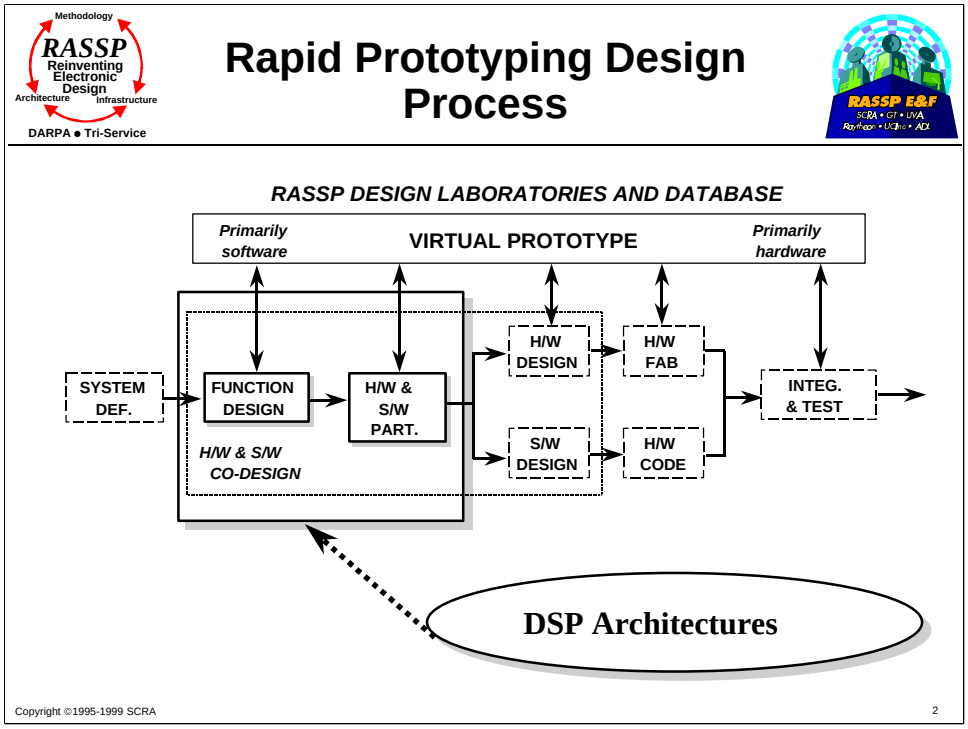
DSP Architectures for RASSP

RASSP Education & Facilitation Program
Module 21

Version 3.00



Architecture design is based on the functional computational and communications requirements of the algorithm or algorithms selected to meet the specification. These trade-offs fall under the functional design and partitioning sections of the RASSP process.

Module Goals

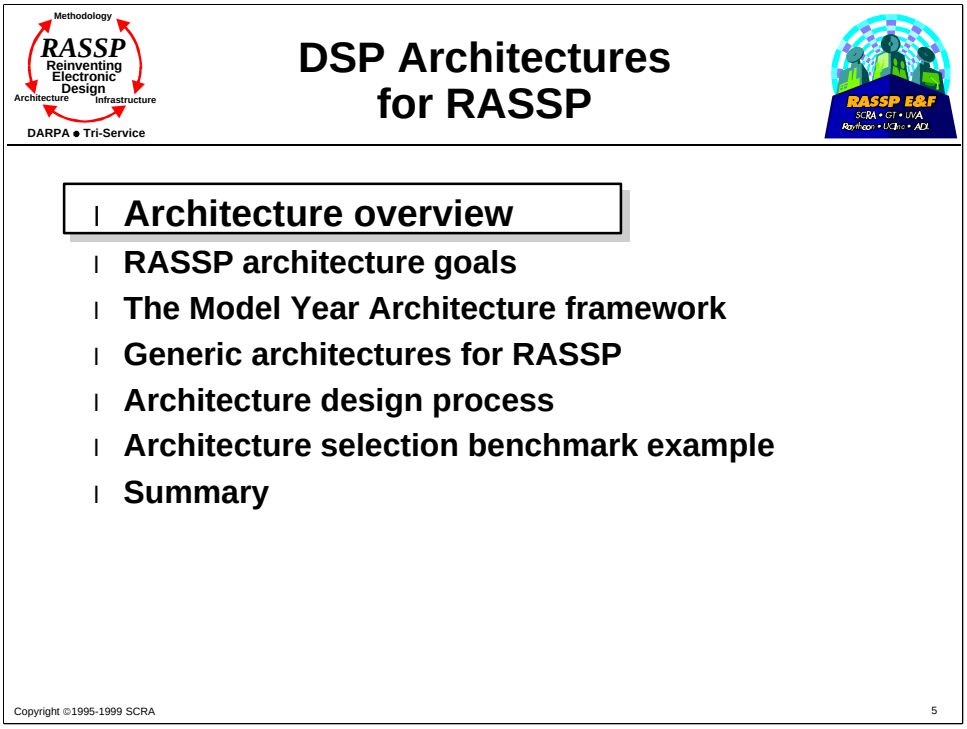
- **Present an introduction to state of the art in DSP architectures**
  - **Architecture overview**
- **Present RASSP-related HW/SW architectures and methodologies**
  - **RASSP architecture goals**
  - **The Model Year Architecture framework**
  - **Generic architectures for RASSP**
  - **Architecture design process**
  - **Architecture selection benchmark example**
  - **Summary**

This module will cover the areas listed above. The purpose of this module is to expose the user of the RASSP process to the area of architecture design.

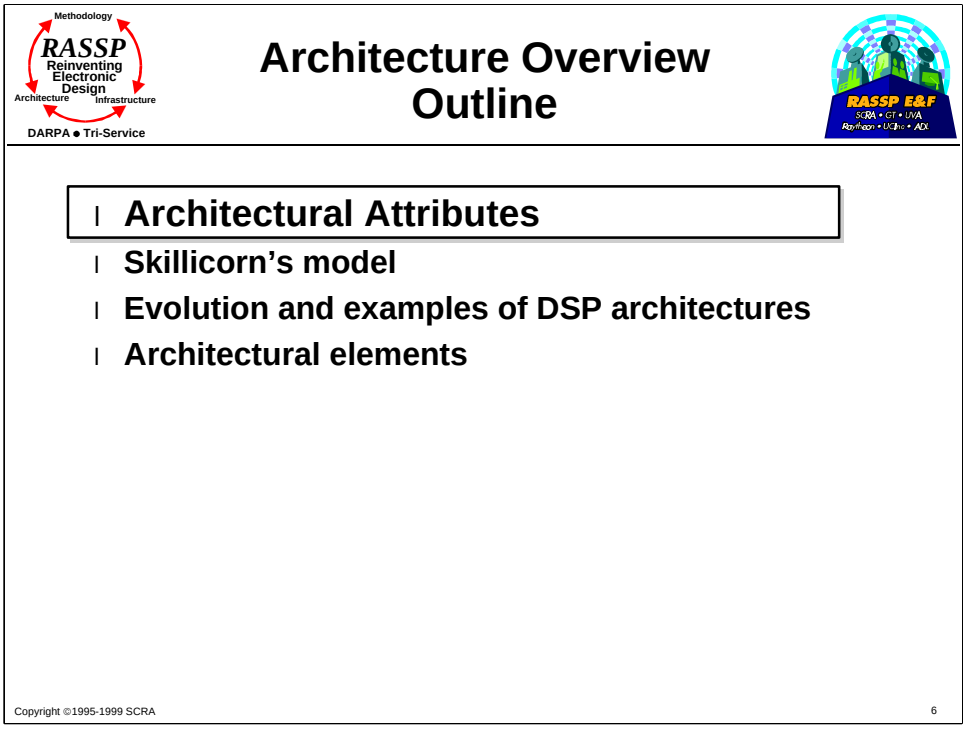
Module Outline

- **Architecture overview**
- **RASSP architecture goals**
- **The Model Year Architecture framework**
- **Generic architectures for RASSP**
- **Architecture design process**
- **Architecture selection benchmark example**
- **Summary**

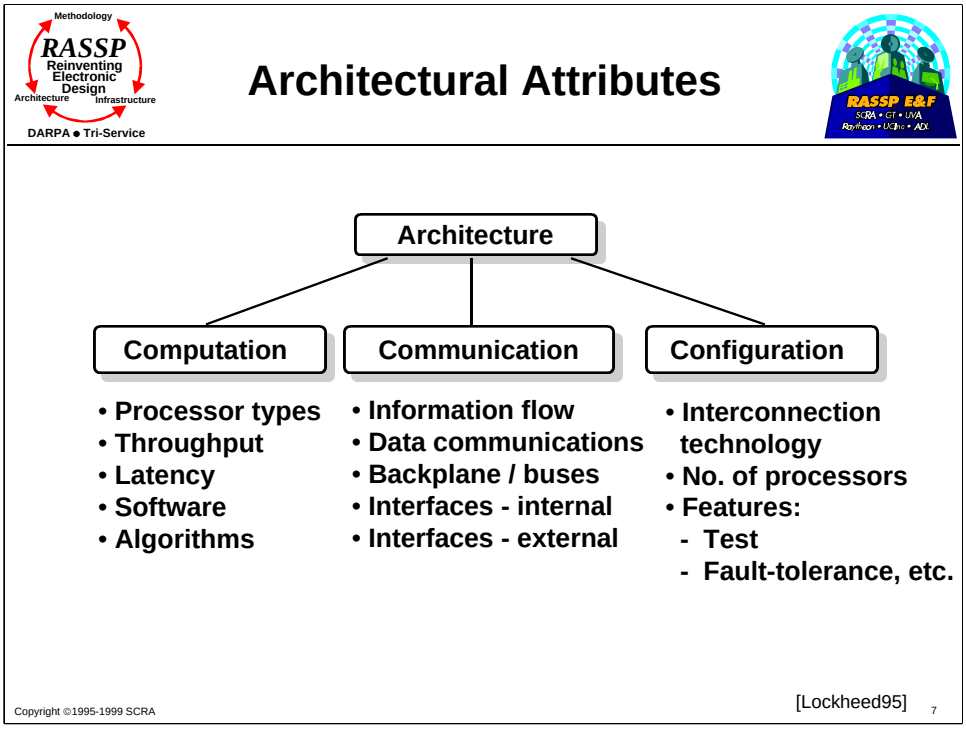
This slide presents the outline of topics to be covered in this module starting with an architecture overview, then presenting the RASSP goals, followed by the model year architecture concept that is used on RASSP. Later sections present the RASSP architecture design process.



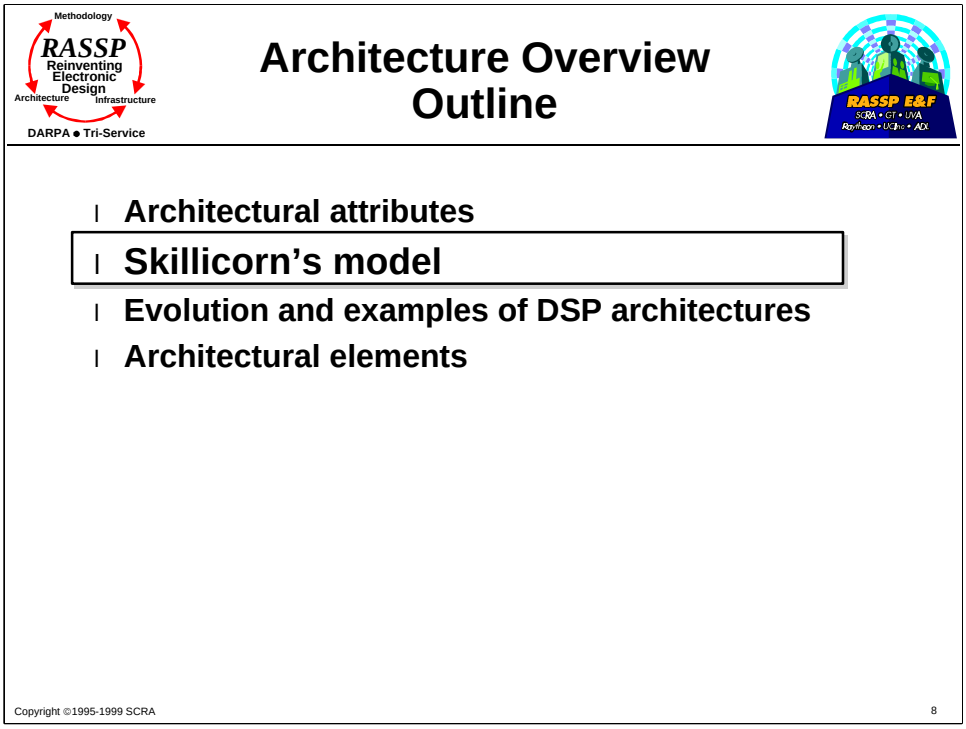
The first topic will present an overview of architectures for DSP.



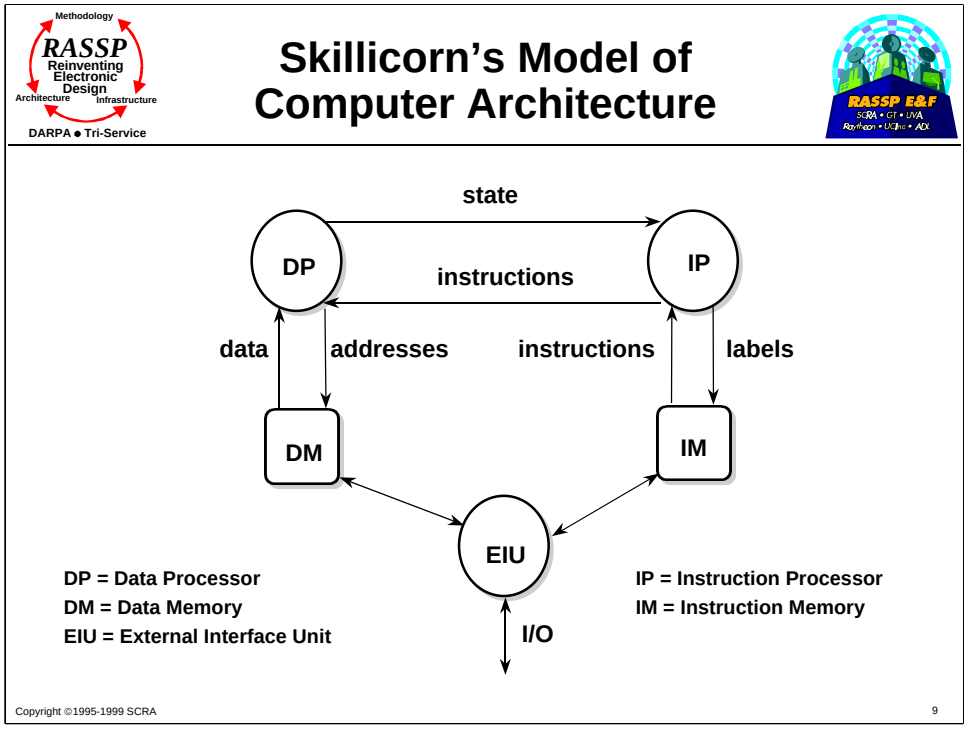
Contained in this section are a listing of the main architectural attributes, a model for architectures presented by Skillicorn, evolution of architectures used for large systems, and a description of architectural building blocks.



The key architectural attributes for a signal processing system include those listed in the chart above. They fall into three broad categories and help specify the computational and communications requirements of the system, along with a system configuration needed to combine all the resources to meet the specification.



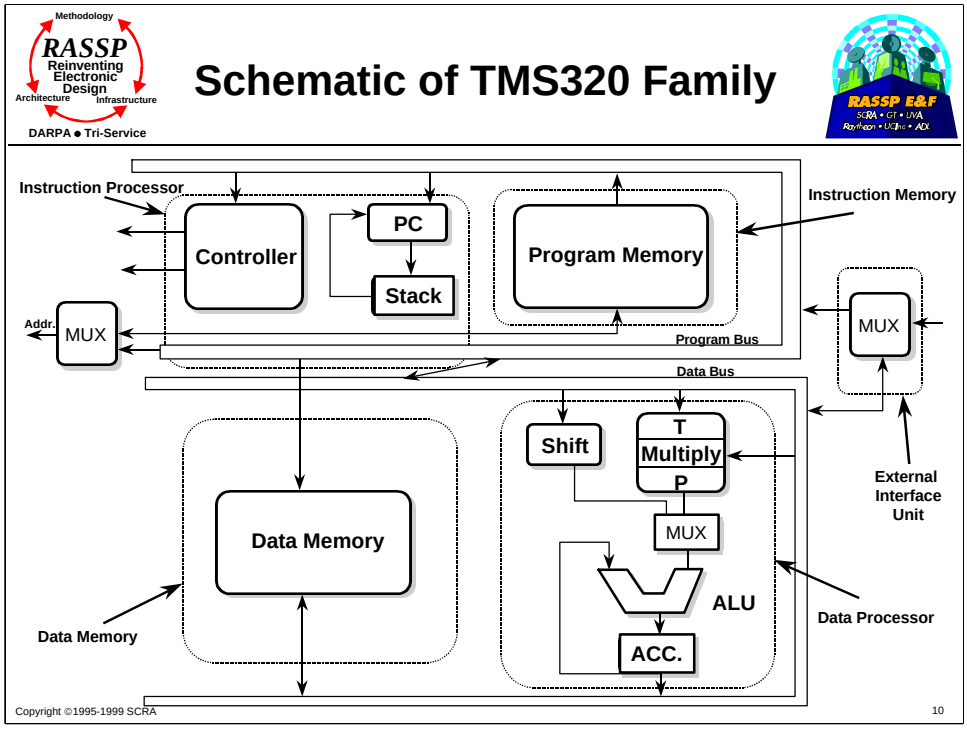
Skillicorns model represents a method for describing uniprocessor and multiprocessor architectures for computing systems.



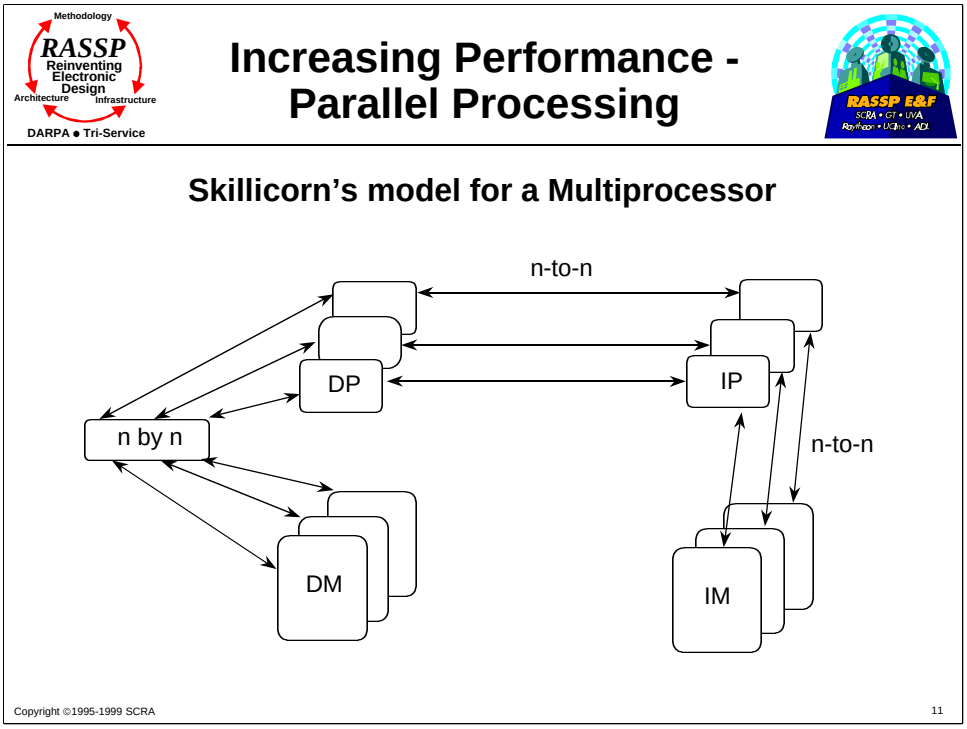
Skillicorn proposed a general model of all computing architectures. It is based on the following elements:

- Data Processor (DP)
- Instruction Processor (IP)
- Data Memory (DM)
- Instruction Memory (IM)
- External Interface Unit (EIU).

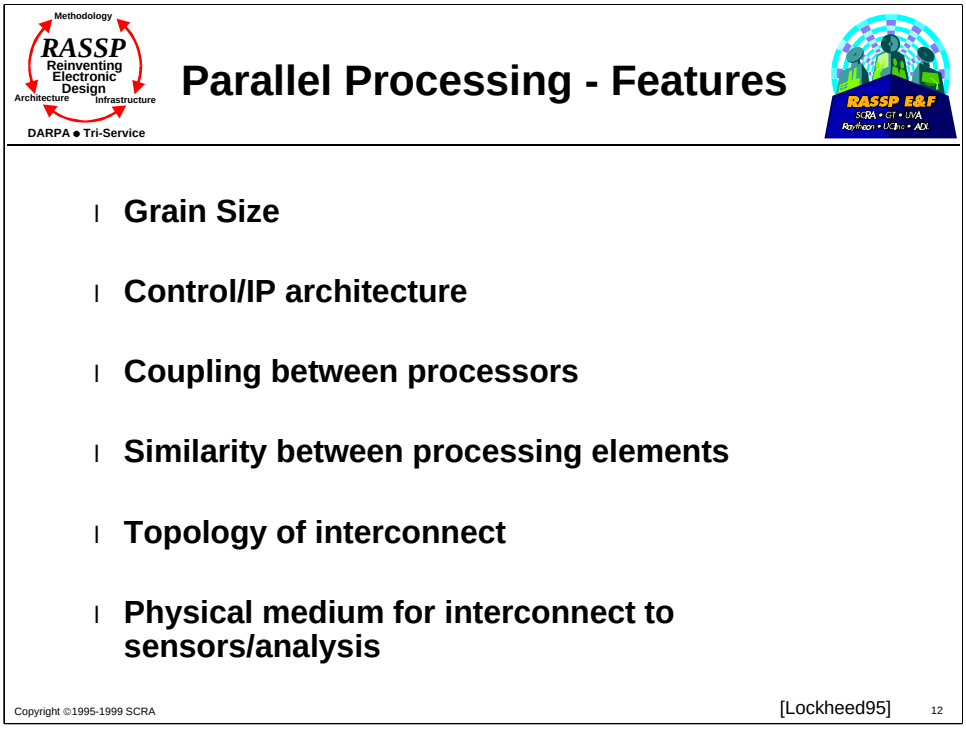
The IP passes information to the DP in the form of control information (Instructions), and the DP returns the state of the computation back to the IP. Both the IP and the DP have some means of obtaining control or data information from their DM and IM and also from the external world through an EIU. (See Madisetti1995)



A typical signal processing chip family can be seen to follow this model closely (TI TMS320 family). In this case, all the representative parts are shown in the figure. The IP consists of a controller that fetches instructions from its IM based on the address in the PC. The control is decoded and passes information to the DP (consisting of an ALU, multiplier, shifter, and accumulator) to process the data from its data memory. The external interface can load instructions or data through a number of elements (serial ports, parallel ports, etc.).



Skillicorn's model can also be applied to a multiprocessor system. In this case there can be N IPs and DPs along with access to their respective memories. In the case above, the IPs each have their own separate instruction memory to process control information, while the DPs have access to all DM through possibly a shared memory scheme.



Some of the features of parallel processors are listed above.

Grain size can be “coarse” or “fine”. In the coarse-grain architecture, the individual processors do large chunks of processing with little communication between them. Fine-grain do smaller chunks and communicate more frequently (require higher throughput interconnects).

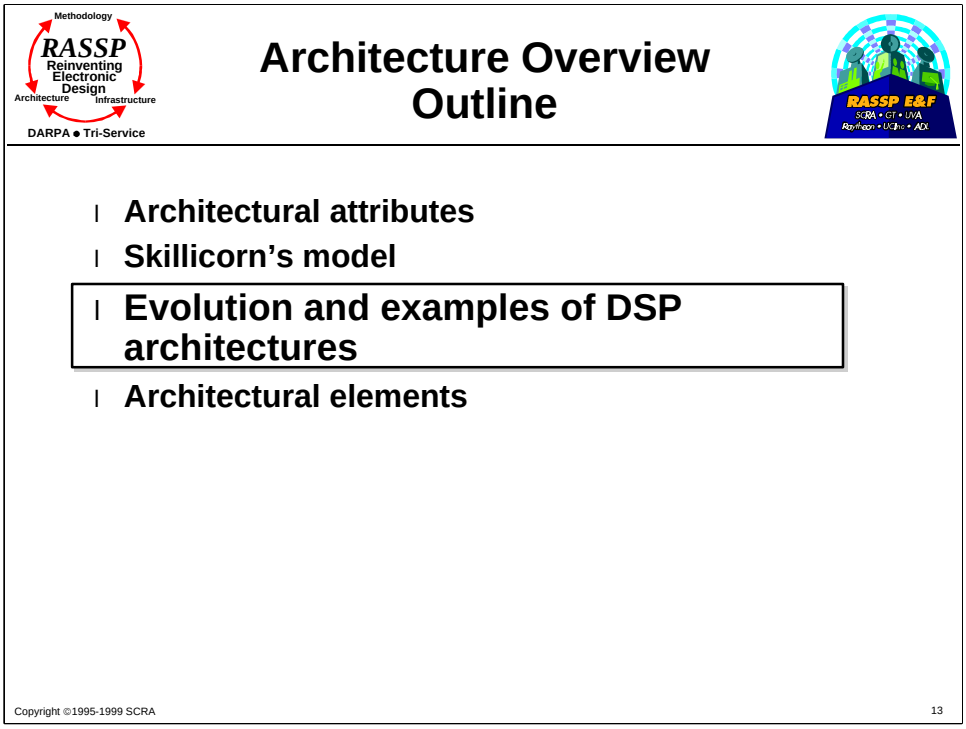
Different control architectures range from Single Instruction Multiple Data (SIMD) to Multiple Instruction Multiple Data (MIMD). SIMD performs the same set of operations synchronously on many streams of data (typically used at front-end sensor arrays). MIMD is more flexible, and each processor works independently with its own set of operations and data streams (typically used when computations can be broken into multiple threads).

Coupling can be done loosely or tightly. Loosely-coupled architectures have processors that act fairly independently from other processors. Tightly-coupled architectures have processors which can share memory and possibly be controlled by a global operating system.

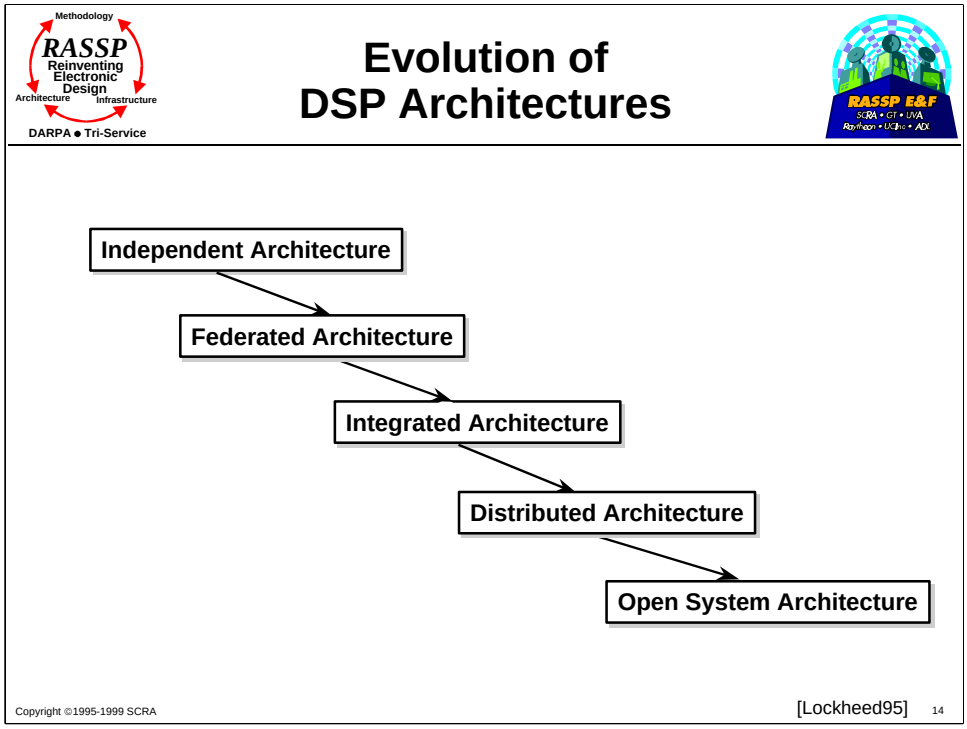
Similarity can involve homogeneous (same type of processing elements) or heterogeneous (various types) type architectures.

Topology: Various methods of interconnect are possible including simple shared multi-drop buses, switch (crossbar) types, and mesh types.

Medium: This includes such things as copper-wire, coax, and fiber-optic materials. Connections can be serial or parallel.

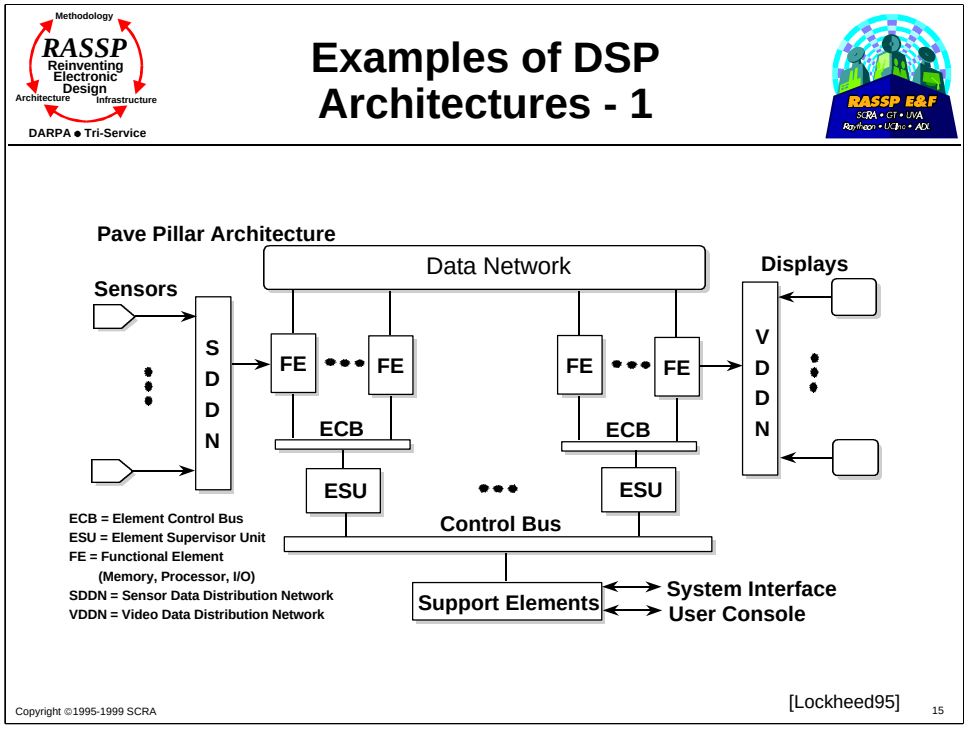


An evolution of DSP architectures is presented in this section along with some examples of recent architectural designs.



This shows architectures in a historical perspective starting from the independent type to the current trend of open systems.

- Independent: No concern about interoperability or the sharing of resources between systems. Each performs its independent functions.
- Federated: The independent systems are interconnected with a low-speed control bus, and a central computer exercised overall control of the function of the various independent systems.
- Integrated: Communication between systems is at a higher speed, and possible resource sharing is done. Fault tolerant systems are possible now because one non-functioning resource can be replaced with another. Data fusion is also possible where multiple resources can share the same data at the sensor input.
- Distributed: Resources are spread according to optimal physical constraints while retaining a strong integrated capability.
- Open Systems: Allows for incremental changes of hardware components as technology evolves. It clearly specifies the critical interfaces for replaceable modular elements. It allows for vendor independence.

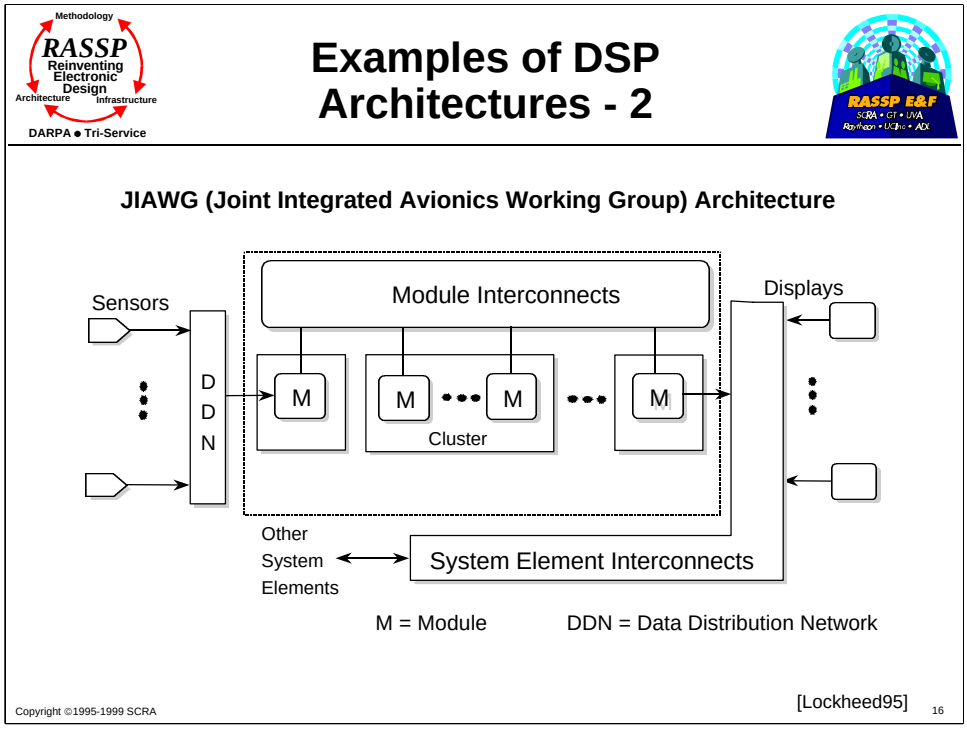


The Pave Pillar Architecture was put together by the Air Force in the early 1980s. It was created with the intent of being modular, open, fault tolerant, and highly flexible.

The control bus carries interprocessor messages and is usually implemented in redundant form for fault tolerance. It has an associated BIT bus for maintenance and debugging.

The data network is usually implemented using a non-blocking crossbar network and transfers large blocks of data. This helps support multiprocessor dataflow.

The architecture can be partitioned into core modules (for local control and communication), functional element modules (for high performance processing, I/O, and storage), and miscellaneous modules (for power regulation and other support functions).

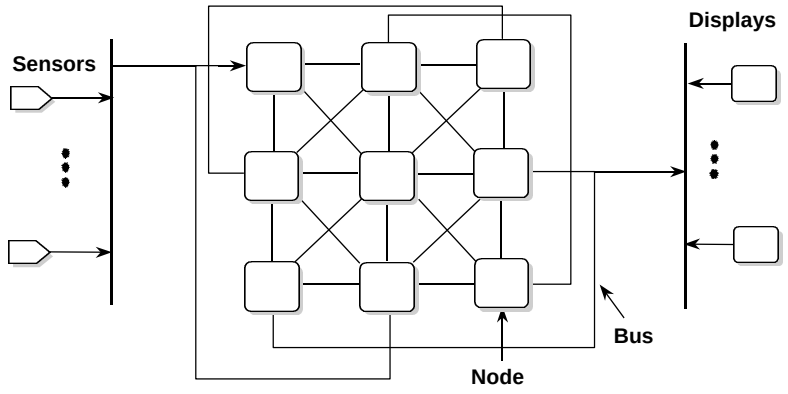


The JIAWG is an implementation of the Pave Pillar Architecture. Use of the JIAWG standard was mandated by Congress for specific contracts developed for the Tri-services.

In this architecture, the control bus is the Parallel Intermodule Bus (PI-bus), and the associated BIT bus is the Test and Maintenance Bus (TM-Bus).

Examples of DSP Architectures - 3

AOSP (Advanced On-board Signal Processor) Architecture



[Lockheed95]

17

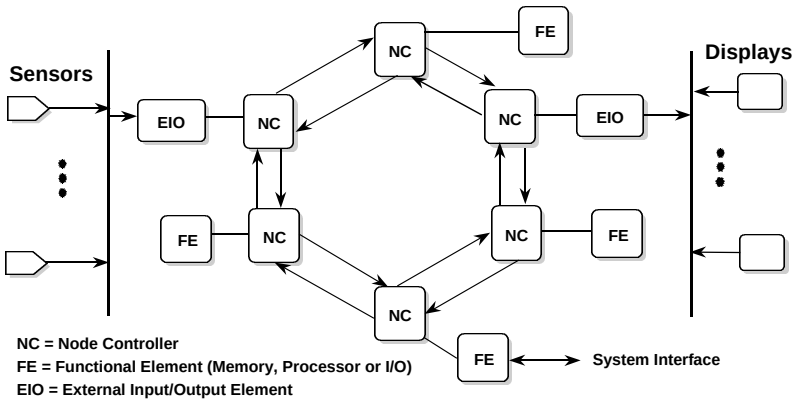
The AOSP architecture was designed for the Air Force in the 1980s for use in space. It implements a mesh architecture with a planar-4 mesh topology.

This topology has four families of buses: horizontal, vertical, left diagonal, and right diagonal. Each node connects separately to the four bus families. using the 2-dimensional pattern shown.

The Node Operating System (NOS) in each controller routes messages to their destination, invokes the application tasks, and monitors error reports.

Examples of DSP Architectures - 4

SAFENET (Survivable Adaptable Fiber Optic Embedded Network) Architecture

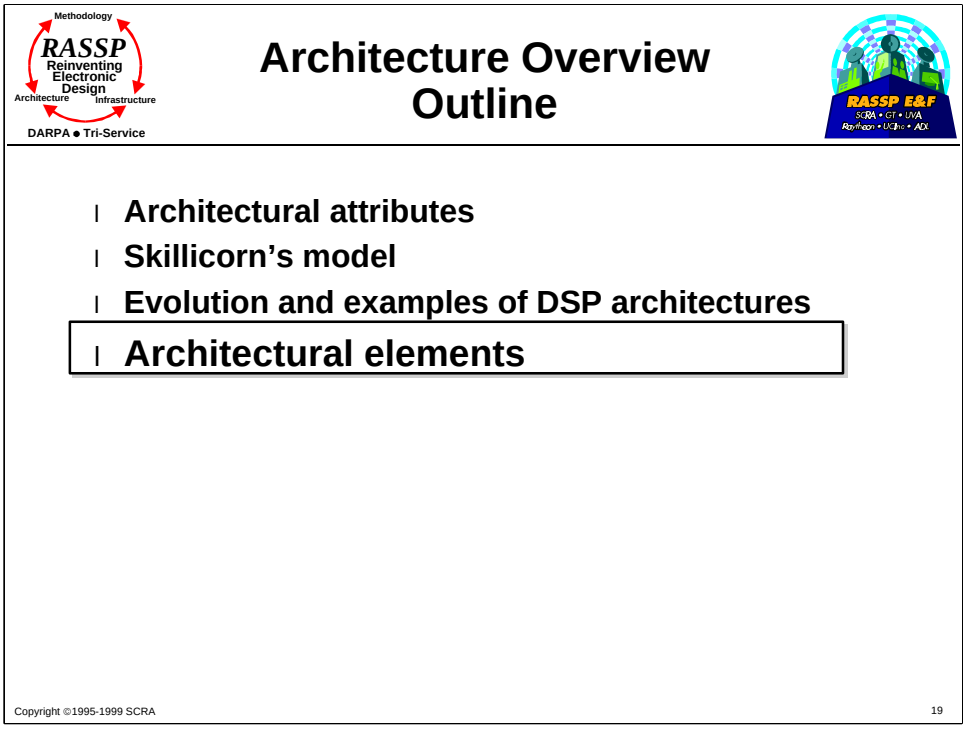


[Lockheed95]

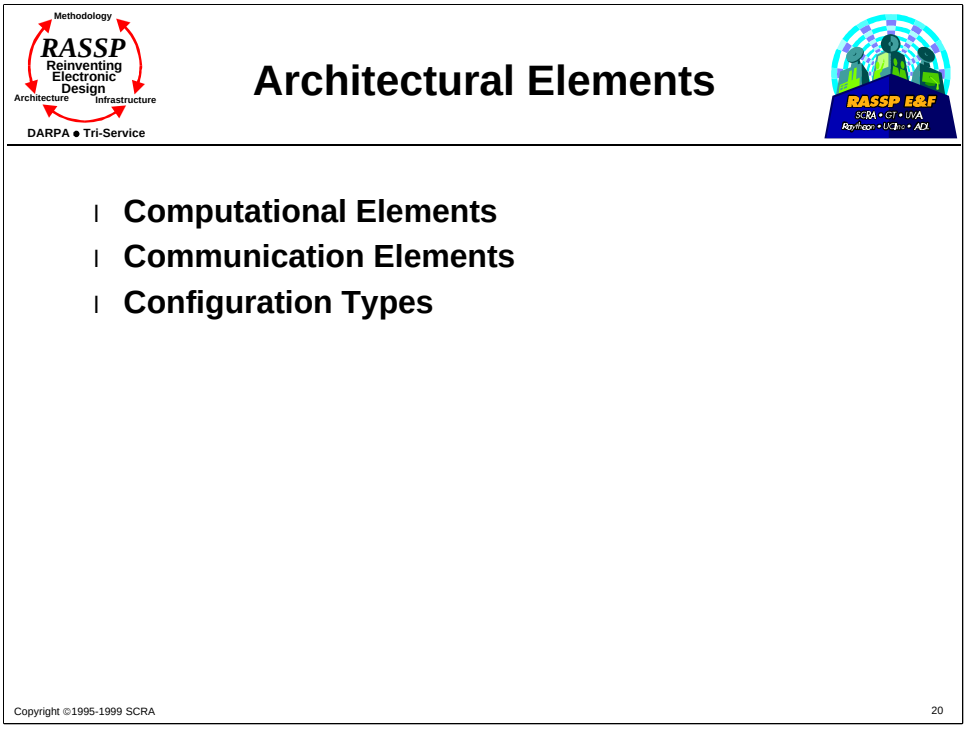
SAFENET is a local area network standard developed by the Navy's Next Generation Computer Resource (NGCR) Program in the early 1990s. This was used to connect multiple digital systems on-board a ship.

The architecture uses Fiber Distributed Data Interface (FDDI) at the network level to implement a dual counter-rotating ring topology. This helps achieve high bandwidth, low cost, and protection from EMI.

Fault-tolerance is provided in two ways. First, a failed node can be isolated from the network via an optical bypass switch contained within the node controller. Second, a failed node can be isolated by reconfiguring the dual ring into a double-length single ring, thereby excluding the failed component.



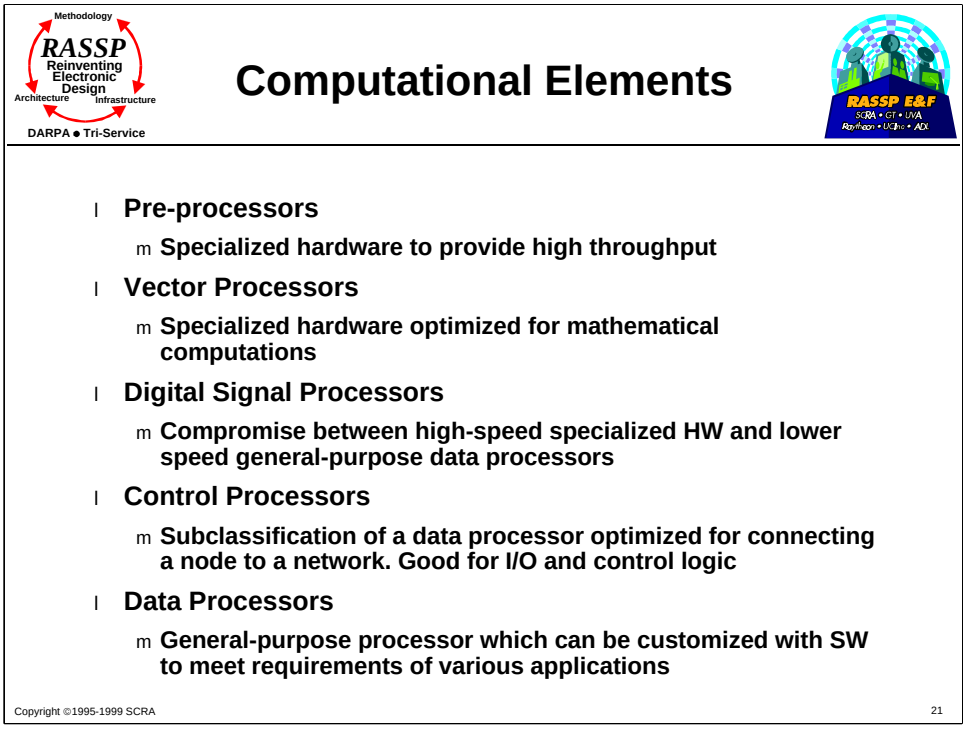
This section will present an overview of elements used to build architectures for DSP systems.



We will now talk about some of the architectural elements that compose the architectures just described on the previous slides.

They can be broken into the categories shown in this slide.

Other important elements of an architecture include the software design and physical constraints, which will not be discussed here.



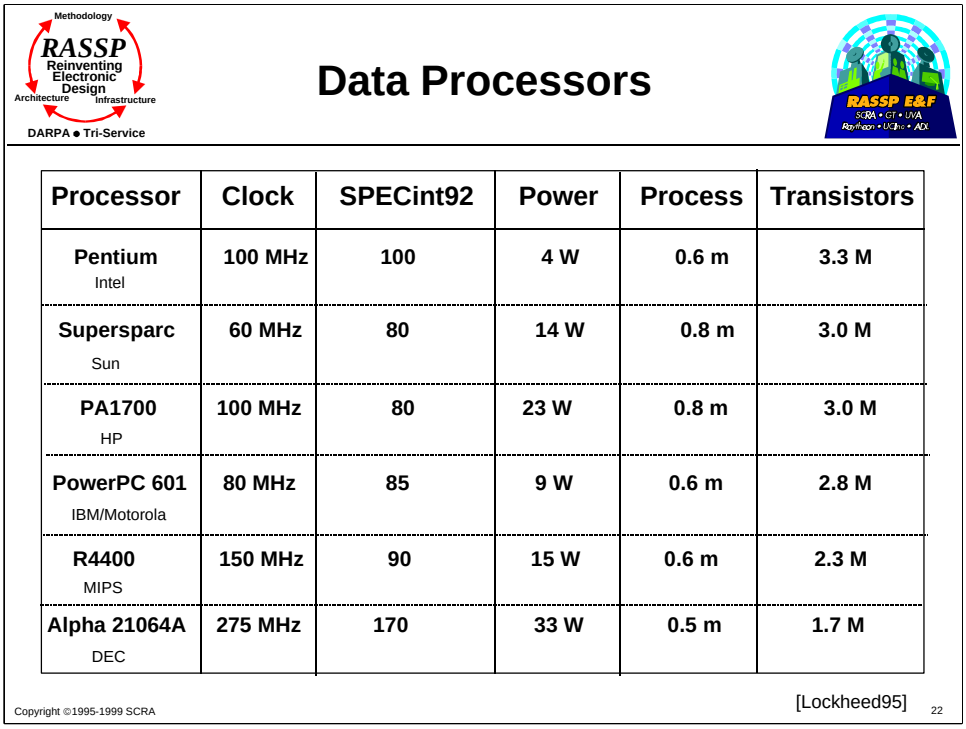
Pre-processors are usually implemented for high throughput application at the front-end, where sensor data input needs to be processed.

Vector processors are specialized for mathematical computations and are typically coarse-grained compute engines. A typical application may include an FFT computation. They are typically pipelined for increased throughput.

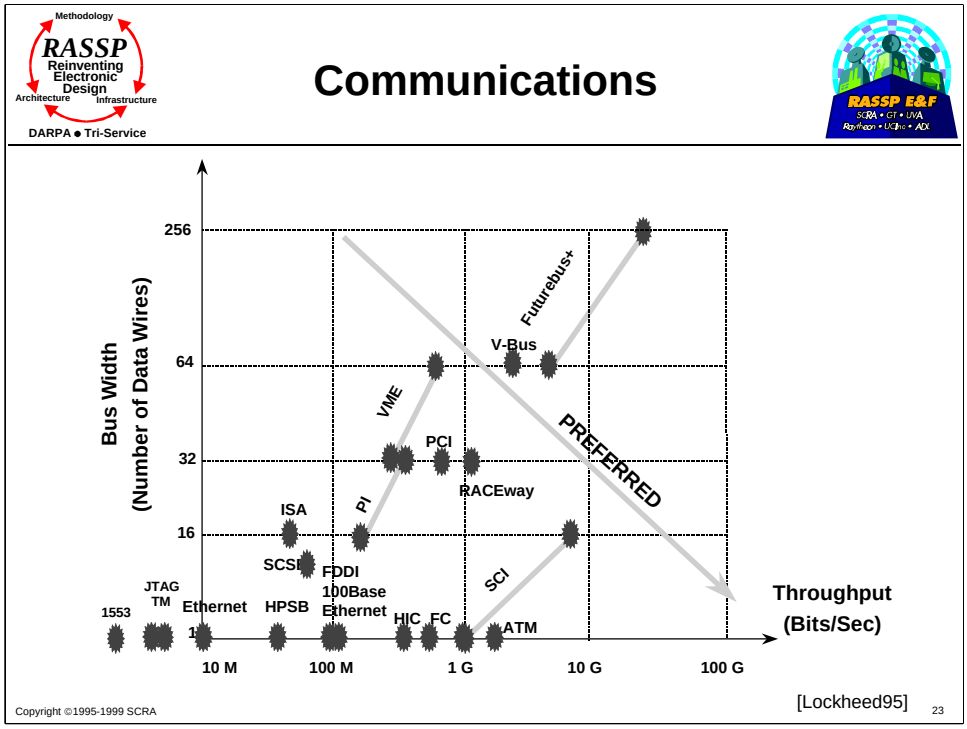
Digital Signal Processors are optimized for multiply and accumulate type operations (MAC). Some currently available choices include the TI TMS320 series, AD 21060, Motorola 56000 and 96000, and Intel i860.

Control processors are usually good at input/output and control logic, but lack the complexity of a high-end processor.

Data processors are general-purpose in intent and meet a large number of application areas.



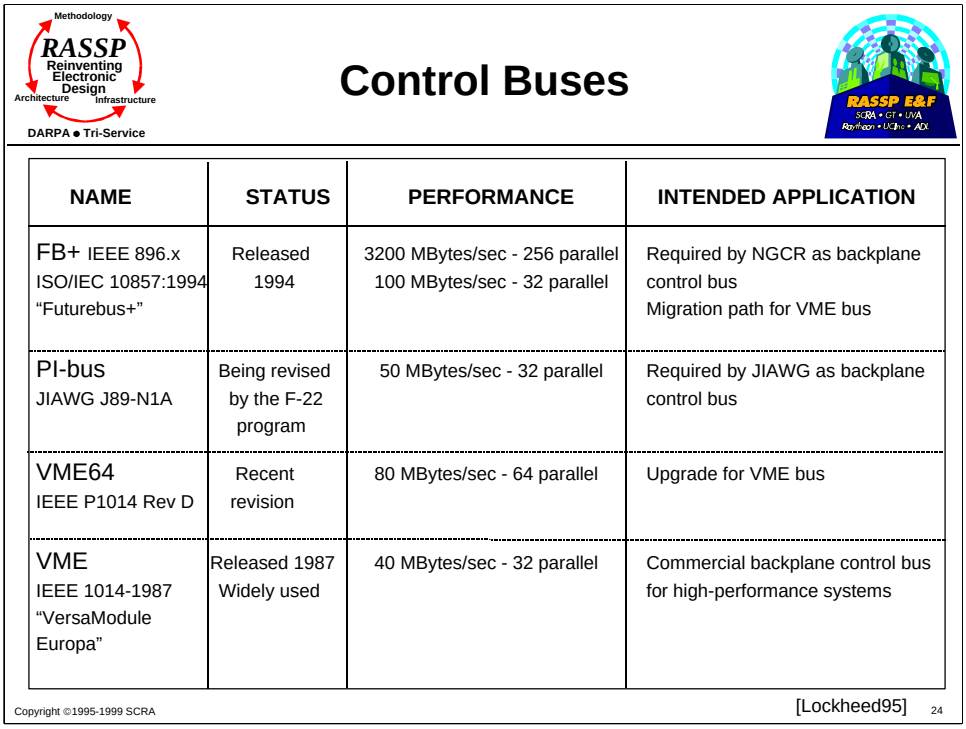
Some commonly known data processors are listed above. There capabilities change on a rapid basis and the numbers above will only reflect their nature over a short period of time (3 months or less).



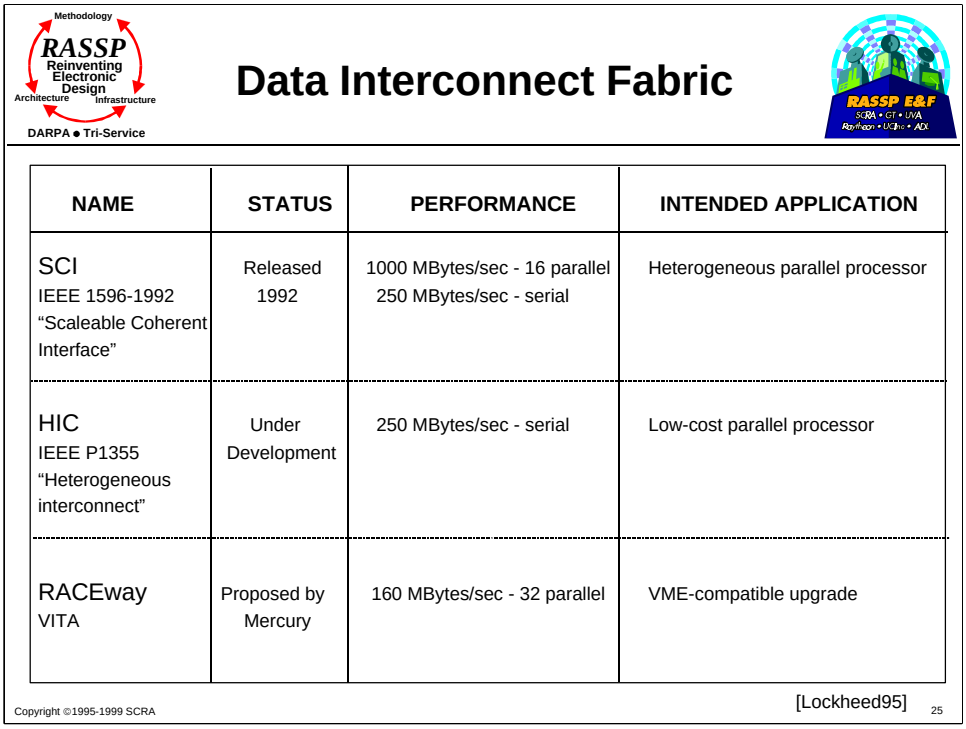
The communications between processors is accomplished by using buses of various types. These include direct point-to-point links; shared multi-drop buses; or networks made of links, buses, or switches.

They can be implemented in serial or parallel form. Some choices of open systems bus standards available today are shown in the chart above with their respective throughputs.

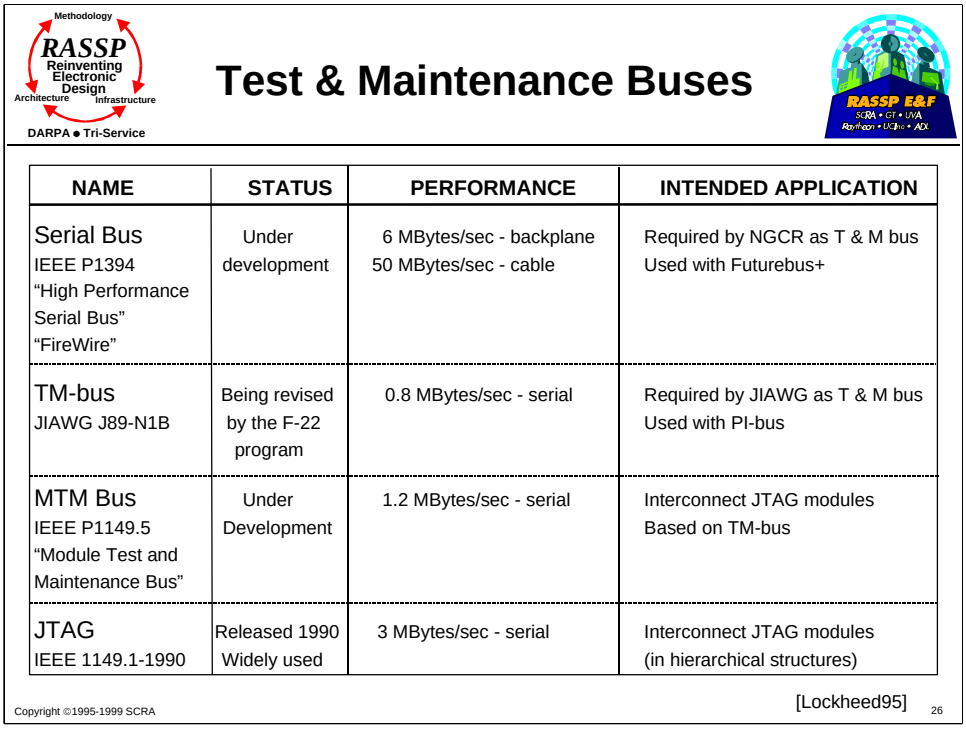
These buses can perform different functions such as control, data transfer, maintenance, I/O, and area networking.



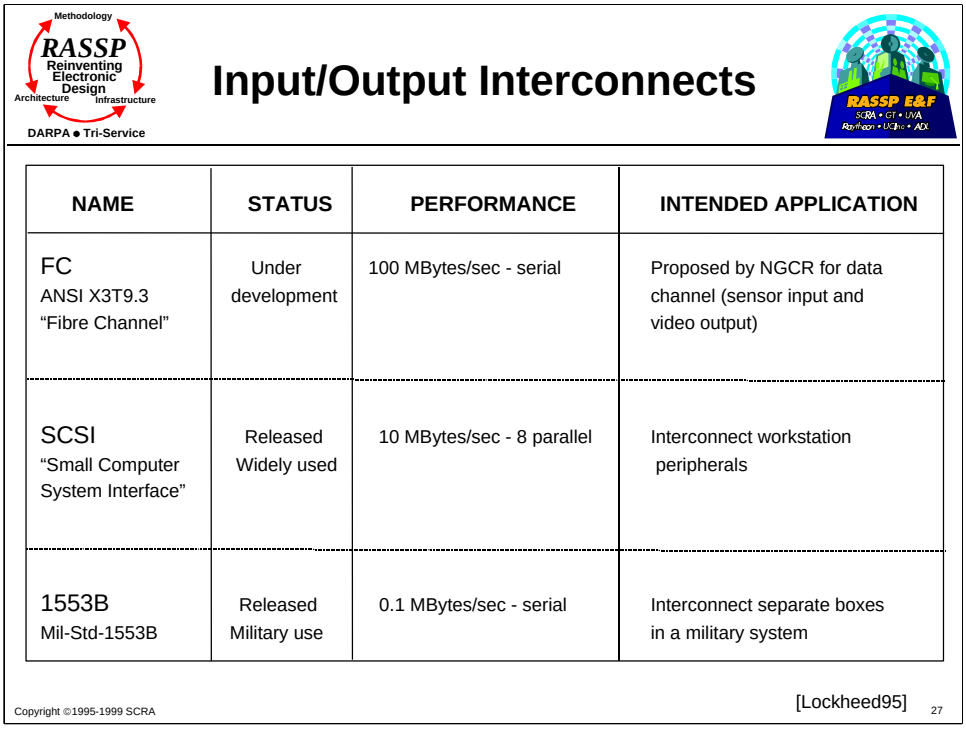
Control buses are typically used to allow multiple processors to inter-operate in a system through the exchange of commands and some data. In small systems with low throughput requirements, this may be the only bus required. Some of the bus choices are listed above.



Data interconnects are used to augment control buses for systems with a higher throughput. To achieve the higher speed, the data interconnect is usually implemented point-to-point with unidirectional links. Some choices are listed above.

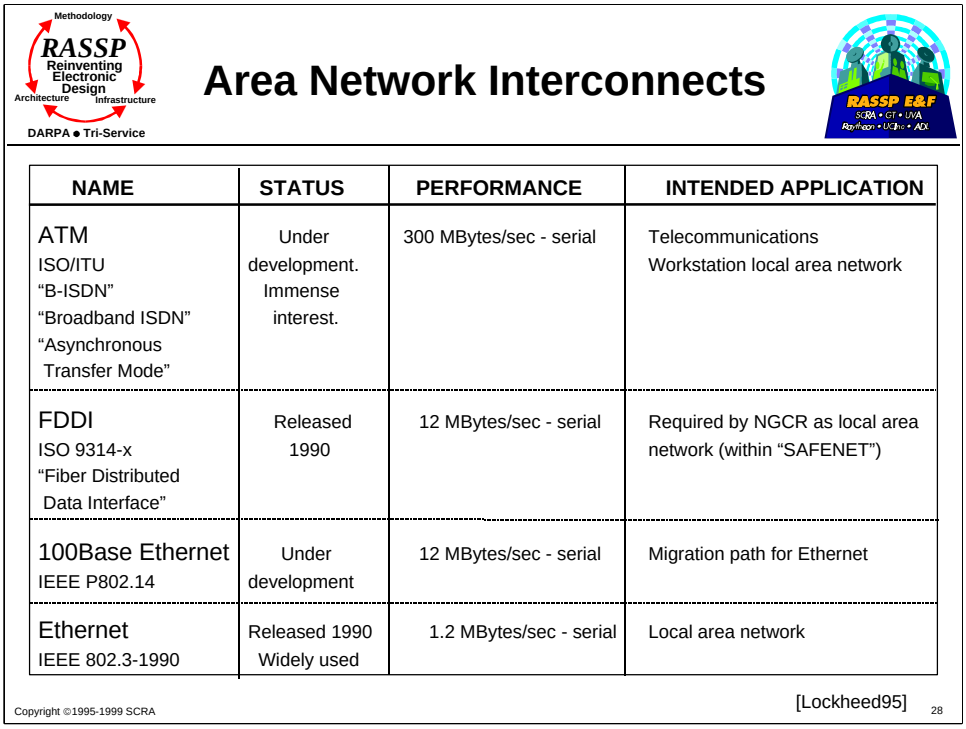


Test and maintenance buses are typically used to provide a minimally-intrusive path to every hardware module in the system to isolate and debug failures. It is typically serial and low speed. It can also be implemented in redundant form for mission critical fault tolerant systems.

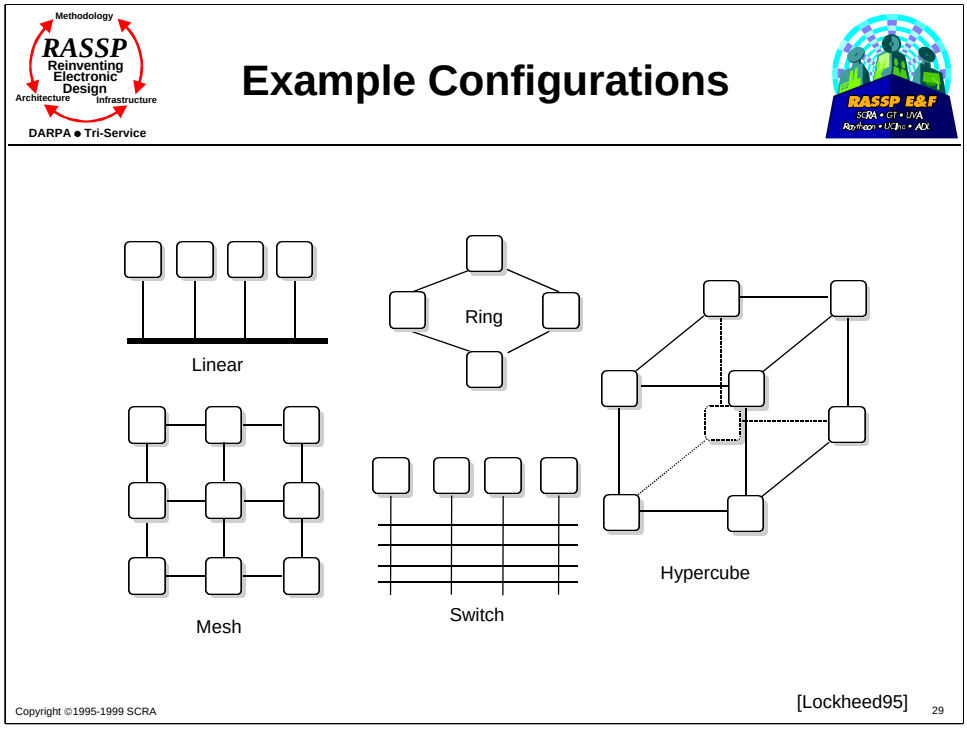


NAME	STATUS	PERFORMANCE	INTENDED APPLICATION
FC ANSI X3T9.3 "Fibre Channel"	Under development	100 MBytes/sec - serial	Proposed by NGCR for data channel (sensor input and video output)
SCSI "Small Computer System Interface"	Released Widely used	10 MBytes/sec - 8 parallel	Interconnect workstation peripherals
1553B Mil-Std-1553B	Released Military use	0.1 MBytes/sec - serial	Interconnect separate boxes in a military system

I/O interconnects are used to collect raw data from the sensors and distribute it to the processors or to the displays of the system. They are optimized to transfer large blocks of data with minimal concerns for error checking and flow control. Some choices are listed above.

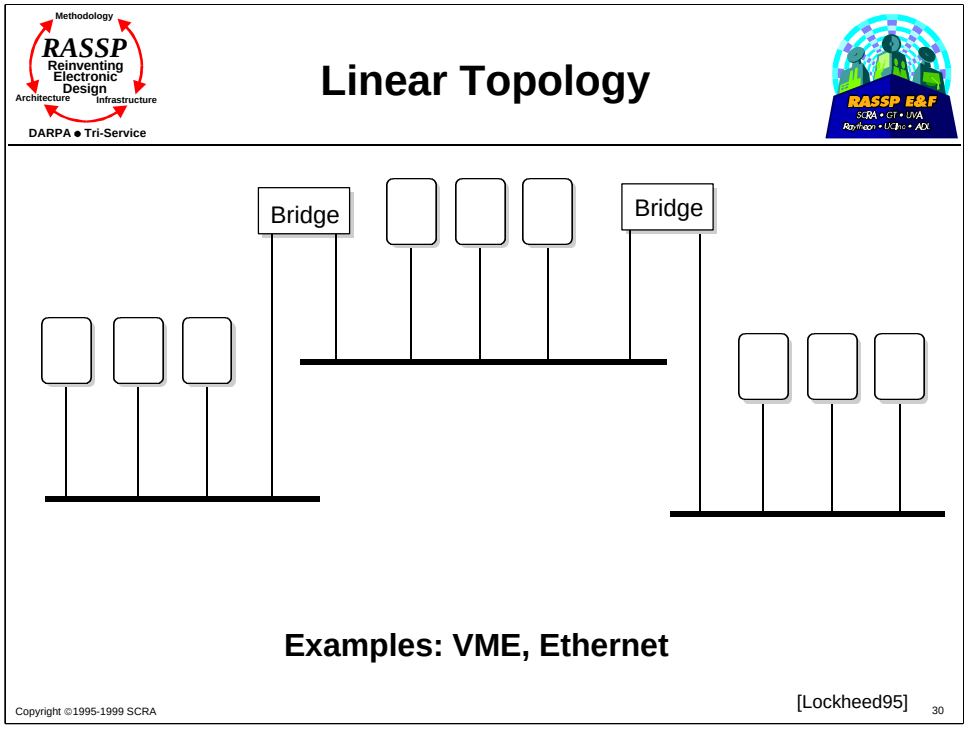


Area network interconnects are used to connect processing systems located in separate physical boxes. They are optimized for bursty traffic but in the future they must be able to handle isochronous traffic for multi-media applications. A list is included above.

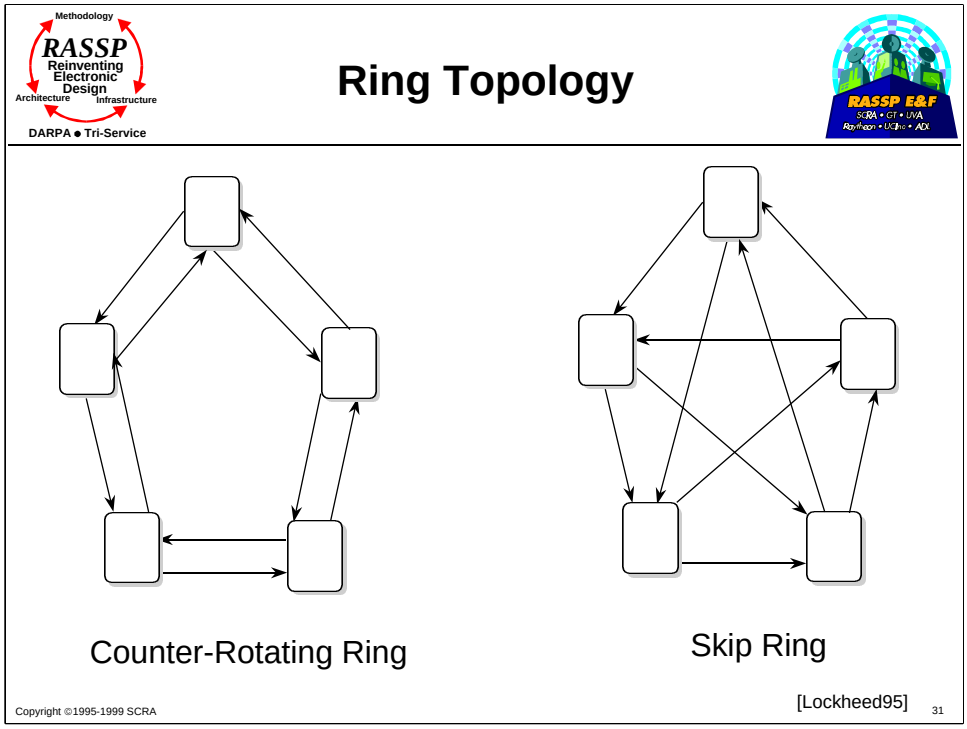


The configuration of an architecture describes how the computational and communications elements are arranged in the digital system.

The primary goal of this slide is to depict network topologies and the basic types: linear, ring, switch, mesh and hypercube. Not included are the connection of sensors to displays and the physical arrangement of the actual hardware.

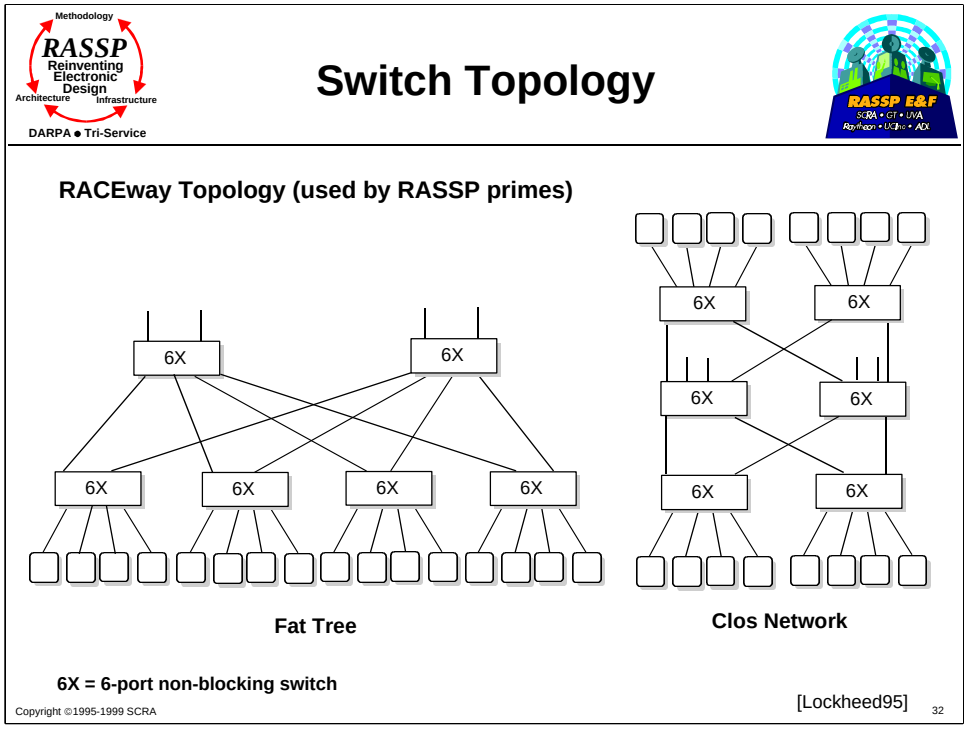


The linear (shared multi-drop) bus is the traditional topology when a computer bus is mentioned. VME, Ethernet, and 1553 are examples of this. A linear bus is inherently fault intolerant. Redundancy is usually implemented for mission-critical computers. Because a linear bus is shared, there usually is a limit to the number of nodes it can support. For example, Futurebus+ and PI-bus each have a maximum limit of 32 nodes. To connect additional nodes, a hierarchical topology as shown above must be configured. The bridge node serves this purpose.



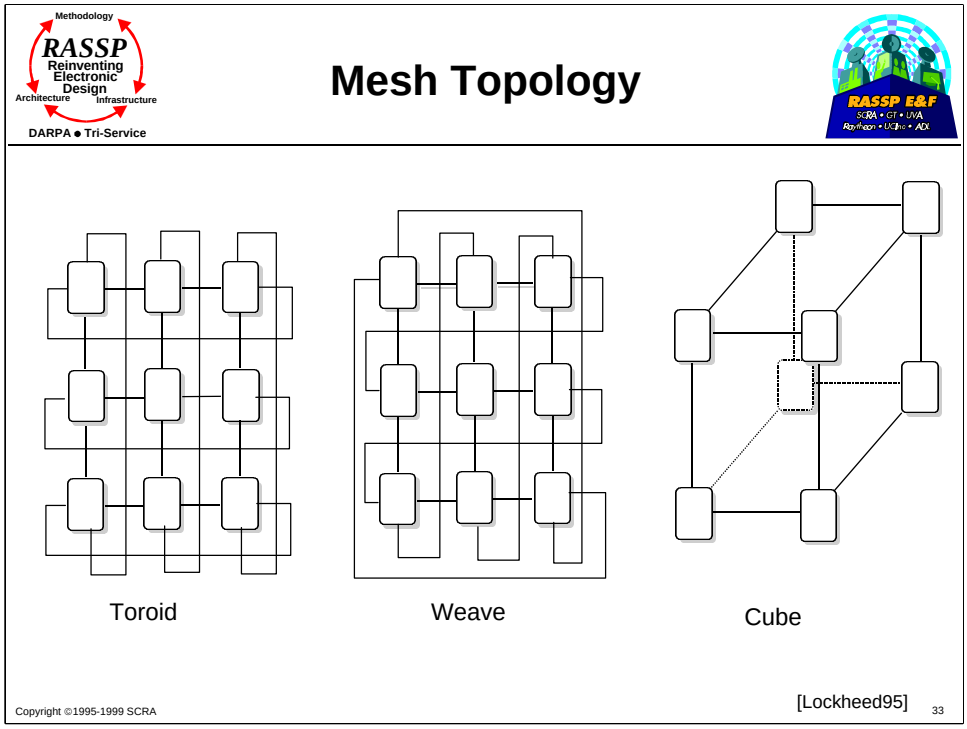
The ring topology is more scaleable than the linear and increases the total system bandwidth. It uses point-to-point links so the number of nodes are not limited and a higher speed can be obtained over a single link. This topology is also fault intolerant, so redundancy is usually built in.

To solve the intolerance problem, two popular ring topologies are the counter-rotating ring and the skip ring. With the counter-rotating ring, a failed link or node can be isolated from the network by its adjacent neighbor. With the skip ring, a failed link or node can be isolated by replacing the bad link with the skip link around the node.



The switch topology provides concurrent independent data paths between pairs of nodes and hence is high performance. It is also flexible because the connections between nodes can be changed dynamically as needed.

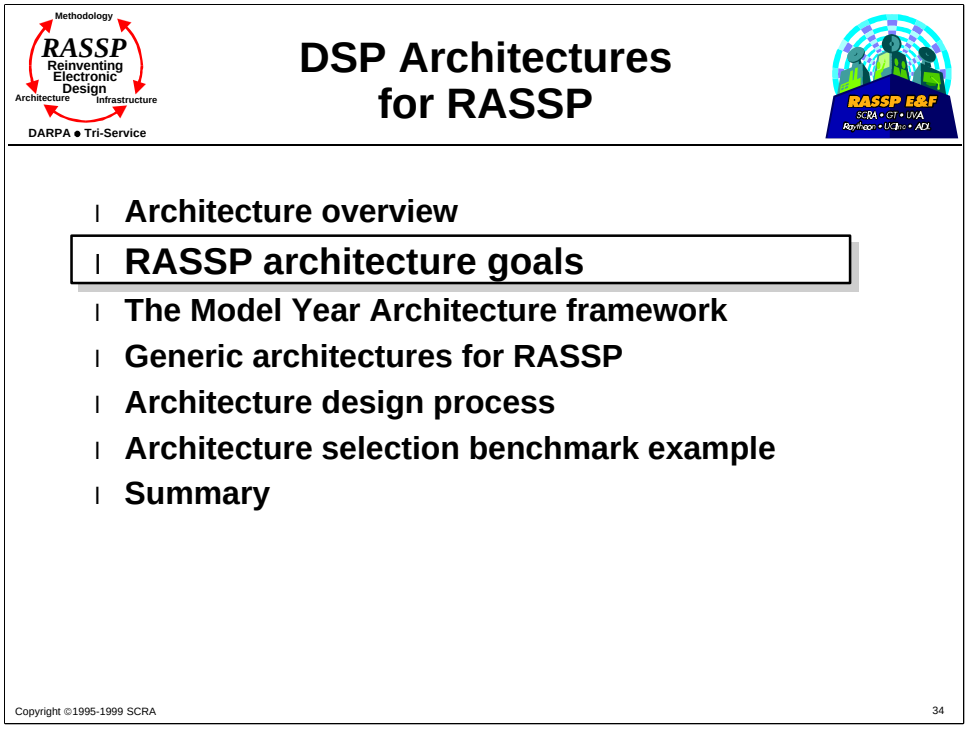
Typical switch types are crossbar and star topologies. The RACEway from Mercury Computer Systems uses the crossbar type. It has a six-port crossbar as its basic building block. It does not scale well because the number of required connections increase more quickly than the number of nodes.



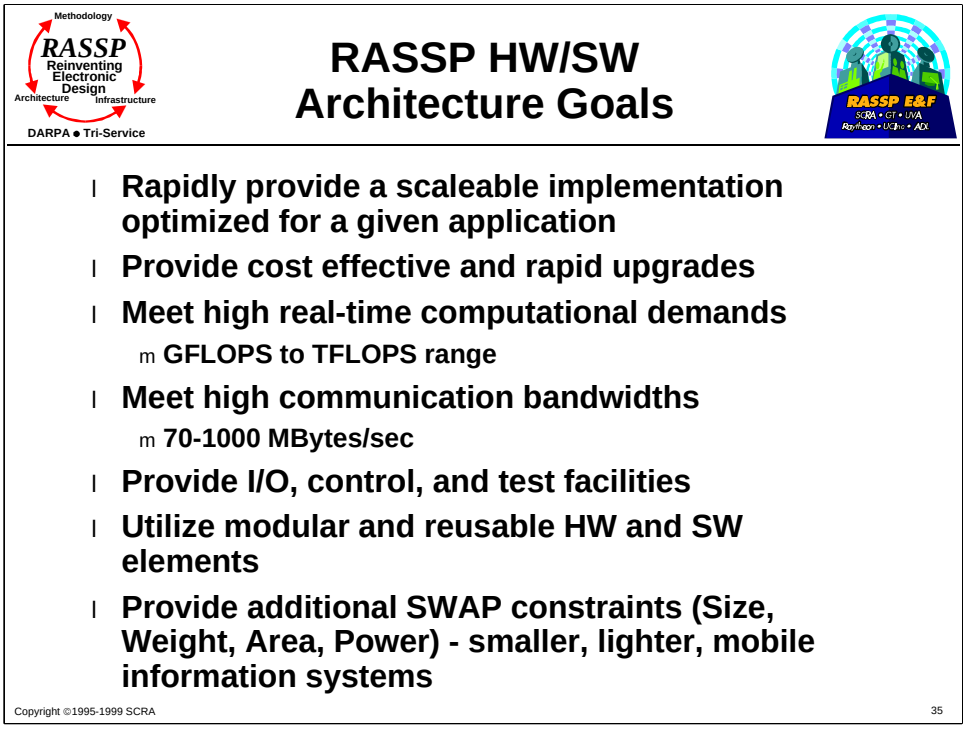
The mesh topology has become the popular choice for scaleable parallel processors and massively parallel processors. Its most important feature is that it scales well. The number of communications links increases linearly with the number of nodes. The disadvantage is that it requires routing of communication through intervening nodes and hence leads to delays and extra processing.

An example is the Intel *Paragon*.

Variants of the mesh topology include the toroid, weave, planar-4 and the N-Cube. The advantage of the N-cube is that there are fewer link hops between nodes. The disadvantage is that more links are required per node and the N-Cube is less scaleable. The planar-4 was mentioned in an earlier slide as part of the AOSP program.

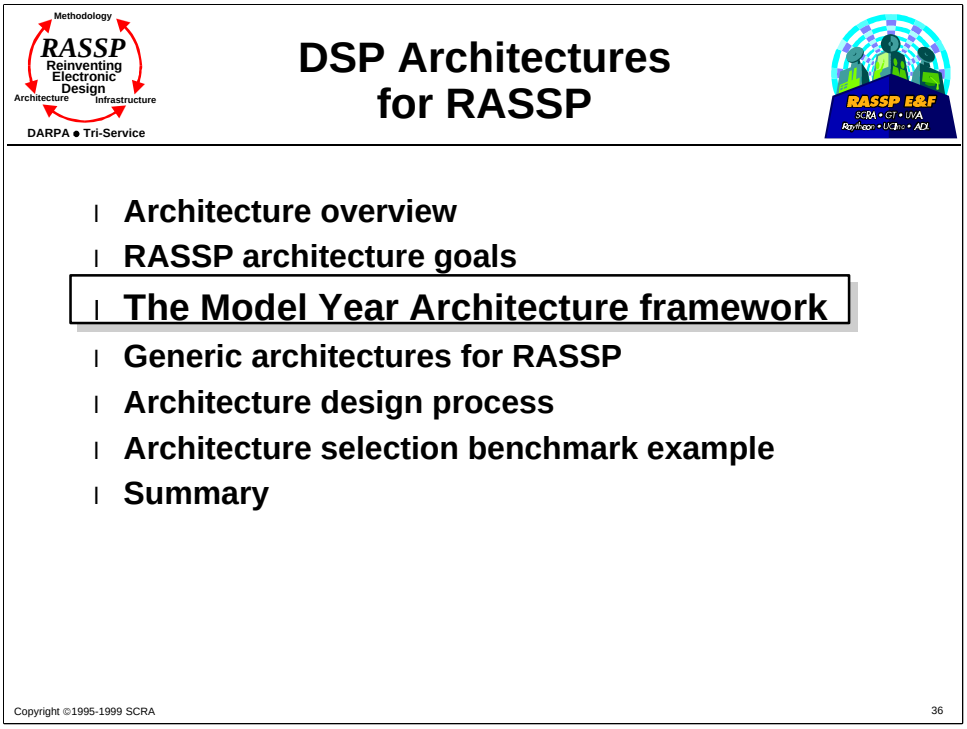


A list of RASSP goals for architectural design are now presented.

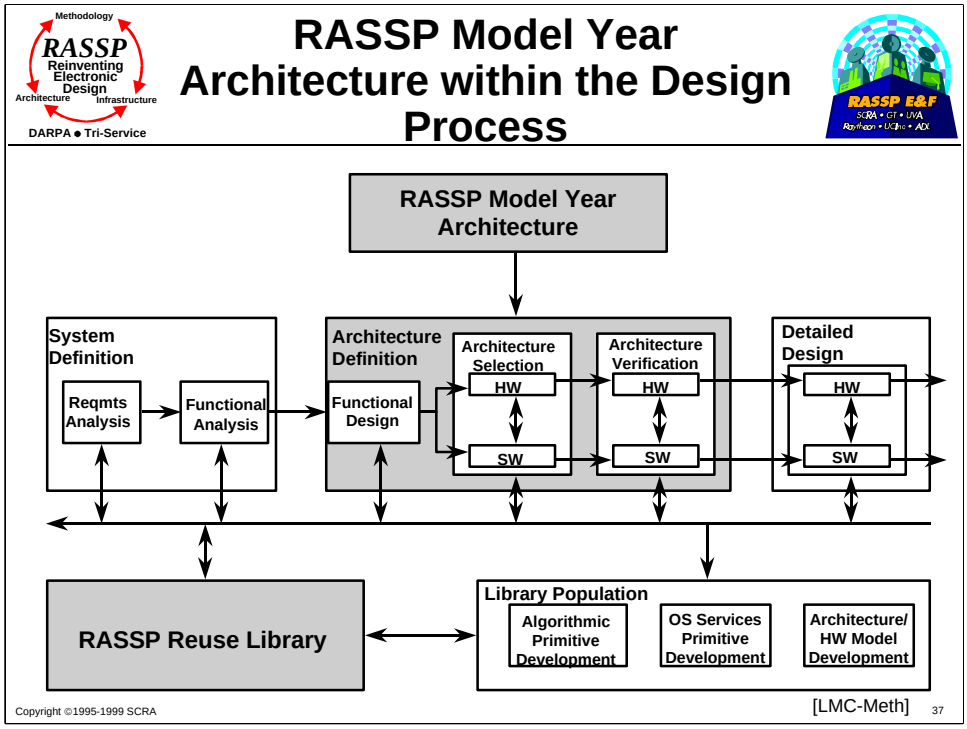


The architecture is what unifies a suite of HW and SW components into a system to accomplish a specified mission. It includes the form, structure, and interrelationships among the elements of the system and between the system and its environment.

The RASSP architecture's main focus is on embedded signal processing systems. The great complexity requires computational demands in the GFLOPS to TFLOPS range, using from 1-1000 processing elements (both general purpose COTS components and application specific designs) with high communication bandwidths. Requirements for cost effective and rapid upgrades are also a major focus of the program. Requirements of testability and fault-tolerance are usually specified. The architectures are typically open ended, modular, and provide I/O, control, and test facilities.



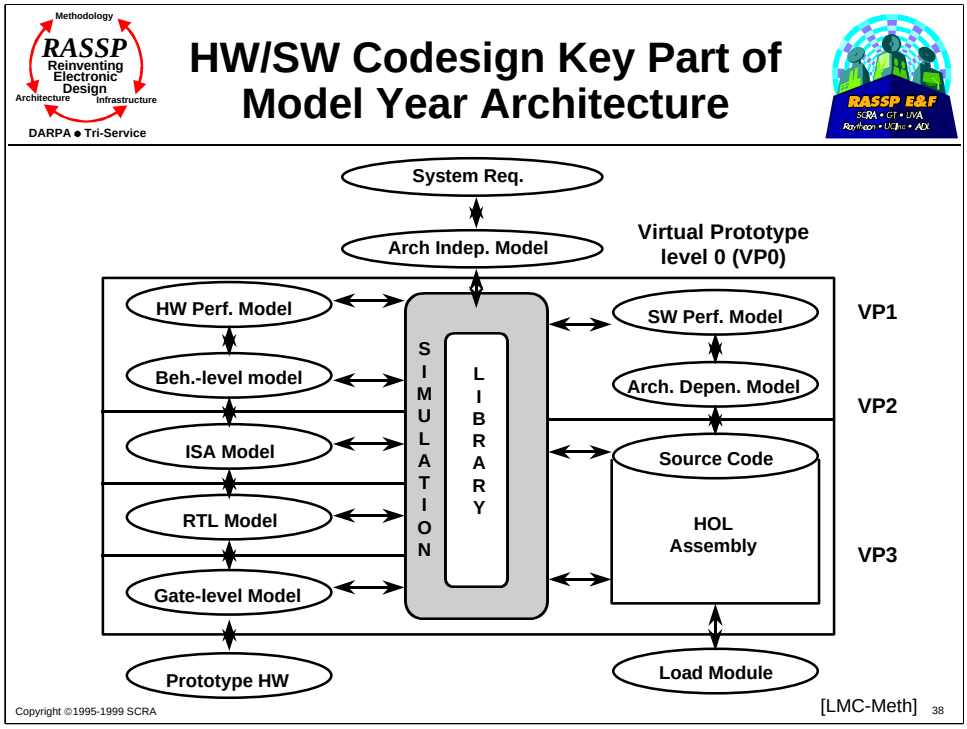
To support these goals, the concept of Model Year Architecture (MYA) will now be presented.



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:

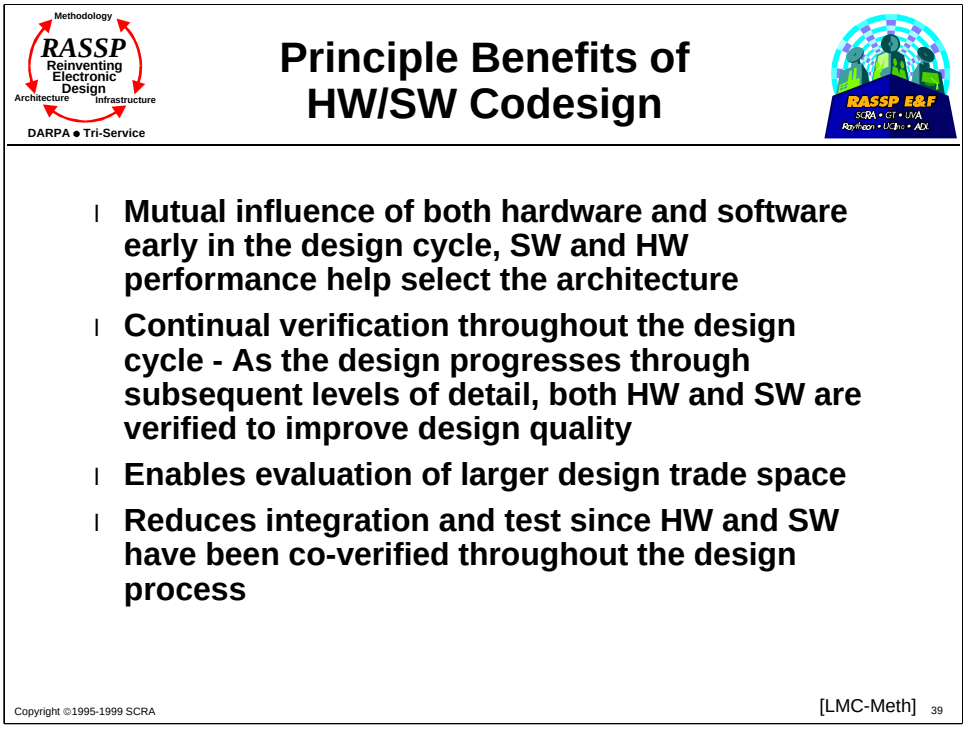
The architectures must be open to promote HW/SW upgradability and reusability in other applications

- 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) costs
- The architectures must facilitate continuous product improvement and substantial life-cycle-cost (LCC) savings in fielded system upgrades
- The RASSP 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 RASSP 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.

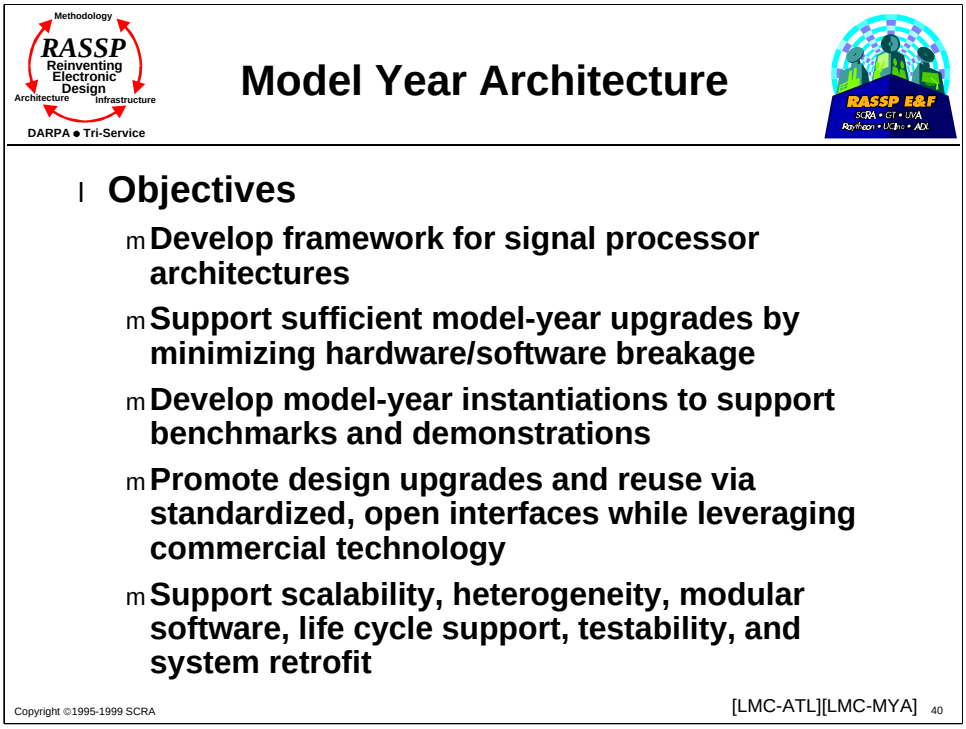


The RASSP design process is based on true HW/SW codesign and is no longer partitioned by discipline (e.g. HW and SW), but rather by levels of abstraction represented in the system, architecture, and detailed design processes. The above figure shows the RASSP methodology as a library-based process that 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.



HW/SW codesign is a major RASSP design methodology to aid in achieving the proposed 4X improvement in design of signal processing systems. The list above presents the principle benefits of the methodology.

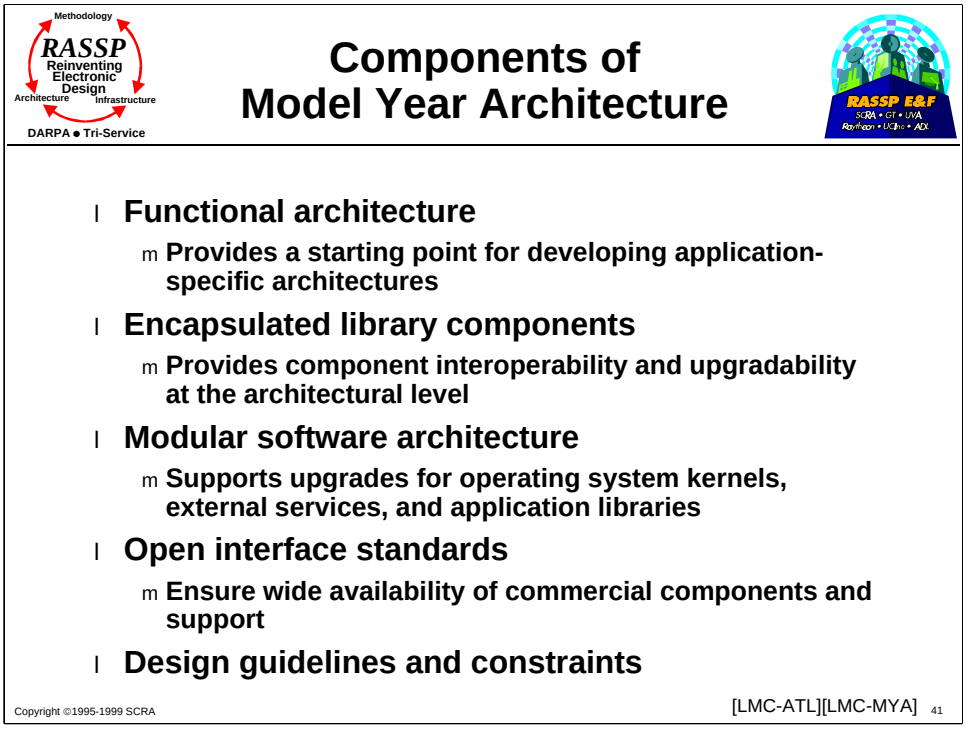


As technology is evolving faster than systems can be developed, the concept of Model Year Architecture allows the incorporation of new technologies to be inserted into the design as they appear.

The objective of the Model Year Architecture (MYA) is to develop a framework for signal processor architectures. The MYA should address the following issues:

Contribute to the 4X reduction in design cycle time required by RASSP

- Provide life cycle support
- Provide scalability to support changing mission scenarios and different deployment environments
- Support heterogeneity in the design process by providing cost effective implementations of functions with a wide range of performance requirements
- Provide flexible interfaces to a wide range of subsystems
- Utilize modular software in the form of reusable components and support upgrades to operating systems, services, and libraries.
- Support hardware upgrades
- Provide for testability in the design process and detect and isolate faults with high probability
- Support for RASSP signal processor retrofit into non-RASSP (legacy) systems
- HW and SW elements within the library of components are encapsulated by functional wrappers, which add a level of abstraction to hide implementation details and facilitate efficient technology insertion



The basic elements of a MYA are listed above.

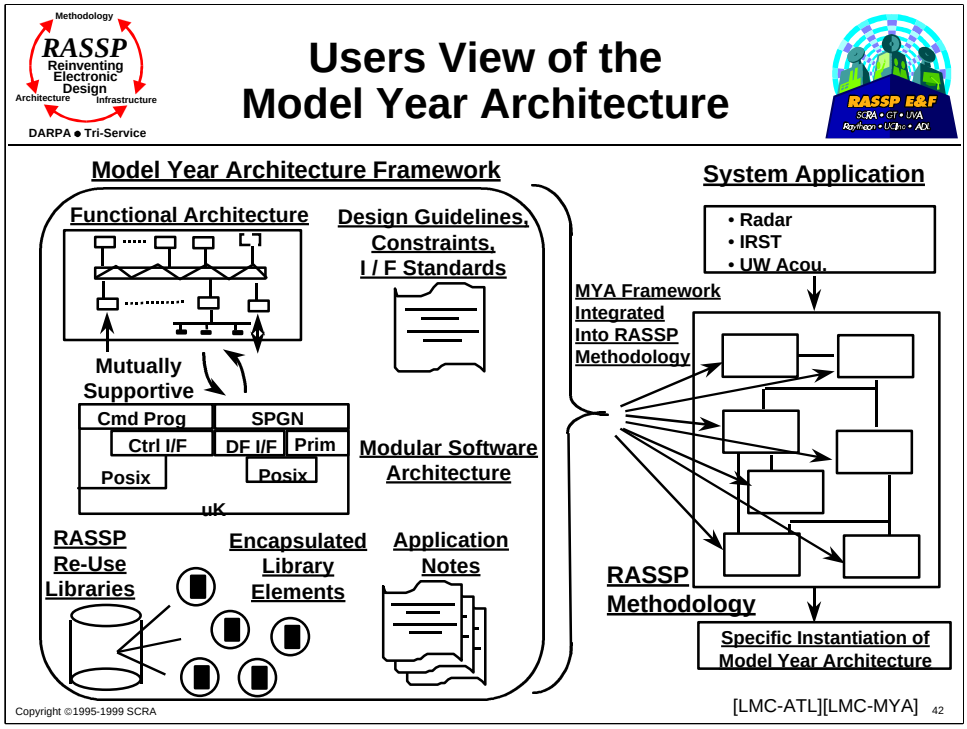
The functional architecture defines the necessary components and the manner in which their interfaces must be defined to ensure that the design is upgradable and facilitates technology insertion. As such, the functional architecture is a starting point for developing solutions for an application-specific set of problems, not a detailed instantiation of an architecture.

An important aspect of the functional architecture is that application-specific realizations of a signal processor are embodied in the proper definition and use of encapsulated library elements. Encapsulation refers to additional structure added to otherwise raw library elements to support the functional architecture and ensure library element interoperability and technology independence.

A modular software architecture simplifies the development of high-performance, real-time DSP applications allowing the developers to easily describe, implement, and control signal processing applications for multiprocessor implementations. It supports upgrades for operating system kernels, external services, and application libraries.

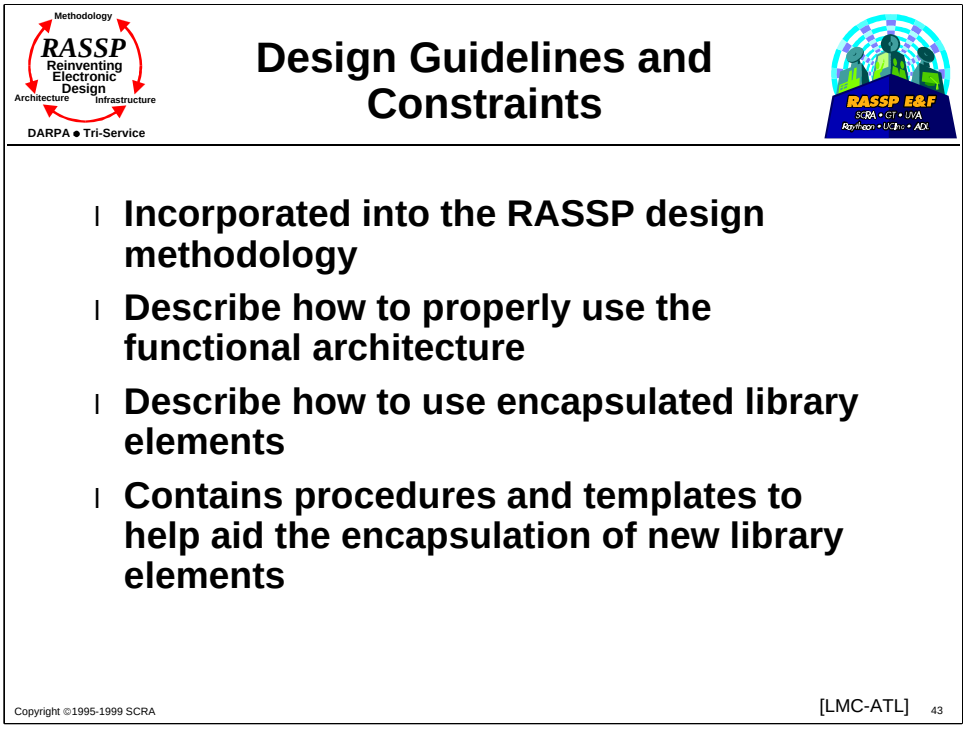
Open interface standards are used to help ensure interoperability between components and ensure a wide availability of commercial components and support.

Design guidelines and constraints are provided for general architectural development, such as how to use the functional architecture framework, use of encapsulated libraries and procedures and templates to encapsulate new library components.

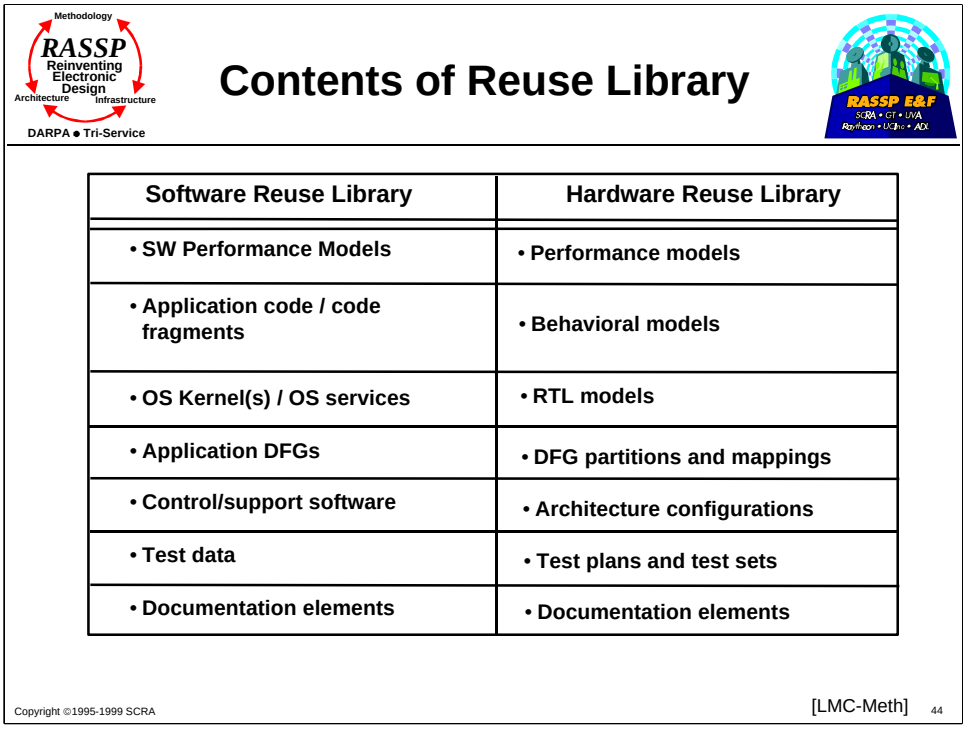


This diagram illustrates the MYA framework as inserted in the RASSP design methodology. Synergism between the MYA framework and the RASSP methodology is required, because all areas of the methodology, including architecture development, HW/SW codesign, reuse library management, HW synthesis, target SW generation, and design for test are impacted by the MYA framework.

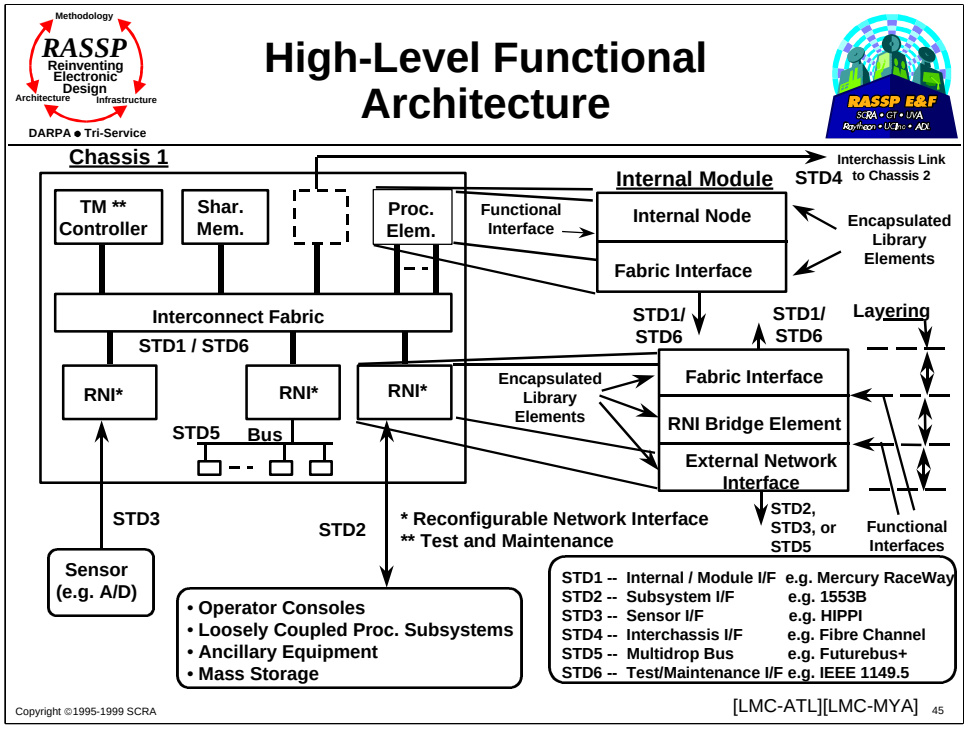
Specific instantiations of the MYA are incorporated into the RASSP methodology to aid in the design of systems.



As part of the MYA framework, an important feature is the capture of guidelines of various workflows in the design process and incorporate them into the RASSP methodology. Guidelines are also described for encapsulating new elements to be placed in the design library.



The contents of the hardware and software component reuse library has models and data at various levels as shown in the chart above. These models support concurrent codesign throughout the selection and verification process. The reuse library drives both the architecture synthesis and the software synthesis processes in an integrated fashion.



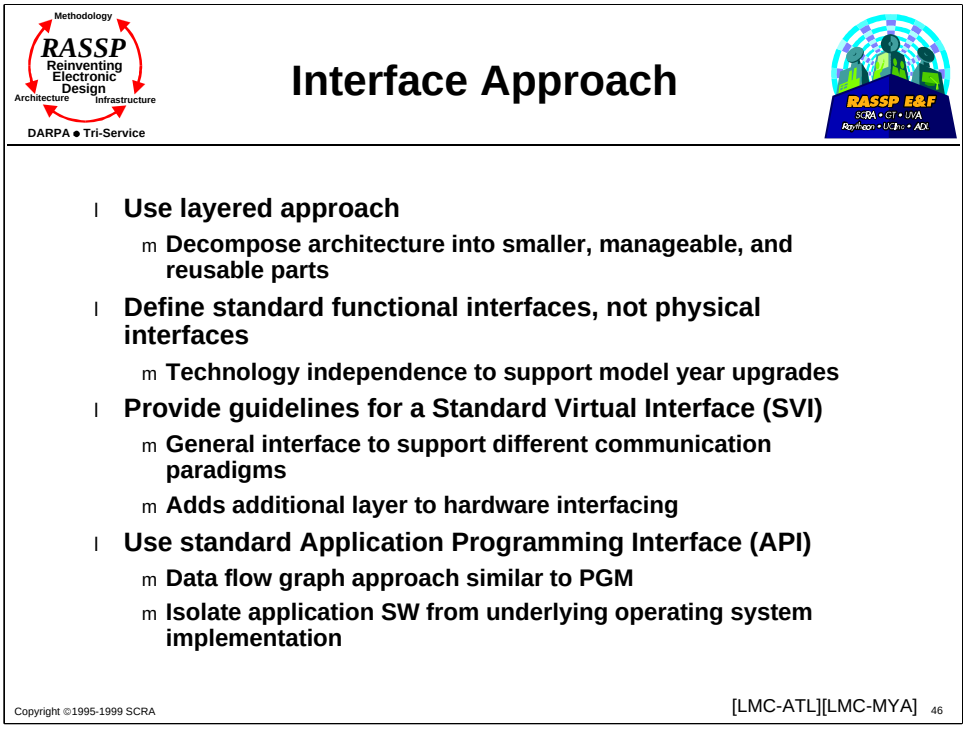
The functional architecture defines the necessary components and the manner in which their interfaces must be defined to ensure that the design is upgradable and facilitates technology insertion. The functional architecture is the starting point for developing solutions for an application-specific set of problems, not a detailed instantiation of an architecture. The functional architecture DOES NOT specify the topology or configuration of the signal processing architecture.

The functional architecture specifies a high-level framework for launching application-specific architecture development. Architecture-level reuse element classes are provided. Open interface candidates for the interconnect fabric, sensor, and interchassis interfaces are provided for selection. The functional architecture also specifies the test methodology to be used for design.

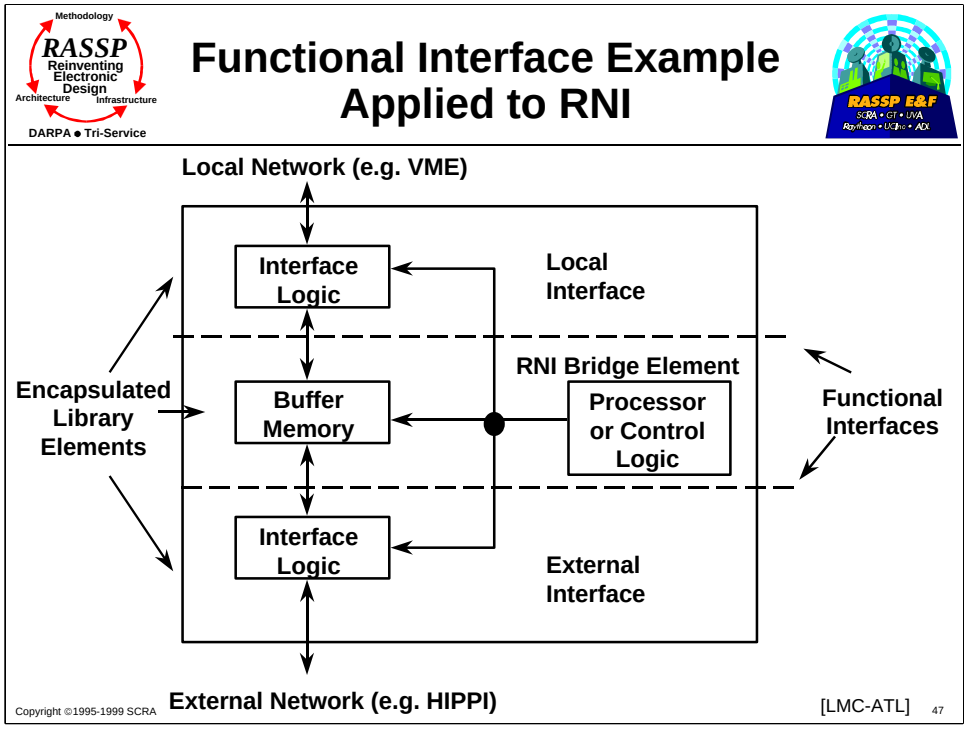
The STDx demarcations illustrate the types of interfaces found in various portions of the functional architecture.

The Reconfigurable Network Interface (RNI) is divided into three logical elements: 1) Fabric interface, 2) External network interface, and 3) Bridge element. The fabric and external interfaces implement the specific protocols to the elements being interconnected, for example a High-speed Parallel Port Interface (HIPPI) could be used for the external interface and a VME interface can be used for the fabric interface. The bridge element, which typically consists of a buffer memory and a controller implemented via custom logic (e.g. FPGA, ASIC) or a programmable processor, performs the actual bridging function. The buffer memory facilitates asynchronous coupling and flow control between the two networks, while the controller coordinates data transfers. The three logical elements of the RNI are implemented as encapsulated library elements that serve to isolate changes resulting from upgrades. For example, the VME interface can be replaced with an encapsulated SCI interface.

The processing element is also encapsulated so links to the internal interconnect fabric is made easier, reusable and provides a better route to upgradability.

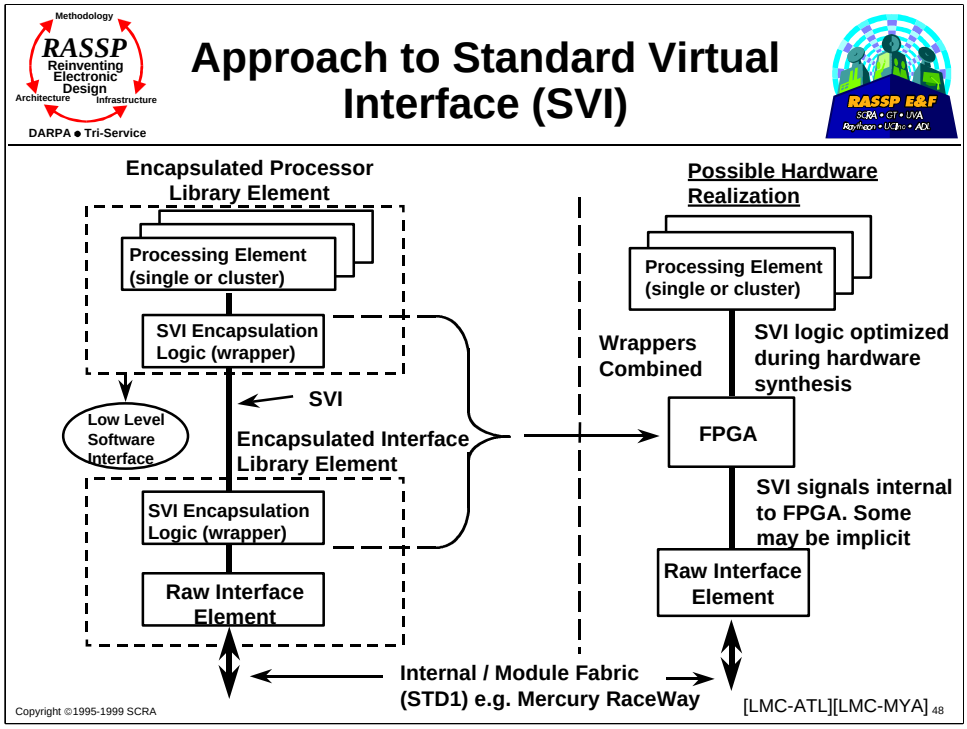


A layered approach can be used for handling the interfacing between components. This decomposes the architecture into smaller, manageable, and reusable parts. A standard functional interface was defined supporting technology independence and model year upgrades. The interface is implemented using a Standard Virtual Interface (SVI) which is general enough to support different communication paradigms and adds an additional layer to the hardware interfacing. SVI will be discussed in more detail in the following slides. An Application Programming Interface (API) is used to isolate the SW from the underlying operating system implementation.

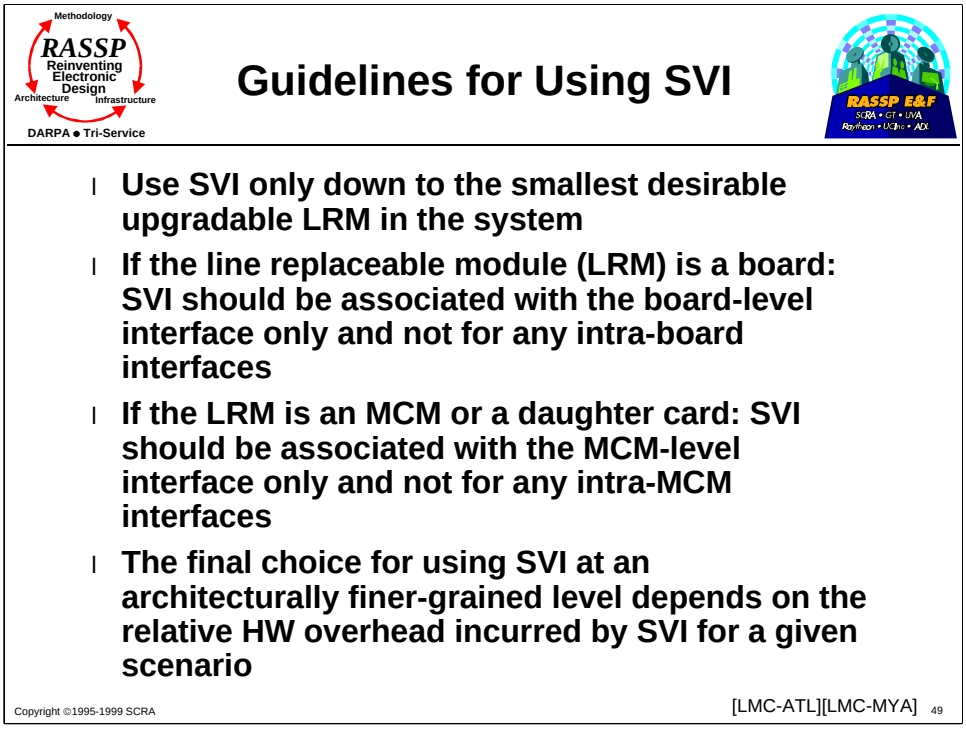


The above diagram illustrates an application of a functional interface at the hardware level for a construct called an Reconfigurable Network Interface (RNI). The RNI is divided into three logical elements: 1) local interface, 2) external interface, and 3) bridge element. The local and external interfaces implement the specific protocols to the elements being interconnected, in this example a HIgh speed Parallel Port Interface HIPPI and VME interface. The bridge element, which typically consists of a buffer memory and a controller implemented via custom logic (e.g. FPGA, ASIC) or a programmable processor, performs the actual bridging function. The buffer memory facilitates asynchronous coupling and flow control between the two networks, while the controller coordinates data transfers.

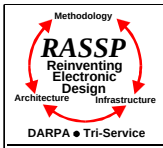
The three logical elements of the RNI are implemented as encapsulated library elements that serve to isolate changes resulting from upgrades. For example, the VME interface could be replaced by another encapsulated interface, such as SCI, with little or no impact on the HIPPI HW and SW.



SVI encapsulates library elements to support reusability and rapid upgradability. The interconnection of library elements is done by connecting their SVIs. A protocol is defined for the SVI to SVI interface. Each library element needs the SVI to operate in this environment. A possible hardware realization is shown above. The SVI interface is implemented on an FPGA, or an equivalent technology, using optimized hardware synthesis tools.



SVI can be used at any encapsulation level (LRM, MCM, component), but should be used where it makes the most sense. Considerations of HW overhead and reusable design elements should be taken into account.

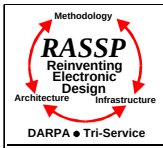


Applications of SVI

- **Internal modules**
  - **Use for processing nodes and shared memory nodes**
  - **Use between internal node and node-to-interconnect interface**
  - **Allows "plug and play" interoperability between internal nodes and node-to-fabric interfaces**
- **Reconfigurable Network Interface (RNI)**
  - **Implements system-level interfaces: sensors, loosely coupled processor subsystems, operator consoles, etc.**
  - **Implements bridge between internal interconnect fabric and particular system-level interfaces**
  - **Contains three logical components: fabric interface, external network interface, and RNI bridge element**

SVI can be applied at the module level to encapsulate processing and shared memory nodes, at interconnect boundaries to allow for plug and play interoperability between internal and interface elements, etc.

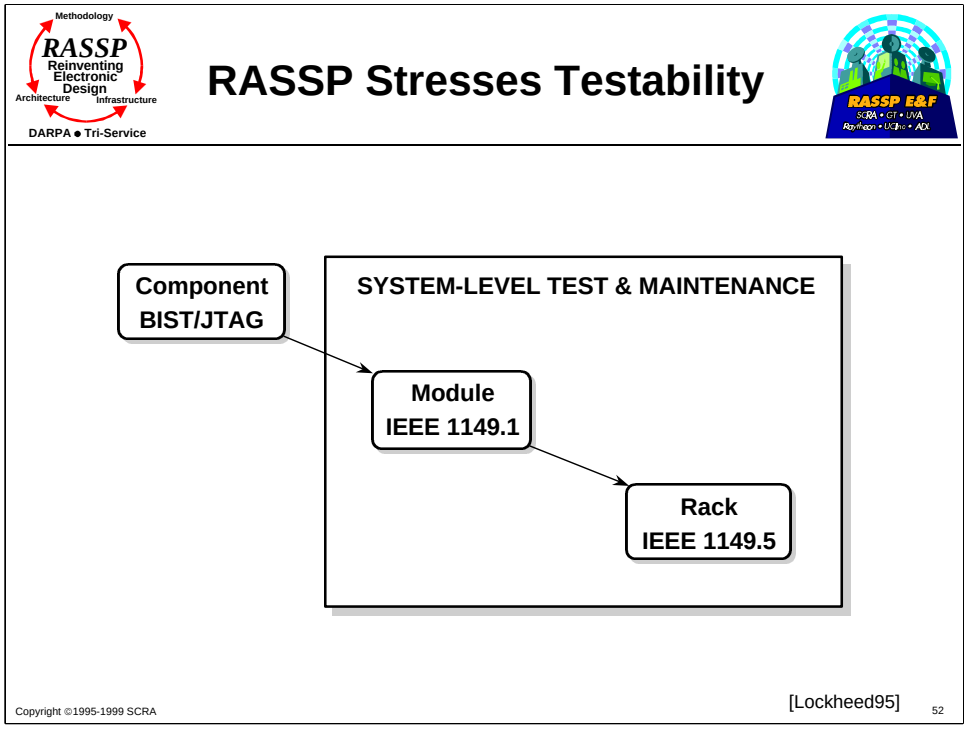
The choice of encapsulation depends on issues of supportability, design overhead, etc..



Implications of Layering

- **Can cause performance penalties and unacceptable hardware or software growth**
- **Resolution**
  - **Trade off performance vs. functionality**
  - **Overhead reduction techniques -> SVI**
- **Use layering judiciously; only isolate important architectural elements subject to upgrades/ technology insertion**
- **Some layering overhead must be accepted**
  - **Tradeoff to realize greater benefits of design/life cycle and cost reduction**

Layering can cause performance penalties due to the additional HW overhead. This can be acceptable if the layering is chosen judiciously and only important architectural elements are isolated where possible technology insertion can occur.



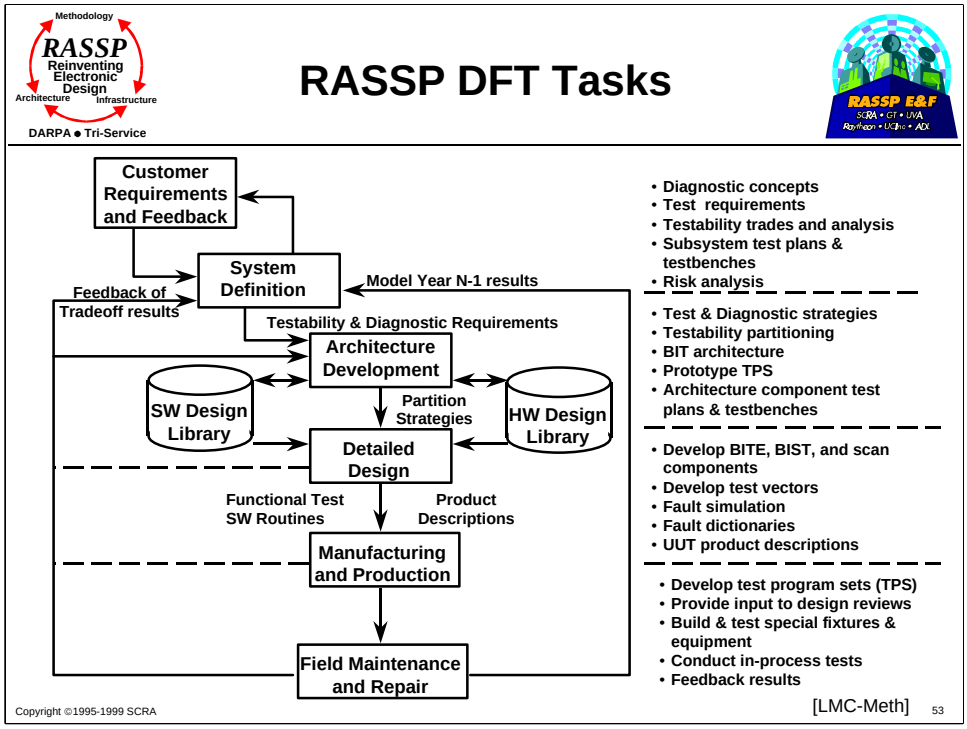
To provide an integrated diagnostic capability, a structured test approach is required for the various levels of system integration: component, module, and box (rack).

Component: High degree of fault coverage (>95%) should be provided. BIST should conform to the IEEE 1149.1 standard (JTAG). Many IC vendors now provide for it.

Module: IEEE 1149.1 boundary scan architecture is used to detect interconnect faults between components. Modules are designed with built-in-test (BIT) to detect, diagnose, and isolate module faults. This is usually controlled by a BIT controller.

Rack: A test and maintenance (TM) controller manages system-level testing, including the initiation of BIT for each of the modules. IEEE 1149.5 proposes a TM bus standard.

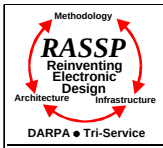
System Test requirements may vary significantly based on the application.



Designers of complex systems cannot afford to postpone test considerations until the final stages of design and still deliver a quality product. Testable systems are not a natural product of a design team unless BIT and scan features are included up front and knitted together seamlessly throughout the system hierarchy.

To ensure consistency between levels of the design hierarchy, a system-level test architecture and strategy must be developed and passed down to each level. The DFT methodology uses the hierarchical partitioning to manage test development complexity and to provide solutions to the incorporation of COTS components.

The RASSP design process is shown above with specific information flow and activities relative for design for test. A prime goal of the RASSP methodology is to eliminate design modification efforts late in the design cycle, including those to correct testability problems. VHDL and WAVES are used, as appropriate, throughout the methodology to capture and refine test and DFT-related information.

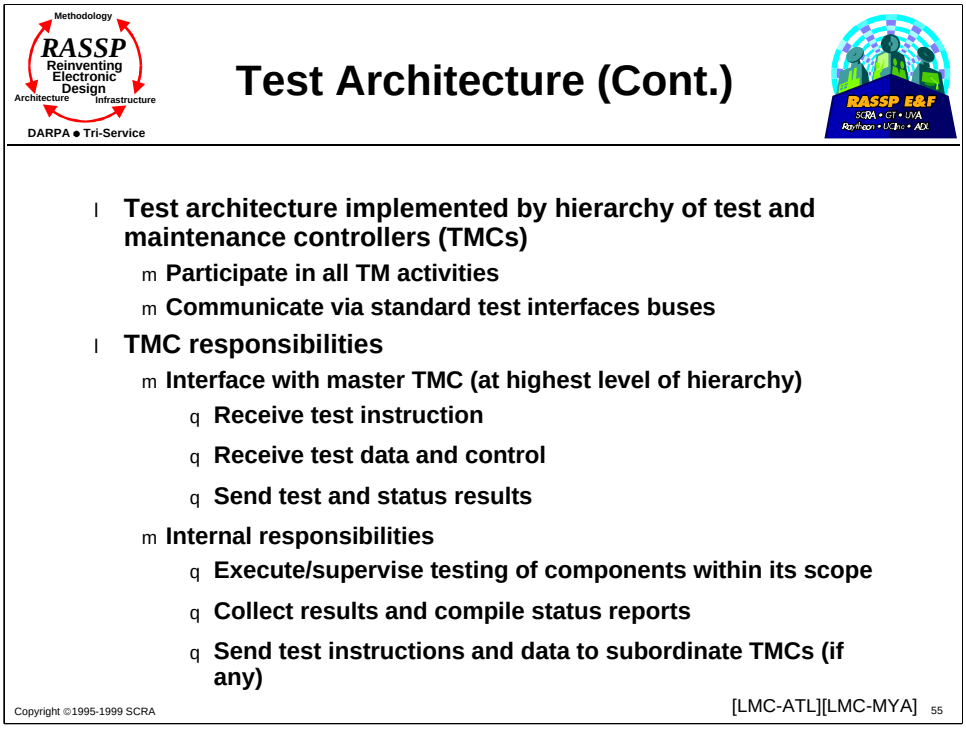


Test Architecture

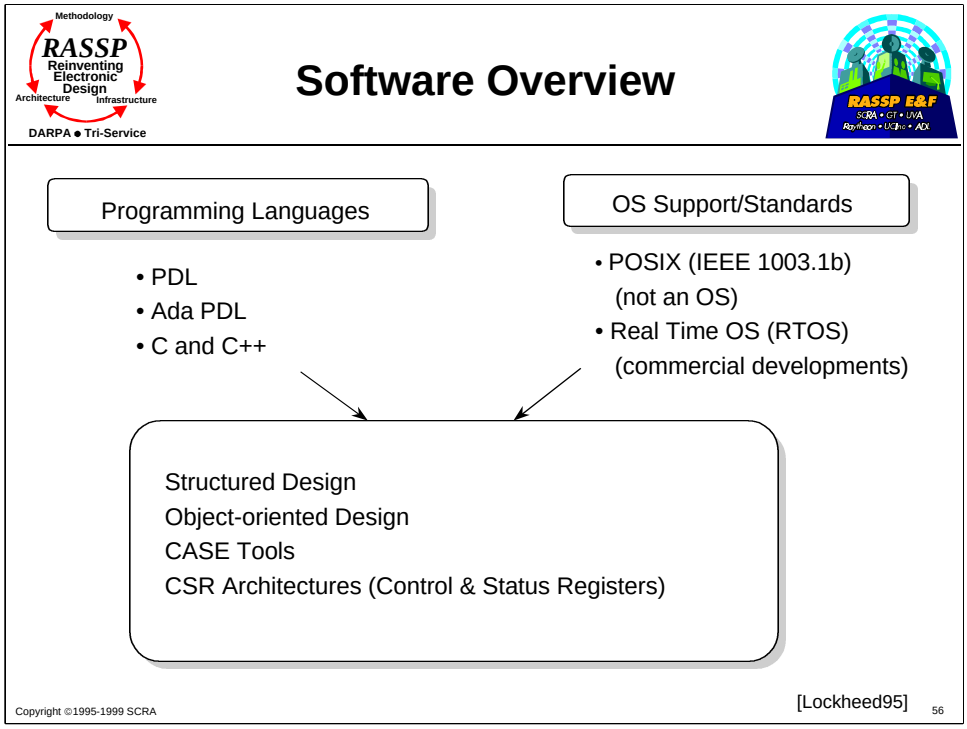
- **Required as an integral part of MYA: Formal structure required to ensure testable design**
- **Use of standard test interfaces as required augmentations to signal/control interface to chips, module, and systems**
- **Test architecture parallels system architecture hierarchy**
  - **Processor system**
  - **Chassis**
  - **Sub-chassis/functional group**
  - **Printed circuit board**
  - **Line replaceable module (LRM)**
  - **Multichip assembly**
  - **Chip**
  - **Functional or logic block**

The test architecture is an important part of the MYA. Standard test interfaces should be augmented to the signal and control interfaces to chips, modules, and subsystems.

The test architecture hierarchy should parallel the system architecture hierarchy incorporating elements at the system level, chassis level, all the way down to the functional or logic block.



Test and maintenance controllers (TMCs) should be used to implement the hierarchy and should communicate via standard test interface buses. The test and maintenance controllers have the responsibility to interface with the master TMC, collect results, and compile status reports.



Attributes of software are considered **architectural** when they express relationships between HW and SW that contribute to long term capacity for change. They are considered **design** when they are implementation specific.

The SW architecture must make provisions for several levels of control and task management. Open systems protocols should be considered. The architecture also must provide for an orderly flow of data throughout the system.

Operating System: An open systems approach should be selected for greater resistance to system obsolescence. POSIX provides for standard interfaces. They are called the Operating System Interface (OSI) and the Application Program Interface (API). Use of the POSIX standards should allow SW to be portable across similar platforms.

Programming Language: PDL (Program Design Language) is a mixture of language statement and control structures. It has the following characteristics:

States design in a easily read fashion.

- It allows concentration on the design logic rather than implementation details.
- Documentation can be done concurrently.
- It is convertible to a high order language (HOL).

Ada is the official language of choice for large complex SW projects of the U.S. Govt. Ada 95 provides for object-oriented features. C and C++ code can be used when COTS technology is specified for use.

Structured Design: A SW development methodology that follows a hierarchical structure of SW module development and test.

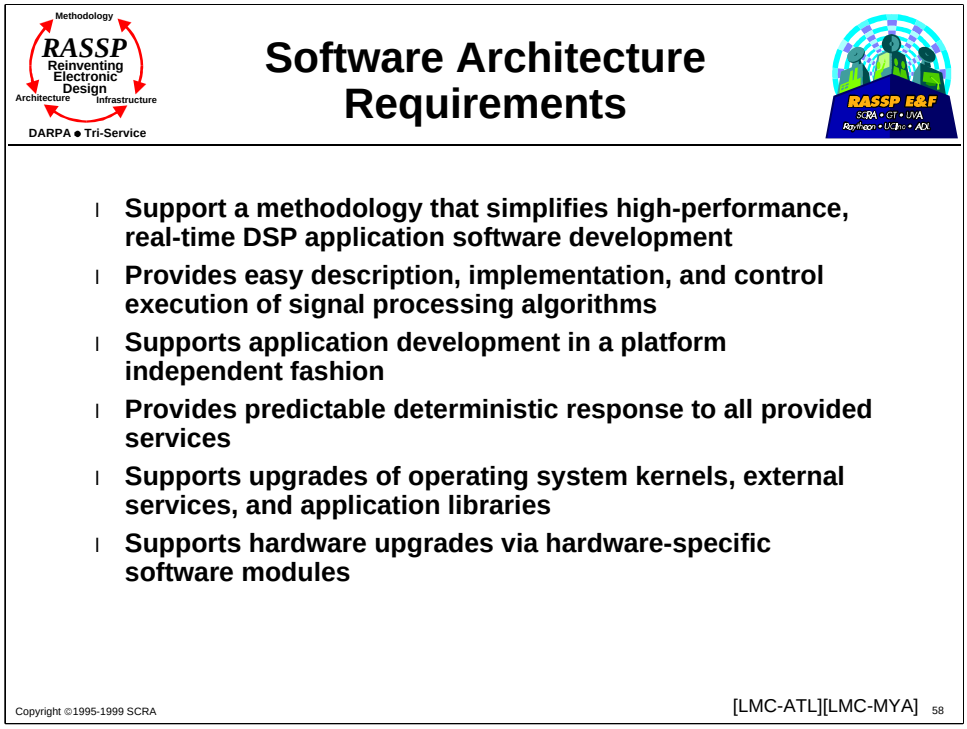
Object-oriented Design: Results in a more modular design. There are three phases to this approach. One, Object Oriented Analysis (OOA), two, Object Oriented Design (OOD), and last, Object Oriented Programming (OOP). OOA and OOD are embedded in the CASE tools such as Cadre's *Teamwork,* and IDE's *Software Through Pictures.*

CSR (Control and Status Registers) architectures can be used to identify the module, select a working subset of its performance capabilities, enable BIST, and record the health status history. IEEE 1212-1991 specifies a standard CSR architecture.

Software Architecture Process Goals

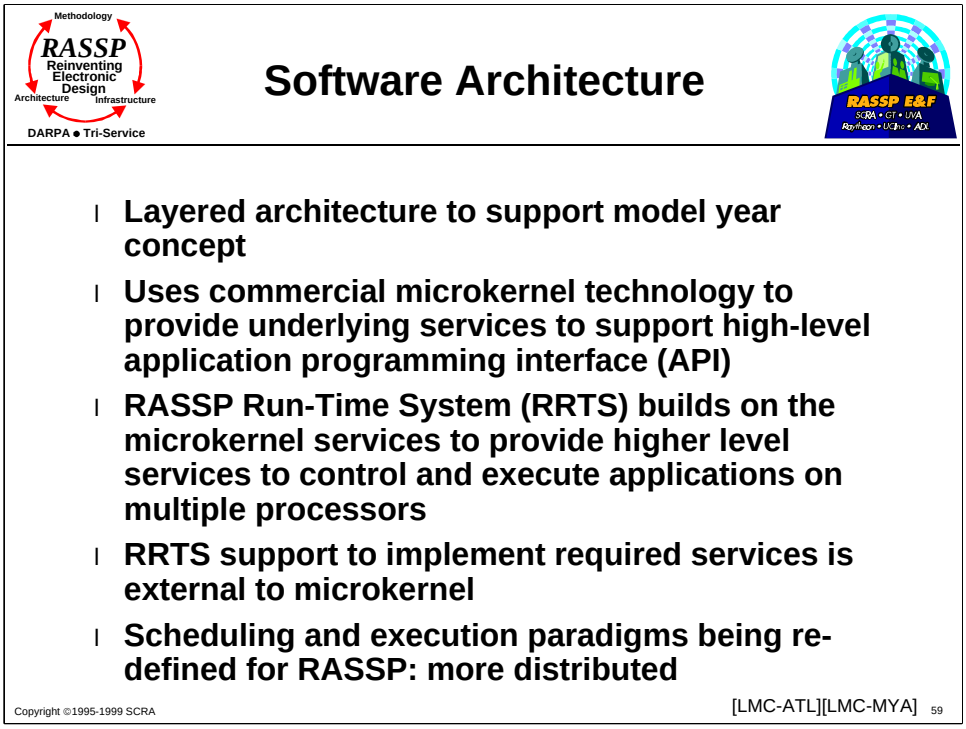
- **Formalize reuse**
- **Emphasize DFG-driven autocode generation**
- **Focus on three major areas of SW functionality**
  - **Algorithm, as specified by a flow graph**
  - **Scheduling, communication, execution, as specified by mapping a graph to a specific architecture**
  - **General command/control software**
- **RASSP is attempting to automate the first two**
- **Use CASE-based code development, documentation, and verification for the general command/control software**

This slide presents a list of the SW architecture process goals desired by the RASSP process. These include a formalized approach to reuse, DFG-driven autocode generation for application code, CASE-based code development for general command and control software when autocode generation is not available.



The requirements of the SW architecture include those listed above.

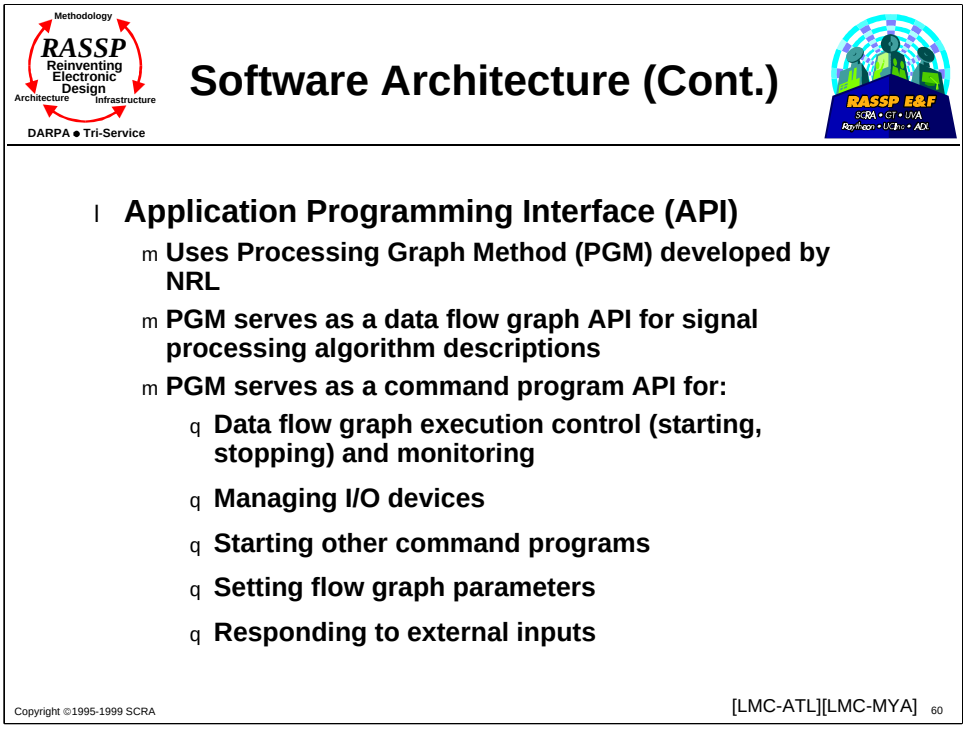
Support should be included that simplifies high-performance real-time DSP application SW development. The SW architecture should provide predictable responses to provided services and easy description, implementation, and control execution of signal processing algorithms. The architecture should support HW upgrades, OS kernel upgrades, and application development in a platform independent fashion.



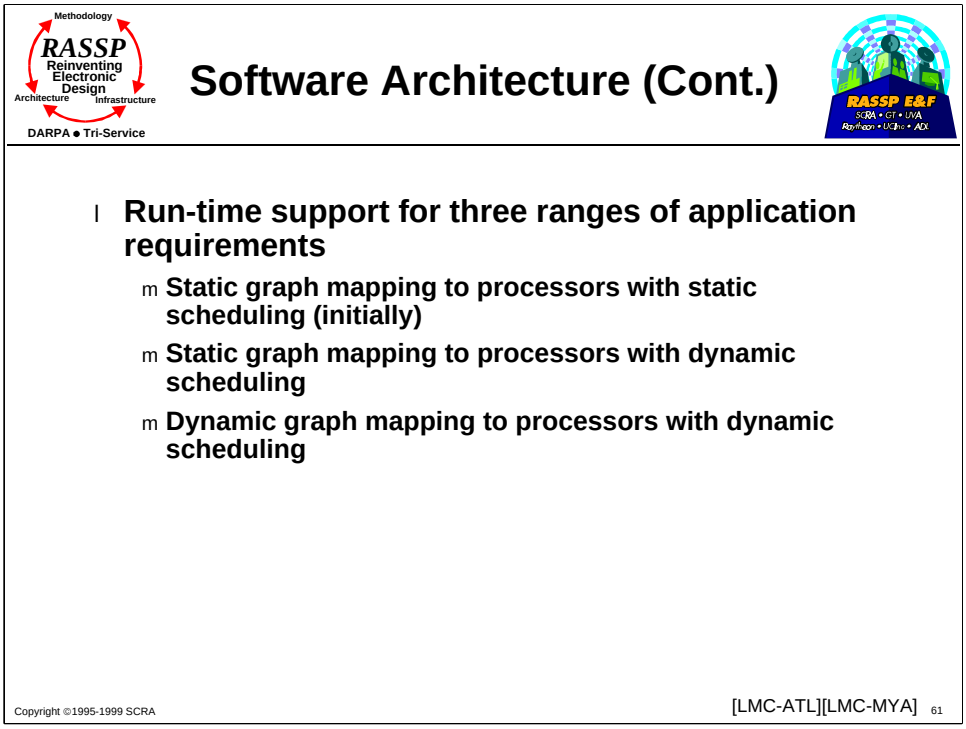
The approach used on RASSP to implement the SW architecture is listed above.

A layered approach is used to support the MYA concept where the replacement of a specific processor and its microkernel would maintain the same API so applications developed for one processor need not be changed when porting it to a new system.

The RASSP run-time system (RRTS) is built on the microkernel to provide higher-level services to control and execute applications on multi-processor systems.



The Application Programming Interface (API) is a set of functions developed in PGM used to develop data flow applications. These functions serve as a buffer between the application program and the microkernel and need not be changed as the kernel is changed during model year upgrades. The API will be highly transportable from platform to platform.

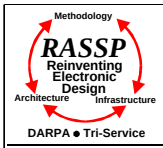


Run time support is provided for static and dynamic graph mapping to processors with static or dynamic scheduling.

Software Architecture (Cont.)

- **Operating system requirements**
  - **Service requirements to support RRTS**
    - **Process creation (spawning)**
    - **Protected address space for processes**
    - **Preemptive multitasking**
    - **Support for dynamic priorities**
    - **Round robin time-slicing for equal priority ready tasks**
    - **Mutex and counting semaphores**
    - **Interprocess communication**
    - **Sequential message passing (sockets)**
  - **Support for COTS products with proprietary O/S**
    - **O/S meets service requirements for RASSP**
    - **O/S provides open interface on which the RRTS and associated API can be ported**

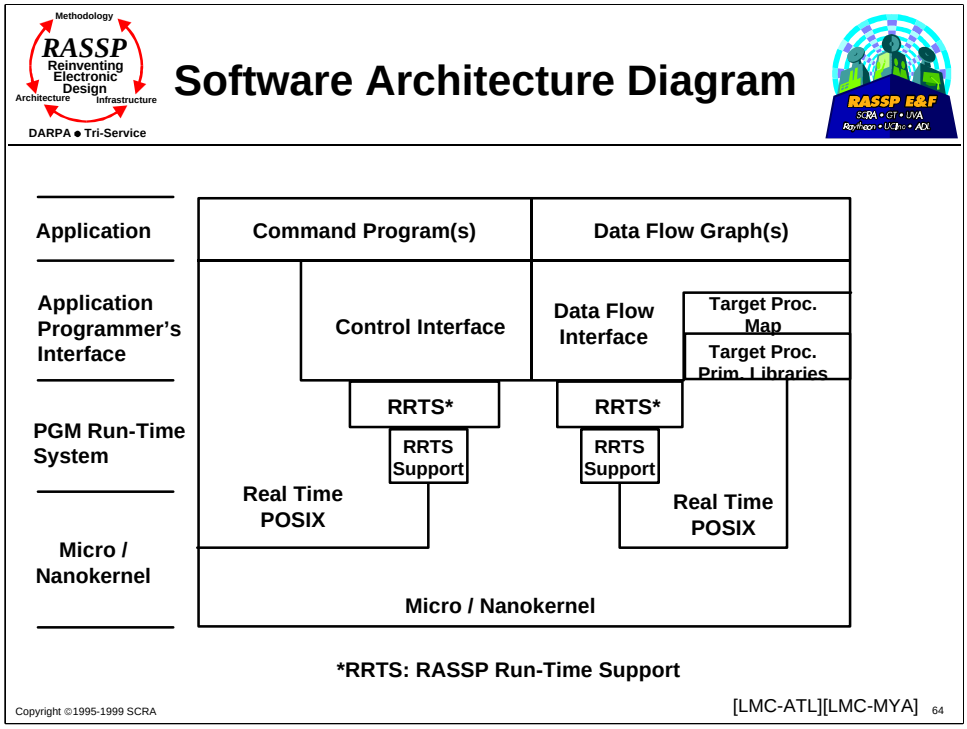
The operating systems requirements for the MYA are presented above. It must support the RASSP run-time system (RRTS) and support COTS products with proprietary operating systems.



Software Architecture (Cont.)

- **Real-time microkernel/operating system candidates**
  - **Large commercial offering**
  - **Not all suited for high performance embedded signal processing**
  - **Current promising candidates**
    - **SPOX - Spectron Microsystems Inc.**
    - **PSOS+/UNISON - Multiprocessor Toolsmiths**
    - **Real Time Executive for Military Systems (RTEMS) - Developed by On-Line Applications Research Corporation under contract to the US Army Missile Command**
    - **Real-Time MACH - Open Software Foundation (OSF)**

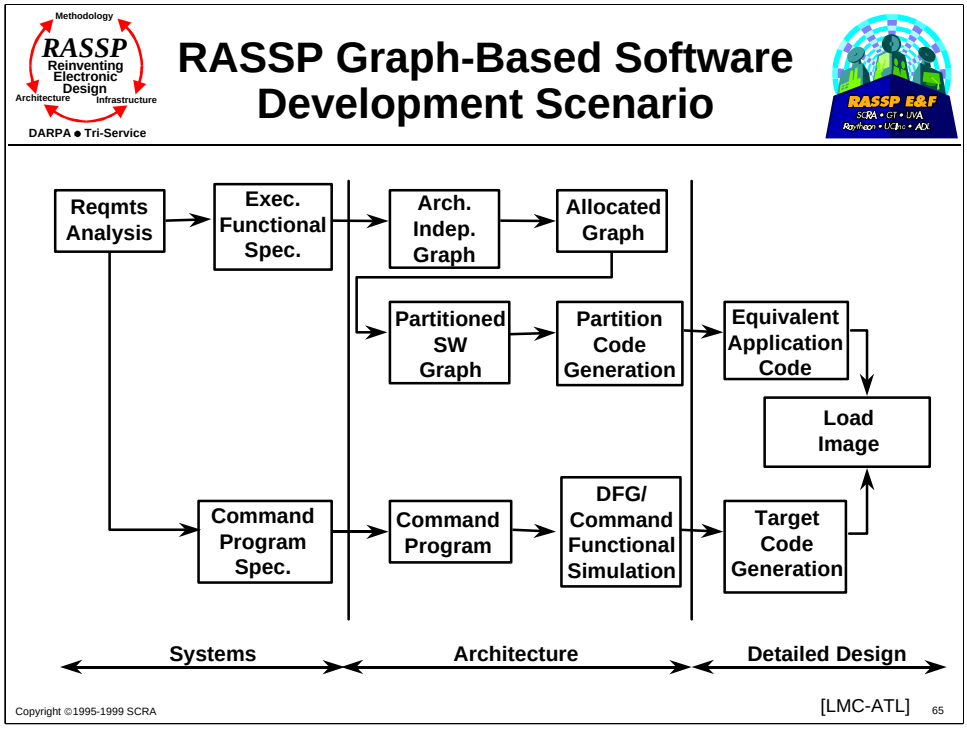
Various real-time microkernels can be used for the operating system. They must be suited for high performance embedded signal processing and a few candidates are listed above.



The software supports the Model Year Architecture (MYA) concept by providing a common Application Programming Interface (API) to the underlying real-time operating system services. This allows a new hardware platform with a new microkernel to change for each model year while maintaining the API. Support for the API is through the RASSP Run-Time System (RRTS), which provides the services required for the control and execution of multiple graphs on a multi-processor system. The RRTS and its support for the API forms the essential component of software encapsulation for a processor object.

The application layer is divided into two parts. The first part is the command program, which provides response to external control inputs , starting and stopping data flow graphs, managing I/O devices, monitoring flow graph execution and performance, starting other command programs, and setting flow graph parameters. The control interface provides services that implement these operations.

The second part of the application layer is the data flow graphs (DFGs) implemented using a data flow language. Services provided by the DFG interface are largely invisible to the developer and include managing graph queues, interprocessor communication, and scheduling. The constructed flow graphs will be converted to HOLs such as C or Ada via autocode generation and will contain calls to a standard set of domain primitives.



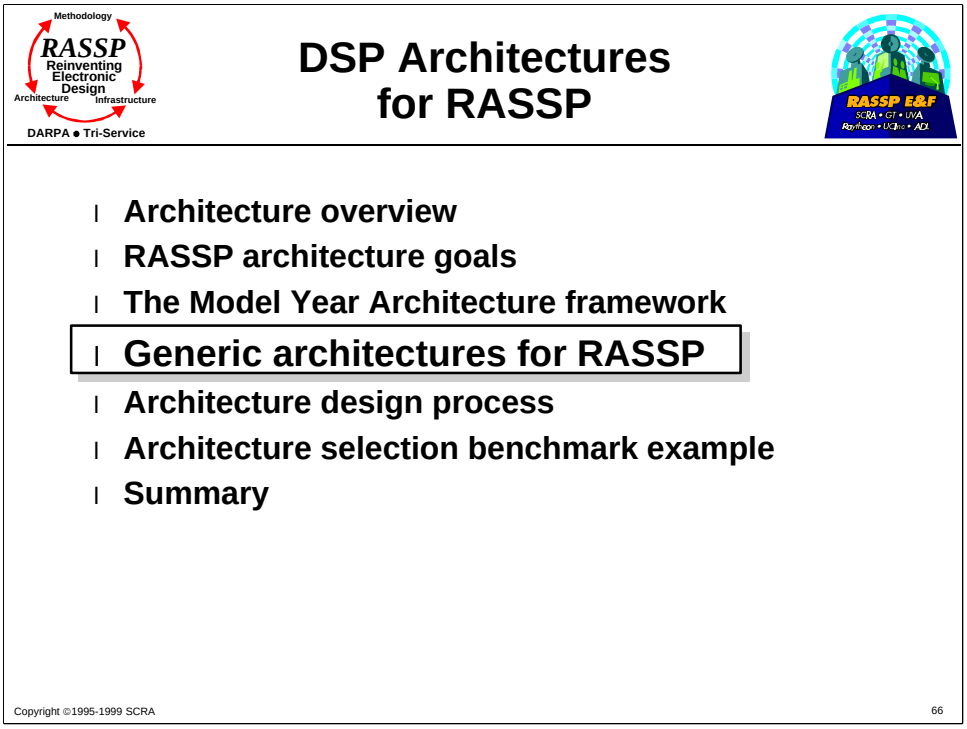
Software development cannot be discussed without its relationship to the architecture. The software portion of architectural objects is handled by the process shown above.

This process depicts the progression of software generation from the requirements to the load image, with emphasis on the graph objects involved and the general RASSP process in which they occur.

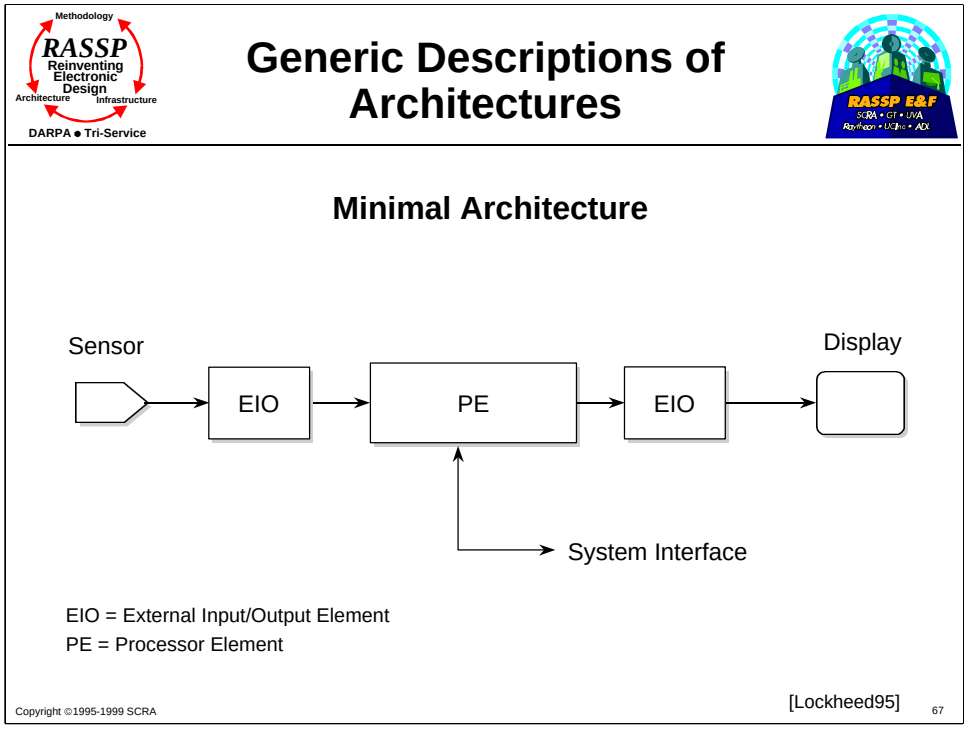
Architecture definition involves the creation and refinement of the DFGs that drive both the architecture design and the SW generation for the signal processor. The DFGs of the signal processor are developed, and the nodes are allocated to either hardware or software. Automated generation of the software partitions is performed to provide executable threads that are to be run on the DSPs. These autocoded partitions are combined into an application graph which is functionally equivalent to the original.

The final step in the SW development, which is the production of the load image, occurs during detailed design. The load image generation is an automatic build process that is driven by the autocode generation results. The inputs to the process include the architectural description, the detailed DFGs describing the processing, the partitioning and the mapping information, the autocode results, and the command program.

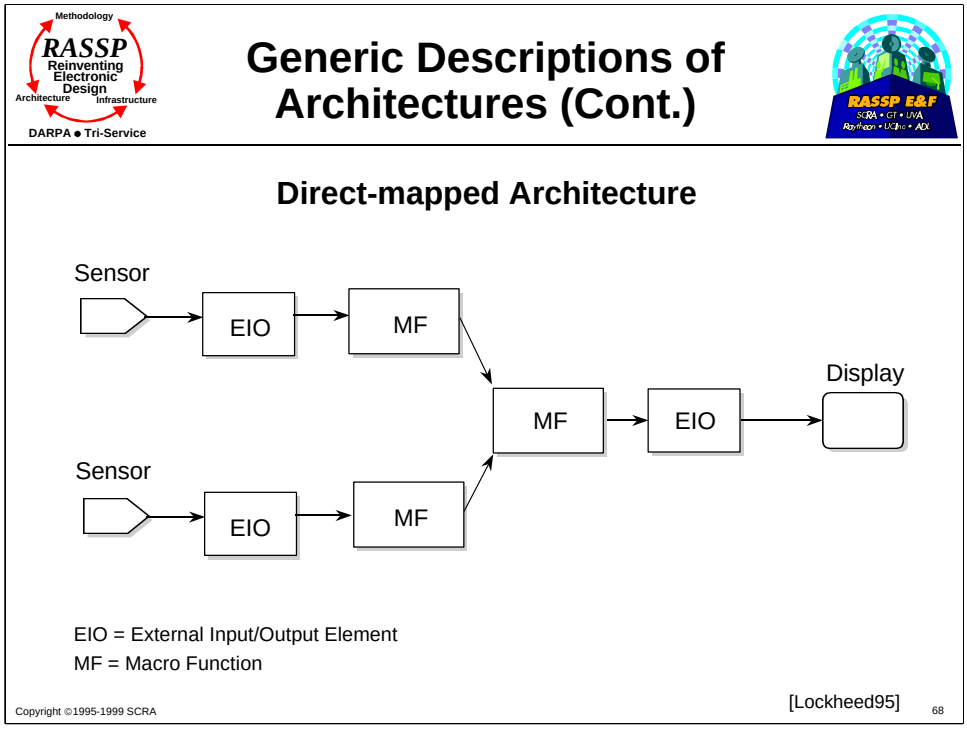
The process is controlled by a software build management function which extracts the necessary information from the library and manages the construction of all the downloadable code.



This section covers a set of generic architectures from which the RASSP user can select a preferred architecture as a starting point for a signal processor design.



This is the minimal architectural configuration consisting of a single HW processor, some I/O, and an interconnect. The interface HW accepts sensor data as its input and displays processed data at its output.

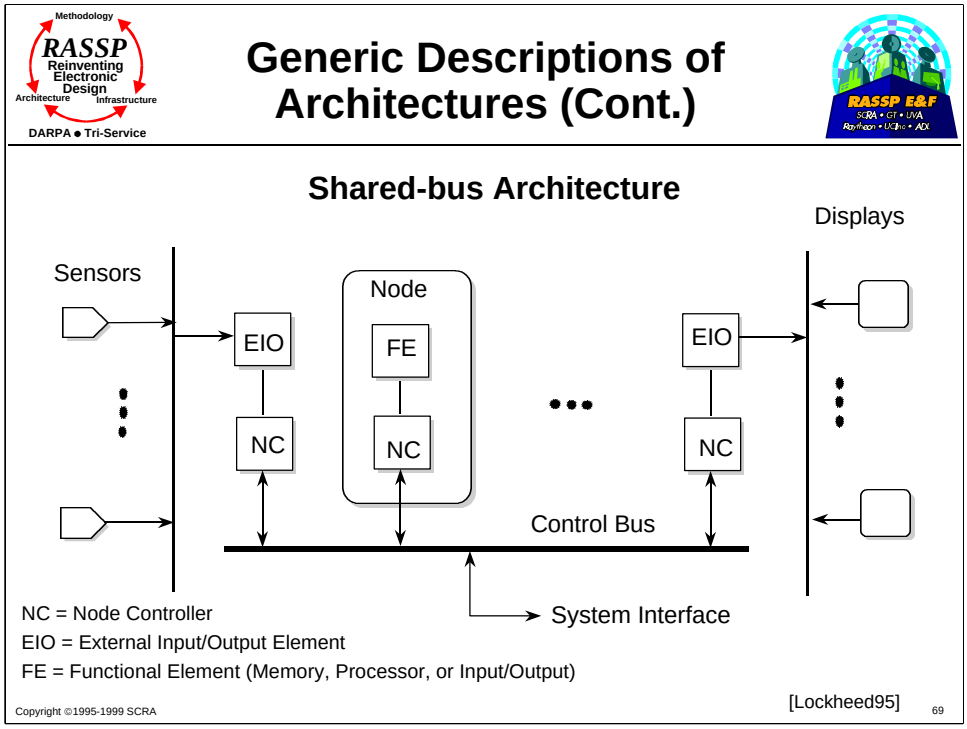


A direct-mapped architecture represents a one-to-one correspondence between the algorithm and the HW module solution. HW modules are interconnected with dedicated point-to-point links and data flows from the raw input through the various modules to the solution output.

Macro function modules include such things as filters, FFTs, etc.

The direct-mapped architecture is inflexible because a change in the problem algorithm requires changing the HW modules and interconnecting links. It is also not readily scaleable and fault intolerant.

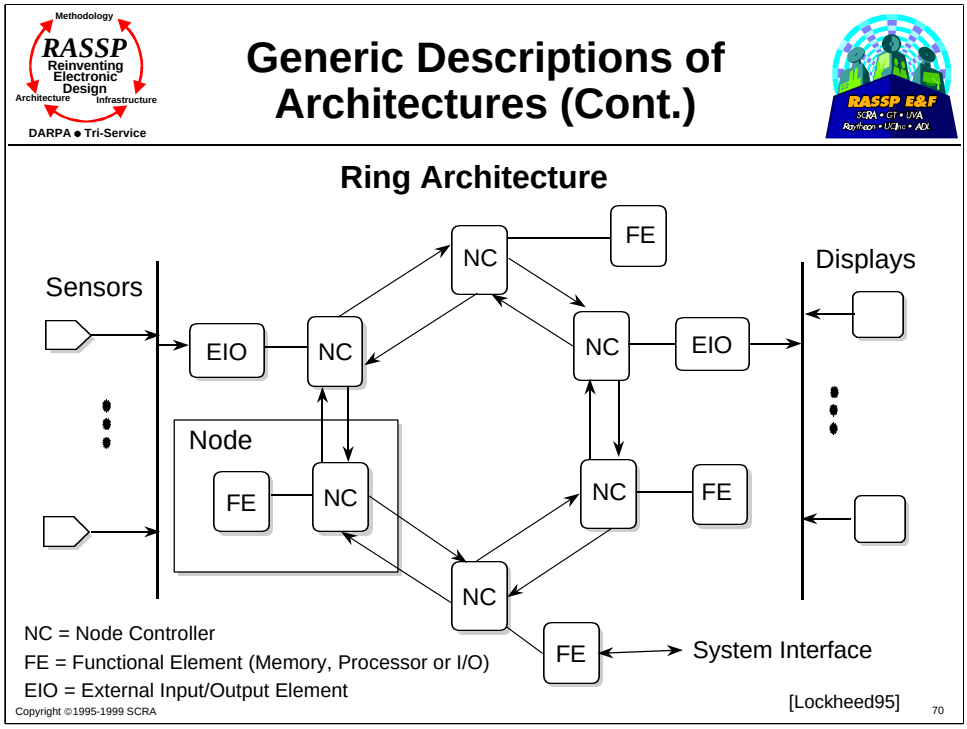
The trade-off is lower cost, size, weight, and power versus a lack of easy future upgradability.



The shared-bus architecture employs a conventional control bus (VME, Futurebus+, etc.) as the mechanism for interconnecting multiple nodes.

The architecture uses standard HW modules with much of the application solution implemented in SW.

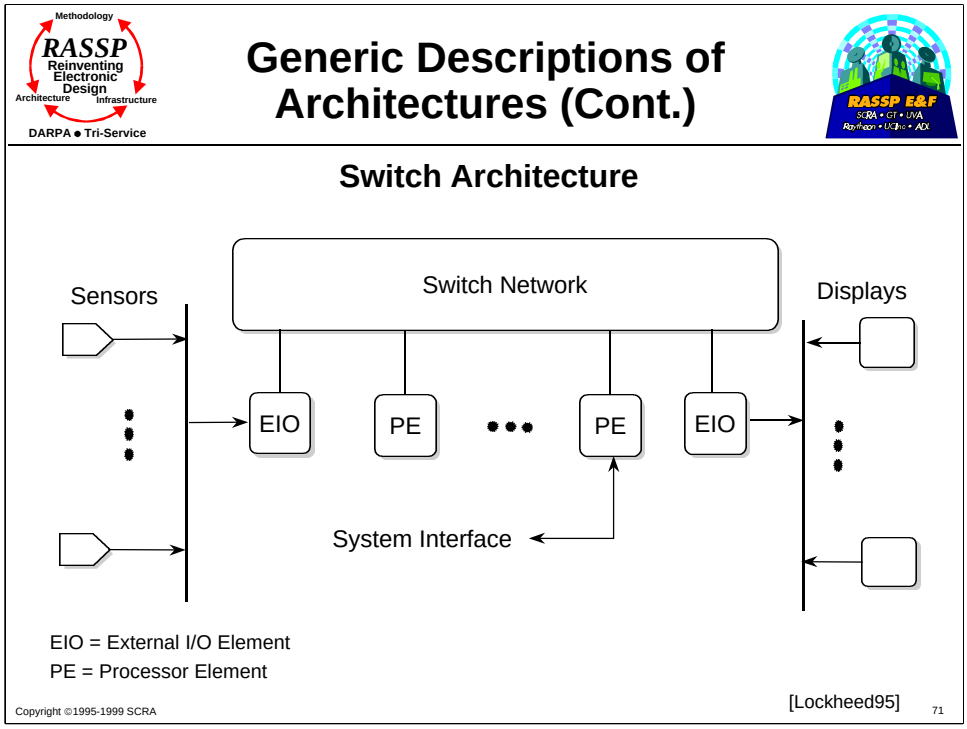
It is scaleable because additional HW and SW modules can be added easily, but is limited in the degree of scalability by the fixed interconnect bandwidth of the shared bus.



The ring architecture employs a set of point-to-point links configured in a ring topology (FDDI, SCI, etc.) as the mechanism for interconnecting multiple nodes.

The main difference with the shared bus architecture is that the communication paths between nodes are separate links rather that common bus.

This diagram shows the dual counter-rotating rings implementation typically used for fault-tolerant applications.



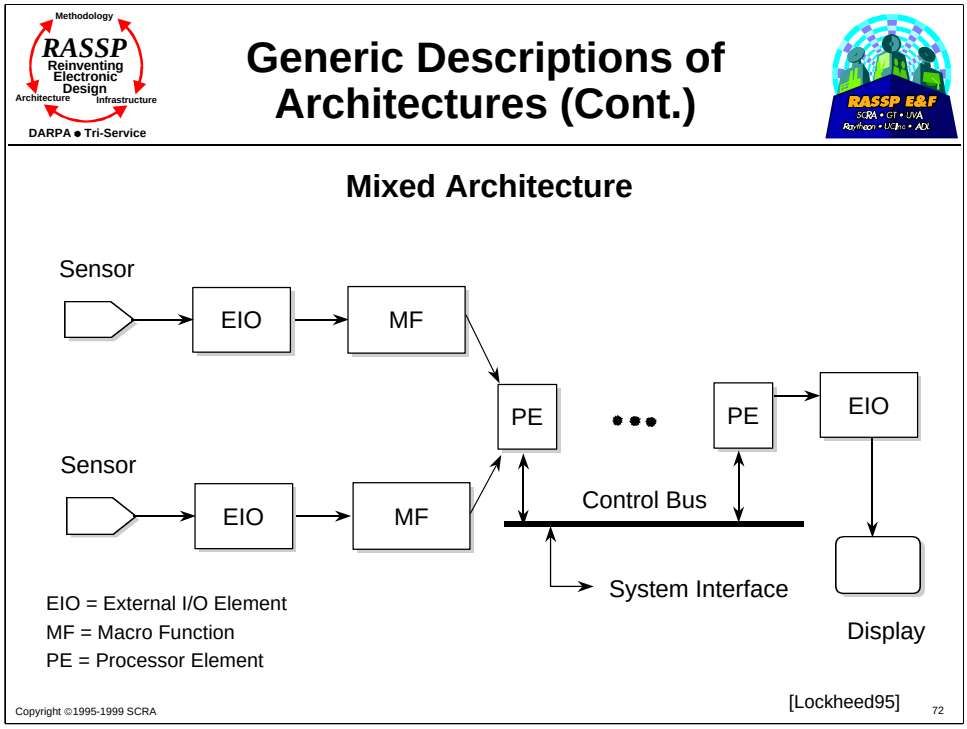
The switch architecture interconnects nodes with direct links that can be reconfigured (switched) dynamically. The simplest is the crossbar switch, where every node can be connected to any other node.

The Mercury RACEway uses a series of crossbar switches connected in a multistage pipeline to achieve a modularly scaleable switch network.

The advantage of a switch architecture is that it achieves very high bandwidth.

One disadvantage for a crossbar switch is that it does not scale well. As the number of nodes in the network increases, the complexity and cost of the switch increases more quickly than linear.

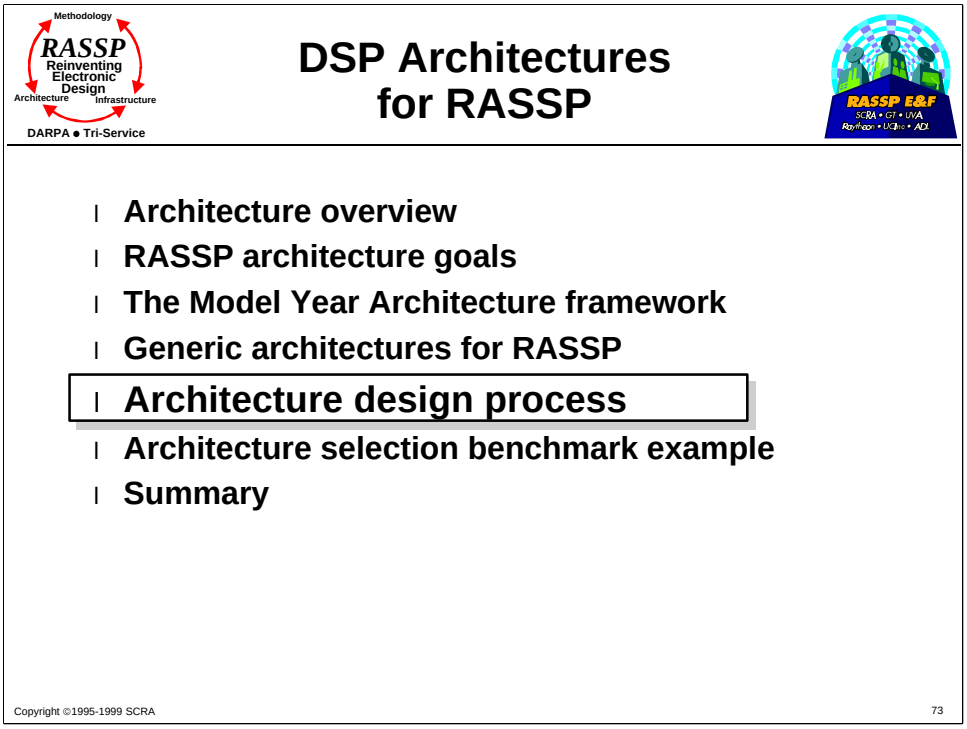
Topologies based on crossbar switches can scale well.



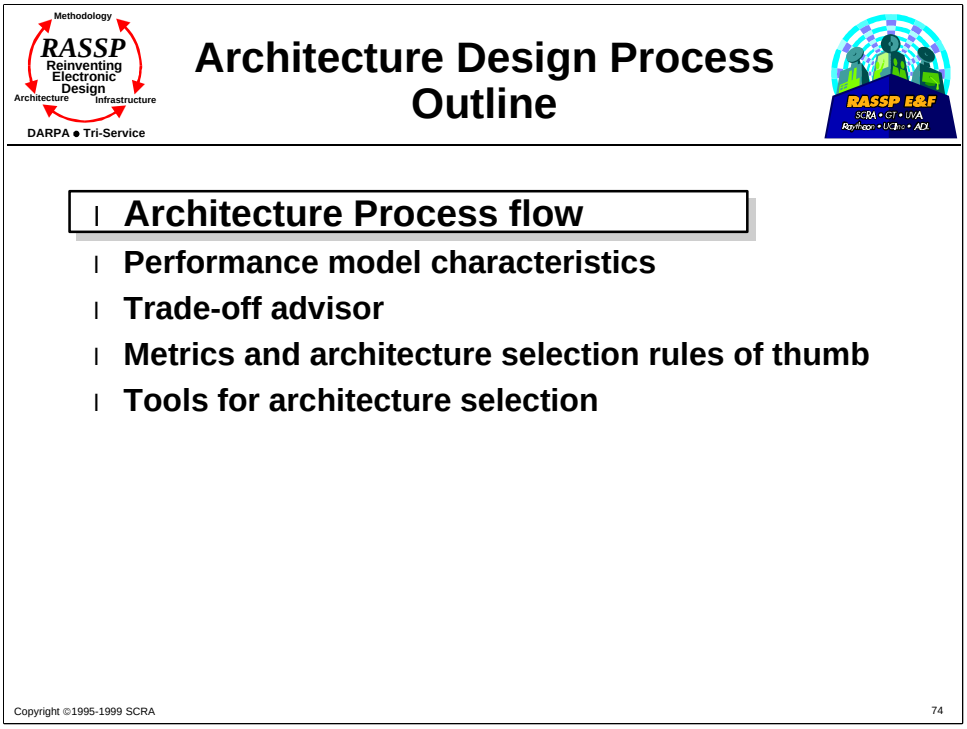
A mixed architecture is a combination of two or more of the previous architecture types.

An example might include a direct-mapped architecture to handle the front-end pre-processing function and a shared bus to handle the less demanding back-end processing.

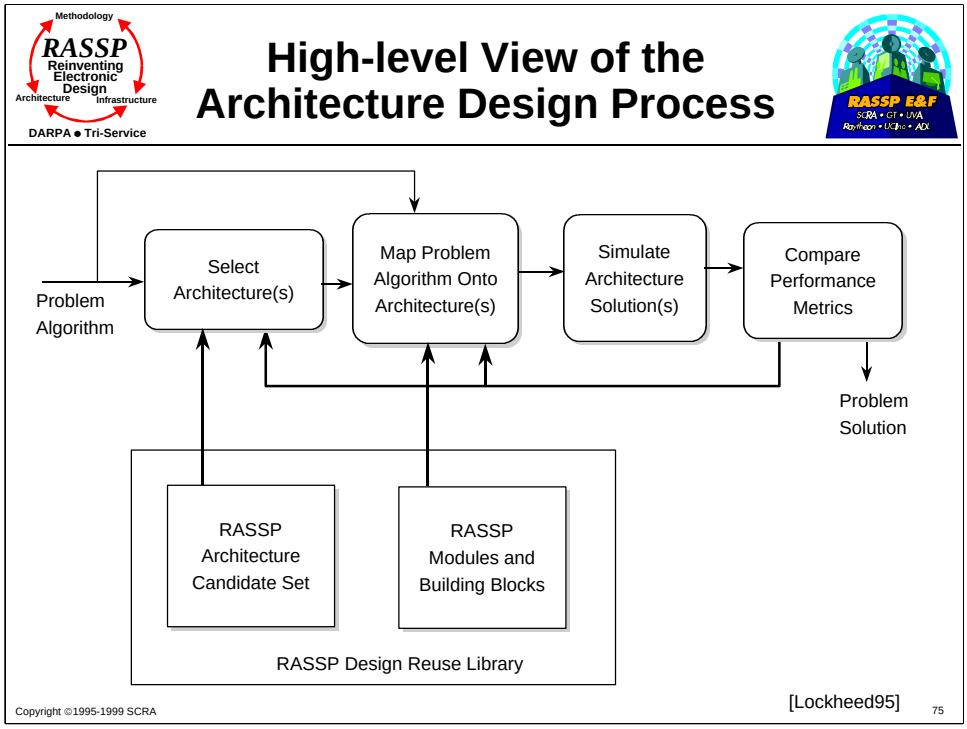
Another example is the hierarchical multi-drop bus approach used in the Pave Pillar architecture.



This section presents an architecture selection methodology, including a set of guidelines in the form of rules of thumb for capturing the key features in a preliminary architecture selection.



The first section describes the architecture process flow in the design process.

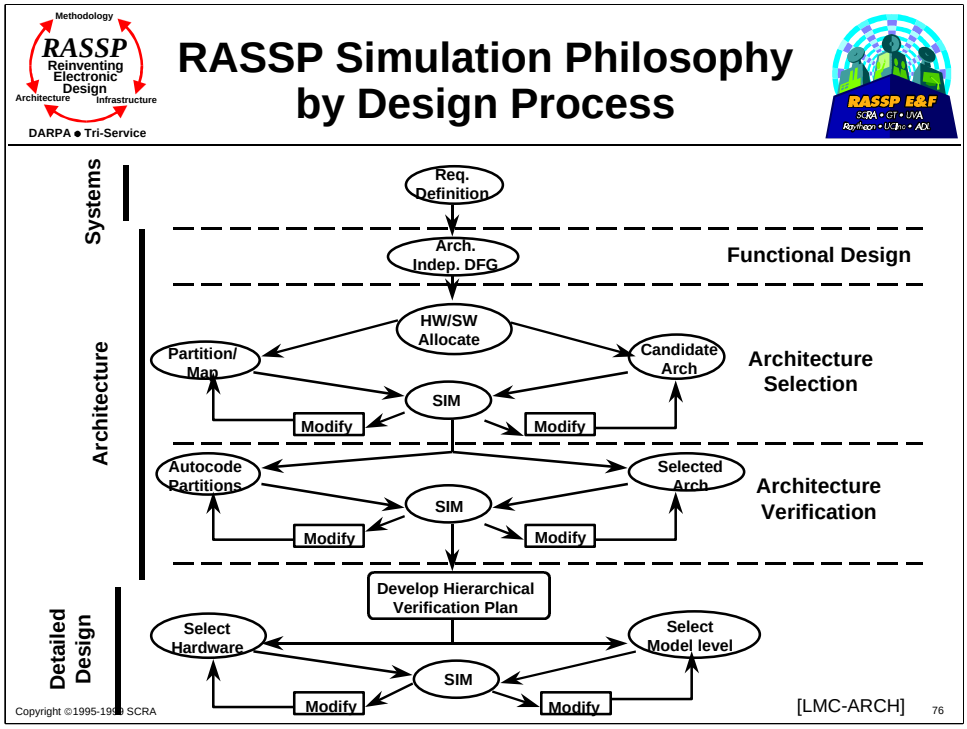


Prior to selecting an architecture, the customer's requirements are examined and expressed in terms of a list of standard RASSP metrics (presented later). The metrics are weighted relative to one another based on their respective importance.

RASSP architecture selection begins with the selection of a small number of candidate architectures from a provided set of template architectures contained in a reuse library. Coarse rules of thumb are used based on the algorithm to be solved.

After choosing candidate architectures, the problem is mapped onto the architectures selected. This is done using performance models from a reuse library.

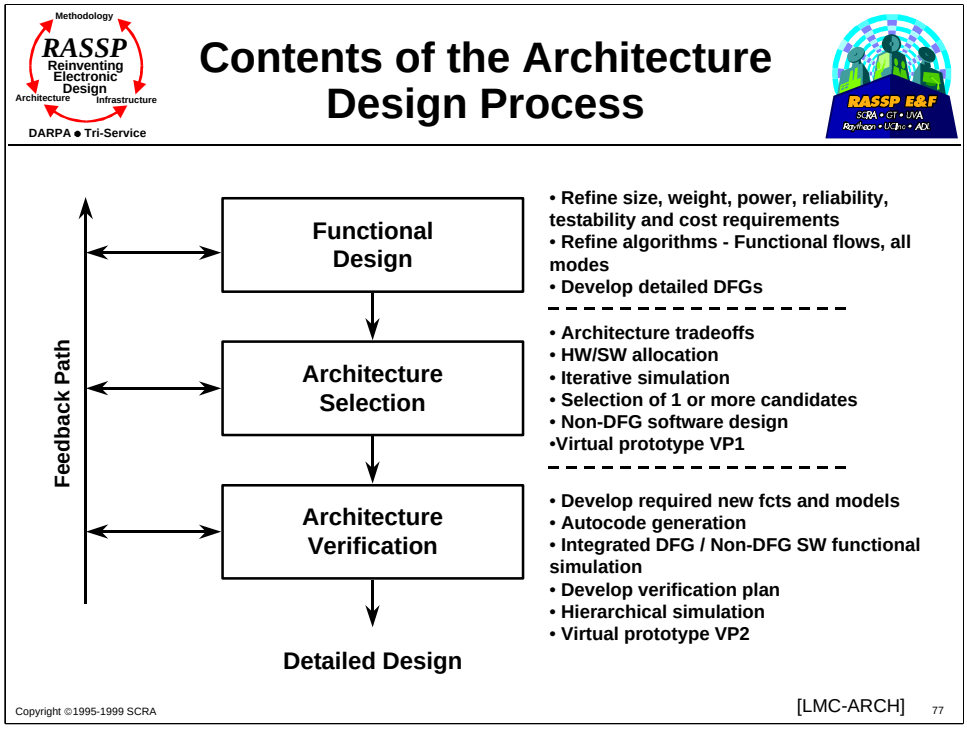
The architectures are simulated at the performance level to compare results via metrics previously defined for the problem.



The systems, architecture, and detailed design processes are the first three main processes in the design of a system. The focus of this presentation is on the architecture design processes. Within the architecture design process, there are three additional processes that need to be defined; functional design, architecture selection, and architecture verification. Simulation is performed at each of the stages of the design.

At the requirements stage, simulation is done using an executable specification developed for the application.

More detail on the functional design, architecture selection, and architecture verification stages will be contained in the following slides.



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

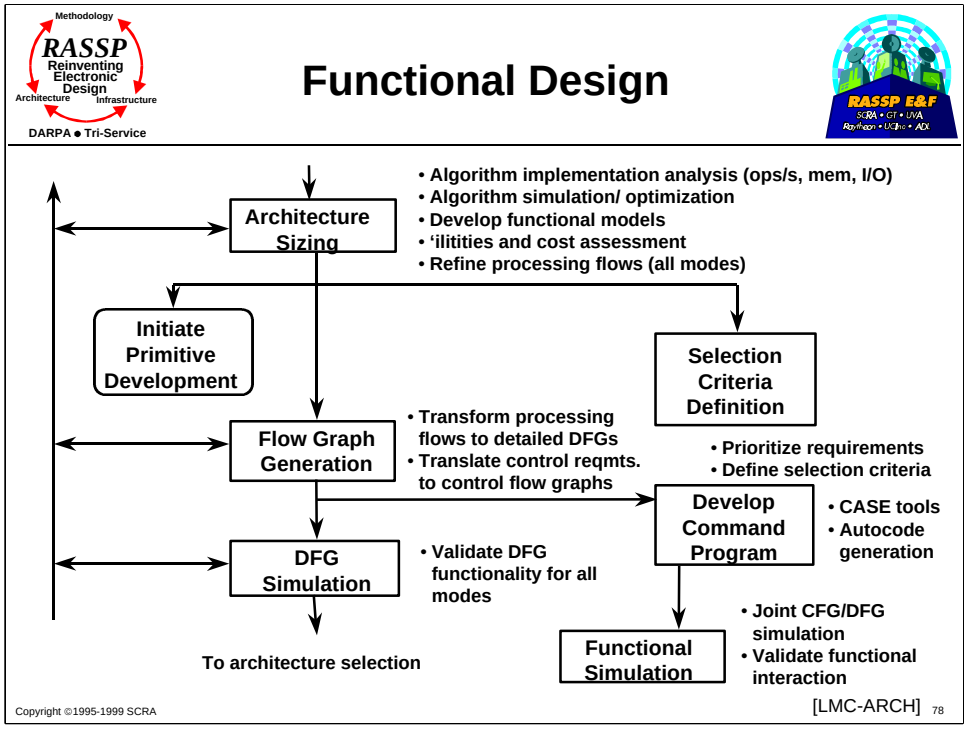
The architecture definition process is a new HW/SW codesign process in the RASSP methodology for high-level virtual prototyping and simulation. The primary concern in the architectural definition process is to select and verify an architecture for the signal processor that satisfies the requirements 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 DFG-driven. Reuse of both architecture elements and software primitives significantly shortens the design cycle. VHDL performance model simulations are used to verify system requirements are met. 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 based upon the Processing Graph Method (PGM) developed by the Navy. PGM is a specification for defining detailed DFGs for signal processing applications. 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.

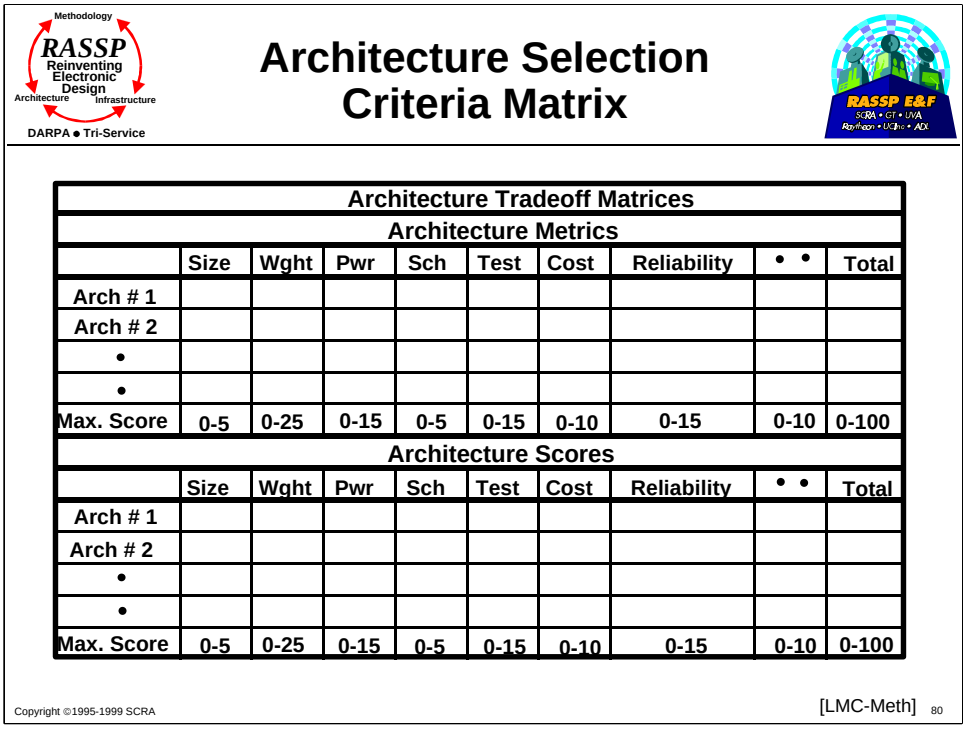
Functional simulation verifies both the DFGs and the CFGs and their interrelationships.

Architecture Sizing

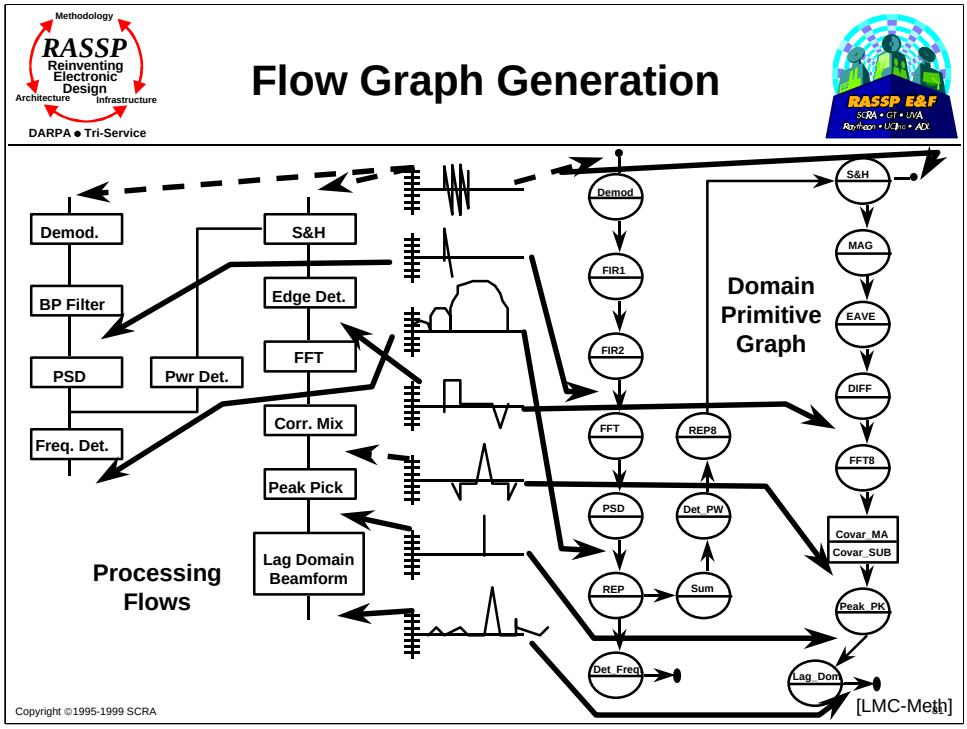
- **Analyze system requirements and processing flows in terms of operations/sec, memory requirements, and I/O bandwidths**
- **Make initial size, weight, power, and cost assessments**
- **Develop first pass partitioning of HW and SW functionality**
- **Develop any necessary functional models**
- **Perform algorithm simulations and optimization**

This slide lists the functions of the architecture sizing step in the design process.

This step analyzes the system requirements and processing flows in terms of operations/sec, memory size, and I/O. Initial estimates are made for size, weight, power, and cost of the system. A first pass at partitioning of HW and SW functionality is done. Simulations are performed on the algorithms developed and optimizations are incorporated to meet the requirements. Models are created as needed by this process.



As part of the selection criteria, a trade-off matrix is defined to help formalize the selection process. This chart contains some a list of the top-level requirements allocated to the signal processor. Satisfying these requirements drives the HW/SW codesign of an architecture. The matrix is populated as the design progresses. Early in the process the entries are less accurate than later on. The goal is to eliminate some of the designs early while carrying the best candidates to subsequent levels of detail.

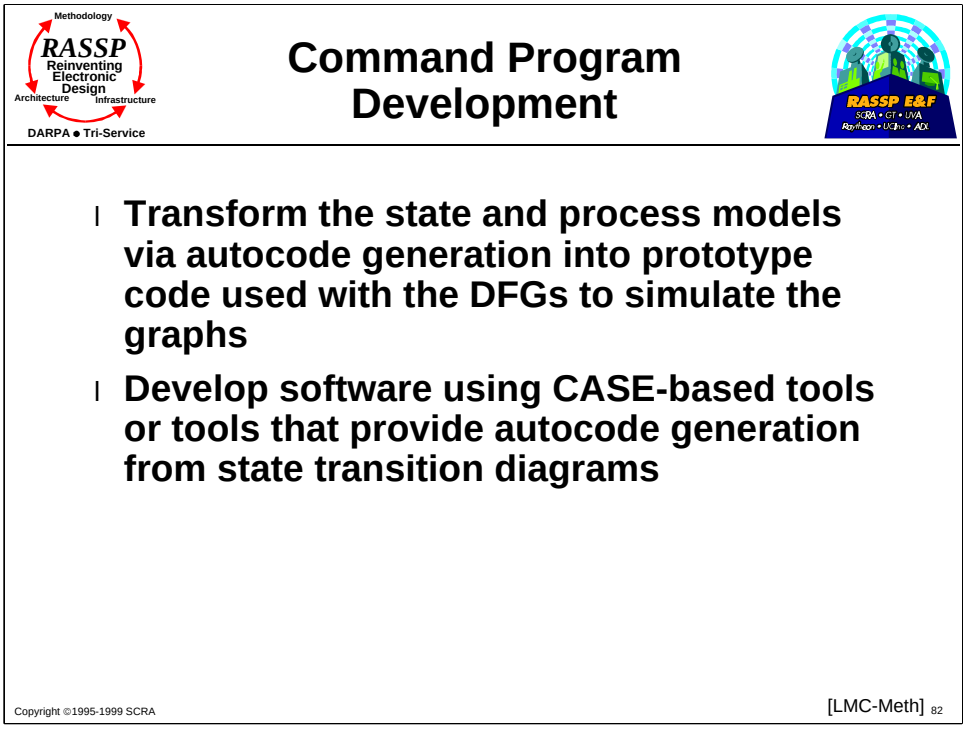


The transformation of the finalized algorithm processing flows into the detailed DFGs is the first step in the HW/SW codesign process. These DFGs are based upon the Processing Graph Method (PGM) developed by the Navy. PGM is a specification for defining detailed DFGs for signal processing applications. The DFGs are made up of reusable library elements, which may represent either HW or SW.

The DFGs are the basis for architecture synthesis, detailed software generation, and potentially custom processor synthesis.

The left side of this figure represents the processing flows as passed down from the systems definition stage. The right side represents the detailed DFG constructed from reuse library elements.

If suitable library components do not exist, then they need to be developed and added to the library.

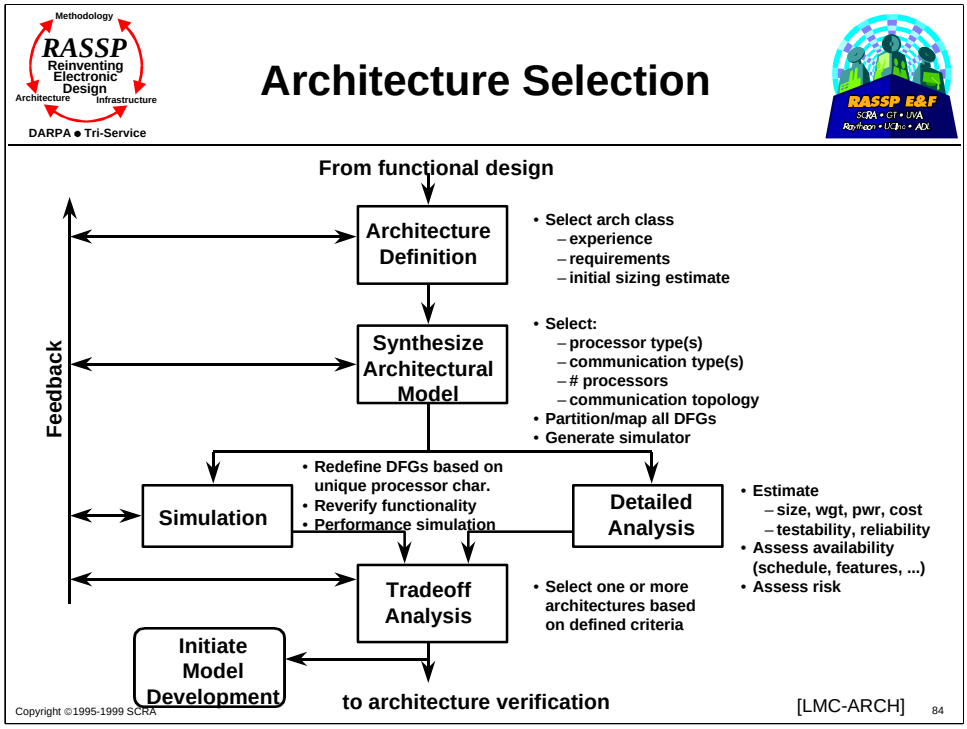


The command flow is modeled using command programs. The command programs are generated by transforming the state and process models using autocode generation into prototype code used with the data flow graphs to simulate the graphs. CASE-based tools or tools that provide autocode generation from state transition diagrams are used to develop the software.

Functional Simulation

- **Simulate both DFGs and CFGs**
- **Verify DFG simulation matches processing flow description**
- **Validate CFGs interaction with DFGs which includes:**
  - **Passing parameter information between external world and graph management software**
  - **Initiating or terminating I/O devices**
  - **Starting and stopping DFGs**

When all the code is created for both the DFGs and CFGs, simulations are performed to verify its behavior. This is compared with the processing flows described by the executable specification (level 0 VP). The CFGs interaction with the DFGs are validated.



Architecture selection is an automation-aided process to rapidly evaluate different architectural designs and instantiations of these designs.

The architecture selection process represents the heart of the RASSP HW/SW codesign, which uses a library-based, DFG-driven approach to SW development combined with iterative performance trade-off analysis to support rapid selection/analysis of candidate architectures. For each architecture offered as a candidate in the selection process, the following steps in the process are done:

Develop a partitioning and mapping for the candidate architecture

- Performance analysis of the partitioning and mapping
- Optimization of the mapping, resulting in processor instantiation
- Analysis of the instantiation size, weight, power, cost, testability, reliability, risk, etc...
- Iteration of the above until one or more acceptable architectures are attained

Inputs to the architecture selection process are the prioritized processing requirements, the selection criteria, the required DFGs for all modes of operation, command program specification, and other non-DFG requirements, and the HW/SW reuse library.

Outputs from the process are the finalized DFGs and one or more architecture instantiations that were selected for more detailed functional and performance verification. Also output is the description of the DFG partitioning and mapping to the processors of the selected architecture(s) for all processing modes.

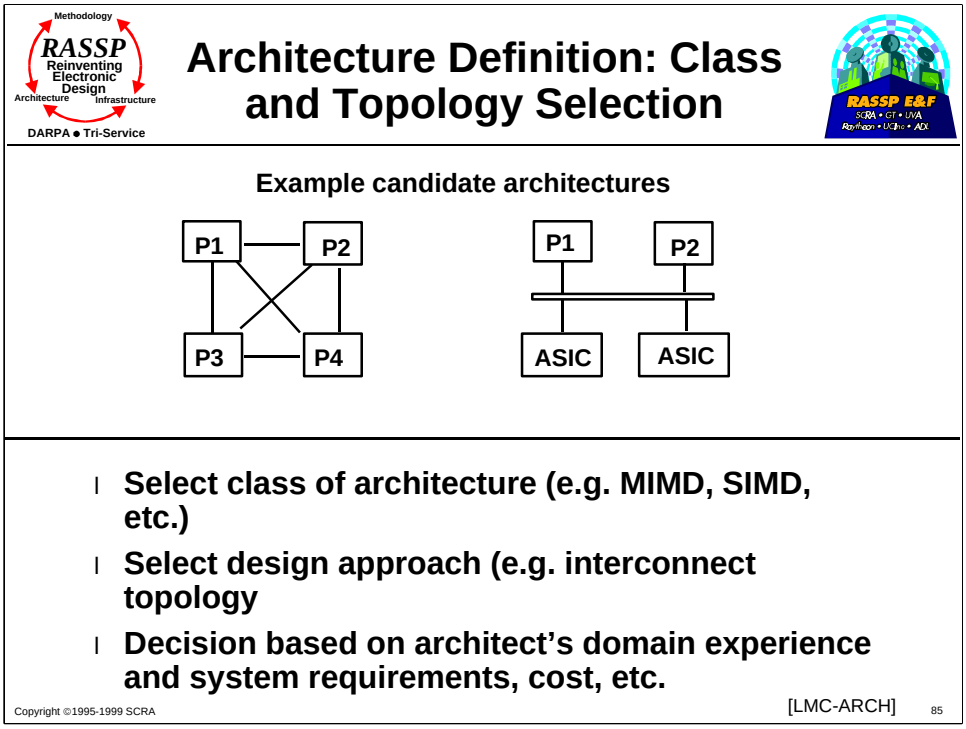
Architecture definition involves selecting the class of architecture to be used (e.g. MIMD, SIMD, etc.) and the design approach within the class (e.g. interconnect topology).

Architecture model synthesis selects the specific processor type(s), number of processors to be used, along with the communication mechanism (e.g. bus, Xbar switch, etc.) for the selected architecture types. The DFGs allocated to software are partitioned and mapped to the available processors of the candidate architectures under consideration. The SW partitions are defined by mapping the primitives in the DFG to the DSPs in the architecture.

Simulation is used to verify the algorithms functionally and to refine the performance of the candidate architectures using available throughput, memory, and I/O estimates for these algorithms. VHDL is used to perform this simulation.

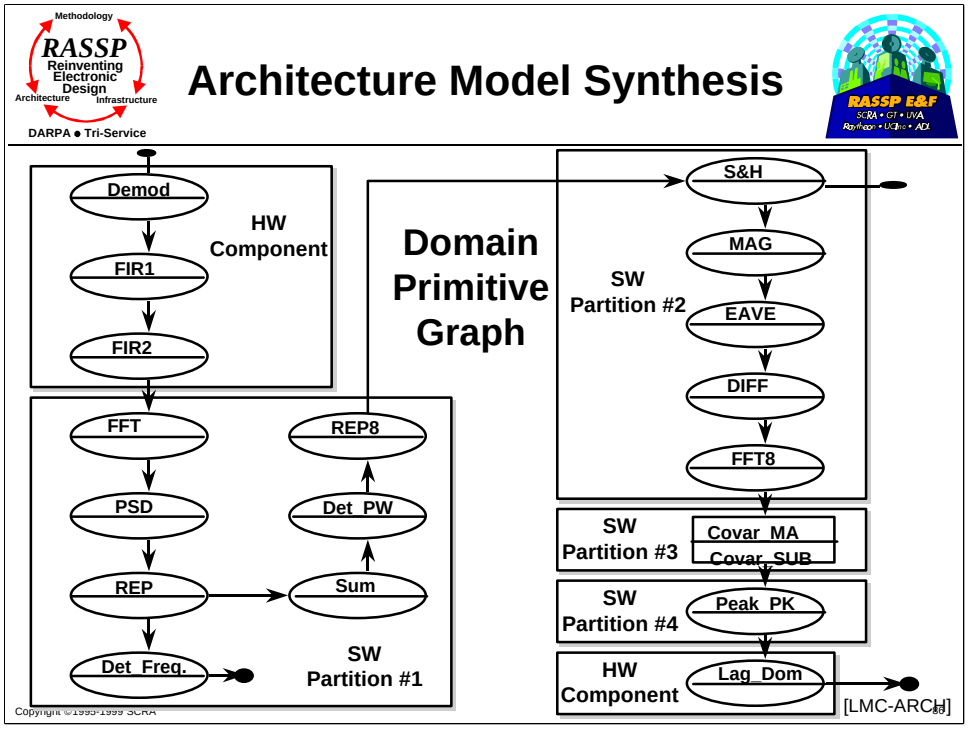
Detailed analysis involves proceeding with implementational analysis of the candidate architecture(s).

Trade-off analysis helps determine an optimized solution by iterating the architecture synthesis, simulation, and detailed analysis process for each of the candidate architectures. These activities are directed toward populating an architecture trade-off matrix that is a record of the design process.



Given the DFGs that describe the processing, the architect must postulate one or more designs that may satisfy the requirements. These choices are based upon the domain experience of the design team. One of the RASSP goals is to facilitate the ability to define and evaluate more alternatives than would otherwise be possible through semi-automated tools that assist the architect.

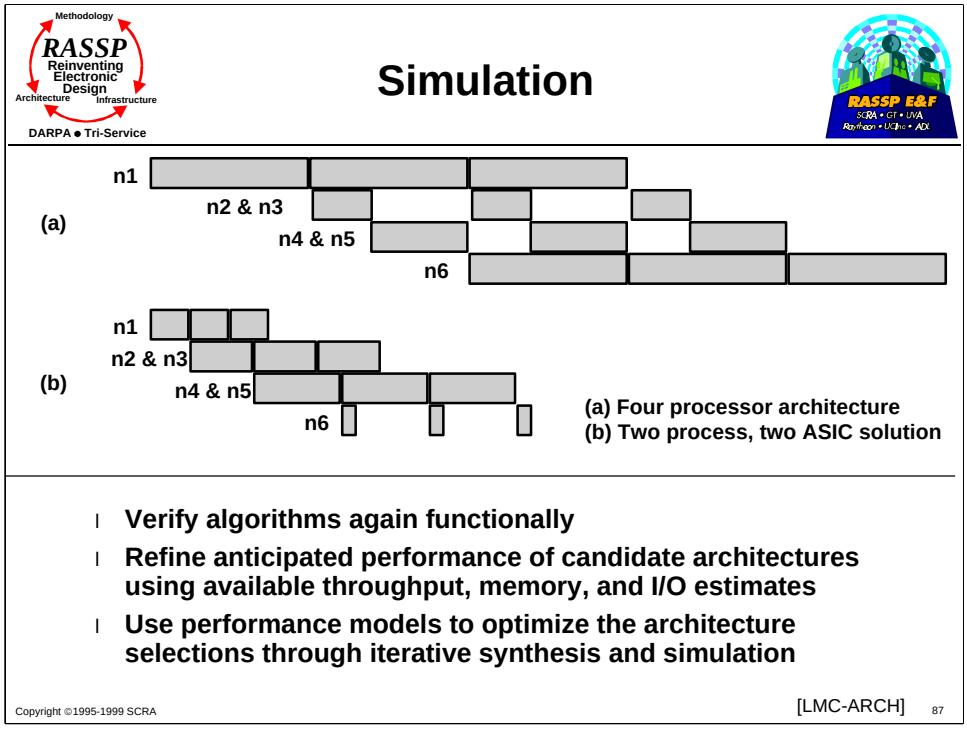
The processing at this point represented by the DFGs has not been allocated to either HW or SW.



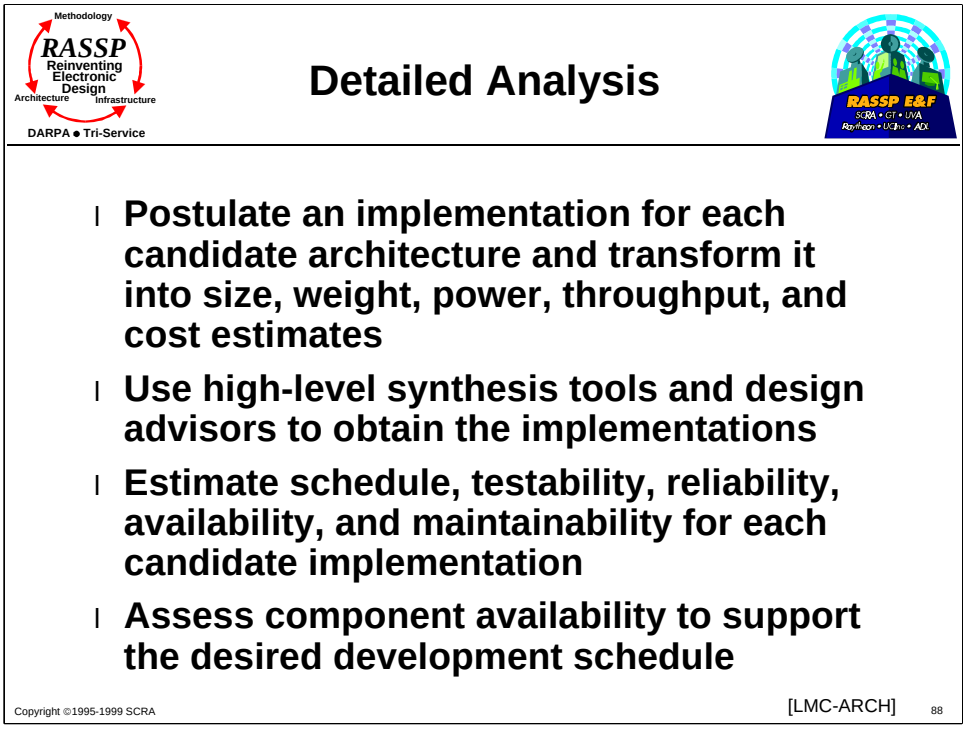
For the selected architecture type, the next step is to select the specific processor types and number of processors, along with a desired communication mechanism. The process of defining the architecture is coupled with the allocation of the DFG to the architectural elements. To support this capability, the library contains a hardware model capable of performing the processing defined by the DFG node.

The portion of the DFGs allocated to SW is partitioned and mapped to the available processors of the candidate architecture under consideration. The SW partitions are defined by mapping the primitives in the DFG to the DSPs in the architecture. The above figure show a DFG in which two portions of the DFG are allocated to hardware and the remainder of the DFG to SW grouped into four partitions. This activity is supported by multiple, automated partitioning/mapping algorithms for graph assignment and a manual capability.

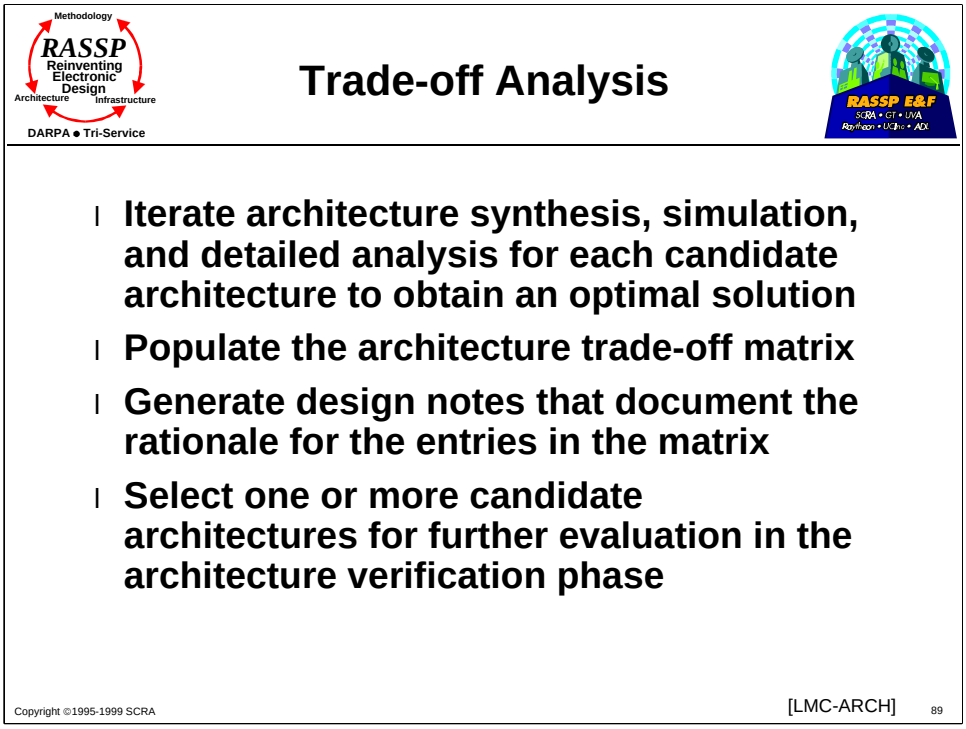
A VHDL performance model is constructed for the architecture to obtain performance metrics for the partition.



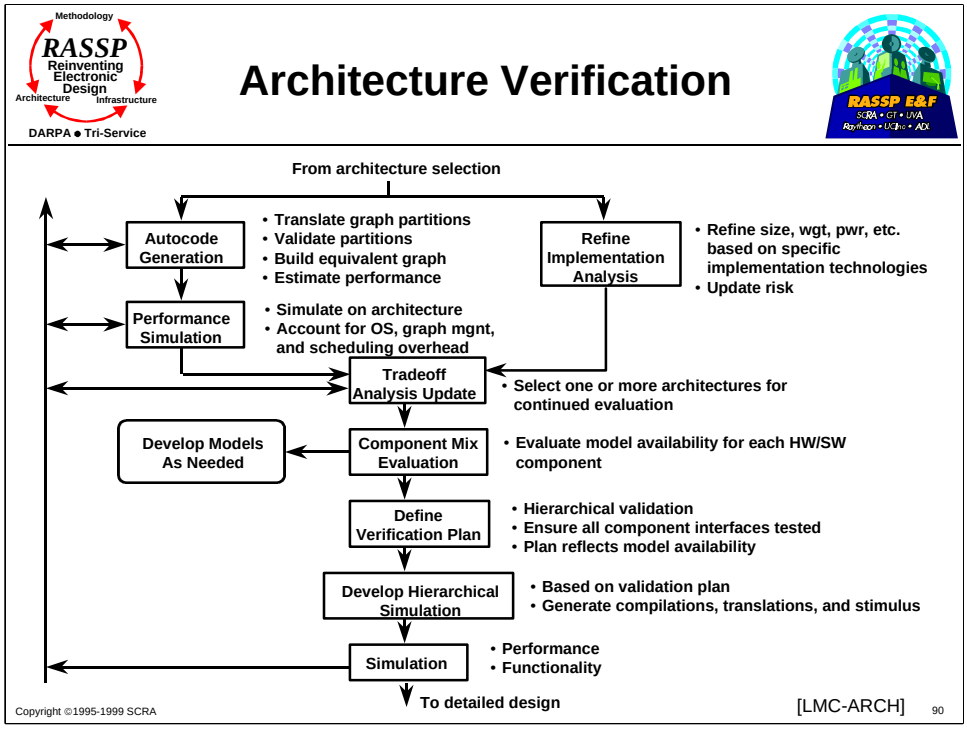
Simulations are performed on candidate architectures with timelines created as shown above. This helps refine the architectural candidates and improve the estimates in the selection criteria matrix. The functionality of the algorithms is verified against previous simulations at the higher level to assure correctness.



Detailed analysis of the results help postulate an implementation for each of the candidate architectures using high-level synthesis tools and design advisors. Improved system requirement estimates are used to help refine the selection criteria matrix and estimates of schedule impact, testability, reliability, parts availability, and maintainability can now be obtained.



This step iterates on architecture synthesis and simulation to obtain an optimal solution to the architecture selection problem. The trade-off matrix is populated and design notes are generated that document the rationale for each of the entries in the matrix. The number of candidate architectures is now trimmed to only a few for further evaluation in the verification phase.



Architecture verification is the process of hierarchically simulating both the functionality and performance of a selected architecture candidate. Simulations are performed at a greater degree of detail as compared to those in architecture selection process. The goal of the verification process is to validate operation of all architectural entities and the interfaces between them before detailed design. Software partitions are autocoded to produce software modules translated from the processor-independent library elements to optimized processor specific implementations, which are interfaced through a set of standard services build on an operating system microkernel.

Inputs to the architecture verification process include the selected architecture instantiation, which includes all or a portion of the implementation partitioning/component list, the optimized DFGs, the CFGs, detailed SW description, and the HW/SW reuse library.

The outputs from the this process include new library elements, detailed specifications for HW development, and performance and functionality verification.

Autocode generation uses the finalized DFGs and the partitioning/mapping data as inputs to generate the software for each of the partitions. The code is generated by translating the processor-independent flow graph primitives to target-specific code which uses the optimized math libraries for the specific DSP.

Performance simulation uses timing estimates from the autocode generation to estimate the performance of the design partitions. It should account for performance impacts due to the target operating system, the graph management system built on top of the operating system, and any scheduling overhead.

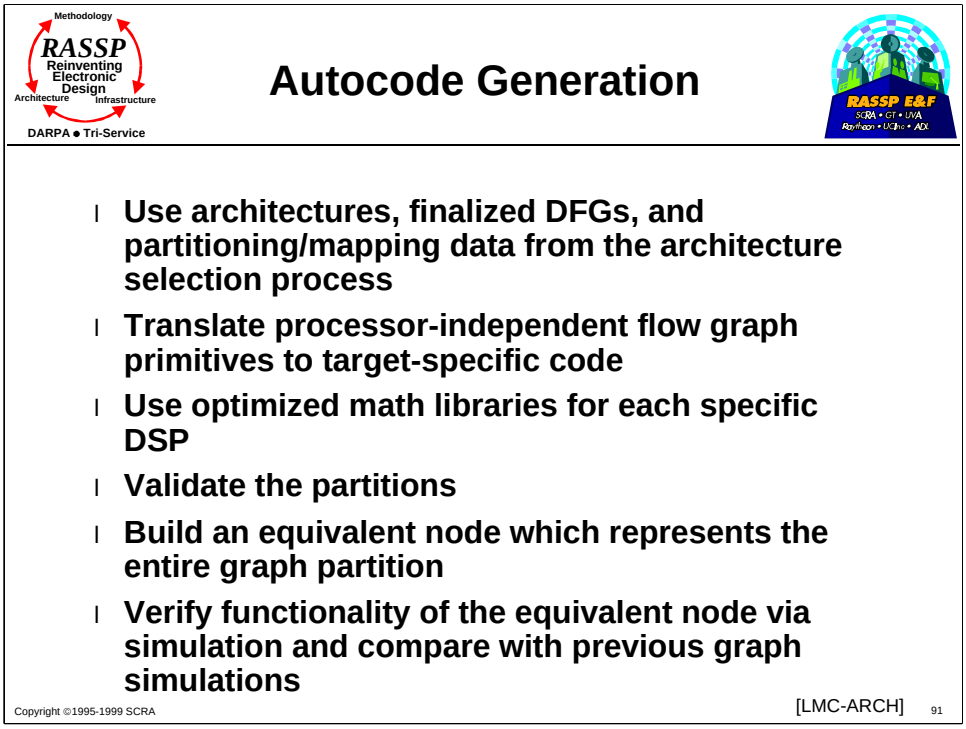
Refining the architectural attributes can now be done because more information is made available by the increased detail of the design.

Trade-off analysis is updated by filling in more detail in the trade-off matrix as it becomes available. Scores are assigned to each of the candidate architectures to aid in the final selection process.

Component mix evaluation is the process of simulating at more detailed levels of models than was previously done. The reuse library is accessed for the required components and if they are not available, models are developed as needed.

A verification plan is developed that ensures, to the maximum extent possible, that all hardware components will function and interoperate as expected and all SW will execute properly on the architecture when built.

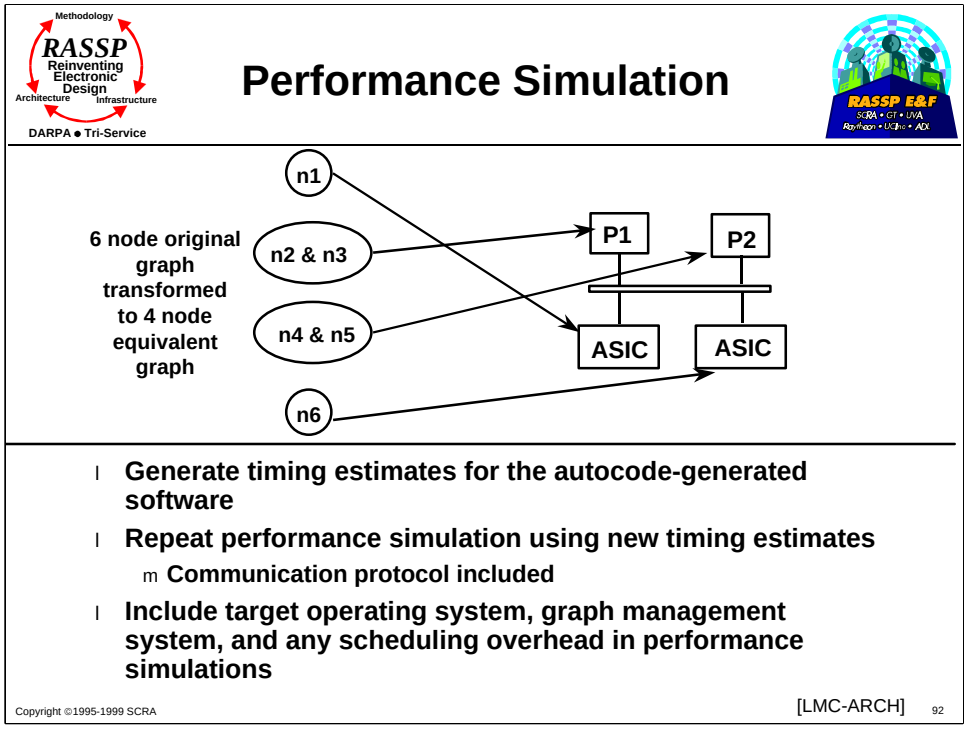
Simulation development enables incremental functional and performance evaluation of the HW and simulation models throughout the design process. Integrated tool suites are used to support the combination of testbed simulation, simulator(s), and/or emulator(s) to fully verify performance and code functionality before hardware implementation.



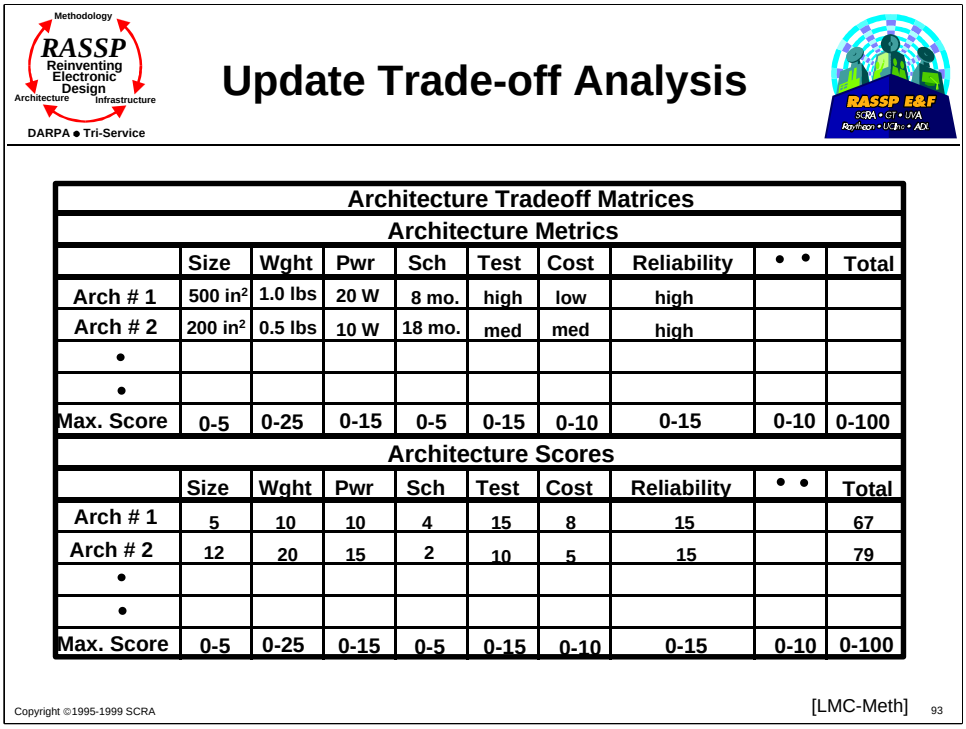
Autocode generation uses the DFGs, candidate architectures, and partitioning/mapping data from the previous process to translate the processor-independent flow graph information to target-specific code.

Optimized math libraries are used for time-critical DSP application code such as FFTs, FIR filters, etc.

The partitions are validated and functionality is verified against previous simulations.

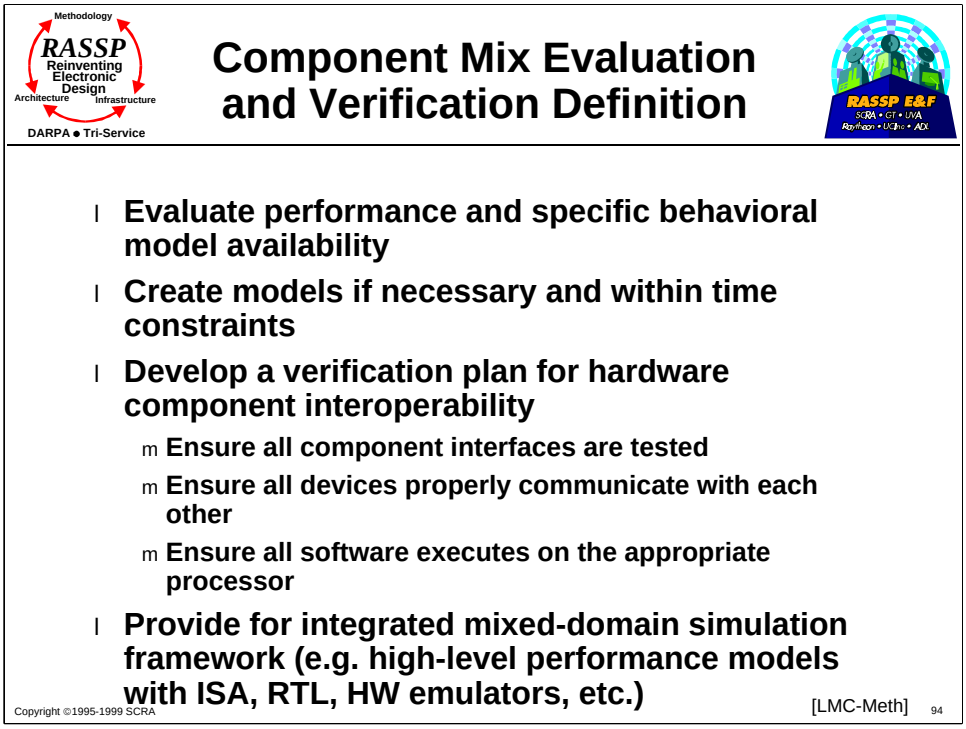


Performance simulations are done using the timing estimates for the autocode generated software. Communications protocol, target OS, graph management SW, and scheduling overhead is included to help improve the quality of the performance model simulations.



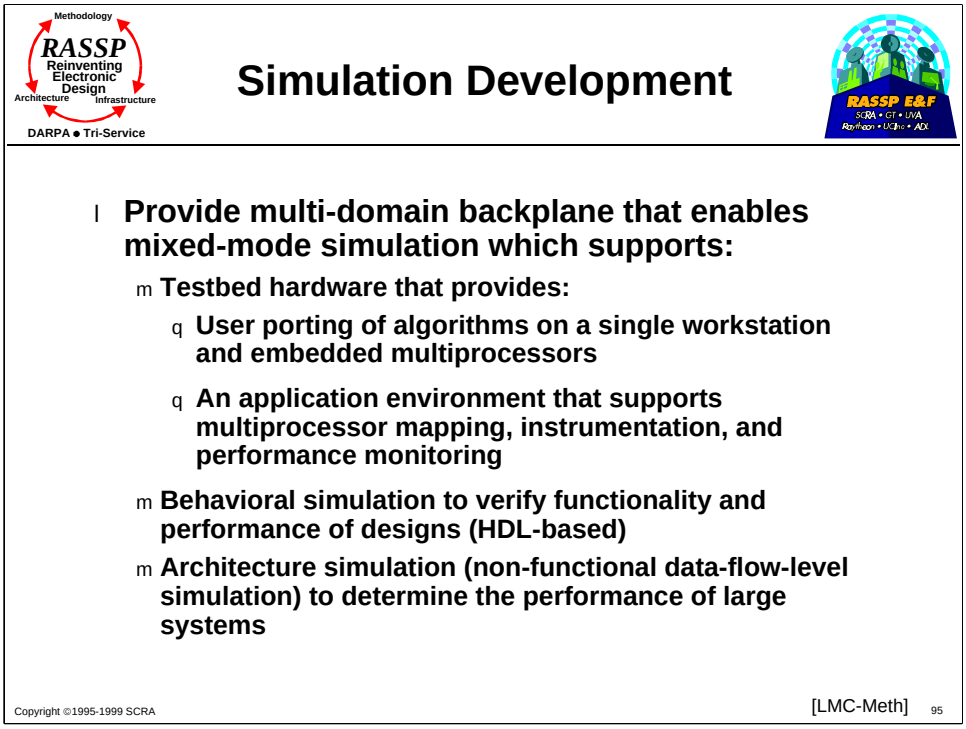
The iterative process of simulating the various architectures and the graph mappings to them results in the completion of the trade-off matrix shown above. This matrix is a record of the design process performed during the architecture phase. The entries in the table should be supported by detailed notes giving insight into the rationale for the numbers.

The output of trade-off analysis update is one or more candidate architectures that satisfy the requirements and are ready for more detailed design and analysis.



This stage in the process evaluates the availability of models and determines if any critical model development should be done in-cycle.

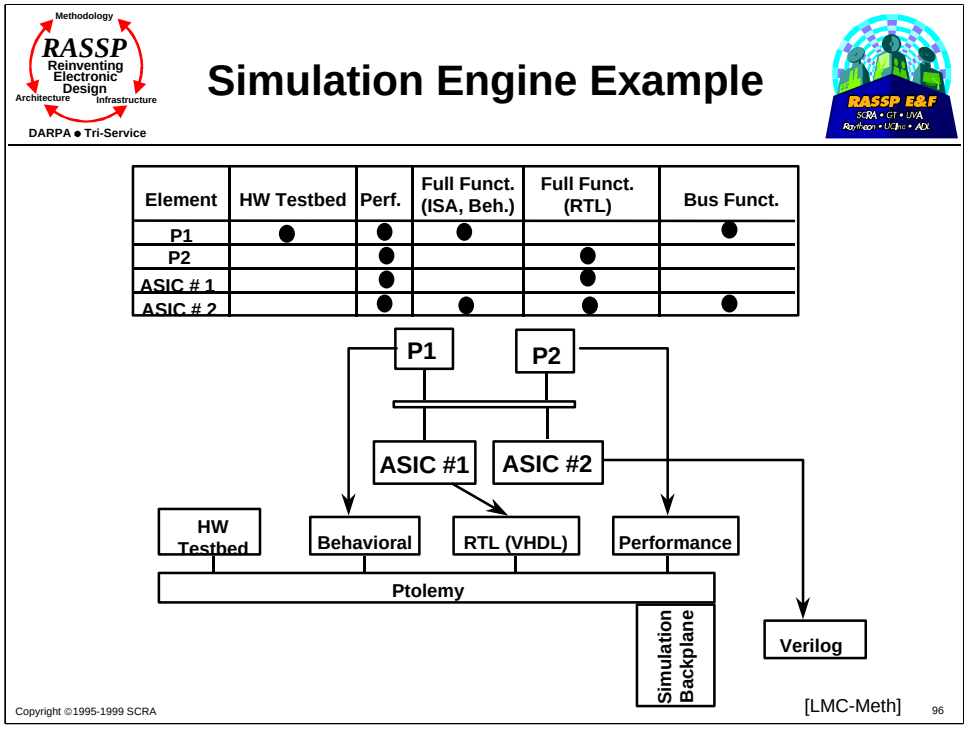
Verification plans are developed to test hardware component interoperability and SW/HW interactions are correct. Mixed-domain simulation should be done in an environment that support this paradigm.



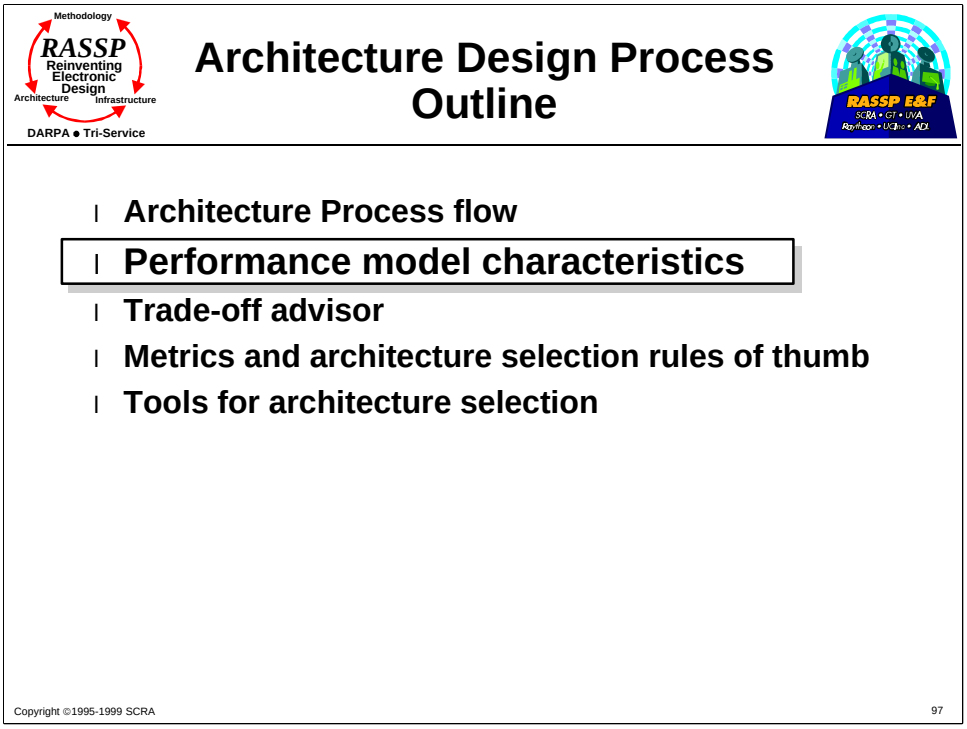
The simulation development should be done in a mixed-mode simulation environment using a backplane that supports its use.

The features which it must support are included above.

Testbed hardware includes such items as HW modelers, emulator boards, non-VHDL-based simulators, etc..



Based upon model availability, the architecture map be mapped to an appropriate simulation engine. The above figure illustrates one such mapping where processor #1 has a behavioral representation selected, processor #2 has both a performance and RTL level model available but possible due to simulation time requirements, the performance model is chosen. ASIC #1 has a VHDL RTL level model that is used and ASIC #2 has a verilog RTL level model. All are simulated in the multi-domain environment. The ideal is to support the interoperability of commercial tools to simulate a complete system in a seamless fashion.



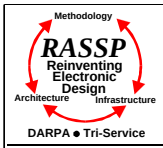
This section will describe the characteristics of performance models used for architecture selection and verification.

Performance Modeling Overview

- Apply during early stages of system development
- Use as a design tool, not a genie
- Aid evaluation of design alternatives
- Capture design decisions and assumptions
- Examine system behavior at boundary conditions
- Determine bottlenecks, overdesign, etc.
- Use for examining system sizing, topology, partitioning and capability issues

[Honeywell] 98

This slide presents an overview of how performance models can be effective used in the selection of architectures for DSP applications.

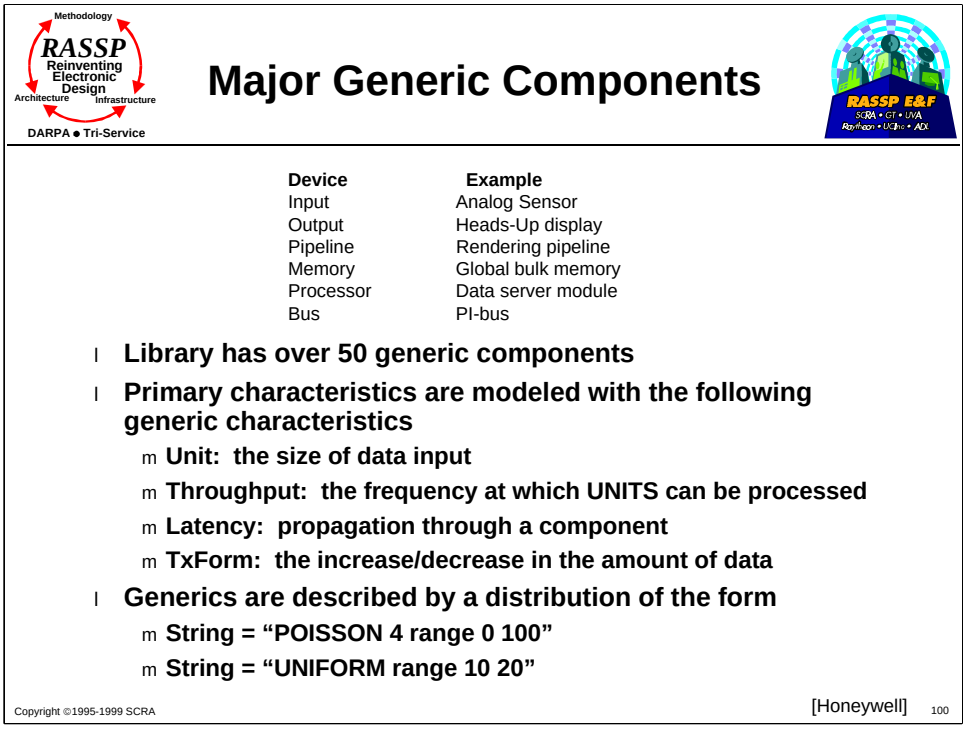


Performance Metrics

- **Performance metrics are generally in three categories:**
  - **Latency: the time it takes to complete a job**
  - **Throughput: the rate at which jobs are processed**
  - **Utilization: time busy as a fraction of the total time**
- **Used to evaluate designs against typical requirements:**
  - **Processor loading no more than 80%**
  - **Communications loading no more than 50%**
  - **Processing rate sufficient to meet requirements**
  - **System produces output with appropriate timeliness**
- **Generally not used for application level functional verification**

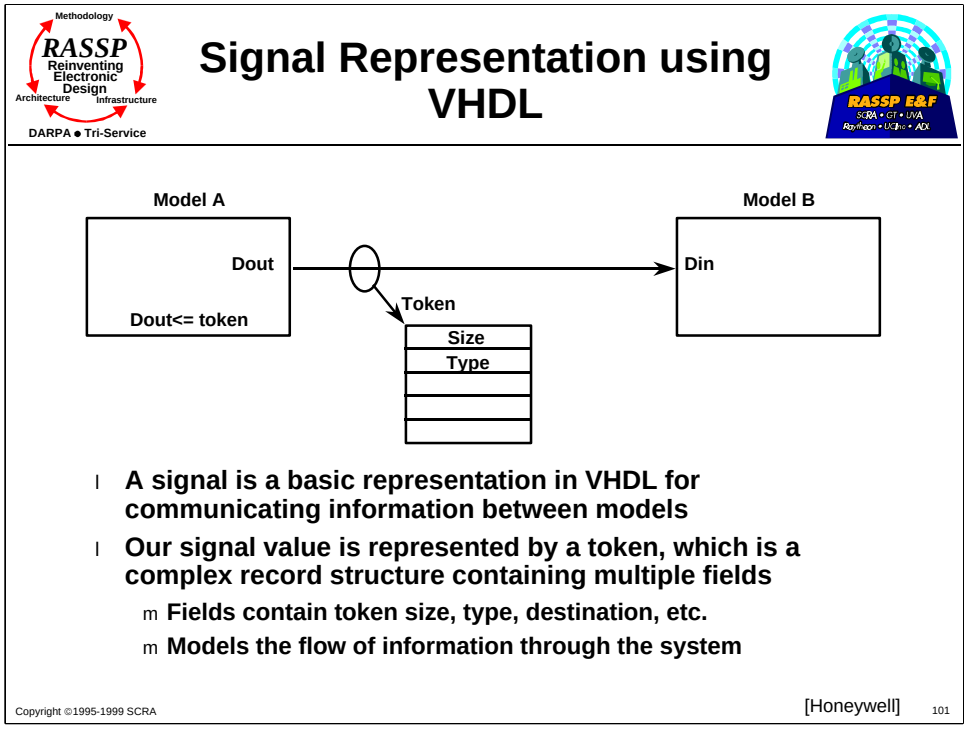
[Honeywell] 99

The metrics captured by using performance model simulations include latency, throughput, and utilization. These metrics are compared against the requirements to help determine the optimal architecture for the application.

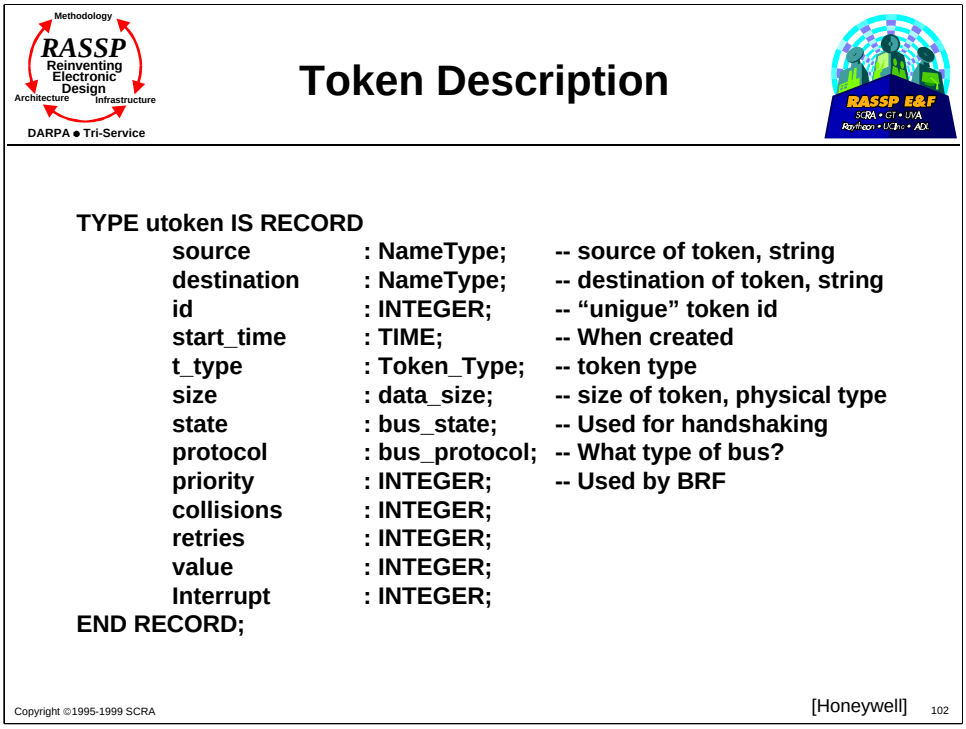


The major components contained in a performance model library are listed at the top of this slide. These include I/O devices, memory, pipelines, buses, and processors. These components are connected to form system architectures. The most complicated is the processor model which includes methods for describing software tasks and scheduling.

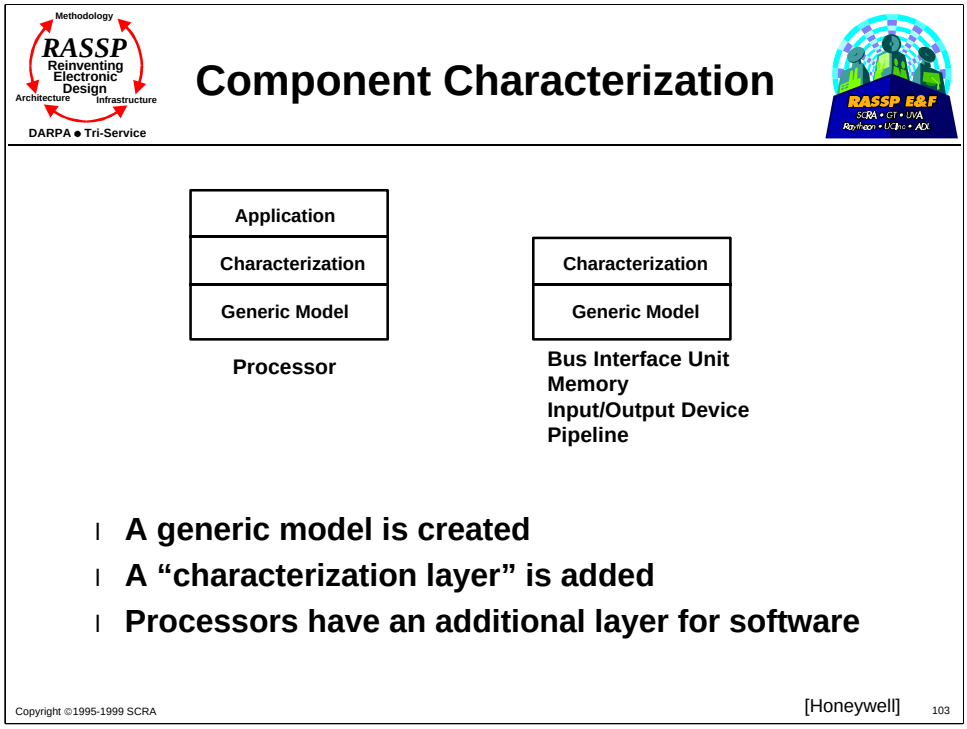
Distributions are used to model rates at which tokens are passed throughout the network.



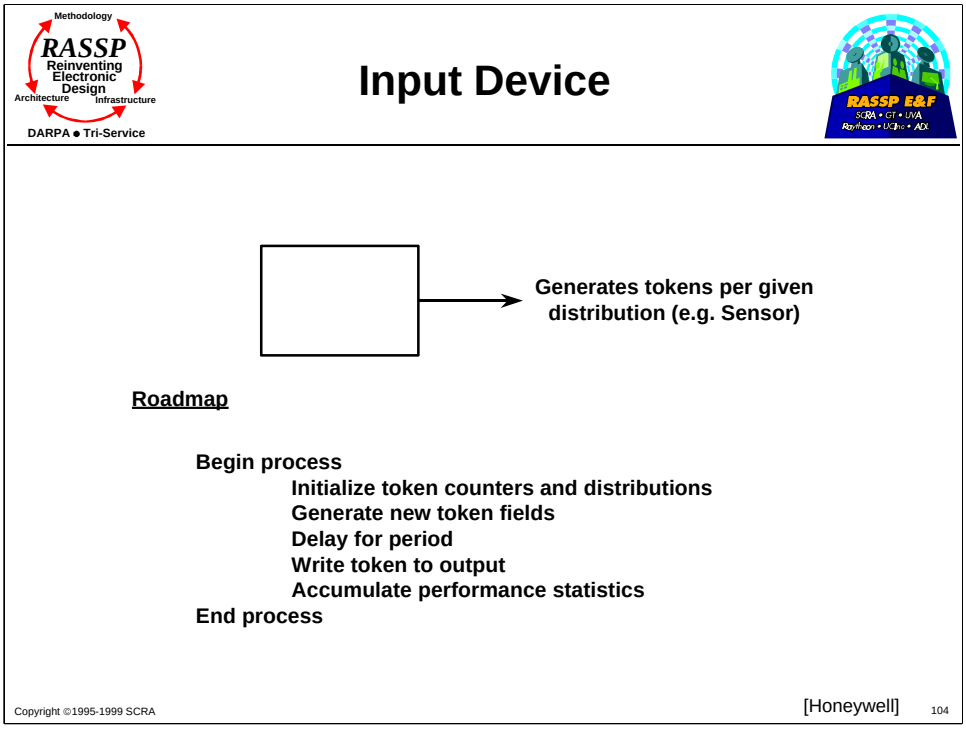
Tokens are the method used to model the information flow between components in a system. These tokens are represented in VHDL by signals with the record data type. The record contains fields containing the token size, type, destination, etc. and is described in more detail in the next slide.



The use of a common token definition is critical for the interoperability of abstract models from diverse sources such as libraries and other project groups. Honeywell Technology Center, under the RASSP contract, has proposed a token type convention for performance modeling, as shown above.

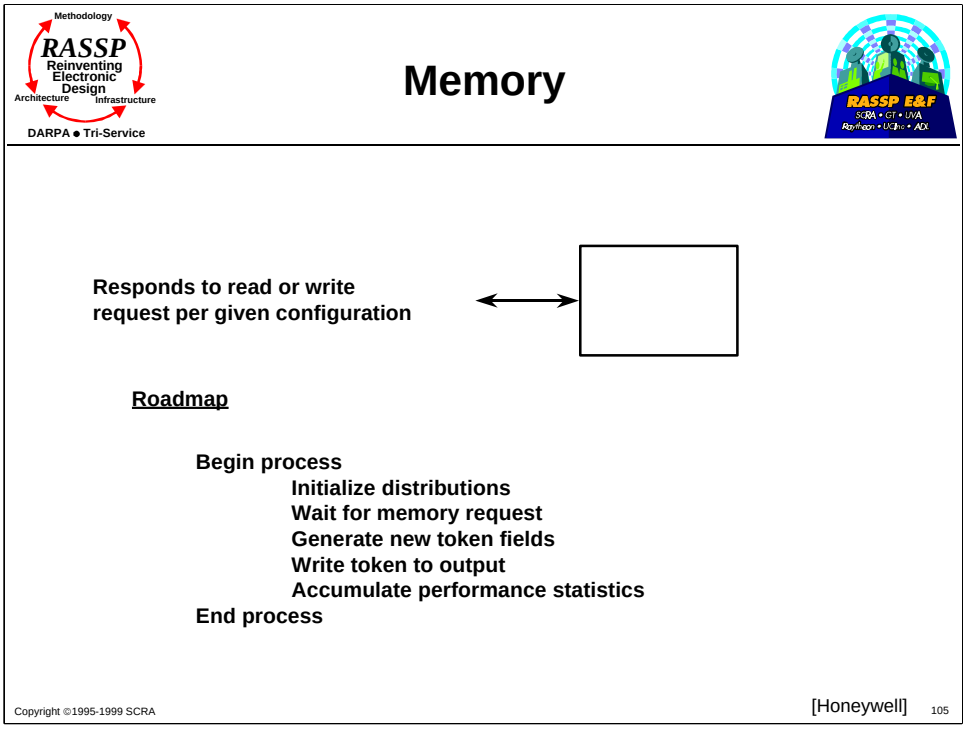


Component models in a performance library consist of layers. For simpler devices such as memories, I/O devices, pipelines, and bus interface units there are two layers while processors contain three. The first layer is the generic model which interfaces to the system architect to allow rapid modification of system configuration and characteristics. The characterization layer contains the functionality of device described at the performance level. This is the code that determines the behavior of the component. The application layer in processors is used to handle SW functions in processing elements.



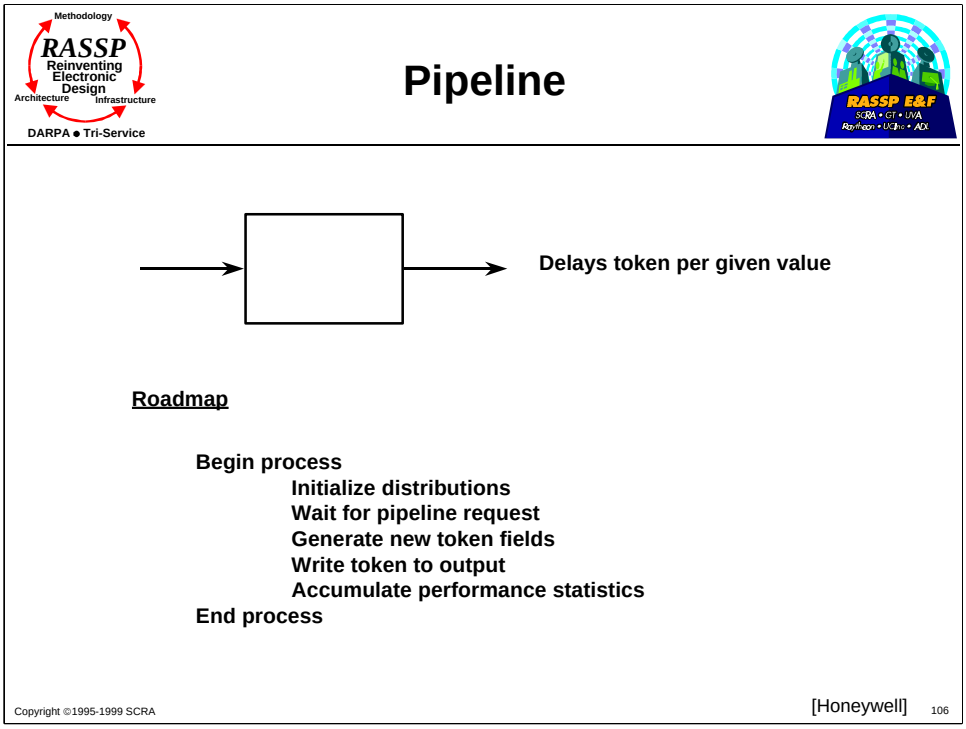
The input device generates tokens for a given distribution. The generic passed to the device would include parameters such as distribution type, values for the distribution (mean, std), etc. The characterization takes the form shown in the roadmap process above. It first initializes token counters and distributions, then generates new token fields, delays as necessary, writes the token to the output, and at the end it accumulates any necessary statistics.

These input models represent devices such as sensors and are the easiest elements to model. They tend to be purely data sources and can be characterized by the rate at which they can produce data.



A memories contribution to system performance can generally be modeled by a simple delay line. A request for data at a particular location results in the memory responding after some delay with that value. Caches are slightly more complex since they may respond with some failure indication or at variable rates. The attributes associated with memories include the physical size, the speed and access type, implementation technology, cache type, and expected hit ratios.

The process description above describes the high-level behavior of the memory device.



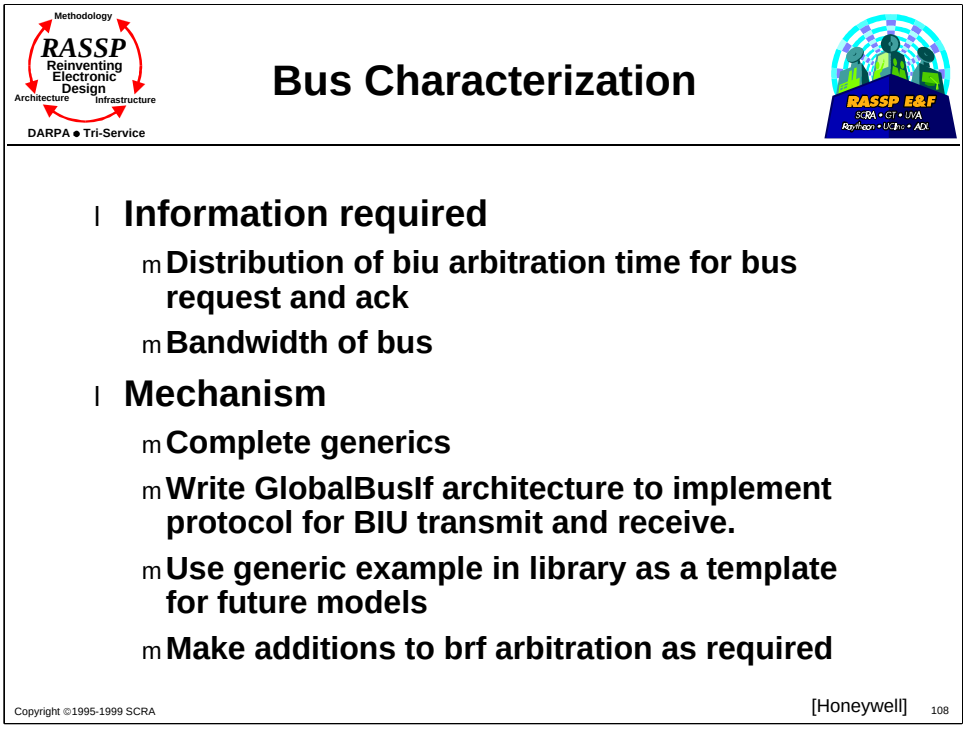
Pipelines are basically delay line elements where inputs cause outputs to be generated at a specified delay time later. As in all the previous examples, it behaves asynchronously because it waits for a pipeline request rather than a specified clock edge. This reduces the number of simulation events that occur.

Output Device

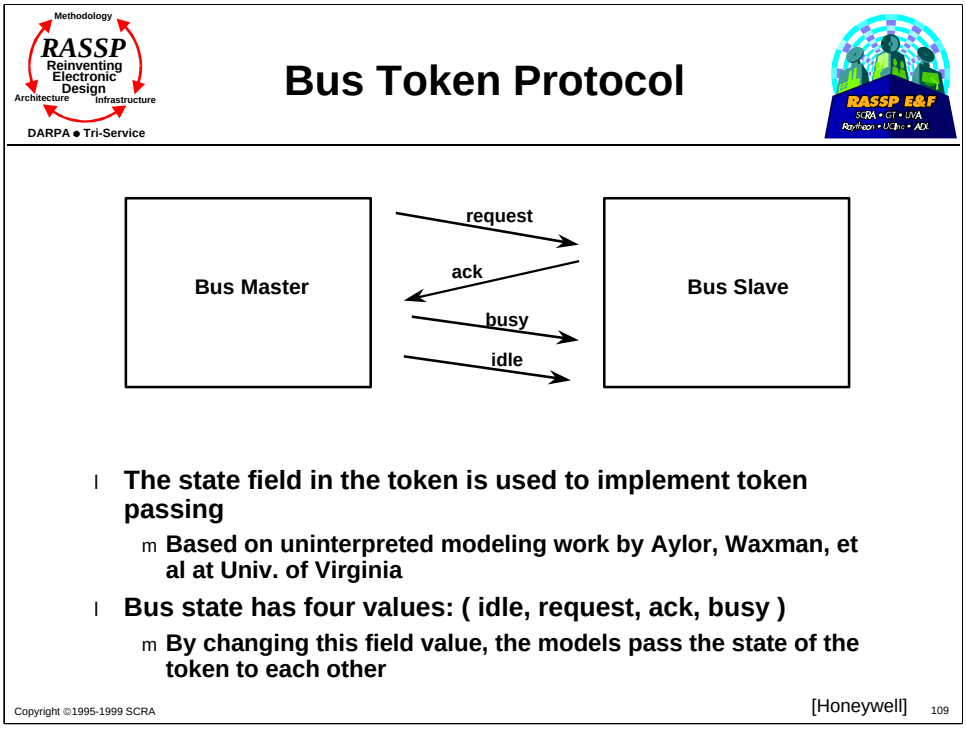
Roadmap

Begin process
Initialize distributions
Generate distributions
Delay for period and await input
Accumulate performance statistics
End process

Output device (e.g. displays) models accept tokens per a given frequency and can be described simply as data sinks. These along with input devices are the easiest to model.

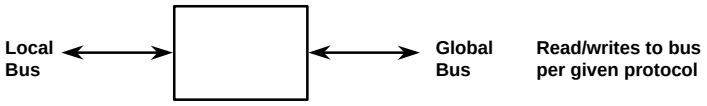


Since the global side of the communications interface needs to fit the protocol being modeled, some modifications to the VHDL code may have to be done. The information required by the model should include the distribution of the bus interface arbitration time for requests and acknowledgments as well as the bandwidth for the bus. The mechanism to due this is through a complete set of generics and a modification to the global interface side that implements the specified protocol. The current library has a template that can be used as a baseline model for which additional information can be added as required. The bus resolution function may also need some modifications based on the protocol being used.



The above diagram describes the bus token protocol used to pass information from components connected to the same bus. The state field in the token is used to implement the token passing. There are four states used to determine the handshaking, idle, busy, request, and acknowledge.

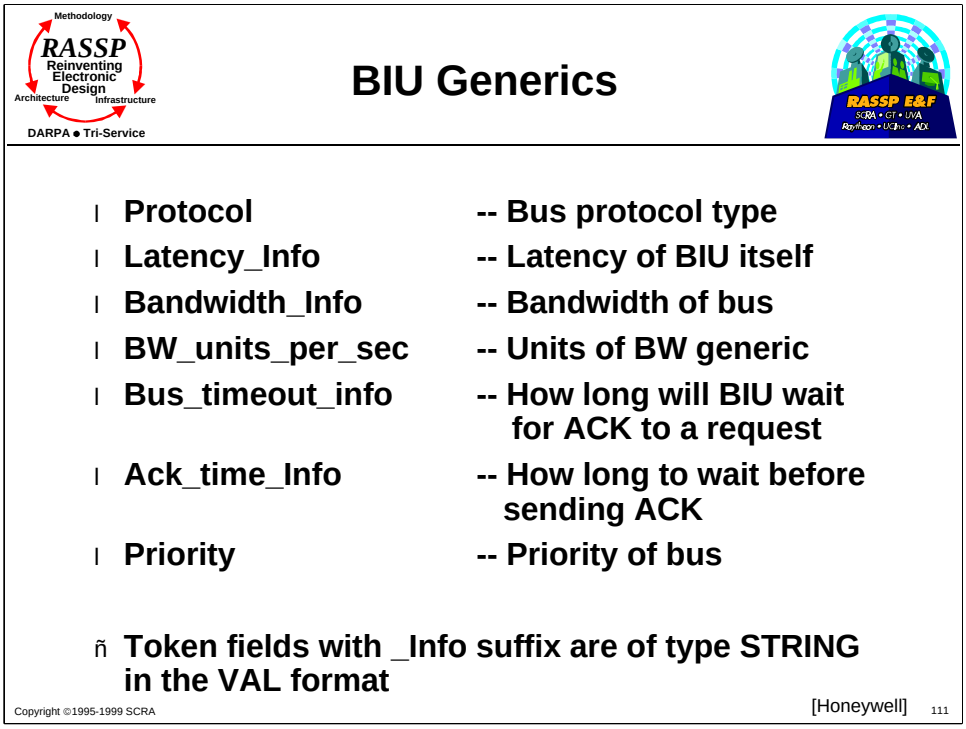
Bus Interface Unit (BIU)



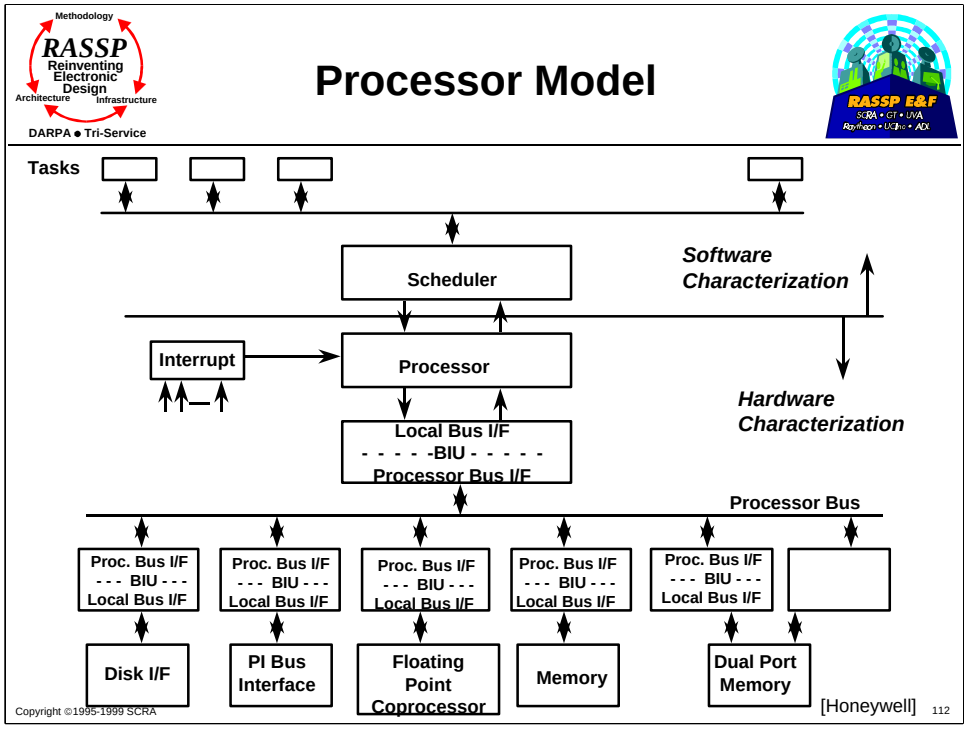
- **The BIU consists of four processes:**
  - **LBRCV receives tokens from a local bus**
  - **LBTX transmits tokens on the local bus**
  - **BIURCV receives tokens from the global bus**
  - **BIUTX transmits tokens to the global bus**
- **Global bus example is the PI-bus**
- **The local bus is a point to point connection to a driving element (processor, memory, etc.)**

[Honeywell] 110

The bus interface unit consists of four main processes to perform functions of reading and writing tokens to the local or global buses. An example of the global bus would be the PI-bus. The local bus connects to a driving element such as a processor, memory, etc.



This is a list of generics associated with the bus interface model. The protocol, latency, bandwidth, and times associated with the handshaking are specified. The VAL format is used for token fields with the _Info suffix.



The processor model consists of both HW and SW characterizations. The objectives of a processor model include:

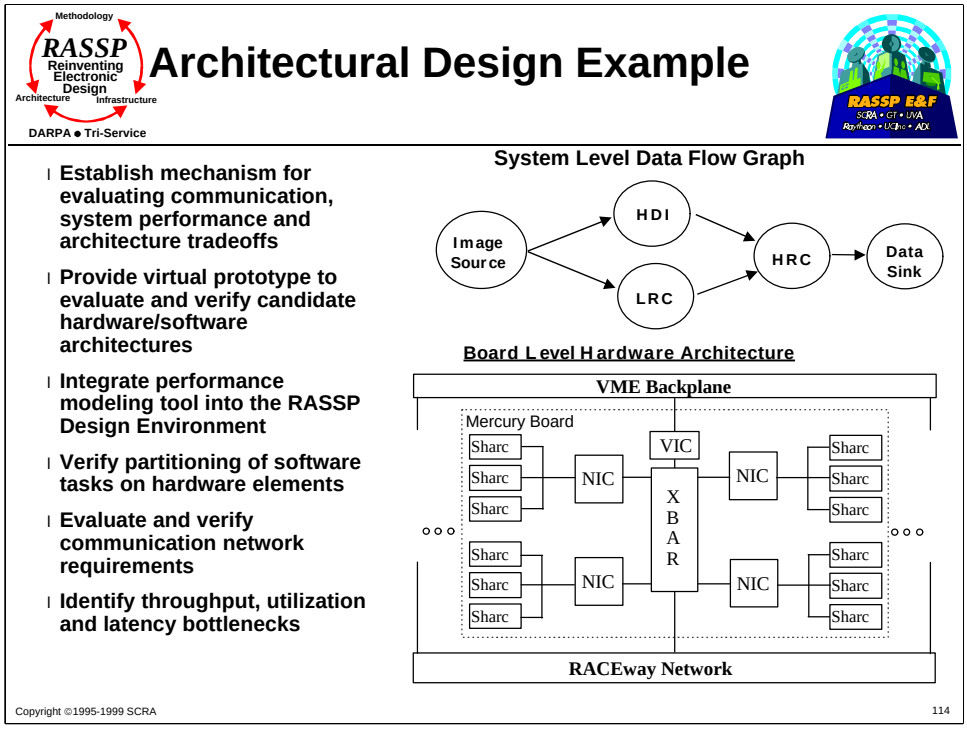
- Provide support for high-level SW modeling
  - Preemptive tasking
  - Static and dynamic task scheduling
  - Rate monotonic task scheduling
  - Interrupt service
  - Task communication similar to Ada rendezvous
  - Task synchronization, i.e., semaphores
- Control of dedicated functions such as coprocessors, BIUs, and memory
- Support command/response model behavior

VHDL Model Output

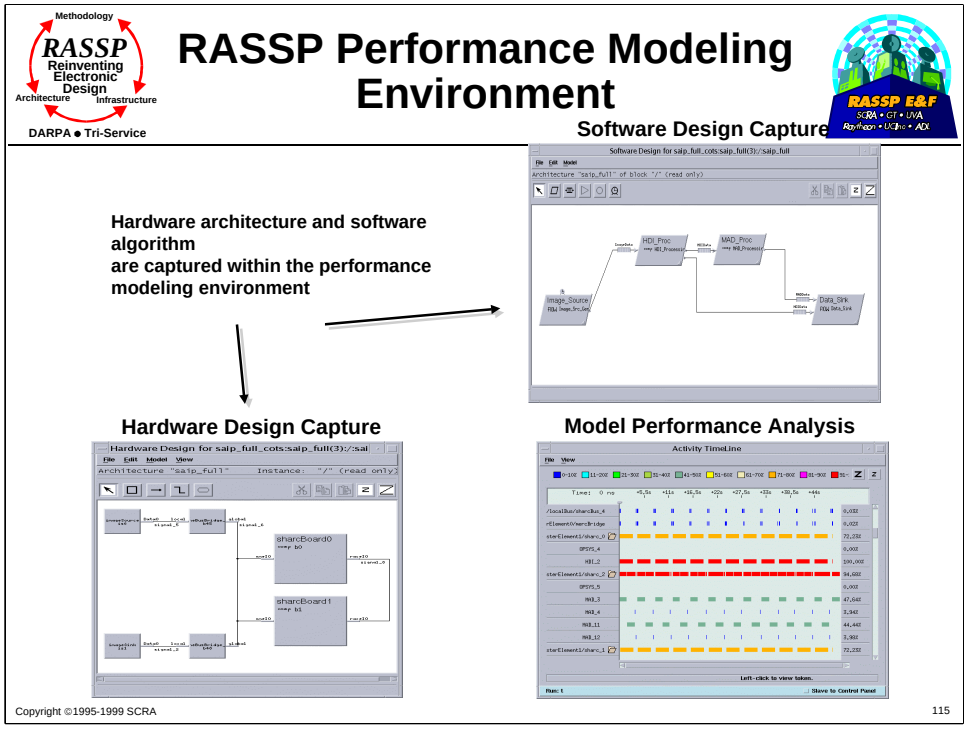
- **The VHDL model automatically captures three types of statistics**
  - **Utilization**
    - **Percent of time busy/total simulation time**
    - **Tracked for each entity**
  - **Latency**
    - **Time to move a token from point A to point B**
    - **Post processed from output**
  - **Throughput**
    - **Data processed/time period**
    - **Tracked for each entity**
- **Trade studies are interpretations of these statistics**
- **Raw data displayed in intuitive form**

[Honeywell] 113

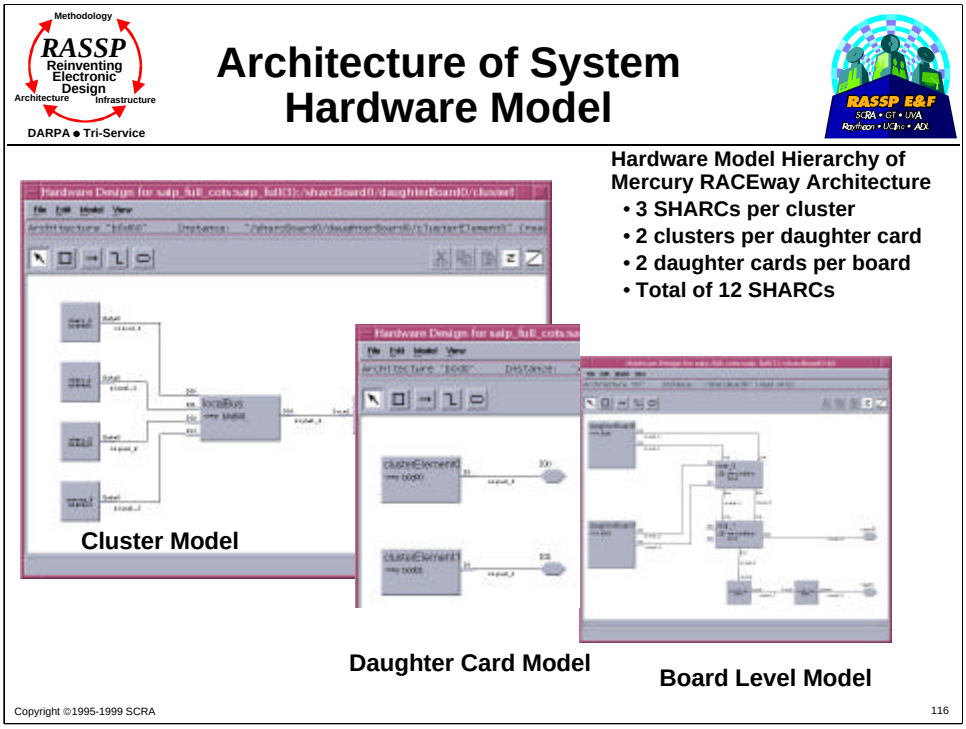
The utilization, throughput, and latency information for the system model are captured automatically by the simulation. Trade studies can be done using these statistics to determine the optimal architecture for the system application. The raw data from the simulations is displayed in an intuitive form as shown on the next slide.



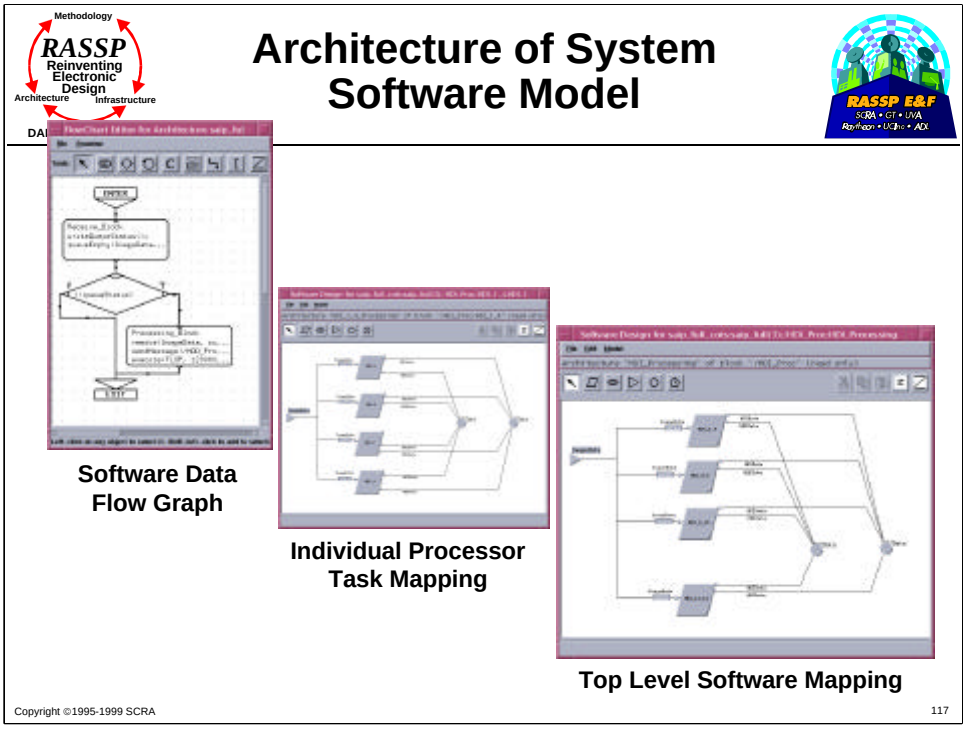
In this example we should some preliminary steps that map an algorithm to a candidate architecture using timing models of software executing on target hardware (called performance modeling). In this case an algorithm shown (top right) in the slide is mapped and scheduled onto a candidate architecture shown in the bottom right part of the slide.



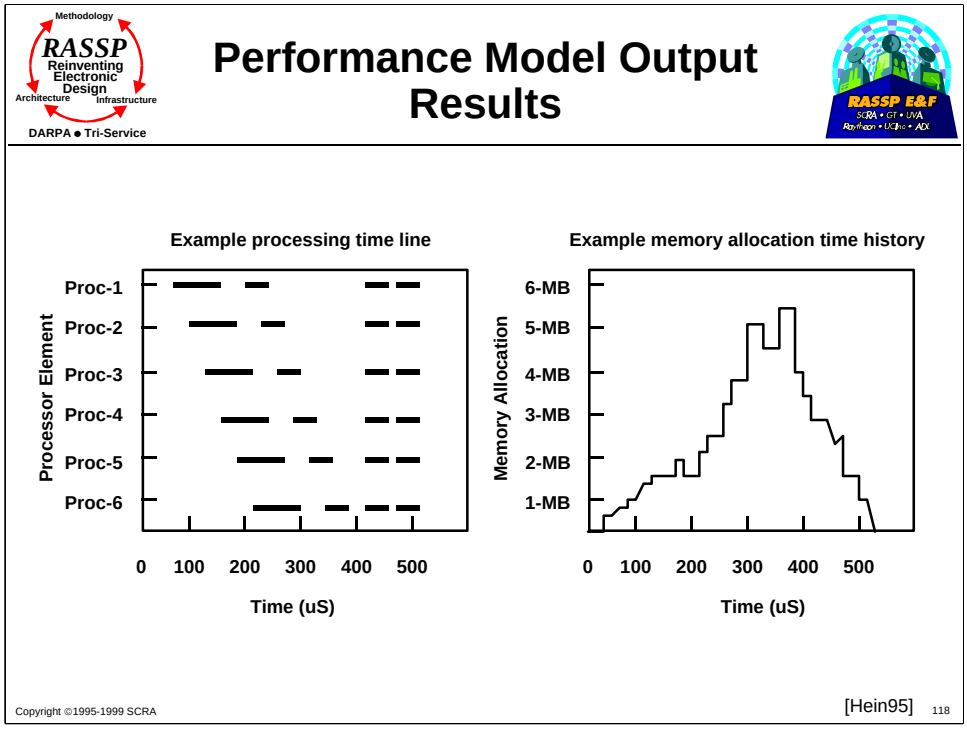
The performance modeling environment (in this case Cosmos) allows capture of the hardware elements of the architecture and also the software elements of the algorithm (according to a certain granularity), and then allows mixed performance analysis of throughput, utilization, and latency of tasks (as shown on the bottom right).



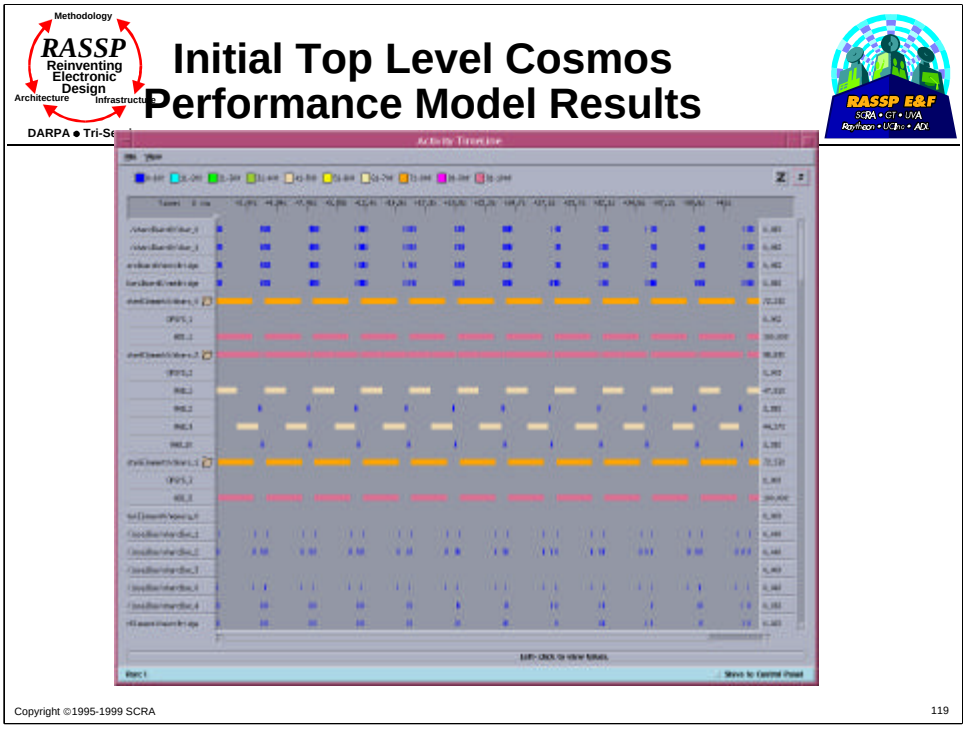
The steps in the modeling of the hardware architecture at the card and the board level are shown, using a total of 12 SHARCs.



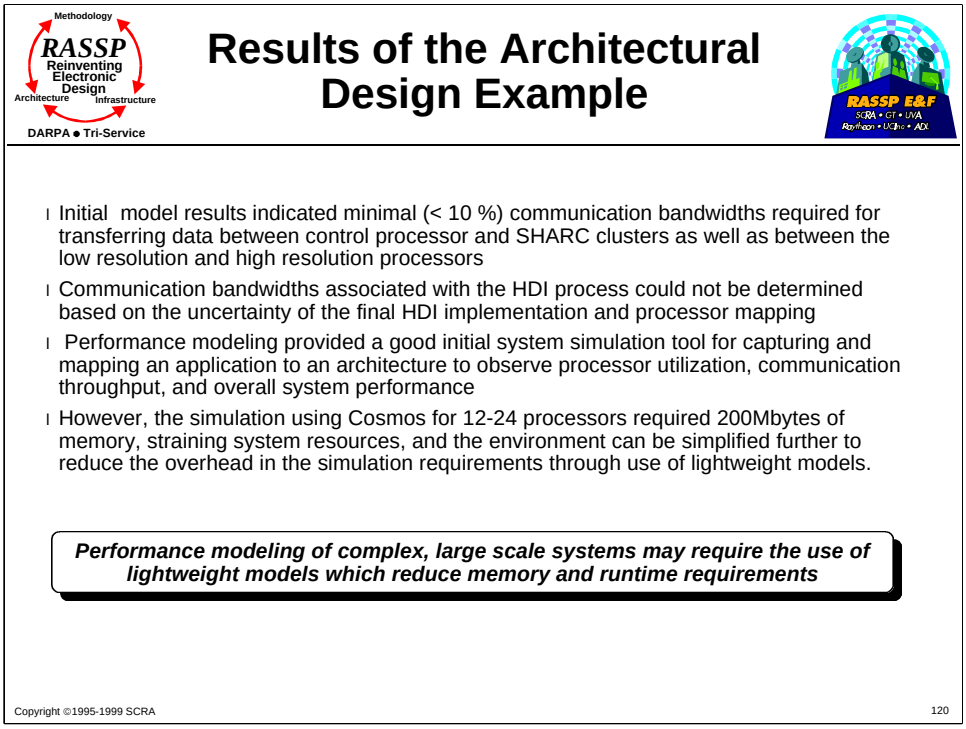
The software task graph is also modeled together with a control flow graph/assignment/mapping schedule that is utilized to merge the hardware and software graphs together.



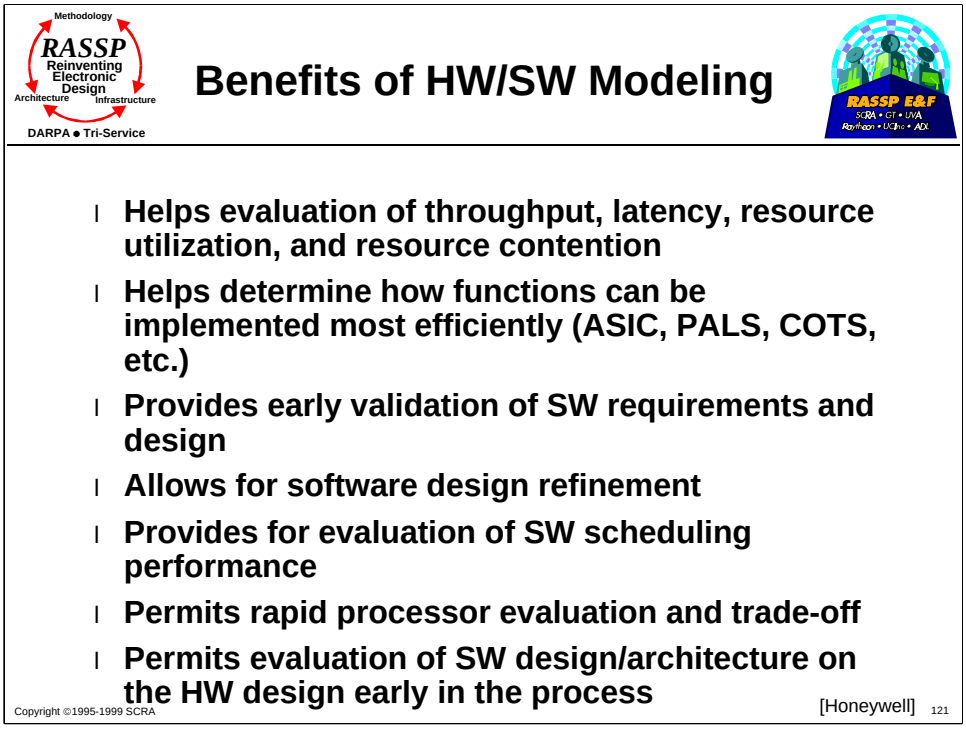
These are some example plots of the output of performance model simulations. The left side plot shows processor utilization vs. time and processing element. The right plot shows how the memory is utilized during the same time line.



The result of the virtual prototyping at the performance modeling level shows various metrics of latency, throughput and utilization for various mapping and scheduling strategies for executing the algorithm on the candidate architecture.



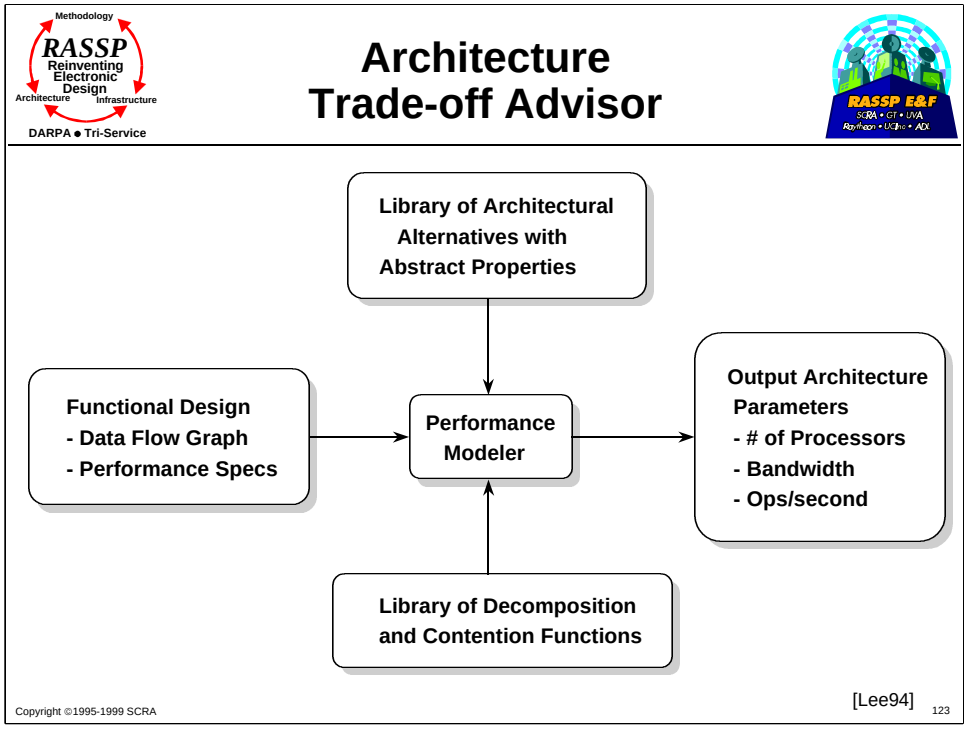
The results of the HW/SW modeling at the architectural level in this example confirm that the bandwidth requirements of the application do not strain the underlying architecture, and provide a good idea of the capability of the architectures. However, when modeling large systems the burden of the modeling environment can be quite severe in terms of memory usage, and light weight custom performance models can also be constructed to perform the architectural design.



The above slide lists the major benefits for modeling both hardware and software in the processing element. With the addition of software characteristics, a more thorough evaluation of the throughput, latency, and resource utilization can be done. It helps determine how the various functions can be implemented most efficiently as well as provide an early validation of the SW requirements. It allows for SW refinement in the traditional hardware design phase. The SW scheduling can be evaluated as part of the trade-off studies in architecture selection.

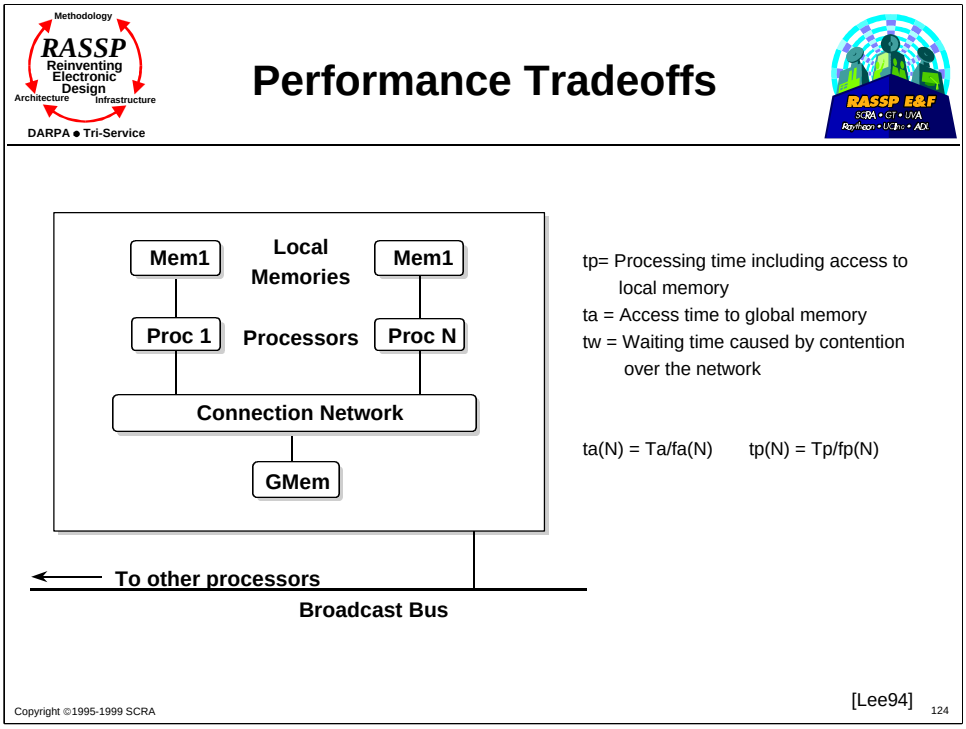
Architecture Design Process Outline

- Architecture Process flow
- Performance model characteristics
- **Trade-off advisor**
- Metrics and architecture selection rules of thumb
- Tools for architecture selection



The architecture trade-off advisor takes as its inputs the functional design in the form of a data flow graph and some performance specification(s). It also uses libraries of architectures and decomposition functions captured for reuse. These are input to a performance modeler.

The output will be a count of the number of processors required to perform the functions and well as bandwidth requirements for communications and a count of the complexity of the algorithm.

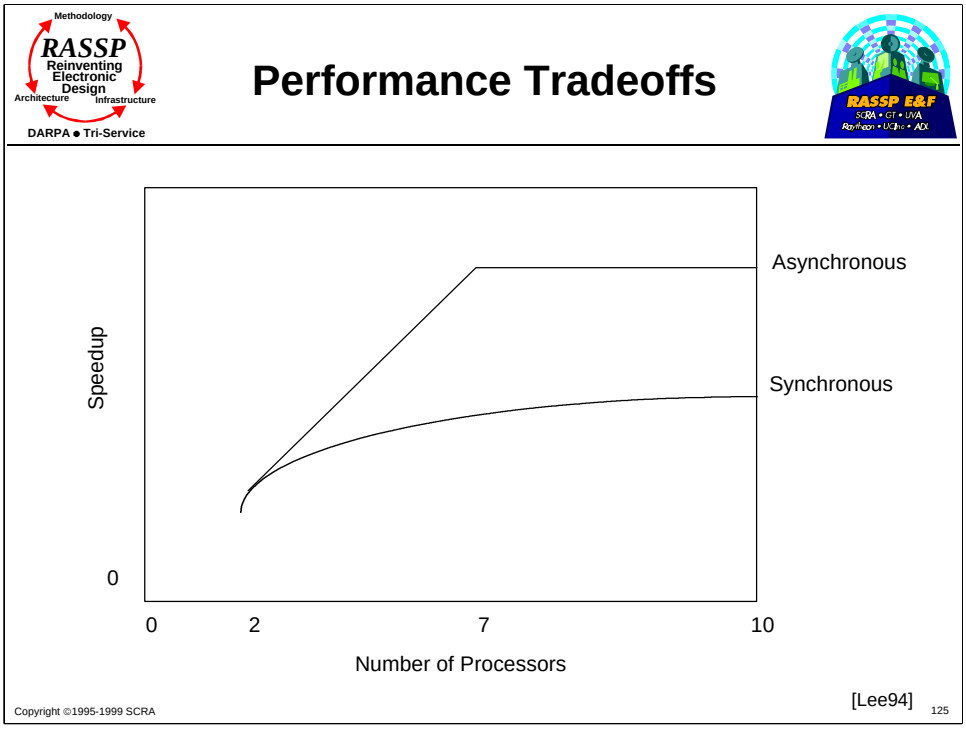


The architecture advisor chooses a hardware architecture based on the predicted performance of the decomposed tasks.

Performance prediction models may exist for multiprocessors containing N processors, each with its own local memory for code and data. In addition, each processor can access shared memory via an interconnection network.

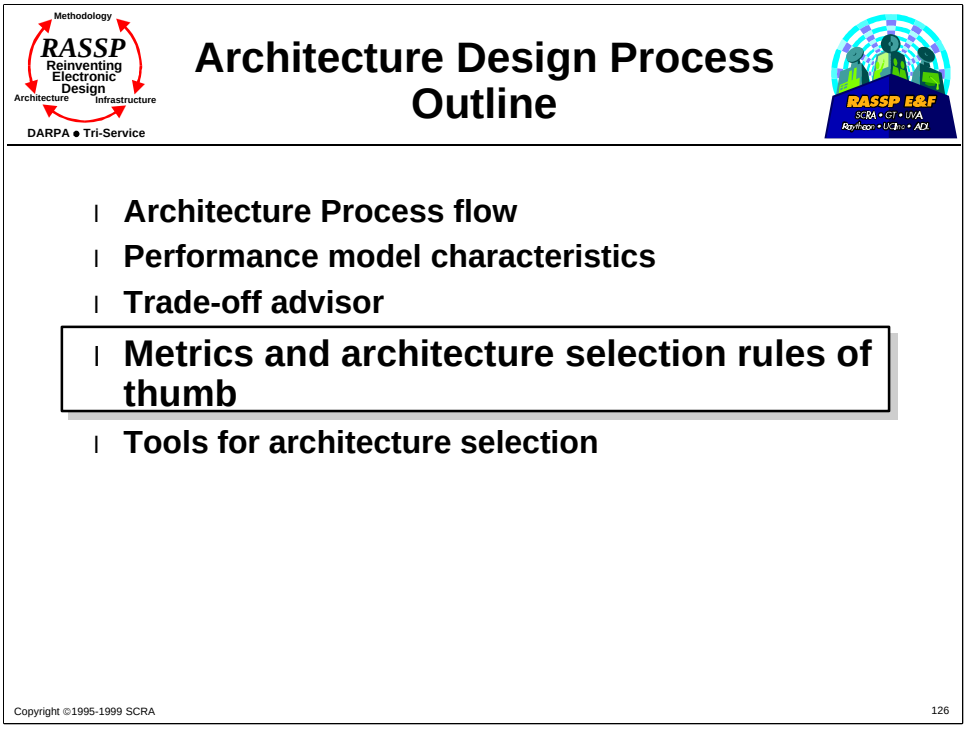
Performance tradeoffs are done using high-level models of the respective components chosen for the architectural candidates.

Timing information such as access time to local memory, global memory, and network data and message passing are included in the model.

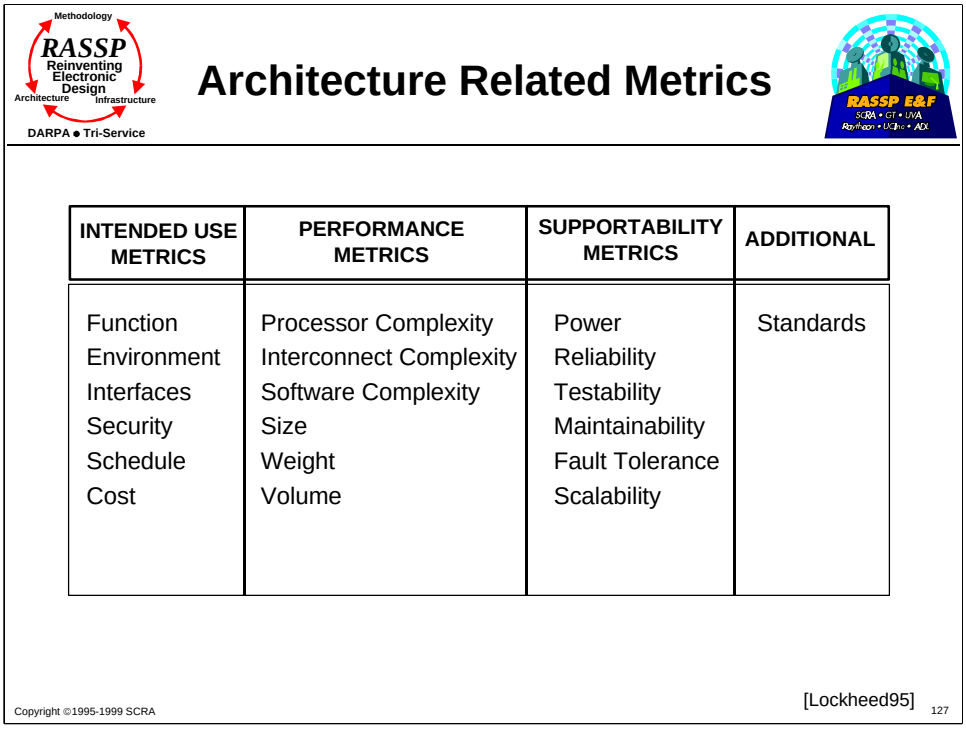


This plot shows how a typical trade-off curve may look as the number of processors are increased for the previous slide's example. We see that the speedup continues to increase in both cases until about seven processors are utilized. At this point an additional processor does little to improve the performance.

We also see the asynchronous case has a higher speedup as compared to the synchronous case.



This section will describe some of the metrics associated with the architecture design process and give some basic rules of thumb in choosing and architecture.

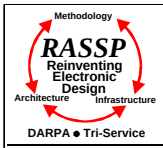


In addition to performance metrics, the above listed metrics are important in helping to choose the correct architecture to use for the application.

A rating can be placed on each metric and, during selection from a few candidates, the weight associated with a particular metric is factored into the equation for choosing the best architecture.

There are three main classes of metrics. They include:

- Intended use metrics
- Performance metrics
- Supportability metrics



Architectural Selection - Rules of Thumb

- **Select interconnect by scope**
- **Select interconnect by speed**
- **Select processor type (scalar, vector, DSP etc)**
- **Use serial interconnect**
- **Use field programmable devices**
- **Use open systems**

[Lockheed95]

This and the next slide list some general rules of thumb to consider when selecting an architecture to solve a particular problem.

Select interconnect by scope: Within a module use direct interconnects; between modules use standard interconnect protocols (VME, SCI, etc.).

Select interconnect by speed: Organize the program data flow from sensor inputs to output. Select a scheme to satisfy the requirements of each section. Sensor input may be much higher than output display information. In this case, different interconnects can be used.

Select processor type: Decompose the algorithm into a set of processing functions interconnected with direct data links. For each function, define the type of processing required (vector, general purpose, etc.). Attempt an initial direct-mapped architecture and, if the solution is overkill, reduce the complexity by using various SW modules.

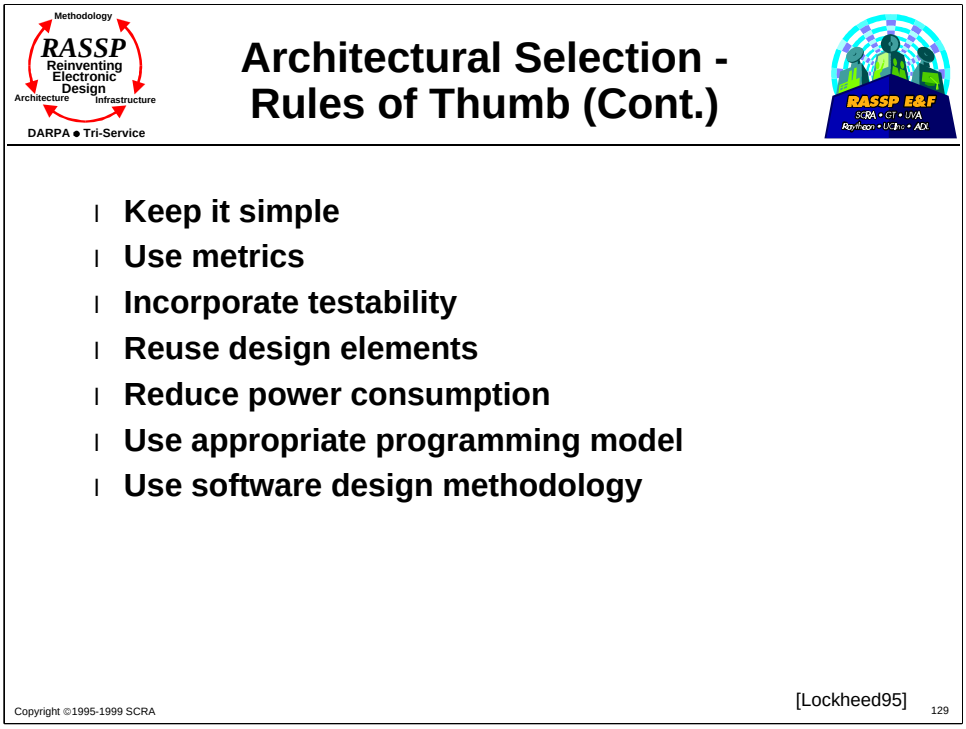
Use Serial Interconnect: It is better to select an interconnect with fewer pins.

Use FPGAs: If maximal speed is not applicable, it is better to use FPGAs as compared to ASIC solutions. They provide support for

Rapid development

- Frequent interactions with the customer
- Commercial IC advances

Use open standards: It is better to use open systems architecture constructs to provide for expansion, upgrading, or functional reconfiguration through the use of replaceable modular elements. This applies to both HW and SW.



Keep it simple: Most often simple and straightforward concepts with clean partitions and interfaces are desired.

Use metrics: The list of metrics was presented earlier. Going over the metrics ahead of time can speed choice of an acceptable design. The metrics can be broken into three categories: Intended Use, Performance, Supportability.

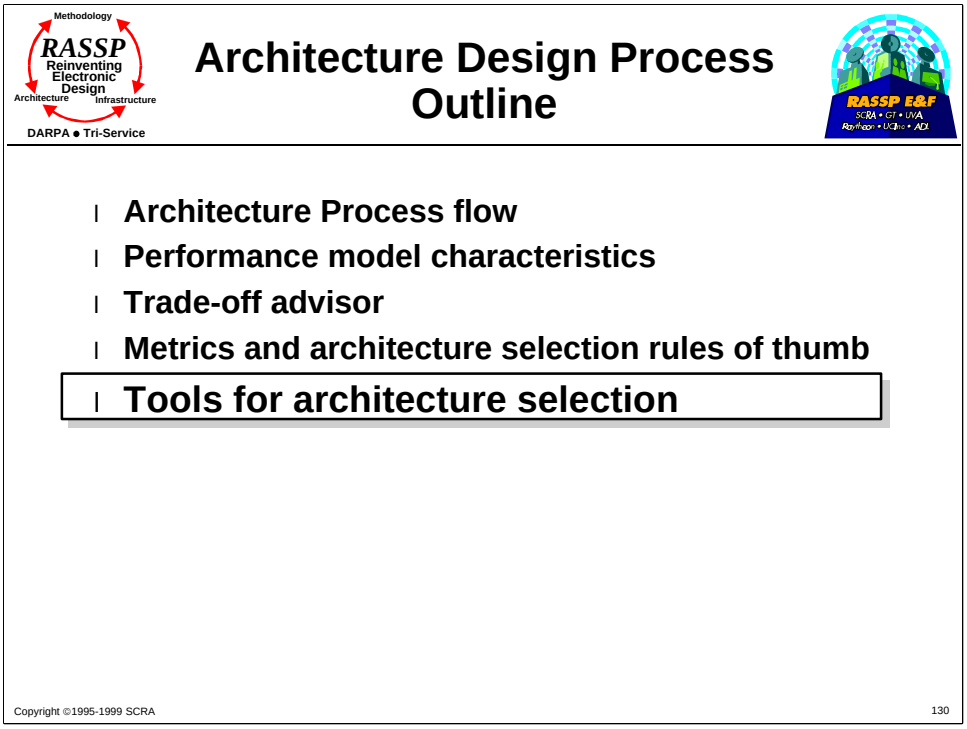
Incorporate testability: Because complex, compact electronics cannot be probed externally, some form of BIST is almost certainly required.

Reuse design elements: Reuse of previous designs improves reliability and reduces development time and cost. Test time is usually decreased because reused modules have already been tested. Life cycle costs can also be reduced because such things as manuals, training, test equipment, and spare parts already exist.

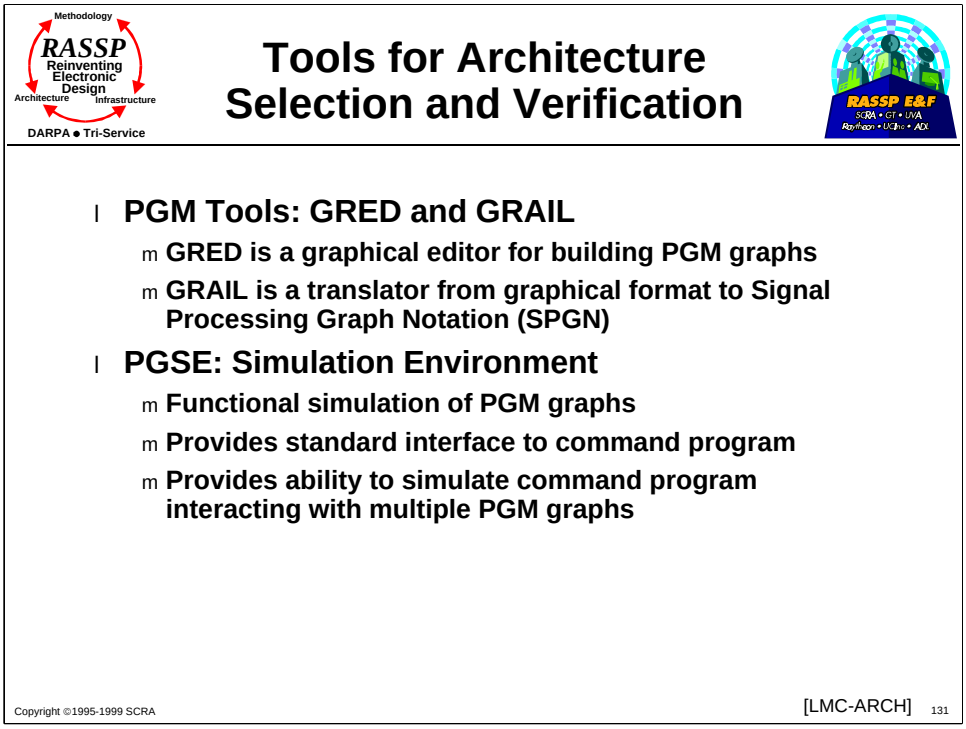
Reduce power consumption: Because heat dissipation is expensive in terms of cost and weight, less heat is preferable. Select components designed for the portable electronics market where this is a critical parameter. Smaller feature size is also helpful and lowers power supply voltages.

Use appropriate programming model: Choose SIMD or MIMD based on the type of processing that is required by the application. Communication with respect to loose or tight coupling is also an issue. The choice of homogeneous or heterogeneous processing elements is also important.

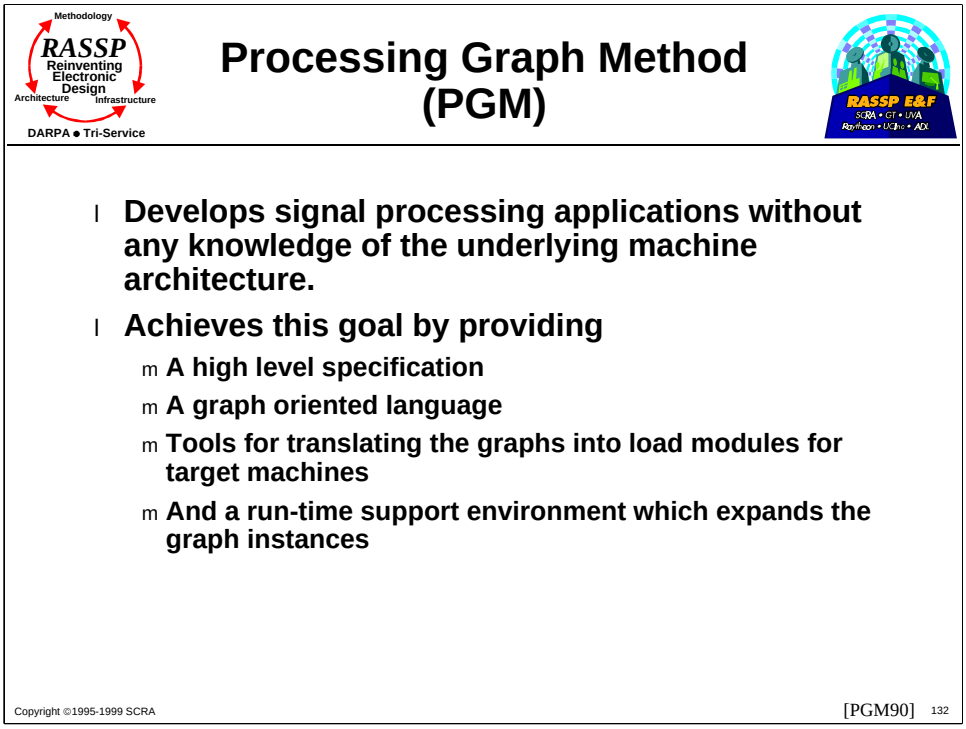
Use software design methodology: Choose OOD when possible. The resulting modular structures are more suitable for automatic code generation, easier to code by hand, and easier to test and maintain.



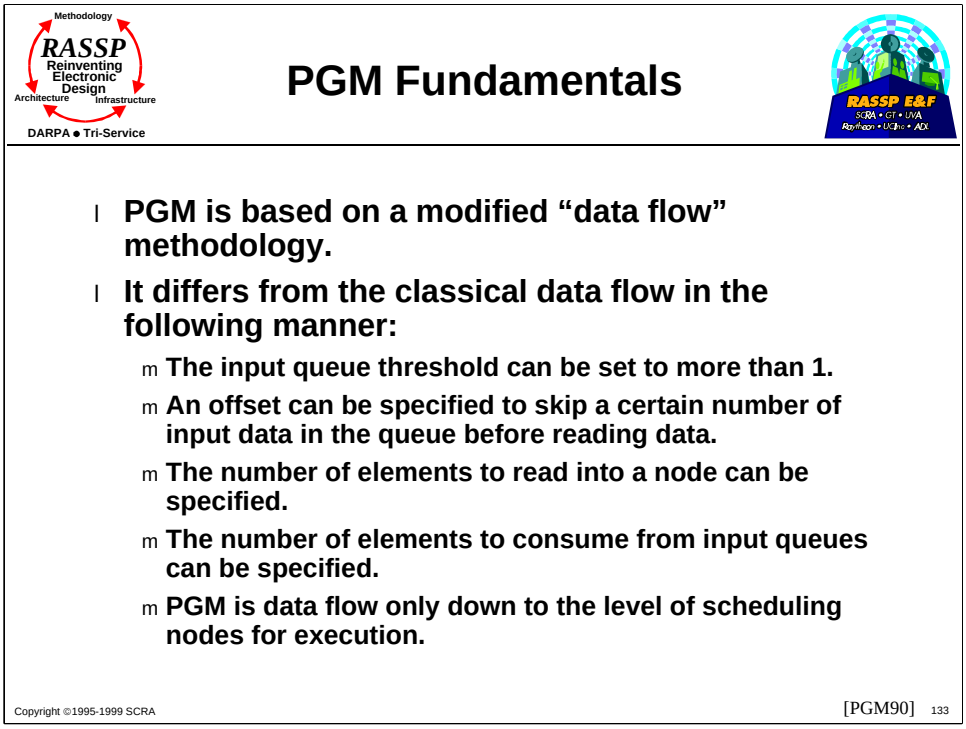
Tools are an important part of the architecture design process. RASSP is attempting to address the need for additional tools to fill in the gaps in the architecture design process.



The main tool used by the ATL branch of Lockheed-Martin is the PGM tool developed by the Naval Research Labs. PGM is used to model the data flow of the system and contains hundreds of primitives written in Ada to develop algorithm designs. GRED and GRAIL are graphical tools to aid in PGM development while PGSE is the simulation environment used to perform the functional simulation on PGM graphs.

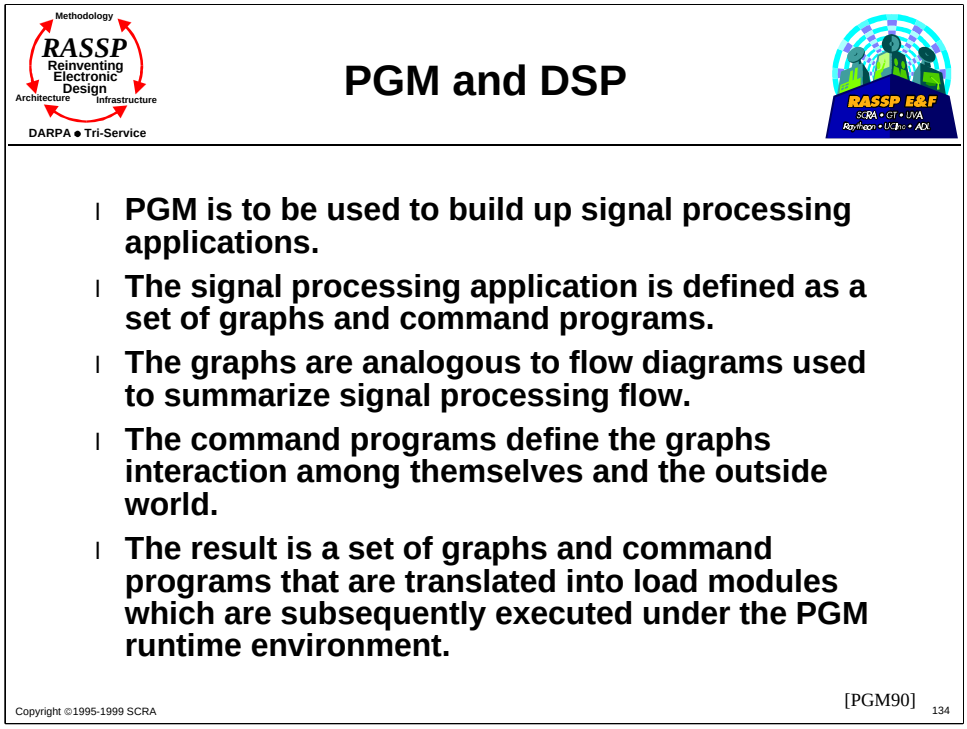


The processing graph method (PGM) develops signal processing applications at the data flow level without the implementation details.

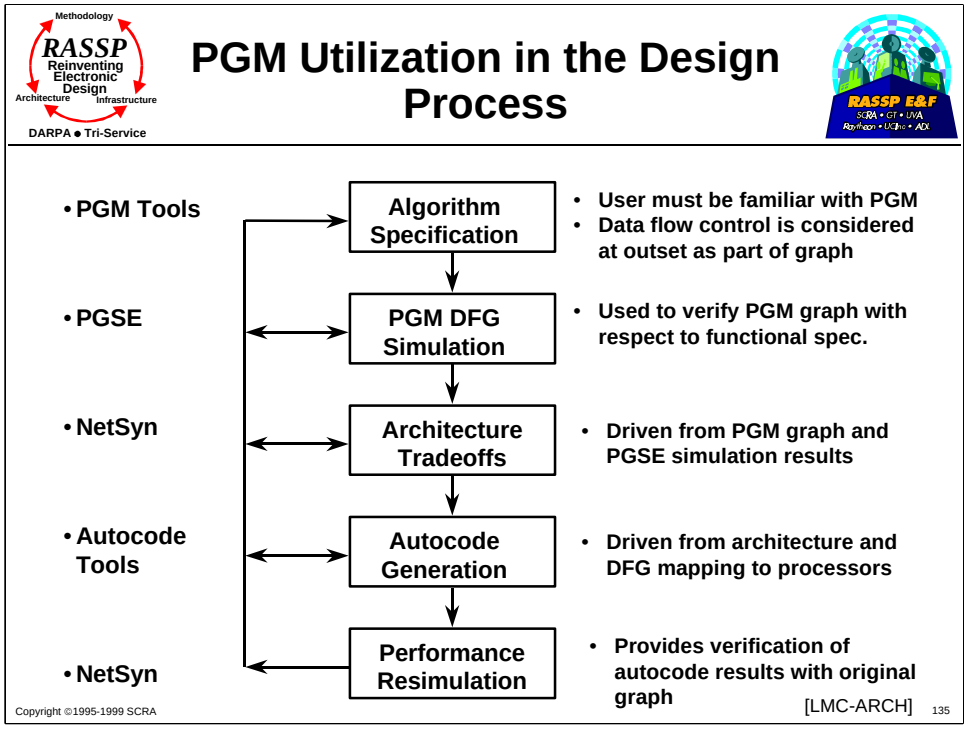


Dataflow Vs. Control flow

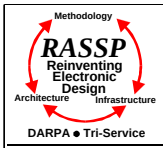
In a Control flow paradigm the order of execution of elements of the program are embedded in the program description. On the other hand, in a Dataflow paradigm the execution of the elements are based on the availability of the data. The environment provides a set of atomic operations that can consume data and produce data. Upon the execution of the program, the elements that have data ready will produce data that will enable other elements to be ready for execution.



Signal processing applications are normally described using block diagrams that lend themselves very naturally to Dataflow paradigm. The block diagram is built out of certain black boxes that perform certain functionalities. These black boxes are further described based on more primitive signal processing elements. Each box in the block diagram processes the data that appears at its inputs and provides the result at the output where it is used by another box.



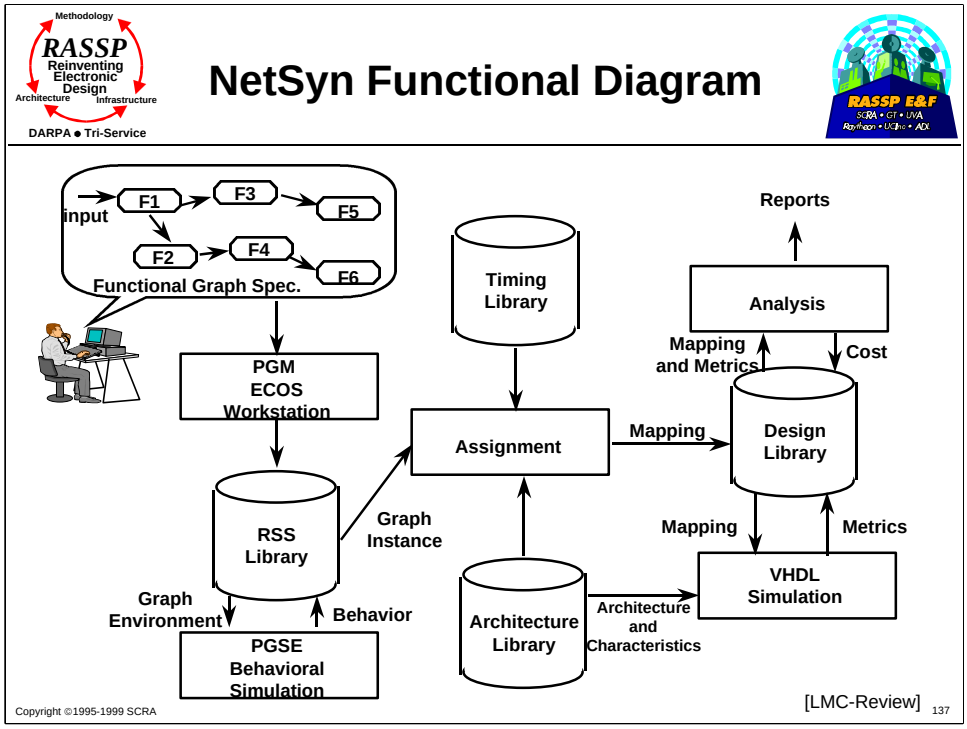
PGM is currently being used in the design process at the algorithm specification level. Data flow graphs are generated in PGM from Ada primitives. These are used to simulate the processing flows for the given application. The PGM primitives are simulated using the PGSE environment. After satisfactory simulation results are obtained, the PGM graphs are input to the NetSyn tool to begin architecture trade-offs.



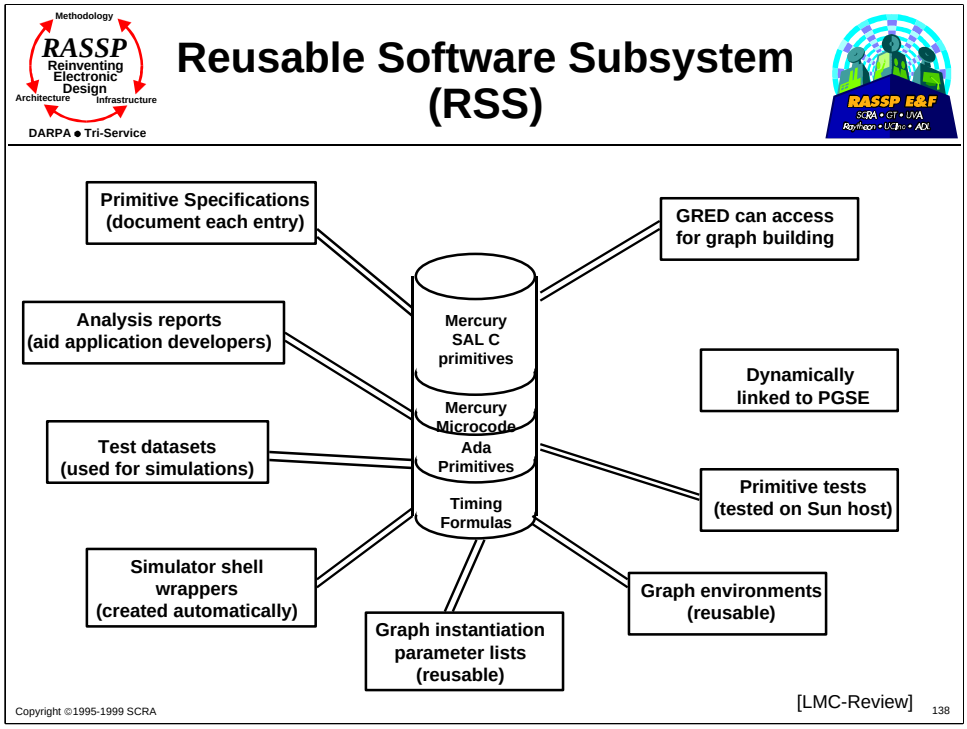
Tools for Architecture Selection and Verification (Cont.)

- **NetSyn: (Assignment/Performance Simulation)**
  - **Driven from PGM data flow graph and PGSE output**
  - **Performance simulation for candidate architectures**
  - **Library primitives**
    - **PGSE executables**
    - **Target dependent timing database**
  - **Outputs include reports, architecture, mapping, and graph**

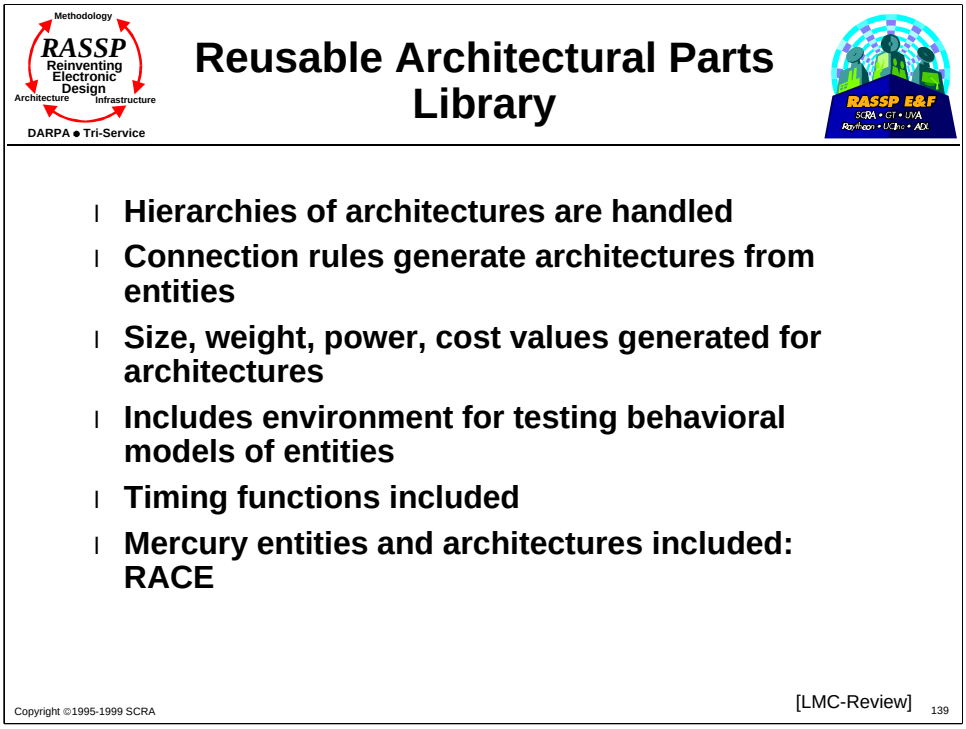
NetSyn allows performance trade-offs to be done in a more automated/user friendly fashion. The input to this tool will be the PGM data flow graph and PGSE output. Rapid performance trade-offs are made within the environment by choosing candidate architectures and simulating them using performance models from Honeywell. Outputs include reports, improved architectural candidates, SW mappings, and improved flow graphs.



The NetSyn tool contains three reusable parts libraries to aid in the selection and verification of an architecture. They consist of 1) a Reusable Software System (RSS) which contains functional graph primitives to help in the building of applications, 2) a reusable architectural parts library which contains architectural classes, components, and configurations, capable of being simulated at the performance level in VHDL and 3) a timing library which contains timing information for the execution of the specific primitives on various processors.



The Reusable Software Subsystem (RSS) captures functional graph primitives in the form of C, Ada, Microcode etc.. It also captures test datasets, analysis reports to aid application developers, reusable graph instantiation parameter lists, reusable graph environments, and primitive tests. The graphical editor for generating PGM graphs (GRED) can access the primitives for building new graphs.

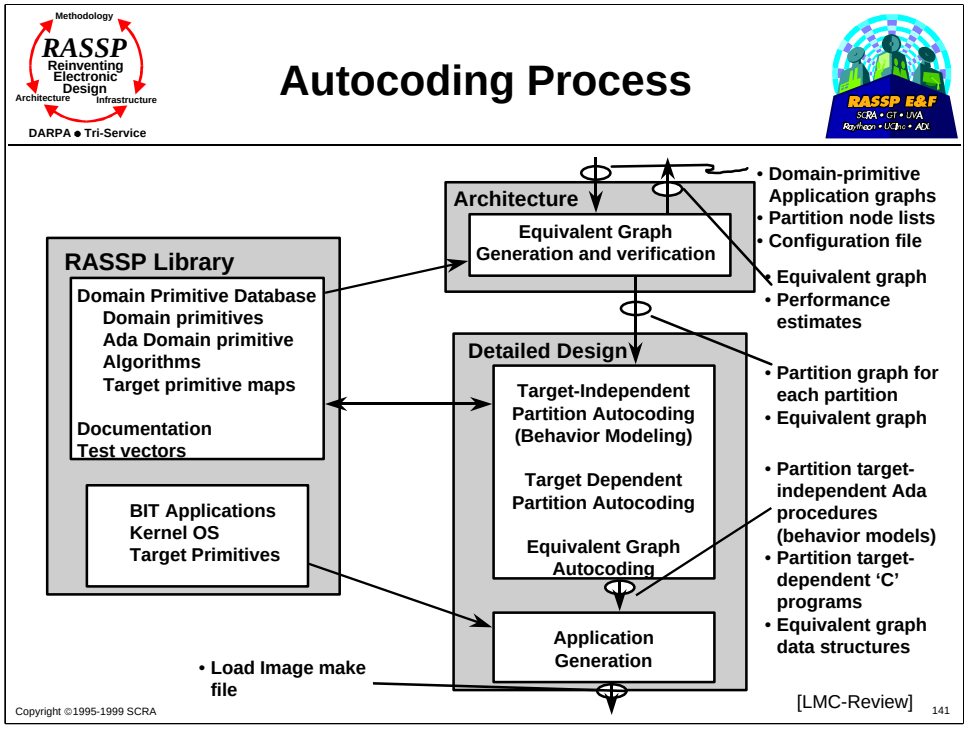


The reusable architectures are hierarchically composed. Connection rules are used to rapidly generate architectures from entities. The performance models use size, weight, power, and cost values so the tool can quickly generate estimates for the entire system. The environment includes capabilities for testing the behavioral models of entities. Timing is included in the library for each of the parts. Currently, the complete RACE architecture from Mercury is part of this library.

Tools for Architecture Selection and Verification (Cont.)

- **Autocode: (Code generation and run-time control)**
  - **Driven from PGM data flow graph, architecture, and mapping**
  - **Library primitives**
    - **Target independent PGSE executables**
    - **Target primitive maps (TPMs) to specific target processors**
  - **Outputs include:**
    - **Modified graph**
    - **Timing estimates for graph**
    - **Target code compatible with run time system**

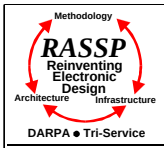
Autocode is a tool that takes as input, PGM data flow graphs, and generates a modified graph with updated timing estimates for the code which has been mapped to a target processor. The target code is compatible with the run-time system. The next slide shows more detail of the autocode generation process.



The autocoding process is shown above. The main elements of the process include:

- Equivalent graph generation
- Partition target-independent autocoding
- Equivalent graph autocoding
- Partition target-dependent autocoding
- Load image specification

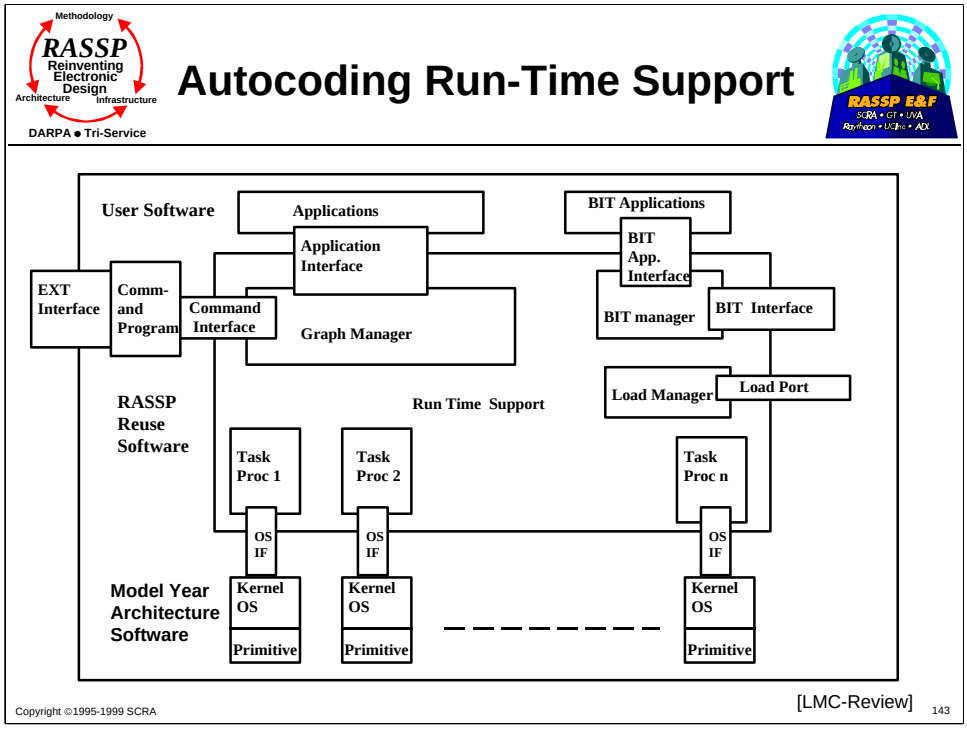
The inputs to the equivalent graph generation process are domain-primitive graphs, configuration files, and partition lists. Equivalent graph generation generates standalone PGM graphs for each partition. Partition autocoding generates Ada procedures implementing each partition (behavior model) and 'C' programs implementing each software partition using target math libraries. Equivalent graph autocoding creates run-time data structures implementing the equivalent graphs. Load image specification generates "make" files specifying complete run-time system.



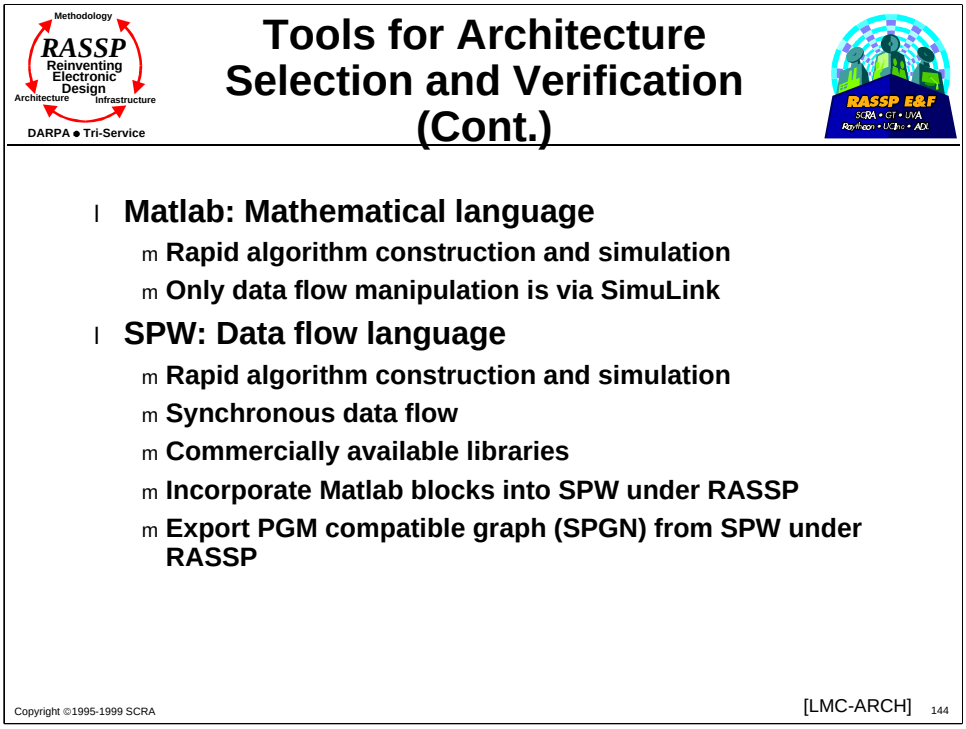
Autocoding Process Inputs and Outputs

- **Inputs to the autocoding process**
  - **PGM application graph SPGN file**
  - **Candidate architecture configuration file**
  - **Lists of nodes in processor software partitions**
  - **Lists of nodes in hardware partitions**
- **Autocoding outputs**
  - **Partition 'C' programs for target processors**
  - **Partition and application performance estimates generated**
  - **Run-time system load image for candidate architecture**
  - **Behavior models for hardware or software partition**

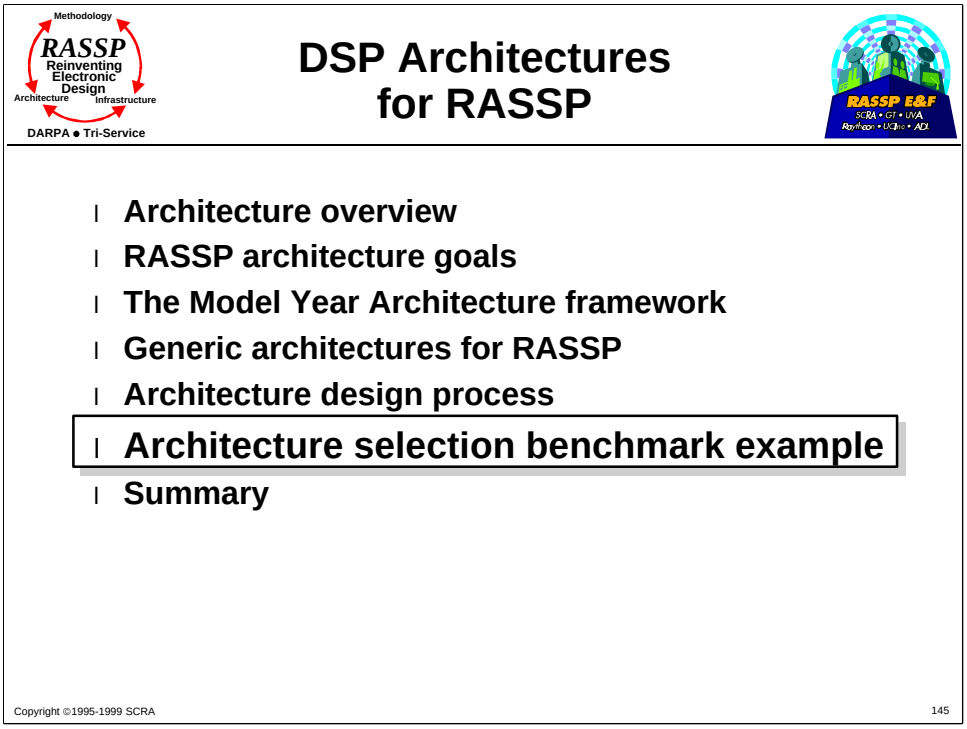
This slide lists the inputs and outputs of the autocoding process.



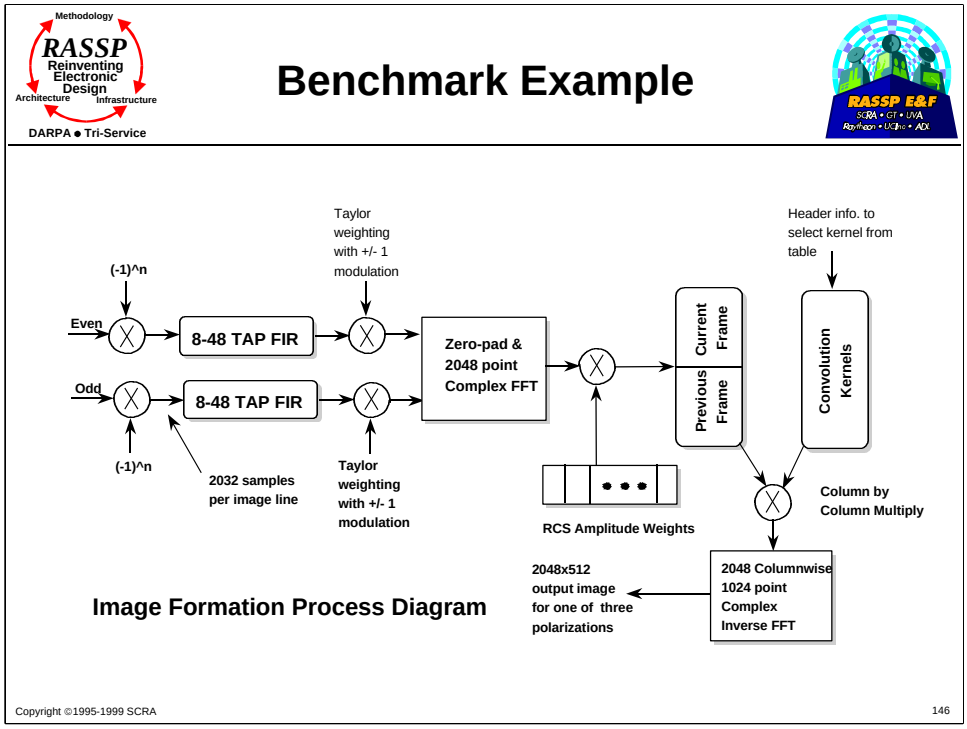
Run-time support is partitioned into user, RASSP user, and Model Year Architecture parts. There are driver-level interfaces between the reuse and model year partitions. Application interfaces are isolated from the target OS. Ports to the external world include the load port, BIT interface, and the command interface. The load manager, graph manager, and BIT manager are run-time managers that execute run-time service routines. Applications are instanced as equivalent node tasks and multiple instances, priorities, and preemption are possible.



Matlab and Signal Processing Workstation (SPW) are used at the high levels of algorithm development and simulation. Current plans are to incorporate PGM code generation into the SPW environment to have a tighter link to the previous set of tools described. This will help close the gap between application algorithm development and the architecture selection tools.



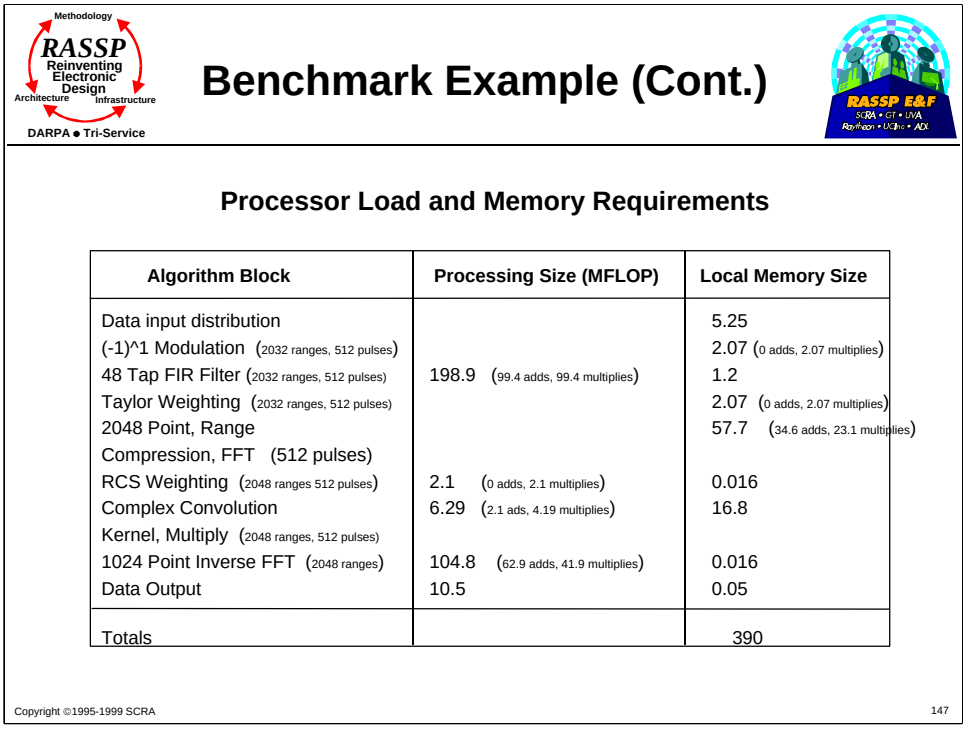
The next section will mention the benchmark example chosen for testing the improvements of the RASSP process. The main focus is on the architecture selection phase of its development.



This is a detailed flow chart of the computation involved in the RASSP Benchmark 1 SAR processing algorithm.

The input was complex data representing the terrain image information. It was first processing by a finite impulse filter and Taylor-weighted before computing a 2048-point complex FFT.

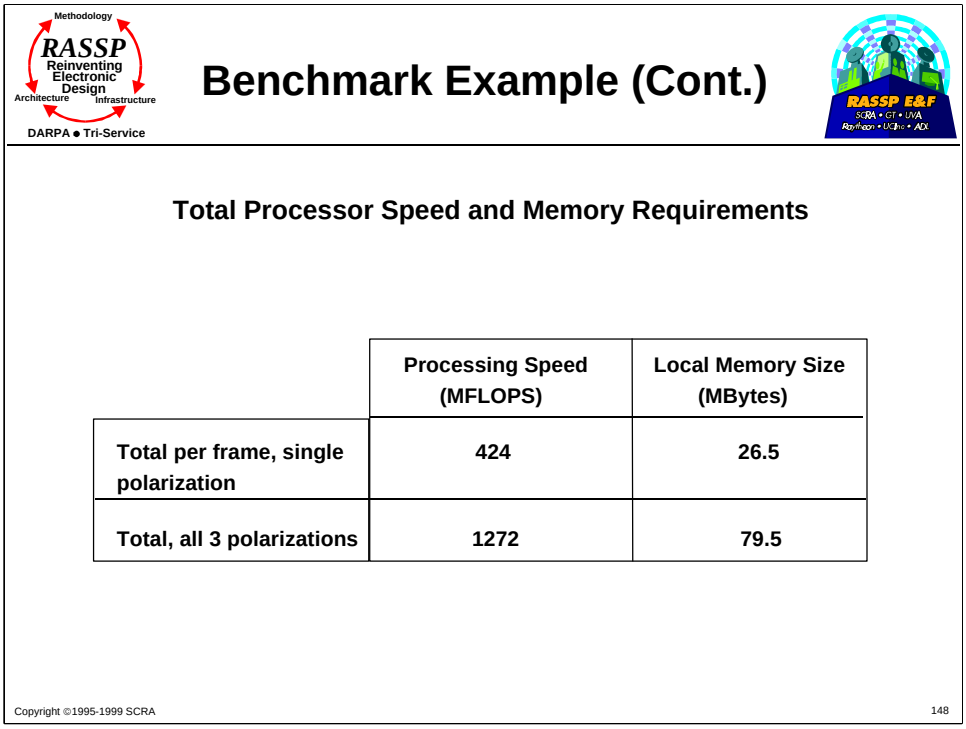
This represents the processing for one of three polarizations.



This chart shows the preliminary analysis of the computational complexity, including the storage requirements for the processor's implementation of the SAR algorithm.

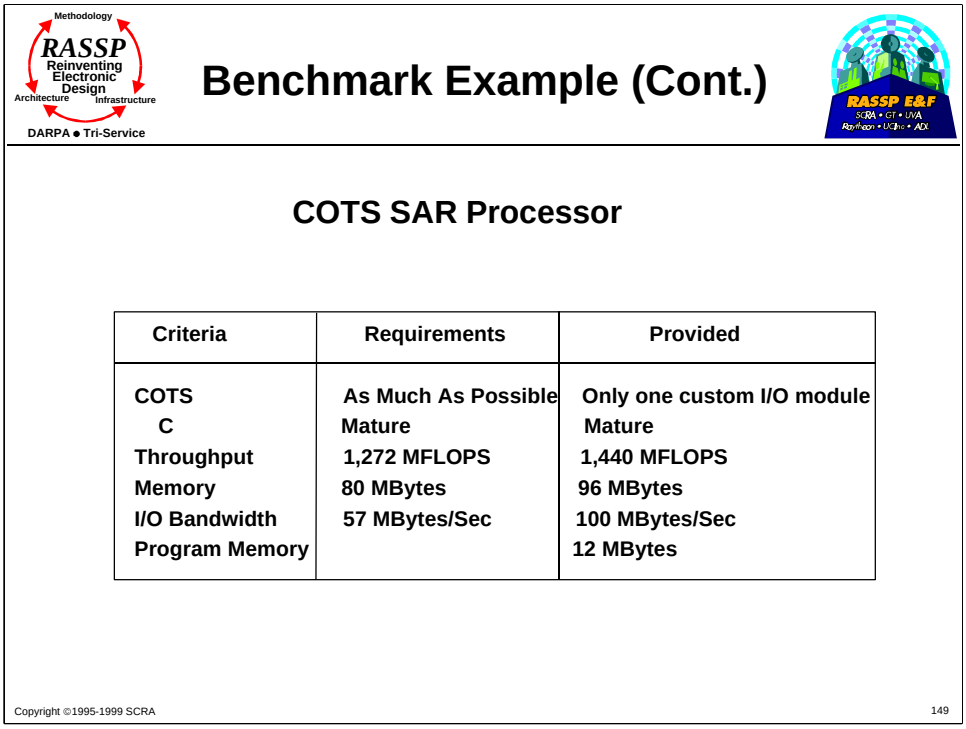
The final estimate requires approximately 400 MFLOPS of processing and 26.5 MBytes of memory.

These numbers are representative for each polarization. There are three polarizations, and the next slide takes that information into account.



Given the three polarizations, the total computational requirements are approximately 1272 MFLOPS.

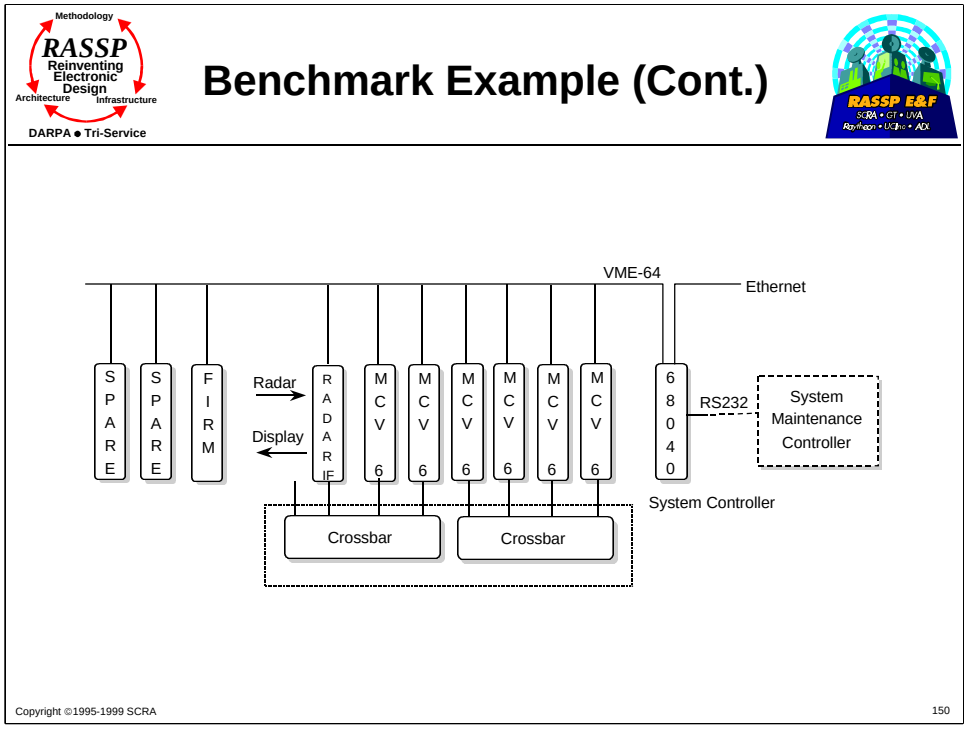
The memory amount needed is on the order of 80 MBytes.



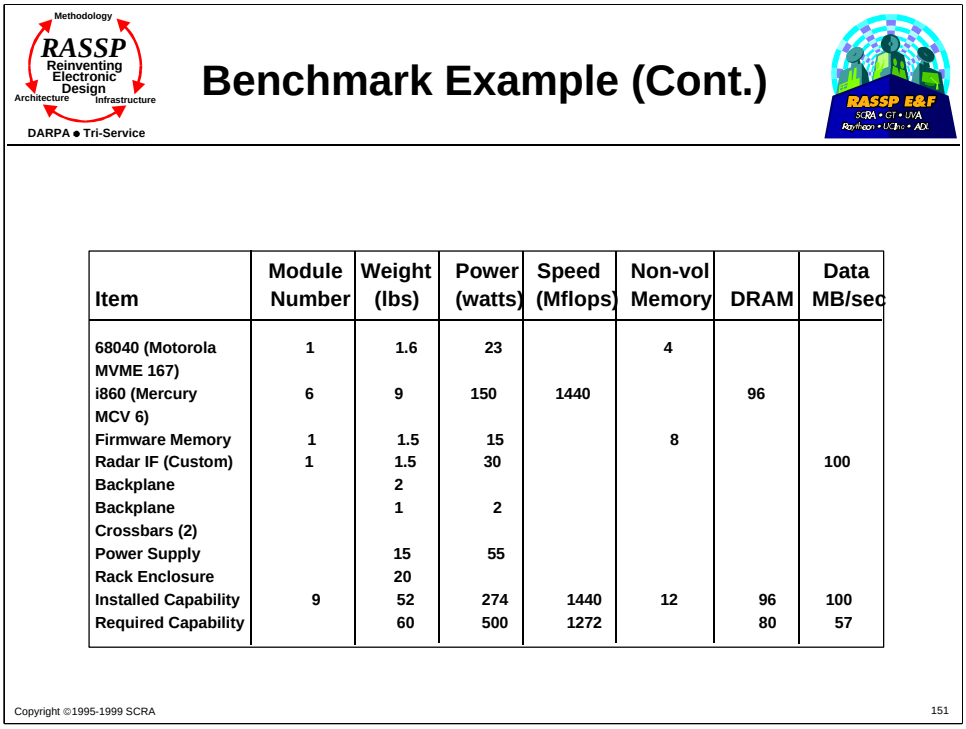
This slides shows the preliminary tradeoffs for architectural design.

It was decided that a COTS solution was the preferred choice. Only one custom I/O module was included for increasing the throughput.

The final solution provided 1440 MFLOPS of processing capability, leaving about 200 MFLOPS of overhead. The memory was set at 96 MBytes for data storage and covered the required 80 MBytes. The I/O BW was selected to be 100 MBytes/sec, which more than met the 57 MBytes/sec requirement. The program storage allotted was 12 MBytes.

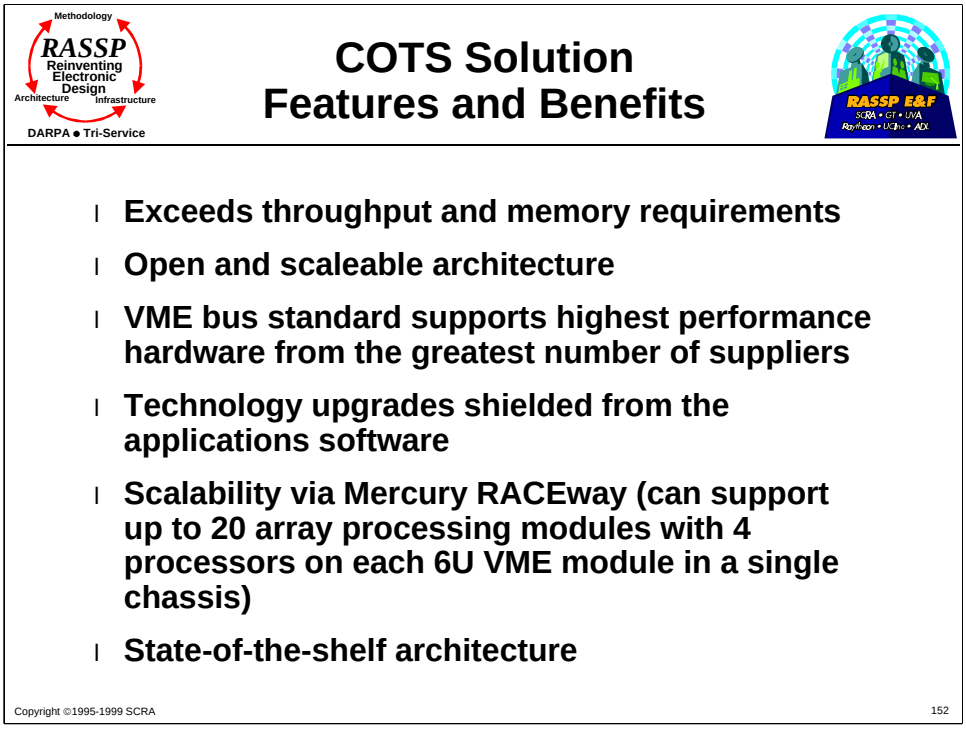


This shows the first architecture selected. It is an all-COTS solution because of the inclusion of the MCV6 boards from Mercury Computing Systems. The network was a collection of crossbar switches to meet the throughput data requirements. The control information was passed over the VME-64 bus from a 68040 controller card.



Item	Module Number	Weight (lbs)	Power (watts)	Speed (Mflops)	Non-vol Memory	DRAM	Data MB/sec
68040 (Motorola MVME 167)	1	1.6	23		4		
i860 (Mercury MCV 6)	6	9	150	1440		96	
Firmware Memory	1	1.5	15		8		
Radar IF (Custom)	1	1.5	30				100
Backplane		2					
Backplane Crossbars (2)		1	2				
Power Supply		15	55				
Rack Enclosure		20					
Installed Capability	9	52	274	1440	12	96	100
Required Capability		60	500	1272		80	57

This shows a breakdown of the individual modules for the COTS solution proposed for the design of the SAR processor vs the various metrics of importance used in the architecture selection process.

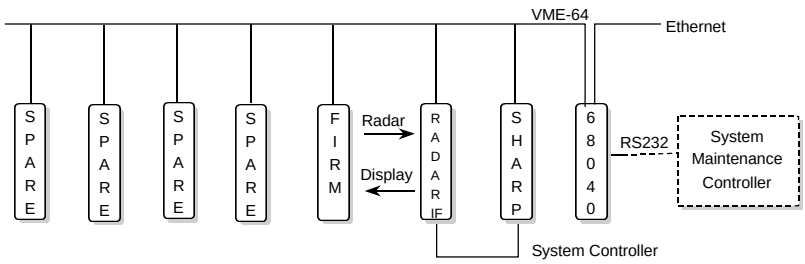


This slide shows some the features of the COTS solution to the RASSP Benchmark 1 study.

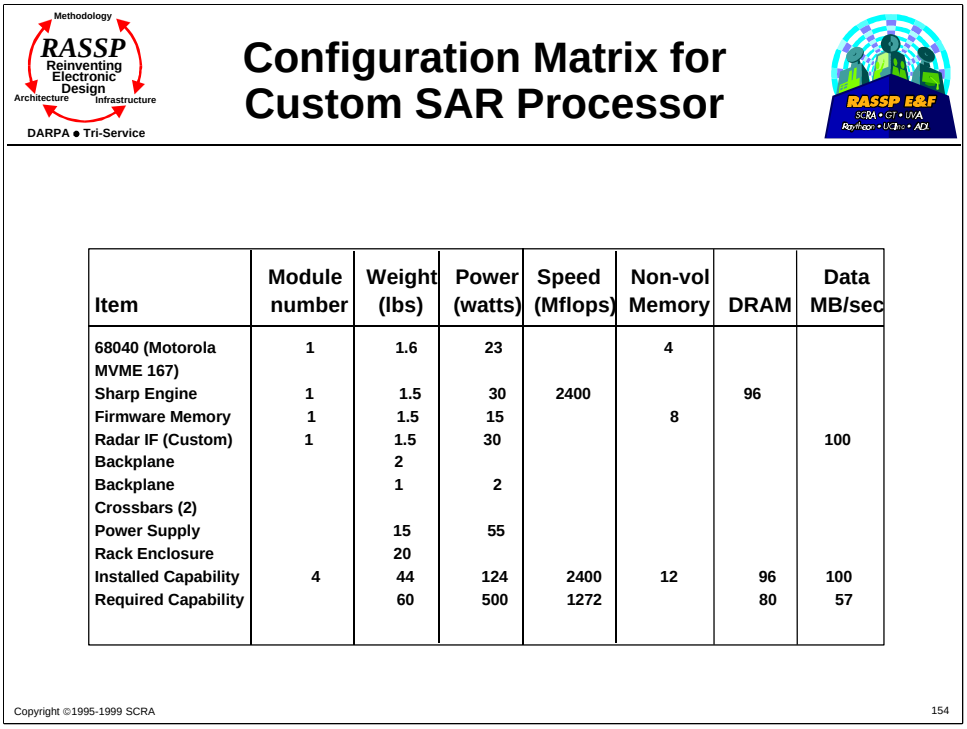
The architecture met or exceeded all its specifications in throughput and memory. It was open and scaleable by using the MCV6 cards. The VME standard bus was used because it is supported by the greatest number of suppliers and with high performance.

The design represented state-of-the-shelf hardware.

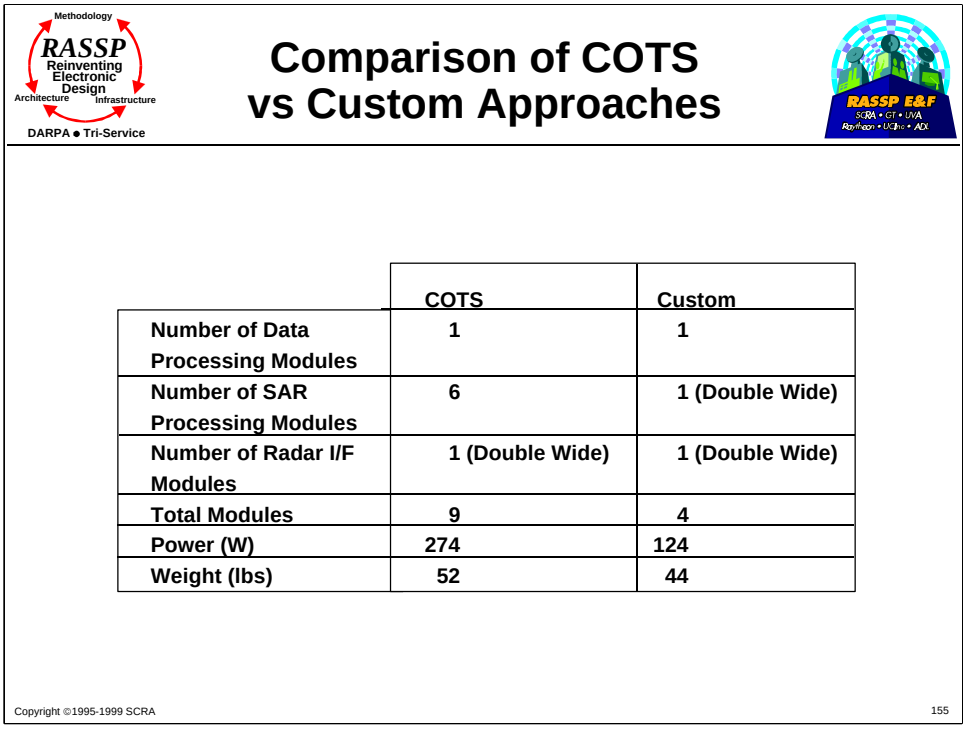
Custom SAR Processor Architecture Overview



This slide depicts the custom architecture selected for the SAR processor. A custom board containing the Sharp FFT chip was used to do the complex FFT computation, and control was implemented using the 68040 card.

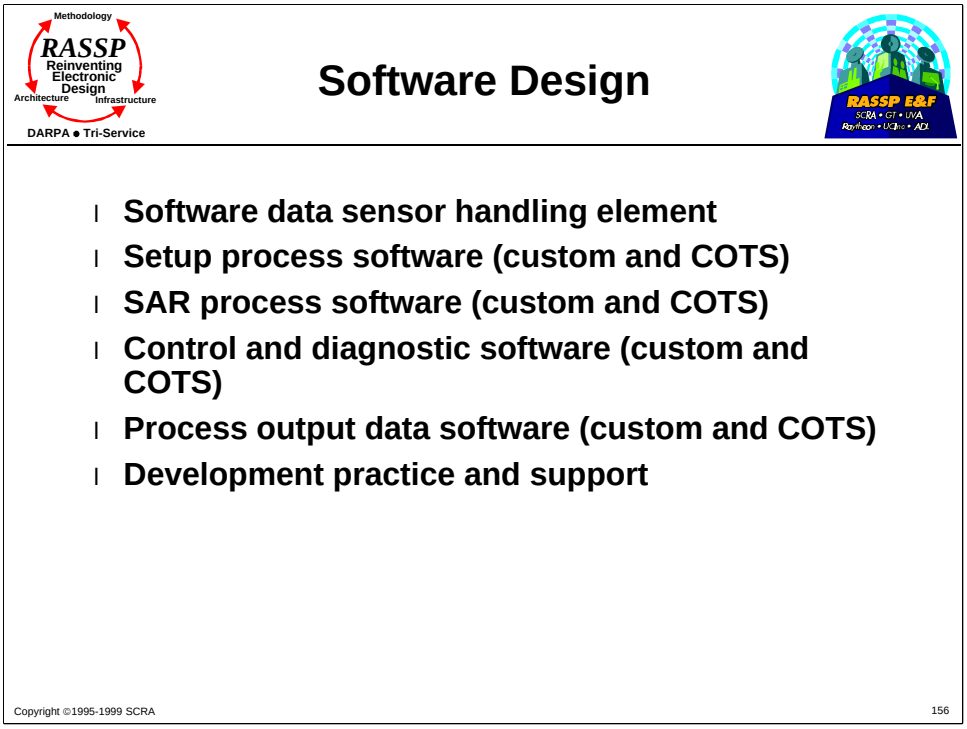


This shows the hardware breakdown of the custom design associated with the design of the SAR processor. It utilized the 68040 controller, the Sharp FFT engine, 2 crossbars, and some custom IF hardware and backplane design.

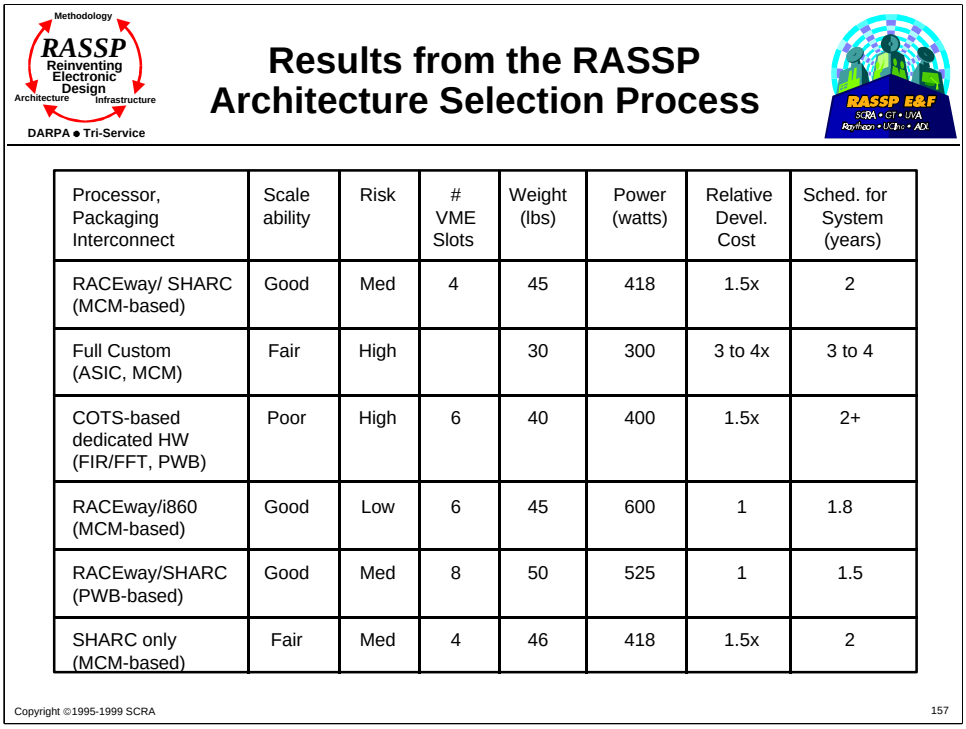


This shows a comparison of the two architectures selected for design of the SAR processing algorithm.

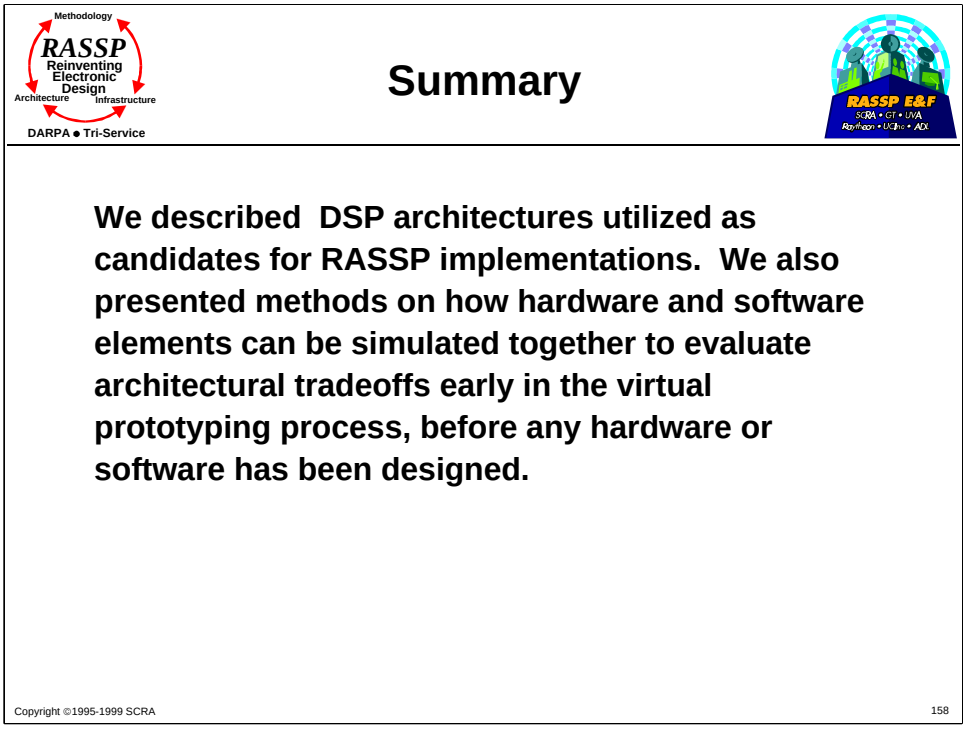
This is a form factor comparison of COTS vs Custom.



This shows some of the software elements required in the development of the SAR processing algorithm.



This shows a spreadsheet of data from the architecture selection process of Martin Marietta.



This concludes the slide presentation. The above topics were covered and architectures were presented as well as methods for choosing these architectures. Some examples of how the process is carried out were also shown.

References

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

[Lee94] Lee J.C.Y., Siewiorek D.P., "*Multiprocessor Architecture Trade-off Advisor Concepts",* Martin Marietta RASSP Team Member, August 1994.

[Lockheed95] Shirley F., et. al., "*RASSP Architecture Guide",* Revision B, Jan 5, 1995.

[LMC-ATL]Caracciolo G., Pridmore J., "*Architectures for Rapid Prototyping of Embedded Signal Processors"* , The RASSP Digest, Vol. 2, No. 1, 1st Quarter 1995.

[LMC-MYA]Caracciolo G., *" Second RASSP Model Year Architecture Working Group Meeting"* , Martin Marietta Laboratories, March 15, 1995.

[LMC-ARCH]Shamming B., *"RASSP Methodology Working Group Meeting Architecture Process"* , Martin Marietta Laboratories, March 16, 1995.

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

[Honeywell] Carpenter T., Rose F., Steeves T., *"Performance Modeling with VHDL",* Honeywell Systems & Research Center, Slide presentation. Used with permission.

[Hein95] Hein C., Nasoff D., "*VHDL-Based Performance Modeling and Virtual Prototyping",* Second Annual RASSP Conference, July 24-28.

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

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

[LMC-Review] *"RASSP Semi-Annual Government Review"* , Martin Marietta Laboratories, February 13-14, 1995.

[Madisetti1995] Madisetti, V. "VLSI Digital Signal Processors," IEEE Press, 1995.