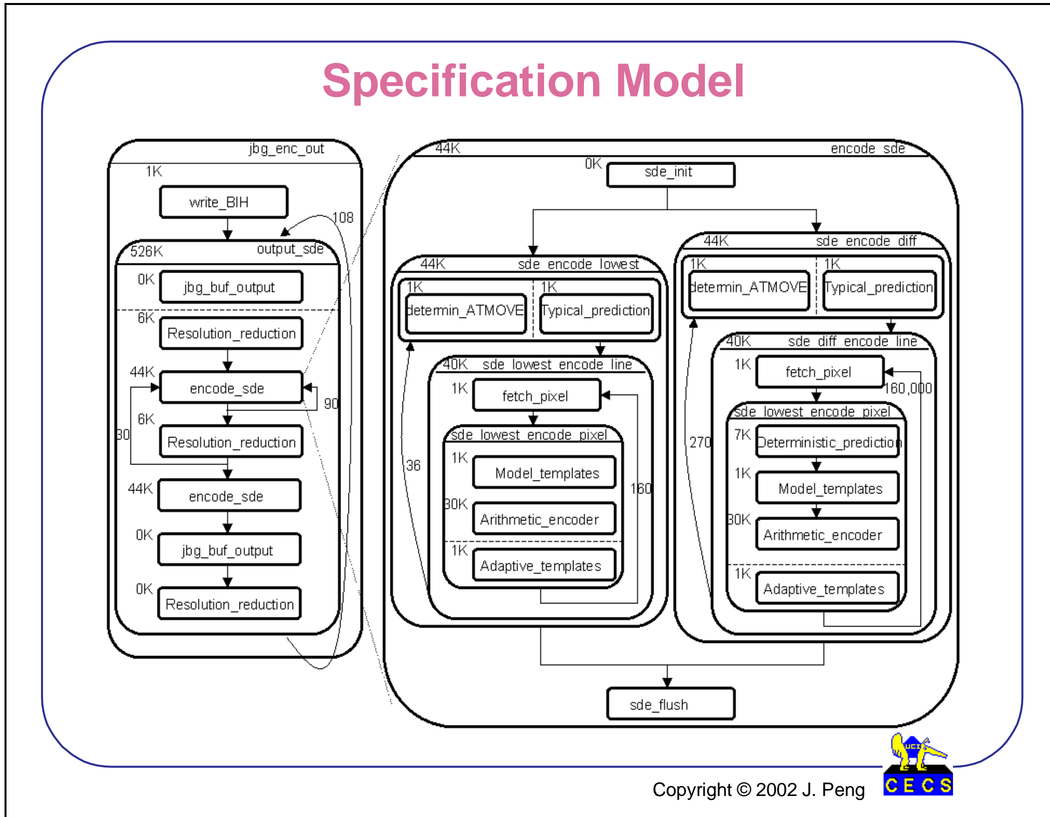
(Source: ICS TR-00-13)

Junyu Peng
Center for Embedded Computer Systems
University of California, Irvine
http://www.cecs.uci.edu/~SpecC

Copyright © 2002 J. Peng

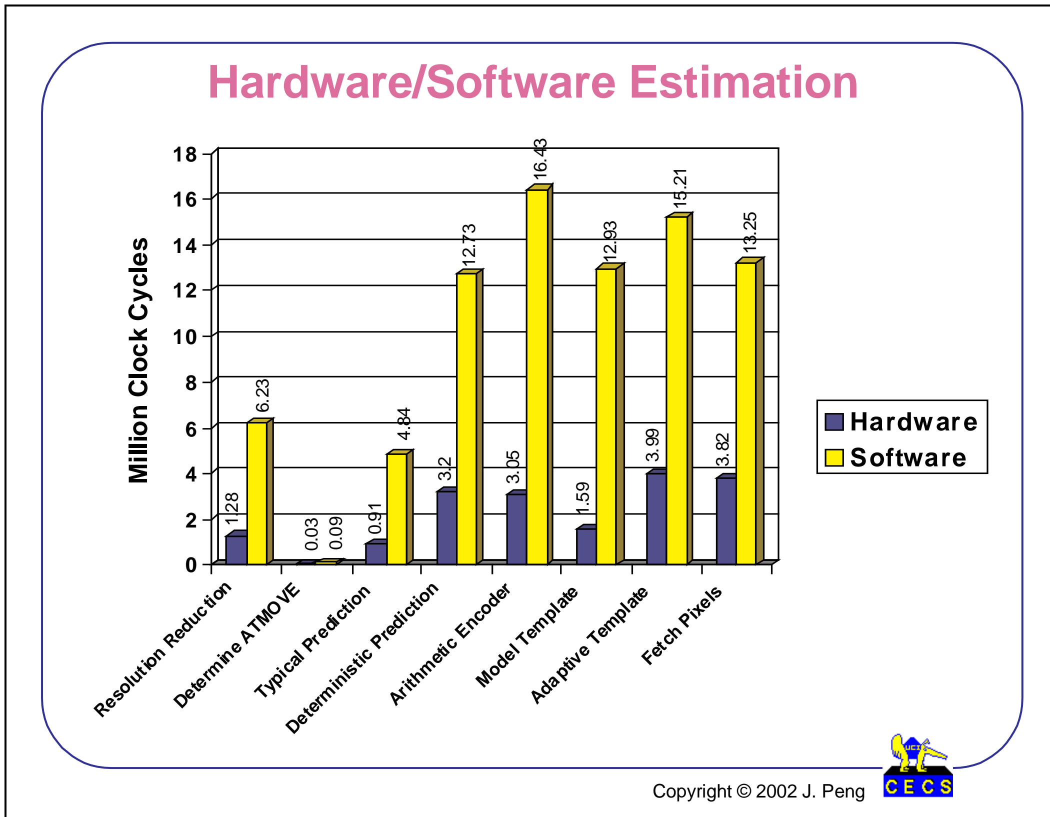


JBIG stands for "Joint Bi-level Image experts Group". JBIG is one of the image compression/decompression standards. Facsimile is one of its many applications.
Specifically, it is a lossless progressive encoding of a bi-level image (an image that has only two colors, like black-and-white). It is lossless because the decoded image is identical to the original without any distortion. The progressive capability enables the transmission of the image with different resolutions over networks. As shown in the figure, when this feature is enabled, a series versions called layers (I_{D-1}, $I_{D-2, \ldots,}$ I_0) of the original input image (I_D) at reduced resolutions are extracted, encoded and transmitted separately.
The specification for the JBIG standard is available as "ITU-T Recommendation T.82". There are several software implementations of JBIG available in the public domain. The one adopted in our design can be found at `ftp://ftp.informatik.uni-erlangen.de/pub/`.

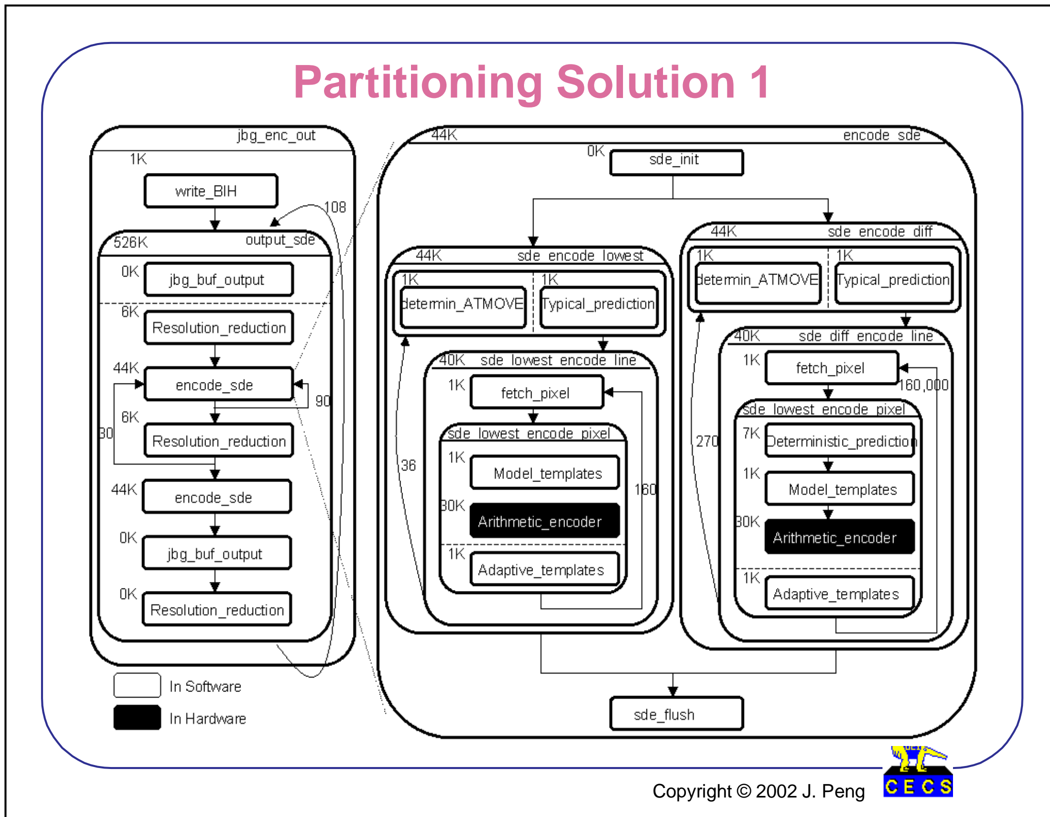


The behavioral hierarchy as well as some profiling information with a test image of the derived Specification model in SpecC are shown above.

The whole of JBIG functionality has been encapsulated as *jbg_enc_out*. It is composed of two sub-behaviors *write_BIH* which writes the header information and *output_sde*. *output_sde* is further composed of *Resolution_reduction* described earlier, *encode_sde* which encodes and buffers the code for one horizontal stripe of pixels and *jbg_buf_output* which writes the buffered code to file upon the completion of that stripe. Inside of *encode_sde*, one of two similar but mutual-exclusive sub-behaviors *sde_encode_lowest* (lowest layer) and *sde_encode_diff* (differential layer) is selected to run. *sde_encode_lowest* is composed of two leaf behaviors, *determin_ATMOVE* and *Typical_prediction,* and *sde_lowest_encode_line* which encodes one line of pixels within the stripe at a time. *sde_lowest_encode-line* is further composed of *fetch_pixels* which extracts the current pixel out of the line and *sde_lowest_encode_pixel* which encodes one pixel at a time.



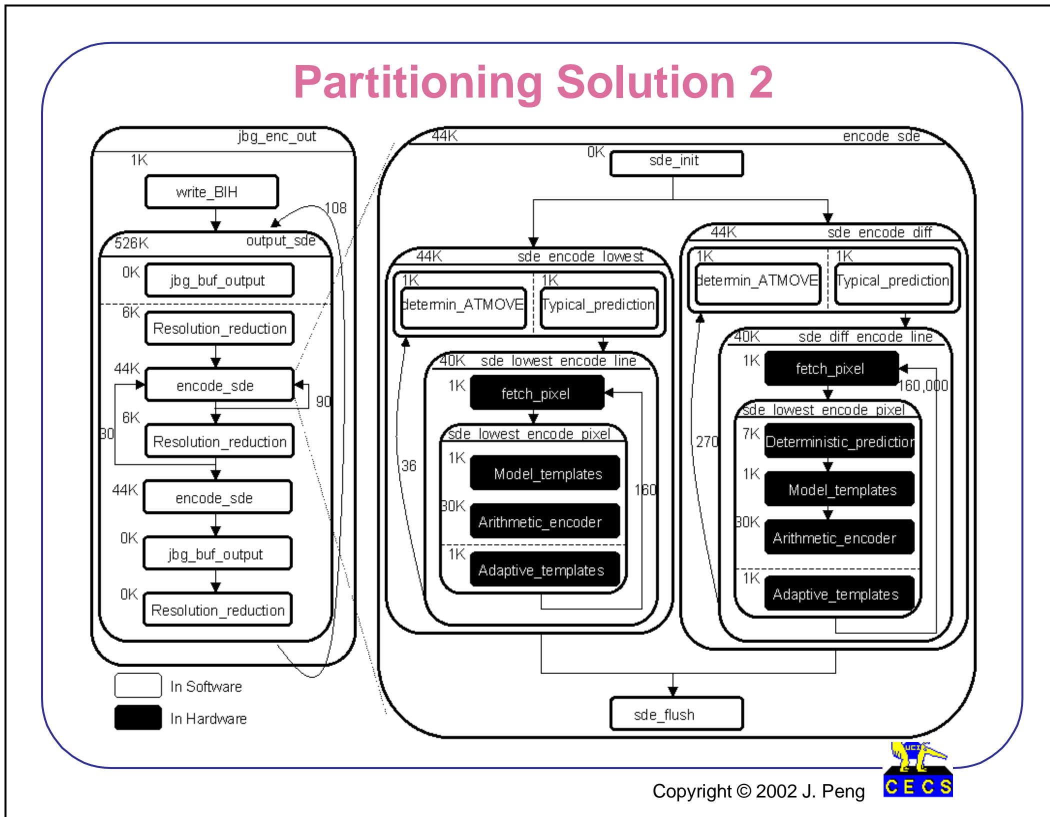
Before we started SW/HW partitioning of the behaviors, a coarse HW and SW performance estimation for different function blocks was performed to give a distribution of computation amount.
In our approach, the function blocks are further divided into basic blocks to statically estimate clock cycles for each basic block.
For estimation of SW implementation, the clock cycles needed to execute a given basic block are obtained by summing up the clock cycles for all instructions compiled for the target processor.
Estimation of HW implementation is a little different. First, an RTL architecture that implements the functionality of the basic block is sketched. Based on the architecture, a FSMD model can be derived. Thus, the number of clock cycles for executing a given basic block in hardware can be found by counting the number of states within that basic block.
Then, we can find out how many times each basic block was executed by simulating the models with a test image.
For each basic block i, if the number of clock cycles is C_i and the number of executions is N_i, the total estimation of SW (or HW) implementation is given by $\sum C_i * N_i$.



Knowing the HW/SW performance of each function block, now we can consider various HW/SW partitioning solutions. Since each behaviors could be implemented either in hardware or in software, theoretically the number of possible partitions is too huge to handle although most of them are not good in terms of quality. Therefore we need some heuristics to construct those promising solutions rather than to enumerate all possible partitions.

First, behaviors that have the most intensive computation are put into hardware to obtain speed gain. Later on, other behaviors can be added into hardware gradually to further speed up the design. The selection of these additional behaviors is based on the profiled information of the specification, which includes the communication between behaviors, behavior hierarchy, loop counts, data-sharing among behaviors and so on.

In the first solution, only the *Arithmetic_encoder* behaviors are implemented in hardware. The rest of the system is to be executed in software. This decision was made based on the estimated complexity of the Arithmetic_encoder and its high calling frequency, while minimizing hardware cost.



Copyright © 2002 J. Peng

As shown in the above figure, composite behaviors *sde_lowest_encode_line* and *sde_diff_encode_line* are completely implemented in hardware.

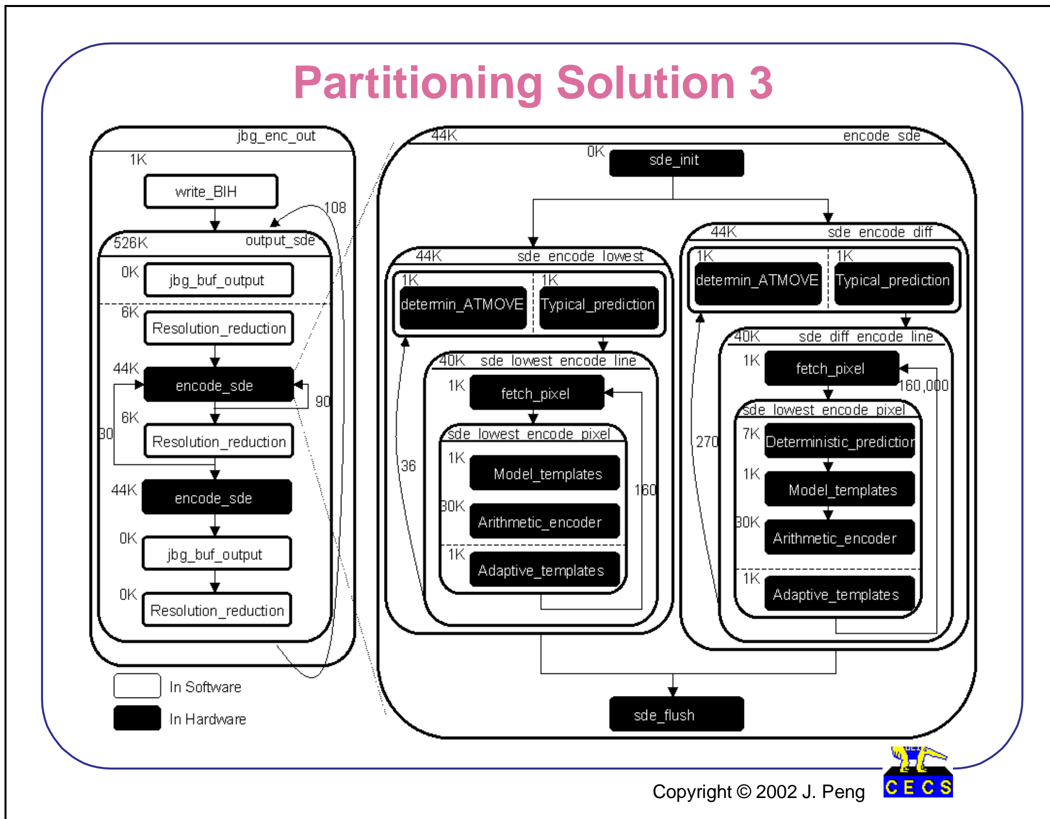
The reasons for making this partitions are following.

First, either behavior is composed of a group of closely related and connected sub-behaviors which indicate data-sharing among them.

Secondly, these two behaviors have similar behavior hierarchy and achieve similar functionality of encoding one line of pixels at a time. The mutual-exclusive execution between them suggests a resource-sharing implementation.

In addition, their sub-behaviors all contribute significantly to the total computational complexity, as shown in the earlier estimation.

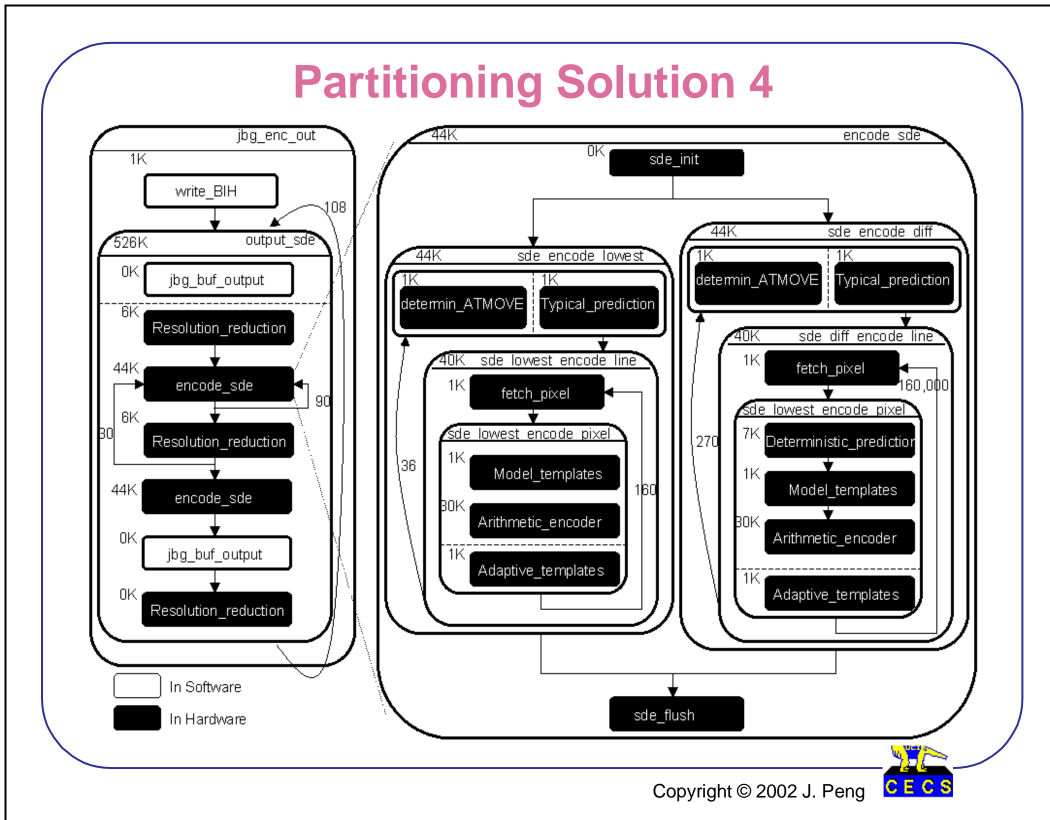
Finally, there is a frequently iterated loop inside each of them therefore we don’t want to make a SW/HW dichotomy inside such a loop since that would introduce large amounts of SW/HW communication overhead.



In the third solution, we decided the whole *encode_sde* behavior to be implemented in hardware.

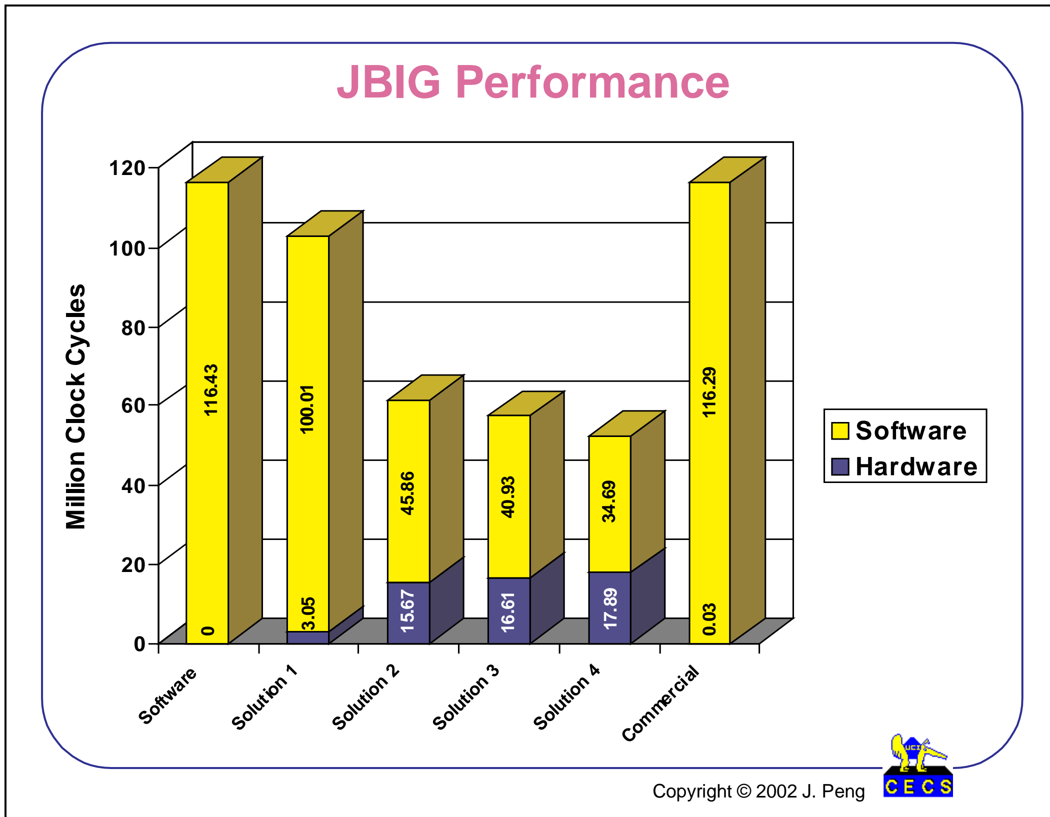
The reasons for this partition are similar to those for Solution 2. *encode_sde* is a relatively independent function block that encodes a whole stripe of pixels. It includes many computational behaviors therefore it is profitable to speed up its execution with a hardware implementation. It also includes a loop that is iterated quite often which discourages a SW/HW split of *encode_sde*.

The drawback, of course, is the increased complexity and cost for implementing the hardware.



In the fourth solution, we put almost everything but two behaviors (*write_BIH* and *jbg_buf_output*) into hardware. These two behaviors are executed in SW because their function is writing the code out to the output file, which we don't want to handle in hardware.

Compared to the third solution, Resolution_reduction is also added to hardware in the hope to further speed up the execution of multi-layer encoding.



The performance of each of the partitions are evaluated and shown in the above figure. The graph shows the total performance for each partition and how it is composed out of time spent in the software and hardware side. Since we used clock cycles as our measure, we need to normalize SW and HW clock cycles. We found the critical path in the HW RTL architecture to be <12ns which was then used as the HW clock. We assume the clock rate of the target processor to be 166Mhz which gives a clock of 6ns. Therefore each HW cycle is normalized to 2 SW cycles.

As expected, the pure SW solution takes the maximum execution time, since software tends to be slower than an ASIC. As seen in the above figure, the performance in terms of speed goes up when more blocks are put into HW. Solution 4 yields the best performance since the maximum number of blocks are implemented in HW.

Communication & Storage

- **Communication requirements:**

	Software	Solution 1	Solution 2	Solution 3	Solution 4
SW ⇒ HW	0	2 B	175 B	3 kB	70 kB
HW ⇒ SW	0	2 B	150 B	4 kB	432 kB
HW ⇔ SW	0	4 B	325 B	7 kB	502 kB
Transfers	0	138,355	2,126	108	1
Total	0	550 kB	690 kB	756 kB	502 kB

- **Memory requirements:**

	Software	Solution 1	Solution 2	Solution 3	Solution 4
Hardware		30 kB	39 kB	44 kB	54 kB
Software	526 kB	496 kB	487 kB	482 kB	472 kB
Total	526 kB	526 kB	526 kB	526 kB	526 kB

Copyright © 2002 J. Peng

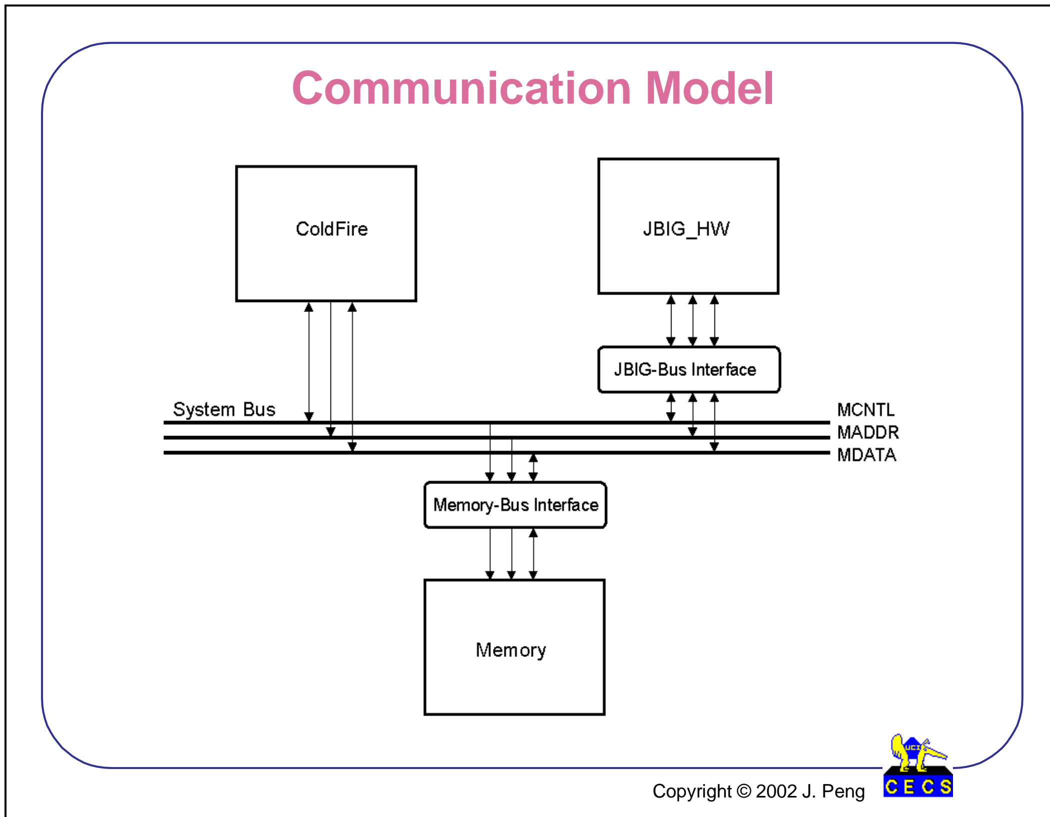
The other metrics we need to look at include SW/HW communication overhead and memory requirement.
The communication overhead was computed in terms of the total number of bytes of data that may need to be transferred. First, we compute the bandwidth of one data transfer between SW and HW. Then we find out how many times such a transfer needs to occur. The total communication is computed by multiplying these two quantities.
The memory size requirement of both HW and SW for each partition was also tabulated. As more blocks are implemented in HW, the amount of HW memory increases as well, but not too dramatically.
All these metrics are needed by the designer in order to choose appropriate partitions for a variety of design constraints.

Selection of Partitioning

- Arithmetic encoding is computation intensive in one-layer-encoding
- Solution 1 minimizes hardware with comparable performance to other solutions

Copyright © 2002 J. Peng

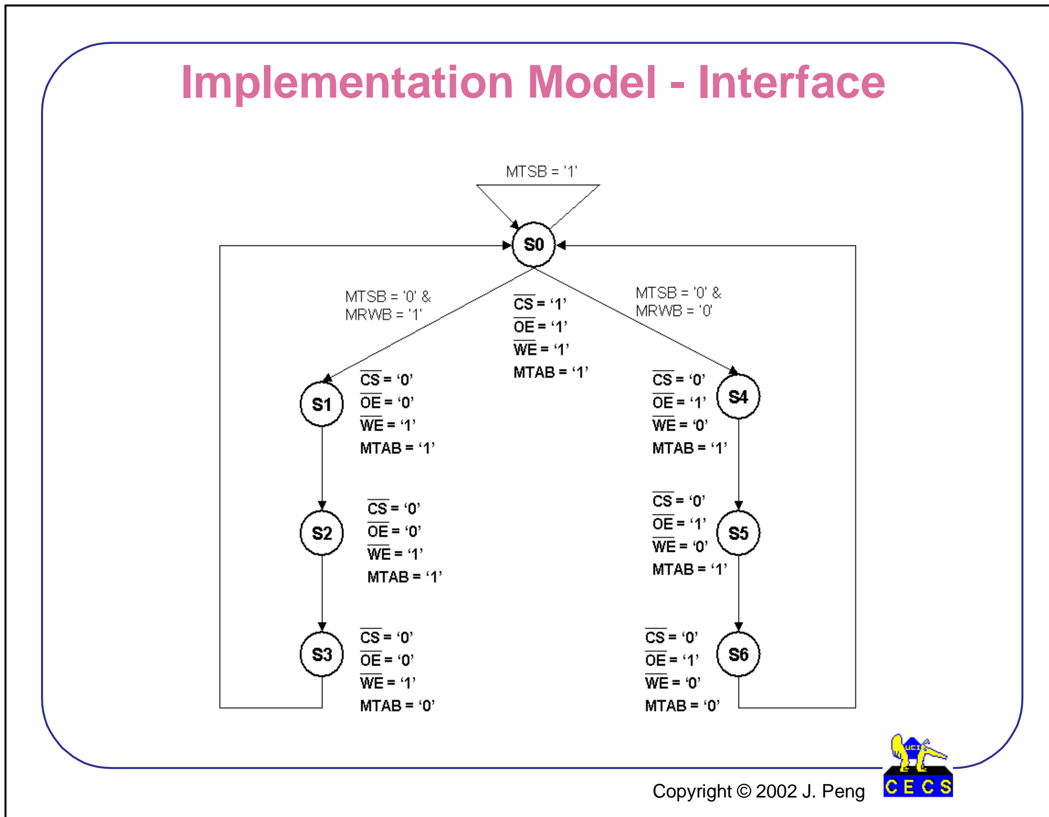
Among all these solutions, we had to choose one of them and proceed in our SpecC methodology. To simplify things, we have chosen Solution 1 since our goal was to verify the SpecC methodology using this JBIG example.
Solution 1 only has the arithmetic encoder block in hardware, but it does give comparable performance to other solutions despite of the minimum hardware it needs. Some JBIG applications, such as FAX, only use one-layer-encoding which relies on arithmetic encoder even more heavily. That is why the existing design mentioned earlier has similar partition to Solution 1.



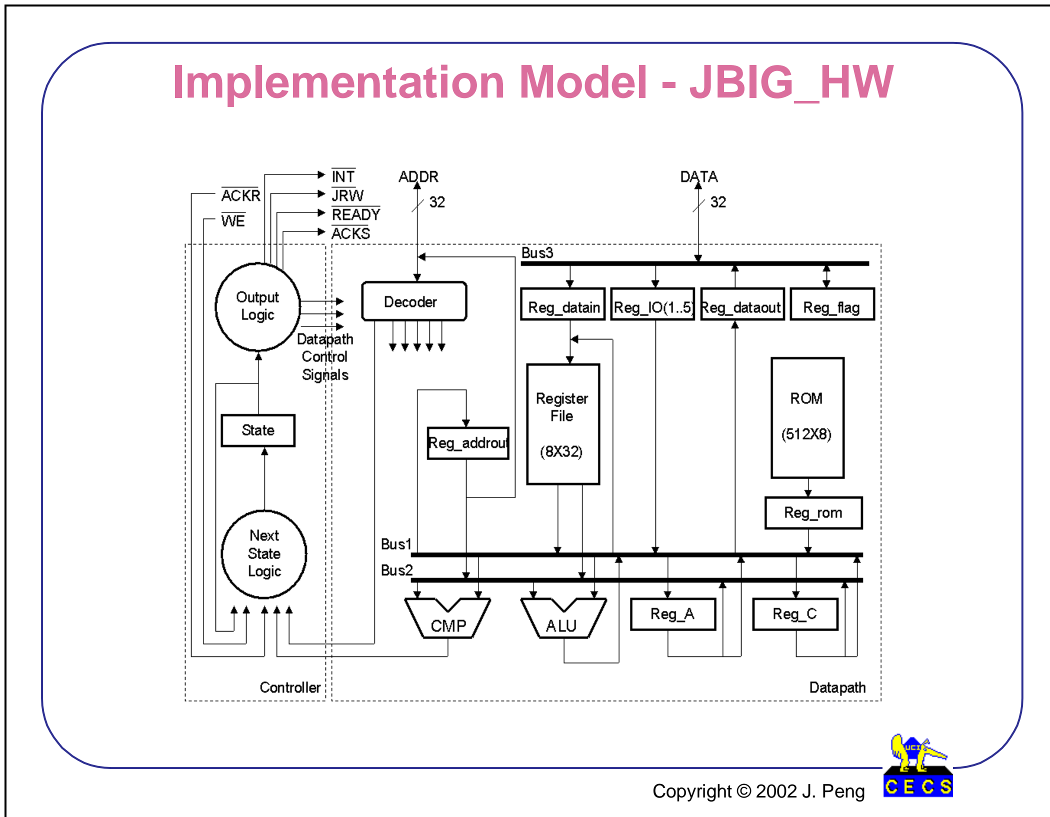
The communication model is shown above. Three components, *ColdFire*, *JBIG_HW* and *Memory* are all connected to *System Bus*.

System Bus uses ColdFire processor's protocol. It consists of control bus *MCNTL*, address bus *MADDR* and data bus *MDATA*. *MCNTL* includes control signals, such as transaction start (MTSB) and read/write signal (MRWB).

All components on the same bus must be clocked at the same speed. The clock of *System Bus* is *6ns*, which is different from the access time of *Memory* (*12ns*) and the clock of *JBIG_HW* (~12*ns*). Therefore, bus interfaces are inserted for both *Memory* and *JBIG_HW* to resolve the timing mismatches.



In order to obtain the final implementation model, the bus interfaces and the hardware JBIG_HW have to be synthesized. We will discuss the synthesis of the interfaces first.
Once we have the timing diagrams for the signals on both sides of the interface, a finite-state-machine (FSM) is constructed to generate desired signals at appropriate time. The clock of the FSM is the same as the bus clock.
The FSM for the Memory-Bus Interface is shown in the figure. At the initial state *S0*, the memory access is disabled. When *MTSB* (transfer start) is asserted, it goes to *S1* or *S4* depending on the value of *MRWB* (read/write). At *S1*, memory control signals are generated. Because the read cycle is *12ns* and the clock is *6ns*, at least one more state (*S2*) is needed to maintain the values of those memory signals.
In the last state (*S3*), *OE* is asserted to enable the output of data and *MTAB* is asserted to notify *ColdFire* that data is valid. The process is similar for a write transaction.
The finite state machine then can be fed into any high-level synthesis tool to produce gate netlist of the interface.
The synthesis of the JBIG_Bus Interface can be repeated in a similar way.



The RTL architecture of the ASIC component *JBIG_HW* is sketched above. It consists of two parts: a *Datapath* and a *Controller*.

The *Datapath* is composed of three types of components: functional units (*CMP*, *ALU*, and *Decoder*), storage units (*Register File*, *ROM* and registers) and interconnection units (*Bus1*, *Bus2* and *Bus3*). The *Decoder* and 5 *Reg_IO* registers are included to enabble the communication between *ColdFire* and *JBIG_HW*. *ColdFire* can access these registers directly using memory read/write instructions by mapping them to the address space of *ColdFire*. The *Decoder* is used to decode the address on the system bus to enable access to the addressed registers.

The *Controller* is composed of a *State* register which stores the current state, a *Next State Logic* unit which decides the next state and an *Output Logic* unit which generates *Datapath* control signals for the current state.

Conclusions

- SpecC methodology demonstrated with a real example.
- SpecC is capable of transforming an executable specification into a RTL implementation.
- Basic refinement rules are established and applied to transform SpecC models.
- With the availability of SpecC automated tools, productivity gain are achievable of 100X over the manual approach.

Copyright © 2002 J. Peng

In this presentation, we illustrated our design flow through the design of the JBIG encoder. It shows that the design flow is capable of refining an executable specification into its RTL implementation.

Architecture exploration starts with a profiling on the specification and initial estimation of pure SW and HW implementation. Among several SW/HW partitions, one solution was selected to determine the system architecture. The SpecC function model was refined into an architecture model to reflect the design decisions.

Communication synthesis refines the abstract communication in the architecture model into signal exchanges over actual wires. Interfaces are generated to bridge incompatible timing protocols. The refined model is the communication model.

Finally, with the compilation of the software component for the selected processor and refinement of the hardware component into behavioral or structural model that can be synthesized using high-level synthesis tools, the implementation model was derived and ready for manufacturing.

At the time of this project, the SpecC tools were not available, therefore most of the work was done manually. We can expect a 100X speed up for the design cycle with the help of these tools when they are available.

References

1. A. Gerstlauer, R. Dömer, J. Peng, D. D. Gajski, *System Design: A Practical Guide with SpecC,* Kluwer Academic Publishers, Boston, MA, ISBN 0-7923-7387-1, June 2001.

2. D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, Boston, MA, ISBN 0-7923-7822-9, March 2000, 336 pages.

3. Junyu Peng, Lukai Cai, Anand Selka, Daniel D. Gajski, *Design of a JBIG Encoder using SpecC Methodology*, UC Irvine, Technical Report ICS-TR-00-13, June 2000.

4. L. Cai, J. Peng, C. Chang, A. Gerstlauer, H. Li, A. Selka, C. Siska, L. Sun, S. Zhao and D. Gajski, *Design of a JPEG Encoding System*, UC Irvine, Technical Report ICS-TR-99-54, November 1999.

5. Andreas Gerstlauer, Shuqing Zhao, Daniel D. Gajski and Arkady M. Horak, *Design of a GSM Vocoder using SpecC Methodology*, UC Irvine, Technical Report ICS-TR-99-11, March 1999.

6. Andreas Gerstlauer and Daniel D. Gajski, *System-Level Abstraction Semantics*, Proceedings of International Symposium on System Synthesis, Kyoto, Japan, October 2002.

7. Junyu Peng and Daniel D. Gajski, *Optimal Message-Passing for Data Coherency in Distributed Architecture*, Proceedings of International Symposium on System Synthesis, Kyoto, Japan, October 2002.

8. W. Mueller, R. Dömer, A. Gerstlauer, *The Formal Execution Semantics of SpecC*, Proceedings of International Symposium on System Synthesis, Kyoto, Japan, October 2002.

9. Rainer Dömer, *The SpecC System-Level Design Language and Methodology, Part 1*, Embedded Systems Conference, San Francisco, March 2002.