A TOP-DOWN APPROACH TO IC DESIGN

Chris Browy
Glenn Gullikson
Mark Indovina

INTEGRATED CIRCUIT DESIGN METHODOLOGY GUIDE

v1.4

A Top-Down Approach To IC Design

v1.4

3

Design Environment Implementation

4

Design Capture Methodology

4

Design for Test Methodology

6

Design Verification Methodology

7

High-Level System Design

8

Logic Synthesis Methodology

9

Timing-Driven Design Methodology

10

Block-Level Implementation

11

Chip-Level Assembly Implementation

A

Programming with the TDSP

B

GNU Free Documentation License

Preface

Overview

A Top-Down Approach To IC Design provides a practical foundation for the top-down design of ASIC and FPGA-intensive hardware systems. This book is intended to be used by engineers and managers who are involved at various stages of top-down design methodology including those just making the transition to top-down design.

The top-down design methodology addresses systems-level, ASIC, and FPGA design issues relating to concurrent design, validation, implementation, and manufacturing. Methodology trade-offs are discussed and specific recommendations are made to facilitate effective top-down design.

The methodologies and environment are described in sufficient detail for readers to be able to both recognize their benefits as well as directly implement them.

Organization of This Book

There are three major sections to this book. For a very high-level overview of top-down design, you can read just the introduction. For more information about a particular area of top-down design methodology, you can read one or more of the methodology chapters. To understand the top-down design flow and how the methodology is applied to a particular design, you can read the design chapters.

- Introduction

  Chpater 1 defines top-down design, describes its basic principles, gives a sample design flow and schedule, and discusses the critical factors related to a successful transition to top-down design.

- A methodology section

  Chapters 2-5, 8, and 9 describes design challenges and how the top-down methodology addresses them. The discussion is divided into the principle areas of top-down design methodology:

  - Design environment
  - Design capture
  - Design for test
  - Design verification
  - Logic synthesis
  - Timing-driven design

- A design section

  Chapters 6, 7, 10, and 11 illustrate the application of the top-down design methodology to a particular design, a Dual-Tone Multi-Frequency Receiver system. The discussion is divided into the principle phases of the top-down design flow:

  - Chaper 6, High-level system design
  - Chapter 7, Design environment
  - Chapter 10, Block-level implementation
  - Chapter 11, Chip-level assembly

  The case study also includes a description of a sample design environment.

1

Introduction

Introduction

The challenges facing the electronics design community today are significant. Advances in semiconductor technology have increased the speed and complexity of designs in tandem with growing time-to-market pressures. The companies that have remained competitive are those that are able to adapt to changing methodology requirements and develop a broad range of products quickly and accurately.

Successful product development environments (PDEs) streamline the design process by creating the best practices involving people, process, and technology. Developing these best practices is based on a thorough understanding of the needed design methods and how to apply them to the system project. This document reviews the basic principles of top-down design for ASIC and FPGA-intensive systems, and provides guidelines for developing best practices based on both semiconductor and EDA technology advances.

What is Top-Down Design?

The strategy of most successful PDEs is to build advanced, high quality products based on a system platform architecture that effectively incorporates leading-edge hardware and software algorithms as well as core technology. This strategy provides integration density, performance, and packaging advantages and enables product differentiation in features, functions, size, and cost. In most cases, to fully exploit these opportunities, this strategy requires a transition from a *serial* or *bottom-up* product development approach to *top-down* design.

The Bottom-Up Design Approach

In a bottom-up design approach, the design team starts by partitioning the system design into various subsystem and system components (*blocks*). The subsystems are targeted to ASICs, FPGAs, or microprocessors. Since these subsystem designs are usually on the critical path to completing the design, the team starts on these immediately, developing the other system components in parallel. Each block is designed and verified based on its own requirements. When all blocks are complete, system verification begins.

The bottom-up design approach has the advantages of focusing on the initial product delivery and of allowing work to begin immediately on critical portions of the system. With this approach, however, system-level design errors do not surface until late in the design cycle and may require costly design iterations. Furthermore, while related products can reuse lower-level components, they cannot leverage any system-level similarities in design architecture, intellectual property, or verification environment. Finally, bottom-up design requires commitment to a semiconductor technology process early on and hinders the ability to reuse designs in other technology processes.

The Top-Down Design Approach

The alternative approach is the top-down design approach. In this approach, the design team invests time up front in developing system-level models and verification environment. Using the system models, the team is able to analyze trade-offs in system performance, features set, partitioning, and packaging. Furthermore, a system-level verification environment ensures that system requirements are met and provides the infrastructure for verifying the subsystems and system components.

The top-down design approach results in higher confidence that the completed design will meet the original schedule and system specifications. Basing the starting point of the system design on a single verified model ensures that critical design issues surface early in the process and reduces false starts in the concurrent design of ASICs, PCBs, and systems. The design team can discover and manage system-level issues up front, rather than having to redesign the system at the end of the design cycle. Because each subsystem is designed and verified within the context of the system verification environment, the overall system functionality is preserved.

> *The key idea is top-down design, where the system is defined in ever-increasing levels of detail...Presumably, one then has everything defined completely before actually specifying a single gate. Traditionally, designers have handled pre-implementation stages informally...Today's systems are too complex for such approaches.*
>
> [Dr. Leventhal, *Printed Circuit Design,* Sept. 1995]

The top-down design approach also effectively leverages the initial product development in the design of related products. The related projects begin with the system environment in place. The design team can reuse and reverify alternative designs, packages, or implementations without having to rebuild a new context or infrastructure.

Success Factors in the Transition to Top-Down Design

Advances in semiconductor technology drive advances in EDA technologies and create new opportunities, as well as challenges, for the electronics industry. The successful companies are the ones that can leverage the opportunities offered by semiconductor advances as well as the new technologies and methodologies offered by the EDA companies. Product development strategies therefore play a prominent role in business strategy.

A company's competitiveness can be analyzed using the Y-diagram shown in Figure 1-1. The more competitive companies are able to develop system architectures and algorithms that exploit the opportunities offered by the new semiconductor technologies, while at the same time adopting the new EDA technologies and methodologies that allow them to deliver quality products. This capability allows them to compete for increased market share with more advanced products.

áp dụng

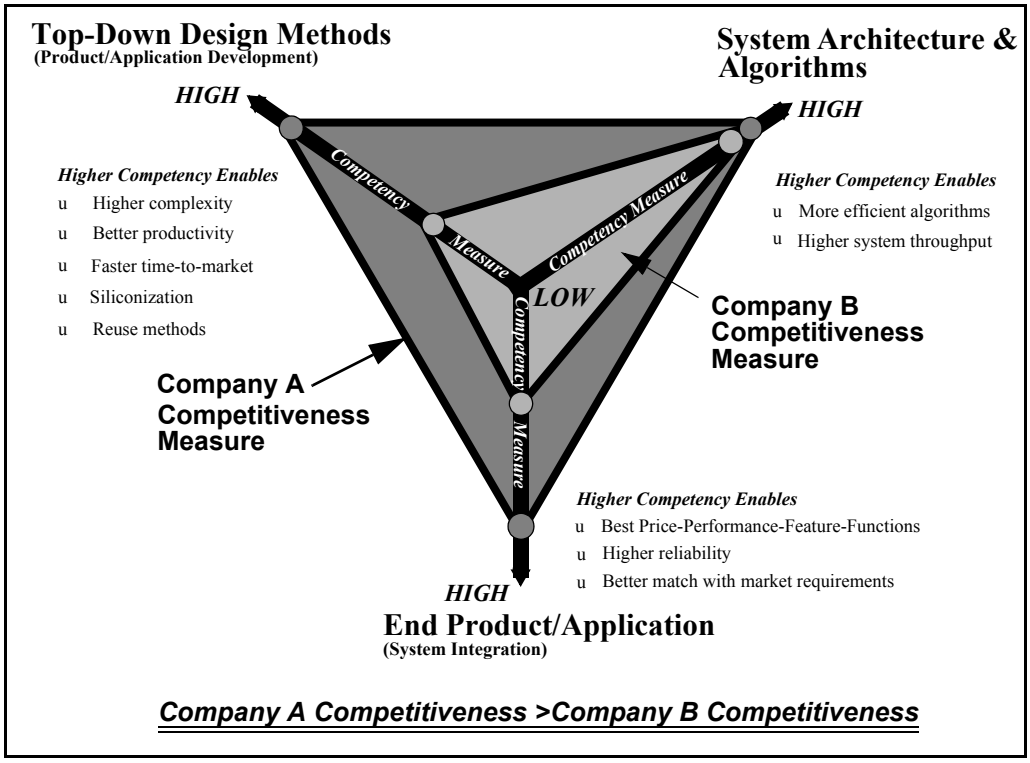
The risk in adopting new technologies is not insignificant, however. In order to enhance, not inhibit, a company's competitiveness, the adoption of a new technology must be based on well-defined business objectives and made with a clear understanding of the need to invest in training the design team in the new methods and tools. Otherwise, the team may not be productive.

There are several essential factors that govern ultimate success

- Rich technology and product development environment
- Effective planning and training in methodology and process
- An integrated, well trained project team working concurrently on design, implementation, validation, and manufacturing

thẩm định

Figure 1-1 PDE Competitive Analysis



Advances in Semiconductor Technology

Recent advances in semiconductor technology have created the opportunity to put more and more functionality, even entire systems, onto a single chip (*systems in silicon*). This trend toward the siliconization of electronic products is detailed in Figure 1-2.

Figure 1-2 Siliconization Trend from the Mid-1990s (Collett International, 1995)

	1994	1996	1998
Average process of top 20% ASIC/IC (microns)	.6	.4	.25
Average size of largest 25% of ASIC (used gates)	250,000	400,000	500,000
Average clock freq of fastest 25% ASIC (MHz)	88	120	150
Average size of top 25% ASIC	$15mm^2$	$17mm^2$	$20mm^2$

The recent advances in technology enable the electronics industry to create new markets and develop new products with unprecedented performance and features while keeping cost, power, and size to a minimum. These advances have had a profound effect in shaping the electronics industry by

significantly raising the level of electronics found in most technology-oriented products. If this siliconization trend continues as expected, it will result by 1998 in a 2X increase over 1994 levels in the relative gate complexity of the top 25% of ASICs developed. Maximum silicon potential found in ASICs will also rise substantially and exceed 2,000,000 gates for the largest available commercial ASIC packages.

Based on the siliconization trend, the concept of an ASIC has been extended to Application Specific Standard Products (ASSPs), which combine ASIC capability along with one or more standard cores. Even now in 1995, the industry has been quick to introduce standard core products supporting a wide variety of rapidly emerging wireless, networking, telecom, multimedia standards, and general purpose processors, DSPs, and controllers. These core products will create a dramatic new level of system integration and functionality.

Advances in EDA Technology

Current directions in EDA focus on providing dramatic improvements in design productivity by integrating the tools and methods for system design, logic implementation, and physical design, and by driving those tools and methods with deep sub-micron technology factors. New "enabling" technologies will emerge just as logic synthesis emerged to enable ASIC design using HDLs in the late 1980s. The key technology directions include:

- Tools and methods for system-level design capture, performance and requirements analysis, and debug
- Methods for core-based top-down design with improved support for customization, verification, and security
- New synthesis and floorplanning tools and methods addressing more effective deep sub-micron ASIC design, high-speed data path design, and low power design
- Advanced verification methods addressing high-speed simulation and emulation, timing verification, and formal verification
- Advanced design-for-test strategies
- Advanced logical/synthesis methods targeting datapath and clock tree generation, and interconnect/gate timing optimization

In order to enhance productivity, all technology advances must also be matched by corresponding advances in design methodology. EDA vendors, who supply most of the new technology, are showing an increasing willingness to provide expert consulting and training in methodology.

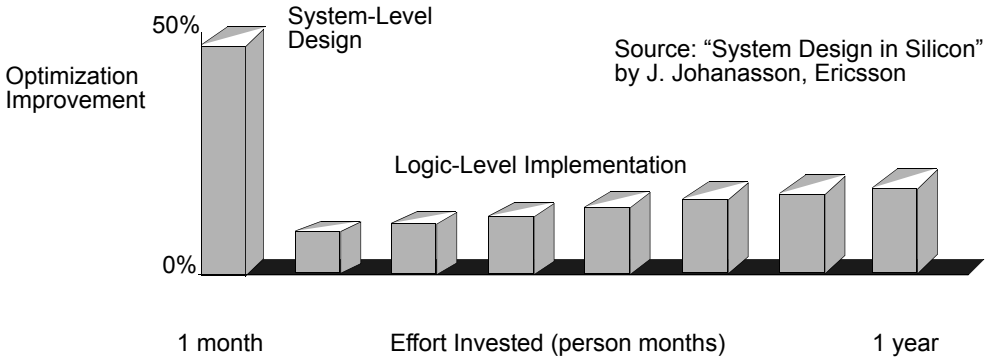
The Challenge to Productivity

For the last ten years designers have struggled to keep pace with siliconization trends. For the top 25% of the largest ASICs developed, design complexity has increased 25X. Meanwhile designer productivity levels have only risen by 5-8X [Collett International, 1994]. The challenge of adopting new technologies while maintaining productivity is clear. For example, Sematech has initiated EDA productivity studies in an effort to drive the rapid adoption of advanced design methods in order to keep pace with the rise in silicon potential.

Electronics companies just making the transition to top-down design methodologies must plan and execute effectively to realize potentially large productivity gains in product development.

Some companies have been able to implement new technologies and methodologies and measure the positive results as well. The results of a study from Ericsson Telecommunications, for example, show that it was possible to realize a 50% improvement in design performance and cost by investing in high-level system design and verification. These results compared to only a 20% improvement based on optimizing a design after implementation.

Figure 1-3 Design Effort vs. Implementation Improvement



Productivity gains which keep pace with advancement in electronics and semiconductors requires a continuous focus on product development process improvement. As was shown earlier in Figure 1-1, the overall competitiveness of a company is, in part, is determined by the performance of its product development process. With ever higher degrees of electronics in the most high-technology products today, companies must focus more on the development process for electronics and ICs.

Some of the largest determinants of a company or product development organization to foster continuous improvement are culture and capacity for

change which can be assessed by evaluating four basic questions which are at the heart of process redesign:

- How does the company's product development process compare to others?
- How will the product development organization gain a performance advantage?
- What will this mean to the company?
- How will the product development organization be converted?

A company's ability and willingness to drive for answers to these questions will realize the initial steps to process redesign steps which can result in higher levels of productivity and effectiveness. A complete process redesign process involves the following range of activities:

- Defining clear process measures and evaluating current performance
- Selecting improvement strategies
- Defining improvement objectives and metrics
- Evaluating the impact and ROI of improvement options
- Identifying process architects and owners
- Planning implementation and transition for the organization
- Identifying roles for 3rd party consulting

Defining Process Measures to Evaluate

The audit and evaluation of a product design process must target all relevant process measures which effect the overall outcome. There are several types of process measures which have been defined from the following definition of process by Tenner and DeToro (1997):

> "*One or more tasks that add value by transforming a set of inputs into a specified set of outputs (goods or services) for another person (customer) by a combination of people, methods, and tools*".

The relationship of these types of process measure is shown in Figure 1-4 and include:

- **Outputs**

  Design results in the form of product features, attributes, and values delivered to the customer.

- **Process**

Design effort and activities employed to render design results including the process, methodology, tools utilized by people

- **Inputs**

  Capability and capacity of the team and design environment, and the input specification for the product and its development

- O**utcome**

  Outcome is typically measured by customer satisfaction based on meeting initial product expectations.

The selection of process measures must consider making sure the measures are:

- Directly measurable
- Clearly defined and agreed upon
- Independent of factors which are beyond the control of the process.

Thorough mapping of the processes used in product development is typically carried out using two different methods concurrently. These methods approach mapping from opposite perspectives yet combine to yield a complete understanding of the processes and subprocesses map.

- Decomposition

  Mapping starts with looking at how the system operates on the work. Processes are understood top-down. The system is dynamic and changes based on the specific workloads.

- Synthesis

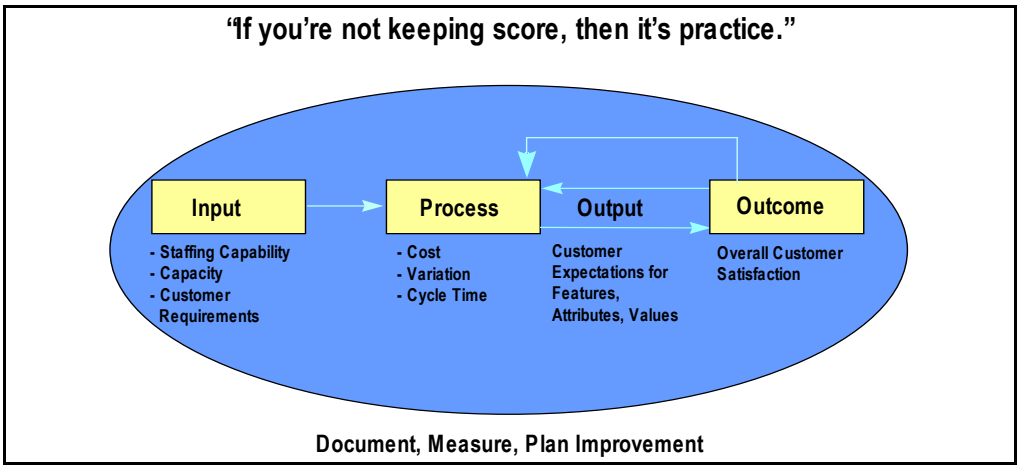
  Mapping starts with looking at how work is performed on the system. Processes are understood bottom-up. The system is a collection of static resources or functions with defined interfaces.

Only after building a complete understanding of how an organization fulfills its mission, the core processes can be identified. Core product development processes most significantly and directly impact the overall performance and need to take the highest priority in improvement planning. Core processes have several characteristics including:

- Crucial to business success
- Strategic importance into the future

- Customer impact
- Cross functional

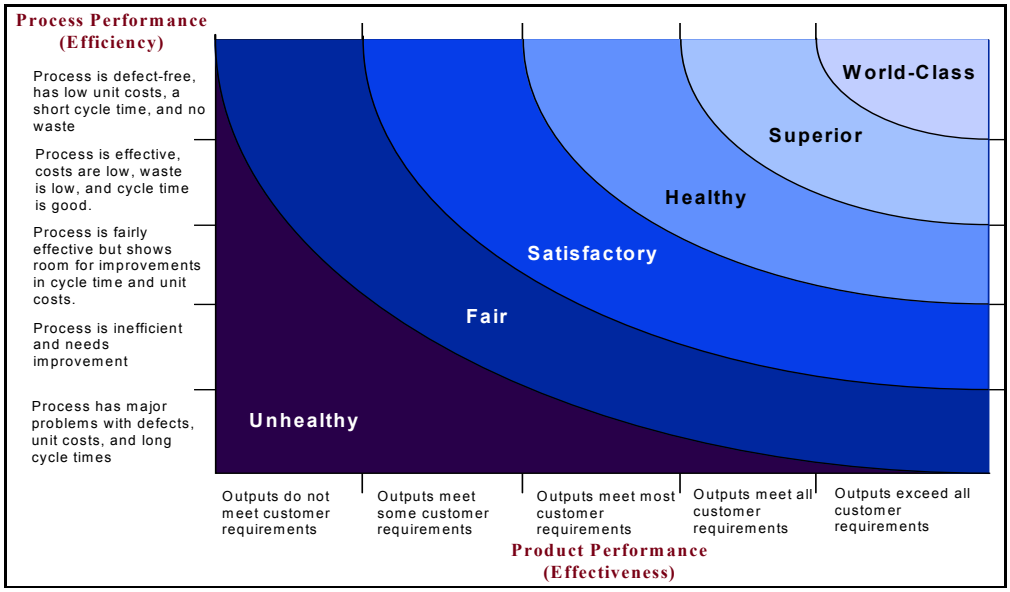
Figure 1-4 Performance Measurement Model



Typically design process optimization will target core processes and define improvement objectives which come in all four categories - inputs, process, outputs, and outcome. Therefore good process measurements are needed to provide a reference on which to base improvements to the product development team, methodology, and technology. Measurements should evaluate both efficiency and effectiveness. So it is important to select criteria to measure which involves people, process, and technology and determine the overall product development function's performance.

Most product development organization's will rate themselves as satisfactory or healthy. Companies who are typically considered industry drivers will have superior or even world-class product development performance.

Figure 1-5 Evaluating Process Performance



Figures 1-6, 1-7, and 1-8 provide examples of performance measures used to evaluate design process effectiveness.

Figure 1-6 Input Performance Measures

Number of engineers assigned versus plan

Number of years of experience in different areas

Compute environment and access

Software capability and capacity

System requirements defined and stable

Figure 1-7 Process Performance Measures

First pass system design process

Number of system design iterations

First pass block implementation

Number of block implementation iterations

First pass chip-level integration

Number of chip-level integration iterations

Elapsed project time

Total project tasks

Average design iteration time

Number of parallel of design tasks

Number of incremental design tasks

Number of serial design tasks that could be parallel or incremental

Figure 1-8 Output Performance Measures

Productivity measures for design (gates/day), and verification (cycles/day)

Design density (gates/mm2)

Operating frequency (Mhz)

System throughput (operations/sec)

System architecture efficiency (throughput/density)

Power factor (operations/watt)

Verification coverage (cycles simulated,% of system modes tested by AVTs)

Fault coverage (% faults detected)

Reliability (MTBF)

Price ($/component)

Selecting the Improvement Strategy

Assessment of the process condition in the current state by means of process performance measures is required to base any goal setting and implementation planning. Several different process redesign strategies which are utilized throughout industry today are contrasted in Figure 1-9 an include:

- Continuous improvement

Practice ongoing cycle of incremental gains in performance through continuous assessment, gap analysis, and selective process improvement to known processes.

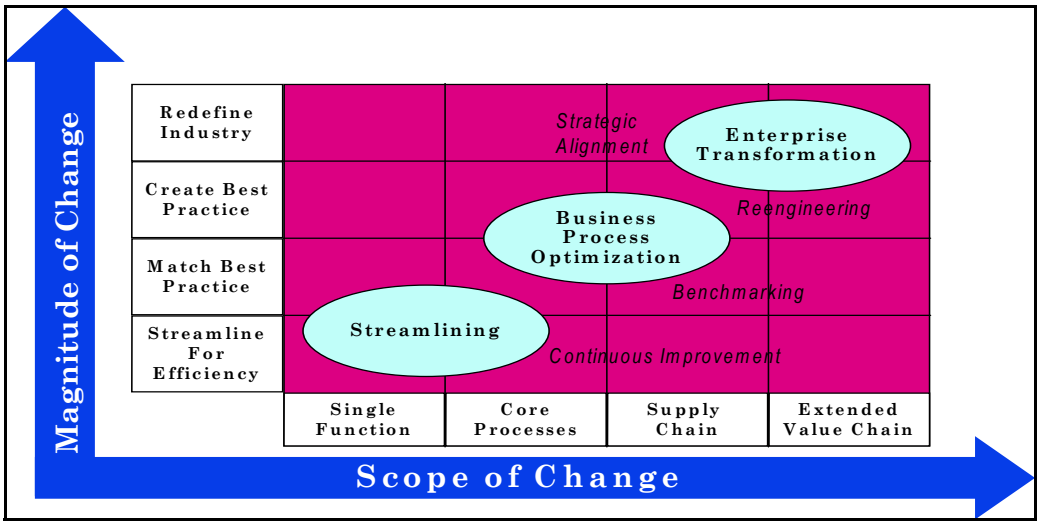
- Benchmarking

  Leapfrog industry performance through combining best practices of one or more competitors.

- Re-engineering

  Drastic and rapid performance improvement accomplished by new process design. Characterized as “breaking the rules”.

Figure 1-9 Process Redesign Strategies



Selection of improvement strategies must balance the level of investment of critical people and funds against desired goal to elevate product development performance. Factors considered in this process include:

- Magnitude of change
- Scope of change
- Timeframe of change

However, these factors alone may not be sufficient in making a selection. The product development organization must also consider more subjective selection criteria and assess its ability to carry out the planned improvement using a specific process redesign approach. Such factors include:

- Level of strategic significance of process redesign to business success

- Culture and attitude of organization to change
- Maturity and stability of management team
- Levels of capital to invest - people and financial
- Skill levels of process redesign team

Together these factors combine to form a total mindset to process redesign which can be summed up as follows:

> *"You can't redesign processes unless you know what you're trying to do. What you're after is congruence among strategic direction, organizational design, staff capabilities, and processes you use to ensure that people are working together to meet the company's goals."*
> *PAUL ALLAIRE, 1995, CEO of Xerox*

Defining Process Improvement Objectives Clearly

Any methodology improvements should have the goal of helping a company achieve certain key business objectives. Some typical business objectives that can be accomplished by methodology improvements include

- Shorten product development schedules
- Lower product cost to manufacture
- Lower product cost to develop
- Increase functionality or performance
- Increase flexibility for related products
- Increase reliability of hardware and software integration

A company's product development strategy may vary based on the scope of the product, type of siliconization (FPGA, ASIC, or deep sub-micron ASIC), and design methodology. This is why no two product development environments are the same. In fact, two companies involved in the exact same niche market will probably have very unique product development environments, formulated by the past experiences, current goals, and objectives of the project leaders. The choices made at any point by this team often have significant impact on the approach to product development and its downstream success.

For example, in the area of design verification, one design team may invest significant engineering and computing resources in full system-level simulation, whereas another might instead adopt a prototype emulation

approach. Although the decisions are different, each may be appropriate to the current objectives and level of team expertise.

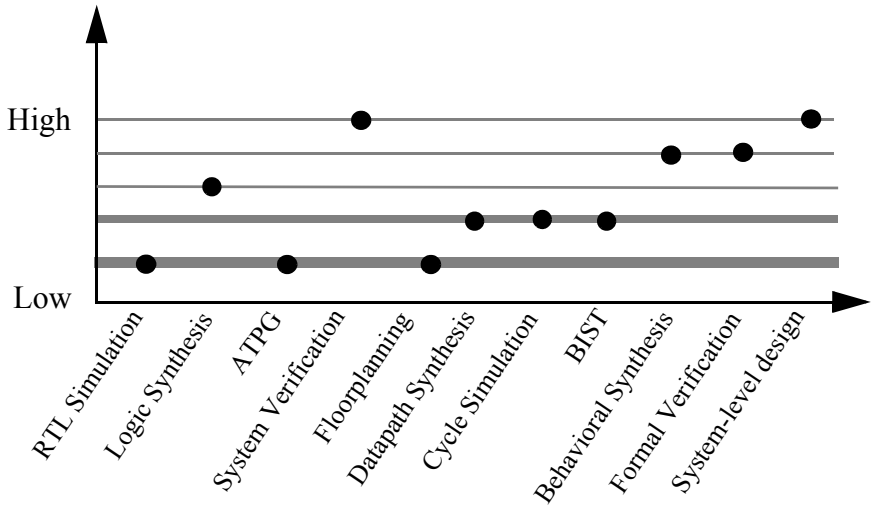
Identifying Process Owners

Ownership and responsibility in planning and leading the transition to new tools and methods is crucial. Project leaders and technology leaders need to be established. These individuals will be responsible for defining and managing specific design methodologies and the design environment. Project leaders need to be established for:

- Verification and simulation
- ASIC and FPGA synthesis, chip composition
- Design for test
- ASIC sign-off and libraries
- CAD
- Networks and workstations

Project leaders should consider the cost of adopting new technology. Investment levels differ for each technology and methodology as illustrated in Figure 1-10.

Figure 1-10 Cost of Technology and Methodology Adoption



Project leaders should articulate the decision criteria governing a methodology shift. For example, the following criteria might be used to evaluate the decision to move into synthesis and system-level simulation. Note that all of these goals are measurable:

- Reduce overall product development schedule by 50%

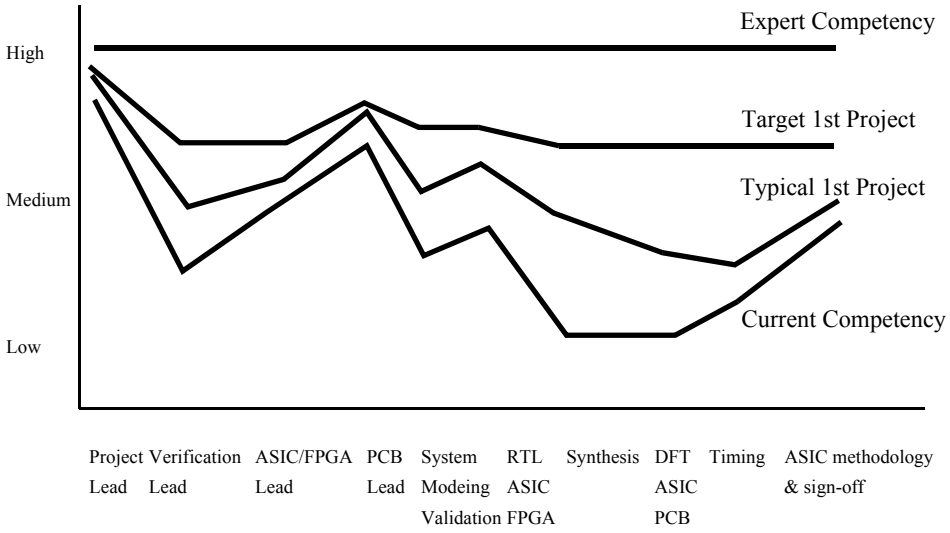
- Reduce time to re-spin and re-validate the design by 50%
- Reduce the chance that a costly re-spin would be required from 100% to 25%
- Decrease time anticipated for debugging hardware prototypes from nine months to three months
- Hire five new engineers to plan/train on use of new methods

The need to define measures is imperative. Top-down design is only as good as the people who practice it. In a recent design contest held at an EDA vendor's user group meeting, 14 design engineers competed for the best design. The design example was a simple counter function. Despite the fact that the group had similar experience levels and over 20 ASICs among them, the results were extremely diverse, varying by 60% in size and 100% in delay.

Planning for Implementation and Transition

Adoption of new methods and technologies must be executed at a pace that allows the design team to learn how to employ them effectively. Companies typically underestimate the time and training required to transition a team to a new methodology or process, as illustrated in Figure 1-11.

Figure 1-11 Expected versus Actual Process Improvement



These shortfalls are usually attributed to

- Difficulty in effecting a change in culture
- Cost of getting started
- Training investment
- Selection process for new technology
- Design methodology development supporting new technology

Evolutionary shifts in the design process can come from factors such as improved designer experience, continual process changes, and incremental tool improvements. This approach normally takes two to three design projects before the expertise levels fully exploit the given design methodology.

Larger, more radical methodology shifts come in the form of paradigm shifts which occur from time to time and result in dramatically improved designer productivity and reduced design intervals. Examples of paradigm shifts include:

- Shift from schematic capture to HDL synthesis
- Shift from prototyping boards in the lab to system level simulation
- Shift from writing test vectors to ATPG and boundary scan tools
- Shift from writing functional vectors to C and C++ software drivers, automatic test bench generation, verifiers, and multi-level regression (system simulation)

A Role for Third Party Consulting

Third party consulting can often provide critical support in a transition to a top-down design approach. There are a growing number of consulting firms that provide a range of services including:

Consulting Services to help plan and design an appropriate product design environment through

- Tools selection and independent benchmarks
- ASIC vendor selection
- Technology library development
- Organization planning and recruiting

Engineering design services to provide resources for ASIC and system development, including

- ASIC and FPGA development
- Design migration
- DFT services
- Model development

Implementation services to develop the team's expertise, including

- HDL and modeling style training
- Top-down design transition training
- Methodology and process support for design, simulation, synthesis, and test

Third party consulting can often mean the difference between costly "on-the-job" training versus starting a project with the expertise required.

Example of Top-Down Design Success

Fulcrum Telecommunications, a subsidiary of Fujitsu, is an impressive example of matching business objectives with product development environment planning and implementation. In the very competitive telephone switching market, Fulcrum has managed to successfully transform its business and regain its position in the market with the successful launch of its next generation switch.

Plagued by reliability issues which caused long delays in getting and keeping new installations on-line, Fulcrum decided that quality and reliability issues were top priority for this project. Enormous erosion of price levels and an explosion in features also made extensive siliconization a must. Unlike many of Fulcrum's larger competitors who had dedicated resources to take on the extensive methodology definition and cross-training, Fulcrum leveraged external consultants to define the design methodology and train the team.

Thorough system verification with rigorous design management and process guidelines, including numerous design reviews and consulting contracts, were crucial in making this product development environment effective for Fulcrum engineers. The telephone switch, which was comprised of six ASICs, ten FPGAs, and 28 PCBs, proved to be the most reliable and robust product ever developed by Fulcrum. The first customer product shipment was installed and carried traffic that same day.

Basic Principles of Top-Down Design

Understanding the basic principles of top-down design is the first step toward implementing the best design practices. These principles influence the objectives for design methodology development. The top-down design approach is based on the following principles:

- Use a hardware description language (HDL) or other high-level programming language to create system and subsystem models as well as reusable cores

  While schematic capture is an appropriate design entry technology for a bottom-up design approach, the Verilog and VHDL languages offer a level of abstraction that makes larger and more complex designs easier to understand. HDLs have multiple abstraction levels, from analytic, behavioral, RTL, and gate-level descriptions. A high-level programming language such as C or other high-level design entry technologies may be appropriate for system models, but current mainstream synthesis tools require RTL descriptions in Verilog or VHDL.

- Validate designs early by developing a system-level verification environment up front

  A system verification environment includes a set of testbenches and models and a detailed, formal test plan for validation of the system. The models and testbenches are a "golden" representation of the design that the team can use to qualify the design of the components. The test plan ensures that the team has considered how to verify all critical aspects of the design as it develops. The verification environment allows the design team to validate the system before implementation and to verify implementations at the RTL level, gate-level, and mixed-level.

- Automate the implementation of the design using synthesis

  Synthesis and optimization technologies allow the design team to explore various implementations of the RTL design before committing to a particular vendor or a particular implementation. This flexibility provided by synthesis tools is critical to attaining performance and designer productivity of large scale ASIC and FPGA designs. Synthesis also enables the reuse of core technologies.

- Develop a design for test (DFT) strategy

  Today's increasingly fast, complex designs offer a formidable challenge to the test engineer. Higher gate-to-pin ratios and a higher

density of board-level interconnects make the traditional bed-of-nails board test unfeasible. The only alternative is to develop a test strategy up front and allow test requirements to influence the implementation of the design, using JTAG, scan, BIST, and other digital logic techniques.

- Provide for consistent data flow between logical and physical design processes for deep submicron ICs and high-speed PCBs

  Logical and physical design of ASICs are no longer separate processes thanks to the advent of deep sub-micron ICs. Chip density and performance after physical design often stray from logical design estimates because of interconnect delays, and the floorplanning of datapaths, cores, RAM/ROM, and system clock distribution. Concurrent optimization of the logic and physical design based on design timing constraints is now required. Chip-level timing analysis and optimization must also be driven by accurate deep sub-micron timing models.

- Manage design data effectively and define design procedures that simplify the effort of iterating design steps

  The amount of design data generated in the process of describing the design, verifying it, and then constraining and analyzing its implementation is overwhelming. In addition, the scope of the design and the breadth of expertise required mean that most design teams have many members. Tracking the status of the various design components and ensuring the integrity of the design data at all phases of development makes a design data management strategy essential.

The core elements of the top-down design process involve HDL modeling of the system and its ASIC components, a comprehensive verification environment, logic synthesis, constraint-driven logical/physical ASIC design iteration process, a complete design-for-test process, and a design environment supporting design data management and release control.

The Top-Down Design Flow and Schedule

Every product development team will implement a design process involving a top-down methodology that best fits the product characteristics and project schedule. Figure 1-12 is a typical top-down design flow diagram. The diagram shows that flow begins with the development of system models and a verification environment (high-level system design). The design team can also start early on to develop a test strategy and to select and to validate vendor libraries.

Once these up-front tasks, including partitioning, are complete, the team can begin the implementation of each block in parallel. Once the blocks are modeled at the RTL level, their functionality needs to be verified within the context of the system. After synthesis and optimization, the gate-level implementation must also be verified within the system context.

Chip integration and sign-off brings all the implemented design blocks together for functional and timing verification, design rule checks, pattern generation, and ASIC vendor sign-off. This part of the design process is very compute-intensive, requiring exhaustive simulations at multiple levels of abstraction. A well-defined test strategy, automated regression techniques and efficient use of network resources help to meet this challenge.

Because of the increasing density of designs, accurate *floorplanning*, or consideration of the overall effect on the design due to the physical aspects of the design and process, is a requirement at all stages of the design process. At the high-level, front-end floorplanners can help to derive accurate boundary conditions, wire models, and timing budgets for design blocks. During block implementation, *silicon synthesis* tools drive synthesis and optimization with the understanding of the physical effects on timing and loading. Floorplanning tools are used during chip-level integration to ensure that the design can be placed and routed successfully.

Figure 1-12 IC Design Flow Diagram

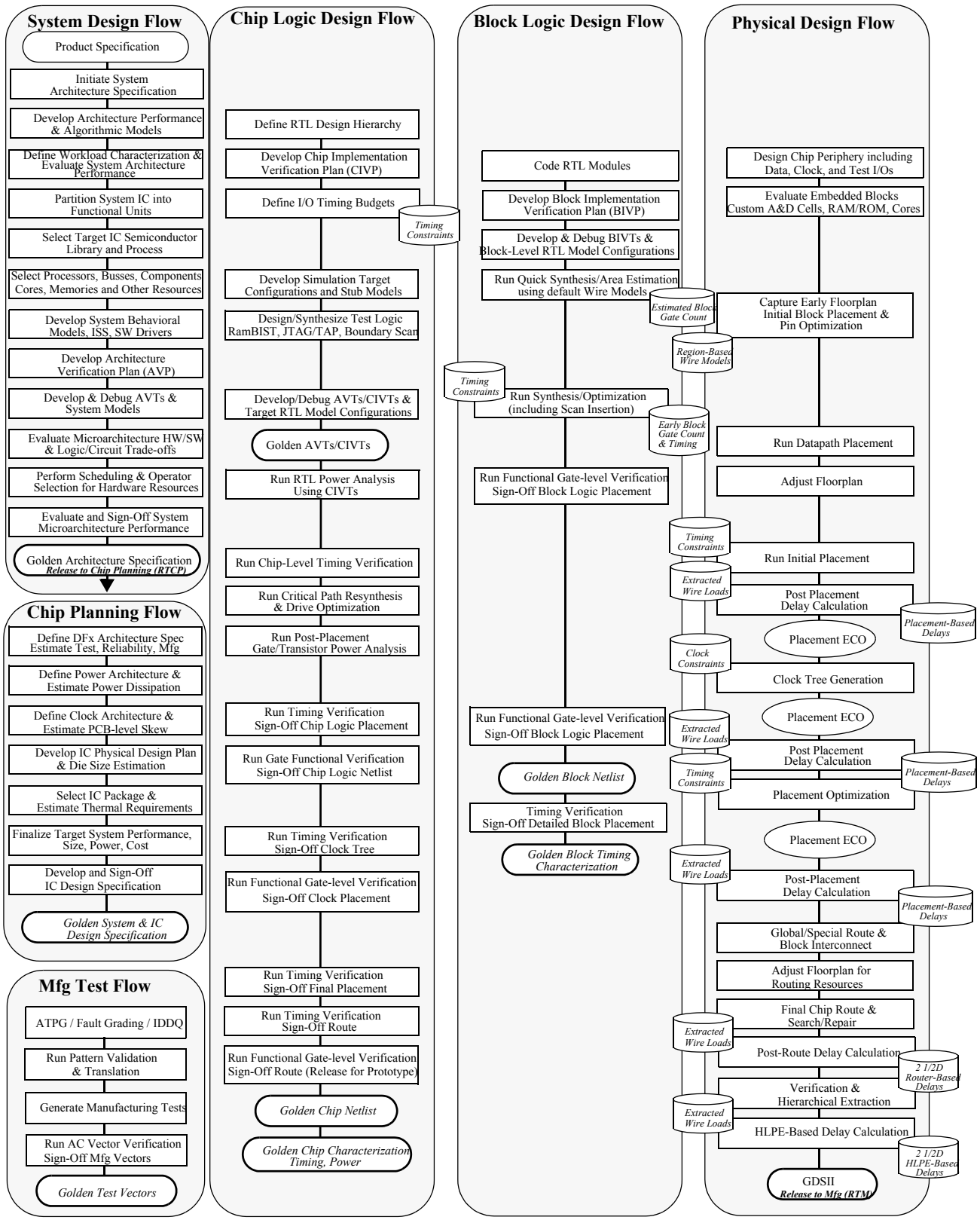


Figure 1-13 shows a sample project schedule for implementing this methodology. The intent of this schedule is to show the major activities and milestones in the process as well as define the relative duration and dependencies of these activities. The actual time line will vary depending on the design.

Figure 1-13 Project Timeline

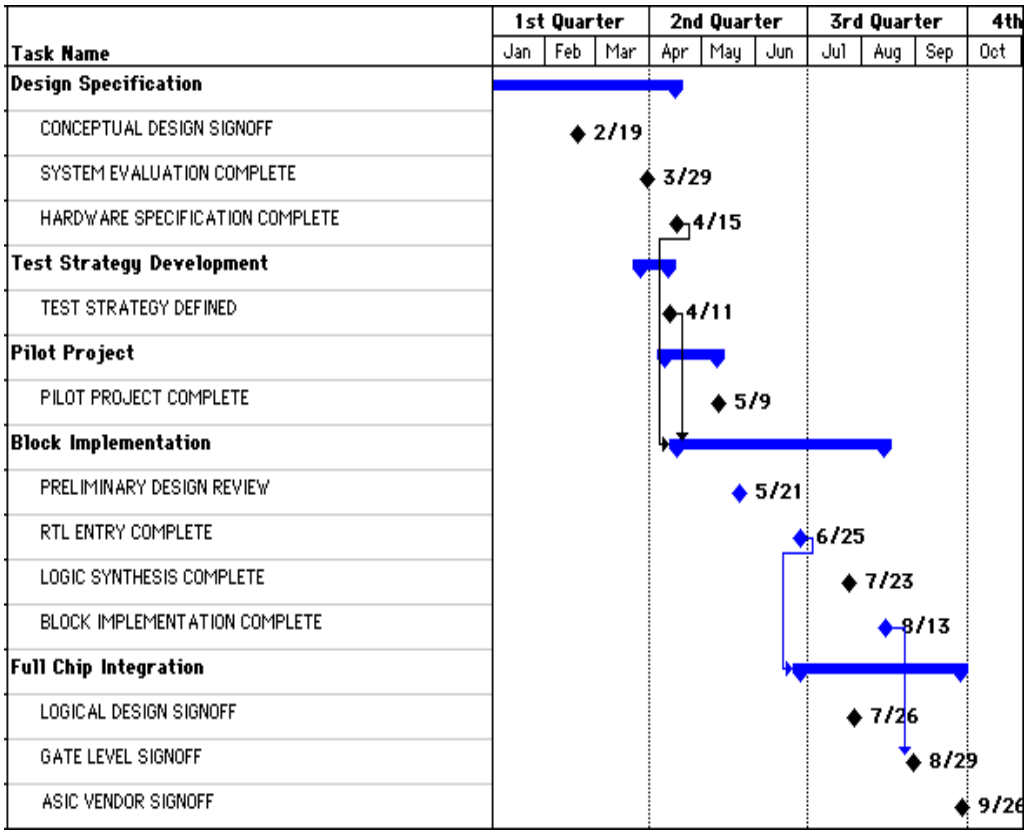
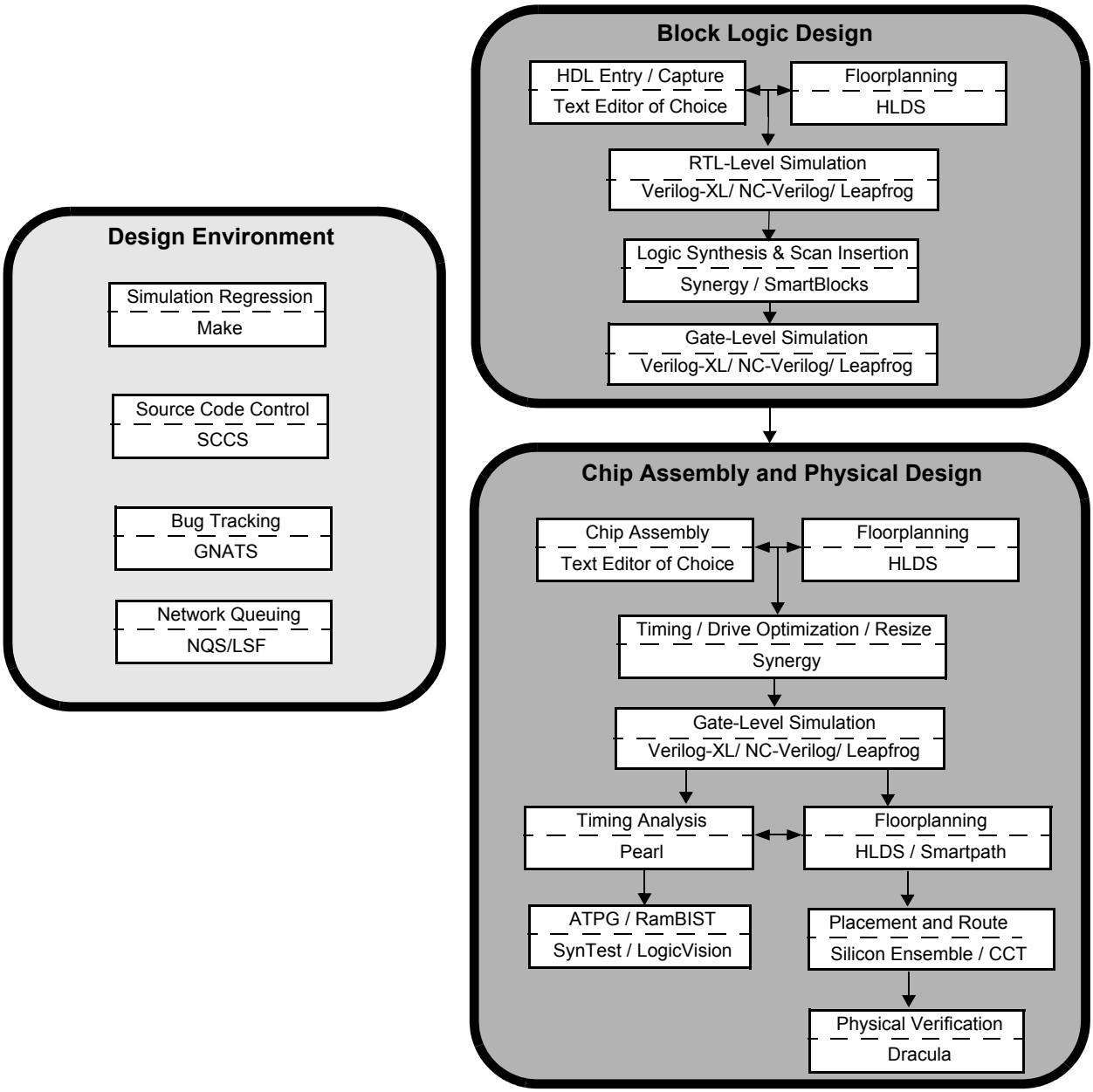


Figure 1-14 shows the tools associated with each part of the block and chip level implementations as well as the design environment. The tools are from a variety of sources including Cadence Design Systems, Synopsys, the Free Software Foundation, and UNIX utilities found on most workstations.

Figure 1-14 Design Tools



Summary

Effective PDEs can create strategic advantages when they enhance a company's ability to produce advanced, high quality products faster and more cost-effectively than competitors. Best practices in top-down methodology for ASIC and FPGA-intensive systems requires continual investments in the people, process, and technology of a company's PDE. Success using top-down design require a solid foundation in the basic principles of the methodology and a focus on continuous development of the methods used. Product development objectives need to impact overall business objectives as directly as possible to exploit market opportunities.

References

[1] Process Redesign, Tenner and Detoro, Addison Wesley, 1997

2

Design Environment Methodology

Design Environment: the Challenges

Virtually every design team and every project has to deal with many complex design environment issues.

First, the sheer amount of design data required to describe, constrain, and automate the implementation and verification of a design is overwhelming. It is very common to have several thousand data files of many different types, including HDL source files, simulation testbenches, synthesis constraint files, and regression scripts, to name just a few.

The scope of the typical design project, the breadth of expertise required, and the time-to-market pressures mean that large design teams are the norm. Without an adequate design environment, it is difficult for multiple designers to share and modify the design files while maintaining the integrity and consistency of the data.

As files are modified and new files are added, the relationships between the files become more complex. The design environment needs to facilitate the grouping or *configuration* of related files.

Many different types of processes, including updates, simulation runs, synthesis runs, and other types of runs, need to be run dozens or even hundreds of times during design development. The design team must also periodically integrate and test the design models to ensure that changes in one portion of the design have not caused problems in other parts.

Given these complexities, it is essential to have a well-defined design environment and consistent data management schemes so that the design process is as predictable and easy to automate as possible. An effective and efficient design environment needs to address objectives in the following areas:

- Design data organization
- Source control
- Configuration management
- Automated processes
- Revision control
- Project tracking

The remaining sections of this chapter discuss these objectives in more detail.

Design Data Organization

An effective strategy for design data organization meets the following objectives:

- Provides a logical and consistent method for storing all types of design data, including source files, configuration files, libraries, executables, and run results
- Gives designers access to the latest tested design data for the entire design
- Isolates designers as much as possible from untested, unstable work in progress

Design data is typically organized and stored in design hierarchies, or *trees*. *Concurrent development* (product development by a team of designers working in parallel on portions of the same design) typically requires three design trees:

- Archive tree

  An archive tree is a set of directories on a disk that is the repository for the files that are under source control. A check-out command copies the appropriate version of the file or files from the archive directory into the designer's local tree.

- Local tree

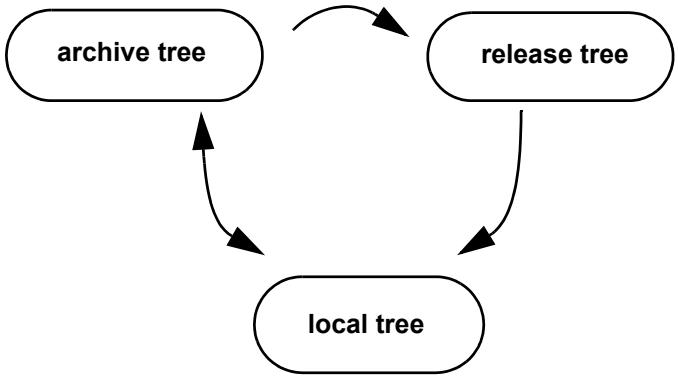
  This set of directories is created by each designer for the purpose of having a work area isolated from other designers. Each designer makes modifications to checked-out source files in this local tree. The designer tests the modifications and then checks the modified file back into the archive tree.

- Release Tree

  This set of directories contains a known good version of the entire design. Because the files in this tree have passed more rigorous tests than the files in the archive tree, designers should reference this hierarchy to get the most recent stable version of the design files needed to test modifications within the context of the entire design.

All of these trees typically have the same directory structure. Figure 2-1 shows the movement of files between these three design hierarchies.

Figure 2-1 Movement of Files between Design Data Trees



Source Control

Source control is a system for archiving versions of files in a central location. The objectives of the source control system are to

- Allow multiple designers access to the same source files while preserving the integrity of the data
- Allow the designers to roll back to an earlier version of the design easily

A source control system typically allows designers to

- Check out a file to view it

  This feature allows multiple designers to view the file at the same time.
- Check out a file to modify it

  When a designer has checked out a file to modify it, the system prevents other team members from checking it out for modification.
- Check in a modified file

  When a designer checks in a modified file, the previous version is archived, and the checked-in version becomes available for other designers to check out. The previous versions of the file are also still available for check out.

Configuration Management

Configuration management is the grouping of a number of distinct files together into a defined set, or *configuration*. There are two basic objectives that configuration management addresses:

- Provide design abstraction management

  In top-down design, a model is first described at a high level of abstraction and then progressively refined to a more detailed representation. Multiple configurations are necessary because different process steps require different sets of files as inputs. It is important to be able to define these configurations once and reference them when needed. These configurations should be simple and explicit so that they can be used as important sources of information about the design.

- Provide version management

  In addition to design abstraction, there will also be multiple versions of each file. The method for configuration management will also need to be able to specify which versions of each file are to be used.

The design environment should support a solution that meets both of these objectives.

Automated Procedures

In order to create, verify, and implement a design, the design team has to repeat many procedures dozens or perhaps hundreds of times. These procedures typically involve invoking a simulation, synthesis, or timing tool with a particular configuration of design data, and then storing the results. The procedure may also include some post-processing of the results to facilitate analysis.

The objectives for automating these types of procedures are

- To reduce the time needed to repeat these procedures manually
- To reduce the errors often involved in repeating the procedures manually

In addition, there are many interdependencies between files. For example, if a component of a design is modified, the other parts of the design that reference that component need to be updated to reflect the modifications. Automating these update procedures is particularly crucial to maintaining the integrity and consistency of the overall design.

Revision Control

Although the terms *source control* and *revision control* are often used interchangeably, in this document the term *revision control* refers to process for promoting and testing files for release, either to the rest of the design team or eventually to the ASIC vendor.

The goals of this process are to

- Preserve the overall integrity and consistency of the design
- Provide a means for tracking progress against project milestones
- Test the procedure for releasing files to the ASIC vendor

A single designer (the *release manager*) should be given the responsibility of chip integration. On a regular basis, this release manager checks out from the archive tree the latest versions of the source files for the entire design. Working in a pre-release area, the release manager runs the regression tests. If the percentage of regression tests that passed is satisfactory, the release manager moves the contents of the prerelease area to the release tree.

The design team needs to agree on the degree of testing required before a file can be checked into the archive tree. This check-in process is not automated. At minimum, each designer should perform unit tests before checking files into the archive tree, to determine that the modifications made have the desired effect and that the files involved are self-consistent.

Bug Tracking

Bug tracking systems are not a requirement for every project. In small design teams, where the designers are responsible for verifying their own blocks and there is only one person in charge of system/chip level verification, it may be adequate to use email and rely on each designer to track his own bugs.

If the design team is large and designers frequently find bugs in portions of the design that other designers are responsible for, the team should adopt a more formal approach to reporting bugs.

The goals of a bug tracking system are to

- Provide a way to report bugs to the person who is responsible for fixing them
- Provide a way to find out the status of a particular bug
- Provide a means for tracking new and fixed bug report rates

An additional benefit of a bug tracking system is that the team can use the new and fixed bug rates as one way of measuring progress against project milestones. This benefit must be weighed against the cost of implementing and maintaining the system.

References

3

Design Environment Implementation

Overview

This chapter describes a particular design environment implementation based on the following software tools:

- UNIX *make* utility
- RCS
- GNATS

Together with common directory structures and file naming conventions, these tools can create a design environment that meets all the goals listed in Chapter 2, “Design Environment Methodology”:

- Provide a logical and consistent method for storing all types of design data, including source files, configuration files, libraries, executables, and run results
- Provide designers access to the latest tested design data for the entire design
- Isolate designers as much as possible from untested, unstable work in progress
- Allow multiple designers access to the same source files while preserving the integrity of the data
- Allow the designers to roll back to an earlier version of the design easily
- Provide a means of configuring design data by abstraction level or version
- Reduce the time needed to repeat these procedures manually
- Reduce the errors often involved in repeating the procedures manually
- Preserve the overall integrity and consistency of the design
- Provide a means for tracking progress against project milestones
- Test the procedure for releasing files to the ASIC vendor
- Provide a way to report bugs to the person who is responsible for fixing them
- Provide a way to find out the status of a particular bug
- Provide a means for tracking new and fixed bug report rates

Design Data Organization

Designs typically have a hierarchical structure shown in Figure 9-1. This structure is used as a basis for the directory structure described in Figure 9-2.

Figure 9-1 Design Hierarchy

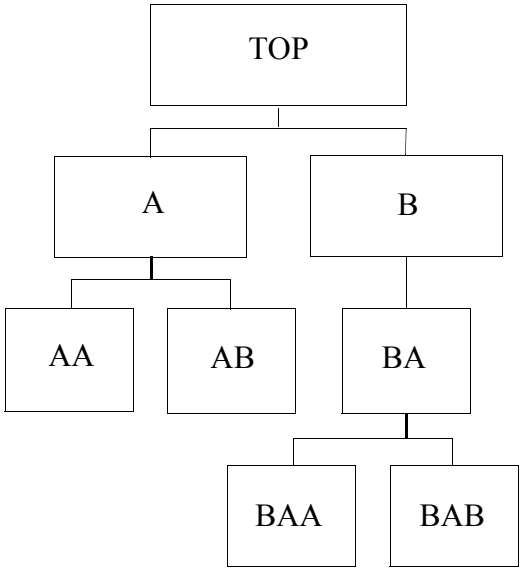


Figure 9-2 Directory Structure

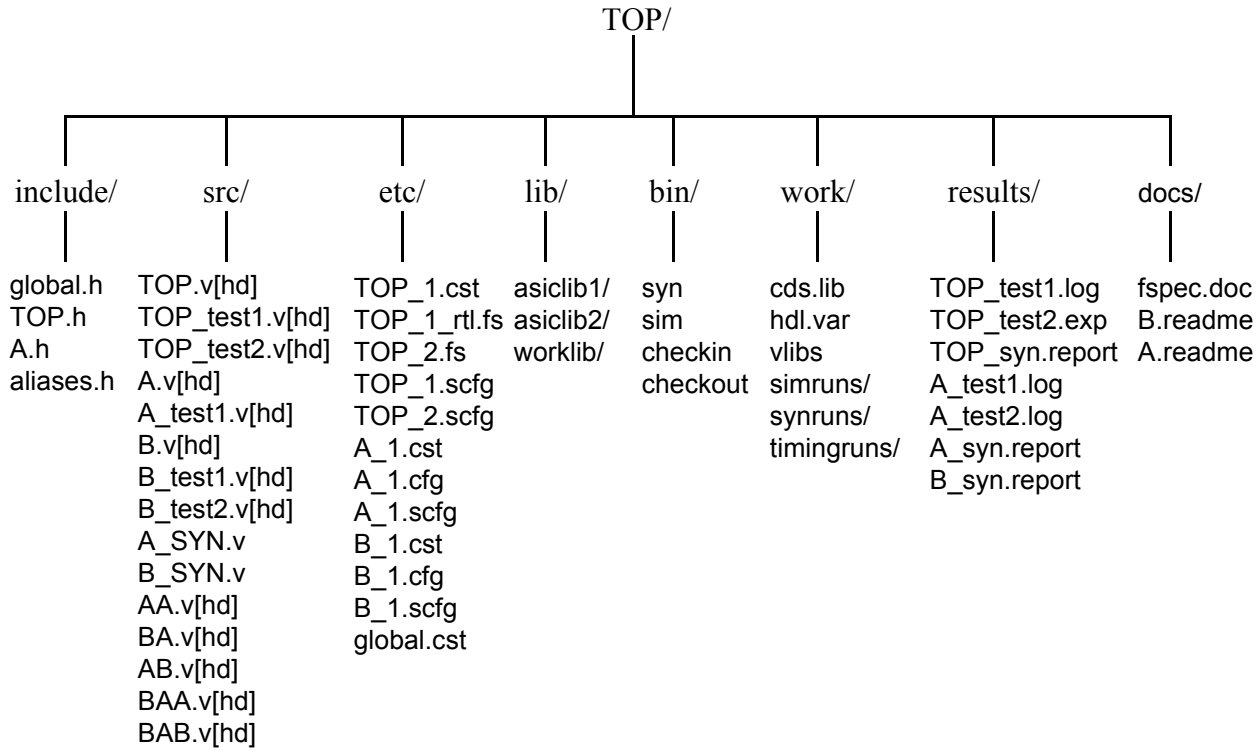


Figure 9-3 describes the function of each of the directories shown in Figure 9-2.

Figure 9-3 Directory structure summary

Directory	Description
include	Location for files containing shared HDL code, such as header files in Verilog, VHDL packages, or memory data files
src	Location of HDL source files, including RTL files, test fixtures, and synthesized netlists
src/RCS	Location of the archived versions of files
etc	Location of design configurations, simulation configurations, synthesis command files, and waveform restoration files
lib	Location of libraries, including reference libraries, ASIC libraries, and VHDL design libraries
bin	Location of scripts specific to this particular project

Directory	Description
work	Location for invoking simulation, synthesis, or timing runs
work/timingruns	Location of timing analysis run directories
work/simruns	Location of simulation run directories
work/synruns	Location of synthesis run directories
results	Location of simulation results, synthesis report files, other relevant report files
docs	Location of any relevant documentation related to the design, including functional specs

File Naming Conventions

In order to make directory and file structure consistent and to facilitate automated procedures, file naming conventions are a requirement. Figure 9-4 describes the recommended file naming conventions.

Figure 9-4 uses the following terms:

- <design> refers to the module name of the block that the particular file is associated with.
- <ident> is a user-definable name to distinguish between multiple types of the same file. For example, there can be many different design configurations or synthesis command files for a particular block in the design. rtl.cst
- <test> is a user-definable name to label a particular test case.

Figure 9-4 File Naming Conventions

File Type	Verilog Extension	VHDL Extension
RTL Source File	<design>.v	<design>.vhd
RTL Design Configuration	<design>_rtl.fs	cfg_<design>_rtl.vhd
Testbench File	<design>_<test>.v	<design>_<test>.vhd
Synthesis Command File	<design>_<ident>.cst	<design>_<ident>.cst
Simulation Configuration	<design>_<ident>_<test>.fs	cfg_<design>_<ident>_<test>.vhd
Synthesized Netlist	<design>.vs	<design>_syn.vhd
Synthesized Design Configuration	<design>_syn.cfg	cfg_<design>_syn.vhd
SHM Database	<design>_<test>.shm	<design>_<test>.shm
Simulation Log File	<design>_<test>.log	<design>_<test>.log
Simulation Expected Results	<design>_<test>.exp	<design>_<test>.exp
Waveform Restoration File	<design>_<ident>.wrf	<design>_<ident>.wrf

Source Control

RCS/SCCS is a standard revision control and source control system available on UNIX. Figure 9-5 summarizes the RCS command set required to implement an effective design environment.

Figure 9-5 RCS Command Set

Command	Description
ci	Check in revisions from work area to RCS tree.
ci -l	Check in revisions from work area to RCS tree and maintain the lock for modifications.
co -l	Check out file from RCS for write.
co	Check out file from RCS for read

Checking In a File

To check in a file, the designer follows this procedure:

1. **Change directory to the local /src directory.**
2. **Type the following at the UNIX prompt**

```
ci file.v
```

RCS responds with the following message

```
RCS/file.v,v  <-- file.v
new revision: 1.3; previous revision: 1.2
enter log message, terminated with single '.' or end of file:
```

3. **Type a description of the changes made to the file.**

RCS removes the file from the local directory and copies it to the RCS directory in the archive tree.

Notice that the command prompts the designer for a description of the changes by asking for a log message. This can be multiple strings followed

by a single "." on a line by itself. It is also possible to give the description in the command line as shown below

```
%ci -m"Changed logic level on rst port.\
." file.v
```

Alternatively, to retain the lock on the file so that edits can continue to be made, the designer can use the -l option as shown below. This command does a check-in and then performs an automatic check-out for modification.

```
%ci -l -m"Changed logic level on rst port.\
." file.v
```

Note that all of these commands can take multiple file names as arguments and therefore wildcards such as "*.v" can be used.

Checking Out a File

To check out a file, the designer follows this procedure:

1. **Change directory to the local /src directory.**

2. **Type the following at the UNIX prompt**

```
co -l file.v
```

RCS sets the lock on the file and makes a copy of the file in the local /src directory with the following message:

```
RCS/file.v,v  -->  file.v
revision 1.1 (locked)
done
```

Configuration Management

There are two type of configurations

- Design configurations
- Simulation configurations

The following sections discuss these two types of configurations.

HDL Design Configurations

Design configurations contain the necessary information to define exactly what files or cell descriptions make up a particular block or entire design. These files are used as input to simulation, synthesis, timing analysis, and fault grading to specify the source files that define the design block.

The run directories for these tools are created within the *simruns*, *synruns, timingruns, and faultruns* respectively. *timingruns* and *faultruns* directories will exist at the top level of the design to be used during the full chip integration process.

Verilog Design Configurations

A Verilog HDL configuration is a list of files that contain all of the modules that make up the design, not including the simulation test bench. File names should use relative path names, because the absolute paths in the local, archive, and release trees are different. For example

```
../../../src/A.v
../../../src/AAA.v
../../../src/AAB.v
```

It is also possible to use -y and -v options to specify a configuration. This method uses a Verilog-XL search path algorithm to resolve all module definitions in the design given the top level.

The designers may create more complicated design configurations by explicitly creating configuration files that are an arbitrary mixture of abstraction levels.

Designers can create these configurations easily by using some basic UNIX commands. The example below shows the creation of a configuration containing all the .v files visible in the /src directory. This would be run from the local /etc directory.

```
% ls ../src/*.v |sed 's:^:../../:' > newcfg.cfg
```

VHDL Design Configurations

A VHDL configuration has a syntax defined within the VHDL language that is independent of file system location. VHDL simulators provide an automated way to generate these configurations. Designers can also manually create configurations by editing a file which contains a VHDL configuration specification. An example of a VHDL configuration is shown in Figure 9-6.

Figure 9-6 VHDL Configuration

```
-- Configurations for top-level unit

VHDL_LIB.TOP:STRUCTURE

-- Configuration Model: Hierarchical
-- No priority list of architectures specified

-- Configuration
configuration CFG_MY_NAND_RTL of MY_NAND is
    for RTL
    end for;
end CFG_MY_NAND_RTL;

-- Configuration

configuration CFG_MY_REG_RTL of MY_REG is
    for RTL
    end for;
end CFG_MY_REG_RTL;

-- Libraries

library VHDL_LIB;

-- Configuration

configuration TOP_STRUCTURE_CFG of TOP is
    for STRUCTURE
        for OTHERS: MY_NAND use configuration
            VHDL_LIB.CFG_MY_NAND_RTL;
        end for;
        for OTHERS: MY_REG use configuration
            VHDL_LIB.CFG_MY_REG_RTL;
        end for;
    end for;
end TOP_STRUCTURE_CFG;
```

HDL Simulation Configurations

A simulation configuration contains all of the input that a simulation needs to run. This includes:

- A design configuration to tell the simulator what design to test
- A testbench to specify what test to use
- Simulation options to further control the simulation run

Verilog Simulation Configurations

A Verilog simulation configuration is very similar in structure to a Verilog design configuration. It is a file that contains the testbench, the design configuration, and any simulator options listed one per line. An example is shown in Figure 9-7.

Figure 9-7 Verilog simulation configuration

```
../../../src/design_test1.v
-f ../../../etc/design_rtl.cfg
+define+debug
+turbo+3
```

VHDL Simulation Configurations

A VHDL simulation configuration is in a format that is similar to the Verilog simulation configuration. The options and command line arguments must be specified one per line.

Figure 9-8 VHDL simulation command file

```
-OUTPUT top_str.sv.log
-INPUT files/presyn_sv.cmd
-BATCH
-RUN
-UPDATE
-SOURCE
-SHMDB top_str.c.shm
VHDL_LIB.TOP:STRUCTURE/SIM
```

Automated Procedures

To achieve the goal of design process automation, it is necessary to have an easy to use and repeatable method for invoking the following processes:

- Interactive RTL simulation
- Batch RTL simulation, which includes regression testing of multiple test suites
- Interactive gate level simulation
- Batch gate level simulation, which includes regression testing of multiple test suites
- Batch synthesis modeling style check
- Interactive logic synthesis
- Batch logic synthesis
- Interactive timing analysis
- Batch timing analysis
- Batch fault simulation
- Updating an entire sub-block
- Updating an entire block
- Updating the entire chip
- Updating the entire system

make is a UNIX utility that automates the running of flows and processes. *make* intelligently decides which steps in a design flow need to be rerun when certain inputs have changed. A corollary to this is that *make* verifies that a design is up to date and does not unnecessarily rerun steps such as simulation or synthesis.

Unfortunately, *makefiles* are somewhat cryptic and it is not desirable to require all designers to maintain or edit them. One way of integrating *make* into a design environment is to provide a script which will automatically create simple *makefiles* for specific simulation or synthesis runs and execute them.

The procedure for invoking any design step such as synthesis or simulation is to type the command followed by the configuration name to be run. For synthesis, the configuration is a command file. This makes the path from

the interactive to batch runs very simple because the command file created during an interactive run can be used as the synthesis configuration.

As an example, assume the following input to a simulation run.

- Simulation Configuration: top_rtl_test1.scfg

```
../../../src/top_test1.v
-f ../../../etc/top_rtl.cfg
+turbo+3
```

- Design Configuration: top_rtl.cfg

```
../../../src/dut.v
../../../src/top_rtl.v
```

By typing **"sim top_rtl_test1"**, the *makefile* shown in Figure 9-9 would be created and executed.

Figure 9-9 makefile for Simulation

```
.top_rtl_test1: ../etc/top_rtl_test1.scfg
../etc/top_rtl.cfg ../src/top_test1.v ../src/dut.v ../src/top_rtl.v
   @echo "*******************************************************************************"
   @echo " Simulation started \`top_rtl_test1' Simulation Configuration"
   @echo " Run Directory    : simruns/top_rtl_test1.run "
   @echo " "`date`
   @echo "*******************************************************************************"
   @mkdir -p simruns/top_rtl_test1.run; \
   (cd simruns/top_rtl_test1.run; \
   verilog -f ../../../etc/top_rtl_test1.scfg); \
   touch .top_rtl_test1

../etc/top_rtl.cfg :
../etc/top_rtl_test1.scfg :
../src/top_test1.v :
../src/dut.v :
../src/top_rtl.v :
```

To create the Verilog makefile shown in Figure 9-9, a Perl script extracts the files upon which the simulation is dependent and builds the *makefile*. If this simulation has already been run and no input files have been modified, it will not be run again.

Tracking Bugs

GNATS is a bug tracking system available from the Free Software Foundation. GNATS has a simple Motif-based graphical-user interface and has the necessary features to implement a useful system for the tracking of bugs. The features include:

- Querying and editing existing problem reports
- Organizing problem reports into a database and notifying responsible parties of suspected bugs
- Allowing support personnel and their managers to edit, query and report on accumulated bugs
- Providing a reliable archive of problems with a given program and a history of the life of the program by preserving its reported problems and their subsequent solutions.

Although the system is customizable, there are four built-in problem states

- Open - the initial state of every PR; this means the PR has been filed and the person or group responsible for it has been notified of the suspected problem
- Analyzed - the problem has been examined and work toward a solution has begun
- Feedback - a solution has been found and tested at the support site, and sent to the party who reported the problem; that party is testing the solution
- Closed - the solution has been confirmed by the party which reported it

Using the Design Environment

In order to use the design environment described in this chapter, each designer has to create a local work tree. The designer also needs to set the following environment variables:

```
setenv RCS_TREE /usr1/<project_name>_RCS
setenv RELEASE_TREE /usr1/<project_name>_RELEASE
setenv WORK_TREE /usr1/<user_name>/<work_area>
```

A typical working model for designers is to have two windows open, one in the local /src directory and one in the local /work directory. The designer checks HDL files out of RCS and modifies them in the local /src directory. The designer invokes simulation or synthesis runs from the local /work directory. To modify a configuration, the designer changes directory to the /etc directory and checks out the appropriate configuration file.

To invoke a simulation, synthesis, or timing run, the designer changes directory to the local /work directory and invokes the appropriate *perl* script to create the *make* file.

References

[1] “Article” Book, Author, Date

Lab Exercise: Introduction to the Design Environment

A recommended design environment is described in *An Approach to Top-Down Design.* This lab walks you through the fundamentals of how the design environment is set up, where data is created, how data is managed, how data is shared, and how tools are invoked.

1. **Change directory to *dtmf_proj* directory. Here you will find the following directories:**

Directory	Use
RCS/ or SCCS/	This is the source control directory for the project. All files for the project are checked in and out of this directory. The labs will use RCS or SCCS (Source Code Control System) as the source control tool.
bin/	Location for any executable scripts that are used for the project.
docs/	Location of any applicable documentation for the project.
etc/	Location of design configurations, simulation configurations, synthesis command files, SDF files, and waveform restoration files.
include/	Location of files containing auxiliary HDL code such as include files, memory data files, or VHDL packages.
lib/	Location of the synthesized netlists for each design unit.
src/	Location of HDL source files that have been checked in to the RCS or SCCS directory and checked out into *src*. These files are verified and can be shared with other designers.
work/	This is the user's work directory. Each designer has a separate work area where changes to design data can be made without affecting other designers. Within the work directory, each designer can optionally create a local archive to control design data.

Directory	Use
results/	Location of results from analysis jobs to be saved; synthesis reports, simulation log files, waveform databases, scan chain files, timing analysis reports, etc.

2. **Change directory to the work directory and source the *.sourceme* file.**

   This file sets up the user's shell environment for this project. Environment variables are set to define the project archive directory and the user's path is set to include the project *bin* directory.

3. **Create the file *digit_reg.v* or *digit_reg.vhd* depending on the language you are using.**

   Code this module given the module description in this *DTMF Design Description* chapter of the Lab Manual.

4. **Verify the syntax and synthesizability.**

   Run synergy and perform a synthesizability check or use the *check* script provided.

   ```
   verilog : check digit_reg.v
   vhdl : check digit_reg.vhd
   ```

5. **Create an RCS or SCCS history file for the initial version using:**

   If you're using SCCS, type

   ```
   sccs create <filename>
   ```

   If you're using RCS, type

   ```
   ci <filename>
   ```

   This creates a file called *s.<filename>* in the SCCS project directory and automatically checks out a read-only copy of this file in your current directory.

6. **Verify the functionality of the module by creating a test case (*digit_reg_test.v* or *digit_reg_test.vhd*).**

   To edit a file and make changes use:

   If you're using SCCS, type

   ```
   sccs edit <filename>
   ```

   If you're using RCS, type

   ```
   co -l <filename>
   ```

   This makes the file writeable and allows you to edit it with a text editor. To update the history file use the command(s):

   If you're using SCCS, type

   ```
   sccs delget <filename>
   sccs deledit <filename>
   ```

   If you're using RCS, type

   ```
   ci <filename>
   ci -l <filename>
   ```

   This checks in the changes and checks out a read-only or a writeable copy of the file.

7. **When the functionality is correct and the block is synthesizable, check the file in, using *sccs delget*, or *ci.***

   ```
   sccs delget <filename>
   ```

8. **Check the latest version out into the project source code area.**

   If you're using SCCS, type

   ```
   cd ../src
   sccs get <filename>
   ```

   If you're using RCS, type

   ```
   co <filename>
   ```

9. **Synthesize the digit_reg using the constraints provided in the *etc/digit_reg.cst* file. You can use the *syn* script provided.**

   ```
   verilog : syn digit_reg.v
   vhdl : syn digit_reg.vhd
   ```

10. When synthesis has finished, install the synthesized netlist in the *work/lib* directory :

```
cp synruns/digit_reg.run/syn.v lib/digit_reg.vs
```

Note the Verilog netlist is the data we will move forward to sign-off with for both Verilog and VHDL users.

Turn in the following:

digit_reg.v or digit_reg.vhd

digit_reg_test.v or digit_reg_test.vhd

digit_reg.chains

4

Design Capture Methodology

Design Capture: The Challenge

Factors that make design capture a challenge today include

- Design complexity

  Two decades ago, designers modeled electronic circuits at the transistor level. When semiconductor technologies advanced and design complexity increased, designers transitioned to a new level of design abstraction—gate-level—to handle the increase in complexity.

  A decade later, when technologies again advanced and design complexity again increased, designers turned to hardware description languages (HDLs) in order to reduce the complexity of the design to a scale that the human mind can grasp.

  Increasing the level of design abstraction is one way to meet the challenge of increasingly dense, complex designs. Today's technologies require an additional level of abstraction—system-level design. System-level design techniques help the designer deal with the problem of complexity. They also allow new opportunities to experiment, explore, optimize, and verify before implementation.

- More logic on the die

  As design complexity increases, designers try to fit more and more logic on the the die which creates a prodcutivity gap in specifying and verifying the functionality.

- Intellectual Property (IP)

  To meet time-to-market pressures, design groups are increasingly outsourcing the design of their components to other companies. A critical concern to those companies is the protection of their IP. Companies want to be able to hide the implementation details of cells while providing the models designers need to verify the functionality and timing of the overall system**.**

- Design reuse

  Design teams can use design capture methods such as using parameterized models that promote the reuse of components.

The Goals of Design Capture

In the top-down design approach, designers create models at increasingly lower levels of abstraction during the design process. The designers typically begin by creating system models, which are at the highest level of abstraction. When the design concept or *system intent* has been validated, the designers create implementation models represented as HDL models for portions of the design at the next lower level of abstraction.

HDL models can themselves be written in varying levels of abstraction—behavioral, functional, and structural. The levels generally represent nonsynthesizable, synthesizable, and gate level, respectively.

Designers use a behavioral model when they are modeling a block they will implement using a method other than synthesis. Designers also use behavioral models as stub models to aid in functional verification.

Beneath the behavioral level is the RTL, or functional level. RTL models are written using modeling styles to ensure that they are synchronous so that they can be implemented by synthesis tools.

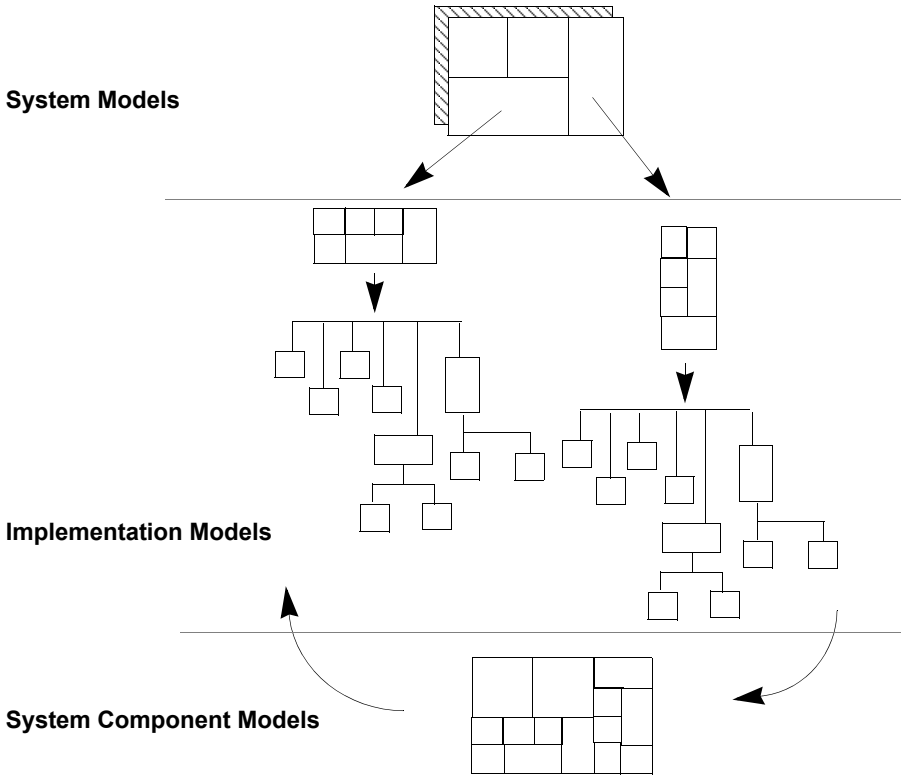
Sometimes the best method for a portion of a design is to directly instantiate existing cells. This level of abstraction is the gate level. The models at this level are usually created automatically by synthesis tools or data path generators. Designers may still choose to create these gate-level implementation models by hand, using a schematic generator.

In the last phase of the design process, the gate-level implementation models are integrated to form a system component model. When this model has been verified, then the design is ready for hardware prototyping.

The scope of the models, as well as the abstraction level, changes during the design process. The system models have the broadest scope. Some of those system models may model key aspects of the entire system as a single unit; others may partition the system into subsystems. In the implementation models, both RTL and gate-level, the subsystems are further decomposed into smaller subblocks.

Figure 3-1 shows the partitioning and integration of design models through the design process in relation to the types of models.

Figure 3-1 Partitioning and Integration of Design Models



Model Development Planning

Model development commences with high-level systems design. During high-level systems design, the system architect determines what models will are developed based on the required amount of detail necessary to implement and verify system function and performance. This information is captured in a model development plan which defines the models and their types. There are two major process steps which drives model development in systems design.

- System decomposition
- Functional unit partitioning

System decomposition determines the high-level system resources including processors, busses, and memories. Functional unit partitioning refers to the allocation of the system functions into one or more ICs where now physical boundaries are considered fully. Factors which often influence functional unit partitioning include the major system dataflow and busses, clock domains, all software controlled registers, memories architectures, processing units and associated datapaths, package pin count, package size and power trade-offs. Functional unit partitioning often results in refining the architecture further in what is commonly referred to as microarchitectural optimization. From this partitioning, functional unit interfaces includ-

ing data formats and encoding are decided.

As system architects and ASIC designers gain a better understanding of the general partitioning from refining the system level specifications out, more detailed model development is specification to the model development plan. Several different model types will be developed through the product development. A brief comparison of model types is shown in Figure 3-2.

Figure 2-2 Comparison of Model Types

	System - Behavioral	System - Performance	Implementation
Purpose	verification	performance analysis	design implementation
Structure	arbitrary	tracks architecture	tracks implementation
Language	full HDL	PLI, C	synthesis style
Datatypes	complex types	tokens, queues	bit, vector, integer
Speed	10-100X implementation	100-10000X implementation	10X structural
Effort	4 weeks to 6 months	4 weeks to 6 months	9-12 months
Timing	asynchronous, zero time, or cycle true	mean system throughput	cycle true
Accuracy	passes 90%+ ASIC tests and 100% subsystem tests	N/A	passes 100% ASIC and 100% subsystem tests

This chapter will review the various model types utilized in ASIC-based systems design.

System Models

The goals of system models are to

- Capture the key performance issues in the design so that the system performance can be measured

  It is important to know whether the system being designed has the capacity to handle the expected work load.

- Capture the key system algorithms

Creating a behavioral model of the entire system allows the opportunity to create new algorithms, optimize known algorithms for a new context, or select the best of several known algorithms.

- Define and optimize the system architecture, partitioning, and packaging

  The high-level system design phase is the time to evaluate alternatives in the size and performance of various subsystems and in the partitioning of functionality and algorithms between hardware and software or between on-chip and off-chip. It is also the right time to select the appropriate packaging. Trade-offs made at this point in the design process far surpass the effects of downstream optimization.

- Clarify system requirements

  Creating a system model can bring to light system requirements that are not feasible, not well-defined, or not well-understood by all members of the team. Once the model is complete and signed off, it becomes the executable system specification.

Behavioral HDL Models

At the behavioral HDL level of abstraction, the goals of design capture are to

- Isolate portions of the design for later implementation

  Designers create behavioral HDL models as black boxes that will pass through synthesis unchanged. For example, they might use behavioral models for embedded blocks of RAM or ROM to be implemented by a module generator. Or they might model data paths as behavioral models to be implemented by a data path compiler.

- Create stub models for verification

  Designers also use behavioral models as stub models, or placeholders for components that have not yet been implemented, to aid in functional verification. In this case, the designer also writes a functional Register Transfer Level (RTL) design for synthesis.

  Create hybrid models

  Designers create a hybrid model with both behavior and analytical models. These models system context as analytic model. They are useful when system context processing rate is much slower than model under test. An analytic model of a sub-system provides a probabilistic distribution function and queuing model that models the system's processing response rates based on high-level system parameters.

RTL Implementation Models

During the block-level implementation phase, the goals of design capture are to

- Create a synthesizable design

  Synthesis tools require designs modeled at the Register Transfer Level (RTL) of abstraction. These models are less abstract than the system models in that they have to model not just the algorithm, but also the behavior of an electronic circuit at each clock cycle.

  Synthesis tools typically require an HDL description as input, and the description has to follow a synthesis modeling style for the tool to recognize the behavior that is being modeled.

- Design at a high level of abstraction to increase the range of possible implementations

  Synthesis tools can implement gate-level HDL descriptions as well as RTL. However, to fully benefit from the flexibility that synthesis tools provide, it is better to model the design at the highest level of abstraction possible—at the finite state machine level, for example, not at the flip-flop level.

- Partition the block properly

  Synthesis tools perform best on *design chunks* of 5000 gates or less, so large subsystems have to be further partitioned. Some issues to be taken in consideration when partitioning include

  - Clock domains
  - Data path and control logic
  - DFT requirements

- Reuse previously designed, optimized cores

  The block-level design should make as much use as possible of previously designed and optimized cores. This approach reduces design development time and increases the confidence in the quality of the end product.

Structural Models

At the structural level of abstraction, the design capture goals are to create design modules that are either non-synthesizable or lend themselves to hand instantiation. Typical modules that are captured at the structural level are:

- Test structures
- Tri-state bus drivers

- Asynchronous logic
- I/O pads

System Models

System models can be divided into the following categories:

- System specifications

  System specifications play an important role in the definition of system requirements.

- Analytic models

  These models analyze the performance of the system.

- Behavioral models

  These models explore different algorithms and partitioning for system control and processing.

- Hybrid models

  These models combine performance modeling and algorithm development in one set of models.

- Block diagrams

  These diagrams are a graphical representation of the partitioning of the system.

System Specifications

The system-level engineering specification documents the system requirements. This specification is not only a document that can be reviewed, but also includes an executable system model that can be used to verify functionality of the system-level HDL simulation.

An additional specification is created for each subsystem. Like the system specification, it is reviewed and agreed upon by all the team members. When properly done, it acts as form of interface contract between the parties involved in the design project. Below is an outline of a specification.

Functionality

This section is an overview of the operations of the system or subsystem. It gives a general outline of each of the major logic blocks and busses of the ASIC.

Performance

This section gives expected performance of the system or subsystem, usually expressed in terms of latency through the device, throughput, or operations/second. It is very important to establish the performance targets so that the designers have specific goals and know what to optimize for. If necessary, the designer can always re-negotiate these performance values should the design not meet them.

Gate count

Overall gate count is estimated here.

Power

Overall power dissipation is also estimated here, according to the target vendors specs.

Block Functionality

These sections describe the functions of each block in the design. The I/O of each block is described. Timing diagrams show cycle accurate waveforms for every block I/O. A target gate count is given. The block performance expectations are described.

Macro Blocks

Macro blocks are the predesigned blocks that will be implemented in this system or subsystem. The design documentation locations for these blocks are noted.

Test

This section describes the scan methodology that will be implemented in the design. IEEE 1149 compliance could be noted. The target test coverage is given.

This section also covers the functional vector tests that will be used to test the device. Expected test coverage may be detailed as well as the test speed.

Vendor Process

This section covers the targeted vendor process. Vendor libraries are noted, including the specific temperature, voltage, process and wire load models.

Custom Cells

This section covers any custom vendor cells required. A detailed specification for each of the cells needs to accompany each cell. The specification is to be agreed upon by both vendor and purchaser.

Clocking

This section details the clocking mechanism to be implemented. This includes: minimum clock rate (max. frequency), maximum clock rate,

internal clock skew, off-chip clock skew, and scan clock rate. Minimum and maximum clock duty cycles are noted.

Package

The package type is noted here. The package pin out layout, minimum and maximum die size, and package dimensions should be noted if available.

Operating voltages

The operating voltages include tolerances are included here.

Operating currents

The operating current ranges are included here.

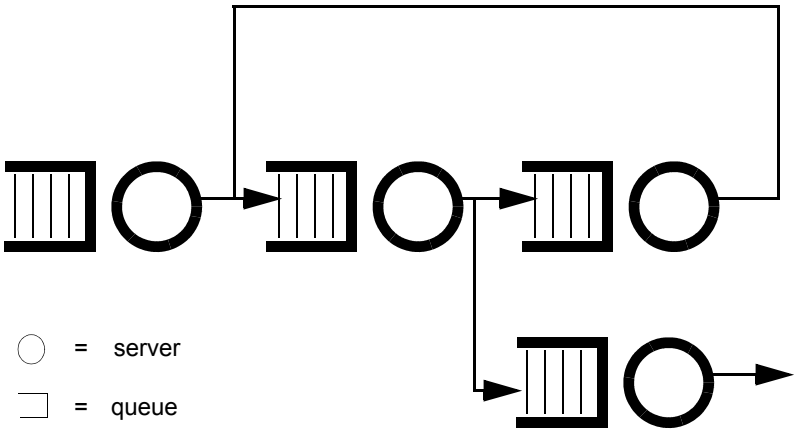
Thermal

The thermal characteristics of the package, heat-sink (if applicable), cooling mechanism, maximum and minimum ambient temperatures, Theta JC, Theta JA, and elevation are provided so that minimum and maximum device temperatures can be calculated.

Analytic Models

The objective of analytic models is to verify that the system has enough capacity to handle the expected work load. In most complex systems, there is a contention for system resources. For example, in a computer system there may be bus contention, where multiple devices access common memory devices across a common bus. In a networking system, many systems may need to transmit and receive data through a common switch. If the system resource—the bus or the switch—does not have enough capacity, then a queue of work—jobs or data packets—begins to form, and system *throughput* (total jobs completed divided by the total time) degrades from the peak performance. Analytic models define the system in terms of servers (system resources) and queues. Analytic models can be represented graphically as shown in Figure 3-3.

Figure 3-3 Analytic Network Queuing Diagram



Each resource in the system is assigned a service time that represents the rate at which work is accomplished. The service time can be constant for a particular resource or described with a probability distribution function based on a mean service time. Figure 3-4 shows the probability distribution functions commonly used to describe service times.

Figure 3-4 Probability Distribution Functions

Function	Parameters
Exponential	Mean service time
Erlang	Mean service time with standard deviation
Hyper-exponential	Minimum service time
Uniform	Upper and lower service time limits
Random	Upper and lower limits
Normal	Mean service time

External inputs to the system are assigned an interarrival work rate that describes the rate of incoming data to the system. These arrival events can also generate other events with different interarrival rates, as the workload flows through the system.

The system model also needs to define how the system handles contention. If a resource is busy when a new event arrives, the resource can process the new event as a priority interrupt or flag this event for a retry. This system behavior is entirely controlled by the system control and protocol.

Different system protocols together with exception and interrupt handling can determine system throughput just as much as resource service times.

The analytic network queuing model for a single system resource (for example, one of the servers and queues shown in Figure 3-3) describes two events, the arrival of a task request and the completion of a task. The arrival event

- Increments the queue
- Calls a function to generate the next request arrival time
- Calls another function to generate the task completion time.

The completion event decrements the queue and, if the queue is not empty, calls another function to generate the task completion time.

Figure 3-5 shows pseudo-code for an executable model of a single system resource.

Figure 3-5 Executable Model of a Single Server Queue

```
while current_time is less than end_of_simulation_time
   if arrival_time is less than completion_time
      set current_time to arrival_time
      increment queue
      generate next arrival_time
      if queue = 1, generate completion_time
   else
      set current_time to completion_time
      decrement queue
      if queue > 0, generate completion_time
         else set completion_time to end_of_simulation_time
```

The functions used to generate the arrival time and completion time need to reflect the anticipated work load for the system and the anticipated time it takes to complete the task. The more accurate these functions are, the more accurate the performance statistics will be.

Behavioral System Models

Designers use behavioral models to capture the key algorithms in a system. In a communications system, for example, the designers may want to test out several different communications protocols. In a graphics system, the designers would want to focus on the algorithm for building a graphical image from data. The protocol involves a sequence of data and control transactions executed in hardware and software.

Behavioral models are functionally accurate, but timing-independent. While behavioral models should accurately model the functionality of the design, they do not model the distribution of events over clock cycles. For example, in the Verilog code shown in Figure 3-6, a series of three events occurs when the *multiply* signal is asserted. The model does not describe these events in relation to a clock signal; instead, the model relies on the simulator to sequence events so that *operandB* is shifted before it is multiplied by *operandA*.

Figure 3-6 Sample Behavioral Model

```
always @(multiply)
   begin
      temp1 = operandA[7:0];
      temp2 = operandB << 1;
      result = temp1 * temp2;
   end
```

The term *behavioral* is also often applied to RTL models as well; for the purposes of this document, we will consider them to be two distinct levels of abstraction.

RTL Implementation Models

Once the system models are complete and system-level verification has begun, the designers can start to create RTL implementation models for the subsystems in the design. The designers use synthesis and/or datapath generators to generate the gate-level implementation models from the RTL models.

To make the most advantage of the synthesis tools, the RTL models should follow the synthesis modeling style and should use hierarchy. The following sections discuss these topics in more detail.

Synthesis Modeling Style

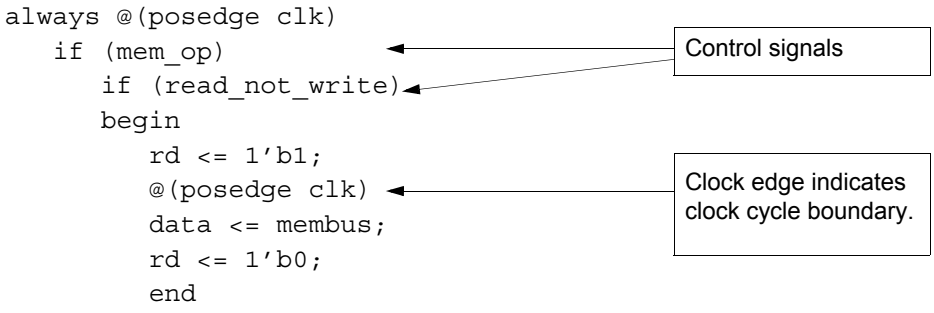
Synthesis tools cannot implement behavioral models because they are timing-independent. For a design to be synthesizable, it must model the behavior of the circuit at each clock cycle.

For example, in a behavioral description, a memory read operation might be represented in Verilog HDL as

```
data <= membus;
```

On the other hand, the RTL description for the same operation has to be more detailed. It should describe the control signals and define the clock cycle boundaries as shown in Figure 3-7.

Figure 3-7 Sample RTL Model



On the other hand, the models intended as input for synthesis tools should not be modeled at too *low* a level. Synthesis tools can implement gate-level HDL descriptions as well as RTL. However, to fully benefit from the flexibility that synthesis tools provide, it is better to model the design at the highest level of abstraction possible.

For example, it is possible to model a carry-look-ahead adder at the equation level:

```
assign
   gen = a & b,
   prop = a | b,

   gg0    =          (             gen[1]) |
                     ( prop[  1] & gen[0]),

   gg1    =          (             gen[3]) |
                     ( prop[  3] & gen[2]),

   c1 =           gen[0] |
                  (prop[   0] & c_in),

   c2 =           gg0 |
                  (&prop[1:0] & c_in),

   c3 =           gen[2] |
                  ( prop[  2] & gg0)|
                  (&prop[2:0] & c_in),
   c4 =           (             gg1) |
                  (&prop[3:2] & gg0) |
                  (&prop[3:0] & c_in),

   carry = {c4,c3,c2,c1,c_in},

   c_out = carry[width],

   c_bund = carry[width-1:0],

   sum = (prop & ~gen) ^ c_bund;
```

The synthesis tools treat these equations as Boolean equations, and therefore cannot build alternative implementations, such as a carry-save adder or ripple-carry adder. A better way to model an addition operations is

```
assign {c_out, sum} = a + b + c_in
```

This approach is the most flexible, in that it does not specify an implementation. The designer can select a different implementation for each synthesis run to determine the most optimum implementation.

Modeling at a high level of abstraction allows the synthesis tools to make high-level implementation trade-offs based on user-defined constraints. This means that given poor constraints, the outcome may not be a desirable one. On the other hand, a properly constrained design yields optimal results for the given technology library.

Modeling at a low level of abstraction produces a more predictable result and the constraints have a lesser effect. However, modeling at a low level of abstraction is much more time-consuming and typically results in less efficient simulation performance.

Hierarchical Designs

Top-down design implies the creation of design hierarchies from the top down. Typically, the design block is partitioned and each of the subblocks are further broken down until a set of lower-level modules and their interconnects have been defined.

One reason for partitioning the design is that synthesis tools perform best on *design chunks* of 5000 gates or less. However, there are other reasons for using hierarchy in a design:

- Clock domains

  Synthesis tools can handle multiple clocks, as long as the clocks have the same period. If the design has clocks with different periods, then the design needs to be partitioned by clock domain so that the logic in the different domains can be synthesized and optimized separately.

- DFT requirements

  If parts of the design use different types of storage devices (edge-sensitive or level-sensitive) or different clocking schemes (single-phase or two-phase clocks), then the design needs to be partitioned so that the test logic insertion can be performed properly on the different parts of the design.

- Datapath vs. control logic

  Separating the datapath logic from the control logic facilitates the sharing of resources for complex operations and the use of datapath generators.

- Resource sharing

  As much as possible, the design should be partitioned so that shared logic is not duplicated in separate parts of the design, as shown in Figure 3-8 on page 4-17.

- Critical path optimization

  Design partitioning should accommodate critical path consolidation and isolation. When possible, inputs and outputs should be registered. This helps in defining timing budgets and creating input arrival and output required times.

  For example, in Figure 3-9 on page 4-17 the output required times for module A and input arrival times for module B are effectively 0. Likewise, the output required times for module C and the input arrival time for module D are effectively the clock period (less the setup time).

  Also, design partitioning should not create critical paths that span multiple preserved modules, as shown in Figure 3-10 on page 4-18.

Figure 3-8 Resource Sharing

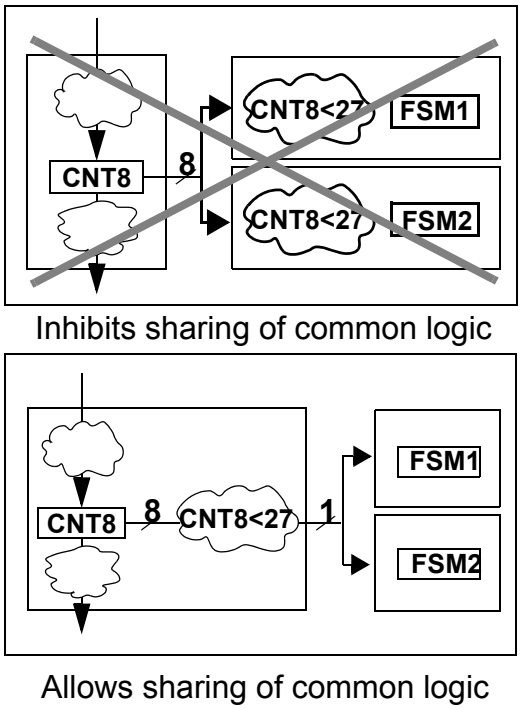


Figure 3-9 Design Partitioning

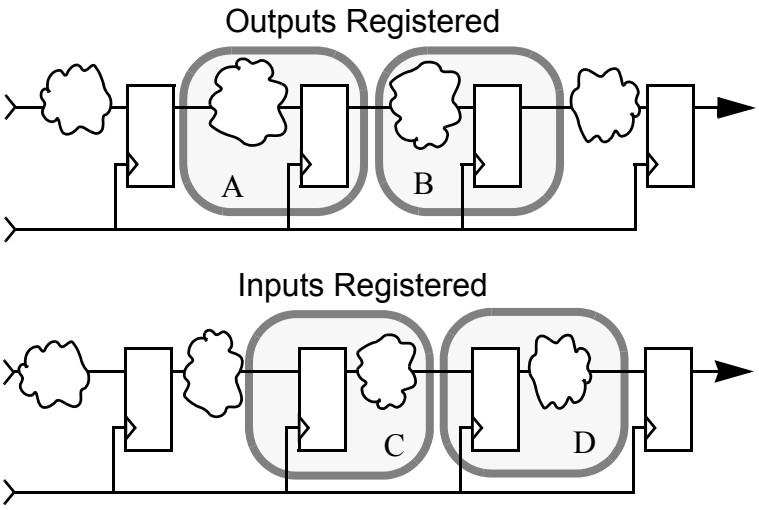
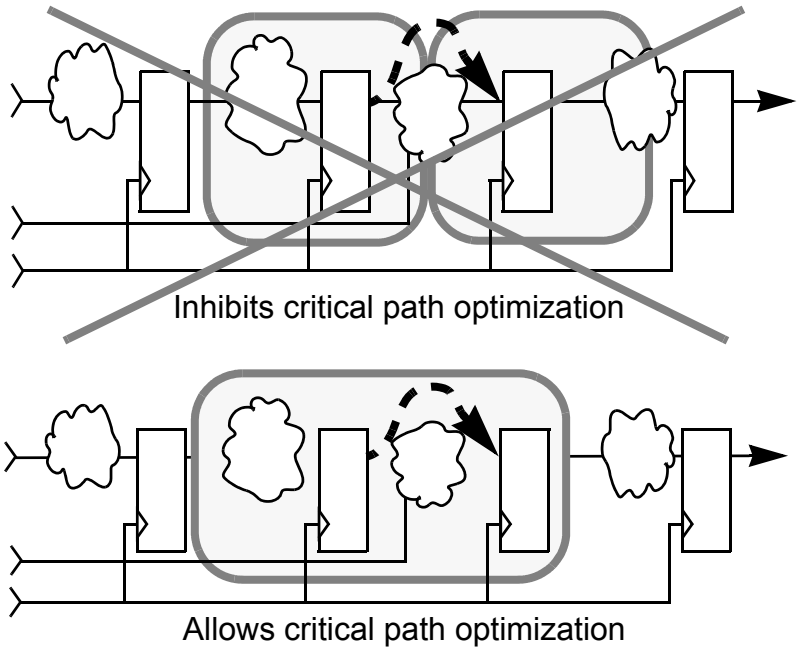


Figure 3-10 Critical Path Optimization



Structural Models

Below is an example of a structural model of a NAND tree for process validation testing.

Figure 3-11 Structural NAND Tree Model

```
module nandtree (nandin, nanout);
   input nandin;
   output nandout;

   nand2 i0(nandio, nandin);
   nand2 i1 (nandi1, nandio);
      .
      .
      .
   nand2 i800 (nandout, nand799);

endmodule
```

Datapath Design

Previously, with system level designs, the datapath portion was generally confined to a dedicated, high performance chip function (DSP, uP, e.g). The nature of these highly computational functions generally dictated a full custom design methodology. Since these typically separate, highly data computational chips were designed in a separate environment from the ASIC portions of the system, many iterations on the ASIC or IP portion of the system design could occur fairly independently of the custom datapath portions. Thus, the datapath IC design may or may not have been the “bottle-neck” for scheduling the finished system-level design. However, as we can see now with the physical possibilities opening up to allow for system-level integration on a chip, datapaths are becoming common requirement of many more types of IC applications. With the continuing trend of designing higher performance, compute-intensive chips, as well as the increased level of system -integration on a chip; the appearance of datapaths in chip design is becoming more and more common.

The most effective method of maximizing a datapath’s performance is to effectively manage where and when the appropriate datapath-functions are executed. It is at the architectural level that the designer has the most versatility and control over the performance of the design. Examples of theses architectural decisions might be determining the number of pipelines necessary to meet the performance specifications or whether a datapath function should actually be split into multiple datapaths in order to meet the performance specifications. Unfortunately, with custom datapath design, the distribution of the designer’s time spent on architectural partitioning and actual IC design versus the physical design implementation - which is the least leveragable phase - is quite disproportionate. Of the total design cycle from concept to layout, a typical time-allotment per design-phase is 10 per-cent for architecture; 10 per-cent for IC design and 80 percent for physical layout and verification.

The reasons behind this disproportion vary; but the perception that the forging ahead with the design implementation before verifying its feasibility will still yield usable results is a common one. Historically, the designer has been forced into this scenario due to insufficient analysis tools at the architectural level. “Fixes” during the physical design phases were common and could still offer controllability over the end-results. However, with the fabrication processes in the DSM range, the effects of altering a physical design are no longer intuitive as what was once second-order-effects are now dominant effects. For example creating another datapath region to process some data in parallel may turn out to be in appropriate due to the global inter-connect delays - which in larger processes would have been negligible. With back-end, pitfalls such as

these, front-end architectural tools which can accurately predict these physical effects are crucial to producing a design which will converge to its specifications.

Design Capture Technologies

Design capture technologies can be divided into the following categories:

- Programming languages
  - C or Perl

    For system models, where the main goal is to verify the performance and the algorithms, the Perl and C programming languages are the modeling language of choice. Designs modeled in C or Perl execute faster than those modeled in HDLs, and thus facilitate design experimentation.
  - HDLs

    For RTL implementation models, where the goal is to model the behavior of an electronic circuit at every clock cycle, a more specialized programming language is required—a Hardware Description Language (HDL). Verilog and VLSI Hardware Description Language (VHDL) are comparable in modeling power, although VHDL is the U.S. government standard.
- High-level design tools
  - Block diagram editors

    These are graphic editors that allow designers to visualize the overall architecture of the system or subsystem. The designer can attach functional or algorithmic descriptions to the subsystems or subblocks in the block diagram.
  - Language-sensitive editors

    These text editors are specially developed to aid in creating models using a particular programming language. The editor "understands" the correct syntax and semantics of the language, and can prompt or highlight errors for the designer.
  - Visualization tools

    High-level, application-specific visualization tools are becoming indispensable in analyzing designs.
  - HDL generator

    An HDL generator is a tool that can generate an HDL description from a system-level model.

- Libraries
  - Application-specific libraries

    These libraries facilitate high-level system design similar to the way that gate-level libraries facilitate block-level implementation. High-level systems design libraries are by nature application-specific. A useful library for a designer creating a system model of a network communications system, for example, would probably contain parameterized models that facilitate building token ring or ethernet LANs.
  - Macro libraries

    These libraries are used with synthesis tools to facilitate the implementation of datapath logic using datapath generators.

References

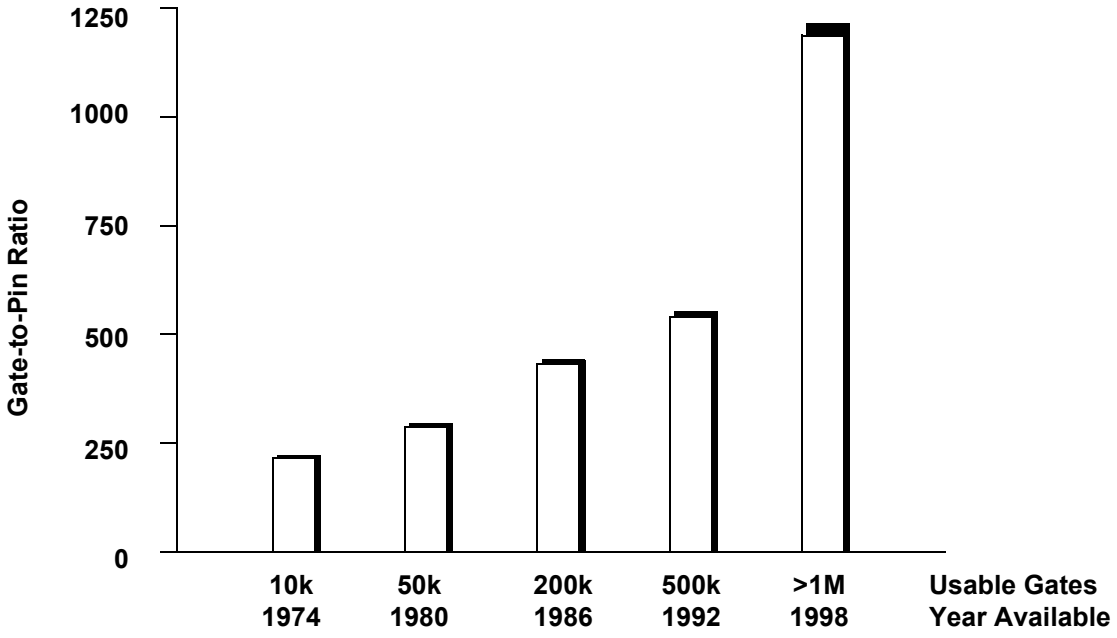
4

Design for Test Methodology

Design for Test: the Challenge

Because of the increasing complexity and density of today's designs, the number of logic gates in VLSI components is increasing at a much faster rate than the number of I/O pins on the device. This trend is shown in Figure 4-1 as the gate-to-pin ratio, which has increased significantly over the past two decades. This high gate-to-pin ratio makes the device's logic less accessible (less controllable and less observable) to the test engineer, thereby creating a formidable test challenge.

Figure 4-1 Increasing Gate-to-Pin Ratios



There are similar test issues at the board and system level. Complex device packaging, such as Surface Mount Technology (SMT) and Ball Grid Array (BGA), along with growing acceptance of multi-chip module (MCM) technology, has resulted in a higher density of board-level interconnects. Physical access to board interconnects for probing is either difficult or non-existent, making the traditional bed-of-nails board test not feasible.

These challenges have created a need for Design For Testability, or DFT, methodology. DFT is the discipline of modifying, or enhancing, the logical design of a chip or system in order to facilitate testing and debug of the design during prototype verification and production manufacturing phases of the design cycle.

Goals of a DFT Methodology

The primary goal for a DFT strategy is to impact the overall business objectives of the product by shortening the product time-to-market and reducing the overall cost of test. A set of objectives in achieving this goal are to

- Facilitate the back-end test process

  DFT allows manufacturing tests to be developed much more easily and efficiently with fewer resources. Proper design for testability can also decrease the actual execution time of production tests, which in turn reduces the labor costs associated with manufacturing the product.

- Improve the quality of the manufacturing tests

  DFT techniques, such as delay testing and I_{ddq} tests extend fault coverage beyond the traditional stuck-at fault model. DFT logic verification and test vector verification also help make production-ready tests available earlier on in the manufacturing cycle.

- Facilitate the front-end design process and the development of related products

  Investing time up-front in the design cycle to develop a thorough DFT strategy can save time during the early phases of development as well as during prototype testing and manufacturing. DFT has proven useful during design verification by providing access to chip and system states via scan paths. This access would otherwise be unavailable to the designer.

  A good DFT strategy also facilitates the production of related products. The time spent in developing a DFT strategy for the initial product can be saved when it is used to develop and test similar products.

- Improve testability at all levels of integration

  A good DFT strategy addresses testing at all levels, including component, board and system levels.

In order to define a DFT strategy that meets these goals, the test engineer needs to have a thorough understanding of the techniques available and a set of guidelines for choosing the appropriate techniques for a particular project. The following sections briefly describe a set of structured DFT techniques and a general set of DFT guidelines.

Structured DFT Techniques

There are various structured DFT techniques that can be used in support of test for sub-micron designs. Because these techniques are designed for particular types of components or logic, it is best to develop a block-based approach to DFT, where different test techniques are selected for different blocks. The standard test techniques for digital logic include

- Internal scan

  This technique is appropriate for blocks of sequential logic, where the system flip-flops or latches can be used in a test mode.

- Boundary scan

  This technique implements scan testing at the board or system level by adding scan registers and other test logic at each I/O pin of each component.

- Test access collar

  This technique improves the testability of embedded blocks of logic, particularly blocks that have been fully laid out.

- Built-in self-test (BIST)

  This technique includes a device for generating test patterns and another for reducing the results to a single no/nogo signature. This technique is appropriate for large memory structures, where scan is not practical.

The design for test techniques discussed so far are generally targeted at detection of stuck-at faults. The stuck-at fault model is intended to abstract the effect of physical manufacturing defects that cause a node of the circuit to be shorted to (or stuck-at) a logic 1 or a logic 0.

Though this fault model is generally well excepted in the industry, and studies have show it to be effective in modeling a majority of IC manufacturing defects, research has also shown that coverage of faults in addition to stuck-at faults will result in a higher quality test. Some common IC manufacturing defects, such as bridging faults, gate-oxide holes/shorts, defects that cause degraded voltages, and defects that cause timing inaccuracies are not generally detected with tests targeted at stuck-at fault detection.

The following test methods provide test capabilities beyond stuck-at fault detection:

- I_{ddq} testing

  This technique is based on measuring the power supply current when the circuit is in a static, or *quiescent* state. This technique is useful in extremely dense circuits.

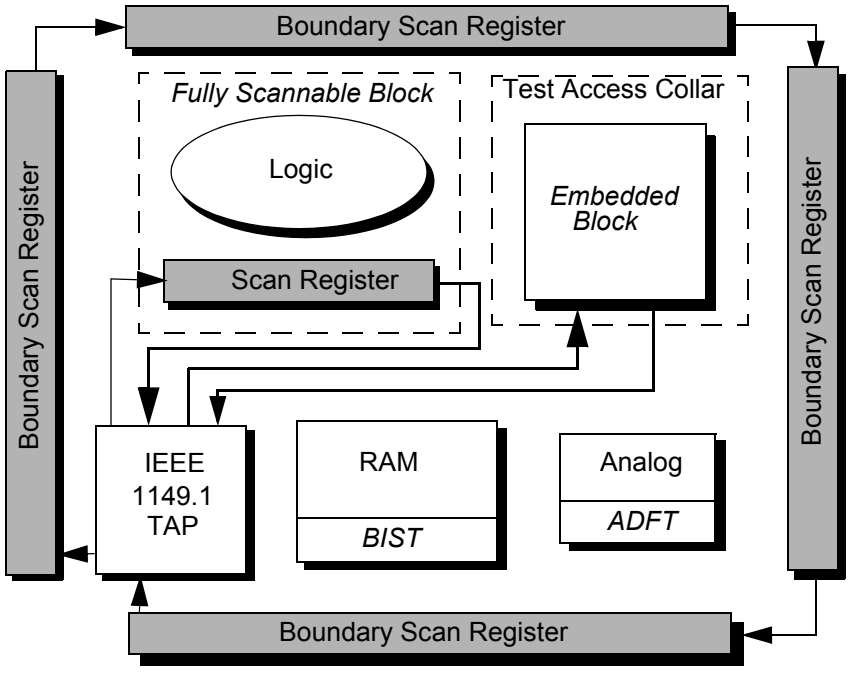
- Delay testing

  Certain manufacturing defects can cause faults that are only observable when the circuit is test *at speed*. This technique is useful in circuits where timing is critical.

  The RAM BIST approach, which usually models stuck-at faults, can provide at-speed testing of the RAM block, if it is implemented using the system clock instead of a test clock.

Figure 4-2 illustrates a block-based approach to providing DFT. The DFT techniques listed above are described in more detail in the following sections.

Figure 4-2 Device Level DFT Techniques

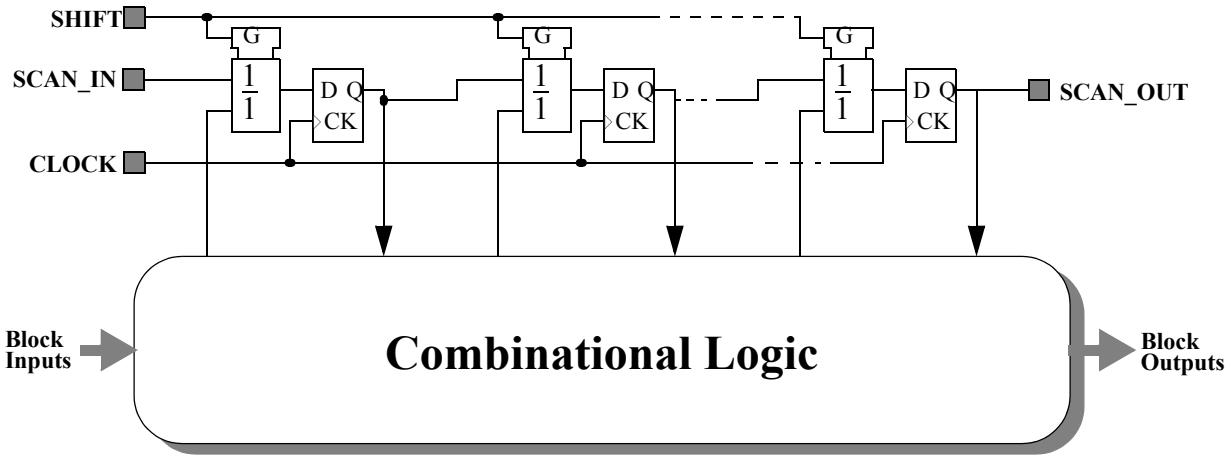


Internal Scan

To implement internal scan, all the flip-flops or latches of the design are connected to form a *scan chain* or *scan path.* A multiplexor is added at the input to each storage device so that a test mode control signal can select either the system data or the scan data as input.

The scan chain thus operates as a shift register when the test mode signal is asserted. Test data can be readily shifted or *scanned* into and out of the scan chain. The basic scan path technique known as multiplexed D flip-flop (DFF) is shown in Figure 4-3 below.

Figure 4-3 Multiplexed DFF Internal Scan



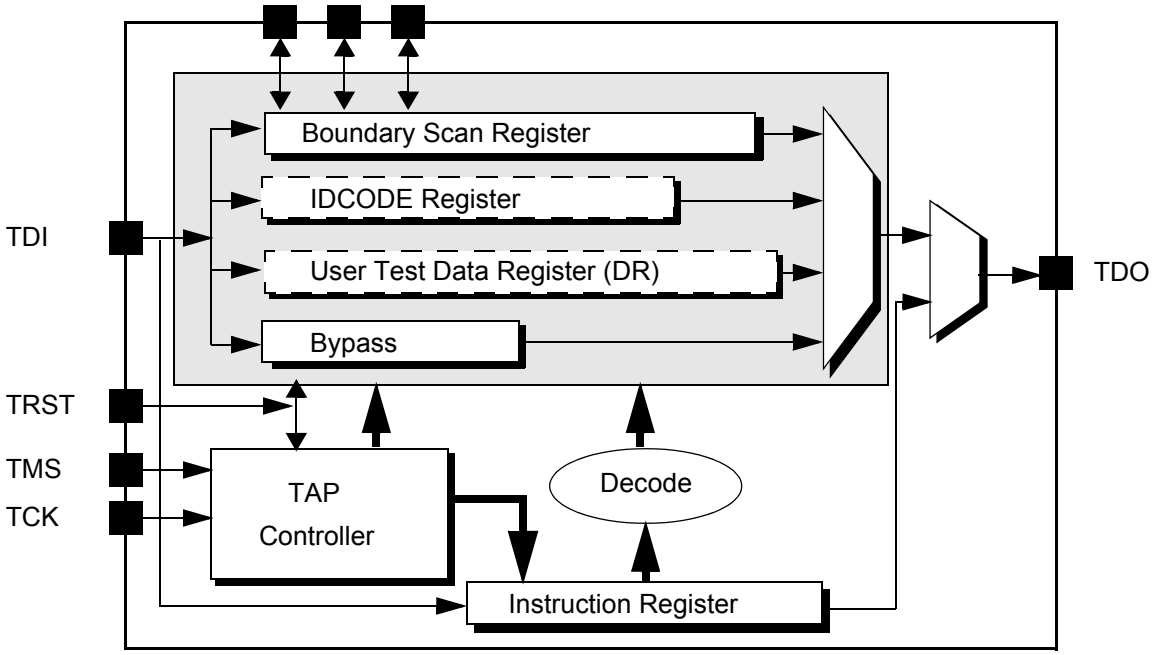
Internal scan allows full access to the register elements in the block or chip under test. The net effect of implementing full internal scan is to reduce the task of testing a complex sequential circuit to the problem of testing a large combinational circuit. This technique results in an easily solvable test problem for Automatic Test Pattern Generator (ATPG) tools and dramatically reduces the test vector development time in the design cycle.

Boundary Scan

Boundary scan is an extension of the internal scan path approach, intended to support board-level and system-level interconnect testing.

Figure 4-4 shows how the IEEE 1149.1 boundary scan architecture is implemented in a subsystem. The IEEE 1149.1 standard, an outgrowth of the work done by the Joint Test Action Group, JTAG, defines a mandatory architecture for controlling access to the boundary scan features in an IC component.

Figure 4-4 Device Level 1149.1 Architecture



The IEEE 1149.1 standard boundary scan includes the following mandatory elements:

- A dedicated scan register at each I/O pin site in order to control and observe logic values directly at the component pins
- A bypass register
- A TAP controller

  The TAP controller is a 16-state finite state machine, whose states and state transitions are defined by the 1149.1 standard.
- A four-pin Test Access Port (TAP), including
  - The scan input and output pins, TDI and TDO, for serial access to scan registers connected through the TAP controller.
  - The test clock, TCK, clocks the TAP controller.
  - The test mode select signal, TMS, controls the state transitions of the TAP controller.
  - An optional TAP reset signal, TRST, may be used to asynchronously reset the TAP controller.

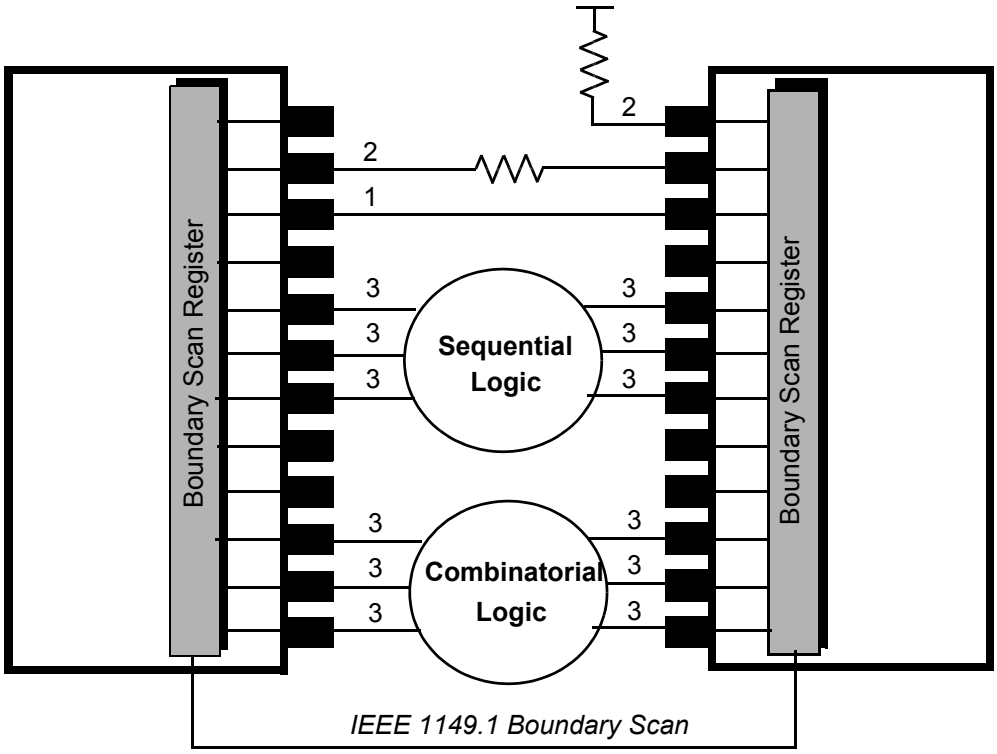
  These TAP pins are dedicated for implementing the TAP protocol and may not be shared by the system functions of the device.

Certain behavioral characteristics of these elements are also required by the standard, such as the TAP protocol and the test behavior of the boundary scan register.

Also shown in Figure 4-4 is the standard's provision for optional test data registers (User Test Data Register), which can be the internal scan register(s) of the subsystem. The IEEE publication *Standard Test Access Port and Boundary Scan Architecture, 1149.1a-1993* has further details on the 1149.1 standard.

Figure 4-5 shows how boundary scan is implemented at the board-level. Studies have shown that, for surface mount boards, 76% to 95% of manufacturing faults are caused by solder opens or shorts. Of course, to maximize the benefits of boundary scan, it must be optimally designed into the product.

Figure 4-5 Board-Level 1149.1 Architecture



(1) Testable by boundary scan ATPG
(2) Potentially testable by boundary scan ATPG
(3) Testable via boundary scan tests

There are several factors that determine the required amount of boundary scan designed into a product:

- Test equipment and test development costs
- Test execution time
- Fault isolation requirements
- Lack of physical access

The last two reasons are the most compelling reasons for boundary scan. In any case, if boundary scan is implemented in a product it is essential that planning be done at the system level because it primarily benefits system test. Since most systems do not contain 100% boundary scan IC's due to availability or cost, intelligent planning can make use of partial boundary scan to test or assist in testing significant portions of non-boundary scan logic. [Wayne Daniel, *Integrated System Design*, September 1995]

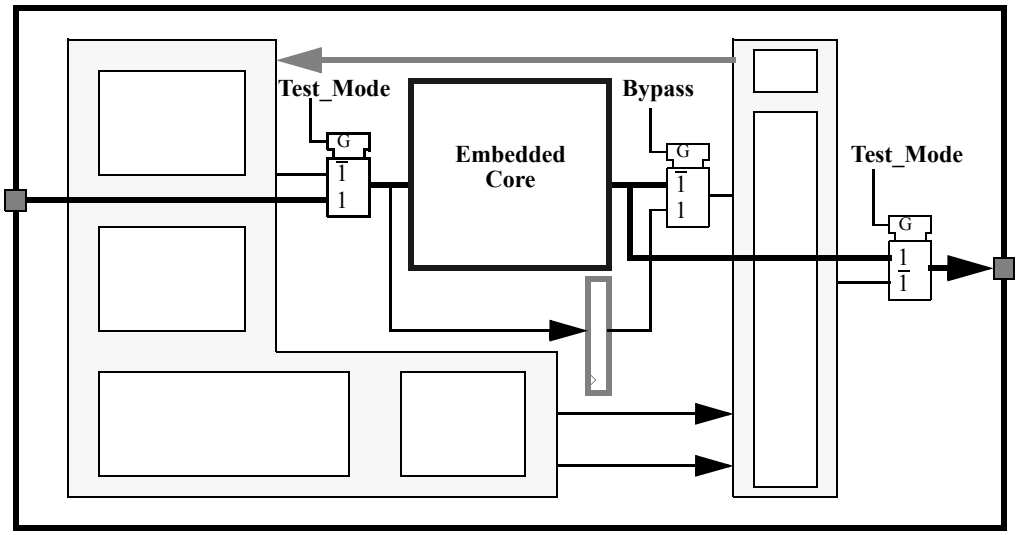
Test Access Collar

Embedded blocks are blocks of the design that have been implemented in the form of a macro or megacell, such as a CPU core or a standard bus interface, for example, a PCI bus macro. These embedded blocks often create a test problem either because they are large and complex, or because designers do not have access to the source description for the block if the block has been imported into the design as a purely physical block.

It is difficult to control the testability of an embedded block, and it may be impossible to generate high quality tests for the block using ATPG tools. For embedded blocks, it is better to have an existing set of test patterns of known high quality and to test the blocks in a stand-alone fashion.

Figure 4-6 shows an embedded block and a method for providing testability around the block. This method enables test patterns to be applied separately either to the core, or to the remaining chip logic.

Figure 4-6 Embedded Block with Multiplexed I/O



In this example, there is a test mode that multiplexes the normal interface signals of the embedded core out to the chips' I/O pins in order to provide direct access to the core block for test. A bypass path around the embedded block is also provided for testing the remaining chip logic independent of the core block. An alternative method is to implement a boundary scan collar around the embedded block. This would then provide the necessary access and isolation of the core block from the rest of the chip for test purposes.

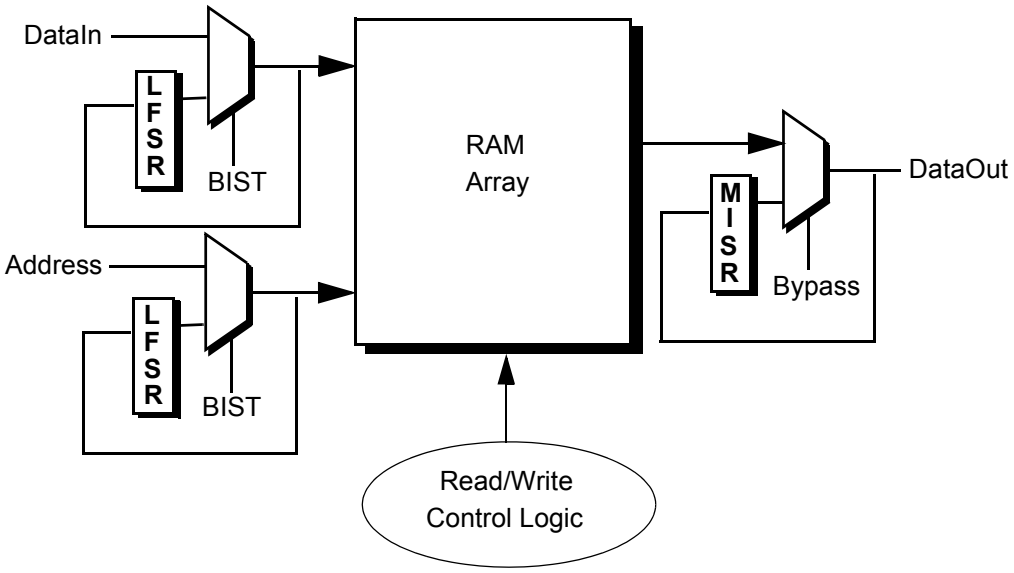
Built-In Self Test for RAMs

Memory structures larger than register files or FIFOs are usually implemented as a RAM array or a latch-based structure. In this case, internal scan is far too costly; a more viable solution is a Built-In Self-Test (BIST).

Figure 4-7 shows as an example an embedded RAM block and the BIST logic added to test it using a pseudorandom pattern algorithm. The RAM BIST implementation requires that multiplexors be added to the inputs and the outputs of the RAM and that dedicated BIST registers be connected through the multiplexors.

The data and address inputs use Linear Feedback Shift Registers (LFSRs) for pseudorandom pattern generation, and the RAM data output uses a Multiple Input Signature Register (MISR) to compact the results of the RAM BIST responses into a go/nogo signature. The BIST register-mux structures are shown as separate structures but may actually be implemented with the functional system registers.

Figure 4-7 RAM Array with BIST



In addition to supporting test of the RAM itself, this RAM BIST approach also provides for the reuse of the BIST registers as added observability and controllability point at the inputs and outputs of the RAM block. At the input an observe path is added, shown as the connection from the output of the mux to the input of the LFSR. At the output a control point is added, shown as the connection from the MISR to the input of the mux. These features provide enhanced ATPG testability of the core logic surrounding the embedded RAM. The result is a self-contained and autonomously testable memory.

Also, BIST results in a methodology that can be easily reused. The use of macro libraries further simplifies this process and fosters standardization of BIST test methodologies.

For large embedded RAM structures the use of memory BIST has little or no impact on the cost. It also minimizes the cost of manufacturing production test by reducing the test pattern overhead required to test the RAM. Thus, the RAM BIST approach can be justified by its cost savings.

I_{ddq} Testing

I_{ddq} testing is a test method based on measuring the power supply current, I_{ddq}, drawn by a CMOS integrated circuit in its quiescent, non-switching, state. This technique takes advantage of the fact that CMOS technology draws very minimal current, typically in the nanoampere range, for a defect free circuit in its quiescent state.

I_{ddq}testing differs from stuck-at fault testing in that it is a current test, not voltage test. Stuck-at tests require that faults be propagated to an I/O pin of the device. However, I_{ddq} faults can be observed at the power supply pins of the chip. Even in cases where the voltage is still within the range for the expected logic level, there is still an abnormal current draw which exceeds the normal quiescent I_{ddq} current, and the defect can be detected.

A single I_{ddq} test vector can detect a large number of faults, and from a test-generation perspective, I_{ddq} vectors are quite efficient. However, several I_{ddq} test vectors, typically in the tens to hundreds of vectors, are still required to achieve high I_{ddq} coverage. One disadvantage of I_{ddq} testing is that I_{ddq} test application rates can be very slow, due both to the long settling time required for the switching current to settle and to the speed of the measurement electronics that are typically used.

The I_{ddq} technique is also not very useful for fault diagnosis and isolation, because it is difficult to determine the specific cause of the failure by means of the current measurement, and because so many possible I_{ddq} faults are detectable by a single test vector. It is important to note that I_{ddq} testing is not intended to replace high quality stuck-at test vectors, but it has been widely acknowledged by the test community as one of the most cost effective means of improving test quality beyond traditional stuck-at fault testing.

I_{ddq} tests are performed as follows:

1. **Apply a test vector which puts the circuit into a known static state.**

2. **Wait for a short period of time, usually some number of milliseconds, for the switching currents to settle out.**

3. **Measure the I_{ddq} power supply current.**

In a defective circuit, the measured I_{ddq} current is typically in the microampere to milliampere range.

As an example, consider a bridging defect, where the output transistors of two logic gates are shorted together. If these two gate outputs are driven to opposite logic values after the switching currents have settled out, then the I_{ddq} measurement would detect a high current draw, indicating that the circuit has a defect.

In order for a subsystem to be I_{ddq} testable, certain design requirements must be met. In general, I_{ddq} testing requires that the design be fully static and that current-draining states be avoided, since such states draw an I_{ddq} current in excess of that expected in the normal quiescent state.

To avoid switching currents during I_{ddq} testing, all clocks are stopped before the current measurement is taken. A fully static design maintains state even when the clocks have stopped. Dynamic logic should be avoided, since in the absence of a clock the dynamic nodes lose charge and may cause some transistors to partially turn-on, creating a current draw.

The following are a general set of I_{ddq} testability rules and guidelines.

- Disable pullups and pulldowns

  Any pullup or pulldown devices in the chip cause current flows when they are driven to the opposite state. A common solution is to add an I_{ddq} test mode signal at the pullup or pulldown device, so that when the test mode is asserted, the device is disabled and a static current measurement can be made.

- Avoid floating nodes and degraded voltages

  States that cause floating internal busses and logic gates that have floating inputs must be avoided. Floating nodes can cause excessive current draw. A degraded voltage driving a FET can partially turn-on the transistor and cause an excessive current flow.

- Avoid drive contention

  There should be no drive contention on internal tristate busses, other internal nodes, or between the tester and the I/O pins of the chip. If any circuit node is driven to opposite states by different drive sources there is an excessive current draw.

- Partitioning of power busses and supply pins

  Digital and analog portions of the design should have separate power supply pins, so that the digital portion can be tested with I_{ddq} vectors. Also, in some very large designs where the process is known to have high leakage currents, partitioning the design into several sections with separate power busses allows I_{ddq} testing to be done on each of the separate partitions independently, where quiescent currents are much smaller.

- Disable devices that have static power dissipation

  Some circuits, such as embedded RAM structures, will dissipate static power in certain states. In this case there should be an I_{ddq} test mode to disable any FETs that are always on in normal operation and would cause static power dissipation in some states.

Delay Fault Testing

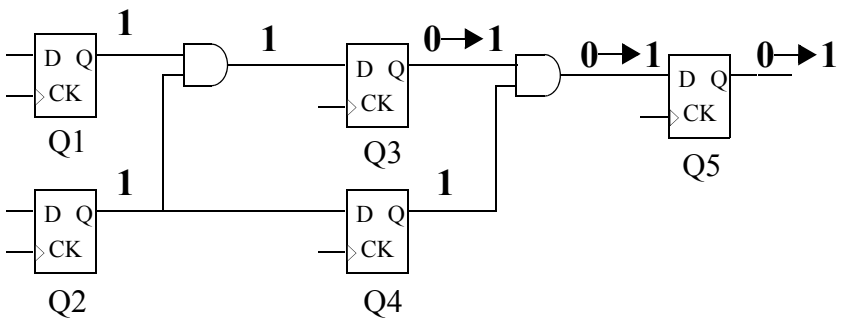
Delay faults, or timing faults that appear when the circuit is run at speed, are very difficult to detect, since in many cases both the underlining defect and its effect on the behavior of a component can only be seen as a parameter being out of compliance with specifications. Delay fault testing is used in conjunction with other test techniques, such as BIST or scan.

For example, if a manufacturing defect causes the resistivity of a metal (or poly) line to be slightly larger than optimum, it may limit the maximum operational frequency of an IC component by increasing certain RC delay values. The final result is excessive internal path-delays.

Such timing faults can only be detected by defining a well-planned signal transition at a pre-determined time and observing whether all signals reach their final, stable values by that time. Timing accuracy in generating both the initial signal transition as well as in observing the final output value is of paramount importance. Furthermore, internal delays along the system clock distribution tree for a subsystem may contribute to the timing problem and must be also considered during timing tests. Thus, both the generation of the initial signal transition and the capturing of the subsystem's response to that transition must be achieved using the functional clock, so that internal clock-tree delays are accounted for in the timing test.

Figure 4-8 shows an example of a delay path test, where the timing path from DFF Q3 to Q5 is to be tested and measured. In this approach simulation or other analysis is performed to determine the state *prior* to the state where the initial signal transition is generated. Then the component is initialized to that state, using scan, for example, and then a system clock advances the internal state and generates the desired signal transition. Next, a second functional clock pulse is applied after a carefully selected time interval to capture the component's response. Results are then scanned out to determine pass or failure.

Figure 4-8 Delay Path Testing



In the example of Figure 4-8, DFF Q3 transitions from a logic 0 to a logic 1 on the first clock edge of the test, and DFF Q5 captures this transition some time later with the second clock edge of the delay path test. The time between the two clock edges determines the delay of the path, and by varying the separation between the first and second clock edges to determine the exact pass/fail point of the path, a direct measure of the delay time along the path can be accurately determined.

The difficulty in using this approach is performing the necessary circuit analysis to determine the desired prior state. In some scan designs it is possibly to utilize the last scan clock edge applied when shifting in the delay path test vector to set up the transition before capturing it with a functional clock. There are also designs for specialized scan DFFs that can load two independent patterns to facilitate delay path testing.

DFT Rules and Guidelines

There are very few universal DFT rules that apply in all design environments. However, several general rules and guidelines have been proven to be beneficial in solving and preventing testability problems in many complex digital (synchronous) systems. A brief overview of these general DFT design rules and guidelines are given below.

- Implement full internal scan

  Implementing internal scan provides controllability and observability of internal chip states and allows for Automatic Test Pattern Generation (ATPG) to easily obtain full test coverage. Payback of this approach is greatest when all internal storage elements are scannable.

- Provide for test access around embedded blocks

  Since it is not practical to make the states of the embedded block scannable, it is necessary to provide for testing of the embedded block itself, and to support testing of the logic around it.

- Use IEEE 1149.1 Boundary Scan and its associated Test Access Port (TAP)

  This allows for board-level interconnect testing using a standard protocol and enables commercially available design automation tools to be used. Access to all on-chip test functions should be provided through the TAP. Using a common interface and protocol to all test functions unifies the formatting of various test suites.

- Use dedicated functional clock signal pins

  Functional clock signals should not be gated with any other signals and must be directly controllable from the component pins. This requirement allows a simple and straight-forward mechanism for applying a functional clock to the system in order to capture its response to a test vector.

- Use dedicated test clock signal pins

  Separation of test clocks from normal functional clocks in a synchronous system environment means that test actions can be performed without affecting the overall system state. For some types of internal scan, the system clock is also used to shift the scan register. In these cases the test clock and system clock pins may be separate, but require multiplexing internal to the chip.

- Avoid asynchronous logic, combinational loops, and uncontrollable latches

  Such designs are difficult to initialize to a desired state and keep stable in that state while other signals around them might be changing. This may prevent proper loading of test vectors into a component or the ability to observe the results in an ambiguous manner.

- The drive state of internal tristate busses must be uniquely decoded

  Test vectors can potentially cause random assignments to the drive state/enables of the bus, so this requirement assures that during testing there will be no drive contention on the internal tristate bus.

- A pre-defined reset state must be identified and a straight-forward mechanism must be implemented to reset the system

  Often, the best solution is to use a dedicated reset pin for this purpose.

- Provide test mode control of the inactive state (no-drive) for bidirectional and tristate chip I/O pins

  The ability to turn off the I/O drive at certain times during chip and board test can help to assure that the chip is not damaged during testing. Implementation of the optional IEEE 1149.1 HIGHZ and CLAMP instructions is recommended.

- Provide for support and testing of fault coverages in addition to stuck-at faults

  For example I_{ddq} tests, at-speed or delay tests, and test structures for monitoring the IC fabrication process. It has been shown that coverage of faults in addition to stuck-at faults can improve the quality of test (i.e., reduce the number of test escapes which are passed on to the next test step in the manufacturing process).

In most cases, when a particular rule can not be followed, there are alternate solutions that still provide the required testability. For example, if asynchronous logic must be used, confine its use to the set or reset of scannable flip-flops. Then provide for the set/reset to be controllable from a primary input or static test mode signal so that the asynchronous logic can be disabled for testing. A test mode can also be used in other cases, for example to “un-gate” gated clocks or to control latches so that they are kept transparent for test purposes.

References

6

Design Verification Methodology

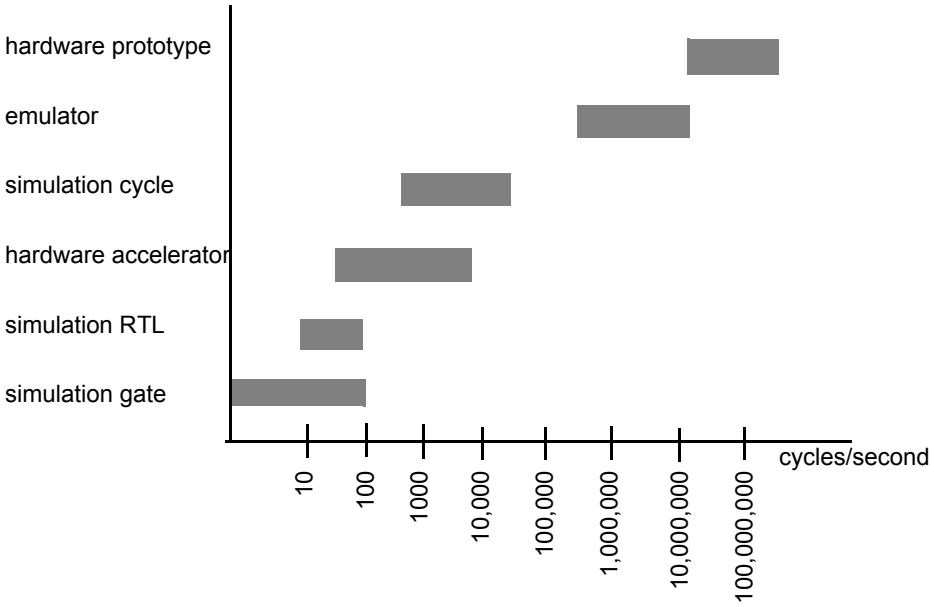
Design Verification: the Challenge

Design verification integrates the phases of the design process—high-level system design, block-level implementation, and chip integration—by verifying that the design continues to comply to all system requirements and has been correctly translated from a higher to a lower level of abstraction at each phase.

In today's product development, the task of verification of large systems determines the product development schedule and constitutes the largest resource demand on hardware engineers. Given the pervasive need for design verification at all phases of the design process, design verification methods need to be made highly coordinated in order to eliminate any possibility of complex system design errors getting missed. In addition, an efficient strategy that reduces the amount of time spent in design verification can have a significant effect on the overall project schedule.

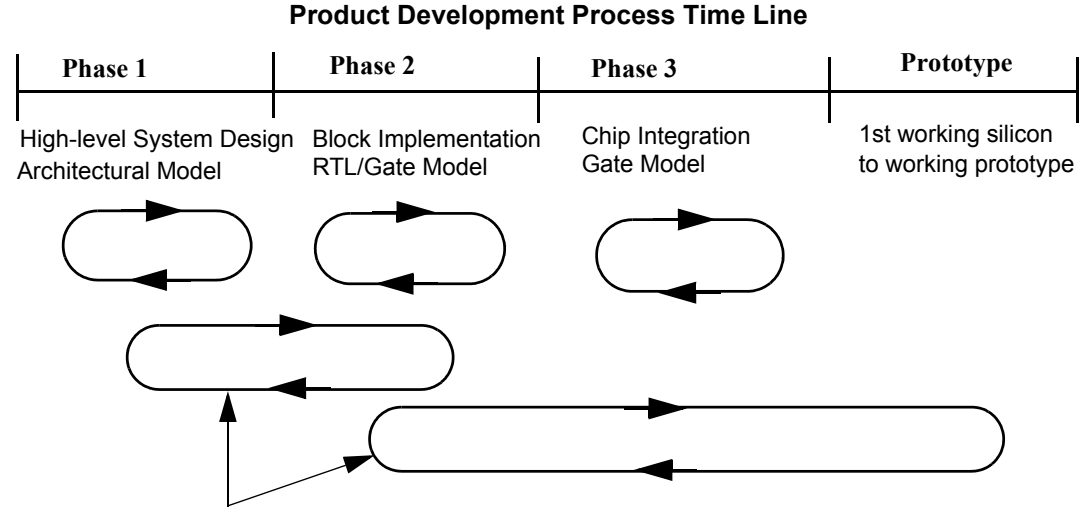
Verification can be a time-consuming process. While event-driven simulation of RTL and gate-level models is the most common verification technology in use in top-down design today, it is also the slowest, as shown in Figure 5-1. It can take 100s to 1000s of simulation hours to execute short sequences of actual system operation in an RTL model of a system, compared to a hardware prototype of the same system which executes at a rate of 100 million cycles per second.

Figure 5-1 Relative Speed of Various Verification Technologies



However, choosing the fastest technology available does not necessarily result in shorter design cycles. As shown in Figure 5-1, because the fastest technologies—emulators and hardware prototypes—cannot be used until after the design has been implemented, complex systems errors that surface during emulation or in the hardware prototype are the most costly to fix, potentially requiring the team to redesign portions of the system and invalidating months of work. While design iterations within a design phase such as block-implementation are unavoidable, design iterations that require the team to go back to a previous phase may require all other work to stop until the redesign is complete and verified.

Figure 5-1 Design Iterations Impact on Project Schedules



Large-scale redesign incurs a greater cost than iterations confined to a single phase.

The key to efficient design verification lies in a carefully thought-out design strategy, not in technology alone.

Design Verification Goals

The challenges posed by design verification cannot be met by technology alone. To reduce time spent in verification and to avoid time wasted by costly design iterations, a thorough, project-specific verification strategy is essential. However, there are two high-level goals that are common to all good verification strategies:

- Validate system intent before starting implementation

  Analyzing system performance and verifying the algorithms of key subsystems or components early in the design process is crucial to avoiding back-end design iterations.

- Verify implementation at both RTL and gate-level

  In general, it is better to reduce the amount of time spent simulating the entire system design at the lowest level of abstraction. Verifying the functionality of the RTL design limits the verification tasks at the gate-level to removing timing and design rule violations.

  It is also better to reduce the amount of time spent in interactive simulation. Verifying the functionality of the RTL design also means that debugging, which often requires interactive simulation, can be completed relatively quickly. Gate-level simulation can be performed mostly in batch.

These goals are discussed in the following sections.

Validating System Intent

The goal for high-level system design verification is to validate the design concept —*system intent*—before design implementation starts. This validation involves

- Analyzing system performance to verify that the design meets all performance and cost requirements
- Verifying system functionality to confirm that the algorithms in the design meet the system requirements
- Verifying that the packaging and the partitioning of the system into hardware and software, ASICs and off-chip, is optimum

It is common for the design verification team to start developing the Verification Test Plan (VTP) to validate system intent soon after the architecture development starts. The verification test plan will define tests which represents the key features and performance factors of the overall product. The verification test plan should define

- Architecture Verification Tests (AVTs)
- Performance Verification Tests (PVTs)

All requirements for the verification process are planned inlcuding the tests which need to be developed, methods and technologies for verifying, the environment and procedures, and the schedule and plan for implementing and performing the verification task. Verification strategies will be discussed in this chapter.

Analyzing System Performance

In most complex systems there is likely to be contention for system resources. Bus contention, for example, is a common problem in any system where more than one processor accesses common memory banks over a common bus. Resource contention also occurs in communications systems, where multiple devices transmit and receive data through a common switch.

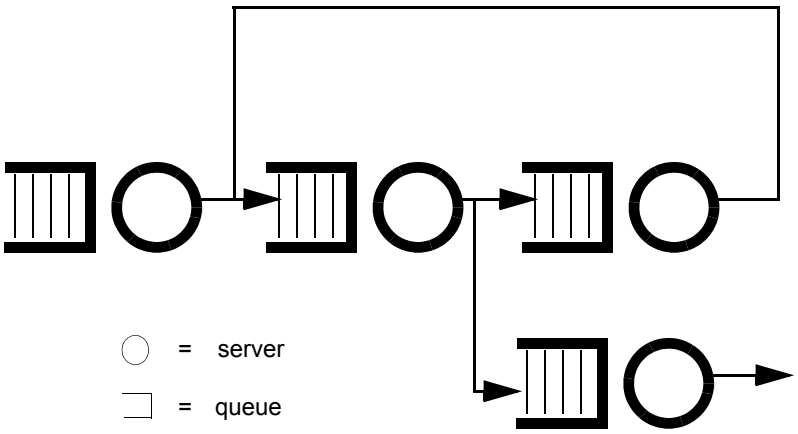
In order to measure system performance it is necessary to determine what performance criteria to measure for. These criteria should be consistent with the system specifications. Then the design team needs to build an analytical model that represents the aspects of the system that need to be measured. Using analytic models, designers can generate system statistics, including

- Throughput—the number of completed tasks/total time
- Server utilization—server busy time/total time
- Average processing time for each request

By linking the analytic models for the whole system together, designers can locate and adjust bottlenecks to optimize the flow of data through the system.

Tests which test system performance are commonly referred to as Performance Verification Tests (PVTs).

Figure 5-1 Analytic Network Queuing Diagram



Verifying System Functionality

Verifying that the key algorithms of the system and its subsystems provide the functionality described in the system specifications is the second goal of this design phase. In audio, video, or graphic systems, for example, it is necessary to test that the algorithm for processing the data has the precision necessary to produce the required resolution or quality. In the case of a communications system, there may be more than one communications protocol available. A test needs to be devised that determines which protocol is more appropriate for this system.

In order to refine, create, or simply test the system algorithms, it is necessary to create behavioral models of the key subsystems. These models, unlike the analytical model, *do* need to be fully functional behavioral models. Unlike implementation models, however, the behavioral models do not need to model system status at each clock cycle.

Tests which test system architectural and microarchitectural features are commonly referred to as Architecture Verification Tests (AVTs). AVTs will be used throughout the development process along with detailed tests which verify the logic implementation of the system.

Verifying the Partitioning and Packaging

The cost, the flexibility, and the performance of a system is determined in part by the partitioning of the system into hardware and software. In general, implementing system functionality in hardware rather than in software results in higher performance and higher cost. Implementing functionality in software rather than hardware, however, may provide the required flexibility, if the goal is to produce a set of related products with a range of features.

Every system, whether it is a computer system, a graphics system, or a telecommunications system, has both control and datapath logic. In a system that is *data-dominant*, where the majority of the system functionality is involved in processing data, a number of products of varied performance, features, and cost, can be created by partitioning of logic into hardware and software.

Verification of hardware and software partitioning ensures that all the required hardware and software resources and their protocols operate correctly. Verification of hardware partitioning ensures that all hardware datapaths exist across chips, boards, and backplanes.

Verifying the Implementation

At this point in the design process, the design has been partitioned into blocks, which are either subsystems—ASICs, FPGAs, microprocessors—or standard components—memory and cores.

Verifying the implementation of systems can be constrained by human and compute resources. Tests which verify system implementation are commonly referred to as Implementation Verification Tests (IVTs) and must be planned so as to limit the extent on the verification constraints while providing high confidence in the system implementation.

The verification needs for subsystems are different from those for standard components. Standard components do not require internal verification; it is necessary, however, to verify that these parts work within the context of the system. In contrast, it is necessary to verify that the subsystems work both internally and within the context of the system.

Full timing verification should be postponed until the chip-level integration phase of the design process, when all the blocks within a subsystem have gate-level models and the most accurate timing information is available. The focus of verification at the block-level implementation phase should be verifying the functionality of the models. For this purpose, cycle-accurate timing is adequate.

However, for timing-critical blocks in particular, designers can do much during block implementation to facilitate the chip-level integration. Using estimated parasitics from high-level floorplanning to drive synthesis, reoptimizing synthesized blocks to remove timing and design rule violations, and driving gate-level simulation with estimated interconnect delays from synthesis are some of the ways to do this.

The objectives for verifying the implementation are

- Verify that standard components work from a functional perspective within the system context

  The standard components should have “black box” behavioral models that perform all datapath and control functions including cycle-level timing, or that at least model the bus protocols and major bus cycles. If these models were not part of the system validation phase, they need to be verified in the implementation phase.

  These models also provide a cycle-accurate functional context for the subsystems that are under design.

- Verify the RTL and gate-level implementation of the subsystems within a cycle-accurate system context

  Subsystems are usually partitioned into subblocks of 5000 gates or so to facilitate synthesis and concurrent design by multiple designers. Each of these subblocks needs to be verified from a functional perspective within the system context at the RTL level and the gate-level.

- Verify critical hardware and software interaction

  Operating systems, application and link-layer control programs, and embedded microprocessor and microcontroller systems require different strategies of co-verification in order to verify control and data processing algorithms partitioned between hardware and software.

The strategy for achieving these objectives includes

- Using appropriate technologies

  Choosing the right technology requires a thorough understanding of the project needs, the verification strategy, and the available technologies. Run time, memory, and model compile time are important considerations.

- Developing a complete verification test plan

  Choosing the right test also requires a thorough understanding of the project needs and the verification strategy.

  The verification test plan defines what the tests are and how they test and verify the system. The different test types and methods are disclosed. Additional requirements are defined for: new models - monitors, checkers, transactors; and integration of software and firmware utilities. Simulation technology decisions are also confirmed.

  When the test plan has been implemented and the design passes all tests, you know that you have finished the verification process and it is now safe to release the design to prototype or production manufacturing.

- Creating the right simulation configurations

  Verifying that a subsystem or component works within the context of the entire system is essential. This does not mean that the entire design needs to be simulated repeatedly throughout the design process with the lowest-level models available. Careful design of configurations that focus verification efforts on key portions of the design and that make use of behavioral models to supply the system context can reduce overall time spent in simulation.

- Developing structured testbenches

  Understanding the structure of a good testbench can reduce the amount of time required to create one. Having test benches available from system-level design testing also facilitates the creation of block-level testbenches.

- Setting up verification procedures

  Working with a common set of procedures allows the team to keep track of model updates and progress against the verification plan

- Setting up automated regression environment

  Leveraging an automated verification regression environment increases throughput and keeps design changes from slowing down the pace of the project.

These topics are discussed in the following sections.

Using Appropriate Verification Technologies

Current design verification technologies include:

- Event-based simulation

  Event-based simulation is the mostly widely used verification technology today. Event-based simulators trace every event on every signal during every clock cycle. Although it is the slowest technology, event-based simulation has the widest applicability. These simulators typically support models at any level of abstraction, from behavioral, to RTL, to gate and transistor level. They also allow at least four states.

- Cycle-based simulation

  Cycle-based simulation is faster than event-driven simulation, but it has narrower application. Cycle-based simulators only calculate the state of the circuit at the end of every clock cycle, and they typically only allow two states. By definition, cycle-based simulation requires synchronous designs and no arbitrary assignment delays. All timing and model parameters need to resolve at compile time which also requires a C compiler.

- Formal verification

  Formal verification tools are most often used to verify the equivalence of two circuit descriptions, where the descriptions are at the same or different levels of abstraction. Because the equivalency is proved mathematically, rather than dynamically, formal verification is much faster than simulation. Currently available tools cannot verify that the design meets the system requirements, however, so formal verification does not eliminate the need for simulation.

- Hardware acceleration

  Hardware accelerators are computer systems that have been especially designed to accelerate logic simulations of models at various levels of abstractions, including behavioral, gate, and transistor.

- Emulation

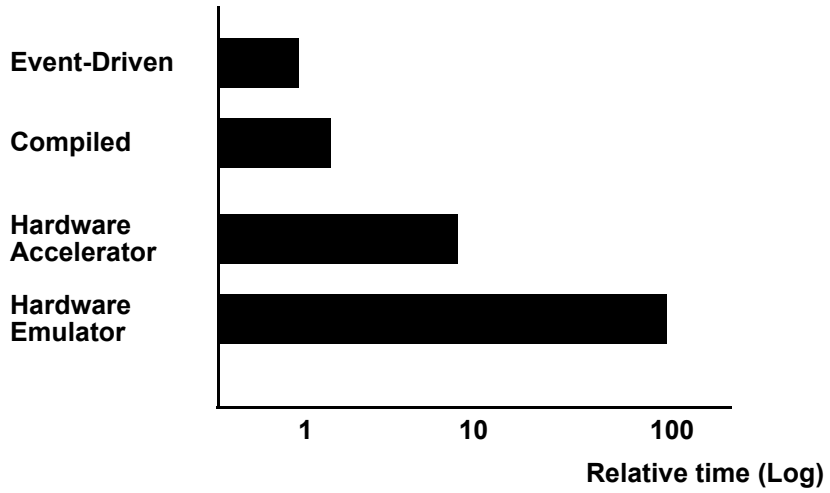
  Hardware emulation tools create a prototype of a design by mapping a gate-level or transistor-level description into FPGAs or other programmable logic devices. Until recently, emulation tools did not support high-level (RTL) descriptions. [*EE Times*, September 25, 1995]

- Hardware prototype

  Verifying a design by testing a prototype is the fastest way, if you measure speed in terms of cycles per second. However, relying on this method will cause schedule delays in the long run, when late-surfacing errors cause costly design iterations.

It is now common to employ several technologies on a project. Knowing when and how to use these technoligies requires planning the size of the effort targeted at each technology. Run time, as shown in Figure 5-1, and model build time should be considered together. Incremental compilation is also used to cut the model build times significantly. Complete model compile times for the various simulation technologies as shown in Figure 5-1. Because of the rate of defects found is highest in initial model debugging of AVTs and IVTs, compiled and event-driven simulation is best suited to verifying design changes. Later stages of design when test suites are nearly completed and a large porion of tests pass, hardware acceleration and hardware emulation can be effective at reducing the run times. Throughput is increased eventhough model iteration is longer becuase of the relatively infrequent model changes and speed of special purpose hardware engines.

Figure 5-1 Model Build Times



Choosing the Appropriate Tests

Design verification efforts have become very sophisticated with the advent of high gate count ASIC-intensive systems. It is now common to find that the investment in verification tests and related models now exceeds the hardware modeling effort of the actual ASIC and standard components. Therefore verification of a system implementation must be thoroughly planned. Starting with the same architectural specification as the ASIC and PCB designers, the verification leader and his team will start to decompose the system for the purpose of defining the layers of verification startegy.

- User application
- Software protocols

- Operating system
- Processors and Caches
- Exception handling and error recovery
- Flow control (HW/SW)
- Dataflow and adaptation
- Functional unit
- Connectivity
- Timing

At each stage of system decompostition, architectural and microarchitectural features are identified and a verification strategy and test plan is developed. A Verification Test Plan (VTP) is a formal specification that plans how to verify design requirements including:

- Overall system performance utilizing a set of system resources
- Adherence to system architecture and microarchitecture features
- Correctness of ASIC and PCB implementation

Relative to these design requirements, three categories of tests will be utilized for each stage of the design implementation process including:

- Performance Verification Tests
- Architecutal Verification Tests
- Implementation Verification Tests

Each test suite which is specified in the VTP will define several key items including:

- Level or Configuration

  Model build necessary to test specific system features including the required hardware, software, firmware, and OS modules. Also includes the appropriate model abstractions that are supported

- Type

  Purpose of each test and how often it is run

- Method

  Definition of what is being checked and how

- Technology

Simulation technology resources and revision levels required for each test including the hardware simulation, software application and drivers, OS levels, and firmware.

- Schedule

  Schedule and order of development and debug

Together test level and type define the scope of system features that are targeted. Test type further defines the purpose of the test and how it will be used throughout the overall verification process.

Verification Test Types

Between AVT and IVT test suites, five types of tests are commonly used in the verification of large systems. Each new type represents refinements to the verification tests used throughout system implementation and results in higher confidence levels of the final product.

- Check-in tests

Represents a small set of tests which validates configuration integrity and basic operations every time a model or test is updated. These tests typically execute in a small period of time.

- Directed tests

Comprehensive set of tests which cycle through many boundary conditions of each architecture feature of the system. These tests also chain sequences together to test subsystems, HW/SW interaction, memories, etc. These tests typically execute in a moderate period of time.

- Random or Exhaustive tests

Generates random values and sequences of system operation to target testing the more combinations of IOs and model state to uncover functional errors, and deadlock and contention of system resources. Random testing is not known for its efficiency because it is very easy to generate duplicate tests. These tests typically execute in a long period of time and are run later stages of the design project.

- Stress tests

These tests target hardware and software exception handling, interrupts, and error recovery by simulating high activity situations. These tests usually require having hardware, firmware, and software necessary to implements higher level system protocols and control functions. These tests typically execute in a long period of time and are run in the later stages of the design project.

- System tests

Runs system boot and initialization, diagnostics, operating system kernal and drivers, and application software. These tests target integrating the complete system. These tests typically execute in a long period of time and constitute 10s to 100s of seconds on actual real-world system operation. These tests are run in the final stages of the design project. These tests rely on all other tests passing before these are run.

System tests cannot replace running diagnostics, stress, and maturity tests on actual system prototypes, however most system integration defects can be eliminated simply by integrating all of the system resources and verifying basic system operations.

Verification Test Method

The method of each verification test specified in the VTP may consider proving different properties about the system implementation. One or more methods are used together to support verifying the system implementation from different architectural perspectives. Methods of verification include:

- Coverage tests

  Evaluating how much of the design is verified

- Compliance test

  Validating compliance with rules, protocols. properties, and attributes of the system operation

- Correctness tests

  Verifying correct design implementation for the applied test.

After all of the VTP is implemented and passes, the design can be released to production manufacturing. Additional production tests will be developed including boot diagnostics, system diagnostics, and design maturity tests.

Verification Test Configuration

As the designers move into the block implementation phase of the design process, the models of the system and its major subsystems start to proliferate. At the end of the system validation phase, there are behavioral models available for each of the subsystems and each of the standard components. At the start of the implementation phase, the subsystems are further partitioned into subblocks; then RTL models and finally gate-level models are created for these subblocks. Each of these models will likely

have various versions as the models are debugged and refined during the process of validation.

Simulating the entire design with the latest versions of the lowest-level models, although conceptually simple, is likely to very inefficient. Simulations on that scale require a lot of time and hardware resources, and in any case, the results would be difficult to analyze. Smaller, shorter, more focused testing is what is required.

A simulation configuration groups particular types of models of particular parts of the system together for a well-defined verification purpose. A simulation configuration includes a test bench—a test fixture and test monitors—designed to address that verification purpose.

The models that make up a subsystem are a configuration, for example. Higher-order configurations, designed to verify the subsystem within the system context, are also required. Several factors that are need to be considered when designing configurations include

- Resources

  The configuration needs to simulate within a reasonable period of time given the hardware resources available. The amount of engineering time required to create the tests and analyze the results also needs to be considered.

- Interaction of a subsystem with other subsystems or components

  Configurations should be designed to test the interaction between subsystems or between a subsystem and its environment.

- Granularity of tests

  Architecture and Implementation Verification Test, AVTs and IVTs, define different feature sets and subsystems which are the target of directed tests

A sample simulation configuration is shown in Figure 5-1 for a heterogeneous hardware-software system. A system application runs on a general purpose computer and interfaces to an embedded hardware system which is controlled through firmware. In this example a behavioral testfixture controls the ASIC operations that are to be tested. Additional models provide the proper interface to the ASICs from other system resources including

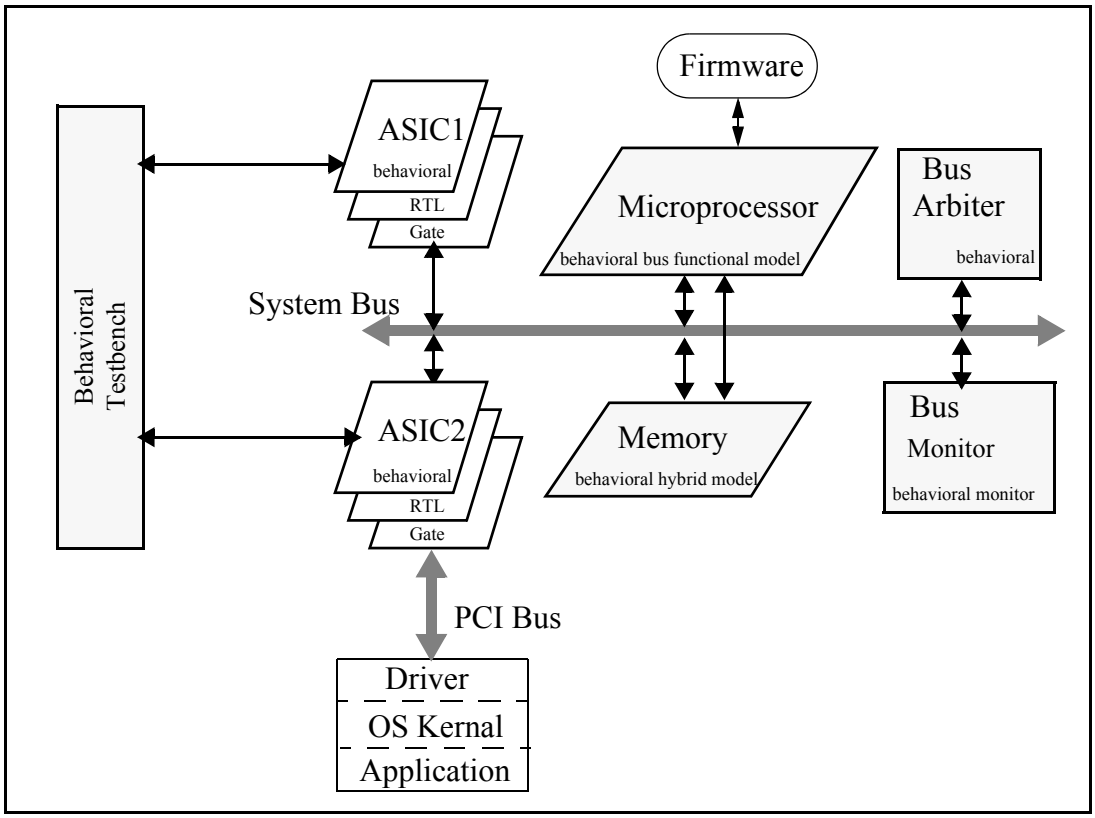
- Heterogeneous HW-SW protocols including device drivers
- HW-HW protocols including bus functional models
- Hybrid models

- Monitors
- Assertion Checkers
- Timing Verifiers

Bus functional and hybrid (mixed behavior and analytic) models model the behavior of the microprocessor, which dynamically controls bus operations and memory access.

The use of bus monitor, sometimes referred to as assertion checker, is also included to test the bus protocol for signal timing, bus contention, and illegal bus operations. Checkers or monitors are very effective at testing adherence to protocols and other complex state sequencing.

Figure 5-1 Sample Simulation Configuration



Developing Structured Testbenches

Developing good testbenches can take as long or longer to develop than the design models themselves. The testbench designer must understand HDL modeling, understand the nature of the configuration under test, and select the appropriate type of test to implement. The discipline and structure

needed to accomplish this often determines the success of a verification strategy.

Many of the same approaches that go into good behavioral model writing also go into testbench development including:

- Separating test content from stimulus timing

  Making changes to the test content should be kept independent from altering the timing of the signals

- Separating the testbench from the compiled model

  The designers can run and alter multiple test suites without having to recompile the model. Enhances flexibility and efficiency in using multiple simulation technologies.

- Partitioning tests into separate testbenches

  The designers can debug tests more rapidly by not waiting for long runtimes before debug state is reached. Computing resources also can be better load balanced by taking advantage of a allocating a large number of relatively short tests.

If possible, test benches should be designed so that the tests can be run in batch and the results compared automatically with expected results. Automatic verification can be accomplished through

- Self-checking tests

  Tests verify the simulation through montiors which check proper protocol transitions and state sequencing, what protocol cases have been verified, toggle coverage, and expected values. Co-verification using a known good model concurrently simulates and compares outputs of both models to verify the newer model.

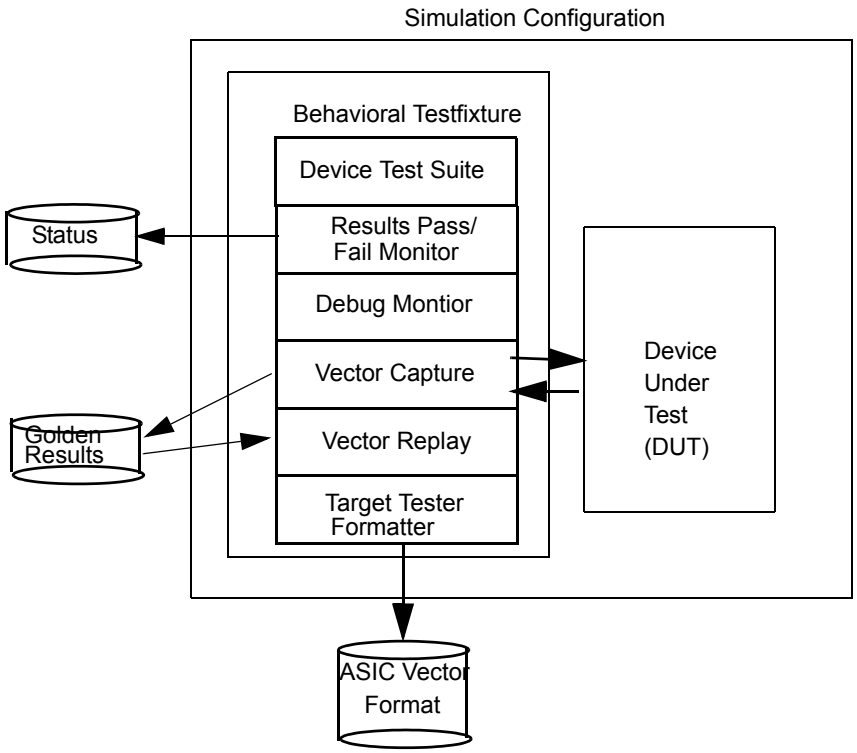
- Comparison to a golden reference

  Comparison to the architectural specification or previously generated results files. A reference model can be generated throughout all levels of abstraction - architecture to gate-level. At the systems level, attributes and properties can be extracted from the simulation through monitors and assertion checkers and then compared to system requirements defined in the architectural specification. At the gate-level, signals and register state values can be compared to golden results generated from previous behavioral, RTL, and gate-level simulations.

After these automated steps uncover a bug, time-consuming interactive debugging with visualization tools such as waveform analyzers can be utilized.

A sample testbench architecture is shown in Figure 5-1. The various functions of the testbench are described below.

Figure 5-1 Example of Reusable Structured Testbench



- Device Test Suite

  This process controls which tests are run during simulation regression. Typically the test suite models the DUT environment, generating stimulus that mimics the I/O in which the device operates.

- Results Pass/ Fail Monitor

  Automatic results analysis can be performed by post-processing or run-time self-checking. Post-processing compares current results to a golden reference database. Run-time self checking can

  - Generate expected system response, possibly by using already verified functionality, and compare to current results
  - Use ATPG and BIST/LFSR to generate and check signature
  - Verify coverage levels of state transitions and assertions for valid and invalid design behavior.
  - Compare results of concurrent simulation of two different models of same device

In most cases, test benches have to handle results comparisons that could mistakenly indicate wrong results. These scenarios arise from comparing two sets of results where inherent timing is different, including

- Different data types
- Different valid comparison periods within a cycle
- Different model initialization cycles
- Different filtering responses

- Debug Monitor

  Conditionally compiled debug code which enables interactive debugging including stopping and starting simulation using breakpoints, configuring visualization and display facilities, and setting signal forces to selectively test different debug hypothesis.

- Vector Capture

  This process manages the opening of and capture of signal activity to a vector trace file, VCD file, or database for future processing and/ or replay. These vectors can become “Golden” vectors if 100% of the tests pass. If these vectors are used for replay on another model, then the vectors need to be sampled at the appropriate times to maintain cycle accuracy.

- Vector Replay

  This function manages the opening of “Golden” vector trace files for replay and comparison with the current circuit activity. Since the behavioral portion of the test fixture is not needed at this point, a significant improvement in simulation performance can be obtained for gate level simulation and RTL regression testing.

- Target Tester Formatter

  This function is used if pattern file formats are required for ASIC vendor sign-off. This process monitors the design modules in the configuration and writes the input and output values at the specified strobe times.

Setting Up Verification Procedures

The verification process involves validating the system intent and the implementation through all of its iterations. A clear procedure must be followed from start to finish. Work can be leveraged work through all of the stages of verification while iterations from rework and errors can be reduced.

Automating the verification model integration and regression environment simplifies the process to verify design changes made at various levels. Simulation job scripts, which automatically run different configurations and test suites based on updating models and tests, greatly streamlines the effort to conduct the verification. A design verification manager should develop the design configurations that will be supported and implement an automated process to perform model integration, builds, check-in testing, and release.

When a change is made to either the design or the test suite there are some commonly used procedures that should be followed:

1. Check updated models into source control system

2. Add new tests to check-in test suite

3. Run check-in verification tests

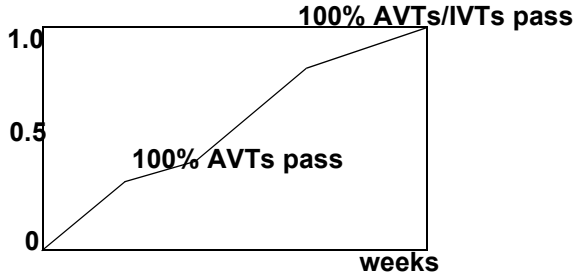
4. Notify group of pass/fail status of tests and/or models in new release

5. Run full regression tests (AVTs or IVTs) for all appropriate model configurations

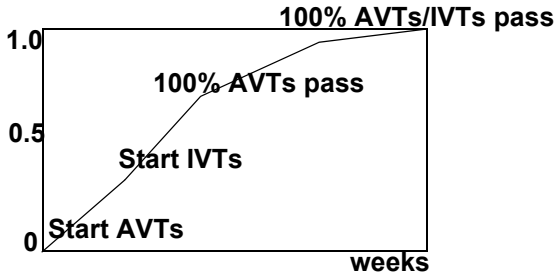
6. Notify group of new release for continued design and verification

Procedures for verification can cut simulation time significantly by streamlining model and test suite promotion. Furthermore formal design verification procedures enable better tracking of the progress of the effort. The design verification manager should keep statistics on progress against plan and defect rate.

Figure 5-1



Verification Cycles Passing vs Time (Normalized)



Cummulative Defects vs Time (Normalized)

A detailed list of verification milestones can include:

- Number of configurations up and running
- PVTs implemented and passing
- AVTs implemented and passing
- IVTs implemented and passing
- Number of defects
- Defect rate per week
- Defect closure rate per week

Automating Simulation Regression

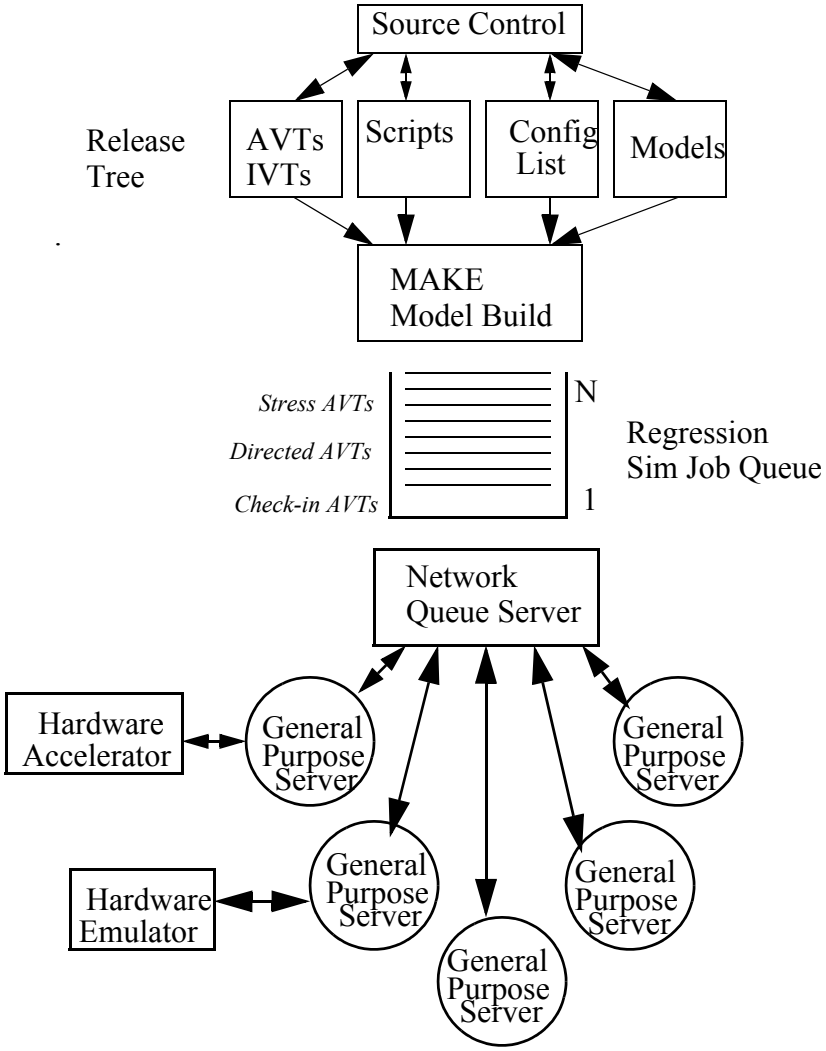
In today's computer, multimedia, and telecommunications systems, verification requires an enormous number of cycles to be simulated and analyzed. It now takes over 10B clock cycles to run all the verification tests for a general purpose RISC processor compute server. Judicious care must be taken when adding new tests or new model builds to the regression environment. Furthermore because cummulative defect levels for large systems typically totals in the 100s to 1000s before system sign-off, the resulting cummulative number of required simulation regression runs will also total in the 1000s or more before all defects are resolved.

A regression server is a good way to enhance verification throughput by providing maximum utilization of all compute resources available for verification. Typically verification configurations and the verification test suites are defined ahead of time. When model or verification test suite changes are made a large number of simulation runs will need to be made if the model configurations and verification tests have been properly partitioned.

An automated build tool, such as *make*, provides dependency checking and compile scripts which can control simulation regression by only running newly released model configurations which have changed from a previously specified snapshot.

To realize high performance verification throughput, concurrent simulation are run on a heterogneous compute cluster comprised of general purpose servers and dedicated special purpose hardware. A network job queue server is used to dispatch and balance the regression runs across the cluster. A regression server model is shown in Figure 5-1.

Figure 5-1 Automated Simulation Regression Environment



For large systems designs it is often mandatory to keep simulation running non-stop throughout the duration of the project to meet current product development schedules.

References

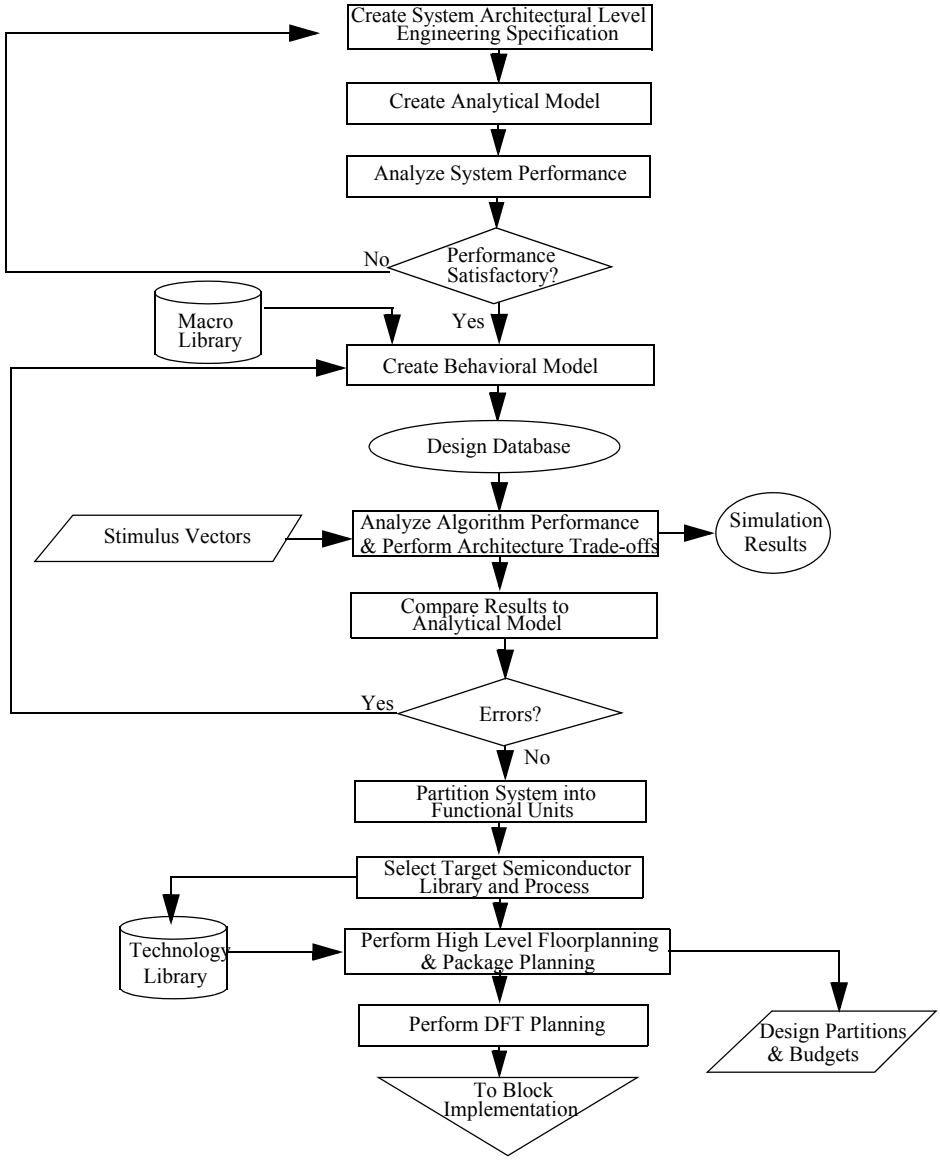
7

High-Level System Design

Process Overview

The high-level design phase focuses on defining the system requirements and on developing and evaluating various executable models that meet those requirements. Once system performance and functionality have been validated, the actual hardware system requirements and partitioning are defined. Figure 8-1 shows the high-level system design flow.

Figure 8-1 High-Level System Design Flow



The following chapter discusses the system design of a Dual Tone Multi-Frequency (DTMF) receiver to illustrate process and methods presented in the previous chapters. Design of the DTMF receiver illustrates the application of specific top-down design methods and processes described throughout this guide covering systems design through implementation. The design models, verification, test, and physical design databases are presented and discussed.

The DTMF design represents a common telecommunications application. The following information summarizes the DTMF design.

Key design highlights and design process decisions:

Design Environment

- Utilized a single design environment for hardware and firmware development making use of source control and
- Developed program to perform automatic hardware/firmware compatibility checking throughout the project
- Selected SCCS and makefile for single, efficient, and low cost source control and hardware, software, and firmware build process
- Utilized Perl, Awk, and Csh for process automation

System Design

- 16-bit DSP with single cycle instruction execution (6 clocks per cycle) and direct and indirect addressing modes
- Single chip solution with on-chip SRAM for small footprint
- DFT algorithm investigation resulting selection of Goertzel algorithm for fast and efficient execution in DSP firmware
- Meets all AT&T compatibility requirements for DTMF signalling
- Microarchitecture supports dynamic range calculation on PCM data resulting in 16-bit DSP architecture versus 32-bit while maintaining frequency response requirements

Firmware

- TDSP assembler developed supporting 65 instructions with direct and indirect addressing modes
- getop utility developed to check and maintain consistency of assembler and hardware models for implementation of TDSP instruction set

Design Modeling and Verification

- Eight model build configurations verified at systems through netlist levels
- genpcm utility developed to generate PCM encoded signal data for use with C and HDL models adjusted for line noise, signal level attenuation, and twist
- Design models in both Verilog and VHDL

Logic Design

- Datapath synthesis (Synergy) utilized due to high content of 16-bit logic and arithmetic operators

Design-for-Test

- Full scan implemented using three scan chains
- RAMBIST with direct IO pin access utilized to test internal ram
- No JTAG or boundary scan due to no pincount restrictions

Timing Driven Physical Design

- Utilized hierarchical physical design to facilitate reuse of the TDSP and to accommodate both standard cell and datapath placement and route
- Over 150 datapath elements used in two datapaths for the TDSP and RCC blocks

Figure 8-2 is an overview of the DTMF chip implementation including:

Figure 8-2 Chip Package Summary

Chip Package	
Number of pins	22
Library technology	Philips 0.35 micron
Library wire levels	3-layer metal
Voltage	3.3V
# Clocks	4
# Gate Equivalent	53,000
Size (mm x mm)	0.075 X 0.075
Operating frequency range	DC-25 MHz
# FSMs	8
Total States in FSMs	35
# flip flops	627
Floorplan regions	6
Datapath	2
# Datapath elements	159
# RAMs	2
# Bits of RAM	6K
# ROMs	1
# Bits of ROM	8K

Figure 8-3 Modeling Summary

Modeling	
# lines of C for goertzel_sim	238
# lines of C for genpcm	754
# lines Verilog behavioral HDL for goertzel_sim	254
# Functional Blocks	31
# lines RTL HDL	5589

Figure 8-4 Design Verification Summary

Design Verification	
Cycles verified	75,000,000
Simulation throughput (cycles/sec)	11,000
# seconds of real time operation	3
# lines Top-Level Testbench	567

Figure 8-5 Firmware Summary

Firmware	
# lines assembler output	536
# lines firmware source	822
# lines of Perl for tdsp assembler	1174
# lines of Perl for getop instruction correlator	227
# words(bytes) in firmware object	307(614)
DSP architecture	16-bit operand, 32-bit accumulator
DSP instruction size	16 bit
# DSP Instructions	65
DSP Addressing modes	2

Figure 8-6 Design for Test Summary

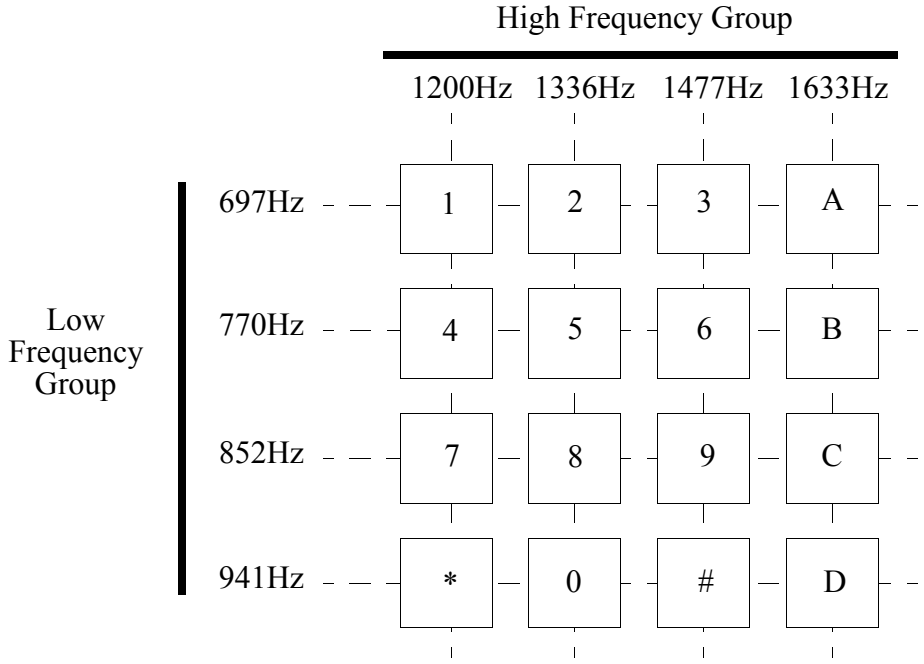
Design for Test	
DFT strategy	Full Scan, RAMBIST with direct IO pin access
BIST patterns	127,500 patterns
BIST tester time	9 ms (80MHz test clock)
ATPG coverage	94%
ATPG vectors	400
# Test Clocks	3

System Specification

In a telephone network, two basic techniques are used for transmitting information between network entities: in-band and common channel signalling. In-band signalling shares the transmission facility for signalling and voice data. Common channel signalling uses one transmission facility for all signalling functions for a group of voice channels.

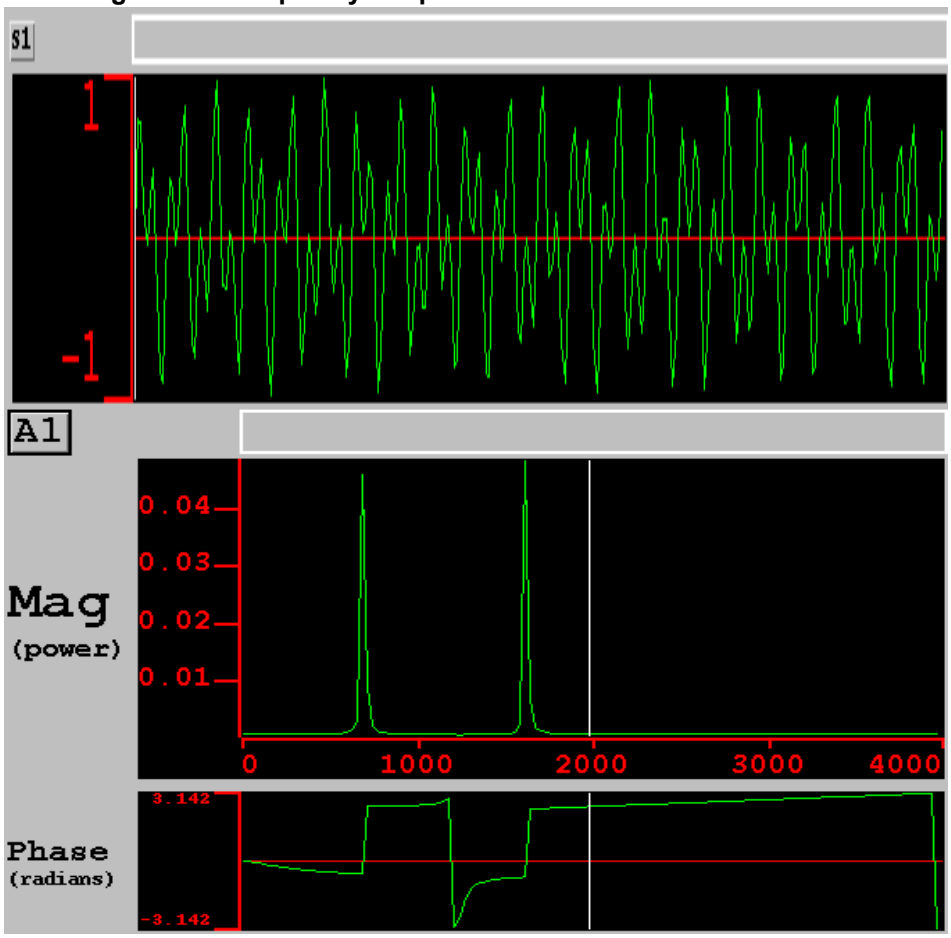
One common form of in-band signalling is dual tone multi-frequency, or DTMF. DTMF signals are commonly generated by “touch-tone” telephones; most of us probably have this type of telephone in our homes today. Figure 8-7, “DTMF Keypad and Character Frequencies,” is a layout of a “full” DTMF keypad.

Figure 8-7 DTMF Keypad and Character Frequencies



Notice that keys “A”, “B”, “C”, and “D” are not usually on telephones for home use. These keys are mainly used in commercial applications with special instruments. Pressing a key on the keypad causes the telephone to generate the indicated pair of tones, one from the high frequency group, and one from the low frequency group. Figure 8-8 is an example of a DTMF signal along with its frequency response.

Figure 8-8 DTMF Signal and Frequency Response



Telephone specifications, such as *Touch-Tone Calling - Requirements for Central Office (*AT&T Compatibility Bulletin No. 105, August 8, 1975) define a DTMF digit as follows:

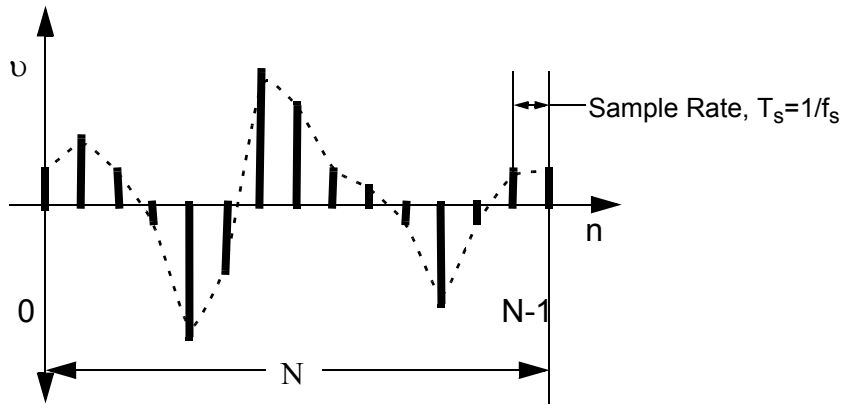
- A DTMF digit is a pair of tones, one from the low frequency group, one from the high frequency group.
- A DTMF digit must have a nominal level, per frequency, of -6 dBm0.
- The maximum rate for DTMF signalling of 10 digits per second (or typically 100mS per digit).
- A DTMF digit must be present for at least 45 mS.
- A “quiet period” must exist between digits for at least 45 mS.
- Upon reception, the signal difference between the low frequency tone and high frequency tone does not exceed 8 dB.
- Upon reception, the signal difference between the high frequency tone and low frequency tone does not exceed 4 dB.

Algorithm Development

A simple DTMF detector can be built using a group of band-pass filters, each followed by a peak detector, which has a natural time constant of about 35 mS. The output of the peak detector would drive a threshold comparator, which in turn would drive a decision logic circuit. This type of implementation would call for a group of analog circuits that would probably require “tweaking” during assembly line production, or later during the product life as components age.

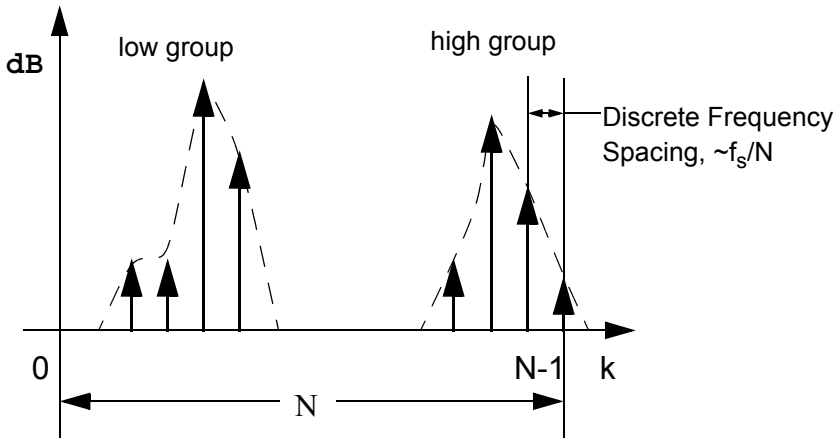
A digital signal processing based system has many advantages over an analog implementation, so we’ll sample the input signal. The discrete signal is referenced using “n”, the sample position number in a waveform window, and “N”, the length of a waveform sample window.

Figure 8-9 Sampled Input Signal



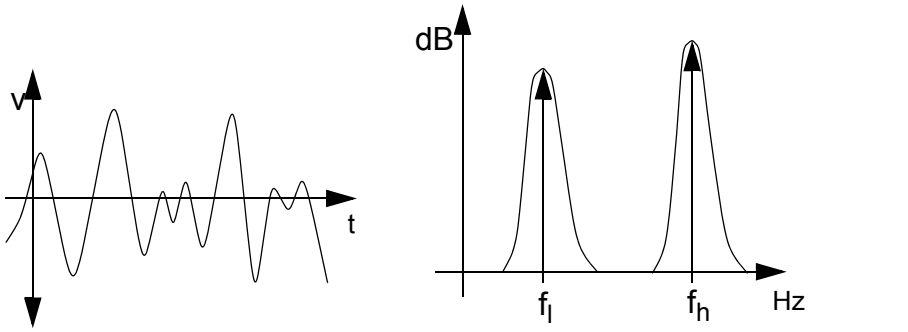
The DSP system will use the “window” of samples to compute a discrete frequency spectrum response. Note that “N” is also the width of the calculated discrete spectrum.

Figure 8-10 Computed Discrete Spectral Response



Because the DTMF receiver only needs to calculate a partial frequency spectrum—the frequency response at the DTMF center frequencies—a comb type filter response is desired. Each bandpass of our comb filter will be centered at one of the DTMF center frequencies.

Figure 8-11 DTMF Signal and Filter Response



Input Signal, v vs t

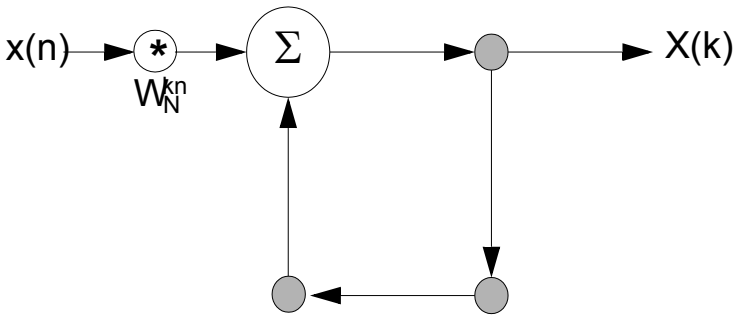
Filter Frequency Response,dB vs Hz
(for clarity, only 2 of the 8 filter responses shown)

The filter response could be computed by direct calculation of the DFT at each frequency as shown in Figure 8-11.

Figure 8-12 Discrete Fourier Transform Computation of X(k)

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} \quad , k = 0, 1, ..., N-1$$

Note: $W(kn,N) = e^{-j\omega} = \cos(2\pi kn/N) + j\sin(2\pi kn/N)$

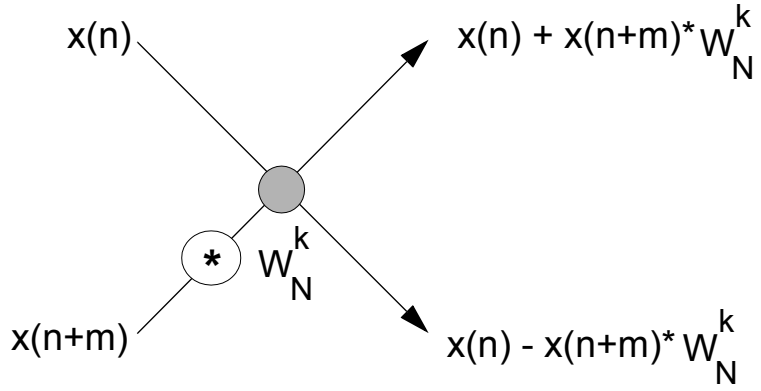


Alternatively, an FFT could be performed to compute the required frequency spectrum. Note that a traditional implementation of both the DFT and the FFT algorithm calculates a full, discrete frequency spectrum with filter resolution defined by the length of the sample window; as previously mentioned, in the case of the DTMF receiver we are interested in a partial spectrum.

Figure 8-13 Radix-2 "FFT Butterfly" for a "Decimation in Time" Implementation

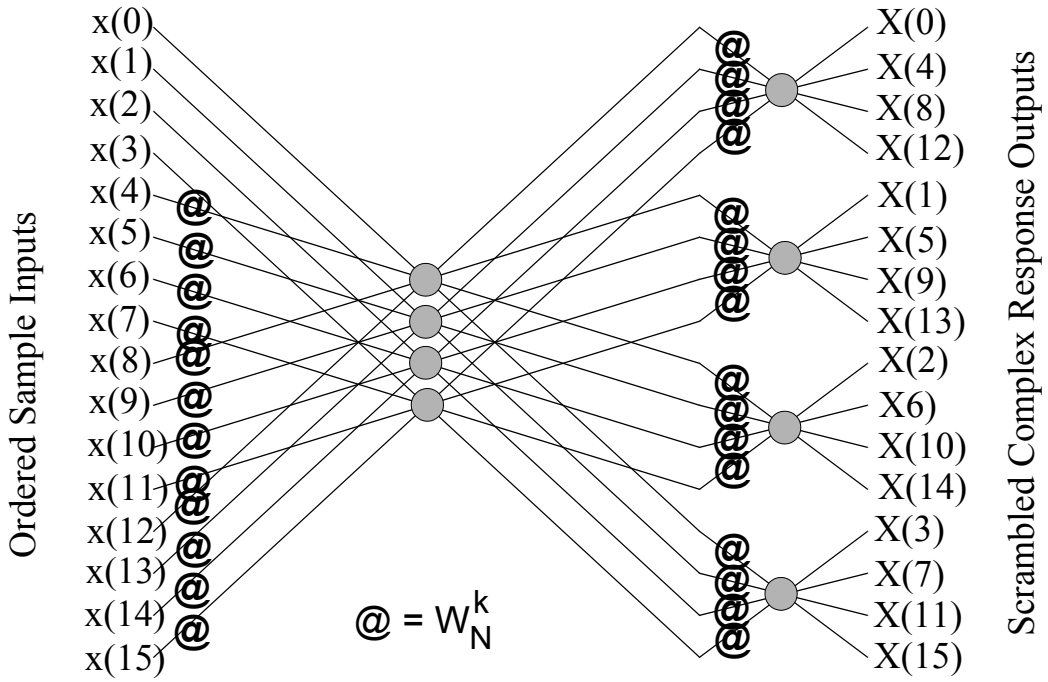
$$X(k) = \sum_{r=0}^{(N/2)-1} x(2r) W_{N/2}^{rk} + W_N^k \sum_{r=0}^{(N/2)-1} x(2r+1) W_{N/2}^{rk} \quad , k = 0, 1, ..., N-1$$

(n = 2r, for even n; n = 2r+1 for odd n)



Indeed, the FFT is very efficient when a full spectrum calculation is required. Lets take a look at a larger spectral response using the algorithm and a 16-point FFT implementation.

Figure 8-14 16-Point, Radix-4 "Decimation in Time" FFT Implementation



One thing to note about the FFT is the amount of memory "inferred" by the directed tree graph as the transform is calculated. Let's assume for the moment that the word size of *storage memory* is equivalent to the word size of the *accumulator*. Through careful optimization of the calculation algorithm, it would be possible for the calculation to be done "in-place" using the same buffer for intermediate variable storage that contained the data sample window. This assumption is rarely the case.

Since our receiver device must be inexpensive to manufacture, we would like to minimize the amount of memory required to perform the spectral calculations. Further research proved that a modified discrete Fourier transform (DFT) algorithm known as Goertzel's algorithm would be a more efficient algorithm since we are only interested in calculating the frequency response at the DTMF center frequencies.

Original DFT:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}, \quad k = 0, 1, \ldots, N-1$$

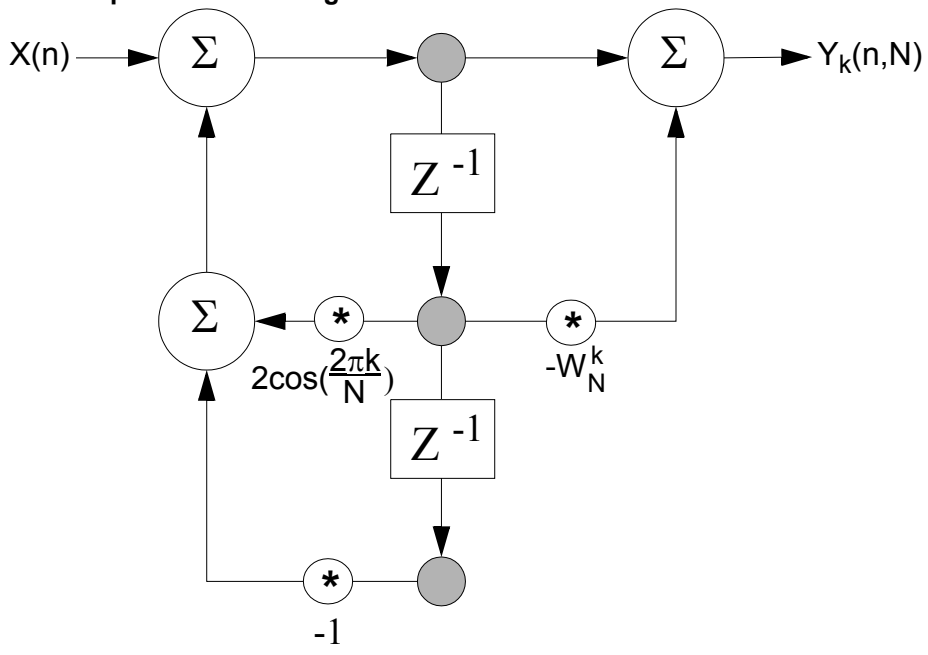
Second-Order, recursive computation of X(k):

$$\frac{y_k(n)\big|_{n=N}}{x(n)} = H_k(z) = \frac{1 - (W_N^k z^{-1})}{1 - 2\cos((2\pi/N)k)z^{-1} + z^{-2}}$$

The Goertzel algorithm is a very efficient method of calculating a partial frequency spectrum using a second-order recursive computation (calculation of the DFT is known as the “direct form”). Because of the recursive nature of the algorithm and the cost requirements of our system, it was decided that the Goertzel algorithm will run entirely in firmware on Tiny Digital Signal Processor (TDSP).

Figure 8-15 is a flow graph of the Goertzel algorithm.

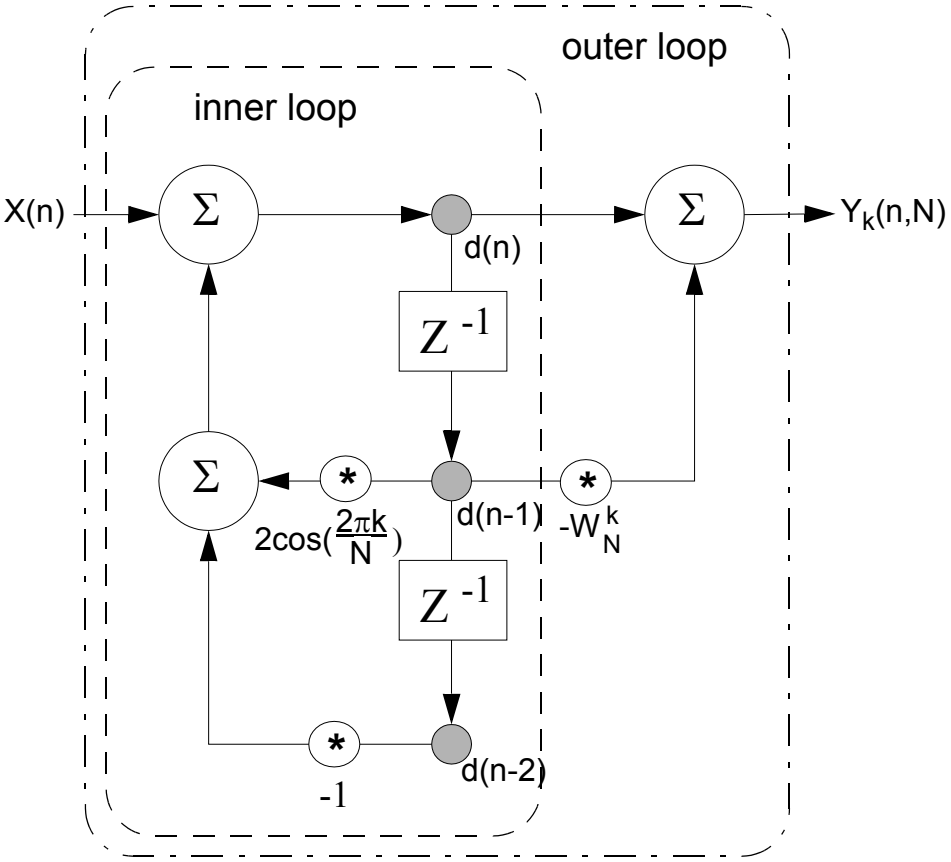
Figure 8-15 Flow Graph of Goertzel Algorithm



As suggested by the flow graph, the Goertzel algorithm takes the form of a second order infinite-impulse-response (IIR) filter. For spectral analysis,

the only interesting calculation is the last iteration (N-1) of the algorithm. At this point $Y_K(N) = X(K)$, the DFT response. What may not be readily apparent is that only the left side of the graph is calculated for most of the input samples ($0 <= n <= N-2$). It is only when $n = N-1$ that both side's of the graph are calculated. Since W^k_N is a complex number, complex multiplication is only required once per algorithm iteration.

Figure 8-16 Goertzel Flow Diagram



The motivation behind choosing the Goertzel was purely to reduce the number of computations (+, -, *) needed per desired spectral component in the calculated frequency response. Let's summarize the required number of computations for our alternatives:

- DFT - requires approximately $4N^2$ real multiplications and N(4N-2) real additions per frequency
- FFT - requires approximately ($N \log_2(N)$) complex multiplications and additions, also requires much more scratch-pad memory for a **full** spectrum calculation

- Goertzel - requires 2(N+2) real multiplications and 4(N+1) real additions per frequency

The sampling frequency of our system is fixed by digital telephone networking equipment at 8000Hz. Using a system model of the Goertzel transform, it was determined that the following algorithm attributes would yield acceptable digit detection performance:

- System Sampling Frequency, $\mathbf{f_s = 8000\ Hz}$
- System Sampling Period, **T= 125 uS**
- Signal Window length & Transform Length, **N = 128** samples
- Frequency Selectivity, $\mathbf{(f_s/N) = 62.50\ Hz}$
- Goertzel Discrete Frequencies, **k = 11, 12, 13, 15, 19, 21, 23, 26**
- Recalculated DTMF Center Frequencies, $k*(f_s/N)$:
  - **687 Hz** (desired center frequency: 697 Hz)
  - **750 Hz** (desired center frequency: 770 Hz)
  - **812 Hz** (desired center frequency: 852 Hz)
  - **937 Hz** (desired center frequency: 941 Hz)
  - **1187 Hz** (desired center frequency: 1200 Hz)
  - **1312 Hz** (desired center frequency: 1336 Hz)
  - **1437 Hz** (desired center frequency: 1477 Hz)
  - **1635 Hz** (desired center frequency: 1633 Hz)

Since we are dealing with discrete frequency in the digital domain, there is a small percentage error between the re-calculated center frequencies and the desired center frequencies of interest as found in the DTMF specification.

Once calculated, the computed spectral response must be analyzed to determine what, if any, DTMF digit was found. This portion of the DTMF algorithm will be using a finite state machine (FSM) module. All DTMF digit parameters are checked as defined except the twist check, which we've relaxed to +/- 12 dB for simplicity to a simple shift function.

An additional requirement which we imposed on our receiver was to accurately receive DTMF signals that had a per frequency level of -45dBm0. Although it was determined that the processing algorithm could easily compute the spectral response using 32 bit arithmetic, the system busses and memory would only be 16 bit wide. Instructions were included in the TDSP to store all 32 bits of the accumulator, but this would require

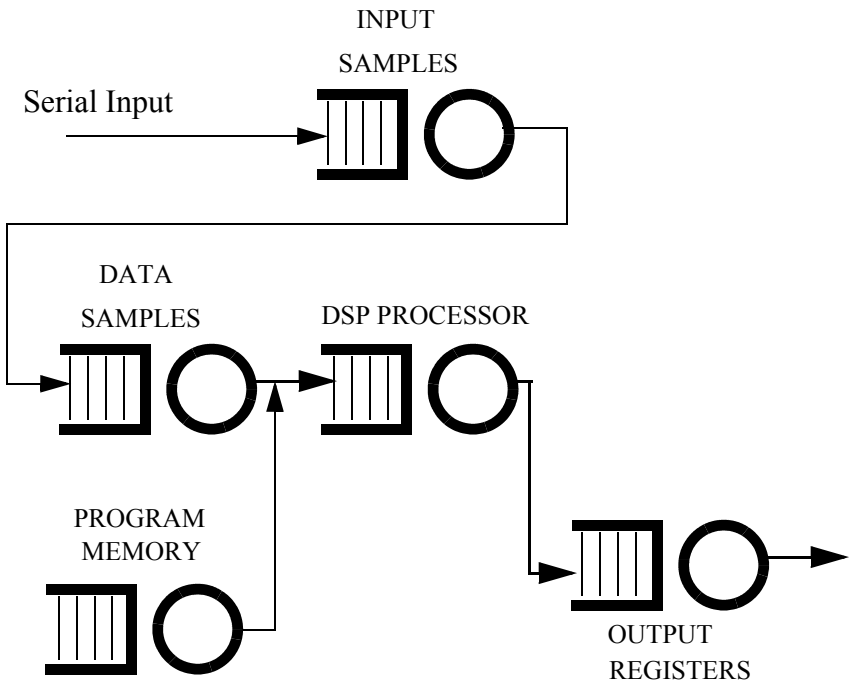
additional memory and load/ store cycles. Instead we chose to add an adaptive gain control algorithm to the receiver firmware that periodically assesses the level of the transform state variables. If any state variable approaches half the fractional value of the accumulator (0.5), the state variables are divided by two. It was analytically determined that the AGC function would review the state variable values every 16 samples, or when an overflow of the accumulator is detected.

Performance Analysis

Figure 8-17 shows an analytic network queuing diagram of the DTMF system, which has five servers with queues. In this closed system, the output sample rate is determined by the work throughput of the DSP processor. Work is performed on the samples in program memory and on incoming samples using the embedded software Goertzel algorithm.

The input sample queue will be processed by a serial-parallel interface which will fill the program memory. Contention for system resources occurs in the DSP processor between processing a batch of input samples and posting the processed samples at the output in time for the next set of input samples to be loaded into program memory. The system model needs to evaluate whether all input samples arriving at a target input rate (I_r) can be processed by the DSP processor and output at a target output rate (O_r).

Figure 8-17 Analytic Network Queuing Diagram



Using this approach, it can be determined that we would collect 128 at a time, this would represent a 16mS snapshot of the input signal. Further, it

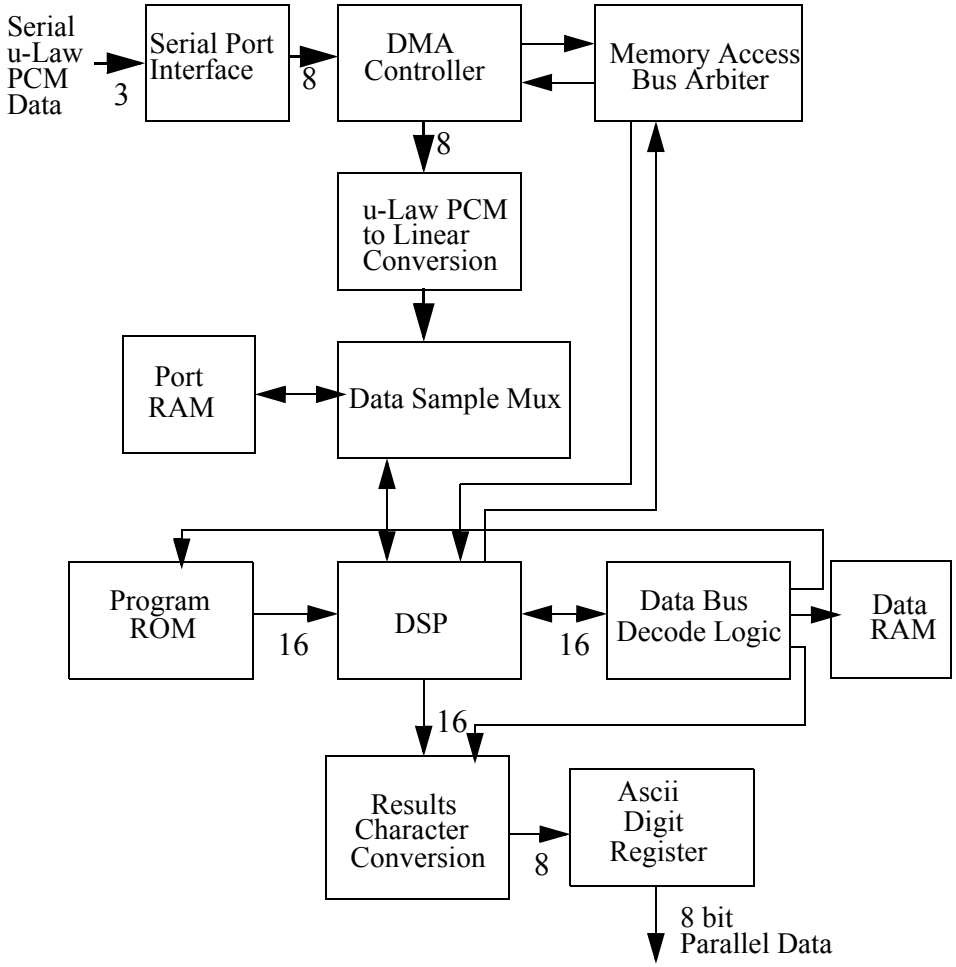
was determined that we would allow one forth of this time, or 4 mS, to our DTMF signal processing algorithm. This would allow for additional algorithms to be added to the TDSP firmware at a later date. For example, in a voice mail application we may also need to compress/ expand voice signals using the TDSP for efficient storage/ retrieval from a disk drive.

A full system simulation of the DTMF receiver, including the Goertzel transform in firmware, indicated that a TDSP operating at 25 MHz requires approximately 3.3mS of real time to calculate the required spectral response. The AGC function mentioned previously makes calculation of the transform somewhat non-deterministic within the range of the minimum and maximum calculation times required for the over 48 dB (+3 dBm0 to -45 dBm0 per frequency) dynamic range of signals presented to the receiver.

System Partitioning

Through algorithm investigation and performance requirements definition the microarchitecture features of the DTMF are developed, evaluated, and selected. System partitioning contributes to this microarchitecture design process where all major subunits and interfaces as well as IC packaging are determined. Initial technology investigation may be conducted to ascertain size, power, performance, and noise considerations. System partitioning leads to a model development plan which defines the modules and the model types that need to be developed for detailed design. This will be used by logic design, verification, and IC design teams to start detailed design. Figure 8-18 shows a block diagram of the partitioned DTMF receiver design.

Figure 8-18 Partitioned DTMF Receiver Design



Functional Specification

The following sections give a quick overview of each of the major blocks.

Serial Port Interface (SPI)

The serial port interface accepts u-law compressed PCM data, serialized least significant bit first, and reformats the data to byte orientation. The interface uses a clock signal to strobe the data on the signal's rising edge. A frame strobe is also used to indicate the start of a new data sample.

Once a character is received, the SPI signals the DMA controller that a new byte is ready to be moved to the Data Sample memory.

THe block will be coded as an explicit state machine with the following signal interface:

spi_clk - serial data clock input

spi_fs - serial data frame strobe input

spi_data - serial data input

clk - system clock input

reset - system reset input

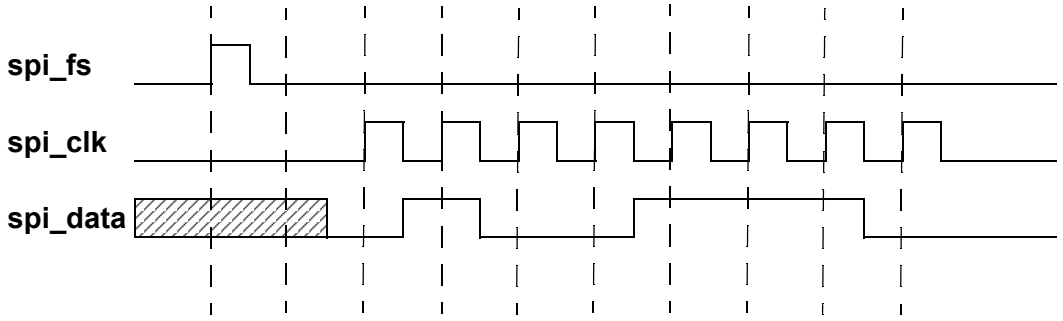
dout[7:0] - parallel data output

read - parallel data output enable input

dflag - new data flag output

The serial interlace timing is represented in Figure 8-19.

Figure 8-19 SPI Timing Diagram



DMA Controller (DMA)

The direct memory access (DMA) controller coordinates byte data movement between the SPI and Data Sample memory. Data transfer is initiated via the SPI. The DMA controller then attempts a data transfer by requesting access to the Data Sample memory via the Bus Arbiter. Once access is granted, the data sample byte is written to the data sample RAM.

The DMA controller will maintain two contiguous buffers (in the same RAM) and provide the DSP with an indication of which buffer is currently being filled.

The DMA block will be coded as an implicit state machine with the following signal interface:

clk - system clock input

reset - system reset input

read_spi - SPI parallel data output enable output

dflag - SPI new data flag input

breq - memory bus request output

bgrant - memory bus grant input

a[7:0] - address bus output

as - data address strobe output

write - data write strobe output

Memory Access Bus Arbiter (ARB)

The memory access bus arbiter (ARB) coordinates DMA and TDSP access to the Data Sample memory. The protocol is a simple REQUEST, GRANT scheme. Note that the arbiter is biased to allow the DSP priority access if both devices request at the same time.

The ARB block is coded as an explicit state machine with the following signal interface:

clk - system clock input

reset - system reset input

dma_breq - DMA bus request input

dma_bgrant - DMA bus grant output

tdsp_breq - TDSP bus request input

tdsp_bgrant - TDSP bus grant output

u-Law PCM to Linear PCM Conversion (ULAW_LIN_CONV)

This block expands the u-Law compress PCM samples to linear PCM samples. The u-Law compression/ expansion mechanism is specified in CCITT standard G.711.

The ULAW_LIN_CONV block has the following signal interface:

upcm[7:0] - u-Law compressed PCM input

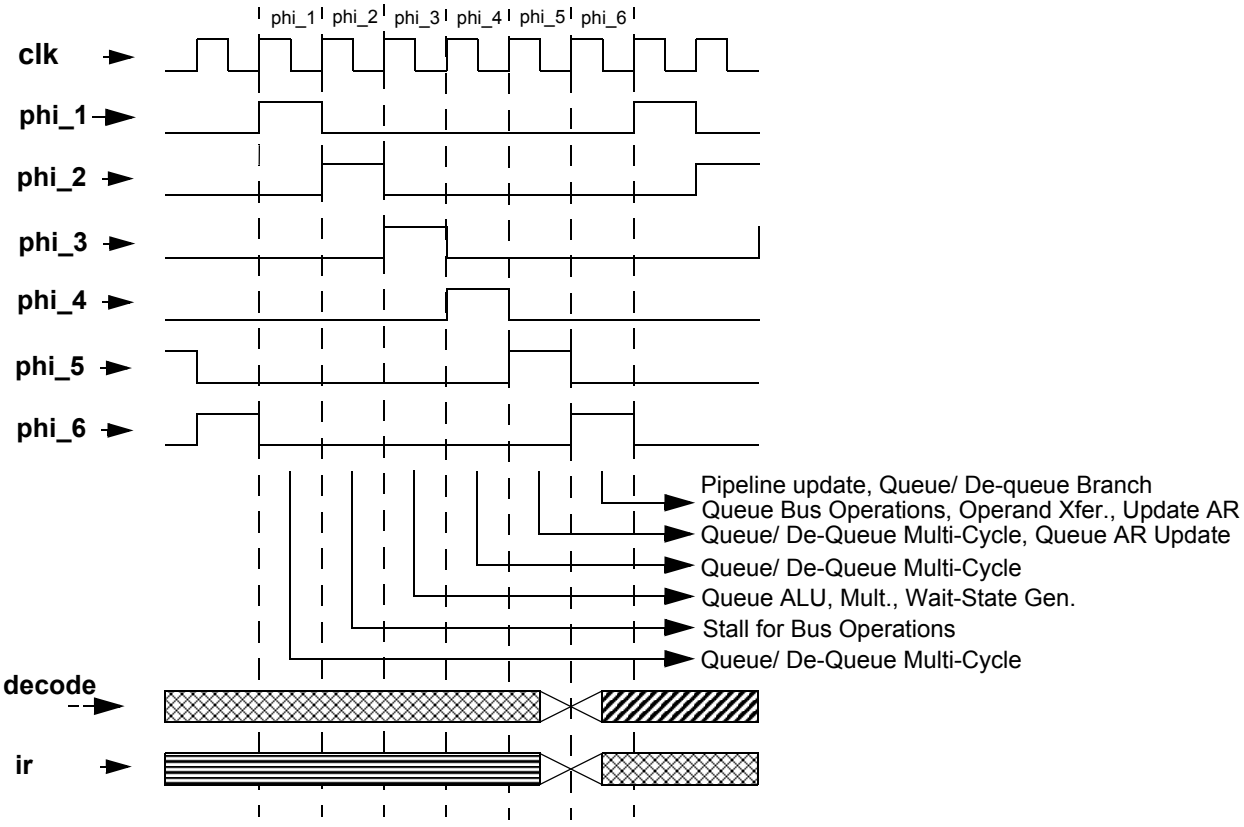
lpcm[15:0]- linear PCM output

Digital Signal Processor (DSP)

The Digital Signal Processor (DSP) mimics the instruction set of the TMS320 family of DSP's (actually its very close in functionality to the TMS32010, with a MAC instruction and bus arbiter interface).

The instruction pipeline can be represented in the timing diagram in Figure 8-20.

Figure 8-20 DSP Instruction Pipeline



The DSP has the following signal interface:

clk - system clock input

reset - system reset input

read - data read output

write - data write output

address[7:0] - data address bus output

data[15:0] - data bus

p_read - program read output

p_write - program write output

p_address[7:0] - program address bus

p_data[15:0] - program data bus

Figure 8-21 shows the DSP read and write cycle timing.

Figure 8-21 DSP Bus Cycle Timing

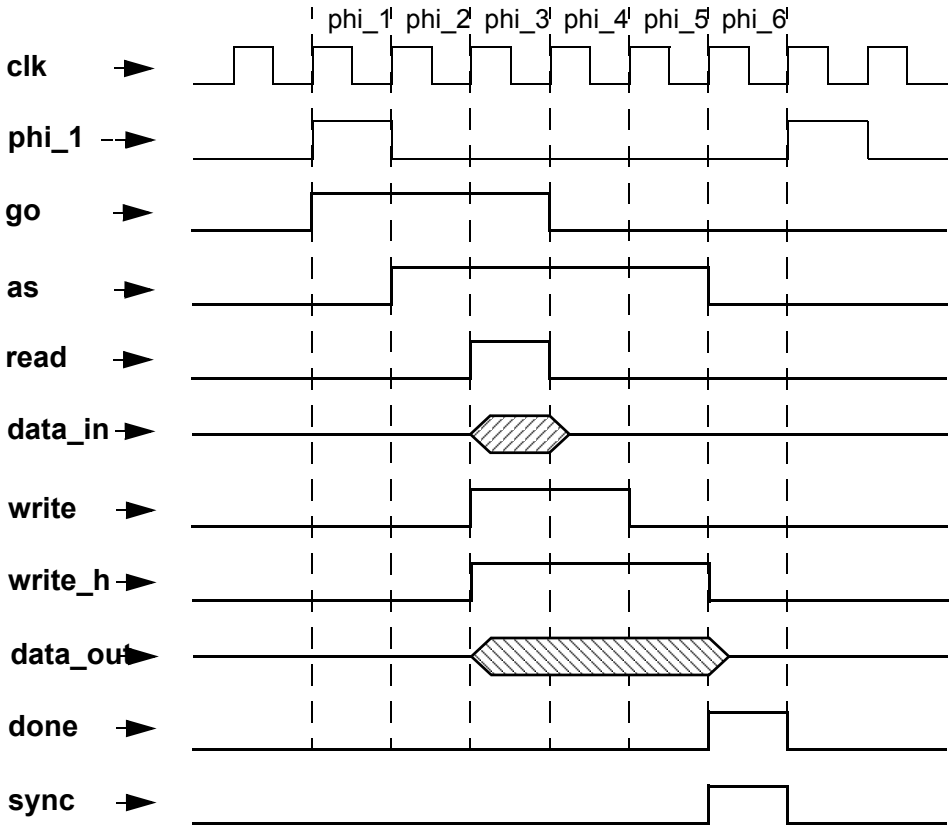
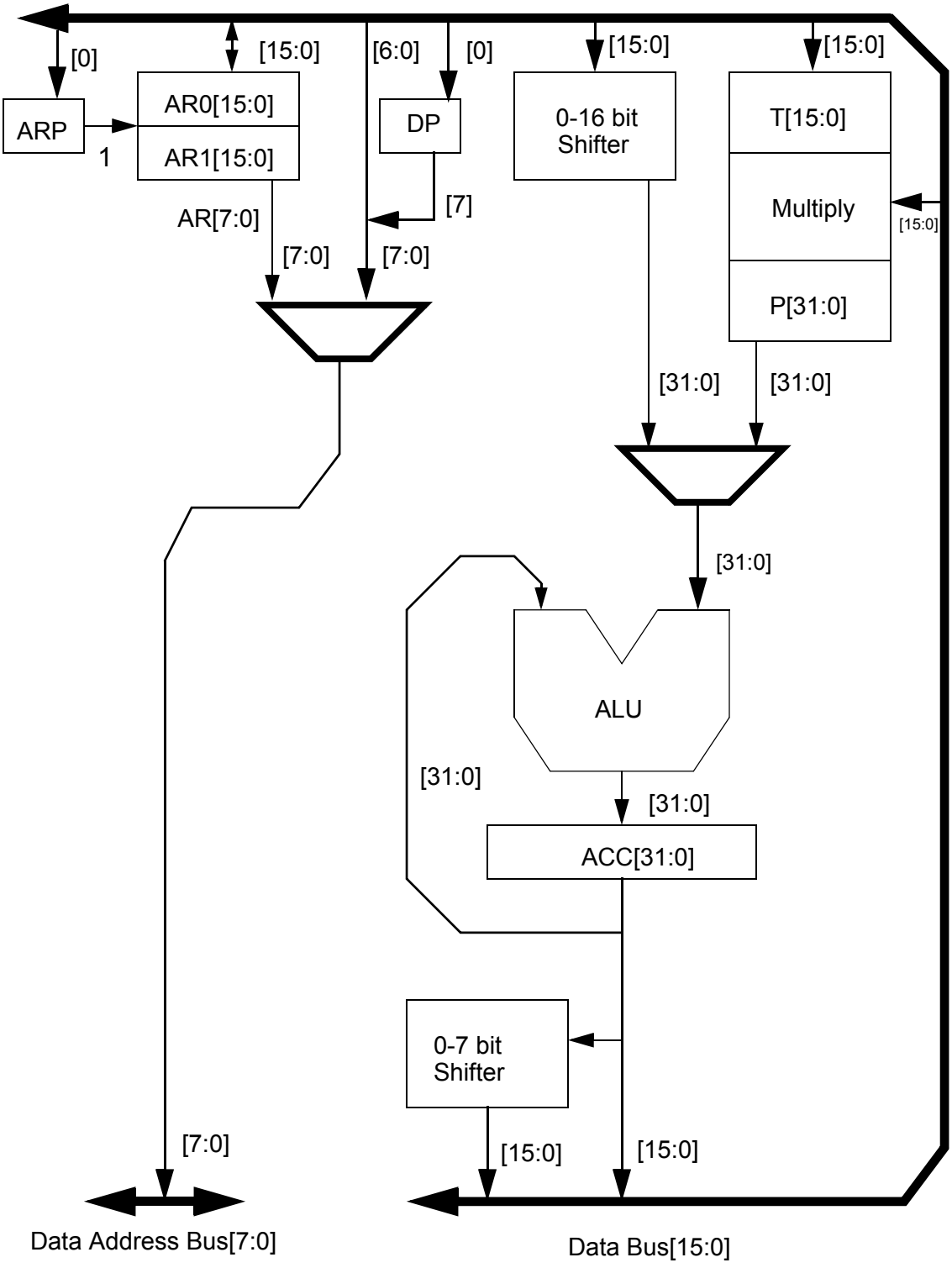


Figure 8-22 shows the DSP data flow.

Figure 8-22 DSP Data Flow



Results Character Conversion (RCC)

Once the DSP has completed the calculation of the signal spectrum, the results are written in block format to the "Results Character Conversion" (RCC) block. Once a block is written, the resulting spectrum is analyzed for DTMF digit content. If a digit is found, the resolved ASCII character representation is written to the Results circular buffer. Once a valid digit sequence is processed, the ASCII character is moved to the ASCII digit register for collection by the host.

RCC is coded as an implicit state machine with the following signal interface:

clk - system clock input

reset - system reset input

address[3:0]- address input

din[7:0]- data input

din_write - data input write input

dout[7:0]- data output

dout_write - data output write output

ASCII Digit Register (DIGIT_REG)

The ASCII digit register is simply a nine (9) bit register for holding the current 8 bit signal character, plus a one bit toggle flag.

Upon reset, the digit holding register is set to 0xff, and the flag is set to 1.

The DIGIT_REG has the following signal interface:

reset - system reset input

clk - system clock input

digit_in[7:0]- digit input

digit_out[7:0]- digit output

flag_in - digit flag input

flag_out - digit flag output

Memory Map

(data space)

0x00 - 0xff-> tdsp program memory (256 bytes)

0x00 - 0x7f-> data sample memory (128 words)

0x80 - 0xdf-> data scratch memory (96 words)

0xe0 - 0xef-> results character conversion (16 words)

(port space)

0x00 - 0x07-> misc. control (8 words)

0x00-> select dma to generate address bit 7

0x01-> select tdsp to generate address bit 7

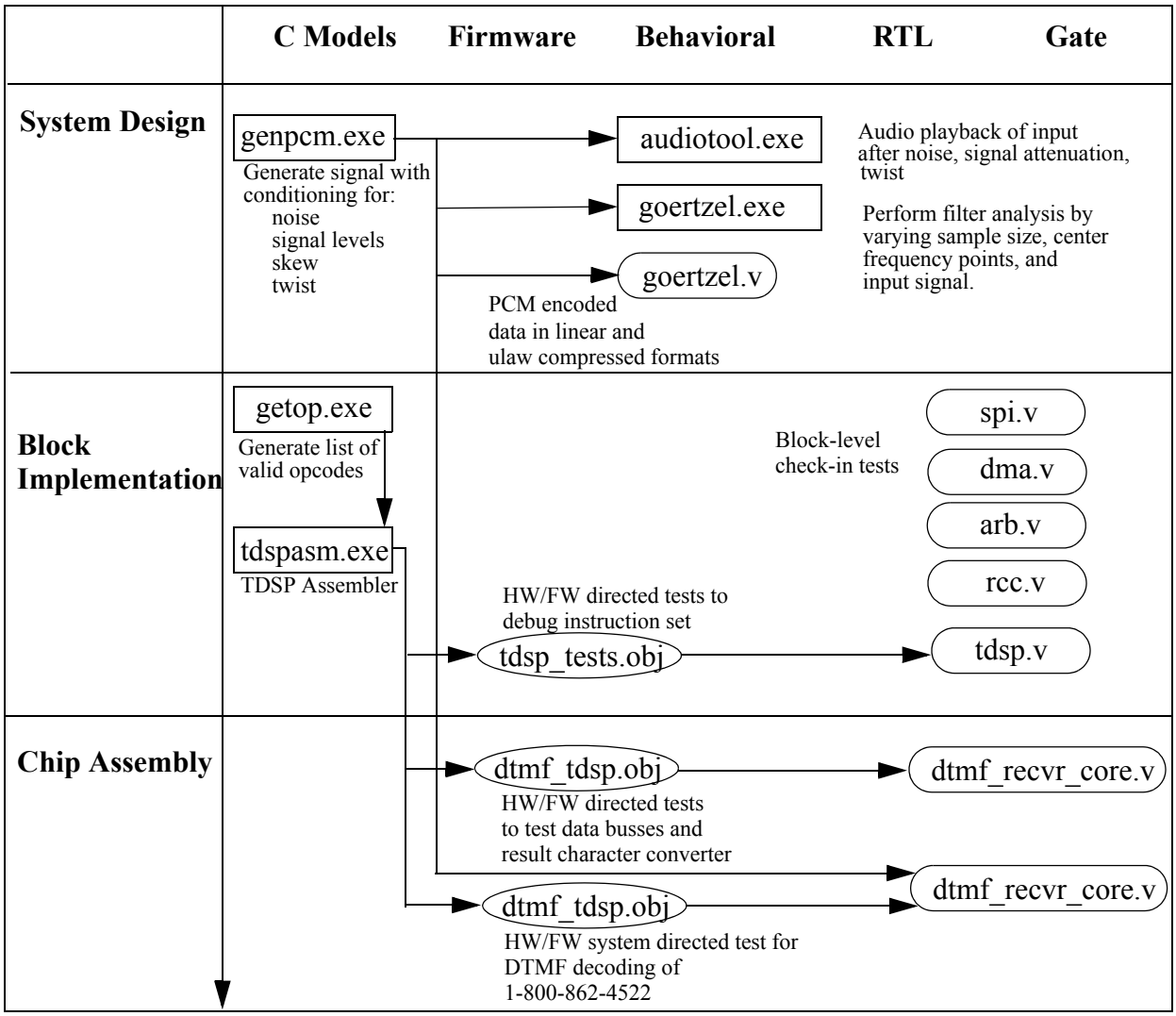
0x02-> tdsp select lower data sample buffer

0x03-> tdsp select upper data sample buffer

Design Verification Strategy

The verification strategy for the DTMF is depicted in Figure 8-23.

Figure 8-23 DTMF Verification Strategy



The model development plan and design verification plans are developed together. The design verification plan defines the overall strategy for verification through all stages of implementation. Architecture and Implementation Verification tests (AVTs, IVTs) are defined in detail in the plan. For AVTs and IVTs tests are further defined as either check-in,

directed, and system application tests. Verification milestones and development schedules are completed.

During the system design phase, the system was modeled in ‘C’ (goerzel.exe). This model analyzed the performance of the algorithm given a input sample stream of varying amplitude, signal skew, and noise levels. The input sample stream was generated by the genpcm.exe utility. In addition, the system model also determined the optimal length of the transform and desired filter center frequencies. A behavioral model was then written for the system (goertzel.v) to explore various architectural structures for computing the transform.

During the block implementation phase, all the major design blocks were verified using integration verification tests (check-in). The design modules verified were; spi, dma, arb, and rcc. Verification of the TDSP required assembly language routines to debug the instruction set. Two utilities were generated for this; getop.exe and tdspasm.exe. The utility getop.exe was written to parse the TDSP rtl code and generate a listing of currently implemented opcodes. This listing was read by the assembler (tdspasm.exe) as a consistency check so that only valid opcodes could be assembled.

At the chip assembly level, software test were written both for direct testing of bus interfaces and full system simulation of signal detection for then phone number 1-800-862-4522 (Cadence Design Systems voice mail network). All the utilities (except for audiotool.exe) were written by the DTM F design team.

High Level Floorplanning

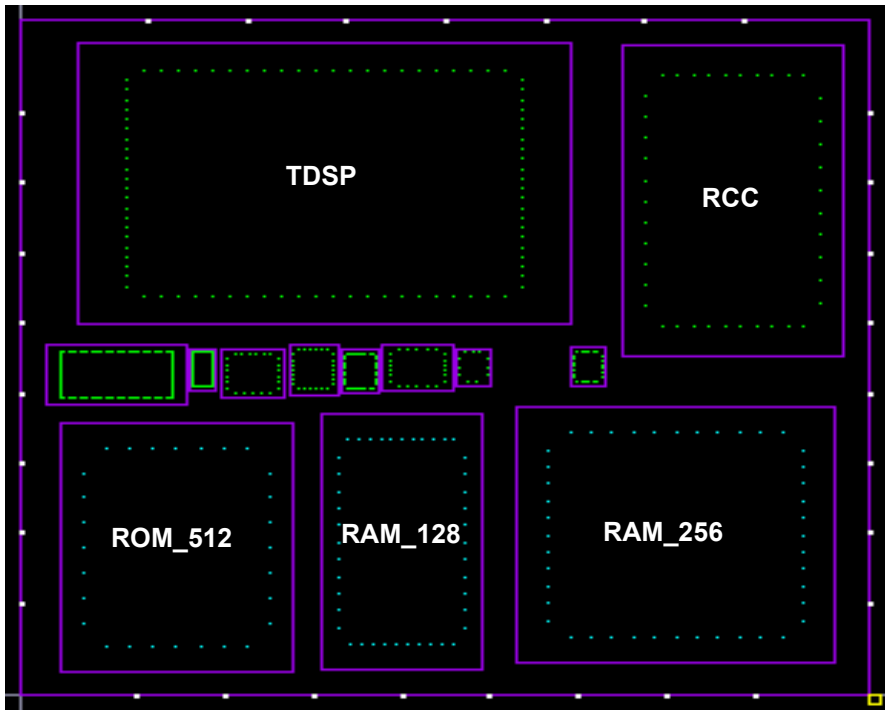
High level floorplanning or logic design planning should be done as early on in the design process as possible. Very little physical information is know at this point other than number of I/O's, number and size of macro blocks, and some approximate sizes for the core blocks.

Initial synthesis runs (with little or no constraints) can be run to get some initial size estimates for blocks and aid in region creation and placement. It is also important to understand the connectivity of the core blocks so that decisions can be made to combine blocks into larger regions without causing long routes between regions.

For the DTMF, an automatic block placement was done at the top level to get an initial block locations based on connectivity assessment. Blocks for each top level module were generated based on gate size estimates from initial synthesis runs.

The initial high level floorplan is depicted in Figure 8-24.

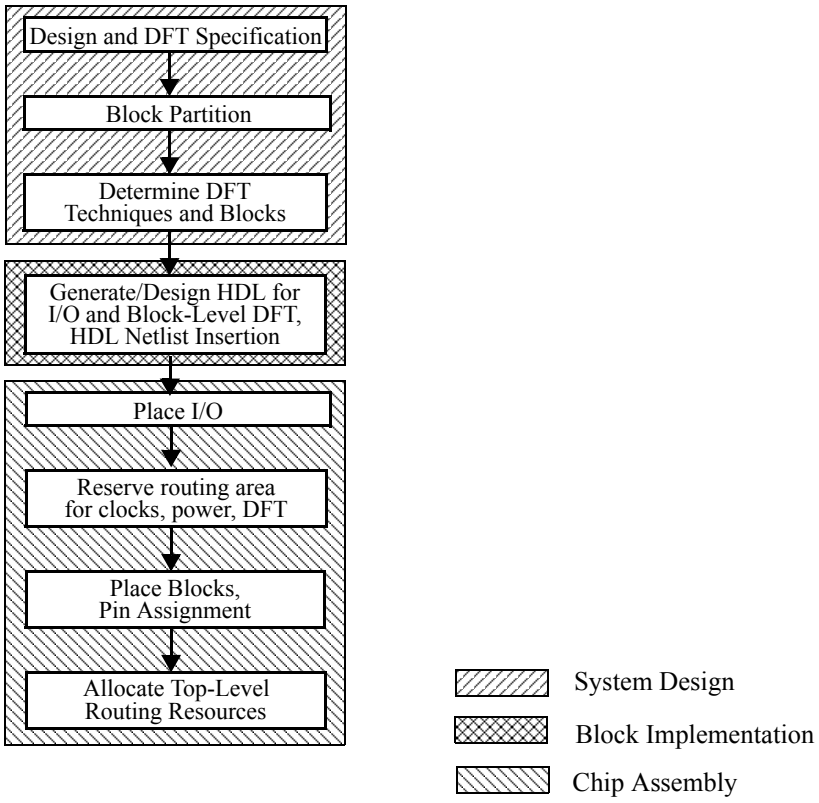
Figure 8-24 High Level Floorplan



DFT Planning and Specification

DFT planning and specification are necessary to insure that testability is taken into account early in the design cycle. The objective of planning is to assure meeting the DFT objectives of the design and to help prevent unnecessary design iterations, or band-aid solutions for test, late in the design cycle. DFT planning and specification need to occur during the system design phase of the design process and implementation follows during the block-level and chip assembly design phases. Figure 8-25 shows the DFT design flow in which these steps take place and the following sections explain the steps of the flow.

Figure 8-25 DFT Planning, Specification and Implementation



DFT Strategy and Testability Analysis

The system design specification must include up-front input for the DFT specifications of the design. The definition of test requirements at both the system and chip level is based on a number of factors including device cost (area and performance impact), vendor requirements, schedule impact, and overall system testability.

A high-level DFT specifications document should be written, which describes the intended test strategy for the overall system design. Also, a set of product specific DFT design rules and guidelines, similar to those in Chapter 4, “Design for Test Methodology”, should be developed at this point in time.

Once the DFT specification is complete, each top-level block, and the ASIC design overall, should be analyzed and a preliminary design review done against the DFT rules and guidelines that were developed. If any blocks require a specialized DFT technique, for example RAM BIST, then a separate specification for part of the design should be written.

DFT Design Considerations

After the initial testability analysis is complete and all DFT specifications are understood, the designer can then determine the DFT structures that need to be added to the design. The chip’s physical block partitions can then be determined, where any additional DFT related blocks that are needed (e.g., a TAP controller or BIST controllers) in the design are included in the top level blocks.

Other issues related to the overall design that are effected by DFT also need to be considered at this point. For example, for internal scan and boundary scan, consider whether specially designed logic cells will be required. For internal scan, either scannable DFF cells can be used or in some cases other logic cells are added to a non-scan DFF (e.g., a 2-to-1 multiplexor) to make the DFF scannable. For boundary scan cells from the chips core logic can be used to implement the require 1149.1 logic, or special I/O cells, with the 1149.1 boundary scan logic included as part of the I/O cell, may be available. The later implementation is more efficient in terms of area and may have better performance.

Also, DFT must be accounted for in terms of routing area (e.g., internal scan chain nets) and it must be included in the budgets of any design constraints (i.e., timing and area). Any additional I/O pins which are used exclusively for test should also be determined. For example, the 1149.1 TAP controller will require 4 to 5 dedicated pins, TMS, TCK, TDI and TDO plus the optional TRST pin. (e.g. 5 for 1149.1).

All of the above considerations should be well understood, as they will effect floorplanning the top-level of the chip and synthesis of the blocks later on in the design cycle.

Tester Resource Considerations

During the planning stages it is also important to consider the target chip tester needs for the design. A tester must be allocated that can handle the intended design. Some of the following tester characteristics need to be considered when selecting the target ASIC tester:

- Number of tester channels

  The tester should have enough tester channels to control each of the I/O pins on the ASIC. If not, then full broadside testing will not be possible.

- Tester's timing performance

  If at-speed functional tests are required, the tester must be able to apply data to the chip at the systems functional clock rate. Edge placement accuracy may also be an issue here and for characterization of high speed I/O - for example if signals on the chip under test need to switch faster than the target tester will allow.

- Tester's vector capacity and capability

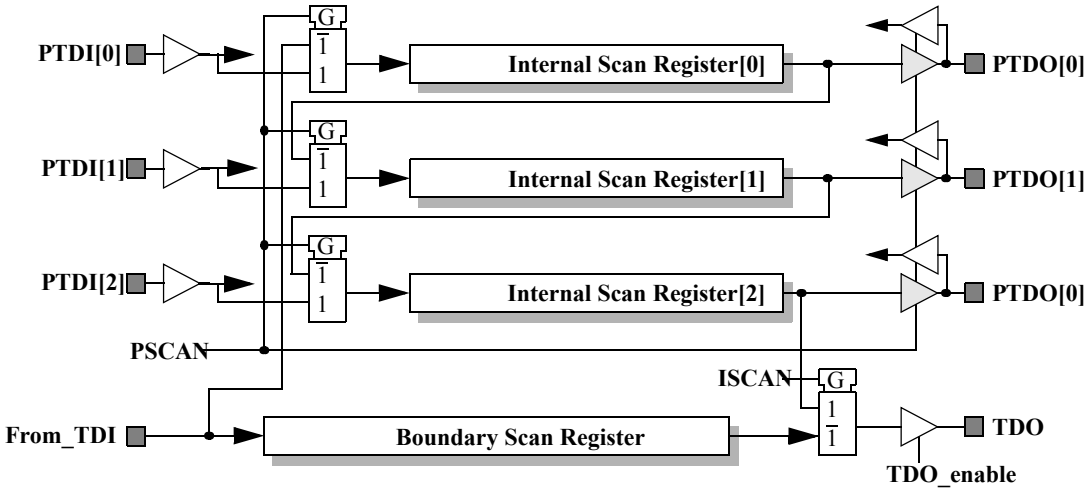
  Estimate the initial size of the test set and make sure that the target tester has enough vector memory to hold the entire test suite, re-loading tester memory is very time consuming and will thus add substantially to the test time of the chip and therefore its manufacturing cost. Also understand if the tester will allow multiple timesets (see the "Tester Formatting and Hand-Off" section on page 11-33) and if the tester can switch timesets without added test cycles (i.e., "on-the-fly"). This is particularly important if functional tests from simulation are used, as the timing the tester will need to handle in this case is often fairly complex.

Also, both design related constraints and tester resource constraints need to be considered in the DFT planning process. Whereas design related constraints may be used to determine allowable DFT area overhead or timing constraints (e.g., test clock frequency), tester resource constraints may be used to help determine DFT features required for end product testing. One important consideration here is providing ASIC tester support for large production test sets. A large test set often is the case for large full scan designs, with several thousand scan elements. The problem arises because ASIC testers typically do not have the necessary support for scan vector application. Those that do often require expensive hardware options

and special formatting software. In most cases, the tester memory is organized for broadside test application, which does not allow for efficient use of the testers vector memory in the case of scan tests. When scan tests are converted to a broadside format for the tester, each bit of scan shift will require a single tester cycle, and therefore take up one bit of vector memory depth per scan shift.

Figure 8-26 shows a method to partition internal scan paths, such that several scan chains can be shifted simultaneously, in a parallel fashion. This reduces the number of shift clock required to load/unload the scan chains of the chip and thus reduces the number of locations of vector memory in the tester required to load a scan vector in a broadside manner.

Figure 8-26 Configurable, Parallel Access Scan Paths



In order to partition the internal scan paths into parallel chains and plan the top-level scan chain configuration of the design, it is necessary to get an accurate estimate of the number of scannable flip-flops in each block. This will be used to determine how many parallel internal scan chains would be needed and which of blocks scan chains could be concatenated together in order to balance the lengths of the parallel chains. This also helps in determining the test pin assignments for each of the top level physical blocks, which will be needed for top-level physical floorplanning and block level synthesis.

In the case of the DTMF, three scan chains will be implemented given some negative edge devices in the results converter block as well as some internally clocked elements. Each chain has a separate scan_input and scan_output but they all share the scan_enable. Implementation will be discussed further in Chapter 11, “Chip-Level Assembly Implementation.”

References

Exercises

8

Logic Synthesis Methodology

Synthesis: The Challenge

Synthesis is a complex, constraint-driven technology, and the quality of the results depends largely on the expertise of the designer who creates the synthesizable model and understands how to constrain the tool to achieve the desired results.

The quality of the results also depends on the accuracy of the timing and load information used to drive the synthesis process. High-level floorplanning tools can help predict timing and load delays due to the location and aspect ratios of physical placement regions; this information is useful in driving synthesis during the implementation phase of the design process. Backannotated wire load information from layout tools, if available, can also effectively drive the synthesis process steps during critical path resynthesis and drive optimization. However, the designer specifies most of the input to the synthesis tool, such as timing, load, and resistance constraints along with the synthesizable model. Thus, an effective modeling style and well thought out contraint budgeting are imperative to the synthesis process.

Deep sub-micron effects are creating the need for a more highly coupled integration between floorplanning tools and synthesis tools. This adds to the complexity of the input data to the synthesis process and puts more burden on the designer to make sure the synthesis tools is working with an accurate and complete data set. Increases in design size and complexity are also contributing to paradigm shift in traditional synthesis technologies by decoupling the synthesis and optimization processes to accommodate for differing optimization strategies for control logic and datapath design as well as size, speed, and power.

All these factors contribute to synthesis begin a potentially time-consuming process. In order to minimize the time spent in the implementation phase of the design process, it is necessary to have designers with an excellent understanding of the technology, and a well-defined strategy for using the tools.

Goals of a Synthesis Methodology

The goal of a synthesis methodology is to produce a structural HDL netlist that

- Meets system requirements in terms of functionality, timing, and area
- Is implemented with optimum technology-specific or vendor specific parts
- Has no timing or design rule violations, either internally or when integrated at the multiblock or full chip level
- Implements the DFT strategy for the design

A set of objectives to reach these goals includes

- Apply the synthesis technology appropriately
- Employ effective modeling style guidelines
- Partition large subsystems appropriately
- Use datapath or module generators
- Synthesize large subsystems from the bottom up
- Select the delay calculation algorithm appropriately

The remaining sections of this chapter discuss these objectives in more detail.

Applying the Synthesis Technology

The logic synthesis process consists of two distinct processes, synthesis and optimization. The synthesis process converts the HDL source code into Boolean expressions through logic structuring algorithms. These algorithms are applied based on the optimization priority, cost (typically area) or timing.

The synthesis process typically performs the following functions

- Modeling style check
- Sequential logic synthesis of registers, latches, and RAMs
- Resource allocation
- Complex operator construction
- Finite State Machine (FSM) construction
- Register collapsing and sharing
- Test logic insertion

The optimization process performs the following functions

- Logic structuring
  - Decomposition and partitioning
  - Technology independent optimization
  - Canonical graph optimization
- Technology mapping
- Critical path resynthesis
- Design rule checks
  - Buffer optimization
  - Maximum fanout transformation

Based on the goals of a particular synthesis run, the designer constrains the synthesis process and selects which optimization functions to perform. By using hierarchy and partitioning, the designer can also set different optimization goals on different partitions of the design by setting specific constraints on these partitions.

Synthesis processes have been specialized to support implementation methods for many different logic structures including:

- Random logic
- Finite state machines
- Datapaths (including random logic)
- RAM/ROM mapping
- Soft cores (parameterized macrocells)
- Hard cores (fixed layout cells)
- Clock trees
- Gate drive sizing and buffer tree insertion
- Scan collar
- Scan logic (partial and full scan)
- Memory BIST
- Boundary scan
- JTAG control

Synthesis tools are best suited to handle control logic and state machines. and should be allowed to optimize freely in this domain to meet the designer's constraints. For example, during optimization, the synthesis tool should be allowed to reencode the state machines.

On the other hand, if the input to the synthesis process is an RTL description containing many datapath operations, the designer might choose a strategy based on the level of abstraction at which the operations are modeled. If the datapath operations are modeled as high-level operands (such as + or *), the synthesis tool can implement the operation using datapath techniques. If the operations are modeled at the equation level (propagate and generate signals for carry lookahead adders, e.g), the designer can prevent the synthesis tool from breaking down the structures and equations further by turning off logic structuring and instead only map the operations to the desired technology.

Another case where the designer may want to turn off logic structuring is when mapping an optimized, technology-specific netlist to another technology or to another vendor's library. In this case, the designer is satisfied that the logic structure of the design is adequate and just needs to be implemented with parts from a different library.

During back-end iterations and Engineering Change orders (ECOs) when the designer wants to maintain the netlist topology, it is possible to further constrain optimization so that it will only resize gates—not add gates—based on loading/timing requirements.

Using Datapath Generators

Traditional synthesis tools were designed to implement complex control logic and state machines, where logic structuring and Boolean reduction techniques are very effective. These techniques are not as effective in optimizing arithmetic or *datapath* operations.

Because of this, different methodologies arose for implementing datapath and control logic, and oftentimes schematic editors were used for the datapath logic. This split methodology required that all datapath operations be created as levels of hierarchy so that the hand-built implementation could be “swapped” out later. This is an unnecessary burden to place on the designer as well as an unneeded data management issue.

With the advent of new synthesis technologies, a designer can aptly experiment with many different architectures at the chip-level, with the added advantage of being able to take into consideration the early estimates of effects due to the physical layout. These capabilities when used in conjunction with realistic, parasitic effects for DSM (using custom regional wire models for interconnect delay) can help determine early in the design phase issues such as the appropriate number of pipeline stages and the number of datapath regions. The level of observability into which particular algorithm is being used by the synthesis engine needs to be guided by the designer. For instance, the synthesis engine can infer which implementation of an adder for example to map into and “add” function. However, such inference mapping is very difficult to debug. The designer must have the skill set necessary to view the synthesized results in layout; and match the corresponding instances from a gate-level netlist to the net names from a higher level, RTL description. This requires knowledge of the synthesis, floorplanning, and placement processes.

Some synthesis tools couple their ability to recognize high-level complex operations with macro libraries to address the problem of implementing datapath logic.

A macro library is typically one of three types:

- Parameterized datapath compilation library

  Synthesis maps the datapath logic to parameterized, high-level cells from the macro library. A datapath generator, invoked either during the synthesis process or as part of the back-end process, performs the actual implementation of the high-level cell.

- Parameterized datapath implementation library

Synthesis maps the datapath logic to the parameterized implementation cells. The implementation cells contain the Boolean equations to implement the given algorithm, a wallace tree multiplier, for example. Synthesis maps the equations to the target library thus preserving the regular structure of the implementation.

- Fixed datapath implementation library

  Synthesis analyzes the datapath logic and selects the implementation from a set of fixed bit-width implementation cells. These implementations are technology-specific netlists that have been characterized through synthesis or by hand.

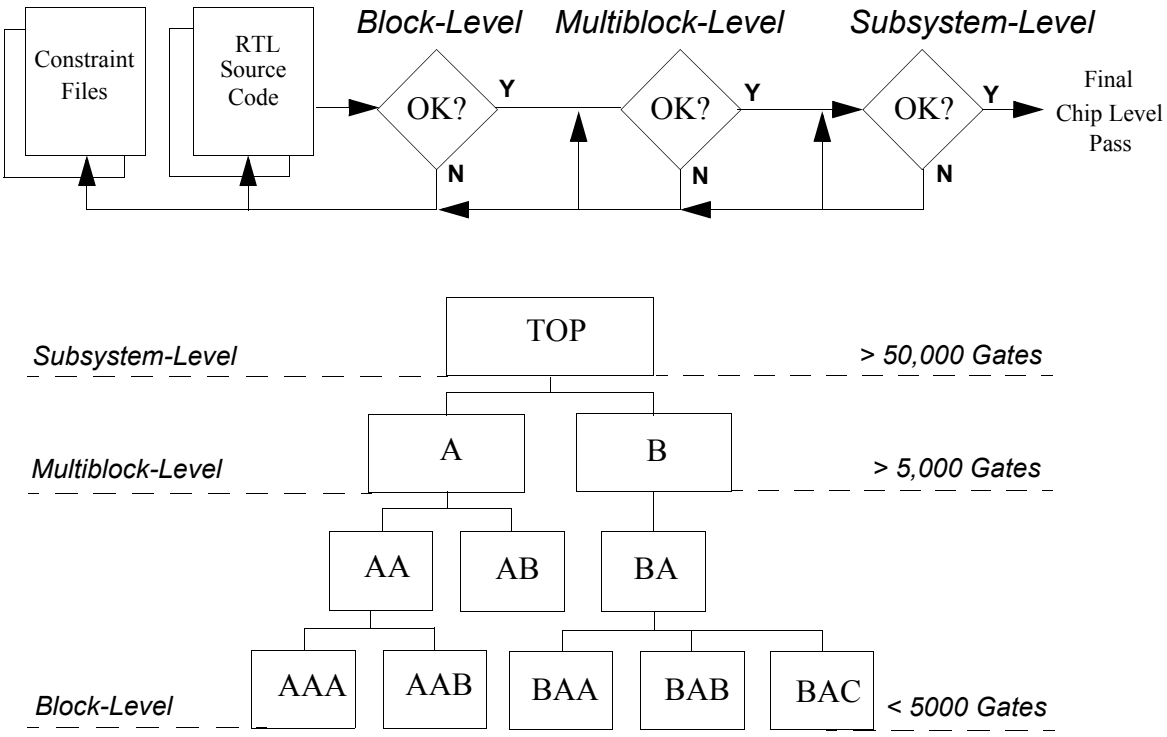
In all three cases, the designer can link modules and functions in the design to specific implementations through user constraints. This is useful for module reuse when the function is too complex to be inferred.

In addition to fixed bit-width datapath operations, embedded blocks such as RAM's can be also be added to the project macro library. This allows the RAM block to be characterized with timing, loading, and area information so that it can be accessed during optimization for accurate timing path extraction and drive buffering.

Synthesizing Large Subsystems

As shown in Figure 6-1, the recommended synthesis methodology is actually a bottom-up process, starting at the block level, progressing to the multiblock level, and then to the subsystem level. At each level, the designer should choose appropriate loading constraints and wire models based on knowledge of the physical design process. Once all the modules at a given level have been completed and the constraints have been met, the next level can be started.

Figure 6-1 Synthesis Process



Block-Level Synthesis

The input to block-level synthesis is an RTL description.

Figure 6-2 shows how to constrain synthesis and optimization during block-level synthesis.

Figure 6-2 Block-Level Constraints

Constraints	Description
Wire model	Should be based on the expected size of the physical placement region, derived from the high-level floorplan. If floorplanning was not done, then knowledge of the vendor's floorplanning tools is essential in wire model selection. If no knowledge of the physical domain is known, then a wire model that reflects the overall gate count of the design block should be used. If the block contains datapath operations, a less conservative wire model can be used. This model would anticipate its 'next nearest neighbor' interconnect, a strategy used in datapath tiling.
Boundary load conditions	Should be based on the physical placement region. If this block maps to a top-level physical placement region, then the boundary conditions should be derived from the subsystem-level wire model.
Timing	Set the clock period, input arrival, and output required times.
Synthesis	Set specific resource-sharing constraints based on the results of earlier synthesizability checks. Set specific FSM state encodings based on known design requirements. Set specific module generation (implementation) constraints based on known design requirements. Otherwise, use defaults.
DFT	Set constraints to insert internal scan registers into the block, and define test clocks, scan control signals, and scan data signals.
Hierarchy	None.
Priority	Select cost (usually area).
Optimization	None.

The designer should analyze the results and consider making the following adjustments, if required:

- Set input arrival and output required times for individual critical paths, if the delays are too long.
- For datapath intensive circuits, use module generation for adders and subtracters.
- If the gate count is greater than 5000 gates, consider partitioning the block further.
- If timing constraints are not met, make timing the priority.

Multiblock-Level Synthesis

The input to multiblock-level synthesis is a design hierarchy containing optimized netlists from the block-level synthesis phase. The goal of the multi-block level of synthesis is to ensure that timing or loading violations were not introduced when the modules were integrated.

Figure 6-3 shows how to constrain synthesis and optimization during multiblock-level synthesis.

Figure 6-3 Multiblock-Level Constraints

Constraints	Description
Wire model	Same as block-level synthesis phase.
Boundary load conditions	If this block maps to a top-level physical placement region, then the boundary conditions should be derived from the subsystem-level wire model. Otherwise, it should be based on the physical placement region.
Timing	None.
Synthesis	None.
DFT	None.
Hierarchy	Preserve the lower-level hierarchy in the design.
Priority	Select cost (usually area).
Optimization	Turn off logic structuring and technology mapping.

The designer should analyze the results and consider making the following adjustments, if required:

- Set input arrival and output required times for individual critical paths if the delays are too long.
- If timing constraints are not met, make timing the priority.
- To fix timing problems that cross block boundaries
  - For datapath intensive circuits, re-run with RTL source using module generation for adders and subtracters.
  - For large state machines, re-run with RTL source using dual table option.

Subsystem-Level Synthesis

The input to subsystem-level synthesis is the full design hierarchy containing the optimized netlists from the multiblock synthesis phase. Like block-level synthesis, the goal of the subsystem-level of synthesis is to ensure that timing or loading violations were not introduced when the modules were integrated.

Figure 6-4 shows how to constrain synthesis and optimization during subsystem-level synthesis.

Figure 6-4 Subsystem-Level Constraints

Constraints	Description
Wire model	Should be based on the chip die size. Often wire models are worst case (versus pessimistic), so the designer should analyze the wire estimation values before selecting a wire model.
Boundary load conditions	Should be based on the pads in the design. Maximum input loads should be set to the maximum load value of the corresponding input pad. Output loads should be set to the input capacitance of the output pad. If the pads are included in the synthesis run, no boundary constraints need to be specified.
Timing	None.
Synthesis	None.
DFT	None.
Hierarchy	Preserve the lower-level hierarchy in the design.
Priority	Select cost (usually area).
Optimization	Turn off logic structuring and technology mapping.

The designer should analyze the results and consider making the following adjustments, if required:

- Set input arrival and output required times for individual critical paths, if the delays are too long.

Once all the design timing constraints and design rule checks have been met, then the designer adds any remaining I/O ring logic (NANDTREES, pads, JTAG/Boundary Scan).

The designer runs the final design through synthesis timing report mode and checks timing and loading reports. If problems exist, the designer should consider flattening hierarchical blocks or re-running RTL synthesis with updated constraints.

If only DRC problems exist, the designer should examine their severity and consider addressing them later during a resizing run with back annotated wire loads.

Selecting the Delay Calculation Algorithm

Advances in integrated circuit processing have placed increasing demands on more accurate delay modeling to account for non-linear effects at the sub-micron level. Synthesis tools may offer a choice between a standard linear delay model and a nonlinear delay model.

Linear Delay Model

A typical linear delay model is described by equation (3.1.1) for $D_{total,}$ which represents the delay across a gate instance, $D_{propagate}$, plus the interconnect delay, $D_{interconnect}$, to the next gate instance.

$$D_{total} = D_{propagate} + D_{interconnect} \quad (3.1.1)$$

The delay from an input pin to an output pin of a gate instance, $D_{propagate}$, is further made up of three components as shown in (3.1.2).

$$D_{propagate} = D_{intrinsic} + D_{load} + D_{slew} \quad \text{where} \quad (3.1.2)$$

$$D_{load} = k_{load} \cdot L \quad (3.1.2a)$$

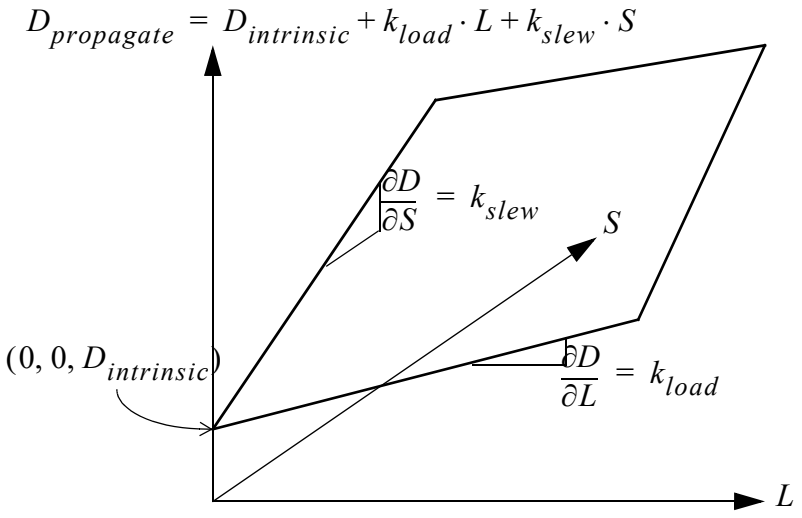
$$D_{slew} = k_{slew} \cdot S \quad (3.1.2b)$$

$D_{intrinsic}$, k_{load} and k_{slew} are constant timing parameters obtained from the ASIC library. They are respectively, the intrinsic gate delay, the output drive factor (driving resistance), and the input slew sensitivity factor. L and S are variables and represent respectively the load (capacitance) seen at the output pin and the slew seen at the input pin which is further determined from D_{load} of the driving gate and $D_{interconnect}$ at the input pin.

$D_{propagate}$ is a linear function of two variables (L, S). As illustrated in Figure 6-5, its equation is that of a plane over (L, S) where k_{load} and k_{slew} are respectively the slopes in the L and S directions, and $D_{intrinsic}$ is the intercept at (L, S) = (0, 0).

$$D_{interconnect} = R \cdot C \quad (3.1.3)$$

Figure 6-5 Illustration of Equation (3.1.2) as a Plane over (*L*, *S*)



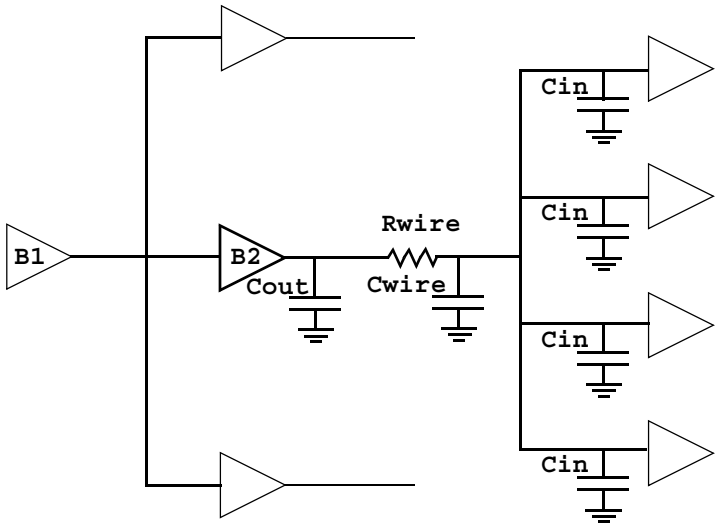
The interconnect delay in (3.1.3) is a product of resistance, R, and capacitance, C, where R and C can be obtained in various ways. C includes pin and wire capacitances. Pin capacitance is a cell library timing parameter. Wire capacitance and resistance can be estimated based on the wire model provided as part of the ASIC library. In the wire model, the nonlinear relationship between wire length and fanout are approximated by piecewise linear functions. Instead of using the wire model to estimate C, designers can back-annotate actual wire loads obtained from physical layout tools.

Figure 6-7 shows the delay calculation for an example circuit, based on the delay parameters shown in Figure 6-6.

Figure 6-6 Delay Calculation Parameters

```
Library Cell Information

------------------------

`timescale 1 ns / 10 ps
`celldefine
module BUF_1 ( Z, A );
output Z;
input A;

buf (Z, A);

specify
specparam cell_area = 567.60;
specparam blocked_tracks = 2.00;
specparam total_tracks = 3.00;
specparam output_cap$Z = 0.50;
specparam input_cap$A = 1.10;
specparam rise_factor$A$Z = 0.020;
specparam fall_factor$A$Z = 0.010;
specparam rise_slew$A$Z = 0.010;
specparam fall_slew$A$Z = 0.011;
specparam max_load$Z = 32.00;

( A +=> Z ) = ( 0.41 , 0.45 );
endspecify

endmodule
`endcelldefine


Wire Model Information
----------------------
Resistance/Unit Length = .000002 ns/pF*micron
Capacitance/Unit Length = .002000 pF/micron
Wire Length [4 Fanouts] = 3500 microns


Previous Stage (B1)
-------------------
Load Delay = 0.210 ns
```

Figure 6-7 Example Circuit




```
Rise Delay Calculation (B2)
--------------------------
Rwire = [Wire Resistance/Unit Length]*[Wire Length]
      = [(0.000002 ns/pF-micron)]*(3500 microns)
      = 0.007 ns/pF
Cwire = [Wire Capacitance/Unit Length]*[Wire Length]
      = (0.002000 pF/micron)*(3500 microns)
      = 7.000 pF

Total Load = Cout + (ΣCin) + Cwire
      = 0.5 pF+ 4*(1.10 pF) + 7.00 pF
      = 11.90 pF

Rise Intrinsic Delay = 0.4100 ns

Rise Load Delay = Rise Resistance*Total Load
      = (0.02 ns/pF)*(11.90 pF)
      = 0.2380 ns

Rise Slew Delay = Slew Factor*(Load Delay)Previous Stage
      = (0.010)*(0.210 ns)
      = 0.0021 ns

Wire Delay = [Rwire/n]*[Cwire/n]*[ΣCin]
      = [(0.007 ns/pF)/4]*[7.00 pF/4]*[0.4*(1.10 pF)]
      = 0.0135 ns

Total Delay = 0.4100 + 0.2380 + .0021 + 0.0135 = 0.66 ns
```

Nonlinear Delay Model

Equation (3.1.1) for D_{total} still applies in the nonlinear model. What changes is a new set of equations for $D_{propagate}$ as shown in (3.2.1).

$$D_{propagate} = f_{propagate}(L, S) \quad (3.2.1a)$$

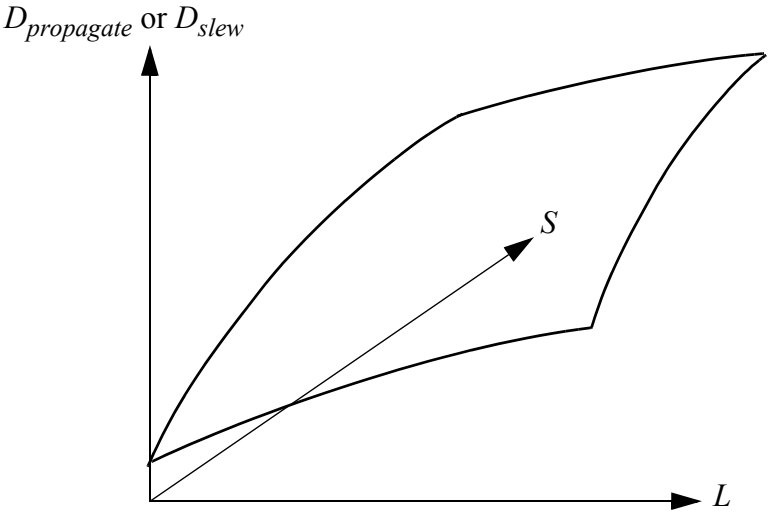
$$D_{slew} = f_{slew}(L, S) \quad (3.2.1b)$$

D_{slew} is the output transition delay. As in the built-in model, L is the load seen at the output pin and S is the slew seen at the input pin. S is further determined from D_{slew} of the driving gate and $D_{interconnect}$ seen at the input pin. In contrast to the linear delay model, recursive input dependency exists in computing (3.2.1b); in other words, output transition delay ultimately depends on transition delays at the circuit's primary inputs.

Functions $f_{propagate}$ and f_{slew} are nonlinear or curvilinear functions of two variables (L, S). Equation (3.2.2) shows an example of such a function from an ASIC vendor for fixed temperature, voltage and process settings. The function is also illustrated in as a curved surface over (L, S).

$$D = k_1 + k_2 \cdot L + k_3 \cdot S + k_4 \cdot \ln L + k_5 \cdot \ln S + k_6 \cdot L \cdot S \quad (3.2.2)$$

Figure 6-8 Illustration of Equation (3.2.2) as a Curved Surface over (*L*, *S*)



In the ASIC library, nonlinear models are described using discrete tables where table entries are sampled values of the nonlinear equation. For example, a $M \times N$ table corresponding to (3.2.1a) will have entries:

$$D_{propagate}[i,j] = f_{propagate}(L_i \quad = 1 \ldots M \quad = 1 \ldots N$$

During delay calculation, values that lie between table entries are interpolated using various techniques.

References

CMOS VLSI Book

9

Timing-Driven Design Methodology

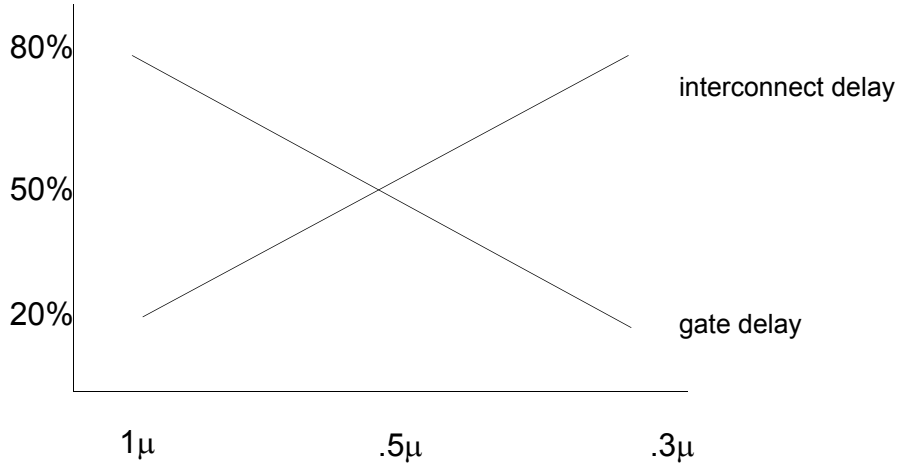
Timing: The Challenge

Because of the increasing density, complexity, and shrinking process geometries of today's designs, the placement and routing or *physical design* of a chip is becoming more complex and less predictable. Synthesis tools today make trade-offs based on the timing or area of various implementations, but they typically have little access to place and route data for the design. As a result, a highly complex synthesized design can be difficult, if not impossible, to place and route. With little access to physical design data, synthesis tools can miss timing goals by as much as 100% based on the estimated versus the actual interconnect.

Interconnect Delay Issues

Because of the increasing number of designs at the submicron and deep submicron levels, interconnect delays due to routing and the placement of logic on a chip are also an increasingly important factor in delay calculation. Figure 7-1 shows the contribution of interconnect delays and gate delays to the total delay in designs at the submicron level.

Figure 7-1 Relative Contribution of Interconnect Delay to Total Delay



Synthesis tools today estimate those interconnect delays based on fanout information and statistical wire models. The wire models characterize the average interconnect delay due to wire length, capacitance and resistance. Due to this limited amount of physical information, the delay estimation is prone to error. It is not uncommon to have as much as 50% error for .5 μ designs and 100% error for .25 μ designs in the timing numbers produced

by synthesis tools. It is important to note that these errors can be in either direction. For example, timing estimates can be larger than the actuals and may lead to overdesign. For designs following design rules above .7μ, this degree of error can be compensated for by most automatic place and route tools. For designs following design rules under .7μ, however, this degree of error significantly reduces the chance that the target circuit performance can be met.

Designs that are difficult to place or route or designs that contain significant timing problems due to interconnect delays can result in costly design iterations if the errors are not discovered until after the entire design has been placed and routed.

A good strategy is to use floorplanning tools to perform preliminary placement of logic on a chip and to estimate interconnect delays based on that placement. This timing information can then be used to drive the front-end synthesis, simulation, and timing tools for more accurate results.

IC Package Issues

For large-scale, high frequency, high pin count designs, IC package contribute to IC design issues which make it imperative that the front-end design team get involved in the early planning of the physical chip and package. This upfront team effort will help to optimize the back-end process in areas such as I/O and macro preplacement, power bus strategy, row utilization, clock buffering, and critical nets timing.

Selecting packages requires an IC designer to make difficult trade-offs between the device's number of I/Os, how much power the circuit will dissipate, the amount of board space the device will require, package price, maximum device operating frequency, and device reliability. These packaging issues will continue to dominate floorplanning activities through the stages of microarchitectural refinements, detailed modeling, and high-level floorplans.

High frequency switching of inputs and outputs can result in separating power and ground planes for inputs, clock, outputs, and core logic to eliminate false switching from ground bounce. This will have significant influence over the floorplan of the design to account for multiple power and ground distribution.

High frequency design also creates other signal integrity issues associated with package inductance and device loads. Study of the specific RLC specifications for signal pins as well as power and ground pins helps determine maximum frequency of the IC in the system environment.

System integration issues of the IC into the board environment is also important. High pin count ICs create large routing congestion problems. Pin locations need to be assigned early in the design to ensure board integration. Initial system floorplanning will be based on establishing pin locations.

Finally, power versus package material (plastic, ceramic, HS, etc.) trade-offs represents the largest IC price decision. The single largest power consumption is based on the clock distribution. Large capacitive loads and high-frequencies can account for as much as 40-50% of the power consumption. Early power planning of the microarchitecture at the high-level systems process is crucial.

Power estimates are essential to understanding the allowable package alternatives and board space required. Package alternatives result in different thermal conditions that should be well understood as it relates to physical package dimensions, cooling requirements and the use of heat sinks. IC device reliability is also based on thermal operating conditions.

Goals of a Timing-Driven Design Methodology

The goals of a timing-driven design methodology is to produce a structural HDL netlist that

- Can be easily placed and routed by the vendor
- Meets the timing requirements after place and route

A set of objectives to reach these goals is to

- Use high-level floorplanning to generate estimated parasitic delays
- Drive synthesis with estimated parasitics
- Back-annotate critical path delay and parasitics from detailed floorplanning into timing optimization, gate-level simulation, and timing analysis
- Perform early delay estimation for timing-critical blocks
- Perform dynamic and static timing analysis

The remaining sections of this chapter discuss these objectives in more detail.

Floorplanning and Placement

Floorplanning tools are graphical tools that allow designers to perform high-level physical design by

- Assigning blocks of logic to particular areas of the chip
- Planning system clock and power distribution
- Placing I/O cells and internal block pins
- Routing top-level buses
- Generating custom wire models

Floorplanning provides useful information about physical design at all stages of the top-down design process. Floorplanning can also perform analysis, including clock tree timing analysis and routability analysis.

Floorplanning is the most critical step in the back-end flow and must be done in unison with the front-end designers. The team will work together starting with developing the functional partitioning and initial IC package plan. In addition the team will look at the requirements driving the chip, to insure they are incorporated in the floorplan design. The floorplanning stage should begin at the start of subsystem model development which follows functional unit partitioning.

The floorplan design and analysis steps results in generating the required control files or script files that will be used later to automate and control the place and route to achieve the desired requirements:

- Die Size requirements
- Row Utilization
- Obstruction and Congestion Mapping
- Constraint Files (Crosstalk, Path, Net)
- Wide wire signal routing control
- Clock tree requirements
- Power and ground distribution and routing control
- Technology/speed requirements

It is essential for the design team to consider all factors, in order to realize the goals of that design, whether they be speed, density, technology choice,

or power consumption.I

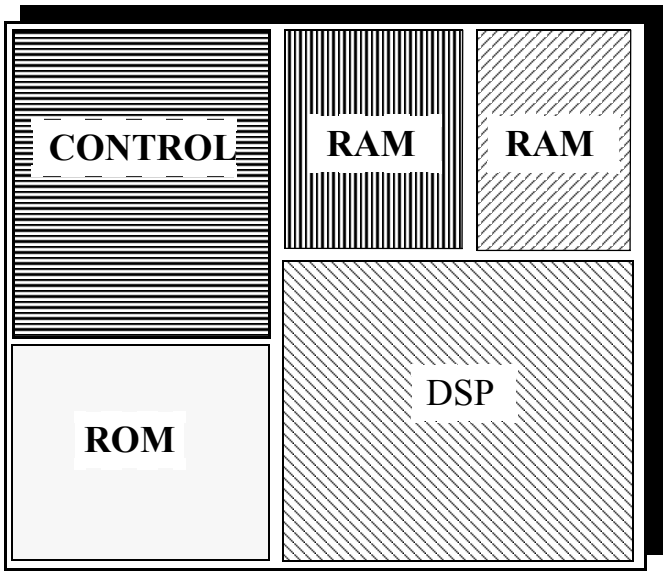
High-Level Floorplans

When a system is going to be implemented in an IC or multi-chip module (MCM), designers can use high-level floorplans generated with floorplanning tools to further investigate the IC package requirements and system partitioning in the physical domain. High-level floorplans define the relative locations of design blocks, global routing resources including clock domains, I/Os pin locations, power and ground distribution, resolve area and timing budgets, generate models for interconnect estimation, and extract boundary condition information. Blocks can be either analog and digital macros, datapath, special cell, or standard cells. Designers use this information to derive synthesis timing and loading constraints.

The goal of high-level floorplanning is to inject estimated block-level interconnect information, region locations, and aspect ratios into the design process as early as possible. These estimates allow the tools downstream to work with more accurate and reasonable information instead of "guesswork," which can lead to multiple back-end iterations, or worst-case estimations, which can lead to overdesign.

Figure 7-2 shows a high-level floorplan.

Figure 7-2 High-Level Floorplan



Detailed Floorplanning

Before delay estimation and timing verification, a detailed floorplan must be created for the overall design. Vendors provide, at minimum, a clustering tool that assigns logic to regions for more accurate interconnect estimates. This type of clustering algorithm is becoming inadequate for deep submicron designs because accurate delay information is not known until a full placement and global route are performed. Without this level of accuracy early on, timing analysis and design rule checking can be misleading, leading to expensive iterations during place and route. Because of the inadequacy of the clustering approach, vendors are beginning to require a placed design as the signoff medium, rather than the traditional netlist.

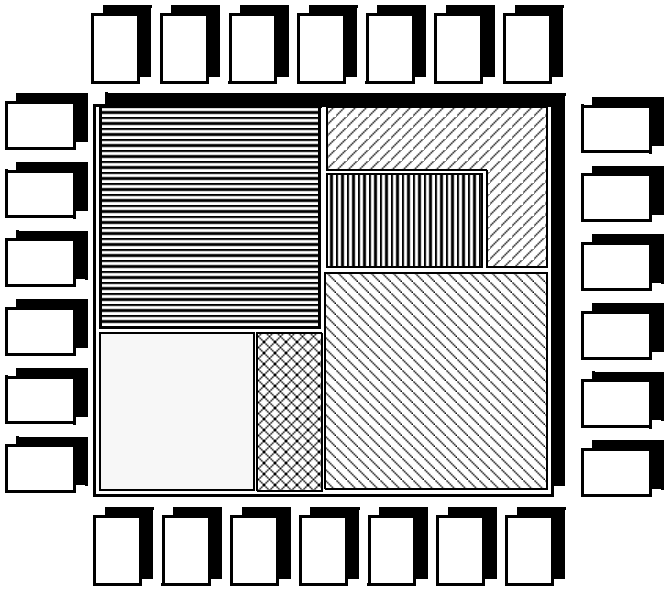
A difficulty with the placement requirement is that the detailed floorplan is created before delay estimation and timing verification (when the degree of accuracy is less than 5%) and therefore the accuracy of the RTL optimization is suspect. Thus, it is recommended that designers perform a quick placement and timing analysis to determine the timing-critical regions and then iterate through synthesis if corrections are needed in these regions.

Some vendor tools include the floorplanning job in the delay estimator and design rule checking tools. This leads to much more accurate delay and load prediction. At this point, I/O location should also be known and included as part of the floorplan.

The finished floorplan will provide obstructions for the placer for any preroutes that have been created during the floorplanning process. For example, any power/ground rings or stripes will have been inserted during the floorplanning step so that the placement tool can avoid placing cells in areas that would cause shorts.

Figure 7-3 shows a detailed floorplan.

Figure 7-3 Detailed Chip Floorplan



Timing-Driven Synthesis

Obtaining the most timing-accurate results from synthesis requires

- Setting accurate timing constraints
- Using the correct wire model
- Setting accurate load constraints
- Using estimated parasitics from placed data

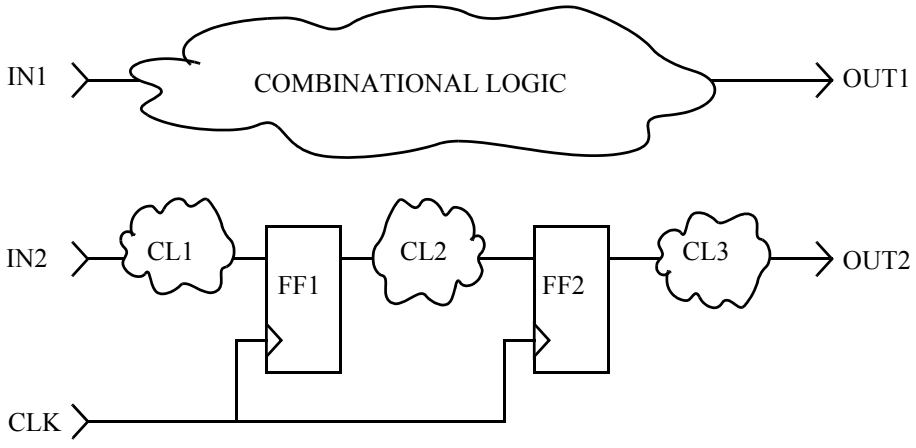
Accurate Timing Constraints

Timing constraints specify signal arrival times and required times and define clock signals, clock periods, and clock skew.

Sequential arrival and required times are set relative to the first edge of the clock. In other words, sequential inputs are clocked by the first edge and arrive after some propagation delay. Sequential outputs are also clocked by the first edge of the clock and arrive after some delay. The required time constraint requires the synthesis tool to make sure the propagation delay does not exceed the required time. Combinational arrival and required times are set relative to one another (required time must be greater than the arrival time). Internal register to register to delays will be constrained by the clock period and the setup checks.

Figure 7-4 shows the basic timing paths through a module.

Figure 7-4 Basic Timing Paths

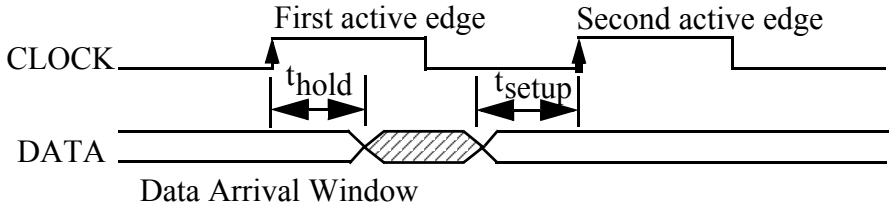


Synthesis tools compute timing paths in the following manner:

- Paths are traced using cell polarity information (inverting versus non-inverting cells). Worst-case is used for non-deterministic cells (xor, xnor, for example).
- Paths originating at storage devices include the clock-to-output delay.
- Clock paths are assumed to be ideal (no load delay is included).

Setup and hold calculations are based on the timing diagram in Figure 7-5.

Figure 7-5 Timing Checks



A clock constraint must be set on all clock nodes. This can be a module input or an internal node. The clock constraint should be set so that the first edge is the active one. This simplifies the setting of arrival times. Skew can be added before and after each clock edge.

It is important to understand the definition of clock skew in the context used by synthesis tools. For purposes of nomenclature, let's describe various clock skew definitions. Also, for simplicity, assume a synchronous positive edge-triggered clocking methodology, although the definitions here apply to most any clocking scheme. The following two diagrams will be used as the basis for discussion.

Figure 7-6 Clock Distribution

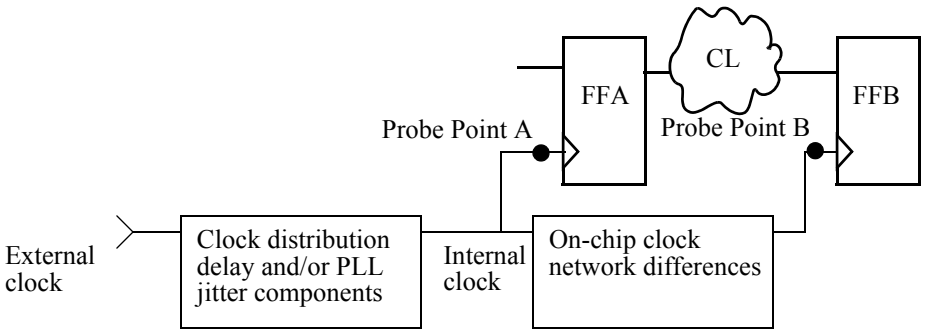
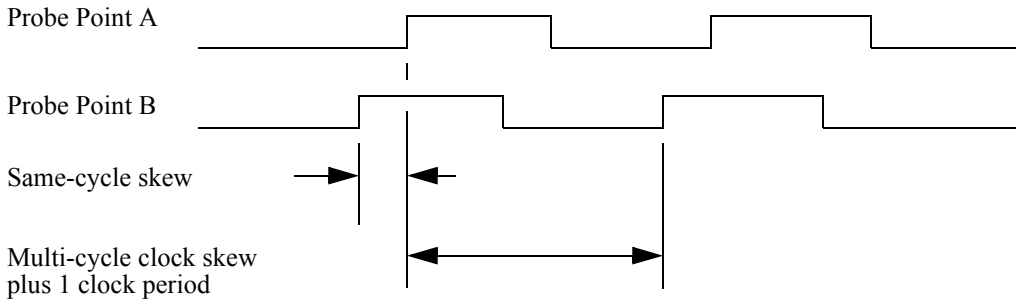


Figure 7-7 Clock Waveforms



Same-cycle/on-chip clock skew

This skew is the worst-case difference of clock arrival times between any same-chip flip-flops that drive each other on the same clock cycle. A diagram of the circuit is shown in Figure 7-6 and a waveform is shown in Figure 7-7. FFA drives FFB through some combination logic cloud, which could minimally be a wire. The probe points are located physically at the FF clock input pins. The skew difference between these points is shown by the box labeled “On-chip clock network differences.” Ideally, this skew would be used in the minimum path timing calculations, but it is not usually supported by synthesis tools. Also, since the difference between this skew value and the different-cycle/on-chip skew value is usually small, it is generally not used.

Different-cycle/on-chip clock skew

This skew is the worst-case difference of clock arrival times between any same-chip flip-flops that drive each other over one clock cycle minus the clock cycle time. This is shown in Figure 7-7. It is this skew that is used by synthesis tools. The major components of this skew are because of variations due to the clock distribution network on the chip and cycle-to-cycle jitter introduced from the chip clock input signal variations and/or on-chip PLL variations. Note that the internal clock is generally distributed after the PLL. It is also important to note that cycle-to-cycle jitter is generally much less than the long-term, or multiple-cycle, jitter typically quoted on PLLs. For on-chip PLLs, the ASIC vendor should spec both cycle-to-cycle jitter and long-term jitter. For synthesis, divide the different-cycle/on-chip skew value by two and use this new value for both the “plusskew” and “minusskew” variables.

Different-cycle/off-chip clock skew or system clock skew

This skew is the worst-case difference of clock arrival times between any two system flip-flops that drive each other on different clock cycles minus the clock cycle time. This is the clock skew that should be used when computing the system timing. It includes skew introduced by the board-level clock distribution network, the differences in the on-chip clock distribution delays as referenced from the clock input pin(s), and any jitter components introduced either on-chip and/or by the board.

Wire Models

Synthesis tools in general typically choose a wire model based on the size of the current block being synthesized. It is recommended, however, to explicitly specify *one* wire model for each synthesis run based on the size of the target physical placement region. This requires knowledge of the vendor floorplanning tools and physical design methodology.

Accurate Load Constraints

Synthesis tools perform one or more of the following design rule checks during optimization:

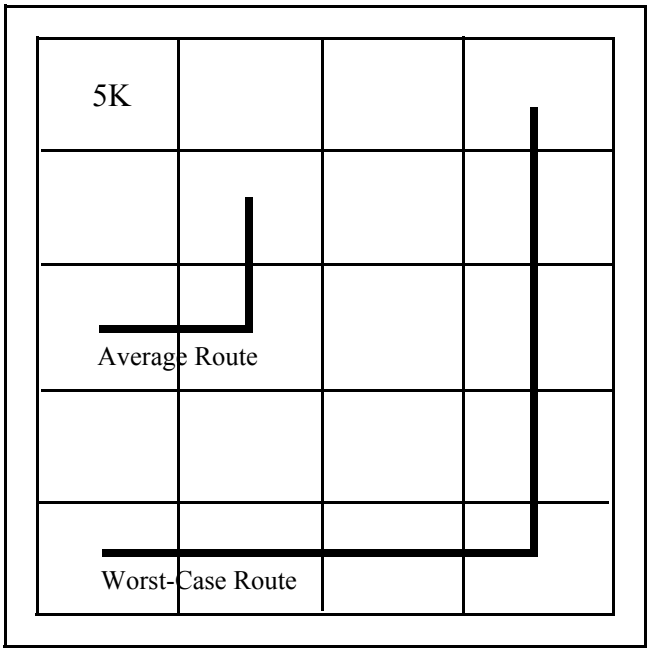
- Maxload <= Cout + ΣCin + Cwire
- Maxfanout <= ΣFanin
- Maxtransition <= [Cout + ΣCin + Cwire]*$\text{DriveFactor}_{\text{Maximum}}$

Typically, either maxload or maxfanout is used (not both). Maxtransition is optional. Synthesis tries to satisfy all rule checks, if possible.

To ensure that final synthesized result passes vendor design rule checks, it is imperative that proper module boundary loading conditions and wire models are used.

For example, if the vendor floorplanner partitions the design into 5K gate blocks in a 100K base array, the wire load of the top level block interconnect must be included in the boundary loading constraints of the lower-level modules.

Figure 7-8 ASIC Gate Array Floorplan



Wire models model worst-case routing. That means that the outputs of modules that cross physical design partitions must be able to drive *at least* a worst-case load. Therefore, output load constraints need to be set accordingly when these blocks are synthesized. This ensures that a buffer can be inserted at the top level during drive optimization.

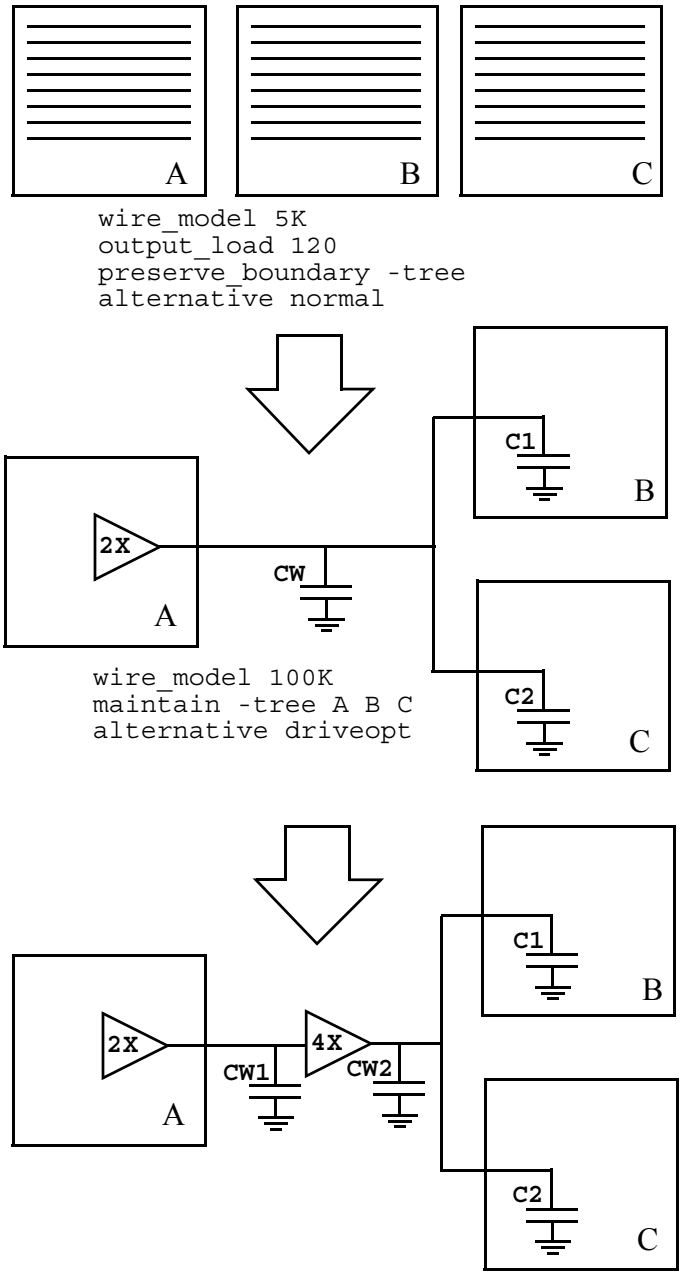
Loading constraints specify the boundary conditions of the module. The designer can specify the drive strength (resistance) of an input along with the maximum load of the gate driving that input. This approach allows inputs with large internal fanouts to be appropriately buffered and accounts for the wire and load delay from the input port to the fanout device(s). Similarly, on the output, loading constraints ensure that the module outputs are properly buffered.

Some general guidelines for specifying these constraints are:

- Input resistance should be set to the resistance of a standard gate (NAND2, e.g.).
- Input maximum load should be set to be less than the maximum load of a standard gate (NAND2, etc.) plus the wire load of the estimated number of fanouts.

- Output load should be set to be greater than one wire load (use appropriate wire model) plus one standard load. Depending on the library, this can be increased.

Figure 7-9 Drive Optimization



```
Cwire (1 fanout) = 120 pF (100K wire model)
Cwire (2 fanout) = 200 pF (100K wire model)
C1 = C2 = 1 pF (Standard Load)
Max Load (2X Buffer) 128 pF
Max Load (3X Buffer) 192 pF
Max Load (4X Buffer) 256 pF
```

Modules A, B, and C were each synthesized separately with an output load constraint of 120 pF. This was used so that each module output could drive at least a wire load (estimated using 100K wire model) between blocks at the top level. As a result, a 2X buffer was inserted (maximum load = 128 pF) to drive the output. A 5K wire model was used so that *internal* interconnect of each of the blocks would be properly driven.

The lower level netlists were consolidated and a top-level drive optimization run was done using a 100K wire model. Because CW had a fanout of two, it was estimated to be 200 pF. Since the 2X buffer was overloaded, a 4X buffer was inserted to drive the 200 pF load (CW2). Since the 2X buffer was driving the output of module A, it could sufficiently drive CW1 and the standard load of the 4X buffer (121 < 128).

It is important to understand that if the output of module A was driven by a standard gate with a maximum load of 32 (for example), the insertion of the 4X buffer could not be done without causing a DRC error within module A. This would have required another iteration to resolve.

Estimated Parasitics

It is recommended to drive synthesis with parasitics (estimated capacitance and resistance from floorplanning), if that information is available.

Synthesis tools can read in a Standard Parasitics Format (SPF) file for timing-driven synthesis. Synthesis takes the extracted interconnect capacitance from the SPF and places a constraint on the output port of the selected cell.

Placement and Route

Placement

The success of placement is dependent on two factors; the quality of the floorplan and the accuracy of the constraint files. The floorplan must address each of the critical requirements driving the design in order for the placer to achieve a routable placement that will meet the timing goals of the design. The constraint files must constrain the right nets with appropriate constraints without overstraining the placement tools. The histogram of net delays generated after synthesis or timing verification should be used to determine what percentage of the total nets needs to be constrained and what the constraints could be.

Several placements can be run simultaneously with different constraint set-ups to achieve the design goals. After an acceptable placement is achieved, the special control files generated from floorplanning would be executed. Again, these files would be used to automate and control clock tree synthesis, wide wire nets and other special requirements. After each step in the placement process, extraction and back-annotation to synthesis and timing verification would be performed. The accuracy of both the estimated and extracted data is vital to guarantee early detection and potential timing problems. The recommendation here is to back annotate to a hierarchical gate level block for the logic team to simulate. This will include a new netlist, if for example, clock tree synthesis was implemented.

Route

At this point in the process, all floorplanning has been completed and all macro blocks and standard cells have been placed,. During the routing phase of the design process, the designer will need to complete any special routing that was not prerouted during floorplanning. Special routing control is typically required for routing of supply, power, and clock nets. The clock net tree configurations will have been created and placed or during automatic clock tree synthesis. Routing proceeds using a balanced router to meet the target delay and minimize the skew between the leaf cells at each clock level and between levels.

The success of the routing step is again dependent on the quality of the previous steps. A placement which has taken routing resources and congestion into consideration will be much more likely to produce a routable design than one that has not. There are several facts that should be taken into account during the routing step depending on the requirements for the design. Examples of these would be crosstalk variables, number of vias used, wrong way wiring, critical nets, wide wire nets, extra spacing nets, phan-

tom logic, and clock tree buffering.

The power routing preroutes that were done during floorplanning would include any power rings that go between the design IOs and stripes through the cell placement area. The macro blocks and standard cells will now need to be connected to those power pre-routes. This will be done with a combination of automatic and manual routing depending on the types of power pins and the desired connection points.

After the special routes are complete, global routing will be run against the signal nets to allocate the routing resources available in the design. The global route function will create a coarse regular wiring layout based on obstructions resulting from special wiring, clock wiring and placement. The global route function will look at the entire design and work to balance wiring congestion with minimizing wire lengths. Detailed or final routing will be created next for the signal nets based on the plan created by the global router. Metal layers and vias will be assigned to each net based on the plan created by the routing layers descriptions in the technology used by the designer. The final route function will attempt to complete all signal connections, reroute some wiring to reduce the number of vias and optimize wire lengths, check for pins with multiple ports and reroute wires to alternate ports to improve results, move wires to account for blockages in cells or a cover macro and do some wrong way routing to avoid floorplan obstructions.

The detailed router will be run in repair mode to remove any DRC violations that may have been created during final routing. Final clean-up can be used to optimize the results of the final route by removing unneeded vias and unnecessary routing jogs, where possible.

Once all violations from froute have been removed a DRC and LVS check of the routing in the design versus the abstract description of the cells should be performed. (A final DRC and LVS check needs to be run against the design when it has been converted to a layout view to account for the complete description of the cells contained in the cell layout views). The design will be checked for DRC violations such as antennas, shorts, minimum spacing rules, insufficient overlaps, missing vias and long wires routed in nonpreferred directions. Additional checks should be run on the special and regular wiring for opens, antennas, partially-routed nets, unconnected pins and pins with more than one used port. If violations are found in the design after running these checking functions, the designer will need to return to the froute and repair functions to try to resolve the violations.

Post Route Extraction

After each step in the routing process, extraction and back annotation would be performed. Again, as in placement, the accuracy of the extraction data is vital to guarantee silicon matches simulation as well as to the over-all cycle time required for the design. Remember the place and route flow will be different for each IC, depending on the goals you are trying to achieve. After final route, connectivity and DRC checks would be run by the routing tool to check for any violations the router may have caused. A subset of the verification rules should also be run to check for any DRC or LVS violations. Once this is clean, the cell layout data would be inserted and full checking would be performed.

Extraction methods need to be flexible enough to handle block interconnect parasitic extraction and incremental (only data which was changed) extraction as well as full chip flat or hierarchical detailed device and interconnect parasitic extraction.

During extraction a single net should be broken into a multitude of R/C networks. These R/C networks must be combined intelligently to form a single R/C element between the terminals of that net.

The move to an area router requires the interconnection extraction to take into consideration over the cell routing as opposed to just channel routing. The topology encountered with over the cell routing is more complicated and results in a tremendous increase in parasitic capacitance structures to be examined and the sheer numbers of elements in a given interconnect to be increased.

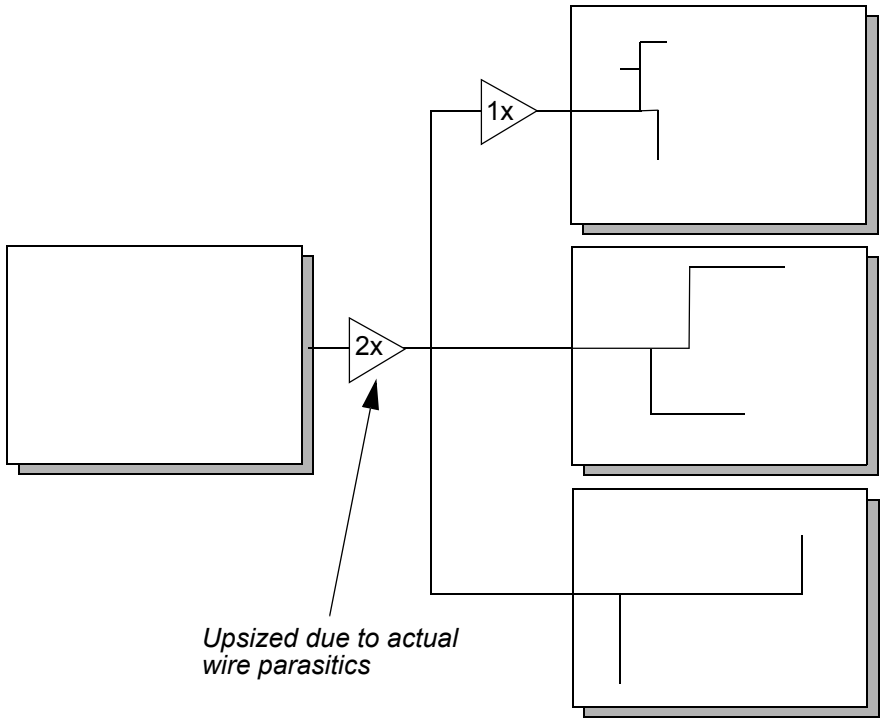
Process feedback is a critical element in the generation of accurate interconnect delays. The actual oxide thickness must be used for capacitance value extraction. The thickness and average width of the actual interconnect material is important also. It is no longer acceptable for the extractor to use drawn widths for area capacitor measurement as this may lead to overly optimistic values. As mentioned above the database must have access to the process generated data. The process line measures line width, oxide thickness and material thickness as well as doping levels and electrical measurements (Wafer Acceptance Test) results.

Synthesis Back-Annotation

Once the design has been run through the vendor floorplanning and/or placement tool, wire load information can be back-annotated into synthesis for timing optimization and final design rule check.

If the vendor has an ECO capability, then use the "resizing" mode so the netlist topology will not change. Constrain Synergy to only work on the problem areas by hierarchy management constraints.

Figure 7-10 Buffer Resizing



Performing Early Delay Estimation

Depending on the vendor kit, it may be possible to estimate delays before the entire chip is at the gate level. If this is possible, then the designers should perform delay estimation for timing-critical design blocks. This step provides an extra level of confidence that the circuit will perform at speed.

The interconnect delay is typically estimated based on an algorithm using technology-specific resistance and capacitance values. In some cases, a preliminary floorplan is created to identify gate clusters and placement regions for more accurate delay prediction. In addition, I/O pin locations, device package, and metal layer information may also be used for further accuracy. The cell delays and timing checks may also be re-calculated based on the additional wire load and the calculated input edge rate. The delay information is typically written to an SDF file, which can be read into a gate level simulation or timing analysis run.

Timing Verification

Timing verification should be used throughout the design process. The primary goal of timing verification is to prove that all design timing requirements have been met. Static timing analysis and timing-based simulation are both required to ensure complete analysis and verification are performed. Both methods should verify the full operating range of the circuit (mil, commercial, industrial). Static analysis performs the most comprehensive analysis. Special timing mode cases can be defined making the verification more accurate for design attributes such as logical false paths, multiple-cycle paths, etc. In timing-based simulation, multiple simulation are run which cover the process corners. Timing-based simulation is typically single delay mode simulation thus several runs are needed to verify the full operating range of the circuit. Like with fault simulation, developing simulation vectors which covers the full range of operation is required to get an accurate verification.

Timing analysis provides critical data which is used in the concurrent synthesis, floorplanning, placement process. Specifically timing analysis produces refined constraints which are directly utilized throughout the process including

- Maximum frequency calculation
- Slack/bottleneck analysis
- I/O timing reports

Gate level timing analysis is necessary after floorplanning, placement, routing, netlist translation, and any layout translation (converting from one design rule set to another). Timing analysis is used to verify that the design meets the system timing objectives. During the logic synthesis portion of the design process any obvious timing critical problem areas should have uncovered. These timing issues should have been addressed during the synthesis phase, and the design should be free of violations from a synthesis tool timing perspective.

After each section of the back-end is completed, i.e. floorplanning, placement and routing, and netlist translation, and layout translation the delays must be back annotated to the timing analysis tool.

Synthesis tools create a Standard Delay Format (SDF) file with every run. The SDF file contains path delay information for the critical paths in the synthesized design. To obtain the most timing-accurate simulation and

timing-analysis results of the gate-level implementation models—particularly timing-critical models—designers should read this SDF file into the simulation or timing analysis run.

However, once the design has been floorplanned, or even placed, a new level of accuracy can be achieved for interconnect and device timing. Because of this, it is recommended that calculated delays be back-annotated and a full chip timing analysis performed.

It is recommended to use a blend of simulation and static path analysis to provide a highly accurate timing analysis. Static path analysis is used to determine fast and precise min/max timing at storage device input, allowing the timing checks built into each storage device model to be verified precisely.

Simulation adds more precision to the path analysis than is possible with purely static algorithms. This capability is generally used to simulate the clock system of the design, enabling timing analysis to handle designs with any form of clocking system - even asymmetrical, multi-phase, and gated clocks.

Final simulations should also cover the testing required by silicon such as AC timing vectors. Any testing which can be simulated with the HDL simulators (Iddq is an exception) should be verified with the tester input and output vectors. This will prove out the test vectors and reduce the time spent debugging final silicon.

References

10

Block-Level Implementation

Overview

Block implementation describes the design process starting from RTL design capture and ending with placement and gate level block logic verification of the implementation. Figure 0-1 shows the block implementation process diagram. For the purposes of this document, a design block is defined as any design partition which will be synthesized and/or functionally verified as a standalone design unit.

Design blocks can be synthesized standalone for many reasons including block size, constraint control, test strategy, optimization goals, etc. In addition, the design block may be simulated standalone because its functionality may be more easily verified by applying implementation verification unit tests (IVTs) outside the context of the whole design.

Figure 0-1 Block Implementation Process Diagram

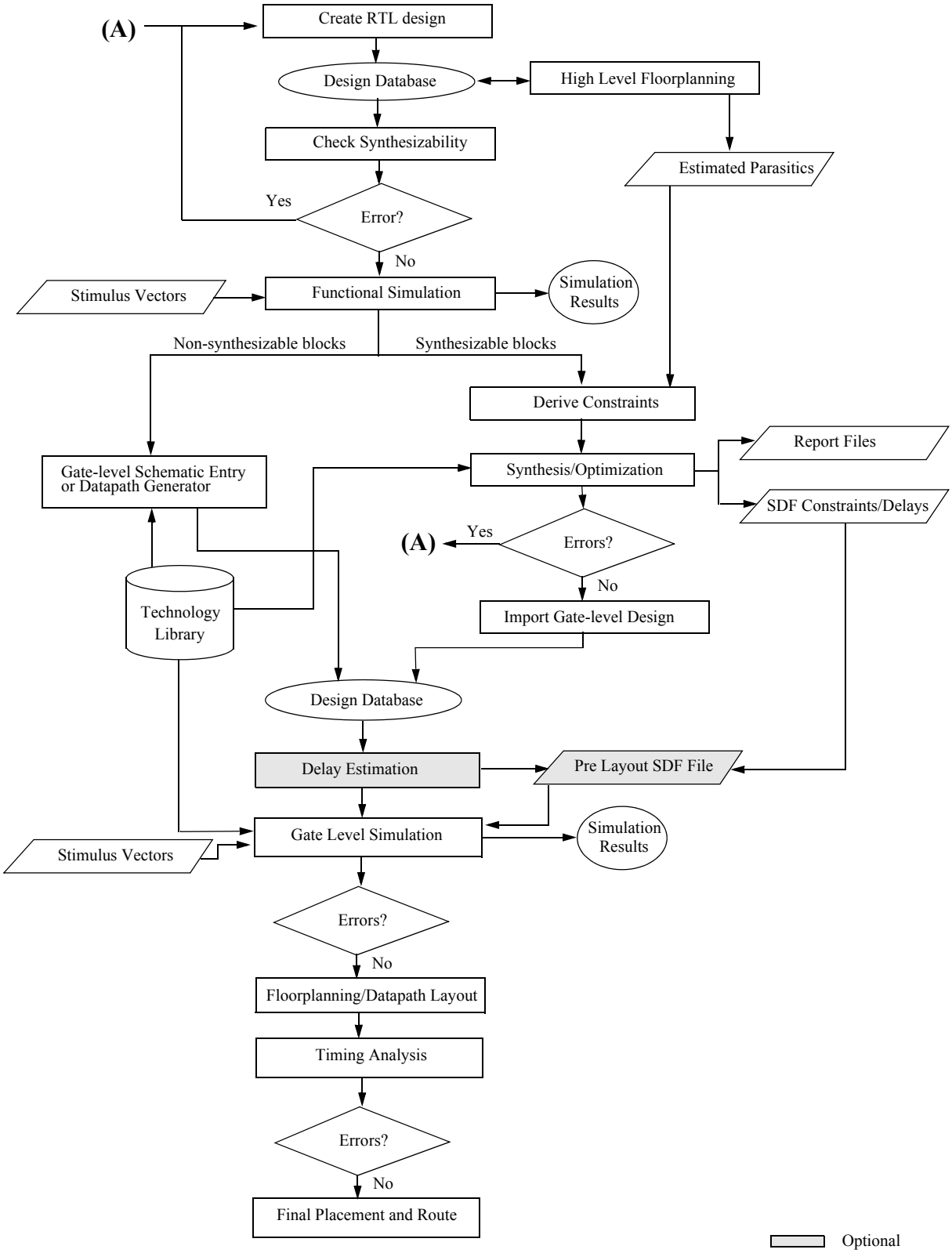
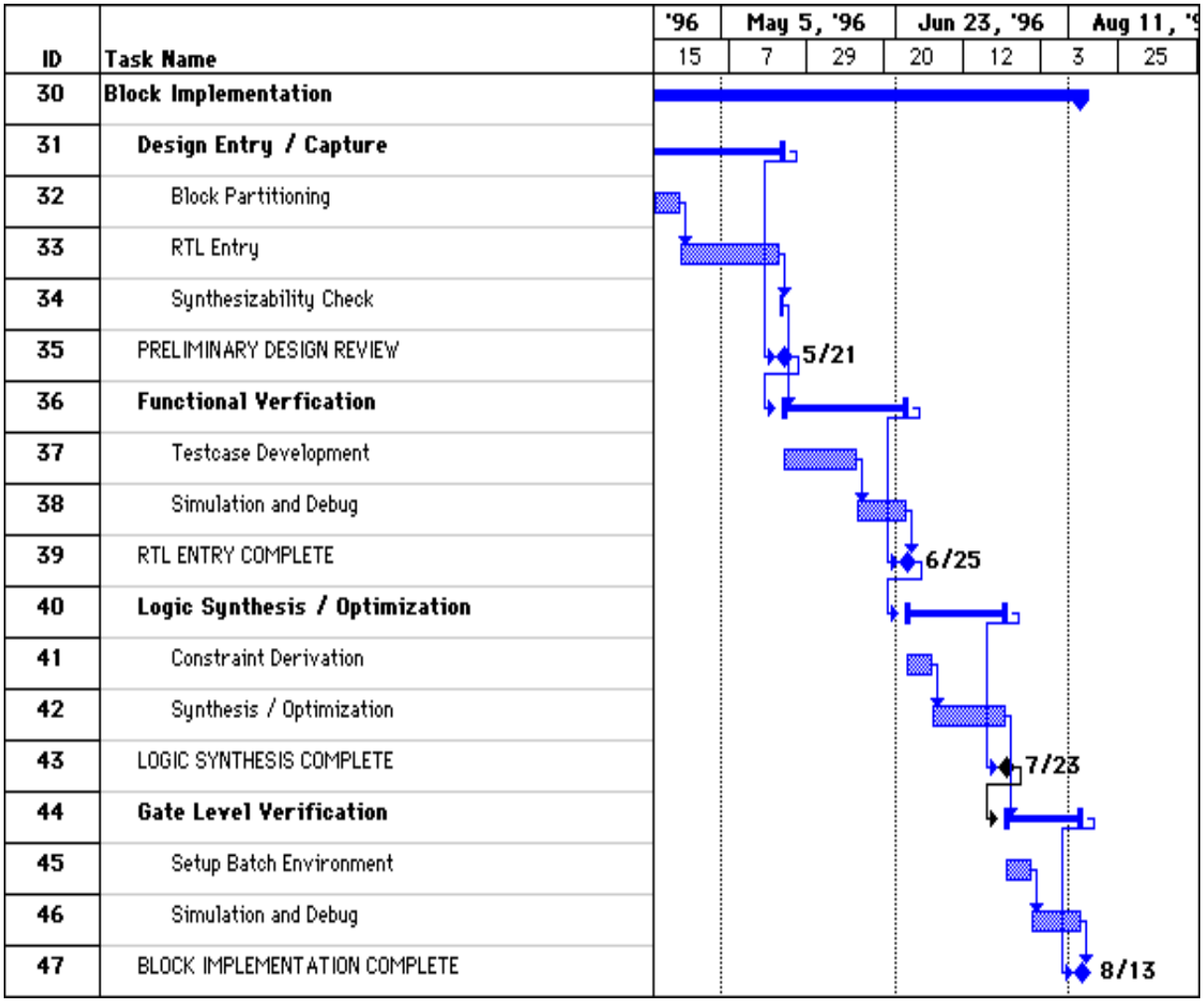


Figure 0-2 is an example schedule of the block implementation process. This is intended to reflect relative allocation of time for each of the process steps and their dependencies. It is not meant to reflect absolute duration of these tasks as they will vary greatly depending on block size and complexity.

Figure 0-2 Block Implementation Schedule



Subsystem Partitioning

The DTMF was partitioned at the system level into firmware (the Goertzel algorithm) and hardware components. The core hardware component of the DTMF is partitioned into the following parts:

- 16-bit DSP processor
- 512x16 read-only memory
- 256x16 data sample random access memory
- 128x16 program random access memory
- Results character converter
- Additional control logic and peripherals

The approximate gate count is 53,000 (including memories).

RTL Models

The principle areas of interest during block implementation for this design are

- The TDSP processor
- The results character converter
- The macro blocks

Tiny Digital Signal Processor

The 16-bit digital signal processor, which has 65 unique instructions in its instruction set architecture, was modeled both in hierarchical Verilog HDL and VHDL. The modules for the DSP are:

- Instruction Decode
- Instruction Execute
- ALU
- Data Bus Interface
- Program Bus Interface
- Port Bus Interface
- Accumulator Status
- Multiplier
- Update Auxiliary Register

The instruction execution and decode modules where written as finite state machines modeling the six clock cycle instruction execution with true clock cycle accuracy. A portion of the Instruction Execute unit state machine is depicted in Figure 0-3.

Figure 0-3 Instruction Execution Module Code Fragment

```
`HI_NIB_C: begin
        case (ir[`S_OP])
        // asm:MAC
        `MAC:
        begin
                if (phi_3 && ! two_cycle & ! three_cycle)
                begin
                        skip_one <= 1 ;
                        sel_op_a <= `OP_A_TOP ;
                        sel_op_b <= `OP_B_MDR ;
                end
                if (phi_6 && ! two_cycle & ! three_cycle)
                begin
                        p <= mpy_result ;
                end
                if (phi_6 && ! two_cycle & ! three_cycle)
                begin
                        two_cycle <= 1 ;
                end
                if (phi_3 && two_cycle & ! three_cycle)
                begin
                        sel_op_a <= `OP_A_ACC ;
                        sel_op_b <= `OP_B_P ;
                        alu_cmd <= `ALU_ADD ;
                        two_cycle <= 0 ;
                        three_cycle <= 1 ;
                end
                if (phi_6 && ! two_cycle & three_cycle)
                begin
                        acc <= alu_result ;
                        if (alu_result[`S_ACC_OV])
                                ov_flag <= 1 ;
                        acc <= alu_result ;
                        if (alu_result[`S_ACC_OV])
                                ov_flag <= 1 ;
                        three_cycle <= 0 ;
                end
        end
```

To maintain consistency during firmware development, a utility was written to only allow the assembler to accept implemented instructions while the instruction set was being implemented. To accomplish this, comments were added to the code (asm:MAC, e.g.) to indicate that an instruction had been implemented. This can also be found in Figure 0-3.

Note also that the opcode definitions were implemented as ‘define constructs in verilog. These were defined in a header file (tdsp.h). Figure 0-4 show a portion of the tdsp.h file. This allowed the opcode definitions

to exist in one file and be accessed consistently by all design, verification, and firmware engineers.

Figure 0-4 TDSP Header File

```
`define LAR     'TP'h38      // Load Auxiliary register
`define LARK    `TP'h70      // Load Auxiliary register with a constant
`define LARP    `TP'h68      // Load Auxiliary register pointer
`define LDP     `TP'h6f      // Load data page pointer
`define LDPK    `TP'h6e      // Load data page pointer with a constant
`define LST     `TP'h7b      // Load status from data memory
`define LT      `TP'h40      // Load multiply temporary operand
`define LTA     `TP'h42      // Load multiply temporary operand and
                             // accumulate previous result
`define LTD     `TP'h46      // Load multiply temporary operand,
                             // accumulate previous result, shift data memory
`define LTP     `TP'h43      // Load multiply temporary operand,
                             // previous result moved to accumulator
`define LTS     `TP'h44      // Load multiply temporary operand and
                             // subtract previous result
`define MAR     `TP'h68      // Modify auxiliary register
`define MPY     `TP'h6d      / Multiply
`define MPYK    `TP'h80      // Multiply immediate
`define MAC     `TP'hc0      // Multiply and accumulate
`define NOP     `OP'h7f80    // No operation
`define OR      `TP'h7a      // Or with low accumulator
```

Results Character Converter

The results character converter reads the contents of the 8 spectral component registers and determines which high and low frequency components exist. In addition, it verifies that the spectral component magnitudes meet the required specification and outputs the detected character. Figure 0-5 shows a block diagram of the dataflow through the RCC.

Figure 0-5 Results Character Converter Block Diagram

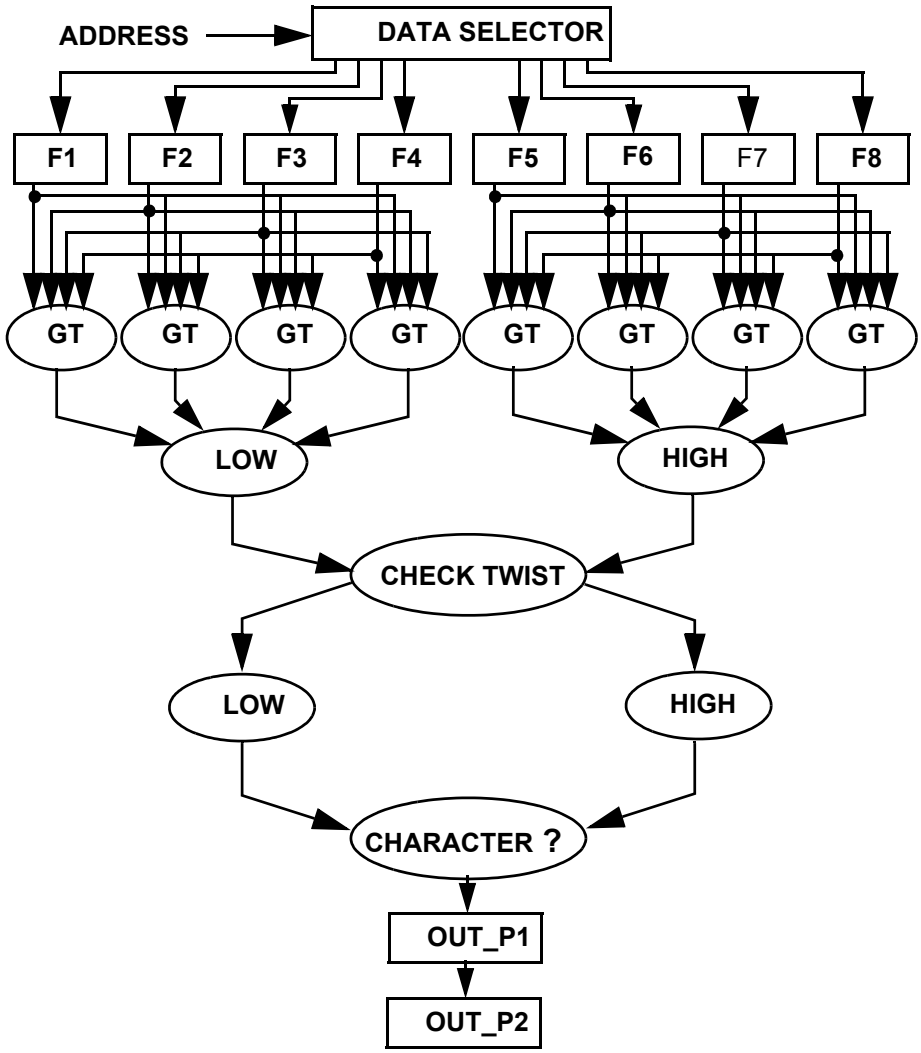


Figure 0-6 shows a portion of the HDL code for the Results Character Converter block that performs the magnitude comparison. Note the three subtractions required for this operation. During synthesis, it is required to constrain the synthesis tool to perform resource sharing on these operations to minimize the area impact to the chip.

Figure 0-6 Results Character Converter Code Fragment

```
always @(posedge clk)
   begin : rcc_machine
   if (reset)
      disable rcc_machine ;
   else if (go)
      begin
      low <= 3'b100 ;
      high <= 3'b100 ;
      clear_flag <= 1 ;
      out_p2 <= out_p1 ;  // digit pipeline
      gt_comp( r697, r770, r852, r941 ) ;
      @(posedge clk)
      clear_flag <= 0 ;
      if (gt)
         begin
         low <= {1'b0, `V_697} ;
         low_mag <= r697 ;
         end
      gt_comp( r770, r697, r852, r941 ) ;
      @(posedge clk)
      if (gt)
         begin
         low <= {1'b0, `V_770} ;
         low_mag <= r770 ;
         end
      gt_comp( r852, r697, r770, r941 ) ;
                                 ...
task gt_comp ;
   input [15:0] opa, opb, opc, opd ;

   reg [16:0] cmpb, cmpc, cmpd ;
   begin

   @(posedge clk)
      cmpb <= opb - opa ;
   @(posedge clk)
      cmpc <= opc - opa ;
   @(posedge clk)
      cmpd <= opd - opa ;
   @(posedge clk)
      gt <= cmpb[16] & cmpc[16] & cmpd[16] ;
   end
endtask
```

Macro Blocks

The DTMF architecture defined requirements for three macro blocks.

- 512x16 ROM
- 256x16 RAM
- 128x16 RAM

These blocks were implemented by the target ASIC vendor. Requirements were given to the vendor early in the design process for implementation. In return the ASIC vendor provided behavioral models with estimated timing information in time for initial RTL simulation of the chip. The physical abstracts (with metal1 and metal2 obstructions) for the blocks were also provided by the vendor so that they could be used in the chip floorplan. The RAM models were modified to provide some additional debug capabilities. Figure 0-7 shows a portion of the RAM code to display write events and dump contents to a file triggered by a named event.

Figure 0-7 RAM Code Fragment

```
/*
 * write logic
 */
always @(negedge wr)
begin: write_block
        mem[a] <= din;
        `ifdef DSRAM_DEBUG
                if (test.debug_print)
        $display("%t %m : Writing %h at address %h", $time, din, a);
        `endif
end

/*
 * dump logic
 */

`ifdef DEBUG_DUMP_RAM
function dumpIt;
        input t ;
        integer i ;
begin
        $write( " **  Dumping ram: (%m)\n" ) ;
        $fwrite( test.dumpRamFile, "//-----------------\n" ) ;
        $fwrite( test.dumpRamFile, "// Dumping ram: (%m)\n" ) ;
        $fwrite( test.dumpRamFile, "// current time: %t\n", $time ) ;
        for( i = 0 ; i < words ; i = i + 1 )
        begin
                $fwrite( test.dumpRamFile, "@%h %h\n", i, mem[i] ) ;
        end
        dumpIt = t ;
end
endfunction
`endif
```

Design Verification

The goal of design verification is to verify that the DTMF system works correctly at both the RTL and the gate level of implementation. The design was verified at two levels; the DSP processor and the DTMF system. An implementation verification test plan (IVT) and necessary test benches were written to exercise the entire instruction set. Note that to verify the TDSP instruction set, architecture verification test (AVTs) were also used.

The system-level testbench for the DTMF verifies a sequence of eleven characters input to the design in ulaw compressed PCM format. Each character is then uncompressed, converted to linear PCM and then written to the data sample memory. The character sequence was (1-800-862-4522) which is the Cadence Design Systems voice mail network number.

The same testbench was used for both RTL and gate-level simulations. Built-in simulator functions were used to automatically generate binary output files and compare them with previous simulation results. The testbench is output (character detection) is shown below for NC-Verilog.

Figure 0-8 Simulation Output

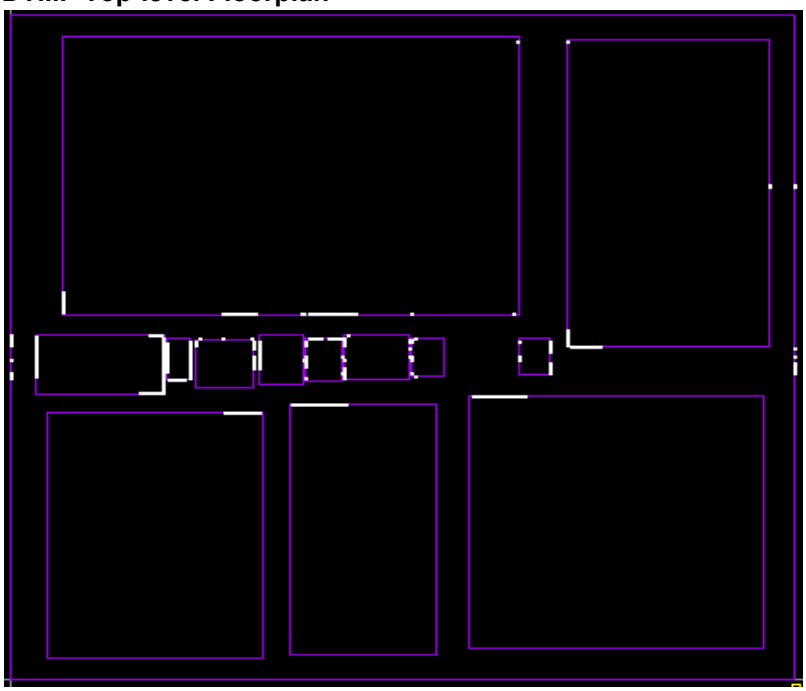
```
ncsim> run
 **  Initializing ROM (test.top.ROM_INST) with (etc/rom.txt)
$readmem - words in file "etc/rom.txt" less than that given by address bounds
 **  Initializing RAM (test.top.RAMS_INST.DSRAM) with (etc/pcm256.txt)

 ** Debug tracing is off, type 'debug_print = 1;' to enable tracing...

 **  Loading u-Law PCM signal
 **  Found digit(1)  @     100.000000ms
 **  Found digit(8)  @     228.000000ms
 **  Found digit(0)  @     356.000000ms
 **  Found digit(0)  @     484.000000ms
 **  Found digit(8)  @     612.000000ms
 **  Found digit(6)  @     740.000000ms
 **  Found digit(2)  @     868.000000ms
 **  Found digit(4)  @     996.000000ms
 **  Found digit(5)  @    1124.000000ms
 **  Found digit(2)  @    1252.000000ms
 **  Found digit(2)  @    1380.000000ms
Memory Usage - text: 2969267, static: 3416336, dynamic: 3076008, total: 9461611
```

Design Planning

Prior to beginning the synthesis process, it is important to understand and analyze the physical design requirements. This allows the designer to drive the synthesis process to arrive and a more accurate first pass results instead of back-annotating and iterating to meet timing and drive strength requirements.

In the case of the DTMF, the design is not I/O limited and therefore the majority of the routes will be in the center of the core. The I/O's were located on the right and left sides for accessibility. Once the locations are determined for the design blocks and the pin placement is determined, accurate wire models can be generated as well as estimated block interconnect. This information is then passed to the synthesis process. Figure 0-9 shows the results of I/O placement and pin optimization on the original high level floorplan from Chapter 7, "High-Level System Design,".

Figure 0-9 DTMF Top-level Floorplan



Implementation

The hardware component of the DSP can be synthesized in four passes (All other top level modules for the DTMF can be synthesized in a single pass). The first two passes synthesize the ALU and the multiplier separately. The third pass synthesizes the rest of the DSP, while preserving the ALU and the multiplier from the earlier passes. The final pass performed drive optimization on the entire design, and is described in Chapter 11, “Chip-Level Assembly Implementation.”

Figure 0-10 shows the synthesis constraints as they would be defined for Synergy for the ALU. The constraints would have the following effect:

- Identify SmartBlocks, a macro library, as the target library
- Select a wire model, based on the expected gate count of 2000
- Require the signals to arrive at the output ports no later than 40 ns after the active edge of the clock
- Preserve the original hierarchy of the ALU description
- Define a clock with a rising edge at 0, a falling edge at 25, and a period of 50
- Require Synergy to use a single resource for all addition operations of 8 bits and a single resource for all subtraction operations of 8 bits
- Identify the name and characteristics of the reset signals for the FSMs
- Run Synergy in normal mode with cost (area) as the priority

Figure 0-10 ALU Constraints

```
library -macro SmartBlocks
wire_model 5K_DLM

timing -required -rise 40.000000 -g
timing -required -fall 40.000000 -g

preserve_hier_tree

clock clk -rise 0.000000 -fall 25.000000 -period 50.000000
operator PLUS -width 16 -opcount 2 -partcount 1
operator MINUS -width 16 -opcount 2 -partcount 1
fsm -s alu -reset reset

synthesis_options -steps synthesizer schematic
synthesis_options -alternative normal
synthesis_options -priority cost
```

Figure 0-11 shows the synthesis constraints as they would be defined for Synergy for the multiplier. The constraints would have the following effect:

- Select a wire model, based on the expected gate count of 2000
- Require the signals to arrive at the output ports no later than 90 ns after a change on the inputs (the multiplier is a two cycle operation).
- Preserve the original hierarchy of the multiplier description
- This is a combinational block so no clock constraints are required.
- Identify the name and characteristics of the reset signals for the FSMs
- Require Synergy to generate explicit (name-based) port lists in the modules.
- Run Synergy in normal mode with cost (area) as the priority

The ALU is a datapath-rich module was also implemented using automated datapath techniques (SmartBlocks library). All datapath partitioning and tiling was done automatically by the synthesis tool through procedural calls to generate the datapath layout information (tile files). There were a total of 15 datapath functions in the ALU which comprised about 75% of the ALU module.

The multiplier is also a highly regular datapath structure. The same process was used to synthesize the multiplier. Figure 0-11 shows the constraints as defined for the multiplier.

Figure 0-11 Multiplier Constraints

```
library -macro SmartBlocks
wire_model 5K_DLM
timing -arrival -rise 0.000000 -fall 0.000000 -g
timing -required -rise 90.000000 -fall 90.000000 -g
preserve_boundary -tree

synthesis_options -alternative normal
synthesis_options -priority cost
```

Figure 0-12 and Figure 0-13 are two placement and route examples of the multiplier; the first the results of a conventional synthesis tool, and the second the results of a synthesis tool with advanced datapath functionality. The regularity of this structure makes it an ideal for datapath placement and routing. This approach can add significant more control to the placement and routing process which reduces the chance of violations due to unforeseen large variances in interconnect lengths for each of the bit-level routes between datapath elements.

Figure 0-12 Multiplier Implementation (Traditional Synthesis)

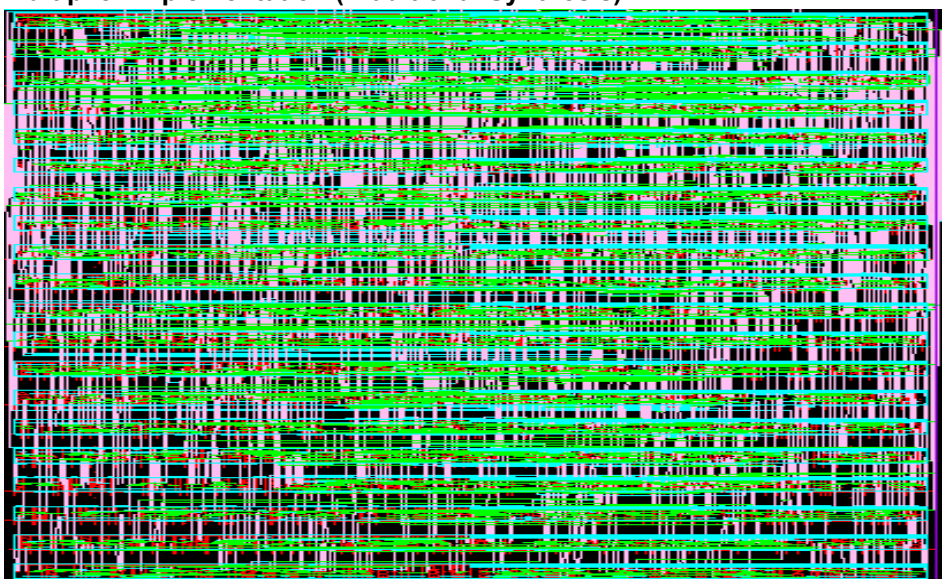
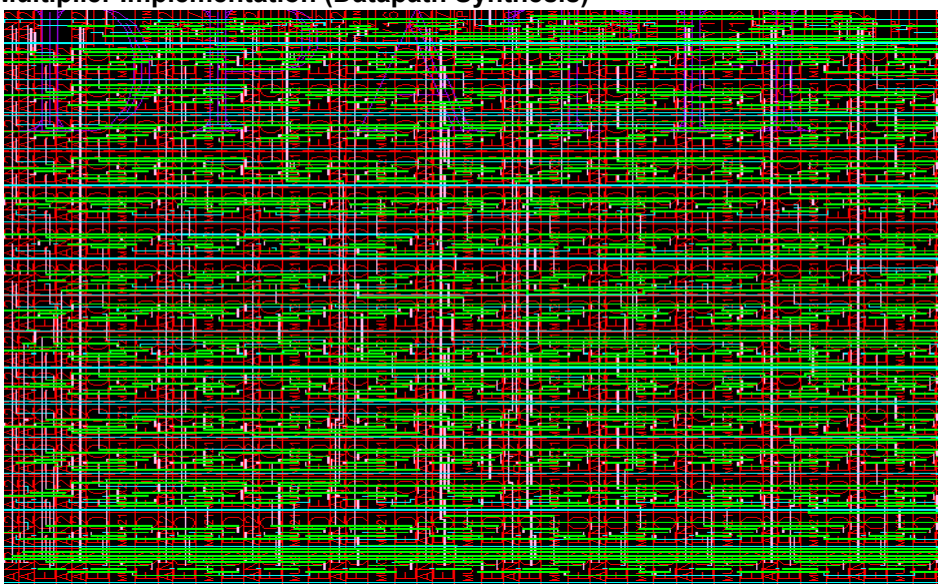


Figure 0-13 Multiplier Implementation (Datapath Synthesis)



The rest of the TDSP was made up of bus interface logic, instruction decode, and instruction execution logic. There were 16 state machines in the TDSP. Figure 0-14 is the constraint file for synthesizing the TDSP which highlights 5 of the state machines. Sequential state encoding is applied to all state machines.

Figure 0-14 TDSP Constraint File

```
        drive 3 -g
        input_maxload 5 -g
        output_load 75 -g
        timing -required -minrise 3 -maxrise 35 -g
        timing -required -minfall 3 -maxfall 35 -g

        scope tdsp_core
        preserve_boundary -tree
        wire_model -tree 10to20K
        timing_path reset -disable yes
        timing_path mpy_result alu_result -disable yes
        clock clk -rise 0 -fall 20 -period 40

        scope execute_i
        wire_model -tree 0to10K
        fsm -block execute_machine  -reset reset -nature asynchronous -edge 1
        scope update_ar
        wire_model 0to10K
        fsm -block update_ar_machine -reset reset -nature asynchronous -edge 1

        scope decode_i
        wire_model 0to10K
        fsm -block decode_machine  -reset reset -nature asynchronous -edge 1

        scope port_bus_mach
        wire_model 0to10K
        fsm -block port_bus_machine  -reset reset -nature asynchronous -edge 1
        output_load 100.0 data

        scope data_bus_mach
        wire_model 0to10K
        fsm -block data_bus_machine  -reset reset -nature asynchronous -edge 1
        output_load 100.0 data

        scope prog_bus_mach
        wire_model 0to10K
        fsm -block prog_bus_machine  -reset reset -nature asynchronous -edge 1
        output_load 100.0 data

        scope tdsp_core.MPY_INST
        maintain -tree

        scope tdsp_core.ALU_INST
        maintain -tree

        synthesis_options -priority cost
        delay_params -type max

    run_synthesizer
```

The TDSP synthesis results are shown in Figure 0-15. Since the block was synthesized maintaining the inherent hierarchy of the TDSP, the results are detailed for each sub-block of the TDSP. During floorplanning regions were automatically created for the ALU and multiplier by the datapath tool which optimally ordered and placed the datapath elements. The remaining portion of the TDSP was placed within the block boundary constraints along with the ALU and multiplier previously placed by the datapath placer. The resulting physical placement is shown in Figure 0-16.

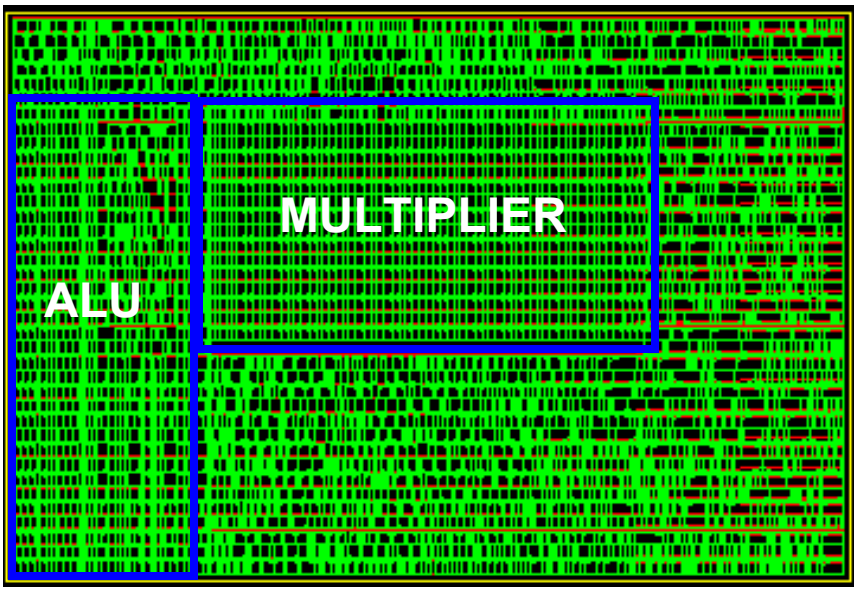
Figure 0-15 TDSP Synthesis Summary

Timing
Maximum Clock Frequency = 33 MHz
Longest Path Delay = 30 ns
Minimum Slack Time = 12 ns

Cost
1303940.000 um^2

Module	WireModel	CellCount
data_bus_mach	0to10K	189
decode_i	0to10K	170
prog_bus_mach	0to10K	192
update_ar	0to10K	311
execute_i	0to10K	938
accum_stat	0to10K	24
imp_cadd32_3	0to10K	237
um32x32_0	0to10K	1461
mult_32	0to10K	1672
alu_32	0to10K	868
port_bus_mach	0to10K	180
tdsp_core	10to20K	4888

Figure 0-16 TDSP Placement



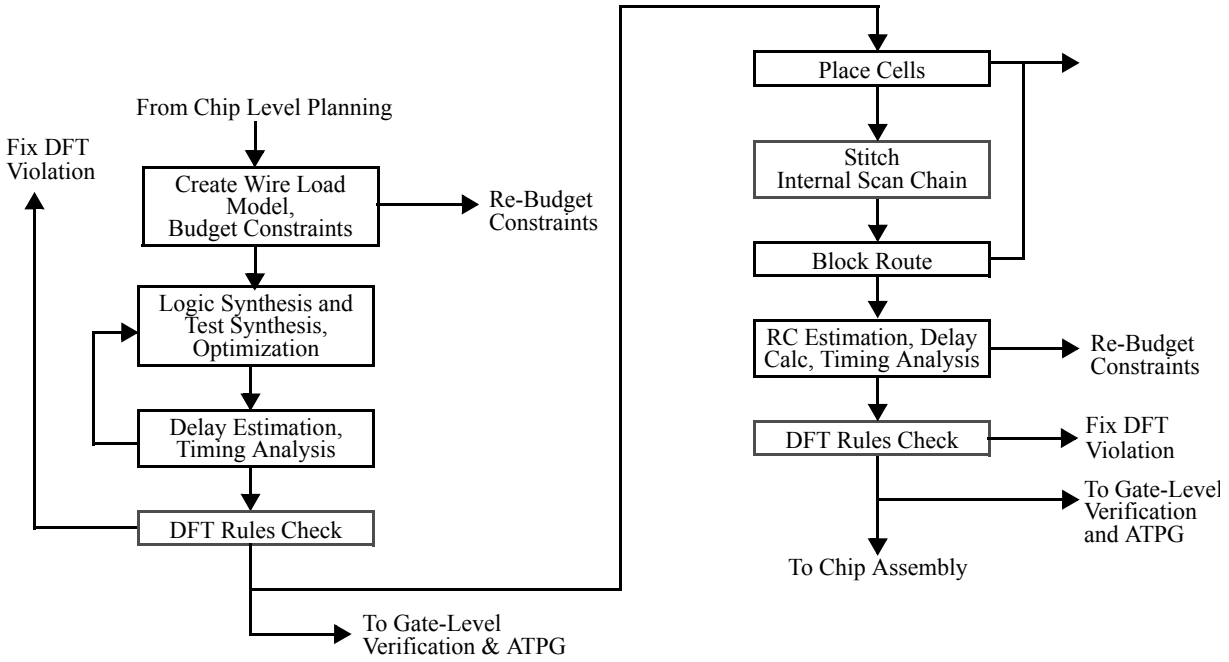
DFT Logic Design and Verification

Once the DFT planning phase is complete, design and verification of the DFT logic can be done. DFT logic design and verification occur concurrently with design and verification of the normal system log. It is important to realize that any DFT logic which is added to the design to support test will require logic and timing verification—just as with the regular system logic—in order to assure that the DFT logic is functionally correct and has accurate timing.

Block-Level DFT Synthesis and Insertion

The flow in Figure 0-17 is used for each top level physical partition of the design. Since test synthesis occurs at this point in the flow, it is desirable to determine what the physical scan order of the internal scan chains will be. This is so that the gate-level netlist will have the correct connections for the final scan chain orders. If the final scan order is not known, it can be determined based on physical cell placement of the scan elements and then the scan chain order can be stitched back into the netlist.

Figure 0-17 Block-Level Synthesis and Design



Scan Path and Test Function Verification

The flow in Figure 0-17 shows various points where the design can be passed either to verification or ATPG test vector generation. Particular attention must be paid to verification of proper scan path shifting and correct scan path ordering.

Other test functions, for example clock de-gating for test purposes, I_{ddq} enable functions, parallel access scan functions, and test access collars should also be verified for correct operation. Though not specifically a test function, proper operation of the system reset function should also be verified, as a reset is often used prior to applying tests and would therefore be important to correct test operation on the tester.

Also, any DFT logic that did not go through the synthesis step in Figure 0-17 should be verified for correct timing. This may typically happen for internal scan path structures that were inserted using a cell-level substitution of internal state elements. In this case it is important to verify that there are no hold violations during scan path shifting. Any hold violations on the scan path will mean that the ASICs scan paths can not be reliably shifted, and therefore the scan path can not be used for testing or debug of the chip.

References

Lab Exercise: Design Entry, Simulation, and Synthesis

Circuit: Bus Arbiter FSM

1. **Code the memory bus arbiter given the module description found in the *DTMF Design Description* section of this manual.**

   All data creation and analysis jobs should be done in the *work* directory.

   ```
   Verilog Module: arb (arb.v)
   VHDL Entity/Architecture: arb/rtl (arb.vhd)
   ```

   Model the bus arbiter as an explicit finite state machine. The state encodings are supplied in the following files:

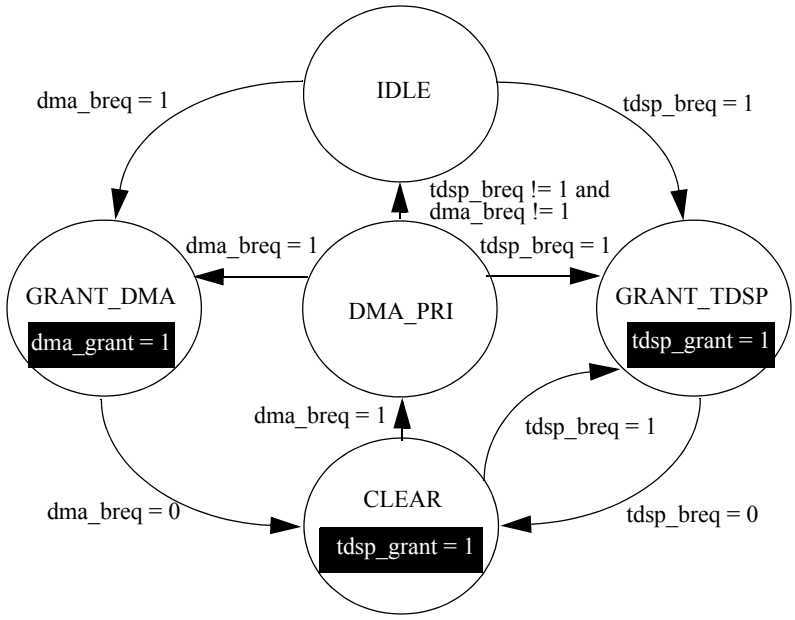
   ```
   Verilog Include File: ../include/arb.h
   VHDL Package: ../include/arb_p.vhd
   ```

   Note the state encodings have been gray-coded:

   ```
   ARB_IDLE        : 001
   ARB_GRANT_TDSP  : 000
   ARB_GRANT_DMA   : 010
   ARB_CLEAR       : 011
   ARB_DMA_PRI     : 111
   ```

   The bus arbiter is clocked by a 25 Mhz positive edge triggered clock (clk) and is asynchronously reset with an active high reset (reset).

Figure 0-18 Memory Bus Arbiter State Machine Diagram



Note: Unless otherwise shown, *dma_grant* and *tdsp_grant* are 0.

2. **Once the design has been entered, verify its synthesizability. A script has been supplied to run a synthesizability check:**

```
Verilog: check arb.v
VHDL: check arb.vhd
```

3. **Once the design is synthesizable, simulate it using the supplied testfixture and configuration files. This will be a pass/fail test.**

```
Verilog: test_arb_rtl.fs
VHDL: cfg_test_arb_rtl.vhd
```

4. **Once the functionality is verified, synthesize the design. Constraints are supplied as well as a script to run batch synthesis.**

```
Verilog: syn arb.v
VHDL: syn arb.vhd
```

5. **When the synthesis job is complete, copy the resulting *verilog* netlists into the *work/lib* directory and rename to the appropriate naming convention:**

```
synruns/arb_rtl.run/syn.v -> lib/arb.vs
```

Note the Verilog netlist is the data we will move forward to sign-off with for both Verilog and VHDL users.

Turn in the following:

arb.v or arb.vhd

syn.report.p

Lab Exercise: Functional Verification

Circuit: DTMF Receiver Core System

‰ **Simulate the Bus Arbiter FSM you completed in Lab 2 in the context of the DTMF receiver system.**

The circuit function is to detect a DTMF signal using Goertzal's algorithm to perform the Discrete Fourier Transform. Consult the *DTMF Design Description* chapter for the circuit description and explanation of the Goertzal algorithm.

Simulation configurations are supplied in the work directory:

```
Verilog: verilog -f dtmf_recvr_core_test_rtl.fs

VHDL: cv cfg_dtmf_recvr_core_test_rtl.vhd
      ev work.dtmf_recvr_core_test_rtl
      sv work.dtmf_recvr_core_test_rtl/SIM
```

The testfixture provided has a built-in pass/fail self-check. To enable this self-check, do the following:

```
Verilog : +define+LAB3

VHDL    : Use 'etc/lab3.cmd' input file
```

Run the simulation from the work directory.

Turn in the following:

Simulation log file

Lab Exercise: Verification Strategies (Pattern Capture)

Circuit: Bus Arbiter FSM

‰ **Update the testfixture for the bus arbiter for unit testing.**

The testfixture should write out a vector file to be used for pattern comparison with gate level simulation results.

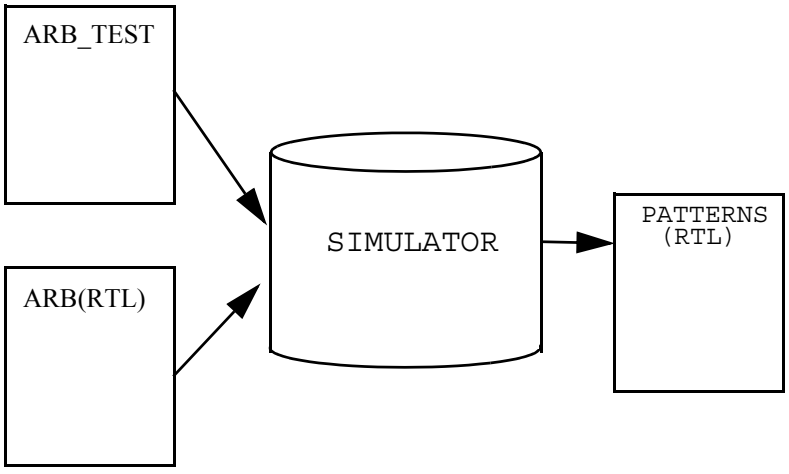
Verilog users can use the *$incpattern_write* function. VHDL users will have to code the appropriate functionality into the testfixture. The testfixtures are in the work directory:

```
Verilog Module: arb_test
Verilog Source File: arb_test.v
VHDL Entity/Architecture: arb_test/behavioral
VHDL Source File: arb_test.vhd
```

The testfixture source files should be source controlled. Again use the simulation configurations to run the simulation.

```
Verilog: arb_test_rtl.fs
VHDL: cfg_arb_test_rtl.vhd
```

Figure 0-19 Pattern Capture



Turn in the following:

arb_test.v[hd]

Lab Exercise: Verification Strategies (Pattern Compare)

Circuit: Bus Arbiter FSM

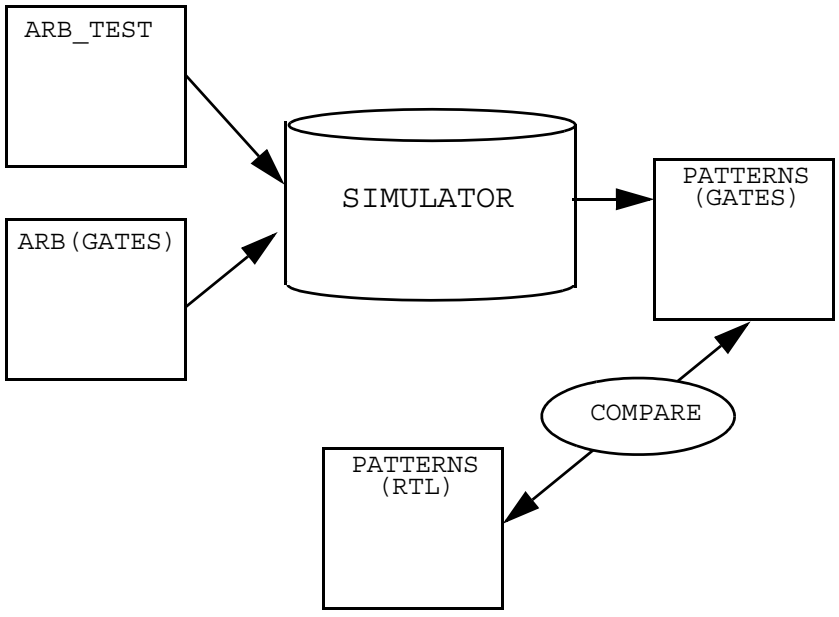
‰ **Verify the synthesized netlist for the *arb* block by reading in the patterns from the RTL simulation (Lab 4) and comparing them with the results from the gate level simulation.**

Update the testfixture from Lab 4 to do this. Verilog users can use the *$incpattern_read*, *$strobe*, and *$strobe_compare* commands for this. VHDL users will have to code the appropriate functionality for this. The only requirement is to be able to show simulation differences, the signals they apply to, and the time when the differences occurred.

Create a simulation configuration to run the simulation using the synthesized netlist from the previous lab.

```
Verilog: arb_test_gate.fs
VHDL: cfg_arb_test_gate.vhd
```

Figure 0-20 Pattern Compare



Turn in the following:

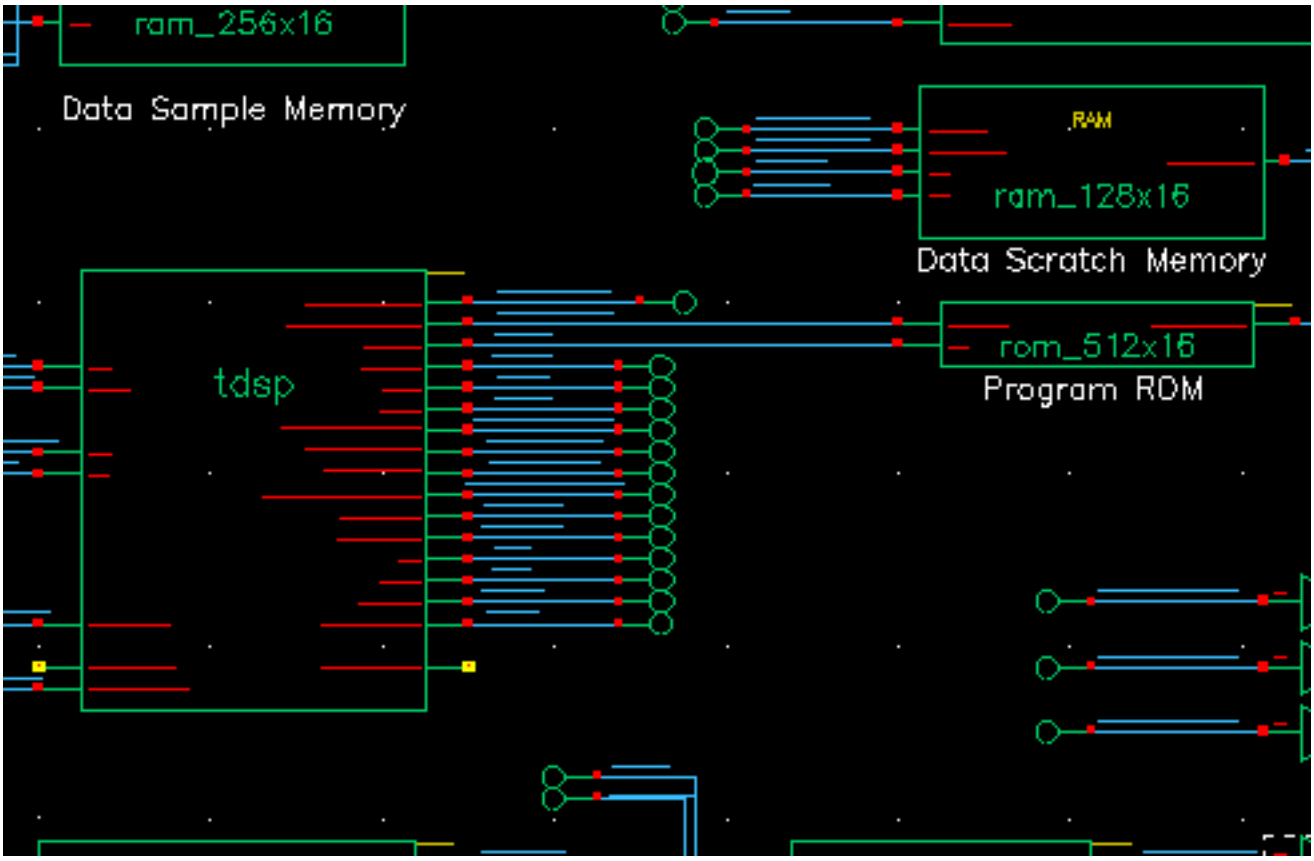
arb_test.v[hd]

Lab Exercise: Hardware/Firmware Co-Verification

Circuit: DTMF Receiver System

The purpose of this lab is to introduce you to the tools and techniques needed to debug HDL within the target simulator environment. To this end you will be creating a small TDSP assembly language program to test the *Data Scratch Memory* that is part of the DTMF Receiver System

Figure 0-21 DTMF Receiver System Block Diagram



The test program you create will write an incrementing pattern to successive RAM locations until the *Data Scratch Memory* is filled. For convenience, the pattern value is the physical RAM address for the location currently being written to. Once the RAM is filled, your program reads from the memory and compares the value found with the physical RAM address your program is accessing. If an error occurs, the program

exits, leaving the erroneous address in the accumulator. If the test passes, the accumulator value should be 0x0000 upon the program exiting.

Sounds easy enough. How about a couple of hints

Hint 1: Reference the *DTMF Design Description* chapter for a complete listing of the TDSP instruction set, instruction execution, and addressing modes.

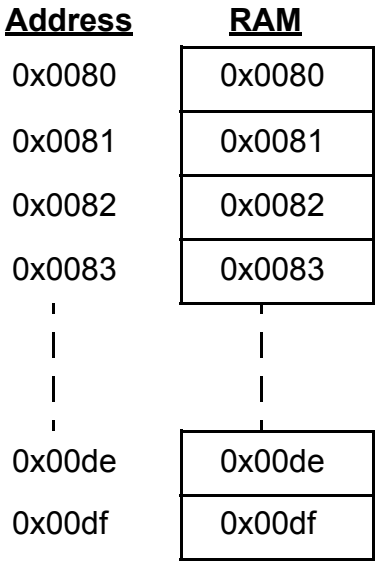
Hint 2: Upon reset, the TDSP starts reading the program ROM at location 0x0000.

H**int 3:** The TDSP direct addressing mode only includes 7 bits in the immediate address portion of the instruction. To address locations higher that 0x007f, you must load the "data page pointer."

Hint 4: Remember the DTMF Receiver memory map that is described in the *DTMF Design Description* chapter? The *Data Scratch Memory* is located at locations 0x0080 - 0x00df in the data memory space.

Hint 5: Your memory should look something like the following if your program is working properly

Figure 0-22 RAM Contents



You may wish to enable the RAM dumping code (review the source code for one of the RAM modules) to assist in viewing the memory contents.

Hint 6: Take a look at the **BANZ** instruction. The instruction can be thought of as a hardware *for* loop.

Hint 7: Upon exiting your program, have the TDSP enter a tight loop that you can easily trace and set break-points for the simulator.

Hint 8: (verilog users) Review and define as necessary any of the *define* options that are available at the head of the testfixture—some of these will be useful to you in completing this lab successfully. Also, review the testfixture, RAM, and ROM models for memory content dump routines and how to initiate the dump process.

Hint 9: (vhdl users) Review the RAM and ROM models for content dump routines and how to initiate the dump process.

Enough hints for now. Here's a procedure to follow when working through the lab exercise:

1. **Create the simulation work directory**

   ```
   dtmf_proj/work/ramtest
   ```

2. **Create a simulation configuration file.**

   You need to include the *dtmf_recvr_core_test* testfixture and the *dtmf_recvr_core_rtl* configuration. Note that you will be making some changes to the testfixture, so fetch a local copy to your run directory that you can edit.

3. **Create a local copy of the current opcodes in your run directory by using the program *getop*.**

   You should invoke ***getop*** as follows (note that you will be referencing the verilog source data for this operation):

   ```
   getop <vlog_path>/src/*. <vlog_path>/include/*.h
   ```

   getop can be found in the *dtmf_proj/bin* directory.

4. **Create your source assembly code.**

   This file should be called ***rom.asm***. You'll assemble your source code using ***tdspasm***. Invoke the assembler as follows:

   ```
   dtmf_proj/bin/tdspasm rom.asm
   ```

   This operation will produce:

   ```
   rom.lst: the composite listing
   rom.sym: the symbol cross-listing
   rom.obj: the machine object
   ```

   Please review these files to familiarize yourself with their contents.

5. **The ROM used in the DTMF Receiver looks for the file *rom.txt* in the run directory for initialization data.**

   Since the assembler file *rom.obj* contains initialization data, make a UNIX file link to *rom.obj* and call it *rom.txt*.

6. **At this point you may wish to invoke the simulation environment as a sanity check to make sure your configuration is correct and the generated ROM file can be opened.**

7. **(verilog users) Modify the *dtmf_recvr_core_test* testfixture and add some code to monitor the TDSP address bus and display the contents of the accumulator when your program exits and enters the tight loop.**

   You can use the hierarchy browser to find the TDSP accumulator so that you can reference it via your testfixture display routine.

8. **(vhdl users) Use the variable browser to find and trace the accumulator during simulation.**

9. **(both) Define a breakpoint on the address bus when your program exits and enters its tight loop to halt the simulation run.**

10. **Assuming all went well when you fired up the simulation environment, open a waveform viewing window to trace the TDSP program and data memory address, data, read and write strobes.**

11. **Resume the simulation and check that your program is operating as planned.**

Lab Exercise: Design Capture

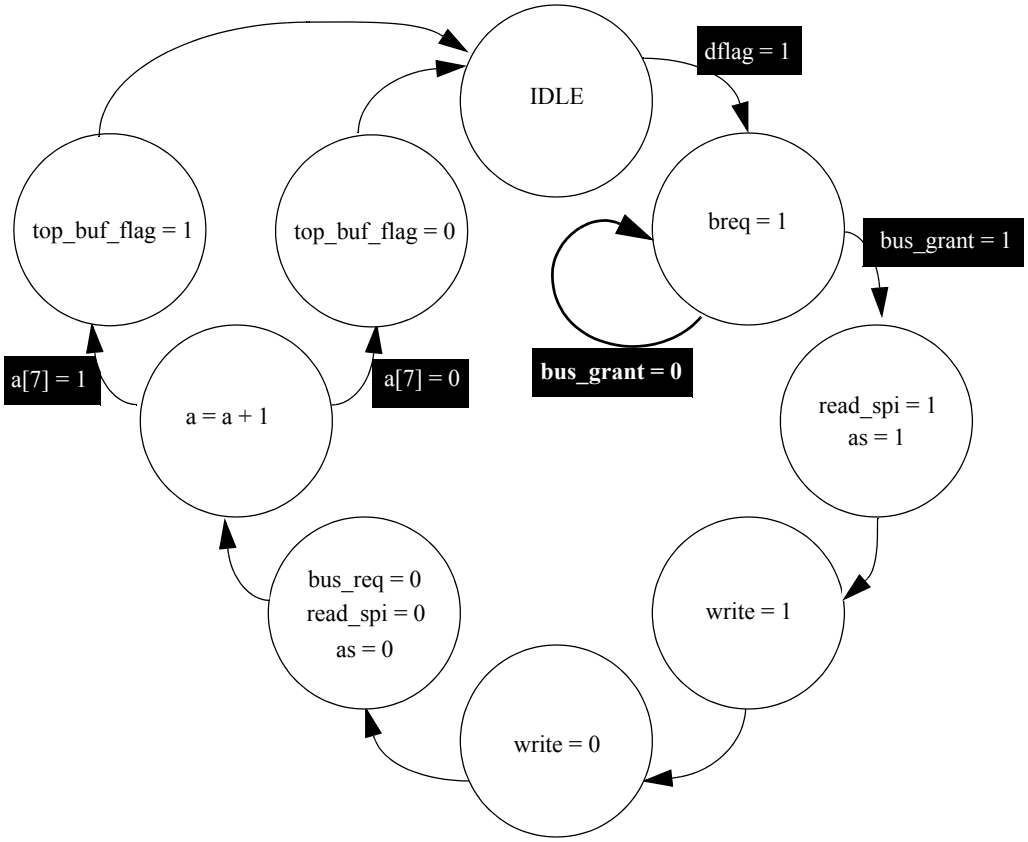
Circuit: DMA Controller FSM

‰ **Code the DMA controller given the description in the *DTMF Design Description* chapter. All data should be created in the work directory.**

```
Verilog Module: dma (dma.v)
VHDL Entity/Architecture: dma/rtl (dma.vhd)
```

Model the DMA controller as an implicit finite state machine. The DMA controller is clocked by a 25 Mhz positive edge triggered clock and the reset is asynchronous and active high. Upon reset all outputs of the DMA controller are cleared (0). When reset is inactive, the state machine shown in
Figure 0-23 is executed, beginning in the IDLE state.

Figure 0-23 DMA Controller State Machine Diagram



Note: Unless otherwise shown, all outputs remain hold their value during state transitions.

Turn in the following:

dma.v

simulation log file

Lab Exercise: Initial Synthesis

Circuit: DMA Controller FSM

1. **Synthesize the DMA controller circuit with no constraints.**

   A script has been provided to run the synthesis job. To invoke the synthesis job, type

   ```
   verilog : syn dma.v
   vhdl : syn dma.vhd
   ```

2. **Examine the results in the run directory, and then answer the listed questions.**

Table 4-1 Synthesis Report Files

syn.v, syn.vhd	Netlist File
syn.report.t	Timing Report
syn.report.p	Cost Report
syn.report.maxfanout	Loading Violation Report
syn.report.stats	Netlist Statistics
syn.report.ffcstr	Flip-Flop Constraint Report
syn.report.int.clock	Internal Clock Report
syn.report.summary	Summary Report

What is the longest path?

What is the maximum operating frequency?

What is the minimum slack time?

What is the total cost?

What is the width of the state register?

How many registers are there in the design?

Are there any loading violations?

Are there any timing violations?

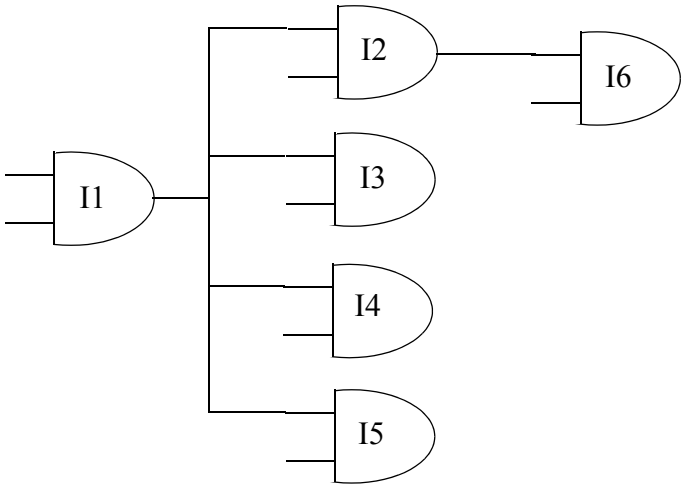
What is the maximum load seen at the input? Can this be easily determined?

What is the minimum load that can be driven? Can this be easily determined?

Lab Exercise: Delay Calculation

‰ **Calculate the maximum fall delay for I1 in the following circuit.**

Use the linear delay equation (3.1.2) on page 6-12 of *An Approach to Top Down Design*. Consult the library file for the characterization data for the NA210 cell (all instances are NA210's). Use the 5K_TLM wire model for wire load estimates.



RW = _____

CW = _____

DW = _____

CL = _____

DL = _____

DS = _____

DI = _____

DT = _____ + _____ + _____ + _____ = _____

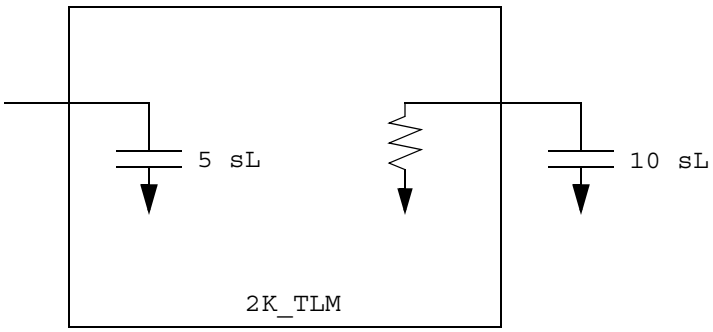
Lab Exercise: Constraint Derivation

Exercise: TDSP Constraint Derivation

Assume that the TDSP is placed and routed as a 10K hard macro. All the blocks in the design are less than 2K equivalent gates. Each module was synthesized separately and the following loading constraints were used:

```
input_maxload 5 -g
output_load 10 -g
wire_model -tree 2K_TLM
```

Figure 0-24 Block Level Synthesis Constraints



Consult the library data for any necessary data.

‰ **Answer the following questions (assume all gate inputs are 1 sL):**

How many fanouts are allowed for each input port?

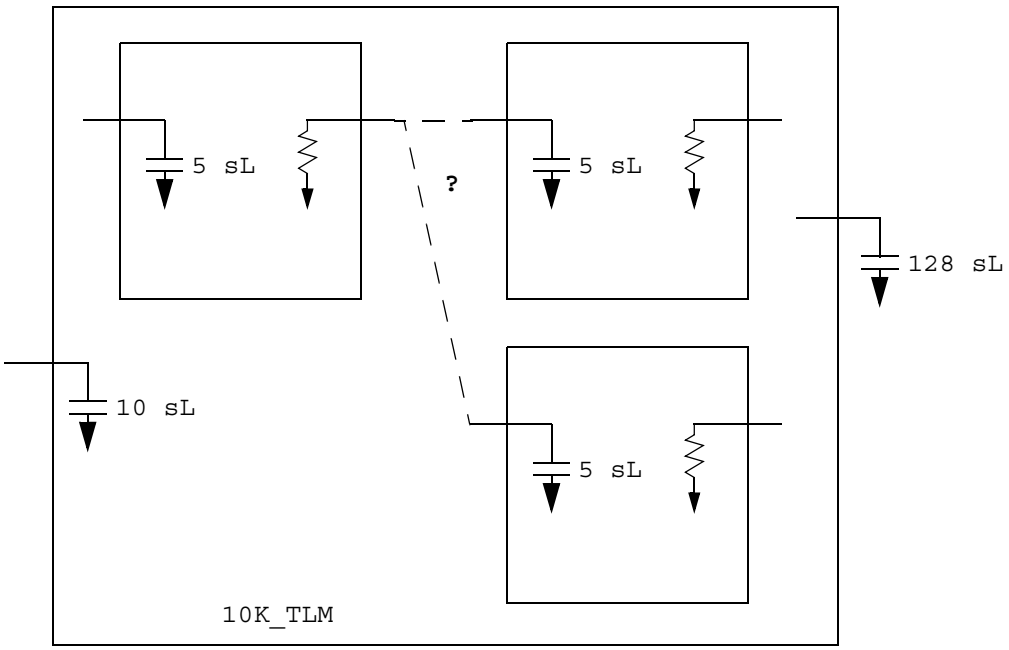
What will be the largest load seen at any input port?

What is the maximum load that can be driven by any output port?

Assume that the TDSP core was then run through synthesis at that gate level for pre-layout drive optimization. The following constraints were used:

```
preserve_boundary -tree -alternative driveopt
input_maxload 10 -g
output_load 128 -g
wire_model -tree 10K_TLM
```

Figure 0-25 Core Level Synthesis Constraints



Consult the library data for any necessary data.

‰ **Answer the following questions (assume all gate inputs are 1 sL):**

How many fanouts are allowed for block interconnect (connections from one block to another) before a buffer is placed?

What will the load be after adding the buffer?

What will be the largest capacitance seen by any internal module output port?

Circuit: DMA Controller FSM

1. **Given the circuit description, the synthesis library and the estimated size of the design, derive a set of global constraints.**

   These constraints should include all necessary boundary conditions (timing, loading, and resistance), wire models, library parameters, and clocking information.

   ```
   Clock: 25 MHz
   Reset: Active High
   Arrival Times: 2 ns (minimum) 5 ns (maximum)
   Required Times: 1 ns (minimum) 10 ns (maximum)
   Wire Model: 5K_TLM
   ```

2. **Enter the constraints in the *work/etc/dma.cst* file.**

3. **Synthesize the design using these constraints.**

   The *syn* script can be used; it automatically looks for the constraints in the *dma.cst* file.

What is the delay of the longest path?

What are the endpoints of this path?

What is the maximum operating frequency?

What is the minimum slack time?

What is the total cost?

What is the width of the state register?

How many registers are there in the design?

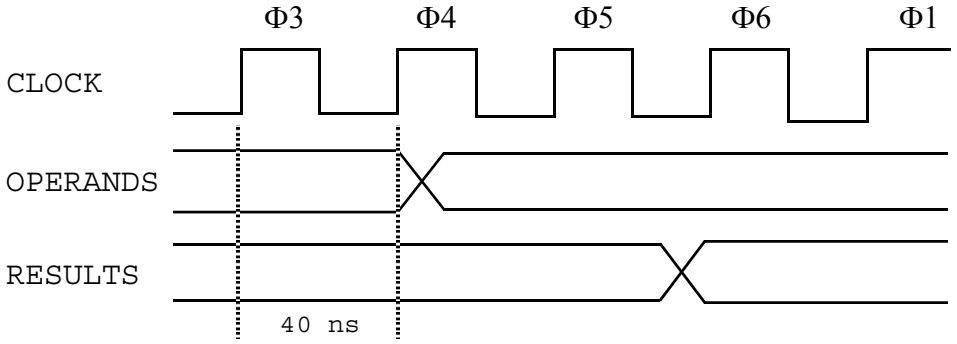
Are there any loading violations?

Are there any timing violations?

Lab Exercise: Timing Analysis

Circuit: Tiny DSP

The TDSP chip is clocked at 25MHz. It is fully synchronous with no internally generated clocks and no transparent latches. Instructions requiring the ALU or the multiplier allow 2 clock cycles for the result.



‰ **Use Synergy to perform timing analysis on the design.**

Use a 20K_TLM wire model for the top and a 2K_TLM wire_model for each lower level block. Run as many synthesis jobs as it takes to answer the following questions:

What is the longest path in the design?

What is the slack time of the design?

What is the maximum operating frequency?

Are there any timing violations?

How many synthesis runs were run?

Lab Exercise: Optimization Strategies

Circuit: DMA Controller FSM

‰ **Synthesize the *dma* design unit.**

A constraint file is provided in the *etc* directory. Constraints may be added to this file as required. Run as many synthesis jobs as it takes to answer the following questions:

What is the smallest implementation?

What is the slack time for this implementation?

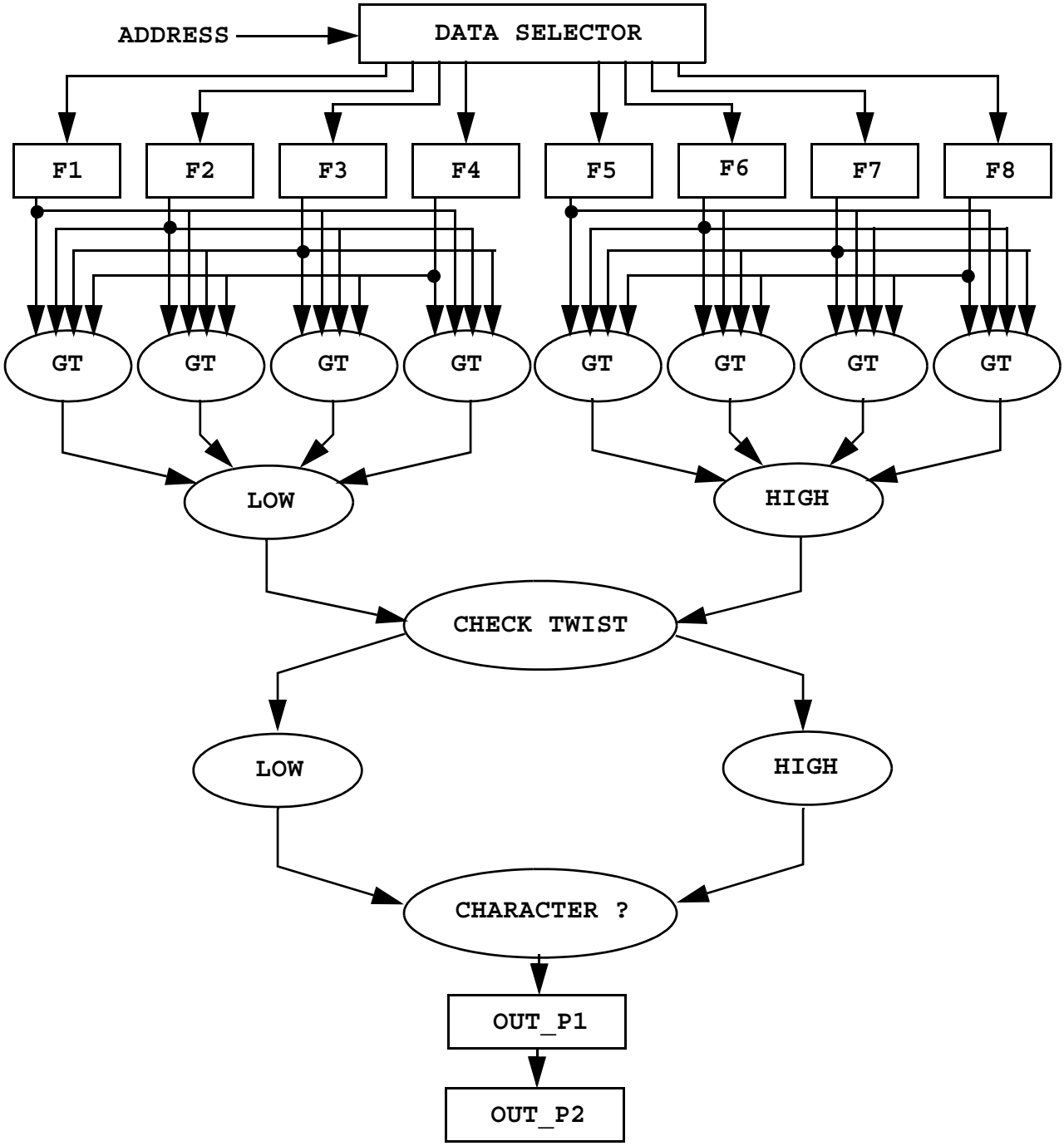
What is the fastest implementation?

What is the cost of this implementation?

Lab Exercise: Resource Sharing

Circuit: Results Character Conversion

The circuit topology for the *results_conv* block is:



Both the *GT* and *CHECK TWIST* functions contain complex operations.

1. Check the source code and analyze the behavior of this design unit.

Note: The GT (*gt_comp*) function calls do NOT happen in parallel.

2. Run a synthesizability check.

How many different types of complex operations are there in this design?

What are they?

How many total complex operations need to be performed?

Can any of the operations be shared?

3. Set up and run a synthesis run (use the SmartBlocks library)

How many complex operations were implemented?

What is the total area?

4. Add the following lines to your command file and re-run synthesis.

```
operator MINUS -width 16 -opcount 2 -partcount 1
operator MINUS -width 17 -opcount 2 -partcount 1
```

How many complex operations were implemented?

What is the total area?

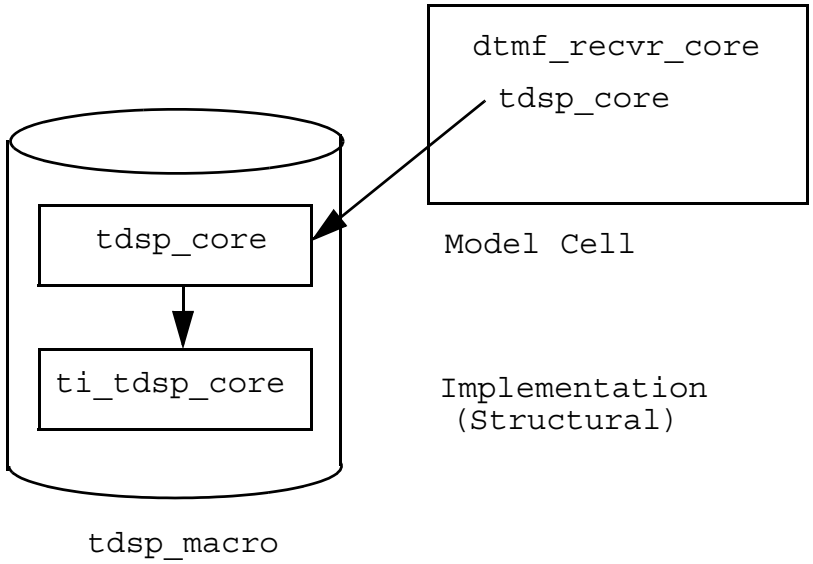
What is the difference in slack time?

Lab Exercise: Macro Libraries

Circuit: DTMF Receiver Core System

‰ **Create a macro library that contains a *tdsp_core* model cell and a technology dependent implementation (*ti_tdsp_core*).**

The library should be called *tdsp_macro*. The hierarchy of the implementation should be maintained during synthesis runs; no drive optimization is required.



Lab Exercise: Test Insertion

Circuit: Bus Arbiter FSM

1. Insert full scan test logic into the bus arbiter module.

The test ports are defined as follows:

```
scan_input: scan data input
scan_enable: active high scan enable
scan_output: scan data output
```

How many elements are in the scan chain?

What is the default order of the elements?

2. Run test insertion again and reverse the scan chain order:

What is the new order of the elements?

What constraint did you use to do this?

11

Chip-Level Assembly Implementation

Overview

Chip assembly is the process of bringing all the gate-level design blocks together for functional and timing verification, design rule check, pattern generation, and either ASIC vendor sign-off or in-house place and route. It is assumed that all design blocks, embedded blocks, and test blocks have been implemented (RTL) and in some cases functionally verified.

Full-chip functional verification is done during this phase of the design process. This part of the design process is very compute intensive. Therefore, the verification strategy must be well defined and testbenches written and in place. Verification of the design will require exhaustive simulations at multiple levels of abstraction and therefore automated regression techniques should employed as well as network “task brokering” to make use of all available resources.

Figure 11-1 shows the detailed process diagram and Figure 11-2 shows the accompanying project schedule (for a commercial ASIC sign-off project).

Figure 11-1 Chip Assembly and Sign-Off Process Diagram

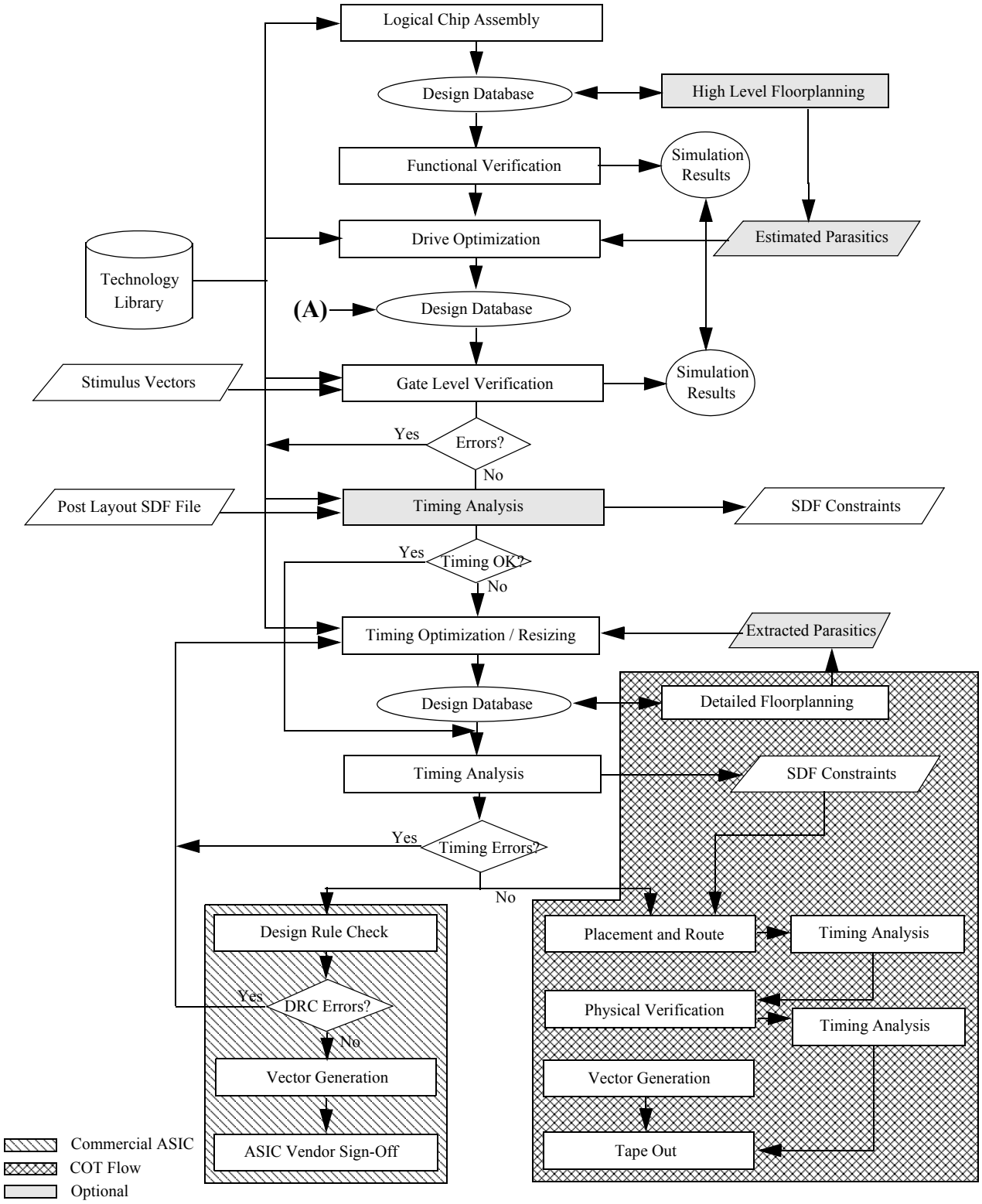
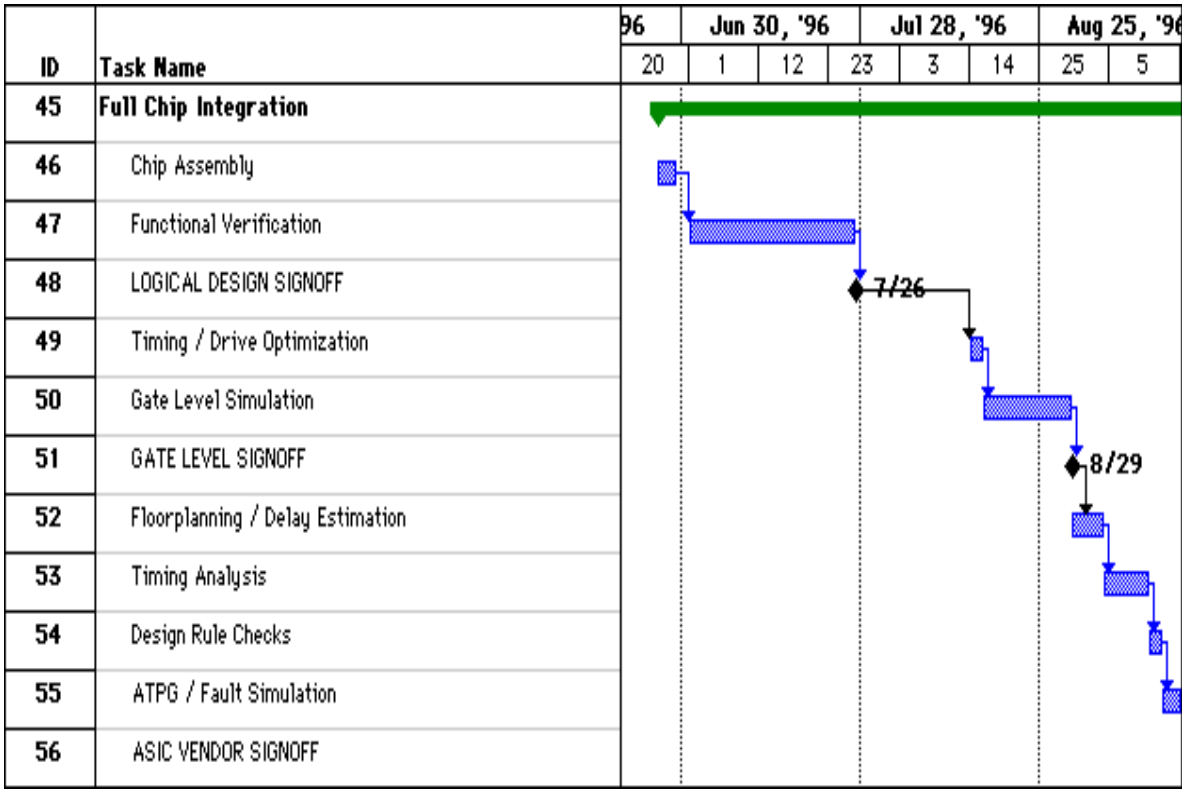


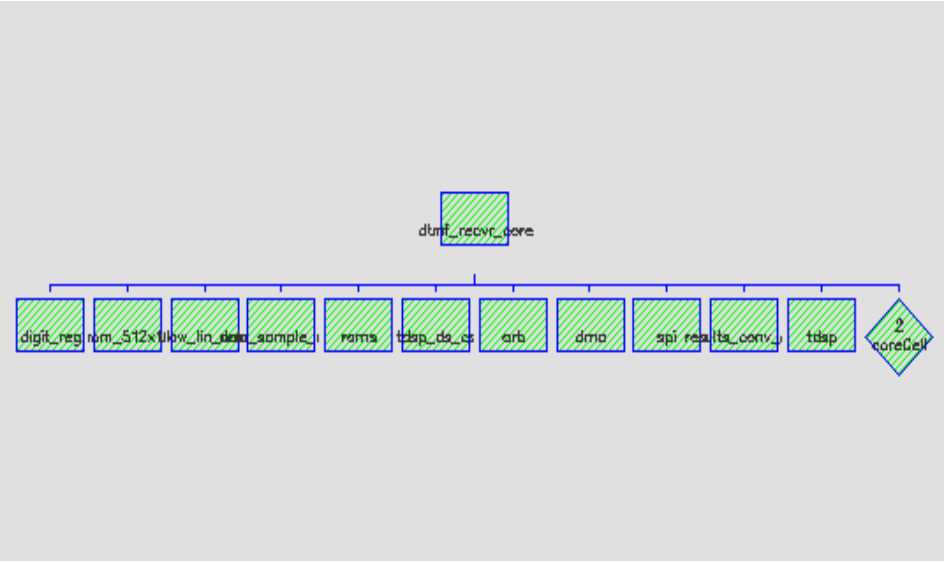
Figure 11-2 Chip Assembly and ASIC Sign-off Schedule



Logical Chip Assembly

Figure 11-3 shows the logical hierarchy of the DTMF core. Once the core logic assembly is complete, additional test logic (NAND trees, JTAG, etc.) can be added to the top-level netlist along with the I/O pads. A level of hierarchy should be created for the core as well as for test logic, so they can be easily isolated and preserved during drive optimization and resizing runs.

Figure 11-3 DTMF Core Logic Hierarchy



Chip-Level DFT Synthesis and Insertion

At the top level of the design, any DFT structures which are part of the chips I/O or are external to other blocks of the design can now be designed or automatically generated, and inserted into the netlist. Examples of such DFT objects are: HDL macro functions for the boundary scan register (BSR) cells, BIST structured such as controllers and LFSRs/MISRs, or an access collar around a large core block. If any of the DFT structures are inserted as HDL they will, of course, need to go through block level logic synthesis.

Scan Chains

In the case of the DTMF, the scan chains were automatically connected at the top level using the synthesis tool. The scan chain input and outputs are defined below.

Figure 11-4 Scan Chain I/O Assignments

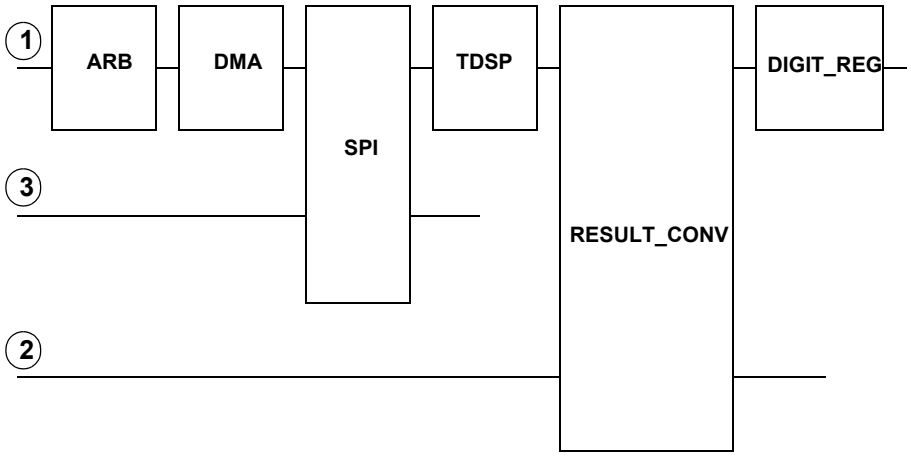
Scan Chain	Scan Input	Scan Output	Scan Enable
1	scan_input_1	scan_output_1	scan_enable
2	scan_input_2	scan_output_2	scan_enable
3	scan_input_3	scan_output_3	scan_enable

Figure 11-5 shows the constraints that were used to affect the I/O connection and scan chain ordering and Figure 11-6 show the resulting scan chain ordering.

Figure 11-5 DFT Constraints

```
dft_number_chains 3
dft_scan_polarity_inv -invert no
dft_donot_scan -scope tdsp_ds_cs

dft_scan_segment
   scan_input_1
   arb
   dma
   results_conv/1
   spi/1
   tdsp_core
   digit_reg
   scan_output_1

dft_scan_segment
   scan_input_2
   results_conv/2
   scan_output_2

dft_scan_segment
   scan_input_3
   spi/2
   scan_output_3
```

Figure 11-6 Scan Chain Ordering



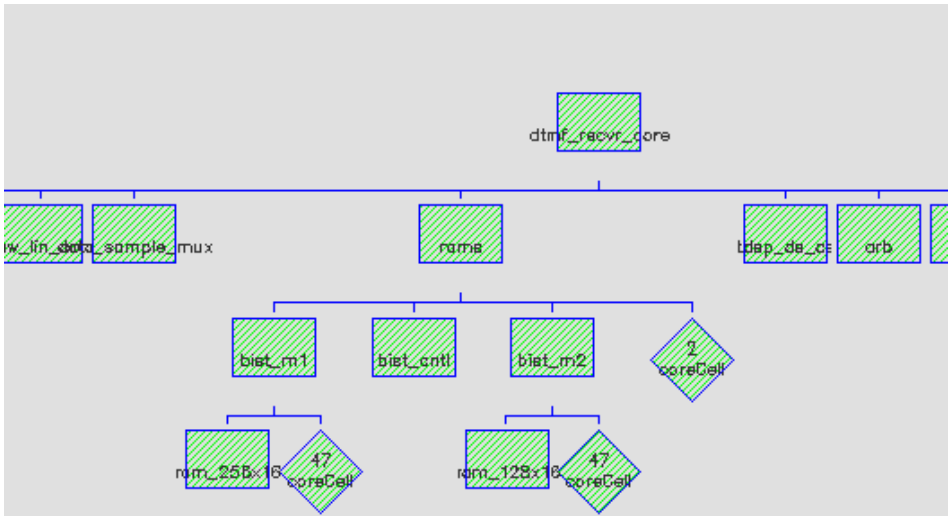
RAM BIST

Often, RAM BIST structures can be automatically generated according to the RAM parameters (e.g., single or multi-ported, RAM address and word widths). The BIST control logic can also be automatically generated. Whether they are automatically generated, or designed by hand, these test structures are typically inserted into the netlist as HDL and connected to the RAM block. The HDL can then go through block level synthesis.

Access collars are implemented either as multiplexed logic, which multiplexes the I/O of the embedded block to the devices primary I/O pins for test, or as a boundary scan ring around the embedded block. Access collars may be automatically generated or designed by hand, and may be done using HDL or special cells (e.g., in the case of a boundary scan collar).

In the case of the DTMF, a RAMBIST tool was used to automatically generate test access collars and BIST controllers for the two RAMs. These modules were then synthesized using constraints that were also generated by the RAMBIST tool. Figure 11-7 below shows the design hierarchy of the RAM block. Note each RAM is instantiated in a BIST access collar (bist_m1 and bist_m2) and the BIST control logic is instantiated as a separate logical block (bist_cntl).

Figure 11-7 RAM Logical Hierarchy



Boundary Scan and TAP Controller

For boundary scan structures, netlist insertion of the Boundary Scan Register (BSR) logic for each I/O cell and the TAP controller can be performed. This includes connection of the added nets between the BSR logic, I/O cells, TAP controller, and the other top-level blocks of the design.

Some ASIC vendors provide hard-macro I/O cells in their libraries, which contain the necessary 1149.1 BSR logic in the I/O cell. This is the desired implementation for boundary scan as it generally achieves the best area and timing performance. The alternative, when such hard-macro I/O cells are not available, is to insert soft macros (HDL that can later be synthesized) for the BSR logic.

In the case of the DTMF, no boundary scan was required.

Once all of the DFT structures that need to be inserted at the top level of the design are completed, chip-level floorplanning can be done and final budgets set for constraints to logic synthesis. The next step is block level design and synthesis of the test structures. Here, any DFT structures which are internal to the top level blocks, for example internal scan paths, are inserted.

Functional Verification

Functional verification of the system at the RTL level should occur as soon as possible. Testbenches and stub models can be written to model peripheral system behavior so that core modules can be verified early in the design process.

In the case of the DTMF system, the DSP was simulated at the block level so that the full instruction set could be verified. This was discussed in Chapter 10, “Block-Level Implementation,”. Chip level functional simulation was used to verify that the system bus interfaces and system peripherals were functioning properly and to debug the system firmware (Goertzel algorithm).

Hardware Verification

Functional verification of the major design blocks (TDSP, RCC, DMA, SPI, and ARB) was performed during block-level implementation. Because of this, RTL verification of the DTMF hardware required only integration testing of these modules to verify bus interface protocols. This saved a significant amount of time as very little debug was required at this level of integration. The bulk of the verification effort was in software verification.

Configurations were used to define block design hiearchies. These configurations could then be synthesized in which case the resulting netlist (hierarchical or flat) would be the gate-level representation of the original configuration. For example, Figure 11-8 shows the RTL configuration for the tdsp_core.

Figure 11-8 TDSP Core RTL Configuration

tdsp_core_rtl.f

```
./include/timescale.h
./alu_32.v
./data_bus_mach.v
./decode_i.v
./execute_i.v
./mult_32.v
./port_bus_mach.v
./prog_bus_mach.v
./accum_stat.v
./update_ar.v
./tdsp_core.v
+incdir+./include
```

The synthesized netlist is tdsp_core.vs and contains the entire design hierarchies for the tdsp_core. Figure show the RTL and gate level simulation configurations for simulating the tdsp_core.

Figure 11-9 TDSP Simulation Configurations

tdsp_core_test_rtl.fs

```
+define+rtl+no_trace+turbo+3+twin_turbo
./include/timescale.h
./tdsp_core_test.v
-f ./configs/tdsp_core_rtl.f
```

tdsp_core_test_gate.fs

```
+define+gate+no_trace+turbo+3+twin_turbo
./include/timescale.h
./tdsp_core_test.v
./lib/tdsp_core.vs
```

The following table shows a complete listing of the DTMF configurations:

Figure 11-10 DTMF Configurations

Configuration File	Type	Contents
arb_test_rtl.f	simulation	arb_test.v arb.v
arb_test_gate.f	simulation	arb_test.v lib/arb.vs
dma_test_rtl.f	simulation	dma_test.v dma.v
dma_test_gate.f	simulation	dma_test.v lib/dma.vs
spi_test_rtl.f	simulation	spi_test.v spi.v
spi_test_gate.f	simulation	spi_test.v lib/spi.vs
results_conv_test_rtl.f	simulation	results_conv_test.v results_conv.v
resutls_conv_test_gate.f	simulation	results_conv_test.v lib/results_conv.vs

Configuration File	Type	Contents
tdsp_core_rtl.f	design	./alu_32.v ./data_bus_mach.v ./decode_i.v ./execute_i.v ./mult_32.v ./port_bus_mach.v ./prog_bus_mach.v ./accum_stat.v ./update_ar.v ./tdsp_core.v
tdsp_core_test_rtl.f	simulation	-f configs/tdsp_core_rtl.f test_tdsp_core.v
tdsp_core_test_gate.f	simulation	lib/tdsp_core.vs test_tdsp_core.v
rams_rtl.f	design	ram_128x16.v ram_256x16.v rams.v
rams_gate.f	design	ram_128x16.v ram_256x16.v lib/bist_cntl.vs lib/bist_m1.vs lib/bist_m2.vs lib/rams.vs
dtmf_recvr_core_rtl.f	design	-f configs/tdsp_rtl.f -f configs/rams_rtl.f arb.v spi.v dma.v digit_reg.v data_sample_mux.v results_conv.v tdsp_ds_cs.v rom_512x16.v ulaw_lin_conv.v dtmf_recvr_core.v

Configuration File	Type	Contents
dtmf_recvr_core_gate.f	design	-f configs/tdsp_gate.f -f configs/rams_gate.f lib/tdsp.v lib/arb.v lib/spi.v lib/dma.v lib/digit_reg.v lib/data_sample_mux.v lib/results_conv.v lib/tdsp_ds_cs.v lib/rom_512x16.v lib/ulaw_lin_conv.v lib/dtmf_recvr_core.vs -v lib/stdCell.v
dtmf_recvr_core_test_rtl.f	simulation	-f configs/dtmf_recvr_core_rtl.f dtmf_recvr_core_test.v
dtmf_recvr_core_test_gate.f	simulation	-f configs/dtmf_recvr_core_rtl.f dtmf_recvr_core_test.v

Software Verification

The Goertzel algorithm, modeled at the behavioral level during the high-level system design phase, now needs to be modeled at the assembly code level. Verification of the algorithm was done by loading the program in the ROM and loading linear PCM data samples in the data sample memory. A PCM generator was written in ‘C’ so that the PCM data (linear and ulaw compressed) could be generated easily. This saved significant simulation cycles because the serial/parallel interface and the ulaw to linear conversion module did not need to be simulated. Figure shows a portion of the DTMF testfixture with compiler directive to conditionally compile the serial data input versus direct memory load. Note the different hierarchical paths to the RAM for RTL and gate level (with BIST logic).

Figure 11-11 DTMF Testfixture Code Fragment

```
`ifdef INIT_SAMPLE_RAM
`ifdef RTL
defparam test.top.RAMS_INST.DSRAM.initfile = "etc/pcm256.txt" ;
`endif
`ifdef GATE
defparam test.top.RAMS_INST.M1.M1.initfile = "etc/pcm256.txt" ;
`endif
`endif

/*
* generate spi interface, data to be shifted out resides in "signalMem"
 */
`ifdef ENABLE_SPI
always
begin
        #1000 spi_fs = 1'b1 ;
        #1000 spi_fs = 1'b0 ;
        j = 7 ;
        spi_data = ulawPcm[j] ;
        for (i = 0 ; i <= 7 ; i = i + 1)
        begin
                #1000 spi_clk = 1'b1 ;
                #1000 spi_clk = 1'b0 ;
                if (i <= 6)
                j = j - 1 ;
                spi_data = ulawPcm[j] ;
        end
        signalAddress = signalAddress + 1 ;
        if (signalAddress == signalSize)
                signalAddress = 0 ;
         #107000;
end
`endif
```

An assembler was also written which output the program codes in memory-loaded verilog format. Figure 11-12 shows a portion of the Goertzel algorithm modeled in assembly.

Figure 11-12 Goertzel Algorithm Fragment Modeled in Assembly

```
;
; **** dtmf dft starts here
;
dfst:
;
; get scale factor
;
          lac       d_scale
          sacl      scale
;
; zero delay elements
;
ddz:    zac
          lark      ar1,dl_len                    ; using all 16 delay elements
          lark      ar0,(dla1+base_page1) ; start with dla1
ddzl:   sacl    *+,ar1
          banz    ddzl,*-,ar0
;
; load ar0 with agc count
          lar           ar0,agc_cnt
          sar           ar0,agc_ptr
;
; load ar0 with data sample memory pointer
          lark    ar0,(ds_ptr+base_page0) ; using all 16 delay elements
;
; load ar1 with transform length
          lark    ar1,xform_len                 ; start with dla1
;
; dft loop, index here goes from 0-(N-2)
;
ddftl:  lac
          sar     *+,0,ar1ar0,frm_ptread sample
          sar           ar1,len_ptr
          sacl    xk                            ; move to xk
;
; this calculates the inner loop for 697hz
; tdsp difference equation:
; dla0 = xk*scale + 2*recf1*dla1 - dla2
;
; actual difference equation:
; d(n) = x(n) + 2*cos(2*pi*k/N)*d(n-1) - d(n-2)
          apac    recf1
          apac
          sach    dla1,15
```

Drive Optimization

Once the logical chip assembly is complete, a synthesis timing report for the entire chip (including pads) should be run to uncover any timing or loading issues. If there are any violations, a drive optimization run can be done to resolve them. A wire model for the top level should be used for proper wire load estimation. This wire model should be selected based on the “size” of the chip not the gate count of the design. Ideally, a custom wire model should be generated for each placement region by the floorplanning tool. This was done for the DTMF; two wire models were generated (TDSP_WIRE and RCC_WIRE). Design modules that do not have timing or loading problems should be preserved. Also, map without the strongest buffer in the library. This will force buffer trees to get built with buffers that can be “upsized” later when wire load data is back-annotated.

Figure 11-13 and Figure 11-14 show an example of block interconnect before and after drive optimization. The inserted buffers are a result of loading estimated based on statistical wire models (generated during high level floorplanning) and the pin capacitances for each destination. Figure 11-17 shows the results of resizing these buffers based on actual back-annotated wire parasitics.

Figure 11-13 Core Level Block Interconnect

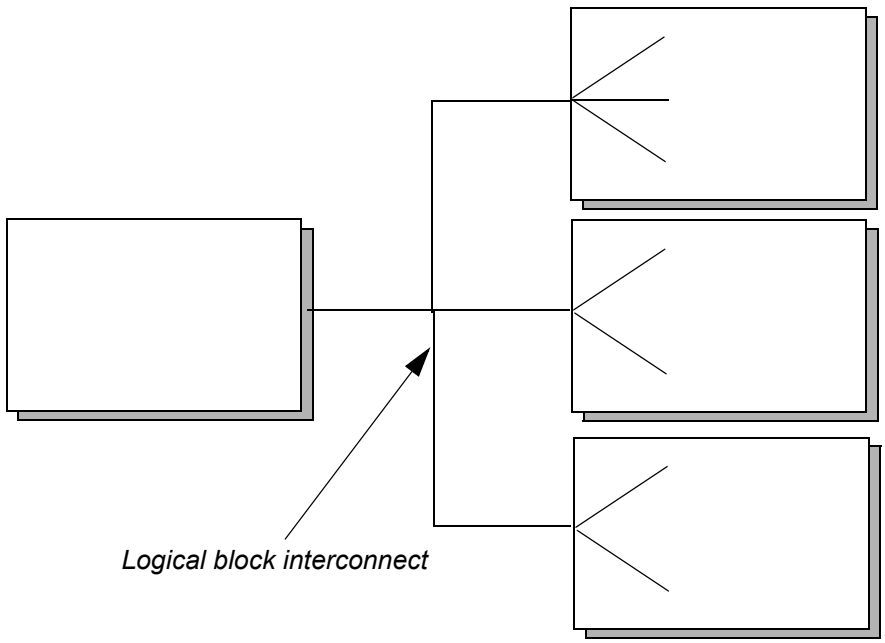
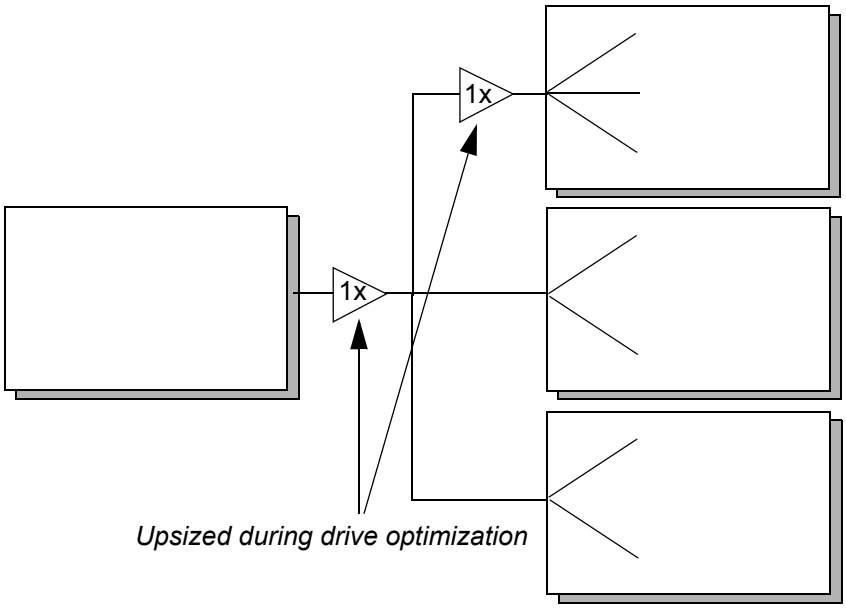


Figure 11-14 Buffered Interconnect



If timing requirements are not met at the chip level, some or all of the design blocks may need to pass through synthesis again with updated constraints. The design hierarchy for certain modules may need to be to flattened to meet design specifications. This process will, however, alter the netlist and require a new physical hierarchy to be generated.

Figure 11-15 shows the constraints for the final synthesis pass. These constraints are similar to the previous passes, expect that the wire model selected is based on the expected chip die size, and Synergy is run in drive optimization mode.

Figure 11-15 Drive Optimization Constraints

```
wire_model FULL_CHIP

clock clk -rise 0.000000 -fall 25.000000 -period 50.000000

timing -required -rise 40.000000 -g
timing -required -fall 40.000000 -g

preserve_boundary -tree tdsp -alternative resizing
preserve_boundary -tree results_conv -alternative resizing
preserve_boundary -tree dma -alternative driveopt
preserve_boundary -tree spi -alternative driveopt
preserve_boundary -tree tdsp_ds_cs -alternative driveopt
preserve_boundary -tree data_sample_mux -alternative driveopt
preserve_boundary -tree digit_reg -alternative driveopt
preserve_boundary -tree ulaw_lin_conv -alternative driveopt

maintain -tree rams
maintain -tree rom_512x16

scope tdsp
wire_model TDSP_WIRE

scope results_conv
wire_model RCC_WIRE

synthesis_options -steps synthesizer schematic
synthesis_options -alternative driveopt
synthesis_options -priority timing
```

Gate Level Verification

Mixed-level simulation allows full chip simulation to occur earlier in the design process. As the design blocks are completed, their gate level implementations are included in the simulation and verified with the rest of the design. By incrementally integrating completed design blocks into the full chip simulation, integration problems can be detected at an early stage. Once all the design blocks are completed, a full chip gate level simulation must be performed.

The full chip gate-level simulation results should be compared with the RTL simulation results. If problems are detected, they can be isolated through interactive mixed-level simulation. If these problems are isolated and resolved quickly, they can be rectified without significant impact to the full chip verification process. An automated regression simulation process to verify the design changes and a synthesis “make” utility to update the gate level database with the design/constraint changes are imperative to this process.

Once the design has been fully verified, delays calculated from estimated wiring parasitics can be annotated into the gate-level simulation to begin timing verification.

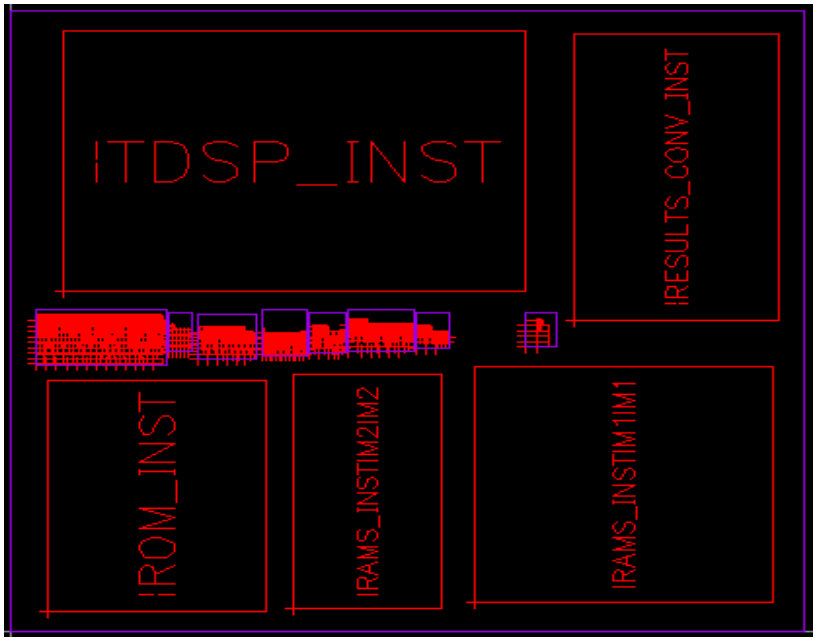
Detailed Floorplanning

Before delay estimation and timing verification, it is useful to have a detailed floorplan for the overall design. Some vendor tools include the floorplanning job in the delay estimator and design rule checking tools. This leads to much more accurate delay and load prediction. At this point, I/O location should also be known and included as part of the floorplan. If the placement and routing is being done through a customer-owned tooling environment (COT) an initial placement should be done at this point. For the DTMF, all physical design was done COT.

A hierarchical approach can be used for designs where reuse is important, designs that lend themselves to physical partitioning or are of a size that requires it. In this case, each block would be placed and routed separately, an abstract generated for the block and then the design would be routed at the top level using a block routing tool.

It was determined that only two hierarchical design blocks should be created for the DSP (containing the multiplier and the ALU) and the results converter for design reuse requirements. Figure 11-16 shows the resulting physical design hierarchy. These blocks were implemented and discussed in Chapter 10, “Block-Level Implementation,”.

Figure 11-16 DTMF Physical Hierarchy



Timing Optimization / Resizing

Once the design has been run through the vendor floorplanning and/or placement tool, wire load information can be back-annotated into Synergy for timing optimization and final design rule check.

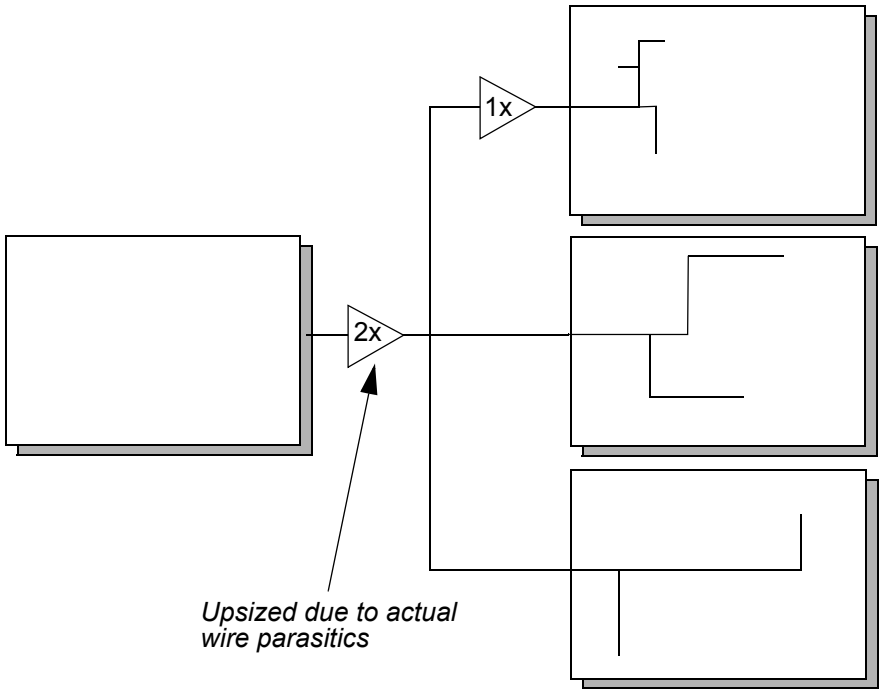
First run Synergy with the timing report alternative:

```
synthesis_options -alternative presynthesis
```

This will generate a full set of synthesis reports and uncover any remaining timing or loading issues. If problems exist, then isolate the blocks that have the problems and run the design back through Synergy in either drive optimization or resizing mode.

If the vendor has an ECO capability, then use the resizing mode so the netlist topology will not change. Constrain the synthesis tools to only work on the problem areas by hierarchy management constraints.

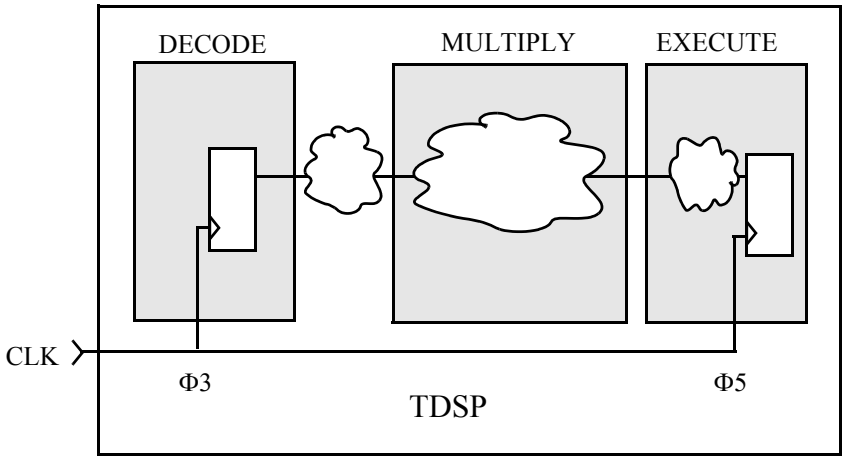
Figure 11-17 Buffer Resizing



Static Timing Analysis

For the receiver design, a large portion of the timing analysis at both the block and chip level occurs within the synthesis tools. The synthesis tool can work with statistical wire models, back-annotated wire loads (SPF), or back-annotated delays (SDF). Figure 11-18 indicates the longest path through the DTMF as reported by the synthesis tool.

Figure 11-18 DTMF Longest Path



As indicated, the longest path passes through the multiplier and is completely contained within the TDSP. The actual delay is 67.94 ns with a single cycle slack of -28.54ns. However, the multiplier is a two cycle path (the operands are registered in phi_3 and the results is registered in phi_5) and therefore needs to be checked against a two-cycle clock constraint. The two cycle slack time would therefore be +11.46ns. Because of this, the synthesis constraints were modified and the timing analysis performed again. During this run, both the multiplier output and the ALU output were disabled so that only the single cycle paths would be analyzed. Figure 11-19 shows the full path trace of the longest multiplier path.

Figure 11-19 DTMF Longest Path (Detailed)

Total	Inc	FO	Load		Cell	Instance
R(0.00)	0.00 <-- clock				FD4QS(CP)	TDSP.DECODE.mod350852
F(2.72)	2.72	12		0.30	FD4QS(CP=>Q)	TDSP.DECODE.mod350852
R(3.77)	1.05	3		0.08	IVP(A=>Z)	TDSP.mod498021
F(4.42)	0.65	2		0.08	ND2P(A=>Z)	TDSP.mod488461
R(5.33)	0.91	3		0.06	IV4(A=>Z)	TDSP.mod501437
F(6.13)	0.80	1		0.06	ND2(A=>Z)	TDSP.mod488418
R(7.20)	1.07	11		0.24	IV4(A=>Z)	TDSP.mod504901
F(8.05)	0.85	7		0.19	ND2P(A=>Z)	TDSP.mod488466
R(8.94)	0.89	2		0.04	IV4(A=>Z)	TDSP.mod501715
F(9.90)	0.96	1		0.02	AO2(A=>Z)	TDSP.mod509179
R(12.10)	2.20	7		0.16	AO3(D=>Z)	TDSP.mod493251(***op_b[2]***)
R(13.50)	1.41	3		0.06	OR3P(A=>Z)	TDSP.MPY_32.mod333422
R(15.22)	1.71	3		0.06	OR3P(C=>Z)	TDSP.MPY_32.mod333434
R(16.93)	1.72	3		0.06	OR3P(C=>Z)	TDSP.MPY_32.mod333446
R(18.65)	1.72	3		0.06	OR3P(C=>Z)	TDSP.MPY_32.mod333458
R(20.37)	1.71	3		0.06	OR3P(C=>Z)	TDSP.MPY_32.mod333470
F(21.62)	1.25	4		0.10	AO7(A=>Z)	TDSP.MPY_32.mod332511
R(22.75)	1.13	1		0.02	ND2P(A=>Z)	TDSP.MPY_32.mod332607
F(23.65)	0.90	4		0.09	AO37(D=>Z)	TDSP.MPY_32.mod332442
R(25.28)	1.63	20		0.42	IVP(A=>Z)	TDSP.MPY_32.mod007140
R(26.97)	1.69	2		0.04	OR3P(C=>Z)	TDSP.MPY_32.mod007140
F(29.28)	2.32	16		0.34	AO34(D=>Z)	TDSP.MPY_32.mod006374
R(30.71)	1.43	2		0.05	AO2(D=>Z)	TDSP.MPY_32.mod008392
F(31.46)	0.75	2		0.05	IVP(A=>Z)	TDSP.MPY_32.mod008394
F(33.72)	2.26	3		0.07	EO3(C=>Z)	TDSP.MPY_32.mod006928
R(35.43)	1.71	3		0.07	EN3P(A=>Z)	TDSP.MPY_32.mod006045
F(37.88)	2.45	3		0.07	EO3(B=>Z)	TDSP.MPY_32.mod004633
R(40.24)	2.36	2		0.04	EN3(B=>Z)	TDSP.MPY_32.mod004659
R(41.78)	1.54	4		0.10	OR2(B=>Z)	TDSP.MPY_32.mod001302
F(42.53)	0.75	1		0.04	ND3(A=>Z)	TDSP.MPY_32.mod001312
R(44.41)	1.88	5		0.13	ND4P(C=>Z)	TDSP.MPY_32.mod001370
F(45.06)	0.65	1		0.05	ND4P(A=>Z)	TDSP.MPY_32.mod001375
R(46.72)	1.67	5		0.13	ND4P(A=>Z)	TDSP.MPY_32.mod001380
F(47.37)	0.65	1		0.05	ND4P(A=>Z)	TDSP.MPY_32.mod001385
R(49.04)	1.67	5		0.13	ND4P(A=>Z)	TDSP.MPY_32.mod001390
F(49.68)	0.65	1		0.05	ND4P(A=>Z)	TDSP.MPY_32.mod001395
R(51.35)	1.67	5		0.13	ND4P(A=>Z)	TDSP.MPY_32.mod001400
F(51.99)	0.65	1		0.05	ND4P(A=>Z)	TDSP.MPY_32.mod001405
R(53.66)	1.67	6		0.13	ND4P(A=>Z)	TDSP.MPY_32.mod001410
F(54.09)	0.43	1		0.04	ND3(C=>Z)	TDSP.MPY_32.mod001482
R(55.40)	1.30	1		0.04	ND3P(A=>Z)	TDSP.MPY_32.mod001485
R(57.15)	1.76	3		0.06	EOP(B=>Z)	TDSP.MPY_32.mod000933
R(58.84)	1.69	2		0.04	OR3P(C=>Z)	TDSP.MPY_32.mod333698
R(60.26)	1.42	3		0.06	OR2(B=>Z)	TDSP.MPY_32.mod332491
R(62.95)	1.00	1		0.02	AO32(D=>Z)	TDSP.MPY_32.mod336006
F(64.49)	1.53	1		0.06	AO6(C=>Z)	TDSP.MPY_32.mod336050
R(65.35)	0.87	1		0.02	IV4(A=>Z)	TDSP.MPY_32.mod336052
F(66.53)	1.18	1		0.03	AO31(B=>Z)	TDSP.EXECUTE.mod439211
R(67.54)	1.01	1		0.06	IVP(A=>Z)	TDSP.EXE-
CUTE.mod439213(***mpy_result[29]***)						
R(67.54)	0.00				FD10Q(D)	TDSP.EXECUTE.mod430041

The actual worst case (critical) path is the address decode of the results character converter memory space during indirect addressing mode. Figure 11-20 shows this path.

Figure 11-20 Actual Critical Path

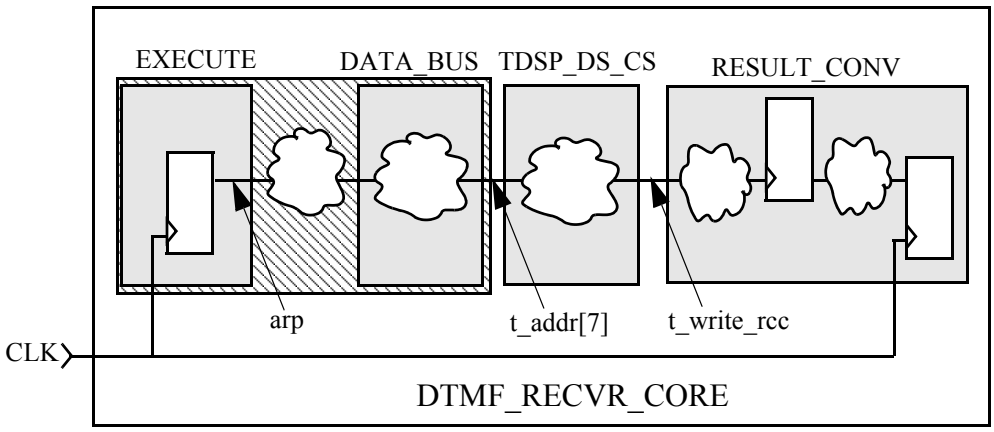


Figure 11-21 shows the full timing report for this path. The critical path originates in the instruction execution module (EXECUTE) within the TDSP and ends in the results character converter module (RCC). The slack time for this path is -9.00 ns. This path actually generates the internal *t_write_rcc* clock for the eight spectral component registers when an address in the results converter is decoded during an indirect write operation. The following alternatives may be utilized to improve this path:

- Modify the RTL and re-synthesize.
- Modify the constraints and re-synthesize from the RTL level.
- Modify the constraints and perform timing optimization at the gate-level.

For the DTMF, it was determined that gate-level timing optimization would be performed first to see how if the critical path timing could be improved; RTL synthesis would be re-run only if necessary.

Figure 11-21 DTMF Critical Timing Path

```
Total      Inc   FO  LoadCell          Instance(net=>net)
R(  0.00)  0.00 <-- clock     FD2Q(CP)
TDSP_INST.EXECUTE_INST.UPDATE_AR_INST.mod278259(clk)
R(  2.94)  2.94 11    0.34  FD2Q(CP=>Q)
TDSP_INST.EXECUTE_INST.UPDATE_AR_INST.mod278259(arp)
F(  4.39)  1.45 5     0.12  MUX21(S=>Z)TDSP_INST.mod488155
F(  5.89)  1.50  3    0.08  MUX21(B=>Z)TDSP_INST.mod488452
R(  7.88)  1.99  3    0.14  EOP(B=>Z) TDSP_INST.mod488079(addrs_in[0])
F(  8.49)  0.61  1    0.02  V4(A=>Z)         TDSP_INST.mod504319
R( 10.31)  1.82  3    0.09  ND4(A=>Z) TDSP_INST.mod491167
F( 11.06)  0.76  2    0.06  IVP(A=>Z)  TDSP_INST.mod506987
R( 12.44)  1.37  2    0.06  ND2(A=>Z) TDSP_INST.mod488479
F( 13.16)  0.73  1    0.04  IVP(A=>Z)  TDSP_INST.mod501117
R( 14.55)  1.39  3    0.07  ND3P(A=>Z) TDSP_INST.mod491271
F( 16.03)  1.48  1    0.02  EO(A=>Z)   TDSP_INST.mod500419
R( 17.58)  1.55  2    0.08  AO4(C=>Z) TDSP_INST.mod488051(addrs_in[6])
R( 19.00)  1.42  1    0.03  OR2P(B=>Z) TDSP_INST.mod488510
R( 20.48)  1.48  1    0.02  EO(B=>Z)   TDSP_INST.mod504439
F( 21.69)  1.21  1    0.06  AO4(A=>Z) TDSP_INST.mod488365(addrs_in[7])
R( 22.65)  0.96  1    0.12  IV4(A=>Z)  TDSP_INST.DATA_BUS_MACH_INST.mod320584
F( 23.28)  0.63  2    0.16  IV8(A=>Z)
TDSP_INST.DATA_BUS_MACH_INST.mod320582(t_addrs[7])
R( 24.42)  1.14  1    0.03  ND2P(A=>Z) TDSP_DS_CS_INST.mod002340
R( 25.74)  1.32  1    0.03  BF2T16(A=>Z) TDSP_DS_CS_INST.mod002965
R( 26.82)  1.08  3    0.18  BF2T8(A=>Z) TDSP_DS_CS_INST.mod002957
F( 27.74)  0.92  1    0.02  NR4P(A=>Z) TDSP_DS_CS_INST.mod002306(t_write_rcc_a)
F( 29.09)  1.35  1    0.02  MUX21(A=>Z) mod006321(t_write_rcc)
F( 30.53)  1.44  1    0.22  BF1T2(A=>Z) RESULTS_CONV_INST.mod823513
R( 32.36)  1.83  129  3.45  IV16(A=>Z) RESULTS_CONV_INST.mod748820
R( 34.72)  2.36  3    0.06  FD2Q(CP=>Q) RESULTS_CONV_INST.mod780517(scan_output_2)
F( 35.68)  0.96  8    0.17  ND3(C=>Z) RESULTS_CONV_INST.mod769349
R( 37.35)  1.67  5    0.12  ND2(B=>Z) RESULTS_CONV_INST.mod749821
F( 38.12)  0.77  3    0.06  IVP(A=>Z)  RESULTS_CONV_INST.mod785865
R( 40.07)  1.95  4    0.11  ND3(C=>Z) RESULTS_CONV_INST.mod771585
F( 41.32)  1.25  9    0.20 ND2(A=>Z)   RESULTS_CONV_INST.mod771591
R( 43.11)  1.79  24   0.51 IVP(A=>Z)   RESULTS_CONV_INST.mod771633
F( 44.07)  0.95  1    0.02 AO2(A=>Z)   RESULTS_CONV_INST.mod792385
R( 45.54)  1.48  1    0.02 AO36(E=>Z) RESULTS_CONV_INST.mod771831
F( 46.56)  1.02  1    0.02 AO38(A=>Z) RESULTS_CONV_INST.mod771880
R( 48.20)  1.64  1    0.06 AO36(E=>Z) RESULTS_CONV_INST.mod771901
```

Figure 11-22 shows the results of the structural re-optimization.Both pessimistic timing numbers (worst case process corner) and wire models were utilized during this run; a positive slack timing margin was our goal. Indeed the slack time was increased from -9.00 ns to 0.50 ns. At this point the logical netlist passes timing analysis as is ready for detailed floorplanning and placement.

Figure 11-22 Critical Path Resynthesis

```
Total     Inc   FO  Load  Cell(path)      Instance(net)
R(  0.00)  0.00 <-- clock     FD2Q(CP)     TDSP_INST.EXECUTE_INST.UP_AR_INST.mod001859(clk)
F(  1.94)  1.94   2   0.04   FD2Q(CP=>Q)TDSP_INST.EXECUTE_INST.UP_AR_INST.mod001859
F(  3.25)  1.31   4   0.13   BF1T2(A=>Z)  TDSP_INST.EXECUTE_INST.mod020971
F(  4.66)  1.42   5   0.10   MUX21(S=>Z) TDSP_INST.mod008140
F(  6.16)  1.50   3   0.08   MUX21(B=>Z) TDSP_INST.mod008144
R(  7.50)  1.34   2  0.04    MUX21N(S=>Z)TDSP_INST.mod008149(addrs_in[0])
R(  9.00)  1.50   4   0.09   OR2(B=>Z)    TDSP_INST.mod008155
R( 10.20)  1.20   2   0.06   OR2(A=>Z)    TDSP_INST.mod008388
R( 11.45)  1.25   3   0.08   OR2(A=>Z)    TDSP_INST.mod008392
R( 12.68)  1.23   1   0.02   MUX21N(S=>Z)TDSP_INST.mod008765
F( 13.55)  0.87   2   0.04   MUX21N(B=>Z) TDSP_INST.mod008769(addrs_in[6])
F( 14.77)  1.22   1   0.04   OR2(A=>Z)    TDSP_INST.mod009030
R( 16.00)  1.23   1  0.02    MUX21N(S=>Z)TDSP_INST.mod009036
F( 17.63)  1.63   1   0.22   MUX21N(A=>Z)TDSP_INST.mod009050(addrs_in[7])
R( 18.46)  0.83   1   0.22   IV16(A=>Z)     TDSP_INST.DATA_BUS_MACH_INST.mod571953
F( 19.03)  0.57   2   0.25   IV16(A=>Z)
TDSP_INST.DATA_BUS_MACH_INST.mod571951(t_addrs[7])
R( 19.83)  0.80   6   0.14   IV16(A=>Z)     TDSP_DS_CS_INST.mod1370866
F( 20.71)  0.88   1   0.02   NR2P(A=>Z)   TDSP_DS_CS_INST.mod1370278(t_write_rcc_a)
F( 22.09)  1.39   1   0.04   MUX21(A=>Z) mod1533429(t_write_rcc)
R( 23.10)  1.00   2   0.05 IVP(A=>Z)       RESULTS_CONV_INST.mod748820
R( 25.46  2.36   3   0.06   FD2QS(CP=>Q) RESULTS_CONV_INST.mod780517(scan_output_2)
F( 26.42)  0.96   8   0.17   ND3(C=>Z)     RESULTS_CONV_INST.mod769349
R( 28.10)  1.67   5   0.12 ND2(B=>Z)     RESULTS_CONV_INST.mod749821
F( 28.88)  0.79   3   0.08   IVP(A=>Z)     RESULTS_CONV_INST.mod785865
R( 30.56)  1.68   4   0.11   ND3P(C=>Z)   RESULTS_CONV_INST.mod771585
F( 31.82)  1.25   9   0.20   ND2(A=>Z)     RESULTS_CONV_INST.mod771591
R( 33.61)  1.79   24  0.51   IVP(A=>Z)     RESULTS_CONV_INST.mod771633
F( 34.56)  0.95   1   0.02   AO2(A=>Z)     RESULTS_CONV_INST.mod792385
R( 36.04)  1.48   1   0.02   AO36(E=>Z)    RESULTS_CONV_INST.mod771831
F( 37.05)  1.02   1   0.02   AO38(A=>Z)    RESULTS_CONV_INST.mod771885
R( 38.70)  1.64   1   0.06   AO36(E=>Z)    RESULTS_CONV_INST.mod771901
R( 38.70)  0.00              FD10Q(D)        RESULTS_CONV_INST.mod780659

The Slack Time of Long Critical Path 1 is  0.50 ns (Req. Time: 39.20 ns)
```

Formal timing analysis was then done to generate timing path constraints for the placement tool. Figure 11-23 shows an example command file for Pearl that checks all sequential paths for a given clock domain. The multiplier and ALU outputs are defined as two-cycle paths and path tracing on test logic was disabled. This command file is setup to run batch timing analysis and will be re-used at every junction in the process where timing analysis needs to be performed (post-placement, post-route, post-verification, etc.). Note that the Pearl command file would be modified to read the appropriate SDF file at each timing check point.

Figure 11-23 Pearl Command File

```
ReadTechnology lib/std_cell.tch
ReadTimingModels lib/std_cell.mod
ReadVerilog lib/dtmf_recvr_core.vs
TopLevelCell dtmf_recvr_core
ReadSDF -process max ../etc/dtmf_recvr_core_synth.sdf
Clock -cycle_time 40 clk 0 20
Input * clk ^ 2 10 2 10
Constraint * clk ^ 2 2 2 2
MultiCycleNode TDSP_INST.TDSP_CORE_INST.mpy_result[31:0] 2 max
MultiCycleNode TDSP_INST.TDSP_CORE_INST.alu_result[32:0] 2 max
Blockpath reset *
Blockpath scan_input_1 *
Blockpath scan_input_2 *
Blockpath scan_input_3 *
Blockpath bist_clk *
Blockpath rcc_sclk *
Blockpath go *
Blockpath done *
Blockpath compstat *
Blockpath biston *
Blockpath bist *
Blockpath test_mode *
Blockpath scan_enable *
Blockpath * scan_output_1
Blockpath * scan_output_2
Blockpath * scan_output_3
TimingVerify
FindMinCycleTiming
ShowPossiblities 1
```

The results of this run (post-synthesis) are shown in Figure 11-24. The back-annotated SDF file was generated by Synergy and therefore the timing numbers should correlate. The maximum clock frequency for the design is 25.67 MHz.

Figure 11-24 Initial Timing Analysis Results

Possibility 1:
Setup constraint slack ***1.05ns*** RESULTS_CONV_INST.mod780671 (FD10Q D v -> CP ^)
Clk edge: clk ^ -> RESULTS_CONV_INST.mod780671.CP ^ at 0.00ns + Tcycle = 40.00ns
Setup time: ***0.40ns***
Data edge: clk ^ -> RESULTS_CONV_INST.mod780671.D v at ***38.55ns***
Required cycle time: ***38.95ns*** (1.0 cycle path)

	Delay	Delta	Node	Cell
*	0.00ns	1.94ns	clk ^	FD2Q
	1.94ns	1.31ns	TDSP_INST.EXECUTE_INST.w024354 v	BF1T2
*	3.25ns	1.42ns	TDSP_INST.w338871 v	MUX21
*	4.67ns	1.50ns	TDSP_INST.w500319 v	MUX21
*	6.16ns	1.34ns	TDSP_INST.w487988 v	MUX21N
*	7.50ns	1.50ns	TDSP_INST.addrs_in[0] ^	OR2
	9.00ns	1.20ns	TDSP_INST.w488575 ^	OR2
	10.20ns	1.25ns	TDSP_INST.w488244 ^	OR2
	11.45ns	1.23ns	TDSP_INST.w488117 ^	MUX21N
	12.68ns	0.87ns	TDSP_INST.w490792 ^	MUX21N
	13.55ns	1.22ns	TDSP_INST.addrs_in[6] v	OR2
	14.77ns	1.23ns	TDSP_INST.w488566 v	MUX21N
*	16.00ns	1.63ns	TDSP_INST.w490794 ^	MUX21N
	17.63ns	0.83ns	TDSP_INST.addrs_in[7] v	IV16
	18.46ns	0.57ns	TDSP_INST.DATA_BUS_MACH_INST.w326421 ^	IV16
	19.03ns	0.80ns	t_addrs[7] v	IV16
	19.83ns	0.88ns	TDSP_DS_CS_INST.w509672 ^	NR2P
*	20.71ns	1.39ns	t_write_rcc_a v	MUX21
	22.09ns	1.00ns	t_write_rcc v	IVP
*	23.10ns	2.36ns	RESULTS_CONV_INST.w230421 ^	FD2QS
	25.46ns	0.96ns	scan_output_2 ^	ND3
*	26.42ns	1.67ns	RESULTS_CONV_INST.w155441 v	ND2
	28.10ns	0.79ns	RESULTS_CONV_INST.w154981 ^	IVP
*	28.88ns	1.68ns	RESULTS_CONV_INST.w175142 v	ND3P
	30.56ns	0.72ns	RESULTS_CONV_INST.w154936 ^	IVP
*	31.28ns	2.12ns	RESULTS_CONV_INST.w181824 v	ND2P
	33.41ns	0.98ns	RESULTS_CONV_INST.w155736 ^	IVP
	34.39ns	1.31ns	RESULTS_CONV_INST.w162202 v	AO2
	35.70ns	0.66ns	RESULTS_CONV_INST.w182726 ^	AO36
*	36.36ns	1.38ns	RESULTS_CONV_INST.w157271 v	AO38
	37.74ns	0.81ns	RESULTS_CONV_INST.w162642 ^	AO36
	38.55ns		RESULTS_CONV_INST.w098728 v	

Design Rule Check

If a commercial ASIC vendor is being utilized, then, at this point, the netlist should be run through the vendor process design rule checker to verify that no test rules, electrical rules, or design rules have been violated. This may require a netlist translation. The netlist format will typically be Verilog, EDIF, or an internal format.

If any problems exist, they must be fixed before the vendor will sign-off on the design. For loading issues, wiring information can be back annotated from the floorplan and used during a drive optimization or resizing run in the synthesis tool. If the vendor DRC tools support an ECO capability and the number of violations is few, then a resizing run is recommended. Otherwise, a drive optimization run should be used. Serious problems may require going back and re-synthesizing certain modules with more pessimistic constraints and/or wire models.

In the case of the DTMF design, an internal standard cell library was used for all physical design and therefore DRC/ERC checks were done during physical verification.

Test Development and Validation

Once all DFT structures have been designed and verified, test development and validation can be done. The sections below discuss what tests need to be developed and the process for automatic test pattern generation (ATPG) and test vector verification.

ASIC Test Vector Suite

The following list describes a typical test suite that can be used for ASIC characterization and production manufacturing tests. The tests are listed in the order they would most likely be run for final production testing. In actual production testing, some tests may be run at more than one voltage point to test that the chip meets established margins—for process, voltage, and temperature—in its intended system application.

- Continuity Check

  Checks for electrical contact between tester channels, DUT (Device Under Test) fixture and the packaged part, or in the case of wafer probe, between the micro-probes and the die pads.

  This test is run automatically at the tester.

- Process Monitor Test

  Used as a measure of the IC fab process this test is intended to monitor the device under test to assure that it lies within a 6-sigma range of the known fab process.

  This is typically done by placing a special process monitor structure on the chip, or by performing a scan ring flush, where the propagation delay of a pulse through the process monitor or scan ring is measured and compared against expected delay times. In the case of a scan ring flush, the scan implementation uses a two-phase clocking style (e.g. LSSD) where both the master latch and slave latch clocks are asserted, making both latches transparent and using the scan chain to form a delay path from the chips scan in pin to the scan out pin.

  The process monitor test is typically developed by hand, using a timing simulation, and requires characterization and verification of the delay times to correlate with the process delay parameters.

- DC Parametric Tests

  These tests are used to check that all I/O pins can drive and receive the correct DC level for both a logic one and a logic zero.

  For output, tristate, open-collector and bidirectional I/O pins, tests that drive out a logic one and tests that drive a logic zero are required. The tester measures that each output pin can be driven to V_OH and V_OL. For input and bidirectional pins, input threshold tests which drive a logic zero and logic one are required. IEEE 1149.1 Boundary Scan will help facilitate generation and application of DC parametric tests, particularly the output tests, where the boundary scan ring can be used to directly drive out the desired logic values. For the input threshold tests the 1149.1 SAMPLE/PRELOAD or EXTEST instructions can be used to capture the input values for testing. Devices without boundary scan may require a NAND tree structure to be inserted into the design for input tests. In this case, V_IL and V_IH are measured at an I/O pin connected to the output of the last NAND gate in the tree.

  DC parametric tests are usually developed by hand, although they can be automated if IEEE 1149.1 Boundary Scan is implemented.

- Hi-Z and Leakage Tests

  These tests place the ASIC's tristate and bidirectional I/O into a Hi-Z state to allow testing and measurement of high-impedance state and I/O input leakage current.

  The device is first put into the Hi-Z state and then the tests apply a logic one and a logic zero to all I/O pins. Any active pull-up or pull-down circuits on the I/O must be disabled for these tests, as they will produce a DC current in the chips static state. Any other sources of DC current must also be eliminated for these tests, such as bus or I/O drive contention, floating nodes, or embedded RAM/Macros that are not normally be in a static state (see Chapter 4, "Design for Test Methodology" for further details).

- I_{ddq} Tests

  These test perform further static I_{ddq} current measurements using test vectors specifically generated or graded for I_{ddq} coverage.

  All sources of DC current must be eliminated for I_{ddq} tests, such as active pull-up or pull-down circuits on the chips I/O, bus or I/O drive contention, floating nodes, or embedded RAM/macros that are may not normally be in a static state.

  I_{ddq} tests can either be developed using ATPG tools, or can be a subset of existing functional tests which have been graded for I_{ddq} coverage. Several I_{ddq} test vectors are recommended in order to achieve high I_{ddq}

fault coverage. Typically between 5 to 100 I_{ddq} test vectors are run for production tests. The limiting factor is usually the test time required to run them. I_{ddq} tests can take a long time to run due to the time required for switching currents to settle (usually in the milliseconds range) and depending on how the current measurements are taken -- typically hundreds of millisecond if the PMU of the tester is used.

- DFT Logic Tests

  These tests are targeted at functional testing of the DFT logic -- such as scan path integrity tests, TAP controller tests, and boundary scan functionality tests.

  These tests are generated by hand in some cases, or in other cases may be automatically generated along with the system logic tests.

- System Logic Tests

  These test vectors are targeted at achieving high fault coverage of the chips core functional logic and its I/O.

  These tests generally consist of one or more of the following types of tests:

  - ATPG Stuck-At Tests

    Automatically generated on full or partial scan designs.

  - Built-In Self-Tests (BIST)

    Generated for designs that have either RAM or logic BIST capabilities.

  - Functional Tests

    Usually provided for non-scannable logic, or in some cases for at-speed testing, or binning purposes. For at-speed functional testing, the tester must be capable of applying broadside test vectors at the ASICs system clock rate, at a minimum, or better at even slightly higher rates. Functional tests are generated by hand.

- Delay Path Tests

  Delay path tests may be provided for testing and measuring critical timing paths.

  These tests are generated by hand in some cases, in other cases they may be automatically generated by ATPG tools. In general, delay path tests are not dependent on the chip testers capability to apply broadside test vectors at the ASICs system clock rate.

- AC Parametric Tests

  AC parametric tests are at-speed test vectors which are targeted at testing the speed characteristics (i.e., set-up and hold times) of the I/O paths of the ASIC.

  These are generally functional tests that are generated by hand using a timing simulation. In some cases they may not be run as part of the production tests for the ASIC and may be developed only for device characterization purposes.

Functional Test Development

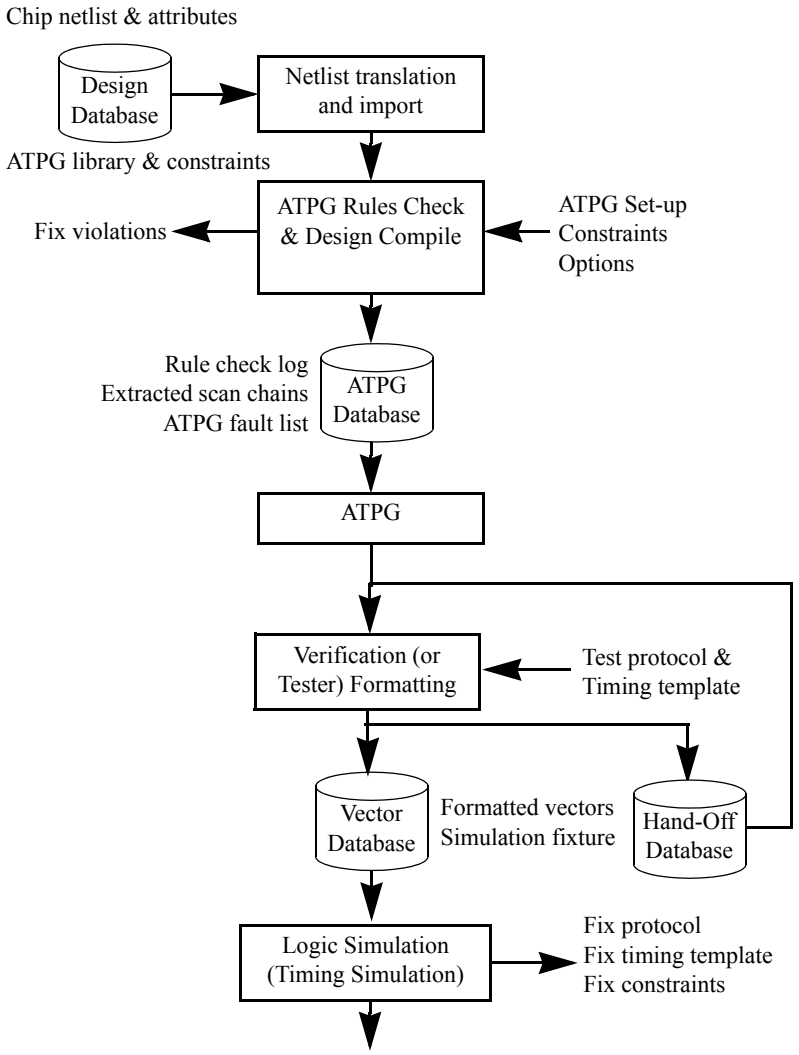
Blocks that do not have any structured testability will need to be tested with hand-generated functional tests. The manual test vectors can be graded using composite fault grading techniques. With composite fault grading, an incremental fault dictionary is maintained while running each set of test vectors. The initial dictionary may be created by injecting faults in the entire design and then as each set of vectors is applied, the fault dictionary is updated. It is passed to each successive job as input and only the "new" detected faults are updated thus producing a composite fault grade for the design.

Automatic Test Pattern Generation

Figure 11-25 show a typical flow for ATPG test vector generation. This same flow can be used, in general, for any tests that are automatically generated. For example, stuck-at tests, I_{ddq} tests, and delay path tests.

The inputs to APTG are a gate level netlist of the chip and a design library for ATPG. ATPG may also need design attributes and any ATPG constraints, which it uses to analyze the design during DFT rules checking and to help generate scan test vectors. The design attributes tell ATPG about features of the design, for example what I/O pins are used for system and test clocks, for scan-in/scan-out in, and for scan shift enable or test mode enables. The ATPG constraints provide ATPG with pre-set boundary values it can use to help generate test vectors. For example, if there is a test mode enable pin which needs to be asserted during test (to enable DFT features for test) its asserted logic value is given as a constraint to ATPG.

Figure 11-25 ATPG and Verification Flow



Once the design netlist has been imported into ATPG, ATPG and design for testability rules checks are done. Often, during the rules checking, the scan chain orders of the chips scan paths are extracted. These can be used to verify the expected scan orders, and/or fed into the ATPG and vector verification steps. After passing the rules check, ATPG can be run on the design.

Test Vector Verification

It is not sufficient to send the ATPG vectors generated by the ATPG tool directly to the vendor. These vectors need to be verified against the design through simulation. Figure 11-25 shows the steps for formatting the vectors for simulation and performing logic simulations (and for some vectors, timing simulations). Simulation will require a simulation protocol, and a timing template, describing how to apply the vectors to the design under simulation. Each test suite will then have a set of formatted test vectors and a test fixture, which runs the vectors under simulation, associated with it. The formatting and simulation fixture can often be done by the ATPG tools, or separate tools. In some cases the simulation fixtures are written by hand, for example for had generated tests.

The vector simulation may be a long and tedious process, since the vectors are serial and the scan chains get to be quite long. An ASIC with 5000 flip-flops will need slightly more than 5000 clock cycles to simulate just one ATPG "vector" provided the ASIC was initialized. Just 200 ATPG vectors will mean over a million simulation cycles in this case. While most problems will probably be found within the first few ATPG vectors, that cannot be assumed. An alternate method for simulating ATPG vectors is to bypass the serial loading mechanism and instead parallel load the vectors directly into the flip-flops using the force functions available in the simulators. This greatly reduces the simulation time but requires a simulation environment that can do vector conversion and provide a force and release mechanism. Even with the parallel-load mechanism a few serially loaded vectors should be simulated.

Tester Formatting and Hand-Off

Once test vectors have been verified in simulation they can be formatted for the target tester —or into a format as required by the ASIC vendor—for design hand-off. This requires another formatting step similar to that for verification, including a test protocol definition and a timing template which describe how to apply the test vectors to the design during testing on the chip tester.

The following should also be considered when formatting test vectors for design hand-off:

- Test Partitioning

  It may often be desirable to partition large test sets, such as functional tests or large scan vector sets, in order to deal with the data more efficiently, either at the tester or in formatting. If test sets are partitioned, care must be taken to assure that each partition is independent of other partitions. Each partition should be capable of

being executed on its own and should not require some other partition to be run first in order to work correctly.

- Formatting Timesets

  Care must be taken during formatting in order to provide a timeset that the target tester will accept. Often, testers expect all tester cycles to be formatted to the same timeset. Some testers will allow multiple timesets for each test vector load, but only up to a maximum number of timesets.

  The tester will also require certain minimum allowable times for various waveforms in the timeset. For example it may require that there be at least 10ns from the beginning of a tester cycle before any inputs can change, or it may require that there be at least 5ns between the edges of different waveform. For outputs, certain strobe window requirements may have to be met. For example, the tester may require a minimum strobe window width of 25ns, during which time the tester will measure the output voltage on the pin and compare it to the expected logic value. Some testers also require that there exist a “dead” cycle between direction changes for bidirectional and tristate pins. In this case, neither the ASIC under test nor the tester is driving during the dead cycle between the direction change; both are at a high impedance state.

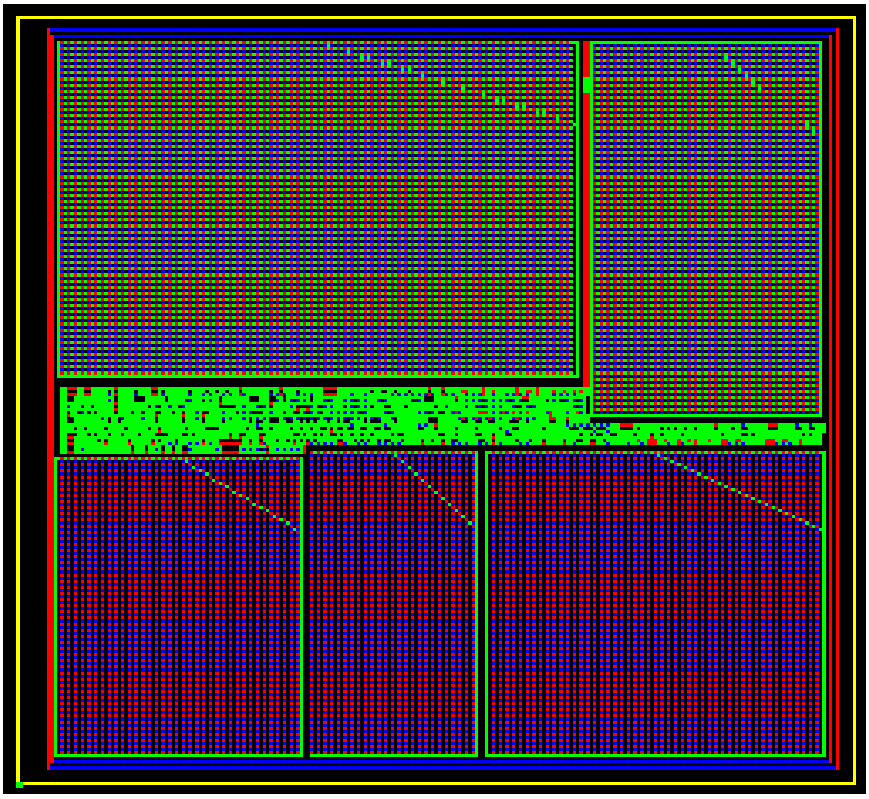
- Tester Format Validation

  After formatting the tests into the tester (or ASIC vendor) format it is sometimes desirable to convert the formatted tests back into a simulation format and re-simulate the test vectors. This will help to assure that there are no errors being introduced by the tester formatting process, and that the tests will run correctly on the tester.

Final Placement and Route

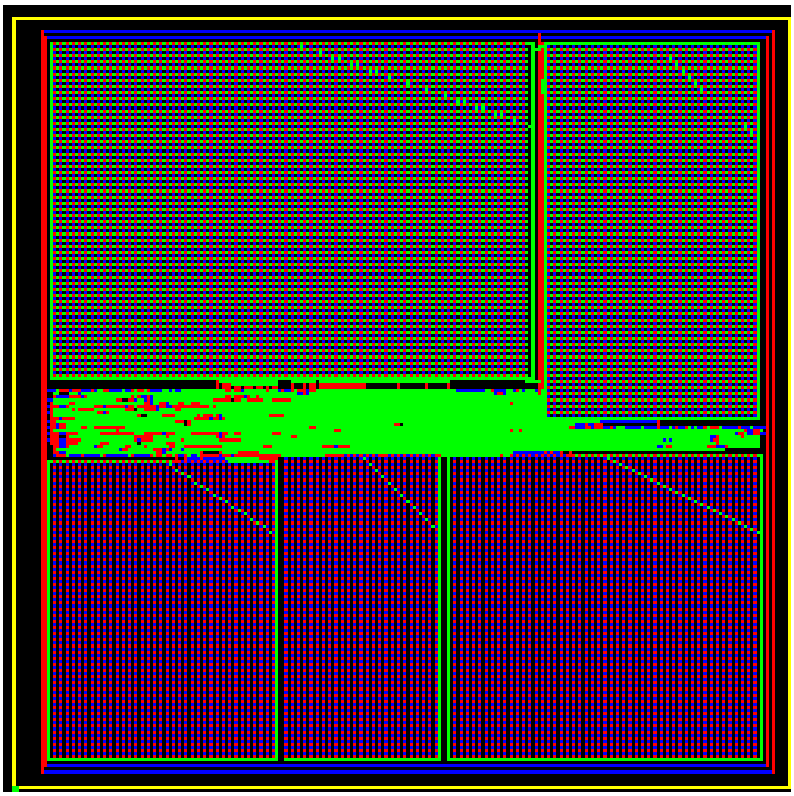
Once the gate level design is verified to be functionally correct and all timing constraints are met, final placement and routing can be done. Since hierarchy was only created for the DSP and the RCC block, the rest of the "soft" blocks were flattened and placed within top level regions by the cell placement tool using the timing path constraints.

Figure 11-26 DTMF Top-level Placement



The macro blocks were implemented standalone as well. This leaves a relatively simple top level place and route. Figure 11-27 shows the top level route.

Figure 11-27 DTMF Top-level Route



The only placement that typically occurs at this point is through ECOs due to netlist changes such as resizing or clock tree generation. Power and ground rings and stripes are routed, followed by any special nets (such as clocks), and then rest of the design is routed.

Once the design is routed, physical verification is performed to ensure that no design or electrical rules are violated. If verification passes, then the design is then ready for tape-out.

References

A

Programming with the TDSP

TDSP Instruction Set

Addressing mode notes:

Direct Addressing Mode - Direct addressing forms the data memory address by concatenating seven bits of the instruction word with the data page pointer. This implements a paging scheme in which each page contains 128 words.The physical address is built by appending the immediate address with the current data page pointer, for example:

```
{DP, OPCODE[6:0]}
```

Indirect Addressing Mode - Indirect addressing forms the data memory address from the least significant eight bits of one of the two auxiliary registers, AR0 or AR1. The auxiliary register pointer (ARP) selects the current auxiliary register for indirect address generation. The auxiliary registers can automatically post increment or post decrement in parallel with the execution of any indirect instruction to permit single-instruction-cycle manipulation of data structures in memory. Specific support for indirect addressing is included in the assemble as:

```
*     address AR(ARP)
*+    address AR(ARP), post increment AR(ARP)
*-    address AR(ARP), post decrement AR(ARP)
```

Immediate Addressing Mode - Immediate instructions derive data from part of the instruction word rather from the data RAM. This can be thought of as a shorthand for loading constants to certain registers. Note that the typical immediate data size is an 8 bit constant, although certain instructions can handle lager constants. For reference, most immediate data instruction opcodes end in “k”.

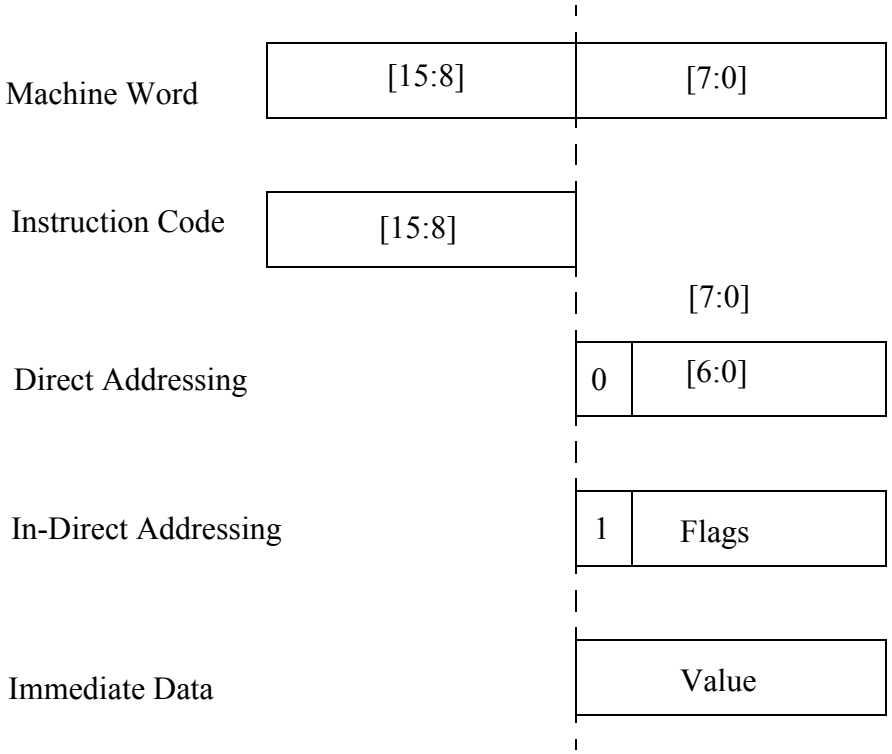
For all instructions, except where noted, (PC) +1 -> PC.

Symbols:

ACC	Accumulator
AR	auxiliary register 0 or 1
ARP	auxiliary register pointer
dma	data memory address
DP	data page pointer
P	multiply product register

PA	port address
PC	program counter
pma	program memory address
T	multiply Temporary register
->	assigned to
||	absolute value
()	contents of

Machine words are built as follows:



ABS- Absolute value of accumulator

Direct Addressing:	ABS
Indirect Addressing:	N/A
Operands:	N/A
Operation:	|ACC|

ADD- Add to low accumulator

Direct Addressing:	ADD dma, shift
Indirect Addressing:	ADD {*|*+|*-}, shift, next ARP
Operands:	0 <= shift < = 15, 0 <= dma <= 127, ARP = 0, 1
Operation:	(ACC) + (dma)*2^{shift} -> ACC
	Modify AR(ARP), and ARP as specified

ADDH- Add to high accumulator

Direct Addressing:	ADDH dma
Indirect Addressing:	ADDH {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(ACC) + (dma)*2^{16} -> ACC
	Modify AR(ARP), and ARP as specified

ADDS- Add to low accumulator with sign-extension suppressed

Direct Addressing:	ADDS dma
Indirect Addressing:	ADDS {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(ACC) + (dma) -> ACC
	Modify AR(ARP), and ARP as specified

AND- And with low accumulator

Direct Addressing:	AND dma
Indirect Addressing:	AND {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	((ACC) & (dma)) & 0x0000ffff -> ACC
	Modify AR(ARP), and ARP as specified

APAC- Add Product to accumulator

Direct Addressing:	APAC
Indirect Addressing:	N/A
Operands:	N/A
Operation:	(ACC) + (P) -> ACC

B- Branch unconditionally

Direct Addressing:	B pma
Indirect Addressing:	N/A
Operands:	0 <= pma <= 0x1ff
Operation:	pma -> PC

BANZ- Branch if auxiliary register != 0

Direct Addressing:	BANZ pma
Indirect Addressing:	BANZ pma, {*|*+|*-}, next ARP
Operands:	0 <= pma <= 0x1ff, ARP = 0, 1
Operation:	IF AR(ARP) != 0, THEN pma -> PC ELSE (PC) + 2 -> PC Modify AR(ARP), and ARP as specified

BGEZ- Branch if accumulator >= 0

Direct Addressing:	BGEZ pma
Indirect Addressing:	N/A
Operands:	0 <= pma <= 0x1ff
Operation:	IF (ACC) >= 0, THEN pma -> PC ELSE (PC) + 2 -> PC

BGZ- Branch if accumulator > 0

Direct Addressing:	BGZ pma
Indirect Addressing:	N/A
Operands:	0 <= pma <= 0x1ff
Operation:	IF (ACC) > 0, THEN pma -> PC ELSE (PC) + 2 -> PC

BIOZ- Branch if bio == 0

Direct Addressing:	BIOZ pma
Indirect Addressing:	N/A
Operands:	0 <= pma <= 0x1ff

Operation: IF (BIO) == 0,
THEN pma -> PC
ELSE (PC) + 2 -> PC

BLEZ- Branch if accumulator <= 0

Direct Addressing: BLEZ pma
Indirect Addressing: N/A
Operands: 0 <= pma <= 0x1ff
Operation: IF (ACC) <= 0,
THEN pma -> PC
ELSE (PC) + 2 -> PC

BLZ- Branch if accumulator < 0

Direct Addressing: BLZ pma
Indirect Addressing: N/A
Operands: 0 <= pma <= 0x1ff
Operation: IF (ACC) < 0,
THEN pma -> PC
ELSE (PC) + 2 -> PC

BNZ- Branch if accumulator != 0

Direct Addressing: BNZ pma
Indirect Addressing: N/A
Operands: 0 <= pma <= 0x1ff
Operation: IF (ACC) != 0,
THEN pma -> PC
ELSE (PC) + 2 -> PC

BV- Branch on overflow

Direct Addressing: BV pma
Indirect Addressing: N/A
Operands: 0 <= pma <= 0x1ff
Operation: IF overflow flag == 1,
THEN pma -> PC && overflow flag -> 0

ELSE (PC) + 2 -> PC

BZ- Branch if accumulator == 0

Direct Addressing:	BZ pma
Indirect Addressing:	N/A
Operands:	0 <= pma <= 0x1ff
Operation:	IF (ACC) == 0,
	THEN pma -> PC
	ELSE (PC) + 2 -> PC

CALA- Call subroutine indirect *(Not implemented)*

Direct Addressing:	N/A
Indirect Addressing:	N/A
Operands:	N/A
Operation:	N/A

CALL- Call subroutine direct *(Not implemented)*

Direct Addressing:	N/A
Indirect Addressing:	N/A
Operands:	N/A
Operation:	N/A

DINT- Disable interrupts *(Not implemented)*

Direct Addressing:	N/A
Indirect Addressing:	N/A
Operands:	N/A
Operation:	N/A

DMOV- Data move in memory

Direct Addressing:	DMOV dma
Indirect Addressing:	DMOV {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(dma) -> (dma) = 1
	Modify AR(ARP), and ARP as specified

EINT- Enable interrupts *(Not implemented)*

Direct Addressing:	N/A
Indirect Addressing:	N/A
Operands:	N/A
Operation:	N/A

IN- Input data from port

Direct Addressing:	IN dma, port address
Indirect Addressing:	IN {*|*+|*-}, port address, next ARP
Operands:	0 <= dma <= 127, 0 <= port address <= 7, ARP = 0, 1
Operation:	(port address) -> (dma) Modify AR(ARP), and ARP as specified

LAC- Load accumulator

Direct Addressing:	LAC dma, shift
Indirect Addressing:	LAC {*|*+|*-}, shift, next ARP
Operands:	0 <= shift <= 15, 0 <= dma <= 127, ARP = 0, 1
Operation:	(dma)*2^{shift} -> ACC Modify AR(ARP), and ARP as specified

LACK- Load accumulator with immediate constant

Direct Addressing:	LACK eight-bit positive constant
Indirect Addressing:	N/A
Operands:	0 <= constant <= 255
Operation:	(eight-bit positive constant) -> (ACC)

LAR- Load Auxiliary register

Direct Addressing:	LAR AR, dma
Indirect Addressing:	LAR AR, {*|*+|*-}, shift, next ARP
Operands:	AR = 0, 1, 0 <= dma <= 127, ARP = 0, 1
Operation:	(dma) -> auxiliary register Modify AR(ARP), and ARP as specified

LARK- Load Auxiliary register with immediate constant

Direct Addressing:	LARK AR, eight-bit positive constant
Indirect Addressing:	N/A
Operands:	AR = 0, 1, 0 <= constant <= 255
Operation:	(eight-bit positive constant) -> (auxiliary register)

LARP- Load Auxiliary register pointer

Direct Addressing:	LARP one-bit constant
Indirect Addressing:	N/A
Operands:	0, 1
Operation:	(constant) -> (ARP)

LDP- Load data page pointer

Direct Addressing:	LDP dma
Indirect Addressing:	LDP {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(dma) & 0x01 -> data page pointer
	Modify AR(ARP), and ARP as specified

LDPK- Load data page pointer with immediate constant

Direct Addressing:	LDPK one-bit constant
Indirect Addressing:	N/A
Operands:	0 <= constant <= 1
Operation:	constant -> data page pointer

LST - Load status from data memory *(Not implemented)*

Direct Addressing:	N/A
Indirect Addressing:	N/A
Operands:	N/A
Operation:	N/A

LT- Load multiply temporary operand

Direct Addressing:	LT dma
Indirect Addressing:	LT {*|*+|*-}, next ARP

Operands: 0 <= dma <= 127, ARP = 0, 1

Operation: (dma) -> T register

Modify AR(ARP), and ARP as specified

LTA - Load multiply temporary operand and accumulate previous result

Direct Addressing: LTA dma

Indirect Addressing: LTA {*|*+|*-}, next ARP

Operands: 0 <= dma <= 127, ARP = 0, 1

Operation: (dma) -> T register,

(ACC) + (P register) -> ACC

Modify AR(ARP), and ARP as specified

LTD- Load multiply temporary operand, accumulate previous result, shift data memory

Direct Addressing: LTD dma

Indirect Addressing: LTD {*|*+|*-}, next ARP

Operands: 0 <= dma <= 127, ARP = 0, 1

Operation: (dma) -> T register,

(ACC) + (P register) -> ACC,

(dma) -> dma + 1

Modify AR(ARP), and ARP as specified

LTP - Load multiply temporary operand, move product to accumulator

Direct Addressing: LTP dma

Indirect Addressing: LTP {*|*+|*-}, next ARP

Operands: 0 <= dma <= 127, ARP = 0, 1

Operation: (dma) -> T register, (P register) -> ACC

Modify AR(ARP), and ARP as specified

LTS - Load multiply temporary operand and subtract previous result

Direct Addressing: LTS dma

Indirect Addressing: LTS {*|*+|*-}, next ARP

Operands: 0 <= dma <= 127, ARP = 0, 1

Operation: (dma) -> T register,

(ACC) - (P register) -> ACC
Modify AR(ARP), and ARP as specified

MAR - Modify auxiliary register

Direct Addressing:	MAR dma
Indirect Addressing:	MAR {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	Modifies AR(ARP), and ARP as specified

MPY - Multiply

Direct Addressing:	MPY dma
Indirect Addressing:	MPY {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(T register) * (dma) -> P register
	Modify AR(ARP), and ARP as specified

MPYK- Multiply with immediate constant

Direct Addressing:	MPYK constant
Indirect Addressing:	N/A
Operands:	-2^{12} <= constant <= 2^{12}
Operation:	(T register) * constant -> P register

MAC- Multiply and accumulate

Direct Addressing:	MAC dma
Indirect Addressing:	MAC {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(T register) * (dma) -> P register then
	(ACC) + (P register) -> ACC
	Modify AR(ARP), and ARP as specified

NOP- No operation

Direct Addressing:	N/A
Indirect Addressing:	N/A
Operands:	N/A
Operation:	N/A

OR- Or with low accumulator

Direct Addressing:	OR dma
Indirect Addressing:	OR {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	((ACC) | (dma)) & 0x0000ffff -> ACC
	Modify AR(ARP), and ARP as specified

OUT - Output data from port

Direct Addressing:	OUT dma, port address
Indirect Addressing:	OUT {*|*+|*-}, port address, next ARP
Operands:	0 <= dma <= 127, 0 <= port address <= 7,
	ARP = 0, 1
Operation:	(dma) -> (port address)
	Modify AR(ARP), and ARP as specified

PAC- Move Product to accumulator

Direct Addressing:	PAC
Indirect Addressing:	N/A
Operands:	N/A
Operation:	(P register) -> ACC

POP- Pop top of stack to accumulator *(Not implemented)*

Direct Addressing:	N/A
Indirect Addressing:	N/A
Operands:	N/A
Operation:	N/A

PUSH- Push accumulator onto stack *(Not implemented)*

Direct Addressing:	N/A
Indirect Addressing:	N/A
Operands:	N/A
Operation:	N/A

RET - Return from subroutine *(Not implemented)*

Direct Addressing:	N/A

Indirect Addressing: N/A

Operands: N/A

Operation: N/A

ROVM- Reset overflow mode register

Direct Addressing: ROVM

Indirect Addressing: N/A

Operands: N/A

Operation: 0 -> OVM status bit

SACH- Store high accumulator

Direct Addressing: SACH dma, shift

Indirect Addressing: SACH {*|*+|*-}, shift, next ARP

Operands: 0 <= shift <= 7, 0 <= dma <= 127,
ARP = 0, 1

Operation: (ACC[31:16])*2^{shift} -> dma
Modify AR(ARP), and ARP as specified

SACL- Store low accumulator

Direct Addressing: SACL dma

Indirect Addressing: SACL {*|*+|*-}, next ARP

Operands: 0 <= dma <= 127, ARP = 0, 1

Operation: (ACC[15:0]) -> dma
Modify AR(ARP), and ARP as specified

SAR- Store auxiliary register

Direct Addressing: SAR AR, dma

Indirect Addressing: SAR AR, {*|*+|*-}, next ARP

Operands: AR = 0, 1, 0 <= dma <= 127, ARP = 0, 1

Operation: (auxiliary register AR) -> dma

SOVM- Set overflow mode register

Direct Addressing: SOVM

Indirect Addressing: N/A

Operands: N/A

Operation: 1 -> overflow mode (OVM status bit)

SPAC- Subtract P register from accumulator

Direct Addressing: SPAC

Indirect Addressing: N/A

Operands: N/A

Operation: (ACC) - (P register) -> ACC

SST- Store status *(Not implemented)*

Direct Addressing: N/A

Indirect Addressing: N/A

Operands: N/A

Operation: N/A

SUB- Subtract from high accumulator

Direct Addressing: SUB dma, shift

Indirect Addressing: SUB {*|*+|*-}, shift, next ARP

Operands: 0 < shift < 15, 0 <= dma <= 127, ARP = 0, 1

Operation: (ACC) - (dma)*2^{shift} -> ACC

Modify AR(ARP), and ARP as specified

SUBC- Conditional subtract *(Not implemented)*

Direct Addressing: N/A

Indirect Addressing: N/A

Operands: N/A

Operation: N/A

SUBH- Subtract from high accumulator

Direct Addressing: SUBH dma

Indirect Addressing: SUB {*|*+|*-}, next ARP

Operands: 0 <= dma <= 127, ARP = 0, 1

Operation: (ACC) -(dma)*2^{16} -> ACC

Modify AR(ARP), and ARP as specified

SUBS- Subtract from accumulator with sign-extension suppressed

Direct Addressing:	SUBS dma
Indirect Addressing:	SUBS {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(ACC) -(dma) -> ACC
	Modify AR(ARP), and ARP as specified

TBLR- Table Read

Direct Addressing:	TBLR dma
Indirect Addressing:	TBLR {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(ACC[8:0]) -> pma
	(pma) -> dma
	Modify AR(ARP), and ARP as specified

TBLW- Table Write

Direct Addressing:	TBLW dma
Indirect Addressing:	TBLW {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(ACC[8:0]) -> pma
	(dma) -> pma
	Modify AR(ARP), and ARP as specified

XOR- Xor with low accumulator

Direct Addressing:	XOR dma
Indirect Addressing:	XOR {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	((ACC) ^ (dma)) & 0x0000ffff -> ACC
	Modify AR(ARP), and ARP as specified

ZAC- Zero accumulator

Direct Addressing:	ZAC
Indirect Addressing:	N/A
Operands:	N/A
Operation:	0 -> ACC

ZALH- Zero accumulator and load high

Direct Addressing:	ZALH dma
Indirect Addressing:	ZALH {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	0 -> ACC[15:0]
	(dma) -> ACC[31:16]
	Modify AR(ARP), and ARP as specified

ZALS- Zero accumulator and load low with sign-extension suppressed

Direct Addressing:	ZALS dma
Indirect Addressing:	ZALS {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	0 -> ACC[31:16]
	(dma) -> ACC[15:0]
	Modify AR(ARP), and ARP as specified

TDSP Assembler

The TDSP assembler, *tdspasm*, supports compilation of source files formatted using the following conventions. The assembler is case insensitive.

File names for assembly must with a ".asm" suffix. The assembly process will produce three (3) separate output files:

<file_name>.lst - composite machine, opcode listing

<file_name>.sym - cross reference table for symbols and their values

<file_name>.obj - machine object readable by your digital simulator

Source Statement Syntax

Typical source statement will be entered as:

[<label>:] <opcode>[<operand, (operand expression)>] [;<comment>]

Optional attributes are included in brackets, "[]". The brackets must not appear in your source listing. A source statement may include a label that is user-defined. The label field will end in a colon, ":". A source statement may include a comment that is user-defined. The comment field will start with semi-colon, ";".

The operand field may be blank or may contain a constant, an expression, or a previously defined symbol.

Operand expressions can be:

```
simple:     +, -, *, /, %

logical:    ~, ^, &, |

complex:    sin(), cos(), tan(), exp(), log(), sqrt()
```

Operand expressions **must** be enclosed in parenthesizes.

Operand values and operand expressions can contain symbols and labels.

Define Assembly Time Constant Attribute

<label>=<constant value, (value expression)>[; <comment>]

The label field contains the symbol to be given a value. Symbols in the operand field must be previously defined.

Value expressions can be:

```
simple:     +, -, *, /, %

logical:    ~, ^, &, | << >>

complex:    sin(), cos(), tan(), exp(), log(), sqrt()
```

Value expressions **must** be enclosed in parenthesizes.

Symbol values and value expressions can contain other symbols and labels.

Constants

Constants can be represented as decimal, or as hexadecimal values if preceded by 0x. Floating point values can be used in expressions. Note that floating point values will be truncated prior to operand assignment.

Example:

```
245         - Decimal 245

0xfc89      - Hexadecimal FC89
```

Initialize Word Attribute

```
.DATA    - Direct, in-line 16 bit constant

.data    - Direct, in-line 16 bit constant

[<label>:].DATA <value, (value expression)> [; <comment>]
```

".DATA" places one value in program memory. Use this directive to place coefficients or other data words in program memory. TBLR can be used to transfer the data words from program to data memory.

Value expressions **must** be enclosed in parenthesizes.

Absolute Origin Attribute

```
.AORG    - Define new, absolute program origin

.aorg    - Define new, absolute program origin

[<label>:].AORG <location, (location expression> [;<comment>]
```

".AORG" places a value in the program location counter. Multiple ".AORG" statements can be included in your source listing.

Location expressions **must** be enclosed in parenthesizes.

Predefined Symbols and Abbreviations

Along with the machine opcodes, the following symbols are predefined for usage in a source listing:

```
AR0     - Auxiliary register zero

ar0     - Auxiliary register zero

AR1     - Auxiliary register one

ar1     - Auxiliary register one

PAn     - Port Address n (PA0 through PA7)

pan     - Port Address n (PA0 through PA7)

*       - Indirect Addressing

+-      - Indirect Addressing with post increment of
          auxiliary register

*-      - Indirect Addressing with post decrement of
          auxiliary register
```

B

GNU Free Documentation License

GNU Free Documentation License

GNU Free Documentation License
Version 1.1, March 2000

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have

any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-

using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

1. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
2. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
3. State on the Title page the name of the publisher of the Modified Version, as the publisher.
4. Preserve all the copyright notices of the Document.
5. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
6. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
7. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
8. Include an unaltered copy of this License.
9. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the

previous sentence.

10. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
11. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
12. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
13. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
14. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or

distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any v