Methodology Appnote

Abstract

The goal of the Rapid Prototyping of Application-Specific Signal Processors (RASSP) program is to improve by (at least) a factor of four the time required to conceptualize, design/upgrade, and field signal processor systems. Similar improvements in design quality and life-cycle cost were also expected. The overall RASSP methodology is based upon two major beliefs:

- that concurrent design practices, using an integrated approach to hierarchical design verification, are required to improve design quality and performance
- that dramatic (>4X) decreases in cycle time can only be achieved by maximizing reuse of both hardware and software elements.

These two beliefs lead to the notion of design using modular hardware and software functions that are represented in a reuse library. To significantly improve the cycle time and cost of developing signal processing systems, major changes are required in the traditional process. RASSP innovations in the areas of processor architecture selection and verification using Virtual Prototyping and software development using graphical design methods and autocoding have demonstrated the largest contributions to the 4X goals.

In addition to speeding up the individual processes, RASSP reduced a portion of the cycle time by eliminating time-consuming and costly design rework cycles. The RASSP process provides a method to support high-quality designs that lead to first-pass success of all hardware and software elements.

Purpose

This application note serves as the gateway to the other application notes that detail the RASSP developed technologies as well as the case studies that document the actual use of the methodolgy and technologies to specific projects. The following provides a generic description of the process to follow in order to achieve the RASSP benefits. It does not get into the use of the particular tools nor does it provide examples for how to perform designs. These are provided in the Case Studies and Application Notes. By adopting and following the processes described in this document a program will be able to realize the same productivity improvements that Lockheed Martin Advanced Technology Laboratories have demonstrated on the RASSP program.

Roadmap

- 7.1 Functional Design
  - 7.1.1Architecture Sizing
  - 7.1.2 Selection Criteria Definition
  - 7.1.3 Define Non-DFG/CFG Software Tasks
  - 7.1.4 Flow Graph Generation
  - 7.1.5 Command Program Development
  - 7.1.6 Functional Simulation
- 7.2 Architecture Selection
  - 7.2.1 Architecture Definition
  - 7.2.2 Architecture Model Synthesis
  - 7.2.3 Performance Simulation
  - 7.2.4 Implementation Analysis
  - 7.2.5 Trade-off Analysis
- 7.3 Architecture Verification
  - 7.3.1 Autocode Generation
  - 7.3.2 Performance Simulation
  - 7.3.3 Refine Physical Decomposititon
  - 7.3.4 Refine Implementation Analysis
  - 7.3.5 Model Availability
  - 7.3.6 Verification Approach Definition
  - 7.3.7 Simulation Development
  - 7.3.8 Simulation
  - 7.3.9 Trade-off Analysis Update
- 7.4 Software in Architecture Design Process
  - 7.4.1Domain Primitive
  - 7.4.2 Domain-Primitive Graph
  - 7.4.3 Allocated Graph
  - 7.4.4 Partitioned Software Graph
  - 7.4.5 Partition Graph
  - 7.4.6 Equivalent Application Graph
  - 7.4.7 Command Program
  - 7.4.8 DFG/Command Program Functional Simulation
  - 7.4.9 Non-DFG Software
- 7.5 Other considerations in the Architecture Design Process
  - 7.5.1Use of VHDL in Architecture Process
  - 7.5.2 Design-for-Test Tasks in Architecture Design
  - 7.5.3 Role of PDT in Architecture Process
  - 7.5.4 Design Reviews in Architecture Process

8.0 Detailed Design Process Detailed Discussion

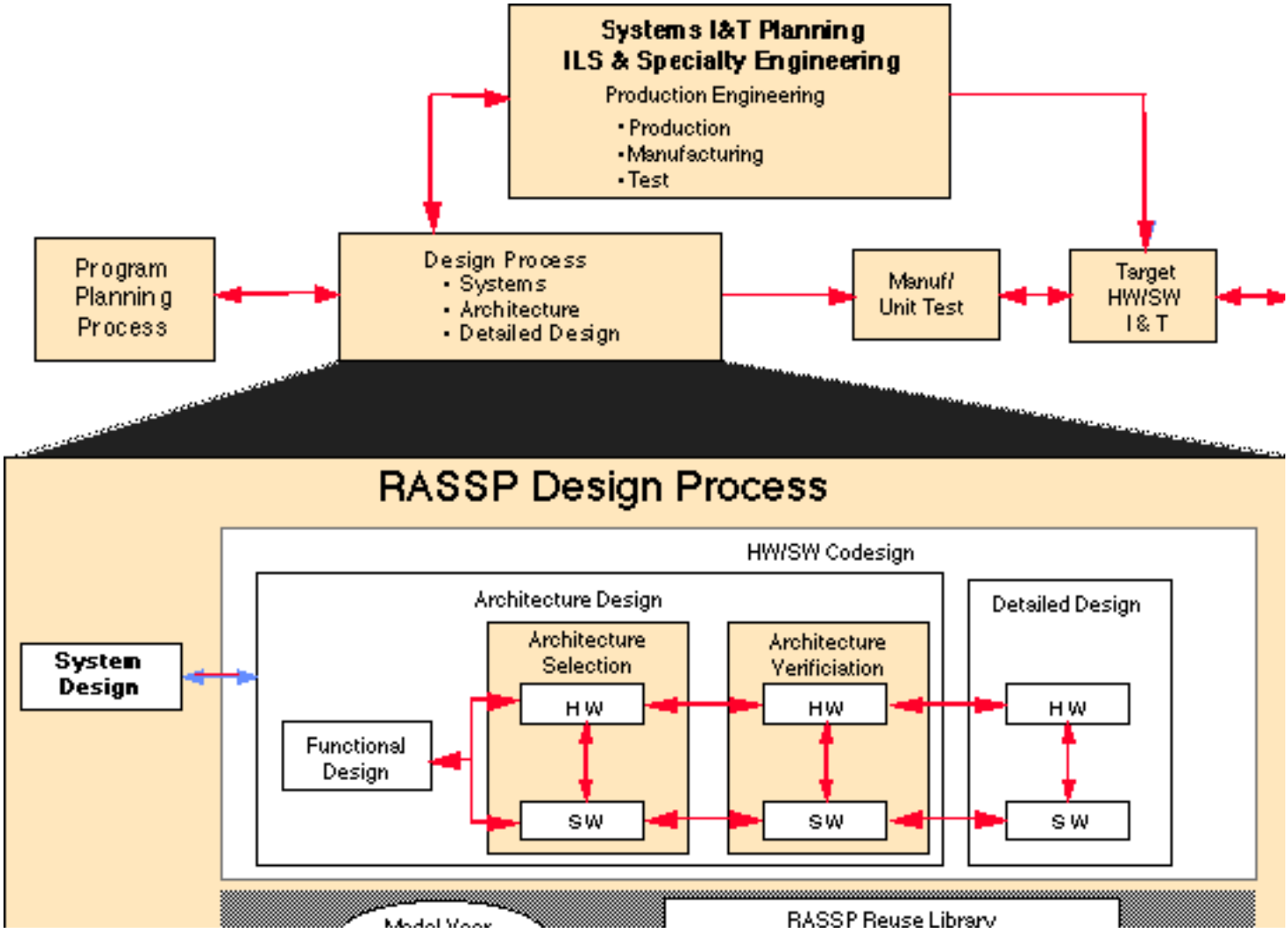
- 8.1 Module/MCM Design Process
  - 8.1.1 Module Preliminary Design
    - 8.1.1.1 Develop Module Behavioral Model
    - 8.1.1.2 Generate Module Test Plan
    - 8.1.1.3 Perform Behavioral Funtional Simulation
    - 8.1.1.4 Generate Module Functional Test Vectors
    - 8.1.1.5 Generate ASIC/FPGA Requirements
    - 8.1.1.6 Search Design Reuse
    - 8.1.1.7 Interactive Logic Design
    - 8.1.1.8 Preliminary Layout (Parts Placement)
    - 8.1.1.9
    - 8.1.1.10 Perform Power/Loading Analysis
    - 8.1.1.11 Perform Functional Timing Simulation
    - 8.1.1.12 Select Parts

RASSP Methodology Application Note

1.0 Executive Summary

1.1 RASSP Methodology Design Process

The RASSP design process has three major functional processes - the systems, architecture, and detailed design processes. It is shown in expanded form in Figure 1 - 1. The process is partitioned as a function of the abstraction level of the evolving design, not as a function of the discipline. This is the result of merging hardware and software into a true codesign process; any distinctions between hardware and software are made within the specific process. Hardware/software codesign is implemented from the initial partitioning of functions to hardware and software elements all the way to manufacturing release. At each step in the hierarchy, interactive simulation using hardware and software models is performed at equivalent levels of abstraction to verify both functionality and performance. This means that each process area is closely tied to the RASSP vision of an iterative (spiral-like) development, resulting in a series of virtual prototypes and data packages.



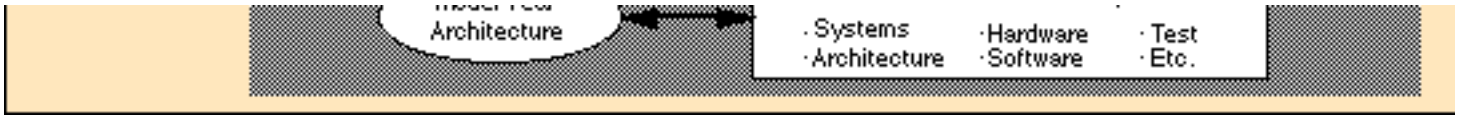


Figure 1 - 1: RASSP Design Process

Each major cycle of the spiral process represents an iteration of a virtual prototype. Within each prototype iteration, pieces of the design can and most likely will be at different levels of maturity, as shown in Figure 4 - 1. Each piece of the design may be represented by a mini-spiral where the spiral cycles correspond to virtual prototypes of the piece. Consequently, for each major spiral cycle, there may be activity in the system, architecture, and detailed design processes.

The overall RASSP development process, shown in Figure 1 - 1, has four major elements:

- Project planning/Management - Project planning/management provides the first step of the process: Develop the technical plan, and form the PDT (product development team) and the overall management approach for the specific signal processor development.

- Design Process - The design process represents development of the signal processor design from requirements capture through release to manufacturing. Is has three major processes: Systems design, architecture design, and detailed design. This process takes customer requirements and results in a fully verified (functionality and performance) virtual prototype of the signal processor.

- Systems Design Process - This process captures customer requirements and converts these system-level needs into processing requirements (functional and performance). Functional and performance analyses are performed to properly decompose the system-level description. The system process has only a very preliminary notion of either hardware versus software functionality or processor implementation.

- Architecture Design Process - The architecture process transforms processing requirements into a candidate architecture of hardware and software elements. Architecture selection initiates the trade-offs between the different processor architecture alternatives. During this process, the system-level processing requirements are allocated to hardware and/or software functions. The hardware and software functions are verified with each other via 'co-verification" at all steps. The architecture verification process results in a detailed behavioral description of the processor hardware and the definition of the software required for each processor in the system. The intent is to verify all the code during this portion of the design, ensuring hardware/software interoperability early in the design process.

- Detailed Design - As with the prior processes, the design is completed and verified for both hardware and software via a set of detailed functional and performance simulations. When this process is complete, the design is established, resulting in a fully verified virtual prototype of the system.

  During the hardware portion of the detailed design process, behavioral specifications of the processor are transformed into detailed designs (RTL, and/or logic-level) through a combination of hardware partitioning, parts selection, and synthesis. Detailed designs are functionally verified using integrated simulators, and performance/timing is also verified to ensure proper performance. The process results in detailed hardware layouts and artwork, net lists, and test vectors that can then be seamlessly transitioned to manufacturing and test via format conversion of the data.

  Since most of the software developments are verified during the architecture design phase, they are limited at this point to generation of those elements that are target-specific. This includes configuration files, bootstrap and download code, target-specific test codes, etc. All the software is compiled and verified (to the extent possible) on the final virtual prototype prior to the detailed design review. Design release to manufacturing marks the end of the RASSP design process.

- Prototype Manufacturing, Integration and Test - The manufacturing, integration and test processes define for RASSP the interface of design engineering to the disciplines required to build and test signal processor prototypes. This means the interaction of manufacturing and test personnel throughout the design cycle to support concurrent engineering through the release of the virtual prototype to manufacturing. This process also supports subsequent testing of the manufactured design and hardware/software integration of the signal processor.

- Specialty Engineering ('illities, etc.) - Specialty engineering covers all the non-design engineering tasks required to successfully develop and field RASSP products. These include reliability, maintainability, parts, and EMI engineering (where applicable), as well as integrated logistic support engineering and services.

 Dennis Basara

RASSP Methodology Application Note

2.0 Introduction

2.1 Methodology Application Note Objective

The objective of this application note is to provide an understanding of the design methodology developed for the LM/ATL RASSP program. It is not the intent to lead you, the reader, on a step-by-step discussion of this methodology as applied to any one particular application. Rather, it will use the workflows generated in IDEF3X format to lead you through a generic discussion of the RASSP methodology. One must always keep in mind that RASSP is an iterative and concurrent development process. Feedback or 'Failback' paths are identified within the workflows. However, because every program/project is unique, the failback paths may be different for each effort and the number of iterations of each design cycle will also vary.

2.2 Organization of the Application Note

Sections 5, 6, 7, and 8 are organized around the IDEF workflows developed for the Systems, Architecture, and Detailed Design processes. Clicking within the major boxes will enable you to explore the hierarchical nature of each of the major process levels. Defining the initial Systems, Architecture, and Detailed Design diagram as the First level, each of the major sections can be navigated to the following levels.

- System Design - 3 levels
- Architecture Design - 3 levels
- Detailed Design - 4 levels

The rest of this application note contains the detailed descriptions of the methodology, the conclusions, and the references for additional detail. Section 3.0 is a high level overview of IDEF (ICAM (Integrated Computer-Aided Manufacturing) Definition). This reference is useful in order to better understand the constructs of IDEF and thus be able to review the process drawings. This allows the reader to get a quick understanding without having to read the detailed explanations.

Section 4.0 contains a description of some of the overall processes that are found throughout the RASSP methodology. They unify the design tasks that are described in subsequent sections. This section starts with a description of the RASSP methodology as a rixk driven iterative hierarchical virtual protoytping, spiral development process. The unifying process of Model Year Architecture, Hardware/Software Codesign, Design-for-Testability, and Reuse Library Management are then presented as a high-level introduction.

Section 5.0 then presents a top level summary of the system, architecture, and detailed design processes. This section provides the first links into the IDEF process flows that can lead the reader graphically through the methodology. These process flow are displayed in a hierarchical fashion and also contain links to the textual descriptions of the process found in Sections 6.0, 7.0, 8.0, and 9.0.

Section 6.0 provides the detailed discussion of the System Design process. It takes you through requirements analysis, functional analysis, system partitioning, and completes by discussing other items that need to be considered during design, i.e., DFT, IPT, design reviews, and use of VHDL in modeling.

Section 7.0 provides the detailed discussion of the Architecture Design process. It takes you through functional design, architecture selection, architecture verification, and software in the architecture design process. It completes with a discussion of other items that need to be considered during design, i.e., DFT,

IPT, design reviews, and use of VHDL in modeling.

Section 8.0 provides the detailed discussion of the Detailed Design process. It takes you through module/MCM design, ASIC design, FPGA design, backplane and chassis design, and subsystem integration and test. It completes with a discussion of other items that need to be considered during design, i.e., DFT and use of VHDL in modeling.

Section 9.0 provides an integrated view of the software development process within the RASSP methodology. This has all been discussed previously in the System, Architecture, and Detailed Design sections since the methodology if a Hardware/Software Codesign one. It is provided here in one place so that those interested in only the software development may have an easier reference.

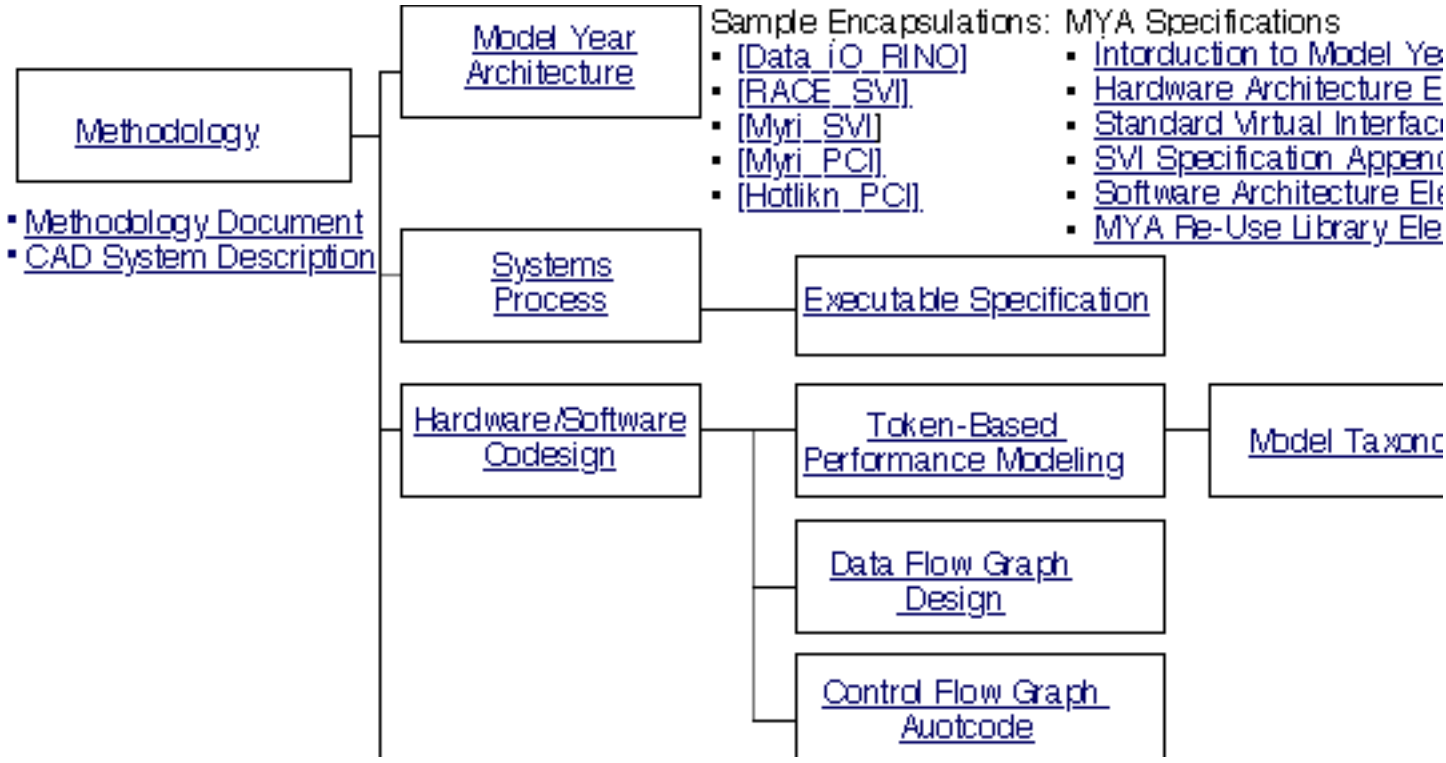
Section 10.0 provides a discussion of the implementation of a reuse management system that will assist in the design ÔforÕ and ÔwithÕ reuse. This is a key ingredient in reducing cost and cycle time for your designs.

Section 11.0 provides the reference information and the links to those references. It also provides links to all of the other application notes and case studies.

2.3 Linkage with other Application Notes

This methodology application note serves as a gateway to all of the other Application Notes and Case Studies created by Lockheed Martin ATL. The case studies (SAR, ETC4ALFS on COTS Processors, and SAIP) will lead you through how this methodology was actually applied and used in the performance of these Benchmark programs. The focus for these case studies is more on what was done and the benefit gained. Not all of the RASSP methodology and process steps were applied to each of the case studies. The Methodology was tailored to meet the goals and objectives of each development project.

The Application Notes serve to provide specific details of the technologies that were developed and applied within RASSP. Detailed examples were taken from the benchmark programs in order to illustrate not only the how and why it was accomplished but also show the benefits to be gained by applying the RASSP technologies. Figure 2-1 illustrates how each of the application notes are tied to this document. This figure, as well as other links in this document will allow access to the rest of the application notes.



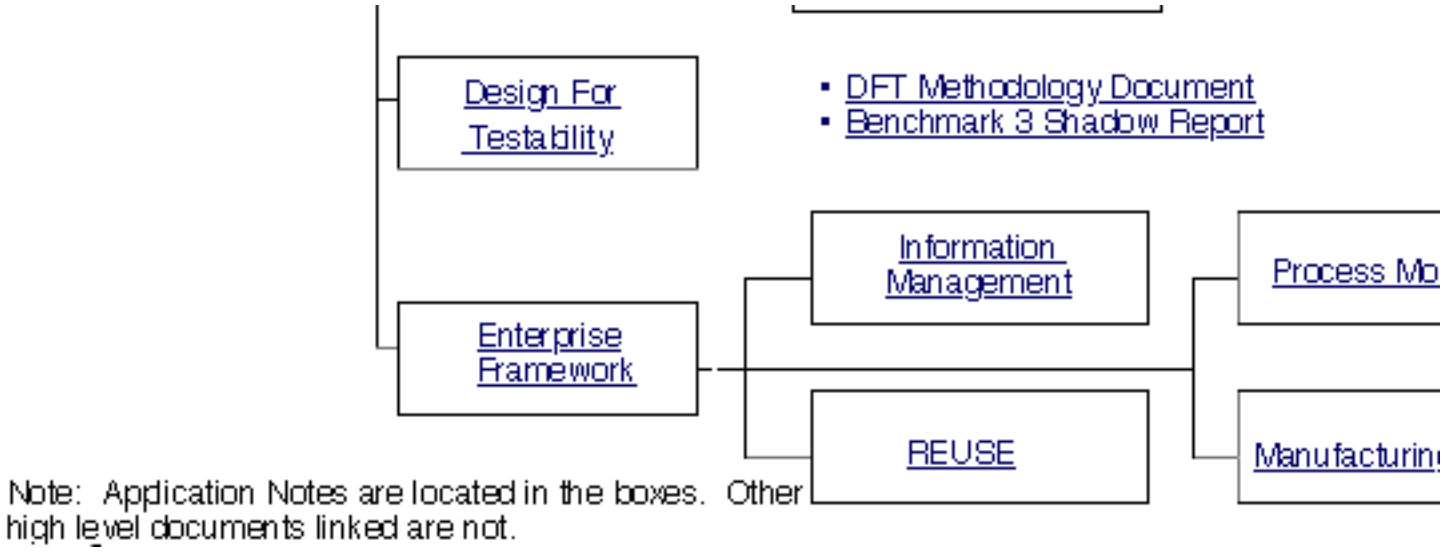


Figure 2 - 1: RASSP Design process.

Each major cycle of the spiral process represents an iteration of a virtual prototype. Within each prototype iteration, pieces of the design can and most likely will be at different levels of maturity, as shown in Figure 4 - 1. Each piece of the design may be represented by a mini-spiral where the spiral cycles correspond to virtual prototypes of the piece. Consequently, for each major spiral cycle, there may be activity in the system, architecture, and detailed design processes.

The overall RASSP development process, shown in Figure 1 - 1, has four major elements:

- Project planning/Management - Project planning/management provides the first step of the process: Develop the technical plan, and form the PDT (product development team) and the overall management approach for the specific signal processor development.

- Design Process - The design process represents development of the signal processor design from requirements capture through release to manufacturing. Is has three major processes: Systems design, architecture design, and detailed design. This process takes customer requirements and results in a fully verified (functionality and performance) virtual prototype of the signal processor.

    - Systems Design Process - This process captures customer requirements and converts these system-level needs into processing requirements (functional and performance). Functional and performance analyses are performed to properly decompose the system-level description. The system process has only a very preliminary notion of either hardware versus software functionality or processor implementation.

    - Architecture Design Process - The architecture process transforms processing requirements into a candidate architecture of hardware and software elements. Architecture selection initiates the trade-offs between the different processor architecture alternatives. During this process, the system-level processing requirements are allocated to hardware and/or software functions. The hardware and software functions are verified with each other via 'co-verification" at all steps. The architecture verification process results in a detailed behavioral description of the processor hardware and the definition of the software required for each processor in the system. The intent is to verify all the code during this portion of the design, ensuring hardware/software interoperability early in the design process.

    - Detailed Design - As with the prior processes, the design is completed and verified for both hardware and software via a set of detailed functional and performance simulations. When this process is complete, the design is established, resulting in a fully verified virtual

prototype of the system.

During the hardware portion of the detailed design process, behavioral specifications of the processor are transformed into detailed designs (RTL, and/or logic-level) through a combination of hardware partitioning, parts selection, and synthesis. Detailed designs are functionally verified using integrated simulators, and performance/timing is also verified to ensure proper performance. The process results in detailed hardware layouts and artwork, net lists, and test vectors that can then be seamlessly transitioned to manufacturing and test via format conversion of the data.

Since most of the software developments are verified during the architecture design phase, they are limited at this point to generation of those elements that are target-specific. This includes configuration files, bootstrap and download code, target-specific test codes, etc. All the software is compiled and verified (to the extent possible) on the final virtual prototype prior to the detailed design review. Design release to manufacturing marks the end of the RASSP design process.

- Prototype Manufacturing, Integration and Test - The manufacturing, integration and test processes define for RASSP the interface of design engineering to the disciplines required to build and test signal processor prototypes. This means the interaction of manufacturing and test personnel throughout the design cycle to support concurrent engineering through the release of the virtual prototype to manufacturing. This process also supports subsequent testing of the manufactured design and hardware/software integration of the signal processor.

- Specialty Engineering ('illities, etc.) - Specialty engineering covers all the non-design engineering tasks required to successfully develop and field RASSP products. These include reliability, maintainability, parts, and EMI engineering (where applicable), as well as integrated logistic support engineering and services.

RASSP Methodology Application Note

3.0 IDEF3 Representation of Process Workflows

3.1 What are Workflows

A workflow consists of a group of process steps to be performed in a particular order to complete a particular design effort. A program plan is created by piecing together the workflow segments which are appropriate to the design task. When instantiated in the enterprise, the workflows can be executed by clicking on the process steps in the order allowed by the precedence relationships and performing the task required by that process step. The concept of operation for the enterprise framework includes the ability for teams of engineers to execute program plans, expressed as workflows. The information manager generates the appropriate objects and work locations to facilitate the workflow. The information models and information manager are therefore closely coupled to the process models and the workflow manager.

The workflows are hierarchical in nature - representing the various disciplines associated with electronic design. The options available to a user organization are either to make use of the workflows in their current form or to develop plans based on a combination of reuse of workflow segments, individual process steps or augmentation of workflow segments with custom user process steps.

3.2 Activity Definitions

The Activity Definitions present short descriptions of what is to be accomplished for each process step, as well as the tools and business item names (placeholders) for the inputs, outputs, required personnel resources and reference data needed to perform each task or process step. In execution of the program plan constructed from the workflows, the activities and business item specifications are instantiated for the particular project. As part of this process, users are assigned to the roles in the project.

3.3 Workflow Capture

Workflows were developed for the entire RASSP design process using the IDEF3X modeling method. IDEF3X was developed by Rockwell International Corporation and is an extension of the IDEF3 process description capture method. The name IDEF originates from the Air Force program for Integrated Computer-Aided Manufacturing (ICAM) from which the first ICAM Definition, or IDEF, methods emerged. IDEF3 was created specifically to capture descriptions of sequences of activities. It can be distinguished from other process modeling methods because it facilitates the capture of the description of what a system actually does.

3.3.1 IDEF3X Overview

IDEF3X is a modeling method which combines the ICOM (input, control, output, mechanism) aspect of IDEF0 with the process flow description of IDEF3, along with some additional features to facilitate implementation by a workflow management tool such as Intergraph's Design Methodology Manager (DMM). The syntactic elements of an IDEF3X model are similar to IDEF3 and include units of behavior (UOBs), junction boxes, and precedence links. Additional features include identifying ICOMs by naming the object and its life cycle state separated by an "*" (e.g., Draft*Publication - where Publication is the name of the object and Draft is its current state), object state links which identify the flow of data between UOBs, feedback links which indicate failback paths, annotating the name of the junction boxes with an "A" or "S" to indicate an asynchronous or synchronous junction (e.g. A&), and annotating the precedence links with a "P:" followed

by a two-letter code which indicates the precedence between the parent and dependent UOB.

Synchronization refers to the relative timing of the process paths that either converge into or diverge out of a junction. A synchronous fan-in junction indicates that all processes connected to the input of the junction must complete simultaneously before the UOB connected to the output of the junction will be activated. If the fan-in is asynchronous, there is no timing constraint imposed on the completion of the input processes.

3.3.2 Precedence Links

A precedence link is defined by two states such as Finish-Start which is the default precedence of links within an IDEF3 process model. The first state indicates the state the parent UOB (where the link starts) must be in before the dependent UOB (where the link ends) can enter the second state. So, Finish-Start means the parent UOB must finish before the dependent UOB can start. IDEF3X identifies additional precedence relationships which are supported by Intergraph's DMM tool. These 8 precedence are:

- Start - Start: The dependent process can begin a soon as the parent process begins, but not before. No completion order is implied.
- Start - Finish: The dependent process cannot complete successfully until the parent process starts. No starting order is implied.
- Finish - Start: The dependent process cannot begin until the parent process completes with the chosen exit status (exit statuses are defined by the implementation)
- Finish - Finish: The dependent process cannot complete successfully until the parent process completes with the chosen exit status. Either process may begin first.
- Concurrent: This is a combination of Start-Start and Finish-Finish. The dependent process cannot begin until the parent process begins, and cannot be completed until the parent process completes with the chosen exit status.
- Cascade: The dependent process cannot begin until the parent process completes with the chosen exit status, and is immediately and automatically invoked when the parent process completes with that status. Cascade connections are used to automatically trigger processes when the design passes a certain point.
- Fail (Reset): The dependent process and any downstream processes are marked as invalidated when the parent process completes with a status that activates a fail connection. It is useful to think of the fail connection as "Reset" -- it resets the workflow to a previous state based on a particular exit status that may or may not be a true "failure" condition.
- Fail Cascade: When a process completes with a status that activates a fail-cascade connection, the dependent process and all downstream processes are marked as never having started. The dependent process connected to the fail-cascade link is immediately launched.

This information will be represented on the workflows by placing a 'P' on the link followed by a ":" and the 2 letter code. For example,

P:SS for Start - Start
P:SF for Start - Finish
P:FS for Finish - Start
P:FF for Finish - Finish
P:CO for Concurrent
P:CA for Cascade
P:FR for Fail (Reset)
P:FC for Fail - Cascade

3.4 Modeling Example

The workflow model captures:

- Process steps
- Precedence relationships between process steps
- Personnel roles authorized/required to perform workflow
- Information objects involved (created, used, modified, destroyed, etc.) in the process step
- Tools to be launched or controlled at each step

The example elaborated in this section is taken from the RASSP Architecture Definition: Functional Design workflow. Figure 3 - 1 contains a portion of the Functional Design workflow represented as an IDEF3X model. The model contains three units of behavior (UOBs) which represent design activities: Architecture Sizing, Selection Criteria Definition and Refine DFT Strategy. Also present in this model are precedence links, object state links, junction boxes, and all of the Inputs, Controls, Outputs and Mechanisms (ICOMs) for each UOB. This model was developed using the TopDown Flowcharter tool by Kaetron Software Corporation.

Although it is not the purpose of this Appendix to present IDEF3X concepts in depth, a quick review is in order. The text attached to the UOBs are called Business Items. These are place holders for actual file system items which contain the design data. There are four different types of Business Items or ICOMs. The inputs and outputs are self-explanatory. The controls are documents that can be referred such as libraries or specifications. The Mechanisms are the tools and personnel resources required to execute the process step. The syntax of an input or output Business Item is:

[Grouped]*State*Business_Item_Name.

The [Grouped] term is optional. If it is present, (as in Grouped*Developed*SD Test Architecture and TSD 2.1.3 in Figure 3 - 1), it indicates that the Business Item actually consists of multiple file system items (files or directories). For example, Grouped*Developed*SD Test Architecture and TSD is made up of the file system entities:

Developed*SD Test Architecture
Developed*SD TSD
Developed*SD Test Plans
Developed*SD Test Procedures
Developed*SD Fault Model

The word "Developed" indicates the state of the Test Architecture and TSD. In this case, the data items have just been created during the process step 2.1.3.

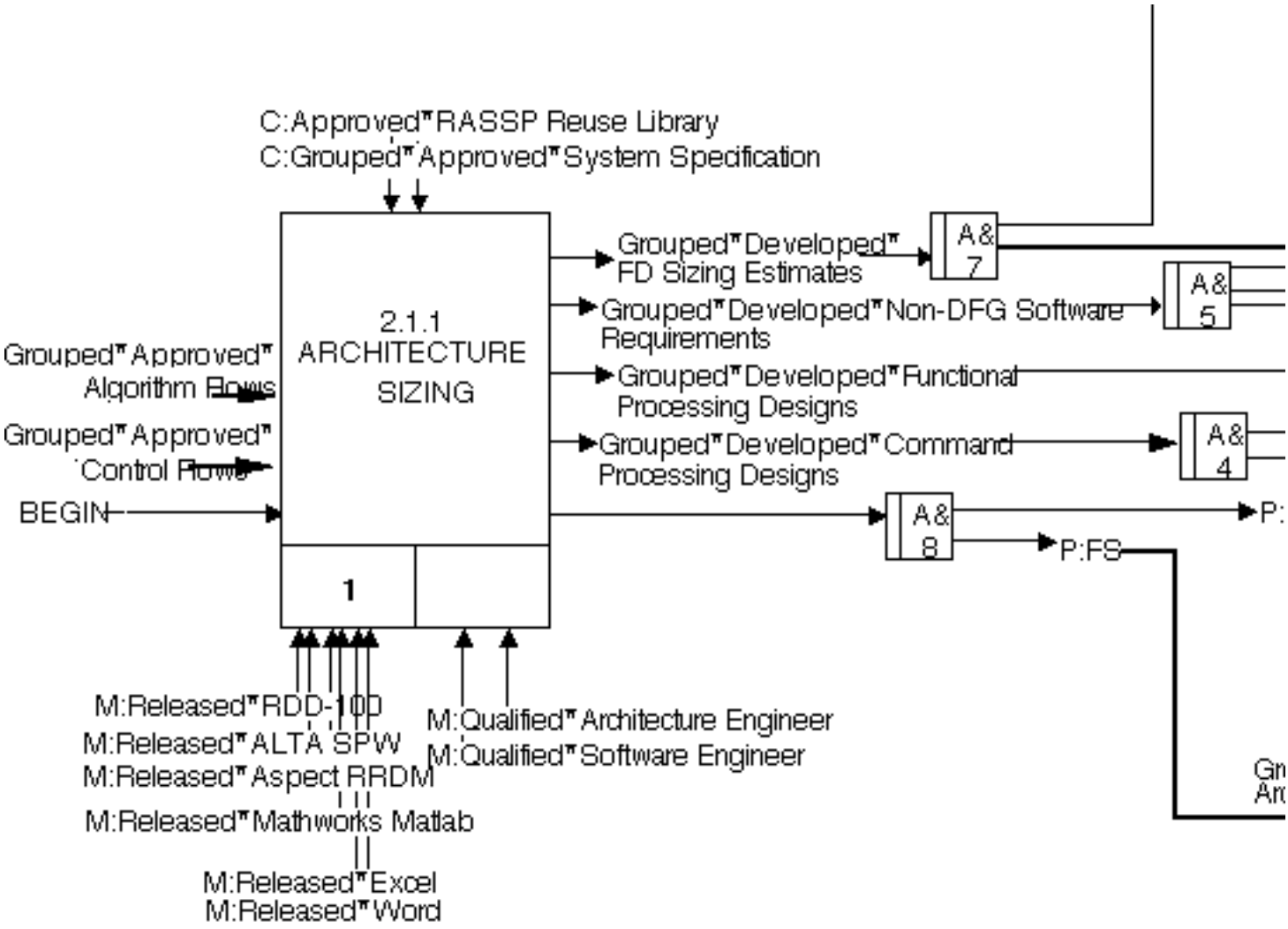


Figure 3 - 1: IDEF3X Workflow Model.

Other states which are used are:

Developed: An item which has just been created during a process step Updated/Refined: A "Developed" item which has been updated/refined during a process step

Preliminary/Verified: An "Updated" item which has been refined and (possibly) simulated and is ready to be approved

Approved: A "Preliminary/Verified" item which has gone through some type of approval or Design Review cycle. The actual level of approval necessary to transition an item to an "Approved" state varies depending on the particular instantiation of the Design Reviews. It can vary from workflow to workflow and also from one implementation to another. There are several different states available:

Generated: An item which is automatically produced from a tool

Qualified: Refers to the skill designation which a person must have in order to be able to execute the process step

Responsible: Indicates which personnel resource will have the authority to transition the process step

Released: Refers to the status of a tool (Currently this is the only status used for tools)

Variable: Indicates that the data item is a pointer to a group of data items, one of which will be chosen based on decisions made during the execution of the program plan. For example, a particular processor may be chosen during a process step. Further on in the program plan, the generic code for that processor will be targeted using a library specific to that processor. The target library for the processor is one of the members of the "Variable*Target Libraries" (on the Architecture Definition:Architecture Verification Workflow).

In Figure 3 - 1, the workflow begins with the process step Architecture Sizing. In the Activity Definitions, it indicates that the purpose of this activity is to analyze the system requirements and processing flows for all required modes in terms of estimated operations per second, memory requirements, and I/O bandwidths. The resulting functional processing flows represent the detailed algorithms that must be performed for each required mode.

The inputs for this activity are the Business Items:
Grouped*Approved*Algorithm Flows
Grouped*Approved*Control Flows

The controls for this activity are:
Approved*RASSP Reuse Library
Grouped*Approved*System Specification
The outputs for this activity are:
Grouped*Developed*FD Sizing Estimates
Grouped*Developed*Non-DFG Software Requirements
Grouped*Developed*Functional Processing Designs
Grouped*Developed*Command Processing Designs

The tool mechanisms are:
Released*RDD?
Released*Aspect RRDM
Released*ALTA SPW
Released*Mathworks MATLAB
Released*EXCEL
Released*WORD

The personnel mechanisms are:
Qualified*Architecture Engineer
Qualified*Software Engineer

The other two activities can be analyzed similarly. It is important to distinguish between junction boxes which are used to distribute/collect data to/from multiple process blocks and junction boxes which are used to indicate precedence. Some Business Items which are outputs of 2.1.1 are directed to 2.1.2 and 2.1.3 via asynchronous "AND" junction boxes. In this case, no timing is implied. Note that similar boxes are used to direct the precedence links out of 2.1.1 to 2.1.2 and 2.1.3. These precedence links indicate that when 2.1.1 finishes, 2.1.2 and 2.1.3 become startable.

For readability purposes, the attached leaf-level Architecture workflows do not have the precedence links indicated on them.

3.5 Summary

This section will serve as a reference for developing program plans. It describes what workflows and Activity Definitions are and how they pertain to the RASSP Methodology in general and the Enterprise, in particular. The workflows should be viewed as templates which can be pieced together to create a program plan. The

Activity Definitions contain supplemental information about each process step.

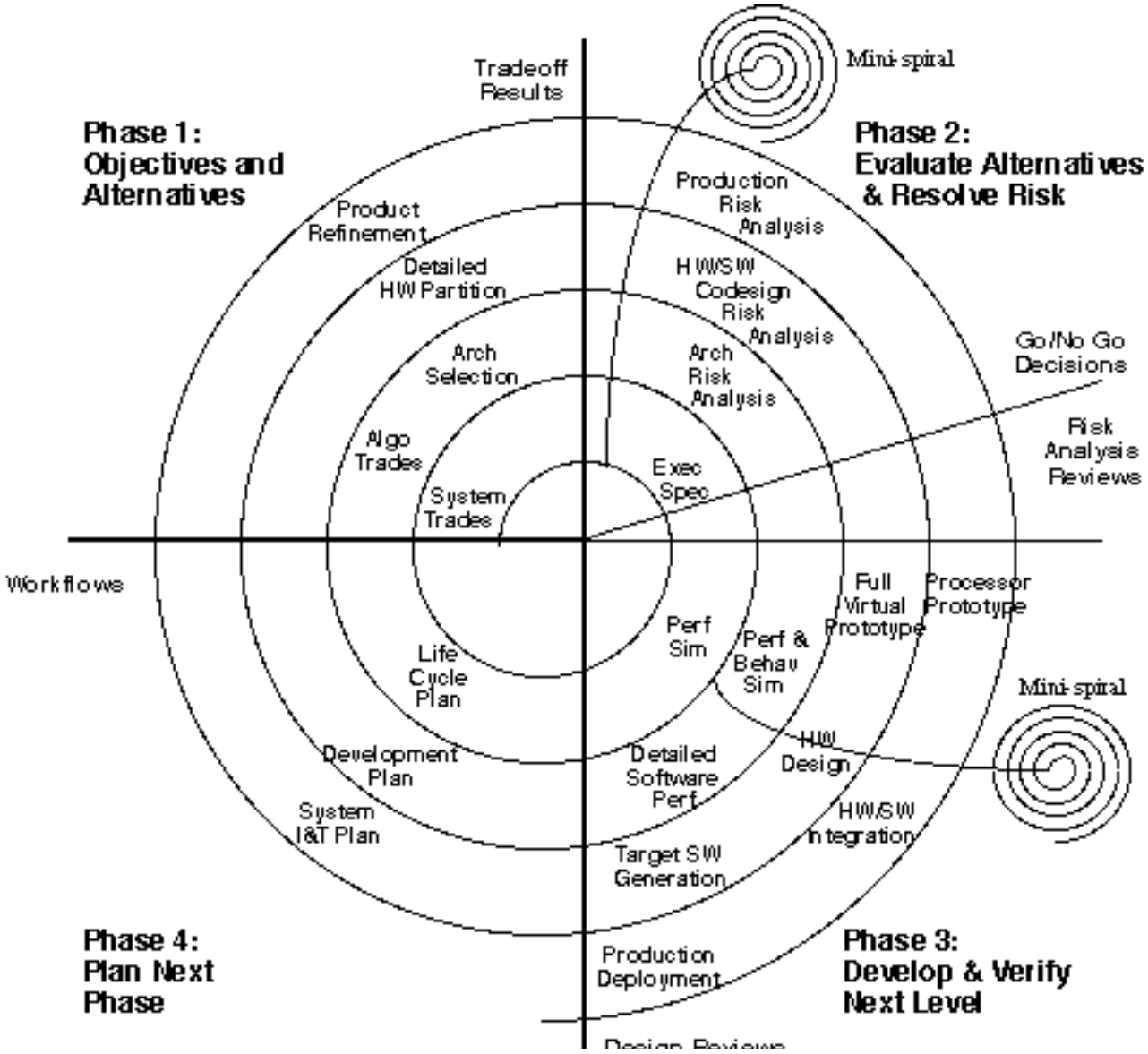
 Dennis Basara

RASSP Methodology Application Note

4.0 Unifying Processes and Roles in RASSP

4.1 Risk Driven Iterative Hierarchical Virtual Prototype Process (spiral model)

The goals of the RASSP program, which include rapid prototyping combined with continual process and product improvements, are more applicable to an iterative, spiral development model than the traditional waterfall development model. The spiral model was originally directed at supporting the software development process. Since the RASSP scope includes more than just software, adaptation of the model is appropriate. A view of the spiral model, as adapted for RASSP, is shown in Figure 4 - 1.



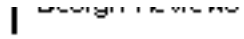


Figure 4 - 1: RASSP spiral model

Four phases are associated with each major cycle of the spiral. In the first phase, the baseline approach and appropriate alternatives are developed to meet program objectives. During the second phase, the approaches are evaluated against the objectives and alternatives, and the risk associated with these approaches is evaluated. During the third phase, the prototype is evaluated and the next level of the product is developed. This phase results in a prototype of the design. In the fourth phase, the product is reviewed and plans for the next development stage are established. This entire process is repeated to the next level of detail by repeating each of the four phases; hierarchical application of the process results in an interactive, risk-driven approach to rapid prototyping.

In the traditional spiral model for software, each cycle might be a full set of software at some level of functionality and/or maturity, or might have critical portions of functionality prototyped (with others "stubbed" out for implementation in subsequent cycles). RASSP applies the concept of virtual prototypes, or simulated versions of the signal processor (encompassing hardware/software functionality and performance) at each cycle of the spiral. During each spiral iteration, a new virtual prototype emerges-from executable specification to high-level behavioral prototype, and then to full functional and timing versions of the design. The design then transitions into a full hardware prototype and matures into a design for full-scale production. Risk analysis and design reviews provide decision points within the cycle and enable backtracking to evaluate alternatives. New spirals are spawned to reduce high-risk elements or continue the design as planned. This model represents a more temporal overlay to the RASSP functional processes and supports concurrency (overlap) between functional disciplines.

In the RASSP methodology the spiral process is applied hierarchically . Each data package may be the result of a series of mini-spirals, while also being part of the primary set of spirals within the design. As high-risk elements in the design process are indentified, "mini-spirals" can be spawned to develop critical items that are "long poles" in the design schedule. Note that this approach allows development across processes as a function of time [e.g., architecture or detailed design tasks can be performed during the early stages (systems process) for high-risk portions of the design]. The logical consequence of the process is that each successive virtual prototype has more complete functionality than its predecessor.

An expanding information view of the spiral process is shown in Figure 4 - 2. The major spiral cycles correspond to the iterations of a virtual prototype associated with the overall signal processor. The mini-spiral cycles correspond to the system, architecture, and detailed design processes associated with portions of the design and/or models (e.g. custom processor, MCM, new hardware model, new software primitive, etc.). This view corresponds to the idea that, based upon risk, pieces of the overall design may be at differing levels of maturity. At the same time, there must be a minimum maturity level achieved and data package produced for the overall signal processor to proceed from one major spiral cycle to the next. The data packages form the basis for the design reviews associated with each major spiral; these are used to drive the next iteration of the design. The data packages are inputs to (and become a subset of) the next stage of the process.

The concept of the virtual prototype as an expanding information model of the processor maps well into this paradigm. At all steps of the design, the model is characterized by four major elements-workflows, requirements, executable models, and test benches. The four elements are summarized as follows:

1. Workflows - The steps used to develop the current/next phase of the development. This is the result of the fourth phase of the spiral model.

2. Requirements - An appropriate set of parameters specifying the performance and implementation goals for the processor (size, weight, power, cost, schedule, etc.).

3. Executable Models - The functional and structural definition of the processor under development. VHDL is being heavily emphasized to specify functionality throughout the design process on RASSP.

4. Test Benches - An appropriate set of test vectors, data sets, etc. that fully verify the functionality and performance of the model. We intend to use VHDL (and the corresponding WAVES standard) to the greatest extent possible for RASSP test benches.

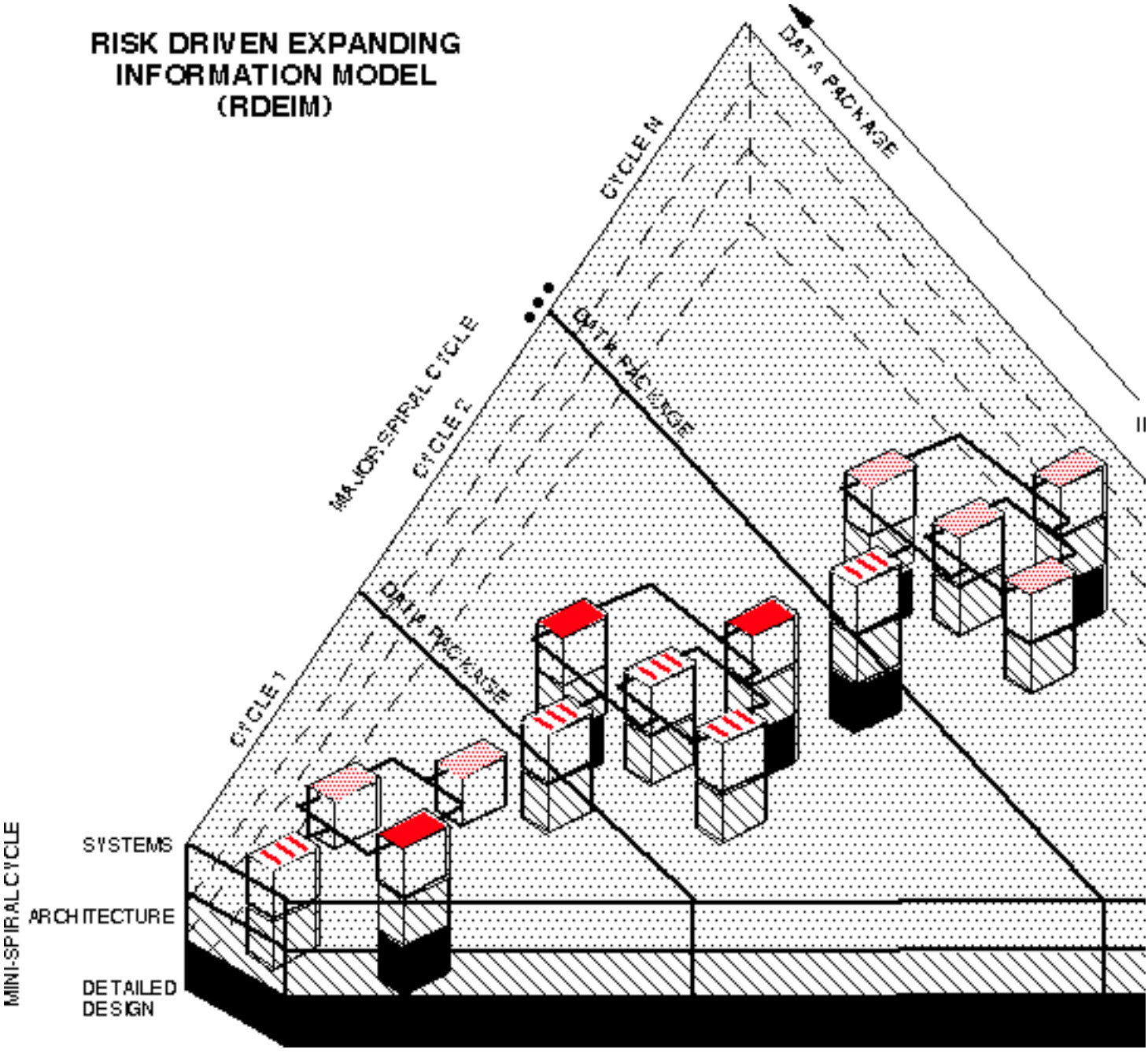


Figure 4 - 2: Risk driven expanding information view of RASSP design process

4.2 Role of Model Year Architecture in the RASSP Methodology

To enable design reuse and minimize upgrade costs, a set of common hardware and software architecture elements is required. A key element to implement this methodology is the selection of a Model Year Architectural approach (both hardware and software) that adheres to the following principles:

- The architectures must be open, promoting hardware/software upgradability and reusability in other

applications
- The architectures must use emerging, state-of-the-art commercial technology whenever possible
- The architectures must support a range of applications to maintain low non-recurring engineering costs
- The architectures must facilitate continuous product improvement

The following section summarizes the approach taken for defining the concept of a Model Year Architecture. A more complete description of the technology, additional references, and actual encapsulations can be found in theModel Year Architecture (MYA) application note.

The Lockheed Martin ATL approach to implementing the Model Year Architecture is based on modular, scalable architectures that use functional standard interfaces. This approach strives to move beyond today's approach of standardizing on physical interfaces; this approach is good, but it does not go far enough to ensure technology independence. Physical interfaces encompass the lower levels of the ISO/OSI protocol hierarchy and are specific to the media and operating conditions (voltage, timing, etc.) specified by these layers. Functional interfaces provide the data-transaction-level capabilities to support communications, independent of the underlying physical technology. By standardizing on functional interfaces, we can maximize independence from technology (electrical versus optical) and specific hardware versus software (processor-based versus dedicated hardware) approaches. We will apply this approach to a number of signal processing interfaces required for RASSP, as shown in Figure 4 - 3. The RASSP team has developed a full description of this approach, which is documented in a set of specifications that can be found in the MYA application note.

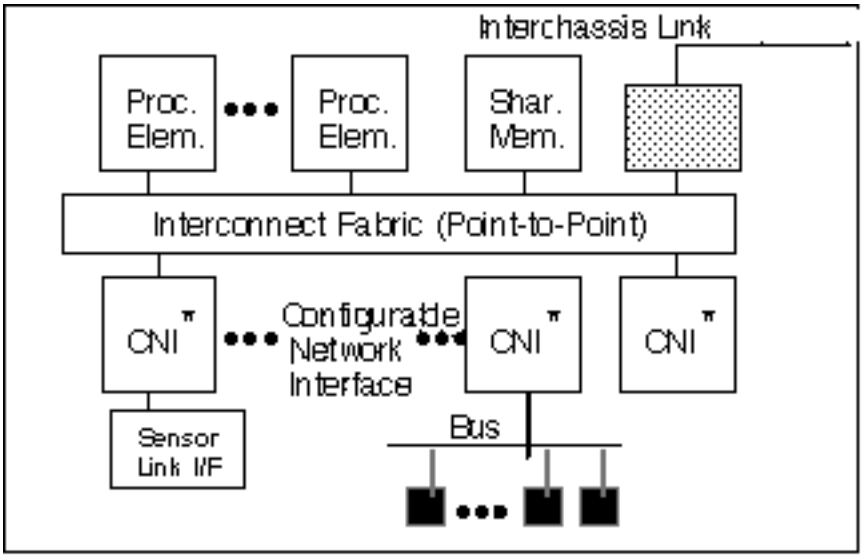


Figure 4 - 3: RASSP Model Year Architecture

The RASSP Model Year Architecture(s) must be supported by the necessary library models to facilitate trade-offs and optimizations for specific applications. Reusable hardware and software libraries facilitate growth and enhancement in direct support of the RASSP Model Year concept. The hardware and software components within the library are encapsulated by the functional wrappers required to enable efficient integration and use. We used the Model Year Architecture methodology to develop all elements for inclusion and use within the reuse libraries. This is key to attaining the 4X cycle-time improvements on RASSP. 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.

4.3 Role of Hardware/Software Codesign in the Methodology

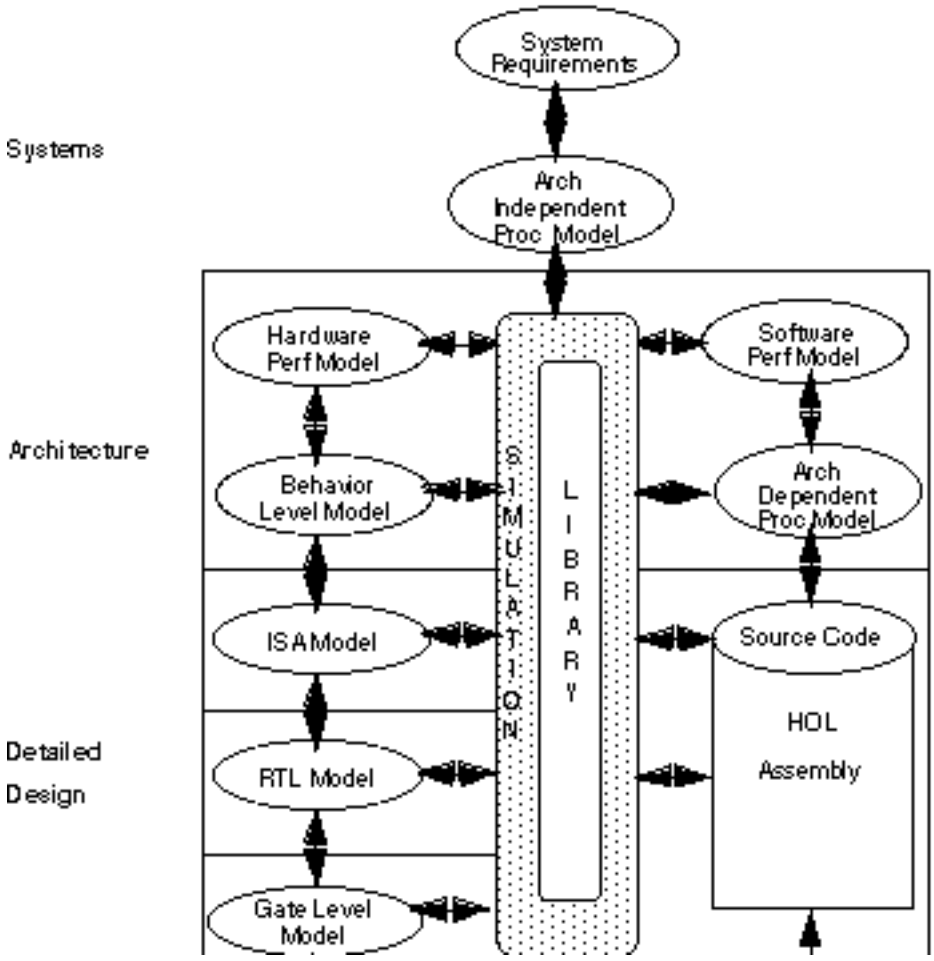
Hardware/software codesign is the joint development and verification of hardware and software through the use of simulation and/or emulation; it begins with initial partitioning and proceeds through to design release.

This section provides an overview of the hardware/software codesign definitions and technologies developed for RASSP. A more complete discussion along with specific application examples can be found in the Hardware/Software Codesign application note.

The principal benefits of hardware/software codesign are:

- Mutual Influence of Both Hardware and Software Early in the Design Cycle - Software performance becomes one of the criteria for selecting an architecture, rather than the more traditional approach of selecting the architecture and forcing the software to fit.
- Continual Verification Throughout the Design Cycle - As the design progresses through subsequent levels of detail, both hardware and software are continually verified to improve design quality. This minimizes iterations after the design is released to manufacturing.
- Enables Evaluation of Larger Design Trade Space - Interoperability of tools and automation of codesign early in the architecture process significantly improves the ability to consider designs which otherwise may be ignored.
- Reduces Integration and Test - Since hardware and software have been co-verified throughout the design process, integration and test is greatly simplified. Test and diagnostics are developed up-front, while there is still the opportunity to impact the design for testability.

The RASSP design process is based on true hardware/software codesign and is no longer partitioned by discipline (e.g. hardware and software), but rather by the levels of abstraction represented in the system, architecture, and detailed design processes. Figure 4 - 4 shows the RASSP methodology as a library-based process that transitions from architecture independence to architecture dependence after the systems process.



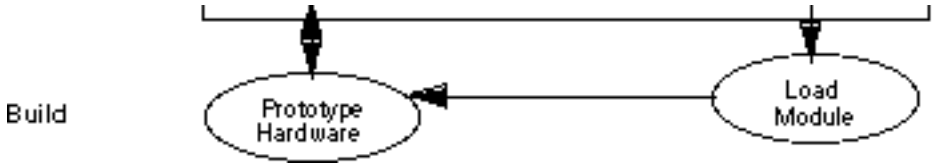


Figure 4 - 4: Hardware/software codesign in RASSP design process

In the systems process, processing requirements are modeled in an architecture-independent manner. Processing flows are developed for each operational mode and performance timelines are allocated based upon system requirements. Since this level of design abstraction is totally architecture-independent, hardware/software codesign is not an issue.

In the architecture process, the processing flows are translated to a standard data and control flow graph description for subsequent processes. The processing described by the nodes in the data flow graph are allocated to either hardware or software as part of the definition of candidate architectures. This becomes the transition to architecture dependence. Requirements and tasks for non-DFG based software and the corresponding software are developed to the maximum extent possible.

The hardware/software allocation is analyzed via modeling of the software performance on a candidate architecture through both software and hardware performance models. For architectures selected for further consideration, hierarchical verification is performed using finer grain modeling at the ISA level and below.

During the detailed design process downloadable, executable application and test code is verified to the maximum extent possible.

Reuse library support is an important part of the overall process. The generation of both hardware and software models is supported in the overall methodology. Software models are validated using the appropriate test data. Hardware models are validated using existing validated software models. Both hardware and software models are iterated jointly throughout the design process.

4.4 Role of Performance Modeling/Virtual Prototyping in the Methodology

Simulation is an integral part of hardware/software codesign. Figure 4 - 5 shows a top-level view of the overall simulation philosophy in the RASSP methodology. During the systems process, functional simulation is performed to establish a functional baseline for the signal processor.

During the architecture process, various simulations are performed at differing levels of detail as the design progresses. Early in the process, performance simulations are executed using high-level models of both hardware and software from the reuse library. Software is modeled as execution time equations for the various processors in the architecture. Models at the behavioral level, for both processing elements and communication elements, are used to describe the architecture. This level of performance simulation enables rapid analysis of a broad range of architectural candidates composed of various combinations of COTS processors, custom processors, and special-purpose ASICs. In addition, many approaches to partitioning the software for execution of the architecture can be rapidly evaluated. Additional detailed information regarding the role and use of simulation can be found in the following application notes:

- Token-Based Performance Modeling
- Virtual Prototyping
- VHDL Modeling Terminology and Taxonomy

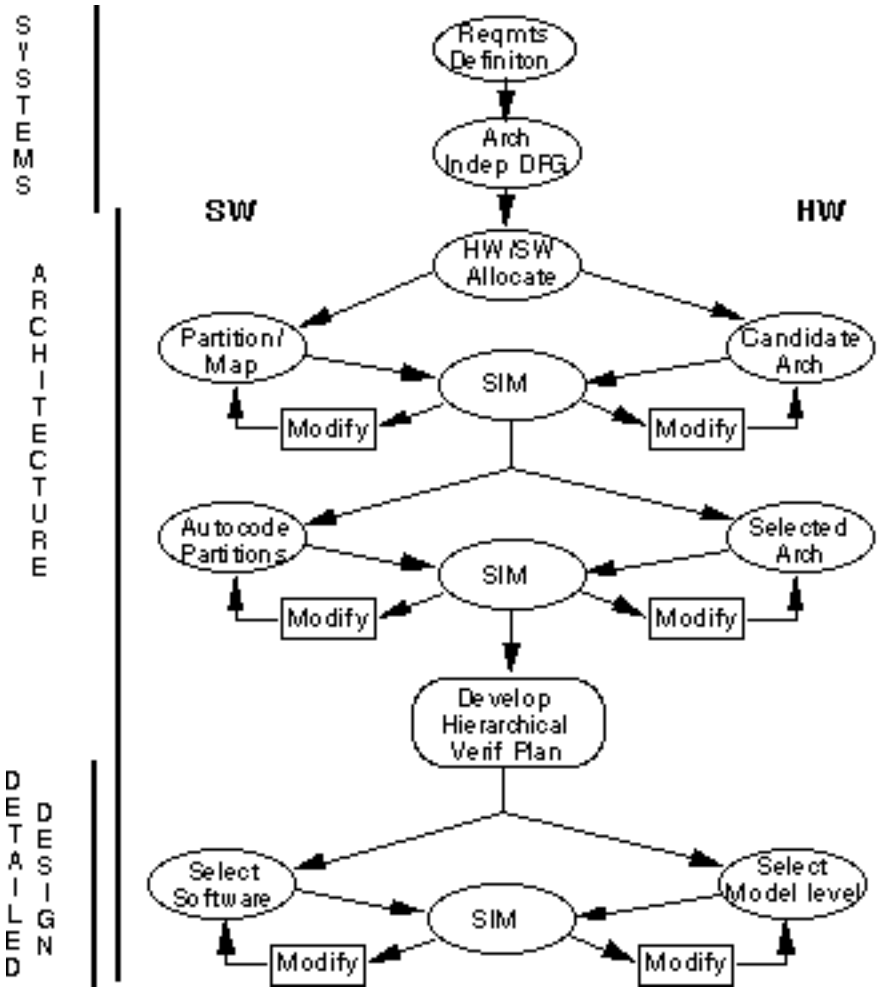


Figure 4 - 5: RASSP simulation philosophy by design process

As the architecture process progresses, each graph partition is translated into a software module for execution on a specific processor in the architecture. Functional simulation is used to verify that the generated code is consistent with the functional baseline. Performance simulation using lower-level models, which include the operating system, scheduling, and support software characteristics, provide the next level of assurance that all throughput requirements are met. Source code for non-DFG based software is developed and tested to the maximum extent possible. Finally, hierarchical architecture verification of the architecture is established using selective performance and functional simulation at the ISA and/or RTL level. The goal is to ensure that all architectural interfaces have been verified.

In the detailed design process, selective performance and full functional simulation are again performed. At this point, however, the design has progressed to where simulation at the RTL and logic-levels is most appropriate. Verification of the designs at this level is necessary before release to manufacturing. It is important to note that pieces of the design may be in different stages of the overall process based upon the risk analysis performed in each development cycle. For example, if it is obvious to the designers during systems analysis that a new custom hardware processor will be required to meet the requirements, the design of the custom processor may be accelerated while the overall signal processor design is still in the architecture process. This approach corresponds to the mini-spirals described in Section 4.1.

4.5 Role of Design For Testability in the Methodology

The Design For Testability (DFT) steps within the overall methodology enables designers to create systems that can be cost-effectively tested throughout their life cycle. Designs that adhere to the methodology are made

testable on the basis of various design for testability (DFT) and built-in-self-test (BIST) techniques. The methodology covers various aspects of test and diagnosis at the chip, MCM, board and system levels, including test requirements capture; test strategy development; DFT and BIST architecture development; DFT and BIST design and insertion; test pattern generation; test pattern evaluation; and test application and control. The methodology provides the designer with a process for introducing testability requirements and constraints early in the design cycle and for addressing DFT and BIST issues hierarchically at the chip, multichip module (MCM), board, and system levels. The payback for early testability emphasis includes lower test cost throughout the life cycle of the product, reduced design cycle time, improved system quality, and enhanced system availability and maintainability.

Elaboration of the DFT steps within the overall methodology can be found in the Design For Testability application note. This application note provides additional detailed references for adopting the DFT methodology. There is a close relationship between the DFT Methodology document and the overall RASSP Methodology Document. DFT and test activities are incorporated into the overall methodology document at a high level. The DFT Methodology document describes these activities in more detail and how they interface with other design activities.

 Dennis Basara

RASSP Methodology Application Note

5.0 RASSP Design Process Description

5.1 Overview

This section addresses the portion of the overall methodology that relates to the RASSP design process. The design process is detailed in Figure 5 - 1. The system design, architecture design, detailed design processes, and library population are addressed in individual subsections. [Click in the shaded boxes of Figure 5 - 1 to obtain the textual overview for that box as well as additional links for more detailed descriptions] Software is discussed in the architecture, detailed design, and library population processes as part of hardware/software codesign; it is also addressed separately to present a consolidated picture. The bold arrow between the individual processes and the library population in Figure 5 - 1 is meant to convey that the overall methodology is an iterative process with feedback from any process to preceding processes. The development of new library elements (software primitives or architectural elements) can be initiated anywhere within the design process, as the need arises.

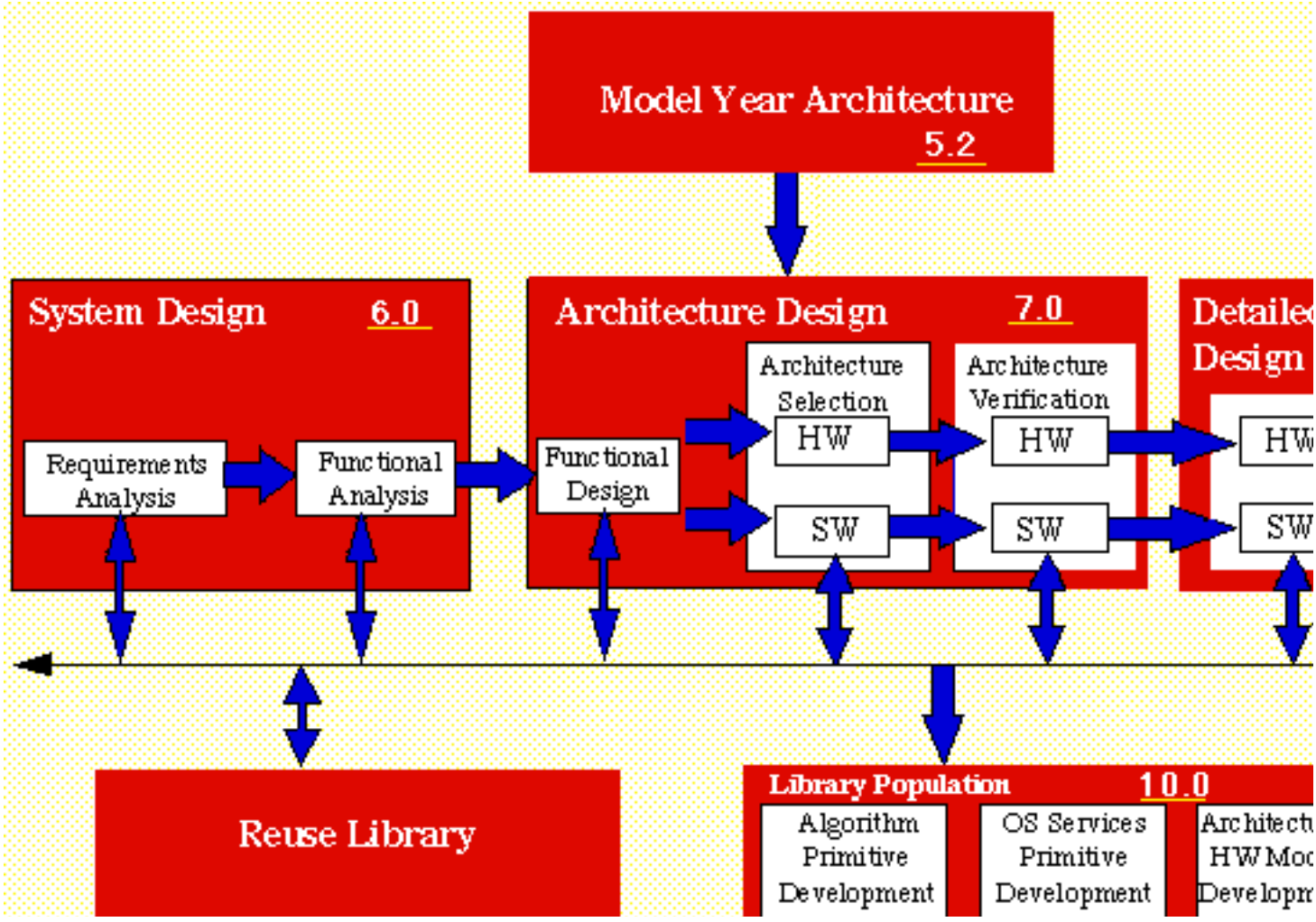


Figure 5 - 1: RASSP design process

Exploring the RASSP design process may be performed in multiple ways in this Section as follows:

1. Click in the shaded boxes of Figure 5 - 1 and a textual overview of the process will be provided. Additional links to more detailed descriptions will also be provided. links will also be found to the Application Notes that provide specific information on various process steps and technology that are employed.
2. Click on RASSP Design Process and the top-level IDEF3X process chart will be displayed. From this top level you can link to the lower, leaf level. IDEF charts that detail the system, architecture. and detailed design phases of RASSP. Links are also provided back to the textual descriptions.

Table 5 - 1 list all of the process steps for RASSP and provides direct links not only to the detailed IDEF3X chart but also to the workflow implementation of that process with the Intergraph Design Methodology Manager. As in the second option above, links to the textual description are also provided.

Design Phase	IDEF3X Model	DMM Workflow
System Design	X	
Requirements Analysis	X	X
Functional Analysis	X	X
System Partitioning	X	X
Architecture Design	X	
Functional Design	X	X
Architecture Selection	X	X
Architecture Verification	X	X
Detailed Design	X	
ASIC Preliminary Design	X	X
ASIC Preliminary Design Review	X	
ASIC Final Design	X	X
ASIC Pre-Release Design Review	X	
ASIC Fabrication and Unit Test	X	X
Backplane Preliminary Design	X	X
Backplane Preliminary Design Review	X	
Backplane Final Design	X	X
Backplane Pre-Release Design Review	X	
Backplane Fabrication and Unit Test Design	X	X
Backplane Final Design Review	X	
Chassis Preliminary Design	X	X
Chassis Preliminary Design Review	X	
Chassis Final Design	X	X
Chassis Pre-Release Design Review	X	

FPGA Design	X	X
FPGA Preliminary Design Review	X	
Module Preliminary Design	X	X
Module Preliminary Design Review	X	
Module Final Design	X	X
Module Pre-Release Design Review	X	
Module Fabrication and Unit Test	X	X
Module Final Design Review	X	

Table 5 - 1: RASSP Process Models and Workflows

5.2 Model Year Architecture Overview

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. Within these domains are many common characteristics that can be served by the same hardware and software architectures. A key element to implement this methodology is a Model Year Architecture approach (both hardware and software) that adheres to a specific set of principles:

- The architectures must be open to promote hardware/software upgradability and reusability in other applications
- The architectures must use emerging, state-of-the-art commercial technology whenever possible
- The architectures 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) must be supported by the necessary library models to facilitate trade-offs and optimizations for specific applications. Reusable hardware and software 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. Details of the Model Year Architecture concept are more fully discussed in the Model Year Architecture application note.

5.3 Systems Design Process

The system process captures customer requirements and converts these system-level needs into processing requirements (functional and performance). Functional and performance analyses are performed to properly decompose the system-level description. The system process has no notion of either hardware versus software functionality or processor implementation.

The first major cycle of the spiral shown in Figure 4 - 1 results in a requirements specification that was captured in an appropriate tool; it is the first instantiation of a virtual prototype (VP0). This information is then translated into simulatable functions, which we refer to on RASSP as an **executable specification.** This cycle represents the first level at which requirements are specified so that we can readily match with simulators to verify performance and functionality in an automated manner. In this phase, processing time is allocated to functions and functional behavior is defined in the form of executable algorithms. At this point, all signal processing functions are implementation-independent. High-risk items can spawn prototype analysis and development efforts in a mini-spiral. The executable specification represents a major portion of the systems design information model. This process results in a System Definition Review (SDR) based upon VP0.

A major portion of the systems behavioral definition is in terms of algorithmic functionality. Generating an executable version of the algorithms using systems-level tools is part of the systems effort. In addition, the detailed specification of all messages that must be passed to and from the signal processor must be generated, along with the definition of all required mode transitions.

Click on Section 6.0 to go to a more detailed description of the System Process.

Click on System Process Application Note to go to the application note that not only discusses the methodology but also the use of the Integrated System Engineering tool environment.

Click on System Design to go to the IDEF3X chart for this effort.

5.4 Architecture Design Process Overview

The architecture design process is broken into functional design, architecture selection, and architecture verification. During functional design, initial performance analysis is conducted based upon the processing flows and requirements flowed down from the systems design process. Processing flows are converted to detailed data-flow graphs based upon reuse library primitives. Non-DFG software requirements and tasks are identified. The architecture selection process transforms processing requirements into a candidate architecture of hardware and software elements. This process, which corresponds to the second virtual prototype iteration (VP1) of the signal processor system, initiates the trade-offs between the different processor architecture alternatives. During this process, the system-level processing requirements are allocated to hardware and/or software functions. A non-DFG software architecture is defined, if necessary, and initial software development is begun. The hardware and software functions are verified with each other via co-verification at all steps. The architecture verification process, corresponding to the next virtual prototype (VP2), results in a detailed behavioral description of the processor hardware and definition of the software required for each processor in the system. The intent is to verify all the code during this portion of the design to ensure hardware/ software interoperability early in the design process.

The architecture design process is a new hardware/software codesign processin the RASSP methodology for high-level virtual prototyping and simulation. Traditionally, the architecture definition has been performed partially in the systems design and partially in the hardware design. RASSP has redefined the major processes by product maturity versus functional area into systems, architecture, and detailed design. Hardware/Software codesign encompasses architecture and detailed design. The primary concern in the architecture design process is to select and verify an architecture for the signal processor that satisfies the requirements passed down from the systems definition process. Although the architecture must include everything required in the signal processor, including any control processors which may be required, this discussion focuses on the approach to defining the number and connectivity of the signal processors required to meet the processing requirements. 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 for size, weight, power, cost, schedule, testability, reliability, etc. so that a more informed assessment can be made. Software and architecture performance is simulated concurrently so that software performance has a direct impact on the selection of an architecture. This direct coupling through co-simulation immediately inhibits any inclination to design the hardware and make the software fit later.

The architecture design process is library based and Data Flow Graph (DFG) driven. Reuse of both architecture elements and software primitives significantly shortens the design cycle. Candidate architectures are constructed from library models. The signal processing software is then partitioned and mapped to the each candidate architecture. A VHDL simulation for the architecture is also constructed and then simulated to estimate performance. Processor behavioral and performance simulations support trade-offs. Mixed levels of simulation (algorithm, abstract behavioral, performance, ISA, RTL, etc.) are used to verify interaction of the

hardware and software. These models are largely hierarchical VHDL models of the architecture. We choose the models, to the maximum extent possible, from the Model Year Architecture elements in the reuse library. The design team develops and inserts new required library elements into the reuse library to support this design phase. The executable specification has now evolved into a more detailed set of functional and performance models that are architecture-specific. Software algorithm implementations are also now specific to the candidate architecture(s). We conduct an Architecture Design Review based on VP1 and VP2 when the architectural trades are completed and the design has been verified to a high degree of confidence. This process is iterated at the high level to achieve one or more satisfactory designs.

The process is supported by an integrated set of tools to foster iterative design and risk reduction. This results in a more detailed evaluation of architectures (from all perspectives) at the early stages of design, which eliminates the false starts that are costly later in the design cycle.

At the conclusion of the systems/subsystems process, a processing requirements specification (both functional and performance) that describes the signal processing functionality is provided. Included in these requirements is a description of all the required interfaces between the signal processor and the outside world, such as mode transition requests, signal processing parameter observation and/or parameter setting, and I/O initiation or termination requests. All requirements flowed down from the systems processare traceable throughout the architecture definition process. Candidate architectures are evaluated against these requirements during the hardware/software codesign process. Updated values of the requirements metrics for candidate architectures are passed back to the systems process. In the event that the requirements cannot be met, additional trades are conducted at the systems level. Note that these inputs to the architecture definition process are completely independent of architecture/processor implementation. It is the objective of the architecture process to transform these processing requirements into candidate architectures and to select and verify an implementation approach.

The architecture design process transforms processing requirements into a candidate architecture of hardware and software elements, through hardware/software codesign and co-verification at all steps. As such, the architecture design process incorporates both hardware and software aspects. This results in a detailed behavioral description of the processor hardware and the appropriate software required to verify these descriptions. In addition, this portion of the process also develops and verifies the implementation-specific portions of both the application (algorithm) and control/support software.

The software portion of the architecture process deviates significantly from traditional (functional decomposition 2167A) approaches. The partitioned software functionality can be broken into four major areas:

1. algorithm, as specified in a data flow graph (see Data Flow Graph Design application note)
2. scheduling, communications, and execution, as specified by mapping the graph to a specific architecture
3. general command/control software (see Autocoding for DSP Control application note)
4. other non-DFG based software, i.e. operating system, board support package, etc.

The intent of RASSP is to automate the first three to the maximum extent possible. This will be accomplished using a graph-based programming approach(es) that supports correct-by-construction software development based on algorithm and architecture-specific support library elements. The execution control of these graphs is provided by a run-time system that provides reusable data flow control, which extends the notion of reuse beyond signal processing primitives.

The general command/control software development (see Autocoding for DSP Control application note) will use emerging, object-oriented code development, documentation, and verification tools. The command program interfaces with the outside world via a messaging system and translates messages into graph and I/O control commands for execution by the run-time system. We design all the non-data flow graph software required for the signal processor during the architecture process. We functionally simulate the joint operation of graph-based and non-graph based software to validate the proper interaction of the command program with the data flow graph execution.

As part of the RASSP methodology, as much design, costing, testing, and manufacturing information as possible will be used throughout the architecture design process through concurrent engineering that is supported by tool integrations and design advisors which cover a wide range of disciplines. The architecture process supports a more formalized trade-off process in which all these disciplines provide valuable inputs.

Click on Section 7.0 to go to a more detailed description of the Architecture Design Process.

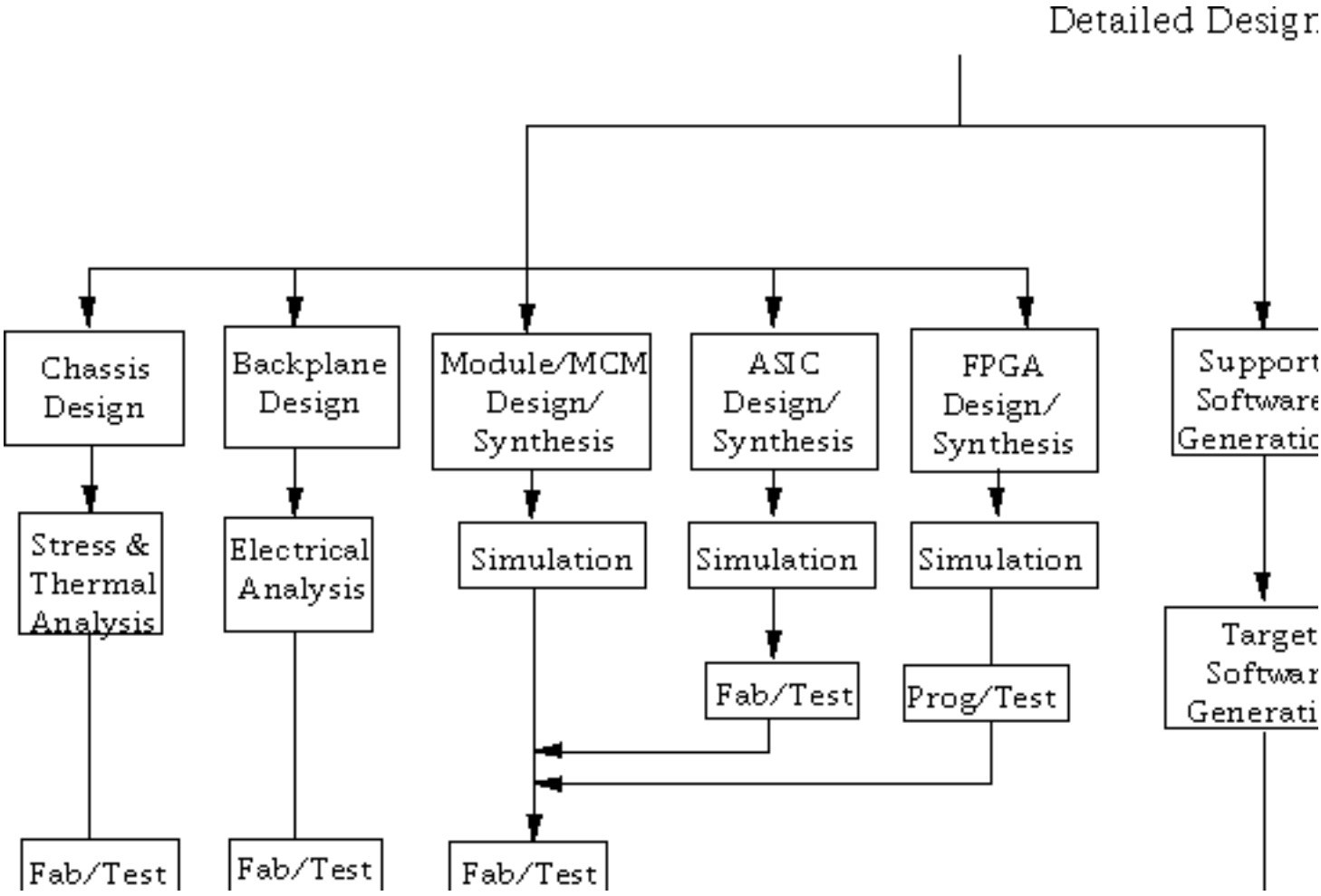
Click on Hardware/Software Codesign Application Note to go to the application note that describes the latest developments in this process as well as the use of the integrated architecture toolset. This will also provide links to more detailed application notes that describe the use of the various technologies involved with this process. These links are also listed below and will be found throughout the detailed discussion in Section 7.0

- Autocoding for DSP Control
- Data Flow Graph Design
- Design-for-Test
- Token-Based Performance Modeling
- Virtual Prototyping

Click on Architecture Definition to go to the IDEF3X chart for this effort.

5.5 Detailed Design Process

The main objective of the detailed design process is to transform the architectural description of the design into the detailed hardware and software components that we will develop, manufacture, and integrate into a prototype processor. Figure 5 - 2 shows the top-level view of detailed design.



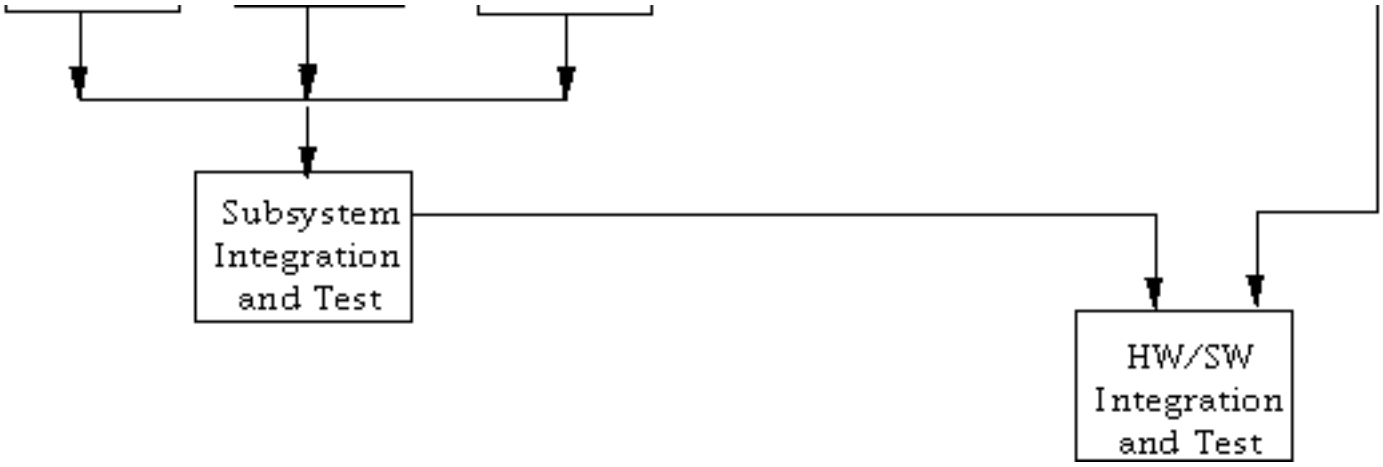


Figure 5 - 2: Detailed Design Process

We partition and develop designs at the behavioral-level (RTL or higher) for processor-level nodes to the appropriate level for all necessary components. This includes ASIC, MCM, and board-level designs, as well as backplane/chassis designs. The process incorporates mechanical and manufacturing elements to ensure that designs meet all specifications for the particular applications. From an industry standpoint, the detailed design process is the most mature area of design, and the improvements on RASSP are optimizations to support rapid prototyping. Toward this end, the process heavily favors two areas:

- Design reuse of validated elements from the RASSP component library to minimize detailed design time
- Synthesis of chip and board-level components from the component libraries.

In the case of software, we must generate any support code that is target specific for the signal processor that has not been completed. This includes initialization, bootup, diagnostics, test software, and any support software required for integration and test which is not part of the operational software. All the software is compiled and verified (to the extent possible) on the final virtual prototype before the detailed design review.

Like the architecture process, this portion of the methodology supports hardware/software codesign, as these elements are fully verified together on the virtual prototype up until the design is released to manufacturing. This virtual prototype iteration (VP3) involves the detailed design of software and hardware elements. As with the prior processes, we design and verify both hardware and software via a set of detailed functional and performance simulations. When this process is completed, the design is established, resulting in a fully verified virtual prototype of the system. We revalidate software that was verified during architecture verification as we develop more detailed models for any of the components. This process provides continual verification as the designs progress toward physical hardware. **The transition of these designs to manufacturing** is supported by the RASSP enterprise system to provide electronic linking capabilities supporting both concurrent interaction between design and manufacturing, and direct transfer of designs to manufacturing sites.

During the hardware portion of the detailed design process, we transform behavioral specifications of the processor into detailed designs (RTL and/or logic-level) through a combination of hardware partitioning, parts selection, and synthesis. Detailed designs are functionally verified using integrated simulators, and performance/timing is also verified to ensure proper performance. The process results in detailed hardware layouts and artwork, net lists, and test vectors that can then be seamlessly transitioned to manufacturing and test via format conversion of the data. We generate the entire design package required for release to manufacturing for the Detailed Design Review based on VP3, which corresponds closely to today's Critical Design Reviews (CDRs).

At the top level, the detailed design process for hardware is, for the most part, the same for boards, MCM, ASICs, etc. However, at the lower levels, these designs are quite different and use different tools. The

hardware process is partitioned as a function of the hardware element, e.g., chassis, backplane, board, chip, etc.

We start the overall hardware design process with the subsystem (e.g., a signal processor) components definition requirements from the architecture verification process. This is documented at the level of detail of a Type C specification (MIL-STD 490). Architecture verification previously analyzed these requirements, performed architecture trade-off studies (supported by VHDL simulations at the algorithm and architecture level), partitioned the functional requirements into analog/digital/mechanical, and generated documents for all individual elements, including backplanes (frames/cabinets), modules, and ASICs.

The hardware design process flow is shown in Figure 5 - 3. In RASSP, partitioning is driven by component requirements from the architecture process, with a heavy predisposition toward using library-based components to support the in-process design concept. Where possible, we use off-the-shelf modules, MCMs, ASICs, chassis, and backplanes, and we will be aided by knowledge-based advisors with data on available hardware elements to drive synthesis-based approaches. We will specify off-the-shelf hardware for procurement, and we will design custom hardware for fabrication. All hardware must be modeled or must have testbed hardware in place for joint hardware/software simulation and verification. The concept of hardware/software codesign at this level is a new element introduced by RASSP.

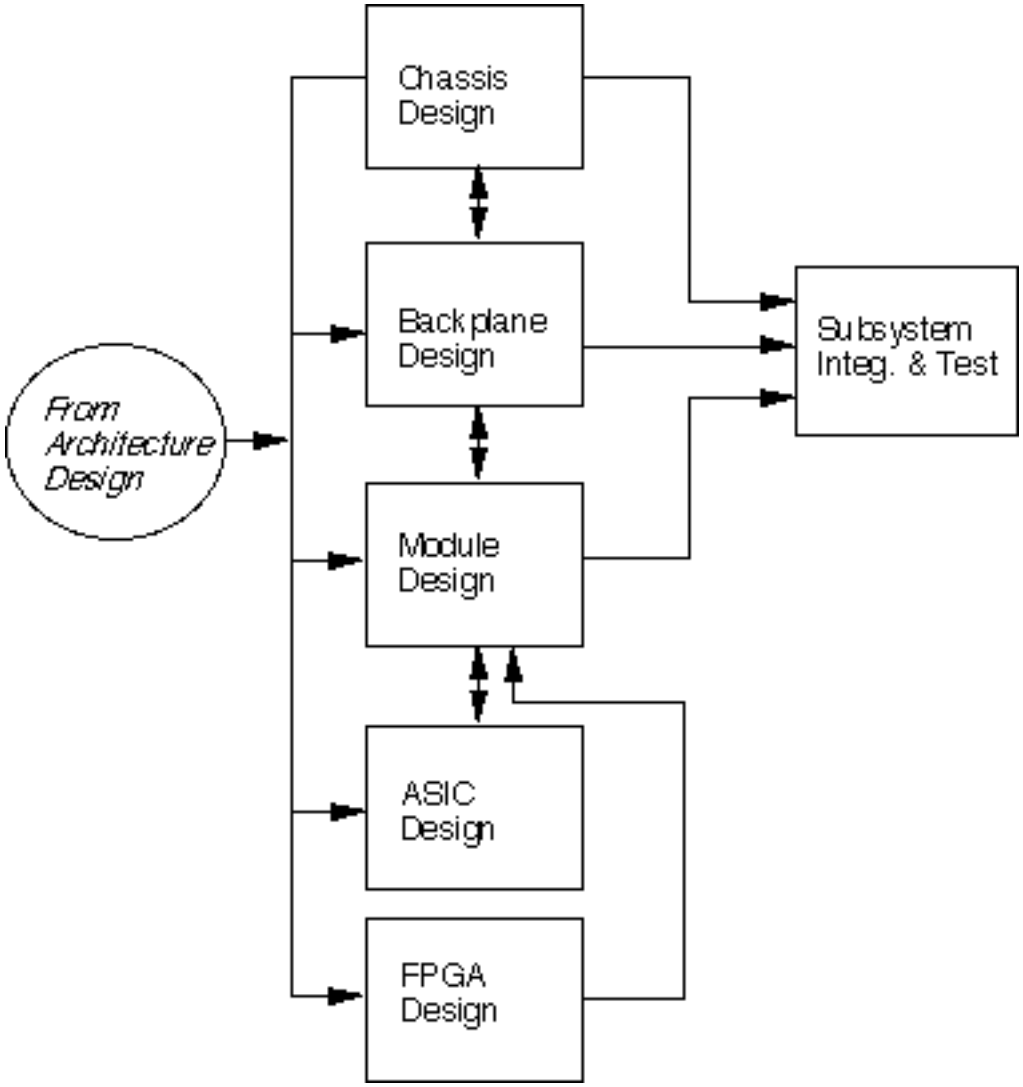


Figure 5 - 3: Hardware Design Process Flow.

In interactive logic design, we develop a structural VHDL description from the algorithm architecture

level-VHDL descriptions, which include DFT structures. We simulate this model to verify functionality and performance, then synthesize it, generate a layout, and perform a thermal analysis. Back-annotated simulations are done followed by a critical design review after verification. We then manufacture hardware and test it.

Click on Section 8.0 to go to a more detailed description of the Detailed Design Process.

Click on Detailed Design to go to the IDEF3X chart for this effort.

 Dennis Basara

RASSP Methodology Application Note

6.0 System Design Process Detailed Description

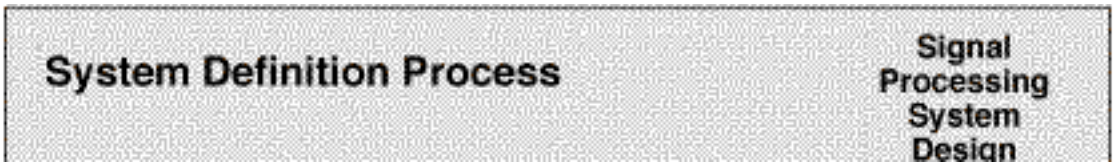
A similar description to the one that follows is in the System Design Process Application Note. This application note also discusses the design automation tools that were developed to support this activity. It also contains an application description for the use of the process and tools on one of the benchmark programs. If your interest is in the System process flows and a general description of the tasks and products of this part of the design process then explore this discussion further. If your interest is more to the details of what and how this part of the process works along with the EDA tools, then proceed to the application note.

The RASSP methodology provides several key improvements over traditional system processes. The output of the system definition process is a set of executable specifications for the signal processor design activity. The definition and functions encompassed by the executable specification is receiving focused attention on the RASSP program. A hierarchical set of simulations is performed at each design level, and the results of these simulations are back annotated in the higher-level simulations. In RASSP, we emphasize considering LCC early in the design process and the reuse of library elements at the system and subsystem-levels.

The system design process is a front-end engineering task in which signal processing concepts are developed to meet customer requirements and top-level tradeoffs are performed to define the signal processing subsystem requirements. Depending upon one's perspective, a system could represent a platform, sensor system, signal processor or processing board. For RASSP, a system is defined at the signal processor level. For this document, a "system" represents a signal processing system and a "subsystem" represents a major component of the signal processor. The system design process for RASSP starts after the requirements have been established for the signal processing system. A multidiscipline Product Development Team (PDT) performs the functions specified by the system design process. The specific roles of the PDT team members are described in Section 6.4.3.

The inputs to the system design process include all the customer documentation detailing the processing system specification. The outputs of this process include the functional, performance, and physical requirements for each signal processing subsystem. Typical signal processing requirements include system mode functional descriptions (search, track, waveforms, algorithms), performance requirements (processing gain, timeline and precision requirements), physical constraints (size, weight, power, cost, reliability, maintainability, testability, etc.), and interface requirements. The system definition process is iterative, requiring constant interaction with the customer and the product development team members as the impact of system-level decisions on LCC are significant.

The system design process is shown in Figure 6 - 1. Top-level trade-offs are performed to determine how the system will operate and what set of subsystems are required. System-level functional and timeline simulations are developed to characterize system behavior. The output of the system design process is a set of functional, performance and physical requirements for each subsystem. As the subsystem designs progress, key subsystem parameters are back annotated, and system-level simulations are re-run to ensure that performance is maintained.



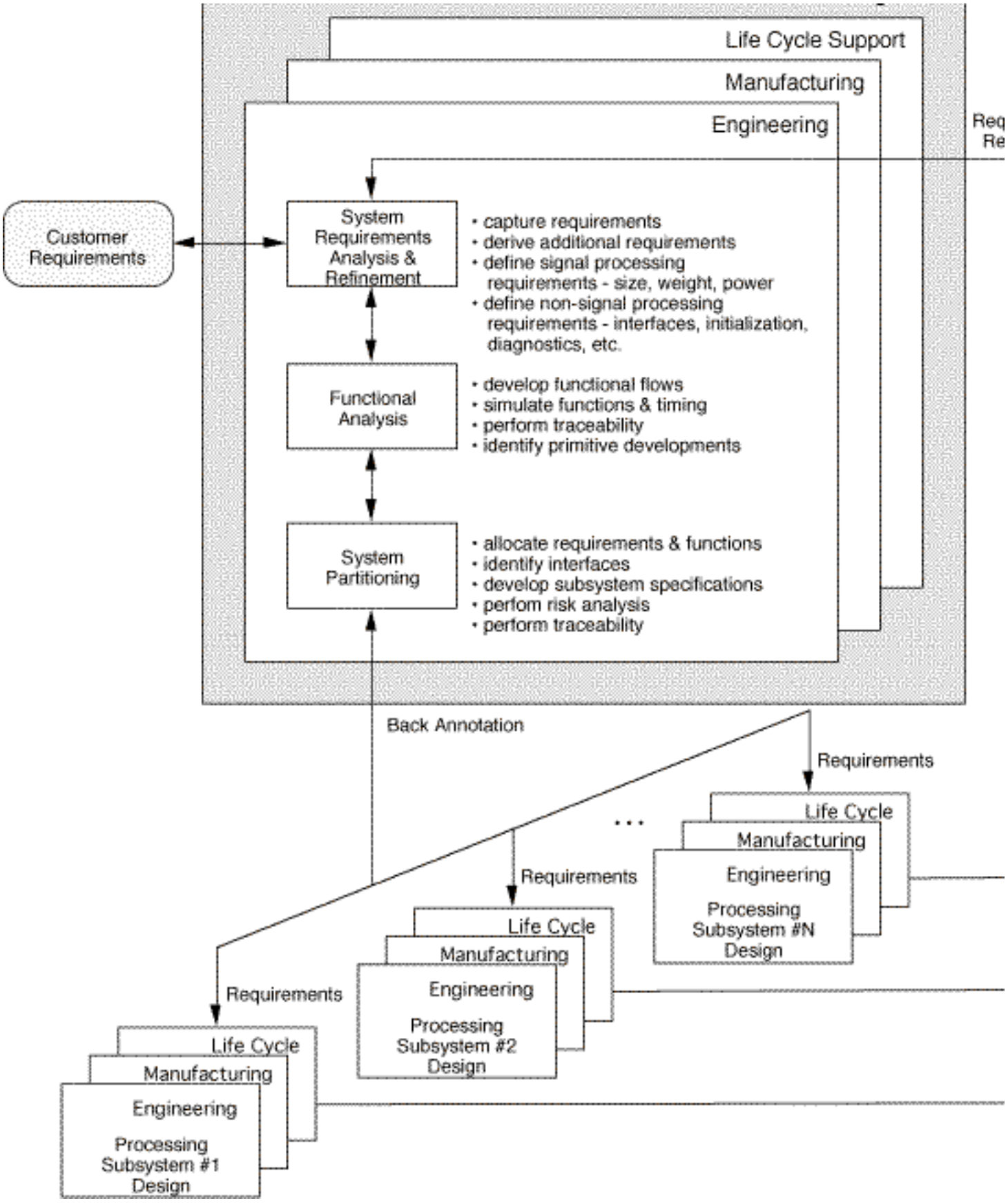


Figure 6 - 1: System Design Process.

Subsystem requirements are constantly monitored to make sure the development risks are balanced among the subsystems. There is a feedback path back to the system-level from each subsystem design; this path is used whenever cost-effective subsystem designs cannot be obtained. System requirements are then reexamined and new analyses is performed to determine a refined partitioning of the requirements. By clicking on the system design process box in the IDEF figure the next level of system design can be explored. The IDEF process model of this second level will be presented and textual descriptions of each subprocess may be obtained by clicking on the text button or by clicking on the link below. This level consists of three main subprocesses:

1. **Section 6.1 Systems Requirements Analysis and Refinement**

   The operational and procurement requirements are initially examined to ensure that all requirements are well understood. There is close interaction with the customer to clarify any confusion with the requirements. A traceable path must be established when requirements are allocated to functions and components. Both mission and threat analyses are performed to understand how the signal processing system should behave. The system is defined by describing the system modes, functions and interfaces. Measures of effectiveness (cost, performance, risk, testability, etc.) are established for the system to provide metrics to compare different system configurations. Operational scenarios are also develop to assist in determining system performance.

2. **Section 6.2 Functional Analysis**

   The system is decomposed into its functional elements while establishing the requirements. This functional analysis is performed by determining what functions are required to implement each system requirement. Functions are described by defining the inputs to the function, the algorithm performed by the function, and the outputs of the function. Constraints and timing requirements are identified for each function. The top-level system behavior is modeled to determine the functional performance of the system. Signal-to-noise ratios, detection ranges, probability of detection, and probability of false alarms are several examples of system-level behavior for a signal processing system. System test and maintenance concepts are developed. All functions within the system must be traced back to the system requirements.

3. **Section 6.3 System Partitioning**

   The functions of the system are allocated to subsystems as the functional requirements are established. At this point, various system configurations are developed and characterized to determine the baseline system. Trade-off analyses are typically performed for the following areas: reliability, availability and maintainability; testability; LCC; schedule and technical risk; integrated logistics support; human factors; and system safety. All system requirements must be traceable to both functions and subsystems. The output of the system partitioning process is a set of executable specifications for each signal processing subsystem.

   Other items that need to be considered during this phase of the process are considered below:
   - Task Concurrency

     The system definition process steps of requirements analysis, functional analysis, and system partitioning are closely interrelated. The tasks shown in Figure 6 - 1 are performed concurrently, often as a series of iterations to trade-off alternative approaches and to successively provide greater detail. The system requirements are first examined before functional analysis can begin. However, the functional behavior of the system can be developed while the requirements are analyzed and refined with the customer and end-user of the system. A preliminary system functional baseline is required at the System Requirements Review. This corresponds to the first level of the spiral model. In addition, the system partitioning process begins after the initial functional baseline is identified. The limitations of various equipment configurations identified during system partitioning must be accounted for in the functional behavior simulations. The system definition process is an iterative process

where requirements analysis, functional analysis, and system partitioning activities are performed concurrently to define the subsystem requirements. The system definition process is closely coupled with the architecture design activities. As the signal processor design matures, quantified processor parameters such as throughput rates and precision must be back annotated into the system-level functional simulations. In addition, the system activities continue throughout the signal processor design. The allocation of system requirements to subsystems is closely monitored to ensure that the development risks are balanced among the subsystems.

- Risk Management

  Risk management is an integral part of the system design process as sources of technical, schedule, and cost risk are identified. Risk analysis is part of all of the trade studies that are conducted. The probability of occurrence for each risk and the resulting impact on performance, development schedule, and LCC as well as their interrelationships are determined. Risk reduction plans for the risks likely to occur or those that have a critical impact on the program are generated. The key to risk management is to identify and quantify the individual risk elements so that the overall system risk can be set at an acceptable level. Another key is to track key design parameters as the system is developed to monitor the risks throughout the program. Risk management activities are formally examined during all design reviews.

6.1 System Requirements Analysis

System requirements analysis converts a user need into a combination of elements that satisfies that need. It involves analyzing customer documentation and conducting discussions with the customer to refine the purpose and manner in which users will operate the system. It is designed to determine what the system is to do and how the system is to be used. The system requirements are thus developed from a user's point of view. External interfaces to the system, usage scenarios, and capacities are developed, and methods are determined to verify each requirement statement. The process iterates with functional analysis and system partitioning efforts to assess feasibility and to structure the requirements cost-effectively. Unnecessary implied designs should not be incorporated in the requirement statements. This iteration also makes the verification process more accurate and cost effective by eliminating ambiguity in the requirements statements.

Baselines are used to control the development process. The functional baseline is established following the system design phase, which prescribes:

- all functional and performance characteristics
- all tests required to demonstrate achievement of functional and performance characteristics
- interfaces between subsystems and their functional characteristics
- design constraints.

Although the functional baseline is not established during the system requirements analysis phase, preliminary data is provided. The RASSP reuse library is examined to determine whether any functional elements are applicable for this baseline. Several baselines can be maintained simultaneously while system alternatives are being evaluated. Typically, the initial baseline is in the form of a requirements database. Formal configuration management is not used until a single baseline has been established. The baseline will be reopened whenever the subsequent design, integration, or test activities indicate the requirements cannot be supported or when customer change requests result in a modification to the requirements.

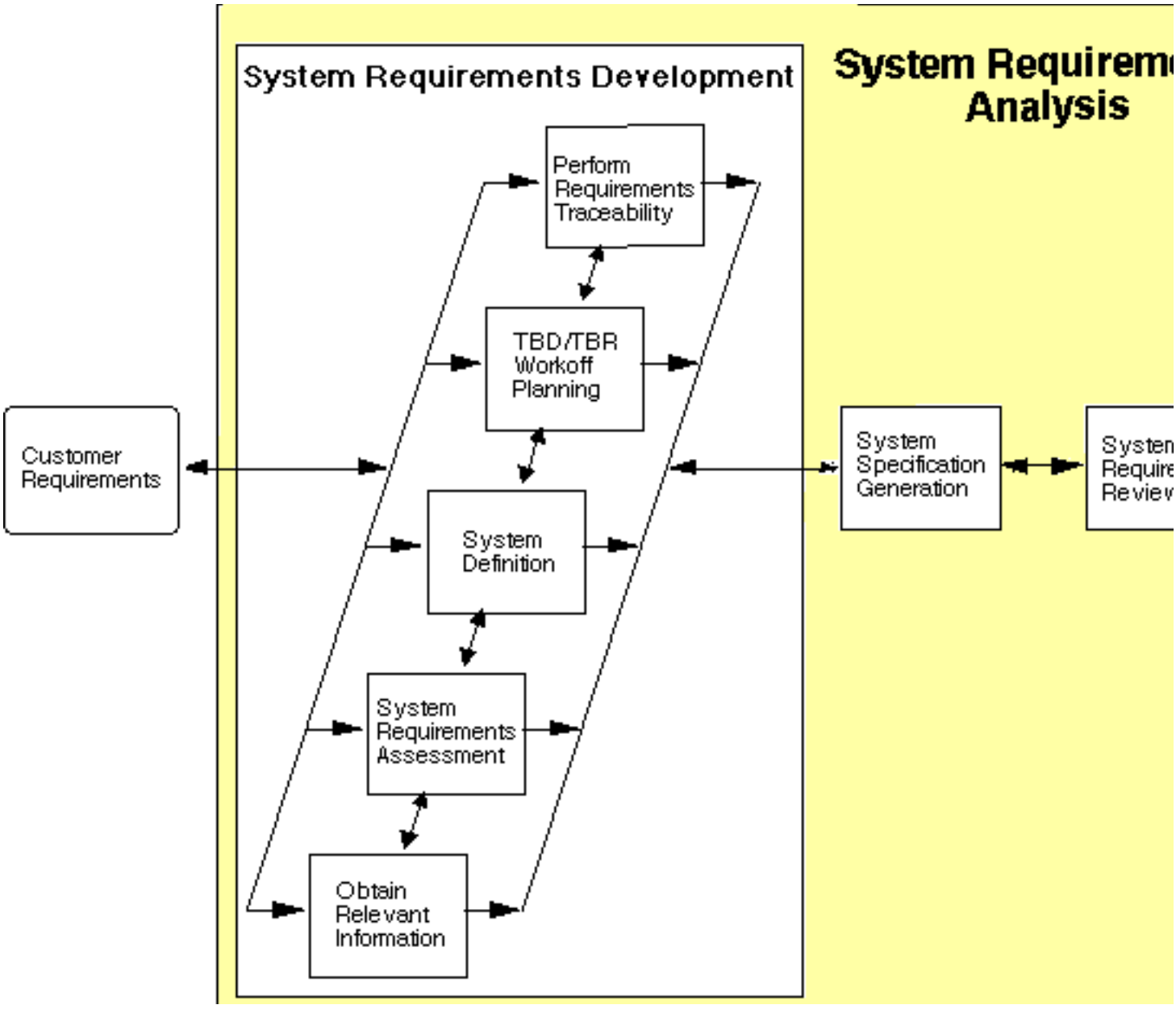
By clicking on the system requirements analysis and refinement process box in the IDEF figure, the last level of the system design process can be explored. The IDEF process model of this third level will be presented and textual descriptions of each subprocess may be obtained by clicking on the text button or by clicking on the link below. The system requirements analysis process is composed of three primary subprocesses:

- Section 6.1.1 System Requirements Development

- Section 6.1.2 System Specification Generation
- Section 6.1.3 System Requirements Review

6.1.1 System Requirements Development

In this process step, the system requirements are checked for completeness and consistency using the steps in Figure 6 - 2. Each step is performed iteratively to determine the system requirements. A complete set of customer documents must be used to determine the system requirements because the contractor must understand how users intend to use the system. The system gets defined in terms of its system modes, states, functions, and interfaces. Trade-offs establish alternative performance and functional requirements to meet customer needs. Any potential conflicts in the trade-off results with the system requirements are resolved. A workoff plan is created for all TBD/TBR items that identifies the responsible individual, schedule for resolution, risk analysis, and key trade-offs to be performed. Traceability of system requirements and decisions ensures that the trade-off decisions made in generating the system requirements can be tracked and that these requirements are completely and accurately reflected in the final design. Traceability is also used to assess the impact of changes at any level of the system. All system-level requirements are traced to their source during this process. --



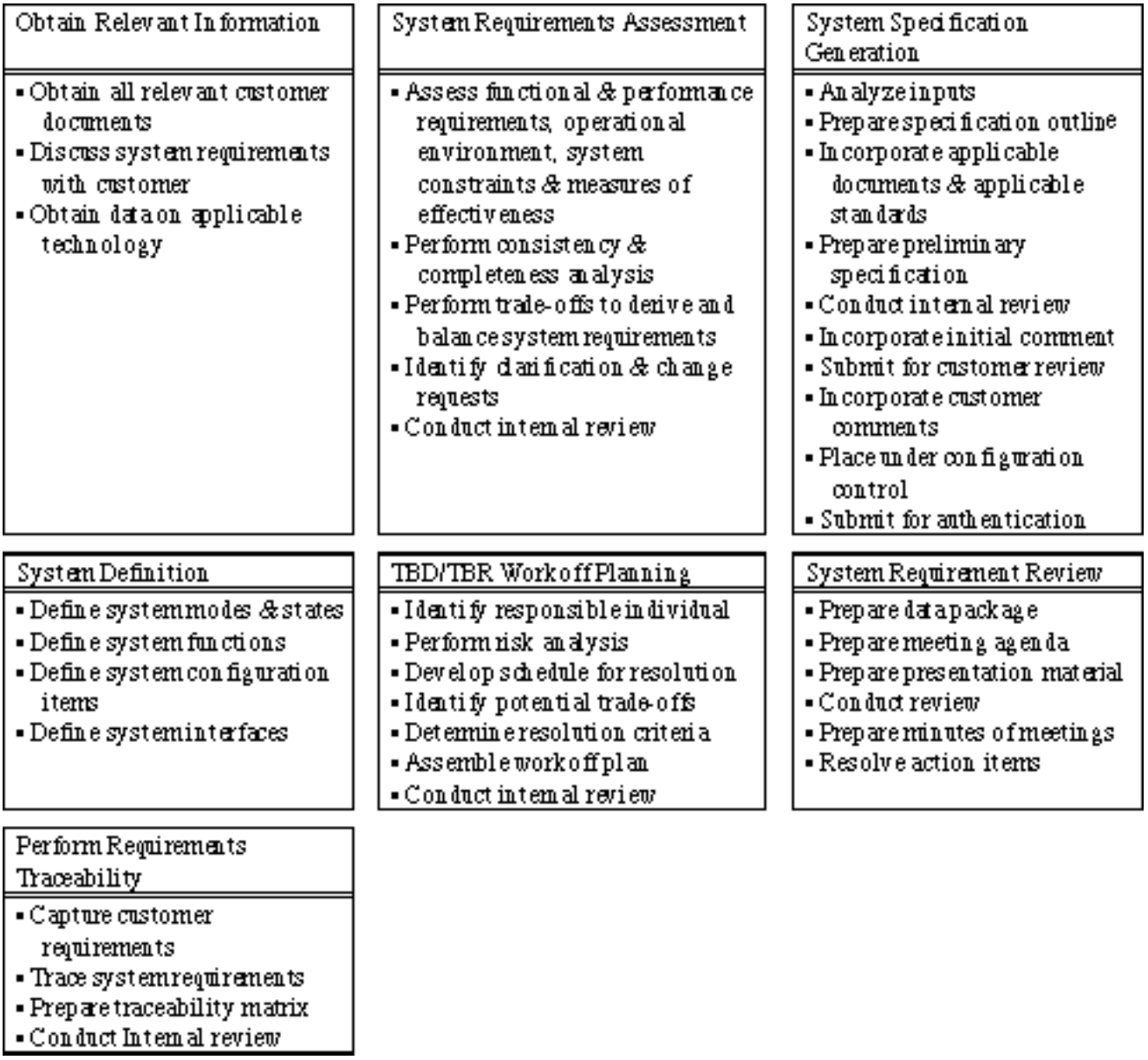


Figure 6 - 2: System requirements analysis

6.1.2 System Specification Generation

The signal processing system specification contains:

- the technical requirements for the system
- allocates requirements to functional areas
- documents design constraints
- defines interfaces between functional areas

It also states all necessary requirements in terms of performance, including test provisions to ensure that all requirements are achieved. Essential physical constraints and requirements for application of any known specific equipment must be included. The use of the system engineering tools, described in the system process application note, assist in automating this task.

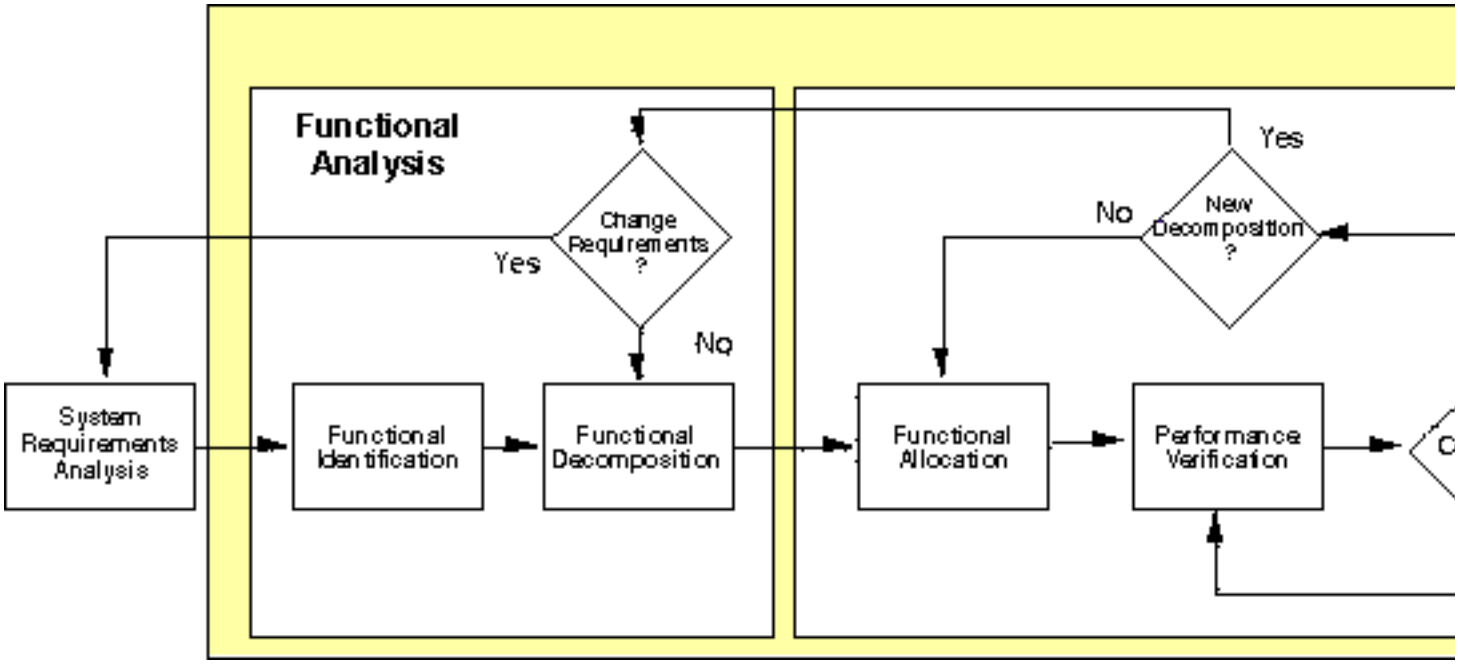
6.1.3 System Requirements Review

A System Requirements Review (SRR) is conducted to ascertain the adequacy of the contractor's efforts in defining the processing requirements. It is conducted when a significant portion of the system functional requirements have been established.

6.2 System Design Functional Analysis

Concurrent with the establishment of the system requirements, the system configuration (the components within the processing system) is defined through iteration of the functional analysis and the system partitioning processes. Functional analysis is designed to identify the functions, decompose the functions into requirements, and allocate these requirements to lower-level functions. System partitioning takes the functions from the functional analysis process and allocates them to entities within candidate configurations. These allocations are analyzed to determine the performance, cost, and schedule impacts of the alternatives. Unsuccessful allocations and configurations are eliminated. Multiple other alternatives are carried to the next phase as the final best configuration may not be determinable until the Architecture design phase is completed.

The functional analysis process describes the requirements as a set of verifiable (simulatable) statements that can be used as a basis for system design. The functions, constraints and performance statements are decomposed to a detailed level that can be allocated to specific candidate design configurations. The output of this process is a functional baseline that describes the system functionality and is the basis for eventual customer sell-off. The functional analysis process is shown in Figure 6 - 3. This process is composed of two steps, functional identification and functional decomposition, which are described in the following paragraphs.



Functional Identification	Functional Decomposition	Functional Allocation
▪ Analyze system requirements ▪ Identify system functions ▪ Develop functional block diagrams ▪ Identify constraints ▪ Establish performance timelines ▪ Evaluate baseline for consistency	▪ Perform trade-offs to eliminate poor system configurations ▪ Identify next tier functions ▪ Develop next tier functional block diagrams ▪ Identify next tier constraints ▪ Establish next tier performance	▪ Create candidate system configurations ▪ Allocate functions to configuration ▪ Allocate constraints to configuration ▪ Identify interfaces

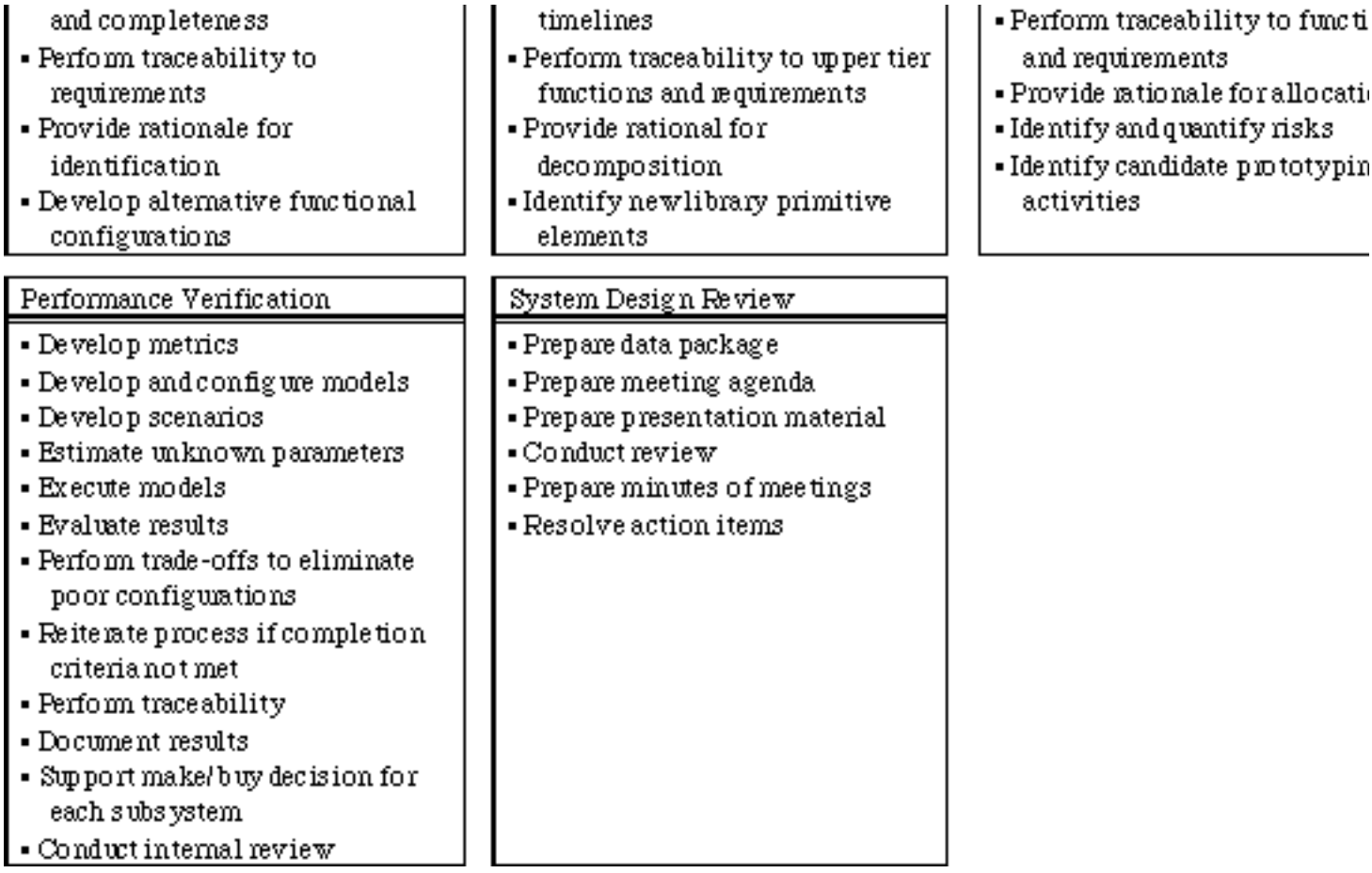


Figure 6 - 3: System Functional Analysis and Partitioning

By clicking on the functional analysis process box in the IDEF figure, the last level of the system design process can be explored. The IDEF process model of this third level will be presented and textual descriptions of each subprocess may be obtained by clicking on the text button or by clicking on the link below. The system functional analysis process is composed of three primary subprocesses:

- Section 6.2.1 Functional Identification
- Section 6.2.2 Functional Decompostion
- Section 6.2.3 Informal Functional Analysis Design Review

6.2.1 Functional Identification

The functional identification process translates system requirements, customer heritage, and customer rationale into functional block diagrams that are used by subsequent processes to create and evaluate system configurations. This step refines and decomposes the functions identified from the system requirements analysis step. Functional identification also develops the design constraints under which the system must operate. Functional elements within the RASSP reuse library are examined to determine whether existing library elements can be used. The key steps in the functional identification process include:

- analyzing system requirements
- identifying top-level system functions
- developing functional block diagrams
- identifying constraints
- establishing performance timelines
- developing alternative functional configurations
- evaluating the functional baseline for consistency and completeness

- performing traceability of the functions to the system requirements
- providing the rationale for the top-level functional description

6.2.2 Functional Decomposition

Functional decomposition creates lower-level functions, constraints, performance, and interfaces from those that currently exist. This more detailed information is used by the subsequent partitioning and architectural trade process. This process is structured and accomplished iteratively to eliminate poor allocation decisions as early in the analytic process as possible. Functional elements within the RASSP reuse library are examined to determine whether existing library elements can be used. If any of the system functions cannot be described by existing library elements, then we identify new primitives for development. The specific steps in functional decomposition include:

- performing trade studies to eliminate poor system configurations
- identifying the next-tier functions and functional block diagrams
- identifying next-tier constraints
- establishing next-tier performance timelines
- performing traceability to the upper level tier functions and requirements
- providing the technical rational for the functional decomposition.

6.2.3 Informal Functional Analysis Design Review

At this time no formal design review is required with the customer. However. it is recommended that an internal informal review be conducted to examine the decomposition that has been accomplished and the initial partitioning that is concurrently being performed. This peer review ensures that requirements and functionality have not been overlooked and that assumptions are valid before beginning the architecture design process phase.

6.3 System Partitioning

During the system partitioning process, we define and evaluate candidate system configurations to determine which configurations most effectively meet the functional and system requirements. As many configurations as feasibly possible should be evaluated in enough detail to rank the alternatives. A structured method must be developed to quickly identify the feasibility of a specific candidate configuration. We typically develop measures of effectiveness for a specific program as the basis to evaluate trade studies. The output of the system partitioning process is the set of functional, performance, and physical requirements for each subsystem in the baseline configurations. For RASSP, an executable specification for each signal processing subsystem is the starting point for the architecture selection process. This executable specification is the first virtual prototype (VP0) of the subsystem. As shown in Figure 6 - 3, the system partitioning process is composed of three subprocesses: functional allocation, performance verification, and system design review. Each of these subprocesses is described in the following paragraphs.

6.3.1 Functional Allocation

The functional allocation process allocates functions from the functional block flows and constraints to entities in a specific candidate design. This process iterates until an allocated baseline can be established. This process is complete when all functions are allocated to subsystems and all requirements and constraints are mapped through functions to subsystems. This process may require many iterations to refine the system configuration. Trade-off analyses assess the risk and LCC impacts for each alternative system configuration. A key feature in RASSP methodology is the ability to quickly assess the LCC for various system configurations. Design decisions and rationale must be documented as the functions are allocated. The specific steps in the functional allocation process include: creating candidate system configurations; allocating functions and constraints to components; identifying interfaces between the subsystems; performing traceability of the system configuration to system and functional requirements; and providing the technical rationale for the functional allocation.

6.3.2 Performance Verification

The performance verification process supports the system synthesis process by providing evaluation criteria to determine which candidate configuration provides more effective performance. This effort must consider all factors of interest to the product development team: technical performance, risk, LCC, producibility, supportability, testability, etc. Critical inputs to the evaluation process are the operational concept, operational scenarios, operational maintenance concept, and usage rates expected by the user. The performance evaluation must reflect objective, demonstrable evaluation metrics and must assure the customer we considered all alternatives. The functional verification process models the allocated functions of candidate configurations to determine whether performance requirements are met. The set of mathematical elements within the RASSP reuse library is used (whenever possible) to quickly construct the functional-level simulation to verify technical performance. This process iterates with the functional allocation process until the performance of a candidate configuration meets the completion criteria established during the systems requirements analysis process. If no candidate configuration meets the completion criteria, the procedure reverts back to the functional decomposition process for an updated decomposition. If no candidate configuration meets the completion criteria for multiple decompositions, the process reverts back to the system requirements analysis process for requirement refinements. The specific steps in the performance verification process include: developing metrics, models and scenarios; configuring models; estimating unknown parameters; executing models; evaluating results; eliminating poor configurations; reiterating process if completion criteria is not met; performing traceability to requirements; documenting results; supporting the make or buy decision for each subsystem; identifying candidate prototyping activities to reduce program risk; and conducting an internal review.

The RASSP architecture selection process transforms the processing requirements for each processing subsystem into a candidate processing architecture of hardware and software elements. The architecture selection process overlaps with the system definition process during the later portions of the system partitioning activity. A set of executable specifications is used to transfer the signal processor requirements to the architecture selection process. A hierarchical set of simulations is performed at each design level, and the results of these simulations are back annotated in the higher-level simulations to verify that performance is maintained.

6.3.3 System Design Review

A System Design Review (SDR) is conducted to ascertain the adequacy of the contractor's effort in defining the baseline system. This review is held after the performance of the candidate system configuration has been verified to meet system requirements.

6.4 Other Considerations during System Design

6.4.1 Use of VHDL in System Design Process

The output of the system definition process is the set of executable specifications for the DSP system. The signal processor specification contains requirements that can be divided into three categories:

1. Timing/Performance (e.g. processing latency, throughput, I/O timing)
2. Function (e.g., algorithms, control strategies)
3. Physical constraints (e.g. size, weight, power, cost, reliability, maintainability, testability, scalability, temperature, vibration)

VHDL is applicable for conveying system function, timing, and performance information and is used to develop a test bench and model of the signal processing system. Many efforts, some of which are being funded under RASSP, are focusing on conventions for associating physical information with VHDL models. As they become available, the methodology will expand to include their use with VHDL.

The system model captures the specification for the timing, performance, and function of the DSP system and its interfaces. It also contains a structural specification of its interfaces.

The test bench provides system stimulus and checks that the applicable system requirements are satisfied. As in all cases, the VHDL test bench is developed before the model it is intended to test. This ensures a thorough analysis and understanding of the requirements before and during design. In essence, the test bench is an interpretation of the aspects of the system requirements that can be described in VHDL, while the system model is an expression of the design solution that satisfies the requirements.

The system model forms an executable specification at the system definition level. The executable specification supports design-for-test (DFT), since the system test concepts must be considered and developed for its verification. The performance model is periodically back annotated with timing and performance data from the more detailed design levels. To continually ensure that system requirements are being met, the performance model is executed within the test-bench to verify continued compliance with the system requirements.

The executable specification consists of a test bench and a system model described in the following paragraphs.

- Test Bench
  The test bench provides test procedures, stimuli, and expected responses for verifying that the system model meets system requirements. The test bench is developed before, and is executed on, the system model.

- System Model
  The system model describes: system timing, performance, and, when possible, system functionality and physical constraints in a way that facilitates automatic testing to verify compliance with system requirements. Elements within the system model are summarized in Table 6 - 1. The algorithm descriptions are in terms of the basic arithmetic operations to be performed. The arithmetic operation sequence is typically expressed in the form of a data flow graph (DFG) that indicates control flow and data dependencies.

System Timing and Performance Data	System Functionality Data	Physical Constraints Data
▪ Signal Processing I/O Data - I/O Timing Constraints - I/O interface structures - I/O protocols - Signal levels - Message types ▪ Signal Processing Latency - Data Acceptance Rate ▪ Signal Processing Stimuli Response	▪ Algorithm Descriptions ▪ Control Strategies ▪ Task Execution Order ▪ Synchronization Primitives ▪ Inter-process Communication (IPC) ▪ BIT and Fault Diagnosis	▪ Size ▪ Weight ▪ Power ▪ Cost ▪ Reliability ▪ Maintainability ▪ Testability (fault coverage, diagnosis BIST goals) ▪ Repairability ▪ Scalability ▪ Environment Constraints - Temperature - Vibration - Pressure - Stress and Strain - Humidity - EMI/EMF/EMP

Table 6 - 1: Elements within the system model

6.4.2 Design For Test Tasks in System Definition

Requirements are derived from customer and/or parent system requirements and maintained during the system definition process. All of the functions including test functions such as BIST are partitioned into processor system and subsystem functions. An overall model of the processor system is developed which is referred to as VP0. This model together with testbenches represents the inputs, outputs and transformations of the inputs by the processor system including latency. Key outputs of the DFT efforts are the consolidated test requirements (which promotes a singular test strategy across design verification, manufacturing test and field support), preliminary test strategy diagram, TSD0, and testability architecture, TA0. Figure 6 - 3 shows the DFT steps which occur during the system definition step. The key step, requirements analysis, is shown in more detail in Figure 6 - 4.

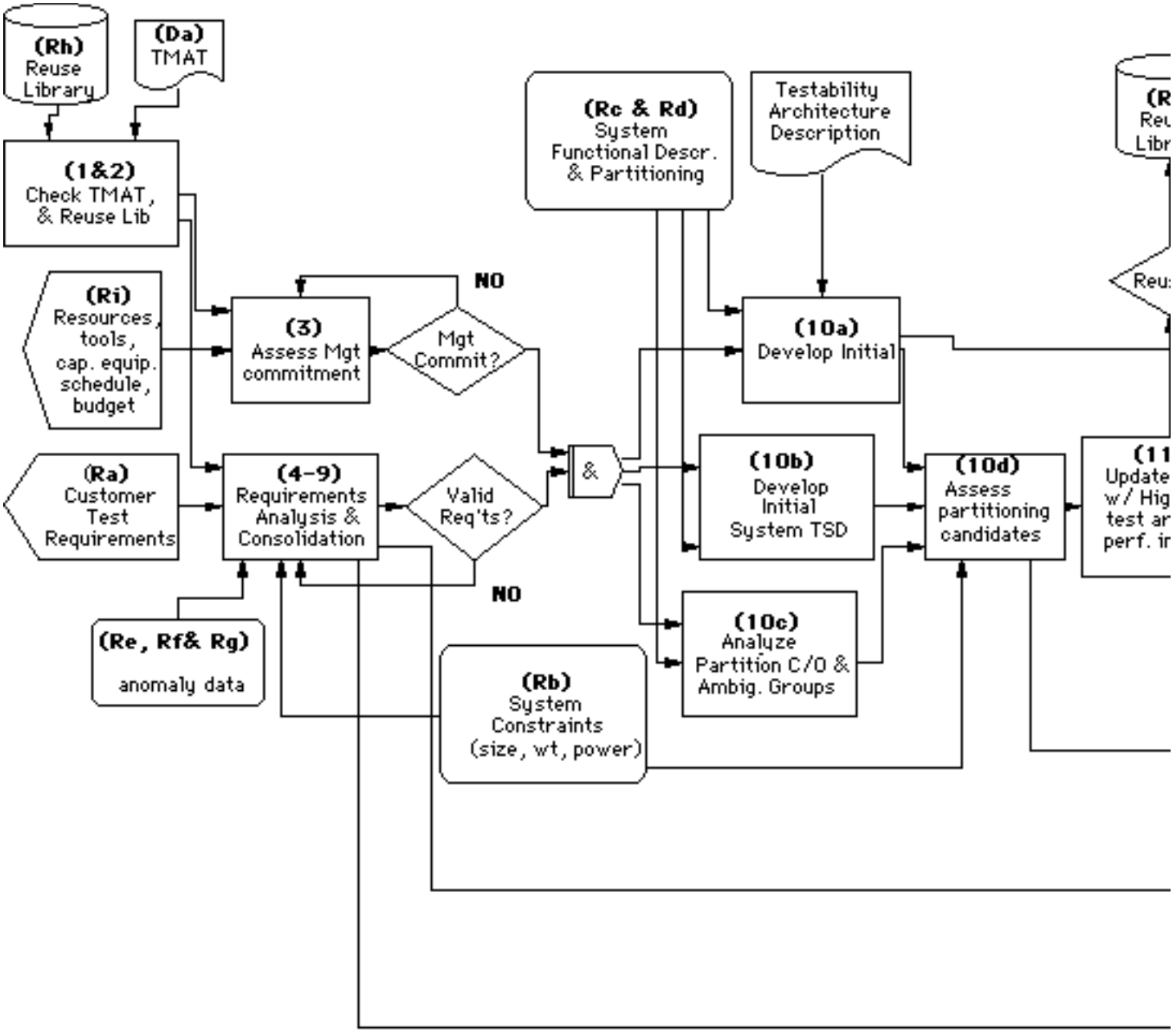


Figure 6 - 4: DFT steps in system definition flow diagram

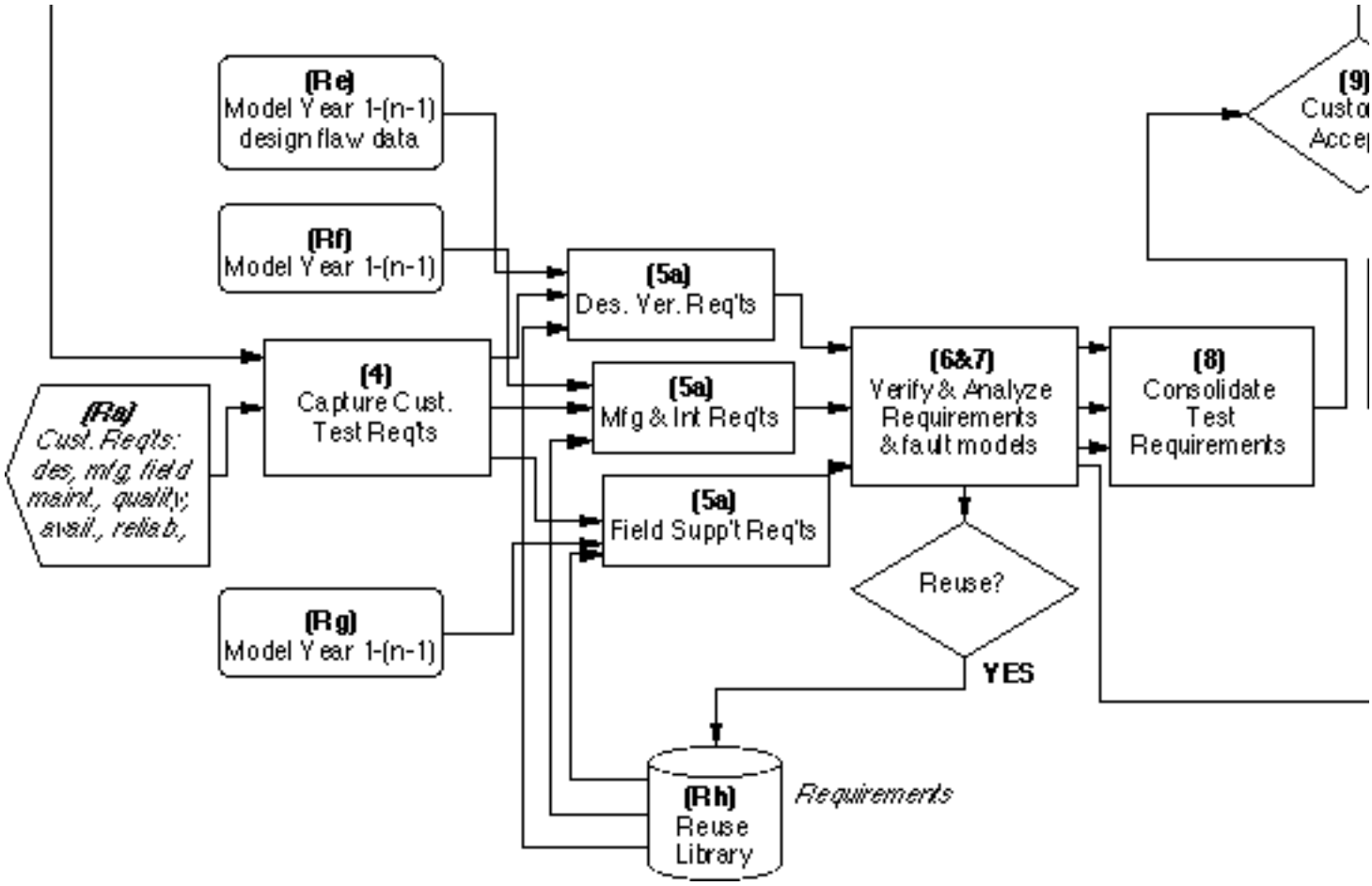


Figure 6 - 5: DFT requirements analysis flow diagram

The presence of COTS may throttle attempts to reach a singular life cycle test strategy. One reason is that black box COTS elements are usually tested functionally (for defect detection and isolation), while most of the rest of the design that is not black box oriented is usually tested using a great deal of structural testing and some functional testing. An analysis of the degree of anticipated COTS usage and the extent to which current COTS incorporates test features, would tend to allow an early, preliminary assessment of what plateau of test architectures might be possible

The impact of using COTS components during system definition is to place practical limitations on the test requirements and subsequent test strategies. For example, it does not make sense today to specify a requirement that can only be achieved with 100% boundary-scan, if a large part of the system will use COTS boards that do not come in boundary-scan versions.

The development of the fault model in this step is also affected by the presence of COTS, particularly if the COTS element is a "black box" in terms of its structural composition. Several options exist for handling fault models for black box COTS elements:

1. Establish a structural fault model, in which the "node" is the lowest level input or output for which a description exists. This level will usually be at the inputs and outputs of blocks at the block diagram level or could be simply the inputs or outputs of the package itself (such as a chip) or board.

2. Attempt to define a "functional fault model." This is a fault model based on the function, rather than the structure of the item; and it describes the erroneous functional behavior that would occur in the presence of a fault. While this approach will work for simple functions, it is not likely to be practical for large VLSI based components or boards.

Refer to the Design-for-test application note for further discussion of DFT and other references.

6.4.3 Role of the PDT in System Definition Process

A multifunctional PDT is required to perform the system definition process. The disciplines needed for this process in RASSP include systems, architecture, software, digital hardware, mechanical, manufacturing, test, producibility, sourcing, logistics support, and cost analysis. The role for each discipline is described in the following paragraphs.

- Team Leader - The team leader for the system definition process is the technical director for the project. The technical director is responsible for the overall integrity of the system-level design. The technical director coordinates all activities during the system definition process. The technical director leads system integration and technical management activities.

- Systems - The systems engineers have the lead role in performing the system definition process. The system engineers perform the system-level design, interpret the customer requirements, control development of the subsystem specifications, and coordinate system and subsystem trade studies.

- Architecture - The architecture engineers perform the hardware/software trade-offs to determine the processing architecture within the signal processor subsystem. During the system definition process, the architecture engineers participate in the system-level trade-offs that allocate the system functional requirements to each processing subsystem. The architecture engineers determine the impact on LCC on the signal processor for various functional and physical requirement allocations to the processor.

- Software - Software engineers have a limited role during the system definition process since allocation of functional requirements to software elements is not performed during this process. The hardware/software codesign activity is performed during the architecture selection process in the RASSP design methodology. During the system definition process, the software engineers analyze the system-level software requirements to determine the impact of these requirements on the system design.

- Digital Hardware - Digital hardware engineers have a limited role during the system definition process since allocation of functional requirements to hardware elements is not performed during this process. The hardware/software codesign activity is performed during the architecture selection process in the RASSP design methodology. During the system definition process, the hardware engineers analyze the system-level hardware requirements to determine the impact of these requirements on the system design.

- Mechanical - The mechanical engineers ensure the structural and thermal integrity for the system during all phases of the product's life cycle. During the system definition process, mechanical engineers analyze the operational requirements for the system and perform the top-level tradeoffs to determine the structural and packaging requirements for each subsystem. The mechanical engineers assess the impact of various packaging approaches on LCC. The mechanical engineers monitor the structural and thermal trade studies performed during the subsystem designs.

- Manufacturing - Manufacturing personnel define the integrated manufacturing program used to build the system. During the system definition process, manufacturing personnel examine the functional and physical requirements allocation to each subsystem and assess the impact of these requirements on current manufacturing processes. Manufacturing personnel participate in the make/buy decisions for each subsystem during the system definition process.

- Test - The test engineers develop the test procedures for the product throughout the life cycle. During the system definition process, the test engineers examine and determine the system-level test requirements needed to determine whether the product is functional working on the manufacturing floor, in the depot, and in the operational environment. The system-level test requirements are then

decomposed and allocated to each subsystem during the system definition process. These test requirements must be an integral part of the entire design process for the system and each subsystem.

- Producibility - The producibility engineers ensure that the system can be safely built and maintained. During the system definition process, the producibility engineers examine the allocation of functional and physical requirements to each subsystem to make sure that the resulting system-level designs are producible. The producibility engineers ensure that the system meets all reliability and maintainability (R&M) requirements. Producibility engineers analyze the R&M requirements and perform top-level trade-offs to allocate these requirements to each subsystem. Producibility engineers work with each of the PDTs throughout all phases of the design to make sure that producibility issues are considered as the design matures.

- Sourcing - Sourcing personnel acquire the materials and purchased services needed to design and fabricate the system. During the system definition process, sourcing personnel develop the overall sourcing plan for the project. Sourcing personnel participate in the make/buy decisions for each subsystem during the system definition process.

- Logistics Support - Logistics support personnel ensure that the system is supportable throughout its entire life cycle. During the system definition process, logistic support personnel examine the supportability operational requirements and perform trade-offs to determine the supportability requirements for each subsystem. Logistic support personnel participate in the LCC analyses for the system.

- Cost Analysis - The cost analysts analyze the LCC throughout the design cycle. During the system definition process, the cost analysts support the trade-off studies by performing cost estimates for each candidate processing system. Development, unit production, and support costs are estimated for each processing subsystem in the candidate baseline system configurations. These costs are tracked and refined throughout the design process.

6.4.4 Design Reviews in System Definition Process

Periodic technical reviews are held to demonstrate that required accomplishments have been successfully completed before proceeding beyond critical events and key program milestones. There are two main design reviews during the system definition process: System Requirements Review (SRR) and System Design Review (SDR). In addition, the system members of the PDT participate in the three design reviews during the signal processor development activities: Architecture Design Review (ADR), Detailed Design Review (DDR), and Production Readiness Review (PRR). The plan for conducting each of these reviews is contained within the system engineering management plan (SEMP). Each major review should address the following issues:

- System engineering process outputs and traceability to customer needs, requirements and objectives
- Product and process risks
- Risk management approach
- Cost, schedule, performance, and risk trade-offs performed
- Critical parameters that are design cost drivers or have a significant impact on readiness, capability and LCC
- Trade studies conducted to balance requirements
- Confirmation that accomplishments in the systems engineering management schedule (SEMS) have been completed.

A multidisciplinary team from the government, contractor, and applicable subcontractors should be involved in the design review process. The two design reviews conducted during the system definition process are described in the following paragraphs.

System Requirements Review (SRR)
The objective of the SRR is to determine that the customer requirements have been properly analyzed and translated into system specific functional and performance requirements. Typically, the SSR is held after the

requirements have been analyzed, a preliminary functional analysis performed, and the requirements initially allocated to subsystems. During the SRR, we identify and quantify risks, and address approaches to manage these risks. We identify key technologies essential to system success and candidate technologies not viable for the system. We present the progress of on-going technical verifications, the preliminary functional analysis of the system, and the draft allocation of requirements to subsystems. The draft system specification is reviewed. During the SSR, we establish understanding of the customer requirements by identifying a complete functional baseline.

System Design Review (SDR)
The objective of the SDR is to ensure that the process used to determine the functional and performance requirements for the system is complete. We accomplish this by addressing the primary product and processes, by demonstrating a balanced and integrated approach to the development of the functional and performance requirements, by reviewing the audit trail established from the customer requirements, and by substantiating any requirement changes made since the SRR. The SDR is held after all system-level trade-offs have been performed, which resulted in an optimum allocation of requirements and functions to each subsystem.

During the SDR, we verify the performance, availability, and design suitability for critical products and process technology. We assess the adequacy, completeness and achievability of proposed system functional and performance requirements and present evidence to confirm that the system requirements can be met by the proposed system. The functional baseline for the system is established. We identify the draft specification tree, work breakdown structure, and risk handling approach for the next phase of the program. Pre-planned product and process improvements and acquisition strategies are presented.

RASSP Methodology Application Note

7.0 Architecture Design Process Detailed Description

The architecture definition process for RASSP, shown in Figure 7 - 1, is composed of three steps: functional design, architecture selection, and architecture verification. The process strives to provide a comprehensive hardware/software codesign capability, where

1. hardware and software are partitioned using interactive trade-off analyses
2. the partitioned software is verified (functionality and performance) using simulation before verification on the final target hardware.

Likewise, hardware functions are verified via simulation before detailed hardware design. An iterative, hierarchical simulation process is used to perform this verification at several levels of complexity.

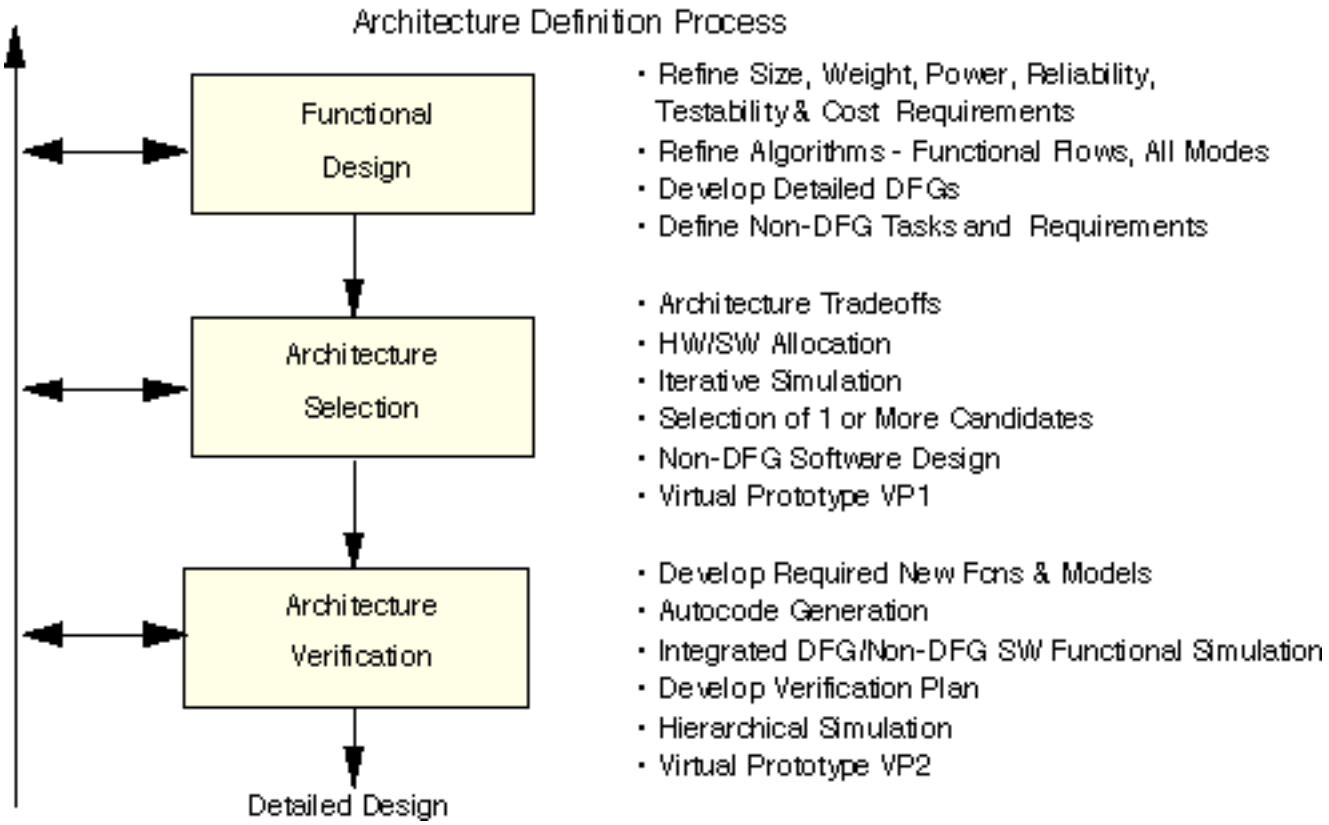


Figure 7 - 1: Architecture Definition Proces

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 hardware/software 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. For complex control applications,

these simulations can be coupled to ensure that all control is properly executed and results in the proper graph actions (e.g. mode transitions).

During architecture selection, we evaluate various candidate architectures through iterative performance simulation and optimize them to appropriate levels of detail. A trade-off analysis based on the established selection criteria results in the specification of the detailed architecture, and software partitioning and mapping. As part of the trade-off analysis, we use information from as many disciplines as possible (either manually or through design advisors) to populate the trade-off matrix. This portion of the process is heavily dependent on the reuse of architectural (hardware and software) components to provide significant time-to-market improvements. In addition, during architecture selection, we design all software not represented by the DFGs. Based on the requirements, the non-DFG software may include BIT, downloading, and diagnostics. The virtual prototype, VP1, produced during architecture selection is not a full system prototype, since function and performance are simulated independently and may or may not be coupled with the overall control mechanism.

During architecture verification, we develop the next level of detail for one or more architectures that meet the requirements and satisfy the established selection criteria. We develop and validate any required library elements (either hardware or software) at this time if they are not completed concurrently. Autocode generation is performed for the DFG based software, and the non-DFG based software (particularly the command program) is developed. The next level of performance simulation should include timing estimates for the generated code, along with a representation of the operating system services, scheduling, and run-time system overhead. We generate a hierarchical validation plan that ensures that all component interfaces are tested. Detailed hierarchical simulations are performed to verify both functionality and performance on the target architecture. The virtual prototype, VP2, produced during architecture verification, represents a functional and performance description of the overall design.

The arrow on the left side of Figure 7 - 1 indicates that the process has feedback within the architecture definition process, between the architecture and systems processes and between detailed design and the architecture processes. In fact, as shown in Figure 4 - 2, various portions of a design may be at different levels of maturity, which implies that more than one of the processes may be active concurrently. Note that as a design progresses, new development activities (mini-spirals) may be initiated. For example, during functional design it may be perfectly obvious to the designer that:

1. custom hardware is required for some portion of the processing
2. the reuse library does not contain all the software primitives required to construct the detailed DFG

In either case, the architecture selection process can proceed by postulating hardware or software elements that can be used in the high-level architecture performance simulations (e.g., defining a new element with 2X performance of existing element). Concurrently, we can initiate new activities to start the hardware modeling effort required for the custom hardware or start the development and validation process for a new software library element.

A hardware and software component reuse library has models and data at various levels, as shown in Table 7 - 1. 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.

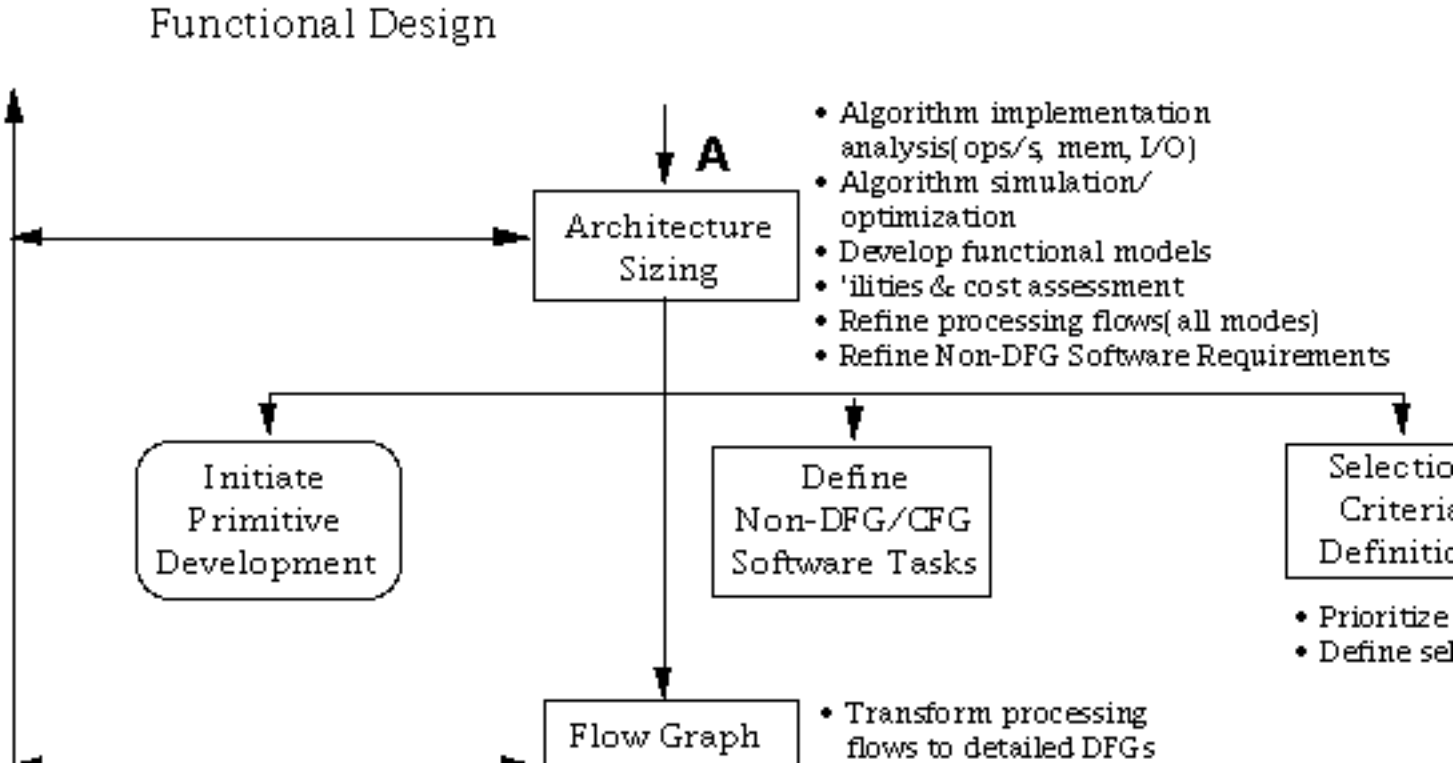
Software Reuse Library	Hardware Reuse Library
• SW performance	models • Performance models
• Application code/code fragments	• Behavioral models
• OS kernel(s)/OS services	• DFG partitions and mappings
• Application DFGs	• Architecture configurations
• Control/support software	• Test plans and test sets
• Test data	• Documentation elements
• Documentation elements	

Table 7 - 1: Hardware and Software Reuse Library

The following sections describe the steps in the architecture definition process.

7.1 Functional Design

Initial efforts in architecture design include an implementation analysis of the algorithms to assess the required operations per second and memory and I/O requirements. Processing flows may be optimized based upon implementation experience and knowledge of reuse libraries. The two primary functions in functional design are formalizing the criteria to select an architecture and translating the processing flows to an architecture-independent DFG constructed from reusable library elements. The functional design process is also an opportunity for the signal processing architect(s) to assess the complexity of the required processing. The goal is to take the functional algorithms and translate these into preliminary implementation form. Initially, we size and establish a criteria to select the architecture, as shown in Figure 7 - 2. We translate processing flows for all modes to architecture-independent DFGs constructed from reusable library elements, which may represent either hardware or software. In addition, we translate control requirements into the appropriate control flows.



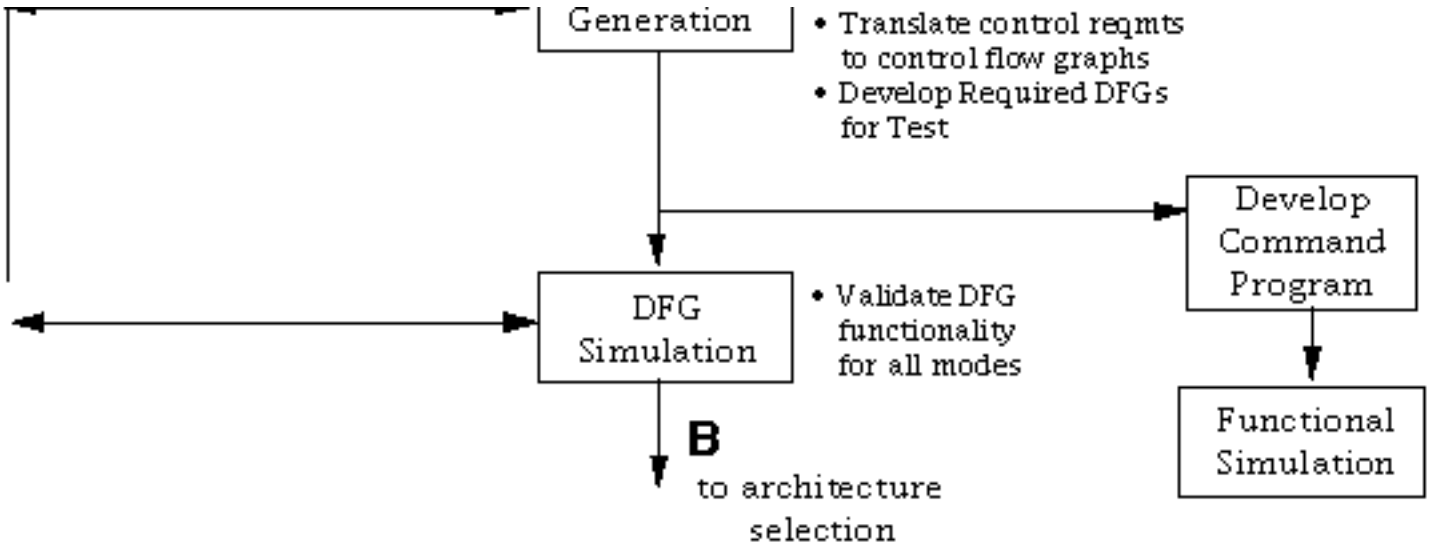


Figure 7 - 2: Functional Design Process.

nputs to this portion of the process are the outputs of the system design activity. These include both signal processing requirements and physical requirements. The signal processing requirements include the algorithm flows for all modes of operation, the control flows (if any) used to control the initiation of processing modes and the transition between modes, and performance timelines which must be met. The physical and/or programmatic requirements include the size, weight, power, reliability, testability, cost, and schedule. Outputs from this portion of the process include prioritization of the implementation requirements, the library-based DFGs representing the processing for each operational mode, software requirements for non-DFG processing (e.g., mode switching), and definition of the criteria and weighting used to select the architecture. Note that the selection criteria may be just as sensitive to cost and schedule as it is to performance.

7.1.1 Architecture Sizing

We analyze the system requirements and processing flows for all required modes in terms of estimated operations per second, memory requirements, and I/O bandwidths processing requirements. We can make initial size, weight, power, and cost assessments based upon rules of thumb and experience sizing estimates. We also develop a first-pass partitioning of hardware and software functionality at this point. We develop requirements for non-DFG/CFG software. Functional models may be developed where necessary and algorithm simulations and optimization performed to refine the processing flows and derive the detailed requirements for the signal processing. The resulting functional processing flows represent the detailed algorithms that must be performed for each required mode.

7.1.2 Selection Criteria Definition

We prioritize the overall system requirements and the derived requirements and establish a selection criteria. The selection criteria provides the necessary basis for subsequent architecture trade-off analysis. An example of typical parameters used in processing trade-offs studies is shown in Table 7 - 2.

Architecture/Hardware	**Software**
Performance	Compilers (availability/efficiency)
Peak	Operating system/support SW
Sustained	functionality
Input/Output Bandwidth	performance
Intraboard	availability/maturity
Board-to-board	Programming tools/environment
Chassis-to-chassis	

I/O pin requirements	**Costs**
Memory (size/performance)	Non-recurring (design, DFT, development, capital)
Cache	Production
Local	Test Cost
Global	Life cycle
Power dissipation	
Static	
Dynamic	**Risk**
Testability	Technical
Fault Coverage	Schedule
Test Time	Cost
BIST Support	
Mechanical packaging complexity	**Quality**
Thermal packaging issues	Defect level
Scalability/Upgradeability	Reliability

Table 7 - 2: Typical processor trade-off criteria

We define a trade-off matrix to formalize the selection criteria for the architecture, as shown in Figure 7 - 3, which contains the top-level requirements allocated to the signal processor. Satisfying these requirements drives the hardware/software codesign of an architecture. We populate the matrix and iteratively update it as any given design progresses. Early in the process, the entries are less accurate than later on. The goal is to eliminate some designs early while carrying the best candidates to subsequent levels of detail. The entire Integrated Product Development Team (IPDT) has rapid access to the ongoing trade studies and participates in populating the trade matrix through various tools and design advisors. Once we have established the criteria, we develop its relative weighting for the particular application to ensure that the proper emphasis between performance, cost, schedule, and risk is reflected in the architecture selection. We use these weighted criteria to drive both the architecture selection and the level of optimization and effort that is applied during this portion of the design. The exact content of the trade-off matrix and the maximum scores associated with different attributes is project dependent.

Architecture Tradeoff Matrices									
Architecture Attributes									
	Size	**Weight**	**Power**	**Schedule**	**Testability**	**Cost**	**Reliability**	**• •**	**T Sc**
Arch # 1									
Arch # 2									
•									
•									
Max Score	0-5	0-25	0-15	0-5	0-15	0-10	0-15	0-10	0-
Architecture Scores									
	Size	**Weight**	**Power**	**Schedule**	**Testability**	**Cost**	**Reliability**	**• •**	**T Sc**
Arch # 1									
Arch # 2									
•									
•									
Max	0-5	0-25	0-15	0-5	0-15	0-10	0-15	0-	0-

Score								10

Figure 7 - 3: Architecture selection criteria

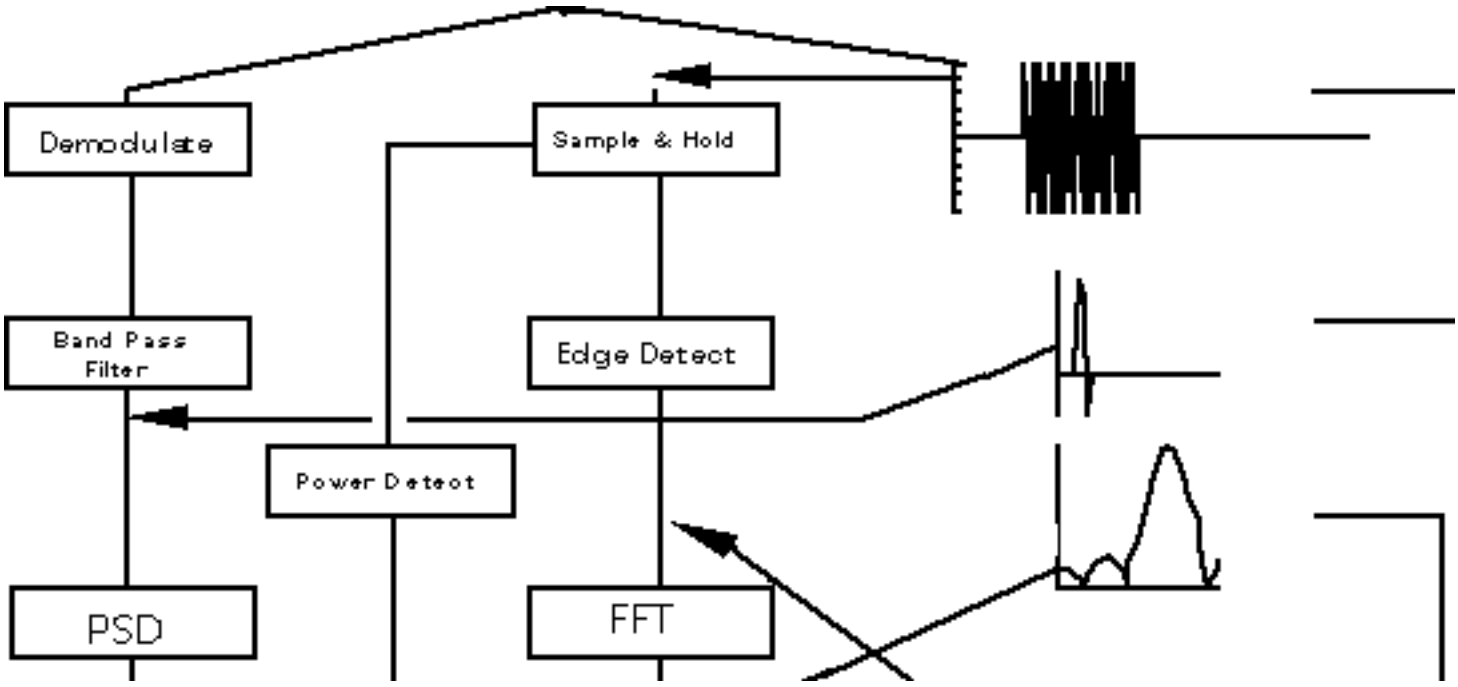
7.1.3 Define Non-DFG/CFG Software Tasks

We review the non-DFG/CFG software requirements and define the tasks required to fulfill these requirements.

7.1.4 Flow Graph Generation

We transform the finalized algorithm processing flows into the detailed DFGs as the first step in hardware/software codesign. In parallel, we optionally develop any DFGs which are specifically for test. These DFGs are based upon either the Processing Graph Method (PGM) developed by the Navy or the GEDAE™ method and tool developed by Lockheed Martin ATL. PGM is a specification for defining detailed DFGs for signal processing applications. It is supported by older, non-graphical development tools. GEDAE™ is a newer, graphically oriented tool environment that is also integrated with a command program generation tool and a performance modeling tool. Either of these will work within the process however, GEDAE™ has become the tool environment of choice due its ease of use and modern programming environment. The DFGs are made up of the reusable library elements, which may represent either hardware or software. The resulting DFGs represent the composite algorithmic requirements for all processing modes. The algorithms to date have been mostly functional in nature, without regard to specific implementation issues such as processing efficiency. For example, the DFG may include a filter. Resultant efficiency will be related to whether a time-domain or frequency-domain implementation is selected, which may in turn depend upon the processor under consideration. Note that if several architectures are considered, unique sets of architecture-specific implementations may be chosen.

The DFGs are the basis for both the architecture synthesis, the detailed software generation, and potentially custom processor synthesis. As indicated in Figure 7 - 4, validating the DFG is an important step that ensures consistency with the simulatable requirements passed down from the systems process. The left side of the figure represents a processing flow that has been simulated during systems definition to establish the baseline algorithm set for the application. If multiple processing modes are required, there would exist a processing flow for each mode. The right side of the figure represents the detailed PGM DFG constructed from reuse library elements.



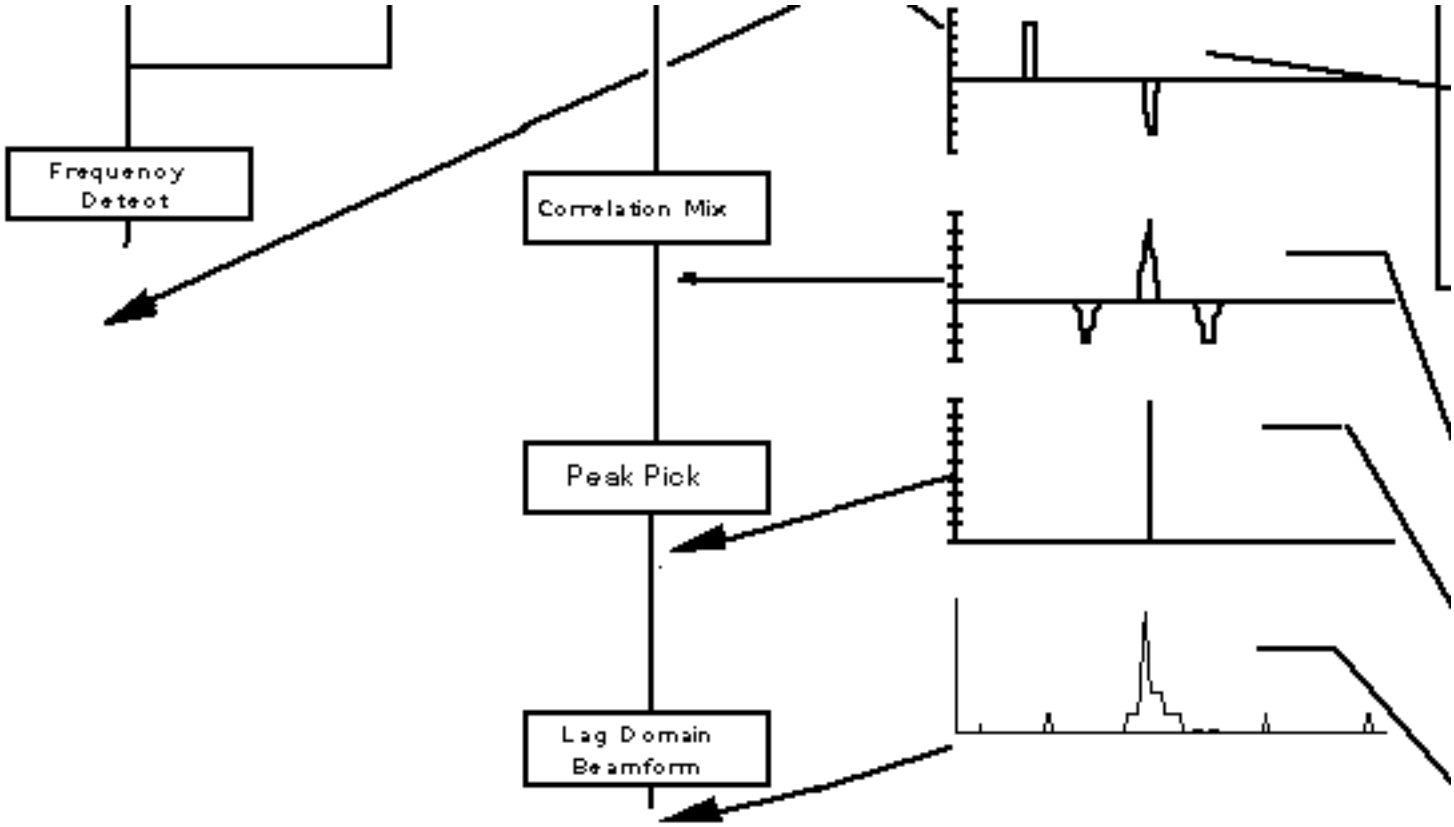


Figure 7 - 4: Correspondence of processing flows and domain-primitive graph

The examples used throughout the rest of this section will use PGM graphs as their basis. More information regarding GEDAE™ data flow graphs may be found in the Hardware/Software Codesign application note, the Data Flow Graph Design applicaton note, or the GEDAE™ web page at http://www.gedae.com.

Each detailed DFG is simulated to provide data for comparison with the algorithmic flows developed during the systems process. As part of this simulation process, the DFG may require modification until the correct functional results are achieved. Validating these two representations ensures that the simulatable functional requirements are captured in the DFGs that will drive subsequent codesign.

In some cases a suitable library component may not exist, which means designing either a hardware or software element. For software, this is referred to as a prototype element. The prototype element will permit the hardware/software codesign to proceed before validating the element for permanent inclusion in the reuse library. For hardware, a similar prototype is established by generating a high-level performance model (see the Token-Based Performance Modeling application note) of the desired component. In either case, the architecture process can proceed using estimates for these elements.

In addition to the DFG generation, the control flow requirements are transformed into the control flow graphs (CFGs)(see the Autocoding for DSP Control application note) required to manipulate the DFGs according to a defined set of rules. This DFG control is referred to as command processing. Command processing is needed to incorporate signal processing into the overall system design. The signal processing functionality needs to be controlled in accordance with the other functionality dictated by the system design. The mechanism that exercises this control over the signal processing subsystem is the command program. The signal processing subsystem receives messages from the external environment and commands the signal processors to perform their detailed function. The command program performs the control operations of the signal processor subsystem. While DFGs describe the functionality associated with the signal processing subsystem, the command processing function is best described by an object-oriented methodology. The design of the two are

interrelated, since design of the DFGs must provide access to or the ability to set graph parameters as dictated by the command program. Consequently these designs must be performed concurrently.

When we develop the signal processor code using the PGM or GEDAE™, a set of operators are needed to control the resultant functionality. The signal processing graphs and their data structures are objects manipulated by the command processing program. The operations upon these objects have to be executed in both realtime and non-realtime. For example, we can initialize the primitive node and the resultant data structures in non-realtime. However, operations such as starting the I/O operation, disabling queues, mode changing, changing dynamic parameters, etc. are all realtime operations and are often specified by commands external to the DFG operation. The command program facilitates interface of the signal processor subsystem to the rest of the overall system. As part of the functional design, the detailed specification of these non-DFG processing requirements is completed. Conceptually, the command program manipulates objects. The objects are the DFGs and their data structures. The top-level requirements define a set of actions that have to be performed by these objects. The designer must specify the actions associated with each of the states of these objects, both for the normal and abnormal conditions. This information will be contained in a state model. Associated with all the states is a process model that expresses the procedures to be executed at each state.

7.1.5 Command Program Development

We transform the state and process models via autocode generation into prototype code that will be used with the DFGs to simulate the interactions among graphs, particularly mode transitions. 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. We can develop the command program either through standard software development CASE tools or through the tools that provide autocode generation capability from state transition diagram descriptions. See the Autocoding for DSP Control application note for more details regarding the process and tools for developing the command program.

7.1.6 Functional Simulation

As part of the functional design process, we must simulate both the DFGs and the CFGs. We simulate the DFGs to validate that they represent the same processing as the mathematical description of the processing flows developed during the systems process. We also have to validate various aspects of the CFGs, which must interact with the DFGs. This includes passing parameter information between the external world and the graph management software, initiating or terminating I/O devices, starting and stopping DFGs, etc. Individual DFGs have been simulated previously and their functionality is not in question. However, it is desirable to simulate the interaction of the various DFGs, as dictated by the CFGs, to ensure that all interactions are functionally correct. In particular, it is necessary to know that mode transitions occur properly.

7.2 Architecture Selection

Architecture selection is an automation-aided process to rapidly evaluate different architectural designs and instantiations of these designs. For example, an existing architecture may be simulated with new DSPs to quickly evaluate inserting emerging COTS technology. This is the Model Year upgrade concept. An integrated toolset will facilitate rapid performance trade-offs to select and size a scalable architecture based upon the processing requirements. These trade-offs are tightly coupled with the performance of the software on the architecture. Automated multiprocessor partitioning and mapping, coupled with user intervention, provides a rapid optimization capability. The trade-off process supports the IPDT concept by integrating tools for DFT, cost analysis, and high-level design advisors. In addition, the architecture process is coupled through VHDL to the detailed synthesis of chips, boards or processors.

The architecture selection process, shown in Figure 7 - 5, represents the heart of the RASSP hardware/software codesign, which uses a library-based, DFG-driven approach to software development combined with iterative performance trade-off analysis to support rapid selection/analysis of candidate architectures. During architecture selection, various architectures are offered as candidates that are selectively optimized. For each candidate, the process includes the following steps:

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

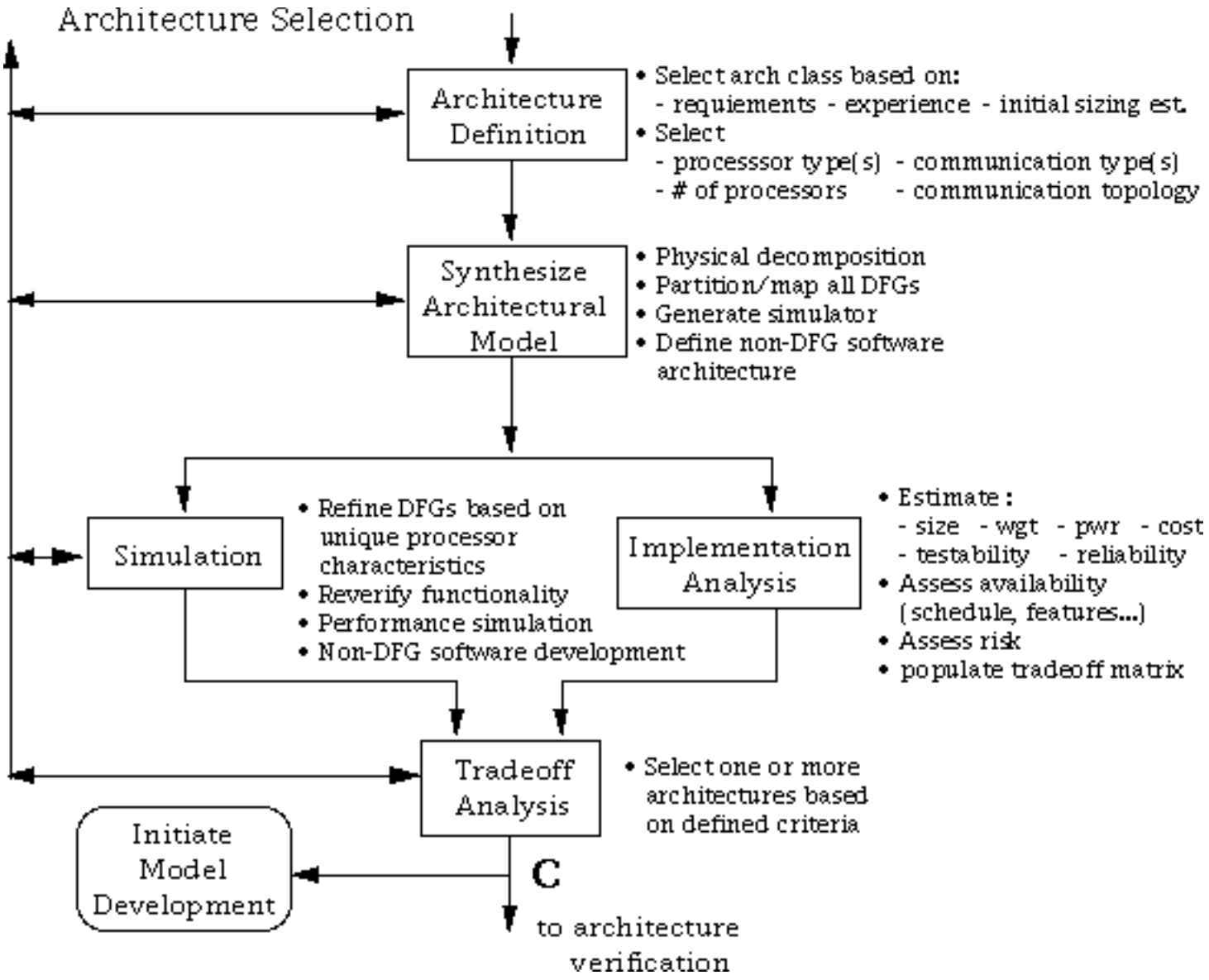


Figure 7 - 5: RASSP Architecture Selection Process

We use a trade-off analysis based on the established criteria to select the detailed candidate architecture and software. Architecture selection is iterative and intended to produce one or more architectures that meet the overall requirements. Ideally, we evaluate candidate architectures that span the design space.

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 hardware/software reuse library.

Outputs from the architecture selection process are the finalized DFGs and one or more architecture instantiations that we selected for more detailed functional and performance verification. Also output from the

architecture selection process is the description of the DFG partitioning and mapping to the processors of the selected architecture(s) for all processing modes.

The steps involved in architecture selection shown in Figure 7 - 5 are discussed below.

7.2.1 Architecture Definition

The next step in the hardware/software codesign process is specifying an architecture. This includes both selecting a class of architectures (e.g. MIMD, SIMD, etc.) and the design approach within the class (e.g. interconnect topology). This decision is generally based upon a combination of the signal processing architects application domain experience and the system requirements, cost, availability, technology maturity, etc. The architecture should include any processor(s) necessary to satisfy the command program requirements.

Given the DFG (or set of DFGs) that describes the processing, the architect must postulate one or more designs that may satisfy the requirements. These architecture choices are based upon the domain experience of the design team. One of the goals of RASSP is to facilitate the ability to define and evaluate more alternatives than would otherwise be possible. This is done through the use of the integration and semi-automation of the architecture tools that assist the DSP design architect. It is important to note that the processing represented by the DFGs at this point has not been allocated to either hardware or software. However, as the architect defines an architecture for consideration, one or more nodes of the DFG may be allocated to custom hardware or to a postulated processor that currently does not exist. Figure 7 - 6 shows two simple alternative architectures. In Arch #2, one or more nodes of the DFG have been allocated to either an existing or postulated special-purpose ASIC. In either case, the architect made a decision regarding hardware/software allocation. In Arch #1, all processing is performed in software on one of the four processors, while in Arch #2 some functions are constrained to the ASIC, while all others are allocated to software on one of the two processors. These architectures illustrate the point. In reality, the RASSP methodology must be capable of dealing with architectures that range from simple cases (such as those illustrated) to multi-chassis configurations of hundreds of processors. Architectures can be constructed from existing entities ranging from single processors to boards.

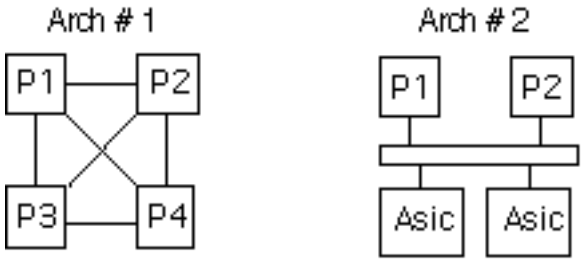


Figure 7 - 6: Example Candidate Architectures

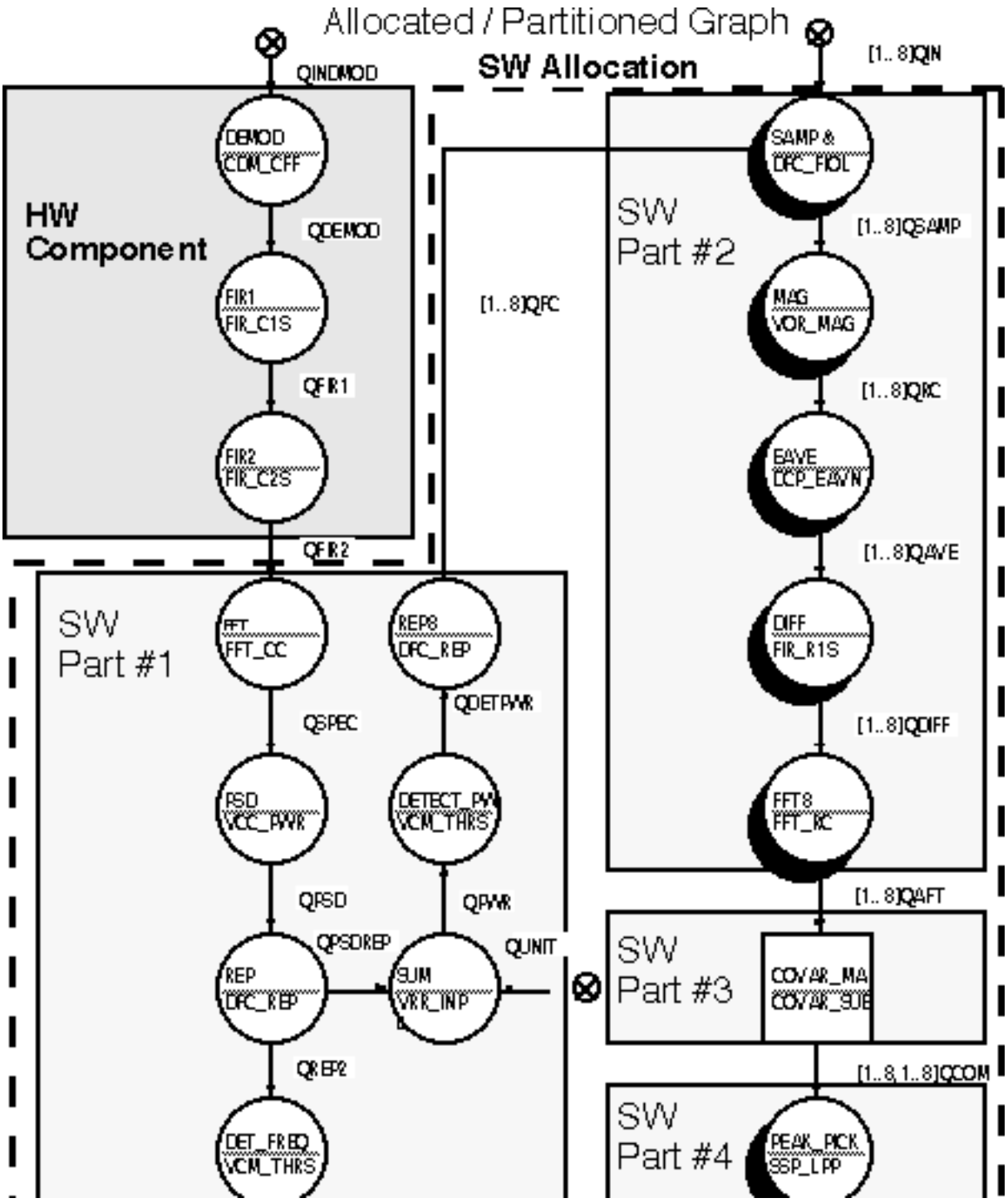
For the selected architecture type, we select specific processor type(s) (e.g., ™S320C40, i860, AD21060, etc.) and number of processors, along with a desired communication mechanism (e.g., bus, Xbar switch, etc.). We choose the number of processors based upon prior estimates of performance and/or validated benchmark data for library code fragments, and knowledge of the system requirements. We can perform a preliminary physical decomposition and do a high level study of routability, thermal issues, etc.

7.2.2 Architecture Model Synthesis & Physical Decomposition

The process of defining the architecture is coupled with the allocation of the DFG to the architectural elements. For example, specific nodes in the DFG may be assigned to a custom hardware element such as an FFT chip. To support this capability, the library would contain a hardware model capable of performing the processing defined by the DFG node.

The portion of the DFG(s) allocated to software is partitioned and mapped to the available processors of the candidate architecture under consideration. The software partitions are defined by mapping the primitives in

the DFG to the DSPs in the architecture. Figure 7 - 7 shows a DFG in which two portions of the DFG are allocated to hardware and the remainder of the DFG allocated to software grouped into four partitions. This activity is supported by multiple, automated partitioning/mapping algorithms for graph assignment and a manual capability. This process may utilize what -if experimentation to evaluate postulated architecture components, as well as software performance estimates for prototype primitives. In addition we postulate architectures for the non-DFG software. We construct a VHDL performance model for the architecture. It may be obvious that special-purpose hardware or a custom processor is required to meet the overall signal processor requirements. If so, we can start a mini-spiral development activity to embark on the required development. The performance of the new hardware may be estimated so that the overall architecture selection may proceed. As models are developed for the new processor, library models may be updated with new timing information and the architecture reevaluated.



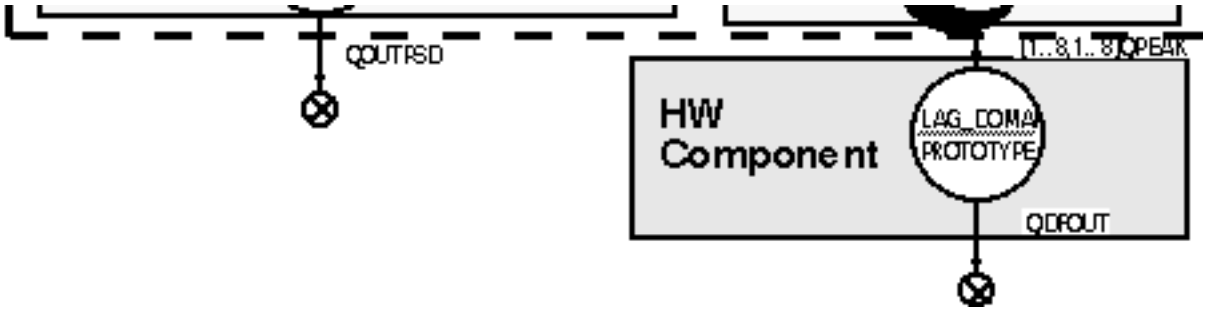


Figure 7 - 7: Graph showing hardware/software allocation and software partitioning

7.2.3 Performance Simulation

Based upon unique characteristics of the individual processors, we can refine algorithms to optimize performance that is verified again via simulation. We also begin high-level language non-DFG software development and simulation. The goal of simulation at this stage is to both verify the algorithms again functionally (if modifications have occurred), and to refine the anticipated performance of the candidate architecture using available throughput, memory, and I/O estimates for these algorithms. We estimate the overall architecture performance via VHDL simulation and analyze it with respect to meeting all signal processing requirements. The performance simulation reflects the established partitioning and mapping of the DFGs. We simulate a particular partitioning and mapping to estimate the overall timeline for the processing. These performance simulations are executed iteratively as we consider different partitioning and mappings in an attempt to optimize the execution timeline. The architecture may be optimized through iteration of architecture synthesis and simulation. After simulation, we can modify the architecture by changing or adding processors, changing communication types, changing interconnect topology or postulating new architectural elements (what-if analysis) that require subsequent development.

Figure 7 - 8 shows example timelines for the two architectures in Figure 7 - 6 for the 6 - node graph in the figure. The 6 - node graph is a simplified representation in which n1 and n6 represent the graph partitions allocated to hardware in Figure 7 - 6, and n2, n3, n4, and n5 represent the software partitions. Timeline (a) represents the mapping of the six graph nodes to the four processors shown in Arch #1 of Figure 7 - 6 and timeline (b) represents the mapping of the graph to Arch #2, which contains two-special purpose ASICs. The example illustrates a much improved overall timeline for the architecture when n1 and n6 are mapped to custom ASICs, all other processing being the same. The importance of the example is to illustrate that different architectures and graph mappings to those architectures can be evaluated quickly, which is especially valuable in large systems.

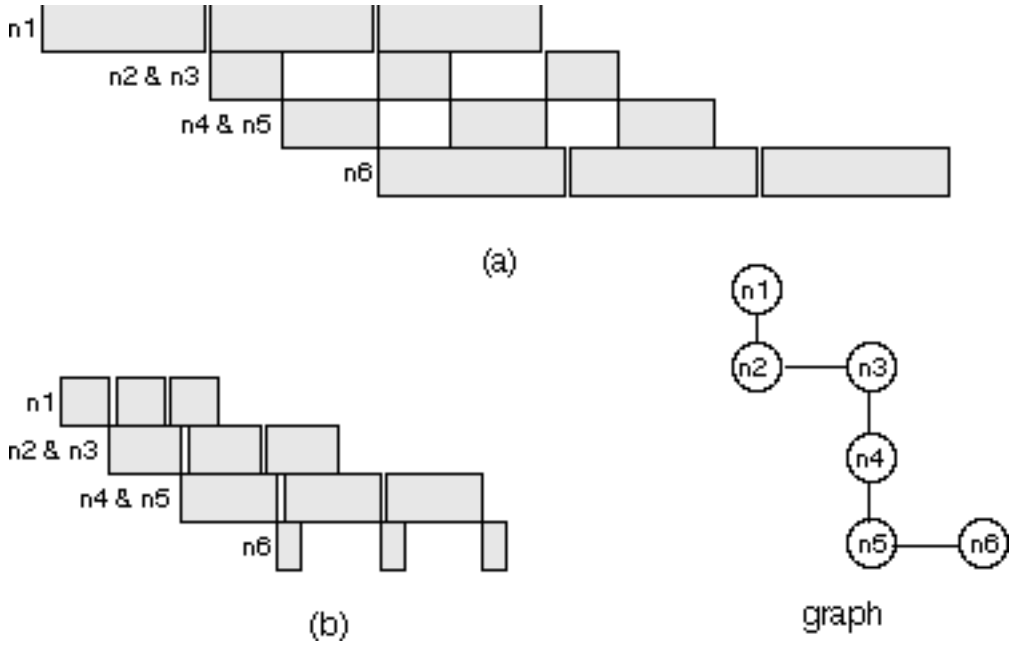


Figure 7 - 8: Timeline for Two Postulated Architectures

7.2.4 Implementation Analysis

Concurrent with the simulation effort to establish performance, we can proceed with implementation analysis of the architecture, if desired (depending on the degree of satisfaction with the architecture). Any given architecture, or multiple candidate architectures, may be in the process of analysis by various members of the PDT (see Section 7.5.3). For each candidate architecture, we postulate an implementation (using high-level synthesis tools and/or design advisors driven from a VHDL description of the architecture) and transform it into size, weight, power, throughput, and cost parameters, as well as schedule, testability, reliability, availability, and maintainability estimates. This process requires the collaboration of the IPDT to properly assess the attributes of the architecture. In addition, we assess component availability with respect to supporting the desired development schedule. The concurrent use of tools and design advisors speeds the process and leads to better-informed, early decisions. Based upon this analysis, we can assess implementation risk for the different architectures.

7.2.5 Trade-off Analysis

We iterate the architecture synthesis, simulation and detailed analysis process for each candidate architecture to obtain an optimized solution. These activities are directed toward populating an architecture trade-off matrix that is a record of the design process. The trade-off matrix is supported by design notes that document the rationale for the entries in the matrix. We analyze the information gathered from the architecture synthesis process with respect to the selection criteria and weighting established in the functional design process. Based on the selection criteria, we select one or more of the candidate architectures for further evaluation.

At this point of the design, it may become obvious to the design team that a custom processor or special-purpose ASIC is required to meet the program requirements. If custom hardware is required for a viable solution to the requirements, we can define and support what-if elements in the library and assign processing times associated with them for the DFG nodes. We can evaluate the new element; if satisfactory performance is achieved, we can start the appropriate hardware model development in parallel with the architecture selection process. Or, if desired, we can invoke a processor synthesis tool that is driven from the DFG primitives. The synthesis tool outputs VHDL compatible with down stream detailed design tools.

In this iterative process, we can modify the partitioning, mapping, and the architecture as part of the optimization process. Tools support interactive design changes to both the architecture and/or the partitioning

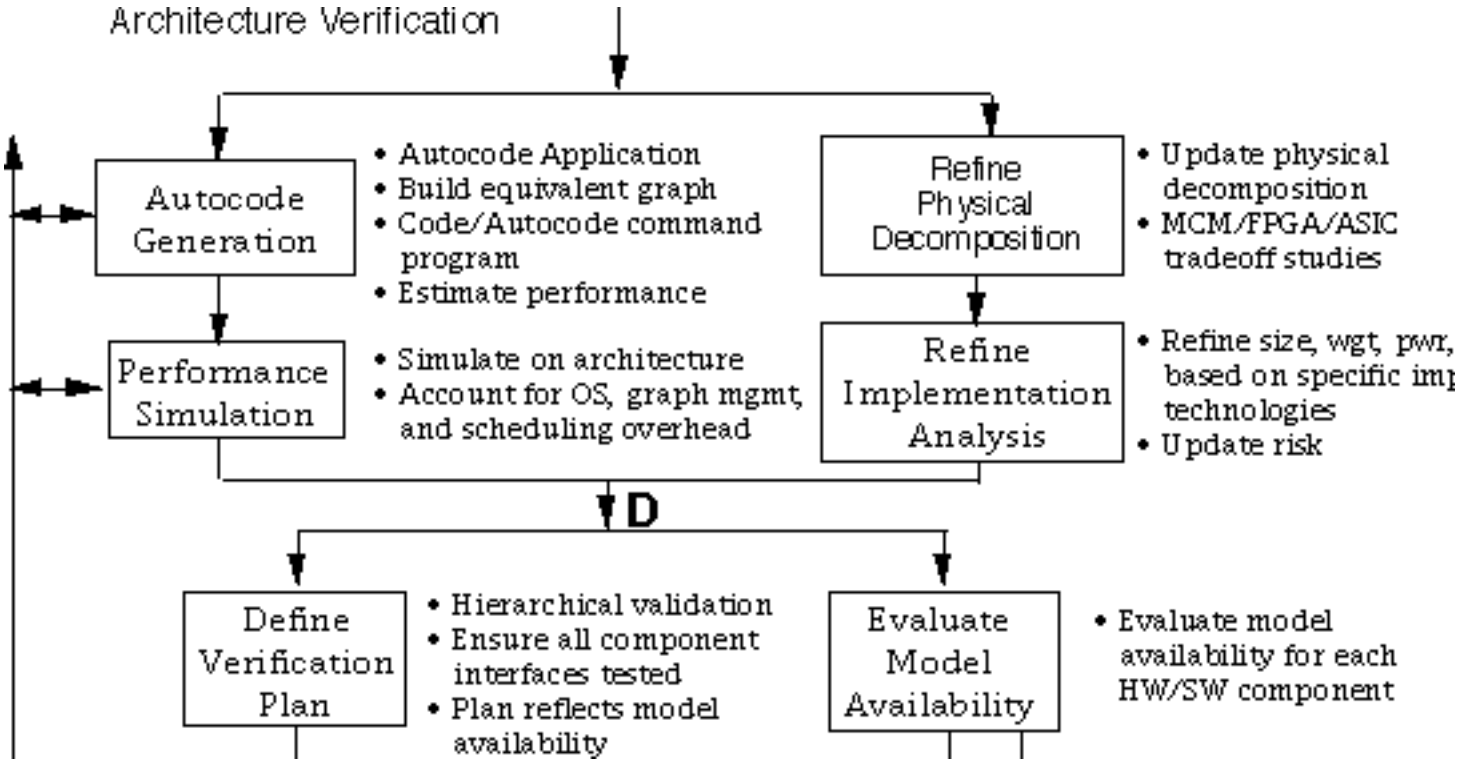
and mapping. Of particular interest is the ability for designers to quickly modify the processor interconnect topology for large architectures. The candidate architectures are each evaluated with respect to the established selection criteria, and then we may select one or more architectures for further consideration.

7.3 Architecture Verification

Architecture verification is the process of hierarchically simulating both the functionality and performance of a selected architecture candidate. Here, simulations are performed at a greater degree of detail than during architecture selection. An integrated framework supports mixed-domain simulation so that high-level performance and functional simulation can be coupled with ISA or RTL VHDL simulators, hardware emulators or hardware testbeds. The goal of the verification process is to validate operation of all architectural entities and the interfaces between them before detailed design. The specific verifications required depend upon whether the design is all COTS, mixed COTS and new hardware, or all custom hardware. 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 built 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 software description, and the hardware/software reuse library. Note that overall functionality has not been verified before this point. The role of architecture verification is to verify functionality and more detailed performance of the candidate implementation using a combination of testbed hardware, simulator(s), and/or emulator(s) before detailed hardware implementation. This is the first real step in hardware/software codesign verification and is important to verify the virtual prototype defined to date. This process can iterate with the architecture selection process to 1) optimize the selected architecture and 2) update model performance in the RASSP library. As shown in Figure 7 - 9, the major steps in this process are autocode generation, performance simulation, reassessment of architecture attributes, new element development, component mix evaluation, definition of a verification approach, simulation development, and performance and functionality verification.

Outputs from architecture verification include new library elements, detailed specifications for hardware development, and performance and functionality verification.



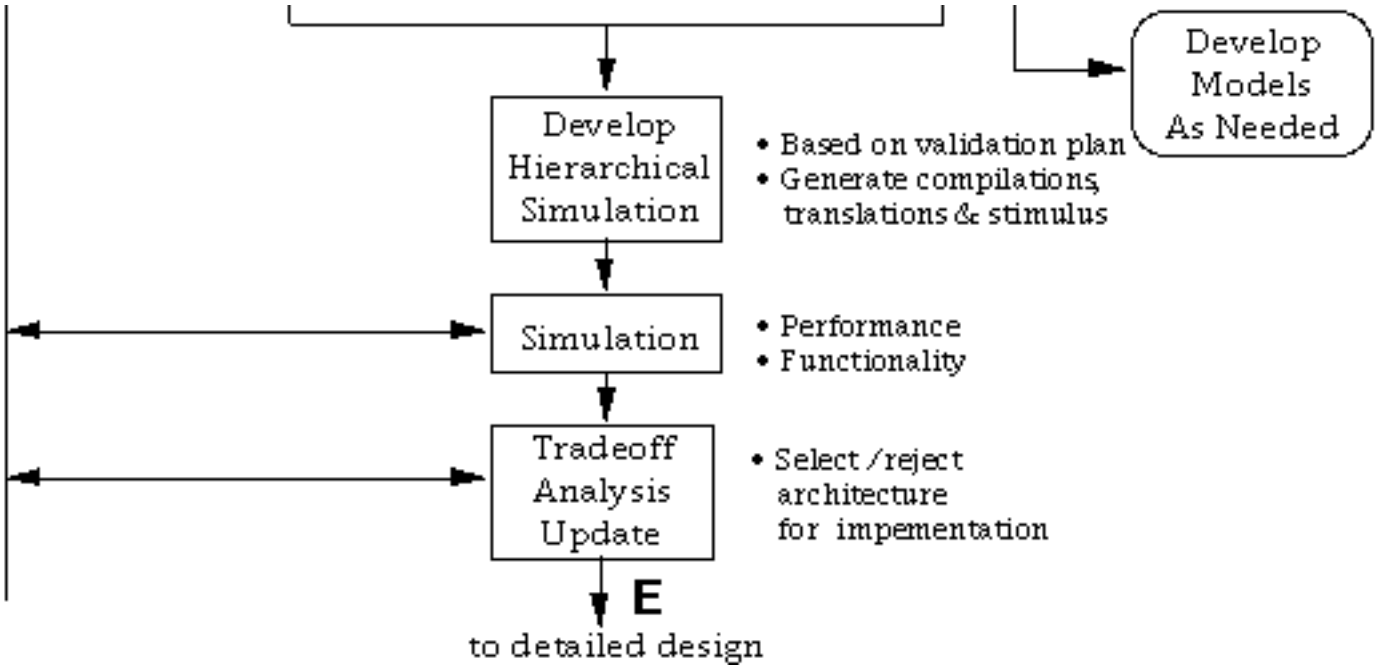


Figure 7 - 9: Architecture Verification Process

7.3.1 Autocode Generation

The architecture(s) produced in the architecture selection process, the finalized DFGs, and the partitioning/mapping data, provide the necessary input to autocode generation. We use the architecture description and the partitioning and mapping data to automatically generate the software for each of the partitions. We generate the code by translating the processor-independent flow graph primitives to target-specific code which use the optimized math libraries for the specific DSP. The result of this process is definition of a new DFG node (called an equivalent node), which represents the entire graph partition. As part of this procedure, we simulate the resultant code to ensure that the resulting node satisfies the same test bench as the original subgraph. The resulting equivalent nodes from all the graph partitions can be combined into a new equivalent graph that has the same characteristics as the original, but each node represents a software module representing the aggregate processing of a subgraph. We can verify the functionality of this equivalent graph again via functional simulation to maximize confidence in the overall translation process.

7.3.2 Performance Simulation

We generate timing estimates for the autocode-generated software, and performance simulation can be repeated using the new timing estimates. Ideally this simulation is at a greater level of detail than previous simulations. 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 necessary. The equivalent DFG(s) performance is simulated on the candidate architecture, much like was done in architecture selection, but with a higher degree of fidelity.

Figure 7 - 10 shows the 6-node graph replaced by a 4 - node equivalent graph in which the nodes are aggregated as dictated by the partitioning and mapping. The performance simulation at this level accounts for communication protocol so that detailed network contention is evaluated. If necessary, we can modify the partitioning, the architecture, and/or the DFG until performance is satisfactory. Modifications, of course, require iterating through the process steps again.

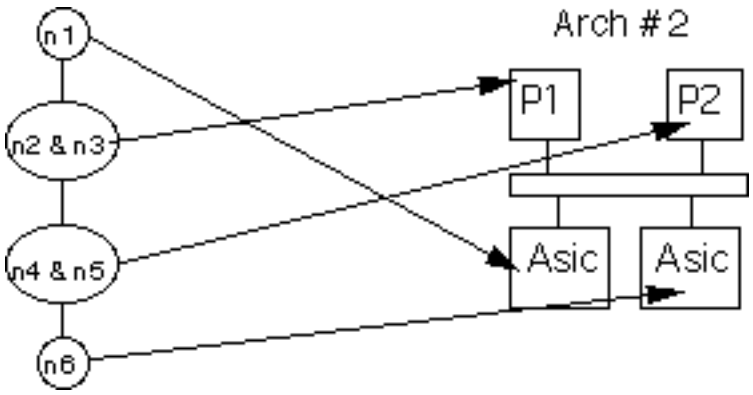


Figure 7 - 10: Mapping of equivalent graph to Arch #2

7.3.3 Refine Physical Decomposition

We continue to refine the physical decomposition and implementation of the design. We perform MCM/FPGA/ASIC trade-off studies. We update the routability, thermal, and other analyses based on more detailed information.

7.3.4 Refine Implementation Analyses

During architecture selection, we produced the initial estimates for size, weight, power, cost, schedule, testability, maintainability, etc.. We used this information to select one or more candidate architectures for further evaluation. After initial selection of candidates, we can perform a more detailed evaluation of the architecture attributes. This may include reiteration of size, weight, power, and cost estimates based upon particular implementation technologies, such as MCMs, surface-mount technology, etc. This activity involves various disciplines from the development team. The architecture description of viable candidates may again be processed using hardware synthesis tools and various design advisors to produce the updated architecture attributes. We can assess increased costs due to architecture modifications necessary to meet reliability requirements, since these could impact final architecture selection.

NOTE: It can be argued that these three steps (autocode generation, performance simulation, and refine implementation analysis) could be allocated either to the architecture selection or architecture verification process.

7.3.5 Model Availability

Architecture verification is the process of hierarchically simulating both the functionality and performance of a selected architecture. In general, we perform the simulations at more detailed levels than previously done. In preparation for architecture verification, we must determine the level of hardware and/or models available for all architecture components. This becomes the basis for defining a functional verification approach. For each architecture component, the hardware may exist and be available, or various levels of models may exist in the library, including behavioral models, ISA models/simulators, RTL models, and/or performance models. In the ideal case, all software components in the reuse library have performance data for each processor supported in the library. In reality, this performance data is most likely generated on an as-needed basis. Consequently, for each software component not supported by performance data for a required processor, we must evaluate the availability of hardware and/or various level models to support validation of that software component. The availability of these hardware components and/or models provides constraints on the verification plan.

7.3.6 Verification Approach Definition

An architecture verification plan should be developed that ensures, to the maximum extent possible, that all hardware components will function and interoperate as expected and all software will execute properly on the

architecture when built. This will likely require a hierarchical approach to the architecture verification. Virtual prototyping the entire architecture at the RTL-level requires an exorbitant amount of time. The plan must, however, ensure that all component interfaces are tested, all devices properly communicate with each other, and all software executes on the appropriate processor.

An integrated framework supports mixed-domain simulation so that high-level performance and functional simulation can be coupled with ISA or RTL simulators, hardware emulators or hardware testbeds. The goal of the verification process is to validate operation of all architectural entities and the interfaces between them before detailed design. The specific verifications required depend upon whether the design is all COTS, mixed COTS and new hardware, or all custom hardware.

7.3.7 Simulation Development

The objective of this process is to enable incremental functional and performance evaluation of hardware and simulation models throughout the design process. We do this by providing an integrated suite of tools that support a combination of testbed hardware, simulator(s), and/or emulator(s) to fully verify performance and code functionality before hardware implementation. The ideal approach is to integrate these disparate simulators through a multi-domain backplane that enables mixed-mode simulation. The multi-domain simulation capability should support the following elements:

- Testbed Hardware - The most critical element in enabling algorithms to be tested on target multiprocessor hardware is a robust distributed application environment. The environment should support user porting and analysis of algorithms on a single workstation and embedded multiprocessors. It will provide users with an application environment that supports multiprocessor mapping, instrumentation, and performance monitoring.

- Behavioral Simulation - Behavioral simulation will verify functionality and performance of new designs. The simulations will be VHDL-based, and will support a number of commercial simulators. The results of behavioral simulation will be used to used to update network simulation model performance.

- Architecture (Network Performance) simulation, which is defined here to be non-functional data-flow-level simulation, determines the performance of large systems. RASSP efforts are focusing on providing interoperable VHDL performance models. Interaction of this capability with behavioral simulation and hardware testbed capabilities provides a method to verify system-to-subsystem performance. The performance results can be used to update the architecture selection processor performance models.

Constructing the hierarchical simulation is based upon the availability of models supporting the entities in the architecture. In some cases, there may be available hardware testbeds for COTS processors, in which case using them is the most effective approach. In general, there may be models available at different levels and in different languages. While it is a goal on RASSP to use VHDL as a common language, we are taking a pragmatic approach to enable use of all available modeling technologies. If sufficient models have not been developed during the design process, we must create them at this point. Based upon model availability, we map the architecture to appropriate simulation engines. Illustrated in Figure 7 - 11 is one of the simulation mappings that could be run; the process envisions a set of hierarchical mappings to efficiently verify the entire hardware/software suite. The interoperability and integration of various commercial tools operating in different domains provides a mechanism to perform efficient simulations for verification. The ideal is to support the interoperability of commercial tools to simulate a complete system in a seamless fashion.

Element	HW Testbed	Perf	Full Func (ISA/behavioral)	Full Func (RTL)	Bus Func
P1	•	•	•		•
P2		•		•	
Asic 1		•	•		
Asic 2		•	•	•	•

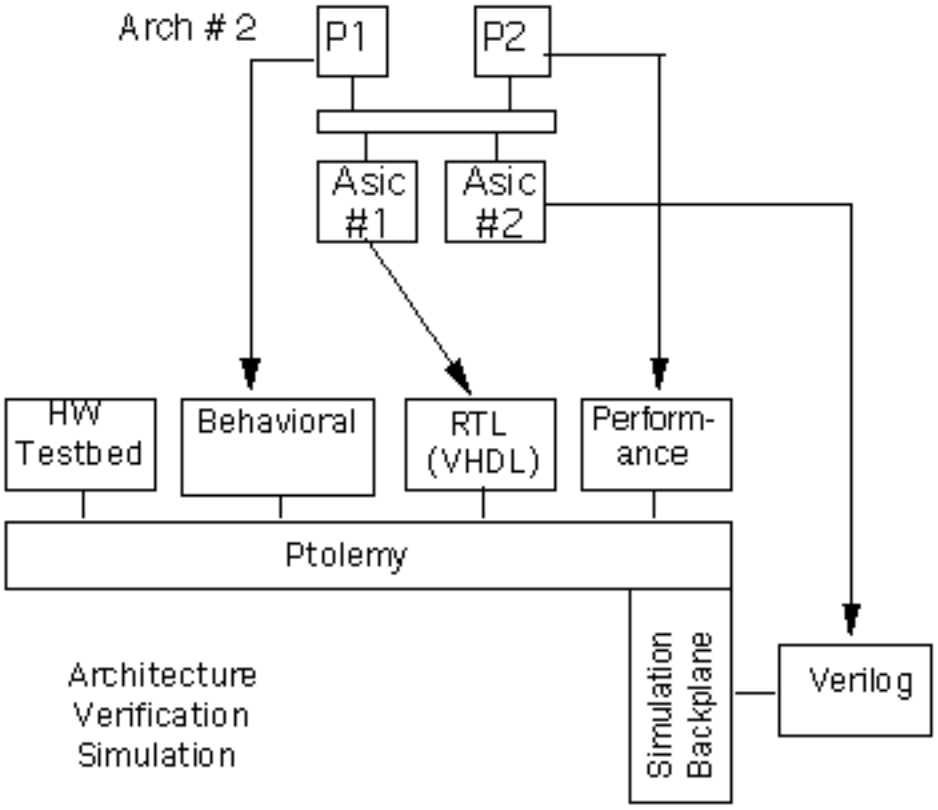


Figure 7 - 11: Mapping of architecture to appropriate simulation engine.

In the event that a hardware testbed is available, detailed evaluation of the performance can be readily obtained through tools that provide detailed timelines of both processing and communication on the architecture. The ability to perform point-and-click mapping of functions to processors provides a way to fine-tune the application, as required. This approach is most useful when performing Model Year upgrades to an existing signal processing system.

7.3.8 Simulation

Executing the constructed, multi-domain simulation provides performance and functionality verification. The detailed simulation provides:

- Concurrent verification of detailed software on hardware and simulation models
- Hierarchical performance verification
- Detailed behavioral specifications for new design elements
- Performance/functional updates to RASSP library models

7.3.9 Trade-off Analysis Update

The iterative process of simulating the various architectures and the graph mappings to them, along with the PDTs estimation of the architecture attributes, using interactive tools where available, results in the completion of the trade-off matrix shown in Figure 7 - 12. The results of the various trade-off studies are reflected in population of the trade-off matrix, which is a record of the design process performed. The entries in the table should be supported by detailed notes to provide both designers and program management insight into the rationale behind the selections.

Architecture Trade-off Matrices									
Architecture Attributes									
	Size	Weight	Power	Schedule	Testability	Cost	Reliability	- -	Total Score
Arch # 1	500 in^2	1.0 lbs	20 W	8 mo	high	low	high		
Arch # 2	200 in^2	0.5 lbs	10 W	18 mo	med	med	high		
-									
-									
Max Score	0-15	0-25	0-15	0-5	0-15	0-10	0-15	0	0-100
Architecture Scores									
	Size	Weight	Power	Schedule	Testability	Cost	Reliability	- -	Total Score
Arch # 1	5	10	10	4	15	8	15		67
Arch # 2	12	20	15	2	10	5	15		79
-									
-									
Max Score	0-15	0-25	0-15	0-5	0-15	0-10	0-15	0	0-100

Figure 7 - 12: Example Trade-off Matrix.

We finish this iterative selection process when one or more candidate architectures satisfies the overall requirements. The IPDT reviews the choices and again selects one or more of the candidates for detailed design.

7.4 Software in Architecture Definition Process

This section provides a discussion of the software related activities in the architecture process. It is intended to provide a consolidated description of how requirements are translated to software in a DFG driven process.

During the system definition process, we develop the initial system requirements. These requirements typically include all external interfaces to the signal processor; throughput and latency constraints; processing flows by operating mode; all mode transitions; and size, weight, power, cost, schedule, etc. We define five items directly related to the subsequent software development:

- Algorithm Flows - Algorithm flows are block diagrams showing the high-level computations associated with an application. They are typically executable descriptions of the processing that must be performed (executable functional specification).

- I/O Requirements - The I/O requirements identify the amount of throughput the application has to handle, the modes of the data, and the source(s) of the data. The I/O requirements and the algorithm flows are the basis for building the architecture-independent domain-primitive graph.

- External Commands - These are the formal definitions of all commands allowable to the operator and/or command program.

- State Transition Diagrams - These diagrams show the major modes of system operation, and the state

changes between them. They, along with the external commands, are the basis for the command program specifications. The external commands and the state transition diagrams are the basis for the command program specifications.

- Command Program Specification - These specifications are derived from the state transition diagrams and the external command definitions defined in the previous phase. They are required to build the domain-primitive graph and are iteratively refined in the production of the allocated graph. This refinement may occur in the architecture selection and validation phase.

This architecture design phase of development involves the creating and refining of the DFGs that drive both the architecture design and the software generation for the signal processor. The input to this phase is the executable functional specification for a particular application and the command program specifications. We then partition and allocate the graph(s) to either hardware or software. During the architecture definition process, the DFG(s) of the signal processing is developed and the elements allocated to either hardware or software. We simulate the flow graph(s) from a functional standpoint and a performance standpoint until an acceptable hardware/software partitioning is achieved that meets the signal processing system requirements. This process contains many intermediate steps and graph objects defined in the following paragraphs.

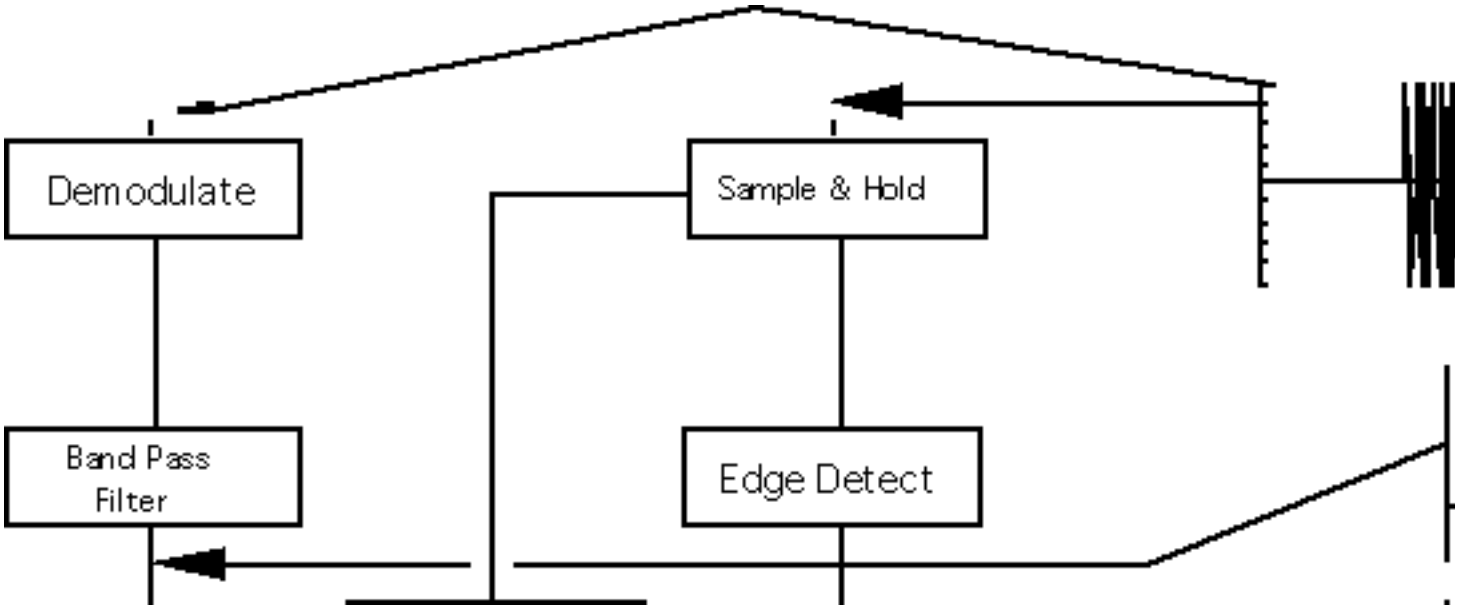
7.4.1 Domain Primitive

Domain primitives are library elements that describe an element of processing to be done. The primitives are HOL programs that perform the graph interface and processing functions of a graph node. Graphs programmed at the domain-primitive level can be automatically translated into graphs executable on any supported target.

7.4.2 Domain-Primitive Graph

The section uses PGM as the examples to explain the Domain-Primitive Graph. GEDAE™ examples can be found in the Hardware/Software Codesign application note, the Data Flow Graph Design application note, and the GEDAE™ web page.

The architecture-independent DFG (domain-primitive graph) is a PGM graph refined to contain only domain-level primitives and subgraphs. The domain-primitive graph created during function design may be refined during the architecture selection process. This graph becomes the functional baseline graph for the application once it has been validated with the test vectors developed for the algorithm flows. The relationship between the algorithm flows and the target-independent DFGs (which are derived in part from the algorithm flows) is shown in Figure 7 - 13.



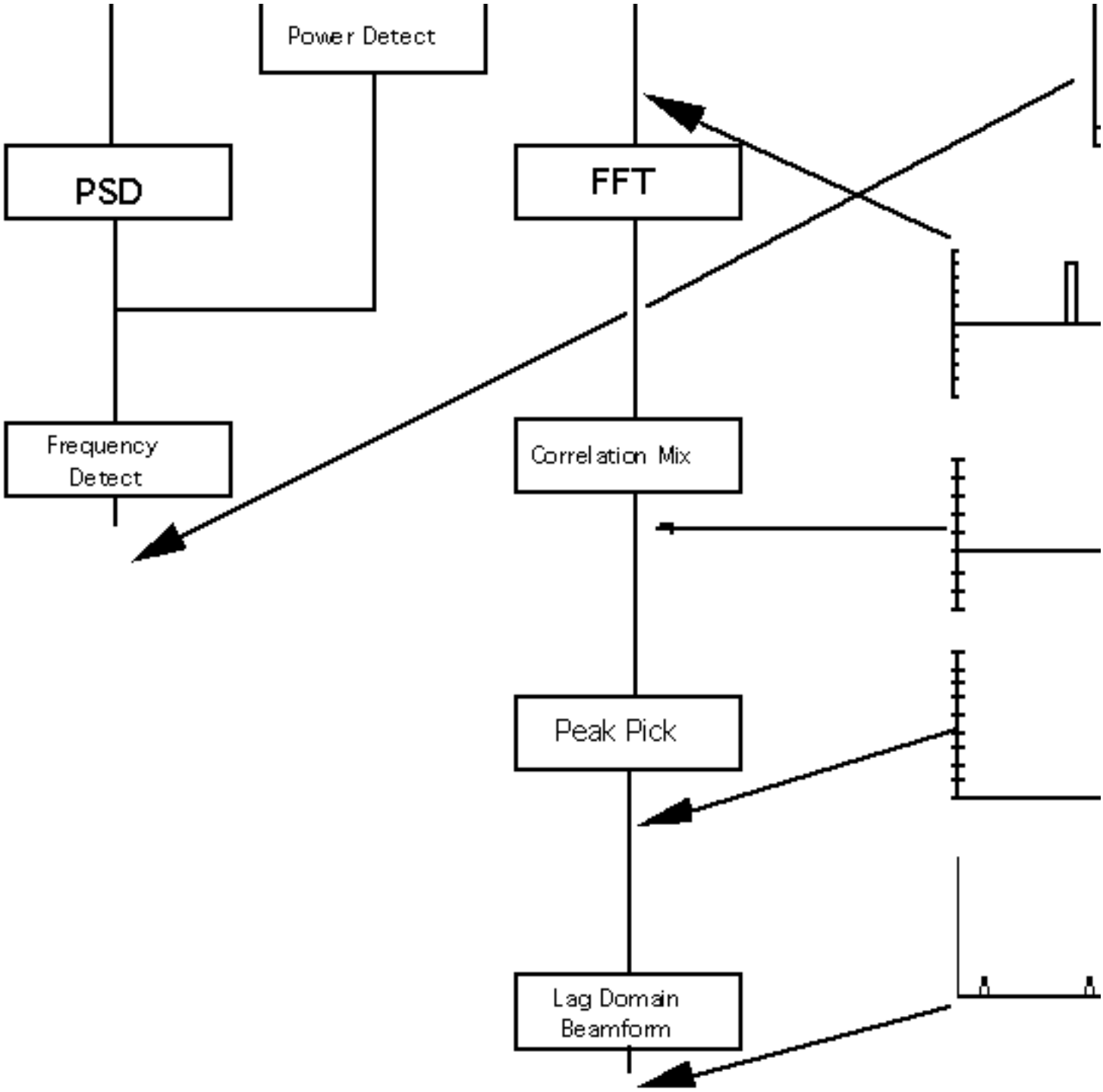


Figure 7 - 13: Algorithm flow correspondence with domain-primitive graph

This figure shows an algorithm flow (shown as a functional block diagram) for a hypothetical application being translated into an architecture-independent DFG (domain-primitive graph). The domain-primitive graph is validated against the algorithm flows produced during system definition. A part of the validation of the domain-primitive graph is the creation of suitable test vectors. The domain-primitive graph is validated once the test vectors produce the same results predicted from the algorithm flows. These test vectors are later used

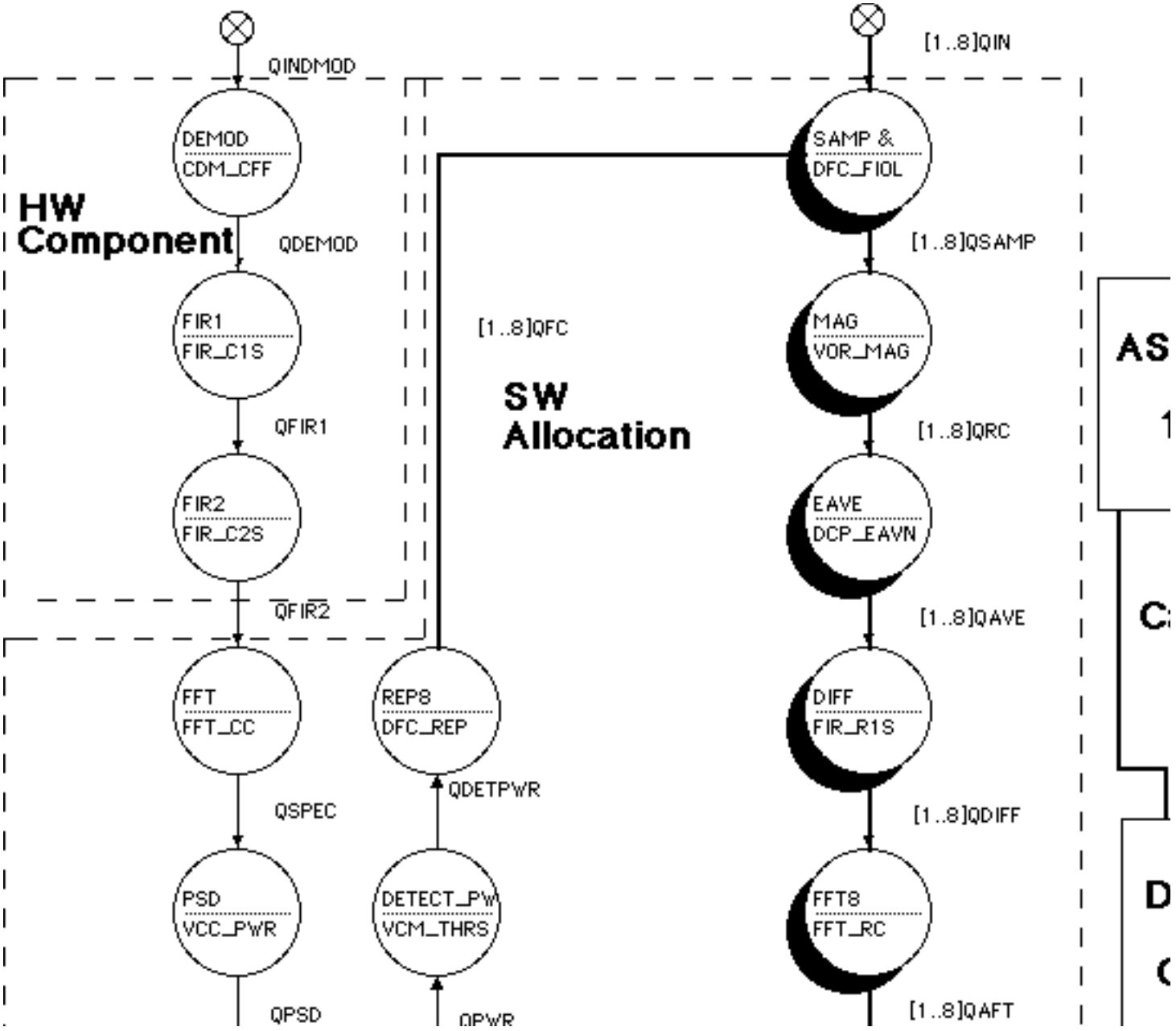
for graph validation throughout the application development process.

Generating the domain-primitive graph may involve an intermediate step in which the PGM graph is composed of domain primitives, domain subgraphs, prototype primitives, subgraphs, or any entity that can specify the workings of an algorithm. This permits continuation of the architecture selection process concurrently with generation of the domain-primitive graph when new primitives are required.

7.4.3 Allocated Graph

We produce the allocated graph by assigning each domain-primitive node or subgraph of the domain-primitive graph to hardware or software. Each node or subgraph must either be allocated to a hardware component contained within the candidate architecture or be identified as a software component to be run upon a processor class contained within the candidate architecture.

Figure 7 - 14 shows a candidate architecture and the segments of the domain-primitive graph assigned to either hardware or software components, which will be executed on the candidate architecture.



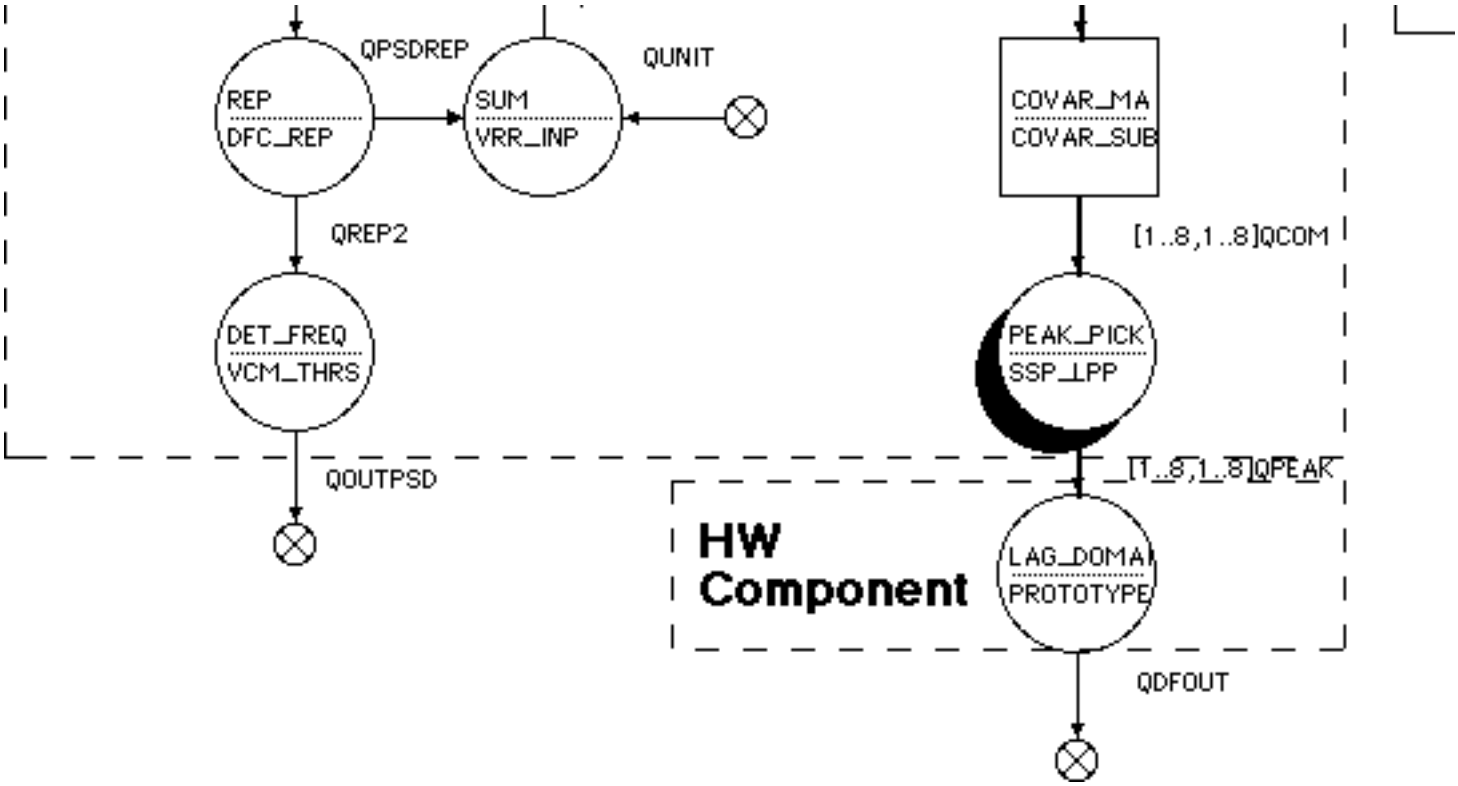
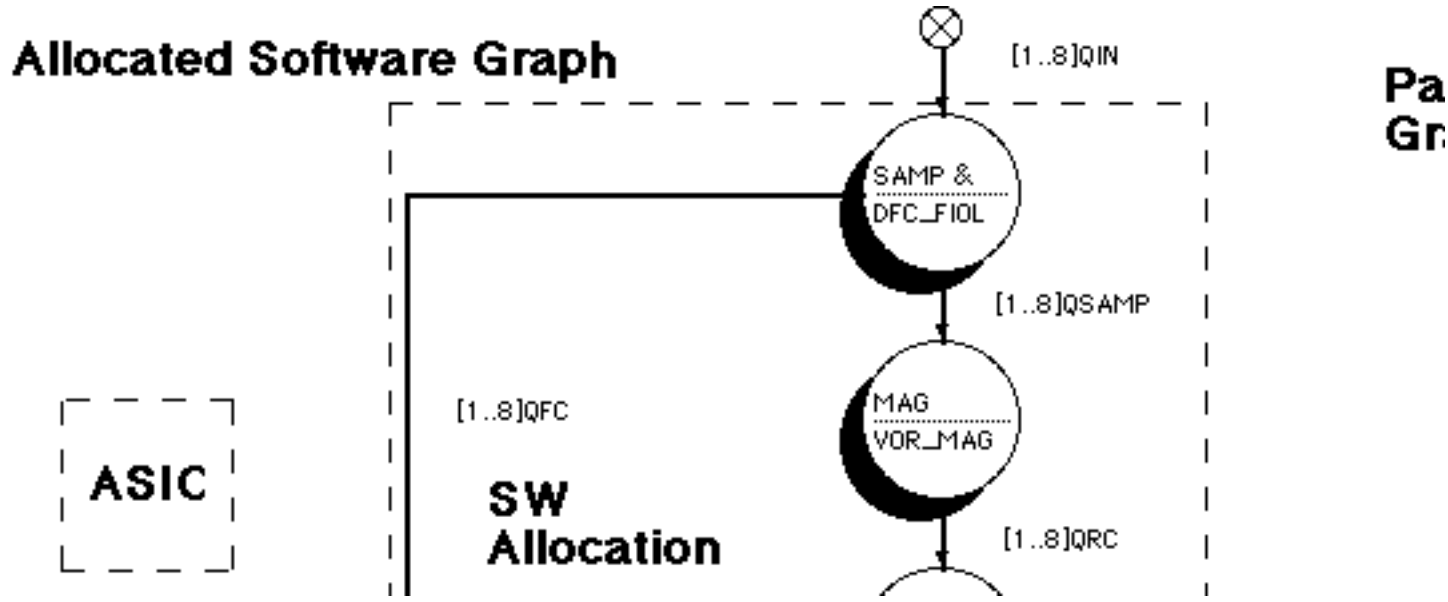


Figure 7 - 14: Graph Allocation to Hardware and Software

7.4.4 Partitioned Software Graph

The partitioned software graph is the software portion of the allocated graph. It has been partitioned into units of software that will be mapped to individual processors in the architecture. It represents all domain-primitive nodes and subgraphs allocated to software partitions. It is important in its own right because it may be partitioned separately from the full graph to optimize software partition performance.

Figure 7 - 15 shows the creation of the software partitions from the allocated graph. The hardware partitions are omitted for simplicity and are shown as ASICs. In this example, ASICs are assumed to be the final form of the hardware partitions as they apply to the code downloadable to the target processor.



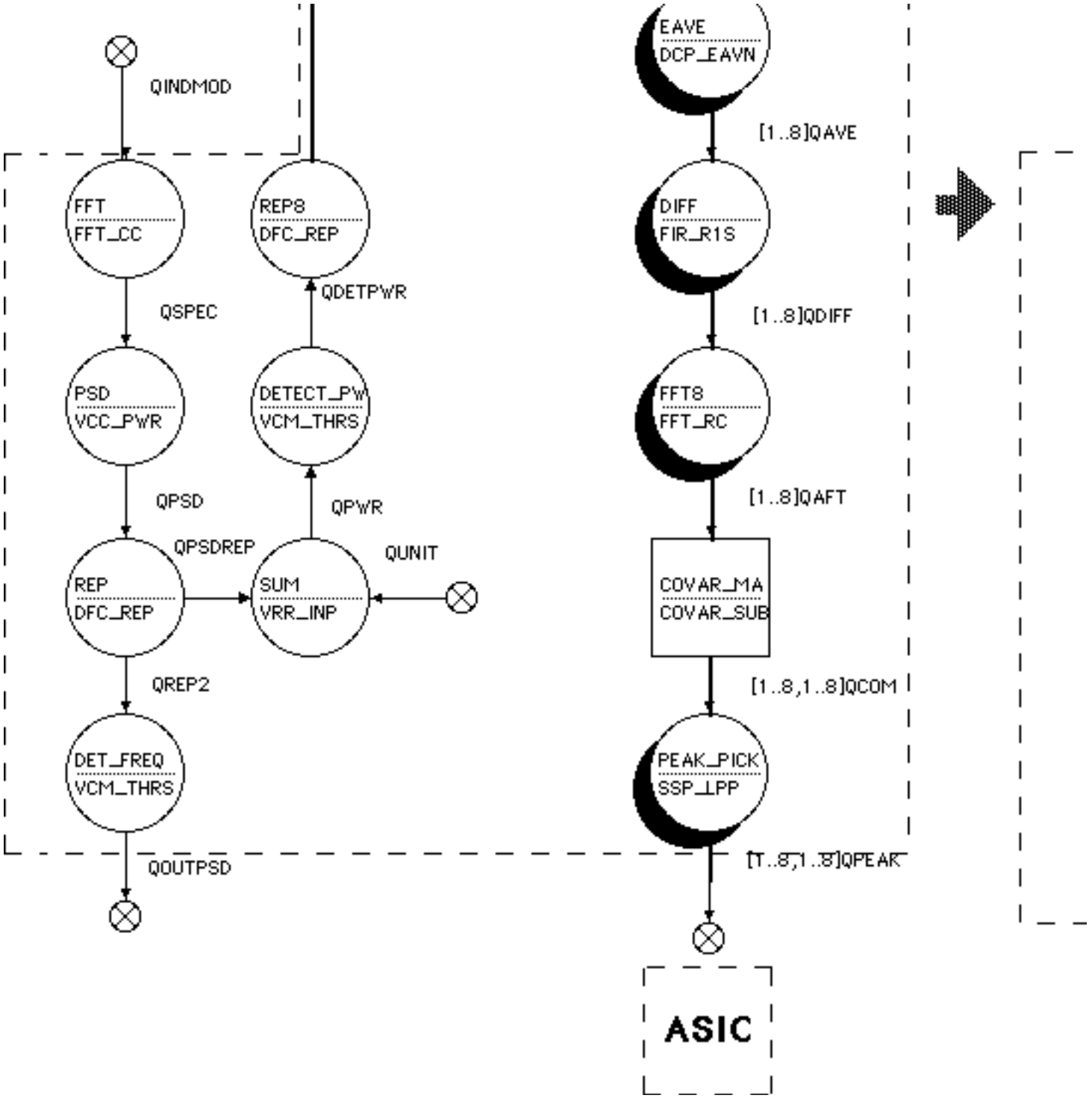


Figure 7 - 15: Software Partitioning of the Allocated Graph

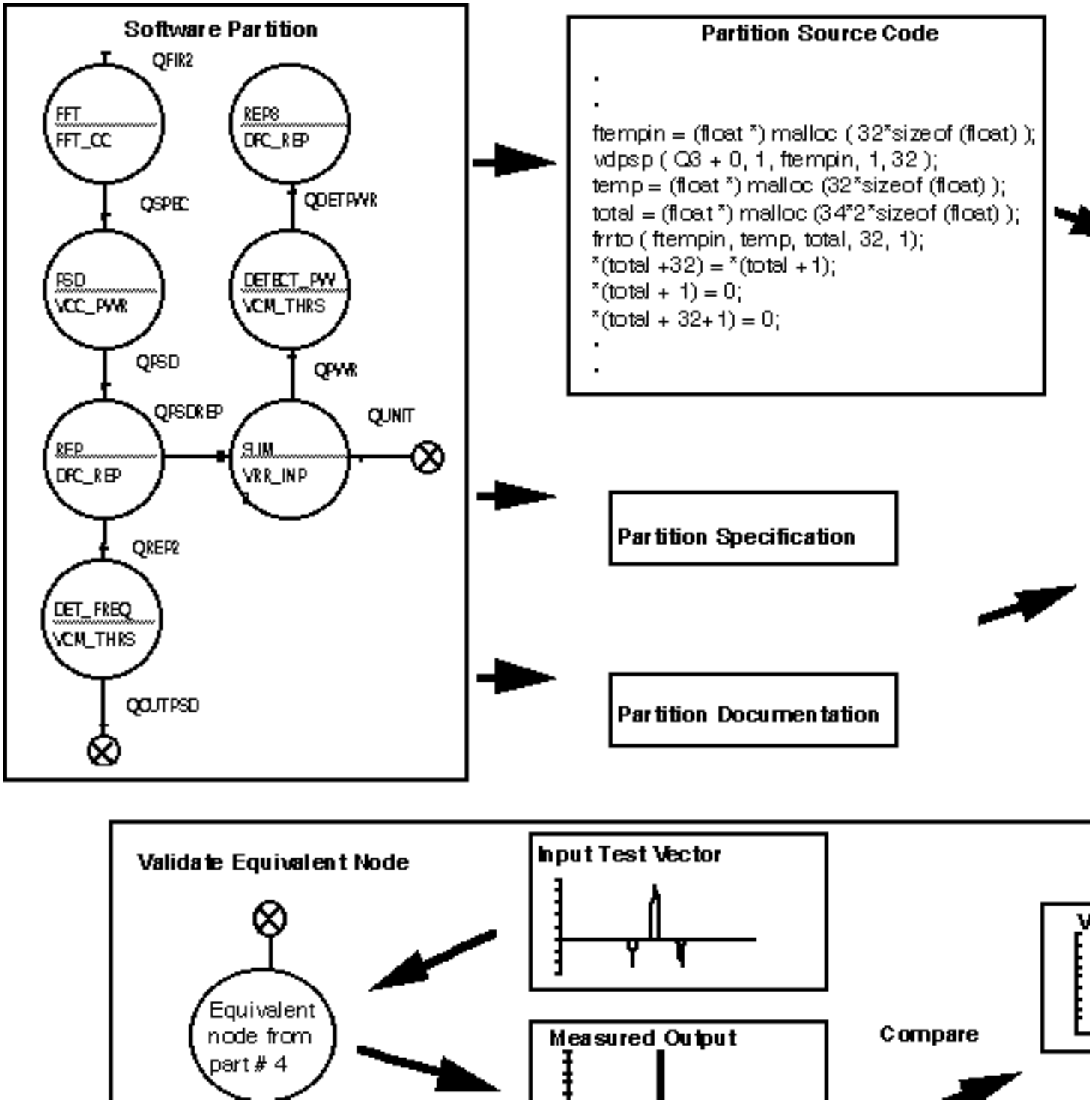
The domain-primitive graph must be re-partitioned for each candidate architecture. The criteria we use in determining the optimal partitioning scheme included the system throughput requirements, latency, memory, etc. We can iterate the partitioning and the architecture to achieve improved performance, cost, reliability etc.

7.4.5 Partition Graph

A partition graph is a stand-alone PGM graph that represents either a hardware or a software partition in the original domain-primitive graph. We build partition graphs in the autocode process, and use them to construct

equivalent nodes for each partition. We also build simulation interfaces for each partition to simulate the application.

Figure7 - 16 shows the derivation of an equivalent node for software partition #1. The software partition is transformed into a stand-alone partition graph that is represented by source code, a partition specification, and documentation. We use the partition specification and source code to create an equivalent node for the partition, which has the same characteristics as the subgraph represented by the partition. We use the equivalent node to build an equivalent graph in which each node represents one of the original graph s partitions, either hardware or software. The equivalent node is validated using the same test vectors used to test the graph partition. This process is equivalent to the traditional software unit test.



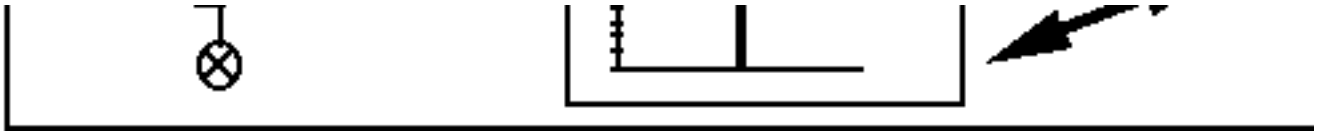


Figure 7 - 16: Creation of Equivalent Application Node for Software Partition

7.4.6 Equivalent Application Graph

We produce the equivalent application graph from the partitioned software graph by replacing each partition with its equivalent node. The equivalent graph is a PGM graph in which all partitions identified in the partitioned graph are replaced with equivalent nodes. The equivalent graph is executed using the same test vectors used during the domain-primitive graph simulations, and it is considered functionally validated if the output vectors of the simulation match the output vectors obtained from the domain primitive graph simulations.

7.4.7 Command Program

In parallel with the graph allocation, partitioning, and performance simulation on a candidate architecture (which is aimed at concurrently optimizing the architecture and the software partitioning), we develop a command program that controls the behavior of the graphs. The command program can interpret the messages received from the external source (operator or higher-level command and control system) and execute the detailed graph commands that control operation of the signal processing graphs within the signal processor. See the Autocoding for DSP Control application note for more details.

7.4.8 DFG/Command Program Functional Simulation

At any point along the development path, we can interface the command program with the DFGs representing the signal processing and these can be jointly simulated to ensure that the graphs are correctly managed within the overall context of the application. Of particular interest is the ability to properly execute mode transitions, which may require stopping, starting, and initializing different graphs; starting and stopping I/O; setting graph variables; or passing data and/or parameters to the external world. This joint simulation ensures that all modes operate properly and transitions between modes occur as required by the specifications.

7.4.9 Non-DFG Software

Non-DFG software requirements tasks and architectures are defined during the architecture process. We develop all code to the maximum extent possible.

7.5 Other Considerations in the Architecture Design Phase

7.5.1 Use of VHDL in Architecture Process

At this point in the process, we have specified DSP system network topology and partitioned the DSP system functionality onto architectural (the hardware/software) elements. We model functionality in terms of timing, resource usage, and algorithmic operations. VHDL is used to model subsystem timing and resource usage. Additional details on performance modeling and its use within the design process can be found in the following application notes:

- Token-Based Performance Modeling
- Virtual Prototyping
- VHDL Terminology and Taxonomy

As we partition functionality between hardware and software, we model the hardware components in VHDL at the abstract behavioral-level; the software components are expressed in a pseudo-language, DFG, program design language (PDL), or ADA. We develop architectural performance (network simulation) models in

VHDL. The software components exist in files that can be interpreted and executed by the VHDL models for co-simulation/co-verification of the candidate processor implementation.

The output of the architecture design process is a test bench and a model for each component.

- Test Bench - The test bench provides test procedures, stimuli, and expected responses to verify that the architecture meets system requirements. The test bench is developed before, and is executed with, the component models during co-simulation/co-verification.
- Architecture Model - The architecture model describes the system in terms of the structural interconnection of components whose functionality is described in terms of a) timing, b) resource usage, and c) algorithmic operations, as shown in Table 7 - 3.

Timing Data	Resource Usage Data	Algorithmic Data
▪ DSP subsystem component I/O - I/O timing constraints - I/O interface structures - I/O protocols - Signal levels - Message types ▪ DSP subsystem component processing latency - Data acceptance rate ▪ DSP subsystem component stimuli response	▪ CPU usage ▪ Memory usage - RAM - ROM - Disk drives - Tape drives - Fast and slow cache - Local and shared memory - Network bandwidth contention	▪ Fast-Fourier Transform (FFT) ▪ QR-decomposition ▪ Finite Impulse Response Filter (FIR)

Table 7 - 3: Elements within the Architecture Model

We must capture the timing and resource-usage behavior of the candidate architecture s hardware components in abstract/non-evaluated VHDL models. (Non-evaluated means that the actual data operations are not performed; rather, just the time required and the control aspects are modeled. Another term for this type of abstract model is "token-based".) Algorithmic operations that have been partitioned into software are expressed in software descriptions. The control aspects of these software descriptions must be interpretable by the non-evaluated VHDL models of the programmable hardware components.

The architecture model, taken as a whole, consists of the structure interconnection of the components, their abstract VHDL models, and the associated software descriptions. The architecture model forms the executable specification of the architecture design, and the VHDL component models form the executable specifications of the hardware components that are passed on to the hardware design process.

Associated with each VHDL component model is a VHDL test bench. The VHDL test bench is a set of VHDL modules that provide stimulus to the component model and check its response, so that we can verify the behavior of the component to meet its requirements. We design the VHDL test bench for each hardware component before the component model is designed. Since the architecture model is more detailed than the system-level performance model, the more precise results of its execution are back annotated to the system model. Likewise, as we obtain the results from the more detailed hardware models in the hardware design process, they are back annotated to the architecture and system models, which makes them precise.

7.5.2 Design For Test Tasks in Architecture Definition

Architecture Design is comprised of three sub-process steps: functional design, architecture selection and architecture verification. The primary focus of the DFT activities is to develop a test strategy and architecture consistent with the requirements captured during system design. DFT interacts with the functional process to affect architecture decisions and component selection. A virtual prototype of the test & maintenance architecture is developed and used to perform Hardware/Software codesign with the test software. Additional details on how to do design-for-test can be found in the Design-for-Test application note.

The functional design step provides a more detailed analysis of the processing requirements (including BIST), resulting in initial sizing estimates, detailed data and control flow graphs for all required processing modes to drive the hardware/software codesign, and the criteria for architecture selection. Figure 7 - 17 shows the DFT steps which occur during functional design.

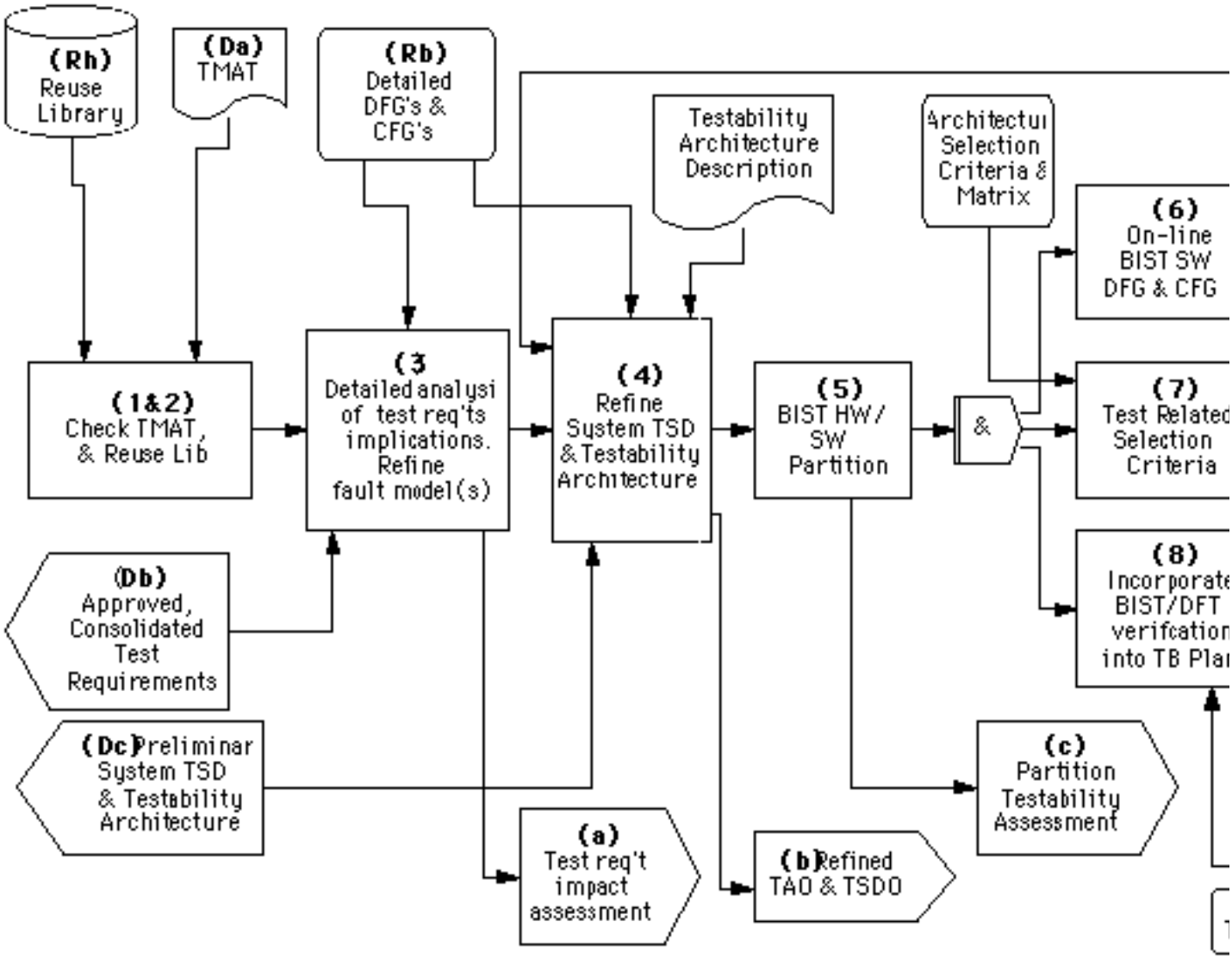
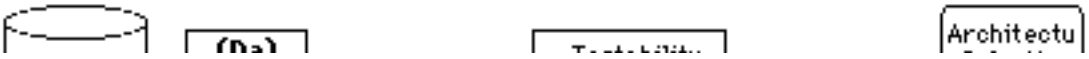


Figure 7 - 17: DFT Steps in Functional Definition Flow Diagram

During architecture selection, various candidate architectures are evaluated through iterative performance simulation and optimized to appropriate levels of detail. A trade-off analysis based on the established selection criteria results in the specification of the detailed architecture, and software partitioning and mapping. As part of the trade-off analysis, information is used from the required cross disciplines such as reliability and testability (either manually or through design advisors) to populate the trade-off matrix. Figure 7 - 18 shows the DFT steps which occur during architecture selection.



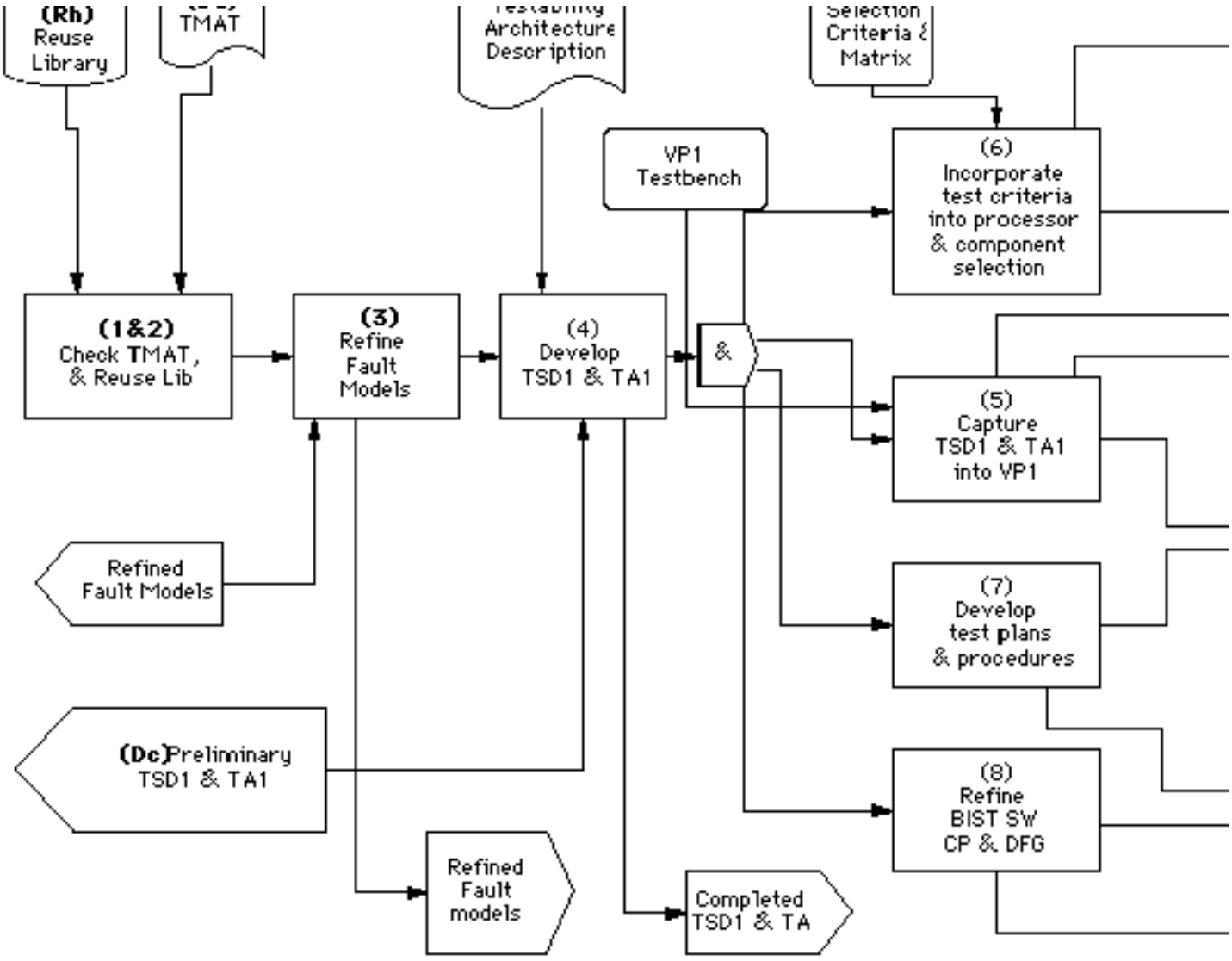


Figure 7 - 18: DFT Steps in Architecture Selection Flow Diagram

This portion of the process is heavily dependent on the reuse of architectural (hardware and software) components to provide significant time-to-market improvements. In addition, during architecture selection, all software not represented by the DFGs is designed. Based on the requirements, the non-DFG software may include BIST1, downloading data and code, and diagnostics. The virtual prototype, VP1, produced during architecture selection is not a full system prototype, since function and performance are simulated independently and may or may not be coupled with the overall control mechanism. Several architectures may be selected for verification during the following step (i.e. primary candidate and risk reduction alternatives). The Architecture Selection level test strategy diagram, TSD1, and testability architecture, TA1, are developed concurrently with VP1 for each candidate architecture.

Architecture verification is the process of hierarchically simulating both the functionality and increased

performance detail of a selected architecture candidate. Up to this point the overall functionality has not been verified. An integrated, simulation-framework supports mixed-domain simulation so that high-level performance and functional simulation can be coupled with ISA or RTL VHDL simulators, hardware emulators or hardware testbeds. The goal is to validate operation of all architectural entities including embedded test 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. The resultant virtual prototype, VP2, is passed onto detailed design. The Architecture Verification level test strategy diagram, TSD2, and testability architecture, TA2, are developed concurrently with VP2 for the selected and verified architecture. Figure 7 - 19 shows the DFT steps which occur during architecture verification.

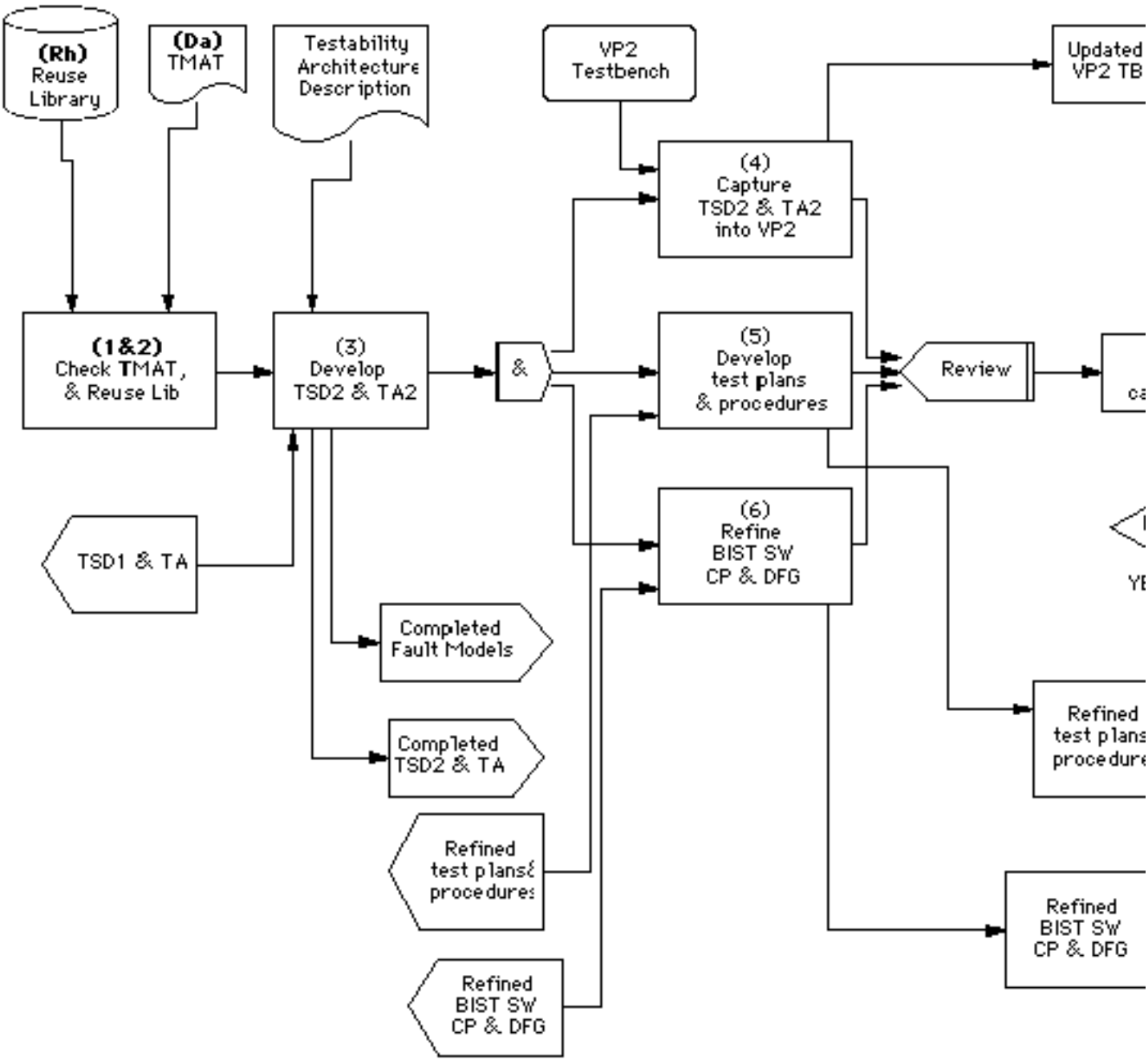


Figure 7 - 19: DFT steps in architecture verification flow diagram

During the architecture definition step, the "lead, follow, or get out of the way" strategy for COTS at the testability architecture level is developed. Controllability and observability analysis results are used to determine what solution(s) could be examined for sections anticipated to employ COTS. The possibilities are explored of absorbing some or all COTS functions (that are in non-testable devices) into ASICs or FPGA type devices that incorporate at least boundary-scan, if not BIST. A refinement of the COTS fault model also takes place.

7.5.3 Role of PDT in Architecture Process

The architecture process is not only an ideal candidate for concurrent engineering, it is defined in such a way that concurrent engineering is mandatory. By definition, the process includes elements of systems, hardware, and software engineering, and it is supported heavily by the other disciplines, as reflected in the integration of hardware estimates, costing, mechanical complexity, thermal issues, software support, and RAM-ILS parameters into the architecture selection criteria. The following paragraphs describe the role of the PDT disciplines in the architecture process.

- Team Leader - The team leader is the principal interface with program management who monitors progress on the subsystem and ensures that all system programmatics are satisfied. It is assumed that the team leader has participated to some degree in the systems process and at a minimum has participated in the SDR. During the architecture process (which encompasses hardware/software codesign and verification of the selected architecture), the team leader and program management schedule the appropriate design reviews at various stages of the process. At the top-level there must be a review (PDR) of the selected candidate architecture, along with all the inputs that make up the weighted selection criteria. This provides a go/no-go gate before the detailed architecture verification. This is not a sequential process, however, since all the appropriate disciplines have been involved in the architecture selection process to some degree. Much of the effort during the architecture verification process is aimed at producing the detailed support for the subsequent Critical Design Review (CDR). In reality, many less formal reviews will take place regularly within the PDT to evaluate ongoing architecture development. Inserting automated tools into the architecture process provides the opportunity to evaluate more architecture options, which increases the need for multidiscipline review. Program management representing the IPDT will hold meetings with the PDT leader(s) to assess status regularly.

- Engineering - The architecture engineers have the lead role in the architecture process. The architecture process is a new process that is being inserted into the design methodology. It includes portions of the processes traditionally performed in the system design and hardware design processes, along with key elements of the software process. The architecture process is truly hardware/software codesign in that the candidate architectures reflect the performance of the software on the architecture. Therefore, the composition of the engineering team is perhaps different from the traditional one in that it probably includes systems, hardware, software, and test engineers as principal contributors, along with support from mechanical, thermal, etc. During the architecture process, all aspects of the candidate architectures are considered to develop the weighted selection criteria that must reflect the overall system requirements. Systems engineering ensures that system requirements are met and performs additional system-level trade-offs based on the results achieved in the architecture selection process. Software engineering evaluates library element sufficiency and initiates both prototype element building and library element validation. Test engineering ensures that testability is addressed in the trade studies.

  Like the other disciplines, the mechanical engineering representative(s) on the PDT primarily contributes to defining the selection criteria parameters based upon the overall program requirements. Given the complexity of a given architectural design, it may be necessary for mechanical/packaging personnel to provide inputs regarding the desirability of one design over another based upon the amount and/or type of cooling necessary to implement the design within the form factor required. The total power requirement of each candidate architecture may meet the overall power budget allocated during the system design, but the power dissipation per volumetric unit may dictate different approaches for the different architectures. These characteristics may impact the overall packaging

approach, accessibility for test, maintainability, and cost; all of these should influence the ultimate architecture selection.

- Manufacturing/Product Assurance - As part of the PDT, the manufacturing representative works with engineering to provide inputs that are factored into the selection criteria for any given architecture. The manufacturing representative provides inputs regarding the degree of difficulty of the manufacturing process for any candidate architecture or packaging approach. If a design requires that a new or special process be implemented, significant cost, schedule, and risk impact may be associated with the design. These inputs must be an integral part of the architecture process.

- Test - Test engineers evaluate the various architectural candidates with respect to test requirements. One aspect of test engineer participation includes providing recommendations or weightings for different architectural elements based upon their ability to support test. As an example, one processing element may support the JTAG standard for boundary scan, which increases the likelihood of available software for testing; another processing element may provide a non-standard boundary scan capability, which increases the cost of testing. Inputs may also be provided on the level of validation, which must be performed in the architecture verification process. Obtaining these types of inputs from test engineering as early in the process as possible optimizes the potential for minimizing test cost and schedules later in the program. The ability to evaluate alternative architectures quickly makes it possible early in the overall design process to make and evaluate product substitutions based on inputs from test engineering.

- Producibility - The architecture selection must be influenced by all aspects of the product development and manufacturing process. We anticipate that the primary method of incorporating these influences at the architecture level is to provide a mechanism for biasing the selection criteria.

- Product Assurance - Minimum input is required during the architecture process.

- Parts Management - Provide technical support to engineering during the trade study process, if necessary. Minimum input is anticipated during the architecture process.

- Reliability, Availability, Maintainability (RAM) and Safety - The RAM inputs are of particular importance during the architecture process because they are directly related to integrated logistics support and LCC. Provision in the methodology to obtain early estimates of these factors for various alternative architectures may prevent the need for additional design iterations later in the process. Ongoing reliability assessments will verify that final architectures meet requirements.

- EMI/EMC/TEMPEST - Provide guidance regarding the appropriateness of architectural library elements with regard to the customer requirements. If appropriate, restrictions can be placed on the usability of various library elements.

- LCC - Work with engineering and manufacturing to assure that the architecture decisions properly reflect the long-term impacts on the program.

- Sourcing - Sourcing personnel provide the interface between the PDT and the suppliers. During the architecture process, it will become obvious which architectural elements are being favored. Sourcing should provide information via the suppliers regarding the availability of components to support the product development cycle. If particular processing elements are available in limited quantities to support early prototyping, but cannot be available in sufficient quantity to support the required program schedule, this information is critical to the architecture selection process. In addition, sourcing may contribute to architectural component selection by providing current information regarding costs, or renegotiating volume prices based upon the design under consideration. All this information should be considered during the architecture selection process, since it can impact the overall program cost and schedule.

- ILS - Logistics personnel ensure that the PDT recognizes the implications of architecture decisions on

the overall support requirements of the signal processor.

7.5.4 Design Reviews in Architecture Process

- System Requirements Review (SRR) Key members of the architecture design team participate in this review. The architects provide inputs during system design with respect to the appropriateness of the allocation of requirements to the signal processor.

- System Design Review (SDR) Key members of the architecture design team participate in this review. The architects provide inputs during system design with respect to the appropriateness of the allocation of requirements to the signal processor.

- Architecture Design Review (ADR)
  The architecture design team leads preparations for this review and prepare the design package. At ADR, we present all architecture trade-off studies performed and the selection criteria developed. We present and review the final recommendation of an architecture to be carried forward. The results of all preliminary performance and behavioral simulations that support the recommendation are included in the design package. We present all architecture verification simulations and any additional architecture selection trade-offs we conducted as a result of the verification process. We present detailed results from the verification steps to ensure that the architecture design meets all specifications before release to detailed design. We establish the influence of all disciplines contributing to the architecture selection and define all external interfaces to/from the signal processing system.

- Detailed Design Review (DDR)
  The architecture design team supports preparations for this review and supports the preparation of the design package.

- Production Readiness Review (PRR)
  The architecture design team supports this review based upon the participation in the integration and test activities and any product refinements that occurred since DDR.

RASSP Methodology Application Note

8.0 Detailed Design Process Detailed Description

We decompose the overall hardware design process into several distinct yet interrelated subprocesses:

- Module design
- ASIC design
- FPGA design
- Backplane design
- Chassis design
- Subsystem integration and test

We start the module design process with inputs in the form of the Module Requirements Specification generated by architecture design. The module is the fundamental line-replaceable unit (LRU) for the signal processor hardware. It is the primary focus of testability, reliability, and maintainability analyses.Here the module architecture is partitioned into a combination of existing components and new or existing ASICs. The ASIC design process is spun off at this point as another mini-spiral in the risk driven development process.

ASIC design takes the Type C specification from architecture design, which may have been augmented by module design, and develops behavioral or RTL models of the ASIC using VHDL. This is followed by functional behavioral simulation. Next, we perform detailed design (synthesis) of the ASIC functional blocks, followed by gate-level functional simulation, using the same test vectors and expected outputs as in the behavioral simulation. The behavioral and gate-level simulations must agree 100 percent before proceeding to the next step.

In parallel with ASIC design, we perform the detailed module design involving interactive logic design, simulation, and timing analysis. Functions are partitioned to FPGAs, if applicable, and FPGA design begins as another mini-spiral.

We perform the first module-level simulations using the ASIC behavioral or RTL model(s) developed by the ASIC designers. As the ASIC design progresses, gate-level ASIC models, and FPGA models may be brought into the overall module simulation environment, depending on overall module complexity. A test bench that is usually generated by architecture design is used on the module as a check for correct requirements mapping and implementation. We then perform timing analysis and fault simulation. When the module is deemed correct (functionally and in terms of timing), after the Module Preliminary Design Review, module preliminary design is complete.

FPGA design, also in parallel with module design, involves the development of a programmable logic device from its beginning as a behavioral VHDL model through detailed design, synthesis, place and route, and programming using a device programmer. This process is similar to the ASIC design process.

We start final module design with the physical layout. We back-annotate added signal delays due to module routing to the simulation model, and the perform a complete module simulation. In some cases, we simulate multiple modules at the structural level to test interfaces and a higher level of functionality. The test bench used to stimulate a multimodule simulation is the same as for earlier behavioral simulations. After the Pre-Release Design Review, manufacturing fabricates and assembles the first-piece module and engineering tests it, using the same test vectors used during module simulation. The Final Design Review is held, after which all module

documentation is signed off.

Backplane design, which began in parallel with module design, involves determining the intermodule wiring scheme (printed wiring board and/or wirewrap) for the collection of modules, and performing detailed analysis of the signal integrity of the backplane, using transmission line and crosstalk analysis tools.

Chassis design also begins in parallel with other design activities and encompasses the mechanical and electrical design and analysis of the structure that holds the backplane(s), power supplies, cable and harness, I/O connectors, and other miscellaneous hardware. With appropriate CAD tools, the chassis is designed and a 3D model is generated. This model is used to analyze stress, temperature, mechanical tolerances, and dynamic characteristics.

Also in parallel with module and ASIC design, we re-evaluate the chassis mechanical conceptual design approach against the program subsystem-level requirements (B2) specification. We develop a mechanical design concept for the enclosures and electrical packaging, then generate solid models and a C-level specification. This effort requires support from manufacturing for producibility and cost trade-offs. Producibility analysis is built into the mechanical design's early stages. The Preliminary Design Phase refines the chassis design concepts developed during the Conceptual phase. We generate and integrate solid models and detail layouts, where applicable, into overall mechanical design layouts that incorporate key features, interfaces, dimensions, etc., for the particular mechanical design. This reduces risk due to misinterpretation or missed revisions, and provides the up-front discipline and integration to assure a correct initial database during the Preliminary Design Phase. A Preliminary Design Review (PDR) is held following completion of preliminary design and prior starting detail design. At this point, we have completed all detail and interface layouts, selected major parts, identified materials and processes, and completed and analyzed major chassis models. We begin preparing test plans and issue long-lead parts lists to manufacturing. The PDR ensures that these tasks have been satisfactorily completed, preliminary design is complete, and detail designs can be generated from the preliminary design.

In the final design phase for the chassis, we generate the actual detail design and assembly drawings used to fabricate and assemble hardware end-items. Designs are transferred electronically from solid models into 2D or 3D formats used to fabricate hardware without unnecessary modification. Outputs for the Final Design Phase include issue of Release Packages that allow manufacturing and sourcing to efficiently fabricate or procure hardware. A Final Design Review (FDR) is held before issue of the Release Packages to ensure that hardware designs are correct and meet the requirements of the subsystem level specification. The major portion of the mechanical engineering effort is complete following the FDR and issue of Release Packages.

Subsystem integration and test is the last process in the overall hardware design process. We follow a phased integration approach where we integrate fully-tested individual pieces into the next higher level. Initially, we integrate the modules into backplanes, then test them; the tested backplanes are then integrated into frames and tested. The eventual result is a complete, fully tested subsystem. Detailed test procedures are followed in all phases of this process.

8.1 Module/MCM Design Process

A module or MCM is a design that encompasses a logical function (or a group of logical functions) that can be implemented with circuit devices on a single circuit board. A module may include CPLDs/FPGAs or ASICs with their own design process. The module designer is responsible for the integrity of the outputs from these lower-level processes and for verifying their proper operation within the target digital module. The module/MCM design process is detailed in Figure 8 - 1.

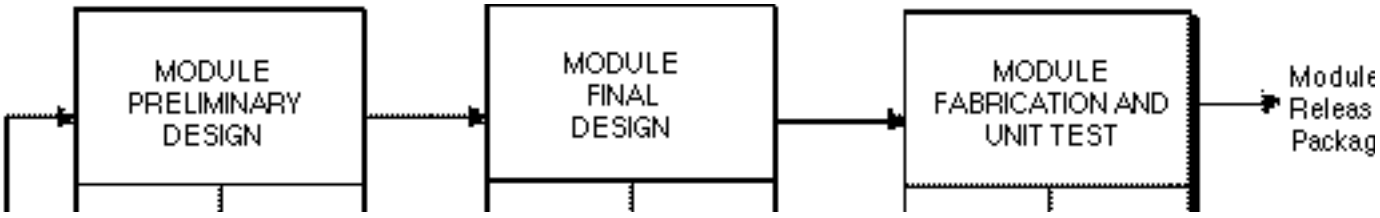


Figure 8 - 1: Module/MCM design process flow

8.1.1 Module Preliminary Design Phase

The module design process starts with preliminary analysis and review of the Module Requirements Specification. We search the RASSP design reuse library concurrently with the preliminary analysis after reviewing the module requirements. If no existing module meets the requirements, then we assess the feasibility of implementing the function and develop the overall architecture of the module. Sizing and partitioning the logical functions required on the module must be undertaken to:

- Verify that the function can be implemented on one module with existing technology within cost constraints
- Allocate the logical functions between CPLDs/FPGAs, discrete logic, and ASICs specified by architecture design.

We review the test requirements, testability goals, BIT requirements, and target tester for this module. This step includes fault coverage goals, fault isolation goals, identifying replaceable units (RUs), type of testing to be performed (i.e., functional, in-circuit), type of test machine that will test the module, design-cycle time, tester-cycle time, level of test (module, system), and information on any standard test busses that may be required for the system.

Another important aspect of testability goals is whether and how a circuit must be probed. For example, it may be acceptable to probe a board in factory test (before conformal coat), but probing may not be allowed for field test. It is important to know and analyze all these constraints and requirements before developing a testability concept and plan.

We generate a concept for designing-in testability for this module. As we identify the test objectives, we select the best method for implementation, such as off-line (external stimulus and response), built-in (internal stimulus and response), and self-checking (e.g., parity checking during operation). The output is a preliminary testability concept document that the designers use when performing the detailed electrical design.

We may need to conduct trade-offs with architecture design and backplane designers following this analysis to achieve the desired module requirements. Manufacturing and mechanical engineering should be involved to ensure that the proposed design can be produced within project, technology, and cost constraints. Reliability and maintainability engineering should be consulted so that the overall architecture and parts chosen meet reliability/maintainability needs. Sourcing should be notified of any possible long-lead parts selected to determine if the procurement time for these devices meets schedule constraints.

8.1.1.1 Develop Module Behavioral Model

All modules are an integral part of a larger system and perform some unique subsystem function. In most cases, the architecture design engineer modeled the unique architectural elements using VHDL to gain confidence that all functional blocks will interact properly. In these cases, each partition (i.e., the module definition) should have had a high-level model developed that can be used by the module design engineer as the starting point for behavioral model development.

The first step in designing a behavioral model of a module is to finalize and define, in detail, the interconnections between functional blocks in the block diagram of the module. After we define all the I/O to each functional block, we can then develop the behavioral model for the function or further break the function

into smaller blocks. Incomplete interconnection definition increases design time and increases interdependence of functional block designs. A module requiring multiple design engineers should have very accurate inter-block definition; changes of I/O should be immediately documented for hand-off and future review.

When developing the behavioral model, the module design engineer should reflect the inherent structure of the design. If, for example, the module has a pipelined architecture, the pipelines and the functions performed by each pipeline stage should be evident from the model. The purpose is to produce a model with sufficient detail to guide gate-level design.

The design database is continually updated with more detailed descriptions of each function as the information becomes available. The I/O definition of each block is detailed enough to allow independent, concurrent design of each function. We then compare functions to the reuse library functions to eliminate duplication of effort. We enter new, common behavioral models into the library database as they are developed and validated.

Also included in this step is development of the 'tester' model, which applies input vectors to the model and collects and analyzes the results. This is a simple model written in the same language as the behavioral model or a complex suite of programs integrated into a complete simulation environment.

8.1.1.2 Generate Module Test Plan

We develop the Module Test Plan in accordance with local standards unless a different format is called for by the customer contract. The Test Plan should define all steps in the simulation process, including the strategy for generating test vectors that exercise all logic blocks, the method of performing comparisons with higher-level simulations, and the merging of ASIC behavioral and gate-level models and FPGAs into the simulation environment. The plan for meeting fault simulation stuck - at - 1 and stuck - at - 0 detection requirements is also addressed in this document. The Test Plan should be sufficiently detailed to allow efficient transfer of all information regarding the simulation effort to anyone associated with the module design. An informal peer review of the test plan is desirable to identify and correct any deficiencies in the plan before the start of simulation. Test engineering and manufacturing should be invited to the peer review.

8.1.1.3 Perform Behavioral Functional Simulation

We simulate the behavioral model of the module to verify that its performance meets the subsystem and ASIC requirements. This effort may be performed jointly by the module design engineer and the ASIC design engineer to ensure full coverage of all subsystem and module requirements. It is quite common for the module design engineer to discover design flaws in the module simulation that are difficult to detect in a single-chip simulation. This may include the detailed interaction of the ASIC during module reset or instruction abort procedures. Using the latest version of the ASIC behavioral model, the module design engineer can discover potential errors before module integration and test.

8.1.1.4 Generate Module Functional Test Vectors

A test vector is a set of bits used to test the operation of the module. In general, the bits define inputs to the module (stimulus) and the expected outputs from the module (response). As each vector is applied, the actual response of the module is compared to the expected response. If these match, the module is performing as designed. Test vectors are required for both simulation of the module and for testing the completed module.

Generating test vectors for a module generally involves more than just specifying bit patterns. In most cases, we must define a 'tester' logic function to generate stimuli to emulate, for example, a bus interface protocol. When a standard microcomputer is involved, the test vectors are limited to the relatively small number of signals needed to reset the processor, plus the (sometimes extensive) firmware that must be downloaded to RAM for subsequent execution by the processor, which in turn generates bus activity, etc.

There are generally two classes of test vectors: functional and production. Functional test vectors verify that the design works as intended. Production test vectors differ from functional vectors in that the production

vectors detect manufacturing defects. The functional vectors may also be used as production vectors, but are usually augmented with additional vectors to detect module faults that the functional vectors do not reveal. Production test vectors generally also use any designed-in-test features of the design, such as controllability and observability points and boundary scan. These vectors must also consider the target tester's restrictions, which include simulation speeds, timing limitations of the tester's drive and detect channels, pattern length, and pattern depth.

In this step, we emphasize generating functional test vectors, although some attempt at vectors designed for fault detection can be made. In "Perform Fault Simulation," the designer grades the vector set on its ability to detect module faults. Additional vectors needed to increase fault coverage will be produced in that step.

We perform this step concurrently with "Perform Functional Simulation." As we develop each group of test vectors designed to test a specific function on the module, the simulation is usually run using that group to verify the function instead of generating all the test vectors before running any simulations.

8.1.1.5 Generate ASIC/FPGA Requirements

The module design engineer generates a high-level description of each function that is implemented as a FPGA. This description may include possible device types for implementation based on an evaluation of the approximate number of inputs, outputs, and product terms for each function. Both the module and FPGA designers review the FPGA requirements to ensure that the documentation is complete and consistent. Typically, the module design engineer and the FPGA design engineer are the same person.

8.1.1.6 Search Design Reuse

Concurrently with the preliminary analysis/design step, we search the RASSP design reuse library to determine if an existing (legacy) design can meet the requirements specified for the target module. In some cases, we can make a minor modification to an existing module to meet current requirements.

The module designer considers all available options before proceeding with the design process. If the library search produces an existing module that meets the requirements, no further analysis is needed. If an exact match is not made, we can check the library for other designs that could be used, either whole or in part. The library also contains entries that make it possible to contact the original designers, who could be valuable consultants. We use the results from this review of existing designs and the cost goals to determine the actual module design approach.

The Module Requirements Specification, generated primarily during architecture design, is a high-level description of the module being designed. We can generate it after completing a firm definition of the architecture. This document should include the following minimum subset of topics:

1. Functional Requirements
2. Functional Block Diagram
3. Preliminary Parts List
4. Register level Block Diagram
5. Preliminary Layout
6. I/O Pin Utilization
7. Physical Characteristics
8. Environmental Requirements

The detailed electrical design process involves the interconnecting the logic devices required to implement the module function. This portion of the process also includes placing the parts on the circuit board, analyzing module characteristics (Critical Path Analysis, generation of FPGA requirements, etc.), and producing a design database. This step is broken into sections explained in the following paragraphs.

The module electrical design process presented here assumes that we will simulate modules after the logic schematic is fully captured or the logic synthesized and a netlist generated. A more structured, top-down

approach can also be accommodated. The following steps can either be completed for the full design, as shown, or they can be repeated as each functional block of the design is generated during the functional requirements generation step. If a behavioral model was generated previously, mixed-level simulations can be run as the logic for the different functional blocks is generated.

8.1.1.7 Interactive Logic Design

We perform the detailed logic design of the module on each logic block concurrently, as appropriate. By designing each functional block separately, we can distribute the design of complex modules among multiple engineers to shorten the overall design cycle. However, each functional block must interface with its neighbors. Therefore, a functional block of the module cannot be considered an isolated design. Information must flow freely between the team of design engineers. Testability should be addressed at the start.

As the detailed design progresses, it should be captured with the appropriate schematic capture tool.

If architecture design specified the use of any ASICs, any additional ASIC requirements must be provided to the ASIC designers in the form of additions to the ASIC Requirements document produced by architecture design.

We must now refine the testability features of the module introduced in the conceptual design. Additional hardware may be required to provide for testing of all functions.

Information is also required regarding the target module tester. This data minimizes the amount of translation required when the production test vectors are generated. This data should include information about restrictions on device names, signal fan outs and loading, and power and ground pins (for standardization of test fixtures).

8.1.1.8 Preliminary Layout (Parts Placement)

After entry of the schematic into the workstation, we must do a preliminary parts placement to ensure that all devices in the design fit on the module. If there is insufficient room to fit the module function on the circuit board, then we must conduct trade-offs. These trade-offs can be alternative logic implementations using logic minimization techniques, using different part types (e.g., CPLDs), or repartitioning the module functionality to other modules.

Representatives from mechanical engineering and manufacturing should be consulted during this step. The mechanical engineer can help with power dissipation and the distribution of heat across the module. These issues can affect parts placement. Manufacturing and mechanical engineering can handle other concerns related to parts placement, such as the minimum distance parts can be placed from each other, and the positioning of Pin Grid Array (PGA) devices so that solder inspections can take place.

8.1.1.9 Critical Path Analysis

The next step in the design process is timing analysis of critical paths. The purpose of the timing analysis is to analyze the timing performance of the design, primarily to determine producibility. Unlike the idealistic typical simulation, a real board is composed of parts whose timing performance varies from typical. In this analysis, we attempt to identify portions of the design where a set of fast or slow parts, or a combination of fast and slow parts, causes the board to function improperly.

We begin the timing analysis by identifying the design's critical paths. A path for this analysis is the set of wires and components between any two pins in the design. A timing path is one that begins at the module's primary inputs or any internal register's outputs and ends at the module's primary outputs or any internal register's inputs. The path delay is the accumulation of propagation times through each wire and each component on the path. Note that the propagation delay through a component is not a constant, but is a function of the capacitive load on the driving pin. The critical paths of a design are the set of timing paths that have the greatest path delay values, and thus determine the limit of the design's timing performance.

We use a static timing analyzer for timing analysis in this step. The static timing analyzer differs from a simulator primarily in that the analysis is independent of the stimulus applied. A static timing analyzer determines the path delay for every timing path in the design, and analyzes each for timing violations similar to those identified by the simulator. Timing problems are identified by timing violation reports. One advantage of the static timing analyzer versus the simulator is that it identifies timing problems and critical paths that are not sensitized by the input stimulus. A disadvantage is that it often reports many false timing errors that would not occur in the actual module.

Critical path analysis using a static timing analyzer provides a quick approach to finding timing problems in the design, such as setup or hold time violations. It does not find all timing problems, especially those involving complex CPLD or controller designs. A worst-case timing analysis that uses a comprehensive set of functional test vectors may be required to broaden the timing analysis.

8.1.1.10 Perform Power/Loading Analysis

Power analysis determines the module's electrical power requirement. There are maximum power specifications for a module, usually based on its size, which cannot be exceeded. If power specifications are not met, some high-power components may need to be replaced with functionally-equivalent, low-power devices.

Loading analysis determines the electrical loading on the outputs of module devices, and includes DC and AC components. A device's DC loading is affected by DC current from other devices that flow through its output. A device's AC loading is related to the capacitance presented to its output due to the capacitance of other devices and interconnections connected to it. Excessive DC loading can cause power dissipation to exceed device specifications; excessive AC loading can cause timing problems and signal distortion.

We send the results of the analyses to various other members of the overall subsystem team: thermal analysis data to mechanical engineers involved in the design of the cooling system; and power and loading data to the backplane designer specifying power supplies and interconnecting modules in the system.

8.1.1.11 Perform Functional Timing Simulation

We use simulation to confirm with a high degree of certainty that the first version of a module will function as desired without modifications. We can work our problems with the design on the simulation without the cost of producing a prototype, finding the problems, and reworking the module. We can also assess various design approaches without building a module.

The purpose of this step is to simulate the module to test its basic functionality, using the previously-generated test vectors. ASICs, if present, will be modeled behaviorally or with hardware models. We use worst-case timing values for all components. We also broaden the timing analysis by performing a worst-case minimum/maximum timing analysis using test vectors and the simulator.

For the timing analysis, we use the simulator to simulate the minimum and maximum delays of components simultaneously. When an input to a component changes, two transitions are scheduled on the component outputs. The first transition is to an unknown state and occurs with the minimum propagation delay. The second transition is to the final state, which occurs with the maximum propagation delay for the component.

Simulation output reports all functional timing hazards, such as setup and hold-time violations, illegal edge violations, and excessively 'long' signal paths based on user specifications. Also reported are structural timing hazards (circuit problems arising from the structure of the circuit).

Once the gate-level design is complete, we must verify the accuracy of the design. The first step is to perform a gate-level functional simulation. This is similar to the behavioral model functional simulation performed previously, except that we use the gate-level design database for simulation rather than the VHDL model. The functional simulation is also concerned with the timing of the gate-level design. The purpose is to verify the

functional correctness and timing of the logic.

Mentor's QuickSim II has a mixed-mode capability that allows VHDL and gate level representations to be simulated in the same design simultaneously. Because of this, it will probably be common to perform this step in parallel with the interactive logic design. We compare the output of the detailed gate-level simulation with the previously-verified behavioral simulation to determine the functional correctness of the detailed design. We must isolate discrepancies between the two models and correct them until both simulations produce the same results.

We can design a subset of the entire VHDL model at the gate level and verify it using the test vectors we developed during the behavioral model functional simulation. Once this function is verified, we design the next function at the gate level and verify it, and so on until the entire design is complete.

8.1.1.12 Select Parts

This step is required to control parts usage on RASSP module designs. Prudent part selection by the module designer enables designers to take advantage of economies of scale in purchasing; it also enables lower inventory costs to the company and its customers. Procurement should be consulted for availability of long-lead items. Also, some new parts are introduced in the literature well before they are actually available for purchase. Procurement or the local components engineering group should be consulted regarding the availability of new parts. We must also consider the cost of parts chosen for the design to meet cost goals.

8.1.1.13 Generate Production Test Vectors

In most cases, we augment the functional test vector set used in functional simulation of the module to support fault simulation and the eventual needs of production test.

8.1.1.14 Perform Thermal Analysis

Thermal analysis identifies problems with heat caused by the power dissipation of the module components. Mechanical engineering sets requirements for heat distribution across a module, and if any areas on the module violate this requirement, we may need to move components to more evenly distribute the heat.

8.1.1.15 Perform Fault Simulation

A fault simulation indicates how well the stimulus tests the module's interconnections for manufacturing. It can also provide information on the set of potential failures based on a particular failure indication (e.g., pattern of incorrect outputs). The fault simulation provides this 'set of potential failures' through a fault dictionary. A fault dictionary lists faults for each output vector that would cause that output vector to fail. The intersection of all sets of potential failures for each failed vector indicates the set of potential faults on the module under test.

The results of a fault simulation often indicate design areas that are inadequately tested and that require additional or modified stimulus to fully test. We use this information to create an adequate set of test vectors to fully test the module and also to identify a reasonably small set of potential failures through the fault dictionary. The test engineer who constructs the module test program uses these vectors, and the associated fault dictionary, to construct the test program concurrently with the module design.

The fault simulation process involves running multiple logic simulations, but each simulation has a fault introduced into the design before it begins. If the response from the faulted simulation differs from the response of the ideal simulation, the fault is either detected or potentially detected. The fault is detected if the response of the ideal simulator was a ' 1 ' while the response of the faulted simulator was a ' 0 ' or vice versa. The fault is potentially detected if either simulator's response was unknown while the other was known. The fault coverage is the percentage of all faults that are detected.

Many types of faults, such as opens, shorts, and faults internal to components, are possible in the physical

module. It is impractical to simulate all types of possible faults, so we usually limit a simulated module fault to a pin on the module that is grounded or connected to power.

If the fault coverage for the set of test vectors does not meet project requirements, we generate additional test vectors to meet the requirement.

8.1.1.16 Generate Module Preliminary Design Document

The Module Preliminary Design document includes all information pertinent to the module design. This document has sufficient data for any engineer or technician who must later work with the module. For example, test engineers use it to debug the test program, and manufacturing uses it to repair boards.

A designer should include at least the following data items:

1. Parts List
2. Block diagram
3. Logic schematic
4. Module layout
5. Timing diagrams
6. Functional Description
7. FPGA description
8. ASIC description
9. Results of all analyses (power, thermal, timing, reliability, testability, etc.)
10. Interface Protocol Timings
11. Power Requirements and Consumption
12. Simulation Results and Data

8.1.1.17 Conduct Preliminary Module Design Review

After completing the module design and before committing the design to fabrication, we conduct the Preliminary Design Review. At a minimum, the project manager, the architecture designer, the module and ASIC designers attend. This review provides a detailed critical technical evaluation of the module design and resolves any outstanding issues. Testability features are evaluated. We review any ASIC or FPGA elements within the module. All disciplines that interact with module design should have representatives present at this review.

8.1.2 Final Design Phase

The final design phase addresses the transition from a functional design to a physical design. During this phase, we accurately place and route the module and analyze the parasitic effects of the module on signal integrity. The simulations performed address actual signal loading, signal crosstalk, line reflections, additional line delay, and skew. We finalize the test vectors and conduct the Pre-release Design Review before releasing the documentation and analysis results.

8.1.2.1 Module Place and Route

In this step, we determine the location of the parts on the module based on the preliminary placement completed previously. The interconnections between parts (routing) are also made. In some cases, we must conduct placement trade-offs due to routing concerns. We may have to move parts to minimize the length of critical interconnections such as clock lines.

8.1.2.2 Update Module Preliminary Design Document

We update the Module Preliminary Design Document and other related documentation to reflect the results of the behavioral simulation and the electrical analyses performed on the gate-level design. We add the final verified version of the behavioral model to the document. We also update other sections of the Preliminary

Design Document that have changed. Such items may include the functional block diagram and other details that might have changed as a result of the modeling and analyses.

8.1.2.3 Perform Final Design Functional Timing Simulation

This functional simulation is identical to that performed in preliminary design, except that we use the design database back-annotated with layout parasitics for final verification.

8.1.2.4 Perform Final Design Thermal Analysis

This thermal analysis is identical to that performed in preliminary design, except that we use the final layout.

8.1.2.5 Perform Final Design Critical Path Analysis

This critical path analysis is identical to that performed in preliminary design, except that we use the design database back-annotated with layout parasitics for final verification.

8.1.2.6 Perform Production Timing Simulation

Module layout parasitics (trace length, number of vias traversed, signal crosstalk, trace parallelism, transmission line effects, etc.) can significantly effect module timing. This step allows the predicted delays caused by layout parasitics to be back-annotated to the simulator for a final simulation run that includes the best known timing information.

We use a transmission line analysis tool at this point. This tool predicts delays, waveform distortion, and ringing due to transmission line effects. Crosstalk, the coupling of signals from one interconnection path (net) to another, is also evaluated with a suitable tool. The engineer can correct these problems by adding terminations, changing parts placement, and making other changes. We must evaluate the impact of any design changes on the module's cost, manufacturability, reliability, and testability.

8.1.2.7 Generate Module Test Procedure and ATE Test Vectors

In this step, we generate the module test procedure document and test database. The test procedure contains the step-by-step procedure for testing the module. It includes such details as module tester setup, assignments of tester pods to module I/O signals, and filenames of test vector files. The information in the Test Plan produced previously can be included in this document. Test engineering and manufacturing should review the test procedure document.

Also in this step, we produce the test database, which is the set of test vectors/expected responses in the correct format to drive the module tester. If the simulation is properly designed, in this step we simply reformat the existing simulation vectors. It is therefore important to design the simulation so that the simulation vectors can be easily used for module test.

We use the module test procedure and module test database, along with the post-processed vectors, to generate the necessary files and procedures for the target tester. We then verify the test program on the target test system. Any changes that may be necessary are reflected back into the module test procedure and/or simulation database. Any conceptual changes are noted in the Testability Document and the Design-for-Testability Design Review Checklist, if necessary.

8.1.2.8 Design and Build Test Adapters

The test engineer is responsible for specifying the electrical requirements for any hardware used to interface the module to the target test system. Depending on program requirements, this hardware may consist of either a totally unique design, or more generally, a generic test fixture design that is common to several module designs, plus unique interface hardware that provides module-specific signal routing, stimulus conditioning and output loads.

8.1.2.9 Generate Module Artwork and Manufacturing Tools

In this step we produce the tools for board fabrication. Artwork is used to generate printed circuit interconnection information. Other tools are needed to control equipment to drill holes in the boards, etc.

8.1.2.10 Prepare Module Release Package

We prepare the module drawing package, designated a 'Module Release Package.' This package contains all pertinent data for building the module and is released to internal manufacturing or an external supplier. The documentation package must fully specify the module to the manufacturing organization. It includes assembly drawings, logic schematics, parts list, and machine tool data.

8.1.2.11 Conduct Pre-Release Design Review

The Pre-release Design Review is the last design review of the completed design of the module before first-item production. This design review must verify that all architecture and module functional requirements have been met. We examine the results of all simulations and address any concerns, including all critical path and timing analyses, and all test-related issues, such as design-for-test and fault coverage. We check all details of the engineering documentation being released. The procedures to follow for this review are found in the local site Design Review Policy.

We evaluate the estimated module recurring engineering costs against cost goals and report actual productivity data at the review.

8.1.3 Hardware Fabrication, Assembly, and Test Phase

Engineering fabricates the first-item module, inserts components, and tests the module.

Fabricate First-Piece Module - Manufacturing or an outside supplier fabricates the first-piece module.

Assemble First-Piece Module - We assemble the first module by inserting components in this step.

First-Piece Module Engineering Test - During this step, we test the first module using the module test equipment according to the test procedure document. We perform design verification in which we check timing, performance under variations in voltage and temperature, and other design performance criteria. Design verification determines if the design will meet environmental requirements. If any problems with the design are discovered, the module design engineer makes any design changes needed to correct them.

We test and characterize the module on an automatic tester. All tests are performed over the full voltage and temperature ranges. Module characterization measures all electrical parameters. We analyze the data to determine module quality and capability.

The ATE test vectors used in the functional and parametric tests are finalized to incorporate any adjustments needed to satisfy module, adapter, and tester interfacing anomalies. We analyze module test results to determine the integrity of the module-to-tester interfacing.

Update Documentation and Release Package - The first-piece module engineering test may reveal deficiencies in the design that need to be changed. If this is the case, then a documentation and Release Package update is needed.

Conduct Final Design Review - The FDR is the last design review of the completed engineering drawing package for the module. We check all details of the engineering documentation being released. This review is crucial, because any errors/discrepancies in the release package that are not found will affect production of the final system. There may be a large number of this type of module in a system, so the effects of an incorrect release package can be great.

Engineering Signoff - After the FDR, engineering signs off the drawings for release to the drawing system. The module design cycle is now complete.

8.2 ASIC Design Process

This process, similar to module design, has three major phases. In the preliminary design phase, we generate a detailed specification for the device based on the architecture design effort. In this phase we develop, simulate, and verify a behavioral model, and develop a gate-level design (manually or via synthesis) and functionally-verify of the design via simulation. In the second phase, we translate the gate-level design to the supplier format, and perform all supplier-related simulations and pre-release procedures.

In the third phase, we fabricate and test the chip with the ASIC supplier. The ASIC design process is detailed in Figure 8 - 2.

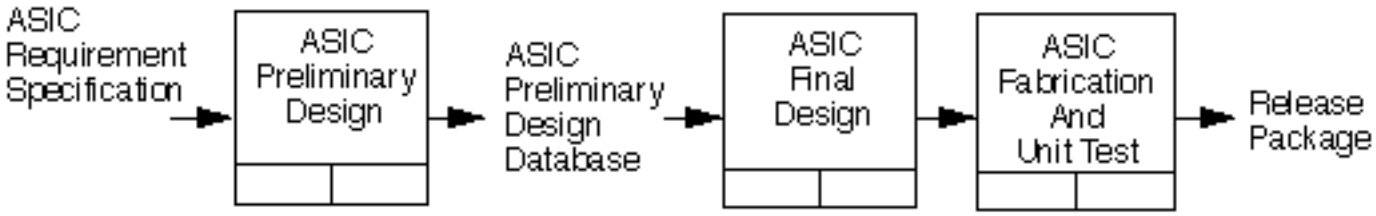


Figure 8 - 2: ASIC design process flow

Critical to success of the ASIC design process is generating an accurate specification, following DFT practices, performing multi-level simulations, and conducting thorough design reviews.

8.2.1 ASIC Preliminary Design Phase

We start the ASIC design process with preliminary analysis and review of the ASIC Requirements Document compiled by the architecture and module designers. We search the RASSP design reuse library concurrently with preliminary analysis after reviewing the ASIC Requirements Document. If the library search reveals that no existing ASIC meets the requirements, we assess the feasibility of implementing the defined ASIC, and develop the overall architecture of the ASIC. This includes defining and sizing a high-level functional architecture, defining the I/O interface signals, developing the instruction set architecture (when applicable), and defining special test features. Technological, packaging, and cost constraints are considered.

We may have to conduct tradeoffs with architecture and module designers following this analysis to achieve the desired ASIC requirements. Manufacturing and mechanical engineering should be consulted to ensure the proposed design can be produced within project, technology, and cost constraints. Reliability engineering should be involved, along with the supplier, to ensure that the fabrication process and packaging meets specified reliability requirements. The test engineer can provide information on the IC tester requirements and limitations.

- Search RASSP Design Reuse Library - We search the RASSP design reuse library concurrently with preliminary analysis/design to determine if an existing (legacy) design meets the requirements specified for the target ASIC. This step is most likely to be performed during the initial proposal phase of a design, when the subsystem designer attempts to configure the subsystem and calculate the cost.

- Generate the ASIC Design Specification - We generate an ASIC Design Specification that specifies the functional, timing, testability, and testing requirements for each unique microcircuit. This specification should combine the ASIC requirements from the subsystem/module design with the results of preliminary analysis and design to produce a complete description of the ASIC for use in behavioral and gate-level design.

- Develop ASIC Behavioral Model - All ASICs are an integral part of a larger system and perform some unique subsystem function. In some cases, the architecture designer modeled the unique subsystems using a high-level hardware description language to gain confidence that all functional blocks will interact properly. In these cases, each subsystem partition (i.e., the chip definition) should have had a high-level model developed that can be used by the ASIC designer as the starting point for behavioral model development.

  When developing the behavioral model, the ASIC designer should reflect the inherent structure of the design. If, for example, the ASIC has a pipelined architecture, the pipelines and the functions performed by each pipeline stage should be evident from the model. The purpose is to produce a model with sufficient detail to guide gate-level design.

  In this step, we also develop the 'tester' model, which applies input vectors to the model and collects and analyzes the results. This may be a simple model written in the same language as the behavioral model, or a complex suite of programs integrated into a complete simulation environment.

- Generate ASIC Test Plan - The ASIC designer prepares a detailed test plan that has two functions. First, it details the step-by-step procedure to verify the functional operation of the behavioral model. This includes the function(s) to be tested, the method and order of testing, and the data to be used. Second, it details the procedures to test the final ASIC devices on an IC tester. This includes all parameters to be tested, the test conditions, the vectors to be used, and the test limits. Timing diagrams for all inputs are included in a format compatible with the target tester. The test vectors are developed during chip simulation and provided to the test engineer along with the Test Plan document.

- Generate ASIC Test Vectors - We usually perform this step concurrently with the behavioral model functional simulation. Using the test plan as a guide, the ASIC design engineer generates vectors to test the functionality of the behavioral model. These vectors are used later in the gate-level and fault simulations.

- Perform Behavioral-Level Functional Simulation - We simulate the behavioral model of the chip to verify that its performance meets the architecture and module requirements. This effort may be performed jointly by the module design engineer and the ASIC design engineer to ensure full coverage of all architecture and module requirements. It is quite common for the module designer to discover design flaws in the module simulation that are difficult to detect in a single-chip simulation. This may include the detailed interaction of the ASIC during module reset or instruction-abort procedures. By providing the module designer with the latest version of the behavioral model, potential errors can be discovered before module integration and test.

- Update ASIC Design Specification - We update the ASIC Specification as required to reflect the results of the behavioral simulations. In particular, we add the final verified version of the behavioral model to the ASIC Design Specification. In addition, we update any other sections of the ASIC Design Specification that have changed, including the functional block diagram, the instruction set architecture, and other details that may change during the model development.

- Perform Interactive Logic Design/Synthesis - The ASIC design engineer performs the detailed logic design, either by capturing the logic interactively or by using logic optimization and synthesis tools that take VHDL as an input. Logic design usually follows a bottom-up approach, where we develop and verify simple logic functions, and create successively larger functional blocks from these smaller blocks.

- Perform Testability Analysis and Generate Additional Test Vectors - In parallel with the interactive logic design, the ASIC design engineer performs a testability analysis on the preliminary netlist. The purpose of this step is to identify sections of the gate-level design that, because of controllability or observability constraints, may be difficult to test. The ASIC designer may then wish to develop

dedicated test logic for these problem areas.

If additional logic is developed in this step, the ASIC designer updates the behavioral model to reflect the new functions, and develops additional test vectors to test the new logic. These vectors, combined with the behavioral vectors, form the nucleus of the final set of test vectors for the chip, and fully exercise all functions of the chip. The updated behavioral model is sent to the module engineer for module simulation.

- Perform Gate-Level Functional Simulation - Once the gate level design is complete, we must verify the accuracy of the resulting netlist. The first step is to perform a gate-level functional simulation. This is similar to the behavioral model functional simulation performed previously, except that the gate-level netlist rather than the VHDL model forms the simulation database. The functional simulation is not concerned with the timing of the gate-level design. The purpose is to verify the functional correctness of the logic.

  We use a simulator that has a mixed-level capability that allows VHDL and gate-level designs to be simulated in the same design simultaneously. Because of this, it will probably be common to perform this step in parallel with the interactive logic design and synthesis. We can translate a subset of the entire VHDL model to a gate-level netlist and verify it using the test vectors developed during behavioral model functional simulation. Once this function is verified, we can translate the next function and verify it until the entire design is complete. The final gate-level design produces the same results as the behavioral model for a given set of test vectors.

- Compare Behavioral Versus Gate-Level Simulation Results - The ASIC design engineer compares the output of the detailed gate-level simulation with the previously-verified behavioral simulation to determine the functional correctness of the detailed design. The ASIC design engineer must isolate and correct discrepancies between the two models until both simulations produce the same results.

- Perform Gate-Level Timing Analysis - The ASIC designer performs a timing analysis simulation on the chip design, or portions of it. The designer compares the results of the analysis with the requirements imposed by the device specifications. There are two ways to assess the performance of the electrical design. The first method is to use a critical path analysis tool to identify critical paths. The second method is to put timing information in the gate-level models and perform functional simulation. This can identify setup, hold, and other timing violations.

  In this step, we assume accurate timing information is available from the supplier, including loading and wire length estimates. In addition, many suppliers have special macrocells or megafunctions as part of their cell libraries. To perform accurate timing analysis at this stage in the design cycle, the timing analyzer must have access to the timing information for any supplier specific functions. If this is not the case, it may be necessary to delay this step until after the design has been translated to the supplier format.

  Since the timing results from this stage are only estimates, the designer should not attempt to squeeze the last nanosecond out of the chip performance in this step. The purpose is to identify obvious problem areas, and make the changes necessary to get the performance as close to the requirements as possible. The detailed timing analysis will be done on the supplier toolset.

  There are usually be two types of changes required to increase performance. The first type is fine-tuning the gate-level design, such as replacing a carry look ahead adder with a carry select adder. We can make this type of change to the netlist only, since the basic structure and functionality are not affected. The second type of change is a major architectural restructuring, such as adding a pipeline stage. We make this type of change to both the netlist and the behavioral model. In both cases, we rerun the functional simulation to verify that the changes did not affect the functional operation of the design.

- Perform Fault Grading/Fault Simulation - The ASIC designer performs a fault analysis on the

functional test vector set, and generates additional vectors until the final set of test vectors provides some predefined fault coverage. As in module design, fault simulation serves two purposes. The first is to provide some quantitative measure of the quality of the test vector set. The second is to obtain a set of vectors that detect flaws in the fabricated device.

- Conduct Preliminary Design Review - After completing the ASIC design, and before committing the design to the supplier for pre-layout simulation and signoff, we conduct a PDR to provide a detailed critical technical evaluation of the ASIC design and to resolve any outstanding issues. All disciplines that interact with module design should have representatives present at this review.

- Translate Design Database to Supplier Format - Most suppliers require the netlist and test vector set in a particular format. We can translate at either the designer's or the supplier's facility. The ASIC designer must also generate any other data about the chip required by the supplier.

- Perform Preliminary Layout (Floorplanning) - Many suppliers allow the designer to perform a preliminary layout or floorplan at this stage. This allows the designer to place major functional blocks of logic on the die, and to assign the ASIC I/O signals to the physical pins of the ASIC package. This step serves two main functions. First, it provides the supplier with a routability analysis of the ASIC. This determines how easy or difficult it will be to perform detailed layout. Most suppliers require a certain routability number before they accept the design for layout. The ASIC designer may have to delete gates or use the next larger die size if this number cannot be reached. Second, it provides updated delay results based on estimated wire lengths from the placement of the functional blocks of logic and the location of the I/O pads. This new delay information provides more precise estimates of the chip performance.

- ASIC Supplier Simulations and Other Supplier Specific Steps - All suppliers require a set of simulations and procedures to be completed before accepting a design for layout. At a minimum, this includes a functional simulation, and timing analysis using the estimated wire length data from the floorplanning performed in the previous step. The functional simulation verifies the functional operation of the design by comparing the supplier simulation results with those from the simulator being used by the designer. The timing analysis includes critical path analysis and setup and hold-time violations for minimum/maximum conditions. In addition, the I/O delays can be calculated with various external loading parameters.

  The exact set of requirements varies according to the supplier, but the following additional simulations are commonly required. Design verification simulation checks the design for proper design rule use. A test file is usually produced that converts the test vector set into a format compatible with the supplier's test facility. A toggle test verifies that the vector set toggles every net in the design at least once. The parametric simulation is used to verify proper voltage levels and/or drive current in the I/O signals.

- Compare Supplier Simulation Results to Contractor Results - We perform this step to correlate the supplier's simulation results with our simulation results to verify functional equivalence across the two sets. Discrepancies should be isolated and corrected so that the two simulation environments produce the same results.

- Conduct Pre-Release Design Review - We review the final design to verify that all subsystem and module functional and performance requirements will be met. We examine the results of the supplier simulations and discuss any concerns. This includes all performance-related issues, including special critical path routing requirements, minimum/maximum timing, and special layout concerns such as the clock tree. The Module Designer examines the final bonding diagram, and the supplier certifies that all pre-layout requirements have been met.

- Prepare ASIC Release Package - The ASIC designer prepares a Release Package containing all the information necessary for the supplier to fabricate, assemble, test, and deliver the ASIC devices. A minimum set of requirements for this document should include a netlist, test vector set, bonding

diagram, packaging information, quality level of the ASIC (MIL, Commercial, etc.), performance requirements, and I/O drive capability.

8.2.2 ASIC Final Design Phase

- Perform Detailed Layout - We perform the detailed layout at the supplier's design center or fabrication site. Using the information from the netlist, the floorplanning, and the bonding diagram, we place and route the detailed netlist using an automatic layout tool. This process does not actually result in the physical device, but in a description of the final ASIC routing mask. This provides detailed, accurate data on the lengths of every net in the design to be used in the next step.

- Perform Post Layout Simulation - Using the detailed wire length data from layout, the ASIC designer reruns the functional and timing analysis simulations to verify that all performance requirements are met. This step is basically a repeat of the ASIC supplier simulations performed previously; the only difference is that we replace the estimated wire length parasitics with the actual layout parasitic data.

  If the performance does not meet the requirements, the designer makes changes and sends the new netlist back to the supplier for layout. This process continues until the performance is acceptable.

8.2.3 ASIC Fabrication and Unit Test Phase

- Fabricate, Assemble, and Test ASIC Device(s) - The ASIC supplier fabricates, assembles, tests, and delivers the specified number of prototype devices.

- Perform Final In-House Test - The ASIC test engineer, using an in-house IC tester, performs final test and characterization procedures on the prototype devices according to the ASIC Test Plan. The vectors used for this step should be the same as those supplied to the supplier.

8.3 FPGA Design Process

This section discusses the Field Programmable Gate Array design process which is becoming increasingly popular among designers of Digital Signal Processing hardware. This process has not been "re-invented" by the RASSP program, but rather reflects current industry standard methods and COTS tools.

The FPGA process described here is kept generic intentionally and may be applied to various classes of programmable logic devices including Field Programmable Gate Arrays, Complex Programmable Logic Devices (CPLDs), and Erasable Programmable Logic Devices (EPLDs). The inputs to the FPGA process are a Requirements Specification, which usually is a part of a higher-level module (board) Requirements Specification and a VHDL behavioral model of the FPGA function. Major outputs from the process are a fusemap used to program the device and test vectors used to verify the functionality of the design. The design can be divided into three phases: Preliminary Design, Final Design, and Programming and Unit Test. These phases depicted in Figure 8 - 3 are described in the following paragraphs.

Figure 8 - 3: Phases of FPGA design

8.3.1 FPGA Preliminary Design

Functional Design --- The purpose of this step is to refine an existing VHDL behavioral model of the FPGA function in sufficient detail so that it can be synthesized down to the specific programmable function unit level. The VHDL behavioral model is generally provided by the architecture definition

process or by the module/MCM design process. The FPGA requirements document should contain the behavioral model, any requirements derived from module design, such as real estate restrictions, timing requirements, power restrictions, or any other module-related information.

Three styles of design entry using VHDL are possible: behavioral, Boolean, and structural. Tabular descriptions are also possible. A complex VHDL design can be built by successively combining building blocks in related layers. Up to this point the designer has not implied any particular hardware. The assumption is that he has captured the scope of the design into one abstract level VHDL description. This description may be realizable in a broad range of hardware devices. Memory, interface, microprocessor, one or more programmable logic devices, and a variety of other integrated circuits may be required to realize the design. Therefore, the design must be decomposed into those specific devices which the designer chooses for realization.

Using a CAE tool the designer can explore the variety of device implementations by interactively varying design criteria - including cost, preferred parts, device count, speed, and power - to find an optimum design solution.

Assuming that partitioning has been performed and a particular FPGA has been selected for a certain function, further decomposition tasks must be performed. Clocks may have to be connected to a clock distribution network and high fanout inputs must be connected to high drive input cells, if desired.

Functional Verification --- During this step the refined FPGA behavioral VHDL model is verified to ensure that it meets functional and performance requirements. This is accomplished by performing a functional simulation using the VHDL FPGA model. The Testbench used for this verification should be the set of test vectors passed down from module design or architecture design, usually augmented by the FPGA designer. Outputs from the simulation are compared to the expected values passed down from the module designer or architecture design.

8.3.2 FPGA Final Design

Logic Design / Synthesis --- After functional verification of the behavioral model the FPGA is synthesized from the refined behavioral model. Synthesis is the transformation of the design input at a high level of abstraction to a lower level. The VHDL description is turned into an unoptimized Boolean level description of the design. This unoptimized Boolean description is flattened, that is, it has no hierarchy and all intermediate variables are removed. At this point the design is not device-specific and is in a sum-of-products form. Next, the synthesized design is transformed into a device-specific netlist. This netlist is passed to the place and route tool.

Placement and Routing --- This step involves the actual placing and routing of the design into the actual device. The place and route software accepts the output of a technology mapper and generates a file for use by the programming equipment and a timing model for use by downstream analysis tools. The place and route determines the actual cell choice for each logic function to be implemented and the wire connections for the signals which pass and from the logic cells and I/O cells. Once the place and route is completed, a performance analysis is usually done to verify that the design will meet expected performance.

Functional and Timing Simulation --- After place and route of the target design, it must be resimulated with backannotated delays to verify that it still functions as expected. A full timing gate-level simulation model is assembled using the backannotation delay information. Test vectors generated for verifying the behavioral VHDL model are used to exercise the model. The resulting post-route simulation outputs are compared against the simulation results generated during the functional model verification. Once validated, the full-timing gate-level model can be made available for use in module-level simulations.

Fault Simulation --- A fault simulation can be optionally performed on the FPGA gate-level design to detect stuck-at-one, stuck-at-zero fault conditions. The procedures for this fault simulation are the

same as those for the ASIC design process.

Refine Test Plans/Procedures --- The detailed test plan and test procedures for the FPGA are refined as necessary. The plan describes the method and the procedures define the step-by-step the instructions to be used to test the device standalone, before it is inserted onto a module.

Conduct Design Review --- This design review is optional, and usually is made part of the module design reviews. This review provides a detailed technical evaluation of the FPGA design function and performance, and serves as the focal point for identifying any outstanding issues. The design review is conducted in accordance with local policies and procedures. A detailed design package is generated by the designer for inclusion in a module-level design review.

8.3.3 FPGA - Program Device and Unit Test

After the FPGA or module design review, the devices can be programmed using JEDEC files that are downloaded to the device programmer. Then the individual devices can be unit tested using the same set of test vectors and expected outputs that were used during functional, timing, and fault simulations.

8.4 Backplane Design Process

The architecture design activity provides the basic requirements for backplane functions. The purpose of the backplane design process is to determine how to interconnect the modules to perform those functions.

There are several terms which are used interchangeably when describing backplanes; these terms include "backpanel" and "motherboard". Although "backplane" and "motherboard" are sometimes used to differentiate between wire wrap and printed wiring, respectively, there is no industry wide standard for these terms. This document will refer to "backplanes" only.

Commercial or Industry Standard backplanes, such as VMEbus or Multibus II, are backplanes which have been manufactured to a specification or standard, with some or all pins predefined, and are available off the shelf.

The backplane design process described here applies to the design of custom backplanes of varying complexities with wirewrap and/or PWB interconnections. It may also encompass the design of backplanes that use off-the-shelf or modified, off-the-shelf backplanes as the basis for a new design. The backplane design process flow is detailed in Figure 8 - 4.

Figure 8 - 4: Backplane design process flow

8.4.1 Backplane Preliminary Design

We begin the backplane design process by forming a design concept, which is then documented in a Backplane Specification that describes how the backplane design engineer plans to implement the required functions. If the design concept calls for any new or modified modules, the backplane design engineer provides mechanical engineering with preliminary mechanical requirements at this stage. Backplane designs should, however, use previously-designed backplanes to the greatest extent possible.

Partitioning can significantly affect the size and complexity of a backplane. Poor partitioning can increase the

number of layers and thus the cost of the backplane, while good partitioning can efficiently use the available space and minimize the number of layers or wiring. Poor partitioning can also lead to noise problems due to long signal runs or wire lengths.

A commercial/industry standard backplane fits the COTS philosophy. It requires no manufacturing time and minimal lead time, and a standard backplane can be chosen so that interface problems in a system can be minimized. It does, however, limit the user to the system functions that are provided in the standard.

Commercial or Industry Standard backplanes have many pins predesignated, and some that are user-defined. It is important for the designer to understand which of the pins are predefined by the standard, and which are left for the backplane designer to designate; obviously, the designer should not use any of the standard-defined pins for uses other than what the standard intended.

- Perform Backplane Detailed Electrical Design - We complete this step when the backplane design engineer determines the partitioning of printed circuit interconnects and wirewrap interconnects on the backplane and establishes a power distribution approach. With this information in hand, the engineer uses a workstation to interactively perform the preliminary partitioning of the backplane schematic design. It is at this stage that the final details of the schematic design must be captured. Signals must be analyzed to determine the signals to be interconnected by PCB and the signals interconnected by wirewrap. Module preliminary placement should consider such things as estimated power dissipation by the modules, voltage requirements, and interconnectivity.

  Detailed design work proceeds at this point, based on the design concept. This task includes input/output definition and the design of timing/control functions.

- Develop Backplane Test Plan and Test Procedures - We generate a detailed test plan and procedure for functional and parametric testing. The plan should contain strategies for all test types, parameters, methods, conditions, and special resources required. Evaluation procedures should include critical signals to be tested and characterized, ands signals to be checked for cross-talk, reflections, etc.

- Perform Preliminary Electrical Analysis - Preliminary electrical analysis consists of evaluating and simulating critical signals on the backplane based on loading characteristics, worst-case signal lengths, and signal interaction. This step helps determine module placement, drive characteristics, and interface requirements by allowing early characterization of critical signals.

- Generate PWB/Wirewrap Netlists - In this step, we extract preliminary PWB and wirewrap netlist data from the schematic design database created during backplane interactive logic design. The objective is to extract all design data from the schematic design database and to generate an ASCII file representing the design.

- Conduct Backplane Preliminary Design Review - After completing the detailed design of the backplane, and before committing the design to PWB layout or wirewrap wiring, we conduct a Preliminary Design Review. Participants include the project manager, the subsystem design engineer, the local Design Review Board, and the backplane design engineer. In addition, all disciplines that interact with backplane design should have representatives present at the review.

  The purpose of this review is to ensure that the functional requirements of the backplane specification have been met. It will provide a detailed critical technical evaluation of the functional electrical design, and we will resolve any outstanding issues.

8.4.2 Backplane Final Design

If the backplane design includes a PWB section, we perform the basic placement and routing steps as in module design. We perform a detailed electrical analysis to verify that proper design rules were followed in devising the interconnection scheme. We then generate artwork and tools for the PWBs, performed as in module design.

- Design Printed Wiring Board - Designing the PWB portion of the backplane includes optimizing placement of the module connectors and the interconnecting printed wires. The complexity of this dictates whether we apply rules during place and route or during the analysis process after routing. The activities are a joint effort between the backplane design engineer, the PWB layout specialist, and the subsystem engineer.

- Perform Electrical Analysis of PWB - Effective backplane system design requires balancing signal density against electrical performance, cost, mechanical reliability, and producibility. As the clock frequency increases and signal rise times decrease, backplane interconnections become more complex and can no longer be considered as 'short circuits.' A basic understanding of transmission lines is vital to analyze and improve these interconnections.

- Generate Artwork and Manufacturing Tools for PWB Fabrication - We generate the artwork and manufacturing tools necessary for fabricating the PWB.

- Perform Wire Wrap Place, Route, and Electrical Analysis - We perform this step to convert the wirewrap netlist into a connection list without routing and levels. The primary function of this processing is to break up the list of pins in each net into a discrete set of wires. We perform circuit analysis to verify that the resulting circuits are functionally correct, i.e., each net contains one output, clock wire length is not less than the longest wire, etc.

  Once we have processed the wirewrap netlist, we route and level the wires using the user-specified rules for optimizing parallelism, line reflections, and wiring density. Wire routing entails searching for the optimum routing path while considering the electrical interaction of wires that were previously routed. For wires that fail the allowed criteria for parallelism, the design engineer can automatically convert the single wires to grounded twisted pair, or manually designate a different route for the wire.

  After we complete routing and leveling for each wire, the wire connection list contains the necessary information to generate manufacturing tools.

- Conduct Backplane Pre-Release Design Review - The Pre-Release Design Review is the final design review of the completed backplane Release Package. All details of the engineering documentation being released are checked. The review is crucial, since any errors or discrepancies in the release package that are not found affect the fabrication and assembly of the first-piece backplane.

8.4.3 Fabricate, Assemble, and Unit Test Phase

- Fabricate and Assemble First-Piece Backplane - This step is a manufacturing function, not an engineering function.

  Once the tools are generated, manufacturing uses the NC data and reports to assemble the backplane. Manufacturing either purchases or manufactures the bare board, and mounts the pins or connectors on the board. An objective of this step is to add the wire-wrapped wires to the backplane and, optionally, to test the wires as they are added to ensure that the proper connections were made. Depending on the design requirements, we test the backplane for continuity and isolation at various stages of assembly. It may be tested before wire installation, after initial hand wires and automatic wires are added, after all the wires are installed, or at all of those stages. Following the completion of this step, we test and debug the first-piece backplane.

- Perform First-Piece Backplane Engineering Evaluation - During this step, we test the first backplane using lab test equipment according to the Test Procedure document. Design verification is also performed, in which we evaluate the backplane for crosstalk, line reflection characteristics, the effectiveness of the power distribution system, settling time, and signal overshoot/undershoot.

- Update Documentation and Backplane Review Package - This step is used when a design engineer

needs to change either the backplane PWB design, the wiring design, or any ancillary documents, such as the Test Plan, descriptive documents, etc. We perform it after those documents are released into a controlled system and it is usually done during and after the first-piece test and debug, or any time thereafter.

- Conduct Final Design Review - After completing the backplane design and evaluation, we conduct a review to provide a detailed critical technical evaluation of the backplane design and to resolve any outstanding issues. All disciplines that interact with backplane or module design should have representatives present at the review.

8.5 Chassis Design Process

We define a chassis as the top-level hardware assembly containing the electronics and mechanical components of a subsystem such as a digital signal processor. A subsystem could be made up of more than a single chassis. Across the industry there is no standard terminology in this area. Other terms used to describe this hardware are "subassembly", "frame", and "box". A chassis usually contains at least one "backplane" which provides the interconnection for enclosed electronic printed wiring boards (also designated "modules" or "circuit card assemblies") as described in the following sections. The chassis design process can be broken up into two major phases which we designate Preliminary design and Final design. These processes are described in the following paragraphs and depicted in Figure 8 - 5.

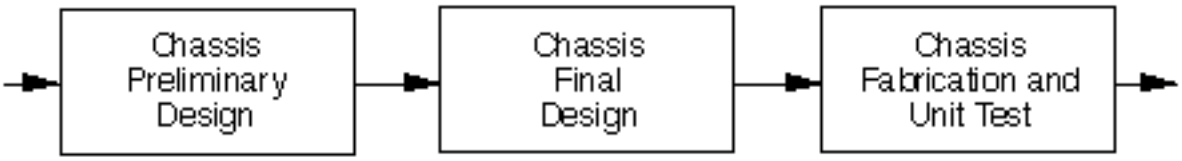


Figure 8 - 5: Phases of Chassis design

Prior to chassis Preliminary design the mechanical and electrical designers review the proposal concept solution against the latest revised requirements. A new look is taken at the previous design and sub-contractors or co-contractors are consulted to explore the advantages of common approaches to packaging design and to determine that requirements that affect packaging have been met.

8.5.1 Chassis Preliminary Design

- Select Parts, Materials and Processes - The mechanical engineer prepares a list of the chassis parts, materials, and manufacturing processes required for the design, and notifies purchasing of any long-lead items needed. The source used for selection is the RASSP Reuse Database. If any new process is required, a process readiness activity must be successfully completed. This step is necessary to ensure that project schedules are not jeopardized by long-lead ordering cycles and/or development problems associated with the introduction of new processes.

- Perform Mechanical Design Analysis - Analyses of weight, stress, tolerances, and thermal distribution are conducted using appropriate models of the chassis.

- Establish Test Requirements - The mechanical, electrical, and manufacturing engineers determine the electrical and environmental test requirements and design or procure the needed test fixtures. Test plan documents are generated.

- Document Structure/Thermal Interface - At this time enough is known about the contents of the chassis to commit to an outline and structural mounting and thermal interfaces. For the design of the chassis to proceed without delay, we must provide firm information on maximum dimensions and dissipations to structural designers for the next higher assembly, if one exists.

- Document Harness Interface - At this point in the design cycle, we have sufficient data on signal flow

and electrical inputs and outputs to establish connector types, quantities and locations, and possibly complete definition of the harness interface. For the harness design to proceed in a timely fashion, we must commit to connector locations and types as early in the chassis design cycle as possible.

- Determine Assembly Processes - Working with manufacturing producibility engineers, the mechanical engineer decides on the sequence of events in the assembly of the hardware, and prepares to document the detailed design. The final drawing package defines only the finished configuration of the hardware, but when properly documented, can indicate an assembly sequence that eliminates synthetic subassemblies which create added paperwork for the manufacturing planners.

- Conduct Preliminary Design Review - The electrical and mechanical engineers participate in this review. Peers, technologists, and responsible engineers from other disciplines review the design and make constructive suggestions for improvement. Others (producibility, sourcing, quality, reliability, etc.) participate as required to ensure cross functional support for the design.

Items reviewed include layouts and solid models of major items, test plans, structural, thermal, tolerance, and weight analyses, source selections, updates to conceptual/proposal data, and interconnection layouts. We conduct this review in accordance with the local policy and procedure guidelines for internal design reviews, and following standard checklists. This step confirms that specifications can be met within established cost objectives, and should reconfirm the acceptability of designs and approaches generated in architecture design.

8.5.2 Chassis Final Design

The final design phase encompasses the detailed level mechanical design of the chassis. Detail design includes the interconnection cables and harnesses. These items require complete definition including, but not limited to: all dimensions and tolerances, material and finishes, marking requirements, test or acceptance criteria, etc. In this step, we begin the formal documentation of the chassis, the end product of which are drawings and parts lists necessary for a supplier or manufacturing to fabricate prototype or production hardware.

- Perform Detailed Design of Chassis - Solid model layouts are finalized, based on the latest changes and models are reduced to the detail part level. The design is driven by stress, thermal, and survivability considerations, with the goal of optimizing weight and producibility. This step is essential to the proper functioning of the electronics during and after exposure to environments. The chassis must provide protection to the sensitive electrical parts and interconnections from the effects of shock, vibration, internal and external heating, radiation, and magnetic effects. Electrical and mechanical test procedures are also generated during this phase.

- Perform Analysis of Chassis - In this step we begin verifying the detailed design. We again perform thermal, stress, tolerance, and weight analyses and compare the results with subsystem and environmental requirements, to ensure that we established adequate margins to provide proper performance, even under worst-case conditions. Cable and harness designs are also finalized.

- Perform Top Assembly Detailed Design - This step integrates the chassis design with the elements of the parallel detail design activities (backplane and module) which are discussed in later sections of this document, into the assembly which is the end product. This step pulls together the lower level detail designs and verifies that they will work together mechanically to meet the intended assembly function. Potential interferences are checked, dynamic clearances determined, and access for tooling, wiring, and assembly processes is established so that manufacturing can proceed smoothly and test problems will be minimized.

  The top-level assembly is subjected to thermal, stress, EMI, shielding, dynamic, venting, weight, and other special analyses to assure that adequate margins have established to provide proper performance, even under worst case conditions.

- Prepare Release Package for Chassis - Next, the required documentation or release to manufacturing or to an external supplier is generated. Solid models of parts are converted to drawings. Detail

drawings, assembly drawings, and parts lists are generated, in accordance with design and manufacturing standards and local documentation practices.

- Conduct Pre-Release Design Review - Electrical and mechanical designers (among many others) participate in this review which, if successfully completed, will affirm the readiness of the design for production. The review is conducted in accordance with local policy and procedures for formal internal reviews.

8.5.3 Chassis Fabrication, Assembly, and Unit Test

The bare chassis is manufactured and assembled locally or by an outside supplier. In either case, electrical and mechanical testing is performed according to the test plan and procedures developed previously.

8.6 Subsystem Integration and Test Process

In the overall RASSP design process, the subsystem (e.g., a signal processor) is usually the highest level of integration proposed. Subsystem integration and test is the last process the equipment undergoes. This process requires significant up-front planning in the form of integration plans, test plans, and test procedures long before the equipment is ready for actual integration and testing. Schedules for the availability of the tested subassemblies from the other design processes and manufacturing have to be negotiated. The subsystem integration and test process flow is detailed in Figure 8 - 6.

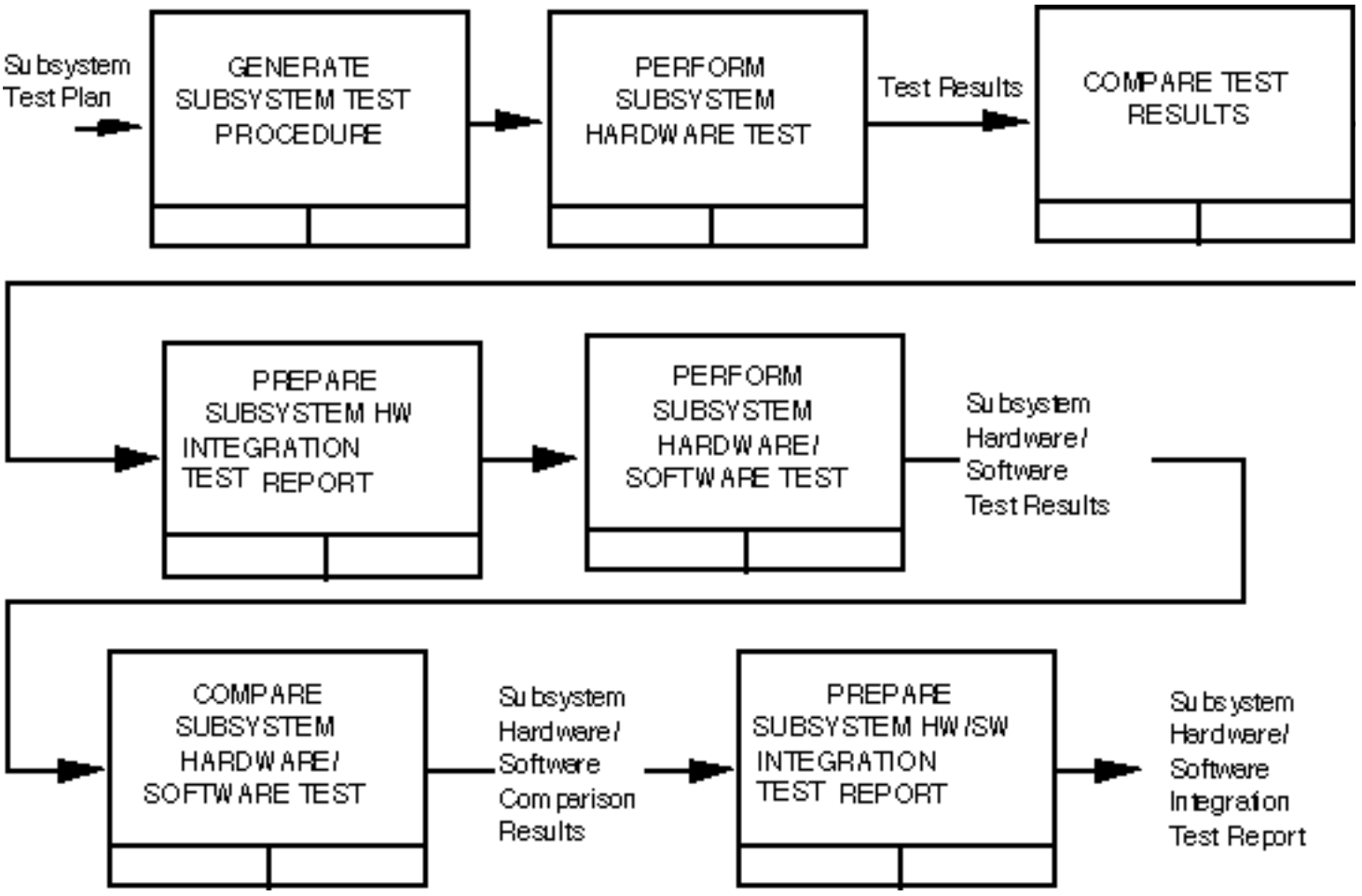


Figure 8 - 6: Subsystem integration and test process flow

8.6.1 Generate Subsystem Integration Plan

In the first step in the subsystem integration and test process, we generate a subsystem integration plan. This plan lays out the events that will occur, and their order and timing, to achieve an integrated and tested subsystem within the cost and schedule allocated to this process. The integration plan defines the interactions and delivery schedules negotiated with other design activities and manufacturing. It also defines the state in which equipment will be delivered to this process.

8.6.2 Generate Subsystem Test Plan

In this step we generate a plan to test the integrated subsystem. This plan specifies all tests that must be performed to verify that the subsystem satisfies the B2 specification. The plan is driven by Section 4 (Quality Assurance) of the specification, which identifies the method (analysis, test, inspection, etc.) by which each of the requirements is verified. For each of these verification procedures, the test plan specifies the inputs, conditions of test, outputs, number of samples, etc., in qualitative terms. It also identifies the test equipment required and how it will be used. Block diagrams of equipment configurations are useful to clarify their intended use.

8.6.3 Generate Multichassis Test Plan

In large subsystems that have more than one chassis, an intermediate integration and test step between chassis test and full subsystem test is useful. This step develops the test plan for this multichassis test. There may be more than one of these multichassis tests. The goal is to integrate as few chassis as possible to make a subset of the subsystem that performs a testable function.

We use the subsystem test plan as a starting point. For each test identified at the subsystem level, we attempt to define a similar test at the multichassis level, although this will not always be possible. As in the subsystem test plan, the multichassis test plans identify, in qualitative terms, the inputs, conditions of test, outputs, number of samples, etc., for each test. They also identify the test equipment required and how it will be used. Block diagrams of equipment configurations are useful to clarify their intended use.

8.6.4 Generate Chassis Test Plan

In this step, we generate the test plan for the lowest level of integration that includes a testable unit. In most subsystems, this is the chassis level, although for some subsystems it is the backplane level. As in previous steps, the goal is to find and solve problems in the equipment as early in the test and integration process as possible. We generate a separate test plan for each unique chassis in the subsystem; again, these identify, in qualitative terms, the inputs, conditions of test, outputs, number of samples, etc., for each test. They also identify the test equipment required and how it will be used. Block diagrams of equipment configurations should be used to clarify their intended use.

8.6.5 Generate Test Procedures for Each Test Plan

In this step, we generate detailed test procedures from each individual test plan. Since these procedures are used for actual testing, they cover all testing configurations. However, test procedures do not normally specify the testing and debug approaches required to solve a problem when a test fails. The test plans and procedures should describe the purpose of the test and how it functions in sufficient detail to make a debug approach apparent. The test procedures specify the test parameters outlined in the test plans. Expected outputs for each test and pass/fail criteria must be specified. Equipment figures and block diagrams from the test plans should be expanded and detail added. Forms should be included for each test which, when filled out, provide the basis for a test report.

8.6.6 Conduct Test Procedure Review

We review the test procedures to verify: (1) that the procedures accurately evaluate the subsystem's performance; (2) that the tests are done at as low a level in the design as possible; and (3) that the test procedures are complete. Incomplete test procedures have a significant cost impact on the testing floor. In

addition to this internal review, many customers require approval of the subsystem level test procedures.

8.6.7 Integrate Backplanes and Test

This step is the first phase of actual integration and test. We integrate assembled and tested modules with the physical structure, backplane, and firmware into a testable chassis. We then use the test procedures developed for this configuration to test the equipment. In a large subsystem there may be several unique chassis tested during this step; each is tested according to its own unique test procedure. We use subsystem-level expected results generated in the architecture design process to verify satisfactory performance of the equipment. Following integration and test, we prepare a test report, and the tested equipment is then ready for the next level of integration. The next level of integration should not proceed until all tests have been completed and problems resolved.

8.6.8 Integrate Chassis and Test

This step is the second phase of actual integration and test. We integrate assembled and tested chassis with the physical multiframe structure into a testable multiframe configuration. In a large subsystem, there may be several unique multichassis configurations tested during this step; each is tested according to its own unique test procedure. In small subsystems there may not be a need for this level of integration and test. We use subsystem-level expected results generated in the architecture design process to verify satisfactory performance of the equipment. Following integration and test, we prepare a test report, and the tested equipment is then ready for the next level of integration. The next level of integration should not proceed until all tests have been completed and problems resolved.

8.6.9 Integrate Subsystem and Test

This step is the final phase of actual integration and test. We integrate assembled and tested multichassis configurations into the subsystem, which is then tested according to its test procedure. We use subsystem-level expected results generated in the subsystem design process to verify satisfactory performance of the equipment. Following integration and test, we prepare a test report, and the tested subsystem is then ready for further integration, if required.

8.7 Other Considerations in the Detailed Design Phase

8.7.1 Use of VHDL in the Hardware Design Process

The hardware design process transforms the architecture component VHDL models into detailed hardware designs. We decompose the abstract, non-evaluated architecture component models into full-functional VHDL models of their constituent entities. We use VHDL extensively throughout this process. In addition to the full-functional models, we create bus-functional models where needed to efficiently test interface designs. The bus-functional models are timing and functionally correct with respect to interface functionality only, whereas the full-functional models are correct with respect to both interface and internal function and timing.

We develop VHDL structural models for the VHDL behavioral models that were developed during the architecture design process. These structural models show the partitioning and interconnection of architecture component modules. We develop a test bench and a VHDL behavioral model for each module, and each module is, in turn, further decomposed and partitioned into other modules, macro-cells, MCMs, ASICs, or COTS components. We continue this iterative process of decomposition and partitioning for custom components until the hardware functions can be described as RTL logic transformations within leaf-level VHDL behavioral models. We can then automatically synthesize the leaf-level models into gate-level netlists. The VHDL models of programmable units are designed to interpret the data files produced by the software design process to facilitate hardware/software co-simulation and codesign. During the partitioning process, we can identify some of the modules, MCMs, and ASICs as custom parts, while others are selected from COTS parts. For custom-developed parts, we develop VHDL models down to the ASIC level, with a fully functional behavior and bus-functional model describing each ASIC. For COTS parts, we develop or obtain only VHDL behavioral models.

The VHDL behavioral models describe the timing and functionality of the module, macro-cell, MCM, and/or ASIC. Timing data includes:

- Module, macro-cell, MCM, ASIC I/O
  - I/O timing constraints - I/O interface structures - I/O protocols - Signal strengths - Message types
- Module, macro-cell, MCM, ASIC processing latency
  - Data acceptance rate
- Module, macro-cell, MCM, ASIC stimuli response

Functionality data includes:

- Control strategies
- Task execution order
- Synchronization primitives
- Inter-process communication (IPC)

The test benches provide test procedures, stimuli, and expected results to verify that the design meets system requirements. The test benches are developed before, and are executed with, the behavioral models of the hardware designs during simulation. The design of test benches before component modeling supports the DFT methodology. The full-functional behavioral model forms the executable specification for a hardware design. We back-annotate the simulation results of the detailed hardware models to the higher level architecture models to make them precise.

8.7.2 Design For Test Tasks in Detailed Design

The focus of detailed design is synthesis and implementation of the selected architecture in hardware and software. From architecture verification, all boards have been broken down into behavioral blocks representing MCMs, ASICs, FPGAs or relatively small logic blocks (i.e. glue logic and/or functions to be implemented in PLDs). Major components such as processors, interconnects and sensor interfaces have been selected, specified and verified at least by simulation. Detailed design elaborates these designs into implementations by schematic capture, synthesis, place, route, autocode and primitive optimization activities.

Physical prototypes are designed, fabricated, tested and integrated with software per the program plan. The extent of the full system can range from a significant sub-assembly (i.e. a MCM or more typically a set of boards where each board type is present) to the complete system. Based upon these results the TSDs are updated with preliminary measurement data on fault population coverage. Design flaw data is collected based upon the extensive simulations of VP3 and the system prototype results. Preliminary results are collected on BIST coverage of manufacturing faults. If the program plan calls for a BIST demonstration, preliminary results are collected on BIST coverage of field support faults.

The generic test insertion process for each design entity is shown in Figure 8 - 7.

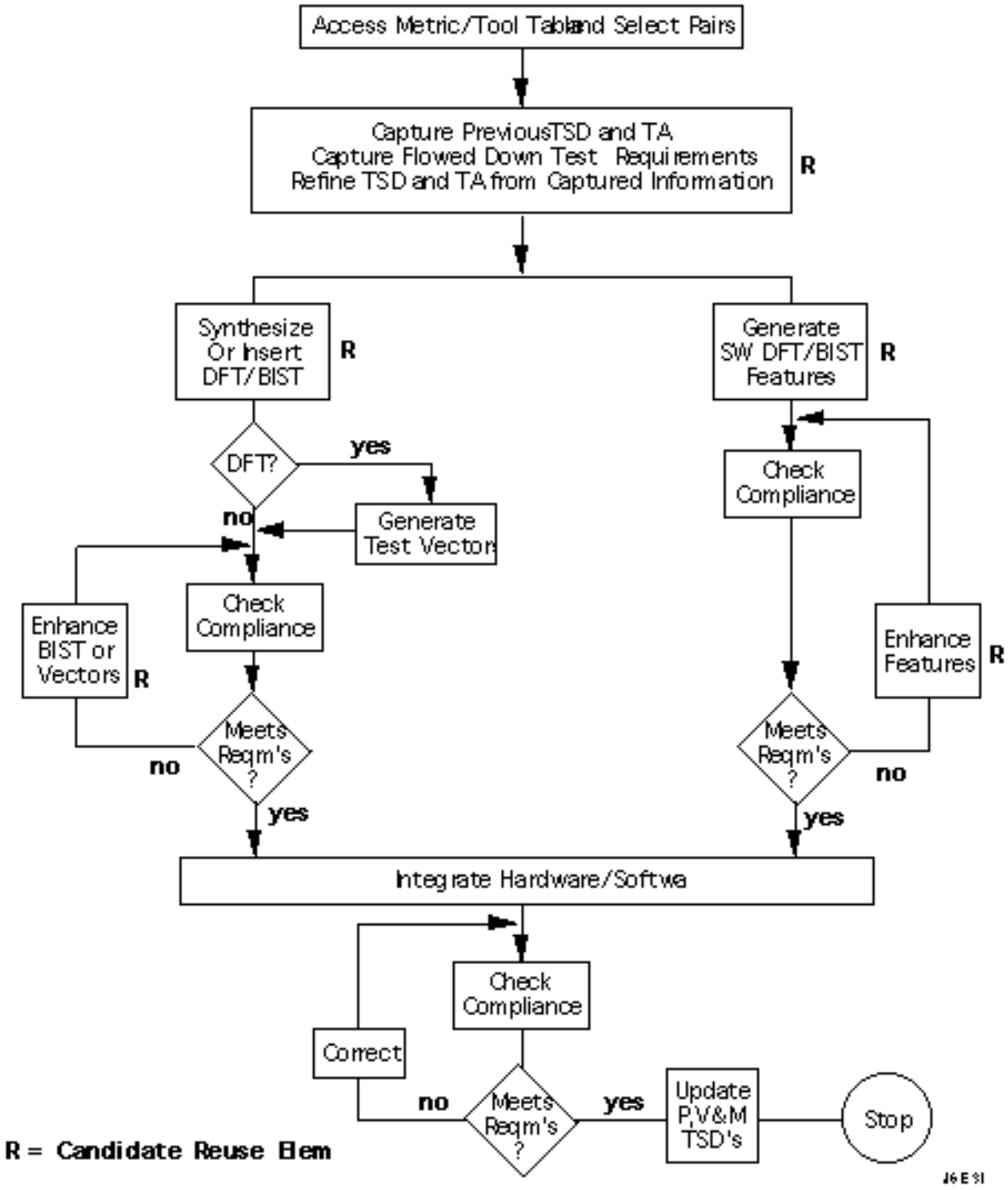


Figure 8 - 7: Generic flow during detailed design

Testable COTS components should be used to the maximum extent practicable. In addition, the "Lead, follow, or get out of the way approach" to dealing with COTS is implemented. This involves adding, or using existing, DFT features to the design to deal with COTS. As illustrated in Figure 8 - 8, COTS devices possessing BIST features, such as the TI SN74BCT8244 (having a PRPG and PSA mode), are given a "lead" role, since they can lead a test process. COTS devices, such as the four non-boundary-scannable octal flip-flops are given the "follow role," since, while they cannot lead, they at least do not impede the BIST test flow originating at the boundary-scan octals. Finally, the RAM chips, being very complex, and having no test features, coupled with an inability to pass pseudorandom patterns through during RAM tests, must be relegated to the "get out of the way" role which means bypassing them during BIST mode using a output

tri-stating approach.

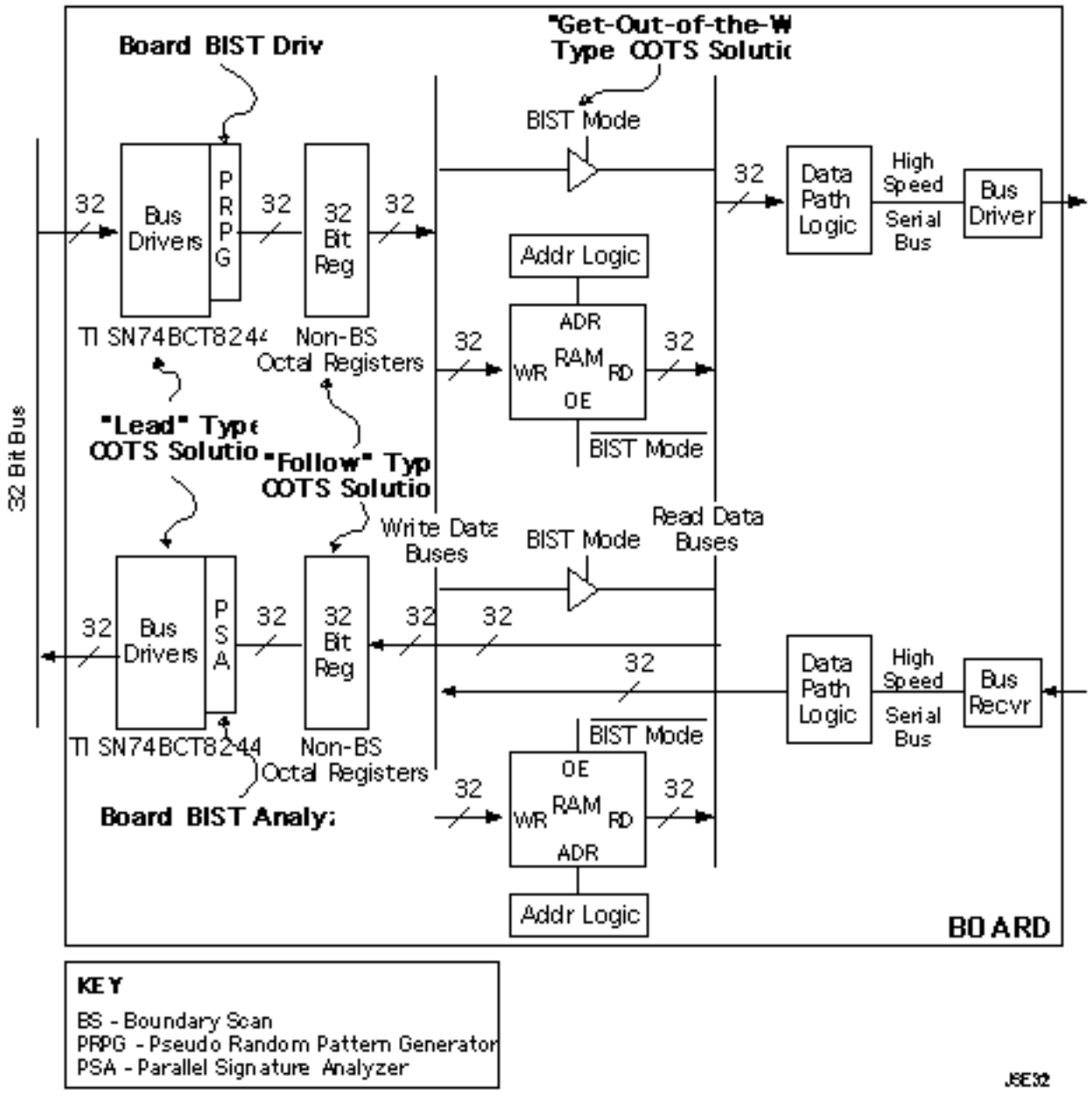
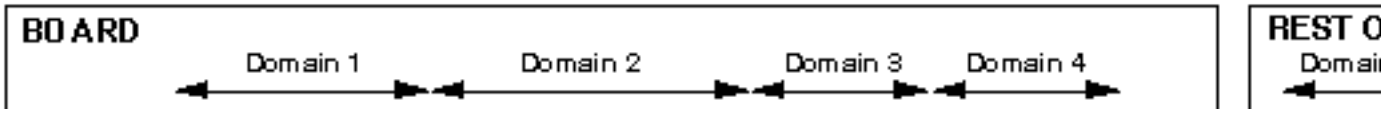


Figure 8 - 8: One aspect of dealing with COTS in DFT solutions

Test domain analysis is performed to stretch the reuse of all DFT/BIST structures. An example of this concept is shown in Figure 8 - 9. In the example, the source of BIST stimuli (8244) is analyzed to determine how much of the board (and eventually the rest of the system) can be tested using the PRPG and the PSA elements in the 8244s. A minimum of four domains (1 - 4) are covered along with additional domains in the rest of the system. By adding loopback capabilities, fault isolation is improved; and the number of domains covered is doubled. Pseudorandom patterns emanate from the source 8244 and responses through the loopback are compressed in the parallel signature analyzer in the sink 8244. Test domain analysis can be used to stretch the limits of reuse across packaging levels, as well as across the same package level.



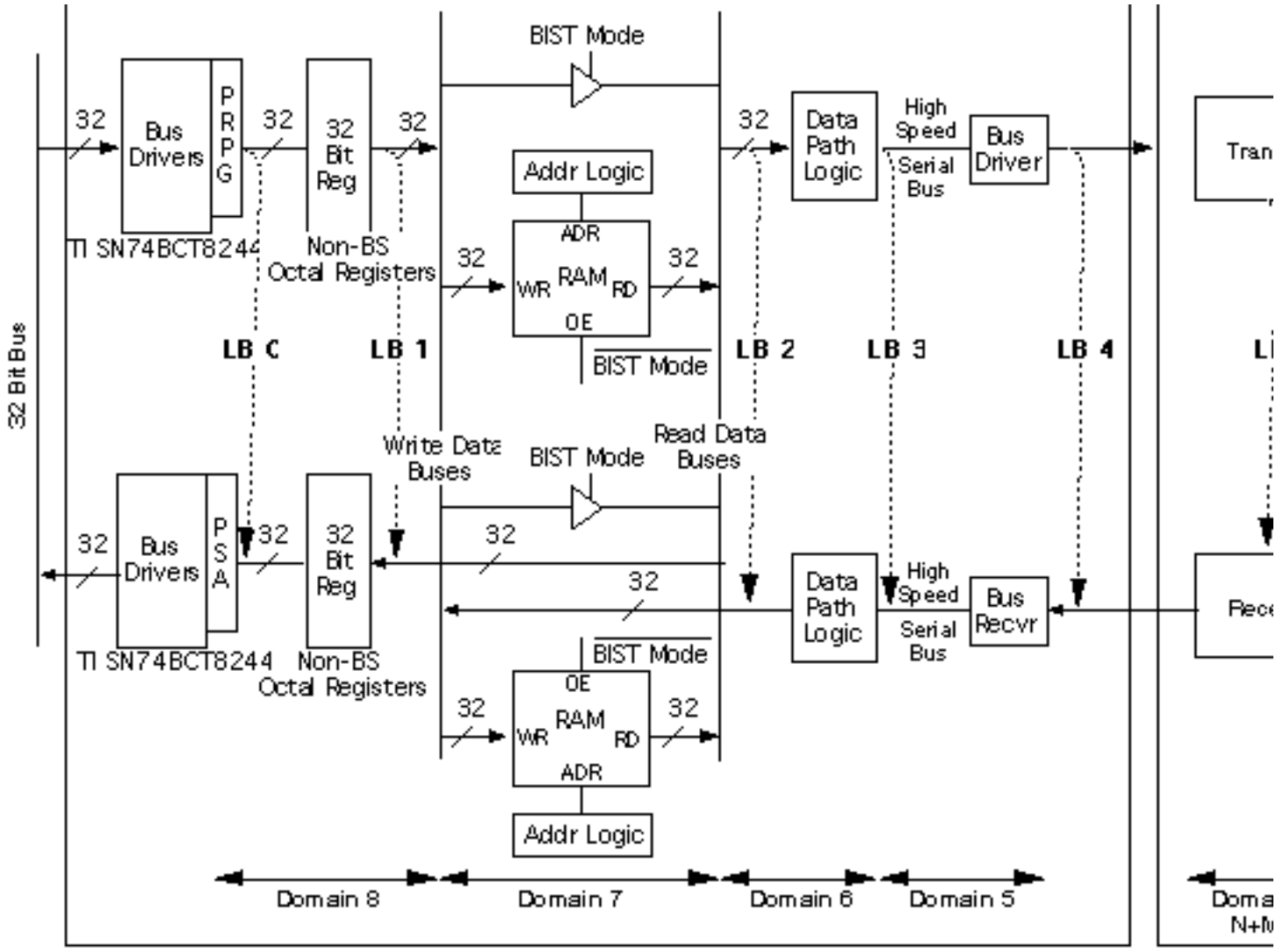


Figure 8 - 9: An example of test domain analysis

Checking for compliance for COTS sections requires definition of the fault model, such that the coverage is measurable, using non-structural simulation models for the black box COTS elements or by using simplified functional fault models.

See the Design for Test application note for further details of the Detailed Design Step.

Next Up Previous Contents

Next: 9 Integrated Software View **Up:** Appnotes Index **Previous:**7 Architecture Design Process Detailed Description

RASSP Methodology Application Note

9.0 Integrated Software View

The overall RASSP process is based upon the philosophy that dramatic decreases in cycle time can only be achieved with maximum reuse of both hardware and software elements. This leads to the notion of a graph-based, correct-by-construction methodology using validated library elements that represent either software fragments and/or hardware instantiation of these elements. This section provides a view of the overall software development that is part of the hardware/software codesign activity. This has all been previously covered but is distributed throughout the architecture design and detailed design process descriptions.

Within RASSP, software development encompasses DFG-based software, graphical-based control flow software, support software, and library population. We assume that a signal processor must communicate with a higher-level system, perhaps a command and control system for the platform. Messages must be accepted from the higher-level system that specify actions to be taken by the signal processor, such as data requests, parameter setting, mode switching, etc.. We also assume that the signal processor software must support diagnostics, error handling, startup initialization, downloading etc. In summary, the signal processor software development must result in the generation of all the code that must be downloaded to each processor within the system.

If all required signal processing can be constructed from validated elements from the reuse library, the signal processing software represented by the DFGs is automatically generated and documented. When all required processing cannot be constructed from validated library elements to meet the requirements (or when new hardware architectural elements are added to the reuse library), we must generate and validate new library primitives and/or operating system support software elements. This is referred to as library population and is discussed separately in this document. In addition to the signal processing represented by the DFGs, it is also necessary to develop the control software. This, as discussed earlier, is referred to as the command program. Command program generation is supported by emerging tools, which include autocode generation from state transition diagrams.

See also the following application notes for specifics on the software development process:

- Hardware/Software Codesign
- Data Flow Graph Design
- Autocoding for DSP Control

9.1 Software in the Design Process

We performed much of the work normally associated with applications, control, and communications software development in the architecture process in the RASSP methodology. Detailed software design is concerned with aggregating and/or translating the compiled HOL code verified at the architectural level to downloadable target code. In addition, we will generate low-level software modules to support initialization, downloading, booting, diagnostics, etc. on the target hardware. Since many of the support software functions have not been tested in the architecture process, we must verify these functions via simulation during detailed design. We generate downloadable code by taking the autocoded and hand-generated portions of application and control code, and combining them with the communications and support software into a single executable for each processor. The run-time system, which provides the reusable control and graph management code, will have all the hooks required to interface with the support code.

The software aspect of the detailed design process is the production of the load image. The items used and/or produced in this phase are described in the following paragraphs.

Graph Realization

A graph realization is a compiled version of the equivalent application graph. It is composed of data structures fully describing the software part of a graph, except for the values for control parameters and the actual parameters that will be bound to formal parameters. These actual values and parameters are defined at graph instantiation.

Partition Primitive Interface Description (PID)

A partition PID is the source code that represents a partition graph. The source code is written in the computational element's native language. Consequently, the architecture description and mapping the partitions to the architectural elements is required to produce the PIDs. The partition PID is produced from a partition in the partitioned graph and executes all the target processor-level primitives necessary to implement the domain-level primitives in the partition graph for the specific target processor. These partition PIDs are then compiled by the computational element's native language compiler so that they may be input to the load image.

Load Image

The load image is the end product of the software development process. To build a load image for an application, we combine the graph realizations produced for each equivalent application graph and their associated partition PIDs with all required microcoded primitives, the run-time object code, command program, and the kernel operating system for the selected architecture. The process is shown in Figure 9 - 1. This load image may then be downloaded to the target hardware and executed.

For the final test of the application, we run the test vectors used to validate the executable functional specification on the target architecture running the load image. The output vectors from this execution should match exactly with the test results from the executable requirement specification.

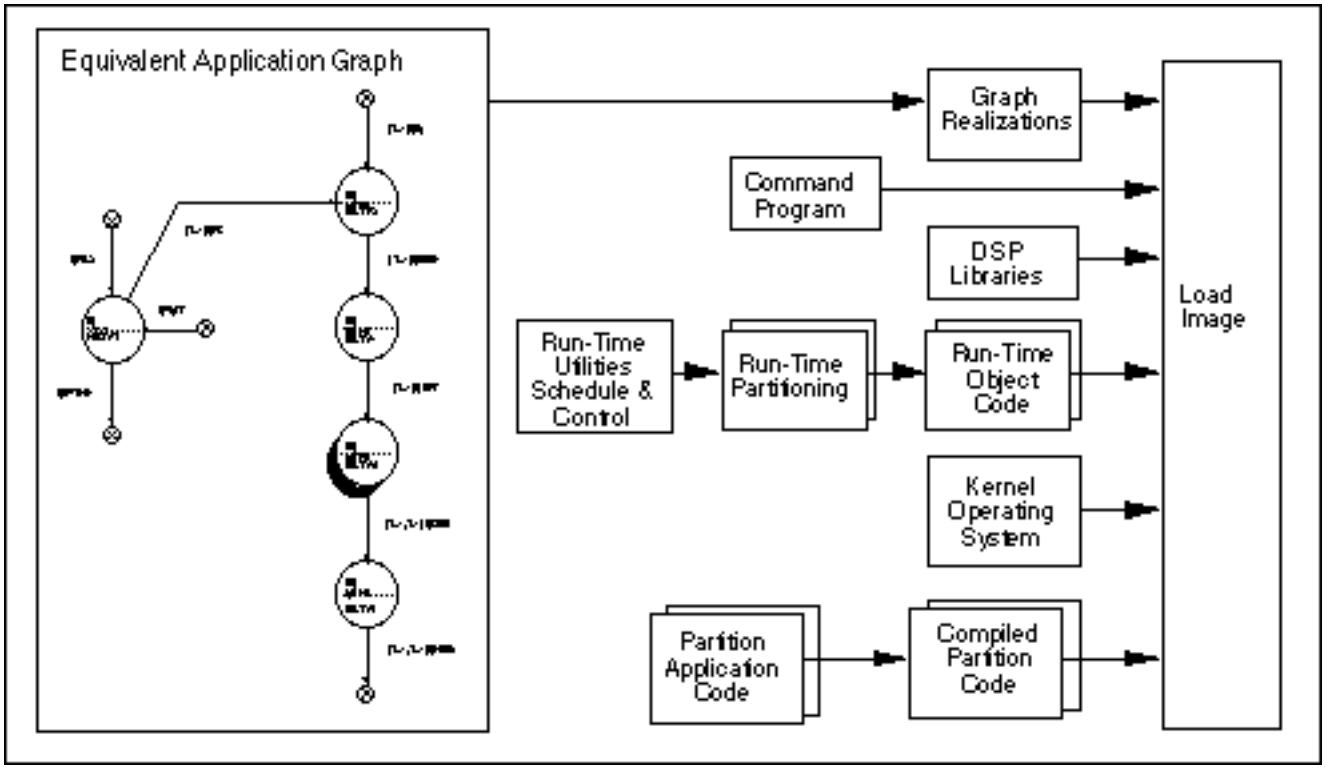


Figure 9 - 1: Load Image Building Process

Run-Time System

A reusable run-time system will be provided that is built on top of operating system microkernel(s). This system is a set of run-time application programs that constitute a virtual machine which performs all graph execution functions and all system interface, control, and monitoring functions. These programs may be reconfigured to execute on appropriate processors of the candidate architectures. We will interface them to the kernel operating systems of supported processors as part of validating a computational or non-computational element of the Model Year Architecture. Translation software translates data flow graphs to forms executable by this system. Interfaces are defined for external data, control, test and debugging, and monitoring system functions. The run-time system provides reusable signal processing control/communication code based on standard operating system services. Availability of this reuse package will make a DFG implementation of the RASSP methodology economically practical for a range of medium to small system users who could not otherwise afford implementing a run-time graph execution system. Traditionally expensive run-time support development and testing will be reduced to run-time reuse package reconfiguration and new element validation in some cases.

Support Software

In addition to all of the DFG-related software, we will generate low-level software modules to support initialization, downloading, booting, diagnostics, etc. on the target hardware during the detailed design process. Since many of the support software functions were not tested in the architecture process, we must verify these functions via simulation during detailed design.

We generate load images by taking the autocoded and hand-generated portions of application and control code, and combining them with the communications and support software into a single executable for each processor. The run-time system, which provides the reusable control and graph management code, will have all the hooks required to interface with the support code.

The top-level software architecture is shown in Figure 9 - 2. The application software consists of an interface to the world outside the signal processor in addition to a command program and data flow graphs. This could either be a user or, more likely, it could be a higher-level system as part of an overall platform. This interface receives messages from the outside which control the processing to be performed, request data from the signal processor, or provide parameters to the signal processor. The DFGs represent the signal processing algorithms that must be applied for each operating mode of the signal processor. The signal processing algorithms are represented as DFGs that drive the autocode generation to produce the executable code for each processor in the system. This process is supported by a reusable primitive library. The command program is the control flow program that provides the overall control, as dictated by the received messages. A run-time support system (represented in the figure by the control and data flow interface) provides the reusable data flow control and graph management for the signal processor. The run-time system is built upon a set of operating system services provided by a real-time microkernel using a standard interface. The run-time system is usable with any operating system microkernel that provides the necessary set of services. The run-time system will be a set of run-time application programs which constitute a 'virtual' machine that performs all graph execution functions and all system interface, control, and monitoring functions.

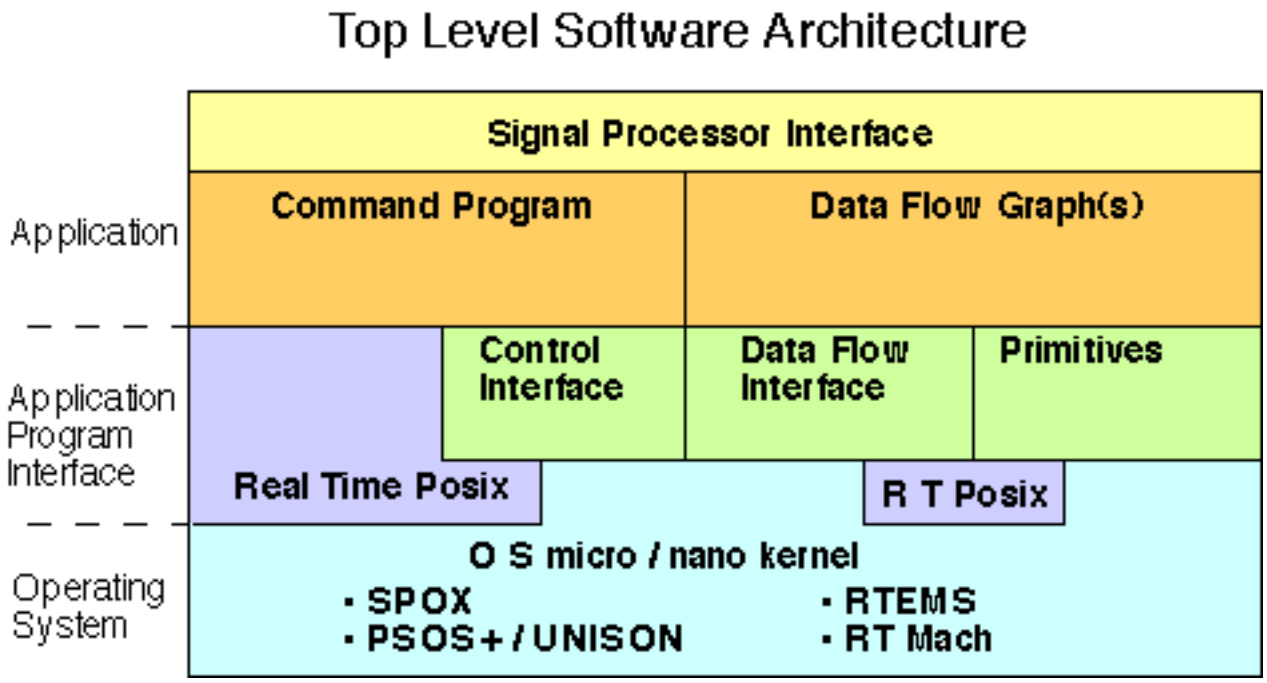


Figure 9 - 2: Top-level software architecture

9.2 Hardware/Software Codesign

The hardware/software codesign discussion presented here is primarily from a software perspective. As part of the virtual prototyping methodology, RASSP uses the results of simulations tailored to the various levels of the design process, which span system design, architecture definition, and detailed design. Figure 9 - 3 shows the progression of software generation from the requirements to load image, with emphasis on the graph objects involved and the general RASSP process in which they occur. Also shown is the parallel development co-simulation of the command program. Feedback is not shown in the figure, since this is not intended to be a process diagram, but rather a walkthrough of how we generate and transform the signal processing DFG to downloadable code for a target processor. In reality, many of these steps (along with intermediate steps not shown) occur iteratively as we develop the hardware and software architecture.

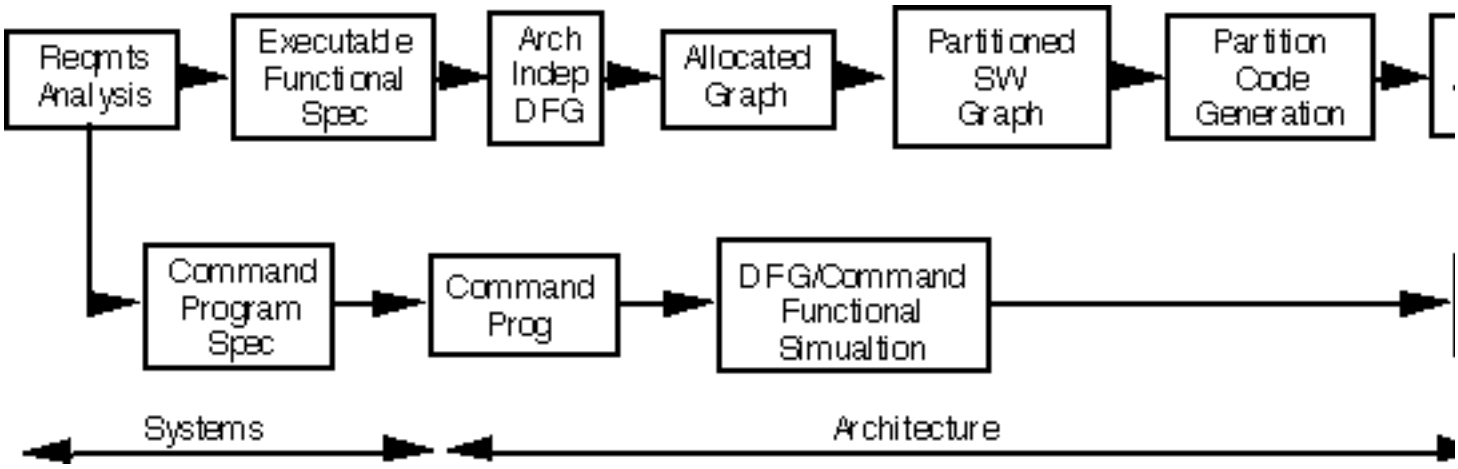


Figure 9 - 3: Graph-based Software Development Scenario

During the system design phase, we develop the initial system requirements. These requirements typically include all external interfaces to the signal processor; throughput and latency constraints; processing flows by

operating mode; all mode transitions; and size, weight, power, cost, schedule, etc. We use high-level simulations to generate overall processing strings that are functionally correct and satisfy the application requirements. At this level, no allocation of processing functions to hardware or software is made.

Architecture definition involves creating and refining the DFGs that drive both the architecture design and the software generation for the signal processor. The input to this phase is the executable functional specification for a particular application and the command program specifications. During the architecture design process, we develop DFG(s) of the signal processing and allocate the elements to either hardware or software. The flow graph(s) are simulated both from a functional and performance standpoint with increasing fidelity until we achieve an acceptable hardware/software partitioning that meets the signal processing system requirements. During the architecture process, we conduct various trade-offs among architecture classes, processor types, specialized hardware elements, and communication mechanisms. We perform high-level, initial performance simulations so that we can quickly evaluate gross architecture trade-offs with size, weight, power, and cost estimations. We also make a first pass at the non-DFG software requirements and tasks definitions.

Once we select a candidate architecture for optimization, the simulations become more detailed and therefore more clearly represent the final system performance. At this level, we perform automated generation of the software partitions to provide executable threads that run on a group(s) of DSP elements. The items to be traded in the detailed software mapping are latency, processor utilization, memory requirements, and computational organization of each of the groups. When the design cycle is completed, the requirements will either be met or the simulation results will show that the candidate architecture is not capable of meeting these requirements. In the latter situation, we use the performance analysis results to indicate where to make architectural changes. The process is iterated until a satisfactory solution is obtained. We further develop non-DFG software code. During architecture verification, we use prototype hardware and behavioral simulations to simultaneously verify the performance and functionality of the overall signal processing system. The final verification is performed hierarchically, since it is unreasonable to simulate the entire system at the RTL or ISA simulator level because: a) the computation requirements of the signal processing application are beyond the scope of feasibility for simulation at the clock level; and b) the verification must be driven by the types and fidelity of the models available in the component library.

The output of the architecture design process is both a description of the detailed hardware architecture and a detailed description of the software (including the computation, scheduling, control, and operating system elements) that must be generated/aggregated for downloading to each of the elements in the architecture.

The final step in the software development occurs during detailed design. The software aspect of the detailed design process is the production of the load image. We must generate and package the final code for downloading and execution on the target hardware. Since we verify most of the software developments during the architecture phase, they are limited at this point to generation of those elements that are target-specific. This may include configuration files, bootstrap and download code, target-specific test code, etc. All the software is compiled and verified (to the extent possible) on the final virtual prototype.

The software 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, and the partitioning and mapping information produced in the architecture design and verification process. This process is controlled by a software build management function that extracts the necessary information from the library and manages construction of all downloadable code, as directed by the partitioning and mapping data.

The software build management function must coordinate construction of the software to be downloaded to each processing element in the target architecture, the bootup software which initializes the processor, and any control code required for scheduling and sequencing the processors. When complete, the 'build' footprint is preserved in the library under configuration control. In addition to preparing this software, the 'build' management function initiates the documentation process. The documentation function uses documentation elements stored in the library to construct the overall software documentation. All library elements, including both operating system services and application primitives, must have associated documentation elements contained in the library that fully describe the algorithm, its implementation, ranges of applicability, and

validation test results.

Since the software is autocode-generated, there are certain assumptions about the validation testing of the resultant software that must be supported by the library process. We assume each primitive has been unit tested. This means that a test suite has been developed for each primitive and this primitive has been instantiated by the software generation process and validated. During the architecture process, we periodically repeat this unit testing as the signal processing graphs are transformed from one form to another. The validation process has also been applied to control program software and we do not have to validate each instantiation of the control program.

The software which is constructed must finally be validated on the target hardware. This should be based upon the system test stimulus that has been carried throughout the development process. This is the first time that the software and physical target hardware has come together. Based upon the testing, simulation, and virtual prototyping that has been performed throughout the RASSP architecture and detailed design processes, the final validation time is considerably shorter than the traditional integration and test times. See the SAR, ETC4ALFS on COTS Processors, and the SAIP case studies for validated examples of these reductions.

9.3 Library Management

Since the overall RASSP methodology is library-based, managing the reuse library is an extremely important function. The domain analysis and organization of the reuse library is key to minimizing the developer's frustration with the process.

The library must contain, for each primitive, a representation or model that supports both behavioral and performance simulation at various degrees of fidelity. In addition, the library management system must manage the application DFGs, candidate architectures, DFG partitioning files and mapping files.

Information available in the library to support the generation of target software is shown in Table 9 - 1. This information is comprised of architecture, application, operating system, and support data. It includes the DFG that describes the processing, the software primitives that make up the processing nodes in the flow graph, the communication elements that facilitate the transfer of data between processing nodes, the partitioning and mapping data that describe how the flow graph is mapped to the individual processors, all the operating system kernels and support software the kernels use, any initialization information that may be necessary, and BIT data. The library also contains a mapping of the target-independent primitive to the specific processors supported in the Model Year Architecture. The software run-time system depends upon a given set of operating system kernel operations. These must be available for each specific DSP type and memory, I/O and message passing structure.

The library stores all information required to generate target software including all the elements shown in Table 9 - 1.

Architecture Data	Application Data	Op Sys/Support Data
▪ Arch description file including processors and interconnections ▪ Partitioning & mapping file describing how the application flow graph is allocated to processors ▪ Documentation elements	▪ Flow graph(s) describing the required processing ▪ Software primitives representing the processing nodes in the flow graph ▪ Initialization data ▪ Documentation elements	▪ Operating system kernel(s) ▪ Operating system services elements required to control the transfer of messages or data between nodes ▪ Built in test data ▪ Boot code

TABLE 9 - 1: Backplane design process flow

The library management system also stores pertinent information relative to the resultant software builds for the target hardware. Version build 'footprints' include all information necessary to reconstruct the build.

9.4 Documentation

The documentation we must produce includes all the required documentation delivered with the target hardware. It is important to note that the familiar 2167A waterfall software development and documentation process is no longer applicable to the library-based, correct-by-construction instantiation of deliverable software. The burden of documentation, validation and test generation is more appropriately allocated to the library population function for the individual software elements. However, when the final software is orchestrated for the target hardware, some form of documentation suitable for both users and maintenance must be produced.

To support the documentation process, detailed documentation of library elements is a required part of the validation process for inserting elements in the reuse library. As discussed under library population, prototype software elements may be generated during the architecture selection process to support design trade-offs. When we select these prototype elements for permanent inclusion in the reuse library, they must be formally validated. Formal validation includes, among other things, generating detailed documentation for the library element. System documentation requires the assembly of the DFGs, along with the detailed library element documentation, the partitioning and mapping of those elements to the overall architecture, and the associated operating system kernel elements.

Next Up Previous Contents
Next: 10 Library Population **Up:** Appnotes Index **Previous:**8 Detailed Design Process Detailed Description

RASSP Methodology Application Note

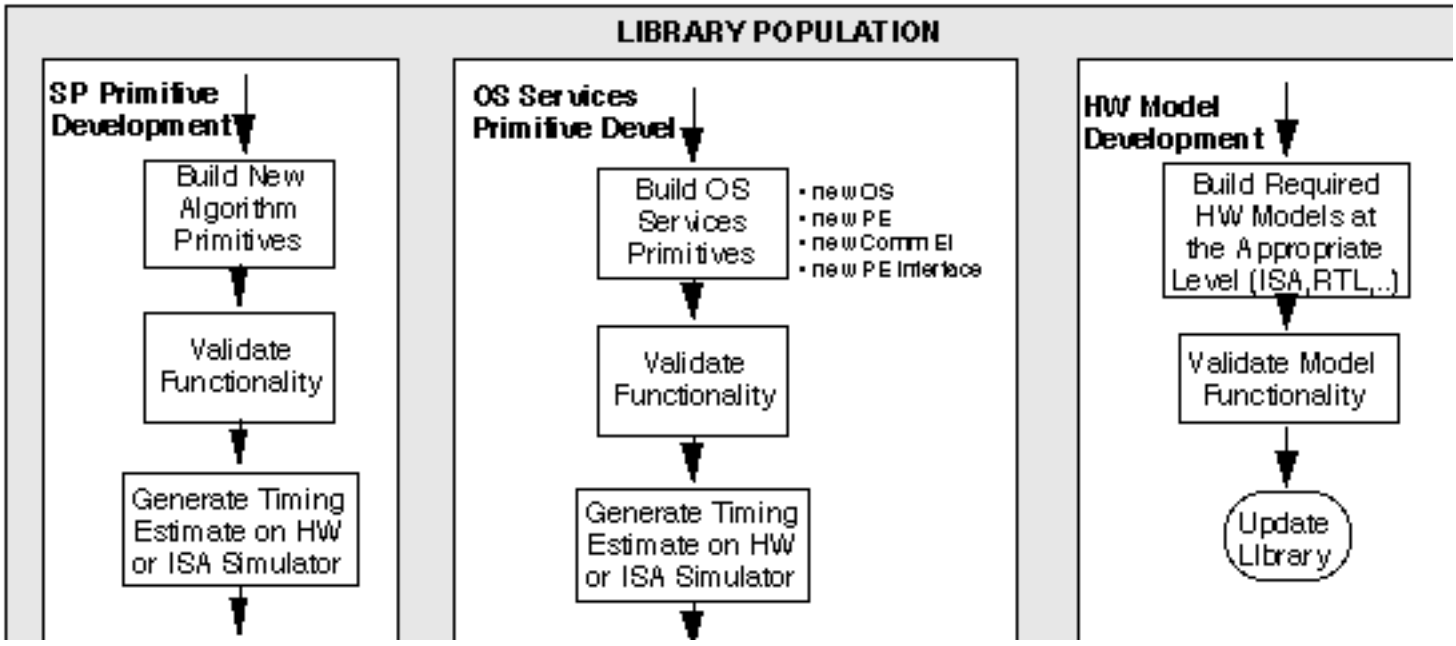
10.0 Library Population

When any of the required support components are not present in the library, we cannot proceed with the design without at least defining a prototype element. We call the process of adding any component function to the library, 'library population'. When the first RASSP example is designed there will exist, by design, a sufficient set of primitives and components in the library to support the example. However, this set must be extended over time to accommodate new technology and applications. The methodology supports the ability to add or modify components of this library. Successful application of the RASSP concept depends upon the ability to perform the library population rapidly and economically so that the project development timelines and cost are not adversely affected. It is important to note that a complete library component is not needed for all the design phases. The methodology allows the design process to proceed in stages, along with the library population effort, to complete the library description. We accomplish this by permitting prototyping of primitives for inclusion in the library as a prototype element. The prototype element may then be used in the hardware/software codesign process to perform high-level architecture trade-off studies during architecture selection. In reality, we can early (based upon the algorithmic processing flows developed during the system design process) that we require new primitives for the application. In fact, application-primitives may be prototyped during system design to simulate the functionality of the required processing. After being confirmed as a desirable primitive, we must validate the prototype element for permanent inclusion in the reuse library. We can perform this validation process in tandem with other portions of the development.

There are three library population software development activities which must be supported:

1. building a signal processing primitive library element,
2. building an operating system primitive library element, and
3. developing hardware models.

These three functions are shown in Figure 10 - 1 and discussed in the following paragraphs.



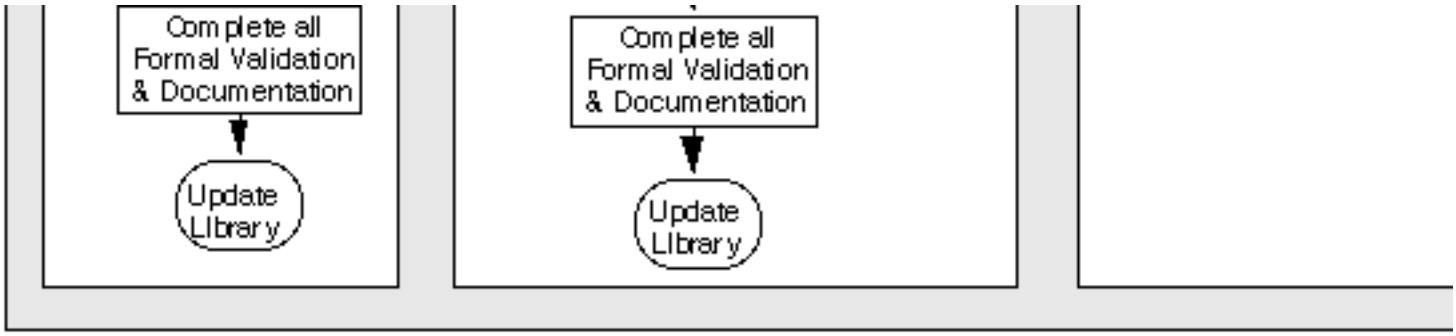


Figure 10 - 1: Library population top-level process

10.1 Signal Processing Primitive Development

We anticipate instances during the architecture definition process where signal processing functions desired by the designer will not be available in the library. This may be recognized at any time during the design process from early systems analysis on through the architecture process. Since the algorithms required to implement the application drive hardware/software codesign, it is most likely that we will recognize the need for new primitives early. The frequency of these instances will diminish over time for a given application domain as the library is continually extended. There are two ways that a new primitive can be prototyped. First, we can construct the new algorithm hierarchically from more elementary library elements to allow an approximate design to proceed with the understanding that the eventual implementation may have to be optimized for run-time by combining these primitive operations into a new primitive. Second, we can implement the new primitive as a HOL description of the algorithm.

To build a new primitive, the primitive generation tool(s) should interact with the user (i.e., application programmer) to construct the prototype for experimentation in the architecture design process. Templates for prototype primitives are stored in the library to support the developer. These templates provide the encapsulation wrappers to interface with the various tools in the system. Approximate timing information for the primitive (for the processor type(s) of interest) must also be generated or estimated to support the architecture trade-off process. This timing information may be updated as the primitive validation proceeds.

After developing the prototype element, the architecture definition process can proceed. We can use the prototype element in the synthesis, evaluation, and trade-off of candidate architectures during hardware/software codesign.

We continue library population by validating the prototype element for permanent inclusion in the library. There is more to the library validation process than simply validating the prototype code. To be a permanent library element, the code must be generated as a domain-primitive suitable for constructing domain-primitive graphs that are architecture-independent. We must validate it over a broad range of test stimulus, document it as a permanent part of the library, and model it in terms of performance on all processor types in the library. This process is aided by templates maintained in the library.

As we validate the prototype code and add support for the code generation process and validate it, the new library element can be used in the autocode generation. It should be noted that library elements may be optimized, which could result in multiple implementations of the primitive being maintained in the library. For some processors, it may be appropriate to optimize the primitive in assembly language or microcode to maximize performance. Translations of the domain-level primitive to the target processors of interest will be maintained in the library in the form of target processor mappings.

In addition to validating that the code produced is functionally correct, the validation process must produce the necessary documentation elements that are also maintained in the library. This documentation should include the test suites used and the resulting test report.

10.2 Operating System Services Primitive Development

There are four instances where we must generate or modify operating system services. These include adding a new operating system, a new processing element, a new communications element, or a new processor interface.

Adding a non-computational architectural element, such as a communication element (e.g., cross bar switch), processor interface, or I/O interface, may require modifications or additions to the application support software maintained in the reuse library. In many cases, the new element requires additions or modifications of operating system kernel support functions, such as drivers or mapping tables, which provide the operating system services to the applications.

The addition of a new operating system requires that all operating system services that are assumed to be available for any processor of interest must be generated for each processor type for which the new operating system is intended for use. We assume that the operating system vendor has provided the operating system for one or more specific processors, and that only the operating system services required to support the application program interface to the run-time system are required.

When we insert a computational element as a new architectural element, we assume that the required HOL compiler(s) is available for the processor. In the RASSP methodology, the atomic computational element may be any general-purpose processor or DSP that is supported by a HOL compiler and/or assembler.

Adding a new computational element requires that we modify other existing library information, or add new library functions. First, the functional performance of all application-primitive elements intended for the new computational element must be verified; that is, they must compile and execute the test suite to produce the correct results. To provide the information required for the hardware/software codesign process, we must update the application primitives and the operating system services elements to incorporate timing estimates for the new computational element. This process must be performed for all primitives in the library available for the new computational element. In addition, to support automated generation of software for the target hardware, an operating system that provides the standard set of services required to support the methodology must be validated for the new computational element. Specialized microcoded functions (such as math libraries) for the new element must also be incorporated into the library so that primitives use the optimum code when translated and compiled for the new computational element.

10.3 Hardware Model Development

When new hardware elements are required in the design process, we must develop VHDL models suitable for use by the various tools operating at different levels of abstraction. This could be a new signal processor or a custom hardware design. Ideally a new signal processor is supported by appropriate models from the manufacturer, but that is not always the case. In the case of a custom hardware element, it may be necessary to develop a high-level performance model for use in the architecture design process, as well as more detailed models suitable for detailed analysis and synthesis. Model generation tools will simplify the process and reduce model generation time.

Next Up Previous Contents

Next: 11 References **Up:** Appnotes Index **Previous:**9 Integrated Software View

RASSP Methodology Application Note

11.0 References

Pridmore, Jeffrey S., and W Bernard Shaming, " RASSP Methodology Overview, " Proceedings 1st Annual RASSP Conference, Arlington, Va., August, 1994, pp 71- 85. [PRIDMORE_94]

Processing Graph Method Specification: Version 1.0, Navy Standard Signal Processing Program, (PMS - 500), December 1987
[To obtain these documents and other information regarding PGM refer to the PGMT home page presented by the Processing System Section, Advanced Infomation Technology Branch at the Naval Research Laboratory, Washington, D.C. www.ait.nrl.navy.mil/pgmt/pgm2.html or contact Mr. David Kaplan at phone number (202) 404-7338 or e-mail kaplan@ait.nrl.navy.mil]

Processing Graph Method Tutorial, Navy Standard Signal Processing Program, (PMS - 500), January 1990.
[To obtain these documents and other information regarding PGM refer to the PGMT home page presented by the Processing System Section, Advanced Infomation Technology Branch at the Naval Research Laboratory, Washington, D.C. www.ait.nrl.navy.mil/pgmt/pgm2.html or contact Mr. David Kaplan at phone number (202) 404-7338 or e-mail kaplan@ait.nrl.navy.mil]

RASSP Methodology Document, Version 2.0, Volume 1, Lockheed Martin Advanced Technology Laboratories, October 1995. [METHODOLOGY_95]

RASSP Design for Testability (DFT) Methodology document, Version 1.0, Lockheed Martin Advanced Technology Laboratories, September 1995. [METHODOLOGY_DFT95]

Saultz, James "RASSP In-Process Report", High-Level Electronic Systems Design Conference, San Jose, CA, October 1997 [SAULTZ_97]

11.1 Application Notes

11.2 Case Studies

Semi-Automated Image intelligence Processor [SAIP-CS]

11.3 Web Pages

GEDAE™
