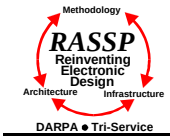
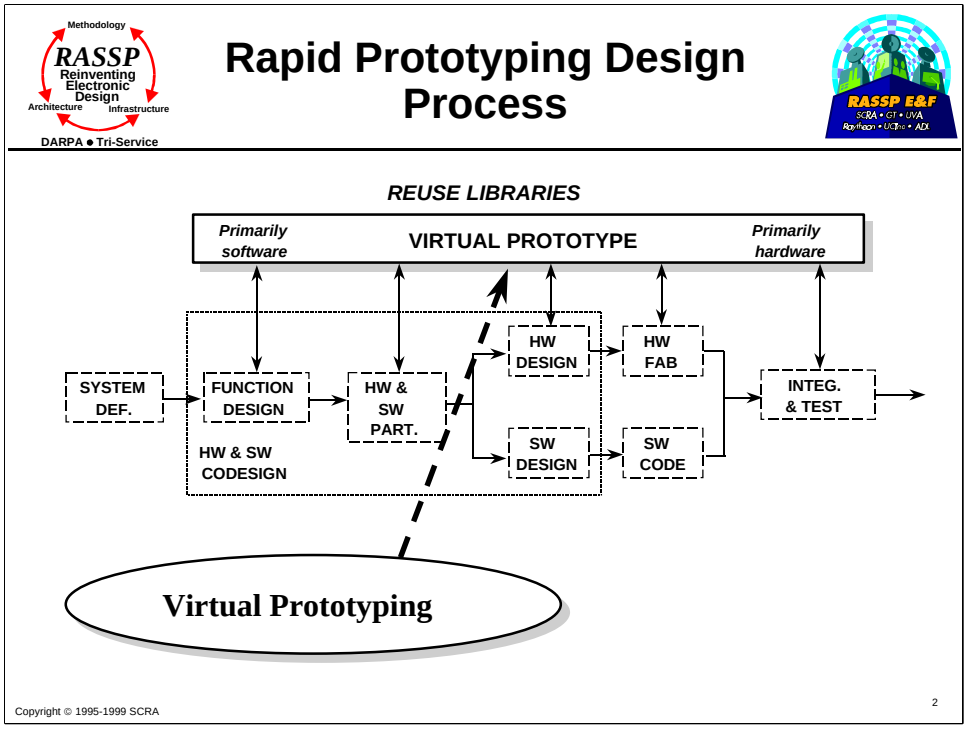


Virtual Prototyping Using VHDL

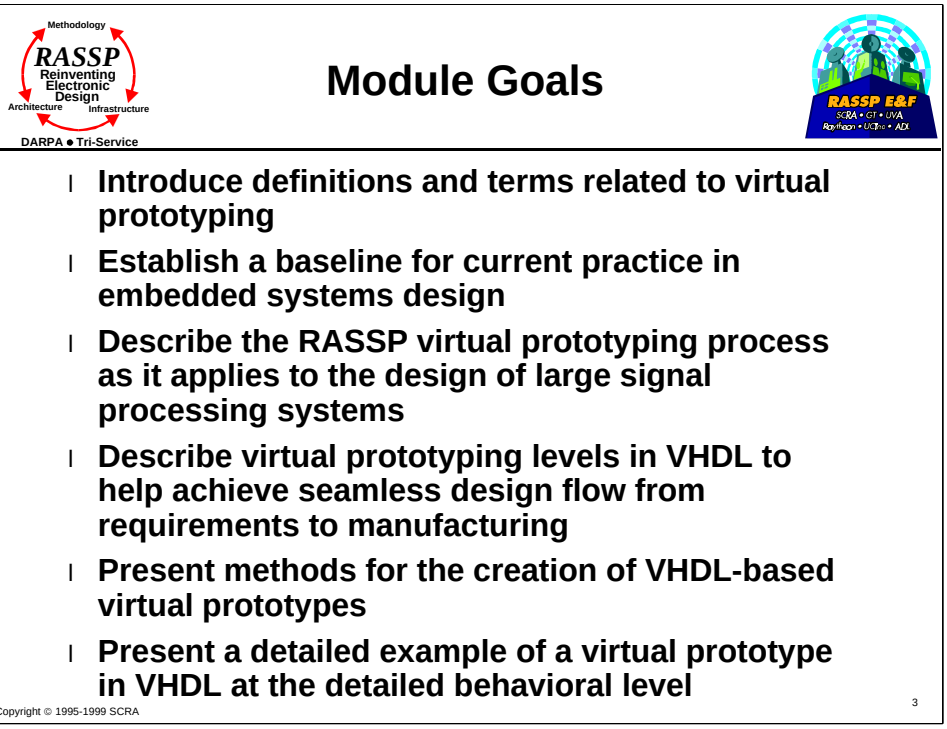
RASSP Education & Facilitation Program

Module 32

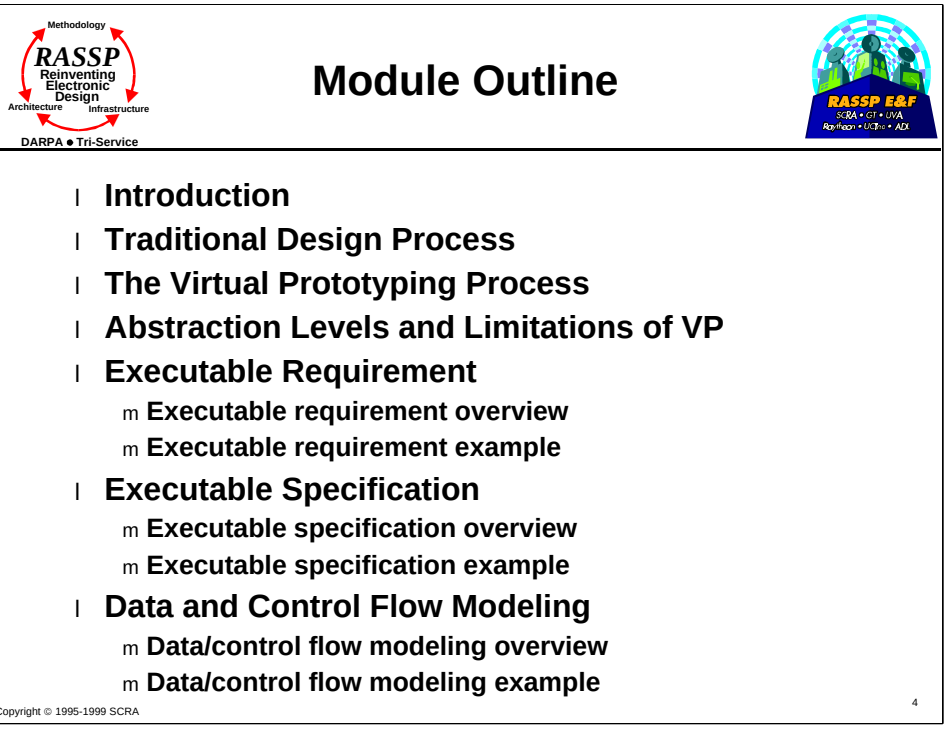
Version 3.00



The RASSP roadmap shows the importance of virtual prototyping in the design process. The virtual prototype carries through the entire process from requirements specification (executable) to HW and SW integration and test. This short course will focus on the entire process with a detailed example of lower level virtual prototyping presented at the end.



This slide presents the goals to be achieved by this module. We start with a presentation of traditional design processes and how the Defense Advanced Research Projects Association (DARPA) Rapid Prototyping of Application Specific Signal Processors Progpam (RASSP) attempted to improve upon it by using the virtual prototyping process. This is followed by a description of the process with emphasis on the use of VHDL to achieve its goals. Small examples will be presented for each section with the final section containing a more detailed example of the use of VHDL to model at the detailed behavioral level.



This module will cover the areas listed above. It will start with a review of traditional design processes based on a survey done by the RASSP Education and Facilitation (E&F) team.

The Virtual Prototyping (VP) process will then be described, focusing on the various levels of virtual prototyping. These include the formation of executable requirements, executable specification, data flow modeling, performance modeling, and more detailed modeling of components and systems.

Important documents and standards will be mentioned because of their applicability to virtual prototyping with respect to VHDL-based methods.

Abstraction levels and limitations of VP will be discussed.

A case study of an example design using an i860XP processor with a memory, memory controller, DMA, and VME bus will be described in detail.

A summary will then be provided.

Module Outline (cont.)

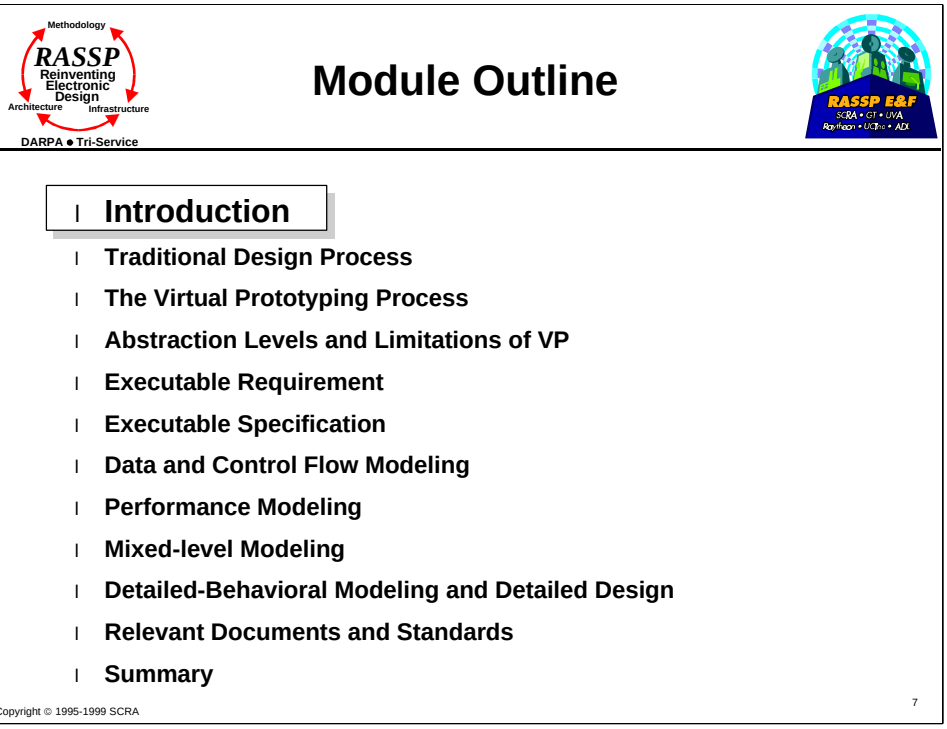
- **Performance Modeling**
  - **Performance modeling overview**
  - **Performance modeling in the architecture design process**
  - **Components for performance modeling**
  - **Laboratory introduction of a VHDL example of a simple performance model Mixed-level Modeling**
- **Detailed-Behavioral Modeling and Detailed Design**
  - **Overview of detailed-behavioral modeling in VHDL**
  - **Design example**
    - **Overview of example**
    - **Modeling the Intel i860XP processor**
      - **Overview of the processor model**
      - **The internal model**

Module Outline (cont.)

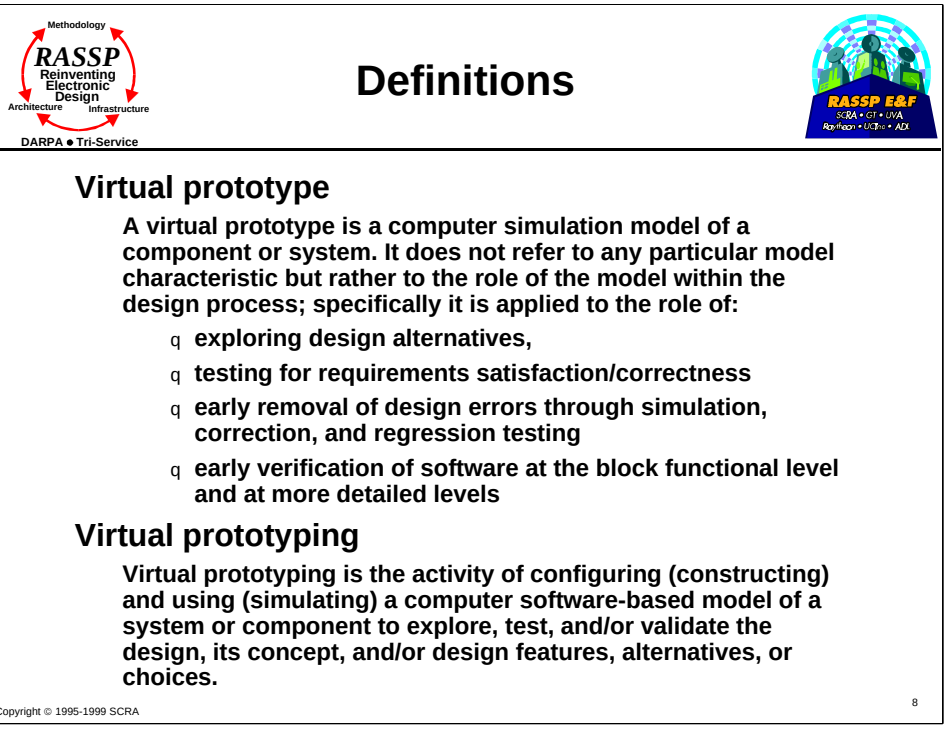
- The interface model
- The testbench
  - Clock/reset generator
  - Memory controller
  - Memory
- Testing the Intel i860XP
- Testing the Mercury Computer Systems MCV9
- Testing the Lockheed Martin-Sanders Infra-Red Search and Track (IRST) system
- Results

- Creating a DMA to the VME in the memory controller
- Simulation results

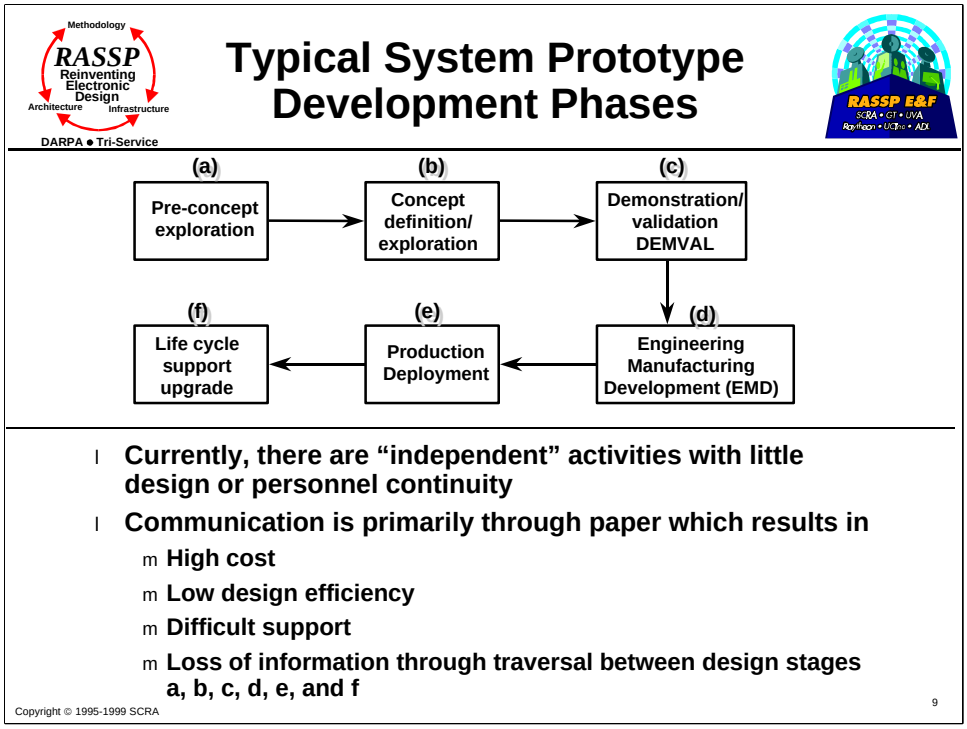
- **Relevant Documents and Standards**
- **Summary**



We now begin the module by introducing the definitions and high-level concepts of virtual prototyping and how a virtual prototype is defined.



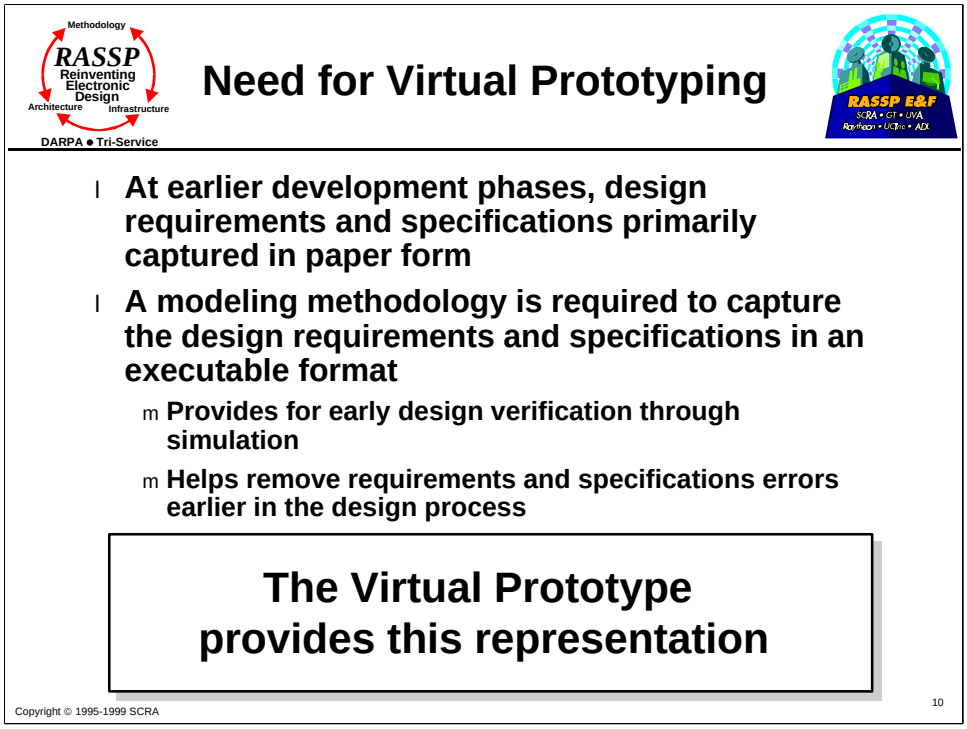
This slide answers the questions of what is a virtual prototype and what is the VP process. Virtual prototypes are used to explore design alternatives at a number of levels in the design process. At the highest levels, the choice of algorithms can be explored to find the best algorithm that meets the functional requirements, while minimizing the amount of computation. As lower levels are developed, possible implementation architectures are explored to determine the most effective solution that can implement the proposed algorithms (cost, size, power, weight, etc.). After a specific architecture is chosen, virtual prototypes of the hardware and the software can be co-simulated to verify proper operation long before the actual hardware is created. In the entire process, verification is done at each level to ensure that the system requirements are being achieved.



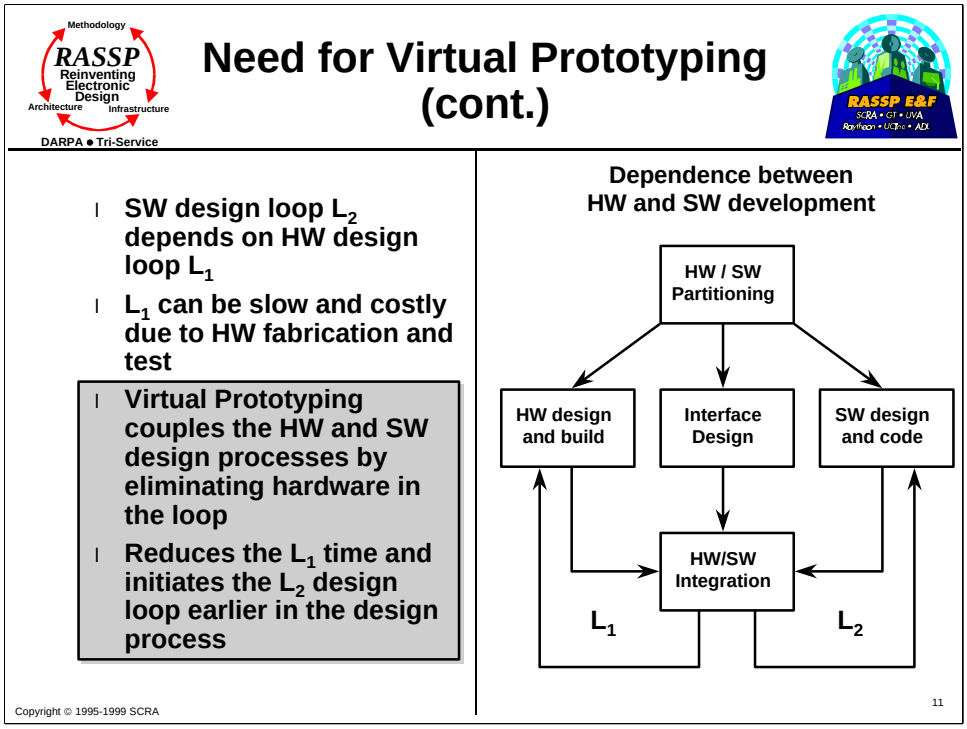
Phases of system development effort:

- Pre-concept exploration/feasibility
- Concept definition/exploration/design
- Demonstration of design validity
- Development
- Production
- Life cycle maintenance

There is little design continuity in the current practice, where communication is done mainly through paper which results in higher cost, lower efficiency, and difficult support mechanisms (i.e. legacy upgrades).



Virtual prototyping provides a means for capturing information in an executable form at the earlier design phases in the previous slide. This provides a method to verify the design requirements and specifications prior to commitment to actual hardware prototyping. It also provides a method to remove errors in the design, prior to the final commitment of the design to silicon (in the case of device modeling) or oversized systems, where the number of processors used may be twice the number actually needed (system sizing error). Through virtual prototyping at various levels of component or system abstraction, the most appropriate and error free design can be realized before committing excessive resources.

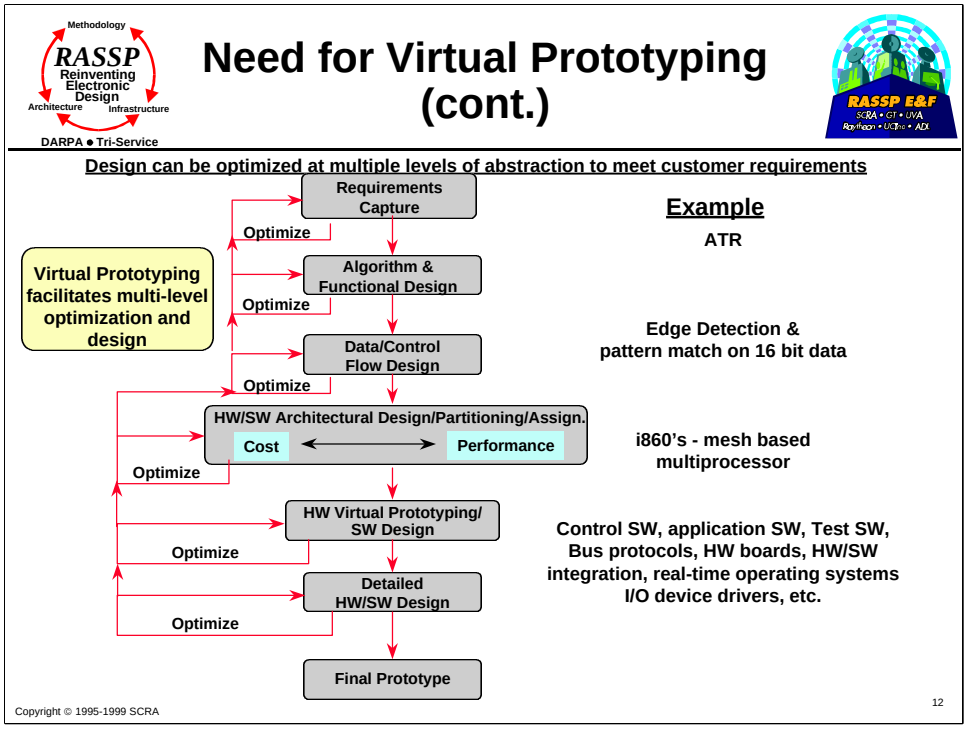


Current practice in HW/SW design/integration requires SW loop L2 to depend on HW loop L1.

L1 can be slow and costly due to HW fabrication and test.

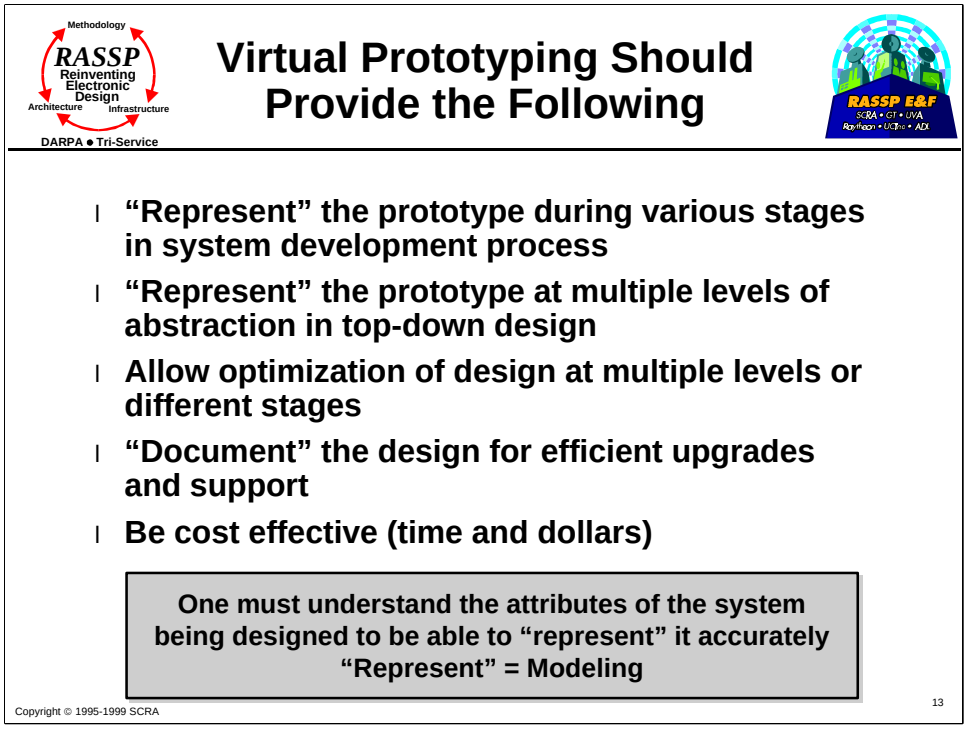
Virtual prototyping allows this codevelopment to occur concurrently through the use of:

- VP's of HW that can run SW or estimate the time-line of events of SW running on a large system
- SW code generation from data flow graphs
- Rapid architecture selection trade-offs using performance models
- Evolving models that capture requirements with refining detail as the design progresses



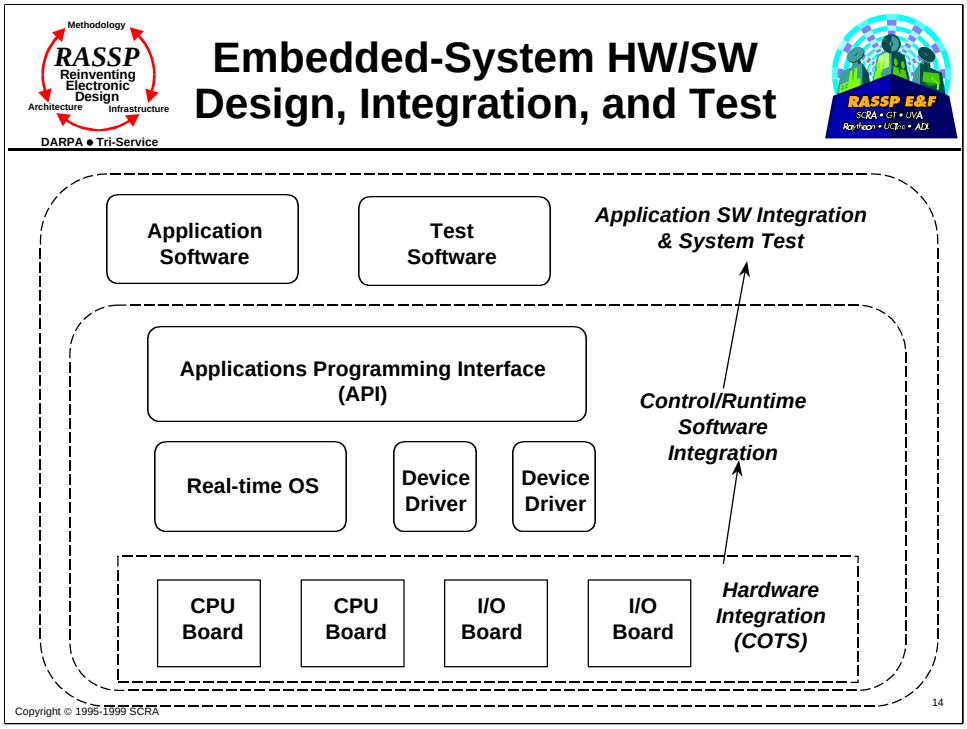
The VP process is a top-down design paradigm with optimization done at multiple levels of abstraction. There are a number of slides in this module that capture the essential flow of this design process, but in a different format. In general, the first stage in any design process is the correct definition of the requirements for the design. Once the customer's requirements have been understood, then the system design team begins working on design specifications that attempt to satisfy all the customers' requirements. In the above process flow, this represents the algorithm definition and functional design stage. At this stage, the algorithms used to implement the functions are refined. The size, weight, area, and power constraints may also be refined as well. At this point, initial computational complexity of the algorithms, as well as fixed-point characteristics, are analyzed. As the process proceeds from the top level, further refinement is done. At the data flow level, the algorithms are refined and data flow graphs that implement the functionality are explored. These data flow graphs are used in the HW/SW architectural design phase, where the nodes in the flow graph are either mapped to hardware or software. At this level in the process, the hardware and/or software are sized to meet the requirements. Timing critical information is captured such as latency, throughput, and resource utilization. As more detailed design is done, the architectural elements are either development as software modules or hardware components (ASICs, Processors, etc.) At the end of the process, final hardware and software are realized. At each stage in the process, the design is verified and optimized to meet the customer's requirements.

Each level captures critical design information appropriate to that level.



Five key elements that VP must provide.

Each stage of the design has its own set of important attributes that must be captured with the proper representation through modeling.



The VP process spans multiple levels and multiple user's viewpoints.

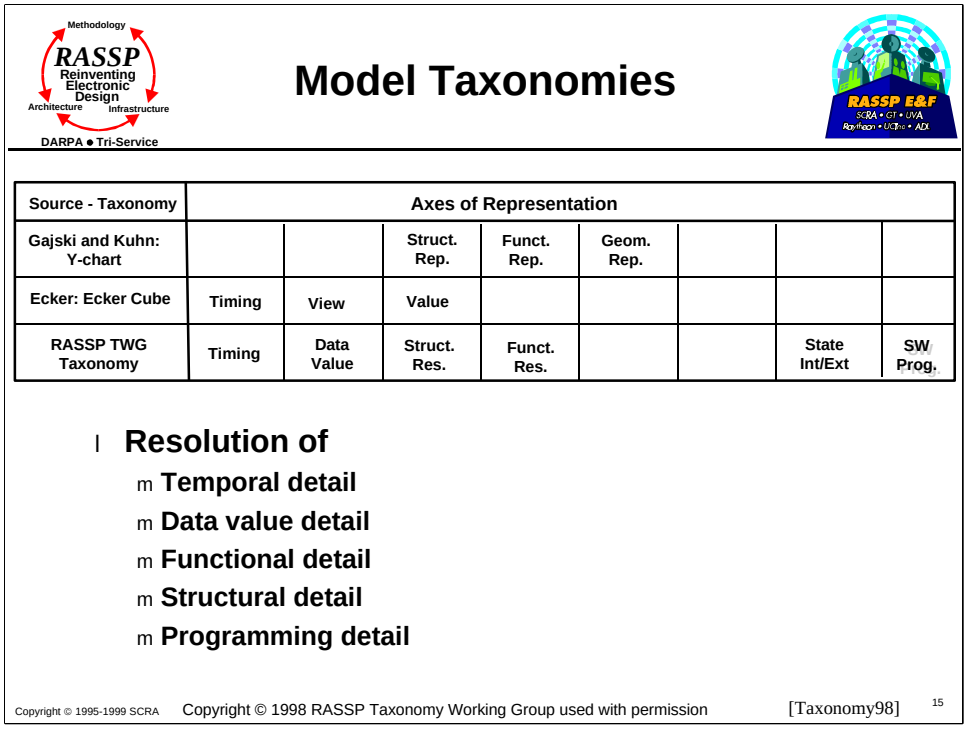
At the lowest level of HW integration, we have HW design being done and one would typically see the following being used:

- Full-behavioral/Interface and RTL level models of application specific and COTS parts being modeled
- Interconnection between devices are tested

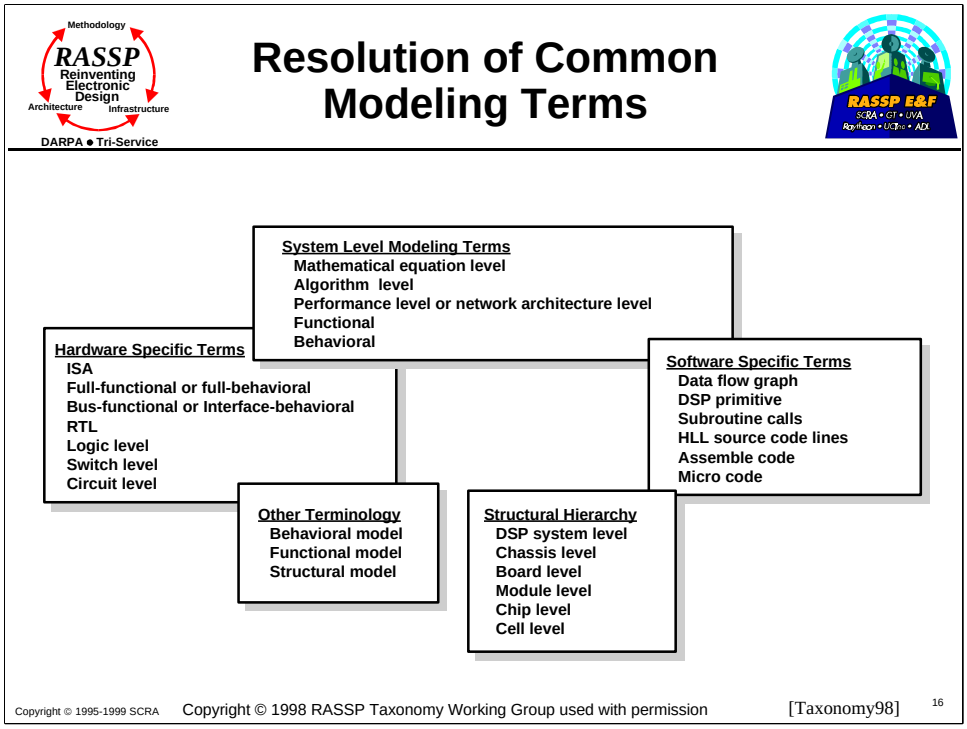
The next level integrates the OS SW and application interface to the HW system. This is where one would see SW running on the HW VPs to make sure the device drivers work as expected.

At the highest level, application and test code is integrated and tested at the system level. Performance level models help determine the number of processing boards required. Detailed-behavioral models help ensure test SW can perform its functions at the node level. Executable specification helps determine the application SW.

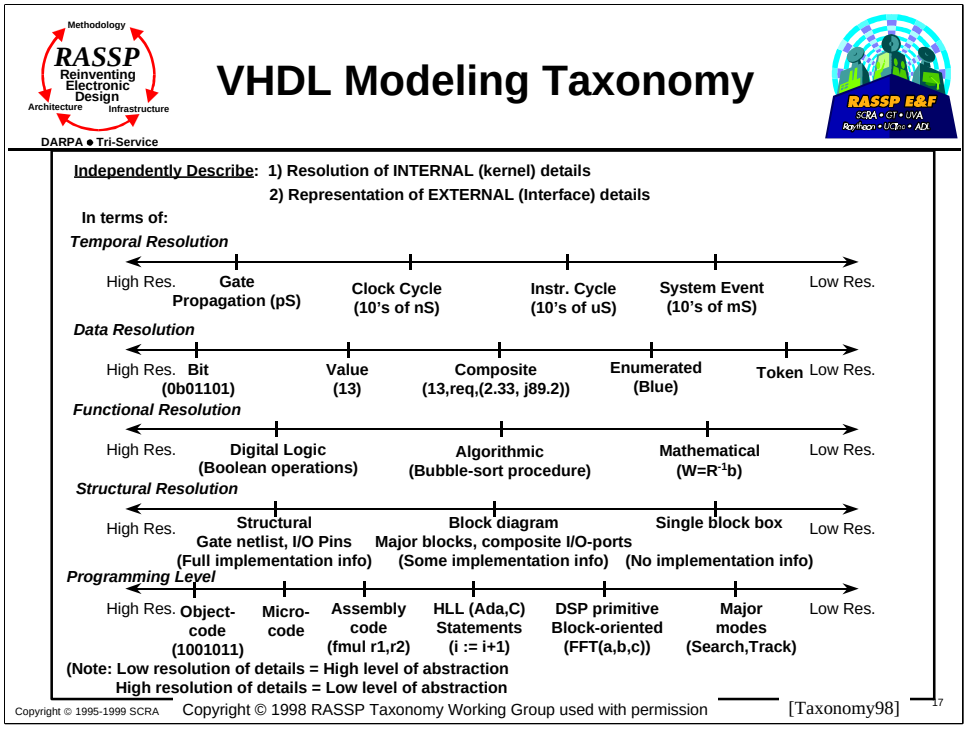
The VP process covers all these domains.



This chart shows the various taxonomies used to describe digital systems. The Gajski and Kuhn chart represented information about a system along three axes: structural, functional, and geometrical. The Ecker cube defined a system using three attributes; the timing information contained within the system model, the view of the model, structural, dataflow, or behavioral and the value representation of the data within the system, e.g. bits, abstract data types, etc. This has steadily been improved with the RASSP Taxonomy to include programming information, timing, structural, functional, and data value information.The RASSP taxonomy will be covered in more detail on later slides.



A number of terms has been bandied about for describing digital systems at various levels of abstraction and a categorization of some of the terms is shown in this slide. The five broad categories included hardware specific terms, software specific terms, system-level modeling terms, structural information descriptions, and other terminology that did not fit into the other categories. Give the diverse set of terms used to describe information within a design process, the RASSP taxonomy was developed. Later slides will cover this in more detail.



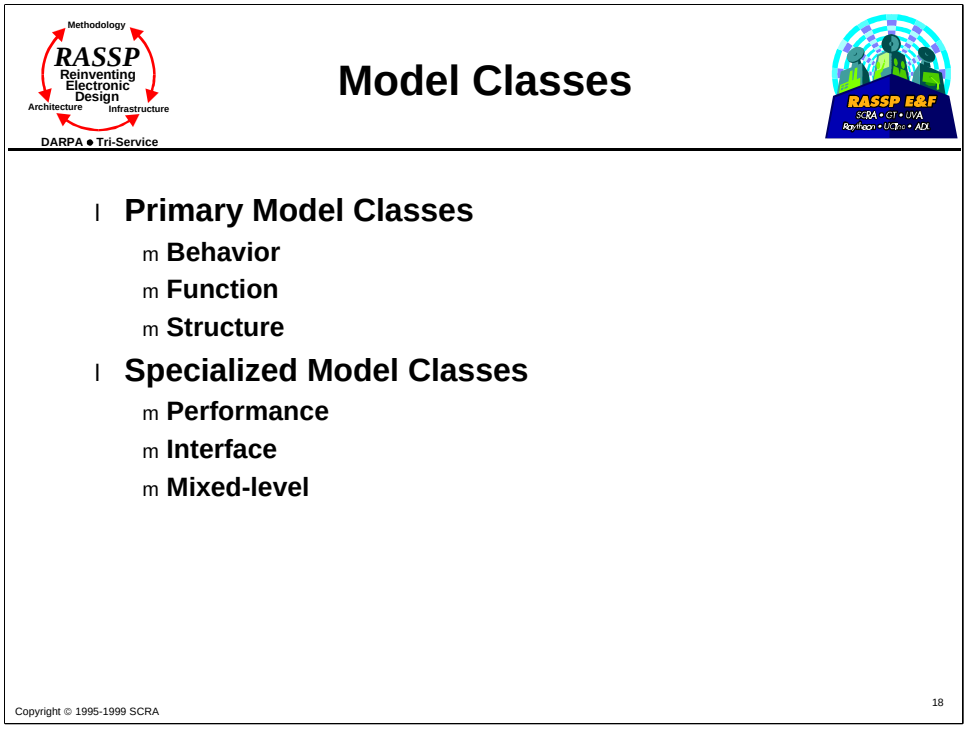
To represent what is meant by each level of the top-down design, we need a method of capturing the information of the models.

Five axes are used to represent:

- Time
- Data
- Function
- Structure
- Programmability

Both internal and external behavior is represented.

For example, time can be represented from low to high resolution. At the high level, we represent gates and at the low level purely functions (i.e. no timing). See the taxonomy document (Taxonomy98) on clarification of terms in the above scheme.



All models can be described in terms of one or more of the five primary model classes. The specialized model classes describe models intended for specific purposes that are not unique to a particular level of abstraction.

A behavioral model describes function plus timing for a specific implementation.

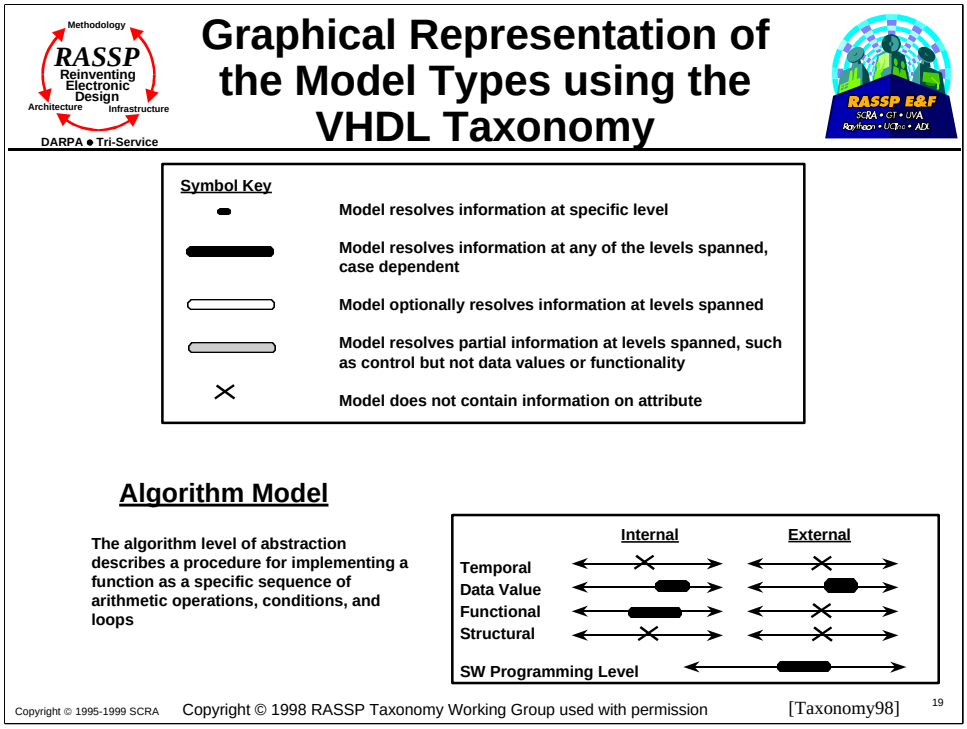
Functional models describe the function of a component or system without describing a specific implementation.

Structural models represent the component or system in terms of the interconnections of its constituent elements.

Performance models measure the quality of the design related to the timeliness of the system in reacting to stimuli.

Interface models is a component model that describes the operation of the component with respect to its surrounding environment.

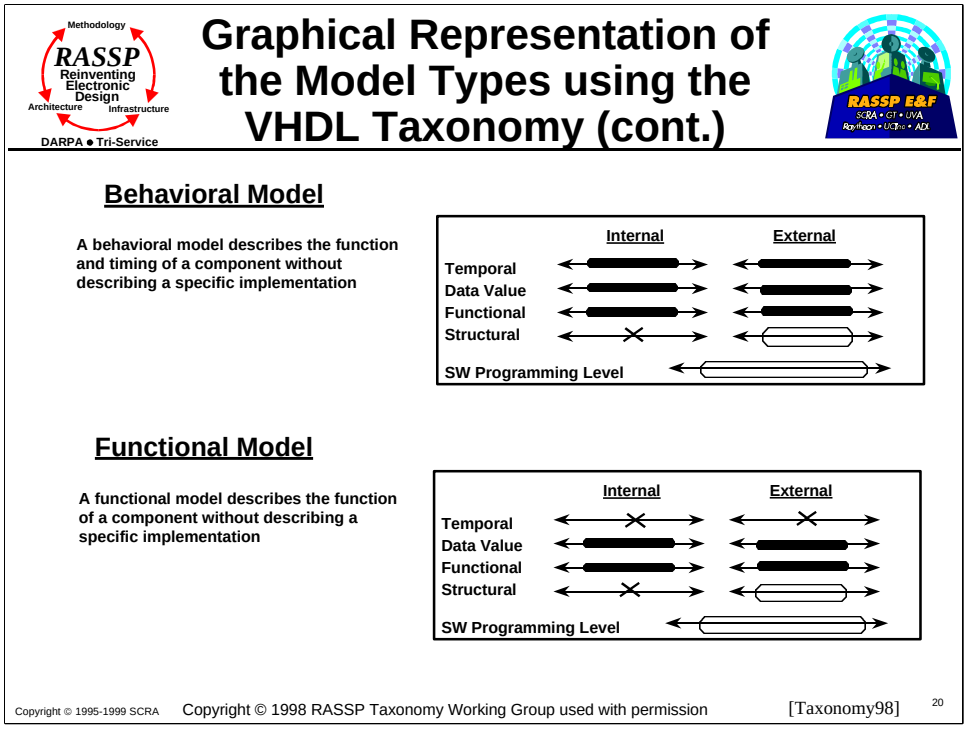
Mixed-level models are a combination of models of differing abstraction levels or descriptive paradigms.



Using the above key code and the previous representations, each model level is described on the taxonomy chart.

For example, the algorithm level shows that:

- It does not contain information on time internally or externally.
- Data is represented at the functional level.
- The function is captured internally, there is no external interface modeled.
- The structure is not defined.
- SW programmability is at the DSP primitive level (i.e., FFT, FIR, etc.) i.e. Matlab.

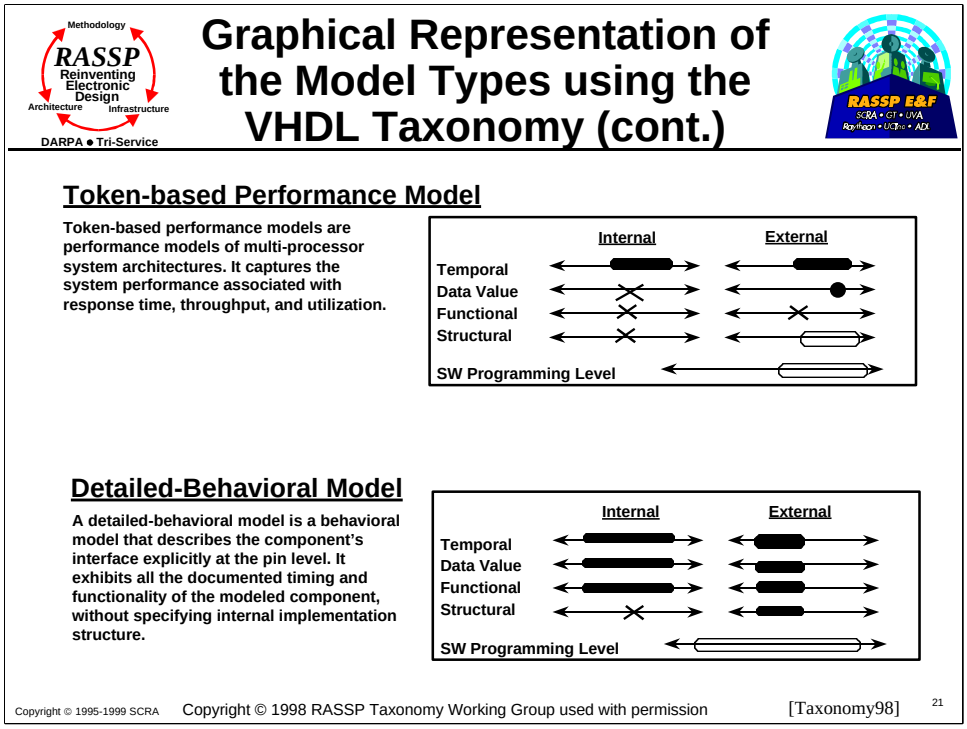


Behavior can be described as shown above. Note: Structure is not maintained internally, which we will see in the ISA model presented later. Timing, function, and data values can be modeled at any level and is case dependent.

A behavior model can exist at any level of abstraction. Abstraction depends on the implementation details.

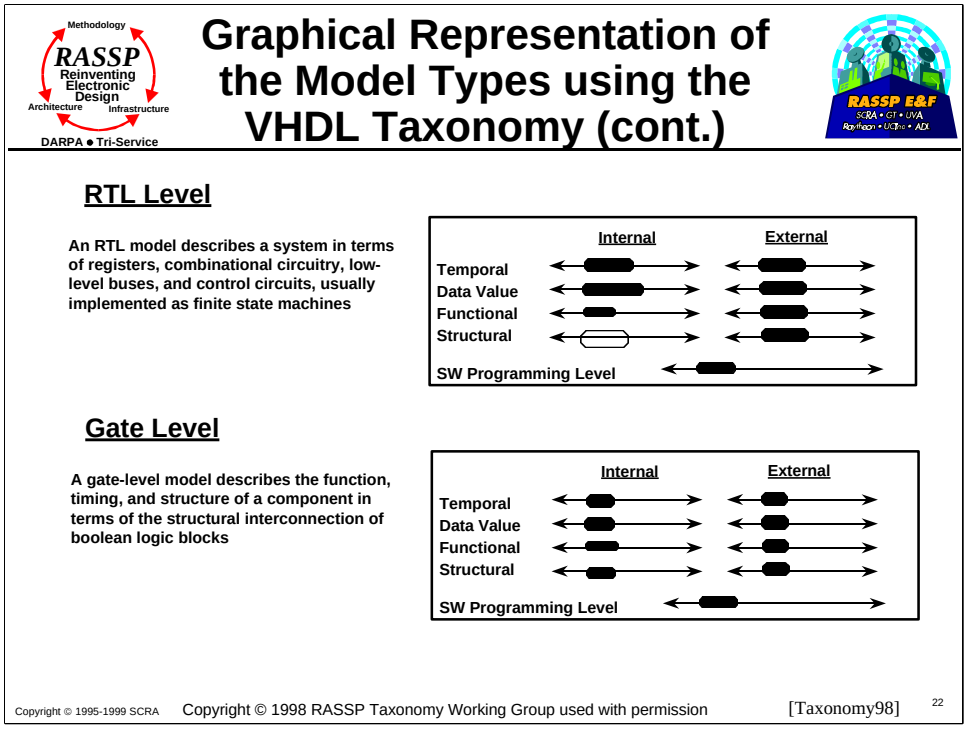
Behavioral : (Behavior = Function with Timing) (Synonym : Interpreted Model)

Functional models are not concerned with the temporal dimension and can exist at any level of abstraction.



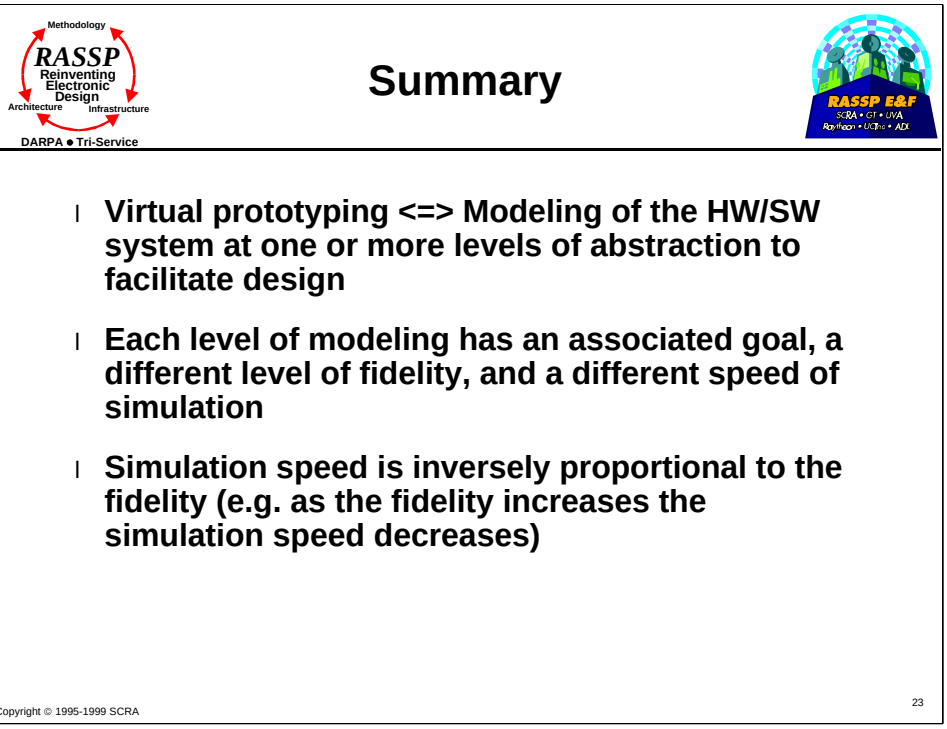
Performance models measure the time effects of the system such as throughput, latency, and utilization. A performance model can be written at any level of abstraction.The token-based performance model is a performance model of a multi-processor system's architecture.

Detailed-behavioral models are a type of behavioral level model. The structure at the external interface level is maintained. Internal structure is typically not specified in detail and SW programmability can optionally span any resolution but need not.This model type has traditionally been called a full-functional model, however the newer term is preferred.



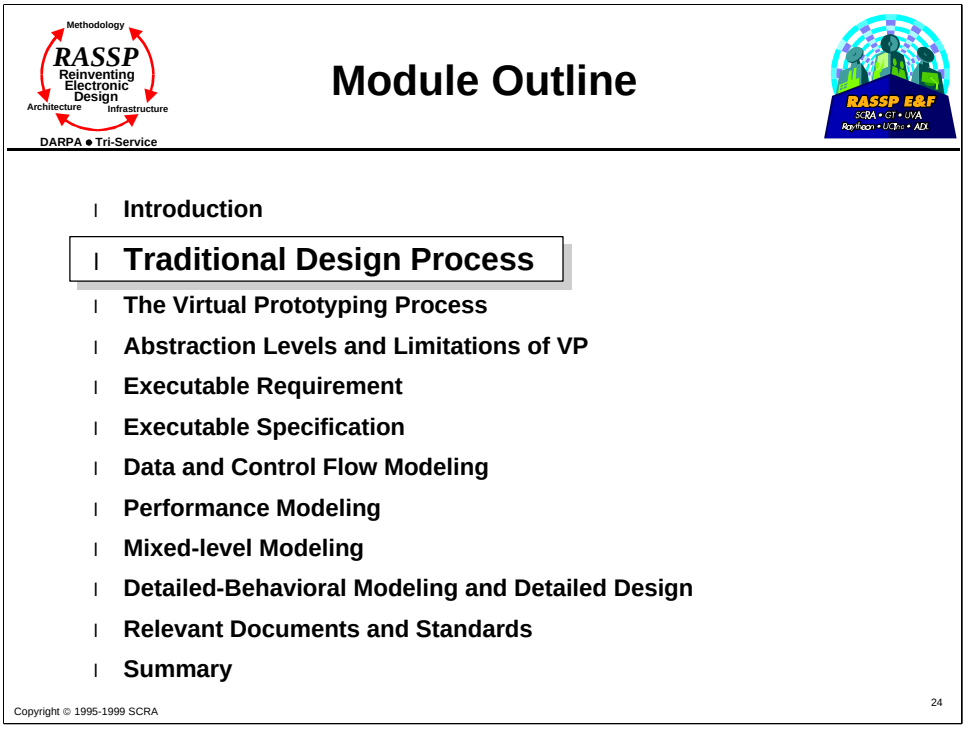
RTL and gate level models specify more of the details of a design and hence tend to be at the higher end of the resolution scales in all categories.

The gate level model includes structure at the higher resolution.

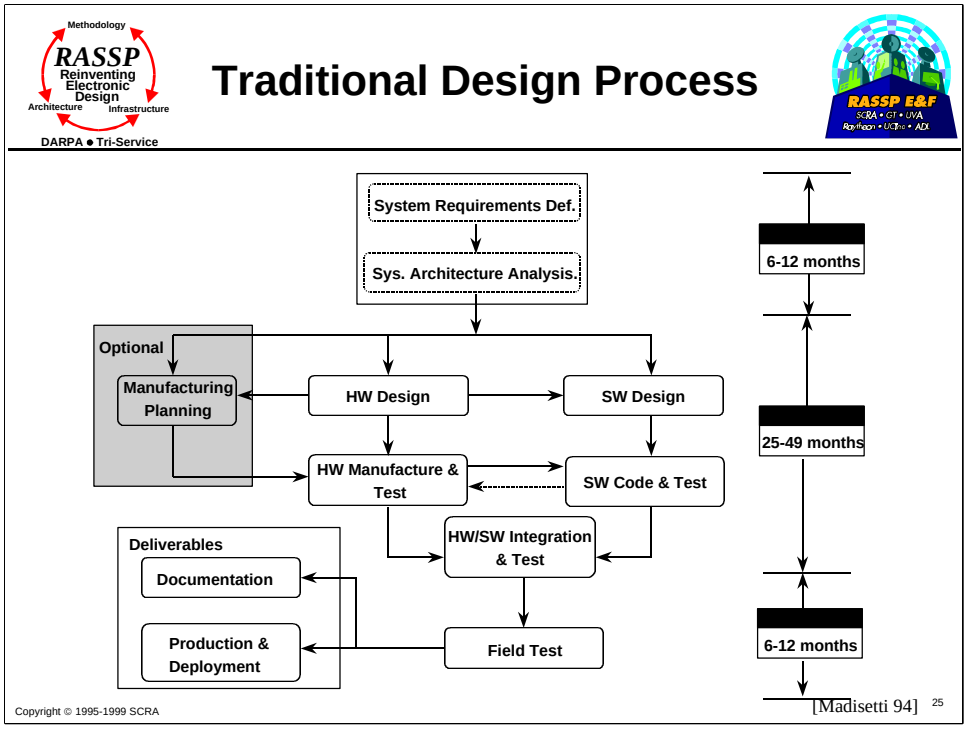


Some important points should be considered before proceeding with the specific topics. These include:

- VP implies the modeling of the HW/SW at one or more levels of abstraction to facilitate design, not all stages need to be modeled. It may be case dependent.
- Each level of model has its associated information it captures, a different level of fidelity, and a different simulation speed.
- Speed is inversely proportional to fidelity as a general guideline.



In this section, the traditional design process for large digital system designs is presented. This information was obtained through extensive interviews with defense contractors who design large digital systems that typically contain a large amount of both hardware and software. This section will give an overview of traditional design processes including the hardware design process, the software design process, the system architecture design process, and how integration of hardware, software, and hardware with software is done. It concludes with an illustration of the virtual prototyping design process and what elements in the traditional design process are modified to improve hardware, software, and architectural design.



A traditional design process model is required as a baseline to help assess the improvements afforded by the RASSP process.

The focus of the RASSP program is on signal processors consisting of a few to hundreds of processing elements.

This diagram shows the time frames related to current practice, broken down into various phases of development. These include:

- Architecture Analysis (6 to 12 months)
- HW and SW Design along with integration (25 to 49 months)
- Field prototyping and test (6 to 12 months).

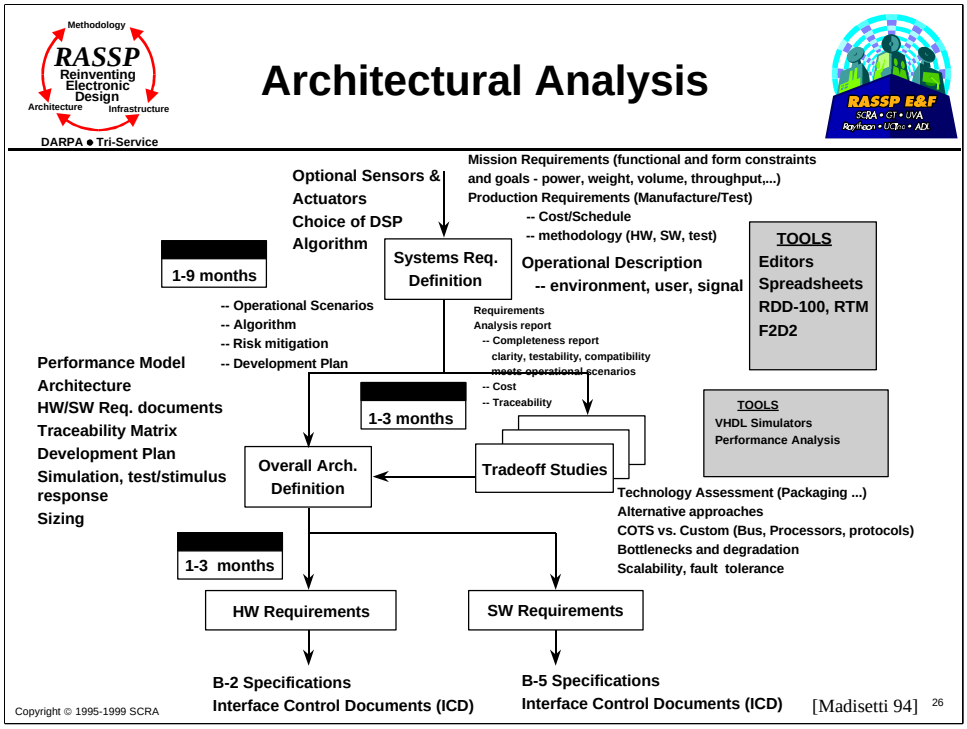
These will be decomposed further in the following slides.

This chart follows a waterfall approach to design methodology which is typical of traditional design processes.

The underlying concept of the waterfall process is a progression through various levels of abstraction, or phases, with the intent of fully characterizing each level before moving to the next.

The following bad design practices tend to result from this processes:

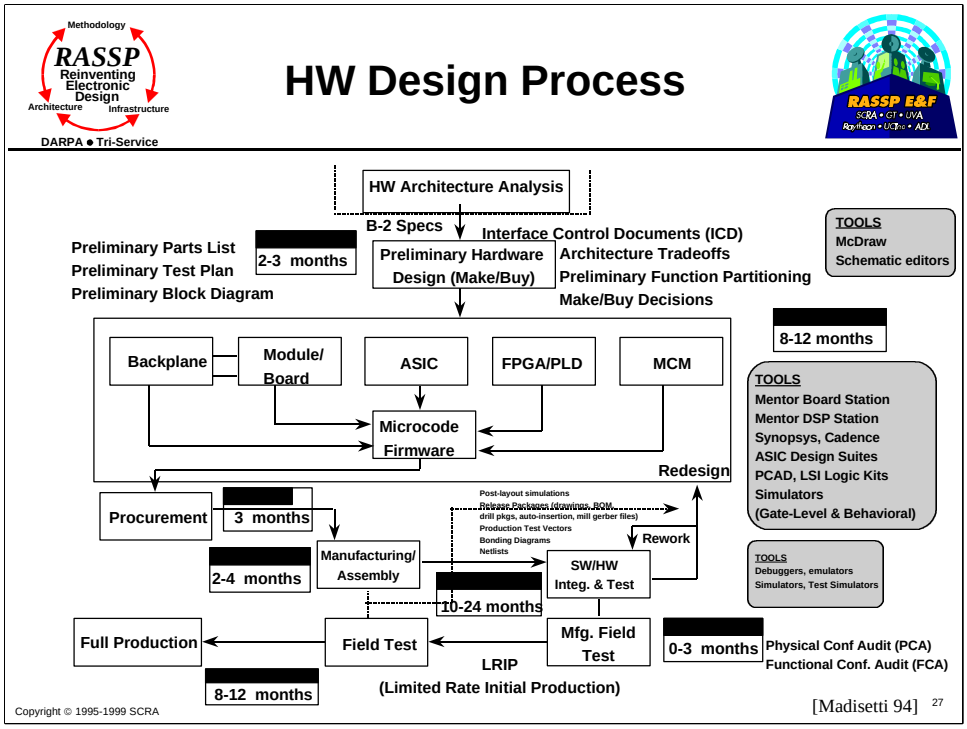
- Limited use of concurrent engineering
- Solving wrong problems early in design process
- Inflexibility late in design process
- Significant rework and cost resulting from design flaws found late in the process



The architectural analysis phase consists of, first, the study of the requirements to verify the feasibility of the work to be performed and to realize and estimate the cost of the design.

The next step is to define an architecture based on the requirements as well as to perform tradeoffs to determine the best architecture for the application.

Once an architecture is chosen, the application is then partitioned between its HW and SW requirements.

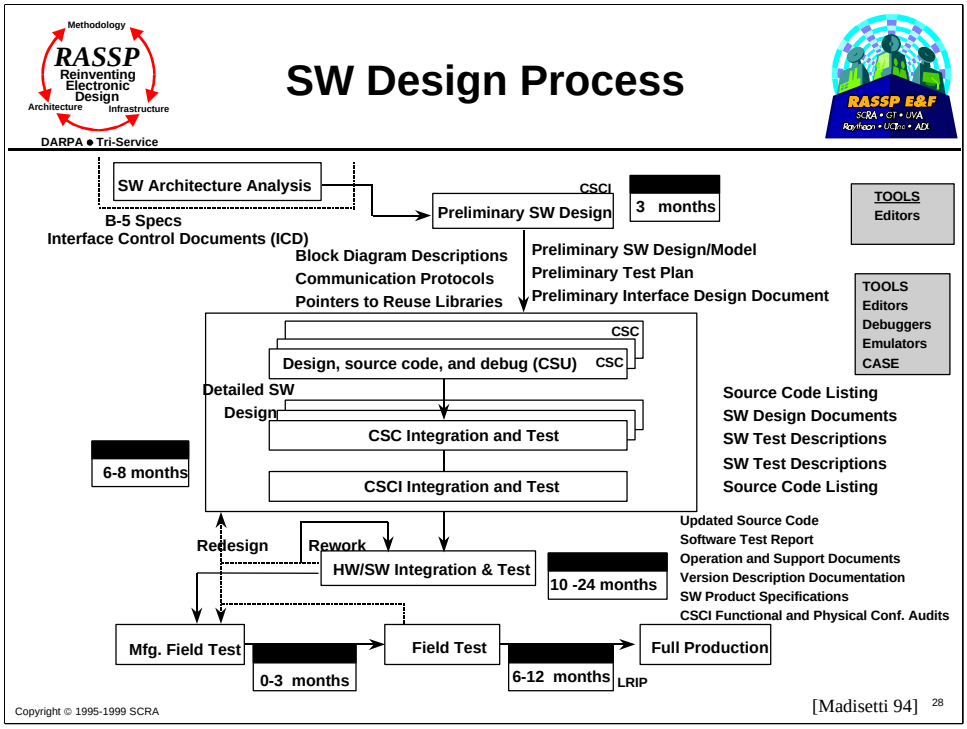


Upon completion of architectural analysis, both HW and SW design begin. The HW design starts with the decision to make or buy parts to solve the requirements handed down drom the previous stage. This can take between 2 and 3 months.

Based on the decision to make or buy, HW design at the chip and board level is initiated. This can include ASIC, FPGA, or COTS based design. This stage can last from 8 to 12 months.

Certain chips may have to be procured if off-the-shelf, and ASIC designs may require fabrication. Depending on availability, a time of 3 months is typical.

Upon HW design completion, integration with SW can begin. This is the first time the HW meets the SW and is a likely location for errors to occur. Errors at this stage can be costly and require long delays for design flaws to be fixed.



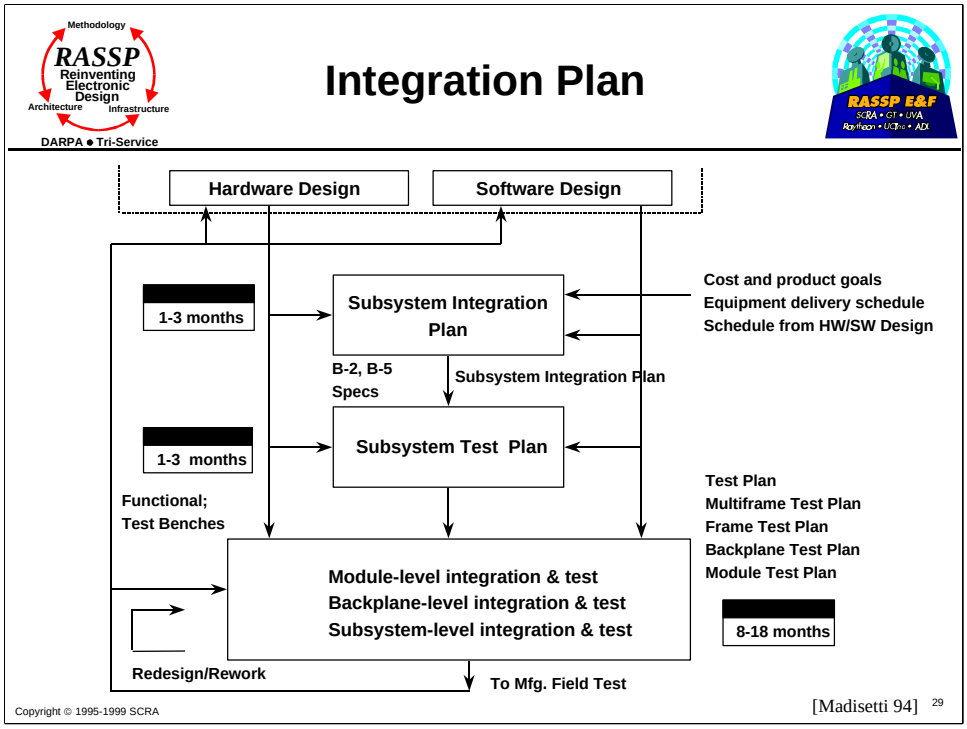
CSCI=>computer software configuration item.

Specified in the B5 Spec. and mentioned in the MIL-STD-490A. The B5 Spec identifies the specifications applicable to the development of computer software.

Preliminary SW design defines the types of code segments to be represented by the SW and typically requires 3 months to complete.

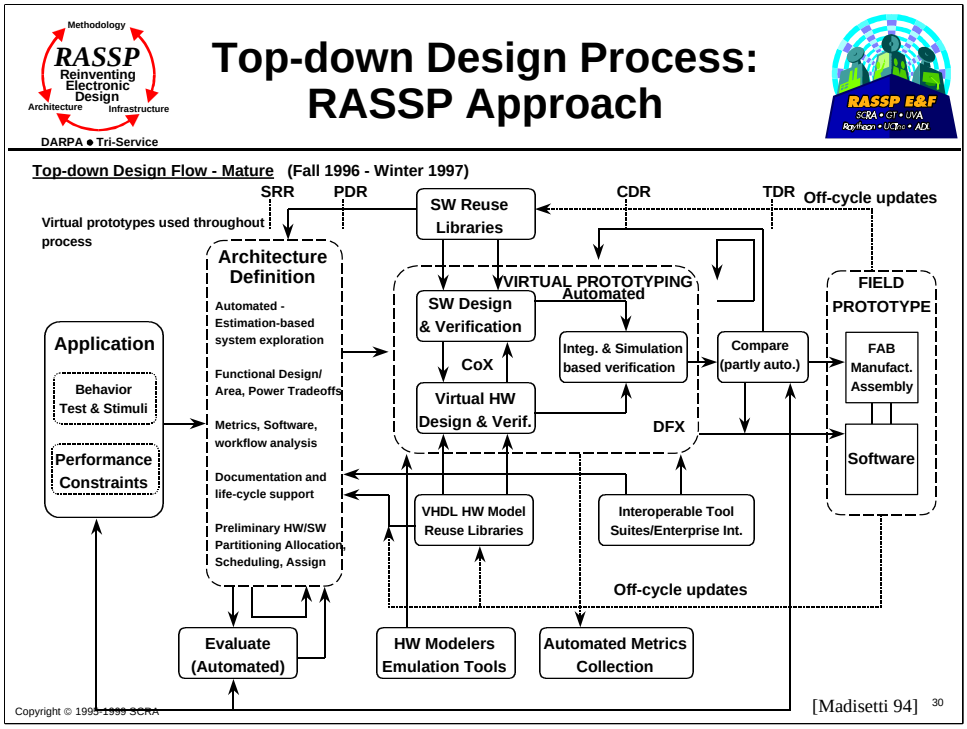
The detailed SW design requires the generation of computer software components (CSCs) and their integration and test.

The final test occurs when the HW is available. This is the point where misunderstandings between the HW and SW developments appear.



Integration occurs when the HW and most of the SW are ready. A plan for integration must be created to guarantee sufficient coverage of the HW and SW.

The actual integration and test can take from 8 to 18 months depending on the number of design flaws and SW work-arounds required.



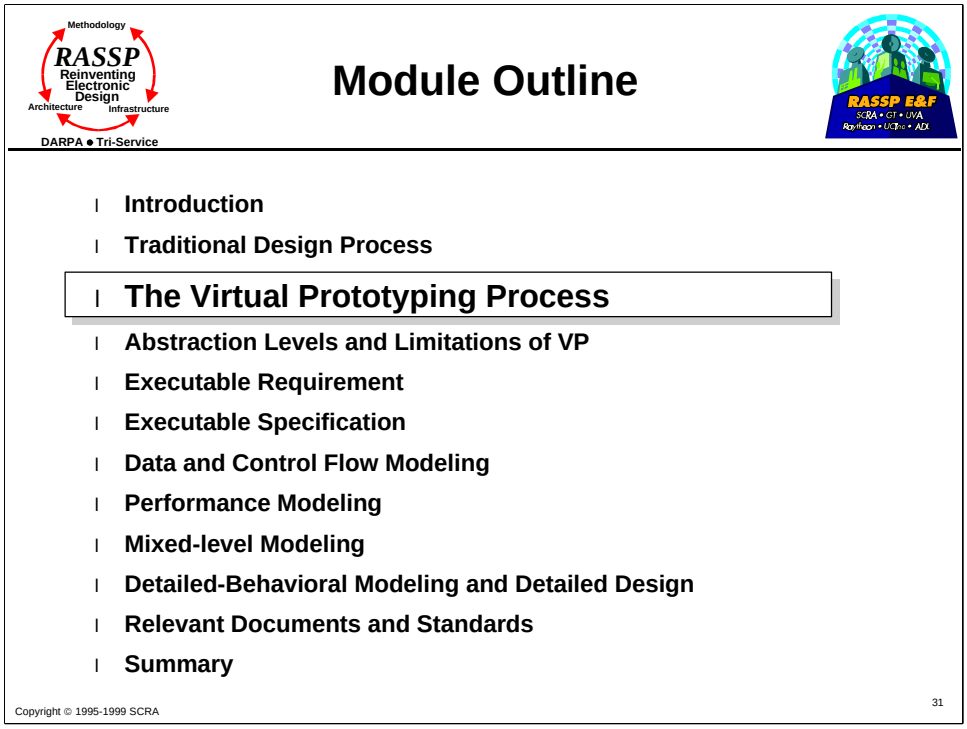
This diagram illustrates the mature RASSP top-down design process with no silicon in the in-cycle design loops; enterprise integration; interoperable tool suites; automated metrics collection; and an additional stage for rapid early algorithm, functional, architectural, and HW/SW partitioning in an automated manner called conceptual prototyping.

HW and SW verification is done earlier in the design process.

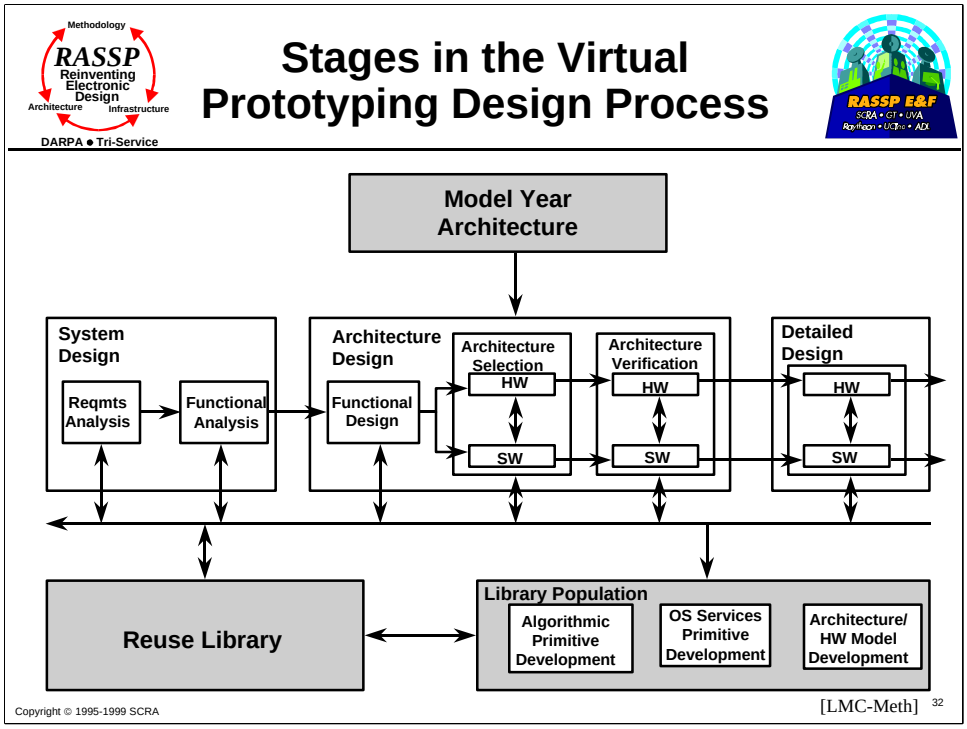
Conceptual Prototyping replaces the current manual HW/SW partitioning typical today.

The significant differences from the traditional design processes include:

- There is no HW fabrication, assembly, and test in the in-cycle design loop until after a large amount of hardware and software verification is done using virtual hardware
- Late binding of HW allows the design product to be state-of-the-shelf
- Extensive use of Conceptual and Virtual Prototyping guarantees first-time success
- Design reuse and the population of libraries are to support reuse
- Use of enterprise integration between design tools promotes design portability and standardization.
- This process uses extensive automation.



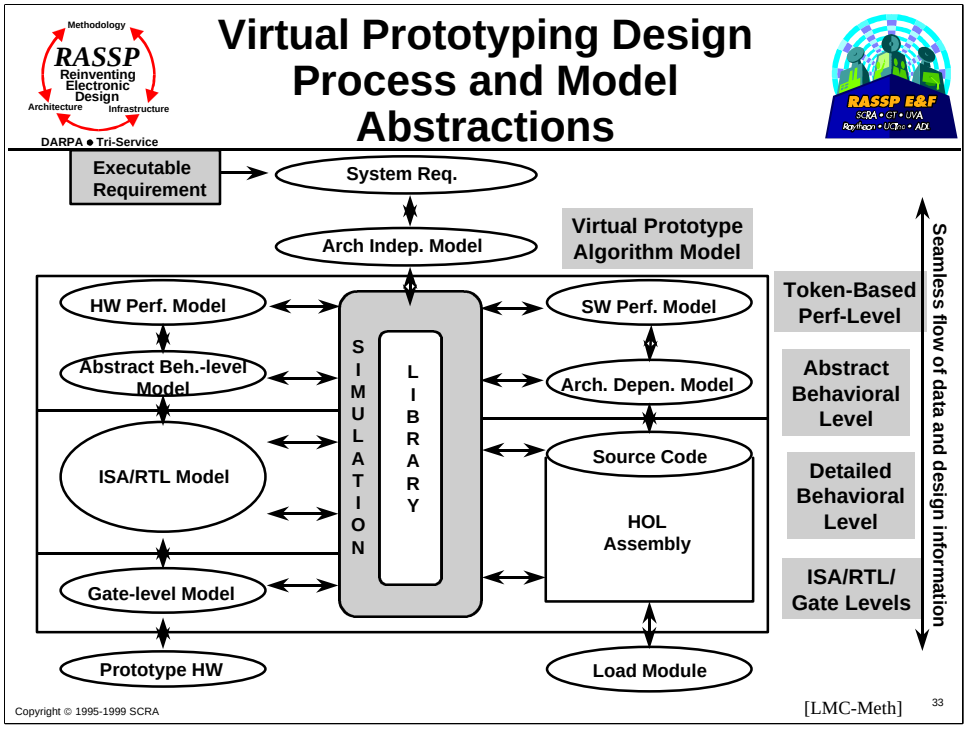
The next topic discussed is the virtual prototyping process and the steps used to define the top-down design methodology. In this section, top-down design is discussed. Virtual prototyping is shown to be a top-down design process where models are used to refine the implementation until actual hardware and software are integrated into a final system.



To dramatically improve the process by which complex digital systems are specified, designed, documented, manufactured, and supported requires a signal processing design methodology that recognizes a number of application domains. A key element to implement this methodology is a Model Year Architecture approach that adheres to a specific set of principles which include:

The architectures must be open to promote HW/SW upgradability and reusability in other applications:

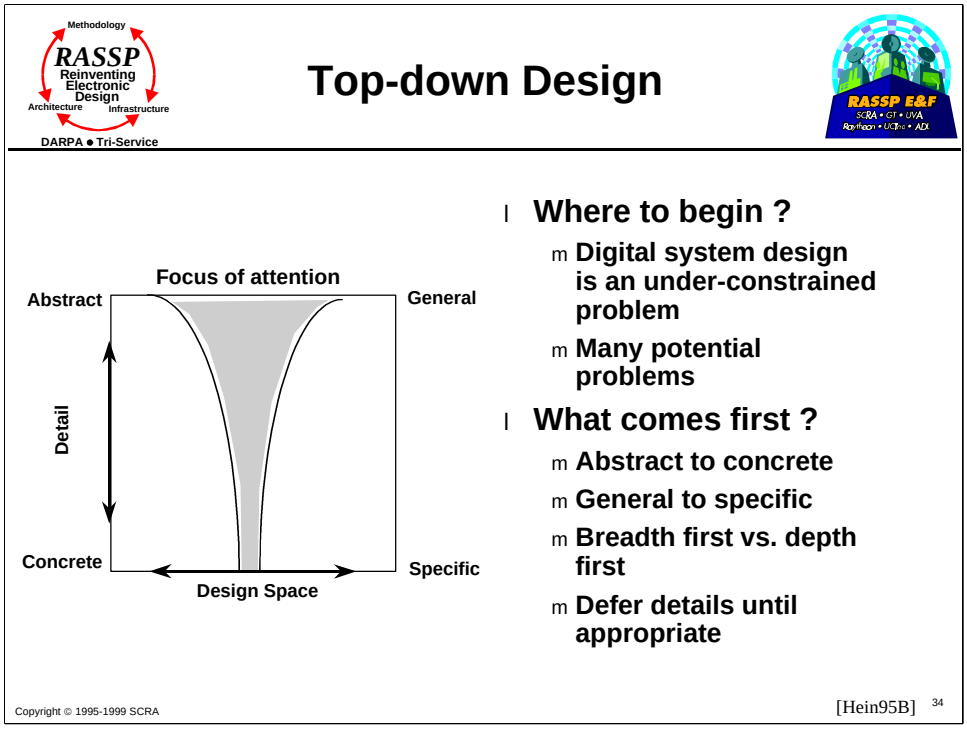
- The architectures must use emerging, state-of-the-art commercial technology whenever possible.
- The architectures must support a wide range of applications to maintain low non-recurring engineering (NRE) and recurring engineering (for design changes later in the process) costs.
- The architectures must facilitate continuous product improvement and substantial life-cycle-cost (LCC) savings in fielded system upgrades.
- The Model Year Architecture(s) (MYA) must be supported by the necessary library models to facilitate trade-offs and optimizations for specific applications. Reusable HW and SW libraries facilitate growth and enhancement in direct support of the model year concept. The notion of model year upgrades is embodied in the reuse libraries and the methodology for their use. As technology advances, new architectural elements may be included in the library. Rapid insertion of a new element into an existing, RASSP-generated design is the goal of the Model Year concept.



The virtual prototyping (VP) design process is based on true HW/SW codesign and is no longer partitioned by discipline (e.g. HW and SW), but rather by levels of abstraction represented in the system, architecture, and detailed design processes. The above figure shows VP as a library-based process that transitions from architecture independence in the systems design process to architecture dependence in the architecture process.

Various levels of virtual prototypes are generated throughout the design process. The first is output from the systems process, where an executable specification is generated, the architecture process generates two more with increasing detail and verification. The final prototype is created before HW/SW sign-off and full system verification is done at the RTL and gate levels with application and test SW running on the prototype. The design flow is seamless with models being interoperable at each stage and data flowing down from the top can be used for verification at each abstraction level.

The design flow is captured by workflows by a design manager.



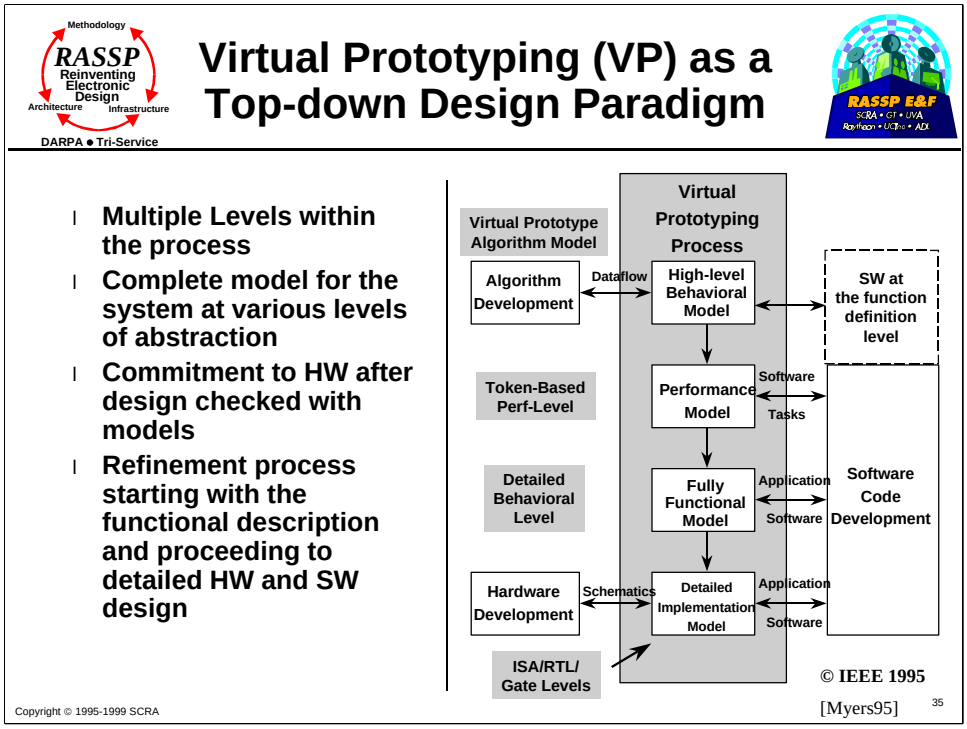
Top-down design is the process of moving from the abstract to the more concrete, looking at the general initially, while moving toward the more specific as the design progresses, and focuses on attention on a large design space early in the process and moves to a single implementation at the end of the design. The goal is to have a final implementation that meets the original requirements.

Digital system design is an under-constrained problem so the design space is large.

VP is a top-down design process so the objective is to explore as much of the design space in the beginning where the details of the design are abstract. Breadth first vs. depth first.

As the design evolves, the focus is placed on one or a small number of design candidates until the final design is determined.

The process should also reduce the risk of poor designs, those that do not meet the customers final requirements, so a breadth first focus initially should analyze multiple design alternatives, settling on one that optimizes or nearly optimizes the design constraints.



The virtual prototyping process can be looked at using the top-down design paradigm. The modeling begins with a functional description and proceeds through refinements to produce HW and SW. The refinement process produces models for functional description, system performance, system detailed behavior, and detailed system design.

VHDL can be used in all phases of the process and is a single-language solution capable of capturing the entire flow.

Functional Modeling: Produces a dataflow model as a set of interconnected sub-functions.

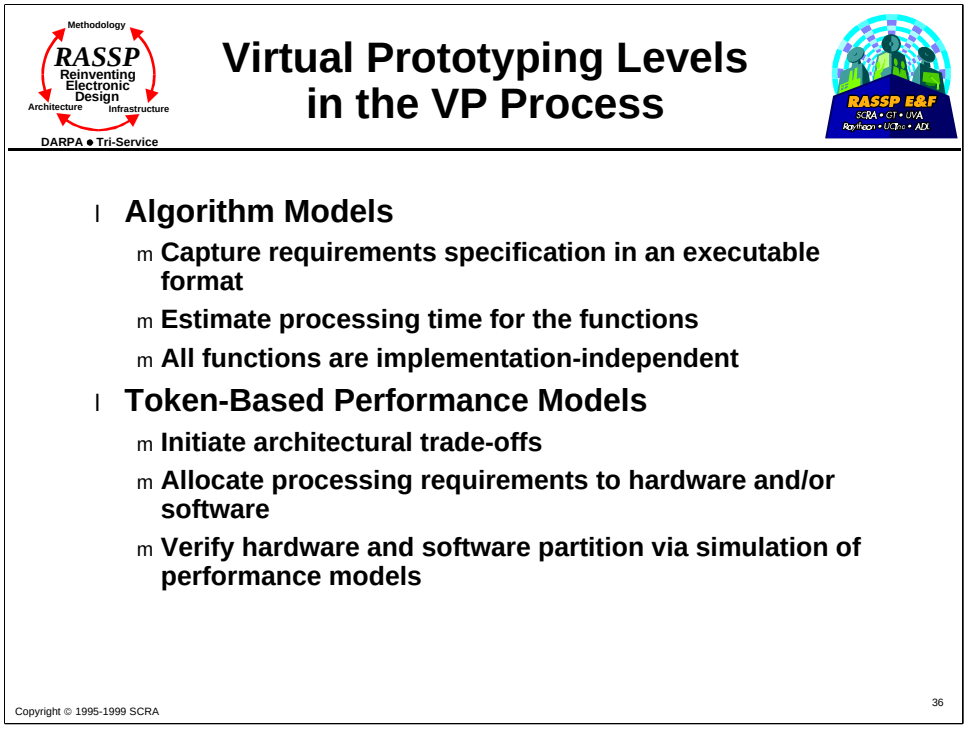
Performance Modeling: Examines candidate architectures for trade-offs in both HW/SW partitioning by measuring time-related system parameters such as latency, throughput, and utilization.

Detailed Behavioral Modeling: Provides a model that is both functionally correct and exhibits the functional and performance characteristics of the devices being modeled. (Used for COTS components in the system; i.e., Processor models, bus models, memory devices, etc.)

Detailed Implementation Models: Provide sufficient details of the device to determine the exact implementation of the components. (Used for in-house designs but not COTS components.)

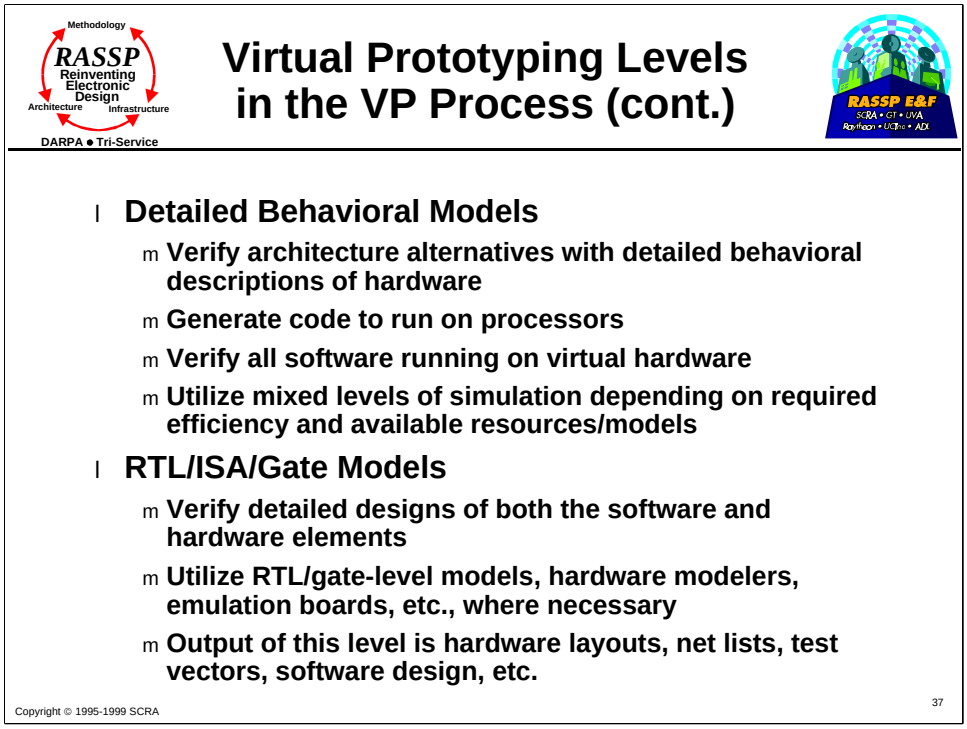
Test data using the same interface format can be passed down the design process for verification.

Software can also follow the top-down design process, where in the early stages, functional breakdowns are designed and at later stages, actual code is developed.



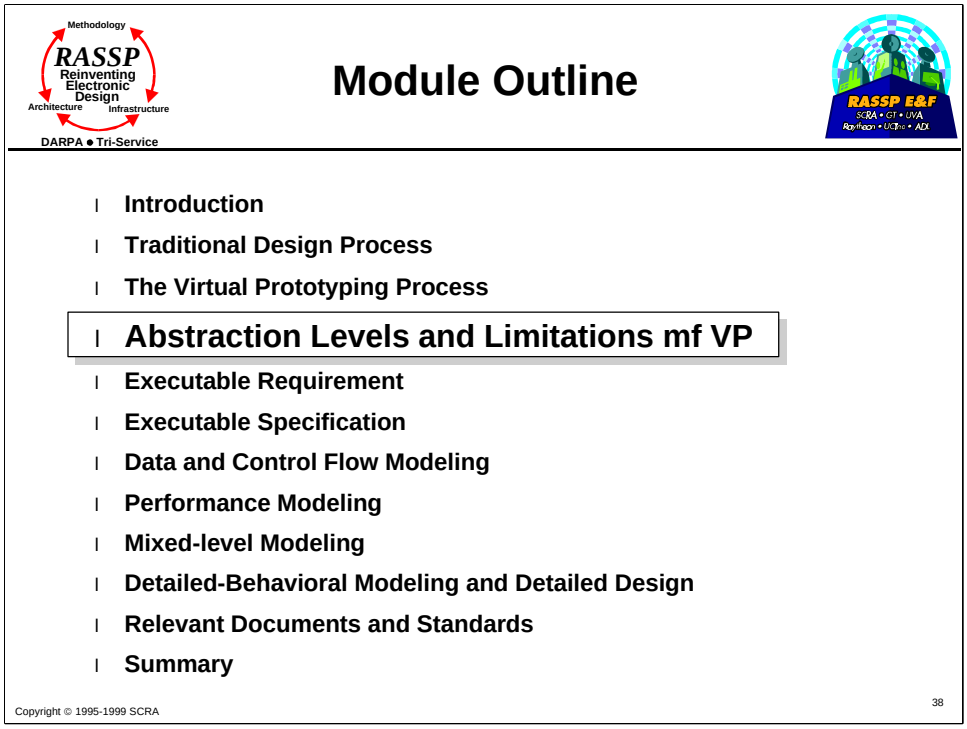
The systems process captures customer requirements and converts them into processing requirements (functional and performance). The requirements are captured in the appropriate tool and are translated to a set of simulatable functions referred to as the executable specification. This constitutes the algorithm type models developed in the virtual prototyping process.

Token-based performance models are used in the architecture selection process. At this stage, trade-offs are initiated between candidate architectural alternatives, e.g. HW/SW partitioning tradeoffs. The system-level processing requirements are allocated to hardware and/or software functions. Simulation is performed on these candidate architectures using VHDL performance models.



Abstract and detailed behavioral modeling in the virtual prototyping process involves the verification of a select number of candidate architectures chosen after completion of the token-based performance modeling stage in the process. At this stage, mixed-level models at the performance and behavioral levels are used to verify the performance of the architectural candidates. Code is generated to run on the processing elements and the final prototype will verify the code. This is the first stage where software touches the hardware.

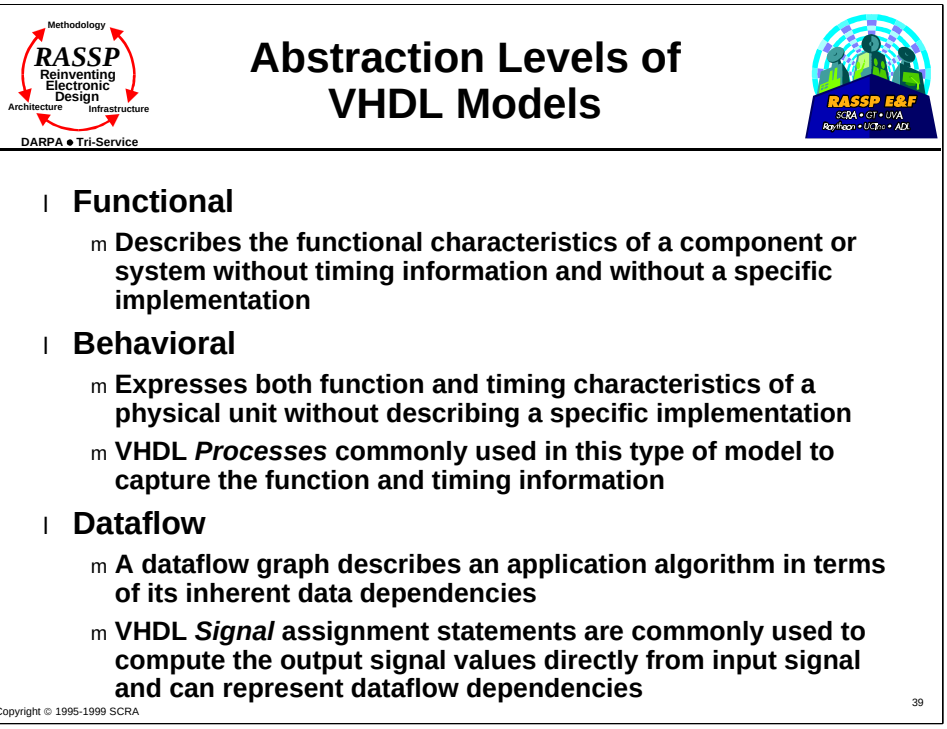
ISA/RTL/Gate model prototypes are generated when a final design is chosen from the previous stage. This design is verified at the detailed level using technologies such as hardware modelers, ISA/RTL/Gate-level models, hardware emulators, etc.



When we discuss virtual prototyping, there are various levels of abstraction used to describe the model we use to represent the system. This section presents some of the possible abstraction levels commonly found in virtual prototyping.

Virtual prototyping using VHDL has some limitations that will also be presented.

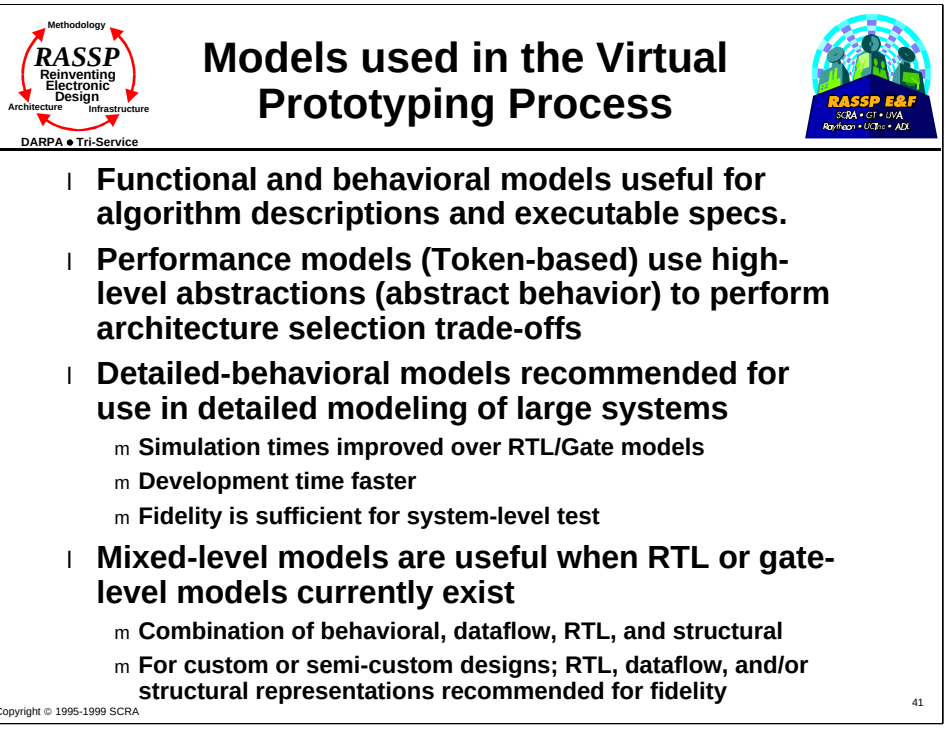
In this short section, we introduce some common abstraction levels used when describing models of components or systems using VHDL.



The virtual prototyping process incorporates models with varying levels of abstraction. The generally accepted levels are listed above and on the next slide. The RASSP working group's taxonomy [taxonomy98] maintains an up-to-date listing of the various terms.

Abstraction Levels of VHDL Models (cont.)

- **RTL (Register Transfer Level)**
  - **Describes the design in terms of registers, combinational circuitry, low level buses, and control circuits**
- **Structural**
  - **Representation of a system in terms of interconnections of a set of components**
- **Refer to [Taxonomy98] for definitions of VHDL related items**

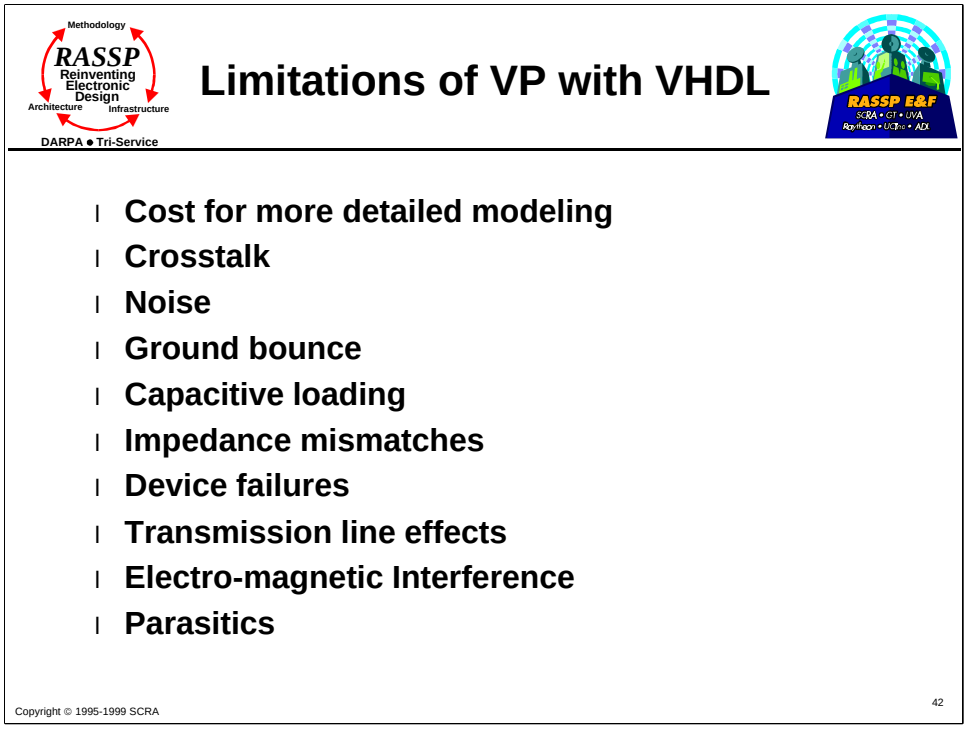


Detailed-behavioral and interface models are the most common type used in system-level virtual prototyping when COTS parts are being modeled.

RTL and dataflow models are the preferred level of model when custom designs are used in a virtual prototype. At this level most synthesis tools can generate the logic required to perform the function.

Structural models are used to connect all the components of the virtual prototype.

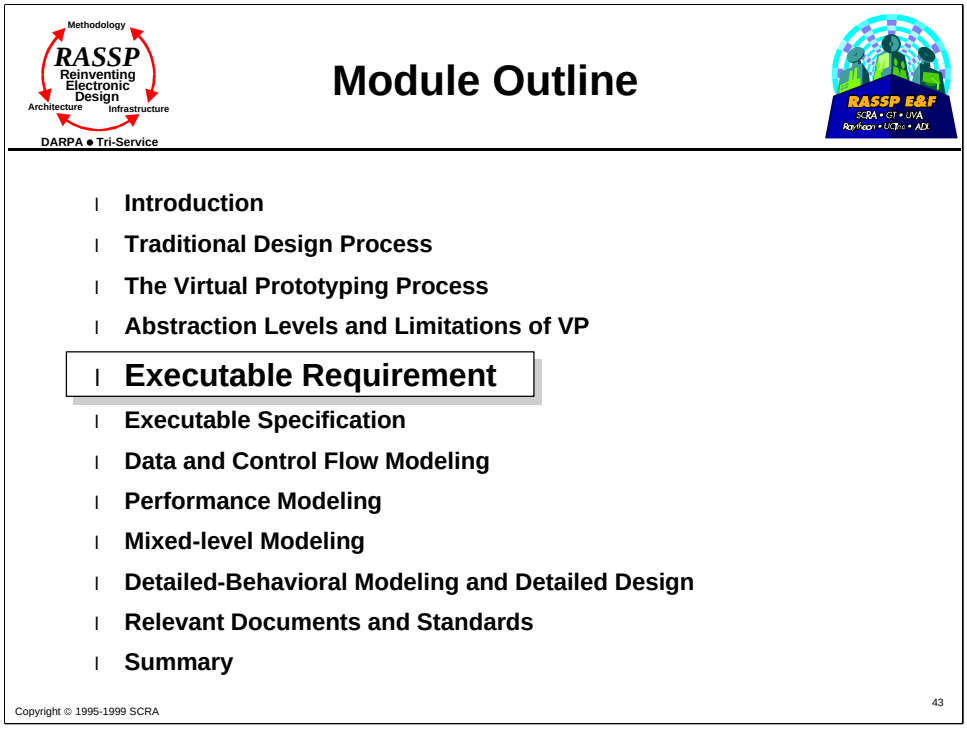
The case study at the end of this module uses the following types of models: Behavioral models for COTS parts, RTL/Dataflow-level models for custom parts, and the structural model to tie the system together.



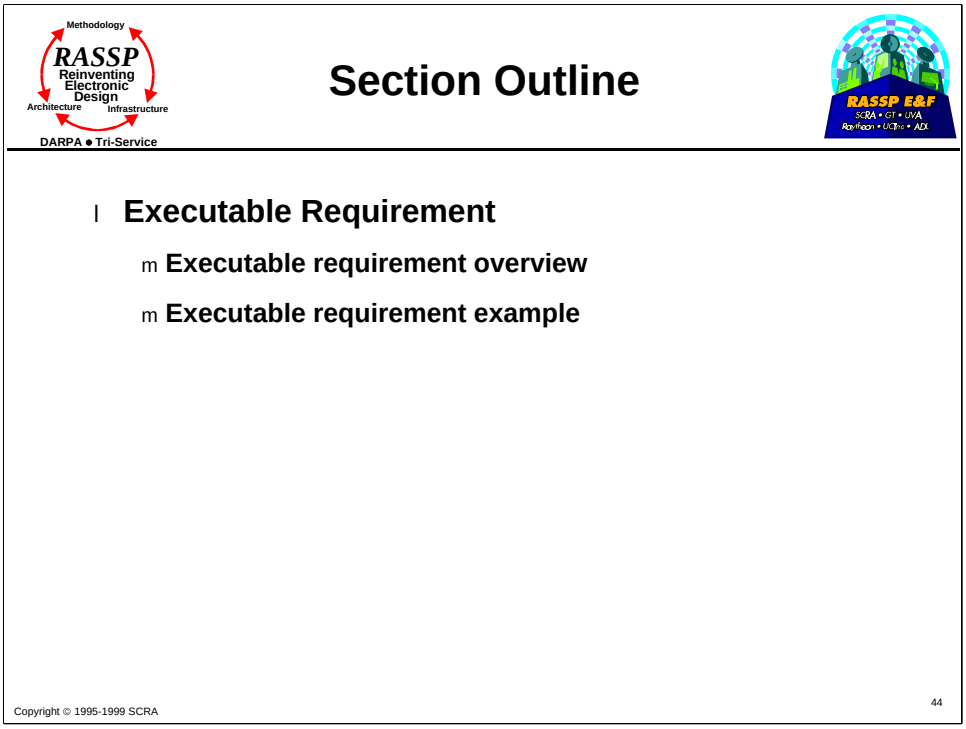
[Lockheed95]

Not all system behavior can be modeled with VHDL or simulation time becomes prohibitively expensive. Some of these system behaviors are listed above.

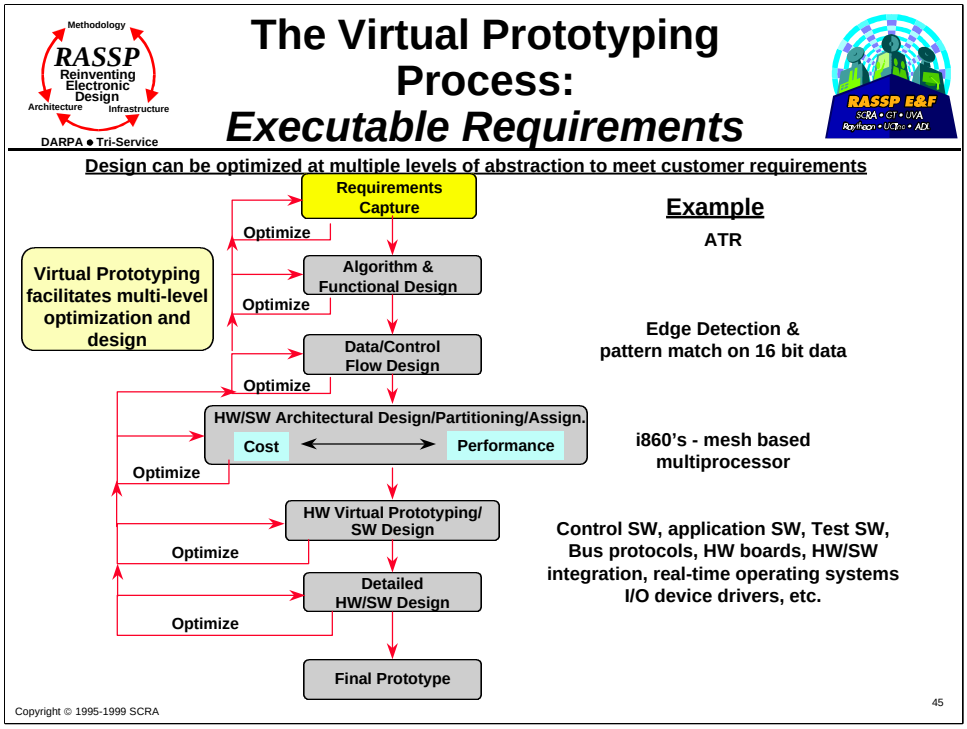
Lower-level modeling tools such as SPICE can be used to model these effects. However, at the system level, enough time is consumed in simulation by the high-level models without the additional expense of simulating these device characteristics. Good design practices can reduce the impact of most of these effects.



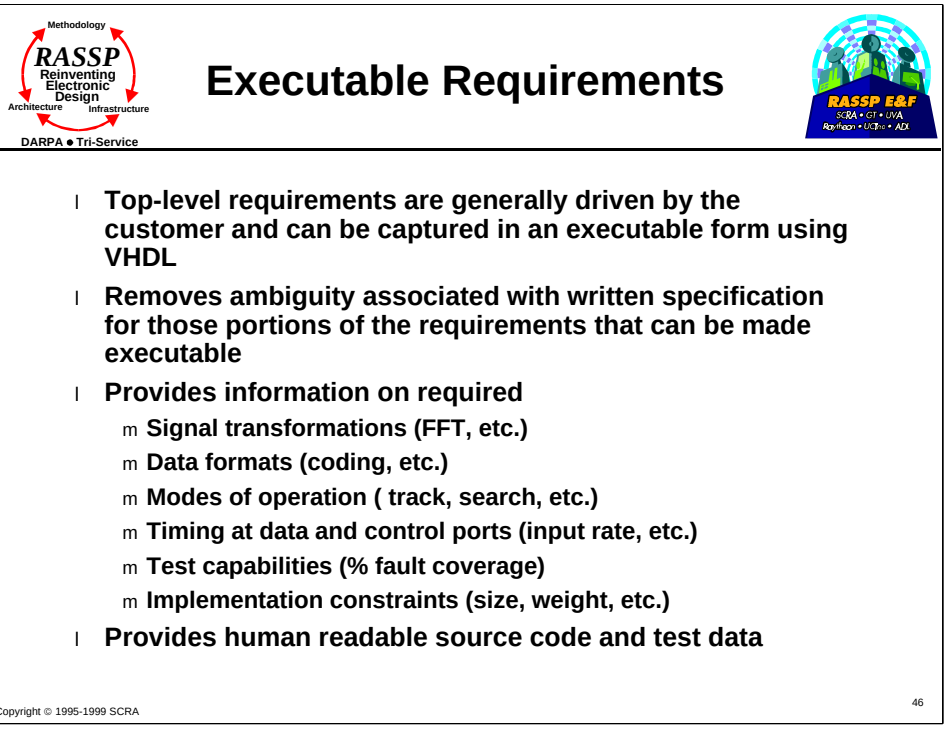
This section will cover the area of executable requirements. Executable requirements represent the traditional paper requirements in a form that can be simulated on existing computer-aided design tools. In this section, the role of the executable requirement is described and it is shown how the requirements of a design can be captured for a simple example using VHDL as the modeling language.



First, an overview of executable requirements is presented followed by an example that uses VHDL as the modeling language.

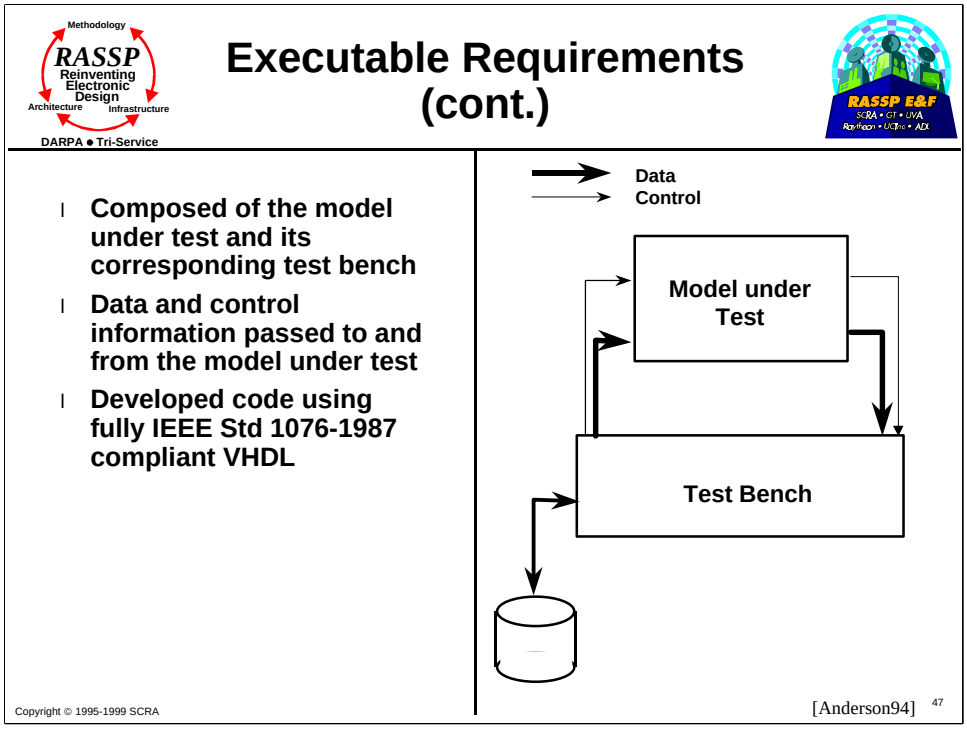


The VP process is a top-down design paradigm with optimization done at multiple levels of abstraction. There are a number of slides in this module that capture the essential flow of this design process but in a different format. In general, the first stage in any design process is the correct definition of the requirements for the design. Once the customer's requirements have been understood, then the system design team begins working on design specifications that attempt to satisfy all the customers requirements. In the above process flow, this represents the algorithm definition and functional design stage. At this stage, the algorithms used to implements the functions are refined. The size, weight, area, and power constraints may also be refined as well. At this point, initial computational complexity of the algorithms as well as fixed-point characteristics are analyzed. As the process proceeds from the top level, further refinement is done. At the data flow level, the algorithms are refined and data flow graphs that implement the functionality are explored. These data flow graphs are used in the HW/SW architectural design phase, where the nodes in the flow graph are either mapped to hardware or software. At this level in the process, the hardware and/or software are sized to meet the requirements. Timing critical information is captured such as latency, throughput, and resource utilization. As more detailed design is done, the architectural elements are either development as software modules or hardware components (ASICs, Processors, etc.) At the end of the process, final hardware and software are realized. At each stage in the process, the design is verified and optimized to meet the customer's requirements.



The executable requirements are in an executable format to help remove ambiguity in the associated written specification. It provides information on the types of signal transformations to perform, the data formats to be used at important interfaces, modes of operation of the system, timing of data and control at the interface ports, the types of test capabilities to be employed by the system, and implementation constraints contained in the specification.

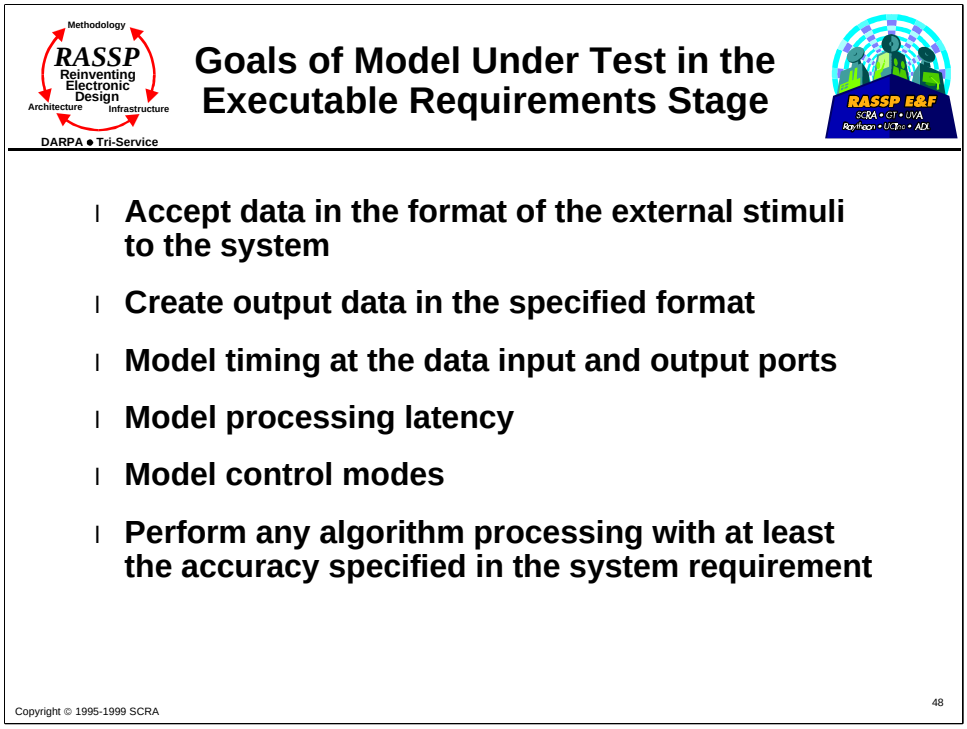
The source code is provided to clarify any misinterpretations in the requirements. Test data is provided to help verify performance of the system as it evolves through the design process.



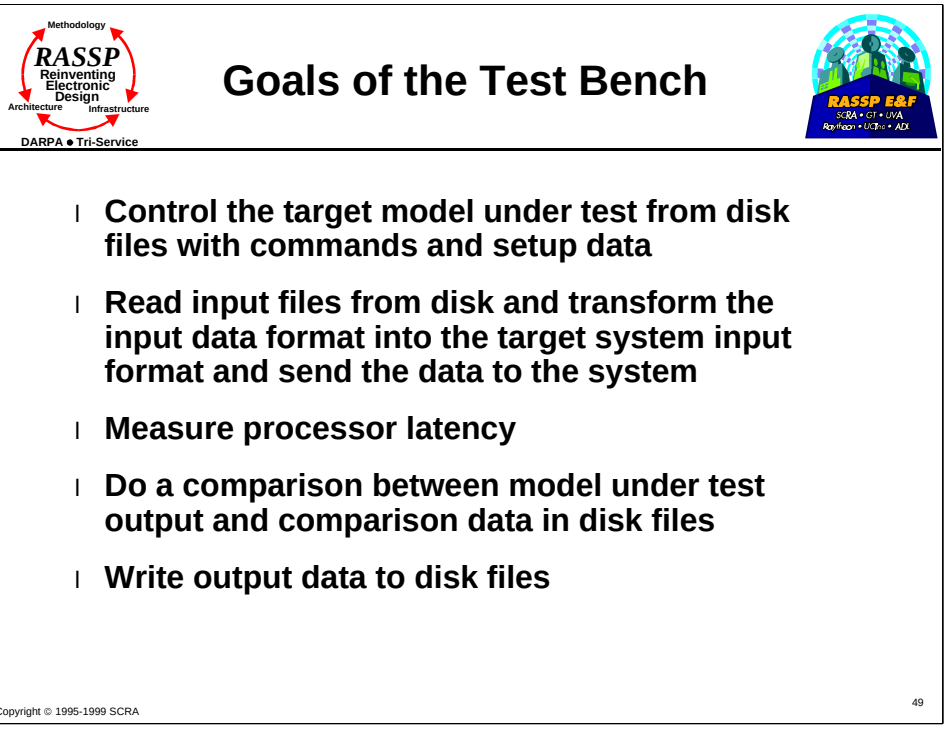
In general, all executable requirements contain 2 main entities, the test bench and the model under test.

The focus of the executable requirements is on the test bench which will be used throughout the refinement process of design. The test bench captures the requirements of the environment for which the component or system will function. It applies the stimuli (control and data) that the model under test shall respond to correctly and captures the responses of the system to compare its behavior with expected responses.

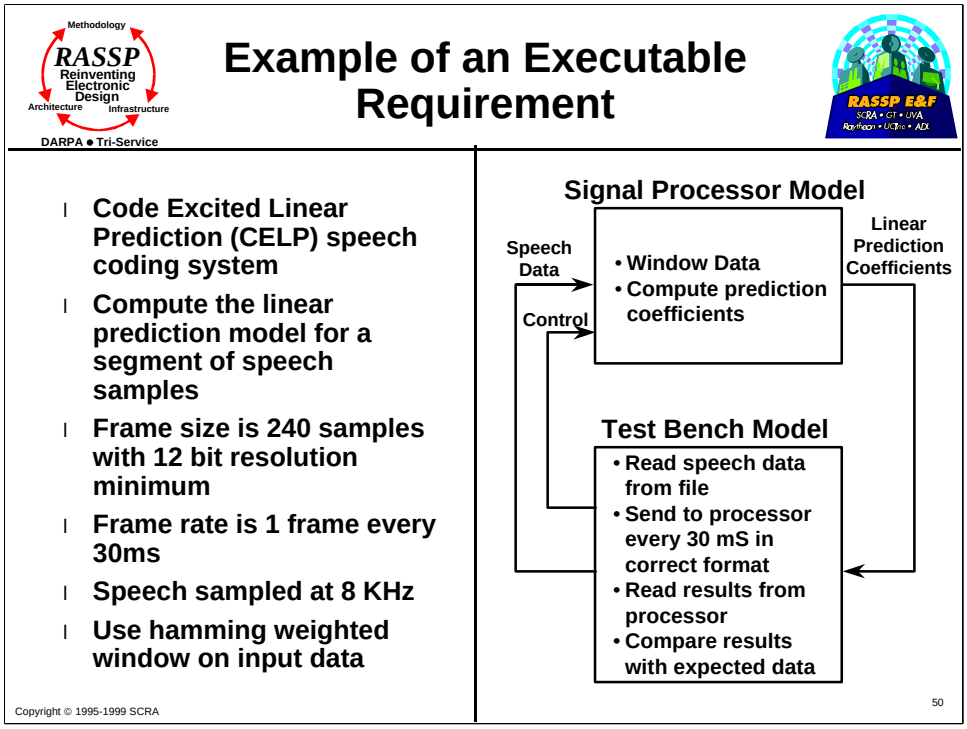
Goals of both are presented next.



This slide presents the goals of the processor model simulation. It accepts data in the format of the external stimuli to the system, creates output data in the required format of the output elements that will be connected to the device or system, models input and output timing as required by the specification, simulates the amount of processing latency expected by the system, models all control modes of operation, and performs any processing algorithms with at least the accuracy specified in the system requirement.



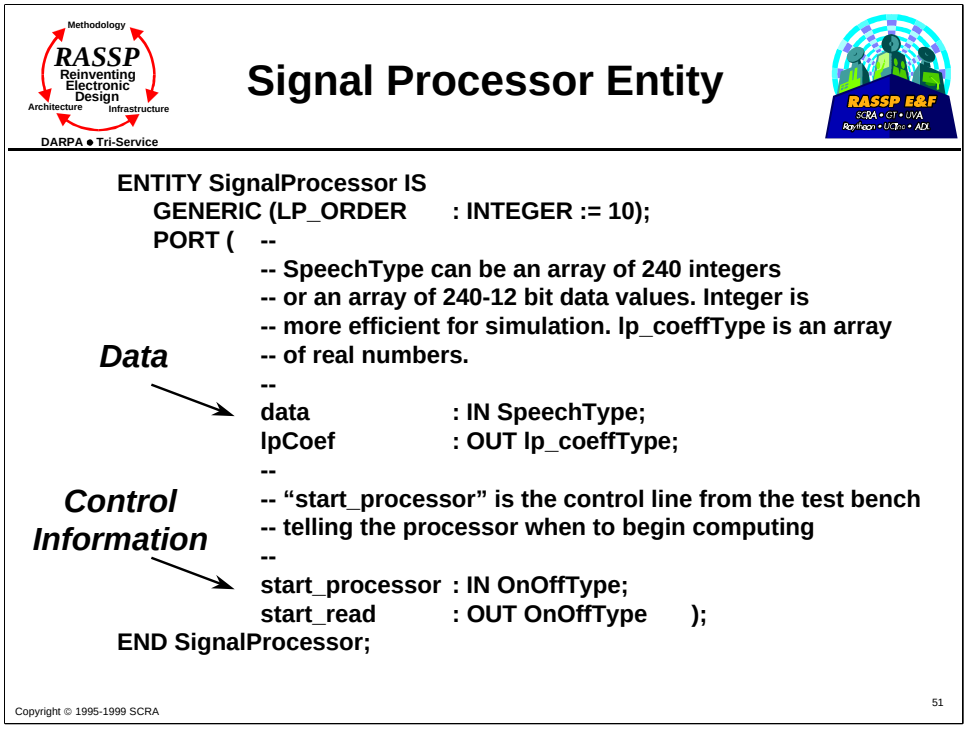
The test bench controls the model under test using commands and setup data read from disk files. Input data is modeled using disk input files and the data read from the files is transformed into that required by the model under test. The test bench also monitors responses of the system under test. The processor latency and algorithmic transformations are computed and written to disk files and compared with expected results based on the specification.



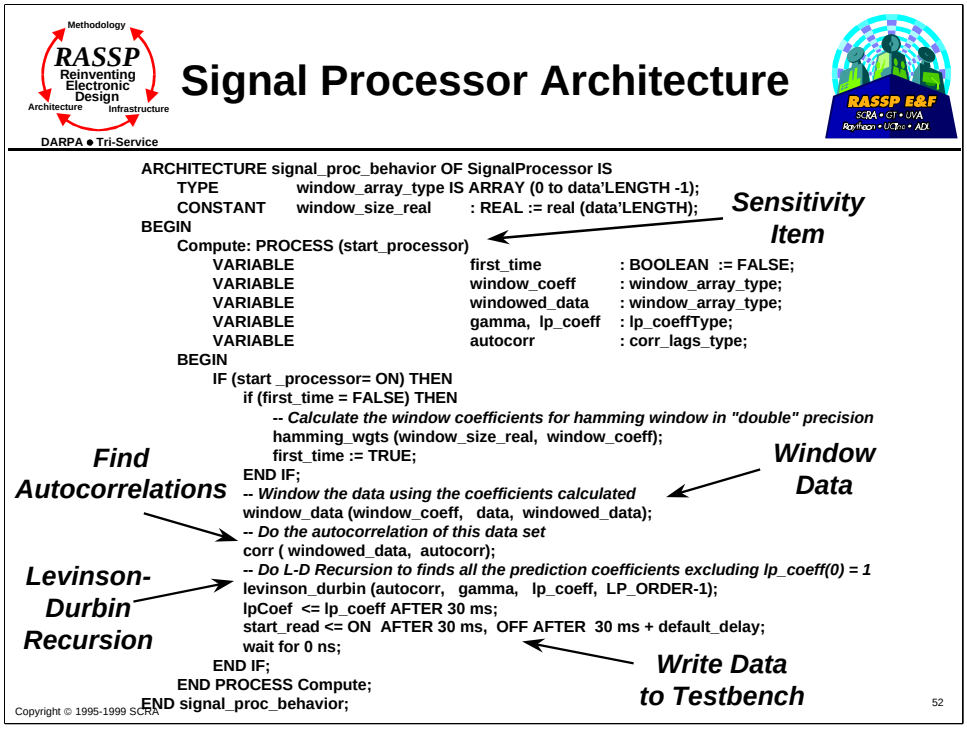
This example will be used to illustrate how an executable requirement may be specified in executable form. It will also be used in the following two sections on executable specification and data flow modeling.

This is a simple example for illustrative purposes only. Speech is sampled at 8 KHz and input to a signal processor in frame sizes of 240 samples representing 30 ms of speech data. The processor windows the speech data using a hamming window function and calculates the linear prediction model for each of the speech segments. The equations for the hamming window and the linear prediction algorithm are presented in a later slide contained within the executable specifications section, where algorithm analyses are performed.

The test bench inputs data to the signal processor and monitors its outputs. The test bench reads its speech data from a file and outputs the linear prediction coefficients to a file after the processor has completed its computations.



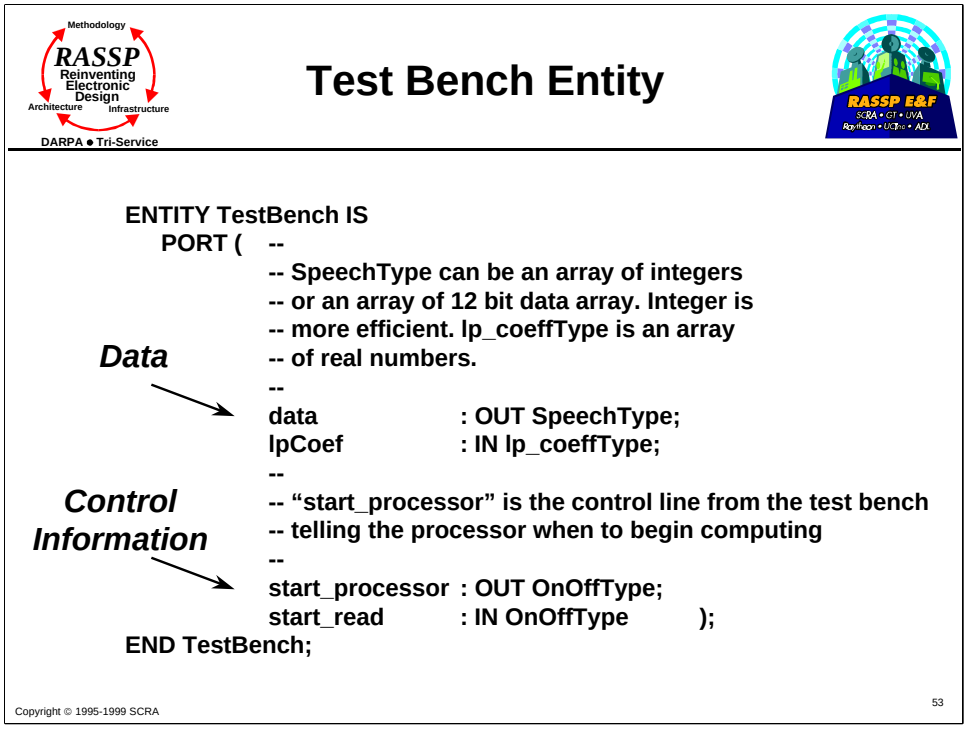
This is the entity description of the signal processor. Since the processor is a linear prediction computational engine, the linear prediction (LP) order is passed as a generic. The ports used for input and output to the processor include its data and control lines. The data lines consist of 240 samples of speech data in the format "SpeechType" and the output data are the LP coefficients. The control input information indicates when the processor starts computing the coefficients and the output control line indicates to the test bench when to read the results.



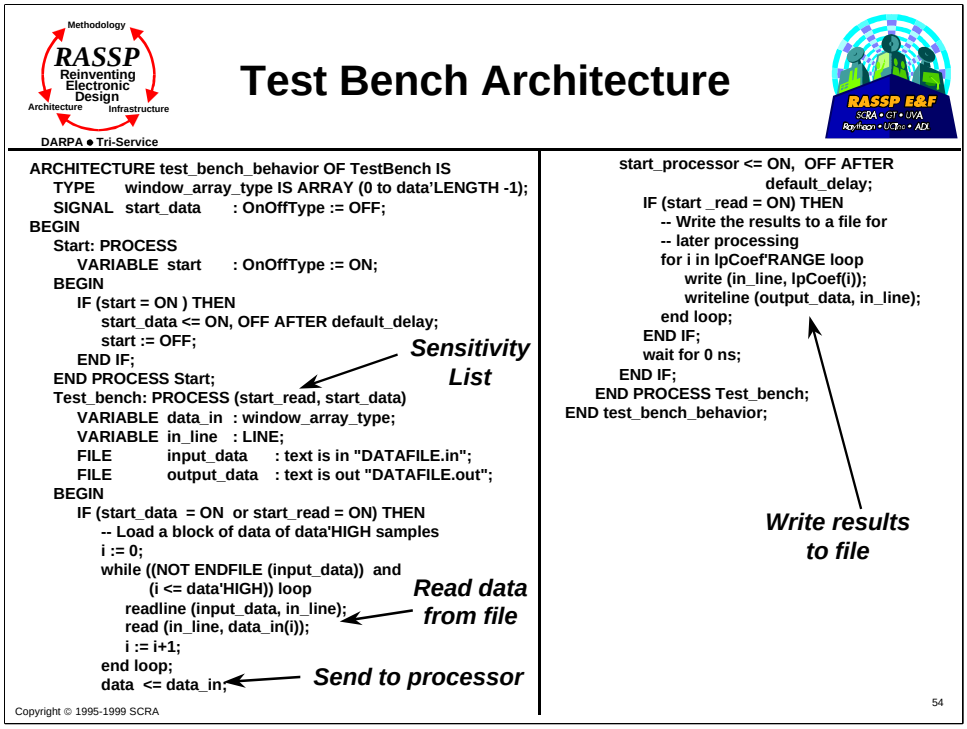
The architecture describing the functionality of the signal processor is shown above. It consists of one process executed in zero time and sensitive to the “start_processor” signal from its input port. When it receives this signal, the speech data is assumed to be present and the processor begins calculating the LP coefficients. It calls a sequence of procedures to perform the necessary functions (window_data, corr, levinson-durbin). When it has finished, it puts the results on the lpCoef lines and sets the trigger to the testbench.

Procedural calls are used to perform functionality.

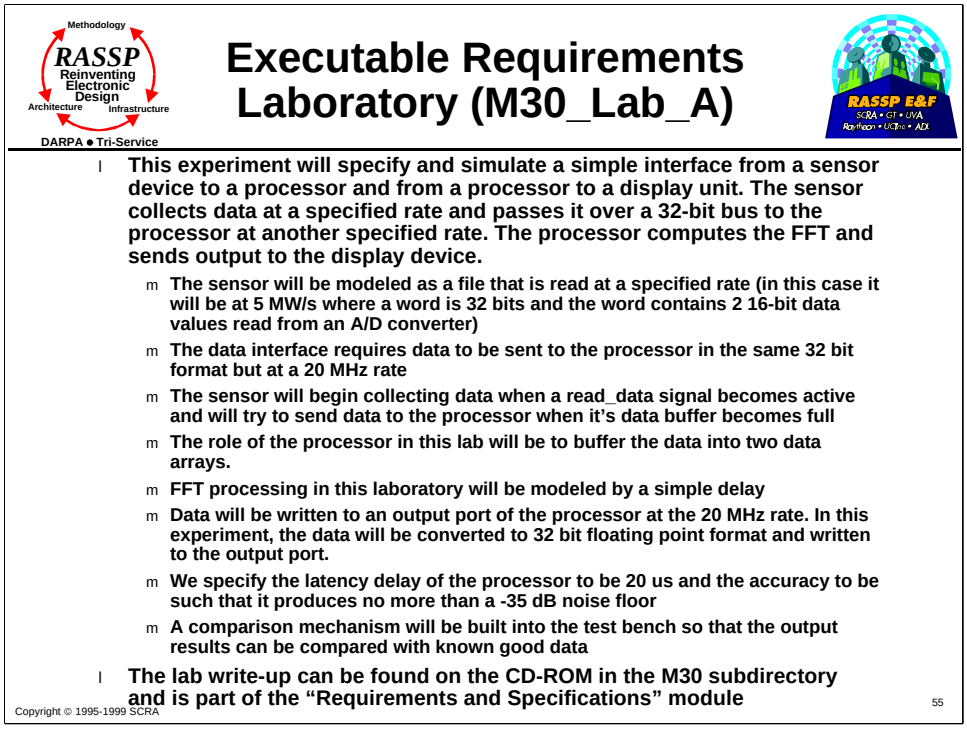
Results are put on output lines after the maximum specified delay time.



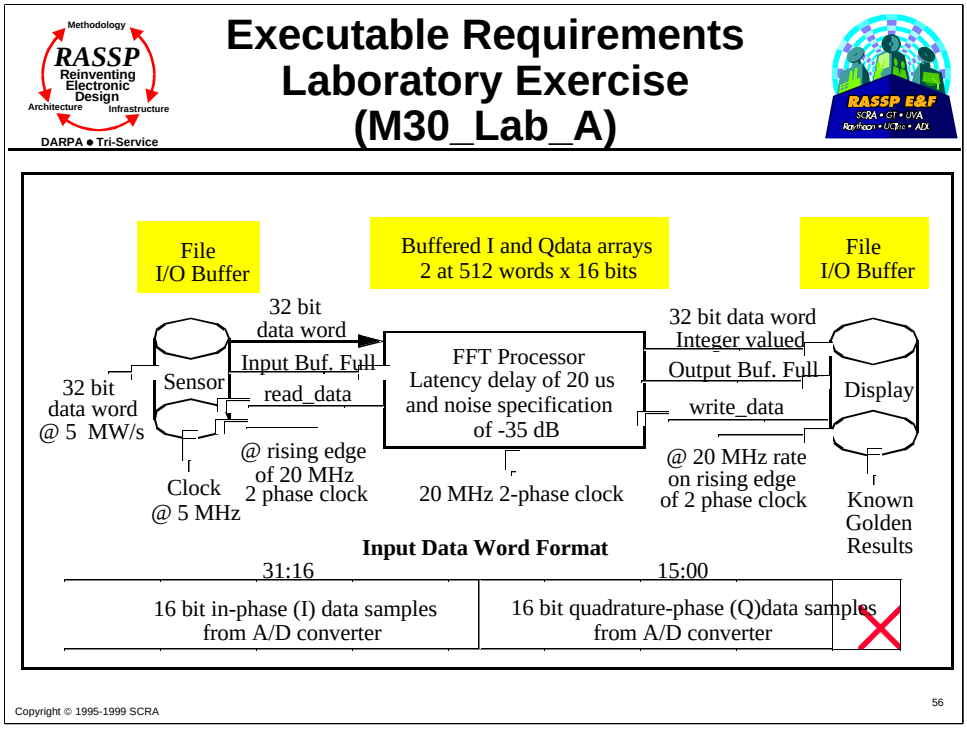
This is the entity description of the test bench for the signal processor. The ports used for input and output to the test bench include its data and control lines. The output data lines consist of 240 samples of speech data in the format "SpeechType" and the input data are the LP coefficients. The control input information indicates when the test bench starts reading the coefficients and the output control line indicates to the processor when to start processing the data sent by the test bench.



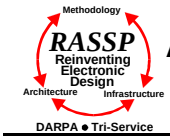
This is a possible architecture for the given test bench. It contains two processes, one to start the reading of data at time zero and the second to read and store data at the appropriate times in the future. The data is read from a file in the form of 12 bit speech samples. It is then sent to the processor by assigning it to the signal “data” and setting the trigger signal “start_processor”. The results are stored at the end of the routine when the “start_read” signal is set to ON.



The executable requirements laboratory is a supplementary part of module 30, "Requirements and Specifications". Since virtual prototyping can model a system or component at any level of abstraction, the executable requirements laboratory was done in VHDL and is representative of how requirements can be captured in a simulation language.

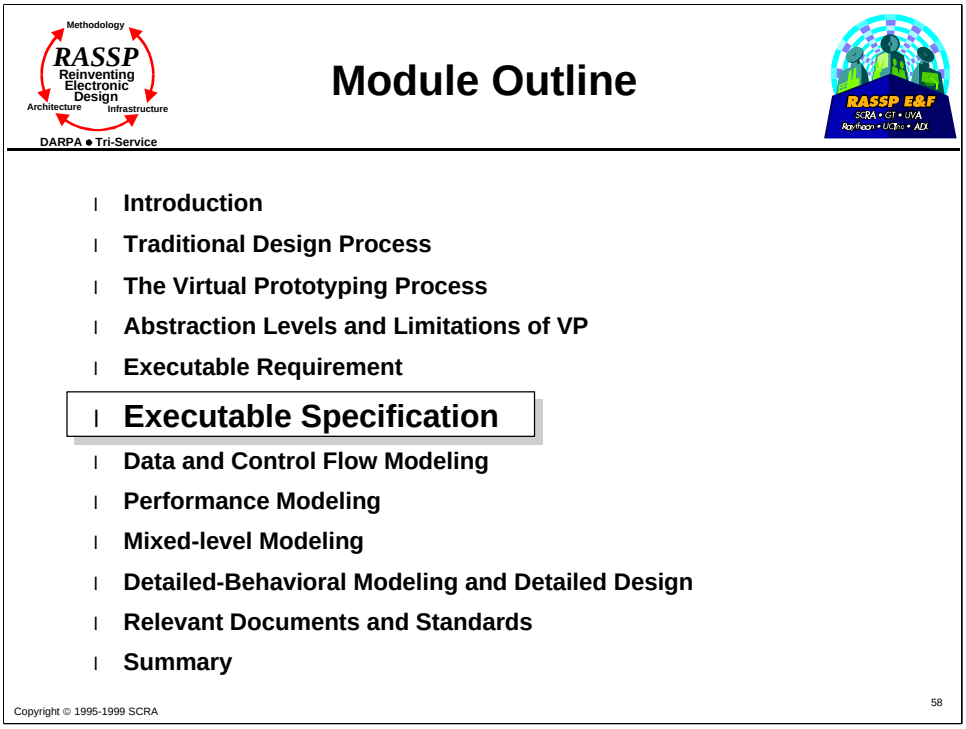


This figure illustrates the requirements for this laboratory. The complete write-up for the lab can be found in the M30 directory of the CD-ROM. It is in Word and PDF format and should be allotted approximately three hours for completion. The laboratory exercise has the student step through the modeling of system requirements. The same test bench is used in subsequent laboratory exercises for executable specifications.

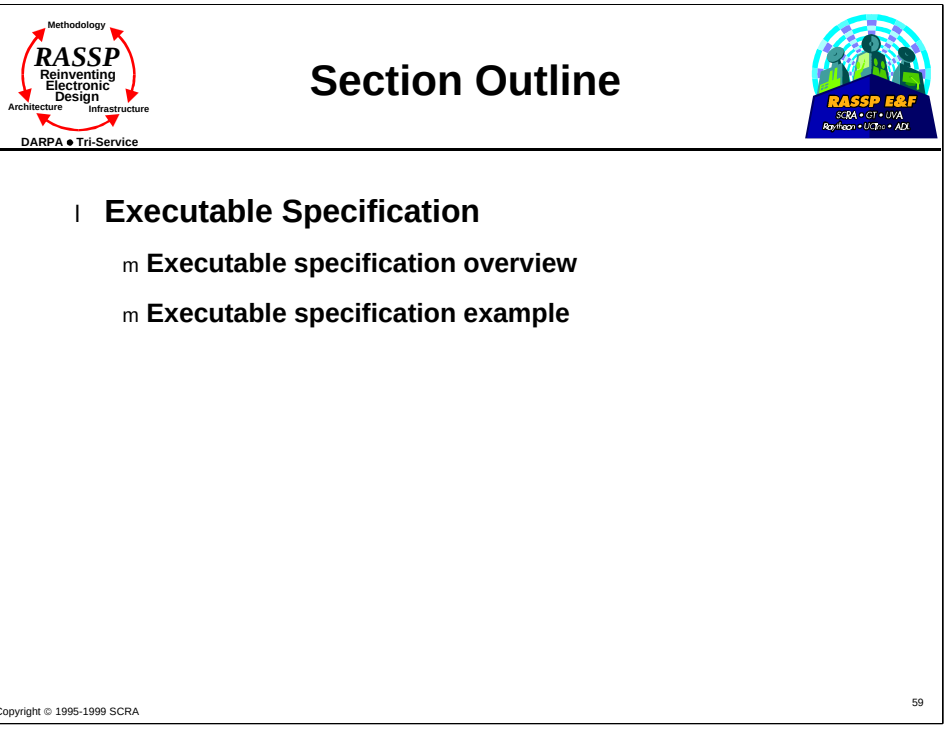


Additional CD-ROM Material on Executable Requirements

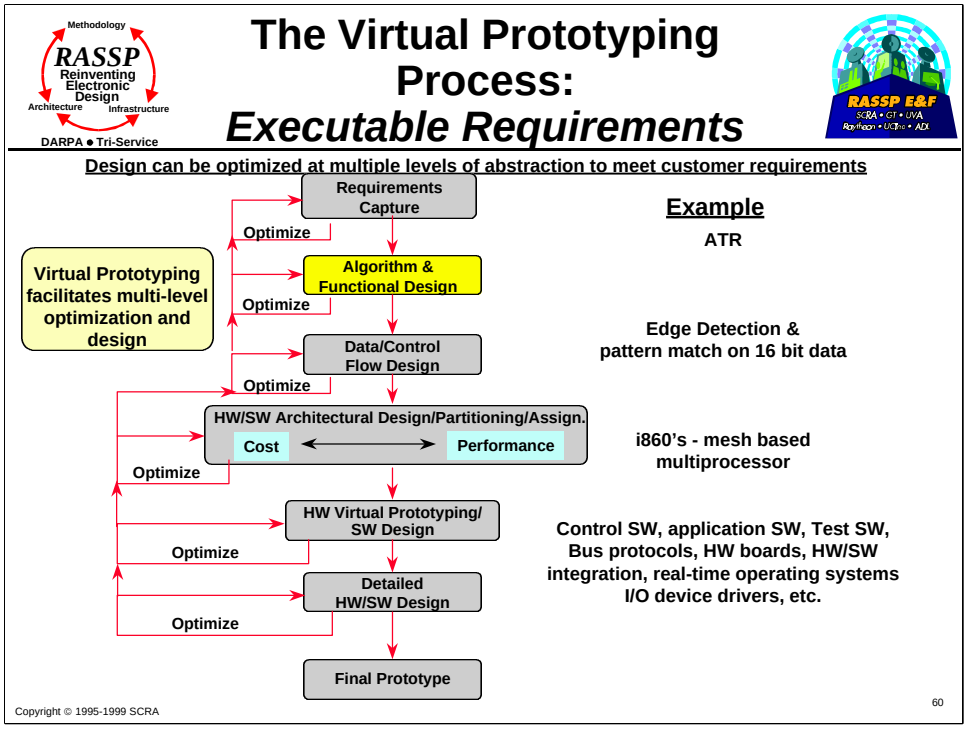
- **Module 30 provides a detailed account of executable requirements and specifications**
  - **This module can also be found on the CD-ROM**
- **Application note on executable requirements and specifications modeling can also be found on the CD-ROM**
  - **See the section on Application notes to obtain this information**
  - **"RASSP Definition and Role of Executable Requirement (ER-SPEC) and Design (ED-SPEC) Specifications Appnote"**



The next section will investigate the role of the executable specification that is created by the system developer in response to studying the requirements and executable information sent by the contractor. In this section, the role of the executable specification is described and it is shown how the specifications of a design can be captured for a simple example using VHDL as the modeling language. The executable specification primarily focuses on the system model and not the test bench. The executable requirements model focuses more on the test bench because it's role is to capture the customer's requirements while is concerned less of how the designers meet these requirements. The executable specification represents the first attempt at an implementation and serves as a guideline for further design refinements. It can be used to generate golden data as the design progresses. Comparisons can be done between later design refinements and the original executable specification in order to verify that the design is implementing the designers original intent.

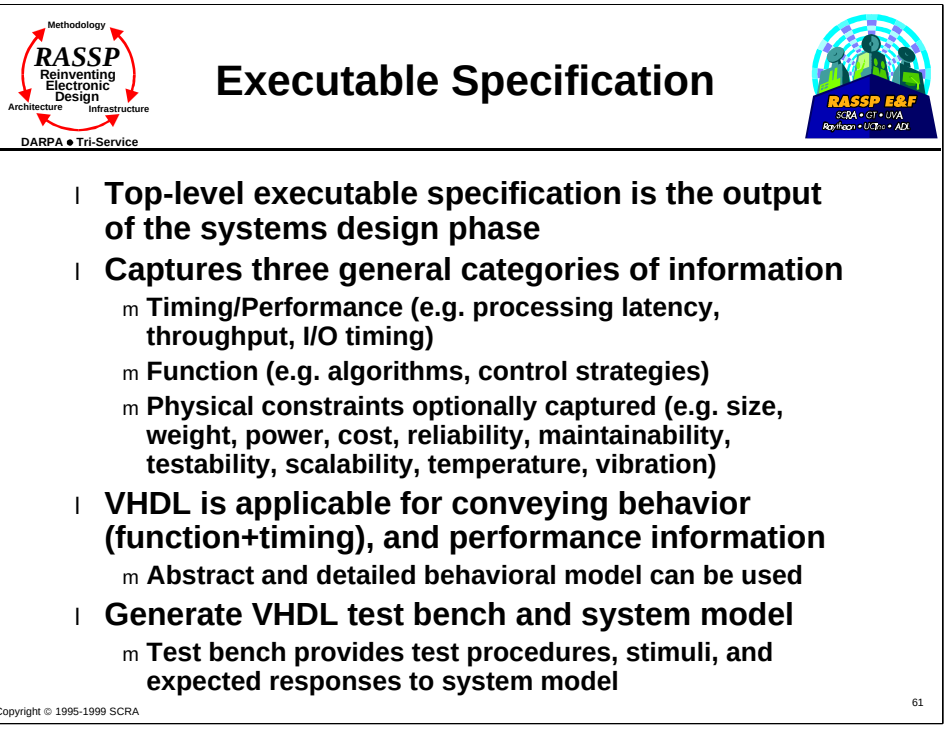


The executable specification evolves with the overall design process. This section will start with an overview of executable specifications and then focus on a short example that captures some of the details of the executable specification.



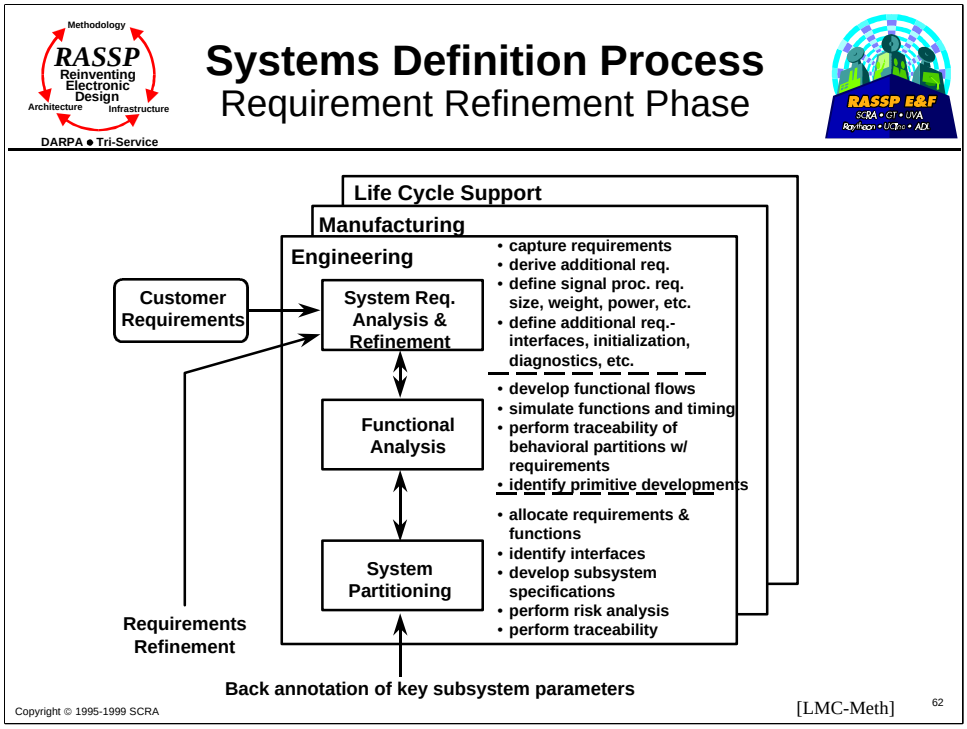
The VP process is a top-down design paradigm with optimization done at multiple levels of abstraction. There are a number of slides in this module that capture the essential flow of this design process but in a different format. In general, the first stage in any design process is the correct definition of the requirements for the design. Once the customer's requirements have been understood, then the system design team begins working on design specifications that attempt to satisfy all the customers requirements. In the above process flow, this represents the algorithm definition and functional design stage. At this stage, the algorithms used to implement the functions are refined. The size, weight, area, and power constraints may also be refined. At this point, initial computational complexity of the algorithms as well as fixed-point characteristics are analyzed. As the process proceeds from the top level, further refinement is done. At the data flow level, the algorithms are refined and data flow graphs that implement the functionality are explored. These data flow graphs are used in the HW/SW architectural design phase, where the nodes in the flow graph are either mapped to hardware or software. At this level in the process, the hardware and/or software are sized to meet the requirements. Time critical information is captured such as latency, throughput, and resource utilization. As more detailed design is done, the architectural elements are either developed as software modules or hardware components (ASICs, Processors, etc.) At the end of the process, final hardware and software are realized. At each stage in the process, the design is verified and optimized to meet the customer's requirements.

This slide illustrates the focus of the current section: Executable specifications



The top-level executable specification is considered the initial virtual prototype in the design process. It is the output of the systems design phase and captures three general categories of information, timing/ performance, function, and possibly some physical constraints. VHDL is a suitable language for conveying this information due to its expressive nature and ability to model behavior at many levels of abstraction. Associated with the virtual prototype at this level is its test bench and system model. The test bench provides test procedures, stimuli, and expected responses to the system model behavior and is usually written prior to development of the system model to serve as an executable requirement for the given design specification. By executing these requirements with the actual design, the design specification can be tested for proper behavior.

The test bench developed as part of the requirements phase can be used in this phase to test the specification.



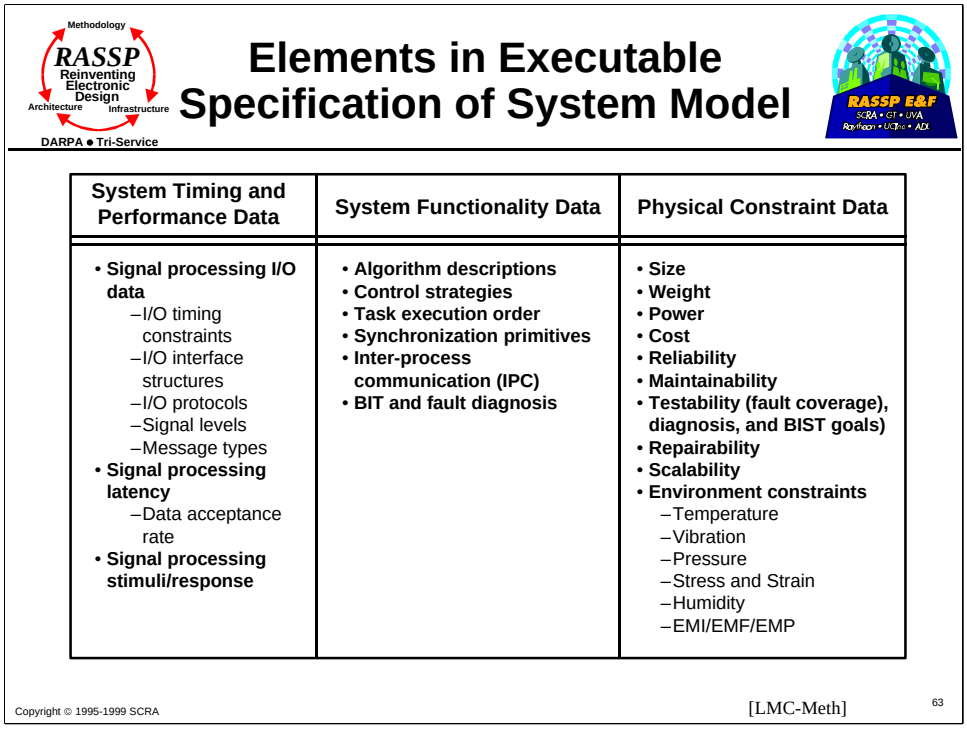
The systems definition process is comprised of three main functions; system requirements analysis and refinement, functional analysis, and system partitioning.

System requirements analysis captures the requirements in a tool such as RDD-100, derives additional requirements required for system design, defines signal processing requirements (size, weight, power, etc.), and non-signal processing requirements (interfaces required, initialization, diagnostics, etc.).

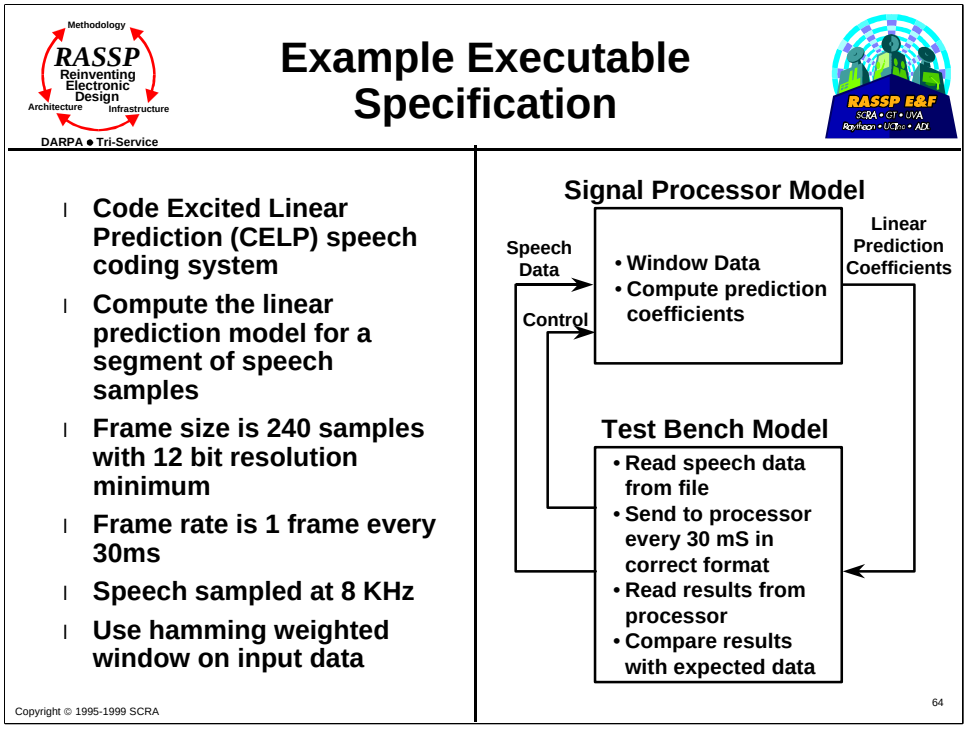
Functional analysis is designed to identify the functions, decompose the functions into requirements, and allocate these requirements to lower-level functions. The functional analysis process describes the requirements as a set of verifiable (simulatable) statements that can be used as a basis for systems design. Functional block diagrams are created that are used by subsequent processes to create and evaluate system configurations.

System partitioning takes the functions from the functional analysis and allocates them to entities within candidate configurations (at the subsystem level). It also allocates constraints to the candidate configurations and identifies interfaces required. The output of system partitioning is a set of functional, performance, and physical requirements for each subsystem in the baseline configurations. Performance verification is done by developing metrics and scenarios, developing simulatable models, and analyzing the results of simulation to determine the best configurations. For RASSP, an executable specification is used to do these trade-offs and is the starting point for architecture selection.

As the subsystem designs progress, key subsystem parameters are back annotated, and system-level simulations are re-run to ensure that performance is maintained. At the same time, requirements can be refined as the subsystems are developed.



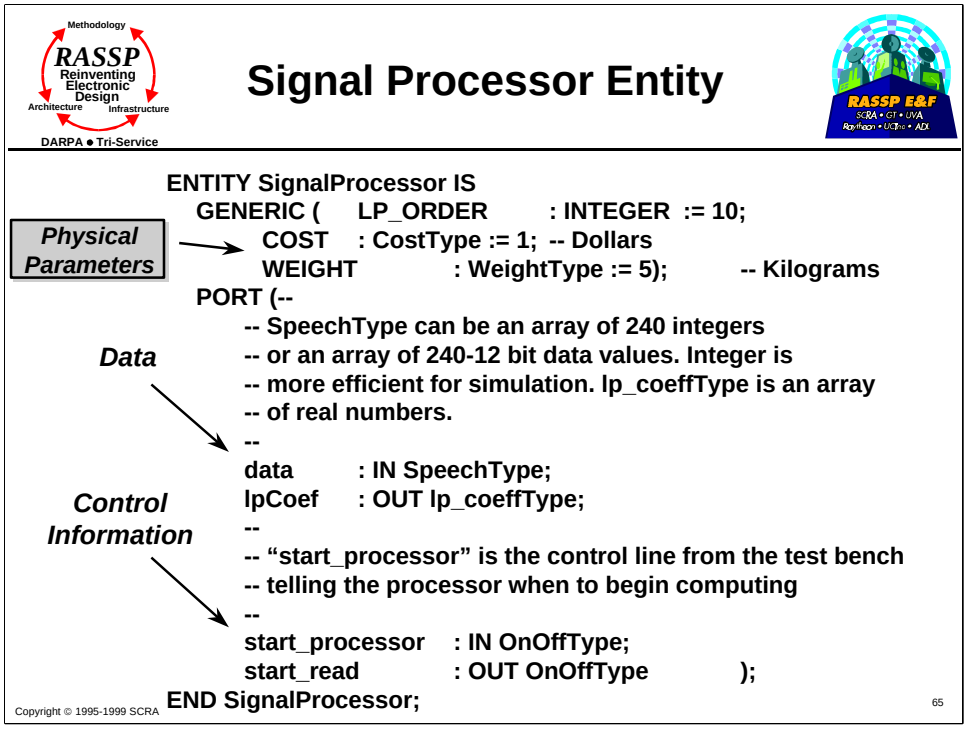
This chart presents the elements contained in the executable specification. The three main categories of the previous slides are expanded upon. The data in this simulation is not fixed at the end of the systems definition design process but can be modified as more information becomes available from future design stages. For example, the size and weight can be estimated initially and as more numbers become available the high level model is updated to track the lower level details.



This example will be used to illustrate how an executable specification can be specified. This example was also used in the preceding section on executable requirements and will be used in the following section on data flow modeling. The executable requirement focused primarily on the test bench. The executable specification will focus more on the system model, in this case the signal processor model. The data flow section will focus on refinements of the signal processor model.

This is a simple example for illustrative purposes only. Speech is sampled at 8 KHz and input to a signal processor in frame sizes of 240 samples representing 30 ms of speech data. The processor windows the speech data using a hamming window function and calculates the linear prediction model for each of the speech segments. The equations for the hamming window and the linear prediction algorithm are presented in a later slide, where algorithm analysis is performed.

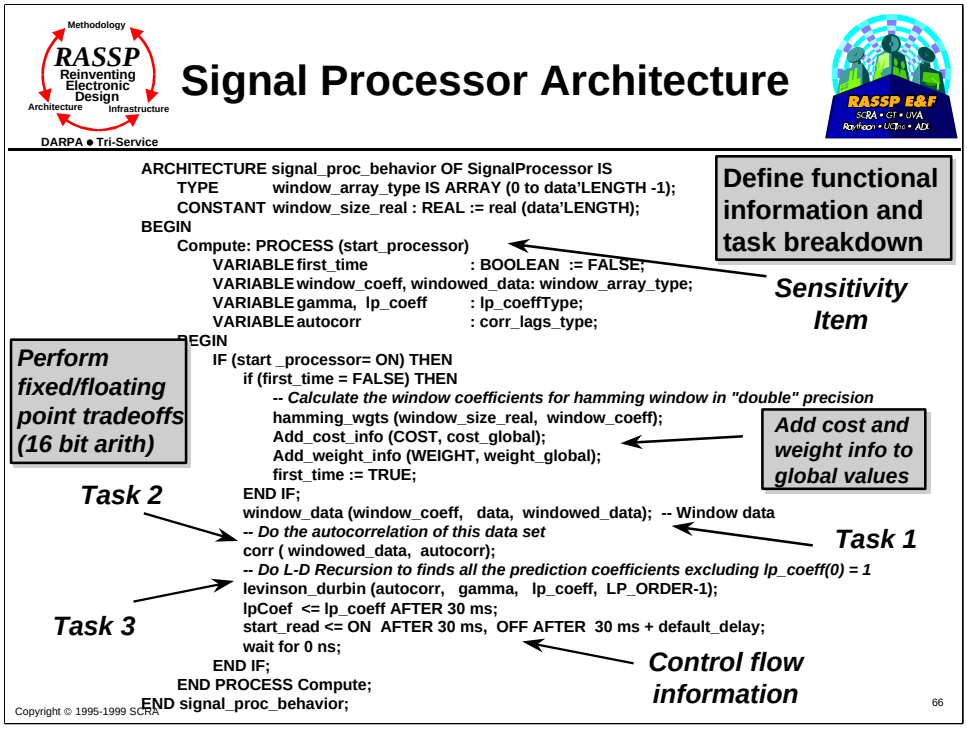
The test bench inputs data to the signal processor and monitors its outputs. The test bench reads it's speech data from a file and outputs the linear prediction coefficients to a file after the processor has completed it's computations. The same test bench developed in the executable requirements section is used here to apply stimuli to the model under test and monitor its outputs for correct responses.



This is the entity description of the signal processor. Since the processor is a linear prediction computational engine, the linear prediction (LP) order is passed as a generic. The ports used for input and output to the processor include its data and control lines. The data lines consist of 240 samples of speech data in the format "SpeechType" and the output data are the LP coefficients. The control input information indicates when the processop starts computing the coefficients and the output control line indicates to the test bench when to read the results. There has been only one addition to the entity model during the executable specification stage and that is the introduction of some of the physical parameters. They are captured by the model in this example as generic parameters.

Additional information passed to this entity include some of the physical constraints from the previous slide such as cost, weight, etc. Additional information could be included such as power, test inputs, etc.

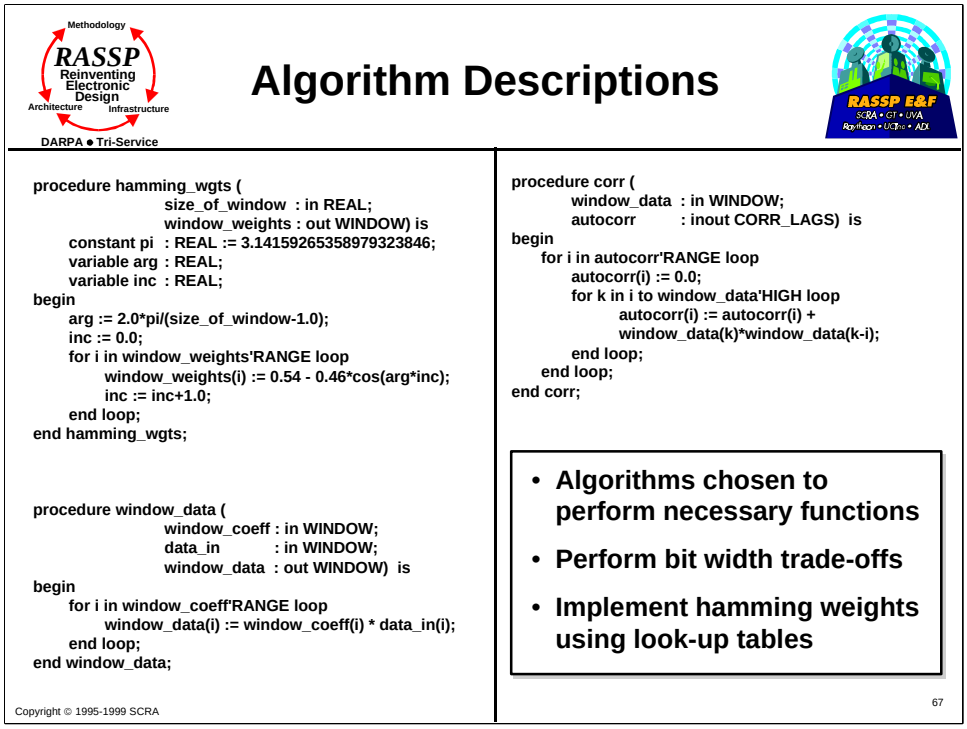
This example shows the case where physical parameters were added to the model entity description.



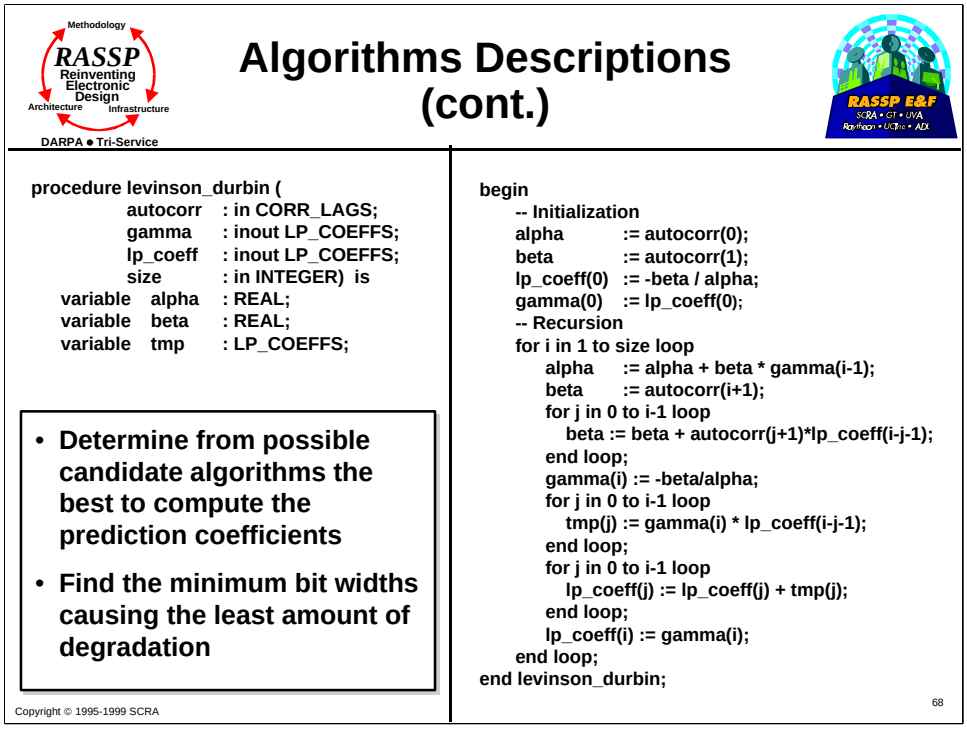
The architecture describing the functionality of the signal processor is shown above. It consists of one process executed in zero time and sensitive to the “start_processor” signal from its input port. When it receives this signal, the speech data is assumed to be present and the processor begins calculating the LP coefficients. It calls a sequence of procedures to perform the necessary functions (window_data, corr, levinson-durbin). When it has finished, it puts the results on the lpCoef lines and sets the trigger to the testbench.

We include additional information at this stage by defining the process flow, functionality, and task breakdown. Fixed/floating point decisions are made at this point by determining the optimal bit widths to do the computations without loosing the quality of the speech data.

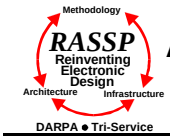
The physical constraint information can be added to global signals to keep information on cost, weight, etc. when there are other components in the system contributing to the total.



Also in this stage, we define how each of the algorithms are to be implemented. The hamming weights would most likely be implemented in a lookup table but computed from the equations above and rounded to the amount of digits required. The procedures for windowing data and computing the autocorrelation are also defined.

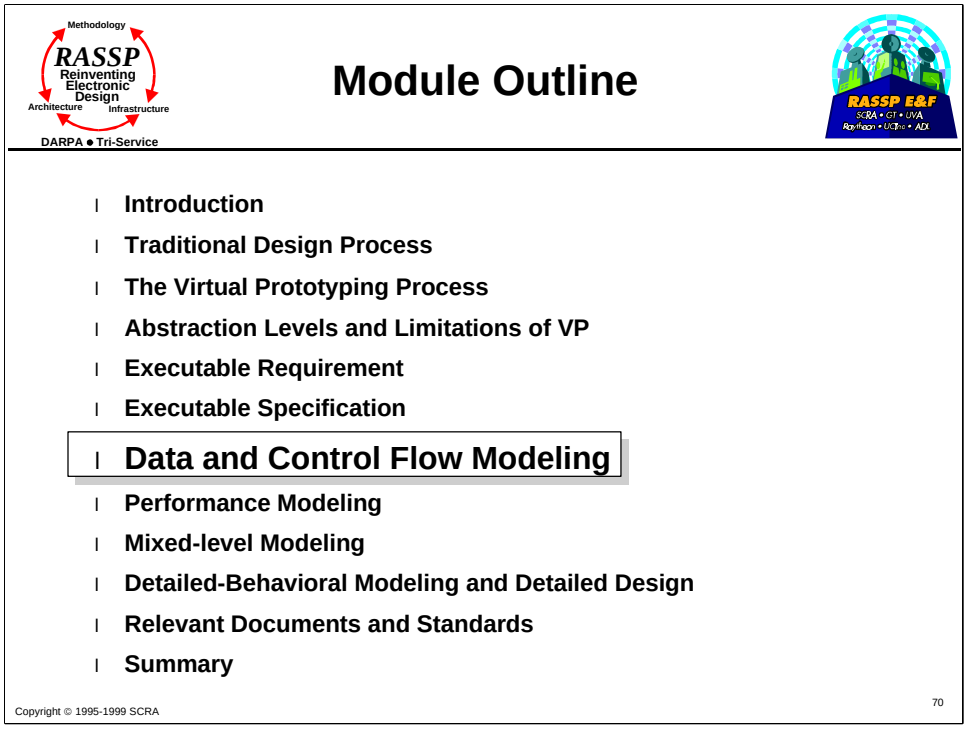


There are many algorithms for computing the linear prediction coefficients. In this design stage, we determine the best algorithm to use for the application. In this case, the levinson-durbin algorithm was chosen due to its efficient method for computation.



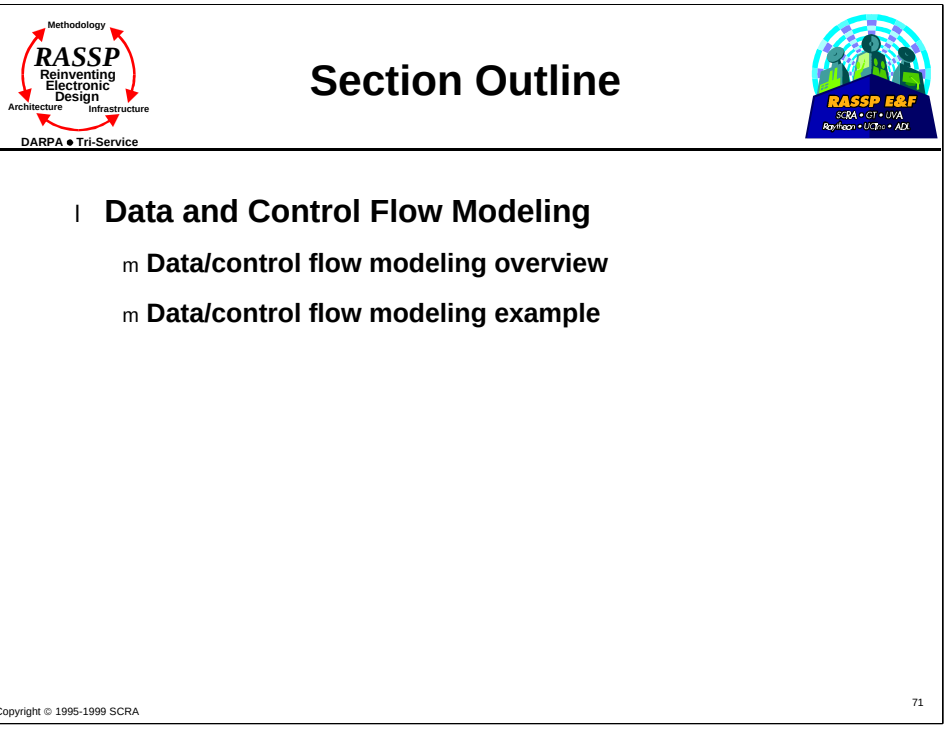
Additional CD-ROM Material on Executable Specifications

- **Module 30 provides a detailed account of executable requirements and specifications**
  - **This module can also be found on the CD-ROM**
- **Application note on executable requirements and specifications modeling can also be found on the CD-ROM**
  - **See the section on Application notes to obtain this information**
  - **"RASSP Definition and Role of Executable Requirement (ER-SPEC) and Design (ED-SPEC) Specifications Appnote"**

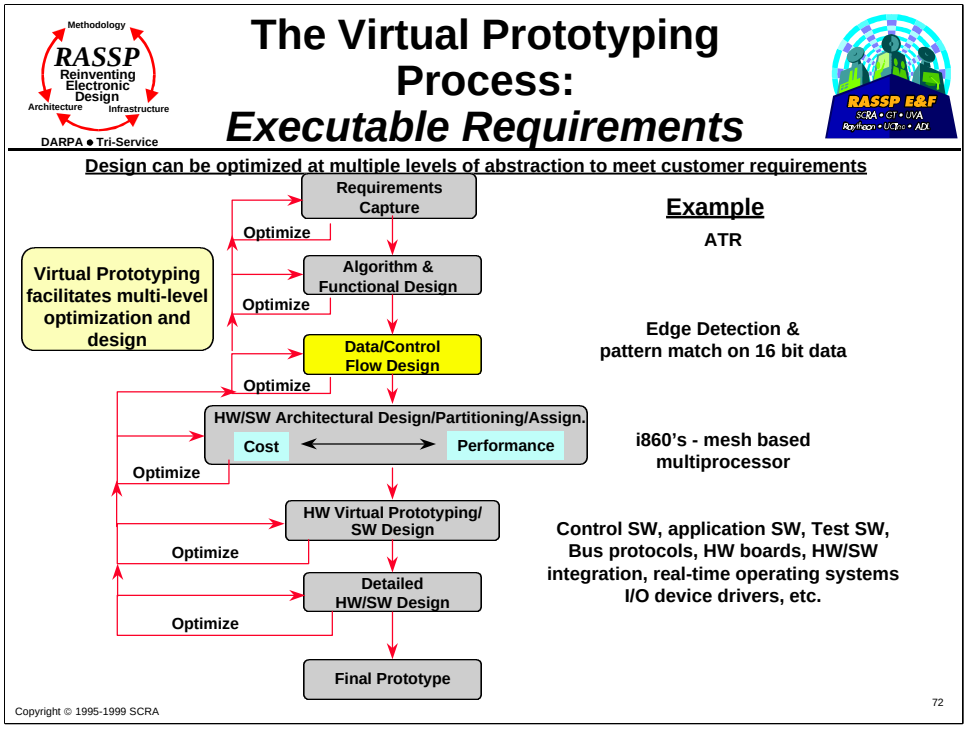


This section presents the next type of model found in the design process. Data and control flow modeling take the executable specification and develop the behavior of the system functionality in terms of how data should flow through the system. Control flow determines how the data flow will be manipulated.

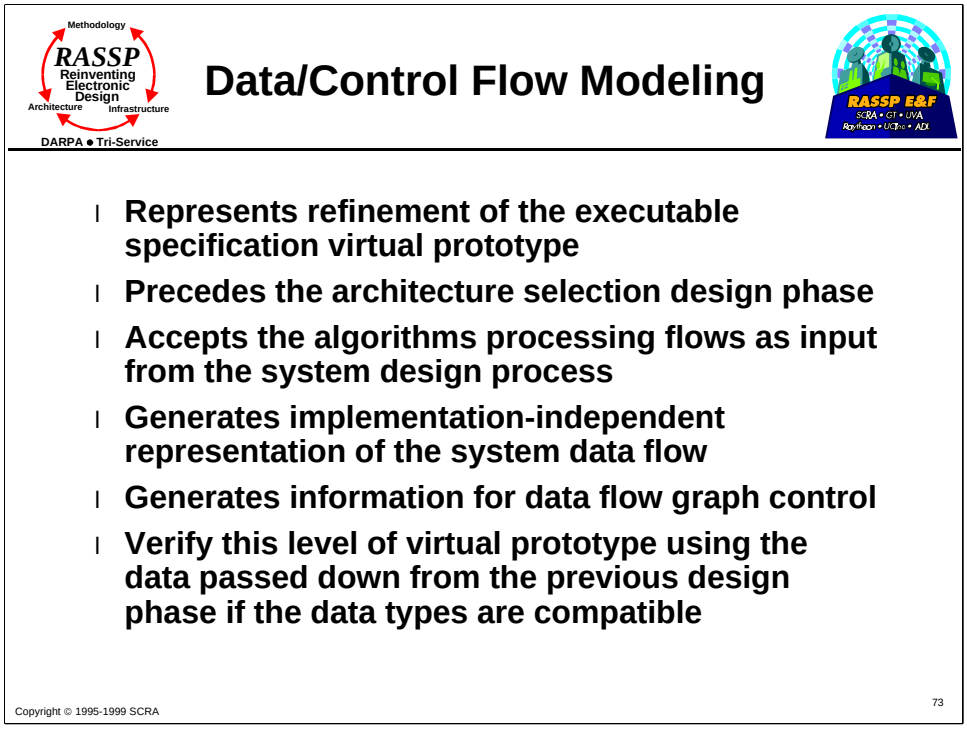
This section first describes what is meant by data and control flow modeling. It then presents an example which is a refinement of the preceding examples presented in the last two sections. The example provides an analysis of how the algorithm is continually refined as the design process progresses.



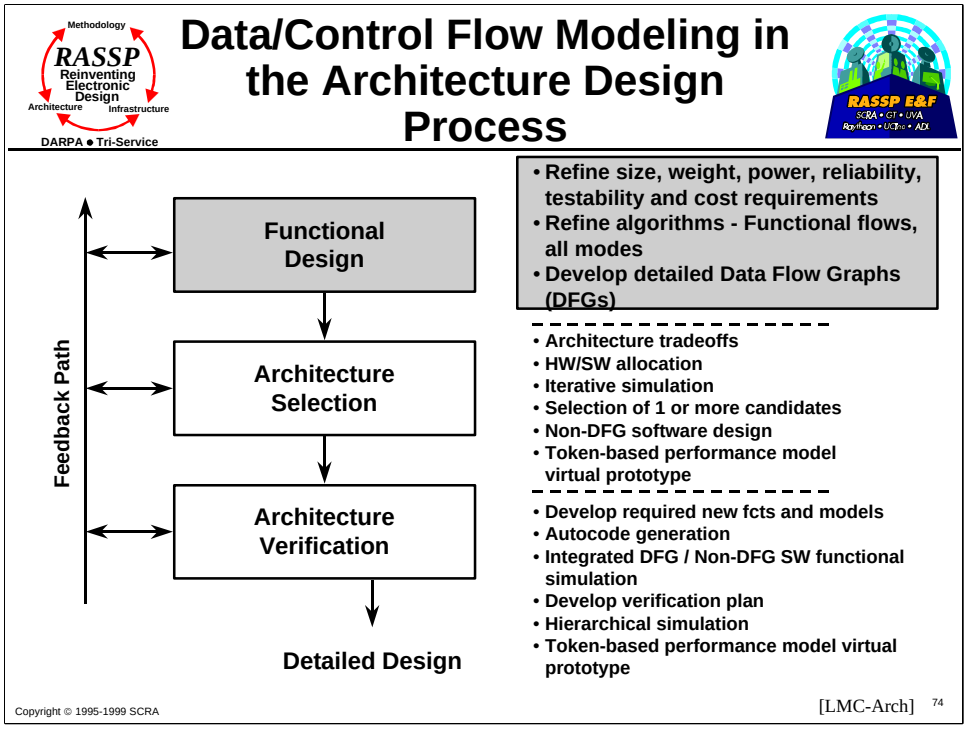
The initial few slides describe what is data and control modeling. The last few slides give an example of how it can be done in VHDL.



The VP process is a top-down design paradigm with optimization done at multiple levels of abstraction. There are a number of slides in this module that capture the essential flow of this design process but in a different format. In general, the first stage in any design process is the correct definition of the requirements for the design. Once the customer's requirements have been understood, then the system design team begins working on design specifications that attempt to satisfy all the customers requirements. In the above process flow, this represents the algorithm definition and functional design stage. At this stage, the algorithms used to implement the functions are refined. The size, weight, area, and power constraints may also be refined as well. At this point, initial computational complexity of the algorithms as well as fixed-point characteristics are analyzed. As the process proceeds from the top level, further refinement is done. At the data flow level, the algorithms are refined and data flow graphs that implement the functionality are explored. These data flow graphs are used in the HW/SW architectural design phase, where the nodes in the flow graph are either mapped to hardware or software. At this level in the process, the hardware and/or software are sized to meet the requirements. Timing critical information is captured such as latency, throughput, and resource utilization. As more detailed design is done, the architectural elements are either developed as software modules or hardware components (ASICs, Processors, etc.) At the end of the process, final hardware and software are realized. At each stage in the process, the design is verified and optimized to meet the customer's requirements.



Data flow modeling represents a refinement of the executable specification virtual prototype of the previous stage by explicitly specifying the data and control flow of the algorithms. It is contained in the architecture selection design process and accepts the processing flows from the systems design process as input. The data flow model is an implementation-independent representation of the system data flow. The control flow determines how the data flow graph will be manipulated to perform the desired functions. Data from the executable specification can be used to verify the performance of the data/control flow models.



The architecture definition process transforms processing requirements into a candidate architecture of hardware and software elements.

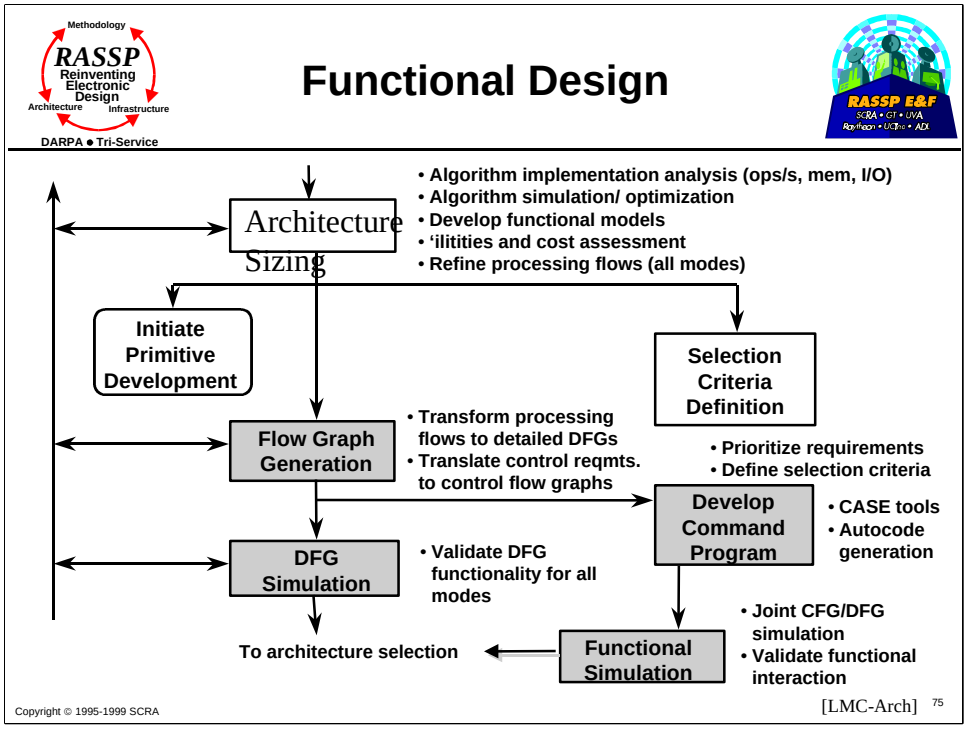
The architecture definition process is a new HW/SW codesign process in the VP methodology for high-level virtual prototyping and simulation. The primary concern in the architectural definition process is to select and verify an architecture for the digital system that satisfies the requirements passed down from the systems definition process.

The overall task is to:

- Define and evaluate various architectures
- Select one or more for detailed evaluation that appear to meet the requirements
- Validate the chosen architecture(s) for both function and performance before detailed design

Concurrently, each selected architecture is evaluated with respect to size, power, weight, cost, schedule, testability, reliability, etc.

The process is library based and data flow graph (DFG) driven. The DFGs are created from the processing flows passed down from the systems definition process. Reuse of both architecture elements and software primitives significantly shortens the design cycle. VHDL performance model simulations are used to verify that system requirements are met and this will be the focus of the following section. Software performance is also modeled for its impact on the total processing time.



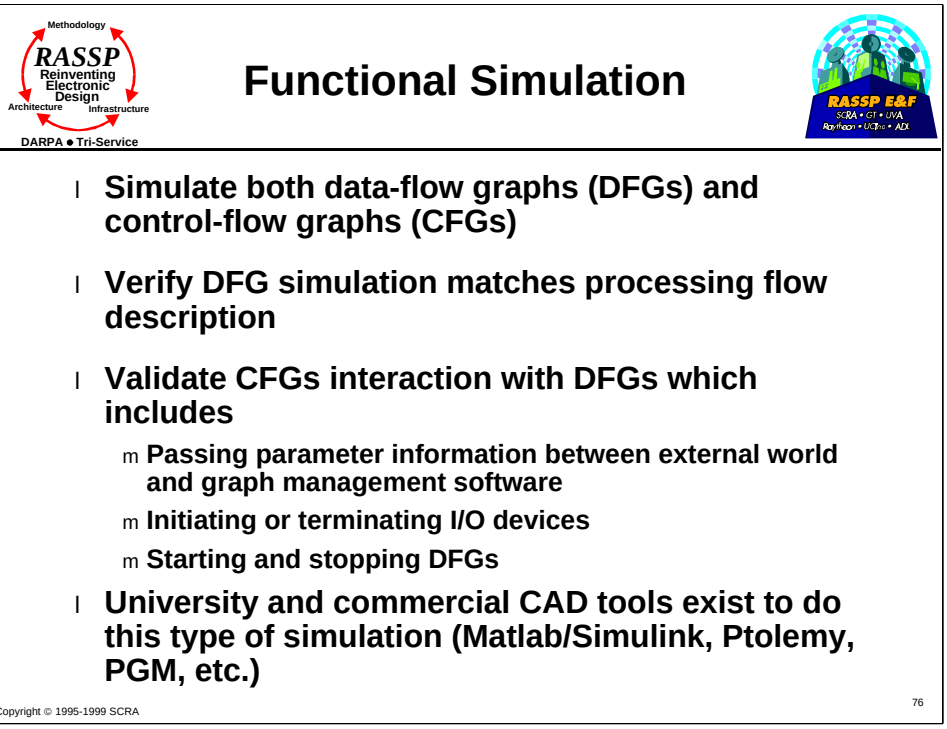
The functional design step provides a more detailed analysis of the processing requirements resulting in initial sizing estimates, detailed data and control flow graphs for all required processing modes to drive the HW/SW codesign, and the criteria for architecture selection. The control flow graphs provide the overall signal processor control, such as mode switching (referred to as the command program). Functional simulators support the execution of both the data and control flow graphs.

Architecture sizing helps to analyze the system requirements and processing flows for all the required modes of the system in terms of estimated operations per second, memory requirements, and I/O bandwidths.

Selection criteria definition helps prioritize the overall system requirements and the derived requirements and establishes a selection criteria. The selection criteria provides the necessary basis for subsequent architecture trade-off analysis. A trade-off matrix is used to formalize the selection criteria. It contains top-level requirements allocated to the signal processor.

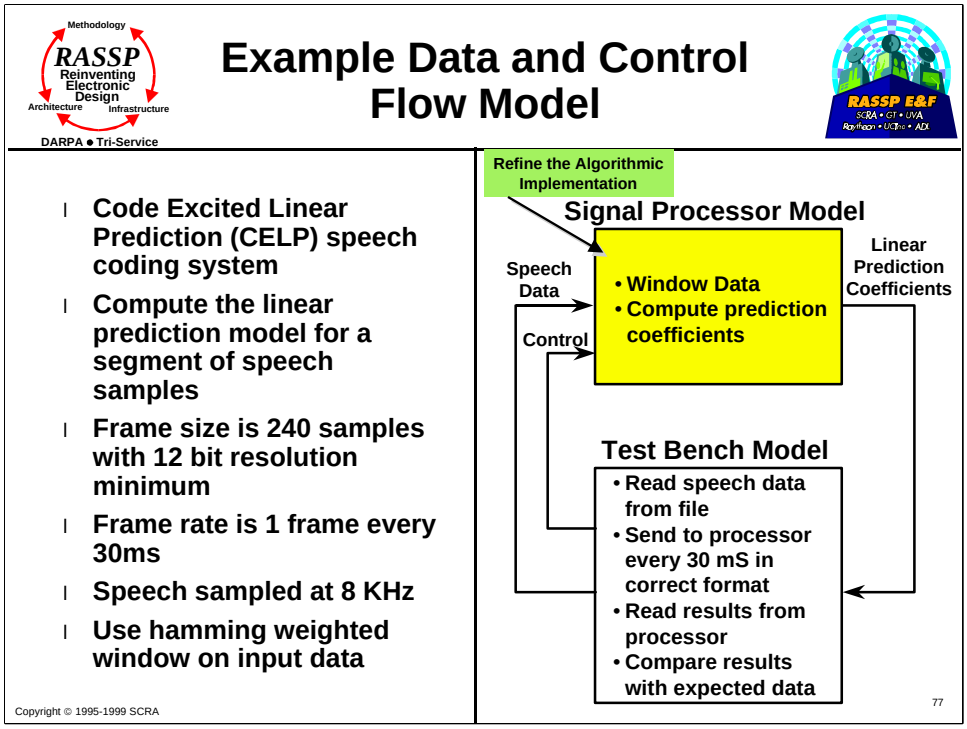
Flow graph generation transforms the finalized algorithm processing flows into detailed DFGs as the first step in HW/SW codesign. The DFGs are made up of reusable library elements, which may represent either hardware or software. The DFGs are the basis for both the architecture synthesis, the detailed software generation, and potentially custom processor synthesis. Each DFG is simulated to provide data for comparison with the algorithmic flows developed during the systems process (executable spec). Control flow requirements are transformed into the control flow graphs (CFGs) required to manipulate the DFGs according to a defined set of rules. This DFG control is referred to as command processing. Conceptually, the command program manipulates objects. The objects are the DFGs and their data structures. The command program must be able to accept messages from outside the signal processor, interpret those messages, and generate the appropriate control information to stop graphs, start graphs, initiate I/O, set graph parameters, etc. The command program can be developed through standard software development CASE tools or through the tools that provide autocode generation capability.

Functional simulation verifies both the DFGs, the CFGs, and their interrelationships.



When all the code is created for both the DFGs and CFGs, simulations are performed to verify the code's behavior. This is compared with the processing flows described by the less refined version of the executable specification of the previous phase. The CFGs interaction with the DFGs are validated.

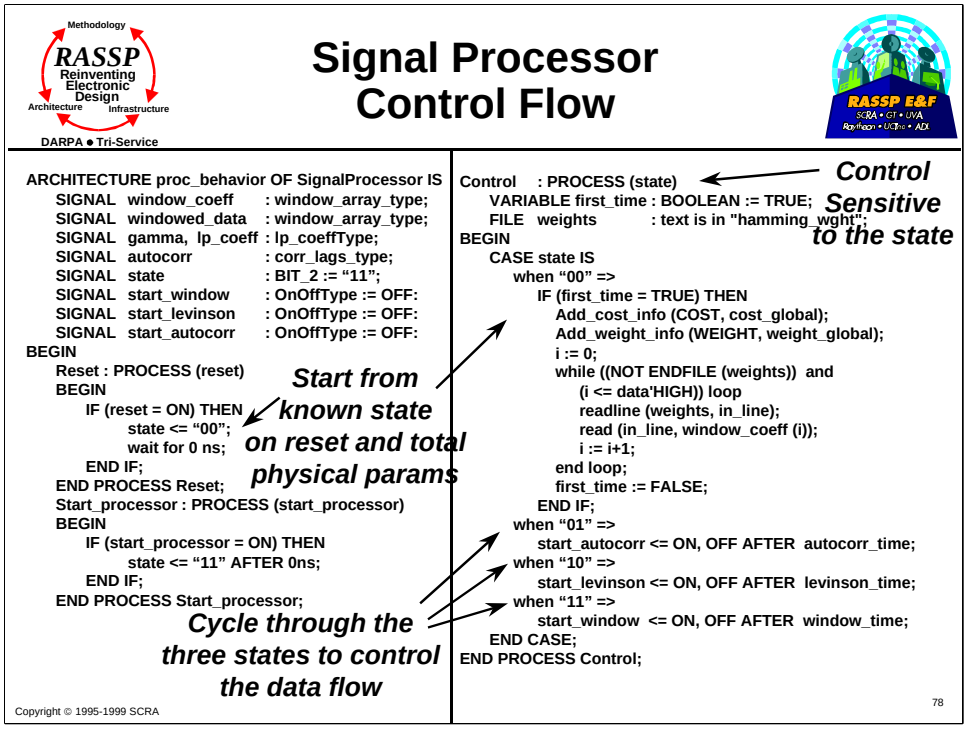
A number of CAD tools exist in both the University and commercial industry to perform data and control flow simulations. Some of these include Matlab/Simulink, Ptolemy from UC Berkeley, and PGM from the Department of the Navy, to name just a few. In this section, it is shown how VHDL can also be used to model the data and control flow of a system's algorithms.



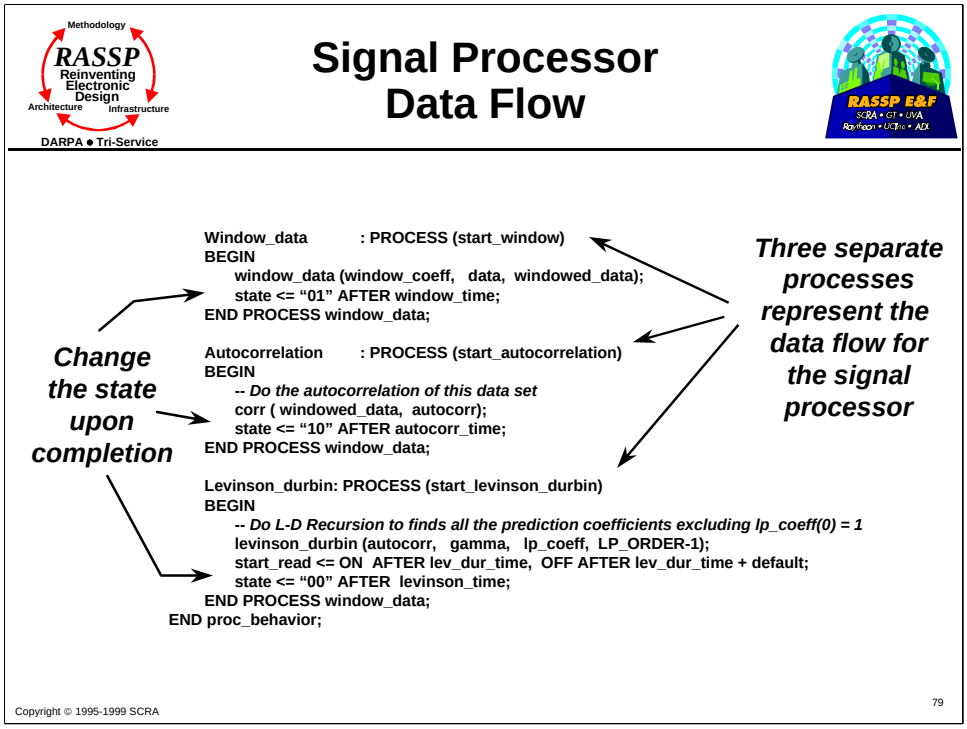
This example will be used to illustrate how an executable specification can be specified. This example was also used in the preceding section on executable requirements and will be used in the following section on data flow modeling. The executable requirement focused primarily on the test bench. The executable specification focused more on the system model, in this case the signal processor model. In this section, the focus is on refinements to the signal processor model. In this case, more details are placed on how selected functions interact, what data flows between the functions, and the control sequence of the execution (scheduling) of the functions.

This is a simple example for illustrative purposes only. Speech is sampled at 8 KHz and input to a signal processor in frame sizes of 240 samples representing 30 ms of speech data. The processor windows the speech data using a hamming window function and calculates the linear prediction model for each of the speech segments. The equations for the hamming window and the linear prediction algorithm are presented in a later slide, where algorithm analysis is performed.

The test bench inputs data to the signal processor and monitors its outputs. The test bench reads it's speech data from a file and outputs the linear prediction coefficients to a file after the processor has completed its computations. The same test bench developed in the executable requirements section is used here to apply stimuli to the model under test and monitor its outputs for correct responses.

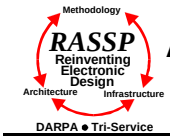


This slide and the next represent the modifications to the architecture of the signal processor with the control and data flow included. This slide shows the control flow. It starts from a known state with the variable "reset" turned ON. When it is ON, the state gets set to "00" and reset gets turned OFF. In state "00", which is only entered once, the estimates for cost and weight along with any other physical constraints are included in the system design. Also, the hamming weights, which are now stored as a lookup table in memory are read from a file and put into storage. Once this is complete the data flow control begins when the "start_processor" signal comes from the test bench. It sets the state to "11" resulting in the "start_window" process to be started. When this process is complete, it sets the state to "01" which initiates the autocorrelation process. Next, the autocorrelation process sets the state to "10" which then triggers the Levinson-Durbin process. Finally, the Levinson-Durbin process triggers the test bench to read the data and it also sets the state back to "00" where now nothing is done. We now wait for another "start_processor" from the test bench.



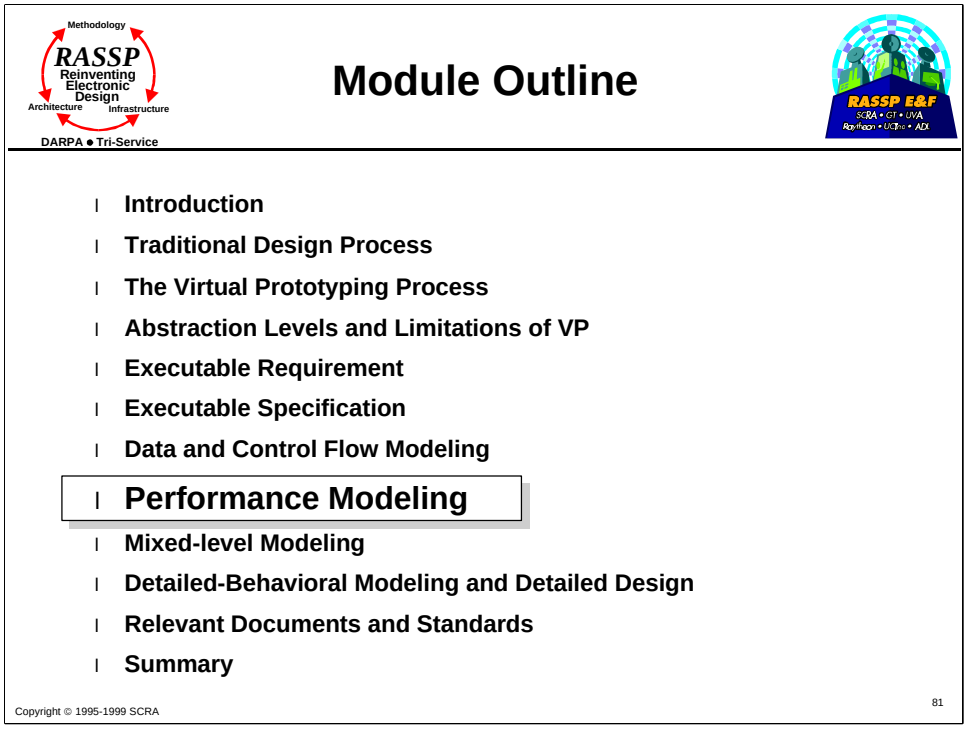
This slide represents the data flow of the signal processor. There are three separate processes with their own respective trigger signals from the control flow. When they are triggered, they do their function and then update the state for the controller.

In this example, each function is modeled using a separate process and the control flow is modeled through a separate state machine process that sends commands via signals to each of the data computational processes when they are required to execute.

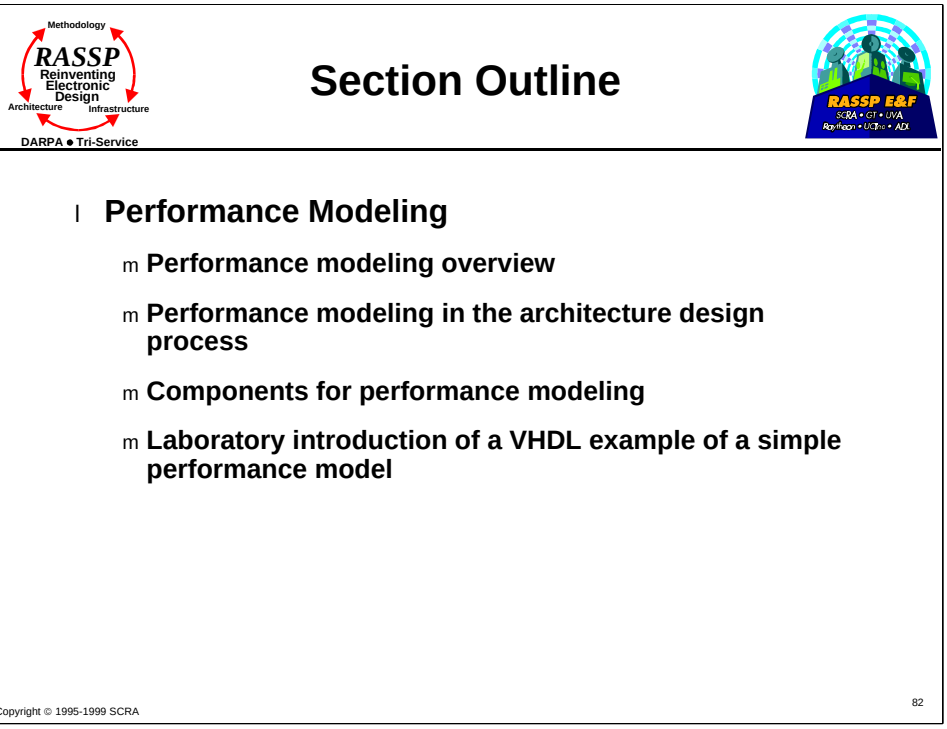


Additional CD-ROM Material on Data Flow Graph Design

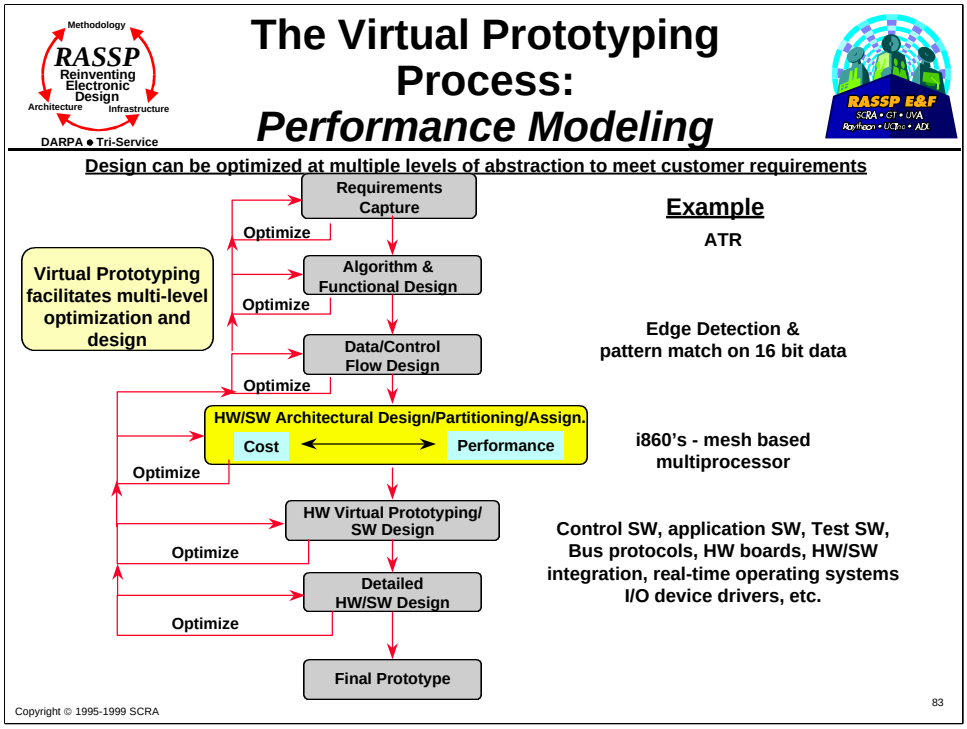
- **Application note on data flow graph design can also be found on the CD-ROM**
  - **See the section on Application notes to obtain this information**
  - **"RASSP Data Flow Graph Design" application note**
  - **See also "RASSP Autocoding for Deliverable, real-time Signal Processing System (DSP) Control" application note**



The next type of model used in the design process is the performance model. It provides a method to rapidly test architecture design candidates that meet system requirements. A thorough investigation of performance modeling is contained in the M59 module on the CD-ROM. It contains over 200 slides on the topic and a couple of laboratory exercises. In that module, VHDL performance modeling is discussed in detail as well as other approaches to modeling at the performance level. This section gives a 30 minute overview of the topic area and defers the details to module 59.

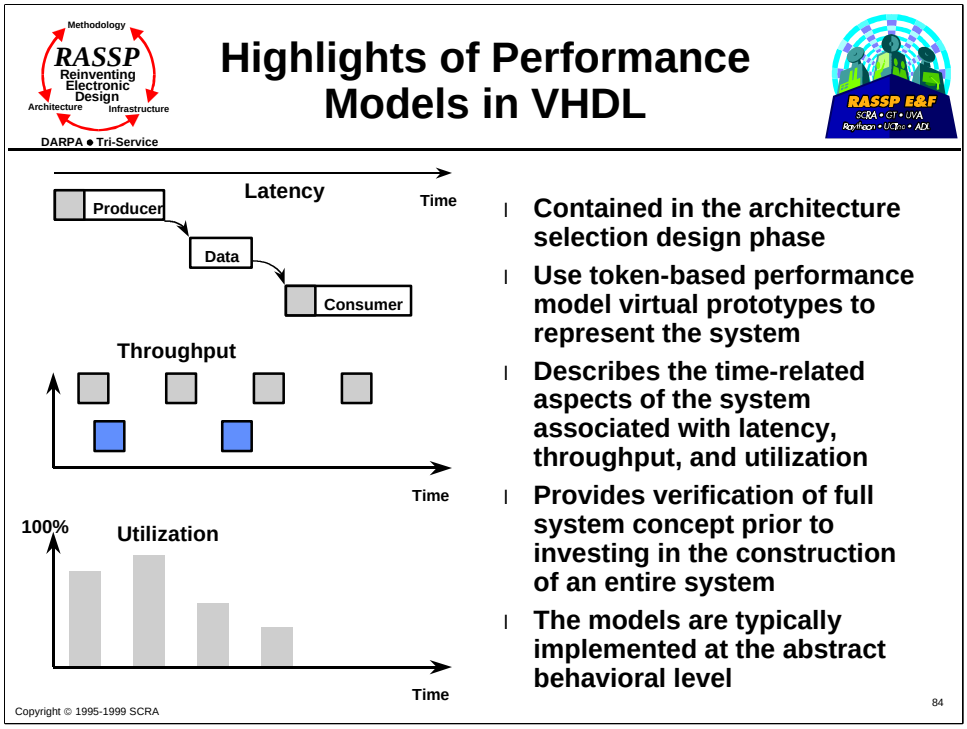


This section will first introduce performance modeling and the critical issues in the design process that it tries to address. It will then look at how it fits into the overall virtual prototyping top-down design process. At the end, it introduces the laboratory for the performance modeling module, M59. More details of performance modeling can be found in that module.



The VP process is a top-down design paradigm with optimization done at multiple levels of abstraction. There are a number of slides in this module that capture the essential flow of this design process but in a different format. In general, the first stage in any design process is the correct definition of the requirements for the design. Once the customer's requirements have been understood, then the system design team begins working on design specifications that attempt to satisfy all the customers requirements. In the above process flow, this represents the algorithm definition and functional design stage. At this stage, the algorithms used to implements the functions are refined. The size, weight, area, and power constraints may also be refined as well. At this point, initial computational complexity of the algorithms as well as fixed-point characteristics are analyzed. As the process proceeds from the top level, further refinement is done. At the data flow level, the algorithms are refined and data flow graphs that implement the functionality are explored. These data flow graphs are used in the HW/SW architectural design phase, where the nodes in the flow graph are either mapped to hardware or software. At this level in the process, the hardware and/or software are sized to meet the requirements. Timing critical information is captured such as latency, throughput, and resource utilization. As more detailed design is done, the architectural elements are either development as software modules or hardware components (ASICs, Processors, etc.) At the end of the process, final hardware and software are realized. At each stage in the process, the design is verified and optimized to meet the customer's requirements.

This slide illustrates the focus of the current section: Performance Modeling.

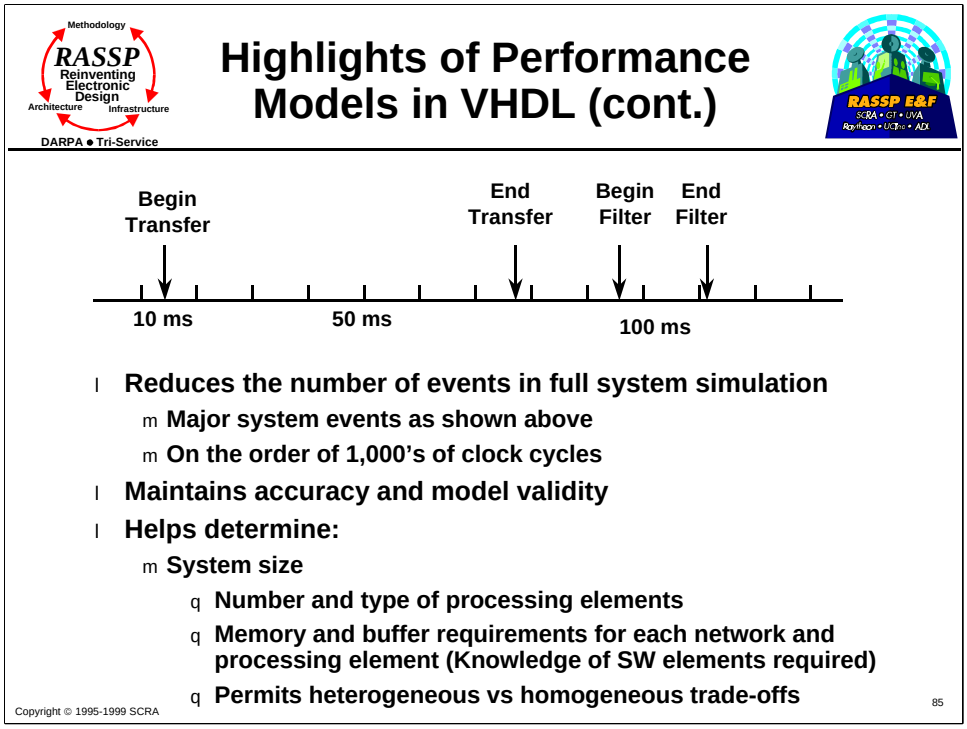


The next three slides list some of the highlights of performance models. It serves as the main simulatable entity in the architecture selection process and is represented as a token-based performance model virtual prototype. It is used to describe the time-related aspects of the system and provides verification of the full system concept at a high efficiency. It is implemented at a higher level of abstraction.

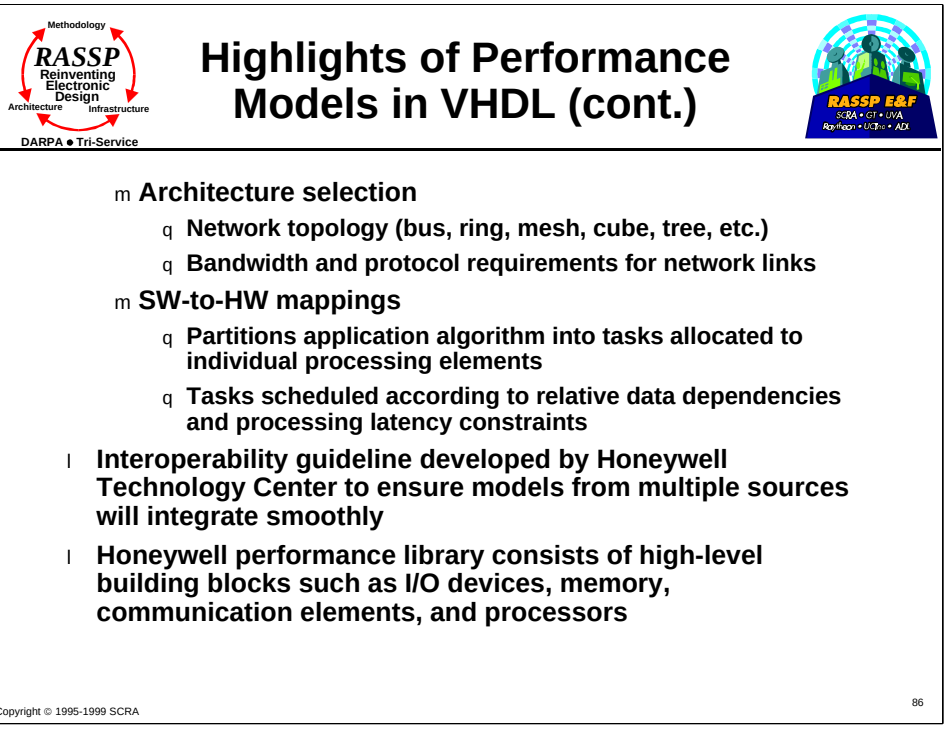
Latency represents the time it takes for a data item to get through the system after it has been introduced to the system.

The throughput indicates the amount of data the system can process in a given amount of time. Throughput and latency are inversely related because by increasing the latency (I.e. taking longer time to get each data item out of the system after it has been inserted into the system), one can have more data being processed internally during a given time frame and therefore potentially increase the overall throughput of the system.

The utilization indicates how much of the time a resource is being used.



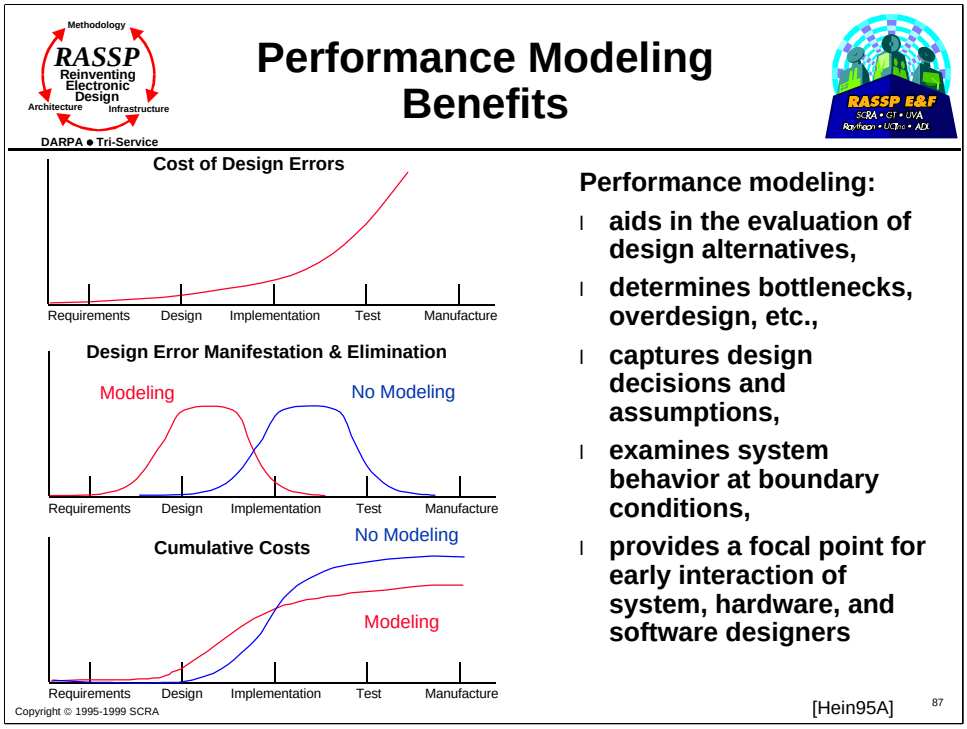
The number of simulation events is reduced because it models major events of the system (data passing) and not clock cycles events while maintaining sufficient accuracy at the system level. Its main functions are to determine the system size (number of processors, etc..), the network architecture (topology, etc.), and the software to hardware mappings.



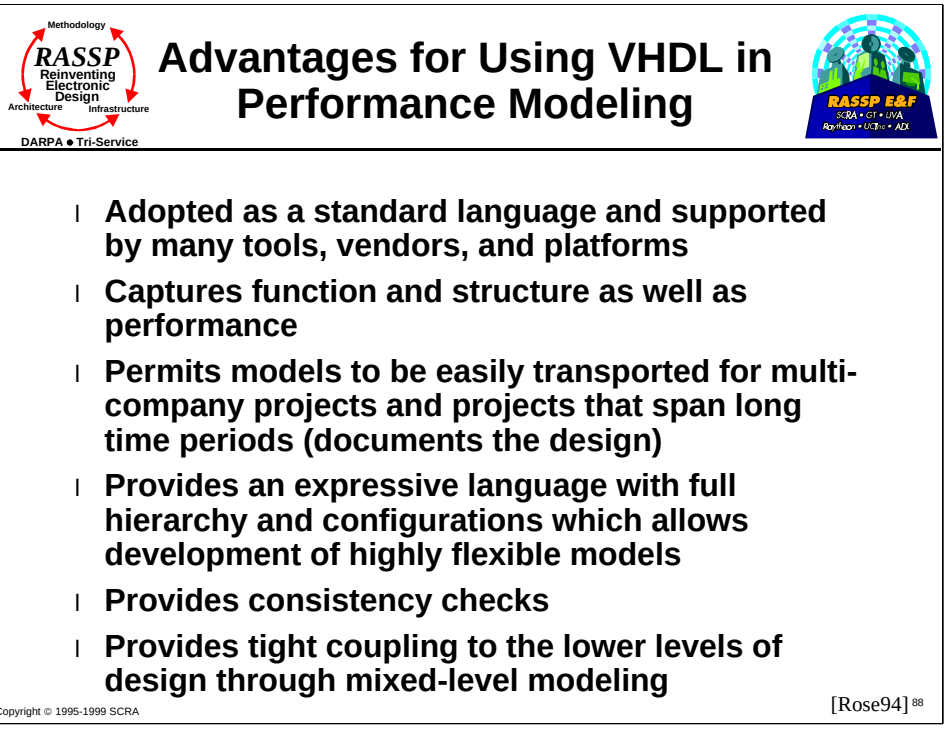
The SW-to-HW mappings represent the partitions of the application algorithm into tasks allocated to the processing elements.

Interoperability guidelines were developed by Honeywell Technology Center to ensure that all models developed on the RASSP program and subsequent programs can integrate smoothly. These guidelines can be found at http://rassp.aticorp.org. The equivalent at the lower levels of design is the IEEE Std 1164-1993 which specifies interoperability at the component interface level.

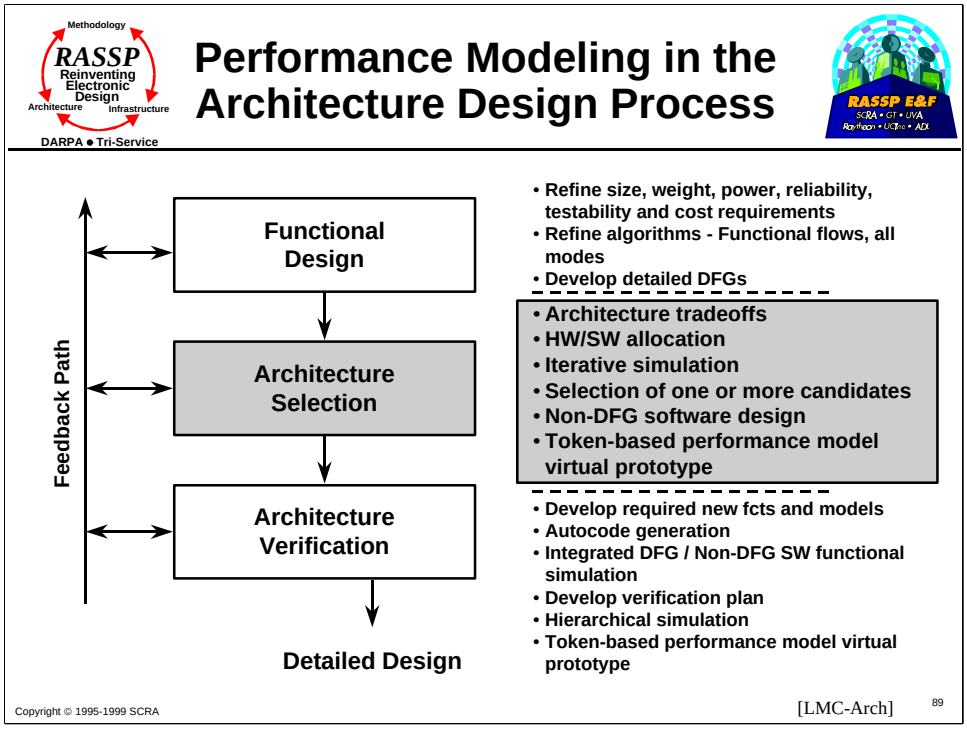
All developed models will be placed in the RASSP reuse library for rapid selection and reconfiguration of candidate system designs.



This slide shows some of the benefits of performance modeling as seen by some industrial users of the technique. Note that using performance modeling results in design errors being manifested and eliminated earlier in the design process where they are less costly. Also note that initially, the cost of a design process with performance modeling is higher, but the overall cost (area under the curve) is lower.



The above slide lists some of the advantages of using VHDL at the performance level.



The architecture definition process transforms processing requirements into a candidate architecture of hardware and software elements.

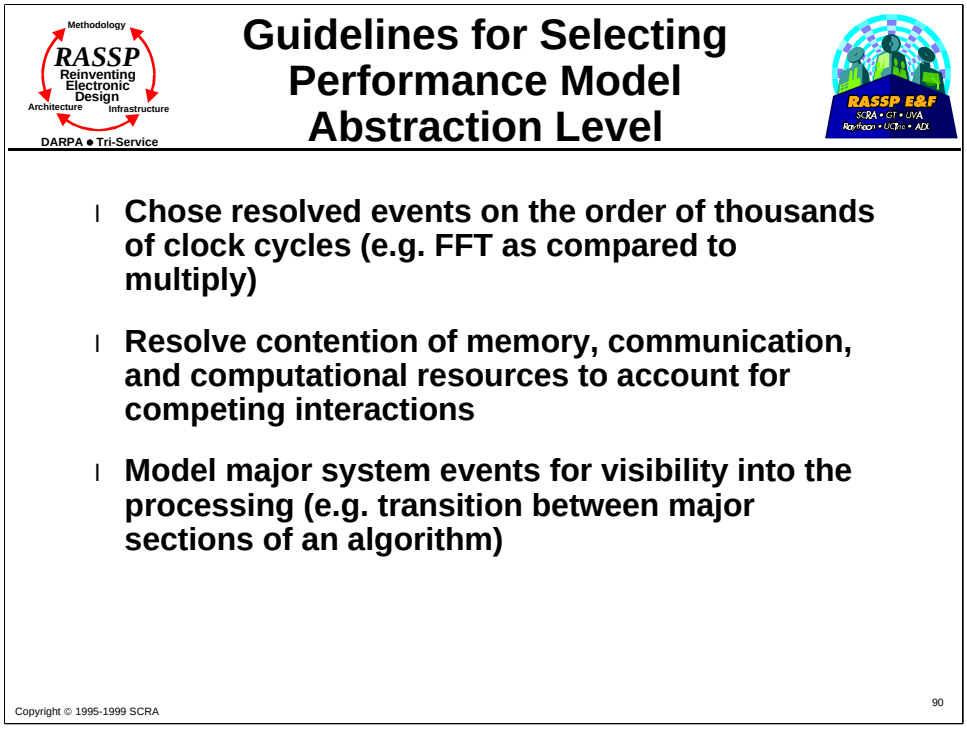
The architecture definition process is a new HW/SW codesign process in the RASSP methodology for high-level virtual prototyping and simulation. The primary concern in the architectural definition process is to select and verify an architecture for the signal processor that satisfies the requirements passed down from the systems definition process.

The overall task is to:

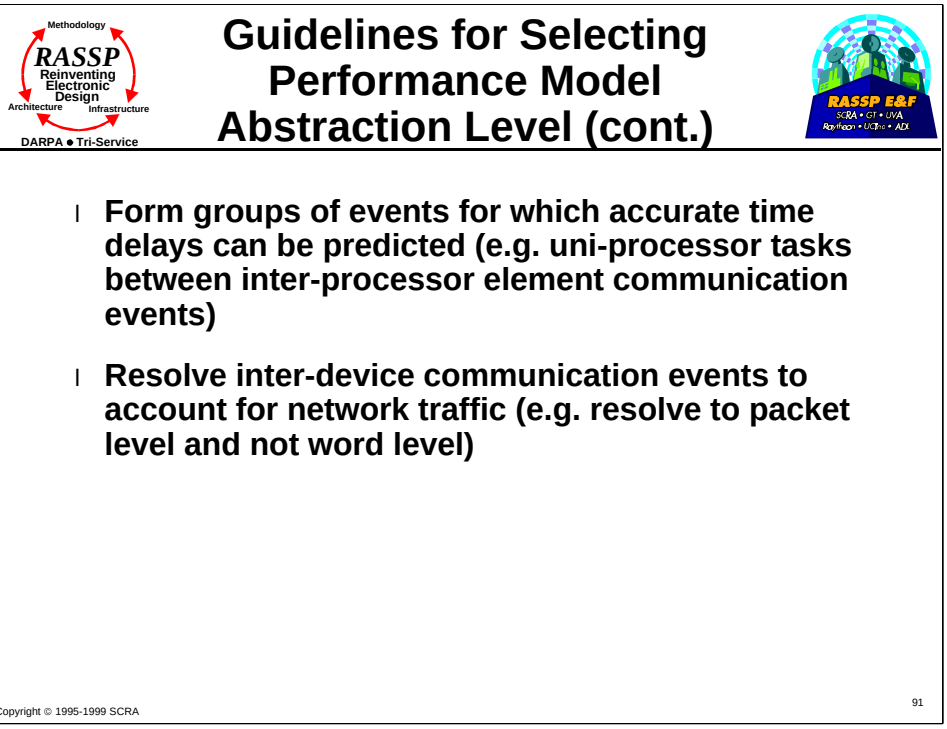
- Define and evaluate various architectures
- Select one or more for detailed evaluation that appear to meet the requirements
- Validate the chosen architecture(s) for both function and performance before detailed design

Concurrently, each selected architecture is evaluated with respect to size, power, weight, cost, schedule, testability, reliability etc.

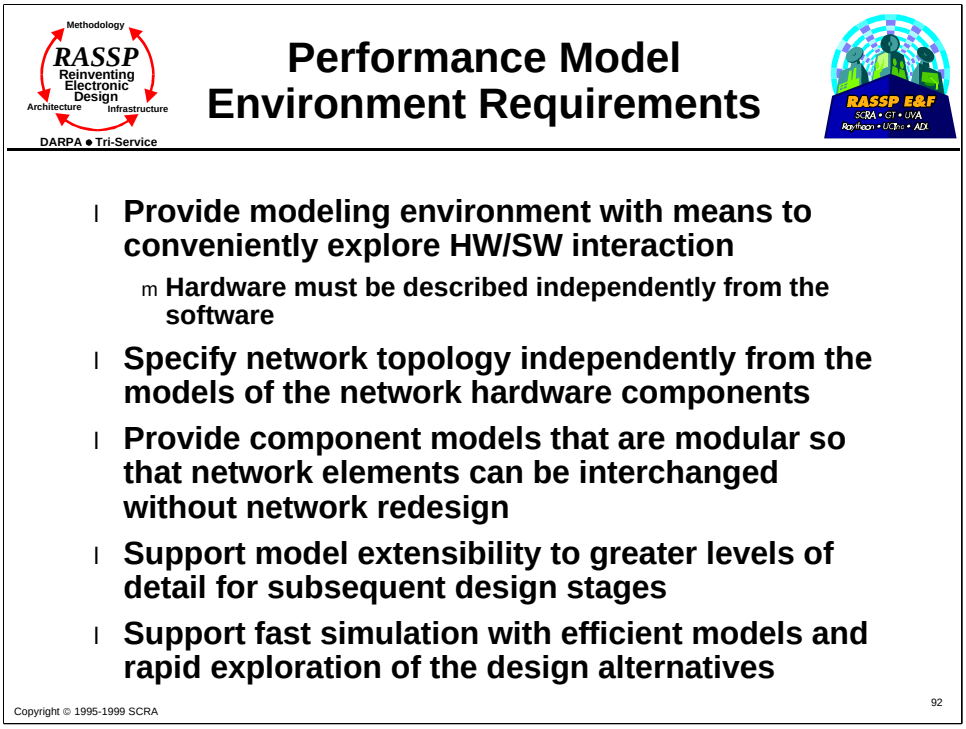
The process is library based and data flow graph (DFG) driven. The DFGs are created from the processing flows passed down from the systems definition process. Reuse of both architecture elements and software primitives significantly shortens the design cycle. Performance models are used to verify the timing aspects systems such as resource utilization, system sizing, latency modeling, and system data throughput. Software performance is also modeled for its impact on the total processing time.



The above list defines some guidelines for selecting performance models at the appropriate abstraction level. When using or developing performance models, the goal is to create an executable model that simulates much longer timelines than more detailed models so that it can analyze system sizing issues in detail, such as how many processors are required, what type of interconnect fabric must be used to meet the data throughput requirements, etc. All the functional details should not be present so that simulation speed can be improved. For example, a fast fourier transform algorithm performance model may only specify the time that it requires to compute the FFT and not the specific details of the algorithm. Major events are modeled in performance models rather than specific details. For example, when data is being transferred across a high speed network, a performance model would only capture the amount of time that it requires whereas a more detailed model will actually perform all the handshaking details of the signals defined for the network protocol.

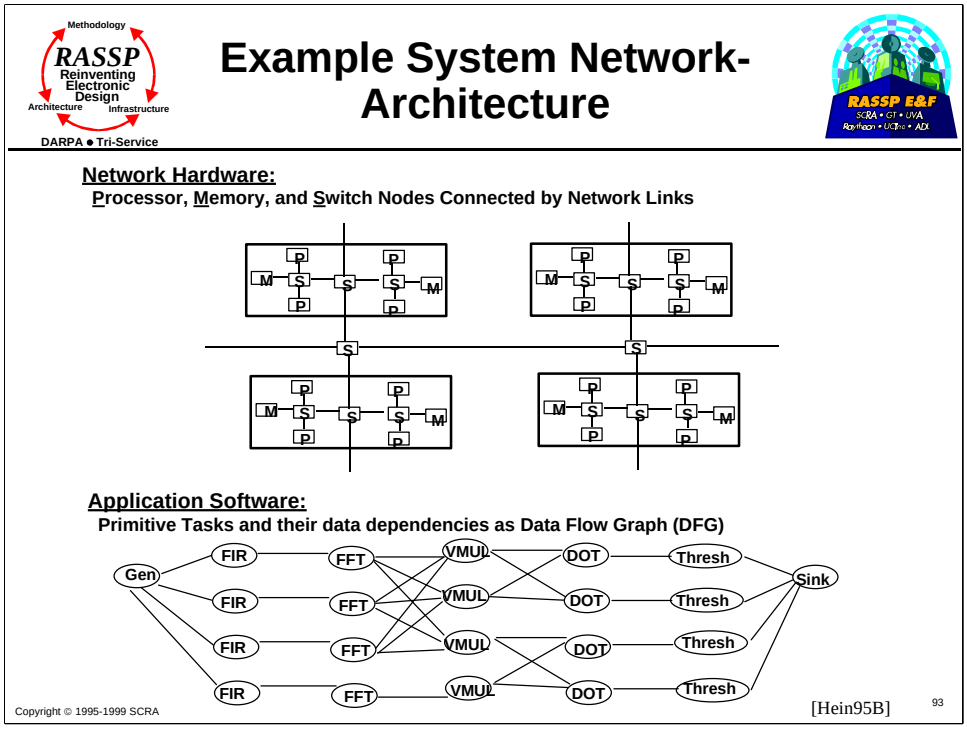


In performance models, it is good practice to combine groups of functions into a single group to reduce the total amount of events that occur during simulation. For example, if a number of tasks are being performed by the same processor, each taking a predetermined amount of time, then the combination of the events will also execute in a predictable amount of time. In this case, the events can be combined into a larger group of functionality that has a new time specified for the entire group.



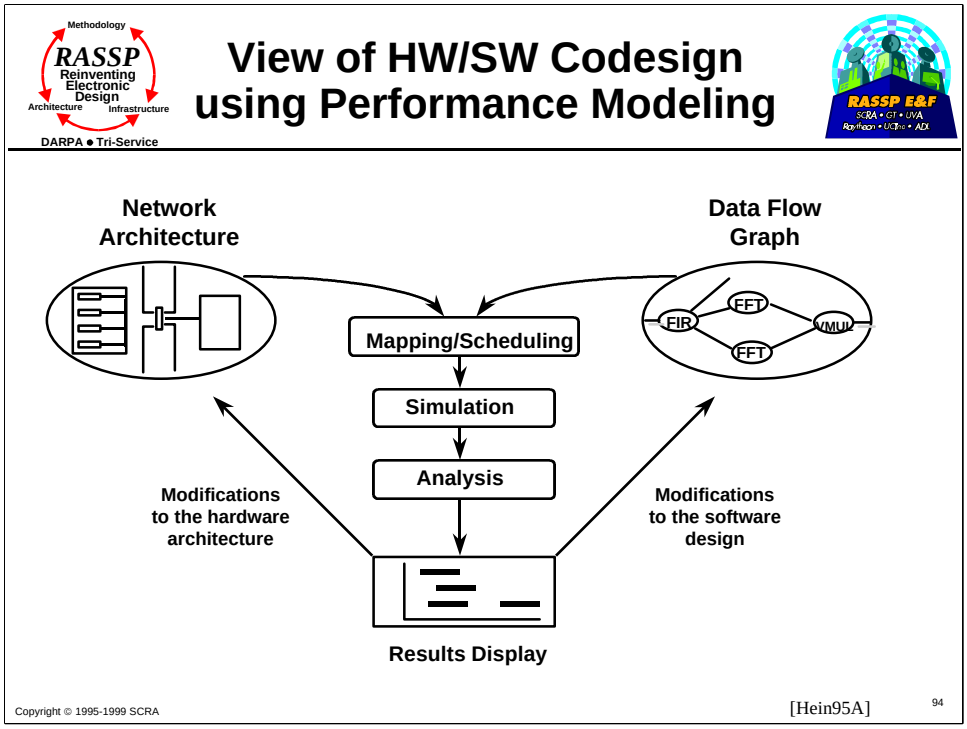
This slide presents requirements that should be imposed on performance models during their development. A performance model should model both hardware and software as well as the interaction between the two. It should be capable of modeling both the computational elements in the design as well as the network communications elements. Storage devices should also be modeled at very high levels so that total memory usage can be estimated. The developed models should be very flexible so that input parameters can be easily modified to create entirely different network architectures. The models should simulate very fast so that large systems can be modeled. This implies that a high level behavioral model is recommended. VHDL can be used to do this. The performance modeling module has examples of how to do performance modeling at the high level in VHDL.

Performance models should also permit the designer to do HW/SW codesign.

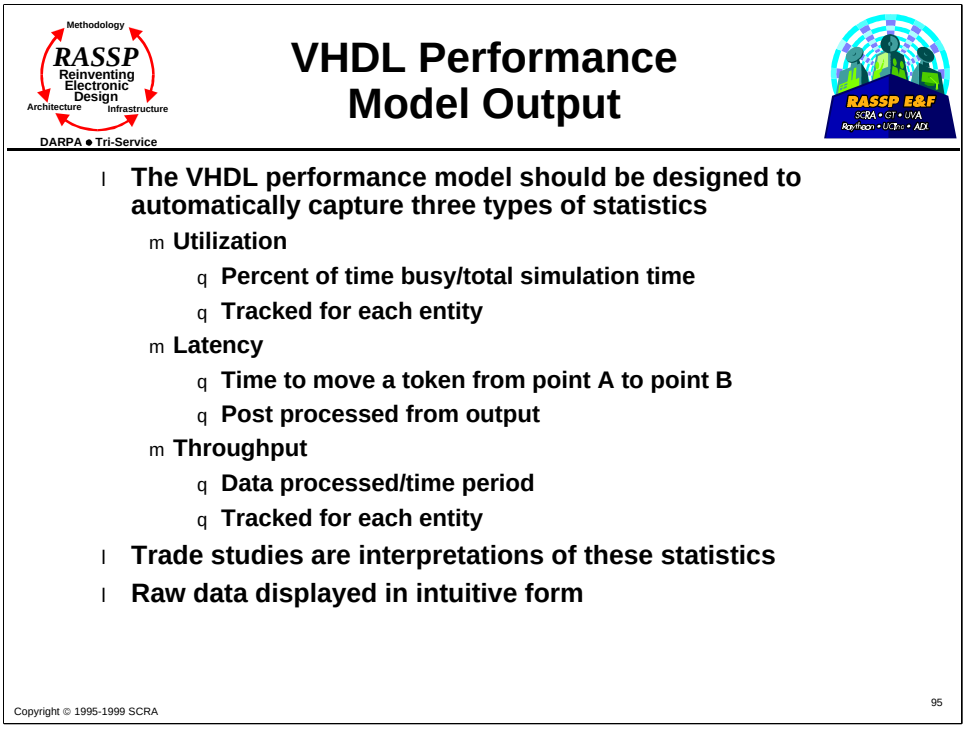


This is a typical network architecture along with application SW that must run on the HW.

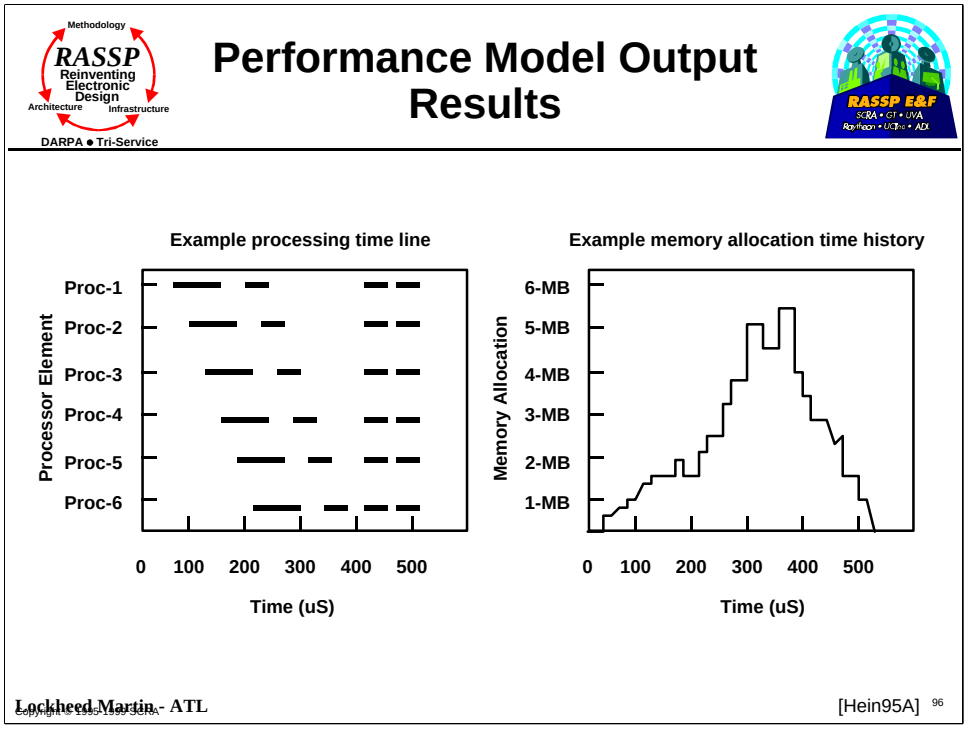
Processor-Memory-Switch type architecture is very general and can model many designs.



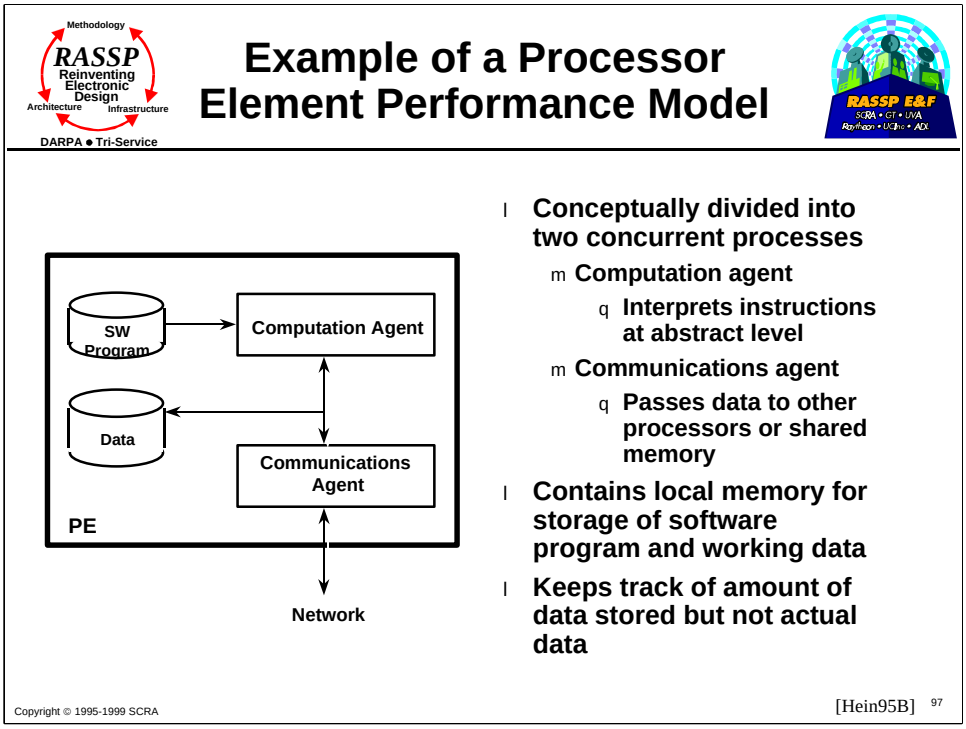
This figure shows the interactive development and optimization of a hardware and software design within an iterative mapping/scheduling process. The hardware/software codesign process begins with a model of the hardware onto which the software application tasks are mapped and scheduled. The joint HW/SW system is simulated, and modifications are made to either the hardware, software, or both in response to the analysis of the simulation results.



VHDL performance models should be developed so that they automatically capture timing-related parameters of the system. These include resource utilization, throughput, and latency information for the system model. Trade studies can be done using these statistics to determine the optimal architecture for the system application. The raw data from the simulations can be displayed in an intuitive form as shown on the next slide using graphical software.



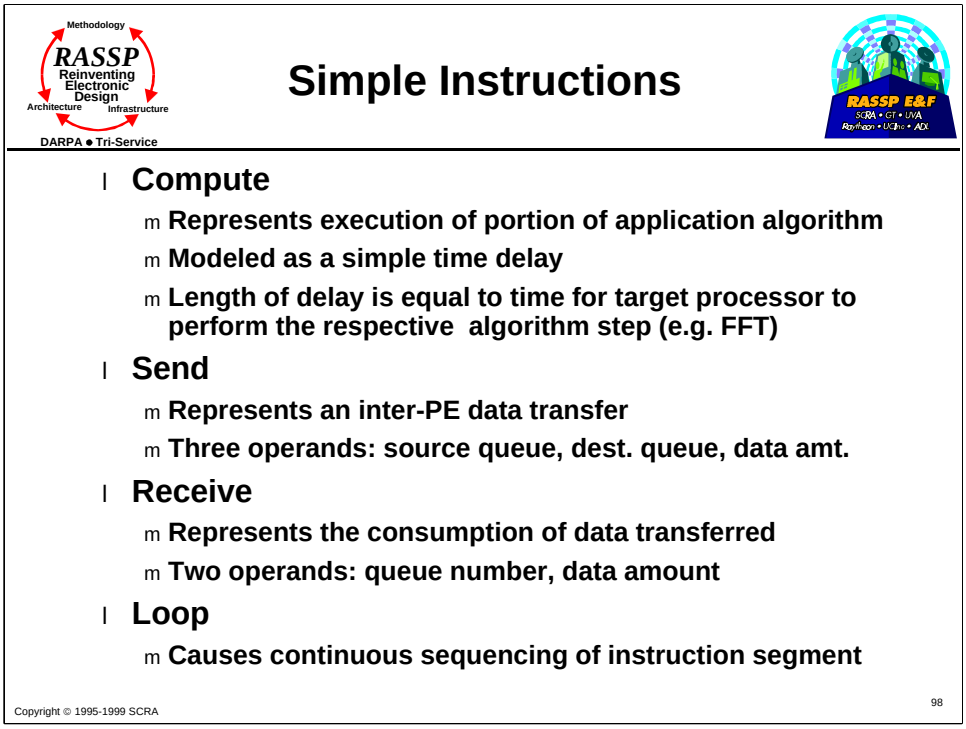
These are some example plots of the output of performance model simulations. The left side plot shows processor utilization vs. time and processing element. The right plot shows how the memory is utilized during the same time line.



The processor element for a MIMD system is conceptually divided into two concurrent processes, the computational agent and the communications agent. The computational agent reads and interprets abstract instruction types from a file. The communications agent sends and receives tokens to other elements in the system through the use of tokens. The processor also contains local memory to store software programs and working data. The performance model does not store actual data but keeps track of the amount of data that would have been stored.

The computation agent represents the hardware side of the interface between the hardware and software because it interprets the SW application program instructions into specific HW actions. The computation agent executes a partitioned flow graph. A simple example could use abstract instruction types such as compute, send, receive, and loop, as its four main instruction types.

The communications agent handles the reliable transfer of data between the other PEs and the local PEs memory queues. It implements whatever link layer protocols, packetization, and retry or blocked message resumption that are needed to transfer and receive arbitrary length data messages over the network. Upon reception of data, the agent increments the data amount of the destination queue by the received amount. If the agent was blocked waiting for the received data, the agent would allow the communications agent to resume. On sending data, the agent decrements the data amount of the local source queue by the transmitted amount.

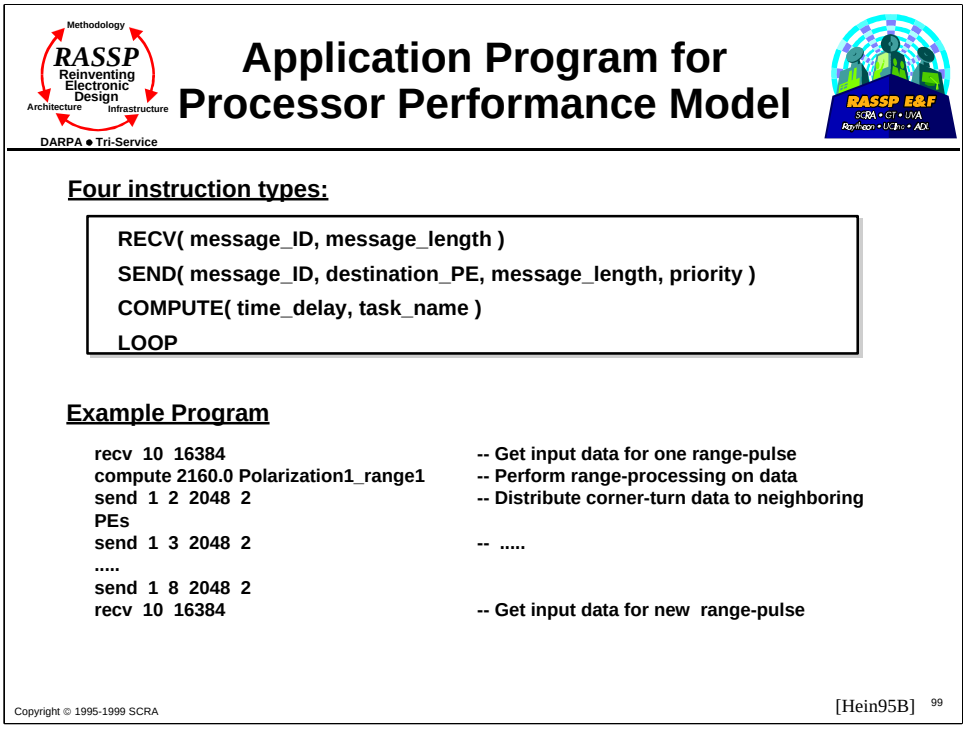


The *compute* instruction represents the execution of a portion of the application algorithm within the PEs local memory. It is modeled as a simple time delay. The compute instruction contains one operand specifying an algorithm step or corresponding computation time. The length of the time delay is equal to the time required for the target PE to perform the respective algorithm step (e.g. FFT, FIR, etc.). Upon completion of a computation delay interval, the computation agent interprets the next sequential instruction.

The *send* instruction represents an inter-PE data transfer. It contains three operands: the local and destination queue numbers, and the data amount to send. Other operands, such as priority, may be modeled. When the agent encounters a send instruction, it directs the local communication agent to transfer data from a local memory queue to a queue in another PE. If the communication agent can accept the command immediately, the computation agent continues sequencing through instructions. If not, the computation agent is blocked until communication completes. No data is actually transferred, but the effects of the transfer are recorded.

The *receive* instruction represents the consumption of transferred data. It has two operands: queue number and data amount. If the sufficient amount of data had arrived in the specified queue prior to encountering a receive command, then the computation agent decrements the specified queue by the specified receive amount and then continues sequencing instructions.

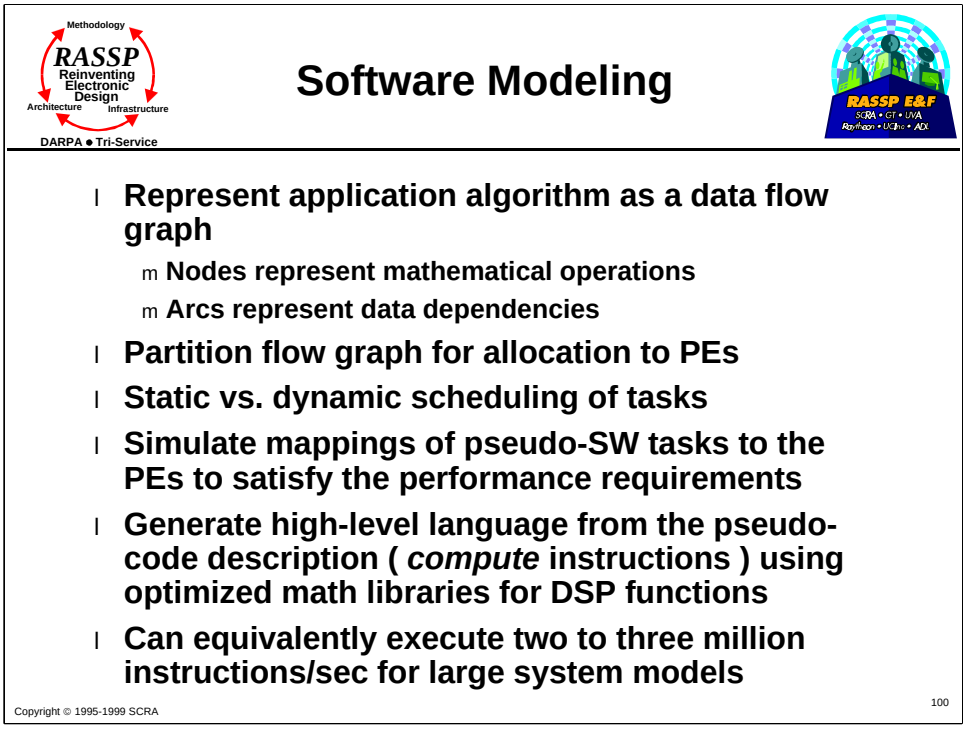
The *loop* instruction causes the computation agent to continue sequencing form a non-sequential instruction of the application program.



The example program is analogous to the ISA level instructions found in more detailed behavioral models (presented later).

Instead of 100-200 instructions, we now only have 4.

Instead of clock cycle resolution, we have resolution at the major event level.



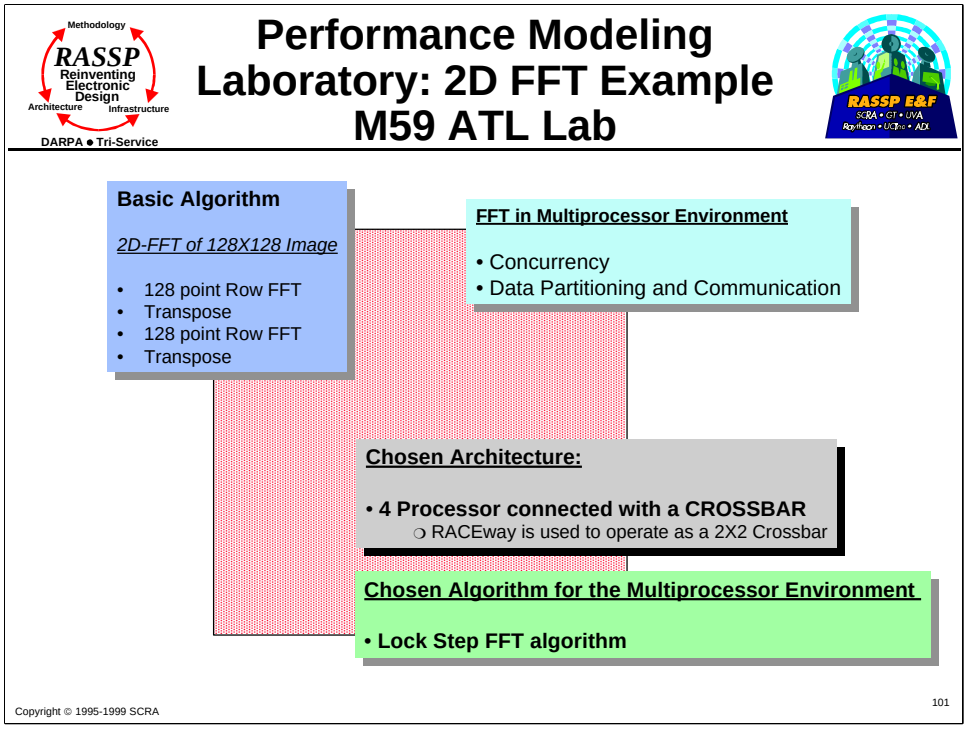
The application algorithm is first represented as a data flow graph (DFG). The DFG is a directed graph where the graph nodes represent mathematical operations, and the arcs represent data dependencies and form the logical data queues. The DFG nodes usually correspond to DSP primitives, such as FFT, vector multiply, convolve, etc.

For a given network architecture, the flow graph is partitioned for allocation to PEs in the network. The flow graph nodes may be allocated statically at design-time or dynamically at run-time. Dynamic mapping/scheduling requires modeling the dynamic mapper/scheduler. Static scheduling requires this to be done prior to simulation.

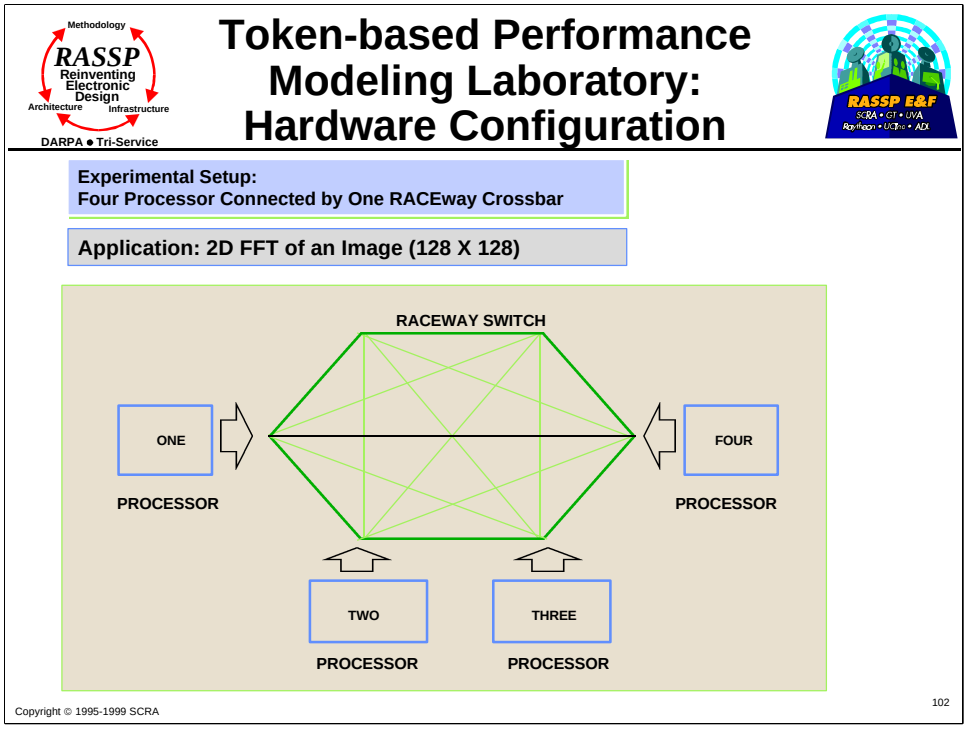
The result of a static partitioning/mapping/scheduling process is a set of pseudo-code software application programs for each of the PEs. The scheduling determines the order the tasks should be executed. The actual time the task is executed is determined by the task sequence and the inherent data flow control of the *send/receive* paradigm.

Once the simulations show a suitable software mapping and HW architecture combination to satisfy the system requirements, the pseudo-code (*compute* instructions) software routines are expanded into high-level language subroutine calls which are compiled for down-loading to the target HW.

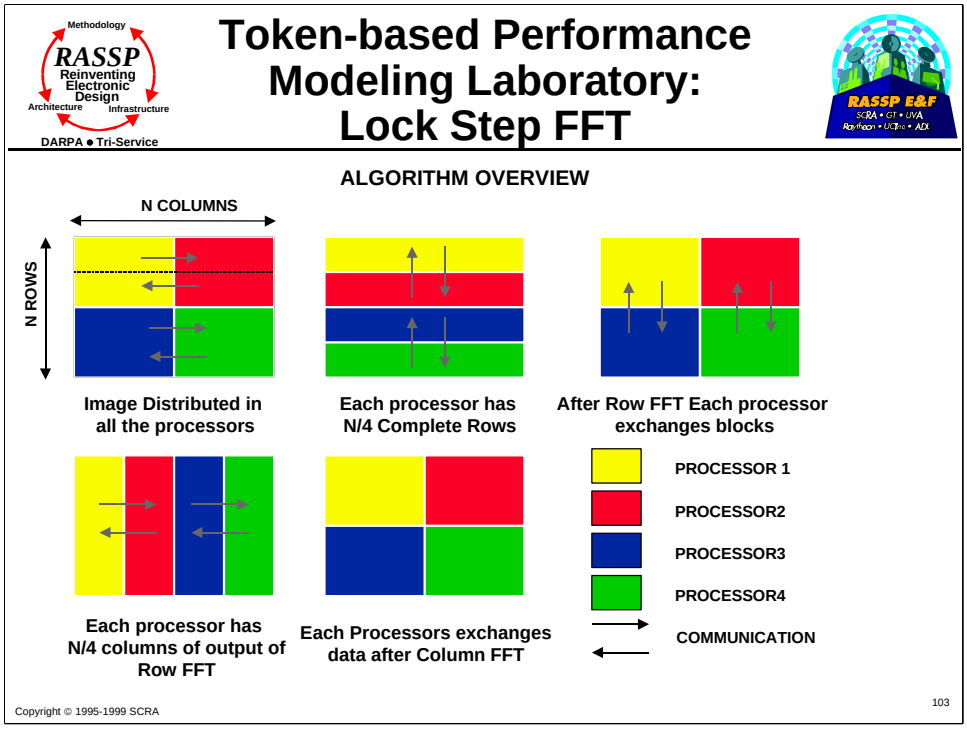
Using these techniques for simulating large systems, a 24-PE system executing 5 seconds of SAR application code executed at an effective rate of 2.8 million instructions/sec. [Hein95]



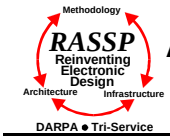
This laboratory uses the processor model (with both computational and communications agents) explained in the previous slides to construct a 4 processor network linked with a Mercury RACEway interconnect. The lab is contained on the CD-ROM in the M59 module, “Token-based Performance Modeling using VHDL.” The lab write-up is also entitled “Token-based Performance Modeling using VHDL” and illustrates the usage of this level of model in describing an application running on a high-level model of a system.The basic algorithm to be executed is a 2D FFT of a 128x128 pixel image. The basic algorithm first takes row FFTs, then does a transpose of the result, then takes another set of row FFTs (equivalent to column FFT without transpose), then transposes the data again to get it back to its original ordering. A lock step algorithm is used and is described in a later slide.



This is a diagram of the 4 processor network interconnected by the RACEway switch. This is the hardware configuration of the M59 ATL Lab VHDL performance modeling lab associated with module 59 and is also contained on the CD-ROM.

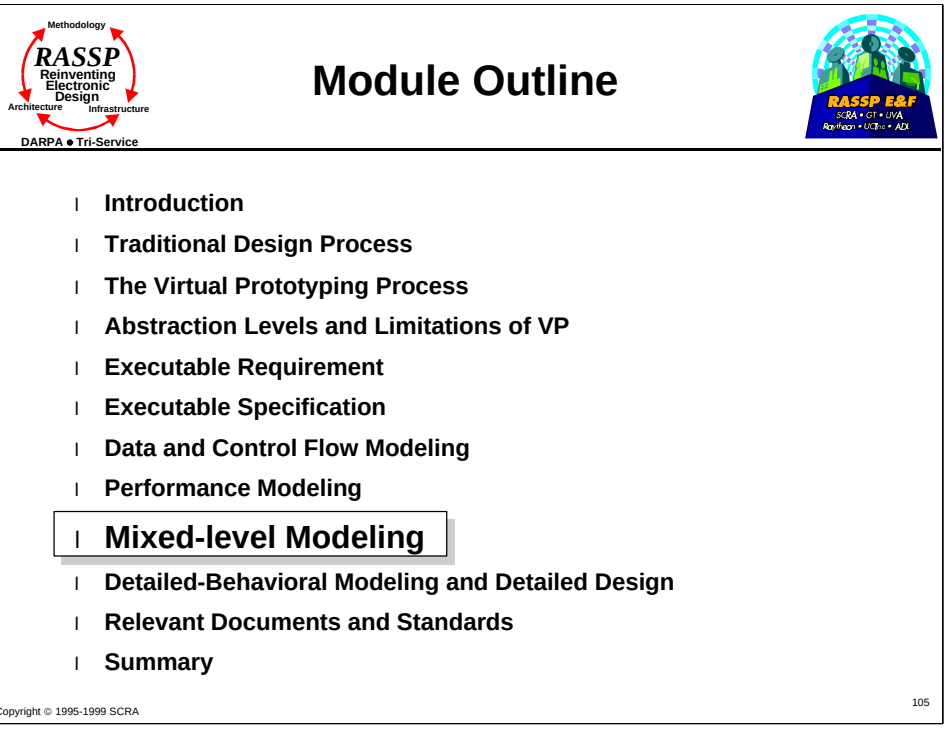


This diagram illustrates the lock-step algorithm that will be executed on the 4 processor hardware performance model used in the M59 laboratory exercise. More details about the algorithm are presented in the lab write-up. The 2-D algorithm is done by performing FFTs on specific segments of the image data and then passing the transformed data to its neighbor processors to do additional FFTs. The image data is first distributed in the memory of all four processors. Each processor then performs row FFTs on their respective image data prior to sending the transformed data to their neighbor processor. After sending the data to the neighbor processor, a final column FFT is performed by each processor on the new data. Data is again exchanged after the final FFT to put it in the correct locations for the image to be viewed.

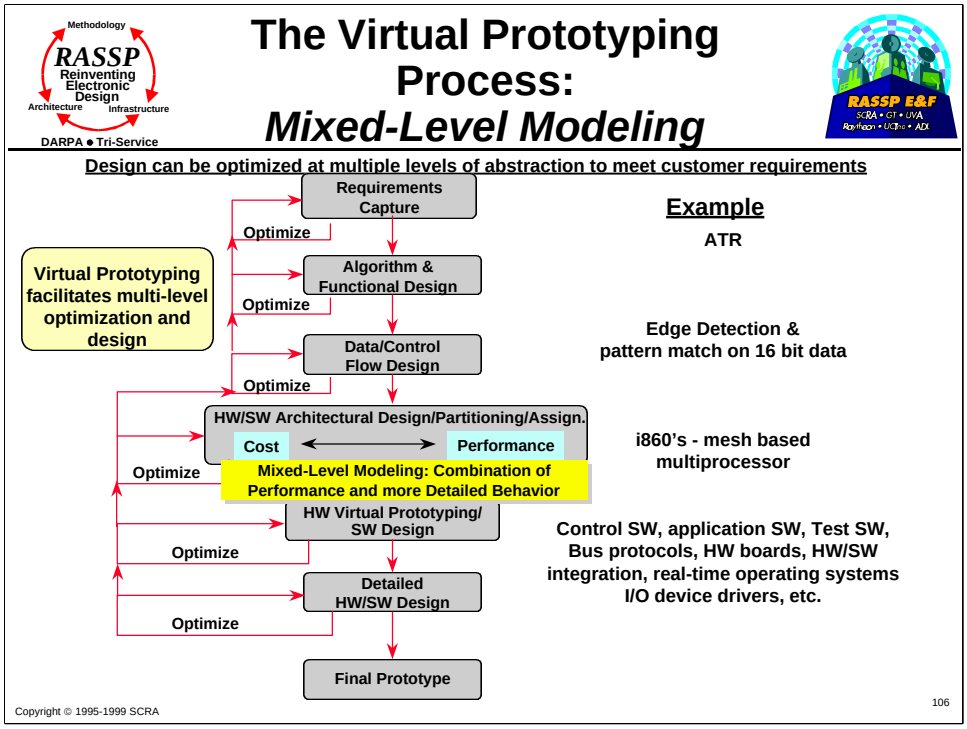


Additional CD-ROM Material on Performance Modeling

- **Module 59 provides a detailed account of performance modeling**
  - **This module can also be found on the CD-ROM**
- **Application note on performance modeling can also be found on the CD-ROM**
  - **See the section on Application notes to obtain this information**
  - **"RASSP Token-based Performance Modeling" application note**
  - **"RASSP Virtual Prototyping" application note**
  - **"Hardware/Software Codesign" application note**

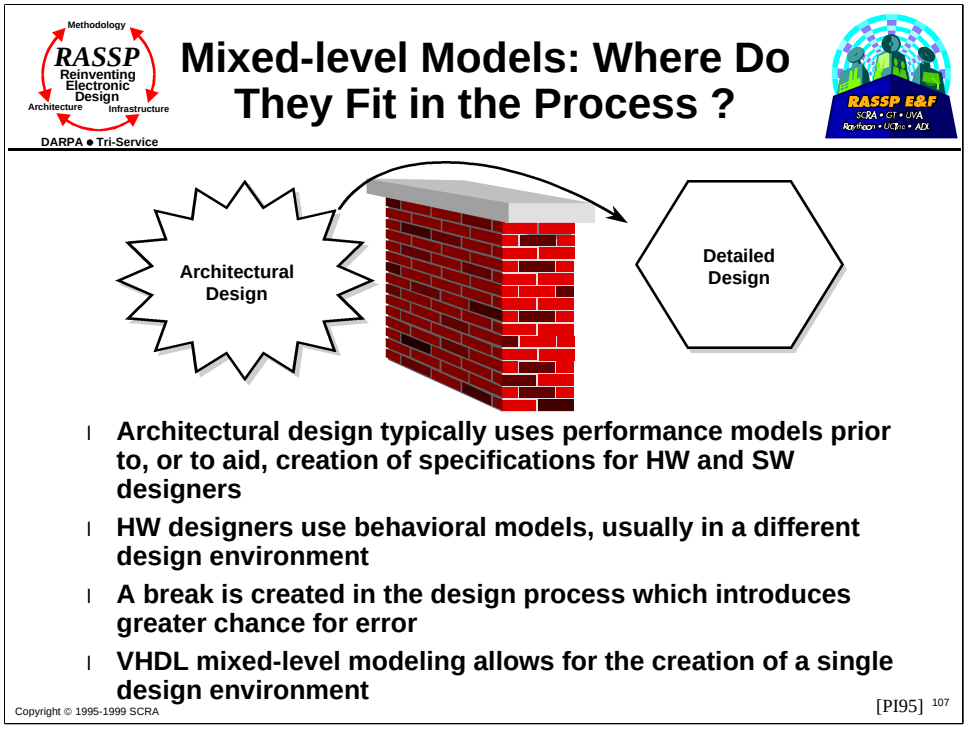


Hybrid modeling attempts to bridge the gap between performance modeling and abstract behavioral modeling. This section will give some of the preliminary results from the RASSP effort in this area.

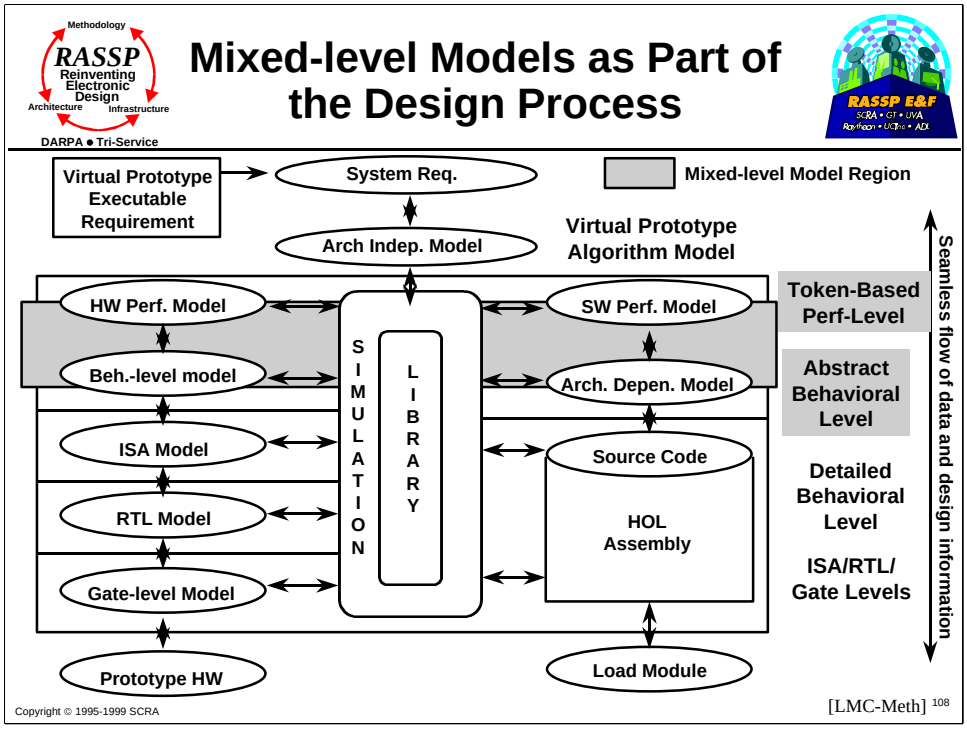


The VP process is a top-down design paradigm with optimization done at multiple levels of abstraction. There are a number of slides in this module that capture the essential flow of this design process but in a different format. In general, the first stage in any design process is the correct definition of the requirements for the design. Once the customer's requirements have been understood, then the system design team begins working on design specifications that attempt to satisfy all the customers requirements. In the above process flow, this represents the algorithm definition and functional design stage. At this stage, the algorithms used to implements the functions are refined. The size, weight, area, and power constraints may also be refined as well. At this point, initial computational complexity of the algorithms, as well as fixed-point characteristic, are analyzed. As the process proceeds from the top level, further refinement is done. At the data flow level, the algorithms are refined and data flow graphs that implement the functionality are explored. These data flow graphs are used in the HW/SW architectural design phase, where the nodes in the flow graph are either mapped to hardware or software. At this level in the process, the hardware and/or software are sized to meet the requirements. Timing critical information is captured such as latency, throughput, and resource utilization. As more detailed design is done, the architectural elements are either developed as software modules or hardware components (ASICs, Processors, etc.) At the end of the process, final hardware and software are realized. At each stage in the process, the design is verified and optimized to meet the customer's requirements.

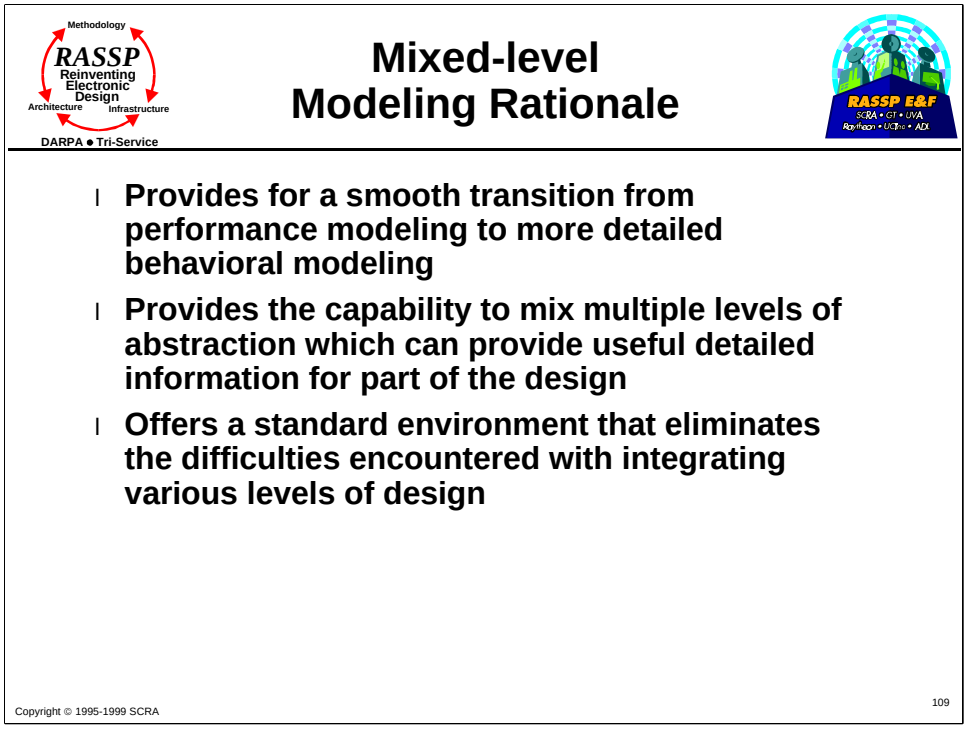
This slide illustrates the focus of the current section: Mixed-level Modeling



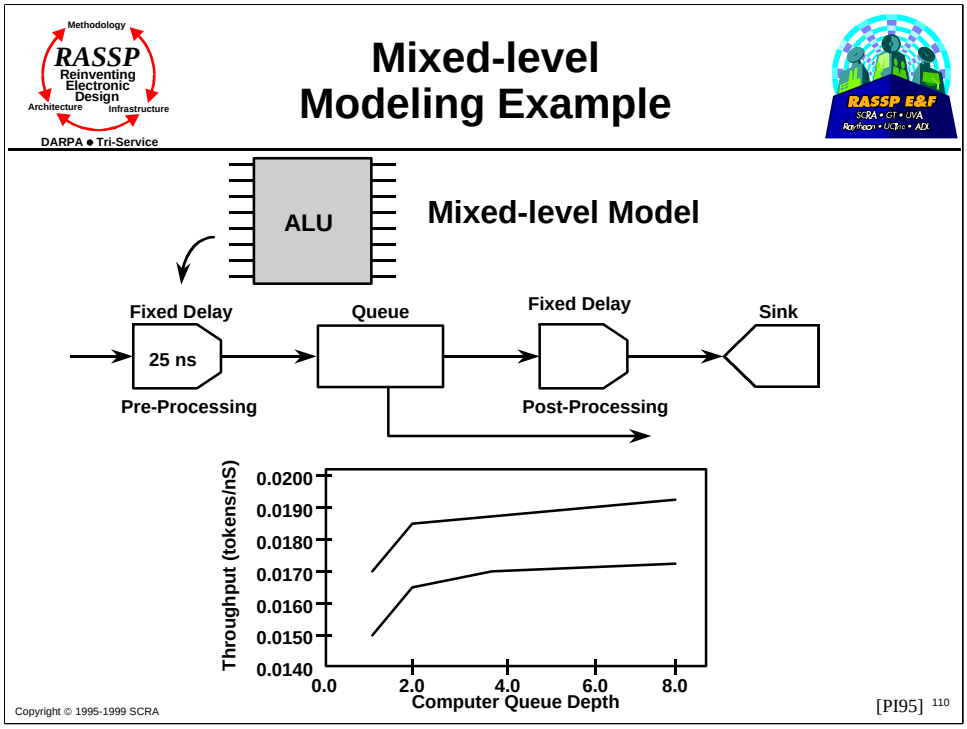
The primary problem VHDL hybrid modeling addresses is the break in the design cycle from architectural design tools to hardware design tools. These two arenas are typically done with different tools, which then means the hardware design must be re-entered. With the advent of VHDL performance modeling, a step was made in the direction of solving this problem. However, a robust hybrid library and methodology is required to successfully complete this solution.



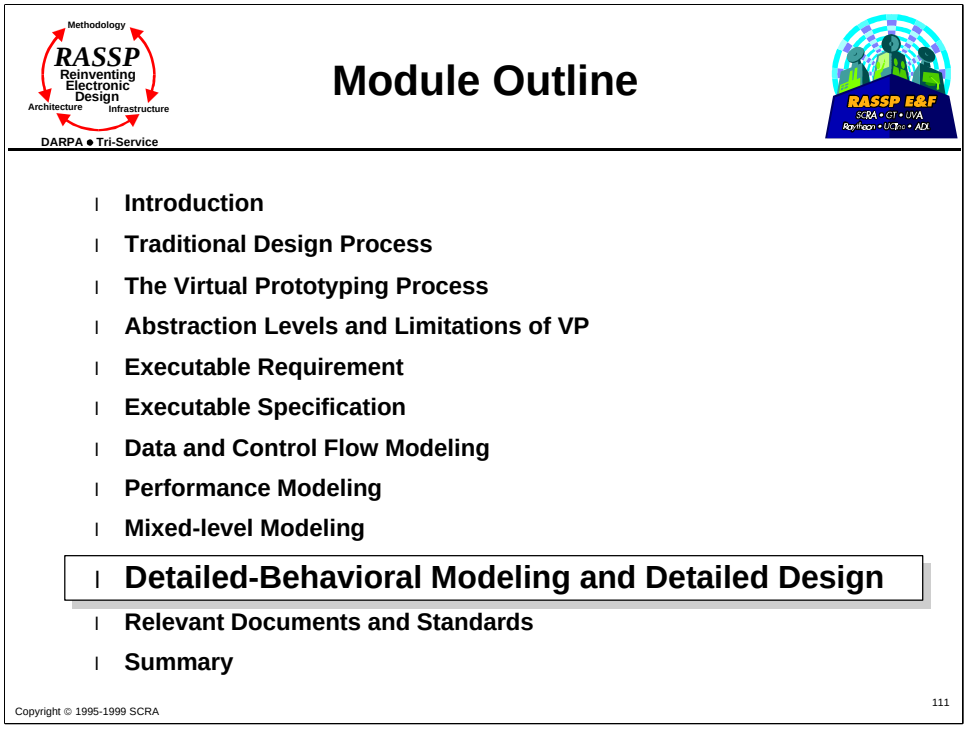
Hybrid models attempt to bridge the gap between performance modeling and more abstract behavioral modeling. The region falls between the virtual prototype using token-based performance models and abstract behavioral models. Hybrid modeling supports a smooth integration of the performance and functional models.



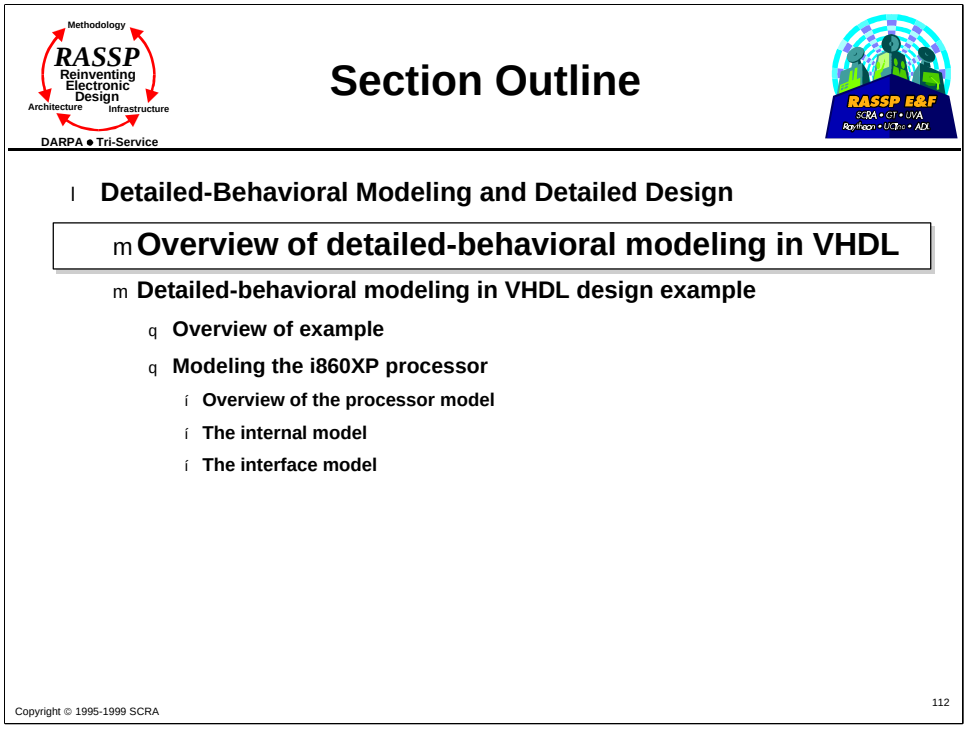
The rationale for developing a hybrid modeling methodology on the RASSP program is described in the above bullets. One of the main reasons for mixed-level modeling is that it provides a smooth transition from token-based performance modeling to abstract behavioral modeling. Token-based performance models are typically used for architectural trade-offs while models with function and timing (behavioral level) are typically used in the first step of the HW design. To seamlessly interconnect the design process, hybrid models fill the gap between the performance and abstract behavioral models. The combination can be simulated in a standard environment using VHDL where the models are represented at various abstraction levels.



This slide illustrates the impact of hybrid modeling on a performance model. In this example, a fixed delay node was replaced with an abstract behavioral model of the ALU. The statistically significant inputs of the ALU in terms of delay were identified. Next, a search of this set of inputs was made to find the input pattern that generated maximum propagation time through the circuit. The shift in the throughput curve shows that the more accurate delay model from the behavior element changed the absolute value of the results but not the trends.



We will now discuss the final stage in the virtual prototyping process where the detailed behavioral modeling and detailed design is done. Candidate approaches for verifying the design are presented as well as an example of a case study done on RASSP for an Infrared Search and Track (IRST) signal processing system. The detailed behavioral model of the IRST will be discussed and the limitations and benefits of the use of such a modeling paradigm will be discussed.

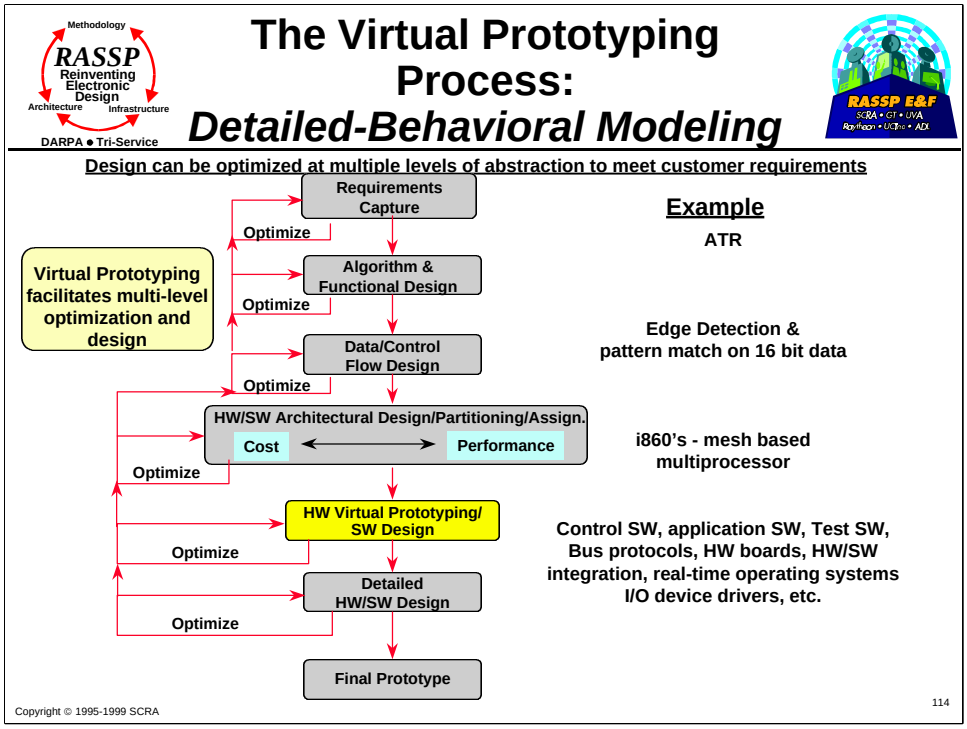


This section starts with an overview of what detailed behavioral modeling addresses in the VP process. After this introduction, alternate approaches to solving this problem are presented along with their advantages and disadvantages. At the end, design examples are presented that illustrate some guidelines in capturing the necessary information at this stage in the process.

Section Outline (cont.)

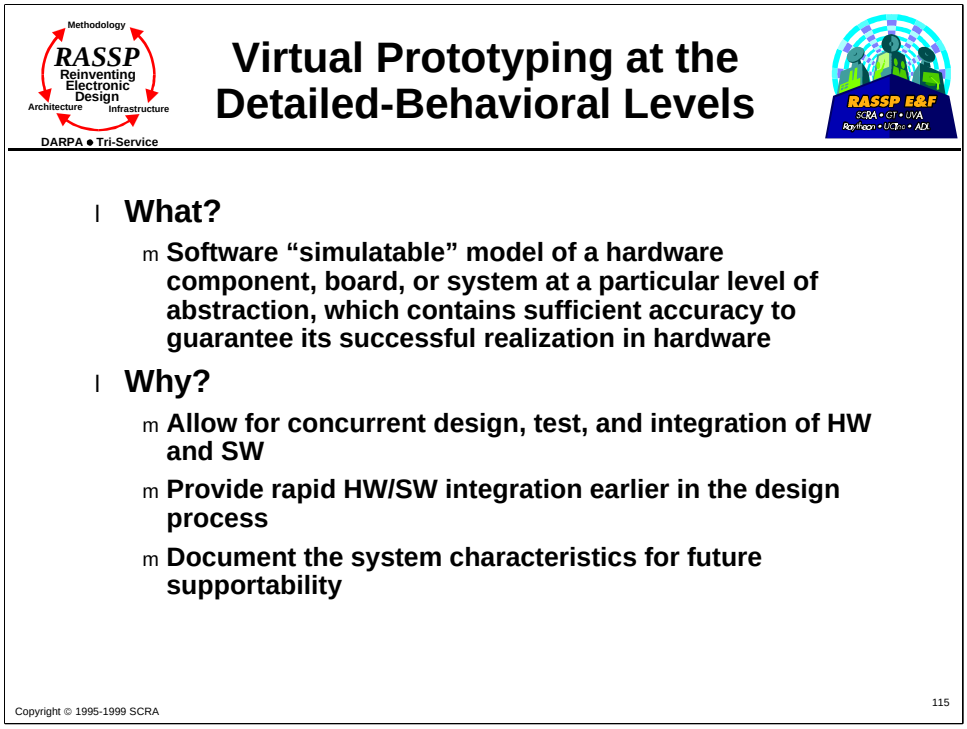
- The testbench
  - Clock/reset generator
  - Memory controller
  - Memory
- Testing the i860XP
- Results

- **Testing the MCV9**
- **Testing the IRST**
- **Creating a DMA to the VME in the memory controller**
- **Simulation results**



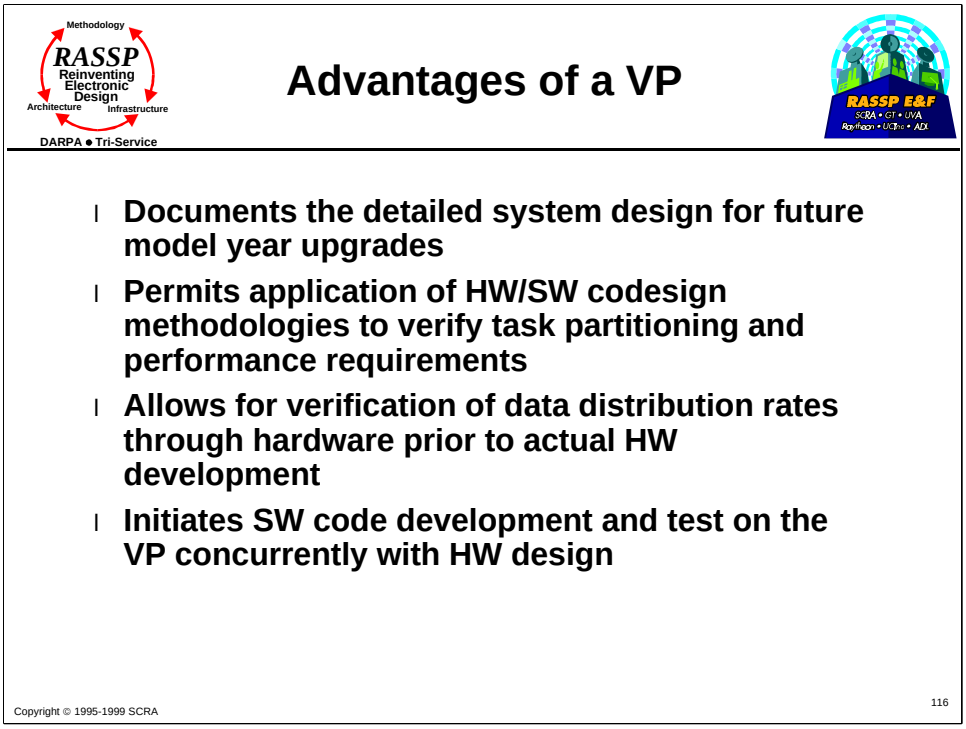
The VP process is a top-down design paradigm with optimization done at multiple levels of abstraction. A number of slides in this module capture the essential flow of this design process in a different format. In general, the first stage in any design process is the correct definition of the requirements for the design. Once the customer's requirements have been understood, then the system design team begins working on design specifications that attempt to satisfy all the customer's requirements. The above process flow represents the algorithm definition and functional design stage. At this stage, the algorithms used to implement the functions are refined. The size, weight, area, and power constraints may also be refined as well. At this point, initial computational complexity of the algorithms, as well as fixed-point characteristics are analyzed. As the process proceeds from the top level, further refinement is done. At the data flow level, the algorithms are refined and data flow graphs that implement the functionality are explored. These data flow graphs are used in the HW/SW architectural design phase, where the nodes in the flow graph are either mapped to hardware or software. At this level in the process, the hardware and/or software are sized to meet the requirements. Timing critical information is captured, such as latency, throughput, and resource utilization. As more detailed design is done, the architectural elements are either developed as software modules or hardware components (ASICs, Processors, etc.) At the end of the process, final hardware and software are realized. At each stage in the process, the design is verified and optimized to meet the customer's requirements.

This slide illustrates the focus of the current section: Detailed-behavioral Modeling



This presentation will cover the areas of virtual prototyping dealing with lower levels of the process where the simulatable models are used to help validate a design to the level where a high degree of confidence is assured before prototyping the actual HW. This detailed behavioral model can serve as an executable specification for more detailed models at the RTL and gate levels.

There are two main reasons this level of model is used: first, the ability to codesign HW/SW prior to the creation of actual HW to remove some functional and timing errors prior to commitment of actual hardware and second, it serves as a model that approximates the actual system so that initial verification and system simulations (interactions with other hardware and software) can be done earlier in the process. It also serves as a form of documentation for the system.



Given a documented model of the system with associated testbenches, modifications to the system can be made by simply replacing the changed portion of the model with its upgrade.

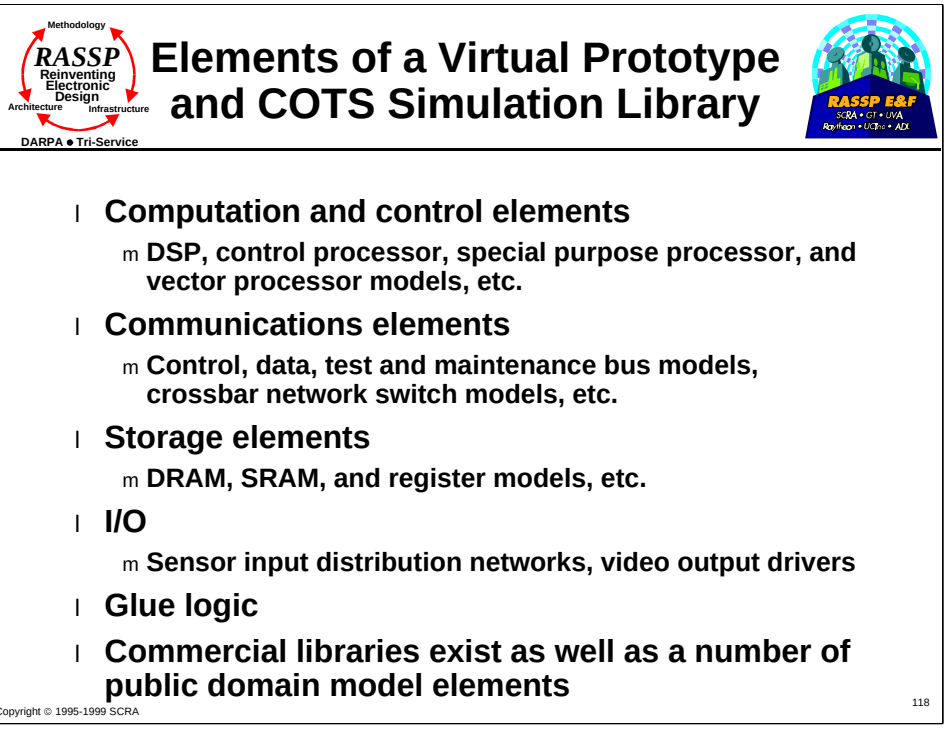
The model is then executed with the previous high-level testbenches for validation of system correctness.

HW/SW codevelopment can take place on the detailed behavioral virtual prototype. There is still a need for an environment where SW designers are able to work effectively on the virtual prototype. This will require an environment that contains the symbolic debugging information typically found in a SW engineering environment.

Early validation can help remove design flaws that would typically only be found in the HW/SW integration stage of current practice.

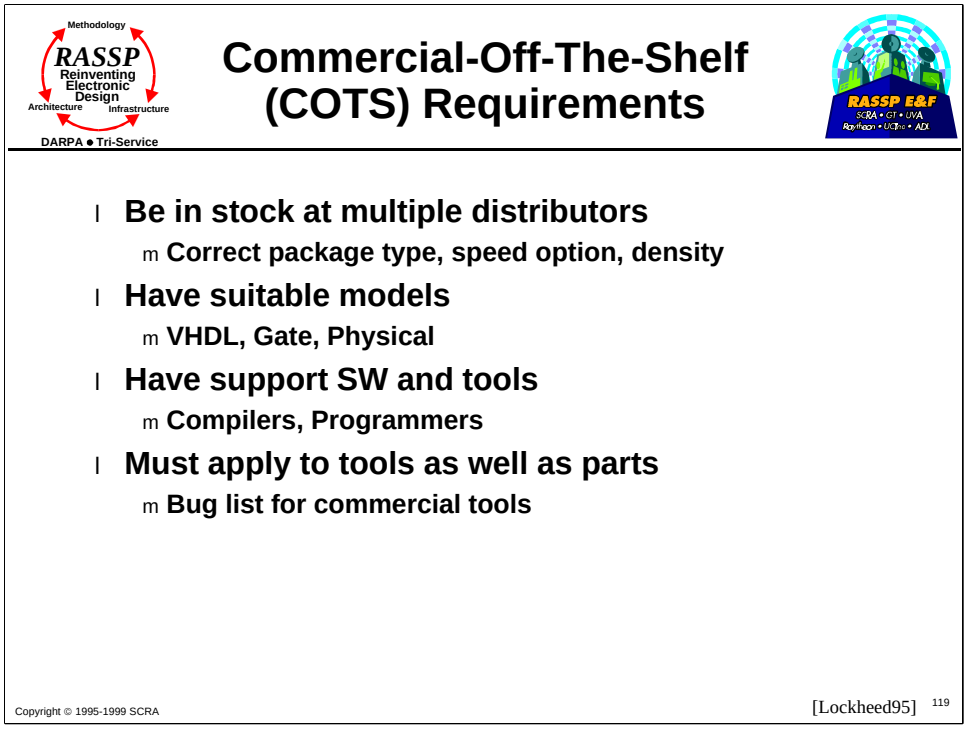
Advantages of a VP (cont.)

- **Tests portions of SW that interface to HW to discover possible design flaws prior to HW commitment**
  - **More observability into the system**
  - **Unlimited set of test probes (Virtual Pins)**
- **Facilitates HW/SW integration and test**



The types of components found in a detailed behavioral virtual prototyping environment include the above commercial-off-the-shelf (COTS) elements.

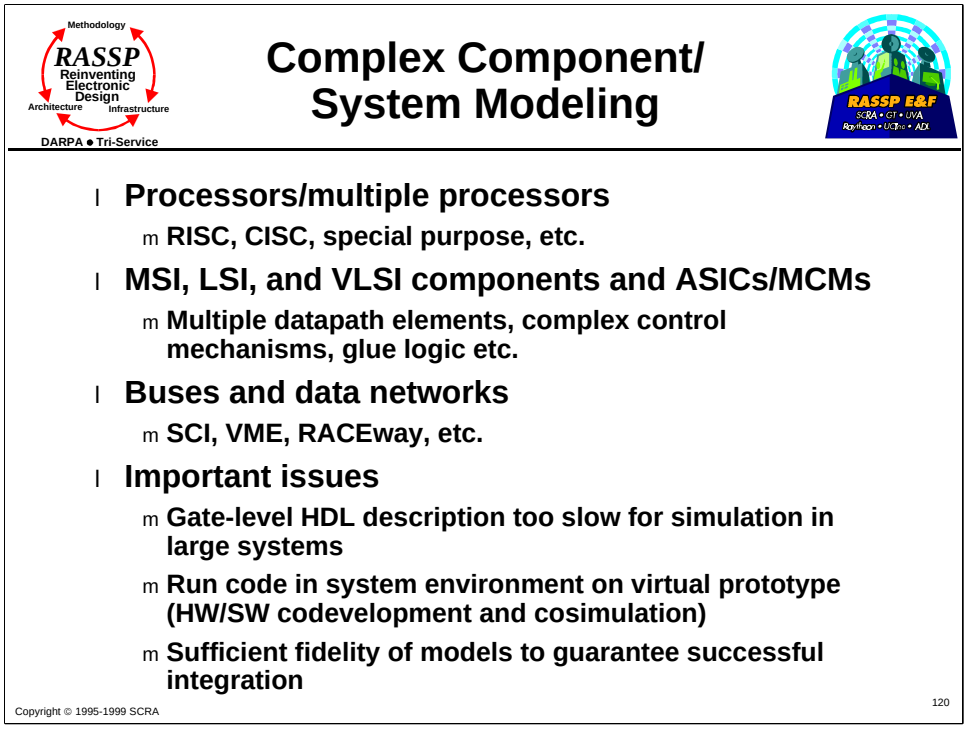
- Ideally, these would be found in a library and not require in-loop creation.
- Glue logic may always need to be designed for communication between elements such as processors and buses. This can take the form of PLDs, PLAs etc.
- Libraries have been developed on tech base contracts to support processors for DSP (GT), bus models (GT), FPGAs (MSU), memory elements (MSU), and lower-level models (Multipliers, adders, etc.) for building special purpose processors (OSU).



Because COTS is a significant theme in the RASSP process, a definition of what it means to be a COTS element and which types of elements this term encompasses is needed. Traditionally, it has been applied to the components themselves, but there is also a need to recognize the importance of some other related factors, including:

The product, as specified, to be available from multiple distributors

- Models of the components to exist so that none has to be created in-loop;
- Adequate compilers and trained software programmers to be available for the processor;
- Tools to have a listing of their problems so that no time is wasted tracking tool-related errors.

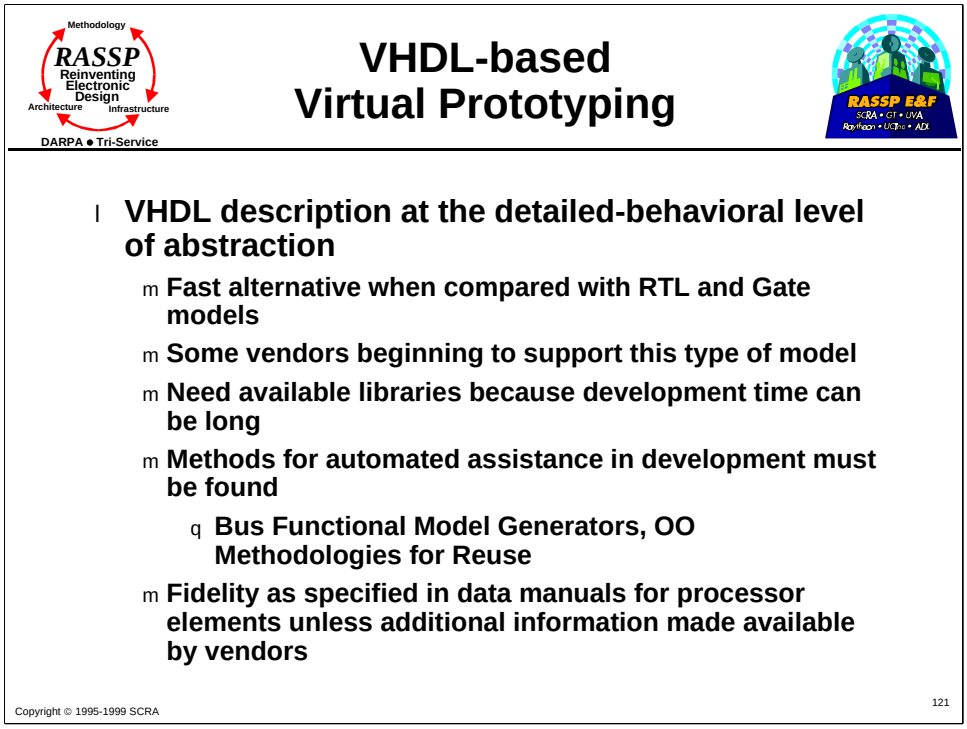


When looking at approaches to system modeling, we must consider the types of components found in a virtual prototype.

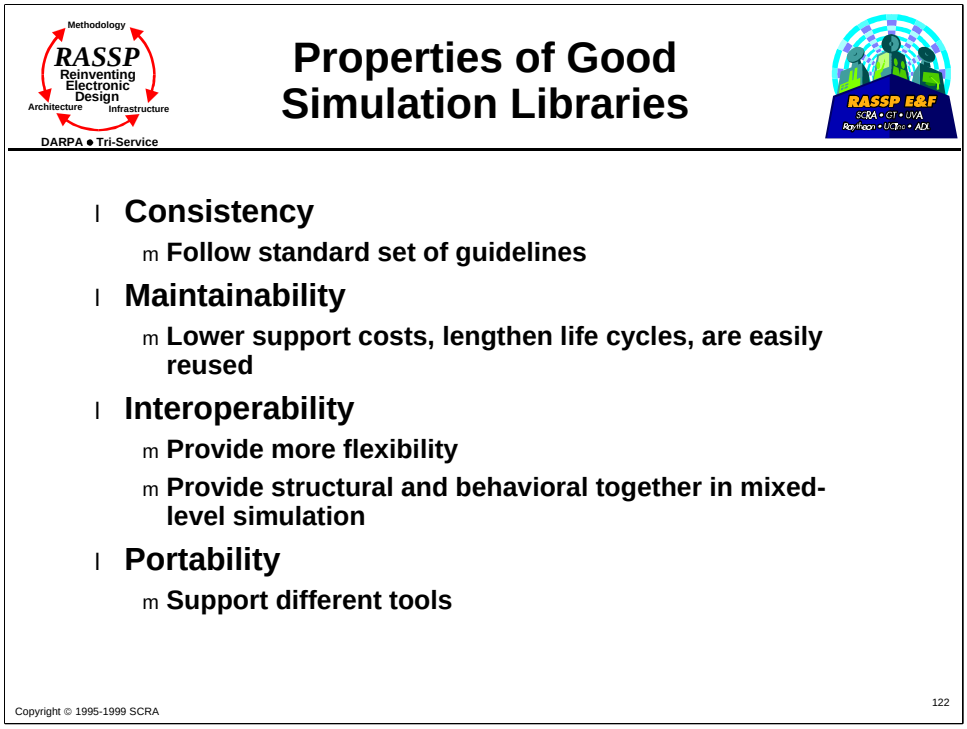
These include processors, ASICs, buses and other data networks, as well as any glue logic required to connect the component models.

Important issues with respect to this area include the use of high-level models and avoiding gate-level models at all cost because of simulation time considerations. We would also like to run code on the prototype. To do this, we need compilers for the processor and a method to translate the assembly/binary code to the model to run.

We also should require sufficient fidelity in the models to guarantee first-time HW design success.



The HDL description at the detailed behavioral level is the main focus of the design example presented later. This approach requires that models be made available by the vendors or third party model developers. The behavioral level helps keep simulation speeds within a reasonable scope. Work is currently being done on improving high-level modeling as well as system-level bus functional modeling. Object Oriented techniques are being developed by Vista Technologies as part of a MMC contract, and Logic Modeling Inc. has been working on a BFM generator. Georgia Tech has been developing processors and bus models as part of the techbase effort.

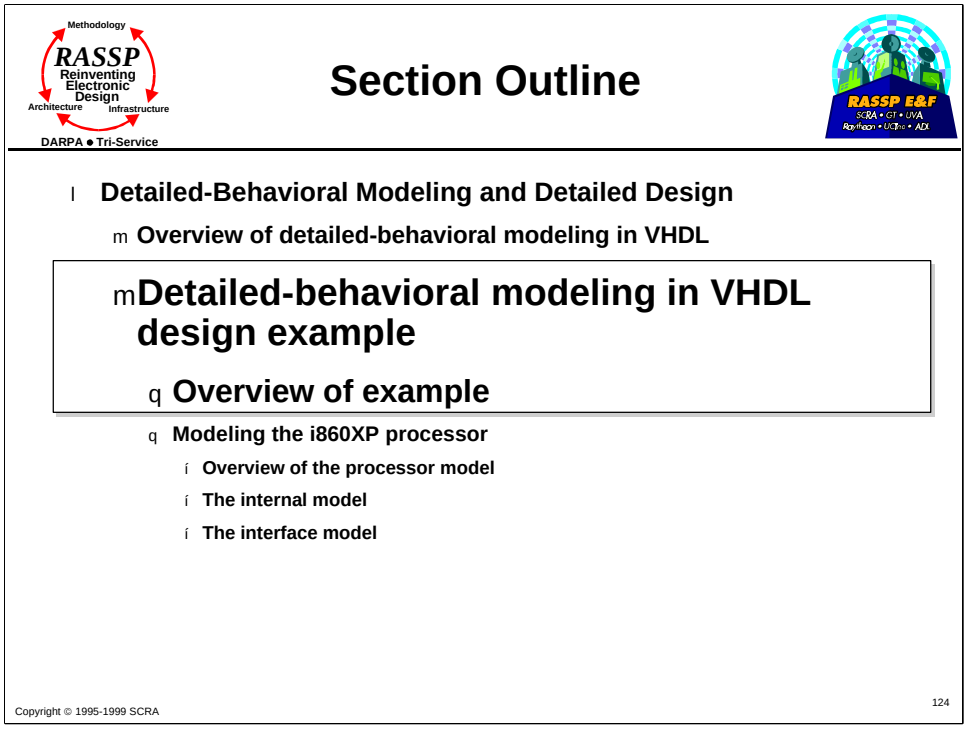


When evaluating or developing a library of models for system simulation, the following two slides cover a list of properties that should be evaluated or used during the process.

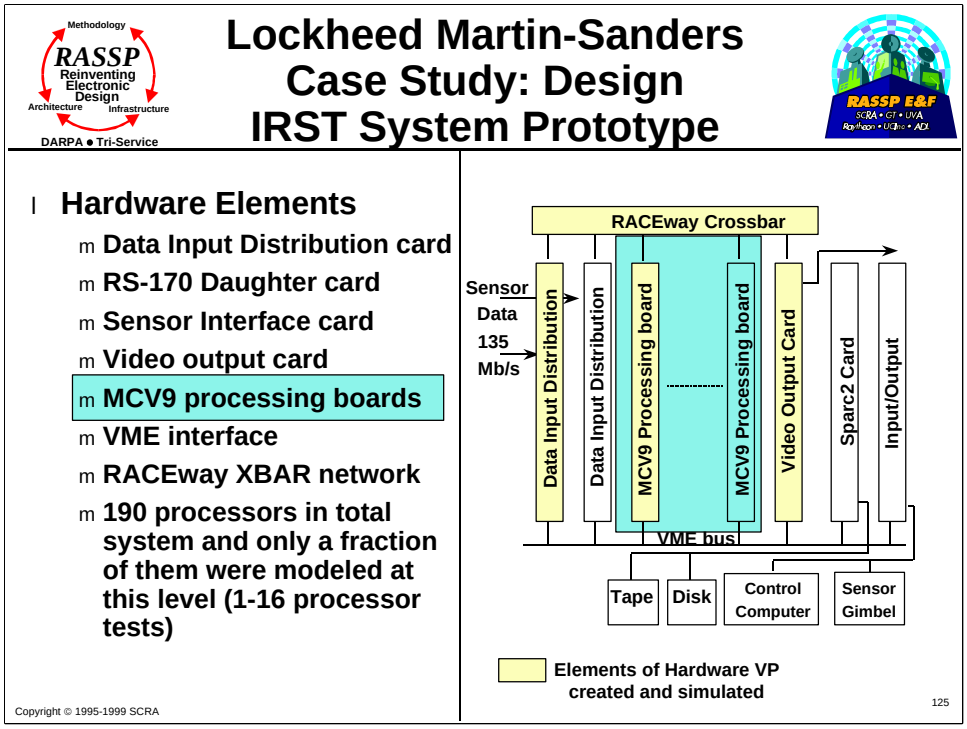
Existing commercial libraries attempt to satisfy all these requirements. The SmartModel™ library from Synopsys follows these properties closely. The SmartModel process is mostly automated to produce accurate, reliable, and reusable elements.

Properties of Good Simulation Libraries (cont.)

- **Accuracy**
  - **Provides fidelity levels, fully functional behavioral, RTL, Gate**
- **Performance**
  - **Higher-level models can run faster than gate-level with loss in resolution**
- **Memory efficiency**
  - **Smaller models use less of limited computer memory resources, especially for large system modeling**
- **Confidentiality**
  - **Protected models let the developers obtain proprietary information more easily**

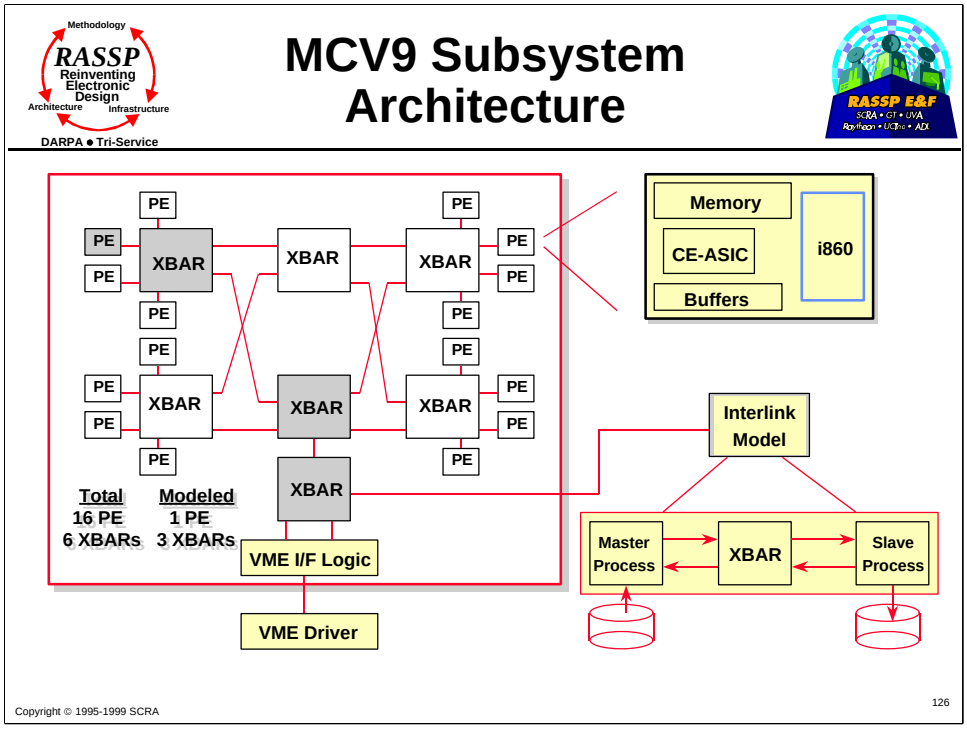


In this section, the detailed design examples are introduced. The IRST example is an example of a large scale detailed-behavioral model that was used to verify the full system with both hardware and software. The second example is also used as the laboratory example for this module. The lab exercise is M32_Lab_A on the CD-ROM.



This figure shows the IRST design case study that was done by the Lockheed Martin Sanders team on the RASSP program. Only those boxes that are shaded were modeled at the detailed behavioral level, mainly due to the size and simulation constraints of the system at this level of model abstraction. The system consisted of a total of 190 processors and only 1 to 16 processors were simulated at any one time. For most cases, only one was required to run the representative software that configured the external boards within the system. This configuration information was sent via the VME bus. Each block in the diagram was a board level model with a number of component elements on it. Because of the large number of components, faster simulation performance can only be achieved if the models are at a much higher level of abstraction. In this case, the models were developed at the detailed-behavioral level rather than the RTL or gate levels. As this module proceeds, examples of the code that was used to simulate portions of the system will be presented.

Focusing in on the MCV9 models that were developed, we see on the next slide the internal details of one of the boards. There were a total of 16 MCV9 boards in the overall system. The boards were interconnected using a RACEway network and control information was sent from the MCV9 board through the VME bus interface model.



This figure shows more detail of what is contained on an MCV9 board. The shaded regions represent the items that were modeled in VHDL. The XBAR models were generated from lower level gate models of the components. The processing element was developed at the detailed behavioral level. In order to run code on the system, only one processing element was required. The control code resided on the processor and configured external hardware via the VME bus. Interrupt information from external hardware was also sent to the processor via the VME bus. When the input sensor buffers were full of data, the processor was notified to configure a transfer to the internal memory the processing elements.

i860 was developed from the data manual description

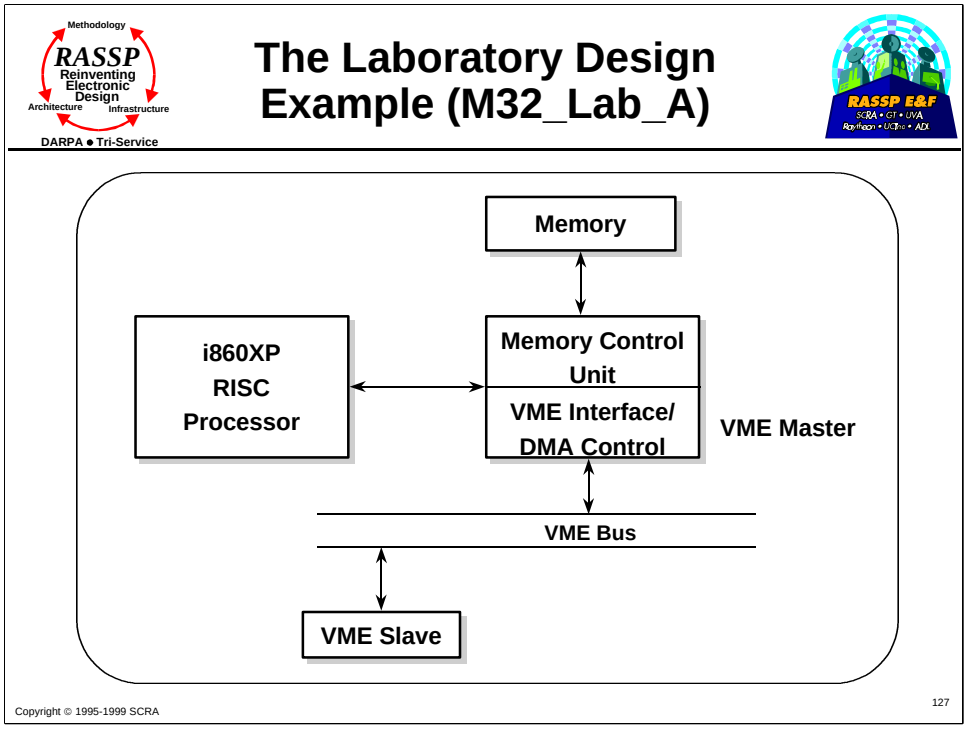
- Clock Cycle accurate
- Behavioral Description

CE-ASIC and XBAR models developed by converting existing schematics

- Translation tools from Mentor Graphics and Viewlogic
- Not straight-forward and not what was expected in all cases

A configuration file was generated for entire subsystem

This slide describes how and at what level the VHDL models for the various component elements of the system were created.The i860XP model was created at the detailed behavioral level while the XBAR models were created from translation tools which used gate-level models. The entire model was configured and tied together using a configuration file at the top-level.



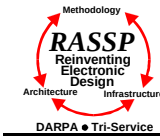
This is a block diagram of the proposed system design. The i860XP processor will be connected with the VME bus through the memory controller unit and a VME interface. When the DMA is configured to do burst writes across the VME bus, it obtains control of the memory from the processor using bus arbitration before doing the transfer.

The i860XP gets control of the VME bus and acts as the master while the slave is configured to write to a file. So any transfers from master to slave involve generating the data in the i860XP and sending it to memory. Following the loading of memory, the i860XP configures the DMA registers to transfer a specific amount of data from memory using the VME handshaking protocol. The data is then written across the bus to the slave address, and the slave places it into a file for later verification.

This laboratory will cover modeling of the dataflow, timing, and control of the major components of a system under design.

The system will consist of an i860XP processor, a memory unit, and a memory controller that interfaces to a VME bus and also contains a DMA controller.

Each of the individual components will be described, and their integration and test will be covered for a specific operating mode of the VME.

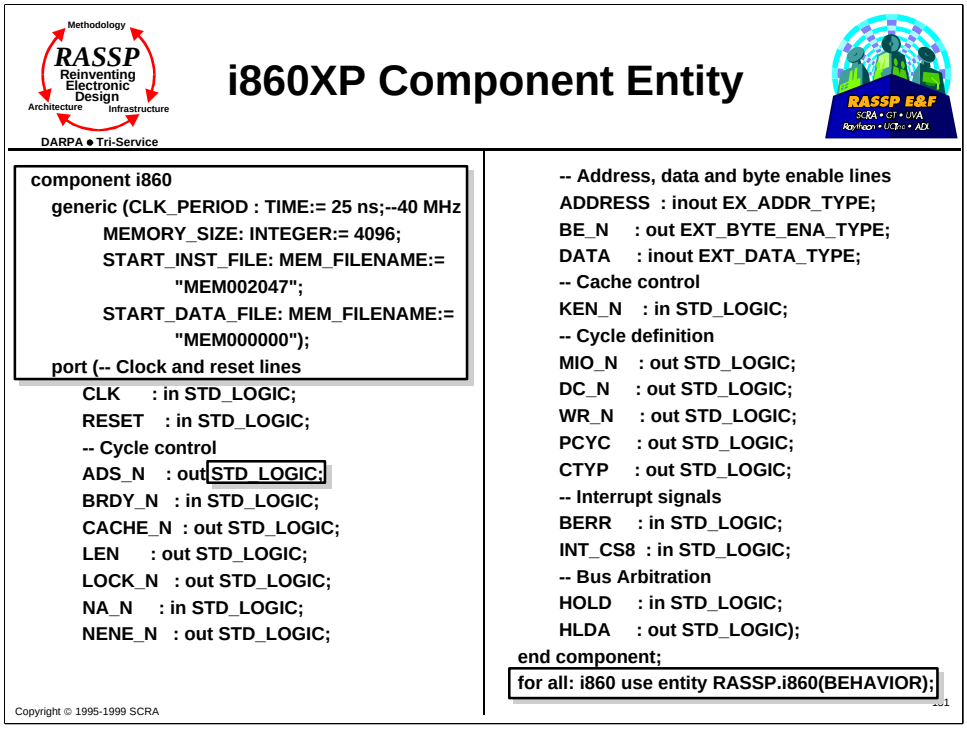


Additional CD-ROM Material on Detailed Behavioral Modeling

- One lab exercise is provided on the CD-ROM
  - Previous slide shows example in M32_Lab_A
- Application note on detailed behavioral modeling can also be found on the CD-ROM
  - See the section on Application notes to obtain this information
  - "RASSP Virtual Prototyping" application note
  - "VHDL Modeling Terminology and Taxonomy" application note

Section Outline

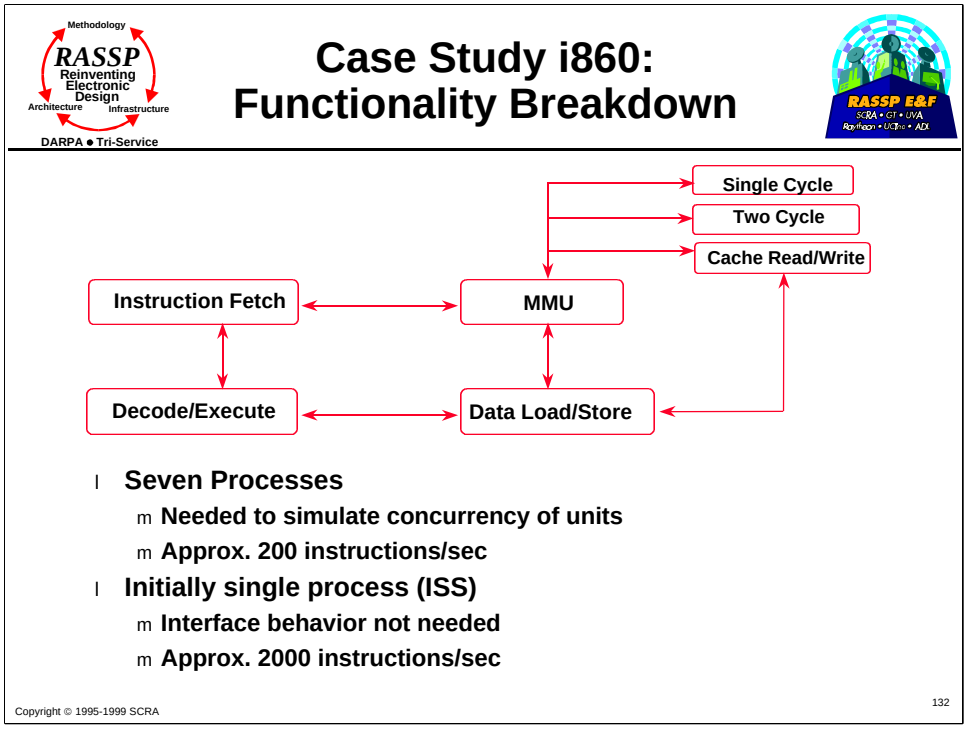
- **Detailed-Behavioral Modeling and Detailed Design**
  - **Overview of detailed-behavioral modeling in VHDL**
  - **Alternate approaches to early prototyping**
  - **Detailed-behavioral modeling in VHDL design example**
    - **Overview of example**
    - **Modeling the i860XP processor**
      - **Overview of the processor model**
      - **The internal model**
      - **The interface model**



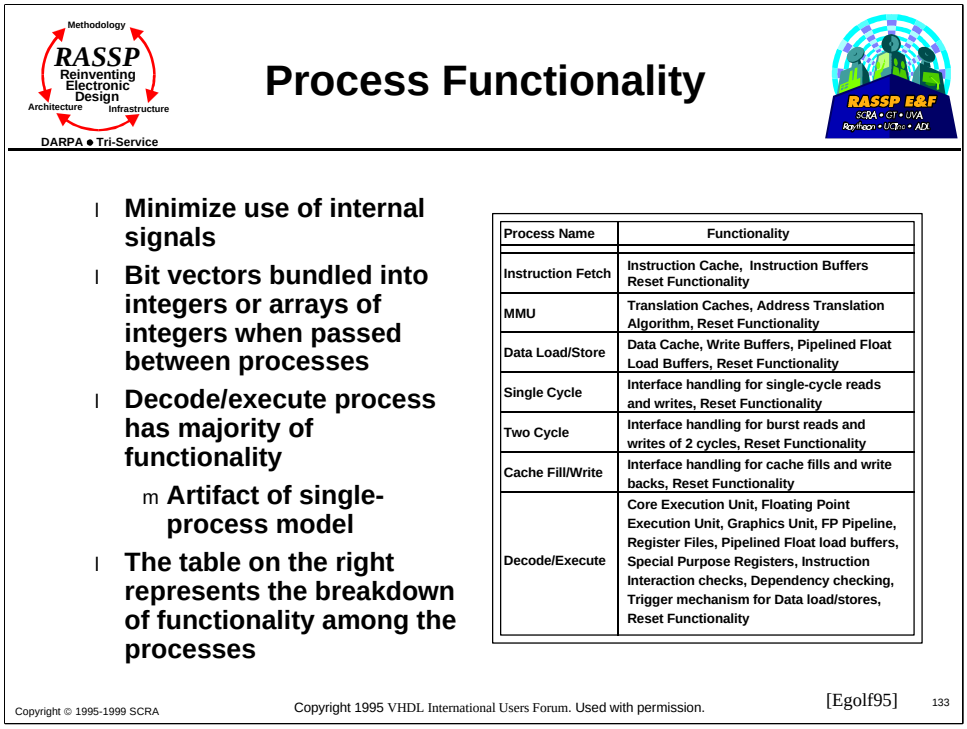
This code represents the i860XP entity pins required for this application. All the interface pins were not required to test this application; only those that had a direct impact were used. All interface pins are of standard logic or standard logic vector. The EX_ADDR_TYPE, EXT_BYTE_ENA_TYPE, and EXT_DATA_TYPE are standard logic vectors. This allows the model to be interoperable with all models that use the standard logic package (IEEE Std Logic 1164) for interface lines. The IEEE standard defines the 9-value logic levels that can be carried on signals of this type.

The same is done with the memory controller, memory, clock/reset, and VME models.

Generic parameters are used to configure the model using late binding (binding at the time of system configuration using the VHDL configuration statement). Alternatively, early binding can be done as shown in the code above. The “for all” clause at the end of this example binds the i860XP entity to the behavioral architectural description.



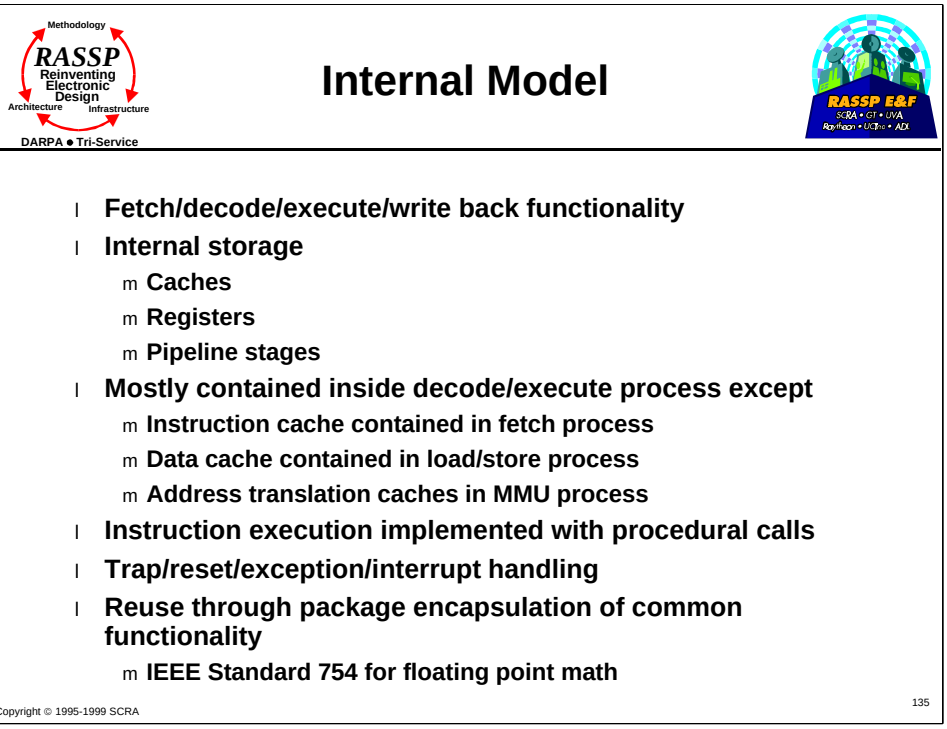
Given the various processor elements, the first step was to break down its functionality into specific processes. The initial attempt was to place all the functionality into a single process similar to instruction set simulator type models. This permitted faster running models, but could not capture all the concurrency issues of the processor. For example, if a cache miss occurred, the processor would need to go to external memory for data. In this case we do not want to stop the main process from executing what was already in, for example, one of the floating point pipelines, to wait for the data to arrive. Because multiple processes were required, the above 7 were chosen to represent the behavior of the device. The next slide lists the type of functionality in each process.



Since multiple processes were required, the attempt was to minimize the number of internal signals needed to pass information between the processes. By minimizing the number of signals used for communication between processes or entities, the number of simulation events can also be minimized. As the number of events is decreased, the simulation times improve. Vectors of bits were always bundled into integers and arrays of integers to help minimize the signal count because each bit was equivalent to an integer in VHDL. The decode/execute process contains most of the functionality because it is an artifact of the single-process model. The major storage elements such as register files, instruction cache, data cache, and translation caches were modeled as variables using record types. The instruction cache is contained in the instruction fetch process, the data cache in the data load/store process, and the translation caches in the MMU process. This helped permit the use of the variable type for each.

Section Outline

- **Detailed-Behavioral Modeling and Detailed Design**
  - **Overview of detailed-behavioral modeling in VHDL**
  - **Alternate approaches to early prototyping**
  - **Detailed-behavioral modeling in VHDL design example**
    - **Overview of example**
    - **Modeling the i860XP processor**
      - **Overview of the processor model**
      - **The internal model**
      - **The interface model**



The internal model is typical of the RISC processors pipeline stages, in such as fetch, decode, execute, and write back.

The internal storage elements such as caches, registers, and pipeline stages are modeled as variables.

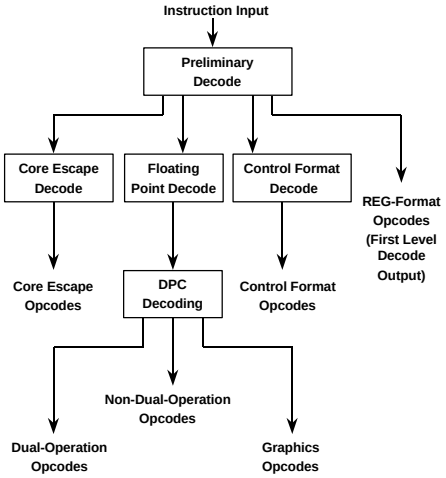
Most of the functionality is contained in the decode/execute process, but the cache elements are contained in other processes related to their functionality.

All instructions are executed using procedural calls. These procedures are encapsulated in packages and can be made reusable across similar RISC processors.

The processor must also be able to detect trap conditions, interrupts, and exceptions.

Instruction Set Decoding/Execution

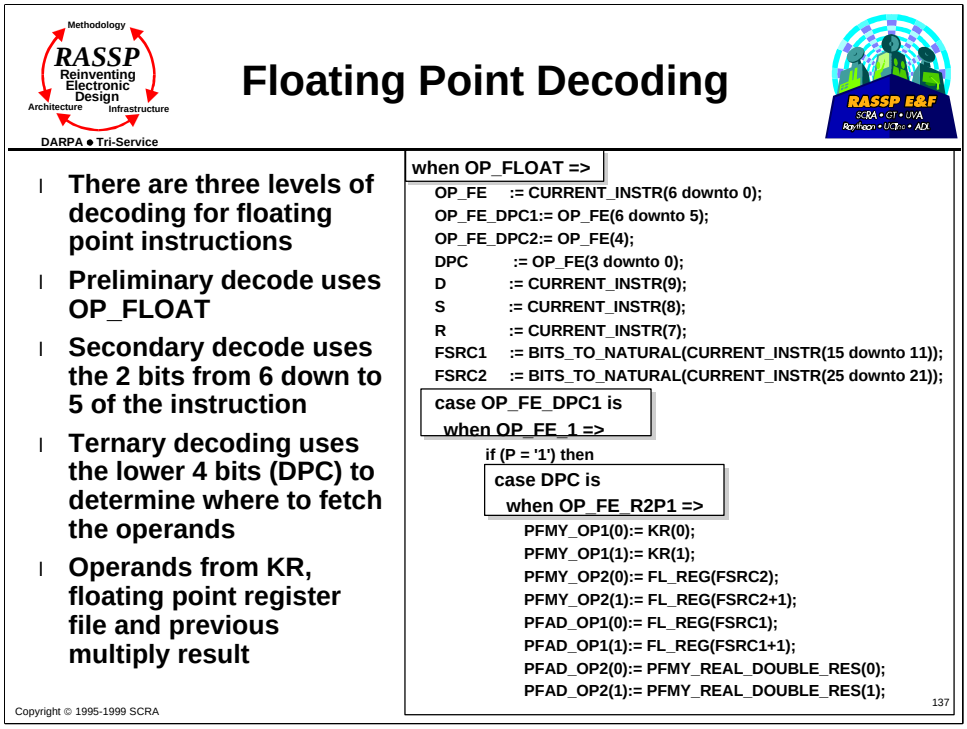
- **"Case" statements used to decode opcode**
  - **REG-Format => 1 case statement**
  - **Core Escape and CTRL-Format => 2 case statements**
  - **FP => 3 case statements**
- **Additional decoding done after opcode determined**
  - **Operand locations**
  - **Branch Offset**
  - **etc.**
- **Execution performed by use of procedural calls**



[Egolf95]

The decoding is modeled using a nested series of "CASE" statements. The instruction is input to the decode section and fields are stripped off based on the instruction type. The largest depth of decoding is three levels and occurs for floating point instruction types. The register format instruction type is decoded in the first stage, so it only requires one compare operation to determine its processing action.

Execution is performed at the leaf of each tree. At this point the operands are determined and loaded. The execution functionality in most cases is contained in procedures in a special package of instruction implementations.



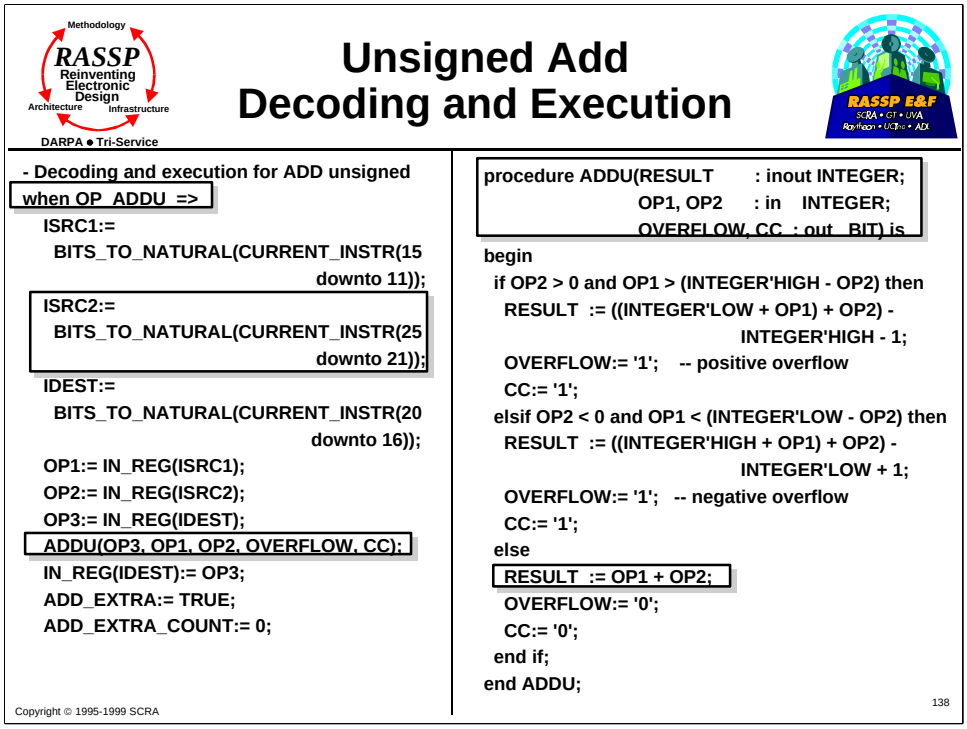
This slide lists a portion of the code used to decode the floating instructions. The upper 6 bits compare to OP_FLOAT and, if they match, then this segment is entered.

At this point, some of the fields are decoded to determine the next level of decoding required. Bits 5 and 6 of the instruction help determine the second level of decoding. In this case, they are compared to the constant OP_FE_1 and, if there is a match, the DPC values are decoded.

The DPC values represent the final stages of decoding, and at this point the operand locations are finalized. In the above case, the operands are fetched from the KR register, two from the floating point register file, and one from the previous result of the multiplier.

The operands are stored in integer array variable of length 2. This permits 64 bit operands to be processed.

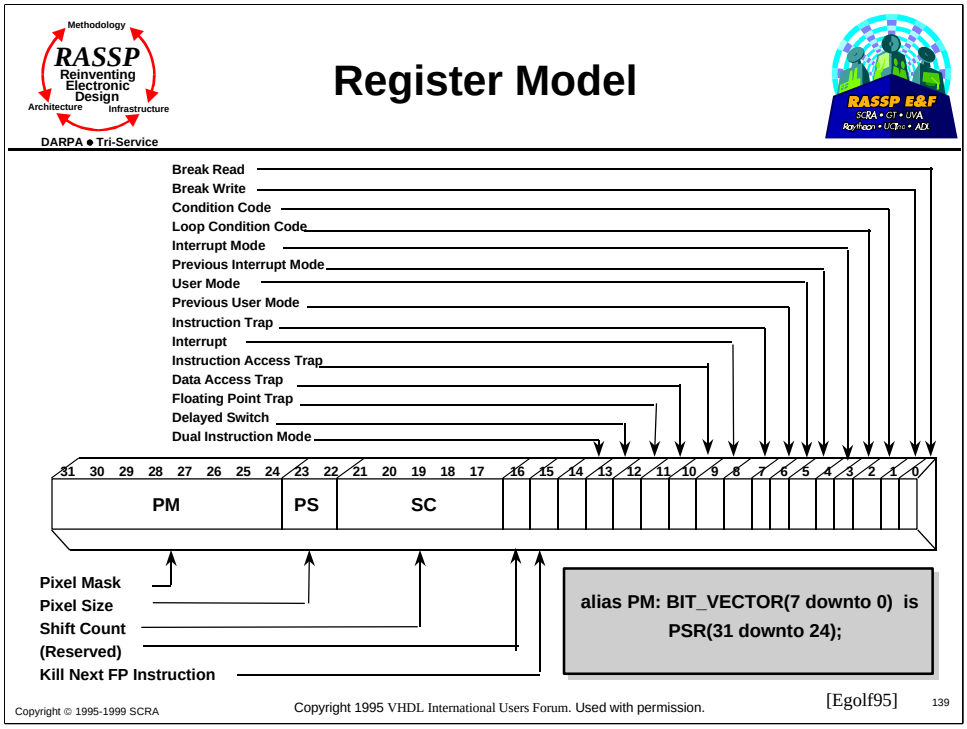
PFMY_OP1, PFMY_OP2, PFAD_OP1 and PFAD_OP2 are the floating point multiply and add operands, respectively.



This is the code for decoding and executing the unsigned add operation using VHDL. The left hand side does the decoding by having a match with OP_ADDU and then decodes the source and destination registers. It then loads the operands from the integer register file and calls the procedure ADDU, which performs the actual operation of an unsigned add.

The unsigned add procedure is on the right hand side of the slide.

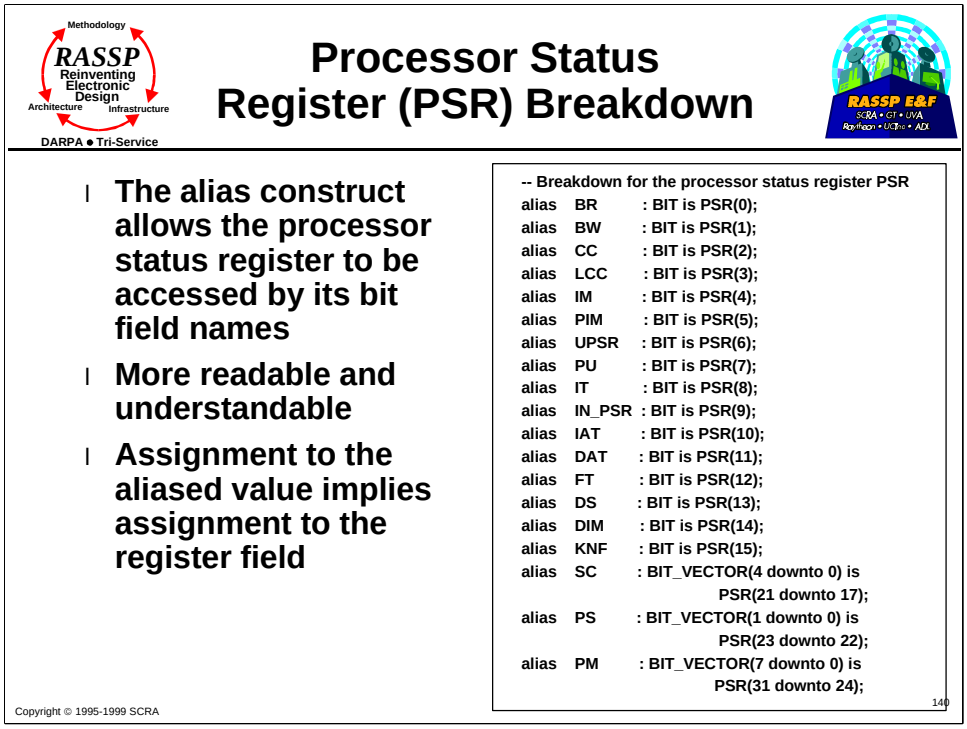
The ADDU instruction is a member of the REG-Format type instructions and hence only needs one level of decoding.



Special Purpose Registers with multiple fields are represented as bit vectors

- Permits “aliasing” of sub-fields for readability and ease of use
- alias PM: BIT_VECTOR(7 downto 0) is PSR(31 downto 24)

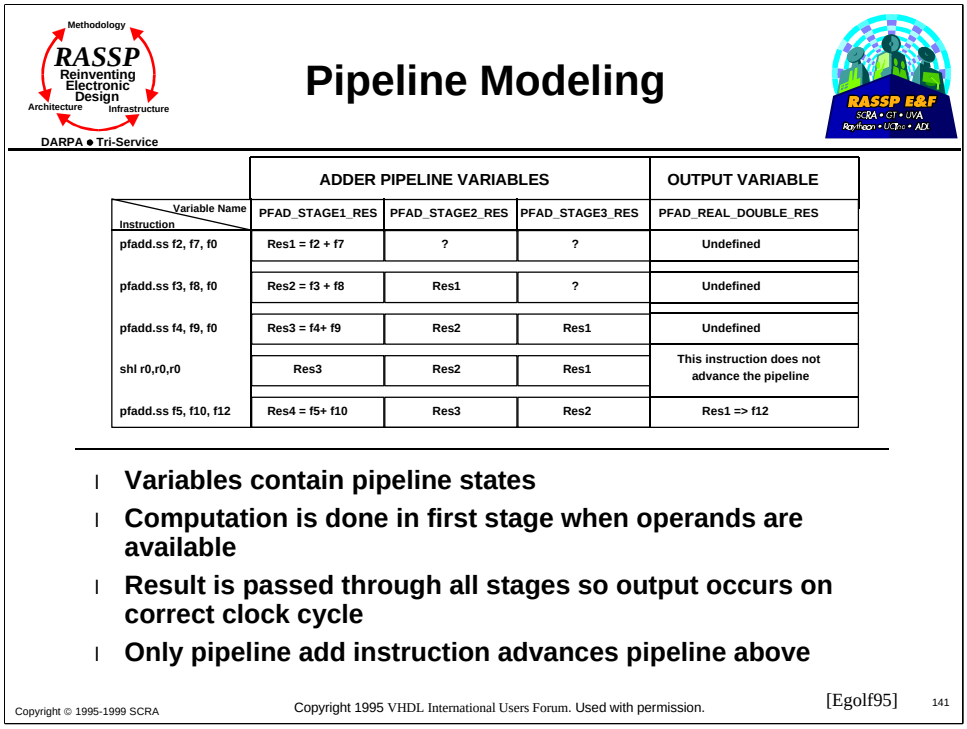
Register Files represented as arrays of integers because simulation Kernel size is reduced by using integers as compared to bit vectors



This shows the complete example code for the aliasing of the processor status register (PSR) using the VHDL alias construct.

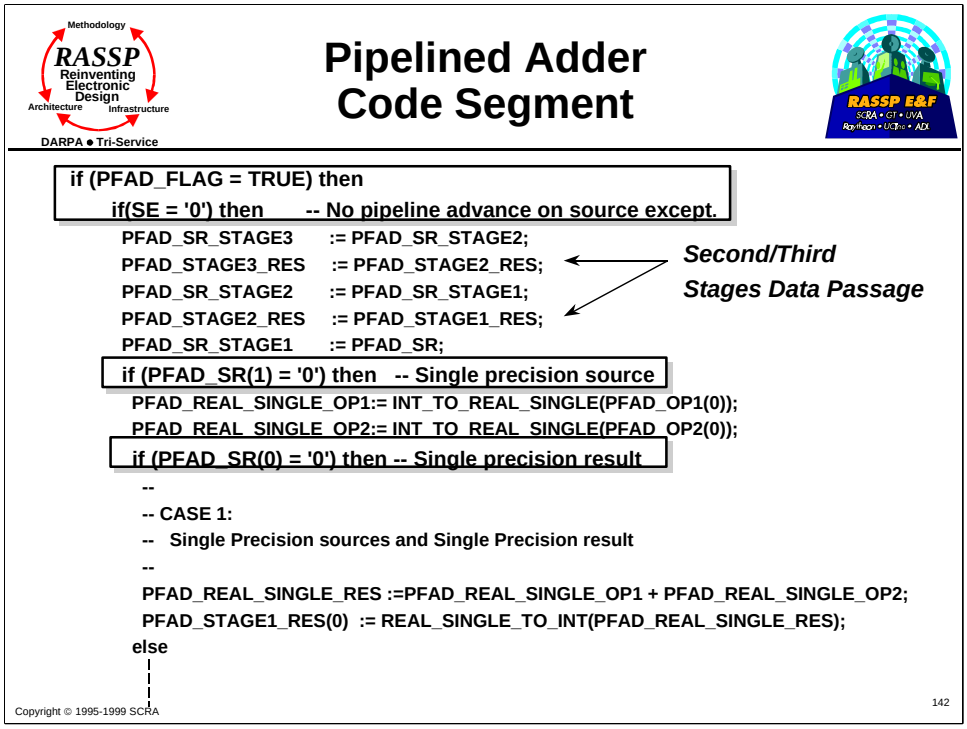
Aliasing allows the model developer to assign values to the aliased variables, which automatically results in the exact assignment being made to that portion of the aliased variable. For example, assignment to PM above would also be written in PSR(31 downto 24).

This creates more readable and understandable code because, when assigning or using the bits from the register, accesses such as: PSR(31 downto 24) can be avoided. Instead, we can use the name PM, and its intended purpose is known.



The floating point and graphics pipelines were implemented using variable data types. To simulate the pipeline performance, the results was calculated on the first stage of the pipeline when the operands were available, and the result was propagated through the various stages by passing the data from variable to variable on each clock cycle. This helped assure the data would arrive at the output of the pipeline on the correct cycle.

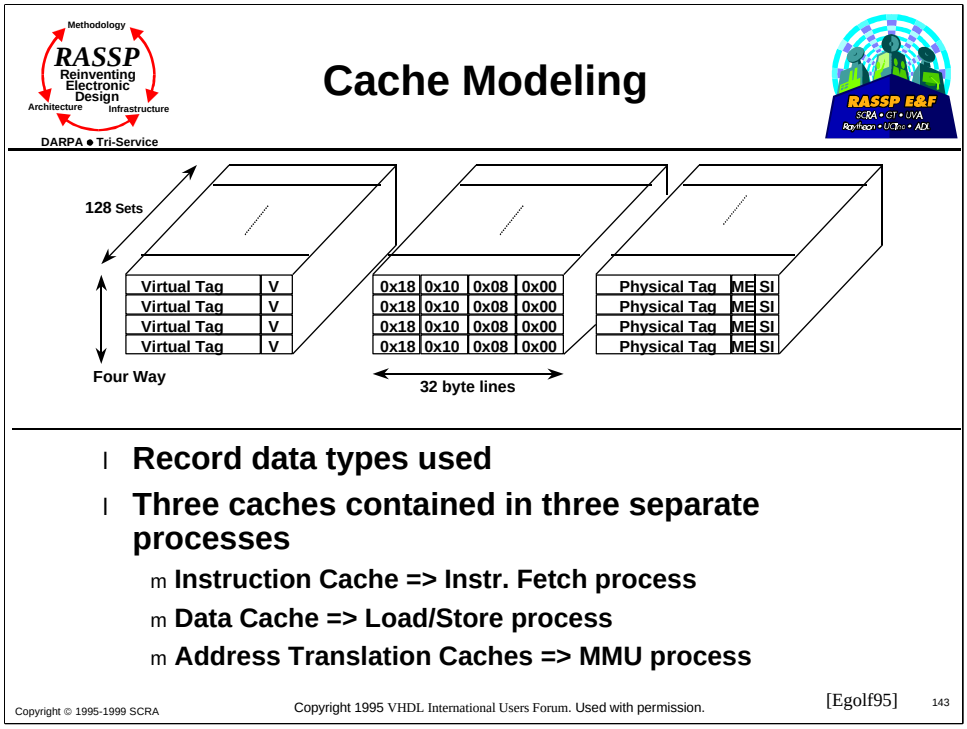
The example above is of the floating point adder pipeline. We see the variable PFAD_STAGE1_RES contains the result (Res1) after the first execution of the pfadd.ss f2,f7,f0 instruction. Each row represents a clock cycle of execution. On the second cycle we see Res1 being passed to PFAD_STAGE2_RES. This continues until the result is output to the variable PFAD_REAL_DOUBLE_RES on the fifth cycle. The pipeline was stalled one cycle because a non-pipelined instruction “shl” was executed in the middle.



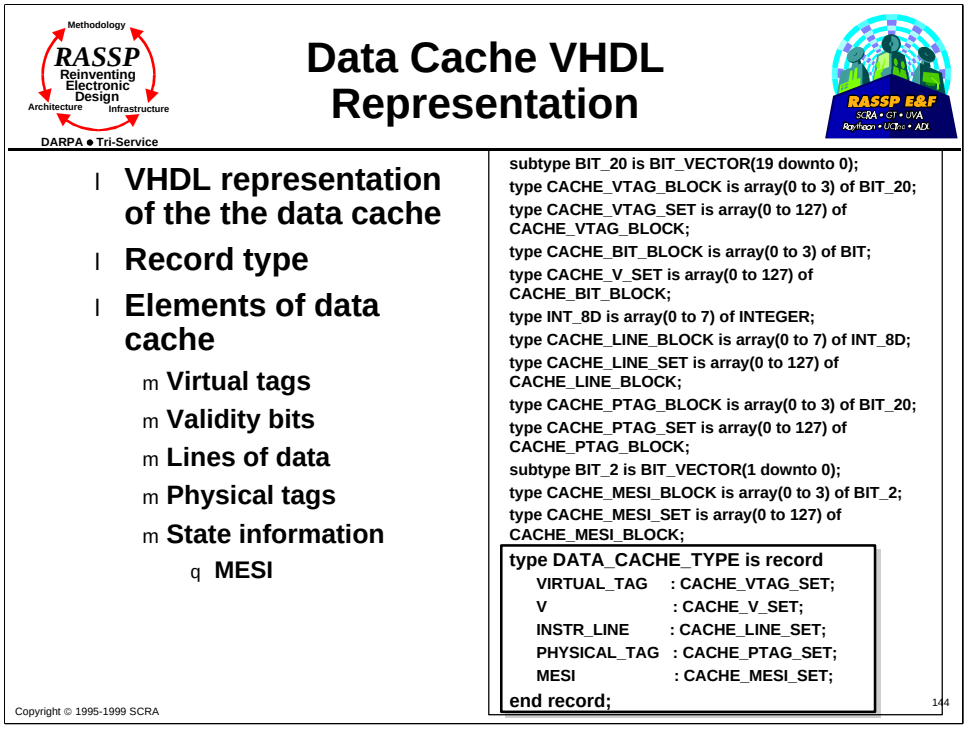
This slide shows the code used for modeling the adder pipeline.

The result is calculated in the first stage, and the results are passed to subsequent stages using the variables PFAD_STAGEx_RES, as can be seen above.

This code segment calculates the single-precision result of adding single-precision floating point operands. The highlighted regions show what needs to occur to enter this stage. First, the pipeline flag must be set. This is set in the decoding stage. Second, no source exception should have occurred. This also gets checked in the decoding stage when the operands are loaded into the variables PFAD_OP1 and PFAD_OP2 (not shown but can be observed if the code in the lab is analyzed). The precision bits are also passed through the pipeline in the form of the variable PFAD_SR, which is two bits wide and represents the precision of the source and result. If the source and result are both single precision as shown above, then we enter a section which performs the overloaded add operation of these types. The result is stored in stage 1. Since the adder is three stages, data is loaded into the first stage on the current clock cycle. On the second clock cycle, data is passed to the second stage of the pipeline and on the third clock cycle, data is passed to the third stage of the pipeline, where the result is finally stored back in the register file.

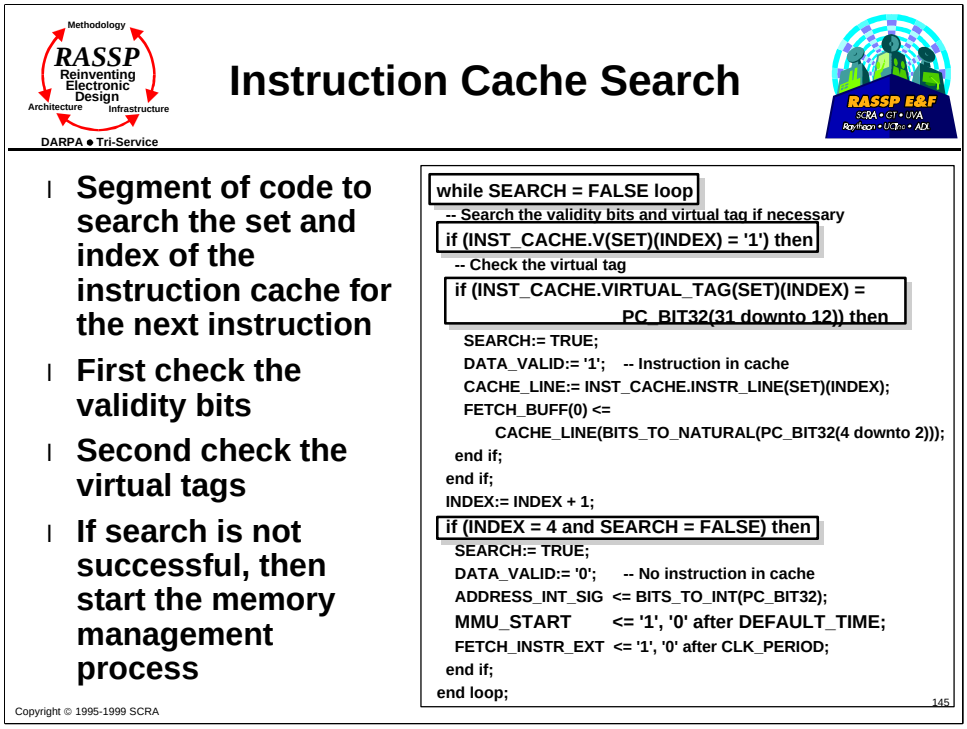


The caches were modeled using record types and stored in variable declarations. There were three caches in this model (instruction, data, and address translation caches) and the above diagram represents the model for the data cache. The instruction and translation caches were of less complexity. Procedures were written to encapsulate the functionality associated with each cache.



This slide shows the representation of the data cache in the i860XP model using the VHDL record construct. The other caches have a similar record structure. The next slide shows how the instruction cache is accessed using a similar structure.

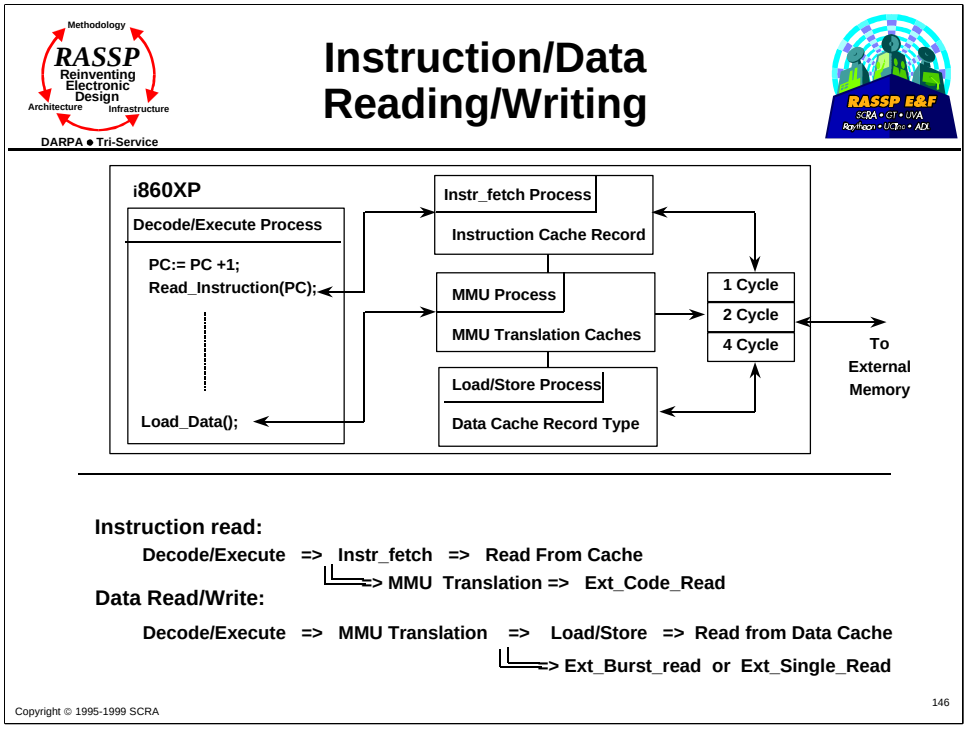
Five elements are contained in the data cache: virtual tags, validity bits, data, physical tags, and state information. These are modeled as fields in the data type for the cache.



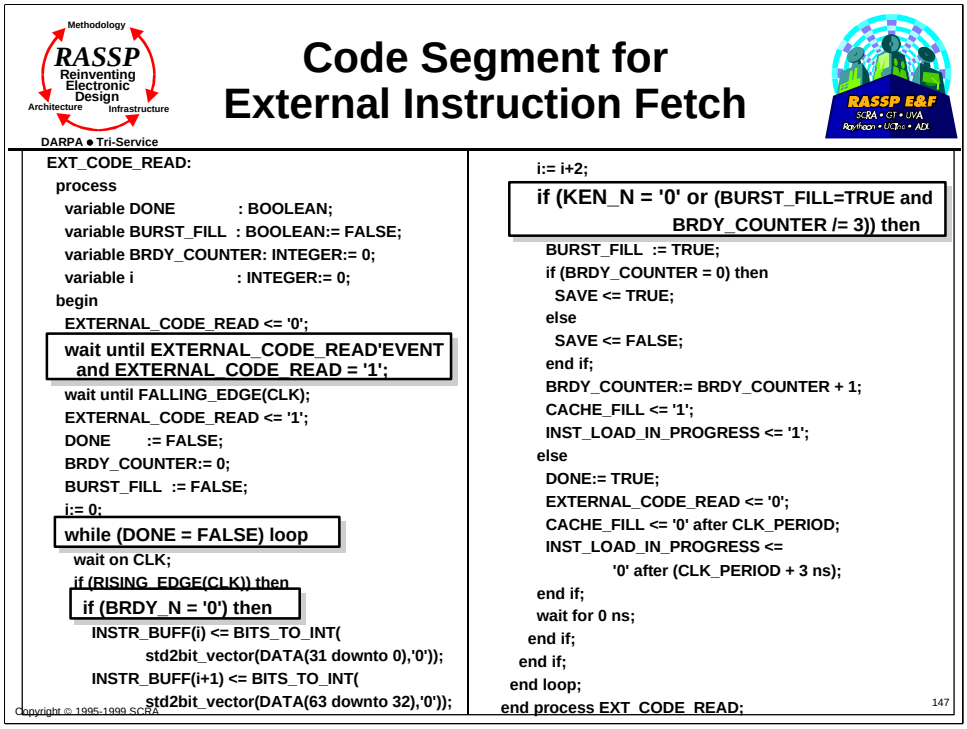
This segment of code searches the instruction cache for the next instruction to place in the fetch buffer (FETCH_BUFF).

Prior to this code segment the search variable is set to false, the cache set is determined, and the index is initialized to zero. It loops through the current set looking for valid data bits set for each line. If it finds one, it then compares the virtual tag for that index with the current PC values upper bits. If it finds a match, then the search is set to true, and the data is loaded into the fetch buffer.

If no match is found, then this would imply the index being equal to four (4-way set associative), and the search variable is still false. At this point the memory management is started to do the translation to a physical address.



Typical loads and stores of data and instructions follow the above process-initiation sequence. The decode/execute process triggers the instruction fetch process when it takes an instruction from the instruction buffer. This allows the instruction fetch process to begin its next fetch while the decoder breaks down the instruction it pulled from the fetch buffer. The fetch buffer was created to feed one or two instructions to the decode/execute process based on the mode of execution (single or dual mode). If address translation was required, the MMU process was triggered next. On data loads and stores, the MMU process was always entered because translation is always done on this type of operation. It then triggered the data load/store process. Once the type of transaction was determined, the handshaking protocol was done by either a single-cycle, two-cycle, or four-cycle process. This was based on whether the transaction type was a 64 bit or less operation, 128 bit operation, or cache fill operation, respectively.

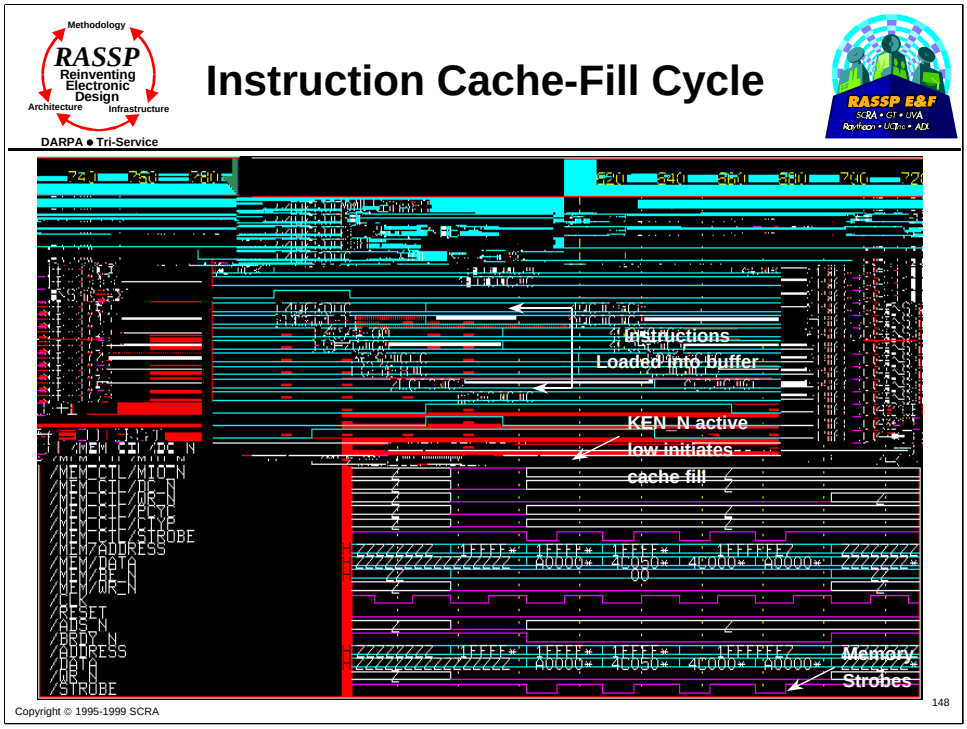


This code segment describes what occurs on a cache miss, and the instruction needs to go to external memory for data. The first thing that occurs is some external process sets the signal EXTERNAL_CODE_READ to “1”. This triggers the process. The role of this code segment is to sample the BRDY_N input signal on each rising edge of the clock and, if it is active, it proceeds loading the data from the bus into the instruction buffer and later into cache if a cache fill is in progress.

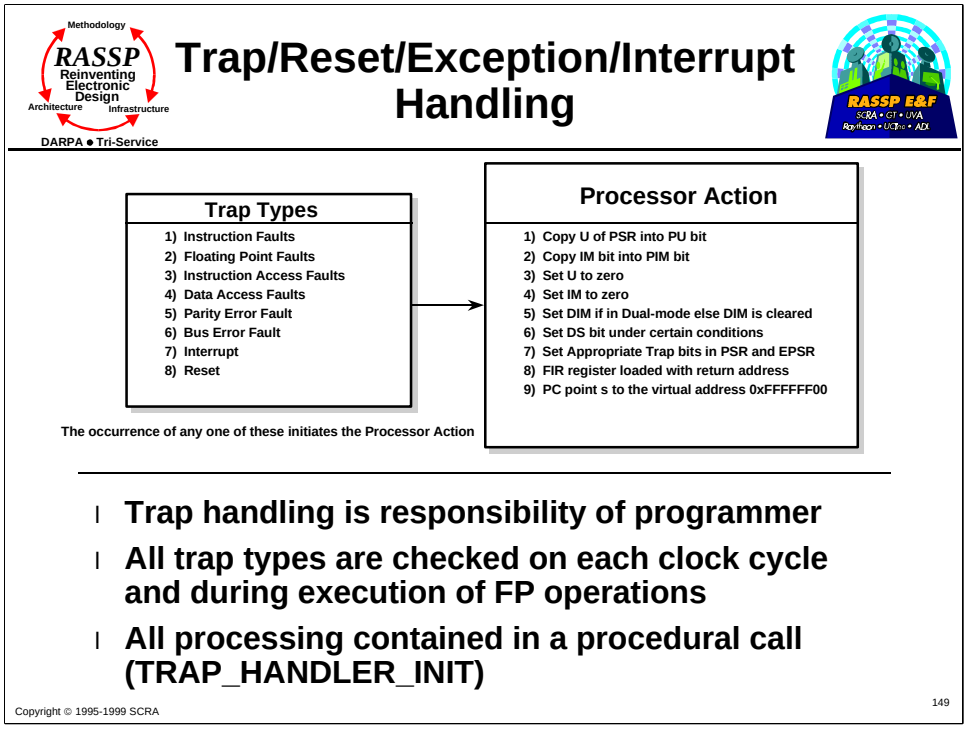
The *while loop* continues until variable DONE is set to TRUE. This samples the BRDY_N line for a “0” on the rising edge of the clock. If the KEN_N input which is set by external hardware is detected to be a “0” also, then the cycle is converted to a cache fill, given that the processor is set up for a cache fill. At this point, the processor is switched to a burst mode line fill (BURST_FILL is set to TRUE and the BRDY_COUNTER is incremented to “1”).

The second time through this segment KEN_N may not be “0” anymore but BURST_FILL is TRUE and the counter is “1”. It gets updated to “2”. On the third load it gets updated to “3” and on the 4th load, because it is “3”, the variable DONE gets set to TRUE.

If KEN_N never goes low, then the normal 64 bit read is performed, and no cache fill is done.



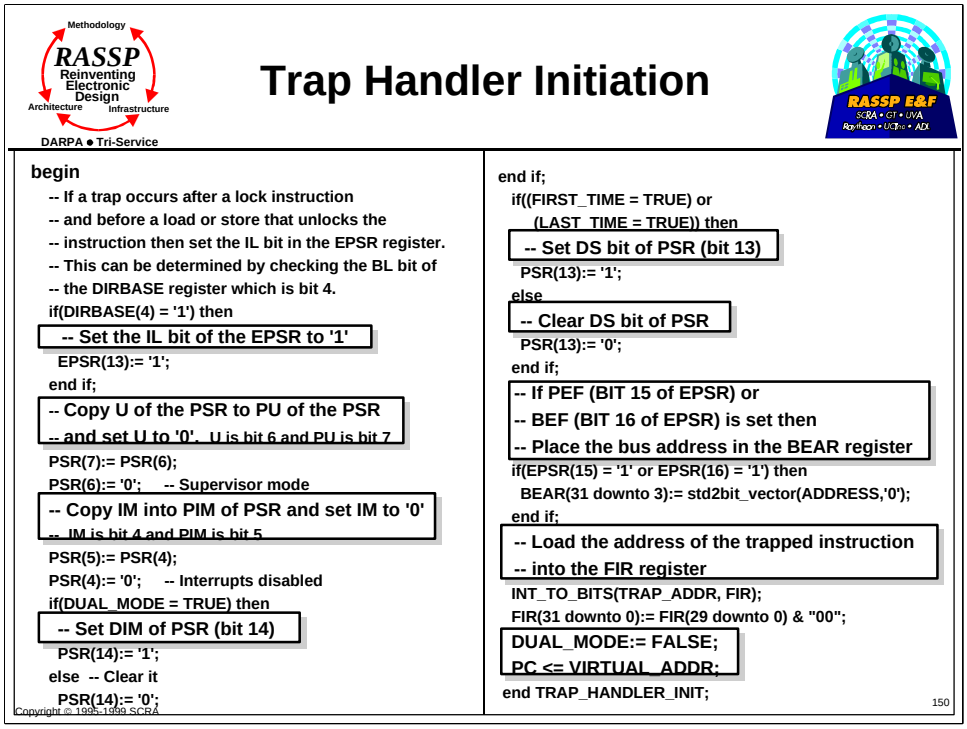
This diagram shows the timing of an instruction cache-fill cycle from an access to external memory going through the memory controller. The cycle begins when the address is not found in the instruction cache. At this point, the “FETCH_INSTR_EXT” signal is set as well as the instruction load in progress signal (INST_LOAD_IN_PROGRESS). The ADS_N signal is set to low to activate the code read, and the external memory controller sets the KEN_N signal from the memory controller unit to tell the processor it plans to do a cache-fill type cycle. At this point the processor stops driving the address lines and the memory controller begins. The STROBE signal indicates the times when the memory is being asked for data. There are no wait states, and in this case data is returned on every clock cycle. The length of memory wait states is programmable by the use of VHDL generic inputs. Data is returned into the INSTR_BUFF array in locations 0 to 7 and is sent to the cache at the same time.



A separate package was created for encapsulating the functionality of exceptions, reset, traps, and interrupts.

The actual code for handling the traps is the responsibility of the operating system or the user.

These types of conditions were checked on each clock cycle and, if a condition occurred, the processor action was taken above. This action was contained in a procedural call.



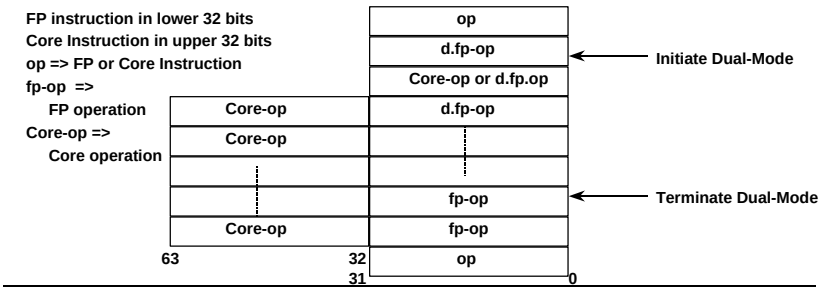
This segment of code shows the processor action taken initially when a trap occurs. The trap types were shown on the previous slide.

This routine is implemented as a procedural call.

The appropriate bits in the special purpose registers are set accordingly.

The only thing not set in this routine are the appropriate trap bits in the processor status register and extended processor status register that indicate the type of trap. This is done outside the procedural call in the routine testing for trap conditions.

Dual Instruction Sequencing

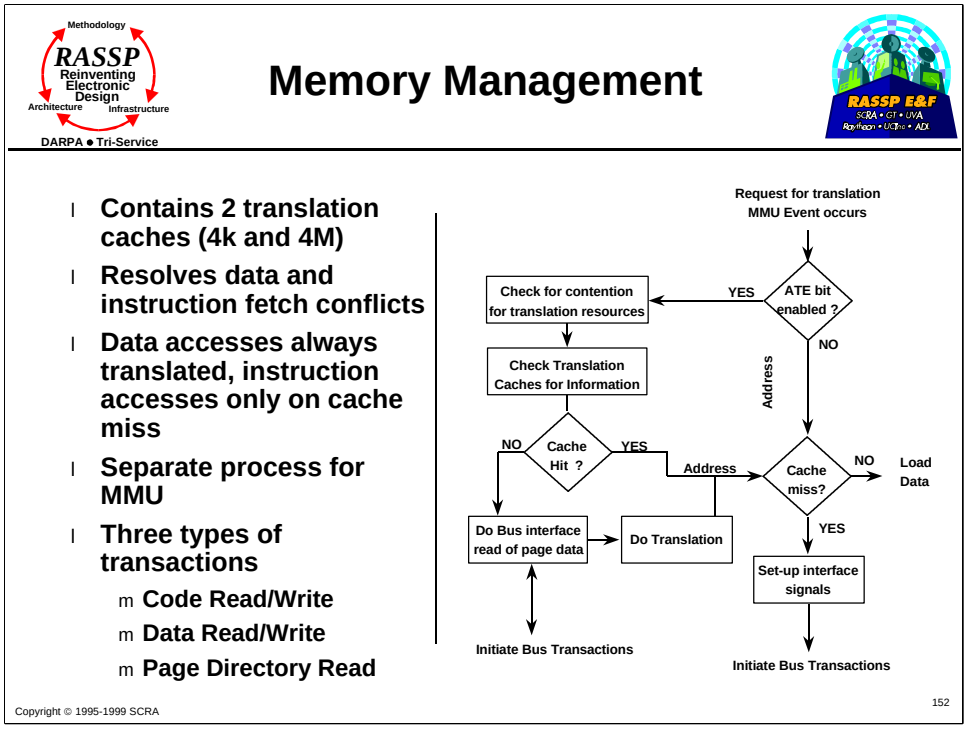


- **Variable set to TRUE when FP instruction has Dual-Mode bit set**
- **DUAL-MODE variable is set after the next execution**
- **When in dual-mode, instruction interactions are first checked before executing the FP and core instructions**
- **Terminating Dual-Mode uses two variables and is similar to initiating**

151

Dual instruction sequencing is done in the decode/execute process and uses boolean variables to determine how many instructions to execute. Dual mode is initiated on the second cycle following a floating point instruction with the dual bit set. When in dual-mode, the decode/execute section of code is executed twice on the given clock cycle. Prior to doing the dual execution, any instruction interaction is first checked.

Dual mode is exited using boolean variables to determine the time of shut-down. When in dual mode and the dual bit is no longer set in the FP instruction, a flag is set to TRUE and if TRUE on the next cycle, the dual mode variable is set to FALSE on the following clock cycle. This allows for the two-cycle shutoff to occur.



The memory management unit is a separate process and it contains two translation caches (4k and 4M). A request for translations comes from either the decode/execute process on data load/stores or the instruction fetch process on instruction cache misses.

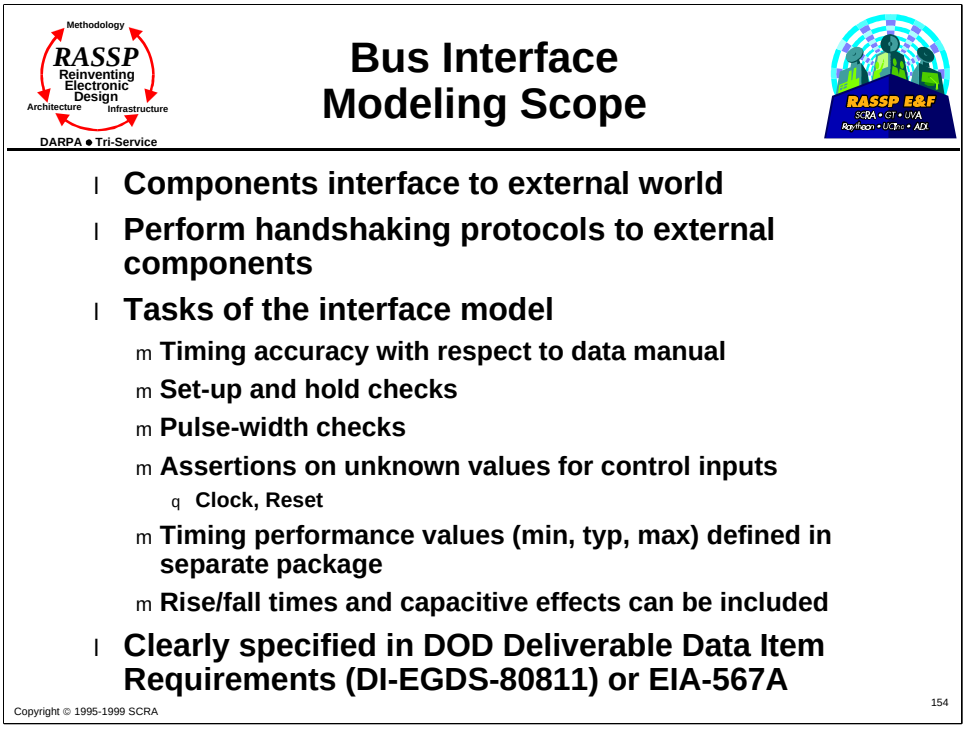
If the translation enable bit (ATE) is not set then no translation is done. The physical address is then checked with the caches for a hit of the physical tags. If none occurs then a bus transaction is initiated with that physical address.

If the translation bit is set and an instruction cache miss occurred, then translation is done to form a physical address. A bus transaction is done using this physical address, and the physical address is then compared to the physical tags of the instruction cache for a hit. If there is a hit of the physical tag, then the entire line is overwritten with the new data returned.

If the translation bit is set and a data cache transaction occurs, then we always do translation. In this case, if the virtual tag was found in the cache, the only thing that could happen would be to update the physical tag associated with it. If the virtual tag was not found, then the translation is done and the physical tag is computed. The physical tag is then searched in the cache at the same time a transaction is done on the bus. If the physical tag is found in the cache, then the data returned on the bus is ignored. If not then the both tag fields are updated.

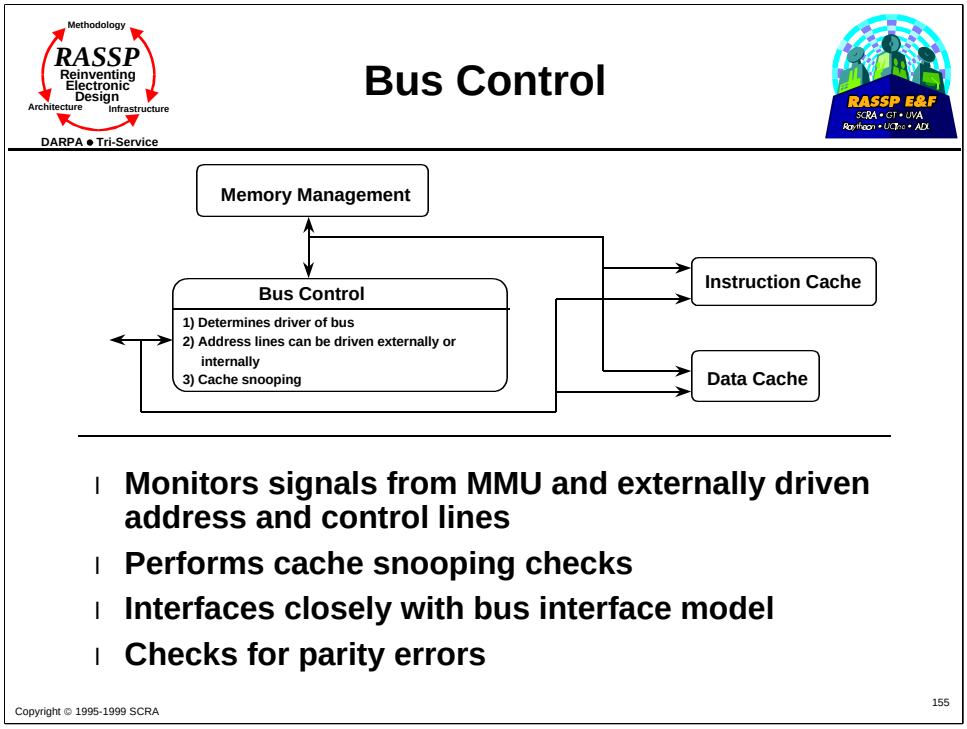
Section Outline

- **Detailed-Behavioral Modeling and Detailed Design**
  - **Overview of detailed-behavioral modeling in VHDL**
  - **Alternate approaches to early prototyping**
  - **Detailed-behavioral modeling in VHDL design example**
    - **Overview of example**
    - **Modeling the i860XP processor**
      - **Overview of the processor model**
      - **The internal model**
      - **The interface model**

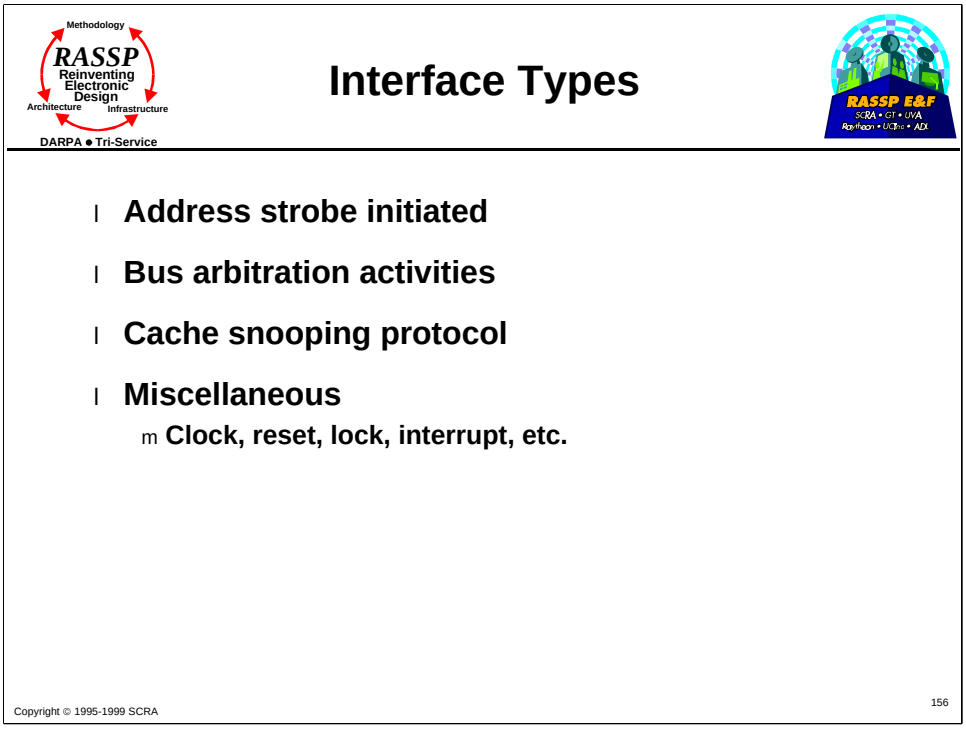


Without the bus interface model, the i860XP would be similar to an ISS model. The interface provides connection to external components and performs the necessary handshaking required to do memory operations, bus arbitration, etc.

The tasks of the interface model are listed above and include most of the timing and waveform checks common to the timing diagrams found in a processor's users manual. The requirements of the interface model are specified in the DID and EIA-567A.



The bus control functionality is dispersed among most of the processes determining who gets control of the buses as well as doing parity checking errors. Input signals related to the snooping function are monitored and, when external access is required, the appropriate processes are triggered. The bus control units within the model perform the handshaking protocol through the interface pins of the device to interact with the external memory controller and memory components.



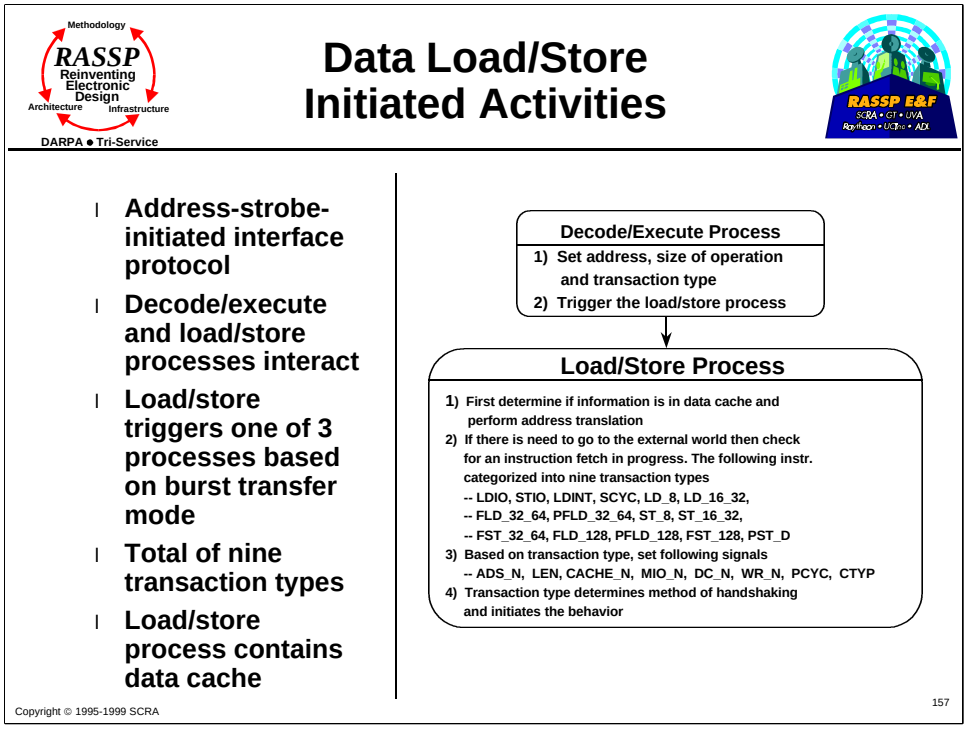
For the i860XP, there are four main categories of interface types listed above.

The address-strobe-initiated type include all activities associated with accessing data and instructions from external memory. This includes cache fills and write backs, 64 bit data loads, etc.

The bus arbitration activities deal with the ability of external devices to gain control of specific lines (i.e., address lines) of the i860XP to perform a defined task. Signals such as HOLD and HLDA are of this type.

Cache snooping interface types allow for external devices to look at the information contained in the i860XPs caches. To achieve this, they need to gain control of the address lines using a specific handshaking protocol.

There are miscellaneous signals that perform such things as reset, clocking, and interrupts.

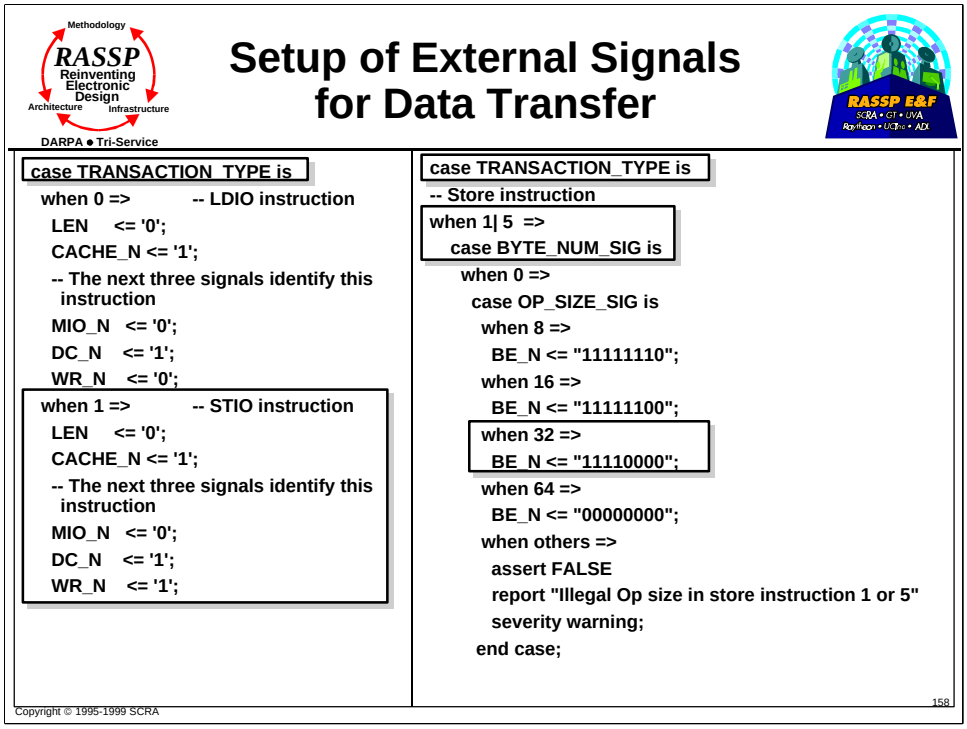


The data load/store process contains most of the address-strobe-initiated interface protocol. The remaining address-strobe-initiated protocol is contained in the instruction fetch and the MMU processes where code and paging information are read.

The data load/store process is triggered by the decode/execute process, and the load/store process triggers the transfer mode type process (single-cycle, two-cycle, or cache-related cycles).

There are a total of nine transaction types defined in this process based on the types of signals to be setup. A list of instructions covered by these nine types is shown in the lower box.

The data cache is also contained in this process.

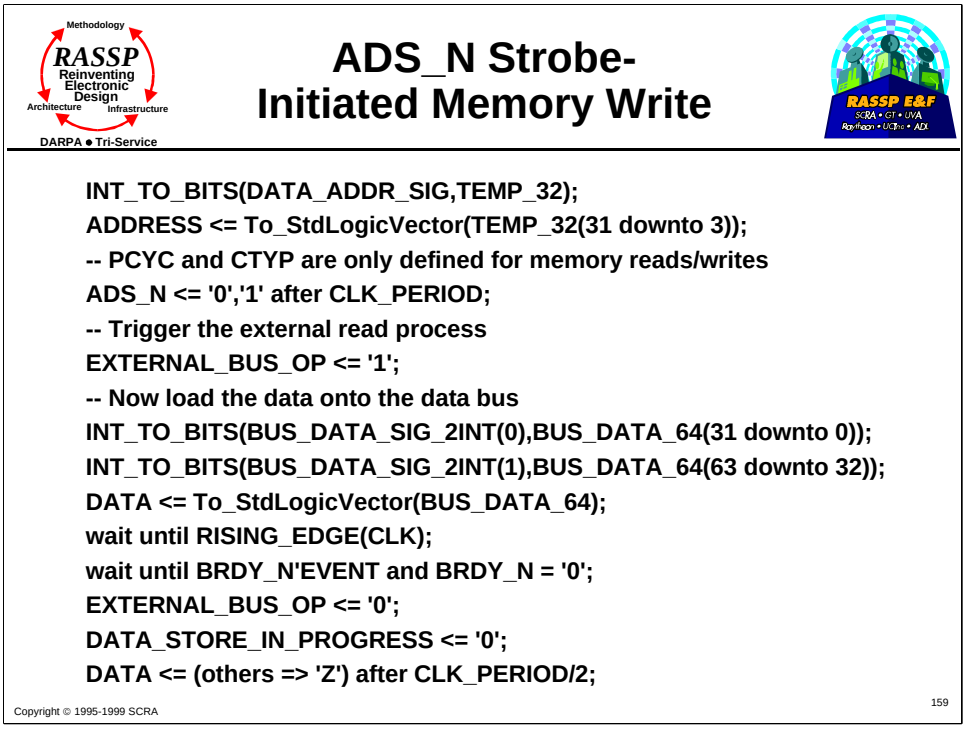


The execution of a load or store instruction sets the transaction type, BYTE_NUM_SIG, OP_SIZE, and passes the address to the load/store process.

This set of code is contained in the load/store process and is based on the transaction type set up by the execution of a load or store instruction; the appropriate signals to the external world are set to describe the type of transaction. The key signals are LEN, CACHE_N, MIO_N, DC_N, WR_N, and the byte enable signals BE_N.

The next slide shows some of the code to initiate the handshake with the memory controller.

For example, if we had a STIO instruction, which is a store to an IO address, then the transaction type would be 1. If the operand size is 32 bits and the byte number is “0” then the code in the boxes is executed.

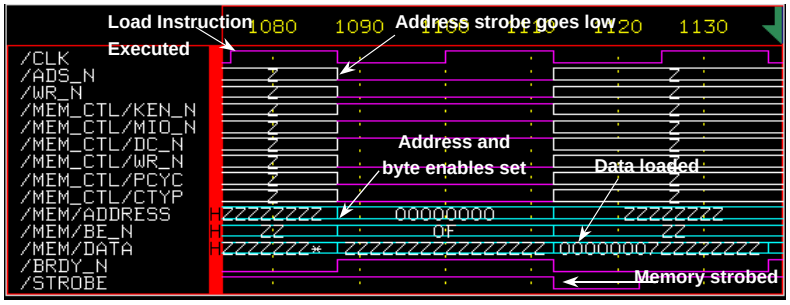


This is the code executed in the case of a floating point store instruction, when the transaction type is 5.

This set of code follows that of the previous page and takes the address supplied by the execution of the store instruction and drives the address bus (ADDRESS <= ....).

At the same time, it asserts the ADS_N line which initiates an address strobe transaction. It puts the data on the bus and waits for the next rising edge of the clock. It then waits for assertion on the BRDY_N line. When this occurs, control is handed over to the single write transaction process (setting EXTERNAL_BUS_OP <= ...). That process continues to drive the data lines until acknowledgment of memory storage is returned.

Timing Diagram for Load Operation



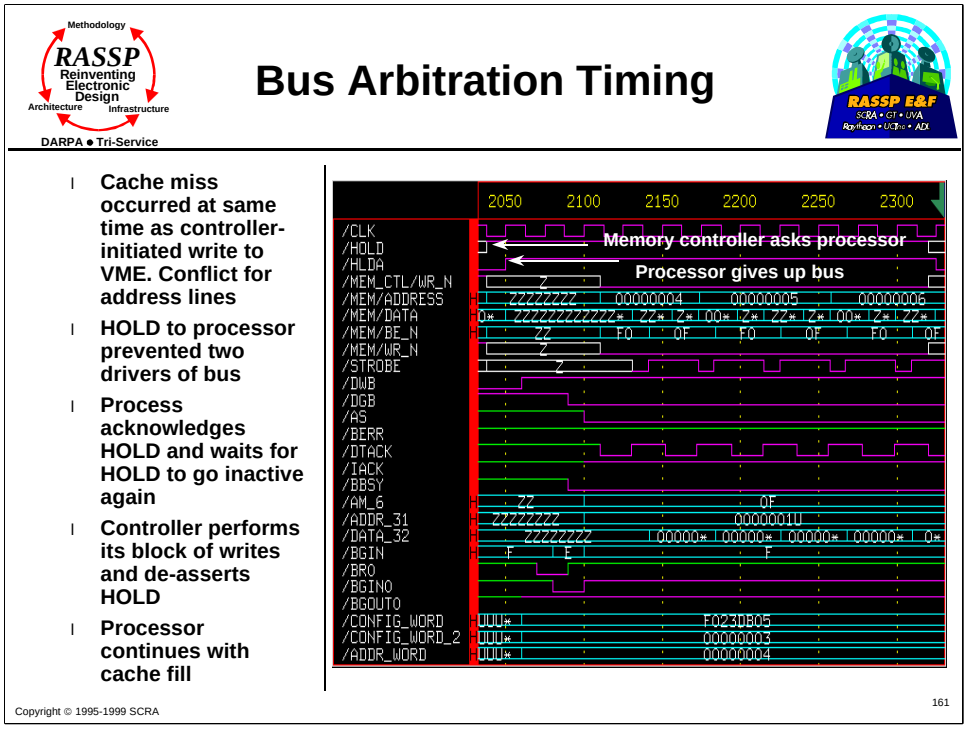
- **Three clock cycles to perform the data store because of memory controller wait states**
- **Clock prior to ADS_N active initiates the store operation**
- **BRDY_N active low from the memory controller signifies data is ready**
- **ADDRESS and BE_N lines determine location in memory**

160

This diagram shows an example of the timing associated with a load/store transaction.

ADS_N initiates the transaction by going low on the falling edge of the clock. Data is put on the bus by the memory after the strobe signal is activated.

The transaction type is determined by MIO_N, DC_N, WR_N, PCYC, and CTYP. 32 bits of data are being driven on the bus during the load operation. The address is set to “0”. When BRDY_N is returned by the memory controller, this means the processor should read the data from the bus on the next rising edge of the clock.

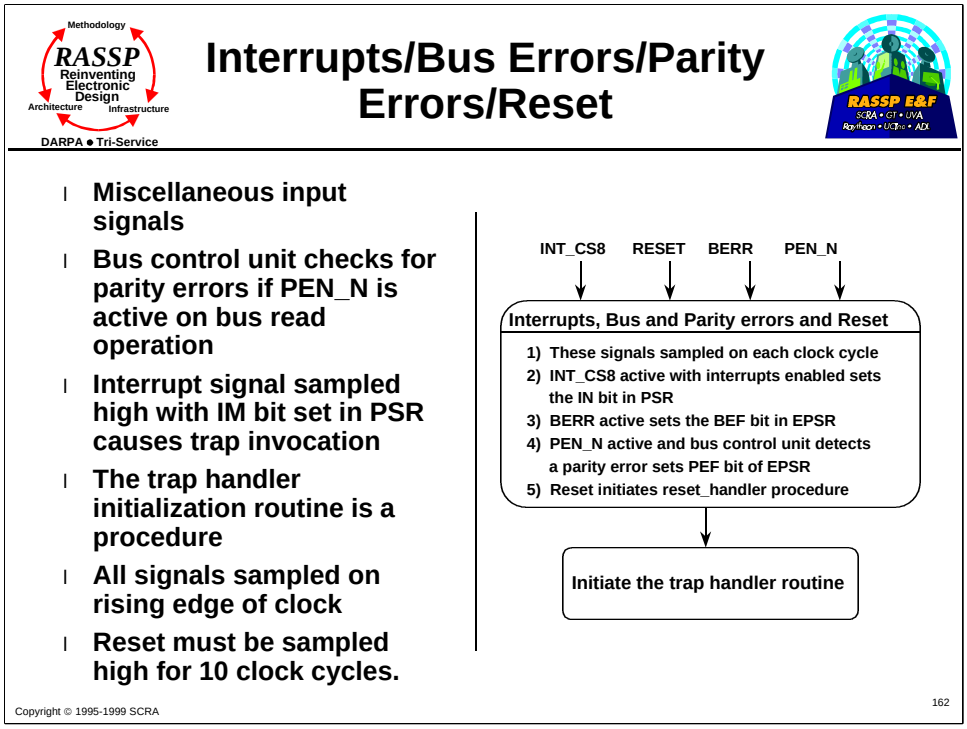


This diagram shows an example of the timing associated with a bus arbitration cycle.

The HOLD and HLDA indicate the handshaking occurring. The controller initiates a HOLD to the processor to indicate it wants control. Because at that time the processor was not doing any address-related operations, it issued a HLDA (Hold acknowledge). At this point the DMA could start driving the address lines and do the DMA transfer across the VME bus. In this case it is doing a quad byte block transfer set up by the configuration words sent to it prior to this diagram. When the DMA is complete, the controller deasserts its HOLD signal and, on the next rising edge of the clock, the HLDA signal is deasserted by the processor.

Not shown is the occurrence of a cache miss when the external DMA controller had access to the memory. The cache miss had to wait until the HLDA was invalidated before performing its code read.

At this point the cache fill can continue.



There are a number of miscellaneous signals that serve a specific purpose. These include the ones mentioned above.

The bus control unit checks for parity errors if PEN_N is active on a bus read operation. This functionality is contained in the decode/execute process when the data arrives from the bus and is placed in the register file. It is sampled on the rising edge of each clock.

The interrupt signal is sampled on each rising edge of the clock and is also a part of the decode/execute process. If the IM bit is set in the processor status register, then a trap invocation is started.

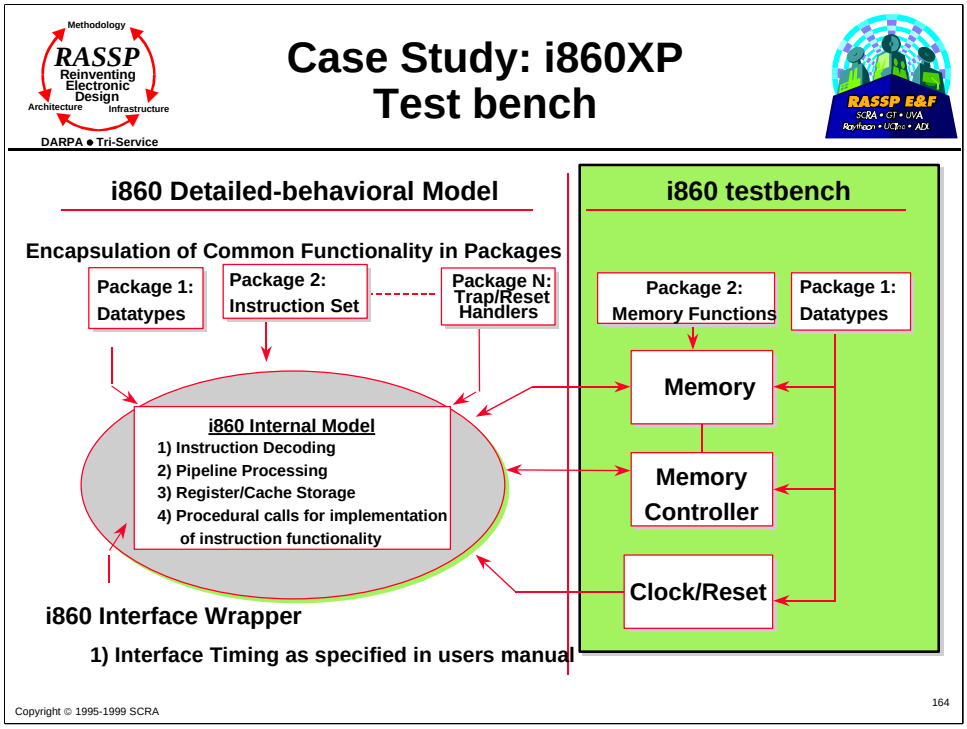
The BERR bit is also sampled on each clock cycle and, if set, the trap invocation is entered.

RESET must be sampled high for at least 10 clock cycles before it activates the reset handling procedure.

Section Outline

- **The testbench**
  - **Clock/reset generator**
  - **Memory controller**
  - **Memory**
- **Testing the i860XP**
- **Results**

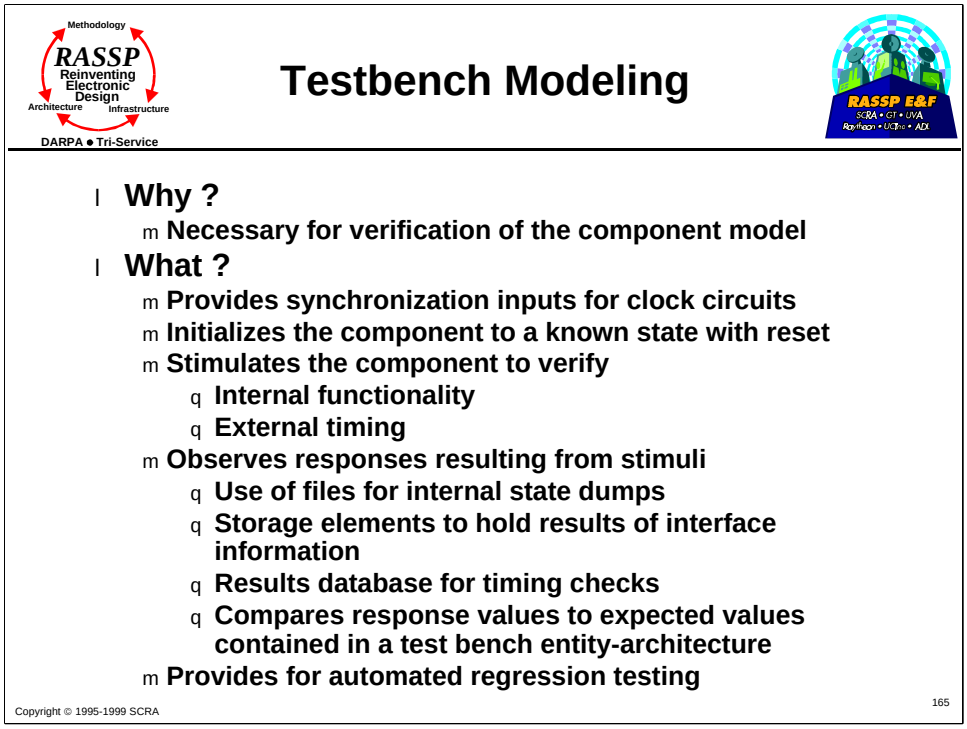
- **Testing the MCV9**
- **Testing the IRST**
- **Creating a DMA to the VME in the memory controller**
- **Simulation results**



This diagram outlines the i860XP component model. It is composed of the i860XP design unit and a testbench. The i860XP is represented as a fully functional model.

- The internal model is at the behavioral level.
- The interface model, which accurately models the timing information at the interface, has hooks to the internal model.
- Packages used for functionality encapsulation:
  - Datatype/Conversion
  - Instruction Set Implementation
  - Trap/Reset/Interrupt/Exception Handling
  - IEEE Standard 754 Floating Point Math.

The test bench consists of a memory, a memory controller, a clock and reset generator for synchronization, and assorted packages to encapsulate specific functionality related to each element. The test bench also contains the test program which is stored in files and read into memory as needed.



The component models testbench is necessary for the verification of the model. The memory controller and memory are part of this testbench.

It consists of segments to provide synchronization inputs in the form of a clock, and it initializes the component to a known state with the reset input.

It also stimulates the component to verify the internal functionality of the model under test as well as its external timing behavior.

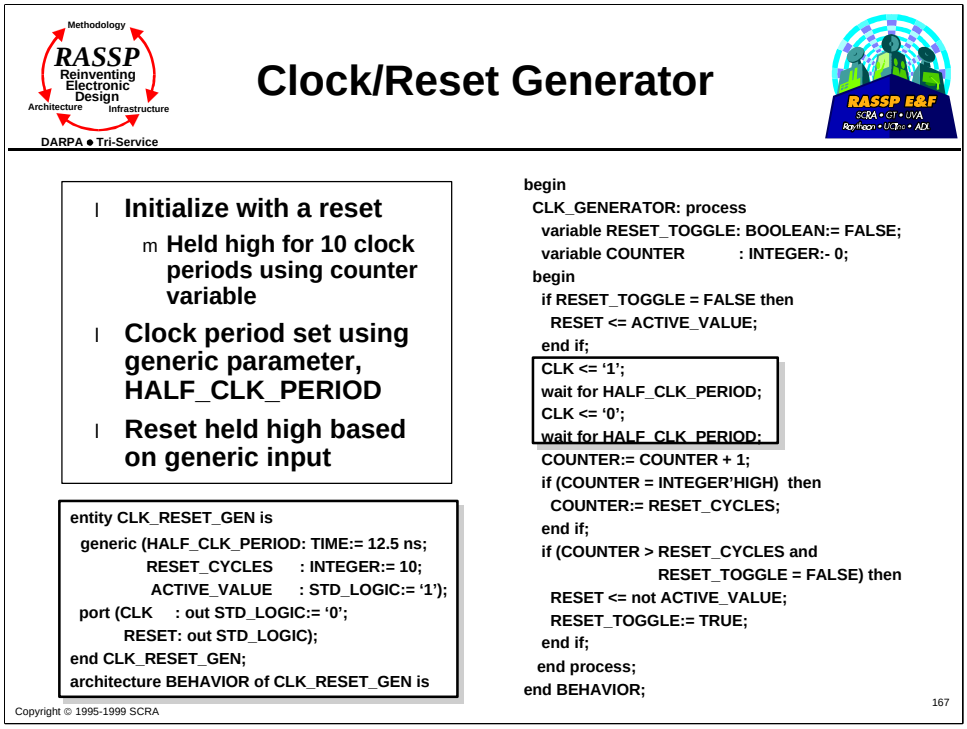
It serves as a monitor of the device under test to compare outputs with expected results. The user of the model should be notified of any deviations from expected responses.

For the i860XP model, files are used to store test programs and are also used to store results for later comparison.

Some form of regression testing should be included to automatically test the component when changes have been made to its functionality.

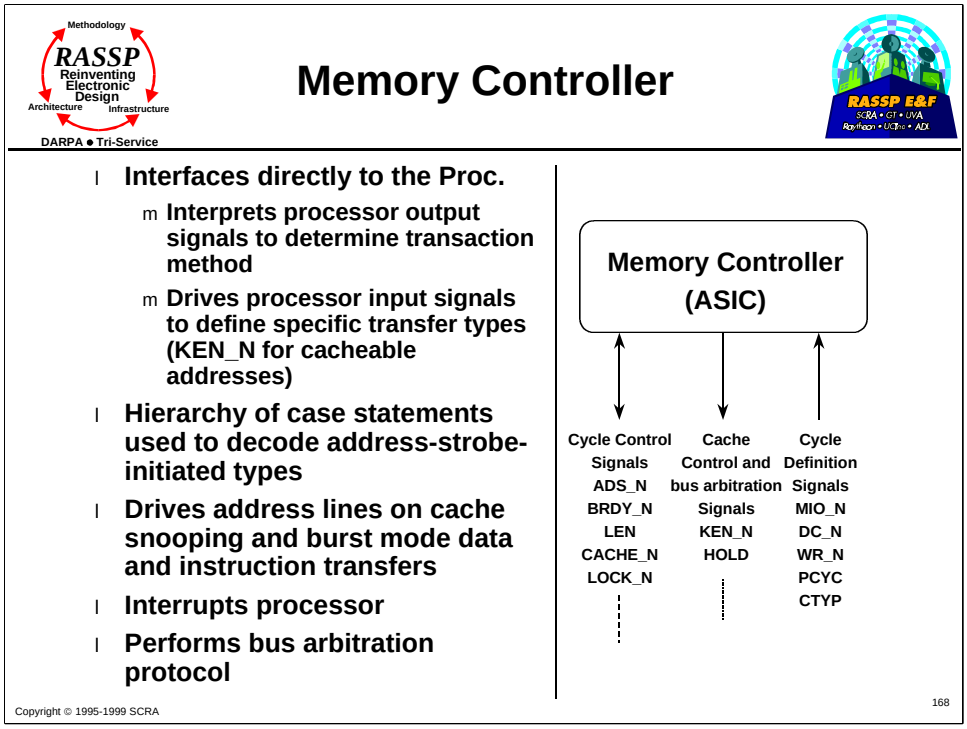
Elements of the Testbench

- **Clock/reset**
  - **Provides the initialization and synchronization**
- **Memory controller**
  - **Provides handshaking and special signals to the processor**
- **Memory**
  - **Stores results and provides test program**
- **Test program**
  - **Test code compiled from high-level language**
  - **Special instructions can also be used as input to the memory and memory controller for monitoring and dumping of specific ranges to a file for viewing**



The clock/reset generator consists of the code above and is reusable across all processor testbenches. The generic clock period time, length of reset (in clock cycles), and its active value are required for simplicity of reuse.

It may be a good move to drive the reset to “Z” because there may be other circuitry that will drive the reset lines. Even better, a special reset process may need to be written that wired-ors a number of lines with the resulting output being the reset state.



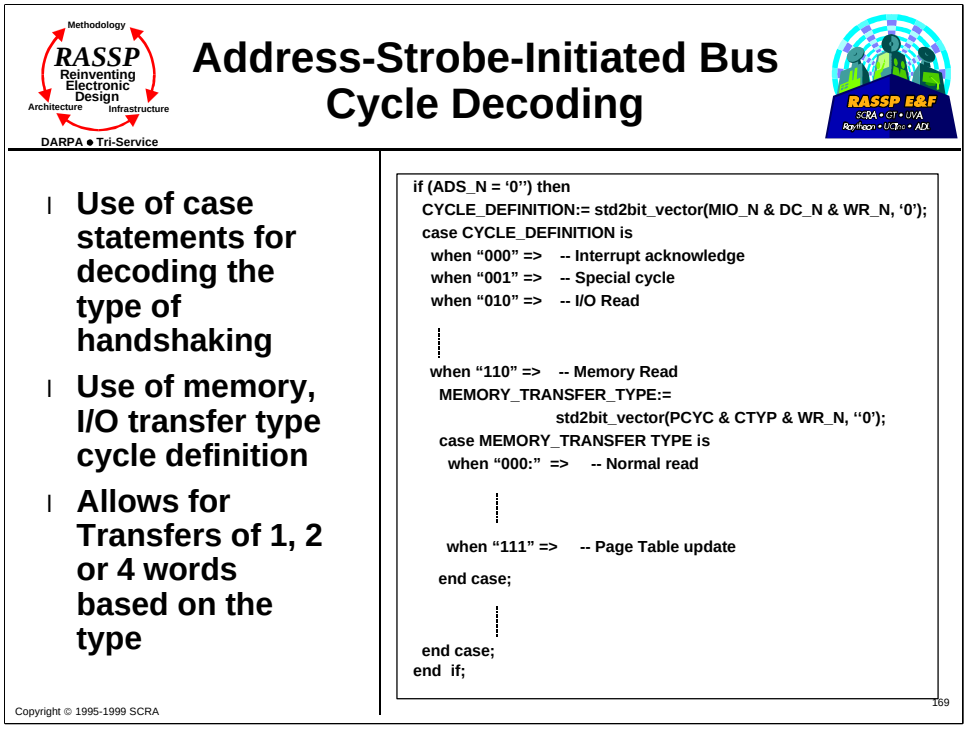
The memory controller interfaces directly to the processor and is required to interpret the processor outputs as well as drive its input lines.

For address-strobe-initiated transactions, there is a hierarchy of case statements used to interpret the important lines from the processor and determine the type of transaction to perform.

In some cases, it needs to drive the address lines; for example, in the case of cache fill cycles and cache snooping.

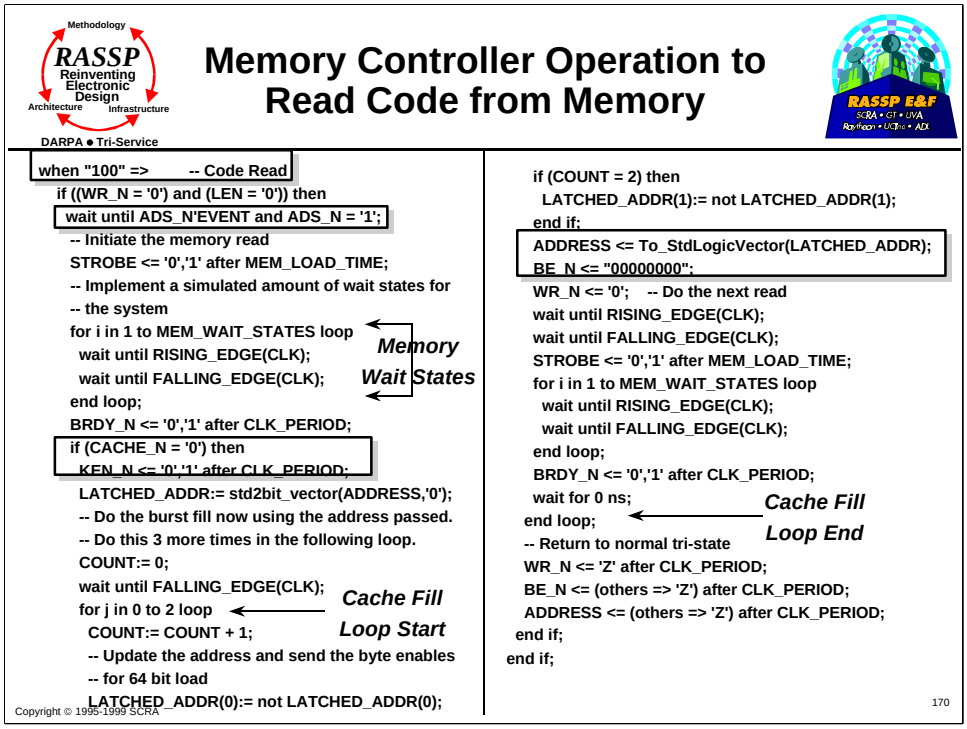
It must also send special signals such as interrupt to the processor to verify its correction response.

It must also perform the handshaking on bus arbitration cycles.



The address-strobe-initiated bus cycle transactions depend on ADS_N going low. This segment of code shows the decoding of the type of transaction based on the signals MIO_N, DC_N, WR_N, PCYC, and CTYP. A case statement is used to structure the decoding section. At the leaf of each decoding tree, the handshaking protocol is performed.

There are 3 transfer lengths, 1 2, and 4 cycles.



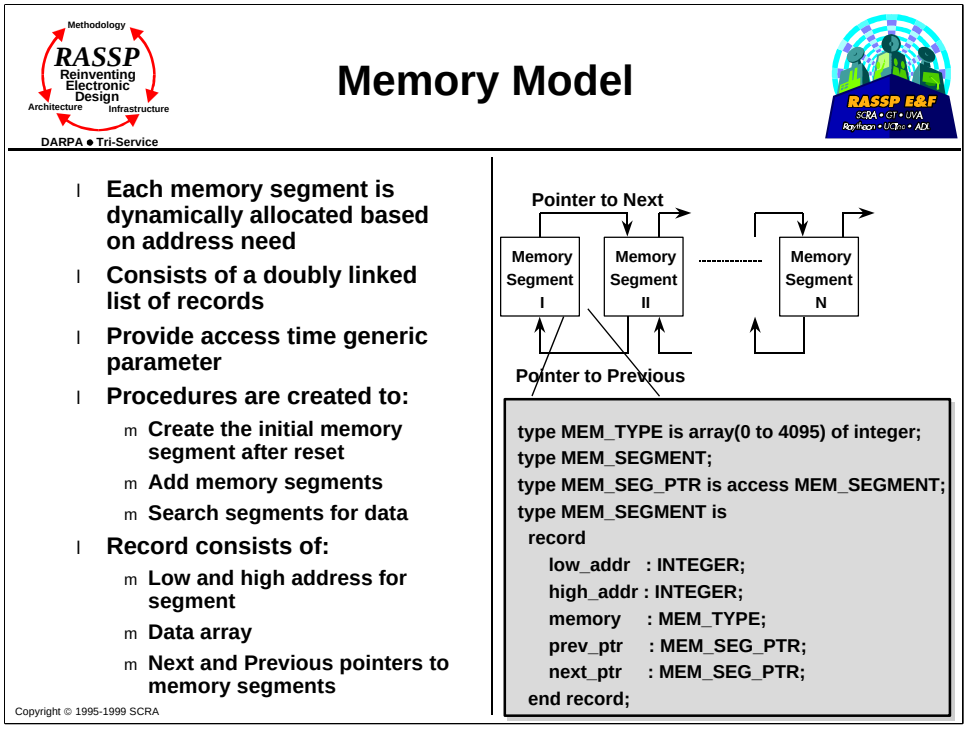
This represents a code read operation. When CYCLE_DEFINITION on the previous slide is “100”, we enter this segment of code.

It starts as a non-cacheable 64 bit operation with burst length determined by CACHE_N and KEN_N. If CACHE_N is set, then memory controller will set KEN_N. This will cause a BURST fill of the cache based on whether the processor wants the data to be cacheable or not.

If CACHE_N is not set the KEN_N will also not be set.

The following is done by this section of code:

- Wait for the address strobe to be “0”
- Strobe the memory for data.
- Wait any specified number of clock cycle delays based on memory access time.
- After waiting a specified number of cycles, the data should be ready. At this point send BRDY_N back to the processor.
- If the CACHE_N line is active, then the memory controller will turn the transaction into a cache fill. In this case KEN_N is set, and the same procedure as mentioned in the previous lines is repeated until three additional data items are loaded by the processor.
- The address is now driven by the controller, not the processor.



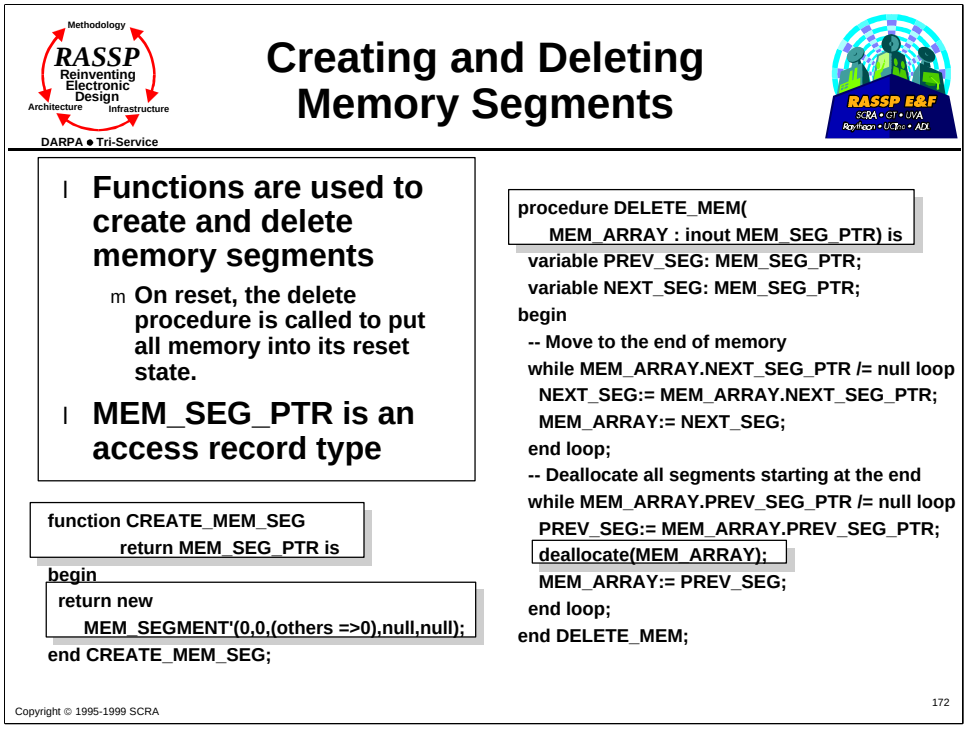
Memory is created using access types in VHDL. A doubly linked list is created to tie groups of memory segments together, and addressing of memory is based on the need to search the segments for the correct address if it exists. If it does not exist, a new segment is added.

Each memory segment was implemented using a record type with: integer fields for low and high address of the segment, an array field for storing the data or instructions for that segment, and two pointer fields to point to the next and previous segments of memory.

The segments are dynamically allocated based on the addresses presented to the input of the memory entity. The code describing the memory data type is shown on the right hand side.

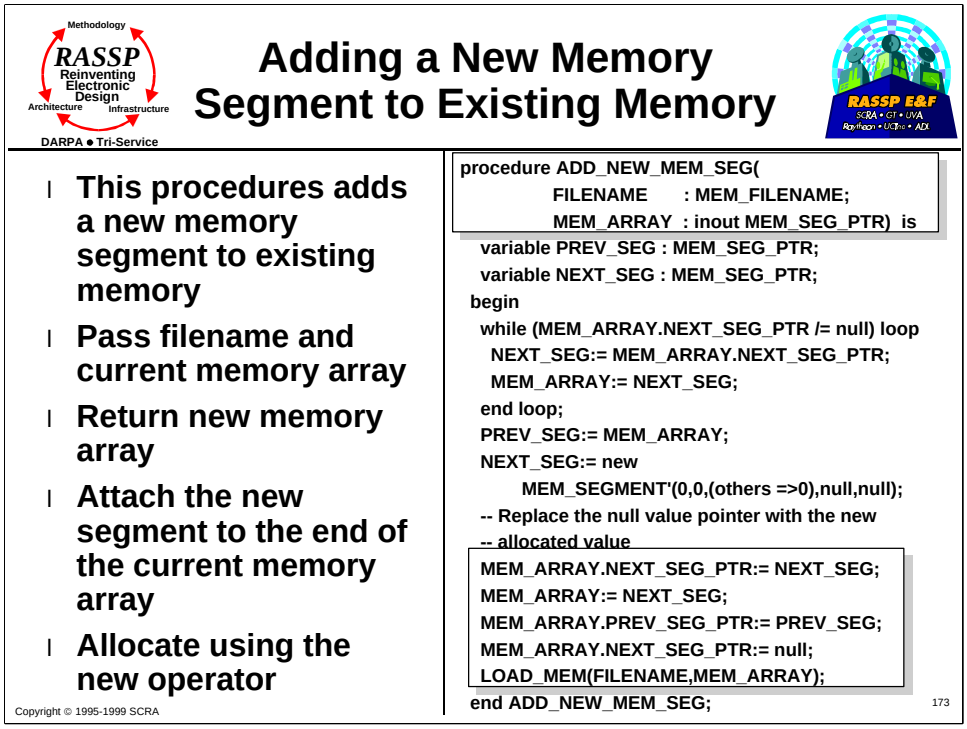
Access time to the memory entity was set by a generic parameter.

Procedures were created for performing various memory functions including: creating the initial memory segment, adding a new memory segment, and searching segments for data.



This slide shows two routines for handling memory segments. The first creates a memory segment, and the second deletes memory (used after reset). On reset, if the memory already exists, then the previous memory is deleted before creating an initial memory segment using CREATE_MEM_SEG.

The record type MEM_SEG_PTR is an access type. When deleting memory, each of the segments previously allocated must be deleted starting at the end of the linked list.

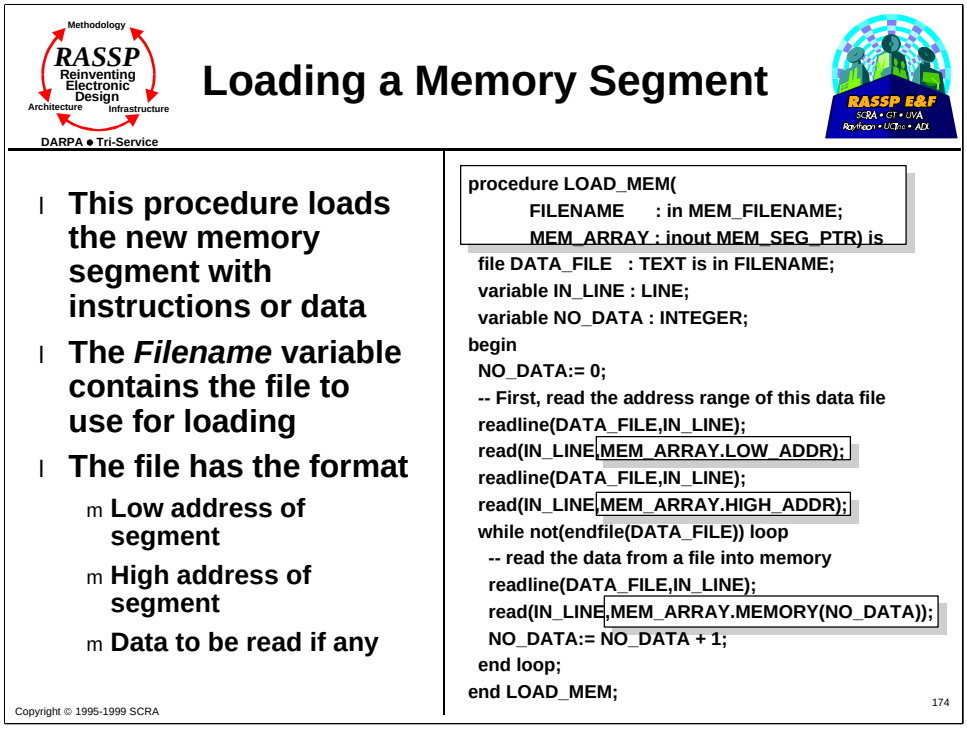


This slide shows how a new memory segment is added to an existing memory array. The filename to load the new information is passed to the procedure along with the current memory array. The updated memory array is returned.

The new segment is attached to the end of the current memory array by first searching for the first null pointer for the next memory segment. At this point, the pointers for the new segment are assigned and the new segment is allocated.

Following allocation of the new segment, the memory array is loaded with the file pointed to by the variable filename. The procedure LOAD_MEM is called to load the new segment.

The next slide shows the LOAD_MEM routine.



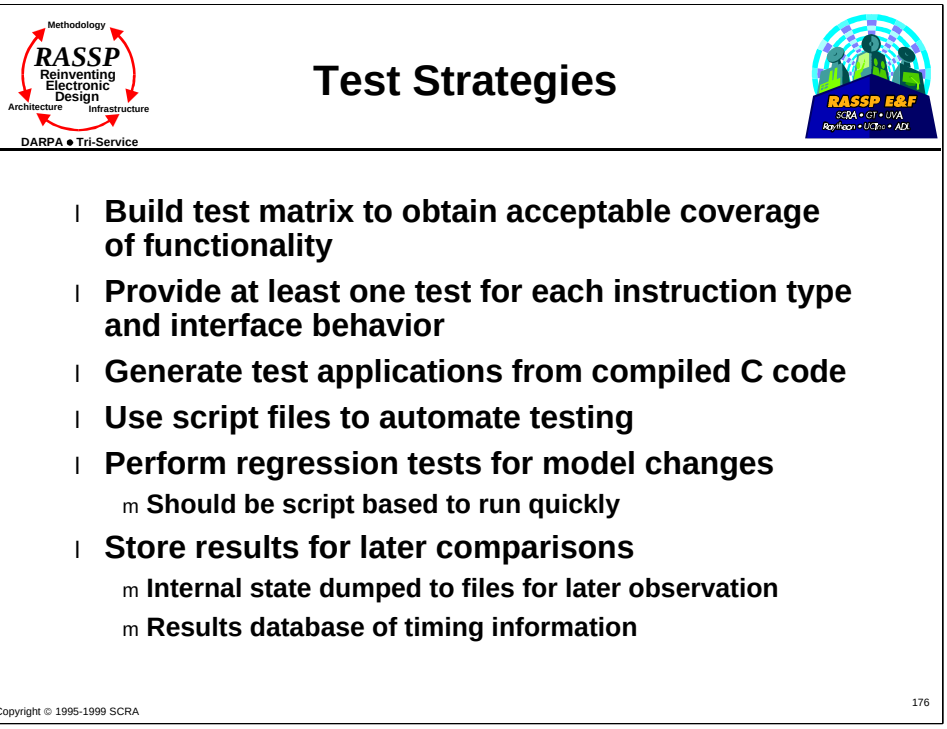
This procedure performs the loading of memory using a specified file containing the data or instructions. The file contains the information to be loaded into the record type. This includes the low and high addresses for the segment along with the data values to be loaded.

The TEXT_IO package in VHDL is used to do the read.

Section Outline

- The testbench
  - Clock/reset generator
  - Memory controller
  - Memory
- **Testing the i860XP**
- Results

- **Testing the MCV9**
- **Testing the IRST**
- **Creating a DMA to the VME in the memory controller**
- **Simulation results**

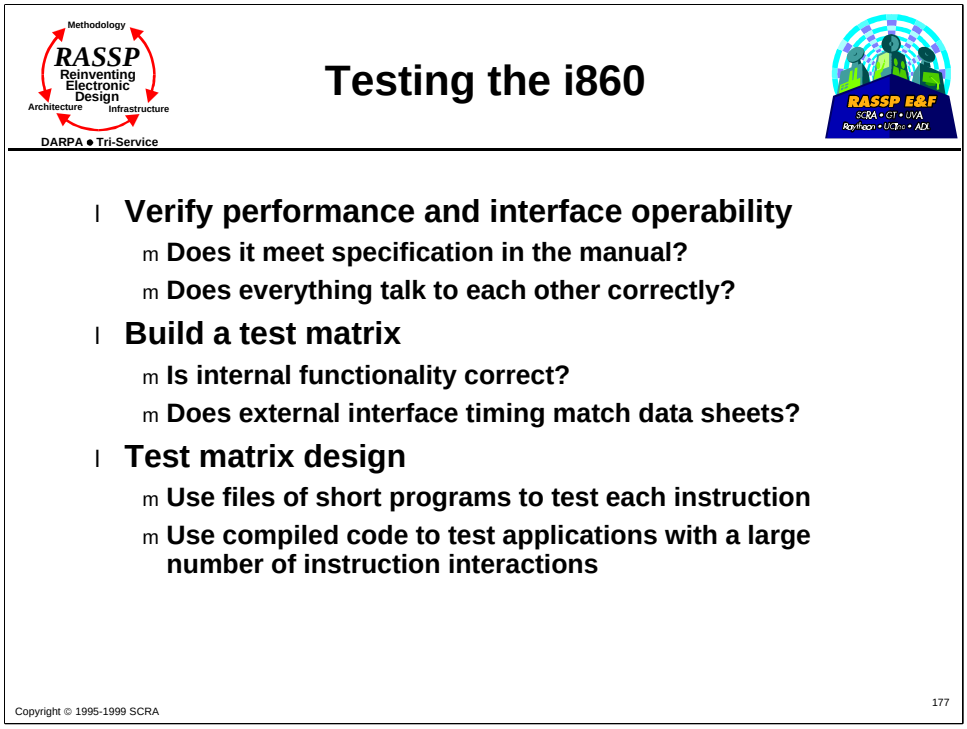


The test strategy should encompass a set of tests to verify the correct functionality of each of the processors' instructions and interface types.

We developed one test for each instruction type and application code to test multiple instructions at a time. The test applications were coded in C or Ada and compiled to the i860 assembly language.

Script files were used to automate the testing and perform regression tests when changes were made to the model. Script files allow the quick verification of previously run tests to ensure that changes did not affect any other parts of the model

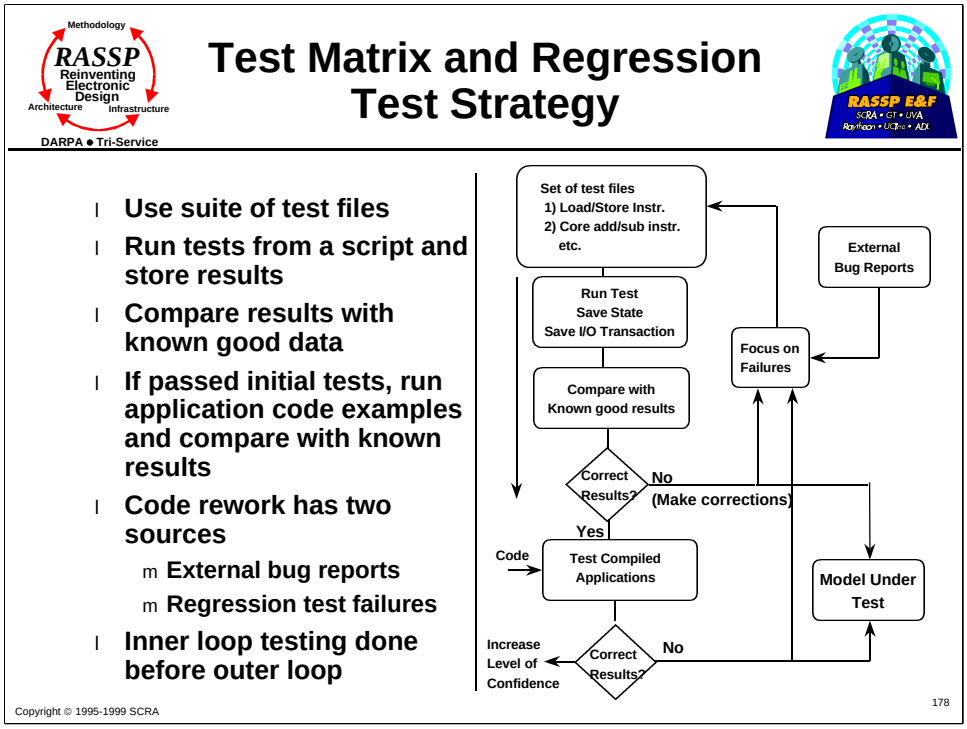
Results are stored in files. The internal state is stored in a file when a special instruction is implemented. The timing information is captured by running the simulator with all the signals archived. The timing information is stored in a results database and compared to known good results at the end of the run.



The reason for testing the processor model is to verify the performance compared to the expected information contained in the users manual. The components of the testbench must interface correctly to the i860 model in order to do this verification.

A test matrix was generated to cover the complete instruction set's internal functional behavior. The test matrix also tests all the interface protocols specified in the user's manual.

The test matrix consists of a set of short files for testing each instruction and some larger application test programs to test the use of a large number of instructions created by a compiler.

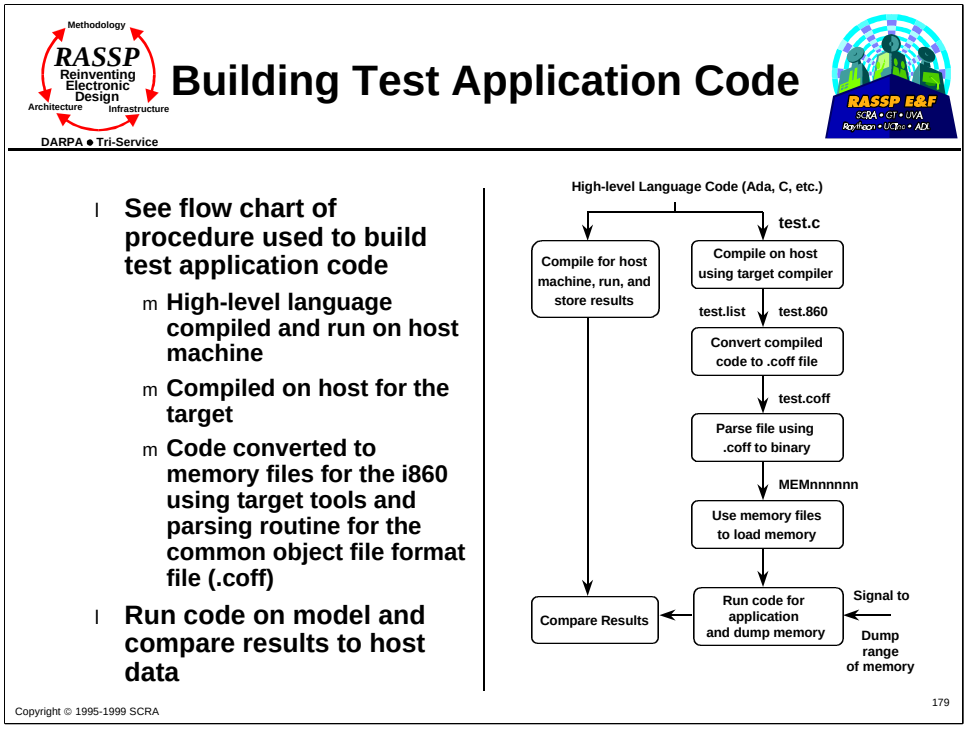


This is the general method used for testing the i860XP processor. The procedure is mostly automated, using script files.

The set of tests is run under two conditions: first, when changes are made to the model and second, when an external bug report is entered.

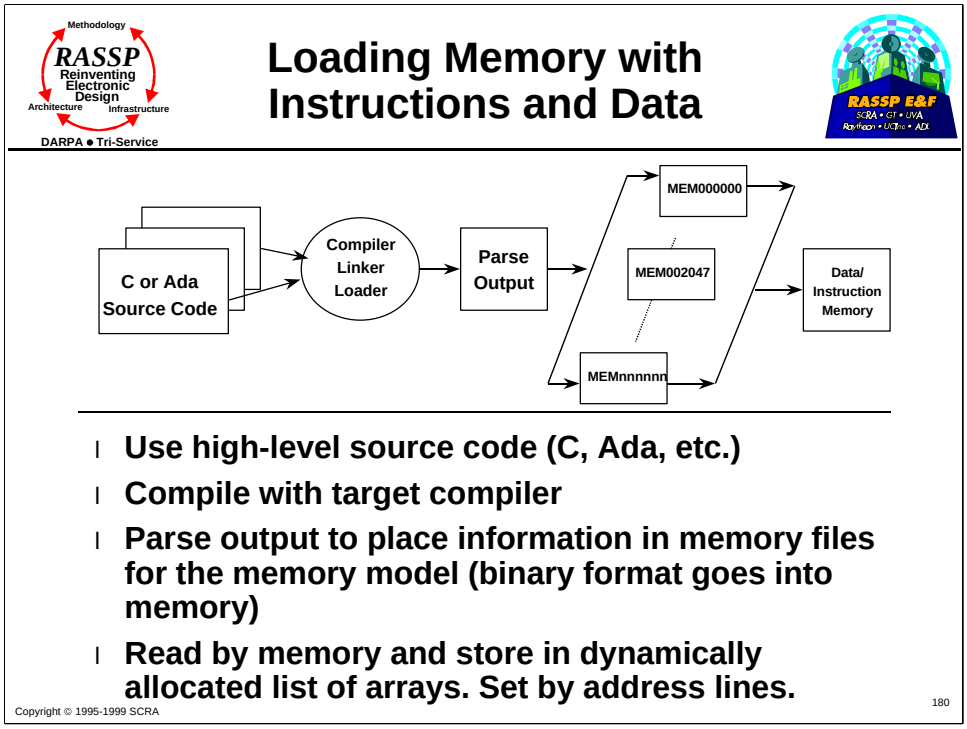
The tests are begun by running the suite of small test files of each instruction. The results are saved in files and results databases. They are then compared to stored good results and, if there is an error, the loop is continued. Errors can occur based on the internal state of the device or on the timing characteristics of its interface. If they pass this test, the large application programs are run and the results are compared with known good results. Again, if there is an error, the model is modified and the tests start again at the smaller tests.

The test is finished when both loop tests complete without error.



The application code was built using a high-level language such as C or Ada. It was compiled on the host for the host and the target. The host was used to compare results based on the precision of the data (single, double, etc.).

The target code was run on the model by first converting the compiled code to a common object file format (.coff) file. This file was parsed using a program developed in C. This program was used to search for the hex memory dump and convert it to the format of the memory files for the processor. Once in the memory file format, the code and data could then be run on the model.



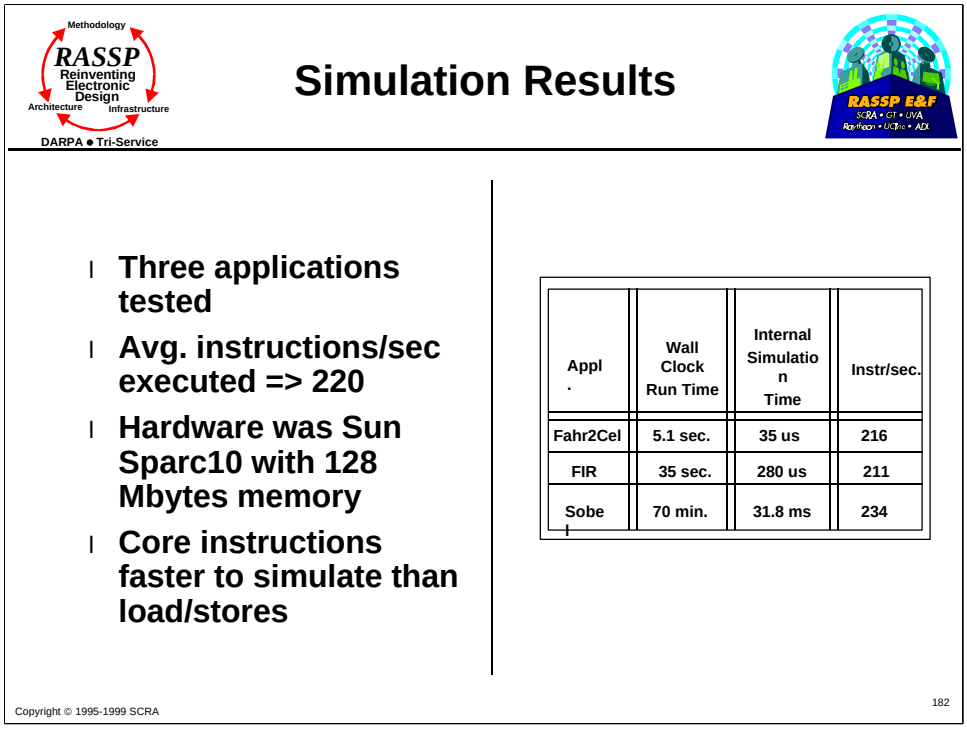
This shows the method used to place the data into the format for each memory file. The high-level language was compiled for the target i860XP and parsed by program to extract the correct information from the .coff file and place them into the correct memory files.

The naming convention for the memory files used the following format:

- The files were named MEMnnnnnn where the nnnnnn was a number from 000000 to 262143. Each file contained 4096 integers, which represented 16364 bytes of data. If an address was 3046, then it was contained in memory file MEM000000. If it was byte address 33497108 in integer, then, because the files are word addresses, we need to divide by 4 to get 8374277, which is the word address. To get the filename, we then divide by 4096 to get 2047. The memory file where the data will be stored is then MEM002047. All addresses were assumed to be word addresses and, if specific bytes were required, then this was made possible by converting the integer to bits and obtaining the correct byte.
- When data or instructions were read into memory, these were done so by creating the memory segments dynamically.

Section Outline

- The testbench
  - Clock/reset generator
  - Memory controller
  - Memory
- Testing the i860XP
- **Results**
- Testing the MCV9
- Testing the IRST
- Creating a DMA to the VME in the memory controller
- Simulation results



Three test applications were used to measure the performance of the i860XP model. These included a small C program to convert Fahrenheit to Celsius and back again, an FIR filter, and a program to do the Sobel edge detection algorithm on an 128x128 pixel image.

It was found that the code ran approximately 220 instructions/sec on a Sun Sparc 10 workstation with 128 Mbytes of memory. The Sobel algorithm took 70 minutes to run because of the amount of data it needed to process.

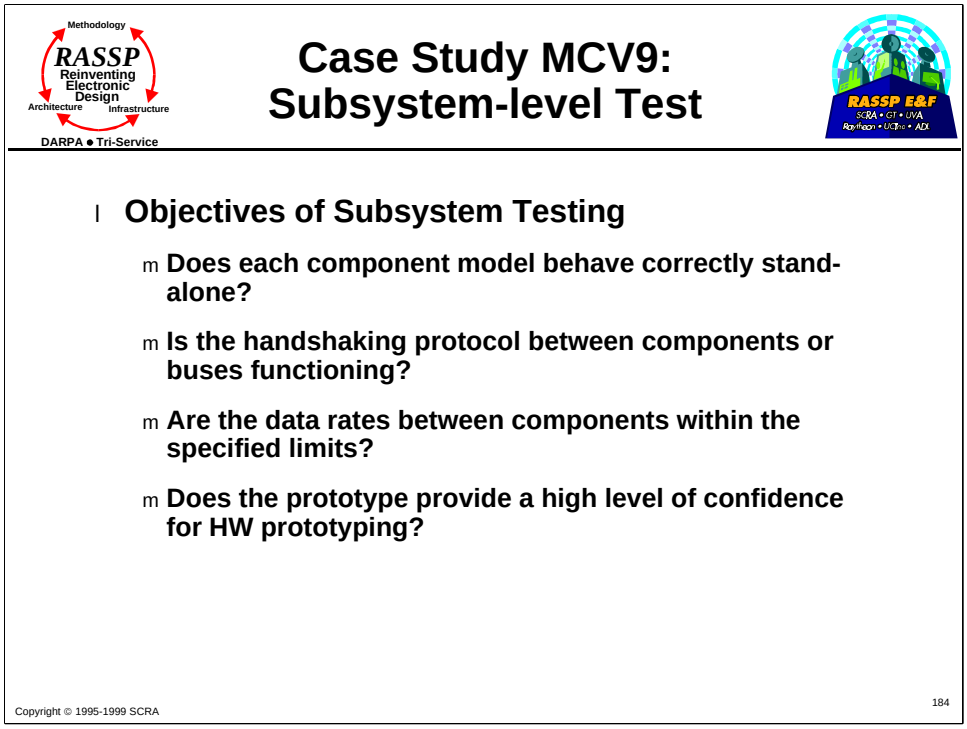
From these results, the core instructions, on the average, ran slightly faster than the floating point instructions. This is probably caused by the lower complexity in computation required to process these types of instructions.

Section Outline

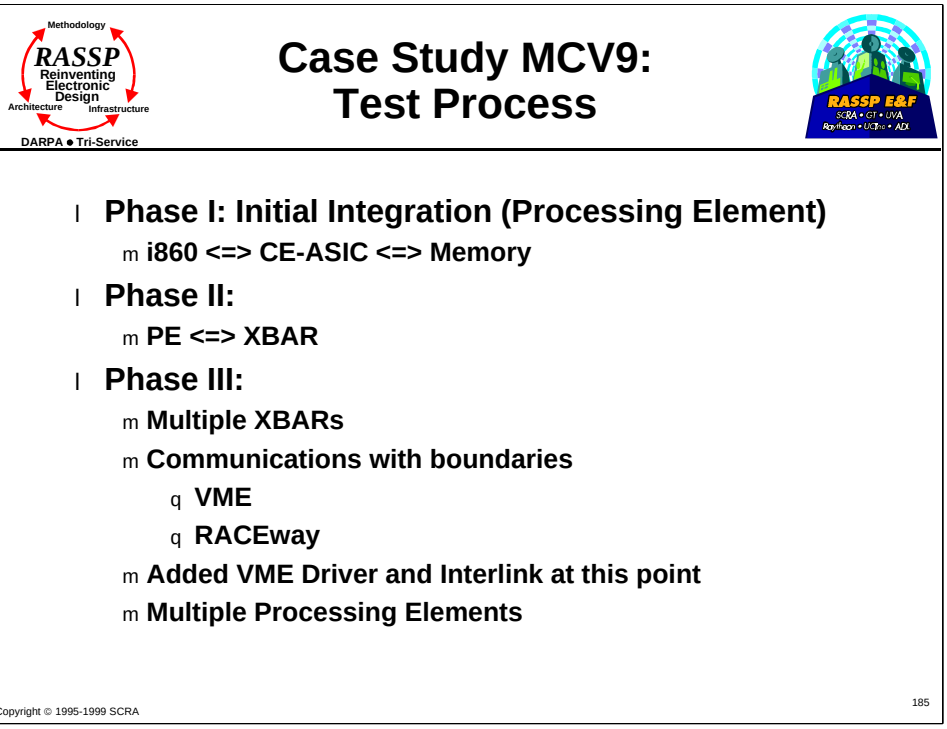
- The testbench
  - Clock/reset generator
  - Memory controller
  - Memory
- Testing the i860XP
- Results

Testing the MCV9

- Testing the IRST
- Creating a DMA to the VME in the memory controller
- Simulation results



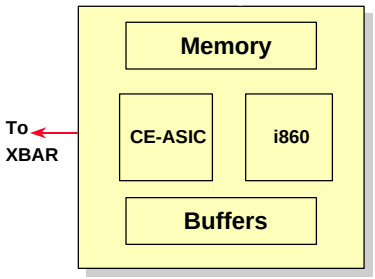
When doing subsystem integration, a plan was required to determine the objectives expected from doing this simulation and when to end. The objectives are listed above. At this level of abstraction the main information to be gathered included whether the components interface correctly and if the data rates between components are at the rate required to meet performance objectives. Lastly, to end simulation runs, one needs to determine how much simulation is enough, and this is based on the degree of confidence the designers have as to whether the actual HW will work based on the simulations. Because the MCV9 is a COTS part, there was a high degree of confidence from the beginning, and detailed simulations were not required. It was sufficient to have the units talk together correctly and have the data was sent across the crossbar network in the correct time.



Various tests were used to verify the integration of the components. These were done in phases and were part of a test and integration plan. The first phase tested the processing element alone, which included the memory, the i860, and the CE-ASIC, along with some buffer registers. Phase II attached the processing element to a single XBAR and phase III connected multiple XBARs. Phase III also included tests to write to the interface of the MCV9 (VME and RACEway Interlink).

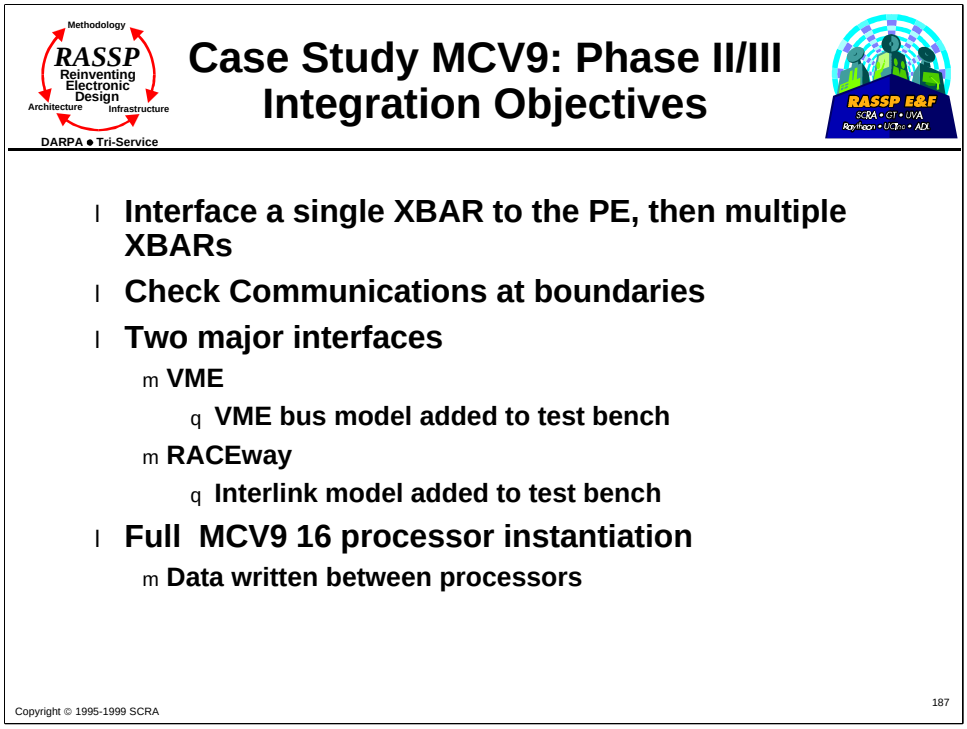
Case Study MCV9: Phase I Integration Objectives

- **Tests at this stage**
  - **Reset**
    - **Did anything reset to an unexpected state?**
  - **CE-ASIC registers**
    - **Can the i860 set key registers in the CE-ASIC?**
  - **Reading/Writing to CE-ASIC**
    - **Is the handshaking logic working between the i860 and the CE-ASIC?**
  - **Memory**
    - **Can we read and write data to the memory?**

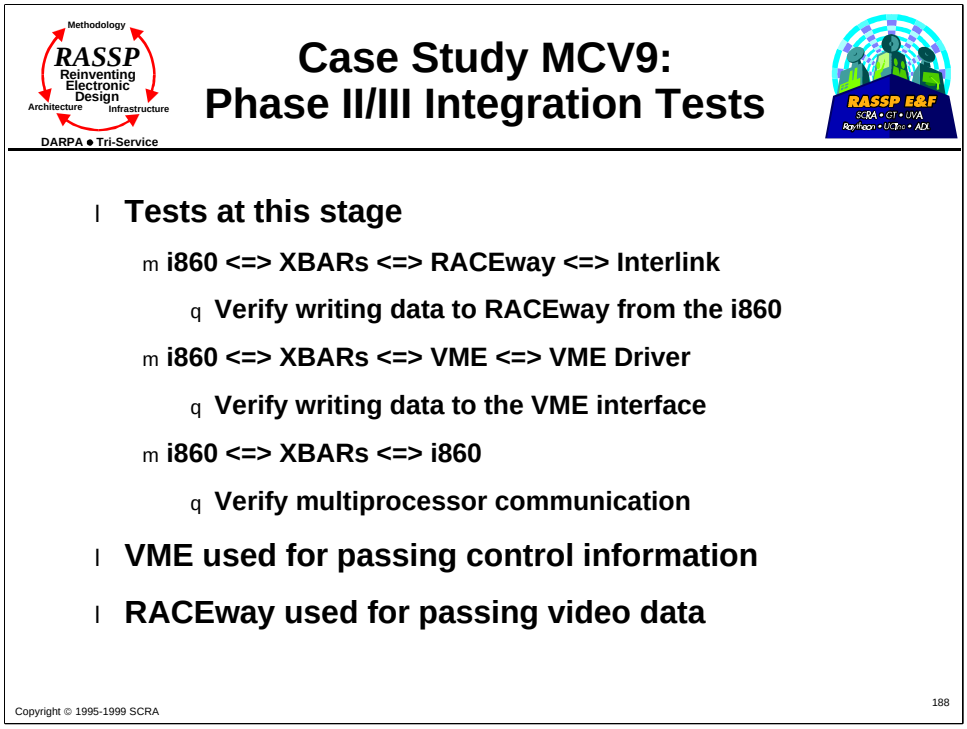


186

The Phase I tests that were run are listed above. Their intent was to test the integration in the processing element of the MCV9. The first test performed was a reset test to verify that all the elements came up in expected states after reset. Once this was verified, various register tests were implemented to test the ability of the processor to set registers in the CE-ASIC. At the same time, the handshaking protocol was verified between the i860 and the CE-ASIC. Finally, reads and writes to memory were tested to verify the interface connection was correct.

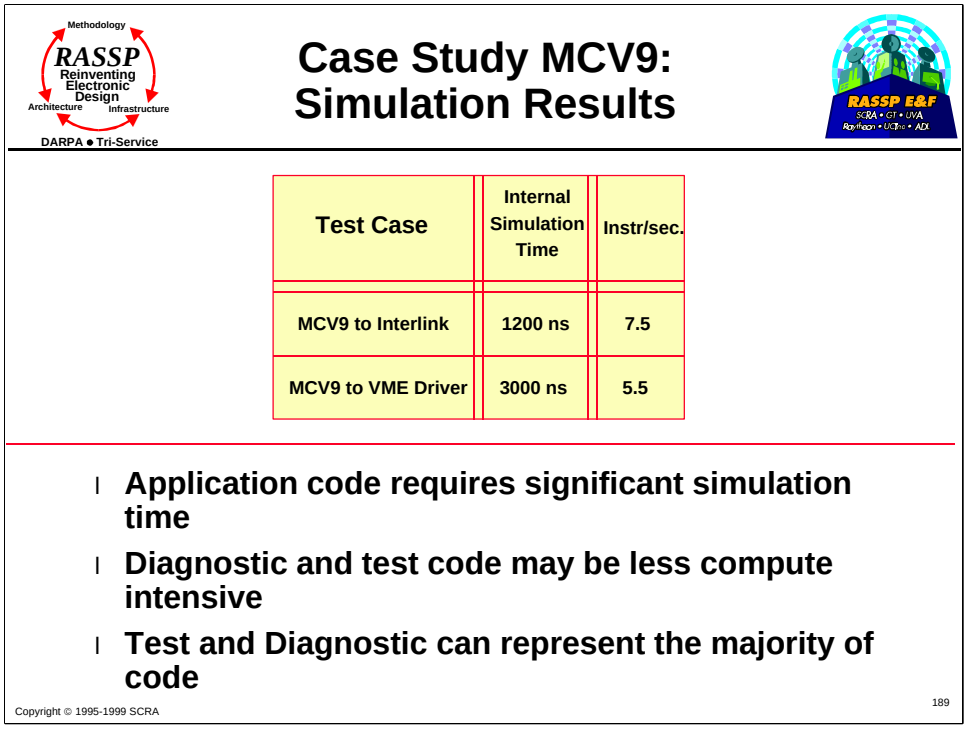


Once it was guaranteed that the processing element was functioning properly, a XBAR, and then multiple XBARs, were added to the model. The handshaking protocol was again verified, and communications were checked to the subsystem boundaries. These included the two major interfaces: the VME and the RACEway interlink. When it was verified that this communication was working properly, a full instantiation of 16 processors was tested on the MCV9. In this case data was written between processors to make sure the routing was working correctly.



The actual tests run at this phase included those listed above.

Control information is passed over the VME bus in the actual system architecture, and video data was passed over the RACEway interlink.



Simulation results were collected for two of the tests. The two tests included the i860-to-interlink data writes and the i860-to-VME writes.

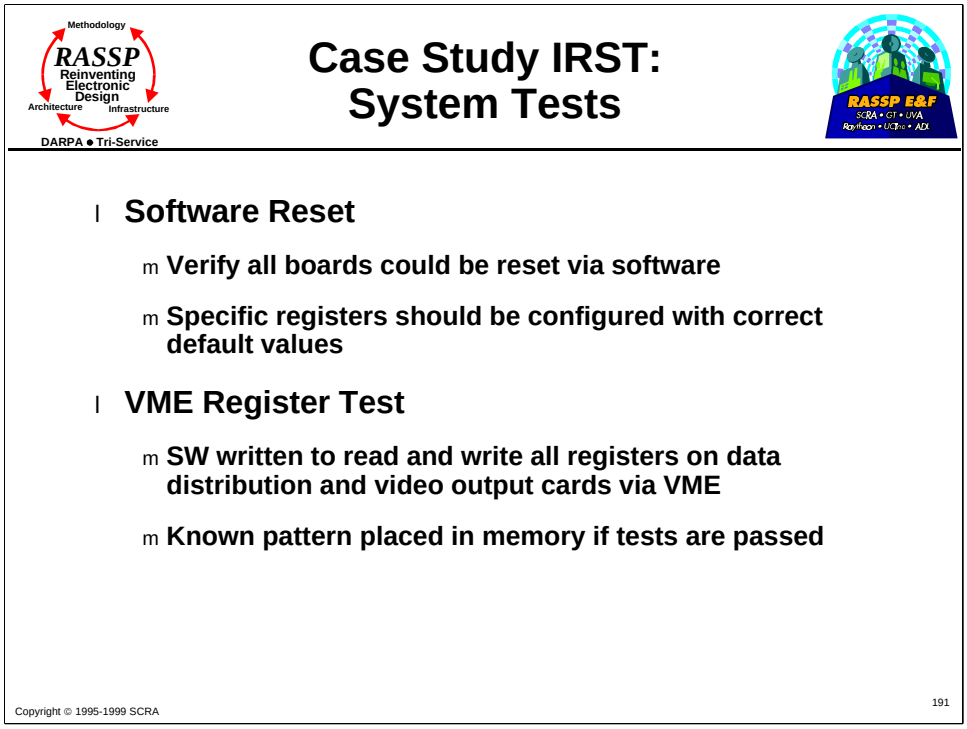
The number of instructions/sec executed with all the additional models added to the subsystem prototype has decreased significantly. This prohibited the running of application code on the virtual prototype.

Application code was not run on the prototype because it required too many computing resources, but test and diagnostic code is a viable candidate for code running on a virtual prototype. In some systems, the test and diagnostic code can represent the majority of the code.

Section Outline

- The testbench
  - Clock/reset generator
  - Memory controller
  - Memory
- Testing the i860XP
- Results

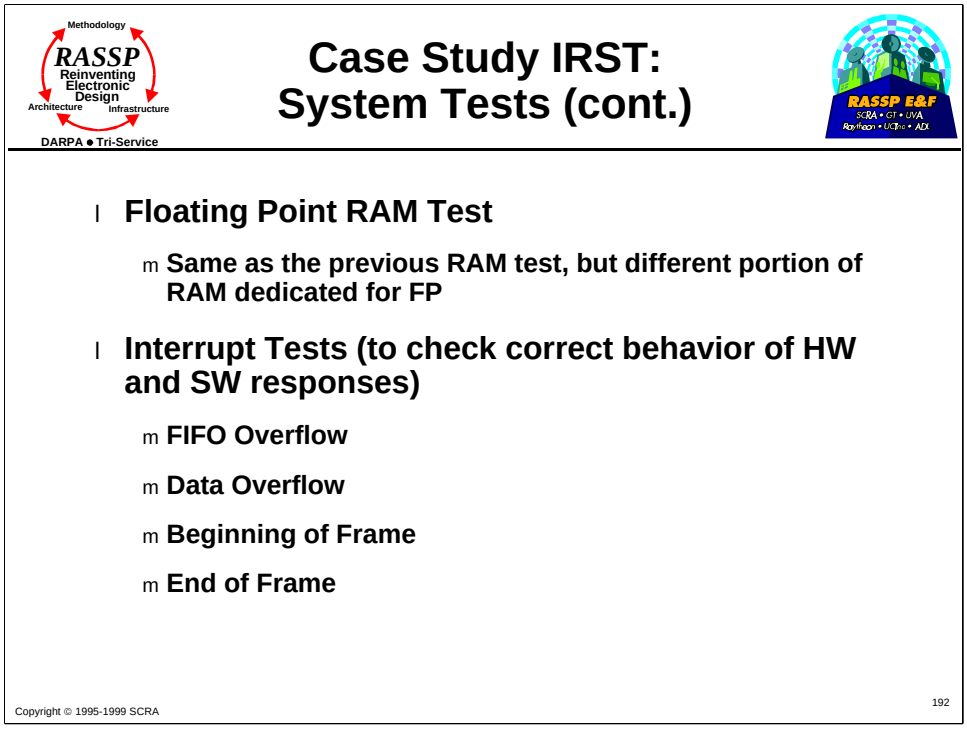
- Testing the MCV9
- **Testing the IRST**
- Creating a DMA to the VME in the memory controller
- Simulation results



This and the following slides describe the types of tests run at the system level to help verify that the system was implemented correctly.

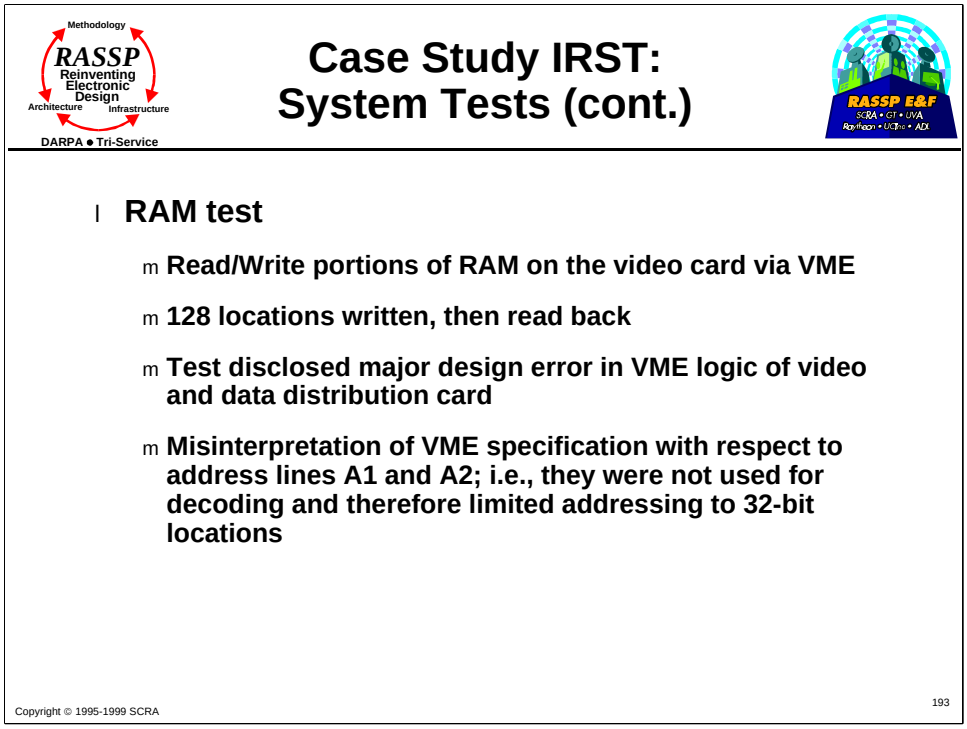
Again, the reset test was the first to be done to verify that everything initializes correctly.

The VME register test was used to verify that registers in the data input and distribution card and video cards could be configured correctly. If the test was passed successfully, then a known pattern was written to memory.

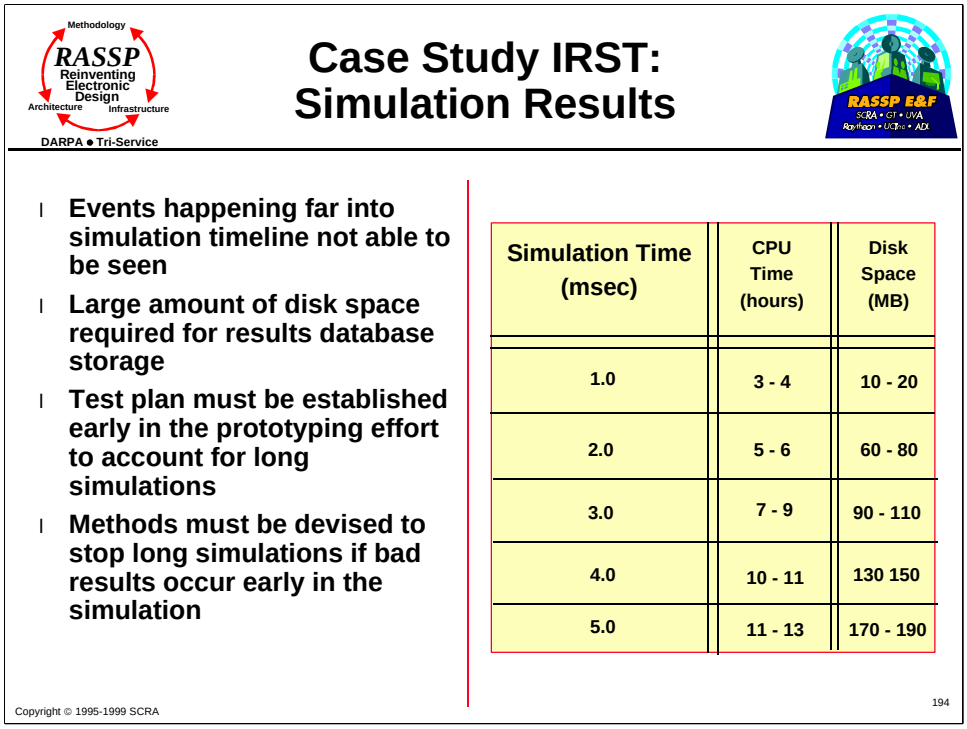


The floating point RAM test was similar to the previous RAM test, but a different memory was verified.

Interrupt tests were important to test both the HW and SW. The HW of the four designed boards could interrupt the processor based on whether their data and FIFO buffers were full. In this case the processor can take the appropriate measures to alleviate the problem. The HW also can send the processor information as to when the frame starts and stops. These tests usually took a long amount of time because some of these events happen much later in the timeline, as the next chart will show.

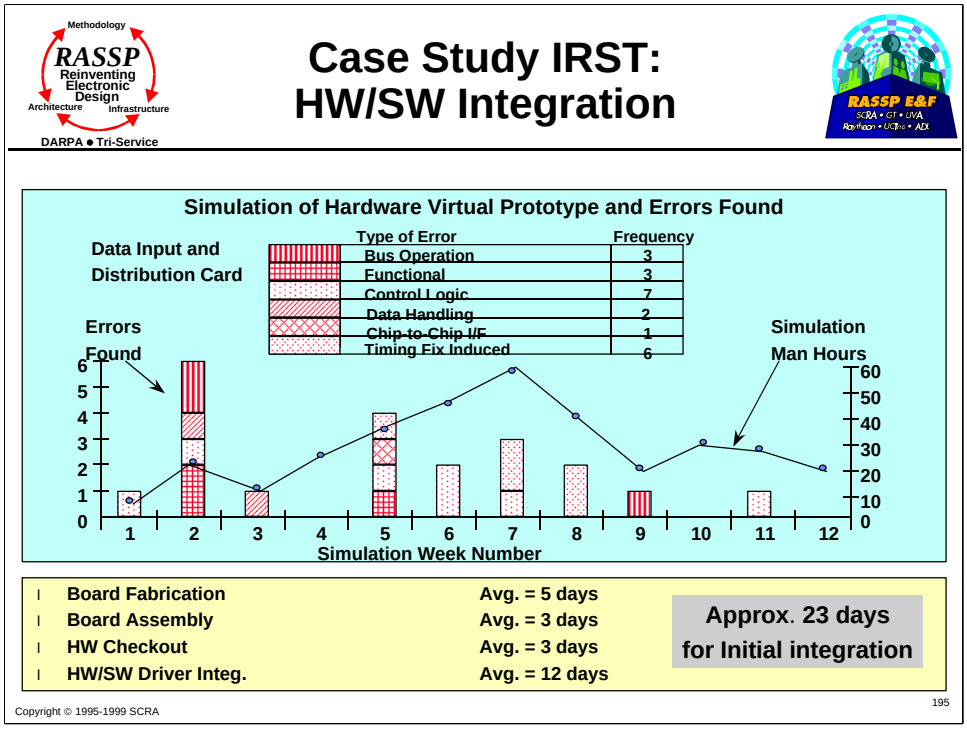


The RAM test was used to read and write portions of memory on the video and the data and input distribution cards via the VME bus. A total of 128 locations were written and read back, and when this was tested an error was found on both cards. The designer misinterpreted the VME specification and did not use address lines A1 and A2 for decoding. This prohibited the use of addressing less than 32-bit locations, and the error was found during this test. This was a significant error that would have required a difficult fix later in the design cycle, but because no HW had been created at the time, the fix was done to the VHDL code. The new code was synthesized again with the fix.

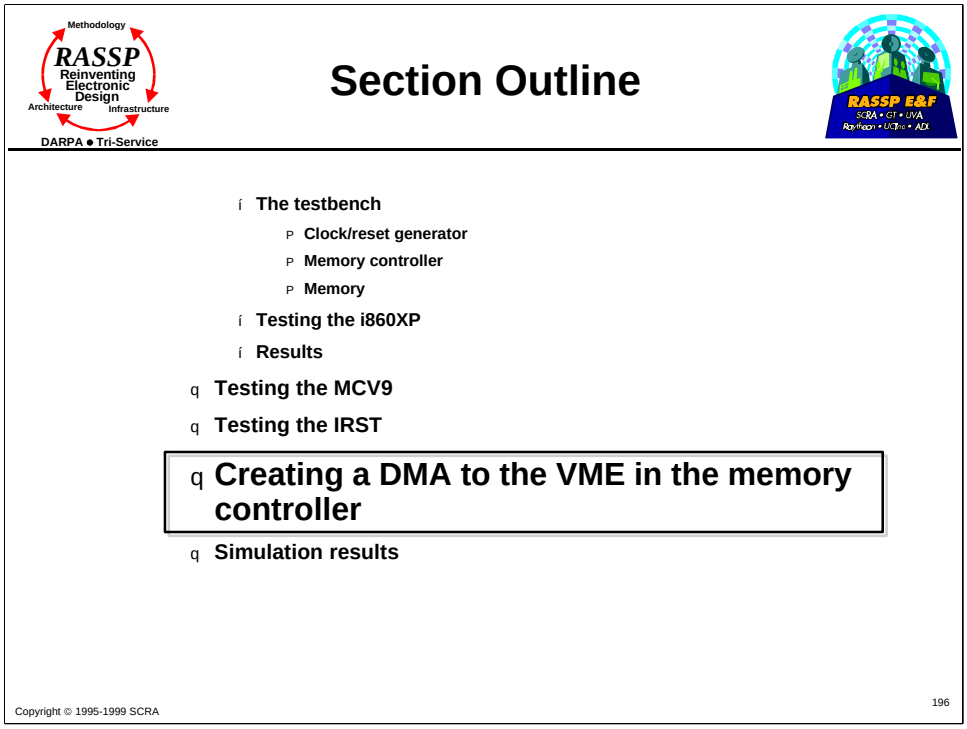


As can be seen from the chart on the right hand side, there is a near-linear relationship between simulation time and amount of CPU time to run the test code. The same applies to the amount of memory required to save the results database.

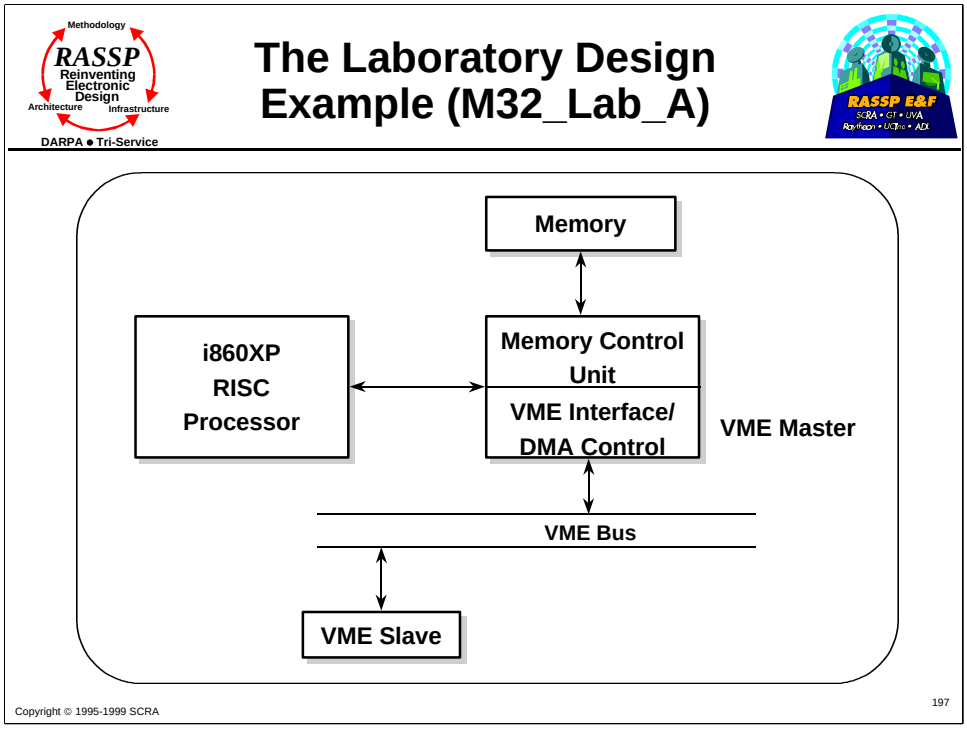
From this information, it is obviously critical that a test plan be devised at the beginning of the modeling effort to account for these long simulations and to institute methods to stop long runs when an error occurs early in the simulation. The test plan must also address the issue of how much simulation is satisfactory before acceptance of the HW design and HW prototyping can begin.



This slide illustrates the types of errors found during early integration and testing using the detailed behavioral virtual prototype in the design process. The errors were found on the data input and distribution card when software was being executed on the processing elements of the MCV9 board. From the figure, it is seen that the errors were tracked over the 12 week period of testing, and as the errors start to decrease, it was determined that the board could go to fabrication with a high degree of confidence in correctness. After fabrication, the final hardware was integrated in 23 days and there were no bugs in the digital hardware that was tested using the virtual prototyping approach.



In this section, the VME bus model for the laboratory example is discussed in more detail. The next slide shows where this model fits into the i860XP/memory/memory controller/DMA/VME board level model. For more details of the actual code, see the lab exercise and the code associated with it.



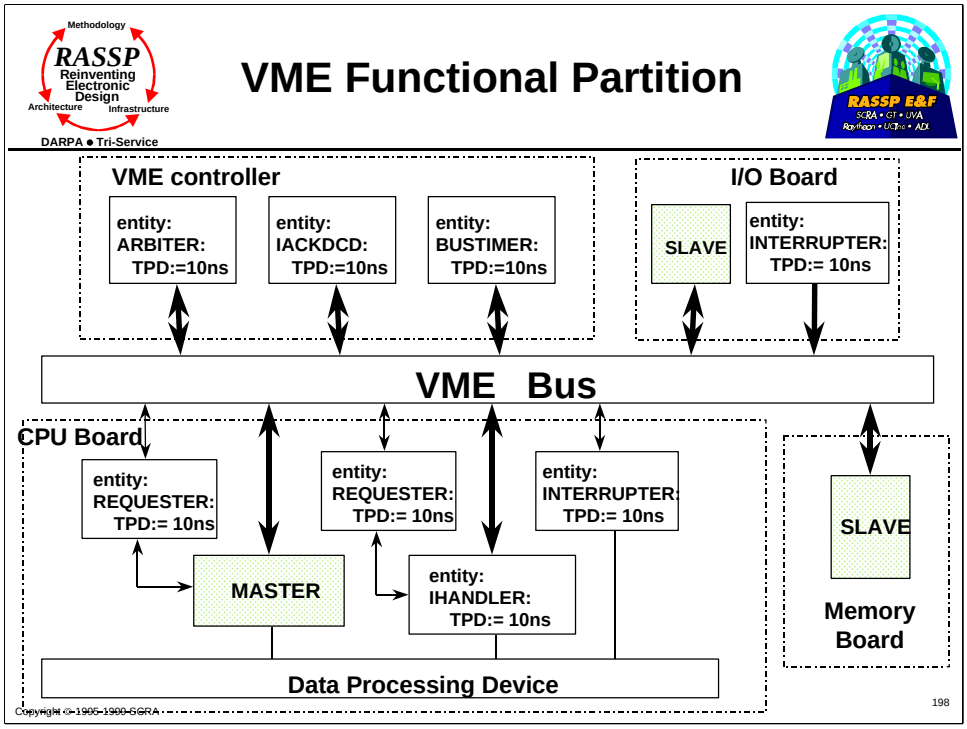
This is a block diagram of the proposed system design. The i860XP processor will be connected with the VME bus through the memory controller unit and a VME interface. When the DMA is configured to do burst writes across the VME bus, it obtains control of the memory from the processor using bus arbitration before doing the transfer.

The i860XP gets control of the VME bus and acts as the master while the slave is configured to write to a file. So any transfers from master to slave involve generating the data in the i860XP and sending it to memory. Following the loading of memory, the i860XP configures the DMA registers to transfer a specific amount of data from memory using the VME handshaking protocol. The data is then written across the bus to the slave address, and the slave places it into a file for later verification.

This laboratory will cover modeling of the dataflow, timing, and control of the major components of a system under design.

The system will consist of an i860XP processor, a memory unit, and a memory controller that interfaces to a VME bus and also contains a DMA controller.

Each of the individual components will be described and their integration and test will be covered for a specific operating mode of the VME.



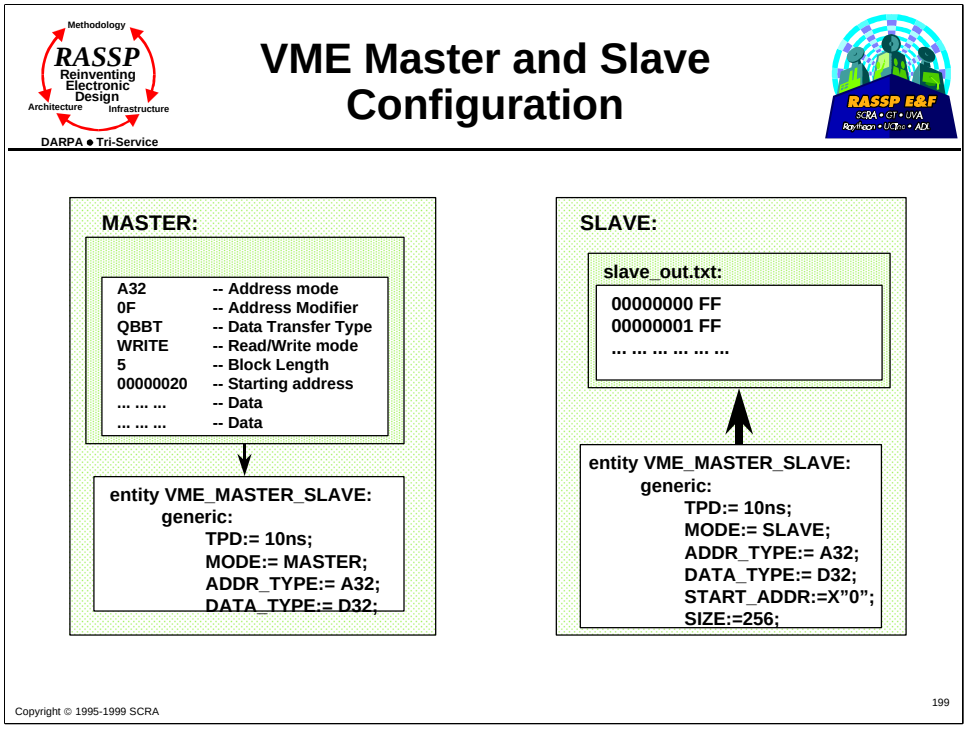
The VME bus functional structure can be divided into four main categories. Each consists of a bus and its functional modules, which work together to perform specific duties. The four main categories are the VME controller, CPU board, memory board, and the I/O board.

Data transfer: Devices transfer data over the Data Transfer Bus (DTB), which contains data and address pathways and associated control signals. Functional modules called masters, slaves, interrupters, and interrupt handlers use the DTB to transfer data between each other. Two other modules, called Bus Timer and IACK Daisy-Chain Driver, also assist them in the process.

DTB arbitration: Because a VME bus system can be configured with more than one Master or Interrupt Handler, a means is provided to transfer control of the DTB between them in an orderly manner and to guarantee that only one master controls the DTB at a given time. The Arbitration Bus modules (Requesters and Arbiter) coordinate the control transfer.

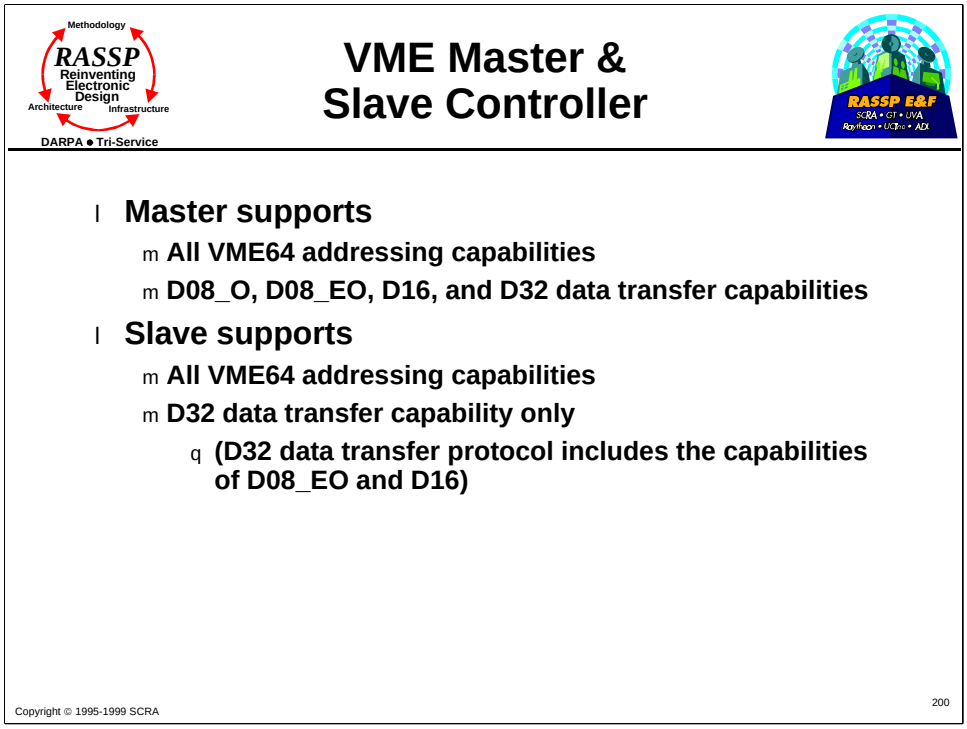
Priority Interrupt: The priority interrupt capability of the VME bus provides a means by which devices can request services from an interrupt handler. These interrupt requests can be prioritized into a maximum of seven levels. Interrupters and interrupt handlers use the Priority Interrupt Bus signal lines.

Utilities: Periodic clocks, initialization, and failure detection are provided by the Utility Bus. It includes a general purpose system clock line, a system reset line, a system fail line, an AC fail line, and two serial lines. Utilities also include power and ground pins.



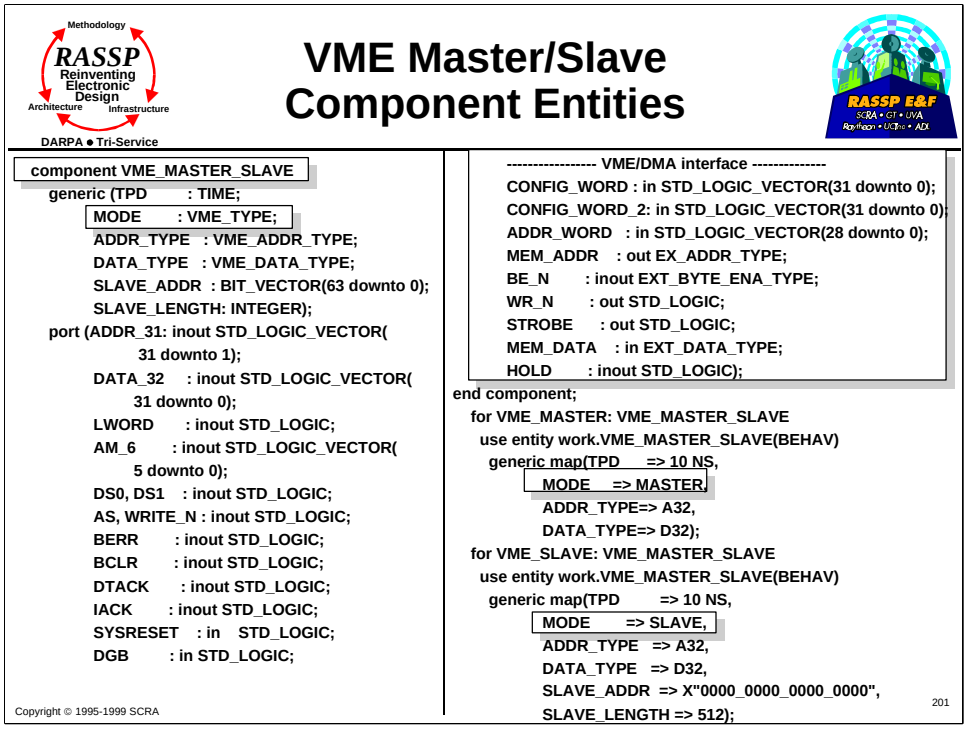
The model takes the above general form.

The master sets up the address mode, modifier type, data transfer type, read/write mode, block length, and starting address. The slave accepts data in the address range and passes it to a file for storage.



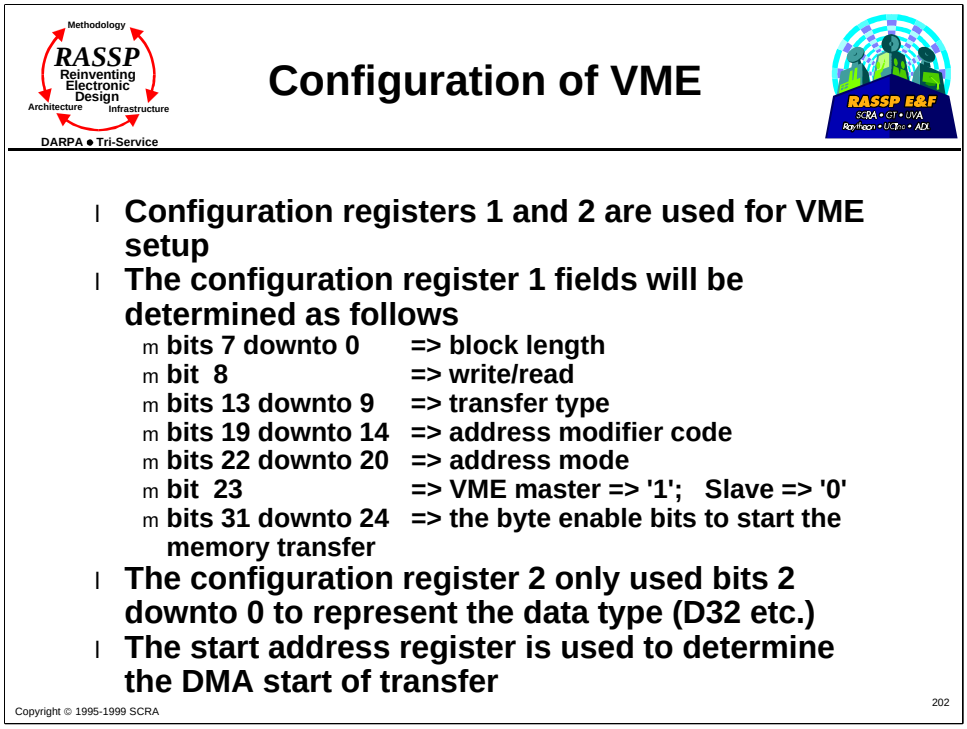
MBLT, MD32, are D64 are modes not supported by the model used in this study.

The mode used for the simulation is the D32 data transfer type.



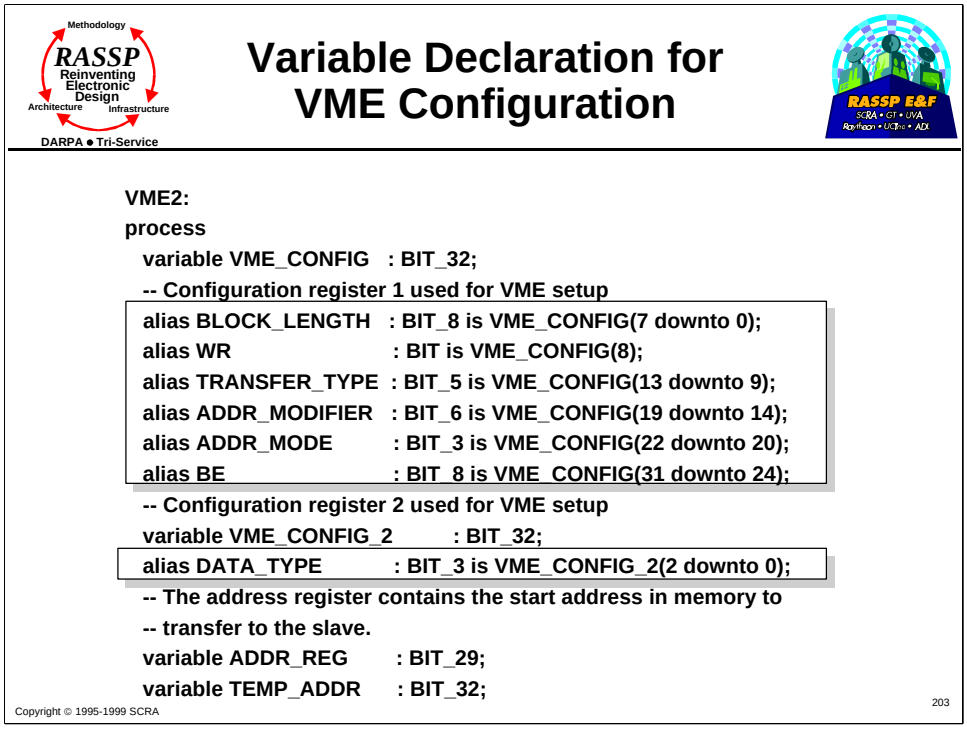
This slide shows the component entities for the VME master and slave. Special to this interface are signals passing configuration word, address word, and DMA handling information. This is shown below the VME/DMA interface comment. This information will be important in interfacing to the i860XP processor through the memory controller. The memory controller has a similar set of lines.

The VME master and slave use the same entity description. The method for differentiating the two is through the use of the generic passed (MODE = MASTER or SLAVE).

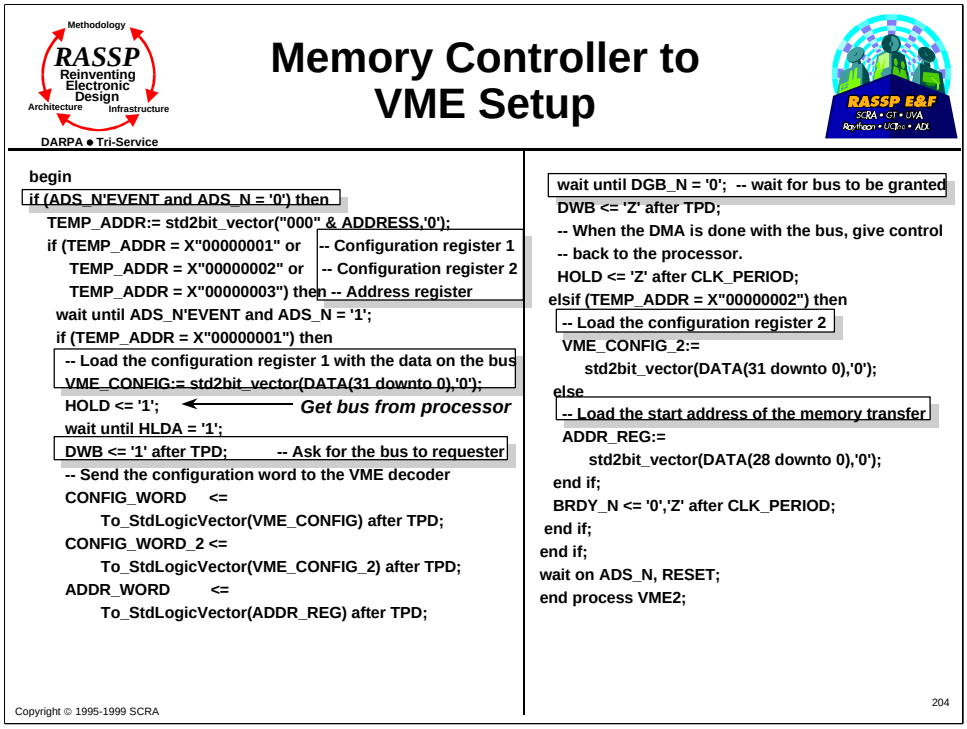


The configuration information is passed to a register contained inside the memory controller in a VME process. The setup information is shown above. All the VME protocol types are setup at this point. The maximum block length is 255 and is placed in bits 7 downto 0 of the configuration word 1. Configuration word two only uses 3 bits and sets the data type (D32 etc.)

The start address for the data transfer is contained in the address register ADDR_REG.



This code shows part of the VME main process where the configuration word information is placed into variable declarations and aliased for ease of reading.



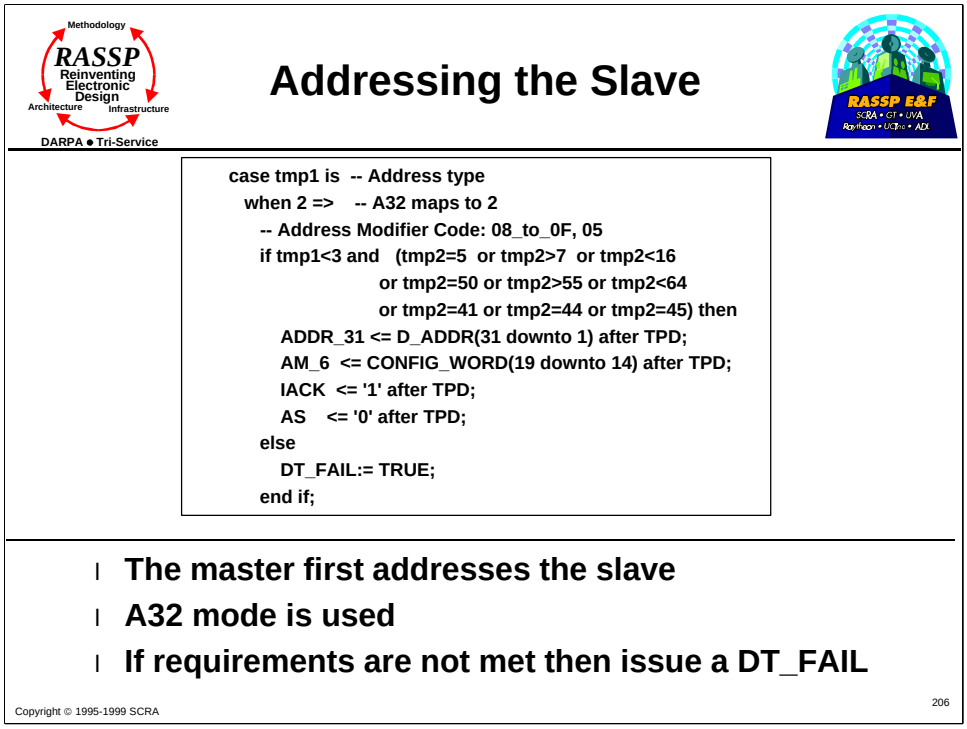
This code segment is in the memory controller and helps set up the DMA and VME for the specific transfer type. The first thing that the controller has to do is to get the start address from the processor and drive the address lines itself for the data transfer. The configuration words and address registers are first set up. When the configuration word 1 is written, the initiation of transfer begins.

The objective of this segment is to check the incoming address for that defined by the configuration register 1. The configuration register 1 will be at the special address defined to be 0x0008 (0x00000001 when addressing 64 bit-wise). This value is on a 64 bit boundary and can be addressed using BE_N and ADDRESS. When it is written to, the transfer begins.

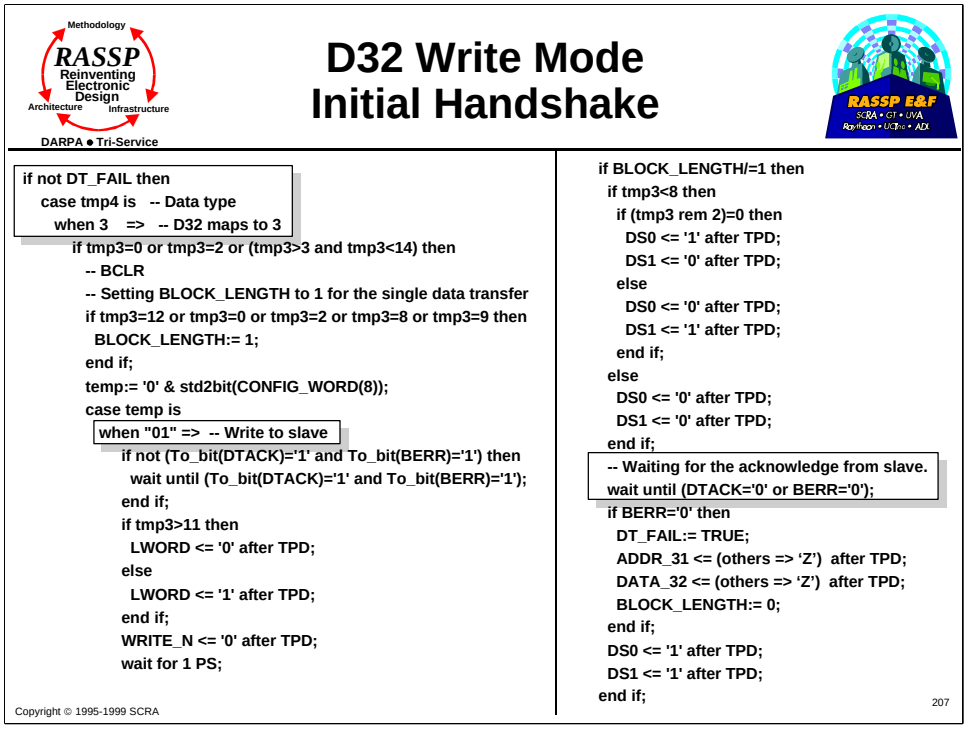
The DMA controller will then take control of the data bus and write the block to the slave.

The setup procedure is implemented as follows:

- Write the data that you wish to transfer beginning at an address.
- Write the start address to the ADDR_REG.
- Write the configuration data to the VME_CONFIG register 2.
- Write the configuration data to the VME_CONFIG register 1.



The first thing after latching all important information is to address the slave. In this case we are using address type A32 and, if the address modifier codes meet the requirements for this mode, then we can drive the specific signals above. If not, then the DT_FAIL flag is issued and no data will be sent.



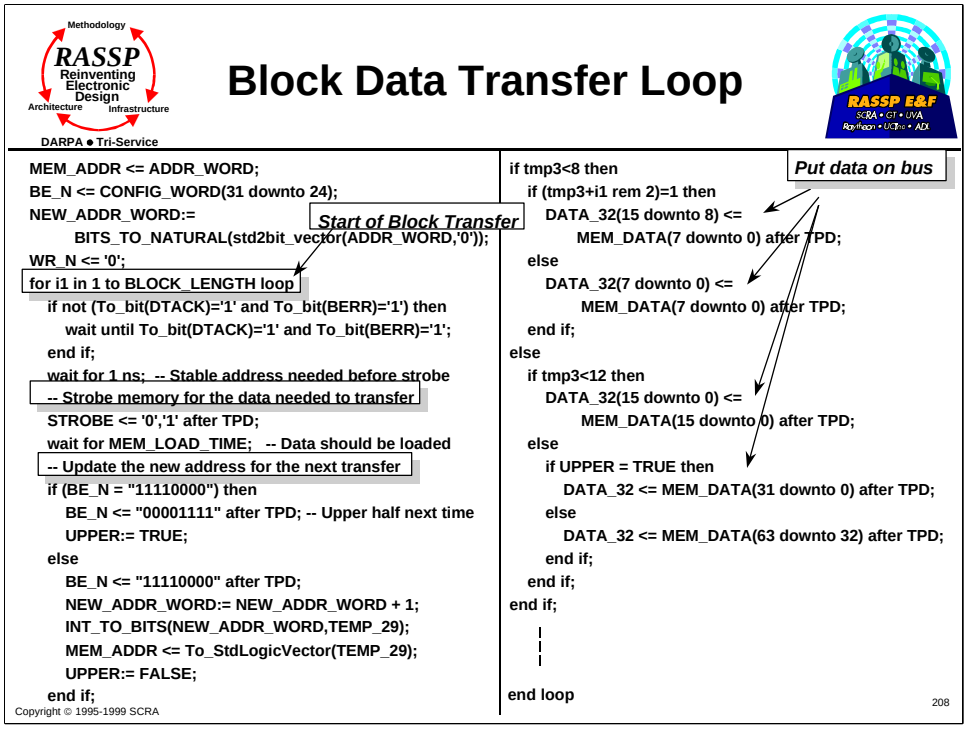
It there was no DT_FAIL, then, based on the data type (D32 in this case), we either do a read or a write. The variable “temp” is used to determine if read, write, or read-modify-write mode is used. In this case we are doing a write (“01” for temp).

Next wait for the previous slave to no longer drive data bus.

Then, set the signal level of LWORD used to select which byte locations are accessed during a data transfer.

Next terminate the address broadcast phase for a block data transfer write cycle.

Finally, set the signal levels of DS0 and DS1 used to select which byte locations are accessed during a data transfer.

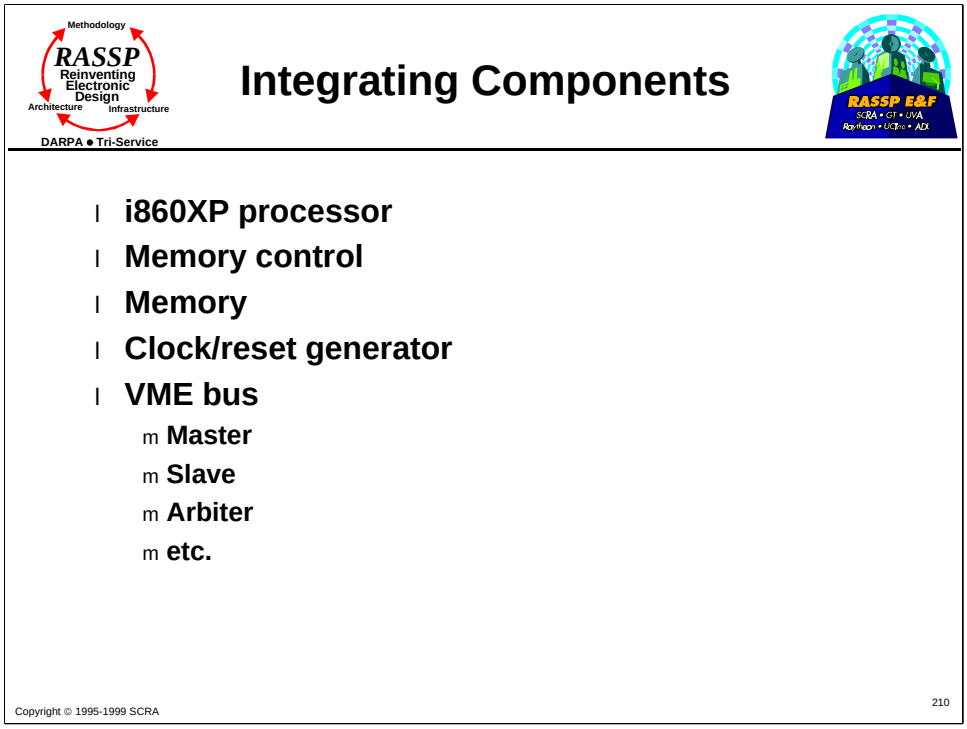


This segment of code does the data block transfer. The “for” loop continues for the entire block length. The addresses are updated in this segment and the memory is accessed for the data to be transferred.

Section Outline

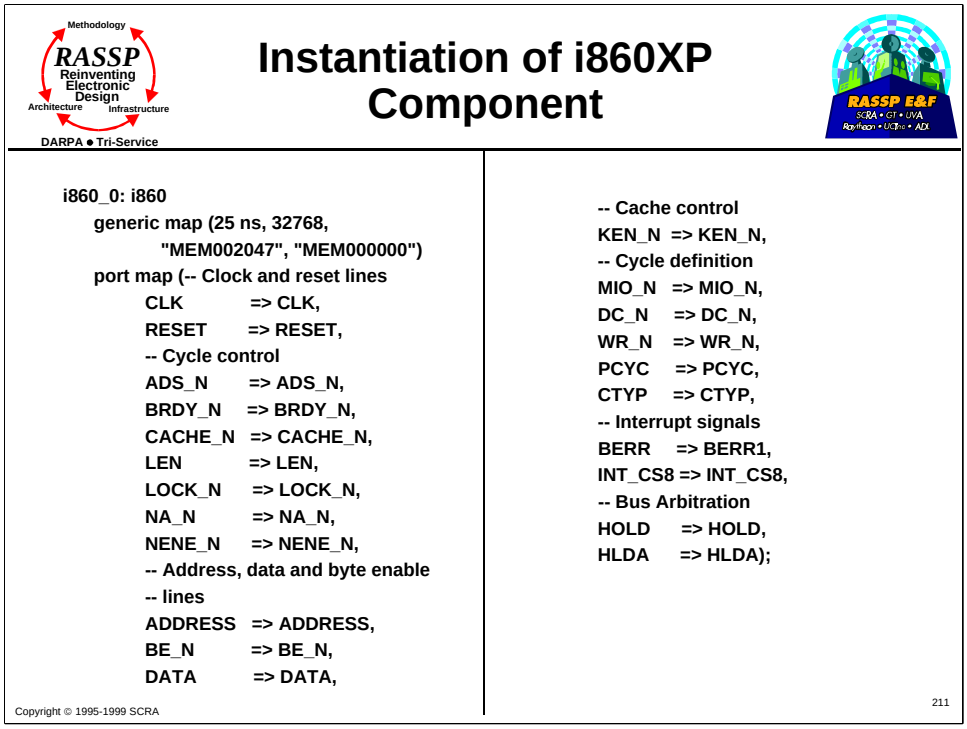
- The testbench
  - Clock/reset generator
  - Memory controller
  - Memory
- Testing the i860XP
- Results

- **Testing the MCV9**
- **Testing the IRST**
- **Creating a DMA to the VME in the memory controller**
- **Simulation results**



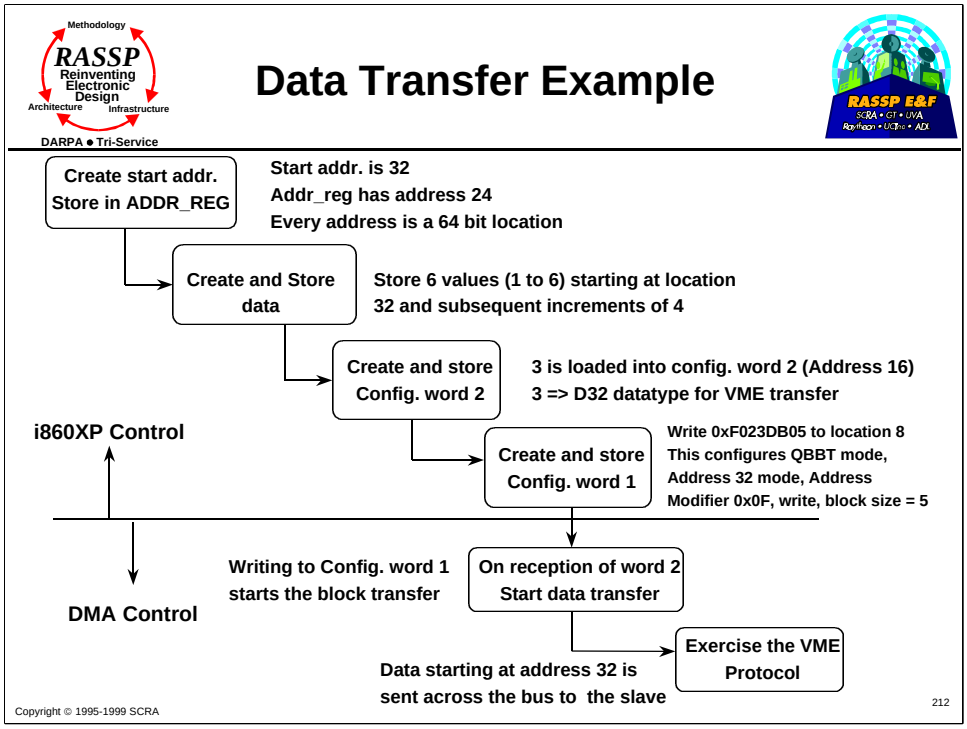
The components that needed to be integrated are listed above. These were all declared as component models in the top-level VHDL description and instantiated as shown on the next slide.

The VME bus had multiple components needed to handle its handshaking protocol. These will not be discussed. Included are such things as the arbiter, bus timer, etc.



This slide represents one instantiation of the i860XP component model in the system design. If multiple processors are required then additional instantiations can be done. In that case, we would have i860_1, i860_2, etc. all of type i860.

The same is done with the memory controller, memory, clock/reset, and VME models.



This slide represents the flow of a test case that was run to do a simple verification of the handshaking protocol and register setup of the memory controller and DMA.

ADDR_REG contains the address to start the DMA transfer.

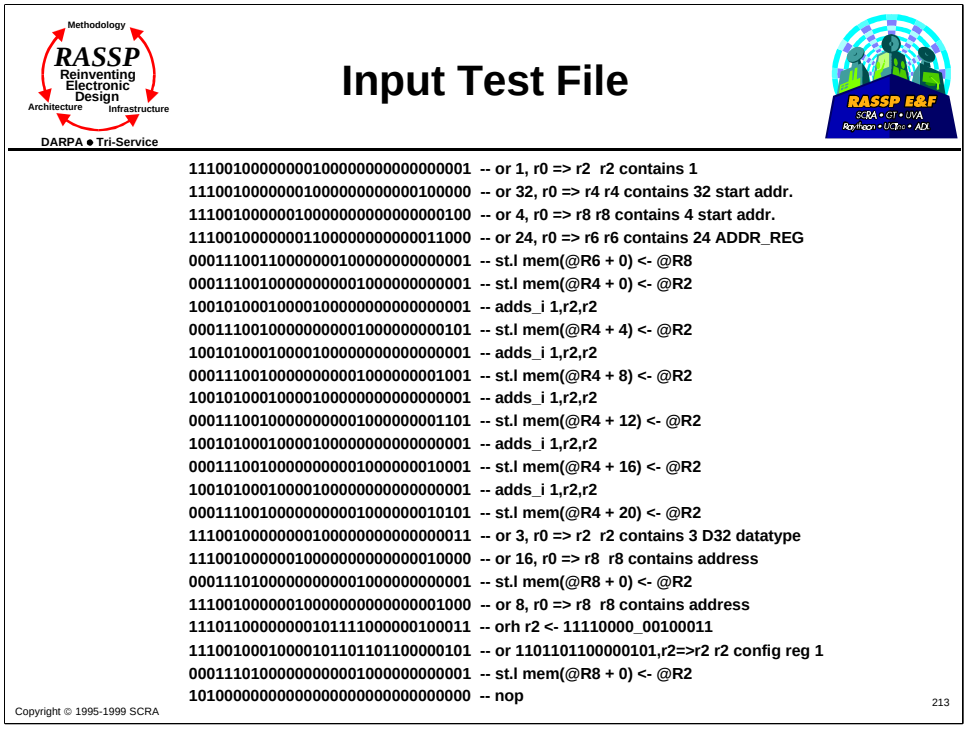
After the ADDR_REG is configured, the data is created by the i860XP and placed in memory starting at location 32. Six words were generated (in this case the numbers 1 through 6).

Configuration word 2 was then generated. This was used to set up the VME data transfer type. In this case D32 mode was selected.

Configuration word 1 was then set up. This has most of the information for configuring the VME mode. It set the VME protocol for a quad byte block transfer, address 32 mode, address modifier 0x0F, write, and with a block size of 5. This is accomplished by writing 0xF023DB05 to address 8.

A write to this register first stores the information, sets up the configuration, and then initiates the transfer.

After the DMA gains control of the address bus from the processor, it begins its transfer of data to the slave file.

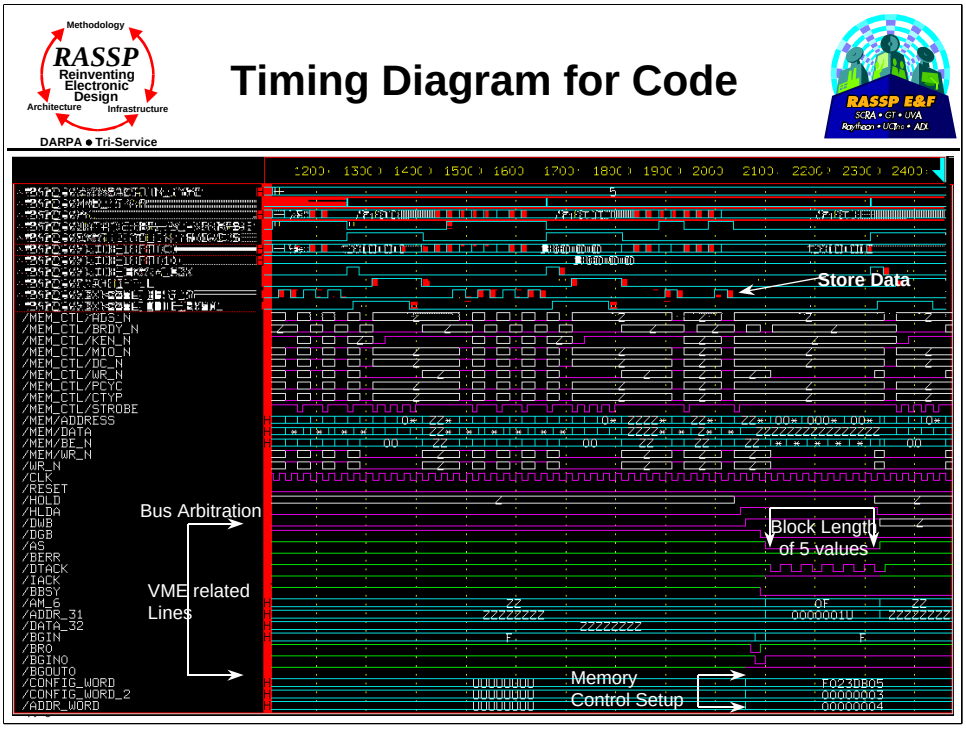


This slide shows the code used to generate the six data values and the configuration information. The input file is loaded into memory when the PC is at the proper address of the file. The 32 bit binary data represents the instruction shown on the right hand side of the slide.

The first part of the code computes the start address of data, and the address register in the memory controller is loaded with this value. Following this the data 1 through 6 is generated and sent to memory. Configuration word 2 (D32 data type) is then set up and sent to the correct address in the memory controller. Lastly, configuration word 1 is set up and sent to the controller's correct address.

This final st.l will cause the data transfer to take place. The next slide shows the timing diagram of this code running on the i860XP model.

Timing Diagram for Code

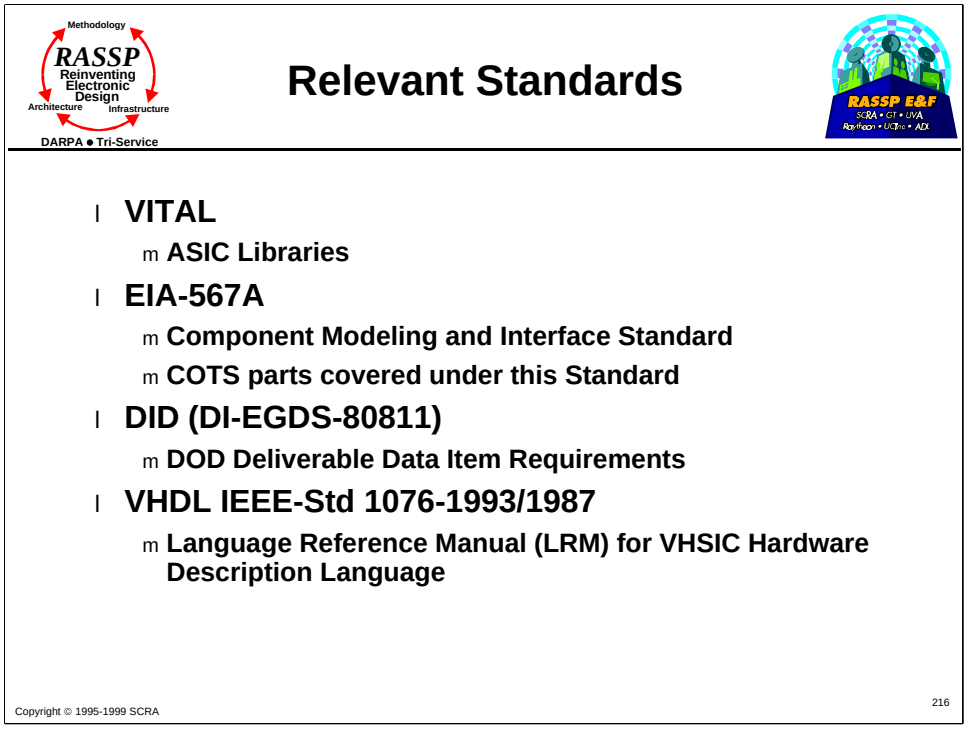


This diagram shows the storing of data into memory by the processor. The arrow pointing to store data shows the total of 9 writes (st.l instruction).

On the last write (to configuration word 1), the VME transfer is initiated. At this point we see the bus arbitration taking effect. The memory controller issues a HOLD to the processor and on the next cycle the processor can release the address bus by issuing a hold acknowledge (HLDA). The memory control setup is shown at the bottom where we see configuration word 1 containing 0xF023DB05. The block length is specified as 5 items and Quad Byte Block Transfer mode was selected. We see the 5 words passed by observing the DTACK signal.

Module Outline

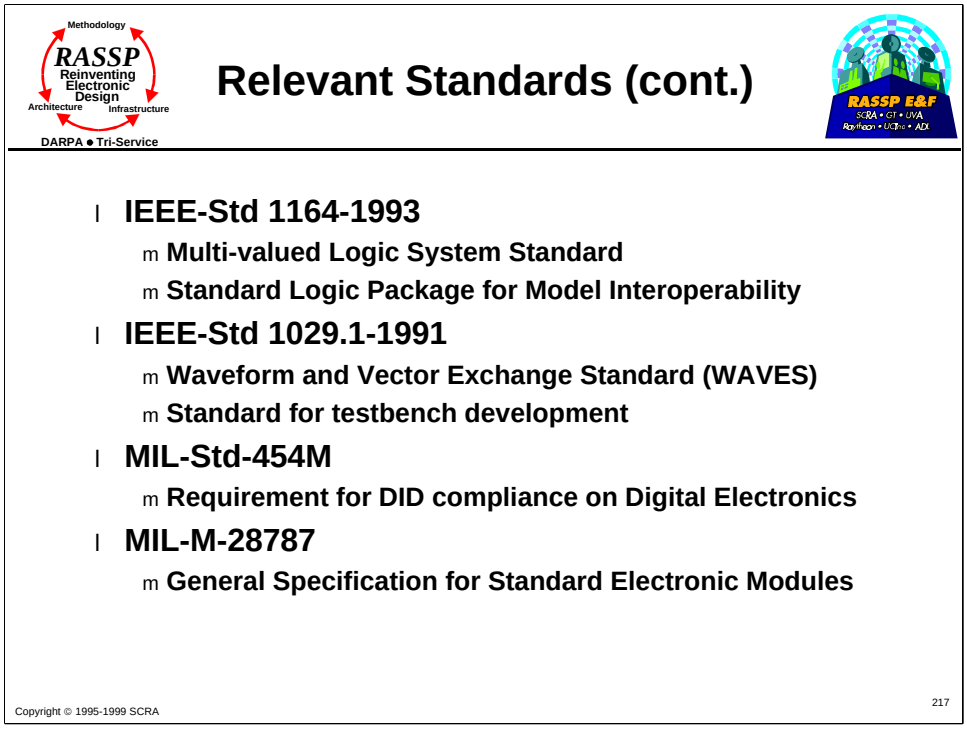
- Introduction
- Traditional Design Process
- The Virtual Prototyping Process
- Abstraction Levels and Limitations of VP
- Executable Requirement
- Executable Specification
- Data and Control Flow Modeling
- Performance Modeling
- Mixed-level Modeling
- Detailed-Behavioral Modeling and Detailed Design
- **Relevant Documents and Standards**
- Summary



Standards exist or are being created for the development of libraries of components for interoperability and portability. VITAL and EIA-567A are two such standards. VITAL is targeted for ASIC libraries and EIA-567A for component models such as processors, memories, etc.

The LRM contains the complete description for the VHDL language. Any model conforming to this standard should be able to run on a compliant simulator.

The DID describes the requirements for delivery of a component model on a defense-related contract.

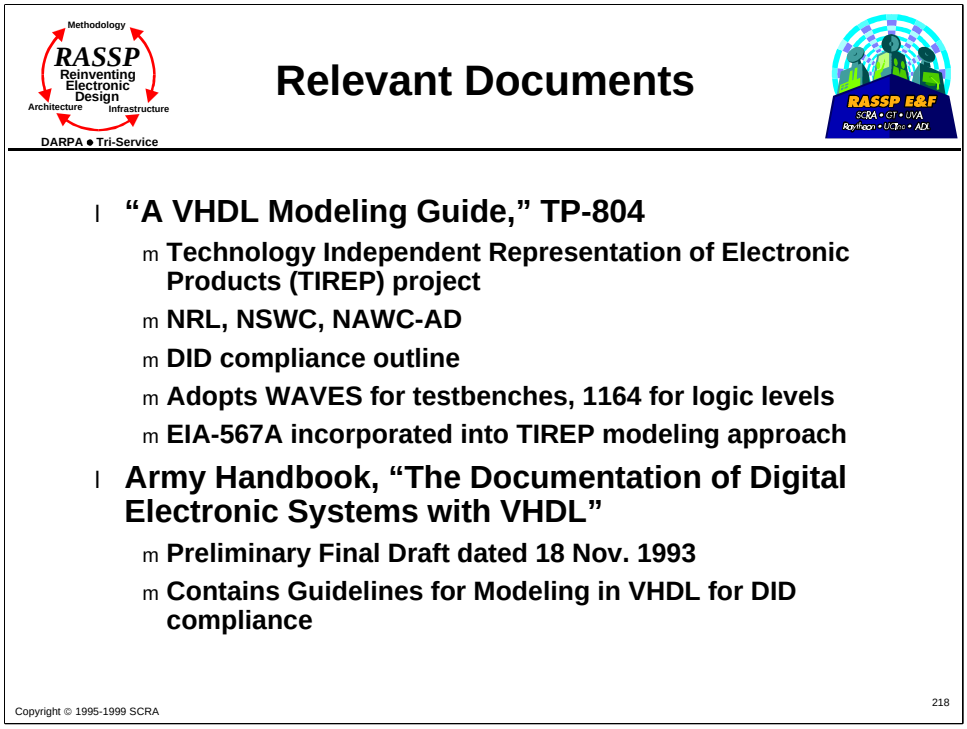


The 1164-1993 standard addresses the issues for model interoperability by defining the standard logic package to be used on all models.

Test benches are important in VHDL, and the 1029.1-1991 standard addresses issues dealing with testbench development and the use of WAVES for this purpose.

MIL-Std-454M defines the requirements for DID compliance on delivered military components.

MIL-M-28787 describes the general specification of electronic modules on military systems.



Two documents relating to VHDL development efforts have been developed by the government. “A VHDL Modeling Guide” was developed as part of the TIREP project by the Navy and describes how users of VHDL can write models to be DID compliant and conform to the EIA-567A standard. It also outlines the use of WAVES as the standard test bench development method and stresses the use of the 1164 standard logic package for signal resolution.

The Army Handbook, to be published in November of 1995 (preliminary versions already available), also contains guidelines for the development of VHDL models for DID compliance.

Relevant Documents (cont.)

- **"VHDL Performance Modeling Interoperability Guidelines"**
  - **Developed by Honeywell Technology Center**
  - **Describes guidelines for developing interoperable models in VHDL at the performance level**
  - **Defines token fields for signals**
  - **Possible precursor to standardization**

Module Outline

- Introduction
- Traditional Design Process
- The Virtual Prototyping Process
- Abstraction Levels and Limitations of VP
- Executable Requirement
- Executable Specification
- Data and Control Flow Modeling
- Performance Modeling
- Mixed-level Modeling
- Detailed-Behavioral Modeling and Detailed Design
- Relevant Documents and Standards
- **Summary**

Summary

- Presented a picture of current practice and how virtual prototyping can help improve the digital design process
- Presented the virtual prototyping process and its application to the design of complex embedded digital systems
- Presented the levels of virtual prototyping in the top down design process and their importance to the capture of system requirements
- Presented VHDL examples at each stage to illustrate the methods used for coding the system requirements important to that stage of development

References

[Anderson94] Anderson A., Shaw G., Sung C., "VHDL Executable Requirements", *1st Annual RASSP Conference Proceedings*, August 15-18, 1994, pg 87-90.

[Arch.Guide94] Lockheed Sanders, Inc. , Hughes Aircraft, Motorola, ISX Corporation, RASSP Architecture Guide , Rev. A, November 23, 1994. See [Lockheed95] for addition information.

[Egolf95] Egolf, T., et. al., "Experiences with VHDL Models of COTS RISC Processors in Virtual Prototyping of Complex System Synthesis", *VHDL International Users Forum* , Spring 1995, pp.7.11-7.27.

[Handbook93] Army Handbook, The Documentation of Digital Electronic Systems with VHDL, Preliminary Final Draft dated 18 Nov. 1993.

[Hein95A] Hein C., Nasoff D., "VHDL-Based Performance Modeling and Virtual Prototyping", *2nd Annual RASSP Conference Proceedings*, July 24-27, 1995.

[Hein95B] Hein C., et al., "Tutorial: VHDL--Based Top-Down Virtual Prototyping for Large DSP Systems", *2nd Annual RASSP Conference Tutorial Program*, July 24-27, 1995.

Honeywell Carpenter T., Rose F., Steeves T., "Performance Modeling with VHDL", Honeywell Systems & Research Center, Slide presentation

[IEEE] All referenced IEEE material is used with permission.

[i86091] i860 Microprocessor Family Programmer's Reference Manual, Intel Corporation, 1991.

[LMC-Meth] RASSP Methodology Version 1.0 , Martin Marietta Laboratories, December, 1994.

[LMC-Arch] Lockheed Sanders, RASSP Architecture Guide 1994. See [Lockheed95] for addition information.

[Lockheed95] Lockheed Sanders 18 Month Review, February 15-16, 1995. This work was performed by Sanders, a Lockheed Martin Company, as a part of the Sanders RASSP program under contract N00014-93-C-2172 to the Naval Research Laboratory, 4555 Overlook Avenue, SW, Washington, DC 20375-5326. The Sponsoring Agency is: Defense Advanced Research Projects Agency, Electronic System Technology Office, 3701 North Fairfax Drive, Arlington, VA 22203-1714. The Sanders RASSP team consists of Sanders, Motorola, Hughes, and ISX.

References (cont.)

[Madisetti94] Madisetti V.K., "Vive La Difference", *RASSP Digest*, 4th Quarter, pp. 17-21.

[Madisetti95] Madisetti V.K., "Rapid Prototyping of Application-Specific Signal Processors: Current Practice, Challenges, and Roadmap", *IEEE ICPP'95 Workshop on Challenges in Parallel Processing",* August 1995.

[Myers95] Myers C., Dreiling R., "VHDL Modeling for Signal Processor Development" , *Proceedings IEEE International Conference Acoustics, Speech, and Signal Processing*, 1995; © IEEE 1995

[PGM90] Prepared by Naval Research Laboratory, Processing Graph Method: Tutorial, January 1990.

[PI95] Proceedings from the Principal Investigators Meeting, January 9-13, 1995

[Richards97] Richards, M., Gadient, A., Frank, G., eds. *Rapid Prototyping of Application Specific Signal Processors*, Kluwer Academic Publishers, Norwell, MA, 1997

[Rose94] Rose F., Steeves T., Carpenter T.,"VHDL Performance Modeling", *1st Annual RASSP Conference Proceedings*, August 15-18, 1994, pg 60-70.

[Shaw95A] Shaw, G.A., et al, "Assessing and Improving Current Practice in the Design of Application-Specific Signal Processors" , *RASSP Digest,* 1st Quarter 1995, pp. 20-24.

[Shaw95B] Shaw, G.A. , et al, "Assessing and Improving Current Practice in the Design of Application-Specific Signal Processors, *Proceedings IEEE International Conference Acoustics, Speech, and Signal Processing,* 1995.

[Taxonomy98] Hein C., et al., "VHDL Modeling Terminology and Taxonomy - Revision 2.4.4", *RASSP Taxonomy Working Group (RTWG),* March 6, 1998

[Yu 93] Yu T., "Create optimal simulation libraries using VHDL", *Electronic Design News*, May 13, 1993, pp. 133-140.