Philip Simpson

FPGA Design

Best Practices for Team-based Design

Springer

FPGA Design

Philip Simpson

FPGA Design

Best Practices for Team-based Design

Springer

Philip Simpson
Altera Corporation
San Jose, CA 95134
USA
Feilmidh@sbcglobal.net

ISBN 978-1-4419-6338-3 e-ISBN 978-1-4419-6339-0
DOI 10.1007/978-1-4419-6339-0
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2010930598

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

In August of 2006, an engineering VP from one of Altera's customers approached Misha Burich, VP of Engineering at Altera, asking for help in reliably being able to predict the cost, schedule and quality of system designs reliant on FPGA designs.

At this time, I was responsible for defining the design flow requirements for the Altera design software and was tasked with investigating this further.

As I worked with the customer to understand what worked and what did not work reliably in their FPGA design process, I noted that this problem was not unique to this one customer. The characteristics of the problem are shared by many Corporations that implement designs in FPGAs. The Corporation has many design teams at different locations and the success of the FPGA projects vary between the teams. There is a wide range of design experience across the teams. There is no working process for sharing design blocks between engineering teams.

As I analyzed the data that I had received from hundreds of customer visits in the past, I noticed that design reuse among engineering teams was a challenge. I also noticed that many of the design teams at the same Companies and even within the same design team used different design methodologies.

Altera had recently solved this problem as part of its own FPGA design software and IP development process.

I worked with the top talent in Altera Engineering to develop a Best Practices Design methodology based upon Altera's experience and the techniques used by many customers successfully in FPGA design. The resulting methodology was presented and implemented at the customer, with great success.

Through the analysis of past customer data and feedback from customers over the last 3 years, it has become clear that this challenge exists broadly in the industry. The challenge is not specific to one specific FPGA vendor; it is an industry wide challenge.

As such, I have tuned the Best practices FPGA design methodology over the last 3 years and deployed it at several customers with great success.

This book captures the Best Practices FPGA design methodology and now makes it available to all design teams implementing system designs in FPGA devices.

San Jose, CA Philip Simpson

Contents

List of Figures

Chapter 1
Best Practices for Successful FPGA Design

1.1 Introduction

This book which describes the Best Practices for successful FPGA design is the result of meetings with hundreds of customers on the challenges facing each of their FPGA design teams. By gaining an understanding into their design environments, processes, what works, what does not work, I have been able to identify the areas of concern in implementing System designs. More importantly, it has enabled me to document a recommended methodology that provides guidance in applying a best practices design methodology to overcome the challenges.

This material has a strong focus on design teams that are across sites. The goal being to increase the productivity of FPGA design teams by establishing a common methodology across design teams; enabling the exchange of design blocks across teams.

Best Practices establishes a roadmap to predictability for implementing system designs in a FPGA.

The three steps (Fig. 1.1) to predictable results are:

1. Proper project planning and scoping
2. Choosing the right FPGA device to ensure that the right technology is available for today's and tomorrow's projects
3. Following the best practices for FPGA design development in order to shorten the design cycle and to ensure that your designs are complete on schedule and that the design blocks can be re-used on future projects with minimal effort

All three elements need work together smoothly to guarantee a successful FPGA design.

The choice of vendor should be a long-term partnership between the Companies. By sharing roadmaps and jointly managing existing projects, you can ensure that not only is the current project a success but provide the right solutions on time for future projects. A process of fine tuning based on experience working together to guarantee success on projects.

These two topics are touched upon briefly in the Best Practices for Successful FPGA Design methodology.

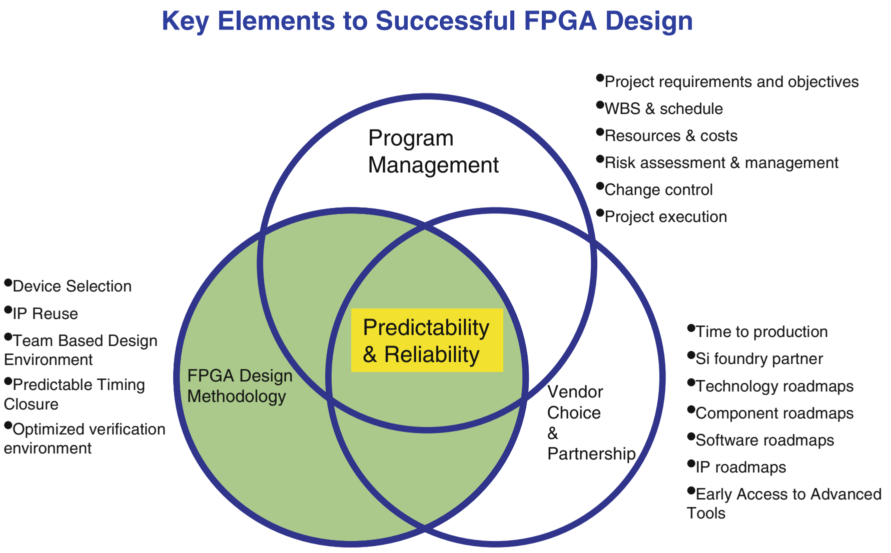

Key Elements to Successful FPGA Design
Program Management
•Project requirements and objectives
•WBS & schedule
•Resources & costs
•Risk assessment & management
•Change control
•Project execution
•Device Selection
•IP Reuse
•Team Based Design Environment
•Predictable Timing Closure
•Optimized verification environment
FPGA Design Methodology
Predictability & Reliability
Vendor Choice & Partnership
•Time to production
•Si foundry partner
•Technology roadmaps
•Component roadmaps
•Software roadmaps
•IP roadmaps
•Early Access to Advanced Tools

Fig. 1.1 Three steps to successful FPGA development

The third topic is the FPGA design methodology.

This is the main focus of the best practices methodology. This covers the complete FPGA design flow from the basics to advanced techniques. This methodology is FPGA vendor independent in that the topics and recommendations are good practices that apply to the design of any FPGAs. While most of the material is generic, it does contain references to features in the Altera design tools that reinforce the recommended best practices.

The diagram that is shown in Fig. 1.2 shows the outline of the best practices design methodology.

Each of the blocks in the diagram is represented by chapters in this book, with an additional chapter on power. Power is its own chapter as it spans many of the other areas of the design methodology. The topics of Board Layout, RTL Design, IP Reuse, Functional Verification and Timing Closure tend to be the areas where design teams have different design methodologies and engineers need guidance on achieving consistent results and shortening the design cycle.

Many of the challenges that are faced in FPGA design are not unique to FPGA design but are common challenges in system design. FPGA devices themselves do provide unique challenges and opportunities compared to ASIC designs. The increase in capability of FPGA devices has resulted in much more complex designs targeting FPGAs and a natural migration of ASIC designers to FPGA design. This has resulted in many design teams migrating ASIC design principles to FPGA designs. In general, this has been a benefit to the FPGA design flow; however it needs to be balanced with the benefits that FPGAs bring to the design flow. The programmable nature of FPGAs opens the door to performing more verification

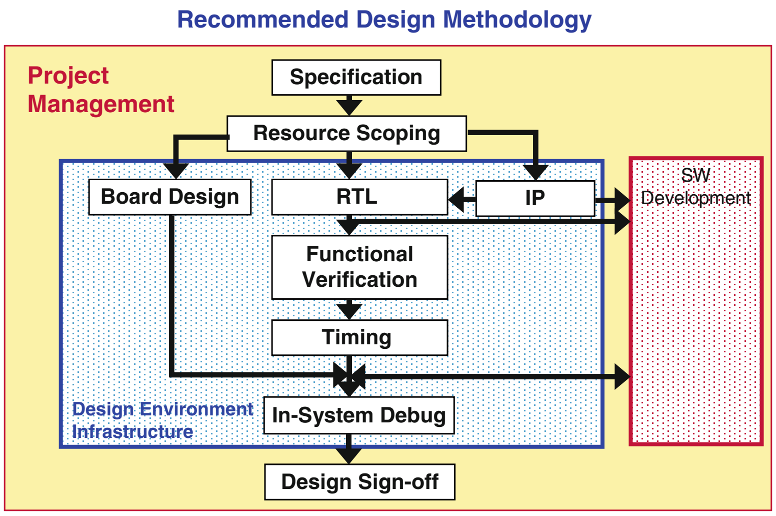


Fig. 1.2 Recommended best practices design methodology for successful FPGA design

in-system. When used correctly, this can greatly speed-up the verification cycle, however when abused it can lengthen the design cycle. The configurable nature of I/Os provides challenges that do not exist in ASIC design. The tools that are used from the EDA industry are also different for FPGAs than for ASICs, in both functionality and cost.

This book will help you adopt the best design methodology to meet your requirements.

While it is recommended that you read the book in its entirety, you can also focus on the individual chapters of the book that target the areas of the design flow that is causing the biggest challenge to your design team.

Acknowledgements Misha Burich for providing the idea for Best Practices. Brian Holley and Rich Catizone for driving the idea at their customer base and providing a constant source of feedback. Chris Balough for encouragement on creating this book. Thomas Sears – Providing access to his development teams, without whom this would not have been possible. YK Ning, Jeff Fox, Ajay Jagtiani, Alex Grbic, Joshua Walstrom, Oliver Tan & Joshua Fender for contributing material to the original presentations on Best Practices. The many customers who have contributed to the material by describing their design environments and the challenges that they have faced in completing their system designs in FPGA devices. My wife Jill and daughter Kayla for their patience and support through the process of gathering data and writing the book.

Chapter 2
Project Management

2.1 The Role of Project Management

The scope of project management is to deliver the right features, on-time and within budget. As such there are three dimensions:

1. Features
2. Development time
3. Resources

The project manager needs to find the right balance of these three dimensions to meet the goals of the project.

There are numerous books and training classes on project management. This chapter provides a brief overview of the elements of project management. It is recommended that you attend formal project management training.

2.1.1 Project Management Phases

Every project can be broken into three project management phases.

1. The planning phase. This is establishing the feature list, creating the project plan and establishing the resource pools and budget.
2. The tracking phase. This involves holding monthly feature reviews, weekly plan updates, reviewing the budget and staffing levels and reviewing any Engineering Change Orders.
3. The wrap-up Phase. This involves project retrospectives, data mining and process improvement review and action plan.

2.1.2 Estimating a Project Duration

Estimating the overall project delivery target is best done with the following steps.

1. Select one of the latest successfully major completed projects.
2. Create a macro model. This involves identifying the major project phases for specification, designing and verification. Extract the exact duration of the phases and any overlap.
3. Set the overall process improvement target. An example would be stating that I want to implement a project of similar complexity 10% faster.
4. Define project complexity metrics such as design characteristics and resource utilization. Design characteristics can include the number of pages of specification, the number of FPGA resources, the number of lines of RTL, design performance technical complexity.
5. Derive the derating factor k.
6. Scale the upcoming project by the derating factor.
7. Evaluate the project with good judgment and make the appropriate adjustments.

2.1.3 Schedule

The project schedule should be updated regularly. It is recommended that it is updated at least once a week.

Any schedule update meetings should be kept brief and should only focus on collecting the status information. This includes information on whether a task has started, is an activity complete, how long will a task take to complete, and any user task information that determines the level of completeness of a task.

The update meetings should also be used to estimate when a task is expected to be complete. The project manager must respect the duration estimates from the resources performing a task but should question any estimates that appear to be wildly wrong.

2.1.3.1 Weekly Schedule Analysis

The project manager needs to rigorously analyze the project schedule on a weekly basis. There are ten main tasks involved in this process.

1. Analyzing and scrutinizing the critical paths.
2. Reviewing the planned tasks for the coming week.
3. Discussing and agreeing on the task priorities with the rest of the review team.
4. Identifying a plan to accelerate the critical path.
5. Identifying other at risk paths that are just behind the critical path.
6. Checking the load on the resources assigned to the critical path.

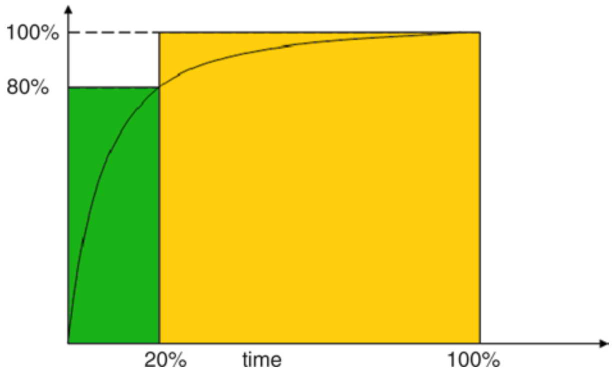


Fig. 2.1 Percentage complete dilemma

7. Confirming the availability of resources with the managers.
8. Determining the part of the project plan that needs more work.
9. Capturing action items.
10. Performing task refinements.

It is critical that the project manager does not get fooled by the percentage complete. It is a non-linear function and is not useful in estimating the remaining task duration (Fig. 2.1).

2.1.3.2 Pro-active Project Management

It requires an extreme degree of pro-active behavior to deliver a project on time. Be sure to dedicate enough management bandwidth to the project.

Due to the dynamic circumstances of design projects, it requires constant management attention with weekly rigorous project schedule updates.

The complexity of the project require the right tools to facilitate the decision making process. The identification and management of the critical path simplifies the priority setting.

Chapter 3
Design Specification

3.1 Design Specification: Communication Is Key to Success

Having a complete and detailed specification early in a project will prevent false starts and reduce the likelihood of Engineering Change Orders (ECOs) late in the project. Late changes to the design specification can dramatically increase the cost of a project both in terms of the project schedule and the cost of the FPGA. The latter occurring as significant changes may result in the need for a larger FPGA device.

The purpose of a specification is to accurately and clearly communicate information.

Another way of saying this is that specifications are a means to convey information between teams/people. Without a thorough specification, which has been approved by all impacted parties, a project is prone to delays and late changes in the requirements; all of which lead to longer project cycles and higher project cost. A key point in this statement is “agreed upon specification”. This implies that a process is in place for the review of the specification.

A fully agreed upon specification ensures alignment between the different teams working on the project. This ensures that the delivered product conforms to the functional specifications and meets the customer requirements. This in turn facilitates accurate estimation of development cost, resource & project schedule. A solid specification enables consistent project tracking, which will ultimately produce a high quality product release. The specification also serves as a reference for the creation of documentation and collateral to be delivered with, or to support the product. All specifications should clearly identify changes that have been made to the specification. In addition, the specification should be stored under version control software.

Specifications are required at different stages of the FPGA design from definition through the development process.

3.1.1 High Level Functional Specification

The high level functional specification is created and owned by the systems engineering team. This document describes the basic functionality of the FPGA design including the required interaction with the software interface and the interfaces between the FPGA and other devices on the board. This document should be officially reviewed with the FPGA design team Manager and the Software engineering manager. After the review, the document should be updated to reflect the recommend changes and to answer any of the issues raised during the review process. This process is iterative until all issues have been resolved and the FPGA design team understands and agrees upon the requirements.

One of the challenges in creating the high level functional specification is successfully describing the functionality in understandable English. Let's be honest here; most Engineers are strong in mathematics and science but will never be the next John Steinbeck.

Executable specifications help resolve this issue. Executable specifications are abstract models of the system that describe the functionality of the end system. It is essentially a virtual prototype of the system. Most executable specifications are created in one of the flavors of "C" (C, C++, System). These languages are good for modeling the desired functionality but do not cover key features such as timing, power and size of design. These need to be covered in an accompanying high level specification to the executable specification. The virtual prototype at this stage is the system model and the testbench which is part of the executable specification. This executable specification can be used throughout the development process to check that the detailed implementation is meeting the requirements of the executable specification.

Not all Companies are using executable specifications as part of the FPGA design process, but its use is becoming more common as more complex systems are being implemented in FPGA devices.

3.1.2 Functional Design Specification

The team that is creating the FPGA design should create a detailed design specification that represents the needs of the high level functional specification. The owner of this specification is the FPGA engineering team. This specification should be reviewed and approved by the FPGA design team, their management and with representation from the systems engineering and software engineering teams. This should finalize the specification for the functionality of the FPGA design and detail the interfaces with the rest of the system including software.

It is critical to agree upon the details of the interfaces to the FPGA with the appropriate development teams that will use these interfaces.

Take for example, the H/W to S/W interface for a design where an A/D converter feeds the FPGA. The FPGA in turn feeds data to a microprocessor. The FPGA requirements specification must cover the interface to the A/D and be designed to avoid any functional failures, even under corner case conditions. Failure to do so can result in functional failures not showing up until testing the design in system. Board tests could show the FPGA passing junk data to the S/W interfaces. The S/W engineers will likely not know how to interpret or debug this issue. This can result in extended board test time and under worst case scenario a redesign of either the software and/or the FPGA design; ultimately this will result in a delay to the schedule.

3.1.2.1 Functional Specification Outline

In this section, we will detail the minimum set of requirements that need to be included in the functional specification.

1. Revision history. A sample revision control page is shown in Fig. 3.1. This includes the date of the changes, the author of the changes and the approval of the changes.
2. Review minutes. This should include details on all review meetings on the specification. The minutes should include the meeting date and location, attendees, minutes and the action items that need to be resolved to gain approval of the specification.
3. Table of contents.
4. Feature overview. The feature overview should provide context of the system in which the feature will be provided. If the feature is a subsystem in the end FPGA system design, this section should describe where it fits in the overall system and its purpose, i.e. the problem it solves. The feature overview should also include a high level overview of its required functionality.
5. Source references. This section should describe the driver of the feature request, e.g. High Level Functional Specification, Software Interface Functional Requirements, etc.
6. Glossary. The glossary should describe any industry standard terms and acronyms that are used in the document. More importantly, it should also do this for any internal Company terminology used in the document. It is amazing how much time is wasted and confusion caused due to the use of internal Company terminology. Many new employees or employees from other groups are often embarrassed to admit that they do not understand the "code" words in review meetings, resulting in confusion, delays in decision and often the stifling of creativity.

Revision History

Version	Author	Date	Changes
0.9	psimpson	4-26-09	Initial revision
1.0	psimpson	5-11-09	Added timing details to CODEC
1.1	aclarke	5-30-09	Modified register map based upon review with SW Engineering on May 28, 2009.
1.2	jjones	6-3-09	Adding a section to describe the interface to host processor.
1.3	psimpson	6-9-09	Updated host processor interface after second review with SW Engineering on June 4.

Fig. 3.1 Sample revision control page

7. Detailed feature description. This is really the meat of the document. This section should include descriptions of any of the algorithms used, details on the architecture of the design and the interface with other parts of the design or system.
8. Test plan. The document should refer to the test plan, or at a minimum state the need for a test plan and be updated when the test plan exists.
9. References. In this section the document should refer to all supporting documents that should be read to understand the functional specification.

Following the creation of the detailed FPGA design specification, the engineering team will create a number of specifications for internal review within the engineering department. These include the Functional Test Plan and QA Test Plan. Each engineer that is assigned to the project will create an engineering plan and functional test plan for the portion of the design that they will be implementing. This should be reviewed within engineering against the overall functional plan. This ensures that it meets the overall requirements of the FPGA design.

3.1.2.2 Test Specification Outline

1. Revision history. A sample revision control page is shown in Fig. 3.1. This includes the date of the changes, the author of the changes and the approval of the changes.
2. Review minutes. This should include details on all review meetings on the specification. The minutes should include the meeting date and location, attendees, minutes and the action items that need to be resolved to gain approval of the specification.
3. Table of contents.
4. Scope. This will provide an overview of what specific features this test plan will cover. If test coverage overlaps with the testing of any subsystems, it should detail what will be covered in this test plan and refer to the other test plans.
5. Test requirements. This should detail any special hardware, software, EDA tools that are required to complete the testing. As part of this it should include any special set-up requirements.
6. Test strategy. This includes the pass/failure criteria. Do the test results require cross-verification with any other sub-systems. Will existing tests be re-used or modified to meet the needs of this test plan. Will the tests be automated and if so, how will the tests be automated. How will the tests be run. An example of this would be an automated regtest that is run each night, or manual testing to verify that the graphics appear correctly on the screen when run on a development board.
7. Automation plan. It is desirable to automate as much of the testing as possible. This section will describe how to automate the test.
8. Running the tests. What is the expected runtime of the tests. If the test is not automated, what is the expected time for the tests to be performed manually.
9. Test documentation. This section should include descriptions of the test cases. As standard practice, the test infrastructure should be set-up to isolate each test.

Thus each test case should have its own test directory. The documentation should detail how to access the results from the regression tests database. This assumes that a regression tests system has been established. Not establishing such a system is setting a project up for failure as it will be incredibly difficult to monitor the quality of the product.

The test documentation should also cover test procedures for the cases where sub-tests cannot be automated. Under this scenario, it is necessary to document how to manually test the sub-feature.

As work begins on the development of the FPGA design, there should be regular design and verification reviews as part of the engineering process to ensure that there are no changes to the plan. These reviews will provide a forum to communicate any changes that may be needed to work around implementation issues and to clear up any areas of ambiguity in the specifications. As a result of these meetings, the specifications should be updated and reviewed. If the recommended changes will impact the high level functional specification or any of the interfaces with the FPGA, there should be formal reviews with the relevant personnel to reach closure on the changes.

In summary, the main purpose of a specification is to communicate information between teams such that the design meets the requirements and can be adequately staffed to deliver on the requirements in the specified timeframe.

The requirements for the functional specification and test specification will be driven by your Company's policy on standards compliance, e.g. ISO 9001 compliance. This book does not discuss the details on ISO 9001 compliance. A detailed description of the ISO 9001 standard is available from http://www.iso.org.

Recommended further reading:

Requirements by Ian Alexander

Chapter 4
Resource Scoping

4.1 Introduction

This chapter is broken down into three main sections. The first section deals with engineering resources. Whether you use internal resources or whether you use external contractor resources.

The second section deals with IP. Do you have IP within the company that you can reuse, or do you use third party IP?

The third and last section deals with device selection. This details how to select the right FPGA with the right resources for your application. It covers the various techniques that you can use to help choose the right device to enable you to meet your project schedule.

4.2 Engineering Resources

The assignment of engineering resource to the project is a project management task. It is key that you adequately resource the project with the appropriate personnel for the tasks in the project. When you are working on the FPGA its not only FPGA designers that you need to consider, you need to look at the team of engineers that are required to create the design. So, from a hardware engineer's perspective you look at who are the engineers that are going to work on the FPGA design. There are the RTL designers, there are the engineers with the experience integrating the design in the FPGA design software and the engineers with design verification experience.

In some companies these roles will be performed by the same individual, or the same pool of engineers. However, depending upon the size of the design or the complexity of the project you may well require a team of engineers with different skill sets from the different engineering disciplines. From a hardware engineering perspective, you also need to look at the board design, so you will to need to ensure that you have board layout engineers on the team. They will have to work close with the FPGA designers, so you want to make sure that the members of the team have a good working relationship. If you are creating a high speed design, particularly if you are looking at design with high speed transceivers or high speed memory interfaces you are likely going to need someone on the team with signal integrity experience.

If your design uses a soft processor such as the Nios® II processor form Altera, you will also want software engineers on the team. Even if the FPGA is interfacing with a microprocessor, you still want the software engineers to be available for when you start to debug the design on the board. You also may need engineers with other system specialties on the team. For example if your design contains DSP algorithms the individual that created the algorithm may not actually be a hardware engineer, thus will not be implementing the design in the FPGA. You need to ensure that the Specialist is available for advice during the design cycle and for debug of the design after implementation. Similarly, for other IP areas of excellence; examples being the main interface protocols such as PCIe or GigE.

An important decision in the assignment of engineering resources is the decisions as to what are you going to implement with the engineering resources that exist in the company vs. what will you implement with external consultants.

4.3 Third Party IP

You need to look at what third party IP is available and will be used in the design. Similarly what internal IP will be reused, do you have IP available from other projects targeting this FPGA family. Or if you are using third party IP you will probably want to look at what are you getting with the IP, do you get a consultancy service or what is your level of confidence that the IP will meet your exact requirements in terms of area, speed and functionality.

4.4 Device Selection

There are seven main factors that influence your choice of device. These are:

1. Specialty silicon features. Are there certain capabilities that you need that dictate that you use a particular FPGA because they are not available in other FPGA devices.
2. Device density. How much logic will your design require? What is the mix of logic to memory blocks to dedicated multiplier blocks that is needed for your application. This will have a big impact on the price of the device that you need.
3. Speed requirements. This will impact the family that you choose and the speed-grade that you need to use. Once again this will have a large impact on the price of the device.
4. Pinout of your device. What kind of package do you require? The choice of package type and the number of I/O in your design will impact both the FPGA

cost and the board design. The package type will also influence the signal integrity and performance of the I/O in your design.

5. Power. What is your power budget for the design and which device is going to help you meet the budget?
6. Availability of IP.
7. The availability of silicon. You want to make sure that production silicon is available when you need it.

So these are the areas that we need to look at in more detail.

4.4.1 Silicon Specialty Features

The first area that you want to look at is the dedicated resources on the device. Does your design require high speed serial interfaces and if so, how many channels and at what performance. Many of the FPGA devices that are available together come with transceivers. The performance of transceivers tends to fall into three ranges, up to 3.125 Gbps, up to 6.5 Gbps and 10 Gbps+. These are important factors in the decision process as they impact both the performance of your design and the cost of the FPGA. You also need to look at your bandwidth requirements. Both the speed of the transceivers and the number of transceivers will determine your bandwidth. Take for example the communications market; if you are trying to implement 100 Gbit Ethernet, you will likely want a minimum of ten channels of 10 Gbps transceivers.

Similarly, if you are completing a design which is math intensive such as a DSP encryption algorithm or radar application, you will require a device with a large number of DSP blocks and adequate RAM blocks to interface with the DSP blocks. The configuration of the DSP blocks is also important. The depth and number of memory blocks will impact how much processing can be performed on chip vs. having to use external memory. Internal memory is important in DSP for caching of processing results between stages of the processing algorithm. You also need to look at both the number and configuration of the dedicated DSP blocks. What is the width of the multiplication operations that you need to perform? If the DSP block does not have sufficient width, you will have to start combining DSP blocks with logic to implement your functionality. This can impact the performance of the operation that you are performing.

How many internal RAM blocks do you need? This is becoming increasingly more important as we look at designs that make use of soft processors. Being able to use internal memory blocks as cache can significantly increases the performance of the soft processor. The sizes of block RAM that is available is also important. If your design will use a lot of FIFOs, it's the number of RAM blocks that are available that matters and not the amount of bits available. FIFO's are notorious for wasting memory bits when implemented in memory blocks.

You also need to consider the debug of your design. Internal block memory is often used in the debug cycle for storing the data from embedded logic analyzers for examination.

4.4.2 Density

When selecting the density of the device, it is unlikely that you will be fortunate enough to have the completed design to determine the size of device needed. You will be choosing the device based upon previous experience. Many designs are based upon previous generations of the design. This can be aid in the device selection process. You should recompile the previous design or the portions that will be used at your target FPGA family to get ballpark density estimates. If you have IP that you will be using, compile it to add to your area estimates and if you are evaluating IP for third party vendors, get an area estimate from the vendor. So, use the previous generation of the design, if it exists, add in the area requirements from IP and then using your experience, add in how much additional resources will be used for the new functionality. Once you have done this, add an additional 25% on top. You should always target a larger device than you think you will need; this is where the extra 25% comes into the equation.

You should always target a larger device than you think you will need. Designs have a nasty habit of growing and you want to guarantee that the design will fit in the targeted device and be able to close timing. You don't want to be struggling to meet timing in a 95% utilized device or be put in the position of having to pull functionality out of your system just to fit in the targeted device.

Another benefit of using a larger device is that it can help you get to in-system checkout quicker. If there is headroom in the device, the place and route software will likely not have to try as hard to meet timing and will result in shorter compile times. This benefits both the hardware and software engineer. The sooner that you have functional silicon, the sooner the software engineer can accelerate his code development process by trying it out on the targeted hardware. You can start the debug of the hardware and software much earlier in the design cycle.

Another benefit of the additional headroom in the device is that it makes it easier to accommodate late ECOs in the device or accommodate growth in future versions of the design after production.

After you have the design working functionally on the device and if there is significant unused resources on the device, you can retarget the device to a smaller device to reduce cost and not have to worry about impacting the project schedule. Some of the FPGA vendor design tools have features that enable you to migrate between device densities in the same family while maintaining the same pin-out. These features restricts you to using only the I/O resources that exist across the density ranges selected in the targeted family; the benefit being that you can retarget your design to a larger or smaller density device avoiding a board re-spin. If this feature is not available in your FPGA vendor software you can design the capability in manually by referencing data sheets and application notes. The manual process is painful and prone to user error, but is worth the investment if the automated flow is not available.

The key point is that you need to ensure that the ability to migrate between device densities while maintaining the pin-out capability is available in the FPGA family that you are considering for your application.

The recommendation is that you select a device that can migrate up in density to accommodate future design growth and can migrate down in density to allow for possible cost reduction.

This functionality is very useful if you intend to ship variations of your product at different price points with changes in the functionality. This enables the same board to be shipped. A single design can be created and functionality removed from the FPGA at the lower price points. Normally the same FPGA is shipped on the same board with a different programming file based on the reduced functionality of the design. By maintaining the same pin-out you can now remove the functionality and retarget the design to a smaller device, further cost reducing your bill of materials.

4.4.3 Speed Requirements

This can be determined from your previous design experience. You should compile designs or design blocks that you already have to get an indication of the performance that they get in the targeted device. This can be used as a good best case indicator as to what you can expect from other design blocks.

The FPGA vendor's data sheets are also a good source of information on performance. They will tell you the absolute maximum that you can hope to get in terms of clock and I/O performance. While these numbers are achievable, it is likely to increase your timing closure cycle achieving these numbers, thus you should back off the numbers by approximately 15% to give you a margin of safety for timing closure.

The choice of speed-grade will impact the price of the device. When choosing device, we recommend that you always start with the fastest speed-grade to enable you to get the device on the board as soon as possible to start software debug and hardware functional check out as early as possible. If the design meets timing comfortably in the fastest speed-grade, you will benefit from faster compilations as the place and route engine does not have to try as hard to close timing. Later in the design cycle, there is the option to retarget the design to a slower device after the functionality is close to complete, for cost reduction purposes.

4.4.4 Pin-Out

The type of I/O interfaces that you need for the design will impact the number of pins required and the package type. You need to understand the I/O standards that you need, the requirements for drive strength. How many pins do you need? What are the power supply requirements? A good way of determining these requirements without the design is by looking at what your device will interface with on your board. You also need to look at the signal integrity requirements for the design. Does your design have interfaces with a large number of pins that are likely to toggle simultaneously;

if so, will you have SSN issues? It is worth noting that wirebond packages typically have worst signal integrity and I/O performance than flip chip devices.

It is recommended that when looking at the pin count for your design, that you reserve pins for in-system debug. The target should be a minimum of 15% of the device pins. They can be used to route internal signals off-chip for analysis with a logic analyzer.

4.4.5 Power

You know the power budget for your design based upon the specification. How many power supplies will be required for the device? Most modern FPGA devices require multiple power supplies as they have separate power planes for the core, I/O's and often the transceivers. The more power supplies that are required, the more expensive the component cost on the board and the more complex the board design.

Once again, your previous FPGA design experience will come into play. Chapter 7 in the book is dedicated to power estimation; it will help master this challenge.

To summarize, it is recommended that you use the FPGA vendor's power estimation spreadsheet together with your previous experience to determine the power that your design will consume.

4.4.6 Availability of IP

IP may be available for a particular family of devices but may not have been ported to or verified on the particular FPGA family that you are considering using. This is often the case with devices that are new to the market. Interface IP in particular is a challenge for devices where the silicon has been available for less than 6 months. The devices are normally not fully characterized thus the timing models are preliminary. High performance interface IP cannot be guaranteed to close timing until the models are final.

4.4.7 Availability of Silicon

If you have a project on the bleeding edge of technology, the chances are that you will be considering using the latest FPGA devices on the market. You will also likely be considering the latest FPGA device knowing that in the future, the pricing will be more favorable. The decision to use the latest FPGA devices on the market makes financial sense if the design will be going into production in 12 months but you know that your volumes will be shipping for 5+ years such that you will be hitting volume production when the FPGA process has matured and pricing is at its lowest.

4.4.8 Summary

We really recommend that when choosing FPGA technology that you quickly stitch together dummy designs effectively to enable the process of successful device selection. You are going to have a good idea of what type of interfaces you are going to need on your device. This will help you to determine the pin requirements and simplify the I/O planning requirements. By creating the dummy design you get an idea of the utilization that you can expect to get out of the device in terms of resources. It will also provide a good guide to the performance that you can expect for your type of design. It also enables you to perform an early power estimate for your design. The creation of a dummy design is instrumental in selecting the appropriate device and should include any known IP blocks that you are going to be used in the design.

Chapter 5
Design Environment

5.1 Introduction

The FPGA design environment is best expressed as a combination of all of the tools, techniques and equipment that is required to successfully complete a FPGA system design. The design environment in each company is usually somewhat unique in that it has been customized to meet the needs of the company. However, there are some common elements that exist across all design elements. The goal of this chapter is to make you aware of the bare minimum requirements for a design environment that will enable the successful creation of an FPGA design on time. The design environment can be represented by five main elements.

1. A scripting environment
2. Interaction with Version Control software
3. Use of a problem tracking system
4. A regression test system
5. Data collection for analysis

5.2 Scripting Environment

One of the challenges for engineers that are designing with FPGA devices, is when to use a scripted design flow vs. when to use the GUI in the FPGA design environment?

Scripts are ideal in the following scenarios:

1. Creation of projects
2. Creation of assignments for the design
3. Compilation of designs. In particular if you utilize a compute farm environment. A compute farm environment enables you to fire off batch jobs to the server for compilation
4. Functional verification and regression testing
5. Integration with version control software

This covers most of the FPGA design flow. It may appear that it is recommended to use scripting for every part of the design flow. This is partially true. You really should deploy scripting for any repetitive tasks. It helps other users to easily reproduce your environment and results.

So, when is it recommended to use the GUI?

The GUI should be used for the parts of the design flow that are interactive. Areas where your actions will change based upon the results that you get. Examples would be the following scenarios:

1. In-system debug of your design
2. Floorplanning operations. This could be looking at the details of the floorplan to gain a better understanding of the device architecture or the resources that are available. This could also be creating a physical layout of your design in the floorplan in a team based design environment
3. Getting started with new tools. The GUI provides a great way for setting up your first project and uncovering the features and capabilities of the tool. Once familiar with the tool, it is recommended that you move to a scripting environment

Through the use of scripting you can save time and effort on repetitive tasks. One of the big benefits is that it simplifies the passing of tasks between team members in a team based design. If someone is taking over a project or design block, from another engineer; Rather than having to write detailed instructions describing what needs to be done to get your results, you give them the script which is self documenting. The new engineer reads the script, runs the script and they get started from where you left off on the project. Nearly all EDA tools that are part of the FPGA design flow have scripting interfaces, both a command-line interface for creating batch files and assignment scripting for creating settings in the project. Most of the EDA industry has standardized on Tcl as the scripting interface for tool assignments.

5.3 Interaction with Version Control Software

Revision Control software provides a record of the history of changes to your design. When you are designing a FPGA, it is necessary to understand the minimum set of files that is needed for check-in and check-out of the version control system. You need to minimize the number of files because the more files that you check-in, the more storage you will need and the more complex the operation will become. Each time you make a change to your design you need to check the FPGA project back in to the version control software. A good scripting environment helps to simplify this process. The initial set-up of the scripts and the identification of the files that need to be checked in and out may be complex. However, once the scripts are established, the scripts can be shared among the engineers that are working on the project. If you can recreate or describe your project with a script, the version control interaction becomes much simpler.

Different FPGA design tools require different sets of files to be placed under version control in order to recreate the results; so the set-up that you use for one FPGA vendor may differ significantly than the set-up used for another. The principle however is the same. If the tools use text files, the interaction with version control systems is much simpler than tools that use binary files that store critical information.

To date, FPGA vendors have done a poor job in publicly documenting which files need to be checked into version control software to enable you to recreate the results of the previous compilation. This process becomes more complex if you use multiple tools in the FPGA design flow. It is recommended that you contact the vendors of each of the tools to understand their recommendations.

One of the major influences on how you use a version control system is the directory structure that you are using for your design environment. This comprises of the location of the RTL design files, location of the RTL and IP libraries, "c" code and programming image if you are using a soft processor, simulation testbenches, location where the results of your regtests are stored and the scripts to compile the design in the FPGA software or in other EDA software. You need to be able to link all of these elements together successfully using the correct versions of the files.

You want to avoid the situation were you are trying to debug the design in the lab and you are using the wrong programming image for the FPGA, or you are loading the soft processor with old source code, or a designer is making changes to an out of date version of the RTL. Proper use of version control will provide an environment that prevents these scenarios from occurring. You also want to be able to store the report files in version control as the report files document the status of the design. This provides valuable information to other designers that work on the same project.

5.4 Use of a Problem Tracking System

A problem tracking system is not a capability that you get from your FPGA vendor. However, I can guarantee that it is a tool that FPGA vendors use as part of their engineering and product planning process. Problem tracking systems tend to be homegrown systems to meet the needs of the individual company. In fact many of the EDA tool and FPGA vendors have a customer interface to their systems for submitting problem reports.

There are commercial systems available on the market. These systems are essentially database system with a customizable front-end to meet your companies needs. In your design environment, you will use the system to track all known issues with your FPGA design. It enables the design engineers to document problems with the design as they occur. This provides the team with an instant status on the design and can be used to track the stability of the design throughout the design process. It makes the other members of the team aware of the problems with your design, avoiding the case were they are trying to debug a problem in their part of the system that is being caused by your design. By looking at this data it can be determined whether to use a particular project build or whether to revert to an earlier build that did not exhibit the problems that were introduced into that particular build.

It also enables users to document the closing of issues. This enables the team to collaborate on solving the issues in the design. This is very helpful in a team based design environment that spans multiple time zones.

As mentioned, the system can be used to provide a snapshot of the health of the project. To do this, it needs to be linked to the regression test system such that test failures automatically file problems reports in the tracking system against the build that is being tested.

5.5 A Regression Test System

As part of your testing, the design engineers will create point tests to show that the design meets functionality. It must be a requirement that you have a set of tests that are run regularly on the design to provide a health check on the design. These tests give you confidence that as your design changes that you do not reintroduce old problems or break existing functionality. Regression tests are discussed in more detail in Chapter 11.

5.6 When to Upgrade the Versions of the FPGA Design Tools

One of the challenges that you will face if you have a design that spans more than 6 months is when to adopt new releases of the tools that are used in the FPGA design environment. FPGA vendors typically have at least two major releases per year plus a selection of service pack releases that include bug fixes and timing model changes. When should you freeze the version of the design tools that you are using?

This decision will be driven by where you are in the design flow. If you are in the early stages of the design, then you should update to the latest release of the FPGA design software unless you are aware of serious problems with the software. This will give you access to the latest bug fixes and features in the software. Normally there is some degree of compile time improvement in the major releases of the FPGA design software.

If your design is mostly complete and the version of the FPGA vendor software that you are using contains the final timing models for the devices that you are targeting, then you should consider freezing the version of the design software that you are using. An exception would be if you come across a bug in the design software that impacts your design. This will likely require you to upgrade the design tools to access the fix to the bug.

If your design is close to complete but the FPGA vendor timing models are still preliminary you will have to upgrade the version of the design software once the final timing models become available. This can be problematic as it may require you to upgrade the versions of the vendor IP blocks, possibly creating more work for you; in particular in verifying the design. It is strongly recommended that you verify your design against the production or final version of the FPGA timing models.

Some of the FPGA vendors provide the capability to read a database from one version of the design software in a later release of the software. Thus the design does not have to be recompiled and only timing analysis rerun to verify that the design still meets timing, with the final timing models.

5.7 Common Tools in the FPGA Design Environment

FPGA design software: This comes from the FPGA vendor and includes the FPGA Place and Route Software and Timing Analysis tools. The major FPGA vendors also include RTL Synthesis, Advanced Timing Closure Features. On-Chip debug and Floorplan Tools.

FPGA synthesis software: This may come from the FPGA vendor or may come from EDA synthesis tool vendors such as Synopsys or Mentor Graphics. Most FPGA synthesis tools support Verilog and VHDL. Some of the tools now support SystemVerilog.

Simulation tools: Some FPGA vendors provide simulation tools but by far the majority of the tools that are used come from EDA tool vendors. The most popular tools are Mentor Modelsim and Questasim, Synopsys VCS, Cadence Incisive and Aldec Active HDL and Riviera Pro. Some of these tools include advanced capabilities for assertion based verification, detection of clock domain crossing, etc.

Formal verification tools: These tools are not commonly used in FPGA designs due to the restrictions that they place on the optimizations that can be performed when using these tools in order to perform a successful verification.

Timing analysis tools: There are timing analysis tools available from EDA tool vendors. However, these are rarely used in FPGA design flows due to the availability of timing analysis tools in the FPGA vendor supplied design software. We recommend that you use the FPGA vendor timing analysis tools for FPGA timing analysis as the timing constraints that are used for timing sign-off are also used by the place and route software for optimization.

It is recommended that the EDA timing analysis tools are not used for FPGA verification, but are used for board timing analysis.

Board design tools: EDA tools are used for board design. These include the board schematic tools, the board layout tools and the signal integrity tools. The HSPICE and IBIS models that are used by the signal integrity tools come from the FPGA vendois.

High-level synthesis: Most of the tools in this space are based on designing in “C or C++” and having the code produce RTL or a netlist for an FPGA. The adoption of these tools in the FPGA market has been slow. These tools have matured a lot and are slowly gaining momentum in creating design blocks for certain applications, as opposed to creating a complete FPGA design. These tools tend to be mainly focus on the High Performance Computing Market and DSP algorithm implementation.

All of the offerings that are available are from EDA Companies.

The next class of High-Level Synthesis is Model based design tools. These utilize optimized libraries in the Mathworks Simulink environment. Their target markets are military markets and Modem designs. These tools rely on the Mathworks Matlab environment and are available from the main FPGA vendors and EDA Companies.

Load sharing software: This is software that is used to schedule jobs that are being processed on compute farms. Load sharing solutions are heavily used in FPGA development, particularly in script based design flows. There are commercially available software packages as well as freeware. Some of the options in the FPGA software include a form of load sharing software.

Version control software: Version control tools are not considered EDA tools per se, but are a major part of the design flow environment, commonly used version control software with FPGA designs are Clearcase, Perforce and PVCS.

Chapter 6
Board Design

6.1 Challenges that FPGAs Create for Board Design

In order to meet the fast performance and high bandwidth of today's system designs, FPGA devices are providing a large number of pins with increasingly faster switching speeds. These higher package pin counts, together with the fact that the devices support many different I/O standards and support different package types, creates a challenge in successfully creating the FPGA pin-out efficiently and correctly. The cost of a board re-spin, due to a problem with the pin-out, is expensive in terms of both the cost of the board re-spin and the impact on the project schedule.

FPGAs provide pin-out flexibility by supporting many different I/O standards on a single FPGA and by providing user control over drive strength and slew rate. This flexibility also results in complex rules for the creation of a legal FPGA pin-out and impacts the termination requirements for the Printed Circuit Board (PCB).

The high package pin counts create an EDA tool flow challenge in the management of data between the board design software and the FPGA design software.

Due to the complexity in designing high performance PCBs, the PCB design cycle needs to begin early in the system design cycle. This creates a challenge in aligning the final FPGA pin-out with the board design cycle. Often the board layout needs to be complete prior to FPGA design completion. In fact, it is becoming increasingly common that the FPGA design and the board development are being undertaken simultaneously and that for many user system designs, the board design is often complete prior to the RTL code for the FPGA existing!

Early in the design cycle, it can be difficult to predict the size of the FPGA device that is required for the project. Most FPGA families have a technical solution to this problem; they support pin migration between devices of different density in the same package. Thus, it is advised that designers select a FPGA device that has several densities in the same package. This creates the challenge for the board designer in creating a pinout that is migratable across all the device densities. Once again, help is at hand from some of the FPGA design tools via a feature that is often referred to as device migration. Device Migration is the ability to transfer a design from one device in an FPGA family to a different density device in the same device family which has the same device package. This enables you to transfer a design from the design's target device to a larger or smaller device with the equivalent pin-outs, while maintaining the same board layout and pin assignments. This is a feature that can be selected in the FPGA vendor software when making the device selection. This feature will prevent the user from making pin assignments to pins that cannot be migrated across the different device densities. It is recommended that you include this requirement as part of your design plan as insurance against unforeseen changes in the FPGA design, particularly if creating a pinout early in the FPGA design cycle. This enables you to use a larger device if the changes to the design results in a significant logic growth or potentially the ability to use a smaller, hence cheaper device, if the design size permits this.

The increase in system performance and bandwidth has resulted in faster pin speeds. At the time of writing, FPGAs are capable of interfacing with 64-bit DDR III SRAM running at 533 MHz. This is a data rate of 1,067 Mbps per pin. This can produce a number of simultaneously switching pins on the FPGA, which can in turn result in functional failures due to noise. The device needs to have a pin-out that avoids Simultaneously Switching Noise (SSN) and the FPGA needs to be terminated on the board in a manner that avoids SSN issues.

Many FPGAs also include transceiver blocks that can operate up to 11.3 Gbps and support various I/O protocols such as PCI Express, Serial RapidIO®, Gigabit Ethernet (GbE), to name a few. These high speed transceiver based interfaces require careful termination on the board to avoid Signal Integrity (SI) issues.

Now that we have identified the potential pitfalls in creating a PCB design for high performance systems containing FPGA devices, we will focus on the techniques that can be deployed to ensure that the board design is right first time. The remainder of the chapter describes the challenges in more detail. It describes the roles of different teams in the board design process. It presents a methodology that addresses all of the challenges that we have described and culminates in a check list that can be used on any FPGA project to achieve successful FPGA pin-out and board design.

6.2 Engineering Roles and Responsibilities

The engineers that are involved in the board design of systems containing FPGA devices can be classified into three distinct engineering skill sets. These are FPGA design engineers, PCB Design Engineers and Signal Integrity Engineers. In some organizations there is overlap in the functionality, but in general they are distinct disciplines and the functions are performed by different engineers or engineering teams.

6.2.1 FPGA Engineers

FPGA Engineers are familiar with the FPGA vendor software. The FPGA engineer is typically responsible for writing and verifying the RTL code for the design. He, or she, is also responsible for implementing the design in the FPGA and helps with the debug of the design in the end system.

The FPGA engineer has a keep role to play in the PCB design. He is responsible for the generation of the FPGA pin-out from the FPGA design software. As such, he interfaces heavily with the PCB design engineer, providing updates to the pin assignments and implementing and verifying any recommended changes from the PCB design engineer.

The FPGA Engineer also acts as the interface to the Signal Integrity engineer. He provides the pin-out information, as well as any HSPICE and/or HSPICE models and netlists that are generated by the FPGA design software.

6.2.2 PCB Design Engineer

The PCB design engineer is familiar with PCB schematic and layout software. The PCB design engineer is typically responsible for creating board schematics, including the generation of device symbols. He is also responsible for creating the board layout, which includes routing the board. The board layout and in particular the routing of the board is heavily dependent upon the pin-out of the devices on the board. As such, the PCB design engineer has a strong influence on the FPGA pin assignments, as these greatly impact his task and the potentially the cost of the board. While the PCB design engineer influences the choice of pin assignments for the FPGA, he typically has no desire to use the FPGA design software. This creates the requirement for an efficient means of passing information to/from the FPGA engineer and the Board Designer. This is effectively the need for a two-way interface mechanism between the FPGA design software and the board schematic software, from EDA tool vendors. Today, some EDA tools provide a two way interface to the FPGA design software. However, the most commonly used interface for the communication of information between these two engineers is Microsoft Excel. Most of the FPGA design software offerings from the FPGA vendors have the ability to read and write the .csv format, which is used as the interface to Microsoft Excel. Similarly some of the board schematic software packages can read the .csv format. It is common practice within industry for board design engineers to create scripts that generate the appropriate schematic symbols from the .csv format or from the FPGA vendor pin report. Thus the .csv format serves multiple purposes.

1. A source of integration between the FPGA and Board design software packages.
2. Documentation of the design pin-out. As such, it should be stored under revision control.

An example of a .csv file that can be used to interface between the FPGA design software and board schematic software is detailed in Fig. 6.1.

A key point is that the csv details much more than the pin assignments. It includes details on the I/O standard and current strength. These are important as they impact the signal quality on the board, as well as the I/O timing.

Pin Name	Direction	Location	I/O Bank	VREF Group	I/O Standard	Current Strength
clk_in	Input	PIN_B13	4	B4_N1	3.3 - V LVTTL (default)	24mA (default)
in_port_to_the_button_pio[3]	Input	PIN_AE6	8	B8_N1	3.3 - V LVTTL (default)	24mA (default)
in_port_to_the_button_pio[2]	Input	PIN_AB10	8	B8_N1	3.3 - V LVTTL (default)	24mA (default)
in_port_to_the_button_pio[1]	Input	PIN_AA10	8	B8_N1	3.3 - V LVTTL (default)	24mA (default)
in_port_to_the_button_pio[0]	Input	PIN_Y11	8	B8_N1	3.3 - V LVTTL (default)	24mA (default)
ext_flash_enet_bus_data[7]	Bidir	PIN_A8	3	B3_N0	3.3 - V LVTTL (default)	24mA (default)
ext_flash_enet_bus_data[6]	Bidir	PIN_B8	3	B3_N0	3.3 - V LVTTL (default)	24mA (default)
ext_flash_enet_bus_data[5]	Bidir	PIN_C9	3	B3_N1	3.3 - V LVTTL (default)	24mA (default)
ext_flash_enet_bus_data[4]	Bidir	PIN_D9	3	B3_N1	3.3 - V LVTTL (default)	24mA (default)
ext_flash_enet_bus_data[3]	Bidir	PIN_G10	3	B3_N1	3.3 - V LVTTL (default)	24mA (default)
ext_flash_enet_bus_data[2]	Bidir	PIN_F10	3	B3_N1	3.3 - V LVTTL (default)	24mA (default)
ext_flash_enet_bus_data[1]	Bidir	PIN_C8	3	B3_N1	3.3 - V LVTTL (default)	24mA (default)
ext_flash_enet_bus_data[0]	Bidir	PIN_D8	3	B3_N1	3.3 - V LVTTL (default)	24mA (default)
out_port_from_the_led_pio[7]	Output	PIN_AA11	8	B8_N1	1.8 V	12mA (default)
out_port_from_the_led_pio[6]	Output	PIN_AF7	8	B8_N1	1.8 V	12mA (default)
out_port_from_the_led_pio[5]	Output	PIN_AE7	8	B8_N1	1.8 V	12mA (default)
out_port_from_the_led_pio[4]	Output	PIN_AF8	8	B8_N0	1.8 V	12mA (default)
out_port_from_the_led_pio[3]	Output	PIN_AE8	8	B8_N0	1.8 V	12mA (default)
out_port_from_the_led_pio[2]	Output	PIN_W12	8	B8_N0	1.8 V	12mA (default)
out_port_from_the_led_pio[1]	Output	PIN_W11	8	B8_N0	1.8 V	12mA (default)
out_port_from_the_led_pio[0]	Output	PIN_AC10	8	B8_N0	1.8 V	12mA (default)

Fig. 6.1 Example .csv file that interfaces between board design SW and FPGA SW

The PCB design engineer also interfaces with the Signal Integrity engineer, by providing details of the board layout characteristics that are used to generate the model of the board for Signal Integrity modeling.

6.2.3 Signal Integrity Engineer

SI engineers are familiar with signal integrity simulation software from leading EDA vendors such as Synopsys, Mentor Graphics, Cadence, Agilent, etc. They are responsible for verifying that the signal quality (e.g. overshoot/undershoot), including simultaneous switching noise (SSN) effects are within specification. Ultimately, the SI engineer is responsible for verifying that the board timing meets the system requirements.

In the past, most FPGAs were designed without using the services of Signal Integrity Engineers. In truth many FPGAs are still being designed today without the services of SI engineers. Board designers have tended to lay the board out conservatively when interfacing with FPGAs and assumed, correctly in most cases, that this will meet their requirements. However, based upon the reasons stated earlier in this chapter, this approach is no longer adequate. The increase in I/O speeds for interfaces such as DDR II/III SRAM memories, plus the addition of high speed transceiver blocks require correct board termination to prevent SI and SSN issues.

These types of interfaces can be successfully designed by following the guidelines that are provided in the application notes provided by the FPGA vendors. However, each board design is different and it is recommended that SI engineers simulate the I/Os that have high performance requirements. This creates the requirement that the board designer interfaces with both the FPGA and the board designer.

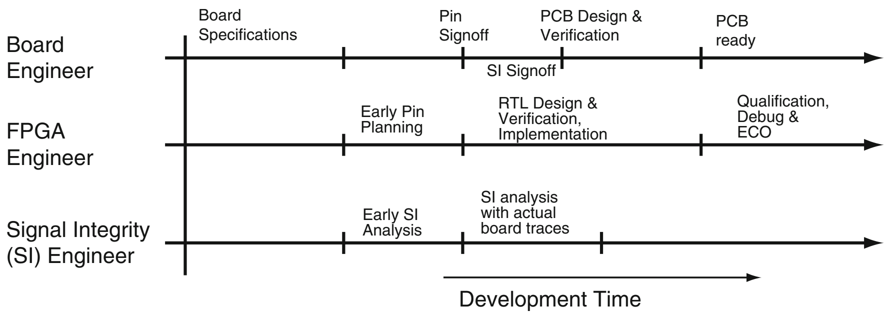

Board Engineer
Board Specifications
Pin Signoff
PCB Design & Verification
PCB ready
SI Signoff
FPGA Engineer
Early Pin Planning
RTL Design & Verification, Implementation
Qualification, Debug & ECO
Signal Integrity (SI) Engineer
Early SI Analysis
SI analysis with actual board traces
Development Time

Fig. 6.2 Design cycle diagram detailing engineering discipline involvement

He requires the HSPICE or IBIS models from the FPGA design engineer and the details on the board traces, etc. from the Board designers. SI simulations tend to be lengthy and should only be performed on the pins of the FPGA that are considered a high risk for Signal Integrity. That is the high performance I/O in the design.

The diagram in Fig. 6.2 details the stage in the design cycle where each of the engineering disciplines should be involved throughout the FPGA design cycle. The diagram is explained in more detail in the section of this chapter on Design Flows for creating the FPGA pinout.

6.3 Power and Thermal Considerations

FPGA power estimation helps guide power supply design for the board.

6.3.1 Filtering Power Supply Noise

In order to reduce system noise it is critical to provide clean and evenly distributed power to all devices on the board. Low frequency power supply noise can be filtered out by placing a 100 μF electrolytic capacitor adjacent to where the power line joins the PCB. If you are using a voltage regulator, the capacitor should be placed at the final stage that provides the Vcc signal to the devices.

In order to reduce the high frequency noise to the power plane it is recommended that decoupling capacitors are placed as close as possible to each Vcc and ground pair.

6.3.2 Power Distribution

A power bus network or power planes are used to distribute power throughout the PCB. A power bus network is the least expensive solution but does suffer from power degradation. As such this should only be considered for cost sensitive applications on two-layer PCBs.

The recommended approach is to use two or more power planes. The power planes cover the full area of the PCB and distribute Vcc evenly to all devices, providing good noise protection. It is recommended that you do not share the same plane for analog and digital power supplies. Virtually all FPGA devices now contain PLLs, thus board design must accommodate an analog and digital power plane for the FPGA.

In summary, the power distribution recommendations are:

– Use separate power planes for the analog and digital power supplies.
– Place a ground plane next to the PLL power supply plane.
– Avoid multiple signal layers when routing the PLL power.
– Place analog and digital components over their respective ground plane.
– Isolate the PLL power supply from the digital power supply.

6.4 Signal Integrity

Digital designs have not traditionally been impacted by transmission line effects. As system speeds increase, the higher frequency impact on the system means that not only the digital properties, but also the analog effects within the system must be considered. These problems are likely to come to the forefront with increasing data rates for both I/O interfaces and memory interfaces, but particularly with the high-speed transceiver technology being embedded into FPGAs. Transmission line effects can have a significant effect on the data being sent. However, as speed increases, high-frequency effects take over and even the shortest lines can suffer from problems such as ringing, crosstalk, reflections, and ground bounce, seriously hampering the integrity of the signal. Poor signal integrity causes poor reliability, degrades system performance, and, worst of all, causes system failures. The good news is that these issues can be overcome by following good design techniques and simple layout guidelines.

6.4.1 Types of Signal Integrity Problems

There are four general types of SI problems. These are Signal Integrity on one net, cross talk between adjacent nets, rail collapse and electromagnetic interference (EMI).

6.4.1.1 Signal Integrity on One Net

Drive strength specifies how much current the driver sources/sinks, while the slew rate specifies how fast it sources/sinks the current. Together, these two settings determine the rise and fall times of the output signal. Process technologies with smaller feature sizes allow faster clocks, but faster clocks also signify shorter rise and fall times. This means that switching times are reduced even on low frequency signals as the rise and fall times are set by the technology. This reduction of the switching time comes together with larger transient current; consequently, larger switching noise. For a high fmax link signal, it might be necessary to have short rise and fall times, but for a low fmax link signal, you may reduce the noise by using longer rise and fall times.

6.4.1.2 Crosstalk

Whenever a signal is driven along a wire, a magnetic field develops around the wire. If two wires are placed adjacent to each other, it is possible that the two magnetic fields interact causing a cross-coupling of energy between the signals known as crosstalk.

The following PCB design techniques can significantly reduce crosstalk:

1. Widen spacing between signal lines as much as routing restrictions allow.
2. Design the transmission line so that the conductor is as close to the ground plane as possible. This couples the transmission line tightly to the ground plane and helps decouple it from adjacent signals.
3. Use differential routing techniques where possible, especially for critical nets.
4. Route signals on different layers orthogonal to each other, if there is significant coupling.
5. Minimize parallel run lengths between signals. Route with short parallel sections and minimize long coupled sections between nets.

6.4.1.3 Rail Collapse

Rail collapse is noise in the power and ground distribution network feeding the chip. Switching I/Os can cause a voltage to form across the impedance of the power and ground paths. This effectively causes a voltage drop with less voltage reaching the FPGA, further accentuating the problem.

The solution is to design the power and ground distribution network to minimize the impedance of the power distribution system.

6.4.2 Electromagnetic Interference

EMI is a disturbance that affects an electrical circuit due to either electromagnetic conduction or radiation. The disturbance may interrupt, obstruct, or otherwise degrade or limit the effective performance of the circuit. The source of EMI is rapidly changing electrical currents.

FPGAs are rarely a source of EMI, however the possibility of EMI being generated increases with the use of heatsinks, circuit board planes and cables.

EMI can be reduced on FPGAs through:

1. The use of bypass or "decoupling" capacitors connected across the power supply, as close to the FPGA as possible.
2. Rise time control of high-speed signals using series resistors.
3. VCC filtering.
4. Shielding. This is typically used as a last resort due to the added expense of shielding components.

The two most common sources of EMI on boards are:

1. The conversion of differential signal into a common signal, which eventually gets onto an external twisted pair cable.
2. Ground bounce on a board generating common currents on external single-ended shielded cables.

These EMI effects can be controlled by grouping high speed signals away from where they might exit the product.

The key to efficient high-speed product design is to take advantage of analysis tools that enable accurate performance prediction. Use measurements as a way of validating the design process, reducing risk and increasing confidence in the tools.

6.5 Design Flows for Creating the FPGA Pinout

There are two flows that are recommended to successfully create an FPGA pinout for the board design. In both flows there is significant communication between the board designer and the FPGA designer.

6.5.1 User Flow 1: FPGA Designer Driven

In this design flow, the FPGA engineer generates the initial FPGA pin-out and provides the FPGA pin-out details to the PCB design engineer. The board design engineer makes suggested pin changes to ease the board design and provides these details to the FPGA engineer. The FPGA engineer makes the pin changes in the FPGA design software and confirms if the changes will work for the FPGA design. This process is continued until a final pin-out is obtained that meets the needs of both the FPGA designer and the board design engineer.

In reality the initial pin-out that is developed by the FPGA designer needs to be created with knowledge of the board layout, i.e. the relative location of the board components, such as memories, transceivers, microprocessors, etc. that the FPGA will interface with. The FPGA engineer can then make flexible pin assignments, such as assigning memory interfaces to particular I/O banks and leave the FPGA design software to make the actual pin assignments. This approach will speed-up the pin planning process such that the communication between the board design

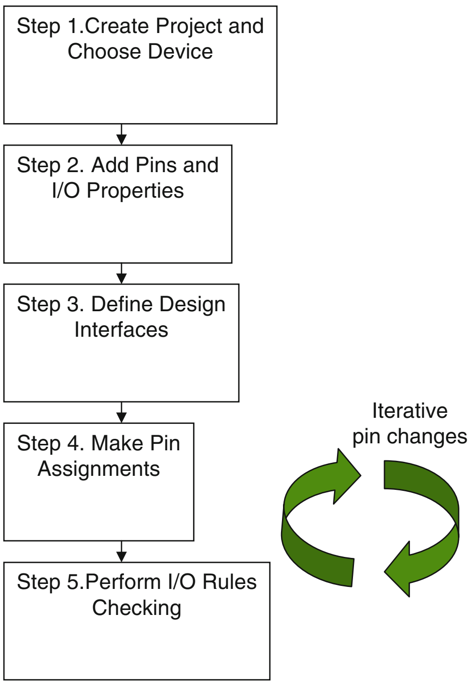

Step 1.Create Project and Choose Device
Step 2. Add Pins and I/O Properties
Step 3. Define Design Interfaces
Step 4. Make Pin Assignments
Step 5.Perform I/O Rules Checking
Iterative pin changes

Fig. 6.3 FPGA designer driven flow for creating the FPGA pin-out

engineer and the FPGA designer is basic pin swapping for ease of board design to minimize board trace crossovers, etc. as opposed to large scale changes (Fig. 6.3).

Step 1: This first step occurs in the FPGA design software. The FPGA designer will create an FPGA design project targeting the appropriate FPGA device and package. At this stage it is recommended that the designer enables any device migration capabilities that exist in the FPGA design software to accommodate future design expansion or contraction.

Step 2: The FPGA designer starts to enter pin information based upon the FPGA design. The FPGA design is unlikely to be complete at this stage in the design cycle however the interfaces must be solid. At a minimum, a top-level design file should exist. This provides enough information for the designer to enter the pin names and to start entering properties of the pins, such as I/O standard, current strength, etc. This information can be entered into the FPGA design software manually or in most cases can be imported from other sources, such as Microsoft Excel. The recommendation is that this information is defined in the specification for the design and that the specification enables this information to be available in the .csv format for import into the FPGA design software. This will greatly shorten this process and reduce the risk of human error.

If interface IP is being used, some of the IP may already contain the pin properties information. The source files should be added to the design. The FPGA design software can usually read in the pin properties information.

Step 3: Define the design interfaces by configuring the ports and parameters of any IP being used to make the port connections to the top-level HDL File. As mentioned previously, it is recommended that a top-level design file already exists, however, in the case were the specification is complete and the design file does not exist, some of the FPGA design software solutions can automatically generate a top-level HDL wrapper file based upon the Pin information that is entered in the FPGA design software. The top-level design file is needed to enable I/O rules checking in the FPGA design software. By creating the design interfaces, you are effectively creating a top-level block diagram of the interfaces to the FPGA design. By providing as much design information as possible to the FPGA design software, the more complete the I/O rule checks that can be performed by the FPGA design software.

Step 4: Make the pin assignments. If you know the exact pin locations that you want, you should enter them directly into the FPGA design software. These can often be imported for IP. If you only know the general area of the device that the pin needs to be assigned to, then you can make broader assignments such as I/O Bank 1 and allow the FPGA design software to select the actual pin location.

Step 5: Perform I/O rules checking and generate a valid pin-out. All FPGA design software has an I/O rule checking capability. This should be run to check the validity of the pin assignments. Some of the FPGA design software packages have the ability to generate pin assignments based upon assignments to a specific area of the device as opposed to specific pins. These assignments can be accepted by the user to replace the board assignments and passed to the board designer.

I/O rule checking options in the FPGA design software is limited in the mount of rules it can reliably check without a complete design. Hence, it is strongly recommended that you create a dummy design that includes all of the IP for the interfaces and clock network details. The interfaces can be terminated with dummy logic such as FIFO's where internal design blocks are not yet available. This approach enables the FPGA design software to check all of the I/O rules with confidence that the same pin-out can be used when the internal design blocks are added to the design in the future.

Steps 4 and 5 are now performed iteratively until an FPGA pinout is achieved that works on both the FPGA and the board.

As the design becomes complete any potential pin-out issues should be communicated back to the board designer and changes made at either the board or FPGA design level. Changes will not be required for dummy designs that are representative of how the final design will communicate with the pins in the FPGA.

6.5.2 User Flow 2

In this design flow, the PCB design engineer generates the initial FPGA pin-out in the board design software and provides the FPGA pin-out details to the FPGA design engineer. Optionally the Board Design Engineer can run the FPGA design software to enter the pin details. In reality this is rarely the case unless the same engineer is performing both the FPGA and board design. The FPGA design engineer makes the pin assignments in the FPGA design software and confirms if the assignments will work for the FPGA design. If there is an issue with the assignments, the FPGA design engineer makes suggested edits that the FPGA design software shows to be legal and feeds these changes back to the board designer. This process is continued until a final pin-out is obtained that meets the needs of both the FPGA designer and the board design engineer (Fig. 6.4).

Step 1: The board designer creates the FPGA pin assignments based upon the components on the board that will interface with the FPGA. This requires details on drive strength and clock restrictions on the FPGA. In reality the Board designer will work with the FPGA designer on this step, asking questions on where the transceivers are located on the device, power rail requirements and other possible restrictions to pin-out. The board designer will then create a first pass at creating the pinout and pass this information to the FPGA designer.

Step 2, 3 and 4: This is the same as steps 1, 2 and 4 in user flow 1. The FPGA designer will create the FPGA project, make the pin assignments and assign the pin properties.

Step 5: The FPGA designer can run the I/O rule checker to validate the pin assignments and communicate any recommended changes back to the board designer. This process will continue until a satisfactory pinout is achieved. As in user flow 1, the FPGA

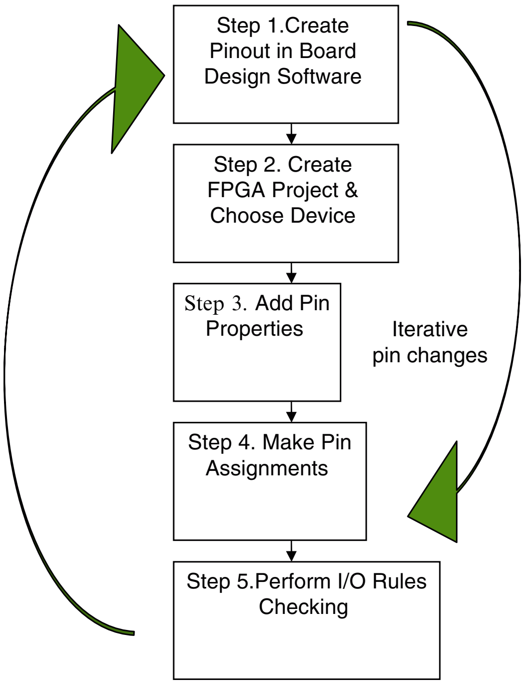

Step 1.Create Pinout in Board Design Software
Step 2. Create FPGA Project & Choose Device
Step 3. Add Pin Properties
Iterative pin changes
Step 4. Make Pin Assignments
Step 5.Perform I/O Rules Checking

Fig. 6.4 Board designer driven flow

designer should create a dummy design or use the real design to ensure that the pin-out will work.

6.5.3 How Do FPGA and Board Engineers Communicate Pin Changes?

There is a tendency to communicate the pin-out changes verbally or via email. However, this approach is prone to error. There needs to be an official document which resides in version control that is used to communicate the changes between the board designer and the FPGA designer. As mentioned earlier in this chapter, Microsoft Excel tends to serve this purpose in many Companies. One of the advantages of using Microsoft Excel is that many of the board design tools and some of the FPGA design software can import and export .csv files.

6.6 Board Design Check List for a Successful FPGA Pin-Out

1. Perform Power Thermal Analysis to ensure that all power planes can deliver the maximum current required while keeping the voltage rail within specification.
2. Perform pin assignment checking.
    a. Check pin assignments in FPGA design software
    b. Terminate unused inputs to Ground
    c. Terminate unused I/Os as desired
    d. Check correct VCCIO for each I/O bank
    e. Does design meet the SSN guidelines?
    f. Select migration devices to accommodate future design growth or reduction
3. Perform configuration mode check against vendor configuration handbook.
4. Check Power supply connections and decoupling against vendor power supply recommendations.
5. Perform board Signal Integrity simulations.
6. Compare I/O Timing to I/O Timing Requirements. This requires the design to be complete or at least the I/O interface portions of the design.
7. Complete board design review between FPGA design team and PCB design team.

Chapter 7
Power and Thermal Analysis

7.1 Introduction

The increase in density and performance of FPGAs has resulted in an increase in power consumed by the FPGA. Both FPGA and PCB design engineers need to consider the power when making the choice to use an FPGA and a particular FPGA vendor, as the power consumed by the FPGA will impact the design of the PCB power supplies, choice of voltage regulators, the heat sink and the system's cooling system. In short, it is crucial in developing the power budget for the entire system.

For applications that are power sensitive and where it is anticipated that meeting the power budget will be tight, the design engineer needs to perform power analysis during the development of the design and deploy power saving techniques as appropriate. Throughout the design cycle, the engineers need to be able to refine the estimates and apply the appropriate power management design techniques.

Today's FPGAs come with a variety of features for reducing the FPGA power, including power optimization options in the FPGA design software. Details on power optimization techniques are covered in the RTL coding guidelines and Timing Closure chapters of the book.

FPGA vendors also provide solutions for estimating the power that will be consumed by the FPGA at different stages of the design flow.

In this chapter we will review the basic elements of power consumption in FPGA devices, as well as the main factors that impact the ability of a designer to obtain an accurate estimation of a design's power consumption. We will look at the tools and techniques for performing power estimation very early in the design cycle, in order to enable the right choice of FPGA technology and to select the right power regulators and components for the board design. Then we will examine the tools and techniques to enable you to perform a more detailed power estimation based upon the design implementation. Finally we will review the best practice recommendations for dealing with power in FPGA designs.

DOI 10.1007/978-1-4419-6339-0_7, © Springer Science+Business Media, LLC 2010

7.2 Power Basics

Thermal power is the component of total power that is dissipated within the device package. Designers need to consider the thermal power in determining whether they need to deploy thermal solutions on the FPGA, such as heat sinks, to keep the internal die-junction temperature within the recommended operating conditions.

The total power consumed by a device, considering its output loading and external termination, is comprised of the following major power components.

7.2.1 Static Power

Static power is the power consumed by a device due to leakage currents when there is no activity or switching in the design. This is the quiescent state. This type of power is often referred to as standby power and is independent of the actual design. The amount of leakage current depends upon the die size, junction temperature, and process variation. This data can be extracted from the FPGA device data sheet or from the vendors Early Power Estimation Spreadsheet. It is recommended that you extract the data from the vendors Early Power Estimation Spreadsheet as the data is generally reported in a much clearer format than in most data sheets.

7.2.2 Dynamic Power

This is the power consumed through device operation caused by internal nodes in the FPGA toggling. That is, the charging and discharging of capacitive loads in the logic array and routing. The main variables affecting dynamic power are capacitance charging, supply voltage, and clock frequency. A large portion of the total dynamic power consumed in FPGAs is due to the routing fabric of the FPGA device.

Dynamic power is design dependent and is heavily influenced by the users RTL style.

7.2.3 I/O power

This is the power consumed due to the charging and discharging of external load capacitors connected to the device output pins and any external termination networks. Again, I/O power is design dependent and is impacted by the I/O standard, data rate, the configuration of the pin as either input or output or bidirectional. The termination on inputs, and the current strength, slew rate and load for outputs impact the I/O power.

7.2.4 Inrush Current

Inrush current is the current, hence power, that the device requires during initial power-up. During the power-up stage, a minimum level of logic array current (ICCINT) must be provided to the device, for a specific duration of time. This duration depends on the amount of current available from the power supply. When the voltage reaches 90% of its nominal value, the initial high current is usually no longer required. As device temperature increases, the inrush current required during power-up decreases, however the standby current will increase.

7.2.5 Configuration Power

Configuration power is the power required to configure the device. During configuration and initialization, the device requires power to reset registers, enable I/O pins, and enter operating mode. The I/O pins are typically tri-stated during the power-up stage, both before and during configuration in order to reduce power and to prevent them from driving out during this time.

7.3 Key Factors in Accurate Power Estimation

Before discussing the best approach to performing power and thermal analysis for an FPGA design, we will look at the key factors for accurate power estimation (Fig. 7.1).

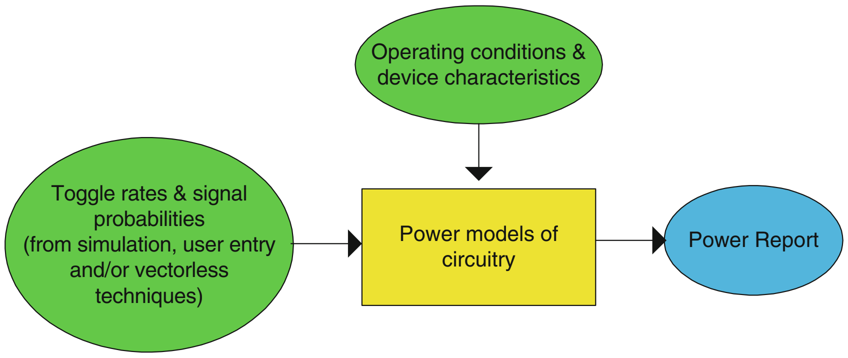

Operating conditions & device characteristics
Toggle rates & signal probabilities (from simulation, user entry and/or vectorless techniques)
Power models of circuitry
Power Report

Fig. 7.1 Key elements in accurate power estimation

7.3.1 Accurate Power Models of the FPGA Circuitry

These are the models that are provided by the FPGA vendors as part of their power estimation solutions. The FPGA design engineer must trust that the FPGA vendor is being honest with the models. These models are typically developed from HSPICE and the models correlated with silicon characterization. This process varies slightly across FPGA vendors. The accuracy of the models will vary depending upon the maturity of the FPGA family. If the FPGA family is new to the market, the power models will be preliminary and subject to change as the FPGA vendor completes characterization of the family. The negative impact of the variation should be minor if the FPGA vendor is conservative in the development of the initial HSPICE models. Asking the silicon vendor for details on how they develop their power models will help set your expectations on the accuracy of the models.

7.3.2 Accurate Toggle Rate Data on Each Signal

Toggle rate data, also referred to as Signal Activity, relates to the performance of the design. While clock speed is important, the average number of times that a signal changes value per unit of time is more important as this transition impacts the power consumption.

A logic ‘1’ condition consumes more power than a logic ‘0’, thus the amount of time that a signal is logic ‘1’ will impact power. This tends to have an impact on I/O power on pins that use terminated standards.

Toggle rate data is under the control of the FPGA design engineer, in that it is dependent upon system operation. This information is usually derived from design simulations or toggle rates which are based upon previous design experience. As such, entering reasonably accurate toggle rate data is an easier task for designs that are derivatives of previous designs than for new designs. I cannot overemphasize the importance of using toggle rate data that is reflective of the end system operation, as gross inaccuracy in the prediction of the toggle rate is the main source of error in power estimation.

In many cases, the simulation data fails to represent real world operation. If simulation is performed for the purpose of measuring code coverage, it is likely to over predict the power that will be used in operation. As a designer, you need to avoid the dangerous situation of under predicting the toggle rate, as this will result in an under estimation of power. However, an over prediction of power may result in a more expensive power management solution.

The power estimation solutions from the FPGA vendors assume a default toggle rate of 12.5% unless specified otherwise by the FPGA design engineer. For many applications, this is sufficient very early in the design cycle, as most designs do not have a high toggle rate on all nodes, and the end application is specified to cope with a margin of error within 30% of the total power. However, this may not be the case for designs in which the majority of the design performs high performance processing, as is the case in many DSP processing applications. These designs will typically exhibit a higher toggle rate.

The FPGA vendor power estimation solutions allow you to easily change the toggle rate values and to quickly see the impact that it has on power. It is recommended that you do what you can to correctly estimate the toggle rate for your application. It is also recommended that if you are not sure of the toggle rate that you try a range of toggle rate values to indicate a possible best case and worst case scenario. Note that it is unlikely that a complete system design will have a toggle rate above 40%.

7.3.3 Accurate Operating Conditions

When we look at the impact of temperature on standby power, particularly for devices at process geometries of 65 nm and below, we can see that there is a dramatic increase in power above Tj of 85°C (Fig. 7.2).

Temperature has a big impact on static power, as the leakage power is an exponential function of Tj. High leakage increases Tj, which, in turn, further increases the leakage, forming a potential positive feedback loop. Tj = Ta + θja × (standby power + dynamic power) where Ta is the ambient temperature, and θja is the thermal resistance between the device junction and ambient air. It is essential to ensure that the junction temperature remains within its operating range and does not enter a positive feedback loop. The more power a device consumes, the more heat it generates and this heat must be dissipated to maintain operating temperatures

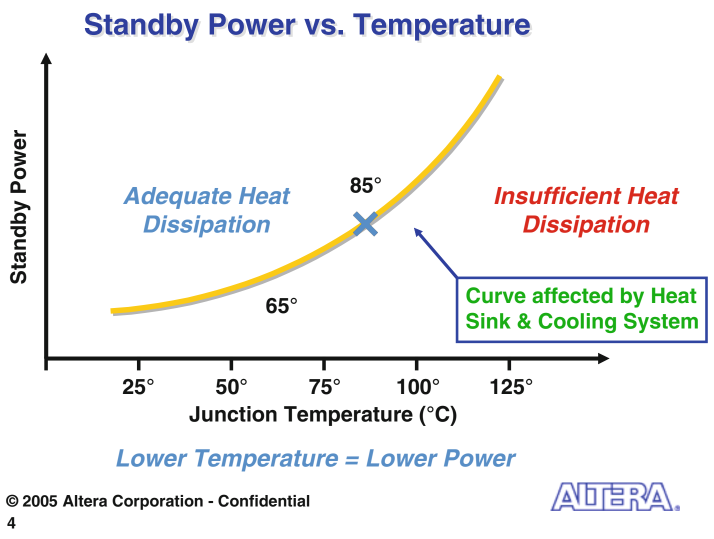


Fig. 7.2 Graph of standby current versus temperature

within specification. For the FPGA and board designer it is essential that this is modeled during power estimation and that the tools used to calculate the power consider the heatsink used, air flow and other factors to correctly model Tj.

Thus it is important that the FPGA and/or board design engineer uses the appropriate thermal management technique to minimize power consumption.

7.3.4 Resource Utilization

There is a fourth element that impacts power and that is the utilization of the resources in the FPGA device. In general, the more logic used, the more power consumed.

However, as a designer you need to be aware of the impact of the different types of resources in the FPGA device on power. As the designer or implementer, you have the ability to trade-off resource type usage, e.g. Logic element usage versus dedicated hardware blocks, such as RAM and DSP Blocks.

If you look at a typical FPGA design, approx. 65% of the power is core dynamic power, 24% is core static power, 10.5% is IO dynamic power and about 0.5% is IO static power.

If we dig into the core dynamic power in more detail, the majority of it can be attributed to routing and combinational logic in the logic elements. RAM blocks also consumes significant dynamic power.

The dynamic power for the clock networks consists of the global clock routing resources plus the power consumed by the local clock distribution within the LEs, RAM and DSP blocks. Designers can control the dynamic via the choice of resource type and the use of clock control blocks. This is discussed in more detail in Chap. 12.

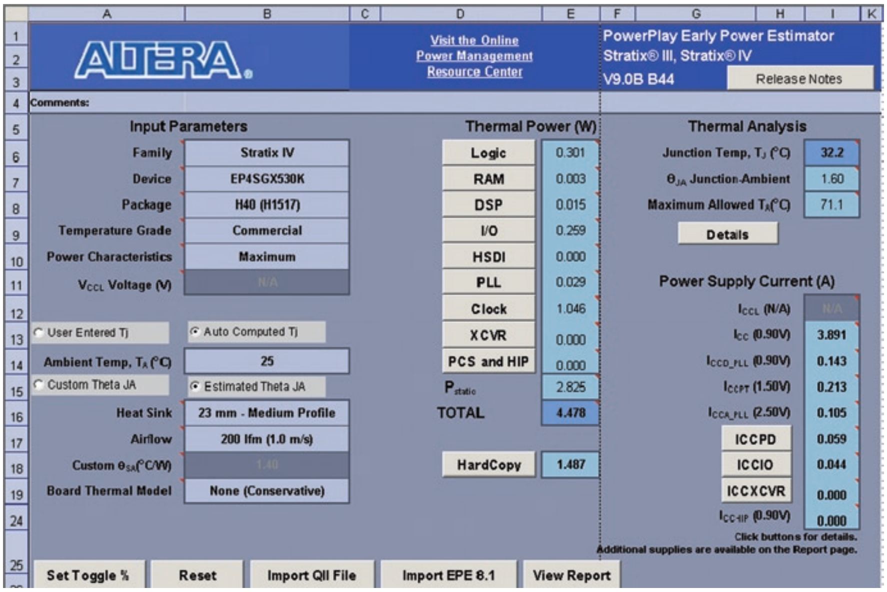
7.4 Power Estimation Early in the Design Cycle (Power Supply Planning)

As mentioned previously, FPGA Vendor data sheets do not provide much data on the typical power consumption of an FPGA family. FPGA vendors do however provide Power Estimation tools to report the power for a given device.

Early FPGA power estimation helps guide power supply design for the board. More often than not, this task needs to be performed before the FPGA design is complete or started. The power estimation spreadsheets provided by the FPGA vendors can be used to estimate the power for your design and to perform preliminary thermal analysis on your design at various stages of the design cycle.

Figure 7.3 shows a sample power estimation spreadsheet for the Altera Stratix® IV GX family

The vendor provided spreadsheets are based upon Excel and can be downloaded from the FPGA vendor website free of charge. The accuracy of the power estimation increases as you provide more information that is indicative of your operating conditions



Visit the Online Power Management Resource Center

PowerPlay Early Power Estimator
Stratix® III, Stratix® IV
V9.0B B44

Release Notes

Comments:

Input Parameters	
Family	Stratix IV
Device	EP4SGX530K
Package	H40 (H1517)
Temperature Grade	Commercial
Power Characteristics	Maximum
V_{CCL} Voltage (V)	N/A
User Entered Tj	[✓ Auto Computed Tj]
Ambient Temp, T_A (°C)	25
Custom Theta JA	[✓ Estimated Theta JA]
Heat Sink	23 mm - Medium Profile
Airflow	200 lfm (1.0 m/s)
Custom θ_{SA}(°C/W)	1.40
Board Thermal Model	None (Conservative)

Thermal Power (W)	
Logic	0.301
RAM	0.003
DSP	0.015
I/O	0.259
HSDI	0.000
PLL	0.029
Clock	1.046
XCVR	0.000
PCS and HIP	0.000
P_{static}	2.825
TOTAL	4.478
HardCopy	1.487

Thermal Analysis	
Junction Temp, T_J (°C)	32.2
θ_{JA} Junction-Ambient	1.60
Maximum Allowed T_A(°C)	71.1

Details

Power Supply Current (A)	
I_{CCL} (N/A)	N/A
I_{CC} (0.90V)	3.891
I_{CCD_PLL} (0.90V)	0.143
I_{CCPT} (1.50V)	0.213
I_{CCA_PLL} (2.50V)	0.105
ICCPD	0.059
ICCIO	0.044
ICCXCVR	0.000
I_{CC-HIP} (0.90V)	0.000

Click buttons for details.
Additional supplies are available on the Report page.

Set Toggle % | Reset | Import QII File | Import EPE 8.1 | View Report

Fig. 7.3 Sample power estimation spreadsheet for the Altera Stratix IV GX family

and of the final design. The maturity of the devices will also impact the accuracy, i.e. are the vendor power models final or preliminary. With minimal effort this can provide a good ballpark estimate on power, i.e. within 30% of real numbers; enabling you to choose the right FPGA technology for your application and to specify the power supply design. By investing more time on entering more detailed data on your design and operating conditions, you can typically get within 20% of the real power. These tools allow designers to enter details on their design and operating conditions. Some of the FPGA vendor tools have the capability to import data from their compiled designs into the Power Estimation Spreadsheet. This feature works well for partial designs or estimating power based upon legacy information. This information serves as a starting point and the details, such as the different resource counts, number of clocks, etc. can be edited in the spreadsheet to reflect the expected size and characteristics of the final design. This is a much quicker and less error prone approach to entering data by using the power estimation and analysis solutions that exist in the FPGA vendor software as discussed in Sect. 7.5.

7.5 Simulation Based Power Estimation (Design Power Verification)

Simulation based power estimation provides the most accurate power estimation solution, providing the simulation vectors are representative of real system operation. Simulation based power estimation uses the results from running a simulation in standard EDA tools, such as Mentor Modelsim, Synopsys VCS and Cadence Incisive, to name a few, in order to simulate the device operation. The resulting simulation data is used as stimulus to the FPGA vendor simulation based power estimation tool.

A Vector Change Dump (VCD) file is normally used to transfer the data from the EDA simulation tool to the FPGA vendor software. The reason why the power estimation solution in the FPGA vendor software is more accurate than the spreadsheet power estimation solutions is that full Place and Route has been completed on the design and at this point the modeling takes into account the actual placement and the routing types used on the design. The ability to use real life operation vectors also has a large impact on the accuracy of the estimation.

Having a design plus accurate simulation vectors implies that the design is complete or is very close to being complete. Therefore it is recommended for most designs that this type of analysis is run towards the end of the design cycle to determine what the real power consumption is for the design. Thus, it is more of a sanity check that the design is within power budget rather than something that is run continuously throughout the design cycle.

An exception is power sensitive designs where this data can be used to determine if the RTL needs to be optimized for power or whether to utilize power optimization options that exist in the FPGA vendor software. Simulation based power estimation can be run early in the design cycle on blocks of RTL that already exist to determine the toggle rate on these blocks for use in the spreadsheet based power estimation solutions. The power report on these blocks of reusable IP can also be included in the documentation on the blocks to give other users of the design blocks or IP, background data on the expected power consumption for the block.

One of the challenges with simulation based power estimation is that the most accurate power estimation is based upon gate level simulation of the design, as the toggle rate data from the simulation will be available for every node in the design. However this type of simulation tends to be runtime intensive for certain application spaces, such as video and image processing. So while this type of analysis provides the most accurate power results, the simulation time may make it impractical for certain applications. Thus, it is recommended that RTL simulations be used for these types of applications. Gate level simulations can be run as a sanity check on the design, i.e. only to model certain operating conditions of the design. It is recommended that you use gate level simulation if the simulation time is feasible for your end application.

An RTL simulation will contain the correct toggle rate on the I/O pins and on most of the registers. There will be some level of inaccuracy on the registers as synthesis will perform register duplication and register merging as part of its optimizations. The combinational nodes will also be inaccurate as the names will not match due to the optimizations performed. This however is not a huge issue, as most of the simulation based power estimation solutions contain a mode called vectorless estimation, which can be combined with RTL simulation based estimation to provide an acceptable level of accuracy.

Vectorless power estimation uses a statistical analysis approach to predict the probability of the nodes between known good data points toggling. If we look at the circuit in Fig. 7.4, if we know the static probabilities and toggle rates of inputs A, B, C, D, E, F, G and H, it is possible to estimate the static probabilities and toggle rates at I, J, K, L; hence the final output M.

This capability can be used to enhance the accuracy of RTL simulation based estimation. As part of best practices we recommend running a sample of gate level simulations, but for long simulations, RTL + Vectorless estimation is the recommended approach. It is also advised that you perform simulation based estimation at certain checkpoints throughout the design process. In reality, at this stage in the project this should be more of a sanity check rather than a necessity. After performing the early power estimation, you ought to have left sufficient headroom on the power budget such that you are not constantly optimizing your design for power. As with Early Power Estimation, you need to vary the operating conditions in terms of temperature and voltage, to ensure that you are reflecting the real world operating conditions.

The simulation based power estimation tools generate reports aimed at facilitating both thermal and power supply planning requirements. These reports pinpoint which device structures and even design hierarchy blocks are dissipating the most thermal power, thus enabling design decisions that reduce power consumption. This provides very high quality power estimates which are usually within 20% of device measurements, provided the toggle rate data is accurate (Fig. 7.5).

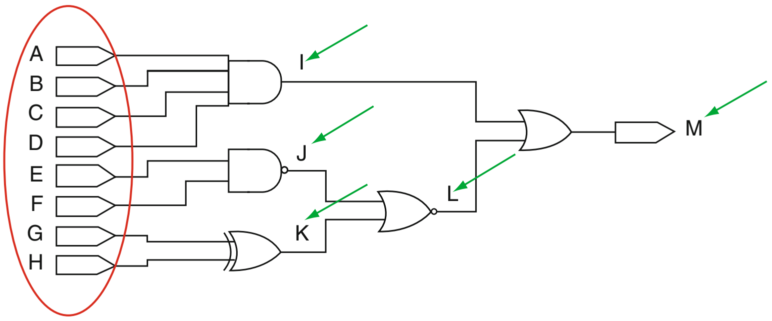
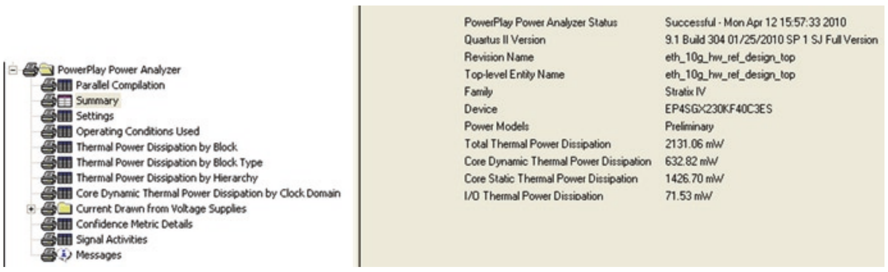


Fig. 7.4 Probability of nodes toggling


PowerPlay Power Analyzer
Parallel Compilation
Summary
Settings
Operating Conditions Used
Thermal Power Dissipation by Block
Thermal Power Dissipation by Block Type
Thermal Power Dissipation by Hierarchy
Core Dynamic Thermal Power Dissipation by Clock Domain
Current Drawn from Voltage Supplies
Confidence Metric Details
Signal Activities
Messages

PowerPlay Power Analyzer Status	Successful - Mon Apr 12 15:57:33 2010
Quartus II Version	9.1 Build 304 01/25/2010 SP 1 SJ Full Version
Revision Name	eth_10g_hw_ref_design_top
Top-level Entity Name	eth_10g_hw_ref_design_top
Family	Stratix IV
Device	EP4SGX230KF40C3ES
Power Models	Preliminary
Total Thermal Power Dissipation	2131.06 mW
Core Dynamic Thermal Power Dissipation	632.82 mW
Core Static Thermal Power Dissipation	1426.70 mW
I/O Thermal Power Dissipation	71.53 mW

Fig. 7.5 Sample power estimation report from Quartus II PowerPlay Estimator

7.5.1 Partial Simulations

One of the challenges in a simulation based approach to power estimation is the initialization time in the testbench and hence simulation. This can reduce your effective toggle rate if the simulation is not run to reflect a long period of operation. You can perform a simulation where the entire simulation time is not applicable to the signal activity calculation, reducing the accuracy of the estimation. For example, if you run a simulation for 10,000 clock cycles and reset the chip for the first 2,000 clock cycles. If the signal activity calculation is performed over all 10,000 cycles, the toggle rates are typically only 80% of their steady state value (since the chip is in reset for the first 20% of the simulation). Some of the FPGA vendor solutions allow the user to specify the useful parts of the *.vcd* file for power analysis, enabling you to ignore the initialization stage as part of the power estimation.

7.6 Best Practices for Power Estimation

See Fig 7.6.

Stage of Design Cycle	Task	Tools	Additional Content
	Device Selection	FPGA Vendor Power Estimation Spreadsheet	Legacy Designs Previous Experience Design Specification Early RTL code
	Board Power Supply Specification	FPGA Vendor Power Estimation Spreadsheet FPGA Vendor Board Design Guidelines	Legacy Designs Previous Experience Design Specification Early RTL code
Early Power Estimation	Board Design	FPGA Vendor Power Estimation Spreadsheet FPGA Vendor Board Design Guidelines	
Evolving Design	Spot check Power Based Upon Evolving Design Estimate Power for Power Optimization	FPGA Vendor Power Estimation Spreadsheet FPGA Vendor Simulation Based Power Estimation Tool	HDL Design Testbench EDA Simulation Tools
Final Design	Determine Final Power Estimate Power for Power Optimization Measure Power on Board	FPGA Vendor Simulation Based Power Estimation Tool	HDL Design Testbench EDA Simulation Tools Final Board and Test Equipment

Fig. 7.6 Best practices for power estimation

Chapter 8
RTL Design

8.1 Introduction

The high level challenges that designers face when writing RTL for FPGA devices are similar to the challenges that are faced when writing RTL code for ASICs.

1. What is the goal for my design block?
2. Am I trying to achieve the highest performance or smallest area?
3. Is my code functionally correct and is it easy to synthesize in the target synthesis tool?
4. Is my RTL code reusable?
5. Is my design easy for place and route to successfully compile the design?

There are however unique high level goals that apply to writing RTL for FPGAs.

1. Is my RTL optimized for the target FPGA architecture or can the RTL be targeted across multiple FPGA architectures?
2. Is my RTL optimized for compile time?

As we look in more detail at writing RTL for FPGAs, we come across more differences compared to writing RTL for ASICs. These differences are due to the architecture of FPGA devices. This provides us with the first rule of writing RTL for FPGA devices; "understand the architecture of the target FPGA."

This chapter provides getting started tips to designers of various backgrounds. It describes some general FPGA architecture features, before covering general good practices in writing RTL. It then provides RTL coding guidelines that are optimized for FPGA architectures, before ending with a summary of best practice recommendations of RTL design for FPGAs.

8.2 Common Terms and Terminology

HDL: Hardware Description Language is a software programming language that is used to model a piece of hardware.

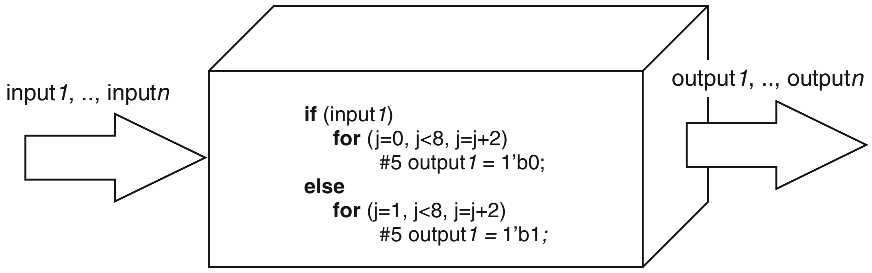
RTL: Register Transfer Level, defines input-output relationships in terms of dataflow operations on signals and register values.

Behavior Modeling: A component is described by its input-output relationship. Only the functionality of the circuit is described and not the structure of the end implementation. There is no specific hardware intent and the coding style is generic such that it can target any technology (Fig. 8.1).

Structural Modeling: A component is described by interconnecting lower-level components and primitives. It describes both the functionality and structure of the circuit.

It is created with the implementation of the hardware in mind (Fig. 8.2).

Synthesis: This is the translation of HDL to a circuit and then the optimization of the circuit. Basically the RTL description of your design is interpreted and hardware created for the targeted FPGA architecture. The synthesis tools require certain coding styles to generate correct logic. The coding style is important for fast and efficient logic (Fig. 8.3).



```
if (input1)
    for (j=0, j<8, j=j+2)
        #5 output1 = 1'b0;
else
    for (j=1, j<8, j=j+2)
        #5 output1 = 1'b1;
```

Fig. 8.1 Behavioral modeling

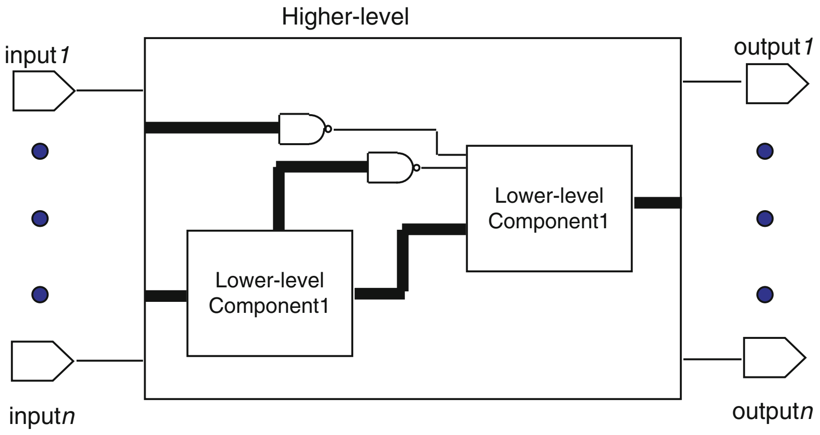
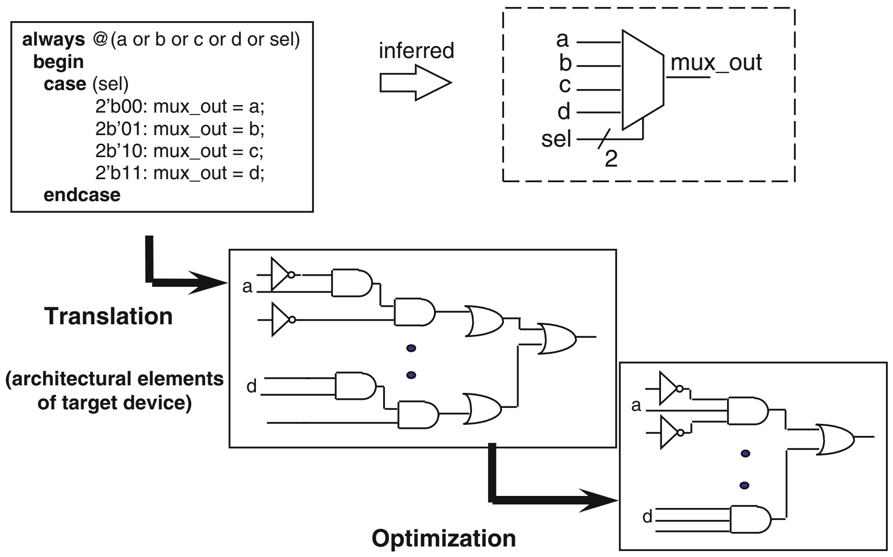


Fig. 8.2 Structural modeling



```
always @(a or b or c or d or sel)
  begin
    case (sel)
         2'b00: mux_out = a;
         2b'01: mux_out = b;
         2b'10: mux_out = c;
         2'b11: mux_out = d;
    endcase
```

Fig. 8.3 Synthesis

8.3 Recommendations for Engineers with an ASIC Design Background

The first thing to be aware of is that FPGAs are loaded with registers. Whether you use them or not, they are in the device that you have purchased. One way to look at it is that registers are free, therefore use them or lose them.

This use of registers is important for the performance of your FPGA design. FPGA logic is generally slower than that of ASICs on the same process geometry. Make use of the registers to pipeline your design to meet the design performance requirements.

Many ASIC designs make use of latches. Do not do this in FPGA designs. Use registers in place of latches. This will significantly improve the FPGA clock performance, albeit potentially at the cost of latency.

A common technique in ASIC designs for power reduction and for design testability is to use gated clocks. In FPGA designs, do not gate the clock. Use the "clock enable" instead. FPGA devices have a limited number of low skew clock networks that are key to running the design at high performance. By gating the clock you will exhaust the number of low skew global signals, thereby limiting the design performance. Clock enable signals are available on all registers in the FPGA and can be used to achieve power reduction and to test the design functionality without inflicting unrecoverable damage on the performance of your design.

FPGA devices do not provide the option of using buffers as a safety net to boost the performance in the design. Thus, when designing timing critical portions of your design, it is best to be conservative and to guard band your timing requirements.

While you pay for resources in FPGA devices, whether you use them or not, the resources are limited to the density of the targeted device. You are limited to the amount of logic, memory blocks and multiplier blocks in the targeted device. In addition, there is a fixed amount of routing in FPGAs. As your design reaches the higher boundaries of device utilization, you are likely to see the performance of your design start to drop off.

8.4 Recommended FPGA Design Guidelines

8.4.1 Synchronous Versus Asynchronous

In summary, practice Synchronous Design. It will help you to meet your design goals consistently.

Asynchronous design techniques can result in a reliance on propagation delays in a device, incomplete timing analysis, and possible glitches.

In a synchronous design, a clock signal triggers all events. As long as all of the registers' timing requirements are met, a synchronous design behaves in a predictable and reliable manner for all process, voltage, and temperature (PVT) conditions. This will enable you to target synchronous designs to different device families or speed grades.

8.4.2 Global Signals

The FPGA design software will automatically select global routing resources. Global signal resources are limited and thus should be treated as being expensive. It is recommended that you try to limit the number of clock domains whenever possible. You can control the selection yourself, but it is rare that you will achieve better results than the automated software.

You must select a reset scheme for your FPGA design, be it synchronous or asynchronous. You need a system reset that puts your entire circuit in a well-defined state and you should verify its operation by asserting it at the start of the testbench simulation.

If you are unsure as to which scheme is best for your system, use synchronous as it is easier to understand.

If you decide to use an Asynchronous reset, the asynchronous reset should be driven by a synchronizer as shown in Fig. 8.4.

Why should an asynchronous reset be driven by a synchronizer?

When the reset is released, there is no sure way of knowing when this occurred in relation to the clock. Some registers may see the clock first, some the released reset resulting in mixed register states. If you have a short reset, it may not be seen at all.

The synchronizer circuit in Fig. 8.4 mitigates all of these issues.

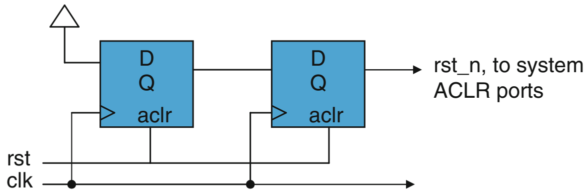


Fig. 8.4 Synchronizer for an asynchronous reset

8.4.2.1 Clock Network Resources

FPGAs provide device-wide global clock routing resources and dedicated inputs. You should use the FPGA's low-skew, high fan-out dedicated routing where available.

You should limit the number of clocks in your design to the number of dedicated global clock resources available in your FPGA. Clocks feeding multiple locations that do not use global routing may exhibit clock skew across the device that could lead to timing problems.

The use of combinational logic to generate an internal clock adds delays on the clock line. In some cases, the delay on a clock line can result in a clock skew greater than the data path length between two registers. If the clock skew is greater than the data delay, the design will not function correctly.

8.4.3 Dedicated Hardware Blocks

All FPGA vendors provide custom resources, designed to perform a small set of functions very efficiently. However, by instantiating these functions in your RTL code, you are locking your code to one vendor or possibly even to one FPGA family. This effectively reduces the reusability of your design. You are also likely to suffer from slower RTL simulation. Your behavioral description of your mode of RAM operation is likely to simulate much faster than the parameterized RAM model from the FPGA vendor. The FPGA vendor model covers every possible usage scenario and subsequently can simulate more slowly.

In some cases you may have no other option other than to use these optimized macros, as they may be the only way to access certain capabilities of the device. Examples of where these would be used are PLLs for the clock tree, or transceiver blocks for high speed serial interfaces. It is normal practice to use the vendor provided building blocks for these types of applications. They can usually be replaced by the equivalent technology primitives from other families or vendors with minimal disruption to your design. Much like using purchased IP.

However, you may want to consider inferring the other blocks such as the internal RAM blocks and DSP blocks. These need only be instantiated if you need access to underlying technology that cannot be reached by RTL inference.

These functions from the FPGA vendor have a limited degree of parameterization and usually come with a wizard to help select the right parameters along with the user documentation.

Fig. 8.5 Instantiation versus inferencing

Instantiation:	Inference:
Pros	Pros
Easy to do, GUI assisted	Architecture independent
Fully leverages HW features	Simple to simulate
Cons	Cons
Architecture specific	Fiddly hand-coding
Requires library files to simulate	Dependency on CAD tool

8.4.3.1 Instantiation Versus Inferencing

See Fig. 8.5.

8.4.4 Use of Low-Level Design Primitives

This section deals with the use of vendor specific low level design blocks, such as carry chains and LUT primitives to implement your design.

FPGA designers have been using this design technique since the invention of the FPGA. In the dark and distant past, it was the only way to guarantee the implementation of your design through synthesis. EDA synthesis tools have become a lot smarter over the years to the point where using this design style has become the exception as opposed to the norm. It really is akin to assembly level programming for hardware design or designing in schematics, only more painful in that you have to declare the wiring connections of the blocks in HDL.

So why has this style of design not disappeared completely? After all it is a tedious way of designing, synthesis tools are now exceptionally smart and the use of these low level primitives can reduce the ability to reuse the design block.

Well, in certain cases a good designer can still outsmart a synthesis tool. Take addition for example. Synthesis tools tend to restructure arithmetic and absorb logic that feeds adder chains opportunistically. The absorption is heuristic and occasionally produces sub-optimal groupings. If a designer thinks about the target hardware and structures the HDL accordingly, he can ensure that he gets the densest possible packing. The use of the low-level primitives makes the intent explicit, independent of the surrounding logic. An example where this approach to design is useful would be where you need to bit slice an adder, to clearly identify the intended carry-in and carry-out signals.

It is recommended that you avoid using these low-level primitives, unless performance or area packing is a problem for your end design. Use standard RTL coding techniques and if you cannot get the implementation that you need for the design, then consider using low level primitives to achieve your goal. It is possible to build up your own library of blocks comprised of low level primitives, e.g. an optimized ternary adder, or CRC. However you need to be aware that these blocks can only be reused with that FPGA vendor and in some cases, only with that particular FPGA family.

8.4.5 Managing Metastability

If the data at the input to a register violates the registers setup and/or hold time requirements, the output of the register may go into a metastable state. In this state, the output of a register oscillates at a value between the high and low states. If this value propagates throughout the circuit, registers may latch the wrong value, causing system failure.

Metastability problems commonly occur when a data signal is transferred between two sets of circuitry that are in unrelated clock domains.

It is good practice for asynchronous signals to travel through two to three registers before being used in order to avoid potential metastability issues (Fig. 8.6).

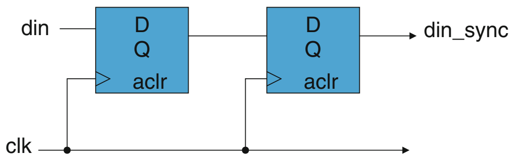


Fig. 8.6 Two-register synchronizer

8.5 Writing Effective HDL

The first rule in writing effective RTL is to divide and conquer. Try to split the design into smaller, unrelated problems for ease of tackling. Start with the areas of the design that you expect to be problematic, particularly the bus interfaces. The system should be designed such that you can exercise and test individual blocks, even if all blocks are not yet present in the design. Besides helping out early in the development process, this practice will allow you to make progress when specific blocks of your design are being revised or are otherwise unavailable.

Follow good synchronous design practices; asynchronous designs that are possible in ASICs because of tight control over timing delays can easily run into trouble in FPGAs. Pipelining your design, as well as registering all ports provides several benefits. First, it breaks combinational logic into more easily synthesizable portions. Pipelining also allows easier debugging since FPGA verification tools can easily access the inputs and outputs of registers. Finally, it allows more options for optimizing performance through register placement.

8.5.1 What's the Best Language

For the purposes of this book we are only going to consider HDLs that have an IEEE standard associated with them, i.e. VHDL, Verilog and SystemVerilog.

In the distant past there were numerous HDLs for targeting PLDs. Some of these were developed by FPGA vendors. Once the IEEE endorsed Verilog and VHDL as standards, these languages quickly conquered the ASIC design market and gained in popularity in the FPGA market. Verilog, including SystemVerilog, and VHDL provide the advantage of allowing users to be able to use the same language for design implementation as for describing the test stimulus for simulation. Today, Verilog and VHDL have effectively obsoleted the old PLD languages.

So, which of these languages is the best language for FPGA design?

There isn't a "best" language. All of these IEEE standard languages have strengths and weaknesses.

VHDL tends to be more verbose than Verilog, but also tends to be more feature-rich. VHDL has strong type checking which makes it harder to make silly mistakes.

Verilog is concise but loosely typed.

In summary Verilog and VHDL both work well for FPGA design. The choice of language is based upon personal preference. The key ingredient is that when you choose a language, make sure that you fully understand the language. Read up on the details of the language, as there are many non-obvious semantics in both languages.

A good starting point is to buy a copy of the relevant IEEE standard. While standards can make for dry reading, they will cover the details that HDL design books often gloss over.

There is an abundance of material on the web from white papers to training courses on HDL coding. These are good for getting a feel for the language and building a base knowledge in the language. I recommend paying for the cost of a hands-on HDL course from one of the many technology training vendors, local Colleges, EDA vendors or FPGA vendors. The instructors will tend to have a wealth of information that is often not covered in books and the hands on experiments will give you experience in the tools that you will use for creating the design.

8.5.1.1 Mixed Language Design

Most of the EDA synthesis tools on the market support designs that contain a mix of HDLs. There are however challenges in doing this and as such, it is recommended that you do not adopt a mixed language design unless you have no option.

So when would you have no other option but to use a mixed language design?

1. If you purchase IP that is written in a different HDL than the one that you have standardized on.
2. You are reusing design blocks from another design that was created in the "other" HDL.

If your organization has a "genius" that prefers a different language to the language that you have chosen, this is not a good reason to use mixed language design. This "genius" needs to comply with the Company's standard.

So, what are the problems that you may encounter when creating a mixed language design.

1. It is easy to make a non-portable design. There is no IEEE standard for mixed language design; consequently EDA tools make up their own rules, which can result in a non-portable design.
2. Verilog is case sensitive, VHDL is not. If you deploy case sensitivity into your naming scheme you could be heading into a minefield.
3. Not all simulators support mixed language design. Most of the major EDA simulation tools do, but it will cost more than the entry level version of the simulation tool.

So while it is recommended that you avoid mixed language design it can work if a module or entity to be instantiated in another language has bit or vector ports and simple parameter types.

8.5.2 Good Design Practices

8.5.2.1 Documented Code

It should be common practice in an organization to include good documentation on major design blocks. This is an additional document to the RTL code for the design. This document should explain the structure of the design, including block diagrams and a description of the hierarchy. It should also include a description of timing details, such as which paths are timing exceptions. Timing exceptions are covered in detail in the timing analysis chapter of this book.

Documentation on major design blocks, such as block diagrams is essential for design reuse. If you do not understand what you are trying to reuse, you are unlikely to be successful in reducing your design cycle through design reuse. Documentation is also very helpful when you are returning to a design that you completed in the past and for the training of new hires in the organization who are taking over the maintenance of, or completion of your design block.

The RTL code for the design block should be self documenting, i.e. the naming conventions used in the RTL should be descriptive of what the signal is doing, e.g. dram_ctrl, regfile0, crc32, egress_buffer. Comments should be used extensively throughout the RTL to explain the functionality of the code, e.g. identification of test signals or multicycle paths and the purpose of certain modules within the design.

8.5.2.2 Recommended Signal Naming Convention

Create a company naming convention and adhere to it!

A standard naming convention needs to exist throughout your Company.

This will make code reviews much more productive. There are EDA tools on the market to help establish coding guidelines and to enforce the coding standards. I highly recommend that you invest in an EDA Lint tool to enforce your Companies coding guidelines. This should also be built into your interaction with your version control software. All RTL code must pass the Lint tool with a clean bill of health in order to be checked into version control.

As discussed previously, all of the names used for ports, signal and variables, should be meaningful.

Here are some standard conventions that you should consider using as part of your signal naming convention.

"reset" or "rst": reset signals.
"clock" or "clk": clocks.
"clk125 or clock_125": 125-MHz clocks.
"rst125 or reset125": reset synchronized to the 125-MHz clock domain.
Suffix "_n": an active low signal and the negative half of a differential signal, e.g. we_n is an active low write enable.
Suffix "_p": the positive half of a differential signal.
Prefix "a": an asynchronous control signal, e.g. aclr is an asynchronous clear signal.
Prefix "s": a synchronous control signal, e.g. sload is a synchronous load signal.
"en or ena": Clock enables.
"_ack, _valid, _wait: bus flow control signals.
Use UPPERCASE: to identify parameters, enums and constants.

While constants generally minimize during synthesis, they are important for understanding the logic structure.

Bus signal rules:

Ensure that you use a uniform bus order. The most common use in industry is MSB:LSB, e.g. [63:0].

Avoid declarations that omit the LSBs, e.g. [7:3]. These increase the likelihood of structural errors in hooking up design blocks.

It is safe to omit unused MSBs, e.g. [12:0] rather than [15:0]. This has the benefit of reducing the analysis time in synthesis tools and also in reducing the number of warnings generated by the synthesis tool.

8.5.2.3 Hierarchy and Design Partitioning

Hierarchy is essential for design partitioning and should be designed for carefully. A good hierarchy is helpful for zooming in on problem areas of the design. Too many levels of hierarchy can also make a design difficult to understand. So, you need to keep the hierarchy depth modest.

A flat design is virtually impossible to understand and will cause problems in debug.

The design should be partitioned along functional boundaries. This makes it easier to see the design's behavior. When looking at the hierarchical partitioning of the design, the hierarchy of the design files should follow the spirit of block diagrams with one Verilog/VHDL module per text file. This improves the understanding of the design and will not impact the optimizations that can be applied by the EDA tools, as synthesis tools will optimize across block boundaries freely, unless you instruct them otherwise.

A benefit of doing this is that it facilitates standalone simulation of sub-designs. It also enables you to perform block performance analysis quickly.

When partitioning designs across functional boundaries you should register all inputs and outputs of the blocks. This may cost you in terms of latency in the design, however the benefits that this will bring will usually far outweigh the cost. This method of insulating the blocks can be a life saver when it comes to timing closure, as critical paths are usually contained within a single partition and can be worked on in isolation from the rest of the design (Fig. 8.7).

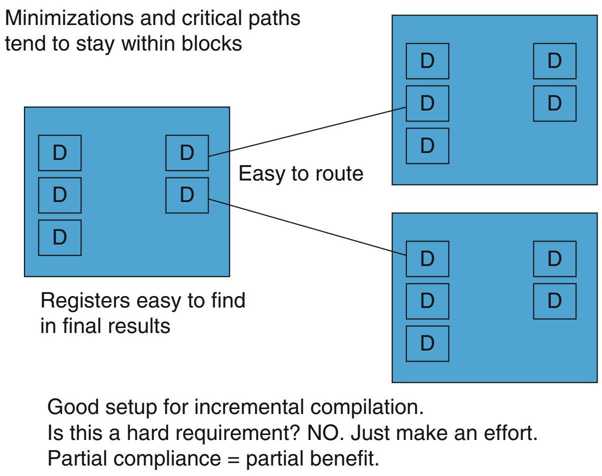

Minimizations and critical paths tend to stay within blocks
D
Easy to route
Registers easy to find in final results
Good setup for incremental compilation.
Is this a hard requirement? NO. Just make an effort.
Partial compliance = partial benefit.

Fig. 8.7 Good design partitioning

In the recent past, this extremely valuable advice was rarely 100% honored by designers, as it requires upfront planning on the design. A common mistake among designers is to design with the mindset, "I can register the ports of the block later if I need it." This statement is a vast underestimation of the effort that this will require. Any late latency changes will ripple through the rest of the design.

When partitioning the design, you must avoid inserting glue logic between partitions, as shown in Fig. 8.8.

Do not use tri-state or bi-directional ports on hierarchical boundaries unless they will always interface with device I/O pins. FPGA devices do not have internal tri-state busses. As such, the hardware versus simulation behavior is difficult to understand as the functionality will be implemented with multiplexers.

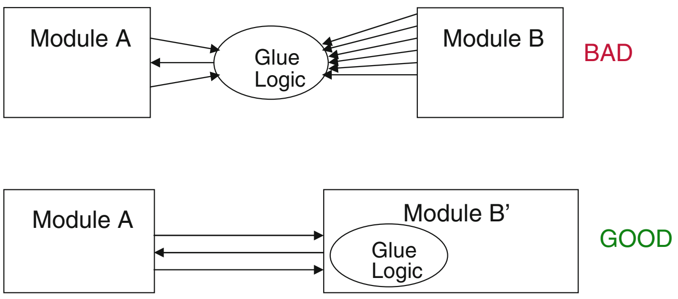


Fig. 8.8 Example bad and good partition

The recommended way to handle this is to use the approach detailed in Fig. 8.9.

```
Input  : my_bus_in [16];
Output : my_bus_out [16];
Output: my_bus_oe;
```

Fig. 8.9 Sample code for dealing with tristates at partition boundaries

Good design partitioning enables you to adopt a divide and conquer approach for building optimized design blocks.

The building blocks can be developed in parallel, potentially by different teams as shown in Fig. 8.10.

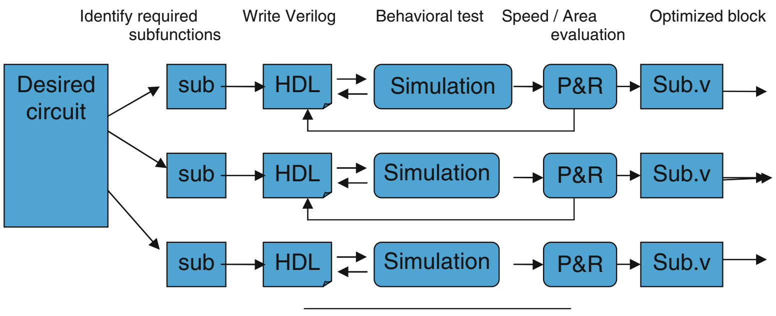


Fig. 8.10 Divide and conquer approach to RTL design

These optimized sub-blocks can be combined to form an optimized system with minimal effort (Fig. 8.11).

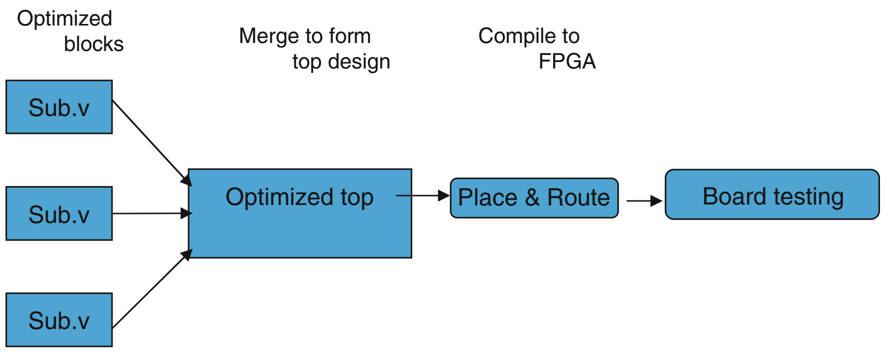


Fig. 8.11 Combine sub-blocks to create an optimized design block

8.5.2.4 Design Reuse

There is a complete chapter in this book dedicated to design reuse. In this section we will cover how the HDL coding style can impact design reuse.

Reusability will happen if the design is synchronous and reasonably partitioned for hierarchy.

It is very common for the FPGA design to be reused in its entirety in the next generation chip. This may happen for cost cutting reasons, i.e. combine multiple designs into a larger device, migration to an ASIC or for the addition of new functionality to the next generation system in a larger FPGA device.

Optimized blocks will be generally reusable but may require some changes in cases where you have used dedicated design primitives that are specific to a particular family.

So, what constitutes a good FPGA building block:

1. Something of which the purpose/functionality can be easily described.
2. It can be customized with parameters.
3. It is standalone testable.
4. It has registered IO. This provides timing closure insurance.
5. It uses a standard protocol interface.
6. The RTL code is self Documenting.
7. The number of signals on the boundary is limited. Too many signals make it difficult to interface with the design block.

What to avoid:

1. Too many levels of hierarchy in the design block.
2. The design block is too small.
3. It is difficult to interface with the design block because it requires a lot of specialized signals.

8.5.2.5 Techniques for Reducing Design Cycle Time

The RTL design cycle time can be shortened through both simulation and synthesis techniques.

Spending effort up from in functionally simulating the sub-designs will catch problems that are hard to catch when you simulate the whole design or when you are trying to debug a problem with the chip while operating on the board. It can be tedious, but it is much faster and easier to eliminate bugs at the lowest level.

There are a number of techniques that you can utilize to reduce the RTL synthesis time.

1. Perform an area evaluation. Run through the synthesis tool to get a ballpark figure of the size of the designs. Now you may be asking yourself why ballpark and not an exact area result? There are two main reasons. Firstly, when your design block is combined with the other design blocks, the synthesis tool performs a number of cross-boundary optimizations. Secondly, FPGA Place and Route tools perform a number of optimizations, e.g. packing unrelated registers with LUTs and merging of memory blocks.
2. Perform place and route on the sub-block for a performance confirmation when the sub design is almost done. If you just meet performance, you should try and build some margin in place for when the complete design is integrated. A 10% margin is good. 15% is better.
3. Try to avoid doing any hand placement or floorplanning early in the design cycle. Instead change the RTL source to meet your performance goals.

There will be times when this is not possible. When you come across one of these cases, you should detail this in the documentation for the design and make use of incremental design practices for locking down the performance of the block.

You need to try and reduce the number of design iterations that you need to run, as iteration time is expensive for large FPGA devices. In most synthesis tools, synthesis runtime is approximately linear with design size. The harder the synthesis tool has to work, the longer the synthesis time and quite likely the place and route time.

When structuring your design, you need to remember that the smaller the cones of logic the faster the design performance and synthesis time. In effect, more pipelined designs have smaller cones of logic and faster performance as well as shorter synthesis time.

If your design has deep tangled cones of logic, the synthesis tool has to try harder to traverse the logic untangling the logic cones, resulting in a longer synthesis time.

8.5.2.6 Design for Debug

This topic is covered in more detail in the chapter on In-System Debug. In this section we will cover some techniques that can be used at the RTL code level to increase the ability to debug your design in-system.

1. Register the signals that you want to see in the chip. These signals are less likely to be optimized away by synthesis.
2. Hierarchically partition the design for ease of debug. For example, if you have an interface that you are concerned about, you can place it at the edge of a device with the interface feeding I/O pins, which makes it easy to monitor.

3. Build test blocks that can easily be extracted from the end design.
4. Ensure that there are free memory and logic resources in the device to enable the use of Embedded Logic Analyzers.
5. Leave free pins on the design for access to debug signals.

8.5.3 HDL for Synthesis

Most Hardware Description Languages were originally developed for simulation and not for synthesis. As such, it is easy to describe functionality that can't be reliably implemented in hardware. You need to be aware that many synthesis tools will synthesize questionable code, which can result in an end result that may not match your simulation results. In this section, I am not going to show you examples of code that can be confusing, but rather recommend that you invest in an RTL coding training course or book. There is a standard subset of Verilog and VHDL that all synthesis tools understand and for which they will provide the same functional implementation. Study and adhere to this standard.

So, what are the guidelines?

1. Keep the hardware in mind when describing your design. What I mean by this is make sure that you can express the functionality in terms of logic gates and registers.
2. Know the limitations of the target device.
3. When your design has run through synthesis successfully, examine and eliminate the warning messages in the synthesis tool.

8.5.3.1 Coding Styles

When creating your design, should you design structurally of behaviorally?

In practice you will and should use both structural and behavioral coding styles. Old school FPGA designers will tell you that you need to use a highly structural design to guarantee the design implementation and performance. In reality, this is only true for designs that are pushing the envelope of performance and in these cases, only for a very small portion of the design; if at all.

The top-level module is invariably a collection of sub-instances, wired together with nets.

The sub-modules mostly implement core functionality with a behavioral style.

It is recommended that you describe your design using the most compact language constructs from the recommended synthesis coding guidelines. This makes it easier to understand the functionality of the design.

It is a general rule of coding that the less lines of code that you write, the less you need to debug.

You should also only instantiate basic primitives when necessary. These may be required to meet your performance requirements or to access device-specific functionality, e.g. I/O primitives, transceiver blocks, etc.

8.5.3.2 General Verilog Guidelines

We are not going to cover Verilog coding guidelines extensively but will touch on a few essential recommendations.

1. Invest in a Verilog RTL coding book or a copy of the IEEE Verilog standard.
2. Appreciate the different between non-blocking assignments (<=) and blocking assignments (=).
   Use = (blocking assignment) when modeling combination logic.
   Use <= (non-blocking assignment) in an edge-triggered always block with the following two exceptions.
   Exception 1: Assignments to temporary variables.
   Exception 2: Assignments to a RAM with write-before-read semantics.
3. Consider expression size.
   You can freely assign a 16-bit vector to an 8-bit vector.
   The context of an expression can alter the size of its operands, i.e. extend their precision.
4. Consider the expression sign.
   A single unsigned operand can coerce the sign of all the operands in a complex expression, e.g. unsigned_a + signed_b + signed_c.
5. Beware of implicit net declarations.

8.5.3.3 General VHDL Guidelines

Again, we are not going to cover VHDL coding guidelines extensively but will touch on a few essential recommendations.

1. Invest in a VHDL RTL coding book or a copy of the IEEE VHDL standard.
2. Standard Packages.
   Use rising_edge(clk) and falling_edge(clk) for edge conditions (ieee.std_logic_1164)
   Use ieee.numeric_std and ieee.numeric_bit for unsigned and signed types/operators
3. Don't use meta-values ("X", "U", "Z", "-") in case statement choices.
   The semantics of built-in VHDL "=" operator requires an exact match.
   In particular, "X" and "-" don't behave as don't cares!
4. Constrain integer subtypes with actual dynamic range, e.g. integer range 7–0.
   This reduces the hardware costs dependence on bit-width optimizations.

8.5.3.4 Designing for Performance

The main rule in achieving the fastest clock performance in a FPGA design is to pipeline your design. Remember, registers are included in the FPGA cell fabric whether you use them or not.

Select a target number for the levels of logic between the registers based upon the data sheet numbers for the LUT and register delays for the FPGA technology that you are targeting. You should aim to maintain this target in all of the sub-blocks of the design.

There are advanced settings in synthesis tools and Physical Synthesis tools that can improve performance using techniques such as register retiming. These are good at fixing a small number of long paths in the design. However, fixing this manually in the RTL, guarantees the performance, reduces the compile time and will make the design block reusable. This approach also guarantees the implementation of the design block if you upgrade to a newer version of the FPGA design software.

Figure 8.12 shows a design with two levels of logic between the registers.

Pipeline your design more than you expect. Figure 8.13 shows how an extra pipeline stage can be used to help the place and route engine meet performance. If the path shown is spread across the chip, possibly due to pin placement at both ends of the path, the "wasted" register can be used to break up the long routing delay, enabling you to meet your clock requirement.

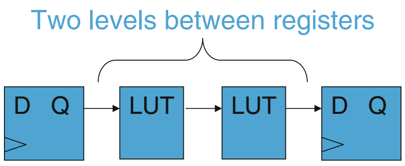


Fig. 8.12 Two-LUT levels between registers

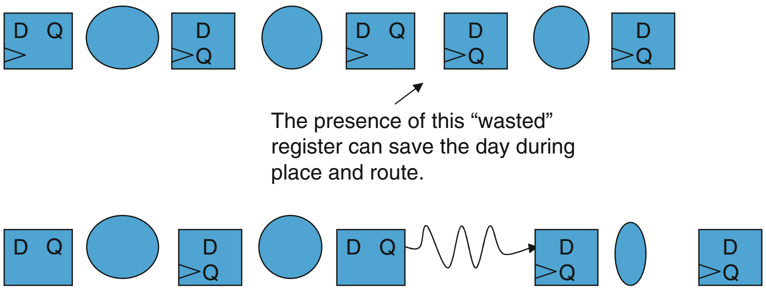

D Q
D Q
D Q
D Q
D Q
The presence of this "wasted" register can save the day during place and route.
D Q
D Q
D Q
D Q
D Q

Fig. 8.13 Use of pipeline stages to break up routing delays

Timing Margin

When designing your sub-block, you should always be looking ahead to system timing closure. Compile the sub-designs standalone and monitor the timing performance using static timing analysis. You should always build margin into the timing requirements for the sub-designs. This will allow headroom for integration with the rest of the design.

Standalone designs get first choice placement and routing. However when the overall design is integrated, not every sub-design can have first choice in a full chip. You should try and budget for a 10–15% speed degradation. It is much easier to avoid system timing problems than it is to fix them later. You do not want to put yourself in the scenario where a change to the specification late in the design cycle results in your module going from narrowly meeting timing to missing timing; making you the delay in being able to ship the product.

Do not trust estimated timing numbers from synthesis. Placement has a big impact on timing.

Sub-designs tend to be relatively small and do not take much runtime to get the true place and route timing numbers.

8.5.3.5 Designing for Area

When you are writing your RTL, think about what logic you are creating. For example, do you want one adder or two? Could you construct the RTL to get one adder?

Be familiar with the logic structure of the target architecture. What control signals are available on the registers and how is the LUT structured, 4-input LUT, 6-input LUT?

Look at the synthesis report to get a good estimate on logic used. Most synthesis tools detail the resource utilization on a hierarchical basis. This is helpful in determining if certain blocks are consuming more logic than anticipated.

For smaller design blocks, you should use netlist viewing tools to analyze the optimization, e.g. one adder versus two, and so on.

If you have very slow logic in the design, consider deploying time division multiplexing. This approach is common place in DSP designs where one FIR runs 2× or 4× required rate to save on resources.

When examining your design, look at duplicate registers and logic. These typically occur due to multiple design blocks duplicating functionality. While a small number of duplicates may be good for speed it is possible that you could achieve heavy area savings by removing the duplication. If you see possible heavy area savings, this may be an indicator of poor design hierarchy partitioning. You should consider creating a separate level of hierarchy for the common portion of the design.

8.5.3.6 Synthesis Tool Settings

All synthesis tools come complete with dozens of options for optimizing your design to meet you target goal. These settings can be very effective, however you may not be guaranteed the exact same impact in a future release of the EDA tool. By using these advanced settings, you are effectively removing the guarantee of your RTL being reusable. Despite the marketing literature on the EDA synthesis tool, it is recommended that you try to maintain the default Synthesis settings and perform your optimizations in the RTL code, ensuring that your design is reusable. If there is a setting that you have to use to meet your goals, this should be fully described in the documentation for the design block.

8.5.3.7 Inferencing of FPGA Design Blocks

RAMs

Most synthesis tools have the ability to infer basic RAMs with a single read and write operation.

A few synthesis tools can also infer true dual-port RAMs.

Synthesis tools cannot infer all of the advanced features of the RAMs in FPGA devices. These capabilities can be utilized either through the addition of attributes to your RTL or through the instantiation of RAM primitives.

When writing the RTL that describes a RAM, you need to be aware that your coding style may be such that the memory blocks require the addition of external logic to match the behavior of your HDL.

When describing RAM blocks, it is recommended that you begin with the RAM templates provided by your synthesis tool. From this, you can then create your own library of RAM modules and re-use them in every design. The philosophy behind this is that you work out all the tool/device inferencing issues in advance. This makes it easy to replace inferred RAMs with instantiated RAMs, as needed.

Avoid unsupported read-during-write behaviors. The synthesis tools will need to insert extra logic to achieve the functionality. This bypass logic will result in an increase in area and slow the performance of the design.

Read During Write Behavior

Does a simultaneous read/write to the same address returns the OLD data or the NEW data? It depends on the HDL.

Figure 8.14 details a coding style that will infer a RAM that returns the NEW data on a simultaneous read/write.

```
always@(posedge clk) begin
    if(we) ram[addr] = data; // blocking write
    q <= ram[addr]; // q reads NEW data if we == 1'b1
end
```

Fig. 8.14 New data on simultaneous read/write

Figure 8.15 details a coding style that will infer a RAM that returns the OLD data on a simultaneous read/write.

```
always@(posedge clk) begin
    if(we) ram[addr] <= data; // non-blocking write
    q <= ram[addr]; // q reads OLD data at addr
end
```

Fig. 8.15 Coding style that will infer a RAM that returns the OLD data on a simultaneous read/write

Figure 8.16 details the coding style for initializing the RAM.

Fig. 8.16 Initialize the RAM contents to all 1 s

```
-- RAM initializes to all 1's
Signal my_ram : ram_t := (others => '1');
```

```
// RAM initializes to all 1's
ram [31:0] ram[0:15];
intial begin
   for(i = 0; i < 16; i = I + 1)  ram[i] = 1;
end
```

ROMs

EDA synthesis tools can detect sets of registers and logic that can be implemented as ROMs in memory blocks.

Figure 8.17 shows how a ROM can be inferred through the use of case statements and registering of the output.

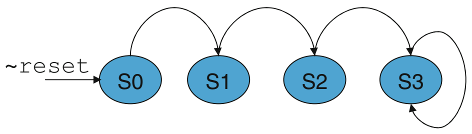
Fig. 8.17 Inferencing of a ROM

```
always @ (posedge clock)
begin
case (address)
8'b00000000: data_out = 6'b101111;
8'b00000001: data_out = 6'b110110;
...
8'b11111110: data_out = 6'b000001;
8'b11111111: data_out = 6'b101010;
endcase
end
```

Finite State Machines

When creating Finite State Machines, you should always specify your reset condition using an asynchronous condition; otherwise, the synthesis tool will guess your reset state which may cause functional issues for your design (Fig. 8.18).

Fig. 8.18 Finite state machine



In VHDL, FSMs are inferred from signals/variables which have enumerated types (Fig. 8.19).

Fig. 8.19 Use of enumerated types in VHDL for state machine inferencing

```
type state_type is (S0, S1, S2, S3);
signal my_fsm : state_type;
State names based on the enum names
```

In Verilog, FSMs are inferred from variables with the following properties.

1. Assigned values are constant expressions or module parameters.
2. Variables are not declared as an output port or used in a port connection.
3. They are referenced or assigned as a whole.
4. The state names are based on binary representation of state value or the name of the parameter that represents the state.

Figure 8.20 details an example of a Verilog FSM.

Fig. 8.20 Verilog FSM

```
localparam S0 = 0, S1 = 1, S2 = 2, S3 = 3;
reg [2:0] state_reg;

always@(posedge clk or negedge reset)
If (~reset)
    state_reg <= S0;
else
    case(state_reg)
        S0: state_reg <= S1;
        S1: state_reg <= S2;
        S2: state_reg <= S3;
        S3: state_reg <= S3;
    Endcase
```

You should always specify your reset state.
In VHDL, use STATE_TYPE'FIRST
In Verilog, state with value == 0 or the state with the smallest value.

State Machine Encoding Styles

Most FPGA synthesis tools have a default state machine style that they will use.

Fig. 8.21 State machine encoding styles

State	Binary Encoding	Grey-Code Encoding	One-Hot Encoding
Idle	000	000	00001
Fill	001	001	00010
Heat_w	010	011	00100
Wash	011	010	01000
Drain	100	110	10000

One-hot encoding is generally used for FPGA devices as the architecture features lesser fan-in per cell and an abundance of registers.

Binary (minimal bit) or grey-code encoding is generally used for CPLD or product-term devices, as these architectures feature fewer registers and greater fan-in (Fig. 8.21).

Safe State Machines

One-hot encoded state machines are commonly used in FPGAs, due to the availability of registers. However, given n encoding bits, there are $2n-n$ illegal states. Many of the synthesis tools targeting FPGAs will optimize away any manual recovery logic that you have created. They tend to have a safe machine option that can be set in the tool or controlled through the use of synthesis attributes. Make sure that you use this option as noise and spurious events in hardware can cause state machines to enter undefined states.

If state machines do not consider undefined states, it can cause mysterious "lock-ups" in hardware. It is good engineering practice to consider these states.

Large Complex State Machines

Embedded Processors are ideal for implementing large complex state machines.

Most FPGA vendors provide soft processors that can be used for this purpose with an easy to use "C" programming environment for describing the state machine operation. When using dedicated hardware to implement state machines, each additional state or state transition increases the hardware utilization. The advantage of using a soft processor is that the hardware resources consumed are fixed, with the exception of the memory resources, which depends upon the size of the state machine. A processor by definition, is a state machine that contains many states. These states can be stored in either the processor register set or the memory available to the processor; the advantage that this provides is that state machines that do not fit in the footprint of a FPGA can be implemented using memory connected to the soft processor.

The FPGA vendors provide guidelines on implementing state machines with their particular flavor of soft processor.

DSP Blocks

Most FPGA devices contain a fixed amount of dedicate hardware that is optimized for multiplication operations.

FPGA synthesis tools recognize the * operator and will infer the appropriate hardware in the FPGA silicon.

Some EDA synthesis tools have the additional capability of being able to detect multiply-accumulate operations and multiply-addition and to infer the dedicated DSP block.

In addition, some of the tools will map input/output registers into the DSP blocks to pack registers, improving performance and area utilization.

However, some of the more advanced features of the DSP blocks, such as high pipeline modes are only available via vendor primitives and these DSP blocks must be instantiated in the design.

Figure 8.22 details a Multiply-Accumulate operation that will infer the dedicated DSP block.

Fig. 8.22 Multiply-accumulate operation

```
assign multa = dataa_reg * datab_reg;
assign adder_out = multa_reg + dataout;

always @ (posedge clk or posedge aclr)
begin
        if (aclr)
        begin
                dataa_reg <= 0;
datab_reg <= 0;
multa_reg <= 0;
dataout <= 0;
        end
        else if (clken)
        begin
dataa_reg <= dataa;
datab_reg <= datab;
multa_reg <= multa;
dataout <= adder_out;
        end
end
```

Registers

FPGA synthesis tools infer registers from the same basic if-else templates.

In verilog, asynchronous conditions differentiate the clock from asynchronous controls, as shown in Fig. 8.23.

Fig. 8.23 Verilog example of a register

```
always@(posedge clk or negedge rst)
begin
    if(~rst) q <= 1'b0;
    else q <= data;
end
```

In VHDL the rising_edge() indicates the clock as shown in Fig. 8.24.

```
In VHDL the rising_edge() indicates the clock as shown in figure sss:
if(rst = '0') then
    q <= '0';
elsif(rising_edge(clk)) then
    q <= data;
end if;
```

Fig. 8.24 Register in VHDL

You must specify all asynchronous conditions first, which takes priority over synchronous conditions.

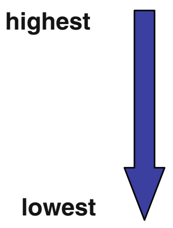
Secondary Signals for Registers

Once again, it is necessary to understand the target hardware.

In some technologies, the device registers support asynchronous clear only, only power up to ground and may not support asynchronous load.

For registers that do not support asynchronous load, it must be emulated with latches and combinational logic that is inherently prone to glitches.

The use of secondary signals also impacts place and route. Many devices have a limit in the amount of secondary resources that are available. An example being the Altera Stratix architectures where clock enable (ena), synchronous clear (sclr), synchronous load (sload) are shared by all logic cells within the same LAB. Too many unique LAB-wide signals will impact the logic utilization of the design (Fig. 8.25).



1. Asynchronous clear, (**aclr**)
2. Preset (**pre**)
3. Asynchronous load (**aload**)
4. Enable (**ena**)
5. Synchronous clear (**sclr**)
6. Synchronous load (**sload**)
7. Data in (**data**)

Fig. 8.25 Synthesis priority of secondary control signals for registers

Conditional Statements

The use of if-else statements infers 2:1 multiplexer trees with preserved priority. This coding style gives the user the control over late arriving signals, as shown in Fig. 8.26 where "a" is a late arriving signal.

```
if(cond1) then
      o <= a;
elsif(cond2) then
      o <= b;
else
      o <= c;
end if;
```

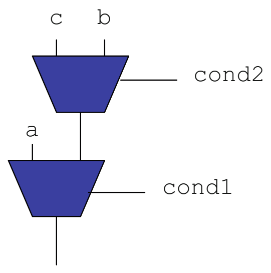


Fig. 8.26 Multiplexer tree

Care must be taken when using this style of coding for inferencing of multiplexers. Too much nesting can increase delay significantly.

It is recommended that if the conditions are mutually exclusive, to recode the multiplexer as a case statement which will infer a N:1 multiplexer (Fig. 8.27).

Case statements infer N:1 muxes.

This type of multiplexer is easier to optimize and provides much better delay than the equivalent priority multiplexer implementation.

```
case (sel)
     2'b00: o = a;
     2'b01: o = b;
     2'b10: o = c;
     2'b11: o = d;
endcase
```

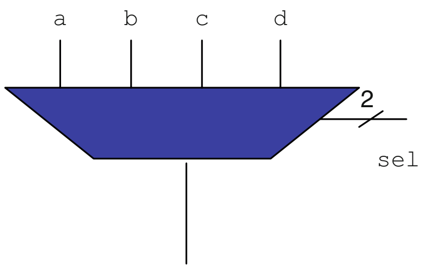


Fig. 8.27 N:1 multiplexer

8.6 Analyzing the RTL Design

All FPGA synthesis tools include a set of tools that report information on your RTL. This information can be used to check that your RTL design description is meeting your goals. They also provide the added benefit of detailing the structure of the design, thus helping in the understanding of design blocks that you have not created yourself.

8.6.1 Synthesis Reports

All synthesis tools generate a report file that details critical information about your design.

8.6.1.1 Source Files

The synthesis report will detail which source files and libraries were synthesized for the design. This is important in ensuring that you are using the intended version of source files in the design.

8.6.1.2 Synthesis Settings

This will detail which options are being used to implement the design in the synthesis tool. This information should be included in the documentation on the design as it is critical for repeatability of results.

8.6.1.3 Resource Usage Information

This is typically broken down by hierarchy. This information is useful for identifying areas of the design that consume a lot of FPGA resources. It can also help identify areas were logic has been optimized out unintentionally or implemented in a manner that is different than what you intended. An example of this would be a multiply operation that is implemented using LUTs as opposed to dedicated DSP blocks.

8.6.1.4 State Machines

Most reports will have a dedicated section that identifies all of the state machines that have been recognized in the design and will detail information on the state machine encoding. This information will identify cases were your coding style resulted in a different encoding than you intended. It will also identify cases were state machines were not recognized. This can result in non-optimal implementation and can impact the debug of your design.

8.6.1.5 Optimization Information

This section of the report contains information on optimizations that have been performed on the design. This is usually with regard to registers that have been optimized out or duplicated. In some tools it will explain why the optimization has occurred, e.g. register has no fan-out therefore optimized out, or a register has been duplicated to reduce fan-out. It also contains connectivity data such as input port to a module or input to a register is stuck at ground. This is useful for uncovering possible errors in the RTL code, in particular for the hook-up of structural code.

8.6.1.6 Timing Estimates

As mentioned previously. The timing estimates from synthesis are inaccurate and should be viewed as a coarse estimate. It is best to perform a place and route operation to get a good feel for the timing of the design or sub-design.

8.6.2 Messages

You should review all of the messages from the synthesis engine to ensure the design gets a clean bill of health.

Synthesis tools will generate a large number of messages of different levels of severity.

The code or synthesis options should be modified to remove any warning messages. If the messages cannot be avoided, you should fully understand the cause of the message and if it is verified that there is not a problem, cover it in the documentation for the module. Most synthesis tools provide the capability to review messages and to suppress them in subsequent compiles. This will greatly simplify the review process for subsequent compiles.

However, we recommend that a full message review be completed before final design sign-off.

8.6.3 Block Diagram View

Most EDA synthesis tools have schematic viewer options that can be used to analyze your design. The viewers create a schematic view of your designs and provide the ability to quickly debug your RTL design. In most cases they can cross-probe between these schematic views and HDL source code for easy tracing of signals and debug of the design implementation.

These tools are excellent for gaining an understanding into RTL code that you did not create but are reusing from another designer. It quickly shows the structure of the design and the flow of data through the design.

Figure 8.28 shows an example of such a tool from the Quartus® II software.

It is very easy to view a state machine design and determine if your description meets the desired implementation.

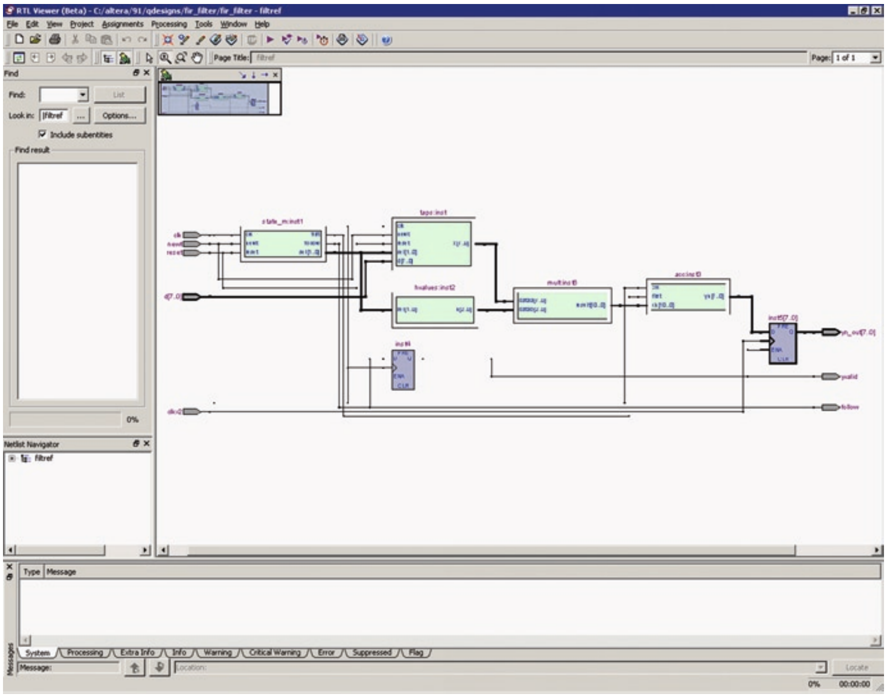
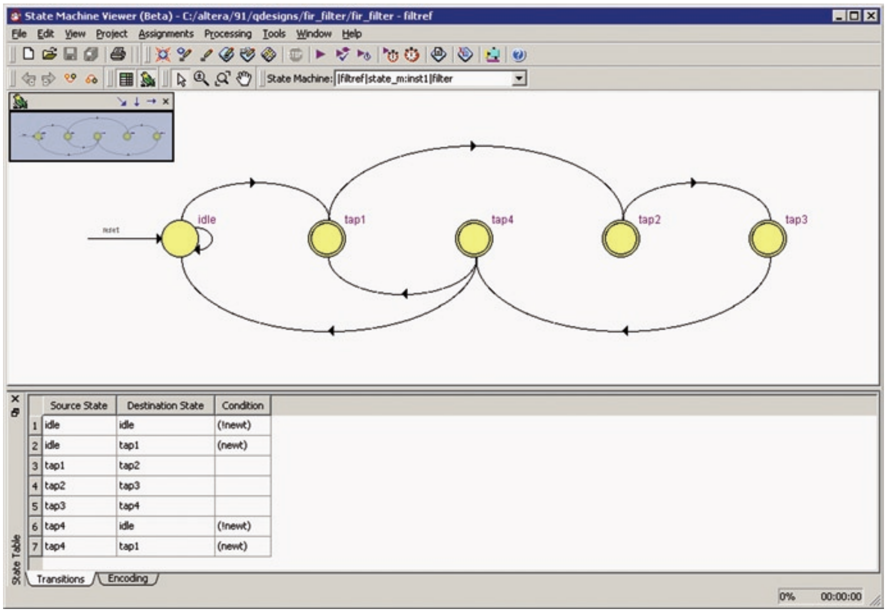


Fig. 8.28 Quartus II RTL viewer

Figure 8.29 shows an example state machine diagram created by the Quartus II software.


State Machine Viewer (Beta) - C:/altera/91/qdesigns/fir_filter/fir_filter - filtref
File Edit View Project Assignments Processing Tools Window Help
State Machine: |filtref|state_m:inst1|filter
reset
idle
tap1
tap4
tap2
tap3
State Table
Transitions
Encoding
0%
00:00:00

	Source State	Destination State	Condition
1	idle	idle	(!newt)
2	idle	tap1	(newt)
3	tap1	tap2	
4	tap2	tap3	
5	tap3	tap4	
6	tap4	idle	(!newt)
7	tap4	tap1	(newt)

Fig. 8.29 Quartus II state machine viewer

8.7 Recommended Best Practices for RTL Design

1. Choose an HDL language
2. Select the EDA synthesis tool
3. Understand the capabilities of your FPGA
4. Create a rough system design
5. Follow recommended HDL coding guidelines
6. Divide and conquer
7. Identify goals for each design block – speed, power or area
8. Run compilations with individual design blocks for area and performance estimates
9. Simulate each block
10. Document each block
11. Remove warnings from synthesis reports
12. Combine blocks to form full project
13. Simulate complete design
14. Analyze synthesis report for complete design
15. Remove warnings from complete design
16. Document complete design
17. Move onto Timing Closure for complete design

Chapter 9
IP and Design Reuse

9.1 Introduction

This main purpose of this book is to guide you in creating reusable design blocks targeting FPGA devices; from specification through RTL design and verification. This chapter on IP reuse is complementary to these other chapters. It focuses on the benefits of IP reuse, how to determine whether to design your own IP versus buying IP and how to package your IP for ease of reuse.

9.2 The Need for IP Reuse

It is universally accepted in the industry that design reuse can result in reduced engineering effort; consequently resulting in faster time to market and reduced development costs.

This is demonstrated with many projects where the next version of the product is a variation of the previous design, hence effective design reuse. In most of these cases the new product has additional functionality to the existing design and the original design is used in its entirety.

However, when it comes to completely new designs or other products that are developed by other design teams, design reuse is not so common.

In practice, design blocks from other designs could be utilized in these other designs by other teams.

So, why does this happen so infrequently?

The main reason is that most companies do not have a design reuse methodology that is adopted across development teams.

Engineers that develop design blocks are not going to drive a design reuse methodology within a corporation. They will be the adopters and contributors to a design reuse methodology.

It is the engineering management that needs to drive the design methodology from the top.

DOI 10.1007/978-1-4419-6339-0_9, © Springer Science+Business Media, LLC 2010

9.2.1 Benefits of IP Reuse

There are five main benefits to a design reuse methodology.

1. *Leverage of existing investment*. It doesn't make sense for every design team to create their own design of a function that is common across all designs. Reusing a functional block across designs makes use of the investment that was originally invested in creating the design block.
2. *Predictable results*. The performance of existing design blocks is a known entity. Through the use of existing design blocks, you are reducing the amount of your design for which the results are unknown. In the case of design blocks that are retargeted to another FPGA technology, if the design block has followed the recommendations in Chap. 8 on RTL coding, it is relatively easy to compile the design block in the new technology and quickly gauge the performance of the design block in the new technology. This is much faster than creating and verifying a new RTL design from scratch.
3. *Enables engineers to focus on their core competencies*. Some of the components of a design may not be an area for which the designer has intimate knowledge. By leveraging design blocks from experts in this area, the designer can focus on their area of expertise. An example could be a packet processing design where the data comes onto the chip via an Ethernet interface. The design engineer may be an expert in packet processing but not in developing an Ethernet interface. By reusing an existing design block that implements the 10 G Ethernet interface, the designer can focus on his core competency of implementing the packet processing functionality.
4. *Minimizes the verification cycle*. The design blocks that are being reused have previously been verified, thus they only have to be re-verified as part of full system verification.
5. *Achieve faster time-to-market*. It may take a matter of hours to add existing design blocks to your system design as opposed to the months that it may take to implement complex functionality, such as an Interlaken or DDR III memory interface.

9.2.2 Challenges in Developing a Design Reuse Methodology

Design reuse does not come for free. While the benefits in turns of cost and productivity are huge, it requires a change in mindset across the engineering teams in a corporation.

9.2.2.1 Engineers Mindset

The first challenge is winning the mindset of the engineers that develop design blocks and that will in turn become the consumers of existing design blocks. Many companies suffer from the not-invented-here (NIH) syndrome. Some engineers view the reuse of other engineers design blocks as reducing their personal value in the designs they are creating. They want to create the design themselves as opposed to using others code.

In addition, when some designers create blocks, they often want to keep the blocks to themselves as their own intellectual property. They may view the sharing of their design blocks as reducing their ownership of the design. There can also be a fear that other designers that reuse their design blocks will criticize their designs.

The largest barrier is the fact that there is extra effort involved in making design blocks reusable, some engineers are not given adequate time or do not want to expend the effort in making life easy for other engineers at cost to themselves.

These challenges can be addressed through formal development policies at the company. After the initial pain of adoption, it will become a way of life for engineers and they will take pride in creating reusable design blocks just as they do today in creating their designs.

9.2.2.2 Awareness of Reusable Design Blocks

IP distribution is a challenge. Engineers need to be aware of where to find design blocks that may benefit them. Consumers of these design blocks need to be able to find information that makes them aware of the capability of the IP, how to use the IP and how the IP has been verified. This will remove any concerns over the quality of the design.

Similarly engineers need to be aware of how to publish their IP; publishing in this context meaning how to make their IP available to other users.

IP distribution and validation can be a hurdle in the adoption of an IP-reuse methodology. Since the IP, is used by the designers who do not directly have access to original design process, they need a lot of information packaged with the IP. This includes documentation, verification plan and tests etc.

These issues can be resolved via a common managed design reuse website, wikisite or sharepoint site that is linked to version control software.

9.2.2.3 Development Effort

There is extra time and effort, hence cost in making a design block reusable as opposed to designing a block for one time use in a single project. The project schedule can be a factor in determining whether a block is developed for reuse. A company that is serious about design reuse needs to ensure that all of their project schedules allows for key design blocks to be designed for reuse. This will allow for more efficient designs in the future.

It is crucial to avoid trying to make every single piece of a design reusable.

Proper definition and selection of design blocks for reuse can be a difficult task. It is not easy to define design blocks that can successfully be used in different applications.

Thus when defining the specification of a design block, it is necessary to understand the functionality of the design block with respect to other applications and products within a company. This information can be used to determine whether the block should be created in a manner for design reuse and documented accordingly in the specification for the design block.

Certain small blocks such as address decoders and arbiters are best left to system integration tools.

Similarly, performance challenged design blocks where the functionality of the design is closely related to the timing, may not be reusable in other FPGA families or even in other devices in the same family. These blocks will have a onetime use model and need not follow all of the design reuse recommendations.

9.3 Make Versus Buy

One of the questions that an engineering manager will face is when to develop IP in-house versus when to purchase IP from a source outside of the company.

One of the influences on the decision for the in-house development of IP is whether an IP is critical to the overall performance of the design. Internally developed design blocks provide more control over design optimization and potentially customization. If this is a concern, then designers should consider designing this functionality in house or re-using design blocks from other teams, for which they have access to the source code.

Similarly, if the design block is one of the areas where you are going to differentiate your product from the competition, you will want a strong understanding of the capability and ownership of the RTL code.

Another factor that will impact in-house development versus purchasing of the IP is cost. It needs to be understood how much it would cost to develop and verify the functionality in-house versus buying a readily available solution.

Time to market may push you in the direction of purchasing IP. If your schedule is tight, purchasing IP may save you several months of development, if your existing resources are already fully occupied.

The availability of IP for your target FPGA technology is another point to be considered. There is usually a delay from the availability of new FPGA families to the porting of IP to these new families. Many of the smaller FPGA vendors will wait for a lead customer prior to performing the port. This can cause a delay in the availability of IP that has tight timing requirements. The risk in being the first adopter of new IP is that you may become the cleaning house on the IP verification in the new technology. This can also be a benefit in that if you are the first to adopt the IP in a leading edge technology, you may gain a lead on your competition.

Anytime that you are receiving design blocks from another source, there will be concerns over the quality the design blocks, in particular if you are purchasing the IP.

There is no industry standard for IP quality that is available to help in the selection of IP. Several initiatives have started in the past, but never reached the level of industry approval and adoption. Consequently, you need to rely on IP provider's reputation or ask for details on the IP provider's verification process and results for the IP that is being purchased.

These are all cases were you can compare the costs of internal development of design blocks versus purchasing of design blocks.

If your design team does not have the knowledge or experience in the area of functionality that you need, it should be a slam dunk to use purchased IP.

9.4 Architecting Reusable IP

9.4.1 Specification

The overall system specification should identify new blocks that are being developed that could be used in other designs. This will impact the schedule and specification for the development of these blocks.

Thus when these blocks are being defined it will be in their requirements that they should be developed for reuse and should follow the IP reuse guidelines.

When the specifications for these reusable blocks are being reviewed, it should include reviewers from the other teams that could be consumers of the IP. This will serve three main purposes. Firstly it will increase the awareness of the IP across teams. Secondly, by involving the other teams in the specification process they will have a vested interest in the IP and will be more open to adopting the blocks in their design. Finally, these other teams may provide feedback that your team may have overlooked.

9.4.2 Implementation Methods

9.4.2.1 Parameterized RTL

Developing IP using parameterized RTL is the most common IP development methodology in the industry. It provides the simplest way to create and maintain reusable design blocks. Some examples would be the use of parameters to set different data widths for memory or FIFOs.

Parameterization provides built-in flexibility through the use of non-constant variables; these are parameters in Verilog and generics in VHDL.

When you are determining what should be parameterized in an IP you should consider the likely uses of the core, anticipate the range of desired features and build parameterized functionality for each desired configuration.

Generate statements which are available both in Verilog and VHDL should be used together with parameters in reusable IP to achieve efficient implementation of

```
module bus_mux (din,sel,dout);

parameter DAT_WIDTH = 16;
parameter SEL_WIDTH = 3;
parameter TOTAL_DAT = DAT_WIDTH << SEL_WIDTH;
parameter NUM_WORDS = (1 << SEL_WIDTH);

input [TOTAL_DAT-1 : 0] din;
input [SEL_WIDTH-1:0] sel;
output [DAT_WIDTH-1:0] dout;

genvar i,k;
generate
        for (k=0;k<DAT_WIDTH;k=k+1)
        begin : out
                wire [NUM_WORDS-1:0] tmp;
                for (i=0;i<NUM_WORDS;i=i+1)
                begin : mx
                        assign tmp [i] = din[k+i*DAT_WIDTH];
                end
                assign dout[k] = tmp[sel];
        end
endgenerate
endmodule
```

Fig. 9.1 Example detailing the use of parameters in a Verilog source file

the design. Generated instantiations and module parameters can be used to remove redundant logic and create flexible designs.

Generate loops allows multiple statements and blocks to be instantiated using ‘for’ loops.

Generate based upon conditions can be used to create parameterized logic. An example showing the use of a generate statement with parameters to generate a bus multiplexer is shown in Fig. 9.1.

More detailed guidelines on creating RTL for IP reuse are available in Chap. 8 on RTL design.

Section 8.5.2.3 of Chap. 8 provides guidelines on hierarchy and design partitioning. Section 8.5.2.4 provides coding guidelines for design reuse.

9.4.2.2 High Level Synthesis

High level synthesis is good for algorithmic exploration; particularly in the DSP space where users enter their design in Ansi C/C++. This class of tools has been shown to provide a large development time reduction over designing algorithms in RTL and opens the hardware design process to a new class of user; the software or system engineer. They are excellent for the architecture exploration phase of the algorithm design as the description is much closer, or the same as the algorithm model. The amount of ‘C’ code needed to describe the functionality is likely to be much smaller than an RTL implementation; hence the gain in productivity. These tools also tend to provide more flexibility in porting the design across FPGA families.

At the highest level of design, the code is not created with a target FPGA family in mind.

Their main disadvantage in these solutions is that they tend not to be an optimal solution for fine tuned optimized Quality of Results; thus can be area inefficient or leave some performance on the table. In recent years, these tools have made good progress in the QoR aspect for certain classes of DSP applications. They should be considered for the creation of non-performance critical DSP IP.

In addition to C/C++ tools there is also another class of design tools which is model based design. These tools provide an interface to the MATLAB environment via Simulink. Once again, these tools mostly target DSP applications. They have been shown to be used successfully in a smaller application space; mostly in modem designs and some military applications. This class of tools should be considered for creating IP in these application spaces.

9.4.2.3 IP Generator

IP generators are programs that are written in C++, Perl, or other high-level languages that build RTL code dynamically, based on parameter settings from the end user. The generators tend to pull together RTL design blocks based upon the chosen parameters.

This technique is commonly used by FPGA vendors to provide complex IP to their customer base.

An IP Generator generates the HDL code based on the customer specification with all of the parameters resolved.

They are suitable for complex parameter combinations, complex legality checking and advanced processing for arithmetic operations.

The disadvantage of IP generators is that they require software programming skills to implement.

9.4.3 Use of Standard Interfaces

It is recommended that you adopt a common interface standard on all of your IP. The use of standard interfaces simplifies the interconnection and management of the functional blocks that makes up a design.

1. It ensures compatibility between IP components from different design teams or vendors.
2. It enables fast system level integration of IP. Consumers of the IP are aware of the operation of the signals to which they are interfacing; greatly simplifying the interface logic to the design block.
3. It also opens the door to using design automation tools for system integration.
4. This simplifies team based design, by enabling individual team members to build and test their individual design blocks. Through the understanding of the common interface protocol, each of the team members will understand how to interface to the

blocks that use the common specification. This simplifies the integration of the individual design blocks into a full system design.
5. It enables plug and play interoperability of IP.
6. It also increases the stability of the IP. The operation of the interface signals are described in the specification for the interface protocol and the operation of the interface signals on the core verified against the specification.

There are various standard interfaces on the market today. The most widely adopted interface standards that are used in FPGA and ASIC design are AMBA (AXI, AHB and APB) from ARM, Avalon (MM and ST) from Altera, OCP from OCP–IP and Wishbone from Opencores.

When selecting a standard interface protocol you need to ensure that the IP infrastructure is in place. When we refer to IP infrastructure we mean that IP is available targeting the FPGA technology that you will be targeting using the standard interface protocol and that the specification for the protocol is solid. IP includes both the IP that will be part of your end design and verification IP such as Bus Functional Models.

The interface standard needs to be easy to understand, compact, and the hardware interfaces should not produce performance or area penalties when implemented. The standard needs to support all of your application needs. This will normally include memory mapped interfaces with address-based read/write interfaces typical of master–slave connections, point-to-point interfaces that support the unidirectional flow of data, including multiplexed streams, packets, and DSP data.

In summary the use of a standard interface protocol really is the heart of a design reuse strategy.

9.5 Packaging of IP

The IP package is the IP core plus the supporting files and utilities.

A good IP package should place everything at the user's fingertips. It should be easy to find, install and to maintain.

User access to the IP could be in a company library of reusable IP or it could require installation on the user's workstation or design environment. If it requires installation, it is recommended that you leverage an off the shelf commercial product to perform the installation, such as install shield, or create a self extracting executable using WinZip or a similar program.

The minimum requirements for an IP package are:

1. IP core. The design that implements the required functionality.
2. Timing constraints and any location constraints.
3. Simulation model.
4. User documentation. This should be the user manual for the IP as well as any errata. This is described in more detail in Sect. 9.5.1.
5. User interface.
6. Compatibility with any system integration tools that you intend using.

9.5.1 Documentation

As mentioned previously the documentation on the IP should include the user manual and any errata. It should include version control on the documentation that details the history of changes to the IP core and documentation. The version of the core needs to be identifiable in the core itself, as well as in the documentation.

While the functionality of the design may be unique to the IP core, the format of the documentation needs to be consistent across all IP cores. This includes the user documentation and the RTL code formatting which in itself should be self documenting.

The documentation should include an example design or testbench for the IP that demonstrates how to connect the IP to the rest of a design. Ideally this can be used to demonstrate the functionality of the IP.

The file structure of the design must be common with all other IP and the naming convention of signals must follow the company coding guidelines.

For parameterized IP, there should be tips on the parameter settings.

9.5.2 User Interface

The most common way that designers make IP available to other designers within their company is that they provide the RTL for the design along with user documentation on the design. While this works, it makes it difficult for the end user to really understand how to use the IP that they are receiving.

IP should come with an interface that makes it easy for the user to understand the constraints that apply to the IP. At a minimum the IP should come with a documented command-line script that enables users to pass values to the parameters in the IP. Ideally it should come with a GUI to help users get started.

Our recommendation is that you provide a simple GUI for your IP and a scripting interface.

The simple GUI should enable users to set parameters, set constraints and be able to validate that the selections are legal.

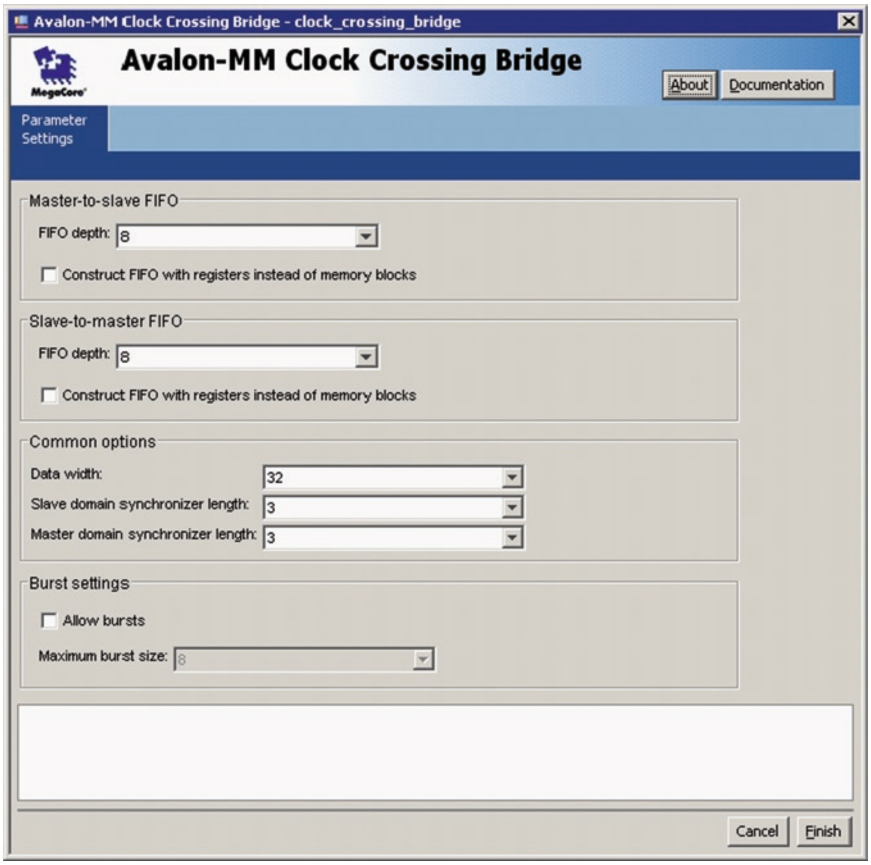
This type of interface will help designers to learn the functionality of the IP, generate the correct verification files and scripts for the block, as well as providing a link to documentation that is available for the IP.

This is the type of interface that you will see in the IP that is provided by the FPGA vendors and in many cases from other IP providers.

The GUI need not be elaborate; it needs to show the user what settings that they can make and enable them to make the settings.

A sample GUI available in the Component Editor from Altera is shown in Fig. 9.2.

If you have reasonable programming skills, you could create a GUI in Tcl/TK or in Java.


Avalon-MM Clock Crossing Bridge - clock_crossing_bridge
Avalon-MM Clock Crossing Bridge
MegaCore
About
Documentation
Parameter Settings
Cancel
Finish

Master-to-slave FIFO
FIFO depth: 8
☐ Construct FIFO with registers instead of memory blocks

Slave-to-master FIFO
FIFO depth: 8
☐ Construct FIFO with registers instead of memory blocks

Common options
Data width: 32
Slave domain synchronizer length: 3
Master domain synchronizer length: 3

Burst settings
☐ Allow bursts
Maximum burst size: 8

Fig. 9.2 Sample GUI for IP demonstrated by the Quartus II Component Editor

If not, you can adopt the IP GUIs from the FPGA vendors. This requires the adoption of the FPGA development tools.

9.5.3 *Compatibility with System Integration Tools*

Standardized design entry and design integration tools can reduce the design entry overhead.

System integration tools auto-generate the HDL for the interconnection of IP blocks. The major FPGA vendor tools provide IP integration tools that perform this function. These system integration tools take care of the relatively mundane tasks that RTL designers have to do such as address decoding, data multiplexing, wait state generation in processor systems, dynamic bus sizing, slave side arbitration and direct interconnect of blocks. This functionality is analogous to a software linker.

A software linker creates an executable program out of MAIN and a selection of precompiled library functions.

System integration tools, such as SOPC builder from Altera, automatically create a system out of a variety of system blocks. This enables designers to focus on value-add architecture ideas, effectively extracting themselves from the low level integration details.

These tools should be used in both the architecture exploration and implementation phases of the design process, where they will increase your productivity. They facilitate architecture exploration by allowing you to plug and play design blocks into your system and to quickly generate the RTL for the given architecture without having to modify the arbitration logic, width adaption logic, memory map, etc. manually. This enables you to quickly try different architecture variations. Once you find the architecture that you want to use for the implementation you can then fine tune the blocks that are in the system to meet your overall goals.

9.5.4 IP Security

The IP that you purchase from IP vendors normally arrive encrypted. The IP vendors do this to preserve the integrity of their RTL and to prevent non-authorized users from being able to design with their IP. The encryption scheme that is used tends to vary across IP vendors and EDA vendors. From the perspective of a consumer of IP, you care about which synthesis tools support the IP and the quality of the simulation model from the IP vendor.

There are moves in the industry to provide a standard encryption methodology. The IEEE has created the IEEE 1499 standard based upon the Open Mode Interface (OMI). The standard enables the RTL to be compiled into a model format that cannot be reversed engineered. These models can be simulated in OMI-compliant simulators. The benefit is that the RTL code for simulation model and synthesis is the same. This reduces the development effort for the IP vendor.

Some IP vendors will provide the source code for the IP. This simplifies the design flow but usually costs significantly more than the encrypted RTL.

If you intend to provide encrypted IP, you must work with your FPGA vendor to utilize their encryption tools.

Some IP vendors provide obfuscated RTL. This provides a limited form of security in that the code is difficult to understand as the signal names appear to be nonsensical. Obfuscation makes it difficult for non-authorized users to reverse engineer the RTL. It does not prevent them from compiling the design.

Some of the FPGA vendors enable you to provide the IP in a post-compilation format as opposed to at the RTL level. An example being a design block that has been compiled using an incremental compilation methodology with the placement and routing locked down. This level of IP guarantees the performance of the IP, thus reducing the support burden on the IP, but restricting its use to particular device.

These are some of the ways that you can provide IP to other users. Most corporations provide the RTL for design reuse within their own corporation and encryption only comes into play on purchased IP. However, some corporations are deploying encryption schemes internally for the distribution of key IP blocks.

Due to the complication of the design flow, it is recommended that you only use encryption or obfuscation on your design blocks if security is a major concern.

9.6 IP Reuse Checklist

1. Purchase or design the functionality?
2. Does the specification state that the design be reusable?
3. Select the appropriate IP implementation method, i.e. RTL, high-level synthesis or generator?
4. For RTL solutions, follow the RTL coding guidelines.
5. For RTL solutions, parameterize the IP.
6. Use standard interfaces on the design block.
7. Is encryption or obfuscation required?
8. Does the IP follow the IP packaging guidelines?

Chapter 10
The Hardware to Software Interface

10.1 Software Interface

The main interface between the application software and the RTL is the Register Address Map. The register address map is shared across multiple disciplines in the design process.

This creates the challenge in the project of synchronizing the firmware, RTL, hardware verification, and the documentation. In the case of documentation this refers to both internal use and in the case of IP development, the documentation that is provided to the end user.

As such, it is essential that the information is strictly controlled and any change in the information is communicated across the design team, with changes being avoided as much as possible to avoid a firmware and/or hardware rewrite.

10.2 Definition of Register Address Map

The register address map is often referred to by many different names including Control and Status Registers (CSRs), Memory Mapped registers, Register File, Register Block, or Register Interface. Registers in the design are used to represent data that is communicated between the hardware and the software. Each block of IP provides a register interface that is mapped to addresses for the software interface. This register address map creates a view of the hardware/software interface for software programmers to read from or write to. Effectively, communicating between the software and the hardware.

10.3 Use of the Register Address Map

As mentioned at the start of the chapter, the Register Address Map is used by different disciplines throughout the design process. Each of the different disciplines will likely require the data in a slightly different format.

DOI 10.1007/978-1-4419-6339-0_10, © Springer Science+Business Media, LLC 2010

10.3.1 IP Selection

As part of your selection criteria for IP, you need to understand how you will interface to the IP from both the hardware and the software perspective. The Register Address Map will address how your software will interface with the IP. The user documentation on the IP core should reflect this information.

10.3.2 Software Engineers Interface

The software engineer needs to know the register map in order to develop the software drivers that interface with the hardware. The software engineer will want the register map information in the form of software header files which define the component base address and register offsets (Fig. 10.1).

10.3.3 RTL Engineers Interface

The RTL Engineer needs to connect the Register Map interface to the rest of the system. This involves writing the logic for each of the register bits and creating address decoders for read/write cycles. The challenge to the RTL design is defining

```
#ifndef __ALT_ETH_10G_REGS_H__
#define __ALT_ETH_10G_REGS_H__

#include "alt_types. h"

/* Revision register */
#define ALT_ETH_10G_REV_REG                          0x00
#define IOADDR_ALT_ETH_10G_REV(base)                 __IO_CALC_ADDRESS_NATIVE(base, ALT_ETH_10G_REV_REG)
#define IORD_ALT_ETH_10G_REV(base)                   IORD_32DIRECT (base, ALT_ETH_10G_REV_REG)

#define ALT_ETH_10G_REV_CORE_REVISION_OFST           (0)
#define ALT_ETH_10G_REV_CORE_REVISION_MSK            (0x0000FFFF)
#define ALT_ETH_10G_REV_USER_REVISION_OFST           (16)
#define ALT_ETH_10G_REV_USER_REVISION_MSK            (0xFFFF0000)

/* Scratch register */
#define ALT_ETH_10G_SCRATCH_REG                      0x04
#define IOADDR_ALT_ETH_10G_SCRATCH(base)             __IO_CALC_ADDRESS_NATIVE(base, ALT_ETH_10G_SCRATCH_REG)
#define IORD_ALT_ETH_10G_SCRATCH(base)                IORD_32DIRECT(base, ALT_ETH_10G_SCRATCH_REG)
#define IOWR_ALT_ETH_10G_SCRATCH(base, data)         IOWR_32DIRECT(base, ALT_ETH_10G_SCRATCH_REG, data)

/* Command register */
#define ALT_ETH_10G_CMD_REG                          0x08
#define IOADDR_ALT_ETH_10G_CMD(base)                 __IO_CALC_ADDRESS_NATIVE(base, ALT_ETH_10G_CMD_REG)
#define IORD_ALT_ETH_10G_CMD(base)                   IORD_32DIRECT(base, ALT_ETH_10G_CMD_REG)
#define IOWR_ALT_ETH_10G_CMD(base, data)             IOWR_32DIRECT(base, ALT_ETH_10G_COMMAND_CONFIG_REG, data)

#define ALT_ETH_10G_CMD_TX_ENA_OFST                  (0)
#define ALT_ETH_10G_CMD_TX_ENA_MSK                   (0x00000001)
#define ALT_ETH_10G_CMD_RX_ENA_OFST                  (1)
#define ALT_ETH_10G_CMD_RX_ENA_MSK                   (0x00000002)
#define ALT_ETH_10G_CMD_XON_GEN_OFST                 (2)
#define ALT_ETH_10G_CMD_XON_GEN_MSK                  (0x00000004)
```

Fig. 10.1 Sample from Header file generated by the Altera SOPC Builder tool

this up front and maintaining the register map throughout the design cycle. It is likely that at sometime in the design cycle that the RTL designers will need to change some part of the Register Address Map. The whole process of coding, documenting, reviewing and communicating the Register Address Map is an error prone task that many RTL designers prefer to avoid.

Fortunately there are several tools on the market that help with this task. The System Integration tools from the FPGA vendors provide an automated interface between the Hardware System Design and the Software Engineer, by automatically generating software header files. In addition they take care of the generation of the logic for the address decoding.

There are EDA tools that provide much more advanced capability. These tools can create the synthesizable RTL for the Register Address Map from register descriptions, generate the software header files, header files for verification and also create user documentation in various formats.

10.3.4 Verification Interface

It is good engineering practice to develop testbenches that verify the operation of the RTL Register Address Map. As such the verification engineer needs the Register Address Map details in a format that can be used with the verification language that is being used.

As part of the verification cycle, you will want to validate that the software can read and write to the Register Address Map as detailed in the specification. This can be tested on the device with the register map document being used as a functional checklist.

10.3.5 Documentation

As mentioned at the start of this chapter, documentation refers to both internal documentation for use among the design team and the documentation that is provided to the end users of IP.

Whenever there are changes to the RTL for the Register Address Map, it is the designer's responsibility to update the documentation and to review the changes with all of the teams that may be impacted by the change.

The format used to describe the Register Address Map must be consistent in terms of the naming convention that is used among all designers. This achieved by having a process for creating the Register Address Map specification which specifies how it should be documented.

There is a standard format that exists in the industry for specifying the Register Address Map for IP. This is the IP-XACT standard which uses XML metadata that can be read by several EDA tools on the market. However, at the time of writing, this standard has not been widely adopted by all IP vendors and EDA tools.

It is recommended that you review the standard prior to beginning your project as you may want to consider adopting this standard as opposed to developing your own format.

10.4 Summary

The Register Address Map Interface is the main interface between the Software Engineer and the RTL Engineer. This information is used by several different functions in the design process, all of which need access to the same information in different formats to fit in with their function. As such this information needs to be strictly controlled and any changes reviewed with the teams that need this information. Due to the fact that it is time consuming and error prone to manually update all of the file formats that use this information, it is recommended that you invest in an EDA tool that specializes in Register Address map Management.

Chapter 11
Functional Verification

11.1 Introduction

There are two simple questions that every design team needs to be able to answer. Does my design function properly and is my design verification complete?

These two simple questions are likely to take more than 60% of your design cycle to achieve acceptable answers. Just defining what is meant by functioning properly and what is deemed acceptable as complete are difficult tasks.

In the past, when FPGA designs were small and many designer were not concerned with the concept of design reuse, FPGA designers deployed the "blow and go" approach to FPGA design verification. They would create the design, perform a cursory functional simulation on the RTL, then program the FPGA and test the design in system. If they found a problem, they would fix the code and repeat. This approach is not practical for large, complex, high quality system designs.

The programmable nature of FPGAs does add a powerful weapon to the design verification armory. However, when used by itself, it is not a method for creating reliable and reusable designs.

There are many publications and EDA tool solutions dedicated to the topic of functional verification.

There are also many different verification techniques that can be used to verify that a design meets the requirements that are dictated in the specification. Many of the techniques that are used in the verification of ASICs are applicable to the verification of FPGA designs. As mentioned, the programmable capabilities of FPGAs provide some additional capability that can be used in the verification of designs that are targeting FPGA devices. This chapter will describe the techniques that are known to work well in functionally verifying FPGA designs and IP targeting FPGA devices.

11.2 Challenges of Functional Verification

At a high level, the goal of functional verification is to verify that the design functions as specified. This applies to the complete design as well as any of the sub-designs.

Functional verification of the design must cover all modes of operation of the design. This includes corner cases. The last thing that you want is that when your design is deployed in a product, that your system enters a mode of operation that you have not considered or tested against, resulting in a catastrophic failure.

The application interface to your design needs to operate as expected, i.e. testing needs to emulate the interaction of your design with the rest of the system.

In the scenario where your FPGA device interfaces to the rest of the system via standard protocol interfaces, such as PCI Express or Serial Rapid I/O, it is necessary to verify that the interface block complies with the appropriate standard.

In the case of parameterized IP, it is necessary to test all architectural variations of the design based upon the parameterization. This will provide confidence to consumers of the IP that the IP meets their requirements.

In the case that the IP has been packaged for reuse and there is a user interface to the IP, it must be possible to verify that the user interface operates as intended and on all supported operation systems.

Finally, you need to verify that the documentation on the design or IP block is clear and matches the behavior of the core.

This may sound like a lot of work…and it is!

The challenges that you face include how do you achieve adequate verification coverage in the given schedule with the resources that are available?

How do you determine what is an acceptable level of coverage?

The answer to these questions will come from the verification plan. The verification plan must detail the coverage goals and other completion metrics. As such, this has an impact on the project plan.

You need to plan the verification of the design at the same time that you are developing the functional specification of the design.

There needs to be a system in place that enables you to monitor the progress against the verification plan throughout the design and verification cycle. This system must be capable of managing the large amount of data that you will receive from the testing and report the progress against the verification plan.

11.3 Glossary of Verification Concepts

1. Device Under Test (DUT): This is the IP being tested.
2. Assertions (coverage points). These describe the behavior of the design that is true when the design is behaving correctly. Assertions are also activated when the design behaves incorrectly. It effectively covers the state of the DUT.
3. VMM. Synopsys Verification Methodology Manual: It details a methodology based around SystemVerilog for verifying complex designs.
4. Testbench: A test bench is an environment that is used to exercise and verify the correctness of the design.
5. Transactors: In a testbench environment, the transactor is a model that defines the sequence of events or tasks to be performed.

6. Scoreboards: The scoreboard is a data structure that holds the expected results from an operation for comparison against the actual results achieved.
7. Register Abstraction Layer (RAL): The VMM Register Abstraction Layer (RAL) automates the creation of the high-level abstraction layer for memory-mapped registers and memories. The VMM specification provides more detail on RAL.
8. Executable specification: An executable specification is a high level model that describes the functionality of the design, hardware and/or software. It is usually written in a high level language such as C, C++, SystemC or SystemVerilog.
9. Regression Tests: Regression tests are a set of tests that are run on the application after every design change and on a regular basis, such as every night or every weekend, in order to ensure that no new bugs have been introduced. It is an automated environment that proves that the design operates to the specification.
10. OVM (Open Verification Methodology): OVM is a standard SystemVerilog library and verification methodology developed by Cadence and mentor Graphics.

11.4 RTL Versus Gate Level Simulation

Simulating at the RTL level performs functional verification without consideration for the timing delays that will occur when the design is implemented. It is common practice to perform RTL simulations to prove the functionality of the design and timing analysis to prove that there is no timing violations in the design.

Gate level simulation utilizes the timing netlist generated after place and route. This contains the device timing delays in the Standard Delay Format (SDF). This provides a more accurate view of how the design will function on-chip as it includes timing information. Timing simulations take considerably longer to run than RTL simulations. In fact they are considered by many designers as prohibitively long for certain application types such as video and image processing applications and for large designs. As such it is recommended that timing simulations should only be performed on critical sub-designs instead of the full design, or when debugging problems that are found during hardware checkout of the system.

11.5 Verification Methodology

In order to achieve success in verifying your design, you must deploy a variety of techniques.

You should use a combination of functional coverage and code coverage techniques.

These are complementary to each other.

In the case of certain protocols, you should also perform hardware interoperability testing.

Finally, let’s not forget that the target devices are programmable. Implement parts of the design in hardware to find those hard to reach bugs that may take days or weeks of simulation to uncover. In-system debug techniques are described in more detail in the Chap. 13.

The verification methodology should use the following steps.

11.6 Attack Complexity

There are three main rules for helping to deal with the complexity of testing your design.

11.6.1 Modularize Your Design and Your Tests

It is extremely unlikely that you will be able to test all of the functionality of your design with a single test. As such you should have different tests for testing different aspects of the design. In addition to providing a more thorough verification environment, this approach will make it easier to transfer the testing to other persons as the tests will be easier to understand.

For large design blocks you should adopt a functional verification methodology that breaks the design into smaller sub-designs, as described in the Chap. 8 and thoroughly verify each sub-design prior to verifying the complete design.

11.6.2 Plan for Expected Operation

Create tests to confirm that the design will work in the planned or normal mode of operation. You should exercise the design under all of the operational modes under the various normal conditions. These tests must cover all of the features listed in the functional description and specification.

Exercise the corner cases and confirm that they operate as defined.

As part of the functional tests, ensure that you exercise every register bit and every signal on every port.

When verifying designs with multiple modules that can be user parameterized, you need to exercise all possible combinations of the modes to verify the interactions between the adjacent modules.

After each operation, verify that the system returns to the correct state.

11.6.3 Plan for the Unexpected

The last thing that you want is that your system enters an unrecoverable state based upon system conditions that you had not tested. As such, you must test exception conditions. These exception conditions will vary from application to application. Examples of such conditions are overflows, underflows, CRC errors, aborts. As part of testing unexpected conditions you should test the functionality in these unplanned conditions and then exercise recovery from the exception conditions. Exceptions aren't necessarily errors; they can be outlier conditions that are unlikely to occur in practice. The key thing is that your system can recover from them.

This testing should test conditions that cannot happen

1. Test illegal conditions
2. Violate design assumptions
3. Violate protocols
4. Change modes in mid-operation

Once again, the key factor is that while the design may behave incorrectly, it should recover eventually.

As part of the functional verification of IP or design blocks, you should test the interaction with other cores in the overall design to ensure that the interfaces operate as expected.

11.7 Functional Coverage

Compliance and corner case testing, as described in the Sect. 9.6, attack complexity, is good but on its own it is not sufficient to fully test your system. It is unlikely that you will be able to predict and exercise all possible conditions. This increases the risk of failure in system. Functional coverage increases the confidence in the verification of your design block or system. It is the determination of how much functionality of the design has been exercised by the verification environment. Each test is created to check the particular functionality of a specification. The key point is that you need to be able to prove that the test executed the functionality that it is supposed to check.

The test plan for your design block and for the overall system should specify the metrics for verification coverage. That is the functional coverage goals for the design.

The challenges that you face when planning for functional coverage are ensuring that the design implements the formal Functional Description and in the case of interfaces, conforms to standard protocol specifications.

Your goal is to ensure that it satisfies formal Functional Test Plan and matches the behavior established by a suitable golden reference model.

In the case of reusable design blocks, you want to ensure that the coverage items capture

1. All features and capabilities of the Device under test
2. All configuration variants
3. Types of stimulus applied
4. The response of DUT

Functional coverage does have limitations in that it is difficult to define a list that proves 100% functionality of the design. Thus it is important to identify the coverage holes in the coverage space.

11.7.1 Directed Testing

Directed testing requires hand crafted test case for each test plan item. Thus the number of tests required to achieve acceptable coverage is enormous. The tests themselves tend not to be easily reusable.

It is best used to test typical behavior due to the time it takes to perform the simulations.

It is recommended that directed testing be used for reasonably small blocks. For much larger blocks and at the system level, you will need to adopt constrained random techniques.

11.7.2 Random Dynamic Simulation

In this verification methodology, random stimulus is used to increase the functional coverage. This method of verification is best performed using a high level verification language. Over the years, many languages and tools have been developed to serve this purpose. SystemVerilog has emerged as the leader in this space. SystemVerilog has been ratified as a standard by the IEEE and provides the broadest tool support among verification languages.

It is recommended that you should consider adopting SystemVerilog for the verification of your system.

11.7.3 Constrained Random Tests

Constrained random testing is built on top of random dynamic simulation. Random simulations are best run in the early stages of the design to catch a lot of bugs. Then as the design nears completion, the random simulations are constrained to fully cover the test space.

A single test run can cover many items in the test plan, resulting in less simulation time.

This approach can also detect problems/bugs that are not part of test plan (Fig. 11.1).

11.7.4 Use of System Verilog for Design and Verification

SystemVerilog is really three languages in one.

1. It contains design constructs that are more powerful than Verilog and VHDL for design and synthesis.

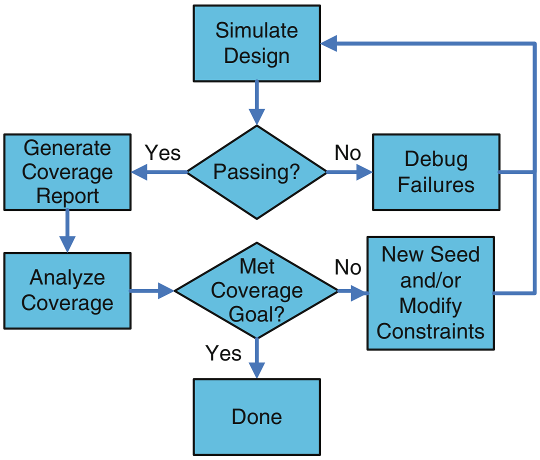


Fig. 11.1 Constrained random test flow

2. It has advanced testbench constructs for stimulus and coverage.
3. It supports assertion constructs to capture the designer intent.

SystemVerilog has built-in support for coverage-driven constrained-random verification.

It has options for pre-verified libraries of assertions with the major EDA simulators on the market.

At this time, the industry is split on the SystemVerilog verification methodology. The two main libraries are VMM (Verification Methodology Manual and OVM (Open Verification Methodology). There is a push to standardize on a single library.

11.7.4.1 Assertions

Assertions are used to check assumptions made by designers and the behavior associated with a design. They are triggered during a dynamic simulation if the design meets or fails the specification. They can be used at both the module and the system level.

They also provide the benefit that they are reusable with reusable design blocks.

Assertions provide early visibility into problems such as FIFO overflow/underflow errors. They also capture inter-block communication such as memory interface behavior.

11.7.5 General Testbench Methods

The simplest testbenches to write do not involve the creation of verification code. It requires that the engineer manually verifies that the design passes. This is normally achieved by viewing the resulting waveforms. One of the challenges with this approach is that while the designer who fully understands the design can understand the waverforms, a different engineer may miss errors or take much longer to understand the results.

This approach is best applied to simple design blocks that are not intended for re-use.

The designer creates the "test harness" code to instantiate the design code and creates stimulus signals (Fig. 11.2).

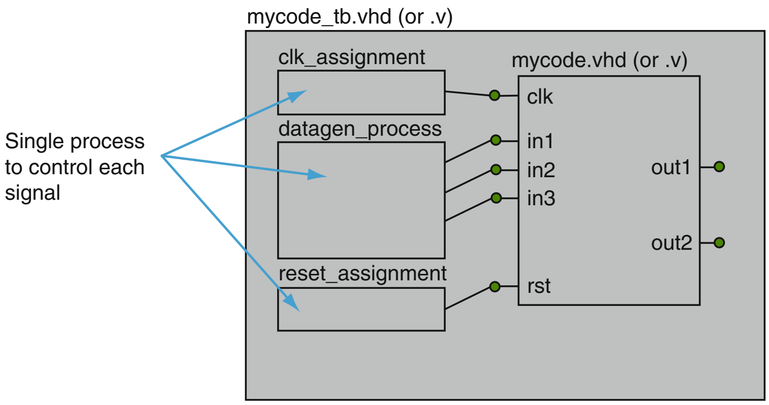


Fig. 11.2 Simple testbench that requires manual checking

11.7.6 Self Verifying Testbenches

Self verifying testbenches are more difficult to create. Being able to write the "expected results" requires a strong understanding of the design block under test. This requires more work up front as any errors in the "expected results" can be hard to catch. However once it is set, you can run the tests and get a quick pass or fail.

This is the approach that you should use for reusable design blocks.

When creating self-checking testbenches, you must add the functions to an existing process so that the outputs can be monitored. A "compare_process" or equivalent is used to check the received results against the expected results (Fig. 11.3).

This class of testbench can contain sequential or concurrent stimulus, as well as the expected results.

Often the signaling is too complicated to model without using vectors saved in "time-slices." This can be achieved using internal arrays or external files.

When using an array containing stimulus and with the expected results inside the testbench, there is no need to perform type translations. This provides faster simulation times, but is difficult to write and can create very large files.

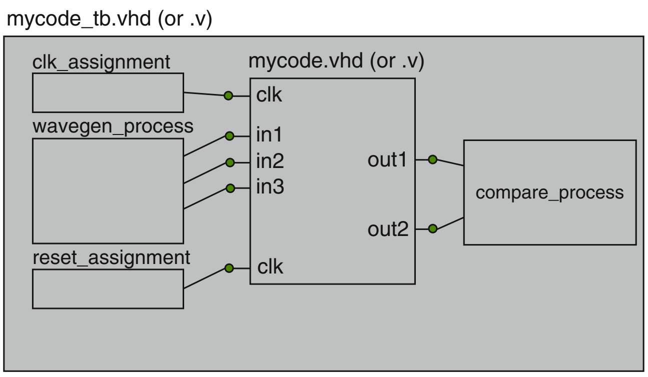


Fig. 11.3 Example diagram of a self-checking testbench

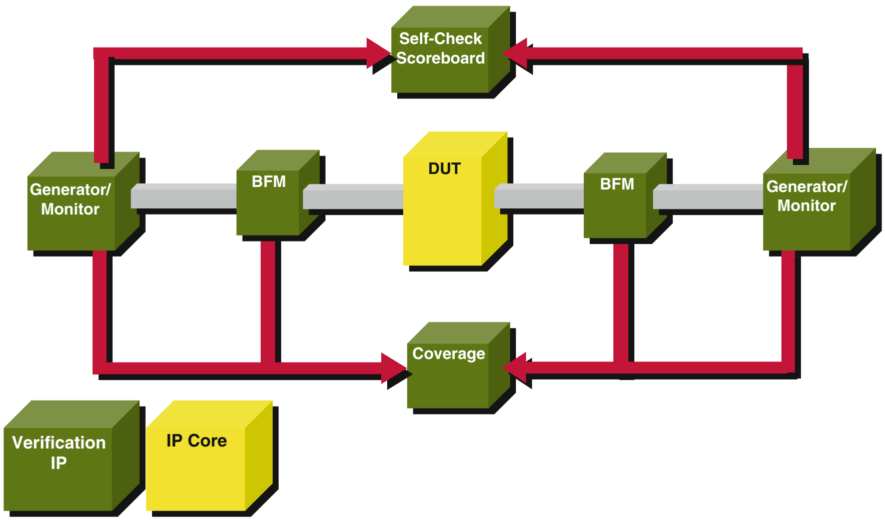


Fig. 11.4 Verification system architecture

When using an external file that contains the stimulus and the expected results, it is likely that you will need to use type translations. This can result in slower simulation times, but is easier to write (Fig. 11.4).

11.7.7 Formal Equivalency Checking

Formal Equivalency Checking compares the logical equivalence between different points in the design flow, or between different netlists. It uses mathematical techniques to compare the logical equivalence of two versions of the same design rather than using test vectors to perform simulation.

It is normally used to compare the RTL code to the post-synthesis gate level netlist to ensure that the synthesis optimizations have not introduced any bugs. It can also be used to compare the RTL or post-synthesis netlist to the post-fit netlist to ensure that the Place and Route optimizations have not changed the functionality of the design.

Whilst Equivalency checking can determine if two netlists are functionally the same, it does not guarantee functional correctness. If the design functionality has been implemented incorrectly in the RTL, equivalency checking will report a "Success" if the netlist it is compared with has the same functionality. Thus equivalency checking is normally used to compare functionally verified RTL to gate level netlists.

Formal Equivalency checking tools tend to be limited in the size of design that they can support and as such are used mostly on design blocks as opposed to complete designs.

It is a particularly difficult technique to use for FPGAs. FPGA synthesis optimizations perform a lot of register optimizations such as register merging, register duplication and register retiming. The first two optimizations can lead to false reports of failures. Investigation of the design can remove these false negatives but is time consuming. The third optimization type, register retiming, is usually a show-stopper. Most Formal Equivalency tools cannot cope with the register retiming that is performed by FPGA synthesis or physical synthesis. Thus Formal Equivalency checking is rarely used in FPGA design flows.

11.8 Code Coverage

Code coverage reflects how thoroughly the HDL code has been exercised.

It provides information about how many lines of code is executed, providing a quantitative measurement of the testing effort and assisting in the directing of future testing effort.

Code Coverage is limited in that it does not look at the sequence of events, nor does it check any interaction between design blocks. It only looks at what is in the design, thus can overlook what has not been implemented. In short, it does not look at the functionality of the design.

One of its benefits is that it can be used to hit the corner cases which are not reached by the random test cases. In order to do this, users have to write the directed test cases to reach the missing code coverage areas.

11.9 QA Testing

11.9.1 Functional Regression Testing

The objective of functional regression testing is to provide an automated environment that proves that the design operates as specified.

Regression testing is necessary to ensure that there is not the reemergence of old faults. It is considered good practice that when a bug is identified and fixed, that a test is created to test that the bug is fixed. This test is then run on any future changes to the design to ensure that the new changes have not re-introduced the bug. Regression testing automates this testing process. This test is combined into a test suite of designs that enables the testing environment to execute all the regression test cases automatically.

Typical automated QA regression testing exercises the IP or design via scripts. It compiles and compares the results against a known good standard. The testing is self-checking with a verification log for reporting exceptions. Note the use of the term exceptions. A test failure is an exception until any analysis determines that the failure was caused by a bug in the design. Often the exceptions occur due to problems with the test environment as opposed to a bug in the design. If this is found to be the case, the problem with the test environment should be resolved and the test rerun to verify that the test passes. The regression test environment must be capable of compiling the test statistics and reporting on the health of the design. This includes reporting on the individual design blocks as well as the final system design that integrates all of the design blocks.

11.9.2 GUI Testing for Reusable IP

While the GUI for IP should be relatively simple to use, it needs to be tested to ensure a good user experience. The GUI is likely to be other user's first exposure to your IP. You want to ensure that it is a good experience and avoid the scenario where your IP is not being used because of bugs in the Graphical User Interface.

There are test programs available in the market that will enable you to perform regression testing on GUIs, however the most valuable testing is Manual GUI testing.

The purpose of the testing is to:

1. Ensure that parameterization GUI functions as intended.
2. Validate the behavior when used correctly.
3. Validate the behavior under user error conditions.

The testing is performed by humans thoroughly exercising the GUI against a checklist. The testers click buttons, load files, examine expected results and perform error reporting.

This method of testing is labor and time intensive but will guarantee a good user experience with the graphical user interface.

11.10 Hardware Interoperability Tests

Hardware Interoperability testing is used where your design is interfacing with standard protocols. Hardware is tested in the lab against industry standard ASSP(s) and/or tested at industry plug-fests and testing laboratories.

11.11 Hardware/Software Co-Verification

There are tools on the market that enable hardware/software co-simulation. This is effectively running the ‘c’ code on the model of the hardware. The ‘c’ code will run much slower than it will on silicon. As such, it is a common technique with FPGA designs to bypass this test and run the code on the FPGA on a development board or in the end system.

11.11.1 Getting to Silicon Fast

FPGAs offer the ability to get preliminary designs on boards fast. In system testing can uncover bugs that cannot be detected using RTL verification. Hardware checkout should be combined with simulation to verify your design. Simulating the FPGA design is most valuable in the early stages of the design. Hardware checkout is useful when debugging interfaces and drivers.

11.12 Functional Verification Checklist

1. Create the test plan. This should detail the interesting test cases to verify the design.
2. Create the functional coverage specification. This should define what should be covered.
3. Build the system testbench.
4. Write functional tests and perform simulations to measure functional coverage.
5. Perform Code Coverage. This should only be run after the RTL is steady.
6. Achieve thorough coverage – if block coverage is at 100%, expand the system level coverage.
7. Perform GUI testing on IP.
8. Complete Hardware Interoperability testing for standard protocol IP
9. Perform In-system debug. This includes hardware–software co-verification with the software running on the targeted hardware.

Chapter 12
Timing Closure

12.1 Timing Closure Challenges

Timing Closure is the area of the design flow that can cause the most frustration to FPGA designers. This is the area which can eat up the compute cycles on your workstation, it can result in feature drop from your system design and may result in you having to pay for a faster speed-grade device than you intended to use.

Most of the chapters in this book have revolved around preventing timing closure challenges in your design.

This chapter presents moves onto the next stage by presenting a design methodology for achieving timing closure.

So, why is timing closure a challenge in FPGA designs?

Over the last decade there has been a huge increase in the FPGA device density and the size of the designs targeting FPGAs. FPGA device logic density has increased by approximately 30×, and the amount of embedded memory has increased by approximately 70×. Over the same period of time, the speed of workstation CPUs have only increased by a factor of 14. All of these create a compile time challenge for high density FPGA designs.

On top of this, the clock speeds of the designs and the interface speeds have risen substantially. Today's devices include transceivers that can reach speeds of more than 11 G and DDR III memory interfaces that run in excess of 533 MHz.

These types of applications require more complex timing constraints such as source synchronous interfaces and inter clock transfers.

The process geometries of modern FPGAs now dictate that timing analysis be performed at two or more timing corners in order to guarantee timing closure. At these smaller process geometries the delays are typically dominated by the delays of the interconnect routing as opposed to the cell delays. This creates a challenge in the placement of the design to avoid long interconnect delays whilst avoiding routing congestion.

The addition of dedicated hardware blocks, such as embedded memory and DSP blocks provide the benefit of increased functionality, but can create a challenge in placement with relation to the logic that interfaces with these blocks.

The good news is that the FPGA vendor software has risen to the challenges and includes a number of features to solve these challenges. In many cases, the default settings will meet your performance goals push-button. For the designs that do not meet your goals there are a number of analysis tools and features to enable you to succeed.

12.2 The Importance of Timing Assignments and Timing Analysis

Timing Analysis is the singly most important topic that you need to understand when it comes to timing closure. Unfortunately, it is also the topic that designers have the greatest challenge in understanding.

In this section of the chapter we will explain the importance of timing analysis and provide a basic background on timing analysis. In depth coverage of timing analysis could be a book in its own right. For an advanced understanding of timing analysis, it is recommended that you attend training from one of the FPGA vendors and download the various application notes from their websites.

Timing assignments serve two purposes in FPGA design.

1. They direct the synthesis and place and route software. The impact on place and route is described in detail in Sect. 12.3.4.1, "understanding the fitter (place and route)." Timing assignments drives where the optimizations are focused for synthesis and determines which paths the place and route engine needs to prioritize in the fitting process.
2. They are used in timing analysis. Timing analysis does not guarantee the functionality of the RTL but does guarantee that your design does not have timing violations. Static timing analysis computes the timing of the design without performing a simulation.

12.2.1 Background

If we step back in time, timing analysis on FPGA designs was relatively simple. The end applications were reasonably simple in that there were a limited number of clock domains and the timing models from the vendors were heavily guard-banded such that designers needed only to analyze the design at a single timing corner. Each FPGA vendor created their own timing assignment language with a heavy focus on the clock frequency. The FPGA vendors effectively sheltered the designers from needing to know the intricacies of timing analysis.

If we look at the current class of designs targeting FPGA devices, designers now face much of the same timing analysis challenges that ASIC designers have been facing for several years. Typical designs now use multiple clock domains, have complex relationships between clock domains and have a heavy focus on interface timing rather than purely finding the maximum clock frequency. On top of this the modern process geometries of 65 and 40 nm require that analysis be performed at multiple timing corners to guarantee operation. The original vendor timing languages were not originally designed for constraining this class of designs. This has resulted in FPGA designers needing to learn ASIC timing analysis techniques.

The good news is that FPGA vendors and the EDA tool industry is standardizing on a timing constraint language. This is the SDC (Synopsys Design Constraints) language from Synopsys.

12.2.2 Basics of Timing Analysis

This section of the chapter explains the common terminology that is used in timing analysis, along with a brief description of the base level of timing constraints upon which timing analysis is built.

12.2.2.1 Static Timing Analysis

Static timing analysis measures the timing delays along the timing paths in the design and reports the timing against the timing constraints. It identifies whether the design will operate functionally based upon the timing characteristics of the FPGA silicon. The timing analysis is performed independent of the functionality of the inputs and determines the delay of the circuit over all possible input combinations with every device path in the design being analyzed with respect to the timing requirements.

Static timing analysis catches timing-related errors faster and easier than gate-level simulation and board testing.

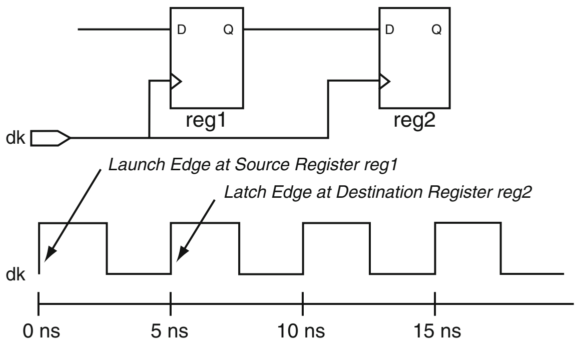
12.2.2.2 SDC

SDC is the acronym for Synopsys Design Constraints. This is the industry standard language for timing constraints that has been adopted by most FPGA vendors and EDA tools that support FPGA devices.

12.2.2.3 Clocks

Clocks are used to specify register-to-register requirements for synchronous transfers and to guide the Synthesis and Place and Route optimization algorithms to achieve the best possible implementation of the design.

Clocks should be the first constraints specified in any design's SDC files. This is important as many constraints reference clocks; therefore, the clocks must be defined first.


D
Q
reg1
D
Q
reg2
dk
Launch Edge at Source Register reg1
Latch Edge at Destination Register reg2
dk
0 ns
5 ns
10 ns
15 ns
Fig. 12.1 Launch and latch edge diagram

12.2.2.4 Launch Edge

The launch edge is an active clock edge that sends data out of a sequential element, such as a register, acting as a source for the data transfer.

12.2.2.5 Latch Edge

A latch edge is the active clock edge that captures data at the data input of a sequential element, such as a register, acting as a destination for the data transfer.

This is detailed, along with the launch edge in Fig. 12.1.

12.2.2.6 Hold Time (t_h)

Hold time is the minimum length of time for which data that feeds a register via its data or enable input(s) must be retained at an input pin after the clock signal that clocks the register is asserted at the clock pin.

A hold time failure occurs when an input signal change too quickly after the clock's active transition on a sequential element. This will result in a timing failure on the sequential element.

12.2.2.7 Set-Up Time (t_{su})

Set-up time is the length of time that the data that feeds a register via its data or enable inputs must be present at an input pin before the clock signal that clocks the register is asserted at the clock pin.

This is detailed in Fig. 12.2.

A set-up time violation occurs when a signal arrives too late at the input of a sequential element missing the time when it should advance. This will result in a timing failure on the sequential element.

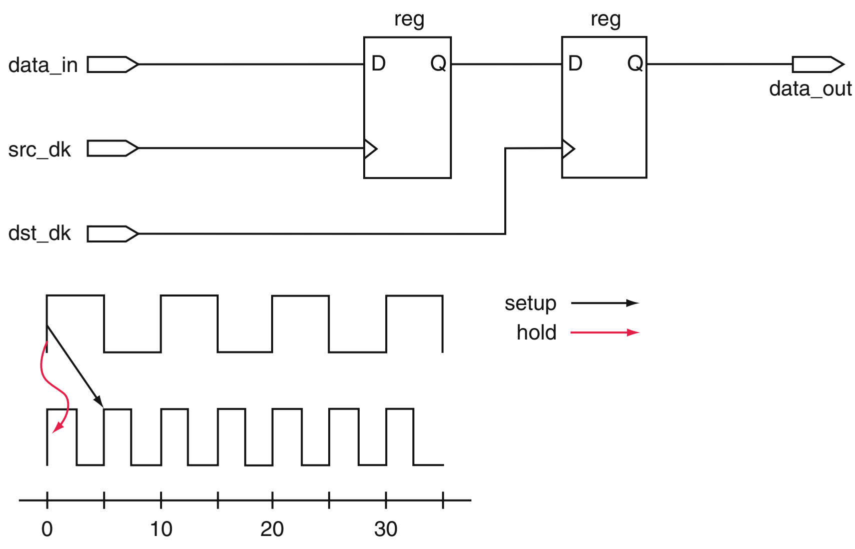


Fig. 12.2 tsu and th diagram

12.2.2.8 Arrival Time

Arrival time can be separated into data arrival time and clock arrival time.

Data arrival time is the delay from the source clock to the destination register.

Clock arrival time is the delay from the destination clock node to the destination register.

Data arrival time and clock arrival time are detailed in Fig. 12.3.

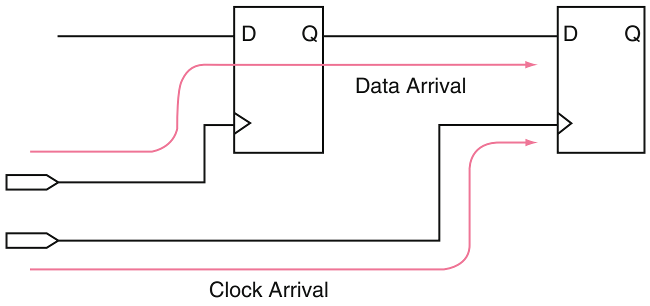


Fig. 12.3 Clock arrival and data arrival diagram

12.2.2.9 Required Time

This is the latest time at which a signal can arrive without making the clock cycle longer than desired.

12.2.2.10 Slack

Slack is the margin by which a timing requirement is met or not met. It is the difference between the required time and the arrival time. A positive slack value indicates the margin by which a requirement was met. A negative slack value indicates the margin by which a requirement was not met.

12.2.2.11 Timing Exception

This is a constraint that is not required, but may be needed to better describe how a design should work. Timing Exceptions adjust how timing analysis is performed on the design. Examples of timing exceptions are multi-cycle paths and false paths.

12.2.2.12 Multi-Cycle Path

Multi-cycle paths require more than one clock cycle for a signal to be updated. These paths need to be identified by the designer of the block, as their identification requires a detailed understanding of the functionality of the design.

A multi-cycle assignment relaxes the setup relationship by allowing you to specify the number of destination clock cycles required before a register latches a value.

Figure 12.4 details a Multicycle value of 2 to a clocked register which delays the latch edge by one destination clock cycle.

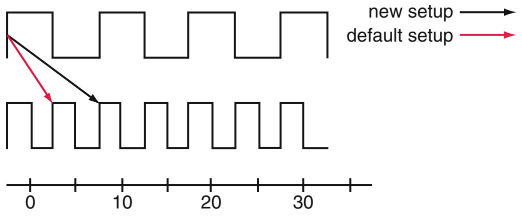


Fig. 12.4 Multi-cycle path

12.2.2.13 False Path

A False path assignment is used to define paths that the timing analyzer should not analyze. Examples of such paths are test logic or any other path not relevant to the circuit's operation. False paths are also commonly used on paths that cross clock domains.

12.2.2.14 Source Synchronous

Source Synchronous clocking is used to describe the technique of sourcing a clock along with the data. In source-synchronous interfaces, the source of the clock is the same device as the source of the data.

12.2.2.15 Rise/Fall Time

The rise time is the time required for a signal to change from a low value to a high value. A low value is typically 10% of the signal value and the high value is 90% of the signal value. The fall time is the time required for a signal to change from a high value to a low value.

12.2.2.16 Input Delay

The input delay (set_input_delay) specifies the required data arrival times at the specified input ports relative to the clock. The input delays are specified relative to the rising edge or falling edge of the clock (Fig. 12.5).

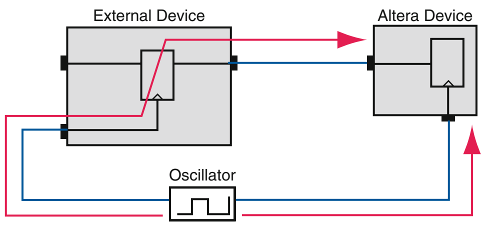


Fig. 12.5 Input delay

12.2.2.17 Output Delay

The output delay (set_output_delay) specifies the required data arrival times at the specified output ports relative to the clock The output delays are specified relative to the rising edge or falling edge of the clock (Fig. 12.6).

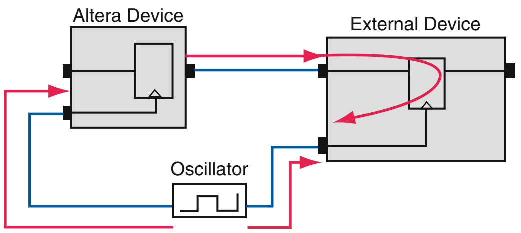


Fig. 12.6 Output delay

12.2.2.18 Operating Conditions

Operating conditions consist of the combination of voltage and temperature settings that are used during the timing analysis of the design. These values impact the delays in the timing models used during timing analysis.

12.2.2.19 Multi-corner Analysis

Multi-corner analysis allows a design to be verified under a variety of operating conditions while performing a static timing analysis on the design. This typically performed on the slow corner model and the fast corner model.

You must perform multi-corner timing analysis on your design before signing off on the design timing. Many years ago, FPGA vendors only provided a single timing model that represented worst case operating conditions. The model had enough timing guard-band built in that users could perform timing sign-off with the one model and be guaranteed that the design timing would work. As the process geometries of FPGA devices have shrunk to 65 nm, 40 nm and below, this statement is no longer true. You need to sign off on the design timing under best and worst case conditions. This means that you will have to optimize your design in both the best case and worst case operating conditions.

12.2.2.20 Slow Corner Model

The slow corner timing model indicates the slowest possible performance for any single path timing under worst case operating conditions. The model represents the slowest device at the max operating temperature and VCCMIN. The Slow timing model is typically used to ensure setup timing is met.

12.2.2.21 Fast Corner Model

The fast corner timing model indicates the fastest possible performance for any single path timing under best case conditions. This model represents the fastest device at the minimum operating temperature and VCCMAX. The Fast timing model is typically used to ensure hold timing is met.

This analysis allows you to verify that short paths meet timing requirements under best-case operating conditions.

12.2.2.22 Clock Uncertainty

Clock uncertainty is often referred to as the skew for clocks or clock-to-clock transfers. It is specified separately for setup and hold times and can specify separate rising and falling clock transitions (Fig. 12.7).

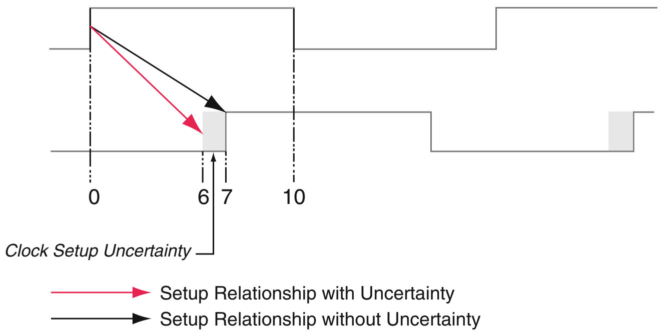

0
6
7
10
Clock Setup Uncertainty
Setup Relationship with Uncertainty
Setup Relationship without Uncertainty

Fig. 12.7 Clock uncertainty

12.2.2.23 Clock Latency

There are two types of clock latency. These are network and source. Network latency is the delay on the clock network between the clock and register clock pins.

Source latency is the clock network delay between the clock and its source (e.g., the system clock or base clock of a generated clock).

The source latency can be assigned to generated clocks for specifying board level delays from a clock output port to a clock input port when the clock input port is acting as a feedback clock.

12.3 A Methodology for Successful Timing Closure

This section of the book will describe a design methodology that will consistently enable you to successfully achieve timing closure in your FPGA design.

12.3.1 Family and Device Assignments

12.3.1.1 Speed-Grade Selection

It is recommended that you start with the fastest speed-grade of the targeted device to enable you to close timing quickly. This will enable you to get to the board quicker for functional checkout and to start on software development sooner.

You can work on optimizing the design for a lower speed device during the verification cycle or later once functional verification is complete.

12.3.1.2 I/O Settings

The drive strength and I/O standards that you select will impact the timing at your pins. They will also impact the power consumption and signal integrity of your device.

The techniques that can be used to improve the I/O timing are, in order of preference:

1. Ensure that the appropriate timing constraints are set on the I/O pins.
2. Examine the report file to determine if the I/O registers are being used. If they are not being used, look at the RTL and recode the RTL such that the output registers drive the pins and the pins drive input registers. The place and route software will normally use the I/O registers in order to meet the I/O timing requirements. If this is not working, you can force the use of I/O registers via settings in the FPGA design software.
3. Look at the delay chain settings for the I/O cells. Use the shortest delays for pins that feed or are fed directly by pins. Most FPGA devices have programmable delays options in the I/O cells that can be used to minimize the tsu and tco times. These are typically set by the FPGA design software based upon the I/O timing settings. If this is not working, you can manually set the delay through settings in the software.
4. Use PLLs to shift the clock edges to meet the I/O timing. If a PLL is providing the clock to the registers that are driving the I/O pin or are being fed by the I/O pin, the PLL output can be phase shifted to change the I/O timing. A backwards shift in the clock will provide better tco at the expense of tsu. Shifting the PLL output forward provides a better tsu at the expense of tco and thold.

12.3.2 Design Planning

As mentioned in Chap. 8, it is important that you plan up front for timing closure. Up front planning will help to identify issues before they arise and avoid delays late in the design cycle.

One of the common mistakes in timing closure is waiting for all of the RTL code to be available before compiling the top-level design. You should compile the top-level design as soon as the RTL for any of the major lower level modules is complete, in order to catch integration and resource issues as early as possible.

In order to be able to do this, you need to have planned for timing closure at the specification stage where you define how the design will be partitioned into functional blocks. This will include the timing requirements for the individual blocks, inter-block timing requirements and any placement restrictions on blocks that interface with dedicated hardware blocks or device pins. These requirements need to be adhered to when compiling the RTL at the top-level. More detailed information on RTL design partitioning is available in Sect. 8.5.2.3.

It is also recommended that you plan to use an incremental design methodology. In reality, by partitioning your design appropriately, as described in Sect. 8.5.2.3 you will have planned for an incremental compilation methodology. The advantage of such an approach is that it makes it easy to apply a team based design methodology to the FPGA design, whereby multiple engineers can work on the design and timing closure of the FPGA design. This design methodology will also enable you to minimize the impact of Engineering Change Orders on the design.

The major FPGA and EDA vendors include features in their FPGA design software to enable an incremental design methodology

12.3.2.1 Incremental Compilation

As mentioned previously, incremental compilation capabilities that are available from the FPGA vendors can dramatically shorten you compile times. This is not the only benefit of this approach. An incremental compilation methodology can shorten the timing closure cycle. The key factor behind the use of this capability is good design planning.

So, how does incremental compilation work?

Incremental compilation provides the ability to preserve the blocks in your design that have not changed and to only compile the parts of the blocks in the design that have changed. The net benefit is reduced compile time as there is less logic to recompile and a reduced number of compilations, as you can lock down the timing critical modules in the design once timing is met, thus preserving the performance of these blocks. A third benefit that is often overlooked is that you can add in debug logic when going to the lab without impacting the design. This is discussed in more detail in Chap. 13.

You should deploy an incremental design methodology.

You should also be aware of the restrictions that it can place on your design so that you can avoid the pitfalls.

1. It requires up front planning on the design partitioning, as described in Sect. 8.5.2.3. This can place restrictions on how your design blocks interface.
2. It prevents optimizations across design blocks. This restriction can be alleviated by maintaining the critical path inside a design block, by registering the ports on the design block and by not inserting combinational logic between design blocks at the next level of hierarchy.
3. It reduces the device utilization that you can achieve. This is true in that some of the area optimizations that exist in FPGA design software are more effective when applied to the complete design. An example of such an optimization is the packing of unrelated registers and LUTs in the same logic cell to save area. If you

are trying to utilize every logic cell in your design, you are likely to have timing closure issues due to the routing resources available in devices. Sacrificing device utilization for faster timing closure and higher performance is a decision that should be addressed in the device selection and specification. Most designs can reach 85%+ logic utilization and close timing using an incremental design methodology

Top-Down Design Flow

In a top-down design flow, the entire design is compiled in one project and timing closure is performed on the whole design. As the RTL for the different blocks in the design are complete, they are added to the top-level design and compiled with the rest of the design. One of the advantages of using this technique is that it provides good visibility into the paths between partitions. Timing closure is performed on the whole design. Once the designer is satisfied with the results for his block, it can be locked down such that it does not need to be recompiled, reducing the compile time and locking down the performance.

Bottom-Up Design Flow

In a bottom-up design flow, the modules are compiled in separate projects and locked down once the designer has achieved timing closure on the blocks. The lower-level partitions are then imported into the top-level project for final integration. This does not require a recompile, but rather a merger of the place and routed netlists followed by a routing operation for the connections between the blocks (Fig. 12.8).

The bottom-up design flow lends itself to a simpler partitioning of the design between different team members, but has the disadvantage of involving total isolation of lower-level modules. This requires more up front effort in the allocation of chip

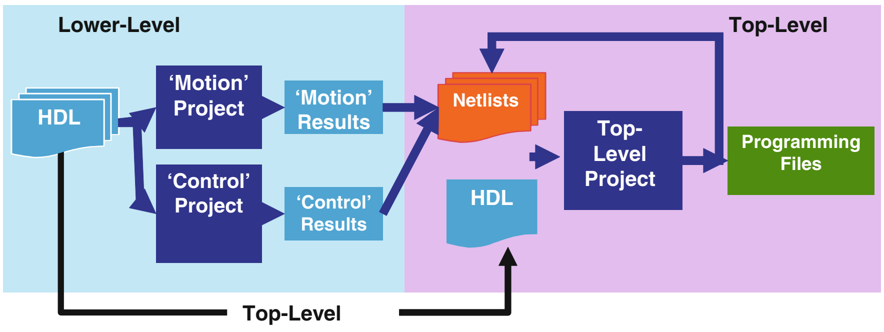


Fig. 12.8 Bottom-up design flow

resources. This creates the need for detailed floorplanning to accommodate each block that will be compiled in a separate project. It also complicates the timing constraints for the overall project as timing constraints need to pass from the top-level project to the lower level project. Any timing constraints that are added in the lower level project will also need to be migrated to the top-level project (Fig. 12.9).

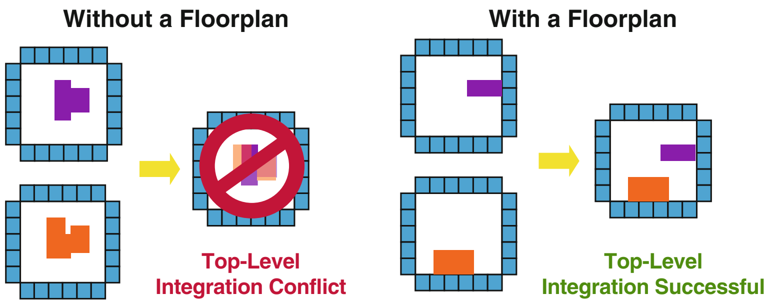


Fig. 12.9 Integration of modules in the top-level design

12.3.2.2 Design Scenarios Using Incremental Compilation

In this section we are going to look at a few scenarios where incremental compilation can significantly reduce the timing closure cycle.

Take the example design shown in Fig. 12.10.

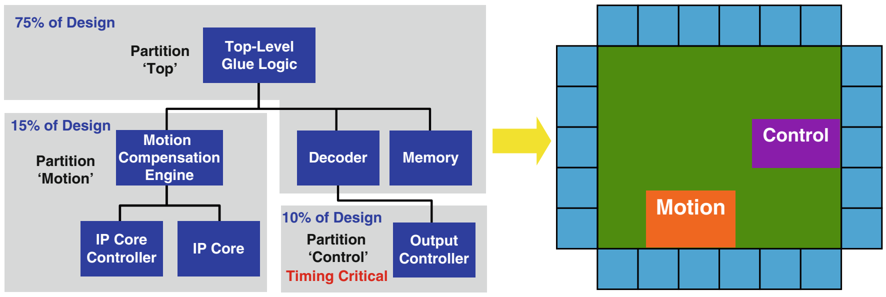


Fig. 12.10 Example design partitioned for incremental compilation

This design has been planned to contain three main hierarchies that have been partitioned for incremental compilation. The hierarchy "Motion," the hierarchy "Control" and the block "Top". Top is the top-level hierarchy of the design and contains the block "Motion," the block "Control" as well as other levels of hierarchy. The block "Motion" is also hierarchical containing two other design hierarchies and the block Controller is a sub-set of the "Decoder" Module which is one of the design blocks in hierarchy "Top". The design has been compiled and meets performance.

Scenario 1: Parameter Tuning

In this scenario, the system needs some fine tuning due to a small change in the specification that will impact the memory module in the top-level file. The user can lock down the place and route on the "Control" and "Motion" blocks, as their RTL will not be changed, make the change to the block "Memory" and recompile the block "Top". This will preserve the performance of the "Control" and "Motion" blocks as they are not compiled and greatly reduce the compile time as only 75% of the design has to be recompiled and the timing critical block that would typically challenge the fitter has not been touched.

If this design typically compiles in 6 hours, a complete recompile means that you can only achieve one iteration of the design in a normal working day. It is usually an iterative process to make a design change successfully.

By using the incremental compilation approach, your compile time would likely drop to less than 4 h, enabling two design iterations in a day, possibly more if these parts of the design are not timing critical allowing you to use the fast compilation options described in Sect. 12.3.3 on early timing estimation.

Scenario 2: Bug Fixing

In this scenario, you have finished the design and are in the final stages of in-system testing in the lab. The system is running at-speed and you have a functional failure. You need to find and fix this bug fast.

You can preserve the place and route of the complete design and utilize some of the debug options available from the FPGA vendors without having to complete a total recompile.

You can route internal signals in the design to unused pins quickly without disturbing the placement or routing of your design.

You can add in the Embedded Logic Analyzer from the FPGA vendor without recompiling the blocks "Top," "Motion" and "Control." As you try to isolate the bug, you can refine the trigger conditions of the Embedded Logic Analyzer and quickly create a new programming file.

A total recompile would take 6 hours and would change the design implementation. Without the incremental compilation methodology, the addition of the Embedded Logic Analyzer, or changes to the Embedded Logic Analyzer may cause the bug to disappear; leaving you wondering is your design functionally correct? Will the problem reappear in production?

Using the incremental compilation capability, the design implementation is preserved and the compile time is likely to be in the order of 45 min; enabling multiple iterations as you debug the design. The design preservation guarantees bug reproduction.

An example of the type of bug that you would capture is an asynchronous signal with a race condition. This type of bug is hard to capture with simulation. Once you find the bug in-system, you correctly constrain the paths and recompile the blocks that are impacted.

This is the recommended methodology that you should adopt for bugs that only occur when running at speed.

Scenario 3: Timing Closure

In this scenario, there is a need to make a few enhancements to the time to increase the overall performance of the design. This may happen if you receive a new version of IP from a third party. In the example that we have been looking at, a new version of the "Motion" core must be used. The specification has also changed such that the block performance must increase from 120 to 150 MHz.

You compile the design and have trouble closing timing in the "Motion" core. You do not have the option to optimize the RTL code, as the design is an encrypted core from a third party. Your only option, outside of waiting for the IP vendor to deliver a new version of the IP core, is to use the advanced optimization settings in the FPGA vendor software. You try the various settings until you close timing on the IP core, "Motion" and lock in the results by setting the block to post-fit and preserve routing.

If there is a change in any of the other design blocks, such as "Top" there will not be a timing closure problem on the blocks "Motion" and "Control" as they are locked down.

12.3.3 Early Timing Estimation

As mentioned in the Chap. 8, timing estimation is inaccurate unless a design has had some level of placement performed. Early in the design cycle, you do not want to go through a complete place and route compilation to get a performance estimate for your design. The FPGA vendors have provided a solution to this problem.

Most FPGA vendor software includes a setting that results in reduced compile time. This is achieved by limiting the number of placement attempts. This can dramatically reduce the compile time, usually at the expense of performance. The timing results using the fast compilation options are usually within 10% of the results that can be achieved by performing a full compile, but in less than half of the compilation time. This is a powerful tool that can greatly reduce your timing closure cycle.

It is recommended that you use this Fast compilation option in the following scenarios.

1. Early in the design cycle when you are determining the performance on design blocks that are undergoing change. Your timing results are likely to be within 10% of what is possible, but your iteration time will be significantly shorter.
2. Use it on complete designs that can easily meet timing. If your design is not high performance compared to the FPGA technology being targeted, this mode will reduce your iteration time throughout the full life of the project.

The project documentation should reflect the fact that this fitter option has been used for the design or for a particular design block.

If your design is missing timing by more than 10%, go back and work on the RTL rather than continuing with a complete compile.

As stated in design planning, you should compile your major design blocks as early as possible at the top-level of the design in order to catch integration and resource issues as early as possible. In order to achieve this, you can create dummy blocks for the blocks that are not complete. These empty blocks need to contain the correct port connections.

12.3.4 CAD Tool Settings

It is recommended that you try to maintain the default Synthesis and Fitter settings. The FPGA vendors provide you with dozens of knobs and switches that will impact the results. You should avoid the temptation to fiddle with them and only use them when you have exhausted your RTL coding capability.

This being said, these settings can be very effective and can drastically change compilation results. However the results that they provide can vary significantly from one release of the FPGA vendor software to the next. Thus they can make your design non-portable between tool versions, effectively making your IP non-reusable.

If you have your back to the wall and have to close timing on this project at all costs, then you should take advantage of these options.

In addition to optimization settings, the FPGA vendor software also provides the ability to influence the result via floorplanning of the logic. You can specify cell placements, in various groups, regions, down to individual routing tracks.

Again it is recommended that you avoid doing this unless the FPGA vendor software is doing a poor job on placement.

It is rare for human architecture experts to beat the tool with hand-work, however it can work in isolated cases and is another weapon in your arsenal if it appears that all hope is lost.

12.3.4.1 Understanding the Fitter (Place and Route)

The Place and Route tools from the main FPGA vendors will adjust their operation to try and meet the requirements for your design. This means that you will see different results based upon your timing constraints. Tougher timing constraints equates to longer compilation time.

The Place and Route engines are timing driven and understand complex timing constraints. Thus it is recommended that you use real timing constraints.

The Fitter tries to find a placement that can be routed to meet your timing requirements.

One of the phenomena of FPGA Place and Route Software is the variation in results based upon the "seed effect."

The initial placement for the logic is random, based upon the starting condition of your design and it is possible that different placements can meet your goals. The Place and Route seed, also known as the Fitter seed, changes the initial starting point of the algorithm for placement, effectively impacting how optimizations proceed. The Fitter's algorithm runs multiple placement attempts based upon the previous results to converge on a successful result. However, by changing the initial starting placement you may result in a different final placement and hence different timing results.

A common technique used in timing closure is "seed sweeping". This is running multiple different seeds to determine which will give the best result for your design. In the past, seed sweeping resulted in large changes in performance. Today, the average change in performance for the latest FPGA technologies is in the ±5% range. Note this can change significantly from FPGA vendor to FPGA vendor and family to family.

It is recommended that you avoid using seed sweeping on design blocks that you intend to reuse or on final designs that are likely to require future updates as the same seed will have a different effect in future versions of the FPGA vendor software or if you make any changes to your design, such as logic changes, assignment changes or pin changes.

So when would you use seeds?

1. If the design can meet timing, however you want to maximize your timing margin.
2. You need to quickly get the design in the lab for functional checkout. You should always go back and remove the need for a particular seed or seed sweeping.
3. This is the final version of the design, it is the only way to meet timing and there will not be future versions of the design. An example of this would be FPGA prototyping of an ASIC design.

An IP, or design block is not reusable if timing closure depends upon a particular seed and hence a particular version of an FPGA vendors software.

12.3.4.2 Advanced Optimization: When You Need More

As mentioned in the CAD tool settings section, FPGA design tools provide dozens of options for optimizing your design. In this section we will cover the options that are typically most effective.

Physical Synthesis Optimizations

Most FPGA vendor tools contain Physical Synthesis optimization options. Physical synthesis is tightly integrated with the place and route engine and re-synthesizes the logic where timing is a problem. Common techniques that are used include register retiming and register duplication. These are techniques that could be fixed at the RTL level, but may require major recoding. There are a lot of other optimizations performed by Physical Synthesis but these are the most common and often most effective.

In certain designs, it can improve the clock performance by greater than 20%. For designs which have been carefully coded with balanced registers, the performance gain may be only 1–2%. This optimization comes with a price. The design compile time will increase dramatically, normally by a factor of 2 or more. It will also limit your use of Formal Verification tools as they typically struggle with register retiming optimizations.

Due to the compile time impact, you should consider limiting the use to problem blocks in an incremental design flow.

The use of Physical Synthesis is fully automated, i.e. you set the option and compile.

Design Space Exploration

Most of the FPGA vendors provide utilities in their tool that will automatically run multiple compilations using different settings and seeds to find the settings in the tools that provide the best results for your design.

Due to the effect of seeds on place and route, you should only use Design Space Exploration in the late stages of your design when the design is effectively complete and you are focused on timing closure.

This type of utility will typically perform ten or more compilations and as such can result in compilation times of several days.

Fortunately the main FPGA vendors have added multi-processing to their utilities such that multiple compilations can be performed in parallel as opposed to sequentially. This greatly reduces the compile time.

The downside of using a Design Space Exploration tool is that if you make a change to the RTL of your design, you will need to rerun the utility due to the random nature of seeds.

Design Space Exploration can be run on individual blocks in your design. This is a powerful technique for reducing the compile time and only focusing the optimizations on the performance critical areas of the design.

This technique is particularly effective in an incremental compilation design flow where Design Space Exploration is only run on the blocks of the design that are timing critical.

If you use Design Space Exploration on a design block or complete design the exact settings used should be documented with the design to enable other users to recreate the results.

12.3.4.3 Compilation Reports and Analysis Tools

Review the messages from the synthesis and place and route reports to help with timing closure. These will often provide information that can be used to help improve the performance of the design. Your design process should dictate that designers should always review and remove all warnings from a project. This is necessary as the messages may indicate problems with the design such as the inadvertent use of latches or missing timing constraints. One of the challenges with reviewing warnings is that the messages may come from purchased IP and you cannot change the RTL to remove the message. In this scenario, you should check with the IP vendor on the message and if they prove that it is safe to ignore the message, you can document this information in the project and ignore the message for future compilations.

The report file itself details information on resource usage in the device and can be used to determine which modules are using the most resources in the device.

Information from the compilation reports, such as the amount of time spent in placement and routing, can help identify challenges to the fitter. Long route time can be due to restrictions created by the placement. This can be improved by possible hand placement of some nodes or increasing the placement effort.

The compilation report also provides details on the optimizations that have been performed, such as the registers that have been removed from the design. This information can help you to find problems in the RTL, or explain why debug logic has been removed, enabling you to fix the RTL.

Similarly messages on ignored assignments can resolve problems caused by typos when creating assignments or identify assignments that are out of date and should be removed from the project.

In addition to the compilation report files, the FPGA vendors provide tools that detail the design in graphical form.

These tools should be used when examining the results for gaining an understanding of the RTL and viewing the results of synthesis and place and route.

These viewer tools provide hierarchical block diagram views of the design, as well as a technology implementation view detailing how the design has been mapped to the target technology after synthesis or after fitting.

The hierarchical block diagram view is useful for understanding the architecture of the design, thus is useful for understanding the design flow as shown in Fig. 12.11.

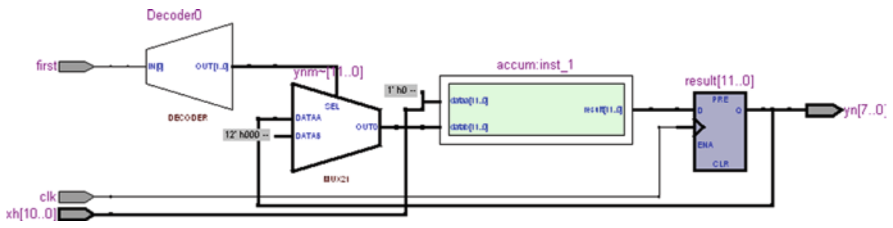
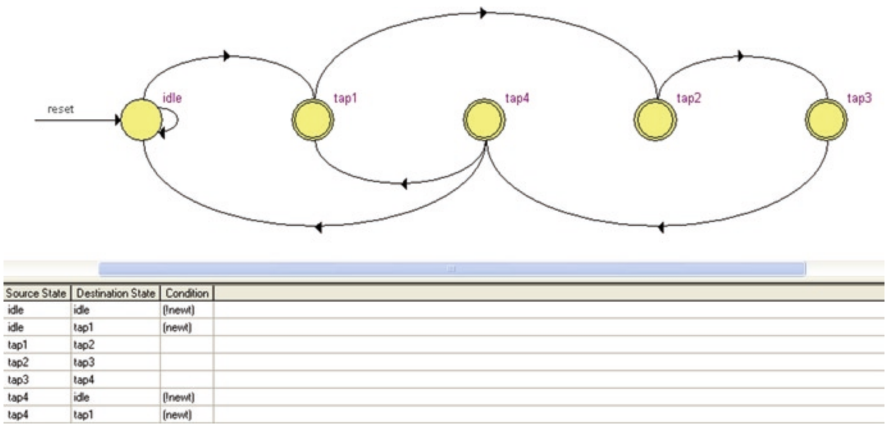


Fig. 12.11 Example of the RTL viewer in the Quartus II software

This should be applied when inheriting design blocks from other users to gain a visual understanding of the design and for planning the floorplan of a device as it will detail the data flow through the design and interaction of the blocks. It also provides visibility into functions such as Finite State Machines as shown in Fig. 12.12.



Source State	Destination State	Condition
idle	idle	(!newt)
idle	tap1	(newt)
tap1	tap2	
tap2	tap3	
tap3	tap4	
tap4	idle	(!newt)
tap4	tap1	(newt)

Fig. 12.12 Example view of a FSM from the Quartus II RTL viewer

The technology-specific view is useful for understanding how the design has been implemented in the FPGA and can be used to determine where optimization is possible.

It can quickly detail the number of levels of logic in the critical path and can link back to the RTL to help relate the implementation to the original RTL.

The technology map view helps in creating legal complex timing constraints for your design when used with the timing analysis tool. It is possible to locate from a path in the Timing Analysis timing report to the Technology Map View. In the Technology Map view, you can examine the implementation, determine whether the path is a timing exception, such as a multicycle path or false path, and then make the appropriate assignment in your timing constraint file.

12.3.4.4 Floorplanning Tools

All FPGA vendor design tools contain a floorplan tool, or in some cases multiple floorplan tools.

In the early days of FPGAs, these tools were critical for both understanding the FPGA architecture and optimizing the design for performance.

Today, the former statement is still true. Floorplan tools help explain what resources are available in the FPGA device and can be useful in analyzing the results of place and route on a design. The latter statement on design optimization is less true. In most cases it is not necessary to floorplan a design to meet the performance requirements. In the cases were floorplanning for performance provides a benefit, you will likely be floorplanning a small part of the design rather than all of the design.

Today there is another area where floorplanning can help. This is in a bottom-up team based design flow. In this scenario, you will assign design blocks to areas of the device rather than designing at the cell level. Each major design block is assigned an area in the device.

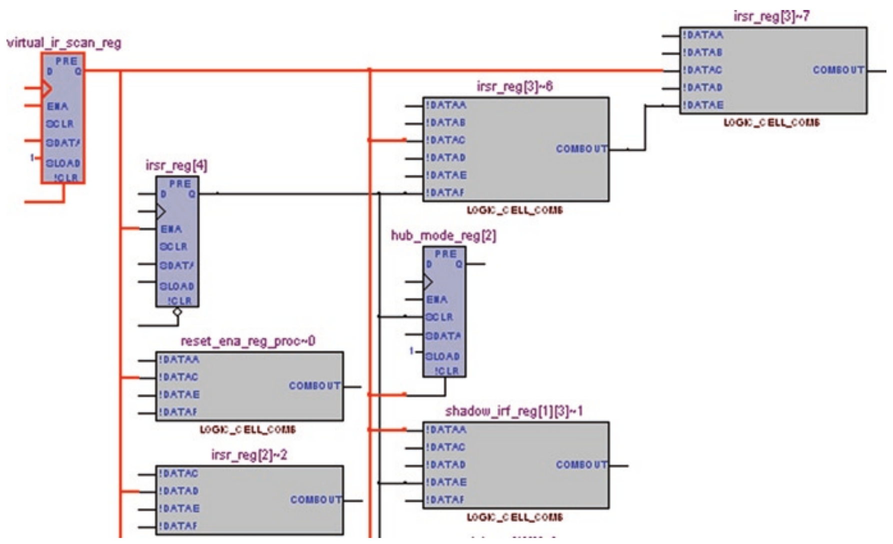


Fig. 12.13 Critical Path View in Quartus II technology map viewer

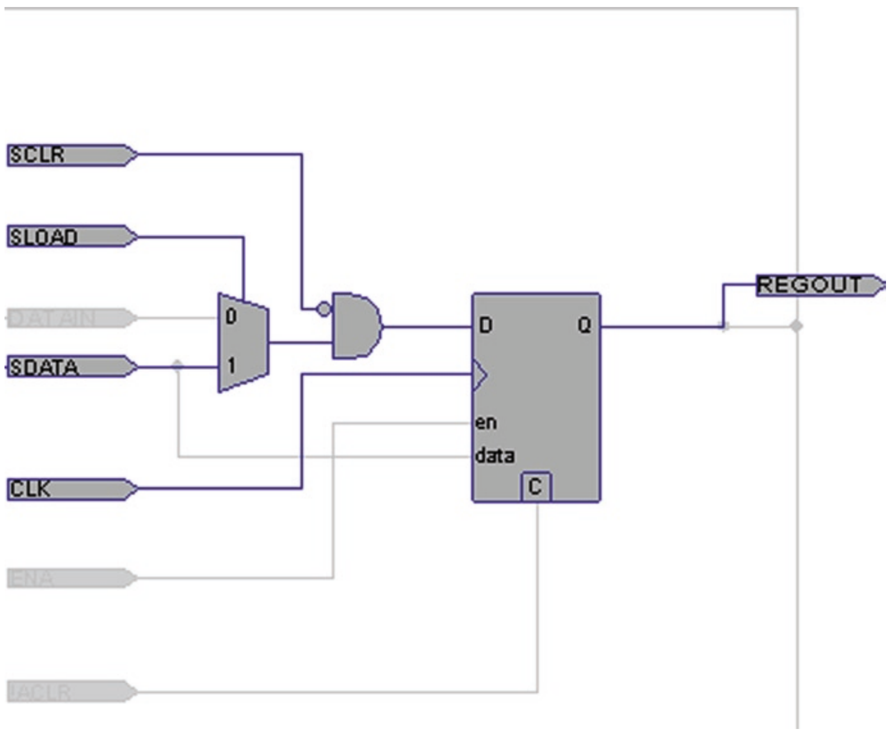


Fig. 12.14 The Quartus II chip planner detailing the Stratix IV ALM architecture

In summary, there are four main uses of the FPGA vendor floorplan tools. These are architecture exploration, analysis of placement and routing, creation of floorplan assignments and Engineering Change Orders.

Architecture Exploration

The floorplan provides a visual display of chip resources. It is akin to having a data sheet on your desktop that details the resources used as well as the resources that are still available. The floorplan can be used to view details on the device architecture, such as the number of registers in a LAB, number of LABs in a row, placement of memories and routing information. It will also allow you to view the logic inside of dedicated blocks, such as the configuration of LUTs and registers.

It provides visibility into the configuration of the I/O cell such as details on the delay chains, I/O standard, direction and use of registers inside of I/O cells.

It is a real benefit in team based designs for viewing the connectivity of your design blocks.

It is also extremely useful for clock network planning. As well as detailing the configuration of PLLs it details which areas of the chip can be driven by the outputs of the PLLs and from the global signals in the device. This capability works well in a team based design environment where you need to assign devices resources to the different engineers and functional blocks, preventing resource conflicts and enabling you to plan for the sharing or merging of resources, such as PLLs.

Analysis of Placement and Routing

The floorplan tool provides an excellent solution for examining design implementation.

It displays logic placement information, detailed routing information, fan-in and fan-out connections and enables the viewing of critical path information.

An analysis of placement and routing need only be performed if you have a problem. In the case of timing failures it can be used with the timing analyzer to locate from failing paths in the timing report to a view of these paths in the floorplan. It is then possible to analyze the placement and routing of the design to determine if the issue can be fixed by location constraints or to get visibility into the congestion in that area of the chip.

The floorplan provides visibility in the number of levels of logics between registers as well as whether the registers in the I/O cell are being used. This information can also be viewed in other tools such as the compilation report and Technology map views.

Floorplan Assignments

The floorplan can be used to optimize the performance of the design through placement assignments. In most cases it is difficult to perform a better placement than what the place and route software does automatically. However there are cases where it can help. A good example is the placement of pipeline registers between nodes that are placed far apart due to resource constraints, such as access to dedicated hardware blocks and/or pins. In this scenario, the place and route software does not always optimize the placement of the registers between the source and destination nodes, Users can move the registers on the floorplan for optimal placement and performance.

Assignments should mainly be used in the floorplan to create region constraints in an incremental or team based design environment. In this scenario, regions are created in the floorplan and blocks of the design assigned to the region. Alternatively region assignments can be used to prevent the resources in a region being used, effectively reserving resources for design blocks that are not yet complete.

One of the challenges in creating region assignments is dealing with internal memory blocks and DSP blocks. Depending upon the resource requirements of the block you may need a non-rectangular region in order to include enough memory or DSP blocks for the design.

You also need to consider how the design block interfaces with the rest of the design so that you do not inadvertently hurt timing closure.

Engineering Change Orders

The floorplan tool can help in the in-system design debug cycle. It provides a means to try out small design changes quickly.

It allows the editing, creation and deletion of logic and connections in the design. It is recommended that you only do this for simple changes, such as changing the polarity on clocks, clock enables, or the insertion of simple test logic.

This method is particularly useful for changing the properties of I/O cells such as delay chain values, use of pull-ups, slew rate, I/O standard and current strength.

It should also be used to modify the PLL settings or for routing a signal out to a pin for analysis.

It is not recommended that you go to production using changes that are made to the logic with this method, as the RTL will no longer match the functionality of the implementation. This method should only be used to try out simple changes and when proven to work in-system, the RTL be modified to match the functionality, the design simulated, recompiled and the new programming image tested in-system. The full verification cycle should be performed on this new version of the design.

12.4 Common Timing Closure Issues

This section lists some of the common timing closure issues that you may face and recommends the course of action that should be taken to resolve the problem.

12.4.1 Missing Timing Constraints

The FPGA vendor place and route software optimizes the design based upon the timing constraints that are provided. If you fail to constrain a critical path, this path will not be optimized by the FPGA software and may fail timing. To further complicate issues, you may not know that you have a timing problem. Timing analysis will only report timing against the timing constraints, thus if a path is not constrained, it will not be analyzed.

Most timing analysis tools have a command to report paths that do not have timing constraints. It is recommended that you run this command to determine if you have unconstrained paths and then set the appropriate timing constraints on the paths.

It is important that you use the correct timing constraints for your design. Analyze the timing report and ensure that any multi-cycle or false paths truly are timing exceptions. It is easy to use wildcards as part of a timing exception and inadvertently apply the constraint to a register that is not a timing exception, resulting in a timing failure in-system that is not reported as a failure by timing analysis.

12.4.2 Conflicting Timing Constraints

It is possible that you create conflicting timing constraints on paths through the use of wildcards. While the use of wildcards is encouraged, you need to be certain that a wildcard is appropriate. If a path has conflicting constraints, the optimization of the place and route engine will only work on one of the constraints. This is generally the last constraint entered. This can result in a timing failure on the other constraint.

Timing conflicts often happen in designs with paths between multiple clock domains.

12.4.3 High Fan-Out Registers

The location of the destination registers for high fan-out registers can result in long routing delays between the source and the destination register. The Place and Route software will normally optimize the placement such that this is not a problem. However it can still be a problem when location constraints restrict the placement options. An example could be a register with a high fan-out that feeds many registers that interface with pins on different sides of the device and there is a tight tco requirement from the registers to the pin. The destination registers have to be placed inside or next to the I/O cell to meet the tco timing. The source register cannot possibly be placed close to all of the destination registers.

The best solution to this is to either:

1. Create better pin assignments, or
2. Duplicate the source register such that it can be placed close to each group of pins. This is best performed at the RTL level.

12.4.4 Missing Timing by a Small Margin

If your design is complete, you are marginally missing timing and your schedule does not permit you to go back to the RTL code, then you should try every option that is available in the FPGA design tool to try and close timing. Most of the vendors have design space exploration features that will cycle through variations of the optimization settings along with seed sweeps to try and find the optimal settings to meet timing on your design. This approach is extremely time consuming as you may have to run 10+ compilations. However, it can provide performance improvements in excess of 20%. In order to reduce the compile time hit of performing multiple compilations, you should compile multiple settings in parallel on multiple machines using the capabilities inside the Design Space Exploration tools.

12.4.5 Restrictive Location Constraints

When location constraints are used early in the design process, there is a tendency to keep the constraints throughout the evolution of the design. This can result in the scenario where constraints that added value to the early versions of the design can hinder the performance in later versions of the design.

There is also the temptation to overly constrain the design. The constraints may work well on individual blocks, but when the design is integrated restricts the optimizations that the place and route tool can perform, resulting in poor performance.

In both of these scenarios, the recommendation is to create a new revision of the design and remove the logic location constraints. If the design does not meet your timing requirements, examine which blocks are having the problem and add back in the constraints on the problem blocks individually. See if it impacts timing. If it does not, remove the constraint. If it does, keep the constraint and move onto the next constraint.

Ideally you want to be able to close timing without using logic location constraints.

12.4.6 Long Compile Times

The first technique is to use an incremental compilation design flow. If you have used an incremental compilation methodology then you will not be suffering from long compile times.

The second technique compliments the first technique. That is to use a workstation with multiple processors or multi-core processors. The algorithms in the FPGA vendor software are multi-threaded and can take advantage of multiple cores or processors to reduce the compile time. To compliment the multiple processors you should ensure that the workstation has plenty of fast RAM. The compilation of designs targeting the latest FPGA devices can use as much as 16 G RAM. The algorithms are constantly accessing RAM, thus fast RAM will help the compilation time.

If your design meets performance reasonably easily, you may consider using one of the FPGA vendor options to quickly fit the design. This can cut the compile time in half but will result in reduced design performance.

12.5 Design Planning, Implementation, Optimization and Timing Closure Checklist

1. Follow synchronous design practices.
2. Follow recommended coding guidelines.
3. Partition the design for an incremental design methodology.
4. Ensure that the RTL is taking advantage of the dedicated hardware resources in the device. This can be achieved by instantiating vendor primitives to access special hardware features that cannot be inferred from RTL.
5. Create complete timing assignments for the design.
6. Ensure that any multi-processor features for reduced compilation are enabled.
7. Floorplan timing critical partitions in the design.
8. Perform timing analysis at all process corners.
9. Analyze all warnings and errors. Make the necessary changes to remove these warnings and document any exceptions.
10. Document the settings that achieve timing closure.

Chapter 13
In-System Debug

13.1 In-System Debug Challenges

The debug of any chip that is operating in-system is a challenging a nerve racking experience. As your board springs to life…. or not, the thought that crosses your mind is “Does my design work?” Then the real discussion starts, is it the system software or the system hardware. Due to the expense in developing system software, the hardware is almost assumed guilty until proven innocent. In this chapter we will look at techniques that can be deployed to identify the problems, quickly.

FPGAs have a distinct advantage over ASICs when it comes to in-system debug. This is programmability. With an ASIC design, you have to design your debug logic up front in order to prove the design operation on the board. With an ASIC, you need to be as close to 100% certain as possible that the design is functionally correct in order to avoid an expensive chip respin. The up front design of debug logic is a critical functionality that should also be used when designing FPGAs. However, the programmability that is inherent in FPGAs enables debug logic to be controlled by a host processor or added to the design as the in-system debug progresses.

The intent of simulation is to catch any design or integration bugs prior to getting to silicon. However, exhaustive simulation of an FPGA design is time consuming and compute intensive. The ability to stimulate a design under real world conditions, can uncover problems that are difficult to detect in simulation. Examples of such problems are asynchronous timing issues, signal integrity peculiarities and hardware/software integration issues.

In this chapter we will recommend a debug methodology that will enable you to verify your design operates in-system as intended and helps you capture problems with your design while operating in-system. The techniques discussed will draw upon the tools and techniques that are commonly available today.

13.2 Planning

When creating designs, most engineers tend not to consider that they will have bugs in the design or implementation. Inexperienced engineers only start to think about in-system debug once there is a problem with the board. The seasoned veteran has been through the pressure of debugging designs many times and wants to minimize the time spent in this high pressure environment. He/she wants to avoid spending evenings and weekends in the lab determining the cause of a problem. As such, these engineers plan for debug up front. This is what you need to do!

In-system debug should be part of the design specification. Each of the major blocks in the design should have a plan for how its operation is going to be verified in-system and what the debug strategy will be for that block. This should include information on the type of information that can be viewed to determine that the block is operating as intended. This includes system level statistics, such as the efficiency of memory interfaces, performance bottleneck analysis on buses and bit error ratio information on high speed transceiver interfaces.

In addition to the debug of blocks, there should be a debug plan for the top-level design, when all of the design blocks are implemented. This information is derived from the information in Chap. 4, where it addresses density and pins.

This plan should specify how many pins and how much logic and memory are reserved for in-system debug. It should also detail the techniques and tools that will be used as part of the in-system debug process.

A good guideline is to reserve 15% of the device pins for debug of the design. This does not include the JTAG pins that are used for programming the FPGA and can be used as part of the debug process. The recommended resource requirements for debug will be discussed further in Sect. 13.3 on debug techniques.

13.3 Techniques

There are multiple tools available from FPGA vendors and EDA Companies that can be used to facilitate the debug of your design in-system. In this section we will look at the mostly commonly used tools and techniques and recommend when they should be used.

13.3.1 Use of Pins for Debug

This is the mostly commonly used debug technique for FPGA designs. One of the reasons that it is so popular has to do with the programmability of FPGAs and the fact that compile times for routing different signals to the pins are fast. Thus when debugging in the lab, you can have a new programming file that routes a different set of signals to the debug pins in tens of minutes. In most cases this can occur without impacting the previous design implementation, outside of adding a fan-out on the signals that you are probing.

If your design is highly utilized, it may be necessary to change routing or placement in order to be able to access the signals. This latter scenario should be avoided; as such a change may cause any asynchronous timing issues to disappear.

This capability requires that you have reserved selected pins or a bank of pins for debug.

There are several ways to route internal signals to pins in the FPGA design software. The most common approach is via the Floorplan tool where you select the required signal as the source and the pin as the destination. The Place and Route software will incrementally route the signals to the pin. This approach is simple for one or two signals. However, it can become laborious for larger groups of signals. A common example is debugging a 32-bit bus on 32 pins. Some of the tools have the capability to allow you to select the source and destination via a signal find utility or scripting interface, and then it will automatically route the signals to the pins.

The timing of the routing of the signals at the pins is important, particularly if routing a bus out to the pins. It is recommended that you register the pins at the pins to synchronize the bus to a clock. You do not want these signals to be the critical path in your design, thus you should add timing constraints to these paths. For high performance designs you may need to insert several levels of pipeline registers between the signal and the pins. Once again this is an automated option in some of the FPGA vendor software offerings.

The steps in using pins for debug signals are:

1. Reserve the pins for debug
2. Set the appropriate I/O standard on the pins
3. Identify the signals that you want to route to the pins
4. Determine if the signals require the insertion of pipeline registers
5. Make the appropriate timing assignments
6. Route the signals to the pins
7. Analyze the timing of the signals
8. Program the device
9. Analyze the data at the pins with an external logic analyzer or oscilloscope

If you want to view different signals at the pin, remove the connections to the pins that you no longer want to examine and repeat from step three.

13.3.2 Internal Logic Analyzer

The internal logic analyzer (ILA) is the tool that has saved the day for many designers. This is the tool that is considered by many as an option in their design flow; until the day when come across a bug in the lab that they cannot find with simulation. They use the ILA to isolate and debug the problem and to verify the fix in system. After this first eye opening scenario, the ILA becomes a key part of their FPGA design flow.

This capability is provided by the major FPGA vendors and some of the EDA tool vendors. The ILA solutions are implemented in the FPGA device using the spare logic and memory resources inside the device.

So what exactly is an Internal Logic Analyzer, or ILA?

Basically it is a tool that is implemented inside the FPGA that provides similar triggering capabilities to the capabilities that is provided by external logic analyzers. ILAs have the advantage that they do not require additional pins to be reserved for debug as they rely on the JTAG interface. They can acquire data on internal signals while the design is running at full speed on an FPGA device at clock speeds exceeding 250 MHz in the latest FPGA technology. However, the performance may vary depending upon the complexity of the trigger conditions being used. They also have the benefit of being able to be used without requiring changes to your design files, as the FPGA vendor software can automatically insert the ILA into the design after the design has been implemented in the FPGA 'without disturbing the implementation of the design.

The captured signal data is stored in device block memory until you are ready to read and analyze the data. In addition, multiple logic analyzers can be implemented in a single device. This provides the benefit of being able to capture data from multiple clock domains in a design at the same time.

So, the question is that if they are so great, why are they not used by all designers?

The answer is quite simply, poor planning. Many designs do not leave sufficient resources in the device to be able to use an ILA. The most common mistake is not leaving adequate memory resources for storing the data for analysis.

As mentioned many times in this book, you must plan for debug up front.

You need to ensure that you have the following in order to use an ILA.

1. A JTAG connection
2. Memory blocks for storing the data for analysis
3. Logic for creating the trigger conditions

Most ILAs come with the following standard feature sets.

Control over the sample depth and the type of RAM that is used to store the data.

Advanced trigger conditions such as state based triggering. This precisely defines upon what conditions the ILA will capture the data.

Continuous storage of data. When the trigger condition occurs, the data that is being tapped is continuously written to memory. This mode of operation can result in the need for large amounts of internal memory in order to prevent data being overwritten.

Transitional storage of data. During acquisition, if any of the signals being tapped have changed since the previous clock cycle, new data is written to the acquisition buffer. If none of the signals being tapped have changed since the previous clock cycle then no data is stored.

Conditional storage. Only stores data if the qualifying condition to write data to memory is true.

The amount of logic and memory that is required to implement the ILA depends upon the complexity of the trigger conditions and the amount of data that needs to be stored.

A useful technique to reduce the amount of logic that is required is to minimize the number of segments in the acquisition buffer to only those required.

Another technique is to use the buffer acquisition control to precisely control the data that is written into the acquisition buffer. This enables you to discard data samples that are not relevant to the debug of your design.

Transitional storage and conditional storage can be used to reduce the amount of internal memory that is required.

13.3.2.1 The Design Flow with an ILA

1. Add an ILA to your design. This can be auto-inserted by the FPGA vendor software without modifying your design or the design implementation in the FPGA
2. Configure the logic analyzer. Define the signals that you want to capture and the storage conditions
3. Define the trigger conditions
4. Compile design
5. Program device
6. Run the ILA application on the host workstation
7. View and analyze captured data

13.3.2.2 ILA Limitations

Not all signals in the design are able to be viewed, or tapped, due to architectural limitations. This includes signals that are part of a carry chain.

You cannot view JTAG signals.

You can only view signals that are available after fitting, unless you want to perform a full design compilation. This can make it difficult to identify combinational signals in the design. This is because RTL synthesis tends to change the names due to the optimizations that are formed during synthesis. These signals can be made available for viewing by using attributes in the RTL to preserve these signals. However, this will change your design implementation. As such it is recommended that you focus your in-system debug on registers, most of which will be available post-fit and not require a full compilation.

13.3.2.3 Tips

Remote Debug

Leave the ILA in the end design. This will enable remote debug of designs in remote locations, if there is JTAG access to the FPGA. This can prove invaluable in debugging designs that are in remote locations or even provides you with the ability to debug designs that are in the lab while you are in your office or at home; this is providing that you have a network connection to the workstation connected to the board.

Interface to MATLAB

Some of the more advanced ILAs provide an interface to the Mathworks MATLAB software. This is a useful option for analyzing DSP data. Once the data has been imported into the MATLAB environment, the view of the data can be displayed in a format suitable to the application being tested.

Insufficient Device Resources

If you are in the position that you have a design that does not leave adequate resources for using an ILA to debug the design, you should strip out functionality from the end design as part of the debug cycle. This enables you to debug isolated blocks in-system, verifying the functionality of these key blocks. This will not enable you to resolve full system integration issues, but will enable you to examine the integration of certain key blocks.

13.3.3 Use of Debug Logic

It is a common and recommended design practice to insert debug logic in the design. This is discussed in Sect. 13.4.3, reporting of system performance.

As mentioned you should build in test logic, monitors and checkers on the interface of major design blocks. The debug logic can be removed after the design is proved to be functionally complete; however leaving the logic in the design provides remote debug capability in the case of in-field failures. If the debug logic is left in the production version of the design, is recommended that the debug logic be disabled and controlled by a pin, JTAG or a soft processor. This will reduce the power consumption in your final design.

Debug logic can also be used with the other debug techniques that are described in this chapter. The addition of a simple multiplexer that interfaces with the debug pins enables the user to more efficiently interface signals that they may want to view to the pins. Which signals are switched through to the pins can be controlled via debug pins that are controlled by the user or via a soft processor in the design. This technique enables fast switching of signals to the debug pins without having to create a new programming file for the FPGA each time that you need to view different signals. This can save you hours of debug time.

The use of debug logic can also be used to force the FPGA into certain conditions, in order to recreate failure conditions or to test the operation under these isolated corner cases.

The main FPGA vendors provide utilities that can help with forcing logic to a particular state via their debug utilities. Using these utilities can reduce the amount of development that you need to do.

Once again these utilities can be combined with other debug capabilities to provide advanced debug solutions. When combined with JTAG it enables you to dynamically control run-time control signals. Similarly it can be combined with ILAs to force the occurrence of trigger conditions setup in the ILA. Through this approach it is possible to create simple test vectors that exercise your design and displays internal signal information without requiring the use of external test equipment.

13.3.4 External Logic Analyzer

The major FPGA vendors provide interfaces to the Logic Analyzers from Agilent and Tektronix. In order to use these optional interfaces in the Logic Analyzers, it requires a JTAG connection and a test header for the Logic Analyzer.

The interface enables viewing of internal signals using an external logic analyzer and using a minimal number of FPGA I/O pins, while the design is running at full speed on the FPGA.

This solution uses a multiplexer, similar to the method described in Sect. 13.3.3 on custom logic, to connect a large set of internal device signals to a small number of output pins.

The multiplexer is JTAG controlled via the user interface of the Logic Analyzer. In addition to controlling the multiplexer, the logic analyzer can display the signal names on the logic analyzer to simplify debug.

This debug approach provides some key advantages over using ILAs.

1. Wider sample depth
2. Ability to handle more data. External Logic Analyzers have much more memory than the amount of memory that is available inside of FPGAs

This debug technique is recommended when you need to store and analyze a large amount of debug data and have room on your board for a test header.

13.3.5 Editing Memory Contents

The contents of the internal memory blocks in your design can be used to force your system into conditions for test and debug. This technique can be extremely effective in testing DSP Applications, such as filters were the memory blocks are used to store coefficients. There are three main approaches to performing this operation.

1. Update the memory initialization files by programming the device with a new image. You can change the memory initialization files without having to recompile the design. You normally only have to run the Assembler to generate the new programming image. This approach works but requires you to bring the FPGA system down in order to change the memory contents
2. The second solution is to generate logic to enable you to write to the internal memory for debug. This is using the technique described in Sect. 13.3.3 on

using logic for debug. This has more flexibility than the previous technique in that you control the writing to the memory blocks while the design is operational. The creation of the logic can be quite complex but the return is invaluable
3. The third technique is to use one of the FPGA vendor supplied solutions that use the JTAG interface to control the writing and reading to the internal memory blocks. This needs to be designed into your system. This means that you will have to replace some of your inferred memories with the primitives from the FPGA vendor. While this offers the simplest and most flexible approach to updating the memory blocks in system, it also comes with some limitations. The biggest limitation being that it does not work with dual port RAM

These techniques work well for other applications outside of DSP applications.

They can be used to test and correct memory parity bits. It can be used to write incorrect parity bit values into the memory to check the ability of your design to handle errors. In addition if you are in the lab and your system is failing due to incorrect parity bits, you can use this technique to correct the errors and to continue the check-out.

This technique can be combined with the other debug techniques that are described in this chapter to provide a very powerful debug arsenal.

13.3.6 Use of a Soft Processor for Debug

Many designers overlook the fact that a processor can be added to your design for the purpose of design debug. The cost of adding a soft processor is 1,000–2,000 Logic Elements, plus internal memory resources.

This is a powerful weapon when combined with custom logic for debug. The processor can take care of controlling the operation of the debug logic or can serve as debug logic itself. It can be easier to describe complex debug trigger conditions, such as state machine trigger conditions, in “C”, rather than in HDL.

The processor can also be used to control the reading and writing to memory. A benefit that it adds beyond the ILA solution is that it can enable the storage of data in external memory, such as DDR III. This enables a larger amount of data to be stored for analysis.

If you are comfortable with coding in “C”, you should consider using a soft processor as one of your debug options,

13.4 Use Scenarios

13.4.1 Power-Up Debug

When the board is first being brought to life, you will want to determine if certain sequences are happening in your design in the correct sequence, to give you confidence that the design can communicate with the rest of the system. In the case were the system does not appear to be operating at power-up, you can use the ILA to capture trigger events that occur during device initialization, immediately after the FPGA is powered on or after the FPGA has been reset. The ILA can then capture data immediately after device programming. This power-up debug feature is available in some of the FPGA vendor ILA solutions.

13.4.2 Debug of Transceiver Interfaces

Just after the board has been powered-up, you will want to determine if the transceiver on the FPGA is operating, i.e. is it capable of transmitting/receiving data from the system.

It is not uncommon that the settings that you have used in your design for the transceiver do not perfectly match the actual board. This scenario can be debugged fairly easily if your transceiver can be dynamically reconfigured, i.e. the settings reprogrammed while the device is operational. Once again the main FPGA vendors provide solutions in this space that can cycle through the settings in the transceiver and report Bit Error Ratio data.

This can be achieved using your existing design if you have built the debug design blocks into the transceiver interface, or you can load the device with one of the debug designs from the FPGA vendor. The latter is the approach that is most commonly used.

These designs consist of Data Pattern Generator and checker blocks along with the dynamic reconfiguration block of the transceiver, which allows modification of the PMA configuration. For the Transmitter, it can change the pre-emphasis settings which affect the eye opening at the receiver end and the Differential Voltage (VOD); which targets different channel medium. On the Receiver, it can change the settings on Equalization and DC gain.

By cycling through the settings and generating and checking data, Bit Error Ratio Testing can be performed on each of the settings. This can serve two main purposes.

1. Analysis of transceiver signal quality
2. Tuning of the transceiver settings to match the board for board bring-up and to mitigate possible signal integrity issues between the transceiver interface and the board

Once the optimal settings have been found they can be applied to the transceiver design in the real design.

13.4.3 Reporting of System Performance

It is likely that you will want to collect system-level statistics on your design to determine if the design is achieving the system performance that you want. The type of data that you may want includes details on the throughput and bandwidth of your system. By identifying the bottlenecks, you can improve the design to meet your throughput and bandwidth requirements. This analysis can be achieved through the use of monitors.

You may want to generate data traffic in order to exercise different transactions in early testing or to isolate corner cases. Normally the system software will take care of this, however early in the board debug, there could be problems with the software or the software may not be ready, so the hardware engineer needs a means to generate traffic to test blocks of the design.

For applications that use specific protocols, you may want to check and report protocol violations. You may want to instrument and analyze the state of the transactions and signals.

These types of data capture, stimulus and reporting are best solved by building verification IP into your design, e.g. monitors that hang off your processor subsystem blocks or protocol checkers that are on your interface IP.

As mentioned previously, by planning for in-system verification, you will hit the ground running when you first receive hardware. If you have been using a standard interface on your design blocks, as recommended in Chap. 9, you will quickly be able to build up a library of verification IP that can be reused on future designs and will easily plug-into your system. It will enable you to use system integration tools, such as Altera's SOPC Builder to drop the verification blocks into your system with minimal design work and impact on the system performance. By having the verification IP available in the final design it will also help in the debug of any systems that fail in the field. The verification IP that you are using can be used with the JTAG control infrastructure, on the FPGAs, to enable you to access/control the data via the JTAG interface.

13.4.4 Debug of Soft Processors

The debug of soft processor designs requires familiarity with multiple disciplines. This complicates the process as it requires the debugging of both the hardware and the application software. The debug of the hardware can be completed using the techniques described previously in this chapter. However it needs to be performed with code running on the processor. Limited debug can be completed using techniques that can force the hardware into known conditions, effectively emulating the operation of the software.

The debug of the software is heavily reliant on the software tool chain that is being used. It is recommended that you read the literature on your soft processor to understand what debug capabilities are available.

In the remainder of this section, we will look at the standard feature set that is available in most software debug tool chains and how they can be used to perform run-time analysis of your design.

13.4.4.1 Software Profiling

Most processor tool chains provide a software profiler. This can be used to provide reports on how long the various functions run in your application. This will identify non-optimal areas of your code that may cause performance issues on your design. You should always profile your software to determine where you need to optimize the software code or potentially accelerate the code via hardware.

13.4.4.2 Watchpoints

The insertion of watchpoints in your code enables the capture of all writes to a global variable. This technique is useful for the debug of a global in the "C" code that appears to be corrupted.

13.4.4.3 Stack Overflow

This technique is applicable to processors that are running a real-time operating system. In this scenario, each task that is running has its own stack. This increases the probability of a stack overflow condition occurring. This type of problem can be more common in FPGA based embedded systems where there is more likely to be restrictions on the amount of memory available for the stack. Most processor IDEs include options to enable runtime stack checking.

13.4.4.4 Breakpoints

Some processor tool chains provide a debug option to set hardware breakpoints on code located in read-only memory such as flash memory. This requires modifying the compilations settings on your code which will result in less optimized code, but code that is much easier to debug.

13.4.4.5 Step Through the Code

By setting the software compiler optimization level to none, you will get software code that runs slower but is much easier to debug as the source code and executable code will now match. This method works well with software breakpoints where the code will run until it hits a breakpoint at which point it will halt. This enables single stepping through the code to examine the values of your variables in order to debug the functionality of the operation.

13.4.5 Device Programming Issues

There is a wealth of JTAG Debug tools from independent Companies and from the FPGA vendors to help you to debug programming issues via JTAG. The most common problem is trying to debug a JTAG chain issue where there are multiple devices from different vendors in the JTAG chain.

The debug tools that come from the FPGA vendors focus on testing the signal integrity of the JTAG chain and to detect intermittent failures of the JTAG chain. The tools check that the devices are connected correctly and provide the ability to run JTAG debug commands.

These tools are excellent for detecting the following type of failures:

1. Open circuits
2. Short to VCC
3. Short to GND

It is recommended that you use a JTAG debug tool on your JTAG chain as soon as you receive your board in house.

13.5 In-System Debug Checklist

1. Plan for debug
   (a) Reserve pins for debug
   (b) Reserve logic and memory resources for ILA use
   (c) Ensure that you use the JTAG interface to the FPGA
   (d) Place a Header on the Board as an interface to an external logic analyzer or scope
   (e) Add debug logic to your design or considering using the FPGA vendor utilities for forcing data to memories and multiplexing data at the pins
   (f) Consider adding a soft processor to your design for debug
2. Perform debug
   (a) Lock down the design implementation using incremental compilation
   (b) For free running data, or for a small handful of control signals, incrementally route the signals to pins for analysis on a logic analyzer or scope
   (c) In order to capture data based upon events, add an ILA to your design. Where possible, use post-fit signal names to avoid a full recompile of the design
3. If there are multiple devices within the JTAG chain, select the device that you want to target
4. Once you have identified the bug, fix the RTL and validate that the fix works with functional simulation

Chapter 14
Design Sign-Off

14.1 Sign-Off Process

There needs to be a process in place to decide at what point to release the design to production. This decision will occur after the design has been fully hardware tested and all of the design and testing processes have been met.

There should be a "GO"/"NO GO" approval process with a management meeting between all of the stake holders in the project. This will review the quality data and decide on whether the design is acceptable for production.

All known bugs should be closed or accepted as not being a gating factor for the release. They should be documented and transferred to the next version of the design for repair.

There needs to be approval for sign-off from all parties and departments.

The sign-off process draws upon the metrics that are captured by the tools described in Chap. 5.

1. The RTL must meet the coding guidelines.
2. The design must meet the functional coverage and code coverage targets.
3. The FPGA project must be free of warnings and any exceptions fully documented.
4. It must meet the timing requirements from the specification.
5. It must meet the in-system debug requirements. In some products, this may involve burn-in testing and full environmental testing.
6. All exceptions to the specification must be fully documented.

14.2 After Sign-Off

After the design has been approved for production, it is necessary to archive the release version and all related design and testing materials. This will serve as the base for any future versions of the design.

The project manager will host a post-project review to discuss what went right, what went wrong, and what was learned from the project. This information will be used in future project plans.

After the well deserved design release party, start working on the next project, which could well be the next version of the design!

Bibliography

Dempster D and Stuart M. Techniques for Verifying HDL Designs. Teamwork International: Hampshire, UK

Bogatin E. Signal Integrity – Simplified. Prentice Hall: Upper Saddle River, NJ

Pellegrin D and Thibault S. Practical FPGA Programming in C. Prentice Hall: Upper Saddle River, NJ

Grotker T, Liao S, Martin G and Swan S. System Design with System C. Kluwer Academic: Dordrecht, The Netherlands

Keating M and Bricaud P. Reuse Methodology Manual for System-on-a-Chip Designs. Springer: New York

Bhasker J. A VHDL Primer. Prentice Hall: Upper Saddle River, NJ

Altera Corporation. Quartus II Handbook v9.0. Altera Corporation: San Jose, CA

Altera Corporation. Altera AN75: High Speed Board Design. Altera Corporation: San Jose, CA

Index