100 Power Tips for FPGA Designers

Evgeni Stavinov

Book Title: 100 Power Tips for FPGA Designers

ISBN 978-1-4507-7598-4

Preface

I've never thought of myself as a book writer. Over the course of my career I've written volumes of technical documentation, published several articles in technical magazines, and have done a lot of technical blogging. At some point I've accumulated a wealth of experience and knowledge in the area of FPGA design, and thought it was a good time to share it with a broader audience.

Writing a book takes time, commitment, and discipline. It also requires a very different skill set. Unfortunately, many engineers, including myself, are trained to use programming languages better than natural languages. Despite all that, writing a book is definitely an intellectually rewarding experience.

I would like to express my gratitude to all the people who have provided valuable ideas, reviewed technical contents, and edited the manuscript: my colleagues from SerialTek, former colleagues from Xilinx, technical bloggers, and many others.

About the Author

Evgeni Stavinov is a longtime FPGA user with more than 10 years of diverse design experience. Before becoming a hardware architect at SerialTek LLC, he held different engineering positions at Xilinx, LeCroy and CATC. Evgeni holds MS and BS degrees in electrical engineering from University of Southern California and Technion - Israel Institute of Technology. Evgeni is a creator of OutputLogic.com, a portal that offers different online productivity tools.

Contents

10. Xilinx ISE Tool Versioning
11. Lesser Known Xilinx Tools
12. Understanding Xilinx Tool Reports

Using Verilog HDL

13. Naming Conventions
14. Verilog Coding Style
15. Writing Synthesizable Code for FPGAs
16. Instantiation vs. Inference
17. Mixed Use of Verilog and VHDL
18. Verilog Versions: Verilog-95, Verilog-2001, and SystemVerilog
19. HDL Code Editors

Design, Synthesis, and Physical Implementation

20. FPGA Clocking Resources
21. Designing a Clocking Scheme
22. Clock Domain Crossing
23. Clock Synchronization Circuits
24. Using FIFOs
25. Counters
26. Signed Arithmetic
27. State machines
28. Using Xilinx DSP48 primitive
29. Reset Scheme
30. Designing Shift Registers
31. Interfacing to external devices
32. Using Look-up Tables and Carry Chains
33. Designing Pipelines
34. Using Embedded Memory
35. Understanding FPGA Bitstream Structure
36. FPGA Configuration
37. FPGA Reconfiguration

FPGA selection

38. Estimating Design Size
39. Estimating Design Speed
40. Estimating FPGA Power Consumption
41. Pin Assignment
42. Thermal Analysis
43. FPGA Cost Estimate
44. GPGPU vs. FPGA

Migrating from ASIC to FPGA

45. ASIC to FPGA Migration Tasks
46. Differences Between ASIC and FPGA Designs
47. Selecting ASIC Emulation or Prototyping Platform
48. Partitioning an ASIC Design into Multiple FPGAs
49. Porting Clocks
50. Porting Latches
51. Porting Combinatorial Circuits
52. Porting Non-synthesizable Circuits
53. Modeling Memories
54. Porting Tri-state Logic
55. Verification of a Ported Design

Design Simulation and Verification

56. FPGA Design Verification
57. Simulation Types
58. Improving Simulation Performance
59. Simulation and Synthesis Results Mismatch
60. Simulator Selection
61. Overview of Commercial and Open-source Simulators
62. Designing Simulation Testbenches
63. Simulation Best Practices
64. Measuring Simulation Performance

IP Cores and Functional Blocks

65. Overview of FPGA-based Processors
66. Ethernet Cores
67. Designing Network Applications
68. IP Core Selection
69. IP Core Protection
70. IP Core Interfaces
71. Serial and Parallel CRC
72. Scramblers, PRBS, and MISR
73. Security Cores
74. Memory Controllers
75. USB Cores
76. PCI Express Cores
77. Miscellaneous IP Cores and Functional Blocks

Design Optimizations

78. Improving FPGA Build Time
79. Design Area Optimizations: Tool Options
80. Design Area Optimizations: Coding Style

1. Introduction

Target audience

FPGA logic design has grown from being one of many hardware engineering skills a decade ago to a highly specialized field. Nowadays, FPGA logic design is a full time job. It requires a broad range of skills, such as a deep knowledge of FPGA design tools, the ability to understand FPGA architecture and sound digital logic design practices. It can take years of training and experience to master those skills in order to be able to design complex FPGA projects.

This book is intended for electrical engineers and students who want to improve their FPGA design skills. Both novice and seasoned logic and hardware engineers can find bits of useful information in this book. It is intended to augment, not replace, existing FPGA documentation, such as user manuals, datasheets, and user guides. It provides useful and practical design “tips and tricks,” and little known facts that are hard to find elsewhere.

The book is intended to be very practical with a lot of illustrations, code examples and scripts. Rather than having a generic discussion applicable to all FPGA vendors, this edition of the book focuses on Xilinx FPGAs. Code examples are written in Verilog HDL. This will enable more concrete examples and in-depth discussions. Most of the examples are simple enough, and can be easily ported to other FPGA vendors and families, and VHDL language.

The book provides an extensive collection of useful online references.

It is assumed that the reader has some digital design background, and working knowledge of ASIC or FPGA logic design using Verilog HDL.

How to read this book

The book is organized as a collection of short articles, or Tips, on various aspects of FPGA design: synthesis, simulation, porting ASIC designs, floorplanning and timing closure, design methodologies, design optimizations, RTL coding, IP core selection, and many others.

This book is intended for both referencing and browsing. The Tips are organized by topic, such as “Efficient use of Xilinx FPGA design tools,” but it is not arranged in a perfect order. There is little dependency between Tips. The reader is not expected to read the book from cover to cover. Instead, you can browse to the topic that interests you at any time.

This book is not a definitive guide into Verilog programming language, digital design or FPGA tools and architecture. Neither does it attempt to provide deep coverage of a wide range of topics in a limited space. Instead, it covers the important points, and provides references for further exploration of that topic. Some of the material in this book has appeared previously as more complete articles in technical magazines.

Software

The FPGA synthesis and simulation software used in this book is a free Web edition of Xilinx ISE package.

Companion web site

An accompanying web site for this book is:
http://outputlogic.com/100_fpga_power_tips

It provides most of the projects, source code, and scripts mentioned in the book. It also contains links to referenced materials, and errata.

2. FPGA Landscape

FPGA landscape is dominated by two main players: Xilinx and Altera. The two hold over 90% of the market share. Smaller FPGA vendors are Lattice Semiconductor and Actel, which provide more specialized FPGA features. All four are publicly traded companies.

Achronix and Tabula are new startup companies that offer unique FPGA features.

The following is a list of FPGA vendors mentioned above along with the key financial and product information. Financial information is provided courtesy of Yahoo! Finance, January 2011.

Xilinx Inc.
URL: http://www.xilinx.com
Market capitalization: $8.52B
Revenue (ttm): $2.31B
Number of full time employees: 2,948
Year founded: 1984
Headquarters: San Jose, California

Xilinx offers several FPGA families with a wide range of capabilities. Two newest families are Virtex-6 and Spartan-6.
Virtex-6 family offers the industry’s highest performance and capacity. It’s ideal for high speed, high density applications.
Spartan-6 provides a solution for cost-sensitive applications, where size, power, and cost are key considerations.

Altera Corporation
URL: http://www.altera.com
Market capitalization: $12.03B
Revenue (ttm): $1.76B
Number of employees: 2,551
Year founded: 1983
Headquarters: San Jose, California

Altera offers the following FPGA families: Stratix, Arria®, and Cyclone.
Stratix-V is the latest addition to the Stratix family. It offers the highest performance and density.
Arria is a family of mid-range FPGAs targeting power sensitive and transceiver based applications.
Cyclone offers low-cost FPGAs for low-power cost-sensitive applications.

Actel/Microsemi Corporation
URL: http://www.actel.com
Market capitalization: $1.98B
Revenue (ttm): $518M
Number of employees: 2,250
Year founded: Actel was acquired by Microsemi Corp. in October 2010
Headquarters: Irvine, California

Actel FPGAs feature the lowest power usage and widest range of small packages.

Actel offers the following FPGA families:

- IGLOO, ProASIC3: low-power small footprint FPGAs
- SmartFusion: mixed-signal FPGA with integrated ARM processor
- RTAX/RTSX: high-reliability radiation-tolerant FPGA family

Lattice Semiconductor Corporation
URL: http://www.latticesemi.com
Market capitalization: $700M
Revenue (ttm): $280M
Number of employees: 708
Year founded: 1983
Headquarters: Hillsboro, Oregon

Lattice offers the following FPGA families:

- LatticeECP3: low power, SerDes-capable FPGAs
- LatticeECP2: low cost FPGAs
- LatticeSC: high-performance FPGAs
- MachX: non-volatile FPGAs with embedded Flash

Achronix Semiconductor
URL: http://www.achronix.com
Headquarters: Santa Clara, California

Achronix is a privately held company that builds the world's fastest FPGAs, providing 1.5 GHz throughput, which is a significant performance advantage over traditional FPGA technology.

Tabula
URL: http://www.tabula.com
Headquarters: Santa Clara, California

Tabula is a privately held company. It developed a unique FPGA technology, called Spacetime, which provides significantly higher logic, memory, and signal processing capabilities than traditional FPGAs.

3. FPGA Applications

A few years ago, FPGAs were mainly used as “glue logic” devices between other electronic components, bus controllers, or simple protocol processors. This is not the case anymore. As FPGAs become larger and faster, the cost per logic gate drops. That enabled FPGA applications unthinkable before, possible. Modern FPGAs became viable ASIC replacement and are now found in consumer electronics devices. Although it’s a common misconception that FPGAs are too expensive for high-volume products, this is not entirely true.

FPGAs became reliable enough to be used in mission critical applications, such as in space, military, automotive, and medical appliances. FPGAs are taking over as a platform of choice in the implementation of high-performance signal processing applications, replacing multiple dedicated DSP processors.

FPGA power consumption became low enough to be even used in battery-powered devices such as camcorders and cell phones.

FPGAs have a consistent track record of capacity growth, and they track Moore’s law better than any other semiconductor device. As an example, XC2064 FPGA, introduced by Xilinx in 1985, had 1024 logic gates. The largest, Xilinx Virtex-7 XC7V2000T, has 15,636,480 gates, an increase of 10,000 times over the past 26 years. If the current trend of capacity and speed increase continues, FPGAs will essentially become highly integrated SoC platforms, which include multi-core processors, wide range of peripherals, and high capacity logic fabric, all in one chip.

Tighter integration of FPGA fabric and embedded processors will in turn raise the level of abstraction. It’ll require less effort to develop a novel software application that includes an FPGA as one of its components. Instead of using multiple, off-the-shelf hardware components, such FPGA SoC can be fine-tuned to optimize the performance of a particular software application.

Another trend is migration of FPGA development tools to popular open-source frameworks. That will make the FPGA development more affordable and enhance interoperability among FPGA and other design tool vendors. Xilinx is already using an open-source Eclipse framework in its EDK.

There is a lot of interest in the industry and academy to raise the level of abstraction of the FPGA synthesis. Currently, most of the FPGA designs are hand-written in low-level hardware description languages (HDL). This approach delivers poor productivity, and there is a lot of customer demand for better tools that support high-level synthesis. There are several promising high-level synthesis tools and technologies, but this hasn’t reached the mainstream yet. Two examples are Catapult-C Synthesis by Mentor Graphics, and AutoPilot by AutoESL Design Technologies (acquired by Xilinx in Jan. 2011).

Virtualization of tools, platforms, and hardware resources is on the cusp of wide acceptance. Virtualization allows better resource utilization by sharing. For example, pooling multiple development boards into a board farm, instead of assigning each board to a developer, will increase effective utilization of a board, and make the maintenance more efficient. Perhaps it’s too early to tell, but virtualization has a potential to introduce new interesting and unexpected FPGA applications.

Virtualization is a key enabling technology for cloud computing environment. Cloud computing is defined as a model for enabling

convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction. Cloud computing augments the power of virtualization by adding on-demand self service, scalability, and resource pooling. At the time of this book's writing, there are two companies that offer cloud-based FPGA development tool products: National Instruments, and Xuropa.
An overview of typical FPGA applications by end market is shown in the following table.

Table 1: FPGA end markets and applications

End market	*Application*
Wireless communications	Cellular and WiMax base stations
Wireline communications	High speed switches and routers, DSL multiplexers
Telecom	Optical and radio transmission equipment, telephony switches, traffic managers, backplane transceivers
Consumer electronics	LCD TV, DVR, Set Top boxes, high end cameras
Video and image processing	Video surveillance systems, broadcast video, JPEG, MPEG decoders
Automotive	GPS systems, car infotainment systems, driver assistance systems: car parking assistance, car threat avoidance, back-up aid, adaptive cruise control, blind spot detection
Aerospace and Defense	Radar and sonar systems, satellite communications, radiation-tolerant space applications
ASIC prototyping	FPGA-based ASIC emulation and prototyping platforms
Embedded systems	System On Chip (SoC)
Test and Measurement	Logic Analyzers, Protocol Analyzers, Oscilloscopes
Storage	High-end controllers, servers
Data security	Data encryption: AES, 3DES algorithms, public key cryptography (RSA), Data integrity (SHA1, MD5)
Medical	Medical imaging
High performance and scientific computing	Acceleration of parallel algorithms, matrix multiplication, Black-Scholes algorithm for option pricing used in finance
Custom designs	Hardware accelerators that offload computing-intensive tasks from the embedded processors: checksum offload, encryption/decryption, parts of image and video processing algorithms, complex pattern matching.

4. FPGA Architecture

The key to successful design is a good understanding of the underlying FPGA architecture, capabilities, available resources, and just as important - the limitations. This Tip uses Xilinx Virtex-6 family as an example to provide a brief overview of the architecture of a modern FPGA.

The main architectural components, as illustrated in the following figure, are logic and IO blocks, interconnect matrices, clocking resources, embedded memories, routing, and configuration logic.

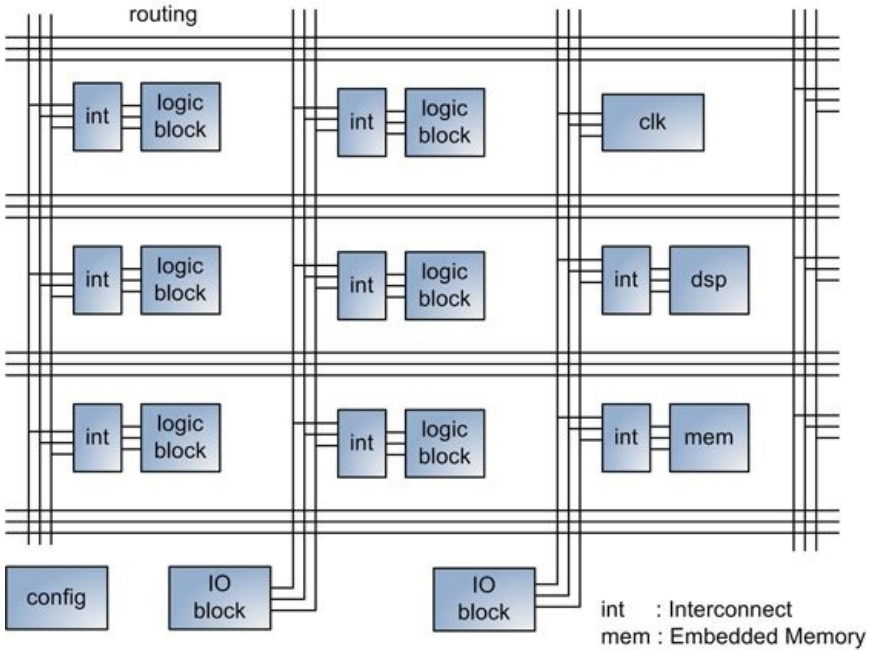

Figure 1: FPGA architecture

Many high-end FPGAs also include complex functional modules such as memory controllers, high speed serializer/deserializer transceivers, integrated PCI Express interface, and Ethernet MAC blocks.

The combination of FPGA logic and routing resources is frequently called FPGA fabric.

The term derives its name from its topological representation. As the routing between logic blocks and other resources is drawn out, the lines cross so densely that it resembles a fabric.

Logic blocks

Logic block is a generic term for a circuit that implements various logic functions. A logic block in Xilinx FPGAs is called Slice. A Slice in Virtex-6 FPGA contains four look-up tables (LUTs), eight registers, a carry chain, and multiplexers. The following figure shows main components of a Virtex-6 FPGA Slice.

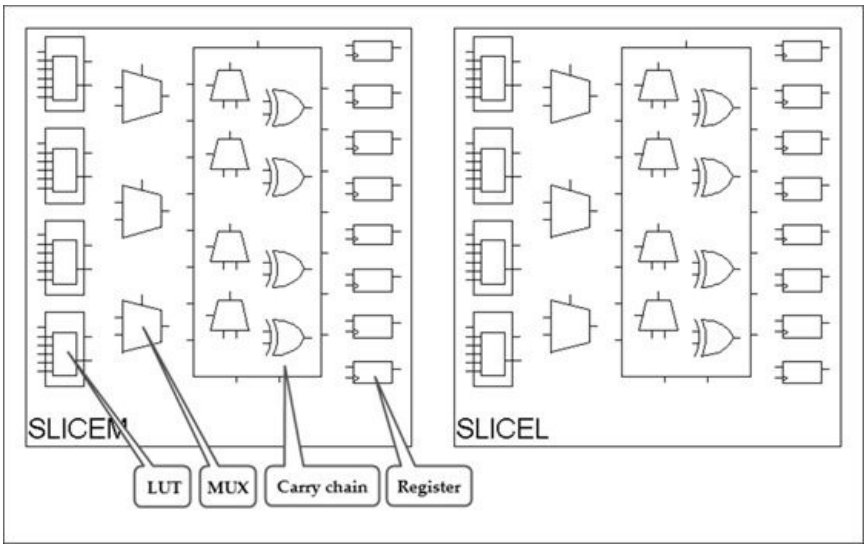

Figure 2: Xilinx Virtex-6 FPGA Slice structure

The connectivity between LUTs, registers, multiplexers, and a carry chain can be configured to form different logic circuits.

There are two different Slice types: SLICEM and SLICEL. A SLICEM has a multi-purpose LUT, which can also be configured as a Shift Register LUT (SRL), or a 64- or 32-bit read-only or random access memory.

Each Slice register can be configured as a latch.

Clocking resources

Each Virtex-6 FPGA provides several highly configurable mixed-mode clock managers (MMCMs), which are used for frequency synthesis and phase shifting.

Clocks to different synchronous elements across FPGA are distributed using dedicated low-skew and low-delay clock routing resources. Clock lines can be driven by global clock buffers, which allow glitchless clock multiplexing and the clock enable.

More detailed discussion of Xilinx FPGA clocking resources is provided in Tip #20.

Embedded memory

Xilinx FPGAs have two types of embedded memories: a dedicated Block RAM (BRAM) primitive, and a LUT configured as Distributed RAM

Virtex-6 BRAM can store 36K bits, and can be configured as a single- or dual-ported RAM. Other configuration options include data width of up to 36-bit, memory depth up to 32K entries, and error detection and correction.

Tip #34 describes different use cases of FPGA-embedded memory.

DSP

Virtex-6 FPGAs provide dedicated Digital Signal Processing (DSP) primitives to implement various functions used in DSP applications, such as multipliers, accumulators, and signed arithmetic operations. The main advantage of using DSP primitives instead of general-purpose LUTs and registers is high performance.

Tip #28 describes different use cases of DSP primitive.

Input/Output

Input/Output (IO) block enables different IO pin configurations: IO standards, single-ended or differential, slew rate and the output strength, pull-up or pull-down resistor, digitally controlled impedance (DCI). An IO in Virtex-6 can be delayed by up to 32 increments of 78 ps each by using an IODELAY primitive.

Serializer/Deserializer

Most of Virtex-6 FPGAs include dedicated transceiver blocks that implement Serializer/Deserializer (SerDes) circuits. Transceivers can operate at a data rate between 155 Mb/s and 11.18 Gb/s, depending on the configuration.

Routing resources

FPGA routing resources provide programmable connectivity between logic blocks, IOs, embedded memory, DSP, and other modules. Routing resources are arranged in a horizontal and vertical grid. A special interconnect module serves as a configurable switch box to connect logic blocks, IOs, DSP, and other module to horizontal and vertical routing. Unfortunately, Xilinx doesn't provide much documentation on performance characteristics, implementation details, and quantity of the routing resources. Some routing performance characteristics can be obtained by analyzing timing reports. And the FPGA Editor tool can be used to glean information about the routing quantity and structure.

FPGA configuration

The majority of modern FPGAs are SRAM-based, including Xilinx Spartan and Virtex families. On each FPGA power-up, or during a subsequent FPGA reconfiguration, a bitstream is read from the external non-volatile memory (NVM), processed by the configuration controller, and loaded to the internal configuration SRAM. Tips #35-37 describe the process of FPGA configuration and bitstream structure in more detail.

The following table summarizes this Tip by showing key features of the smallest, mid-range, and largest Xilinx Virtex-6 FPGA.

Table 1: Xilinx Virtex-6 FPGA key features

	XC6 VLX75T	***XC6 VLX240T***	***XC6 VLX760***
Logic cells	74,496	241,152	758,784
Embedded memory (Kbyte)	832	2,328	4,275
DSP modules	288	768	864
User IOs	240	600	1200

5. FPGA Project Tasks

Any new project that includes FPGAs has a typical list of tasks that a development team encounters during the project's lifetime, which range from the product's inception until getting a working FPGA design out of the door. There are two types of tasks: project setup and design/development.

Project preparation and setup include project requirements, architecture documentation, FPGA selection, tools selection, and

assembling a design team.

Design and development tasks include pin assignment, developing RTL, design simulation and verification, synthesis and physical implementation, floorplanning and timing closure, and board bring-up.

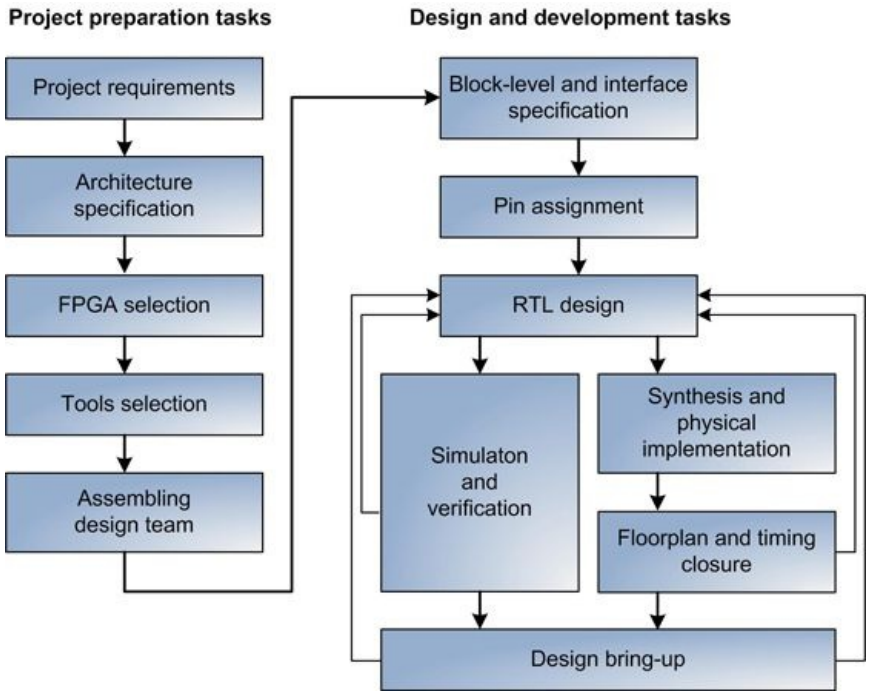

Figure 1: FPGA project tasks

Project preparation and setup tasks

Product and project requirements

It is hard to overestimate the importance of having properly documented product and project requirements. It puts all of the project/product stakeholders on the same page and eliminates ambiguity and second guessing later in the project life cycle. Having proper requirements documentation applies to commercial, research, and even student projects.

Typically, product requirements are driven by the sales and marketing team, and the transition to engineering project is handled by engineering managers and architects.

The level of detail varies. In a small startup it can be a document of a few pages, or even an email. For a large multi-FPGA product, such as an ASIC prototyping platform, military communication system, or high-end switch, the document is hundreds of pages long and written by several people.

Architecture specification

Architecture specification is based upon project specification. It is typically written by the most experienced engineer in a team, the system architect, who has broad experience in hardware, logic, and software design, is well-versed in writing specifications, and has good interpersonal skills. This engineer interacts closely with the sales and marketing team to clarify and adjust the requirements if needed.

Architecture specification addresses issues such as the technical feasibility of the project, top-level block diagram (which describes the partition between different components), main interfaces, and possible implementation options.

It is important to perform feasibility analysis at this stage to find that the project requirements are realistic and attainable and can meet performance, cost, size, and other goals.

An example of unrealistic requirement is a required processing speed of a deep-packet inspection engine that cannot be done in real time using the latest FPGAs. One solution is to plan using the next FPGA family that will be ready for production by the time the project enters the debug phase. Engineering samples might be used in the interim.

Software, hardware, and logic top-level block diagrams should include main interfaces. The interfaces have to be accurate, unambiguous, and interpreted the same way by all team members. The specification also includes overall partitioning into multiple FPGAs and interfaces between them.

Architecture specification is a living document. It has to be maintained and kept up-to-date. Engineering team members refer to the document to write more detailed block-level documents and to do design work. Sales and marketing teams use it to establish bill of material (BOM) and final product cost.

FPGA selection

FPGA selection process involves selecting FPGA vendor, family, capacity, and speed grade. It is not uncommon that a multi-FPGA project contains FPGAs from different vendors and families. It is typically not an issue to interface between heterogeneous FPGAs as soon as they meet project requirements.

Selecting FPGAs from different vendors complicates some of the project management tasks, such as having multiple tool flows and purchasing from different distributors. One of the reasons to select FPGAs from different vendors is that each vendor offers unique features. Examples are integrated memory controllers, SerDes that can operate at certain speed, logic capacity, or number of user IOs.

FPGA capacity estimate

FPGA capacity estimate, general guidelines and criteria for ASIC designs are discussed in more detail in Tip #38.

The most reliable way to do size estimate is to build a final RTL implementation. In some cases this is possible, such as in projects that require migration of an existing RTL base to the latest FPGA with improved characteristics. However, in most cases this is wishful thinking because most of the design is not yet written. The only remaining option is to estimate the size of each sub-module and add them up. It is possible that some of the modules are reused from the previous designs. Also, third party IP cores often include capacity utilization data. The rest of the modules can be prototyped.

A frequently made mistake is to determine only logic utilization in terms of slices, LUTs, and FFs. Other FPGA resources are overlooked, which can cause problems later on. Size estimate should include all FPGA resource types used in the design: embedded memories, DSP, IOs, clocks, PLL, and specialized cores such as PCI Express interface and Ethernet MAC.

FPGA speed analysis

Correct analysis of both FPGA fabric and IO speed is important for a couple of reasons. Maximum design speed affects selection of FPGA speed grade, which in turn affects the cost. Another reason is design feasibility.

A high speed network switch or a protocol processor can require running FPGA fabric at 300+ MHz frequencies, and is close to the limit of that FPGA family. Prototyping a small part of that design might meet timing. However, the complete design might not, because other factors, such as routing delays between modules, come into play. This is a common pitfall of using this method.

Another example is ASIC emulation/prototyping platform, which has different design characteristics. ASIC emulation designs are typically LUT-dominated, and run at low, sub 100MHz frequencies. However, if such a platform contains several FPGAs, the IO speed becomes the bottleneck that directly affects overall emulation frequency.

Higher design speed causes longer build times and more effort to achieve timing closure. Therefore, the goal is to use the lowest possible speed for the design. This will bring material and development costs down.

FPGA power analysis

An increasing number of FPGAs is used in power-sensitive applications. Power estimate can affect the FPGA selection, or even determine whether it's feasible to use an FPGA in a product at all.

An accurate power estimate requires a lot of design knowledge: logic capacity, toggle rate, and exact IO characteristics.

Tip #81 has a more detailed discussion on low-power optimization techniques.

Tools selection

Tools selection is largely determined by the FPGA vendor. There isn't much flexibility in choosing design tools; typically, there is a well-established vendor specific tool ecosystem. Some of the tools are described in Tip #6.

Assembling a design team

It is no longer practical for a single engineer to handle all project design tasks. Even a small FPGA design project requires a wide spectrum of expertise and experience levels from team members.

Some design tasks require a deep domain-level expertise, such as protocol or algorithm understanding. These tasks are carried out by engineers with advanced degrees in a particular field and little FPGA understanding. Other tasks require a lot of logic design experience to do efficient RTL implementation. Tool knowledge is required to do synthesis, floorplanning, and timing closure.

Another trend is workforce globalization. Members of the same team might reside in different locations around the globe. This adds a lot of communication overhead and makes collaboration and efficient project management more difficult.

Assembling a team and delegating the right tasks to each team member is essential for determining accurate and reliable project schedule.

FPGA design and development tasks

Detailed block-level and interface documentation

Detailed block-level and interface documentation is a task that is often done haphazardly, especially in small companies without well-established procedures.

Interface specification includes register description and access procedures between FPGA and software.

Block level specification describes interfaces, such as busses, between FPGAs and modules within an FPGA.

Pin assignment

FPGA pin assignment is a collaborative process of PCB and FPGA logic design teams to assign all top-level ports of a design to FPGA IOs. It usually involves several back-and-forth iterations between the two teams until both are satisfied with the results.

PCB designers are concerned with board routability, minimizing the number of board layers, signal integrity, the number of board components, bank assignment for power management scheme, and clocking.

FPGA logic designers are mainly concerned with IO layout and characteristics. IO layout affects floorplanning and timing closure. Certain IOs, such as high speed busses, have to be grouped together. Clocks have to be connected to clock-capable pins. Tristate and bidirectional pins have to be taken care of.

At the end of the pin assignment process the design has to pass all tool DRC checks, such as correct bank assignment rules and simultaneous switching output (SSO). It is difficult to observe all the individual rules. Using the Xilinx PinAhead tool is indispensable to streamline this task.

In most cases the logic designer will need to create a dummy design to prototype the right IO characteristics, because the real design is not available yet.

Pin assignment requires a significant amount of time and attention to detail. If not done correctly, it can cause a lot of problems at the later project stage. It might require board layout change and costly respin. The importance of this task is often underestimated, even by experienced designers.

RTL design

RTL design is the part of the project that usually consumes the most time and resources.

There are several sources that can cause RTL design change:

- Incorrect simulation
- Issues during synthesis and physical implementation
- Floorplanning and timing closure
- Problems during design bring-up

Simulation and verification

RTL code, whether it's Verilog or VHDL, is written for synthesis targeting a particular FPGA. It is not sufficient to pass all simulation test cases and assume that the code is ready. It is not, because the code might change as a result of various issues during synthesis, physical implementation, floorplanning, and timing closure stages.

This kind of problem often happens with inexperienced logic designers and incorrectly established design flows. Developers spend countless hours doing simulation of a module. When that module is put through the synthesis, numerous errors and warnings are uncovered, making all previous simulation results invalid.

A better design flow is to concurrently run newly developed RTL code through synthesis and simulation, and to fix all errors and critical warnings in the process. It is expected that this step will take several iterations.

Therefore, doing simulation and synthesis in parallel will lead to better designer time utilization and will improve the project's overall progress.

Synthesis and physical implementation

Synthesis and physical implementation are two separate steps during FPGA build. Synthesis can be done using Xilinx XST, as well as third party synthesis tools. Synthesis produces a design netlist. Physical implementation is done using Xilinx tools and produces a bitstream ready to be programmed into an FPGA.

In larger design teams, different team members might be involved in synthesis, physical implementation, and other build tasks, such as developing automation scripts.

Floorplanning and timing closure

Floorplanning and timing closure are the most unpredictable tasks in terms of correctly estimating completion time. Even the most experienced FPGA logic designers underestimate the amount of time and effort it takes to achieve timing closure. It might take

dozens of iterations, and weeks of development time, to make floorplanning and RTL changes to close timing in a large design.

Design bring-up

Design bring-up on a hardware platform is the final stage before a design is ready for release.

Potential issues arise due to simulation and synthesis mismatch. This is discussed in Tip #59.

Post-mortem analysis

Many design teams perform a formal project post-mortem analysis upon project completion. It is used to identify root causes for different problems, such as missed deadlines or unforeseen technical issues. The goal of the post-mortem is to learn a lesson and design a plan to prevent such problems from occurring in the future.

6. Overview Of FPGA Design Tools

Over the course of FPGA design cycle, engineers use variety of tools: simulation, synthesis, physical implementation, debug and verification, and many other. This Tip provides a brief overview of the most commonly used tools.

Simulators

Tool: **ISIM**
Company: Xilinx
http://www.xilinx.com/support/documentation/plugin_ism.pdf

Xilinx ISIM is an excellent choice for small to medium size FPGA designs. It's integrated with Xilinx ISE tools, and it is free.

The biggest ISIM disadvantage is that it doesn't scale well for larger designs. Comparing it to other commercial simulators, ISIM is much slower and requires more memory.

ISIM also offers a co-simulation option, which requires a separate license.

Tool: **ModelSim PE, DE, SE**
Company: Mentor Graphics
http://model.com

ModelSim simulator is the most popular choice for FPGA design simulation. It comes in three versions: PE, DE, and SE.

Tool: **VCS**
Company: Synopsys
http://www.synopsys.com/Tools/FunctionalVerification/Pages/VCS.aspx

VCS simulator is at the high end of the spectrum. It is the fastest, but the most expensive simulation tool. It is mainly used for doing functional simulation of ASIC designs, but is often used in large FPGA designs as well.

Tool: **NCSim**
Company: Cadence
http://www.cadence.com/products/ld/design_team_simulator/pages/default.aspx

NCSim is a core simulation engine, and part of Incisive® suite of tools. It is intended for design and verification of ASICs and FPGAs.

Tool: **Active-HDL, Riviera**
Company: Aldec
http://www.aldec.com/Products/default.aspx

Active-HDL and Riviera are tools for FPGA and ASIC functional simulation and verification.

Tool: **Icarus Verilog**
http://bleyer.org/icarus

Icarus Verilog is an open source compiler implementation for Verilog HDL. Icarus is maintained by Stephen Williams and it is released under the GNU GPL license.

Tool: **Verilator**
http://www.veripool.org/wiki/verilator

Verilator is an open source Verilog HDL simulator written and maintained by Wilson Snyder. Its main feature is simulation speed.

Simulation tools are reviewed in more detail in Tip #61.

Synthesis tools

Synthesis tool is an application that synthesizes Hardware Description Language (HDL) designs to create a netlist. There are several synthesis tools for Xilinx FPGAs: Xilinx XST, Synopsys Synplify, Mentor Precision, and several others.

Xilinx XST produces a netlist in a proprietary NGC format, which contains both logical design data and constraints. Other synthesis tools produce a netlist in an industry standard EDIF format.

Tool: **XST**
Company: Xilinx
http://www.xilinx.com/itp/xilinx10/books/docs/xst/xst.pdf

Tool: **Synplify Pro, Synplify Premier**
Company: Synopsys
http://www.synopsys.com/tools/synplifypro.aspx

Tool: **Precision RTL**
Company: Mentor Graphics
http://www.mentor.com/products/fpga/synthesis/precision_rtl

Tool: **zFAST**
Company: EvE
http://www.eve-team.com/products/zfast.html

EvE zFAST is a synthesis tool dedicated for EvE ZeBu ASIC emulation platforms that use Xilinx FPGAs. The main feature of zFAST is the execution speed. For large ASIC designs it is an important requirement to be able to quickly produce a netlist.

Synopsys Synplify and Mentor Precision are synthesis tools used for general-purpose FPGA designs.

Providing a detailed comparison between synthesis tools is beyond the scope of this book. Each tool offers its unique advantages: faster execution speed, more compact netlist, better support for HDL language constructs, unique optimization features, modular design flows, and many others. It is recommended that you evaluate different synthesis tools before starting a new project.

Physical implementation tools

FPGA physical implementation tools are provided by the FPGA vendor itself. Xilinx FPGA physical implementation requires *ngdbuild, map, par,* and *bitgen* tools. Optional tools to perform different format conversions and report analysis are *trce, netgen, edif2ngd, xdl,* and many others. Those tools are integrated into ISE and PlanAhead GUI environment, and can also be accessed from the command line. The tools are installed in $XILINX/ISE/bin/{nt, nt64, lin, lin64} and $XILINX/common/bin/{nt, nt64, lin, lin64} directories. $XILINX is an environmental variable that points to the ISE installation directory.

Design debug and verification

Tool: **RocketDrive, RocketVision**
Company: GateRocket
http://www.gaterocket.com

GateRocket offers FPGA verification solutions. RocketDrive is a peripheral device that accelerates software-based HDL simulation. RocketVision is companion software package that provides advanced debugging capabilities.

Tool: **Identify**
Company: Synopsys
http://www.synopsys.com/iools/identify.aspx

The Identify RTL debugger allows users to instrument their RTL and debug implemented FPGA, still at the RTL level, on running hardware.

Tool: **ChipScope**
Company: Xilinx
http://www.xilinx.com/tools/cspro.htm

Lint tools

Lint tools perform automated pre-simulation and pre-synthesis RTL design rule checking and analysis. They help uncover complex and obscure problems that cannot be found by simulation and synthesis tools. Examples of design rules checked by lint tools are: clock domain crossing, combinatorial loops, module connectivity, coding style, implied latches, asynchronous resets, and many others.

Tool: **nLint**
Company: SpringSoft
http://www.springsoft.com

nLint is part of the Novas verification environment.

Tool: **vlint**
Company: Veritools
http://www.veritools.com

Basic and Advance Lint tools are integrated into Riviera-PRO tools from Aldec.

ModelSim simulator offers limited lint capability enabled by “-lint” command line option.

7. Xilinx FPGA Build Process

FPGA build process refers to a sequence of steps to build an FPGA design from a generic RTL design description to a bitstream. The exact build sequence will differ, depending on the tool used. However, any Xilinx FPGA build will contain eight fundamental steps: pre-build, synthesis, ngdbuild, map, place-and-route, static timing analysis, bitgen and post-build.

Ngdbuild, map, place-and-route, and bitgen steps are often referred as FPGA physical implementation, because they depend on the specific FPGA architecture. At the time of writing this book, there are no commercially available third party physical implementation tools for Xilinx FPGAs.

Xilinx provides two IDE tools to perform an FPGA build: ISE, and PlanAhead. There are several other third party IDE tools that can do FPGA builds: Synopsys Synplify, Aldec Riviera, and Mentor Graphics Precision.

The following figure illustrates Xilinx FPGA build flow.

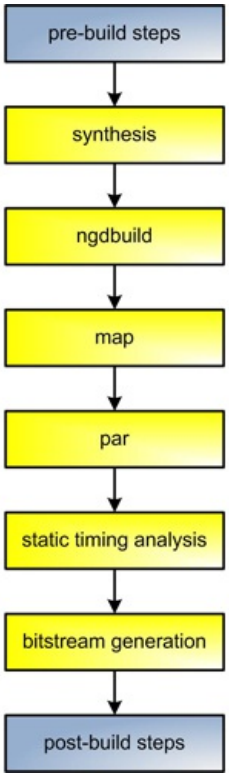

Figure 1: Xilinx FPGA build flow

Build preparation tasks

Build preparation tasks greatly depend on the complexity of the project and the build flow. The tasks might include the following:

- Getting the latest project and RTL file from a source control repository
- Assembling the project file list
- Setting environment variables for synthesis and physical implementation tools
- Acquiring tool licenses
- Incrementing build or revision number in the RTL code
- Replacing macros and defines in the RTL code

Synthesis

The simplest and most generic way to describe a synthesis is as a process of converting a design written in a Hardware Description Language (HDL) into a netlist. Xilinx Synthesis Technology (XST) synthesis tool produces a netlist in a proprietary NGC format, which contains both logical design data and constraints. Other synthesis tools produce a netlist in an industry standard EDIF format. Different synthesis tools are reviewed in Tip #6.

The following is an example of calling Xilinx XST:

```
$ xst -intstyle ise -ifn "/proj/crc.xst" -ofn "/proj/crc.syr"
```

XST contains over a hundred different options, not discussed in this Tip for brevity reasons. For the complete list of XST options refer to the Xilinx XST User Guide.

Netlist Translation

Xilinx NGDBUILD tool performs netlist translation into Xilinx Native Generic Database (NGD) file that contains a description of the design in terms of basic logic elements. Required NGDBUILD parameter is design name. Optional parameters are user-constraint files in UCF format, destination directory, FPGA part number, and several others.

The following is an example of calling NGDBUILD:

```
$ ngdbuild -dd _ngo -nt timestamp -uc  /ucf/crc.ucf -p xc6slx9-csg225-3 crc.ngc crc.ngd
```

MAP

Xilinx MAP tool performs mapping of a logical design into Xilinx FPGA. The output from MAP is a Native Circuit Description (NCD) file, which is a physical representation of the design mapped to the components of a specific Xilinx FPGA part. MAP has over thirty options described in the Command Line Tools User Guide. Some of the most frequently used are –p (part number), -ol (overall effort level), and -t (placer cost table). MAP has several options used during timing closure, area and performance optimizations, and discussed in more detail in other Tips.

The following is an example of calling MAP:

```
map -p xc6slx9-csg225-3 -w -ol high -t 1 -xt 0 -global_opt off -lc off -o crc_map.ncd crc.ngd crc.pcf
```

Place and Route (PAR)

A design place and route is performed using Xilinx PAR tool. PAR outputs an NCD file that contains complete place and route information about the design. Note that this is the same file type as produced by MAP. Some of the most frequently PAR options are –p (part number), -ol (overall effort level), and -t (placer cost table). Command Line Tools User Guide describes PAR options.

The following is an example of calling PAR:

```
par -w -ol high crc_map.ncd crc.ncd crc.pcf
```

Static Timing Analysis

Static timing analysis is performed using the Xilinx TRACE tool, described in Tip #91.

Bitstream generation

Bitgen is a Xilinx tool that creates a bitstream to be used for FPGA configuration. The following is a simplest example of calling Bitgen:

```
$ bitgen -f crc.ut crc.ncd
```

Bitgen has over a hundred command line options that perform various FPGA configurations, and is described in Command Line Tools User Guide.

Post-build tasks

After the FPGA bitstream is generated, the build flow might contain the following tasks:

- Parsing build reports to determine if the build is successful or not
- Copying and archiving bitstream and intermediate build files
- Sending email notifications to subscribed users

8. Using Xilinx Tools In Command-line Mode

The majority of designers working with Xilinx FPGAs use ISE Project Navigator and PlanAhead software in graphical user interface mode. The GUI approach provides a pushbutton flow, which is convenient for small projects.

However, as FPGAs become larger, so do the designs built around them, and the design teams themselves. In many cases, GUI tools can become a limiting factor that hinders team productivity. GUI tools don't provide sufficient flexibility and control over the build process, and they don't allow easy integration with other tools. A good example is integration with popular build-management and

continuous-integration solutions, such as TeamCity, Hudson CI and CruiseControl, which many design teams use ubiquitously for automated software builds.

Nor do GUI tools provide good support for a distributed computing environment. It can take several hours or even a day to build a large FPGA design. To improve the run-time, users do builds on dedicated servers, typically 64-bit multicore Linux machines that have a lot of memory, but in many cases, lack a GUI. Third-party job-scheduling solutions exist, such as Platform LSF, to provide flexible build scheduling, load balancing, and fine-grained control.

A less obvious reason eschewing GUI tools is memory and CPU resource utilization. ISE Project Navigator and PlanAhead are memory-hungry applications: each tool, for instance, uses more than 100 Mbytes of RAM. On a typical workstation that has 2-Gbytes memory, that's 5 percent--a substantial amount of a commodity that can be put to a better use.

Finally, many Xilinx users come to FPGAs with an ASIC design background. These engineers are accustomed to using command-line tool flows, and want to use similar flows in their FPGA design.

Those are some of the reasons why designers are looking to switch to command-line mode for some of the tasks in a design cycle.

Scripting Languages Choices

Xilinx command-line tools can run on both Windows and Linux operating systems, and can be invoked using a broad range of scripting languages, as shown in the following list. Aside from these choices, other scripts known to run Xilinx tools are Ruby and Python.

Perl

Perl is a popular scripting language used by a wide range of EDA and other tools. Xilinx ISE installation contains a customized Perl distribution. Users can enter Perl shell by running *xilperl* command:

```
$ xilperl –v    # display Perl version
```

TCL

Tool Command Language (TCL) is a de facto standard scripting language of ASIC and FPGA design tools. TCL is very different syntactically from other scripting languages, and many developers find it difficult to get used to. This might be one of the reasons TCL is less popular than Perl.

TCL is good for what it's designed for—namely, writing tool command scripts. TCL is widely available, has excellent documentation and enjoys good community support.

Xilinx ISE installation comes with a customized TCL distribution. To start TCL shell use *xtclsh* command:

```
$ xtclsh –v # display TCL version
```

Unix bash and csh

There are several Linux and Unix shell flavors. The most popular are bash and csh.

Unlike other Unix/Linux shells, csh boasts a C-like scripting language. However, bash scripting language offers more features, such as shell functions, command-line editing, signal traps and process handling. Bash is a default shell in most modern Linux distributions, and is gradually replacing csh.

Windows batch and PowerShell4>

Windows users have two scripting-language choices: DOS command line and the more flexible PowerShell. An example of the XST invocation from DOS command line is:

```
>xst -intstyle ise -ifn "crc.xst" –ofn "crc.syr"
```

Build Flows

Xilinx provides several options to build a design using command-line tools. The four most popular options are direct invocation, xflow, xtclsh and PlanAhead.

The direct-invocation method calls tools in the following sequence: xst (or other synthesis tool), ngdbuild, map, place-and-route, trce (optional) and bitgen.

Designers can auto-generate the sequence script from the ISE Project Navigator.

The following script is an example of a direct-invocation build.

```
xst -intstyle ise -ifn "/proj/crc.xst" -ofn "/proj/crc.syr"
ngdbuild -intstyle ise -dd _ngo -nt timestamp -uc  /ucf/crc.ucf -p xc6slx9-csg225-3 crc.ngc crc.ngd
map -intstyle ise -p xc6slx9-csg225-3 -w -ol high -t 1 -xt 0 -global_opt off -lc off -o crc_map.ncd crc.ngd
crc.pcf
par -w -intstyle ise -ol high crc_map.ncd crc.ncd crc.pcf


trce -intstyle ise -v 3 -s 3 -n 3 -fastpaths -xml crc.twx crc.ncd -o crc.twr crc.pcf


bitgen -intstyle ise -f crc.ut crc.ncd
```

Using Xflow

Xilinx's XFLOW utility provides another way to build a design. It's more integrated than direct invocation, doesn't require as much tool knowledge, and is easier to use. For example, XFLOW does not require exit code checking to determine a pass/fail condition.

The XFLOW utility accepts a script that describes build options. Depending on those options, XFLOW can run synthesis, implementation, bitgen, or a combination of all three.

Synthesis only:

```
xflow -p xc6slx9-csg225-3 -synth synth.opt ../src/crc.v
```

Implementation:

```
xflow -p xc6slx9-csg225-3 -implement impl.opt  ../crc.ngc
```

Implementation and bitgen:

```
xflow -p xc6slx9-csg225-3 -implement impl.opt -config bitgen.opt ../crc.ngc
```

Designers can generate the XFLOW script manually or by using one of the templates located in the ISE installation at the following location: *$XILINX\ISE_DS\ISE\xilinx\data*. Xilinx ISE software doesn't provide an option to auto-generate XFLOW scripts.

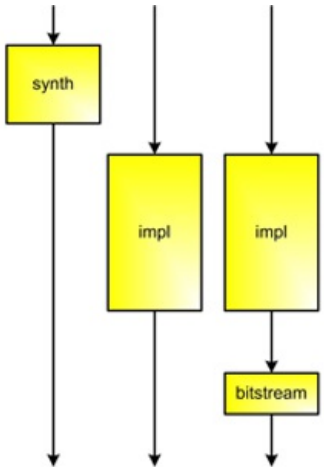

Figure 1: Xflow options

Using Xtclsh

Designs can also be built using TCL script invoked from the Xilinx *xtclsh*, as follows:

```
xtclsh crc.tcl rebuild_project
```

The TCL script can be written manually, and then passed as a parameter to *xtclsh,* or it can be auto-generated from the ISE ProjectNavigator.

Xtclsh is the only build flow that accepts the original ISE project in .xise format as an input. All other flows require projects and file lists, such as .xst and .prj, derived from the original .xise project.

Each of the xst, ngdbuild, map, place-and-route, trce and bitgen tool options has its TCL-equivalent property. For example, the xst command-line equivalent of the Boolean "Add I/O Buffers" is –iobuf, while –fsm_style is the command-line version of "FSM Style" (list).

Each tool is invoked using the TCL "*process run*" command, as shown in the following figure.

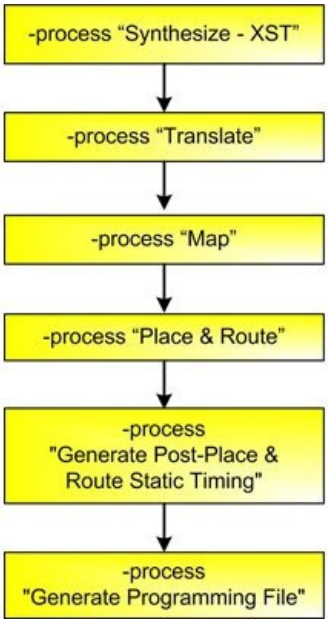

Figure 2: Xtclsh processes

Using PlanAhead

An increasing number of Xilinx users are migrating from ISE Project Navigator and adopting PlanAhead as a main design tool. PlanAhead offers more build-related features, such as scheduling multiple concurrent builds, along with more flexible build options and project manipulation.

PlanAhead uses TCL as its main scripting language. TCL closely follows Synopsys’ SDC semantics for tool-specific commands.

PlanAhead has two command-line modes: interactive shell and batch mode. To enter an interactive shell, type the following command:

```
PlanAhead –mode tcl
```

To run the entire TCL script in batch mode, source the script as shown below:

```
PlanAhead –mode tcl –source <script_name.tcl>
```

The PlanAhead build flow (or run) consists of three steps: synthesis, implementation and bitstream generation, as illustrated in the following figure.

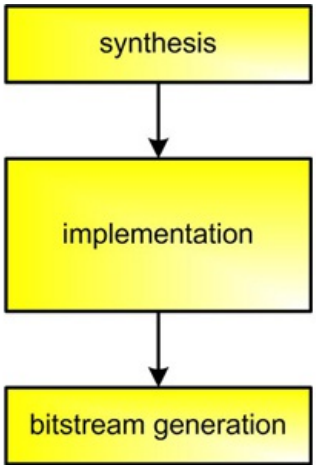

Figure 3: PlanAhead flow

The PlanAhead software maintains its log of the operations in the *PlanAhead.jou* file. The file is in TCL format, and located in C:\Documents and Settings\<user name>\Application Data\HDI\ on Windows, and ~/.HDI/ on Linux machines.

Users can run the build in GUI mode first, and then copy some of the log commands in their command-line build scripts.

Below is an example PlanAhead TCL script.

```
create_project pa_proj {crc_example/pa_proj} -part xc6slx9csg225-3
set_property design_mode RTL [get_property srcset
[current_run]]
```

```
add_files -norecurse {crc_example/proj/.. /src/crc.v}
set_property library work [get_files -of_objects [get_property srcset [current_run]] {src/crc.v}]
add_files -fileset [get_property constrset [current_run]] -norecurse {ucf/crc.ucf}
set_property top crc [get_property srcset [current_run]]
set_property verilog_2001 true [get_property srcset [current_run]]
launch_runs -runs synth_1 -jobs 1 -scripts_only -dir {crc_example/pa_proj/pa_proj.runs}
launch_runs -runs synth_1 -jobs 1
launch_runs -runs impl_1 -jobs 1
set_property add_step Bitgen [get_runs impl_1]
launch_runs -runs impl_1 -jobs 1 -dir {crc_example/pa_proj/pa_proj.runs}
```

In addition to providing standard TCL scripting capabilities, PlanAhead offers other powerful features. It allows query and manipulation of the design database, settings, and states, all from a TCL script. This simple script illustrates some of the advanced PlanAhead features that you can use in a command-line mode:

```
set my_port [get_ports rst]
report_property $my_port
get_property PULLUP $my_port
set_property PULLUP 1 $my_port
get_property PULLUP $my_port
```

The simple script adds a pull-up resistor to one of the design ports. The same operation can be done by changing UCF constraints and rebuilding the project, or using the FPGA Editor. But PlanAhead can do it with just three lines of code.

Many developers prefer using the *make* utility for doing FPGA builds. The two most important advantages of *make* are built-in checking of the return code and automatic dependency tracking. Any of the flows described above can work with *make*.

This Tip has discussed advantages of using Xilinx tools in command-line mode, explored several Xilinx build flows, and examined different scripting languages. The goal was not to identify the best build flow or a script language, but rather to discuss the available options and parse the unique advantages and drawbacks of each to help Xilinx FPGA designers make an informed decision.

Xilinx FPGA designers can base their selection process on a number of criteria, such as existing project settings, design team experience and familiarity with the tools and scripts, ease of use, flexibility, expected level of customization and integration with other tools, among others.

9. Xilinx Environment Variables

The very first task an FPGA designer encounters while working with command-line tools is setting up environment variables. The procedure varies depending on the Xilinx ISE version. In ISE versions before 12.x, all environment variables are set during the tool installation. Users can run command-line tools without any further action.

That changed in ISE 12.x. Users now need to set all the environment variables every time before running the tools. The main reason for the change is to allow multiple ISE versions installed on the same machine to coexist by limiting the scope of the environmental variables to the local shell or command-line terminal, depending on the operating system.

There are two ways of setting up environment variables in ISE 12.x. The simpler method is to call a script settings32.{bat,sh,csh} or settings64.{bat,sh,csh}, depending on the platform. The default location on Windows machines is C:\Xilinx\12.1\ISE_DS\; on Linux, it's /opt/Xilinx/12.1/ISE_DS/.

Linux bash example of calling the script is:

```
$ source /opt/Xilinx/12.1/ISE_DS/settings64.sh
```

The other option is to set the environment variables directly, as shown below.

Windows:

```
> set XILINX= C:\Xilinx\12.1\ISE_DS\ISE
> set XILINX_DSP=%XILINX%
> set PATH=%XILINX%\bin\nt;%XILINX%\lib\nt;%PATH%
```

UNIX/Linux bash:

```
$ export XILINX=/opt/Xilinx/12.1/ISE_DS/ISE
$ export XILINX_DSP=$XILINX
$ export PATH=${XILINX}/bin/lin64:${XILINX}/sysgen/util:${PATH}
```

UNIX/Linux csh:

```
% setenv XILINX     /opt/Xilinx/12.1/ISE_DS/ISE
% setenv XILINX_DSP $XILINX
% setenv PATH  ${XILINX}/bin/lin64:${XILINX}/sysgen/util:${PATH}
```

To test if the variable is set correctly on UNIX/Linux csh/bash:

```
% echo $XILINX
```

UNIX/Linux bash:

```
$ echo $XILINX
```

Windows:

```
> set XILINX
```

To disable the variable on UNIX/Linux csh:

```
% unsetenv XILINX
```

There are several other environment variables that affect Xilinx tools behavior. Some of them are shown in the following tables:

Name	XILINXD_LICENSE_FILE LM_LICENSE_FILE
Values	License file or license server in the format port@hostname
Details	Xilinx application licenses

Name	XIL_MAP_NODRC
Values	1
Details	Disable physical DRC errors. Applicable to MAP application.

Name	XIL_MAP_SKIP_LOGICAL_DRC
Values	1
Details	Disable logical DRC errors. Applicable to MAP application

Name	XIL_MAP_NO_DSP_AUTOREG
Values	1
Details	Disable the optimization where registers are pulled into DSP48 primitive. Applicable to MAP application

Name	XIL_PAR_DEBUG_IOCLKPLACER
Values	none
Details	In cases where clock design rules are violated, PAR aborts the execution with an error. To debug partially routed design and allow PAR to complete, set this environmental variable.

Name	XIL_PAR_NOIORGLLOCCLKSPL
Values	1
Details	Disable IO, regional and local clock placement. Applicable to PAR application.

Name	XIL_PAR_SKIPAUTOCLOCKPLACEMENT
Values	1
Details	Disable global clock placement. Applicable to PAR application.

Name	XIL_PROJNAV_PROCESSOR_AFFINITY_MASK
Values	Processor mask. For example, on a system with four processors setting 5 will enable processors 0 and 2, according to the OS allocation.
Details	On multi-processor machines, this environmental variable controls which processors are allowed to run.

Name	XIL_TIMING_HOLDCHECKING
Values	YES or NO
Details	Enable analysis of hold paths on FROM:TO constraints

Name	XIL_TIMING_ALLOW_IMPOSSIBLE
Values	1
Details	In case MAP application cannot meet a timing constraint, it aborts the execution. To allow MAP to bypass the error and complete, set this variable. The variable is intended to be used for mainly debug purposes during timing closure.

More detailed information about the above environmental variables can be found in answer records on Xilinx website and constraints documentation.

10. Xilinx ISE Tool Versioning

Xilinx ISE software is using two versioning systems: numeric and letter.

Numeric version is used for customer software releases. It uses <major version>.<minor version > format. For example: 10.1, 11.2, or 12.4.

Letter versioning is used internally at Xilinx. The format is <letter>.<build number>. For example: K.39 or M.57. The <letter> part corresponds to the <major version> of the numeric format. For example, K corresponds to 10, L to 11, and M to 12.

There isn't one to one correspondence between <minor version > and the <build number> parts of the numeric and letter formats. Xilinx does frequent tool builds internally, and when the software reaches production quality, it's released as a new <minor version >.

The software version is typically shown on top of each report. The following example is the top two lines on the XST report:

```
Release 12.1 - xst M.53d (nt)
Copyright (c) 1995-2010 Xilinx, Inc.  All rights reserved.
```

Both numeric and letter software versions are typically shown in the "About" dialog.

11. Lesser Known Xilinx Tools

Xilinx ISE installation contains several lesser-known command-line tools that FPGA designers might find useful. ISE Project Navigator and PlanAhead invoke these tools behind the scenes as part of the build flow. The tools are either poorly documented or not documented at all, which limits their value to users.

The tools listed below are some of the most useful ones.

data2 mem: This utility, which is used to initialize the contents of a BRAM, doesn't require rerunning Xilinx implementation. The new BRAM data is inserted directly into a bitstream. Another data2mem usage is to split the initialization data of a large RAM consisting of several BRAM primitives into individual BRAMs.

fpga_edline: A command-line version of the FPGA Editor, fpga_edline can be useful for applying changes to the post-place-and-route .ngc file from a script as part of the build process. Some examples of use cases include adding ChipScope probes, minor

routing changes, LUT or IOB property changes.

mem_edit: This is not a tool per se, but a script that opens Java applications. You can use it for simple memory content editing.

netgen: Here is a tool you can use in many situations. It gets a Xilinx design file in .ncd format as an input, and produces a Xilinx-independent netlist for use in simulation, equivalence checking and static timing analysis, depending on the command-line options.

ncgbuild: This is a tool that consolidates several design netlist files into one. It's mainly used for the convenience of working with a single design file rather than multiple designs and IP cores scattered around different directories.

obngc: This is a utility that obfuscates .ngc files in order to hide confidential design features and prevent design analysis and reverse-engineering. You can explore NGC files in FPGA Editor.

pin2ucf: You can use this utility to generate pin-locking constraints from NCD file in UCF format.

xdl: An essential tool for power users, Xdl has three fundamental modes: report device resource information, convert NCD to XDL and convert XDL to NCD. XDL is a text format that third-party tools can process. This is the only way to access post-place-and-route designs.

Example:

```
xdl -ncd2xdl <design_name>.ncd
```

xreport: This is a utility that lets you view build reports outside of the ISE Project Navigator software. It has table views of design summary, resource usage, hyperlinked warnings and errors, and message filtering.

xinfo: A utility that reports system information, xinfo also details installed Xilinx software and IP cores, licenses, ISE preferences, relevant environment variables and other useful information.

lmutil: A license management utility distributed by Macrovision Corporation, and used by Xilinx software. It is used to communicate with a license server to install, check-out, and poll for floating licenses. The following command is an example of polling for available ISE licenses on server with IP address 192.168.1.100:

```
$ lmutil lmdiag -c 2100@192.168.1.100 ISE
lmutil - Copyright (c) 1989-2006 Macrovision Corporation.
-----------------------------------------------------
License file: 2100@192.168.2.52
-----------------------------------------------------
"ISE" v2010.05, vendor: xilinxd
   License server: 192.168.1.100    floating license no expiration date
   TS_OK: Checkout permitted when client is using terminal client
Feature:       ISE
Hostname:      shiitake
License path:  2100@192.168.1.100
```

reportgen: A utility that takes an NCD file as an input, and generates net delay, IO pin assignment, and clock region reports.

All the tools mentioned above are installed in $XILINX/ISE/bin/{nt, nt64, lin, lin64} and $XILINX/common/bin/{nt, nt64, lin, lin64} directories. $XILINX is an environmental variable that points to the ISE installation directory.

More inquisitive readers might want to further explore these directories to discover other interesting tools that can boost design productivity.

12. Understanding Xilinx Tool Reports

FPGA synthesis and physical implementation tools produce a great number of reports that contain various bits of information on errors and warnings, logic utilization, design frequency, timing, clocking, etc. It requires a significant amount of experience with the design tools to efficiently navigate the reports and quickly find the required information. Xilinx and other FPGA design tools provide a GUI view of some of the most important and frequently used information in the reports, but this is not always sufficient.

Most of the reports have a consistent structure. They are organized into multiple sections, where each section contains a particular type of information. Examples of report sections are errors, warnings, IO properties, utilization by hierarchy.

The following list briefly describes most of the reports produced by Xilinx XST and physical implementation tools.

Report name: **XST synthesis report**
Tool: Xilinx XST
File Extension: .srp, .syr

A XST synthesis report contains information about synthesis options, HDL and low-level synthesis, design summary, and the estimates of logic utilization and performance.

Report name: **translation report**
Tool: NGDBUILD

File Extension: .bld

A translation report contains information about NDGBUILD run, including user-defined constraints, and partition implementation.

Report name: **MAP report**
Tool: MAP
File Extension: .mrp

This report contains information about the MAP run: design summary, removed logic, IO properties, logic utilization by hierarchy, and several other sections.

By default, a MAP report contains only basic information. Use `-detail` MAP option to enable the complete report.

Report name: **physical synthesis report**
Tool: MAP
File Extension: .psr

The physical synthesis report file contains information about MAP options responsible for different timing and area optimizations: global optimization (-global_opt), retiming (-retiming), equivalent register removal (-equivalent_register_removal), combinatorial logic optimization (-logic_opt), register duplication (-register_duplication), and power fine grain slice clock gating optimization (-power).

Report name: **physical constraints report**
Tool: MAP
File Extension: .pcf

A physical constraints report contains all physical constraints specified during design entry and added by the user.

Report name: **place and route report**
Tool: PAR
File Extension: .par

A place and route report contains different information about the PAR run: command line options, design utilization and performance summary, detailed clocking resources report, and partition implementation status.

Report name: **IO pad report**
Tool: PAR
File Extension: .pad

A pad report is a file that contains a list of all IO components used in the design, their associated FPGA pins, and characteristics, such as direction, IO standard, slew rate, and drive strength.

Report name: **unrouted nets report**
Tool: PAR
File Extension: .unroutes

This report contains a list of signals that could not be routed during Place and Route (PAR) stage. The presence of any signals in that report indicates a design error.

Report name: **design rule check report**
Tool: BITGEN
File Extension: .drc

The report contains various design rule checks (DRC) results performed by BITGEN. DCR can be disabled by specifying the –d (no DRC) BITGEN command line option.

Report name: **bitstream generation report**
Tool: BITGEN
File Extension: .bgn

This report contains information options used during the BITGEN, and the overall results.

Report name: **timing report**
Tool: Xilinx TRCE
File Extension: .twr, .twx

Report name: **timing constraints interaction report**
Tool: TRCE
File Extension: .tsi

Timing reports are described in greater detail in Tip #91.

Many reports contain overlapping or complementary information. The following list organizes reports by types of information they provide.

Logic utilization

Logic utilization is reported in several reports during the FPGA build process.

Synthesis report (.syr, .srp) contains estimated utilization information, excluding IP cores that are added during translation stage, and before logic optimization performed during MAP.

MAP report (.mrp) provides post-placement logic utilization for the entire design, and the utilization break-down by individual module.

PAR report (.par) provides the most accurate post-routing logic utilization information.

Timing

Synthesis (.syr, .srp), and MAP (.mrp) reports contain estimated timing information performed at the logic level.

The complete timing information is generated after place and route stage. PAR report (.par) contains the overall timing score.

TRCE static timing analysis reports (.twr, .twx) contain detailed timing information organized by signals, endpoints, clock, or timing group.

IO information

MAP report (.mrp) provides post-placement information about the IO properties. The report doesn't contain any routing details.

PAD report (.pad) contains the complete post-routing IO information.

Clocking

Synthesis (.syr, .srp) report provides basic information about design clocks and the clock load (how many registers the clock is driving).

MAP (.mrp) report contains information about used clocking resources, such as MMCM, and global buffers. It also provides detailed information about control sets: the combination of clock, reset, clock enable, and logic load count.

PAR report (.par) provides detailed timing information about each clock in the design: skew, maximum delay, positive or negative slack, and summary of timing errors.

TRCE timing reports (.twr, .twx) contain the most complete and detailed timing information.

13. Naming Conventions

With an increasing complexity of FPGA designs, design practices, methods, and processes are becoming increasingly important success factors. Good design practices have an enormous impact on an FPGA design's performance, logic utilization, more efficient use of FPGA resources, improved system reliability, team productivity, and faster time-to-market. Conversely, poor design practices can lead to higher development and system cost, lower performance, missed project schedules, and unreliable designs.

Tips #13-16 provide several rules and guidelines for Verilog naming conventions, coding style, and synthesis for an FPGA. The guiding principles are improving code readability and portability, facilitating code reuse, and consistency across different projects.

To be effective, rules and guidelines have to be formalized in a document, disseminated throughout design teams, and enforced by periodic code inspections and design reviews.

The scope of the documents may vary. In a small company it can contain a brief overview of acceptable naming conventions and coding styles. In a large, established company it can be an extensive set of documents covering different aspects of design practices, methods, and processes in great detail. In safety and mission-critical designs, such as military or medical applications, the emphasis is on following strict coding standards and processes that prevent a device failure from all possible sources. It is beneficial for the document to include checklists for signing off different stages of the design process: simulation, synthesis, physical implementation, and bring-up.

The goal of the document is twofold. One is establishing consistency and the same "look and feel" of the code written by different designers within a project or a company, regardless of their background and experience. The other is improved code readability and clarity, which helps reduce the number of defects or bugs.

The list of suggestions offered in Tips #13-16 is by far incomplete. Its main goal is to establish a basis for developing a more comprehensive set of rules and guidelines tailored for specific project or a team.

File header

It is hard to overestimate the importance of including a file header in every source code and script file. The header has to contain at least the proper copyright information and disclaimer. If desired, it can contain a brief description, name and e-mail of the design engineer, version, and a list of changes to that file. The most important reason for including the header is a legal one. A source code or a script file constitutes an intellectual property. The header establishes ownership and rights for that file, which can be used as

evidence in a court of law during litigation, patent dispute, or copyright arbitration.

The following is an example of a file header.

```
/*-------------------------------------------------------------
Copyright (C) 2011 OutputLogic.com
This source file may be used and distributed without restriction provided that this copyright statement is not
removed from the file and that any derivative work contains the original copyright notice and the associated
disclaimer.


THIS SOURCE FILE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION,
THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
Filename: top.v
Engineer: John Doe
Description: top-level module
-------------------------------------------------------------*/
```

Verilog file organization

It is a good design practice to implement a single Verilog module in a file, as shown in the following code example.

```
// file: my_module.v
module my_module;
// module implementation...
endmodule // my_module
```

A module name should match the file name. Following this rule makes it easier to process and manage the projects. An exception might be a library file that contains multiple short modules.

File and directory names

Although it's legal to include spaces and other special characters in file and directory names in Linux and Windows operating systems, it is not a good design practice. Having spaces can cause obscure problems in tools and scripts. Linux command line utilities and shells were designed based on a premise that a space delimits a field value, rather than being an acceptable component of a file or directory name. So do tools that rely on those utilities and shells. Including quotes around the filename with spaces and special characters might not always work, for example when this filename is passed through multiple subshells and pipes.

As a rule, use only alphanumeric and underscore characters in file and directory names.

Try to give files and directories that are meaningful and unique names that may help describe their contents.

Uppercase vs. lowercase

Net, variable, module, instance, and other names in Verilog and SystemVerilog languages are case sensitive. Synthesis tools enforce that rule by default. However, there are several tool options that allow some flexibility. For example, XST provides `-case` option, which determines whether the names are written to the final netlist using all lower or all upper case letters, or if the case is maintained from the source. The synopsis of this option is:

```
xst run -case {upper | lower | maintain}
```

It is recommended that you name parameters, macros, constants, attributes, and enumerated type values in uppercase. Module, file, function, instance, and task names can be in lowercase. Don't use mixed case in names.

Comments

The liberal use of meaningful and non-obvious comments is encouraged in order to improve the code readability and help you to understand the design intent. Comment consistency is an important requirement, such as observing the maximum line size. Verilog allows single-line and multi-line comments.

```
reg my_reg; // this is a single-line comment
/*
   this is a multi-line comment
*/
```

There are comment styles, such as /** */, used by various tools to perform comment extraction to be used in documentation.

Using tabs for code indentation

Tabs are used for code indentation. However, tabs are not displayed in the same way on different computer systems. It is recommended that you configure a code editor to replace a tab character with whitespaces. It's customary to use four white characters to replace a single tab.

Newline characters

Code editors in Windows operating system use carriage return and line feed (CR/LF) character pair to indicate a newline. This is different than in Unix/Linux systems, which only use LF character. That difference will cause extraneous CR characters appear as ^M in files developed on Windows and viewed on Unix/Linux. Moreover, some tools and scripting languages might be sensitive to these extra characters, leading to incorrect results.

Some code editors provide an option of only using LF character for a newline. Also, there is a utility called dos2unix that can be used to remove extra CR characters.

Limit the line width

Standard column width for most terminals, editors, and printers is 80 characters. The motivation behind limiting the line width to 80 characters is to make the code more readable on different systems.

Identifiers

Two most popular HDL identifier coding styles are the following:

1. Mixed case, no underscores. Each word starts with an uppercase letter. Two or more consecutive uppercase letters are disallowed.

```
reg MyRegister;
wire MyNet;
wire DataFromCPU; // incorrect: consecutive uppercase letters.
```

All lowercase, underscores between words.

```
reg my_register;
wire my_net;
wire data_from_cpu;
```

The second coding style is more prevalent. Most of the tool and IP Core vendors, including Xilinx, have adopted this style. It is important to adopt and follow a single style that can be used by all developers and across different projects.

Escaped identifiers

Verilog standard provides a means of including any of the printable ASCII characters in an identifier. Such an identifier has to start with the backslash escape character (\) and end with space, tab, or newline character. The leading backslash and the terminating white space are not part of the identifier.

Escaped identifiers are used by automated code generators, ASIC netlisting, and other CAD tools that use special characters in identifiers. Escaped identifiers are extensively used in the code generated by various Xilinx cores. FPGA designers should avoid using escaped identifiers, because they make the code less readable, and that can cause it to be hard to find errors, such as mistyped trailing space.

The following is an example of a Xilinx shift register core that uses escaped identifiers. There is a terminating white space between the end of an identifier and a semicolon or a bracket.

```
module shift_ram_coregen (input sclr,ce,clk,
       input [0 : 0] d,
       output [0 : 0] q);
  wire \BU2/sinit ;
  wire \BU2/sset ;
  wire \BU2/U0/i_bb_inst/Mshreg_f1.only_clb.srl_sig_62_1_9 ;
  wire \BU2/U0/i_bb_inst/Mshreg_f1.only_clb.srl_sig_62_0_8 ;
  wire \BU2/U0/i_bb_inst/N1 ;
  wire \BU2/U0/i_bb_inst/N0 ;
  wire \BU2/U0/i_bb_inst/f1.only_clb.srl_sig_62_5 ;
  wire NLW_VCC_P_UNCONNECTED;
  wire NLW_GND_G_UNCONNECTED;
  wire [0 : 0] d_2;
  wire [0 : 0] q_3;
  wire [3 : 0] \BU2/a ;


  assign d_2[0] = d[0], q[0] = q_3[0];
  VCC   VCC_0 (.P(NLW_VCC_P_UNCONNECTED));
  GND   GND_1 (.G(NLW_GND_G_UNCONNECTED));


  FDE #(.INIT ( 1'b0 ))
   \BU2/U0/i_bb_inst/f1.only_clb.srl_sig_62  (
   .C(clk),
```

```
   .CE(ce),
   .D(\BU2/U0/i_bb_inst/Mshreg_f1.only_clb.srl_sig_62_1_9 ),
   .Q(\BU2/U0/i_bb_inst/f1.only_clb.srl_sig_62_5 ));


   SRLC32E #( .INIT ( 32'h00000000 ))
   \BU2/U0/i_bb_inst/Mshreg_f1.only_clb.srl_sig_62_1 (
   .CLK(clk),
   .D(\BU2/U0/i_bb_inst/Mshreg_f1.only_clb.srl_sig_62_0_8 ),
   .CE(ce),
   .Q(\BU2/U0/i_bb_inst/Mshreg_f1.only_clb.srl_sig_62_1_9 ),
   .Q31(),
   .A({\BU2/U0/i_bb_inst/N1 , \BU2/U0/i_bb_inst/N1 ,  \BU2/U0/i_bb_inst/N1 , \BU2/U0/i_bb_inst/N0 ,
\BU2/U0/i_bb_inst/N1 }));


   SRLC32E #(.INIT ( 32'h00000000 ))
   \BU2/U0/i_bb_inst/Mshreg_f1.only_clb.srl_sig_62_0  (
   .CLK(clk),
   .D(d_2[0]),
   .CE(ce),
   .Q(),
   .Q31(\BU2/U0/i_bb_inst/Mshreg_f1.only_clb.srl_sig_62_0_8 ),
   .A({\BU2/U0/i_bb_inst/N1 , \BU2/U0/i_bb_inst/N1 , \BU2/U0/i_bb_inst/N1 , \BU2/U0/i_bb_inst/N1 ,
\BU2/U0/i_bb_inst/N1 }) );


   VCC   \BU2/U0/i_bb_inst/XST_VCC  ( .P(\BU2/U0/i_bb_inst/N1 ));
  GND   \BU2/U0/i_bb_inst/XST_GND  ( .G(\BU2/U0/i_bb_inst/N0 ));


  FDRE #( .INIT ( 1'b0 ))
  \BU2/U0/i_bb_inst/gen_output_regs.output_regs/fd/output_1  (
   .C(clk),
   .CE(ce),
   .D(\BU2/U0/i_bb_inst/f1.only_clb.srl_sig_62_5 ),
   .R(sclr),
   .Q(q_3[0]));


endmodule
```

Name prefix

A meaningful name prefix can be used to categorize the net, register, instance, or other identifiers. For example, all the nets that originate from the memory controller can have "`mem_`" name prefix.

Name suffix

Name suffix can be used to provide additional information about the net, register, instance, or other identifiers.

For example, "`_p`" or "`_n`" suffix can indicate positive or negative polarity.

The following table provides a few examples of name suffixes.

Table 1: Examples of name suffixes

Suffix	***Description***
_p, _n	Positive or negative polarity
_ff , _q	An output of a register
_c	A signal driven by combinatorial gates
_cur	Current state
_next	Next state
_tb	Testbench

Clock names

Clock signal names can include frequency, and other characteristics, such as single-ended or differential clock, to make the name more descriptive. It is also customary to include "`clk`" as part of a clock name. The following are few examples of clock signal names.

```
wire clk50; // single-ended 50MHz clock
wire clk_200_p, clk_200_n; // 200MHz differential clock
wire clk_en; // clock enable
wire clk_333_mem; // 333MHz memory controller clock
```

Reset names

Reset names can include information about reset polarity (active high or low), synchronous or asynchronous, and global or local. The following are few examples of reset signal names.

```
reg reset, phy_reset; // active high reset
reg reset_n, reset_b;  // active low reset
reg rst_async; // asynchronous reset
```

Port names

Port names can end with a suffix that indicates port direction, as shown in the following examples.

```
   input  [9:0]   addr_i;  // input
   input  [31:0] DataI;  // input
   output write_enable_o; // output
   inout  [31:0] data_io; // bidirectional
   inout  [31:0] ConfigB; // bidirectional
```

Module names

Module names have to be unique across the entire project, regardless of the hierarchy level the module is instantiated. Reserved Verilog or SystemVerilog keywords should not be used as module names.

Literals

Verilog provides a rich set of options to specify literals. Literals can be specified as a simple integer, for example 32, or as a based integer, for example `6'd32`.

The syntax of specifying a based integer is:

```
   <size>'s<base><value>
```

`<size>` field specifies the total number of bits represented by the literal value. This field is optional. If not given, the default size is 32 bits.

"`s`" field indicates a twos complement signed value. It is an optional field, and if not provided, the value is treated as an unsigned.

`<base>` field specifies whether the value is given as a binary, octal, decimal, or hex. If the base is not specified, a simple literal defaults to a decimal base.

`<value>` field specifies the literal integer value.

A simple literal integer defaults to a signed value. A based literal integer defaults to an unsigned value, unless explicitly specified as a signed using "`s`" field.

It is recommended that you describe literals using the base specification to avoid the possibility of introducing a mistake. For example, use `6'd32` instead of `32`.

A common mistake is a mismatch of `<size>` and `<value>` of the same literal. For example, using 5 `'d32` will not cause any synthesis error, but results in truncation of the most-significant bit of value 32 (binary 'b100000), which will become 0.

If `<size>` is greater than `<value>`, `<value>` will be left-extended, as shown in the following examples.

```
wire [7:0] my_value;
assign my_value = 'b0; // left-extended to 8'h00
assign my_value = 'b1; // left-extended to 8'h01;
```

Verilog defines several rules for dealing with different literal sizes in assignment and other operators. A good overview of the rules and methods to avoid potential problems is described in the [1].

Resources

[1] *"Standard Gotchas: Subtleties in the Verilog and SystemVerilog Standards That Every Engineer Should Know",* by Stuart Sutherland. Published in the proceedings of SNUG Boston, 2006.
http://www.sutherland-hdl.com/papers/2007-SNUG-SanJose_gotcha_again_paper.pdf

14. Verilog Coding Style

Using `include compiler directive

The file inclusion `include compiler directive" is used to insert the entire contents of a source file into another file during synthesis. It is typically used to include global project definitions, without requiring the repetition of the same code in multiple files. Another use case is inserting portions of a code into modules, as shown in the following example:

```
// file test_bench_top.v
// top-level simulation testbench
module test_bench_top;


   `include "test_case.v"


endmodule


// file test_case.v
initial begin
   //…
end


task my_task;
   //…
endtask
```

>The syntax for `include compiler directive is defined as:

```
`include <filename>
```

<filename> can be the name of the file, and also contain an absolute or relative pathname:

```
`include "test_case.v"
`include "../../includes/test_case.v"
`include "/home/myprojects/test/includes/test_case.v"
```

It is recommended using only the filename in the `include, and not absolute or relative pathnames. That will make the code location-independent, and therefore more portable. Pathnames can be specified using synthesis or simulation tool options, such as XST -vlgincdir <directory_name> (Verilog Include Directories), and ModelSim +incdir+<directory_name>.

Another recommendation is to keep included files simple and not to use nested `include directives.

Using `define compiler directive, parameter, and localparam

```
`define is a text macro substitution compiler directive. It is defined as:
`define <text macro>


<text macro> can include single- or multi-line text with an optional list of arguments.


`define has a global scope. Once a text macro name has been defined, it can be used anywhere in the project. Text
macros are typically simple identifiers used to define state names, constants, or strings.

Text macro also can be defined as a synthesis or simulation tool option. For example, XST supports "-define<text
macro>", and ModelSim has "+define+<text macro>" command line option.

The advantage of using command-line option is that, unlike using `define in a source code, it is independent on
the compile order of the files.


parameter keyword defines a module-specific parameter, which has a scope of specific module instance. Parameter is
used to provide different customizations to a module instance, for example, width of input or output ports. The
following is an example of using parameter keyword.


module adder #(parameter WIDTH = 8) (
   input [WIDTH-1:0] a,b, output [WIDTH-1:0] sum );

   assign sum = a + b;
endmodule // adder

// an instance of adder module
adder # (16) adder1 (.a(a[15:0]),.b(b[15:0]),.sum(sum[15:0]));
```

localparam keyword is similar to parameter. It is assigned a constant expression, and has a scope inside a specific module. It is defined as:

```
localparam <identifier> = <value>;
```

It is recommended not to use `define and parameter for module-specific constants, such as state values, and use localparam instead.

Operand size mismatch

Verilog specification defines several rules to resolve cases where there is size mismatch between left and right-hand side of an assignment or an operator. The following are few examples.

```
reg [3:0] a;
reg [4:0] b;
reg c;
always @(posedge clk)
   if (a == b)  // a and b have a different size       c <= 1'b1;

reg [3:0] a;
wire b;
wire [2:0] c;
wire [3:0] d;

a = b ? c : d;  // c and d have a different size
wire [9:0] a;
wire [5:0] b;
wire [1:0] c;
wire [7:0] d;
assign d = a | b | c;  // a,b,c,d have a different size
```

The rules that resolve the different cases of size differences can be complex and hard to follow. When two operands of different bit sizes are used, and one or both of the operands is unsigned, the smaller operand is extended with zeros on the most significant bit side to match the size of the larger operand. When the size of right-hand side is greater than the size on an assignment destination, the upper bits of right-hand side are truncated.

It is recommended to exactly match the bit size of the right and left side of operators and assignments to avoid automatic truncation or size extension by the synthesis tools.

A good overview of the rules and methods to avoid potential problems is described in [1].

Connecting ports in a module instance

Verilog defines two ways to connect ports in instantiated modules: by name and by port order. Using the ordered port connection method, the first element in the port list is connected to the first port declared in the module, the second to the second port, and so on. Connecting ports by name allows more flexibility. For example, an unconnected output port can be omitted. Although it requires larger amount of code than using “by order” method, it is more readable and less error-prone, especially for large modules with hundreds of ports. Port connectivity options are illustrated in the following example.

```
// Lower level module
module low(output [1:0] out, input in1, in2);
   // …
endmodule

// Higher-level module:
module high(output [5:0] out1, input [3:0] in1,in2);

   // connecting ports by order
   low low1(out1[1:0], in1[0], in2[0] )

   // connecting ports by order
   // error: in2 and in1 are swapped    low low2(out1[3:2], in2[1], in1[1] )

   // connecting ports by name
   low low3(.out(out1[5:4]), .in1(in1[2]), .in2(in2[2]) )

   // connecting ports by name; an output is left unconnected
   low low4(.out(), .in1(in1[3]), .in2(in2[3]) )

endmodule
```

It is recommended to use “by name” port connection method in module instantiations.

Using variable part-select to define bit range

Variable or indexed part-select is defined by a starting point of its range, and a constant width that can either increase or decrease from the starting point. The following is an example of using variable part-select.

```
wire [31:0] part sel in,
```

```
reg  [0:31] part_sel_out

assign part_sel_out[24 +:8]  = part_sel_in[7 -: 8];
assign part_sel_out[16 +:8]  = part_sel_in[15 -: 8];
assign part_sel_out[15 -:16] = part_sel_in[16 +: 16];
```

Using variable part-selects allows writing more compact and less error-prone code. In case of the above example, it prevents making a mistake in a bit range bounds.

```
assign part_sel_out[24:31] = part_sel_in[7:0];
assign part_sel_out[16:24] = part_sel_in[15:8]; // error
assign part_sel_out[15:0]  = part_sel_in[16:31];
```

Using Verilog for statement

Verilog for statement can be used in many situations to implement shift registers, swapping bits, perform parity checks, and many other use cases. Using for statement for parallel CRC generation is discussed more in Tip #71.

The following is an example of using for statement in a loop.

```
module for_loop( output reg loop_out,
                 input [15:0] loop_in);
   integer iy;
   always @(*) begin
      for(iy=0;iy<16;iy=iy+1) begin
         loop_out = loop_out ^ loop_in[iy];
      end
   end

endmodule // for_loop
```

It is recommended to avoid using for statements for logic replication, and to restrict it only for simple repeat operations, such as swapping bits. Using it for statement in loops can create cascaded logic, which may result in sub-optimal logic utilization and timing performance.

Using functions

The following is a simple example of a Verilog function that performs XOR operation.

```
module function_example( input a,b, output func_out);
   function func_xor;
      input a, b;
      begin
         func_xor = a ^ b;
      end
   endfunction
   assign func_out = func_xor(a,b);
endmodule // function_example
```

It is recommended to use Verilog functions to implement combinatorial logic and other operations that don't require non-blocking assignments, such as synchronous logic. Using functions allows writing more compact and modular code. Verilog functions are supported by all synthesis tools.

Using generate blocks

Generate blocks were introduced in Verilog-2001 to enable easy instantiation of multiple instances of the same module, function, variables, nets, and continuous assignments. The following are two examples of using generate.

```
// a conditional instantiation of modules
parameter COND1 = 1;

generate
if (COND1) begin : my_module1_inst
   my_module1 inst (.clk(clk), .di(di), .do(do));
end
else begin : my_module2_inst
   my_module2 inst (.clk(clk), .di(di), .do(do));
end
endgenerate

// using for loop in generate block
genvar ii;
enerate
   for (ii = 0; ii < 32;  ii = ii+1) begin: for_loop
      my_module1 inst (.clk(clk), .di(di[ii]), .do(do[ii]));
   end
end
endgenerate
```

Developing portable code

In the context of this book, code portability refers to the ability to synthesize an FPGA design with different synthesis tools, and targeting different FPGA families with minimum code modifications.

Even if code portability does not seem to be an important requirement in the beginning of the project, it's highly likely to have its benefits later on in the project's lifecycle. In the high-tech industry business, market requirements change rapidly. Because of this, large parts of the code will need to be reused in a completely different project, which utilizes different FPGA family or even a different vendor than was originally anticipated. It's almost impossible to predict how the code is going to be used over several years of its lifecycle. The code can be packaged as an IP core and licensed to various customers; the code be inherited by another company as the result of an acquisition; or the code may need to be migrated to a newer, more cost-effective FPGA family.

One recommendation for developing portable code is separating FPGA-specific components into separate files or libraries, and using generic wrappers. For example, Xilinx, Altera, and other FPGA vendors have different embedded memory primitives. However, components, such as RAMs, ROMs, and FIFOs, that take advantage of those memory primitives, have very similar interfaces. Not instantiating Xilinx BRAM directly in the code, but using a simple wrapper with generic address, data, and control signals will make the code much more portable.

Another method of improving portability is to develop generic RTL code instead of directly instantiating FPGA-specific components. Tradeoffs of inference vs. direct instantiation are discussed in Tip #16.

Synthesis tools support different subset of Verilog language. For example, XST doesn't support Verilog switch-level primitives such as tranif, while Synplify does. It is recommended to use language constructs supported by most of the synthesis tools.

Developing simple code

Always strive to develop a simple code. As in every programming language, Verilog allows writing elaborated statements, which are elegant from the functional standpoint, but unreadable. The following simple example illustrates this point.

```
reg [5:0] sel;
reg [3:0] result1,result2,a,b;

always @(*) begin
   result1 = sel[0] ? a + b : sel[1] ? a - b :
      sel[2] ? a & b : sel[3] ? a ^ b :
      sel[4] ? ~a : ~ b;

   if(~|sel)
   result1 = 4'b0;
end // always


always @(*) begin
   casex(sel)
      6'bxxxxx1: result2 = a + b;
      6'bxxxx10: result2 = a - b;
      6'bxxx100: result2 = a & b;
      6'bxx1000: result2 = a ^+ b;
      6'bx10000: result2 = ~a;
      6'b100000: result2 = ~b;
      default:   result2 = 4'b0;
   endcase
end // always
```

The logic that implements result1 and result2 is functionally equivalent. However, using nested ternary operator and two assignment statements in result1 is less transparent, and takes more mental effort to understand comparing to a more clear case statement of the result2 logic.

Often, code clarity trumps implementation efficiency. The same piece of code is read by multiple developers over its lifetime. A more clearly-written code is easier to debug, and less prone to contain bugs in general.

Using Lint tools

Lint tools perform pre-simulation and pre-synthesis RTL design rule checking. They help uncover complex and obscure problems that cannot be found by simulation and synthesis tools. Examples of design rules checked by lint tools are clock domain crossing, combinatorial loops, module connectivity, coding style, implied latches, asynchronous resets, overlapping states, and many others. Tip #6 overviews some of the popular Lint tools.

Resources

[1] *"Standard Gotchas: Subtleties in the Verilog and SystemVerilog Standards That Every Engineer Should Know"*, by Stuart Sutherland. Published in the proceedings of SNUG Boston, 2006.
http://www.sutherland-hdl.com/papers/2007-SNUG-SanJose_gotcha_again_paper.pdf

15. Writing Synthesizable Code for FPGAs

The Verilog language reference manual (LRM) provides a rich set of capabilities to describe hardware. However, only a subset of the language is synthesizable for FPGA. Even if a particular language structure is synthesizable, that doesn't guarantee that the code will pass physical implementation for a specific FPGA. Consider the following example.

```
reg [7:0] memory[1:2**22];

initial begin
   memory[1] = 8'h1;
   memory[2] = 8'h2;
end
```

The example will simulate correctly but will result in FPGA physical implementation failure. The code infers four megabytes of memory, which most modern FPGAs don't have. Also, synthesis tools will ignore initial block, which initializes the lowest two bytes of the memory.

This Tip provides several guidelines and recommendations to facilitate writing synthesizable code for FPGAs.

Follow synchronous design methodology

It is recommended that developers adhere to the principles of synchronous design for FPGAs, which include the following:

- Use synchronous design reset. This topic is discussed in more detail in Tip #29
- Avoid using latches; use synchronous registers whenever possible
- Avoid using gated, derived, or divided clocks
- Use clock enables instead of multiple clocks
- Implement proper synchronization of all asynchronous signals

Understand synthesis tool capabilities and limitations

A good knowledge of synthesis tool capabilities and limitations will improve FPGA design performance, logic utilization, and productivity of a designer. It is important to get familiar with the inference rules for a particular FPGA family, what language constructs a synthesis tool ignores or does not support, and the recommended coding style for registers, state machines, tri-states, and other constructs.

Ignored language constructs

Delay values and `timescale compiler directives are ignored by FPGA synthesis tools. Designers often use delay values to make it easier to analyze simulation waveforms, as shown in the following example.

```
`define DLY 1
always @(posedge clk) begin
   data_out <= #`DLY data_in;
end
```

Using such constructs is discouraged because of the potential synthesis and simulation mismatch. For example, if `DLY in the above example exceeds the clock period, the synthesized circuit will likely be functionally incorrect because it will not match simulation results.

Compiler directives such as `celldefine and `endcelldefine are ignored my most of the FPGA synthesis tools.

initial blocks are ignored by FPGA synthesis tools.

Synthesis tools provide various levels of support for Verilog gate-level primitives such as nmos, pmos, cmos, pullup, pulldown, tranif0, tranif1, tran, and others. For example, XST doesn't support Verilog tranif primitive, while Synplify does. Although Xilinx FPGA architecture doesn't have a direct equivalent for gate-level primitives, some synthesis tools convert them to a functionally-equivalent switch. The following is an example of an nmos conversion.

```
module nmos_switch (output out, input data, control);
   assign out = control ? data : 1'bz;
endmodule
```

Unsupported language constructs

User-defined primitives (UDP), repeat, wait, fork/join, deassign, event, force/release statements are not supported by synthesis tools.

Hierarchical references of registers and nets inside the modules are not supported.

```
module my_module1;
  assign my_net = top.my_module2.my_net;
endmodule
```

Level of support for case equality and inequality operators (=== and !==) depends on the synthesis tool. Some synthesis tools will produce an error when they encounter a case equality and inequality operator, while others will convert the operator to the logical equality one (== and !=).

2-state vs. 4-state values

4-state values ('0', '1', 'x', 'z') are inherently not synthesizable. FPGA architecture only supports 2-state values (logic '0' and '1'), and synthesis tools will apply different rules to optimize 'z' and 'x' values during the synthesis process. That will cause a mismatch between synthesis and simulation results (also see Tip #59) and other side effects. Only use 'z' values during the implementation of tri-stated IO buffers.

translate_on/translate_off compiler directives

translate_off and translate_on directives instruct synthesis tools to ignore portions of a Verilog code. These directives are typically used to exclude portions of a behavioral code in the models of IP cores. The following is an example of using these directives.

```
module bram_2k_9 ( input clka, input [0 : 0] wea,
  input [10 : 0] addra,
  output [8 : 0] douta,
  input [8 : 0] dina);

  // synthesis translate_off
  // …
  // behavioral description of bram_2k_9

  // synthesis translate_on
endmodule  // bram_2k_9
```

syn_keep, syn_preserve, syn_noprune compiler directives

syn_keep directive prevents synthesis tools from removing a designated signal. It works on nets and combinatorial logic. This directive is typically used to disable unwanted optimizations and to keep manually created replications. The exact effect of syn_keep might be different between synthesis tools. For example, XST doesn't propagate the constraint to the synthesized netlist, which doesn't prevent optimization by the physical implementation tools.

syn_preserve prevents register optimization. XST also supports "Equivalent Register Removal" option, which is equivalent to syn_preserve.

syn_noprune ensures that if the outputs of an instantiated primitive are not used, the primitive is not optimized. XST also supports the "Optimize Instantiated Primitives" option, which is equivalent to syn_noprune.

parallel_case, full_case compiler directives

If the full_case directive is specified in a case, casex, or casez statement, it prevents synthesis tools from creating additional logic to cover conditions that are not described. Using a full_case directive can potentially result in more compact implementation.

parallel_case directive forces a case statement to be synthesized as a parallel multiplexer instead of the priority-encoded structure. Using parallel_case directives can potentially improve timing performance of a circuit.

The exact effect of full_case and parallel_case directives depends on the synthesis tool. Also, using those directives will also cause synthesis and simulation mismatch. For those reasons, using full_case and parallel_case directives in FPGA designs is not recommended. Instead, designers should change the implementation of a case statement to achieve the same effect. For example, removing overlapping case conditions will render a parallel_case directive unnecessary.

A more detailed discussion of using full_case and parallel_case directives is provided in [1].

`default_nettype compiler directive

Verilog-2001 standard defines a `default_nettype compiler directive. If that directive is assigned to "none", all 1-bit nets must be declared.

```
// no `default_nettype
wire sum; // declaration is not required
assign sum = a + b;
`default_nettype none
wire sum; // must be declared
assign sum = a + b;
```

case, casez, casex statements

Verilog defines case, casez, and casex statements. The case is a multi-way decision statement that tests whether an expression matches one of the case items. The following is an example of a case statement.

```
reg [1:0] sel;
reg [2:0] result;

always @(*)
  case(sel)
    2'b00: result = 3'd0;
    2'b01: result = 3'd1;
    2'b10: result = 3'd2;
  endcase
```

Using case statements instead of a nested if-else will result in more readable code, better logic utilization, and the achieving of higher performance.

The casez and casex statements allow “don’t care” conditions in the case item comparisons. casez treats ‘z’ values as don't-care, and the casex treats both ‘z’ and ‘x’ values as don't care. If any of the bits in the casez or casex expression is a don't-care value, then that bit position will be ignored. The following are examples of casez and casex.

```
reg [1:0] sel;
reg [2:0] result;

// using casez
always @(*)
  casez(sel)
    2’b0?: result = 3’d0;
    2’b10: result = 3’d1;
    2’b11: result = 3’d2;
  endcase

// using casex
always @(*)
  casex(sel)
    2’b0x: result = 3’d0;
    2’b10: result = 3’d1;
    2’b11: result = 3’d2;
  endcase
```

The case expression can be a constant, as illustrated in the following example.

```
reg [1:0] sel;
reg [2:0] result;

always @(*)
  case(1
    ~sel[1]: result = 3’d0;
    sel[1] & ~sel[0]: result = 3’d1;
    sel[1] & sel[0]:  result = 3’d2;
  endcase
```

Using the don't-care condition in casex and casez items may easily lead to overlap or duplication of the case items. Also, using those statements will cause synthesis and simulation mismatch. The following is an example of a case statement with overlapping case items.

```
// casez statement contains overlapped case items
reg [1:0] sel;
reg [2:0] result;

always @(*)
  casez(sel)
    2’b0z: result = 3’d0;
    2’b10: result = 3’d2;
    2’b11: result = 3’d3;
    2’b01: result = 3’d1;  // overlap with 2’b0z
  endcase
```

Overlapping or duplication of the case items is permitted and in most cases will not trigger any simulation or synthesis warning. This is a dangerous and hard-to-debug condition. It is recommended that the developer avoids using casex and casez statements altogether.

Adding a default clause to case statements is a simple way to avoid the range of problems. There are two ways to achieve the same effect, shown in the following examples.

```
reg [1:0] sel;
reg [2:0] result;

// using default clause
always @(*)
  case(sel)
    2’b00: result = 3’d0;
    2’b01: result = 3’d1;
    2’b10: result = 3’d2;
    default: result = 3’d3;
  endcase

// assigning default value before case statement
always @(*)
  result = 3’d3;
  case(sel)
    2’b00: result = 3’d0;
    2’b01: result = 3’d1;
    2’b10: result = 3’d2;
  endcase
```

Mixing blocking and non-blocking assignments in always blocks

Verilog specifies two types of assignments that can appear in always blocks: blocking and non-blocking. Blocking and non-blocking assignments are used to describe combinatorial and sequential logic respectively. Blocking and non-blocking assignments should never be mixed in the same always block. Doing that can cause unpredictable synthesis and simulation results. In many cases synthesis tools will not produce any warning, but synthesized logic will be incorrect. The following two code examples illustrate mixed use of blocking and non-blocking assignments.

```
reg blocking, non_blocking;
always @(posedge clk) begin
   if(reset) begin
      blocking    =  0;
      non_blocking <= 0;
   end
   else begin
      blocking     = ^data;
      non_blocking <= |data;
   end
end

always @(*) begin
   blocking = ^data;
   non_blocking <= |data;
end
```

The correct way is to implement the above examples is as follows:

```
reg blocking, non_blocking;
always @(posedge clk) begin
   if(reset) begin
      non_blocking <= 0;
   end
   else begin
      non_blocking <= |data;
   end
end
always @(*) begin
   blocking = ^data;
end
```

Multiple blocking assignments

Blocking assignments in an always block are executed in order of their appearance. Although it's often convenient, it is recommended that the developer limits the use of multiple blocking assignments to modify the same variable inside an always block. The following two code examples show potential problems with using multiple blocking assignments.

```
reg signal_a, signal_b, signal_c, signal_d;

always (*) begin
   signal_a = signal_b & signal_c;
   // …
   // additional code
   signal_d = signal_a & signal_e;
end
```

Accidentally changing the order of signal_a and signal_d assignments will break the functionality of a signal_d.

```
reg [15:0] signal_a, signal_b;

always (*) begin
   signal_a[15:12] = 4'b0;
   // …
   // additional code
   signal_a = signal_b;
end
```

The last assignment to signal_a takes priority, and bits [15:12] of signal_a will never be reset.

Using named always blocks

The following is an example of a named always block.

```
reg reg_unnamed;

always @(posedge clk) begin : myname
   // only visible in the "myname" block
   reg reg_named;

   // post-synthesis name : myname.reg_named
   reg_named <= data_in;
   // post-synthesis name : reg_unnamed
```

```
   reg_unnamed <= ~reg_named;

end // always
```

Named blocks can be useful in several situations. Because the always block is uniquely identified, it is easier to find it in the simulation. Also, limiting the scope of the variables allows for the reuse of the same variable name.

Resources

[1] *"'full_case parallel_case', the Evil Twins of Verilog Synthesis"*, by Clifford Cummings. Published in the proceedings of SNUG Boston, 1999.

16. Instantiation vs. Inference

Instantiation and inference are two different methods of adding components to an FPGA design. Each method offers its advantages, but also has drawbacks.

Component instantiation directly references an FPGA library primitive or macro in HDL. The main advantage of the instantiation method is that it provides a complete control of all the component features, and FPGA resources it will use. Also, instantiation makes it simpler to floorplan the component. Instantiation has to be used for complex components that cannot be inferred. For example, it is not possible to infer Xilinx MMCM primitive, and the only way to use it is by direct instantiation. Xilinx CoreGen utility can be used to generate more complex components for instantiation, such as MMCM, DSP48, or FIFOs. Another reason for direct instantiation is when synthesis tool is unable to correctly infer the component.

Component inference refers to writing a generic RTL description of a logic circuit behavior that the synthesis tool automatically converts into FPGA-specific primitives. Inference results in more portable code that can be used to target different FPGA architectures. It also produces more compact and readable code. The main disadvantage of the inference method is that the inference rules vary between different synthesis tools, so that the same RTL may result in a synthesis error, or a functionally different circuit. Xilinx recommends using inference method whenever possible.

Inferring registers

All synthesis tools are capable to correctly infer registers for Xilinx FPGAs with rising or falling edge clock, clock enable, and synchronous or asynchronous reset. It is recommended to use a non-blocking assignment (<=) in Verilog always block for a register inference. The following are several Verilog examples of inferring registers.

```
module register_inference( input  clk,areset,sreset,
            input  [4:0] d_in,
            output reg [4:0] d_out);

   // positive edge clock, asynchronous active-high reset
   always @( posedge clk , posedge areset)
      if( areset )
         d_out[0] <= 1'b0;
      else
         d_out[0] <= d_in[0];

   // negative edge clock, asynchronous active-low reset
   always @( negedge clk , negedge areset)
      if( ~areset )
         d_out[1] <= 1'b0;
      else
         d_out[1] <= d_in[1];

   // negative edge clock, no reset
   always @( negedge clk )
      d_out[2] <= d_in[2];

   // Positive edge clock, synchronous active-high reset.
   // The register is initialized with '1' upon reset.
   // The circuit is implemented using a register and a LUT.
   always @( posedge clk )
      if( sreset )
         d_out[3] <= 1'b1;
      else
         d_out[3] <= d_in[3];

// Positive edge clock, both synchronous and asynchronous resets
// The circuit is implemented using a register and additional logic
always @( posedge clk , posedge areset)
      if( sreset )
         d_out[4] <= 1'b1;
      else if( areset )
         d_out[4] <= 1'b0;
      else
         d_out[4] <= d_in[4];

endmodule // register_inference
```

Register initialization

There are several ways to perform register initialization, illustrated in the following examples.

```
// Register initialization in initial block.
// That will work in simulation. However, initial block is ignored during synthesis

initial begin
   my_reg = 1'b0;
end

// Register initialization during declaration
reg my_reg = 1'b0;


// Register initialization upon reset
always @( posedge clk , posedge reset)
      if( reset )
         my_reg <= 1'b0;
      else
         my_reg <= d_in;
```

Inferring memories

Xilinx FPGAs provide several types of embedded memories that can be inferred using various techniques. Tip #34 provides more detailed description of those techniques.

Inferring Shift Register LUT (SRL)

Xilinx FPGAs contain dedicated Shift Register LUT (SRL16 or SRL32) primitives to conveniently implement shift registers. SRL primitives are more limited than general-purpose shift registers. They don't have a reset, and only have access to serial out bit. The following is an example of inferring an SRL.

```
wire srl1_out;
reg [7:0] srl1;

always @(posedge clk)
   srl1 <= srl_enable ? {srl1[6:0] , d_in[0]} : srl1;
assign srl1_out = srl1[7];
```

Inferring IO

If a port in a top-level module doesn't contain any constraints, it'll be implemented using an IO with default characteristics, which depend on the FPGA family. For Xilinx FPGAs, a synthesis tool will automatically infer IBUF, OBUF, OBUFDS, or another primitive that best describes the IO. Designers can specify port characteristics by using Verilog attributes or UCF constraints. However, Verilog attributes are specific to a synthesis tool, and UCF constraints are specific to FPGA family. The following are examples of Verilog attributes related to the IO.

```
Synopsys Synplify: xc_padtype, xc_pullup, xc_slow, xc_fast, xc_loc,
          syn_useioff.
Xilinx XST: (* IOSTANDARD = "iostandard_name" *)
         * SLOW = "{TRUE|FALSE}" *)
```

Inferring IOB Registers

Every IO block in Xilinx FPGAs contains storage elements, referred to as IOB registers. The available options are input register, and output register for single or dual-data rate (DDR) outputs. Using IOB registers significantly decreases clock-to-input/output data time, which improves IO performance.

Placing a register in the IOB is not guaranteed, and it requires following a few rules. Register to be pushed into IOB, including output and tri-state enable, has to have a fanout of 1. Registers with a higher fanout have to be replicated. All registers that are packed into the same IOB must share the same clock and reset signal.

Designers can specify the following UCF constraint:

```
INST <register_instance_name> IOB = TRUE|FALSE;
```

The other way is to use a MAP "Pack I/O Registers into IOBs" property, which applies globally to all IOs. The command line MAP option is -pr {i|o|b}. "i" is only for input registers, "o" only for output registers, and "b" is for both input and output.

In most cases inferring DDR registers in the IOB requires an explicit instantiation of a primitive. However, the following example might work in a few cases.

```
module ddr_iob( input clk,reset,
            input ddr_in, output  ddr_out);
reg q1, q2;
assign ddr_out = q1 & q2;

always @ (posedge clk, posedge reset)
      if (areset)
         q1 <=1'b0;
      else
         q1 <= ddr in;
```

```
   always @ (negedge clk, posedge reset)
      if (areset)
         q2 <=1'b0;
      else
         q2 <= ddr_in;
endmodule
```

Inferring latches

Latches are inferred from incomplete conditional expressions. For example, when *case* or *if* statements do not cover all possible input conditions, synthesis tool might infer a latch to hold the output if a new output value is not assigned. Xilinx XST will also produce a warning. Latch inference is shown in the following example.

```
module latch( input  clk, input latch_en, latch_data,
           output reg latch_out);
   always @ (latch_en)
      if (latch_en)
         latch_out = latch_data;

endmodule // latch
```

In many cases latches are inferred unintentionally due to poor coding style practices. One way to avoid unintentional latches due to incomplete sensitivity list is to use Verilog-2001 implicit event expression list @(*) or @* in the *always* blocks. Also, always assign the default *case* or final *else* statement to a "don't care" or other default value.

In general, it is recommended to avoid using latches in FPGA designs. Tip #50 provides more information on problems related to using latches, and different methods of porting them.

Tri-stated buffers

The following is an example of how to infer a tri-stated output buffer in Xilinx FPGAs.

```
module tri_buffer( output tri_out, input tri_en, tri_in  );
   assign tri_out = tri_en ? tri_in : 1'bz;
endmodule // tri_buffer
```

It is possible to use more complex expressions to infer tri-stated buffers. However, it's not recommended to mix tri-stated buffers with other logic, such as conditional if and case statements. The following is an example of not recommended way of inferring tri-stated buffers.

```
module tri_buffer2( output reg tri_out,
  input tri_en1, tri_en2, tri_in);
   always @ (*)
      if ( tri_en1 & tri_en2 )
         tri_out = 1'bz;
      else
         tri_out<=tri_in;
endmodule // tri_buffer2
```

Another recommendation is to include logic that infers tri-stated buffers at the design top-level.

Tip #54 provides more examples on using tri-stated buffers in FPGA designs.

Inferring complex arithmetic blocks

It is recommended not to attempt to infer complex arithmetic blocks, such as multipliers and dividers. That particular implementation might work, but it unlikely to be portable across different FPGA architectures and synthesis tools. Xilinx supports very configurable multiplier, divider, and other arithmetic cores that can be generated using CoreGen utility.

17. Mixed Use of Verilog and VHDL

The need to use both Verilog and VHDL languages in the same design arises in several situations. A design team might decide for various reasons to switch to another language for the next project, while reusing some of the existing functional modules. Most often, it is the integration of a 3'rd party IP core written in a different language that results in a mixed language design. One example is Xilinx Microblaze processor, and most of its peripherals, which are written in VHDL, and need to be integrated into a Verilog project.

Xilinx XST synthesis tool and ISIM simulator provide full support of mixed Verilog/VHDL projects.

The following is a simple example that illustrates mixed use of Verilog and VHDL. The module lfsr_counter is written in VHDL, and instantiated in the module top written in Verilog. Verilog modules can also be instantiated in VHDL.

```
// Verilog implementation of the top module
module top(input enable, rst, clk,
         output reg done_qo);
   always @(posedge clk, posedge rst)
```

```
      if(rst)
         done_qo <= 'b0;
      else
         done_qo <= lfsr_done;

   lfsr_counter lfsr_counter(
         enable , rst, clk, lfsr_done);

endmodule // top

-- VHDL implementation of the lfsr_counter
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity lfsr_counter is
  port (
   ce , rst, clk : in std_logic;
   lfsr_done : out std_logic);
end lfsr_counter;

architecture imp_lfsr_counter of lfsr_counter is
  signal lfsr: std_logic_vector (10 downto 0);
  signal d0, lfsr_equal: std_logic;
begin
  d0 <= lfsr(10) xnor lfsr(8) ;

  process(lfsr) begin
   if(lfsr = x"359") then
     lfsr_equal <= '1';
   else
     lfsr_equal <= '0';
   end if;
  end process;

   process (clk,rst) begin
     if (rst = '1') then
      lfsr <= b"00000000000";
      lfsr_done <= '0';
     elsif (clk'EVENT and clk = '1') then
      lfsr_done <= lfsr_equal;
      if (ce = '1') then
        if(lfsr_equal = '1') then
         lfsr <= b"00000000000";
        else
         lfsr <= lfsr(9 downto 0) & d0;
        end if;
      end if;
     end if;
   end process;
end architecture imp_lfsr_counter;
```

Source code and project files for this example can be found in the accompanied web site.

As long as all boundary rules are met, the process of mixing Verilog and VHDL modules is straightforward. Each synthesis tool might have slightly different rules and limitations. Xilinx XST user guide defines the following rules:

- Mixing of VHDL and Verilog is restricted to design unit (cell) instantiation only. A VHDL design can instantiate a Verilog module, and a Verilog design can instantiate a VHDL entity. Any other kind of mixing between VHDL and Verilog is not supported.
- In a VHDL design, a restricted subset of VHDL types, generics and ports is allowed on the boundary to a Verilog module. Similarly, in a Verilog design, a restricted subset of Verilog types, parameters and ports is allowed on the boundary to a VHDL entity or configuration.
- Configuration specification, direct instantiation and component configurations are not supported for a Verilog module instantiation in VHDL.
- VHDL and Verilog libraries are logically unified
- To instantiate a VHDL entity in the Verilog module, declare a module name with the same as name as the VHDL entity that you want to instantiate, and perform a normal Verilog instantiation. The name is case sensitive.
- To instantiate a Verilog module in a VHDL design, declare a VHDL component with the same name as the Verilog module to be instantiated, and instantiate the Verilog component as if you were instantiating a VHDL component.

Note that simulating mixed language designs with commercial simulators might require separate license.

18. Verilog Versions: Verilog-95, Verilog-2001, and SystemVerilog

Verilog started out as a proprietary hardware modeling language in 1984 and enjoyed considerable success. In 1990, Cadence Design Systems transferred the language into the public domain as part of the efforts to compete with VHDL. In 1995, Verilog became an IEEE standard 1364-1995, commonly known as Verilog-95.

Verilog-95 had been evolving, and in -2001 IEEE released another standard, 1364-2001, also known as Verilog-2001. It contained a lot of extensions that cover the shortcomings of the original standard, and introduced several new language features. In 2005 IEEE published a 1364-2005 standard, known as Verilog 2005. It included several specification corrections and clarifications, and a few new language features.

IEEE published several SystemVerilog standards. The latest one is 1800-2009, which was published in 2009. SystemVerilog is a superset of Verilog, and contains features intended for better support of design verification, improving simulation performance, and making the language more powerful and easy to use.

Verilog-2001 is a predominant Verilog version used by the majority of FPGA designers, and supported by all synthesis and simulation tools.

Verilog-2001

Xilinx XST and other FPGA synthesis tools have an option to enable or disable Verilog-2001 constructs. In XST it's done by using *-verilog2001* command line option . In Synplify, use "*set_option -vlog_std v2001"* command .

The following is a brief overview of some of the most important differences between Verilog-95 and Verilog-2001.

Verilog-2001 added explicit support for two's complement signed arithmetic. In Verilog-95 developers needed to hand-code signed operations by using bit-level manipulations. The same function under Verilog-2001 can be described by using built-in operators and keywords.

Auto-extending 'bz and 'bx assignments. In Verilog-95 the following code

```
  wire [63:0] mydata = 'bz;
```

will assign the value of `z` to `mydata[31:0]`, and zero to `mydata[63:32]`.

Verilog-2001 will extend 'bz and 'bx assignments to the full width of the variable.

A `generate` construct allows Verilog-2001 to control instance and statement instantiation using `if/else/case` control. Using `generate` construct, developers can easily instantiate an array of instances with correct connectivity. The following are several examples of using `generate` construct.

```
// An array of instances
module adder_array(input [63:0] a,b, output [63:0] sum);

generate
genvar ix;
   for (ix=0; ix<=7; ix=ix+1) begin : adder_array
      adder add (a[8*ix+7 -: 8],
                 b[8*ix+7 -: 8],
                 sum[8*ix+7 -: 8]);
   end
endgenerate
endmodule // adder_array

module adder(input [7:0] a,b, output [7:0] sum );

   assign sum = a + b;

endmodule // adder

// If...else statements
module adder_array(input [63:0] a,b, output [63:0] sum);

parameter WIDTH = 4;

generate
if (WIDTH < 64) begin : adder_gen2
   assign sum[63 -: (64-WIDTH)] = 'b0;

   adder # (WIDTH) adder1 (a[WIDTH-1 -: WIDTH], b[WIDTH-1 -: WIDTH],sum[WIDTH-1 -: WIDTH]);
end
else begin : adder_default
   adder # (64) adder1 (a, b, sum);
end
endgenerate
endmodule // adder_array

// case statement
module adder_array(input [63:0] a,b, output [63:0] sum);
generate
case (WIDTH)
   1: begin : case1
         assign sum[63 -: 63] = 'b0;
         adder #(WIDTH) adder1 (a[0], b[0], sum[0]);
   end
   default: begin : def
    adder # (64) adder1 (a, b, sum);
   end
endcase
endgenerate
endmodule // adder_array
```

Verilog-2001 adds support for multi-dimensional arrays. Synthesis tools place several restrictions on synthesis of multi-dimensional arrays. For example, XST supports arrays of up to two dimensions. It doesn't allow selecting more than one element of an array at one time. Multi-dimensional arrays cannot be passed to tasks or functions.

The following code example describes an array of 256 x 16 wire elements, 4-bit wide each:

```
wire [3:0] multi_dim_array [0:255][0:15];
```

More concise port declaration.

```
// Verilog-95
module adder(a,b,sum);
     input [7:0] a,b;
     output [7:0] sum;
assign sum = a + b;
endmodule // adder

// Verilog-2001
module adder(input [7:0] a,b, output [7:0] sum );
assign sum = a + b;
endmodule // adder
```

Verilog-2001 adds support for exponential or power operator "**". It's convenient in many cases, for example, for determining memory depth calculations. Synthesis tools support exponential with several restrictions. XST requires that both operands are constants, with the second one is non-negative. The values x and z are not allowed. If the first operand is 2, the second can be variable.

The following are code examples of using the exponent.

```
   localparam BASE = 3,
              EXP = 4;
   assign exp_out2 = BASE**EXP;
   // this code is synthesized as a shift register
   assign exp_out1 = 2**exp_in;
```

Comma-separated and combinational sensitivity list.

```
// Verilog-95
always @(a or b);
   sum = a + b;

// Verilog-2001
always @(a,b);
   sum = a + b;
always @(*);
   sum = a + b;
```

Required net declarations.

Verilog-95 requires that all 1-bit nets, that are not ports, and are driven by a continuous assignment, must be declared. That requirement is removed in Verilog-2001. Instead, the Verilog-2001 standard adds a new `` `default_nettype `` compiler directive. If that directive is assigned to `"none,"` all 1-bit nets must be declared.

```
// Verilog-95
wire sum;
assign sum = a + b;

// Verilog-2001
wire sum; // not required
assign sum = a + b;
`default_nettype none
wire sum; // must be declared
assign sum = a + b;
```

SystemVerilog

SystemVerilog standard is designed to be a unified hardware design, specification, and verification language. It's a large standard that consists of several parts: design specification methods, embedded assertion language, functional coverage, object oriented programming, and constraints. The main goal of the SystemVerilog is to create a unified design and verification environment by taking the best features and building on the strengths of Verilog, VHDL, and hardware verification languages. SystemVerilog enables a major leap in productivity by combining a diverse set of tools and methods into one, removing fragmentation of a system design between software and hardware engineers, and sharing the results.

SystemVerilog contains several extensions to the existing Verilog specifications that are designed to reduce the number of lines of code, encourage design reuse, and improve simulation performance. SystemVerilog is also fully backwards compatible with previous Verilog versions.

SystemVerilog is supported by most of the commercial simulators: ModelSim, VCS, NCSim, and others. Synthesizable portion of the SystemVerilog standard is supported by Synplify and Precision synthesis tools. At the time of writing of this book, Xilinx XST doesn't provide any SystemVerilog support.

Design synthesis and verification environments are often written using both SystemVerilog and Verilog languages. One approach used in large designs is illustrated in the following figure.

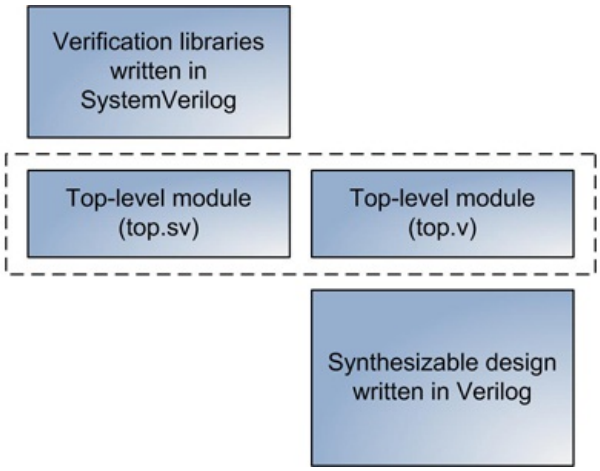

Figure 1: Mixed use of Verilog and SystemVerilog

The top-level module is implemented in both Verilog and SystemVerilog. That makes it compatible with all FPGA synthesis tools that don't support SystemVerilog. The design can be readily integrated with the verification libraries written in SystemVerilog.

The following is a brief overview of some of the features unique to SystemVerilog.

Data types

SystemVerilog defines the following new data types:

Table 1: SystemVerilog data types

Data type	*Description*
int	32-bit signed integer, 2-state
shortint	16-bit signed integer, 2-state
longint	64-bit signed integer, 2-state
bit	User defined vector size, 2-state
byte	8-bit signed integer, 2-state
logic	User defined vector size, 2-state

2-state data types accept 0 and 1 values.
4-state data types accept 0,1, z, and x values.

SystemVerilog provides the ability to create a custom grouping of signals and variables, similar to C programming language. It defines the following features to support grouping: Typedefs, Enumerated types, Structures, Unions, and Static casting.

Below are several examples of SystemVerilog data types.

```
bit x;
enum {STATE1, STATE2, STATE3} state;
typedef enum { red=0,green,yellow } Colors;
integer a,b;
a = green*3 // 3 is assigned to a
b = yellow + green; // 3 is assigned to b
struct {
   bit [31:0] characteristic;
   bit [31:0] mantissa;
      } float_num;
float_num.characteristic = 32'h1234_5678;
float_num.mantissa = 32'h0000_0010;
typedef union{
   int u1;
   shortint u2;
} my_union;
```

SystemVerilog provides strongly typed capabilities to avoid multiple interpretations and race conditions that plague Verilog designs.

Port connectivity

SystemVerilog provides ".name" and implicit ".*" methods to describe port connectivity. That allows significant code compaction.

```
// Verilog
adder add (a,b,sum);

// SystemVerilog
adder add (.a,.b,.sum);

adder add (.*);
```

Interfaces and Modports

The `interface` and `modport` SystemVerilog constructs enable encapsulation of port lists and interconnect between module instances. The following is a simple example:

```
// Define the interface
interface adder_if;
   logic [7:0] a, b;
   logic [7:0] sum;
endinterface: adder_if

module top;

   // Instantiate the interface
   adder_if adder_if1();
   adder_if adder_if2();

   // Connect the interface to the module instance
   adder add1 (adder_if1);
   adder add2 (adder_if2);

endmodule

module adder(adder_if if);

   // access the interface
   assign if.sum = if.a + if.b;

endmodule // adder
```

Object Oriented Programming (OOP)

SystemVerilog enables better support for Object Oriented Programming (OOP). OOP raises the level of abstraction by building data structures with self-contained functionality that allows data encapsulation, hiding implementation details, and enhances the re-usability of the code.

The following table shows SystemVerilog constructs and structures that enable OOP.

Table 2: SystemVerilog OOP structure

	Verilog	*SystemVerilog*
Block definition	Module	Module, class
Block instance	Instance	Object

Block name	Instance name	Object handle
Data types	Registers and wires	Variables
Functionality	Tasks, functions, always block, initial block	Tasks, functions

Constraints, Coverage, and Randomization

SystemVerilog provides extensive support for coverage-driven verification, directed and constrained random testbench development.

Assertions

Assertions augment functional coverage by providing multiple observation points within the design. Usually, both RTL designers and verification engineers instrument the design with assertions. Assertions can be placed both inside the RTL code, which makes updates and management easier, or outside, to keep synthesizable and behavioral parts of the code separate. SystemVerilog provides assertion specification that is used in verification environments of many ASIC designs. SystemVerilog assertions can be applied to the following elements of a design: variable declarations, conditional statements, internal interfaces, and state machines.

Assertions are supported by several commercial simulators, for example ModeSim and VCS.

A small subset of the SystemVerilog assertions specification allows synthesizable assertions. Unfortunately, only a handful of FPGA design tools support synthesizable assertions. One of them is Synopsys Identify Pro® that has assertion synthesis and debug capabilities.

19. HDL Code Editors

Source code editors have features specifically designed to speed up, simplify, and automate creation, browsing, and searching of the source code. Choosing the right source code editor can make a significant impact on personal productivity. Important features of a code editor are: syntax highlighting, auto complete, bracket matching, and management of large projects. Other useful features include: interfacing with different tools such as source control, build manager, or script interpreters, cross platform support, and customizations. Because Verilog and VHDL are not as widely used as programming and scripting languages such as C, Java, and Perl, not all source code editors support them.

The choice of a code editor is mainly driven by personal preferences, availability of specific features or lack of thereof, and operating system support. The following is a brief overview and basic comparison of some of the popular code editors that support Verilog and VHDL languages.

Vi/Vim

URL: http://vim.sourceforge.net
OS: Unix/Linux, Windows, Mac
License: open source
Vi is a lightweight command-line based editor. Vim is an extended version of Vi, and stands for "Vi Improved." Vi/Vim is designed to perform all editing functions without using a mouse or menus, which allows for very fast code editing. Some Vim versions, for example gVim, have basic GUI support.

Vi/Vim macros and plug-ins enable a high level of customization. There are several Vim macros for Verilog and VHDL syntax highlighting.

Emacs/XEmacs

URL: http://www.gnu.org/software/emacs
http://www.veripool.org/wiki/verilog-mode
OS: Unix/Linux, Windows, Mac
License: open source
Emacs is another venerable code editor that has been in use for over three decades. Both Emacs and Vi editors are undisputed leaders in terms of adoption by ASIC and FPGA developers on Linux platforms.

XEmacs is a version of Emacs that offers a GUI toolbar, and other features.

Both Emacs and Vi were originally developed for Unix, and later ported for Windows and Mac operating systems.

A popular Verilog mode for Emacs is available on Veripool.org. It provides context-sensitive highlighting, auto indenting, and macro expansion capabilities. It supports all Verilog and SystemVerilog versions. It is written and actively maintained by Michael McNamara and Wilson Snyder.

UltraEdit

URL: http://www.ultraedit.com
OS: Windows, Linux, Mac
License: commercial
Vendor: IDM Computer Solutions, Inc

UltraEdit is a popular and highly acclaimed text editor. Its main features include: macros, tag lists, syntax highlighting, and autocomplete files for different programming languages, including Verilog and VHDL, project management, regular expressions for search-and-replace, remote editing of files via FTP, integrated scripting language, file encryption/decryption, and many more.

Crimson Editor

URL: http://www.crimsoneditor.com
OS: Windows
License: open source (GPL)
Crimson Editor is a source code editor for Windows. It features a small installation size, fast loading time, and syntax highlighting for many programming languages, including Verilog and VHDL.

Notepad++

URL: http://notepad-plus-plus.org
License: open source (GPL)
OS: Windows
Notepad++ is a popular code editor that supports Verilog and VHDL syntax highlighting. It is written in C++, and designed for high execution speed.

IDE Code Editors

Most of the IDE tools used during FPGA design have an integrated code editor. Examples are Xilinx ISE, Mentor ModelSim, and Synopsys Synplify. Integrated code editors offer unique features such as hyperlinking, and excellent project management capabilities.

20. FPGA Clocking Resources

This Tip uses Xilinx Virtex-6 as an example to overview different types of FPGA clocking resources: dedicated clock routing, clock buffers, and clock managers.

FPGAs provide dedicated low-skew routing resources for clocks. A skew is defined as the difference in arrival times of a clock edge to synchronous logic elements connected to it. Not only can a skew impact the achievable clock speed of a circuit but, in the worst case, it can cause incorrect operation of the circuit. Low-skew clock routing can also be used for non-clock signals, such as reset or high-fanout controls.

Xilinx FPGA die is divided into several clock regions as is illustrated in the following figure.

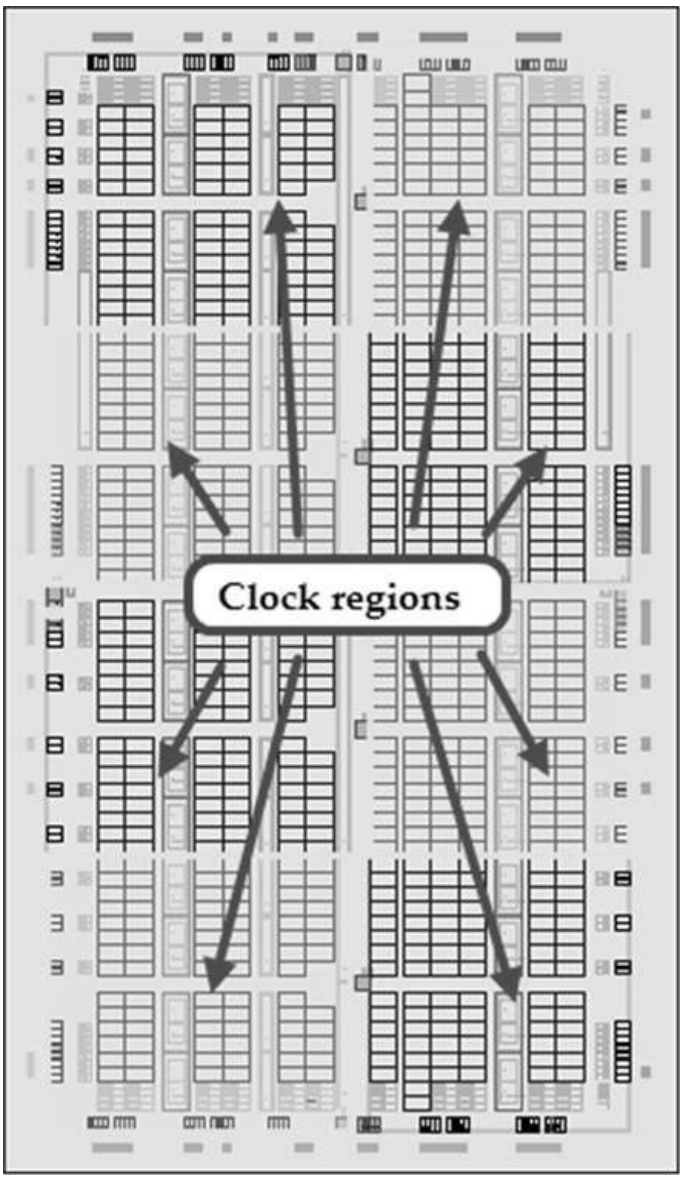

Figure 1: Eight clock regions

Virtex-6 FPGAs have between six to 18 clock regions, depending on the device.

There are three types of clocks: global, which can drive synchronous logic on the entire die; regional, which can drive logic in specific and adjacent regions; and IO, which can serve the logic specific to that IO.

Tradeoffs for selecting clock type are performance and availability. IO clocks are the fastest, but have a very limited amount of logic they can drive. Global clocks are easier to use, because they can reach the entire die. However, there are more regional clocks than global clocks.

Clock buffers

Xilinx provides several clock buffer primitives to support different clock types, listed in the following table:

Table 1: Clock buffers for different clock types

Buffer type	***Primitive***
Single-ended input buffer	IBUFG, BUFGP*
Differential input buffer	IBUFDS, IBUFGDS, IBUFGDS_DIFF_OUT

Global buffer	BUFG, BUFGP*, BUFGCE, BUFGMUX, BUFGCTRL
Regional buffer	BUFR, BUFH, BUFHCE
Local clock buffer	BUFIO

*BUFGP includes both IBUFG and BUFG

Synthesis tools will automatically infer BUFG or BUFGP primitive for simple clocks, as illustrated in the following example.

```
module clock_buffer_inference(input clk, reset,
                 input data_in,
                 output reg data_out);

always @(posedge clk, posedge reset)
   if (reset)
    data_out <= 0;
   else
    data_out <= data_in;
endmodule // clock_buffer_inference
```

Design summary of XST synthesis report indicates that BUFGP clock buffer was used.

```
=====================================
*  Design Summary
=====================================
Top Level Output File Name       : clock_buffer_inference.ngc

Primitive and Black Box Usage:
------------------------------
# FlipFlops/Latches         : 1
#      FDC                  : 1
# Clock Buffers             : 1
#      BUFGP                : 1
# IO Buffers                : 3
#      IBUF                 : 2
#      OBUF                 : 1
```

It is possible to disable automatic insertion of a global buffer. XST provides "`Add I/O Buffers (-iobuf)`" option to enable or disable automatic global buffer insertion.

Regional and local clocks, differential clocks, and circuits with clock-enable signal require explicit instantiation of appropriate clock buffer primitives.

Mixed-mode Clock Manager

Mixed-mode Clock Manager (MMCM) is a clock management primitive in Xilinx Virtex-6 devices. Its main functions are support for clock network deskew, frequency synthesis, phase shifting, and jitter reduction. MMCM leverages features from PLL and Digital Clock Manager (DCM) primitives in previous Xilinx FPGA families. The following figure shows a detailed MMCM block diagram.

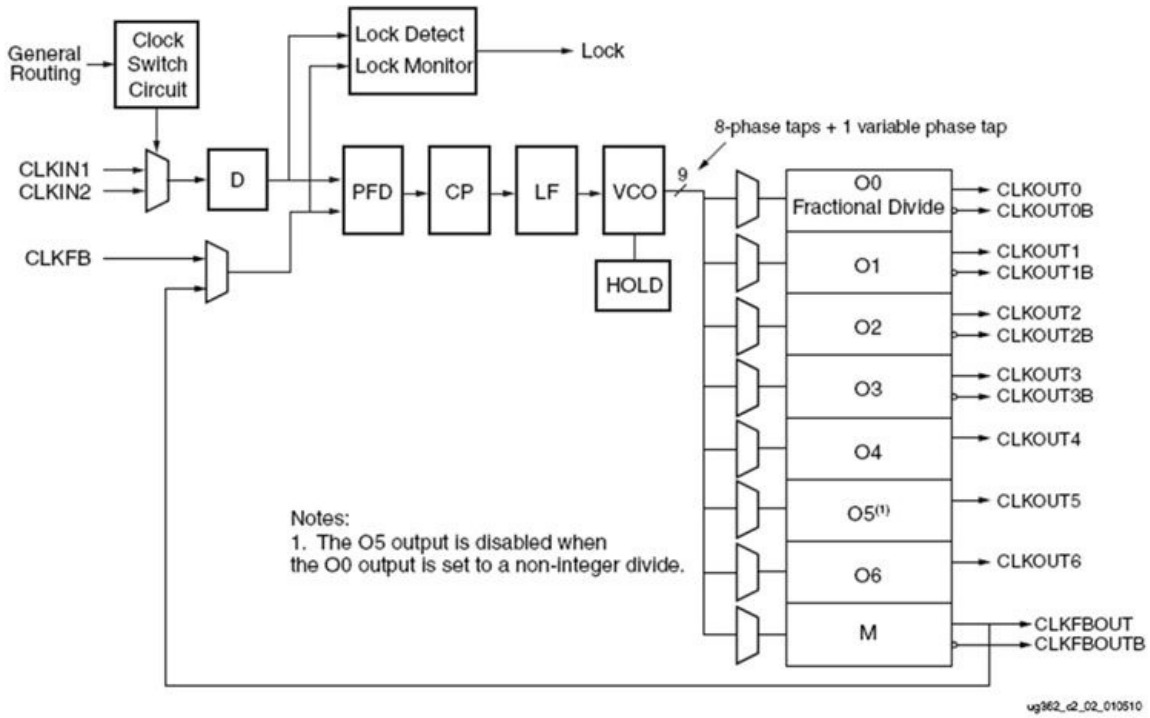

Figure 2: MMCM block diagram (Source: Xilinx User Guide 362)

At the heart of the MMCM is a voltage-controlled oscillator (VCO) with a frequency ranging between 400 MHz and 1600 MHz. VCO is connected to programmable frequency dividers and eight equally-spaced output phases (0°, 45°, 90°, 135°, 180°, 225°, 270°, and 315°). MMCM also supports dynamic, fine-grained phase shifting.

The MMCM can also serve as a frequency synthesizer for a wide range of frequencies and clock duty cycles.

The following is an example of using MMCM.

```
module clock_mmcm(input clk, reset,
             input data_in,
```

```
            output reg data_out);
  wire clk_i,locked,clkfb;
  clk_mmcm mmcm(
   .CLK_IN1          (clk),
   .CLKFB_IN          (clkfb),
   .CLK_OUT1          (clk_i),
   .CLKFB_OUT         (clkfb),
   .RESET            (reset),
   .LOCKED           (locked));

always @(posedge clk_i, posedge locked)
if (locked)
  data_out <= 0;
else
  data_out <= data_in;
endmodule // clock_mmcm
```

Design summary of XST synthesis report shows that the circuit used MMCM, IBUFG, and BUFG clocking resources.

```
=================================
*  Design Summary
=================================
Top Level Output File Name       : clock_mmcm.ngc
Primitive and Black Box Usage:
------------------------------
# FlipFlops/Latches         : 1
#      FDC                  : 1
# Clock Buffers             : 1
#      BUFG                 : 1
# IO Buffers                : 4
#      IBUF                 : 2
#      IBUFG                : 1
#      OBUF                 : 1
# Others                    : 1
#      MMCM_ADV             : 1
```

MMCM is a highly configurable primitive. Instead of manual instantiation of MMCM, it is recommended to use Clock Wizard in Xilinx CoreGen tool to create a customized clock manager instance. Clock Wizard provides an easy-to-use interface to configure attributes, and include all the necessary clock buffers, which ensures correct use of the MMCM primitive.

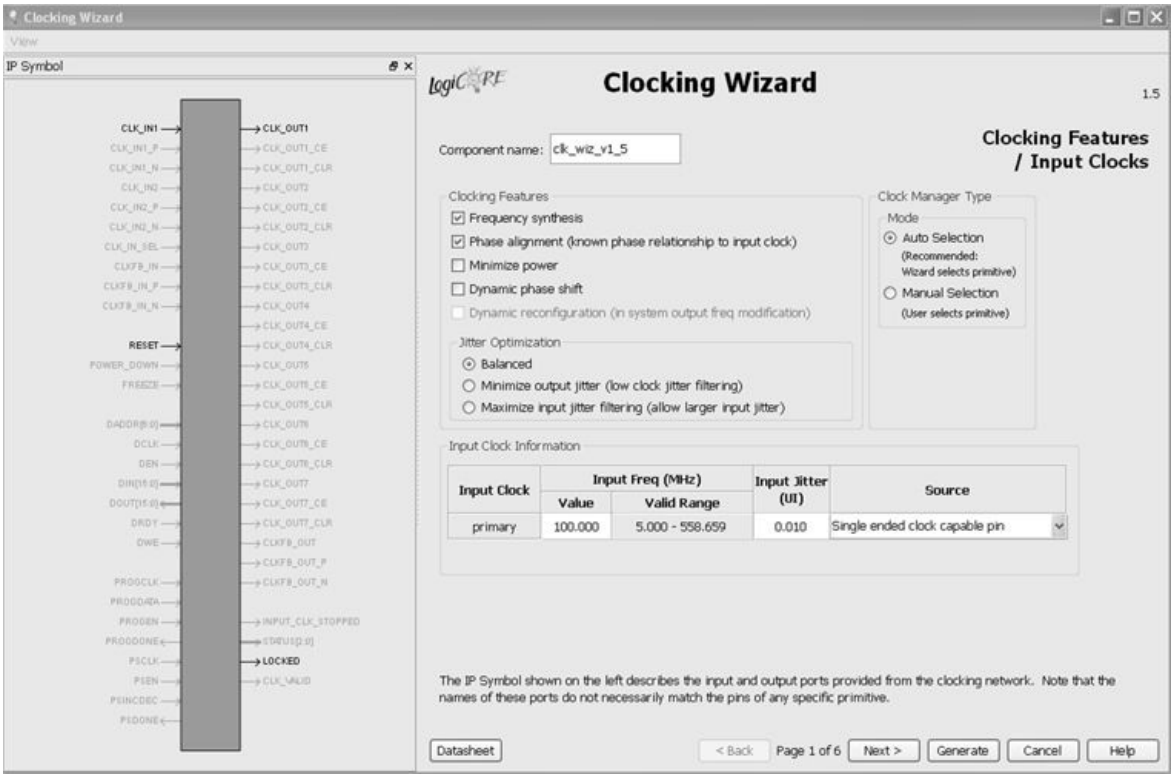

Figure 3: Xilinx CoreGen Clocking Wizard

Digital Clock Manager

Digital Clock Manager (DCM) is a clock management primitive used in Spartan-6, Virtex-5, and earlier Xilinx FPGA families. Its main functions are eliminating clock insertion delay, frequency synthesis, and phase shift. DCM has a very different design than MMCM or PLL. DCM block diagram is shown in the following figure.

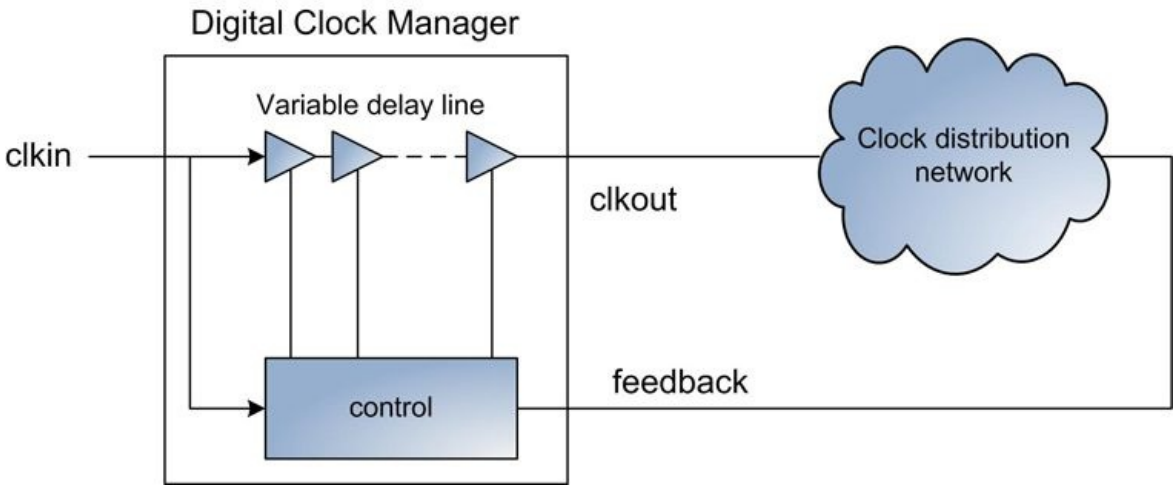

Figure 4: DCM block diagram

The control module inside the DCM compares the phases of the input clock and the feedback from the clock distribution network to estimate the delay. Then, the control module adjusts the delay between the input and output clocks using a variable delay line to achieve phase alignment.

Delay removal is shown in a timing report as a negative delay from the output to input clock of a DCM, as is illustrated in the following figure.

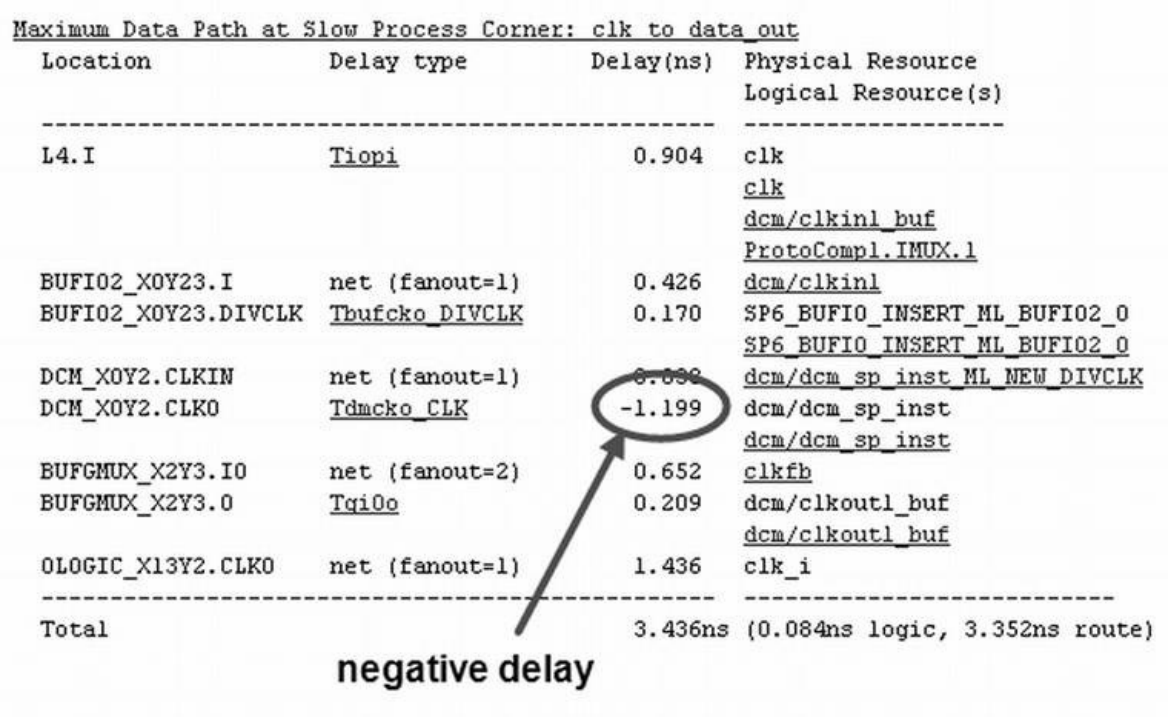

Figure 5: DCM negative delay

Clock multiplexing

Xilinx provides a BUFGMUX primitive that can multiplex between two global clock sources. It also ensures a glitch-free operation when the input clock selection is changed.

Resources

[1] Xilinx Virtex-6 FPGA Clocking Resources, User Guide UG362
http://www.xilinx.com/support/documentation/user_guides/ug362.pdf

21. Designing a Clocking Scheme

Designing a clocking scheme in complex FPGA designs is a challenging task that requires a good knowledge of clocking resources supported by the target FPGA and their limitations, an understanding of the tradeoffs between different design techniques, and the following of good design practices. An incorrectly designed or suboptimal clocking scheme can lead to poor design performance in the best case, or random and hard-to-find failures in the worst.

Examples of the FPGA clocking resources knowledge include the amount of different resources in a target FPGA, clock types (such as regional and global), frequency limitations and jitter characteristics of different clock managers, and the maximum number of clocks that can be used in a single clock region.

This Tip discusses options to implement different parts of the clocking scheme and provides an overview of some of the recommended design practices.

Using dedicated clocking resources

An internally generated clock is an output of a combinatorial logic or a register, as shown in the following figure.

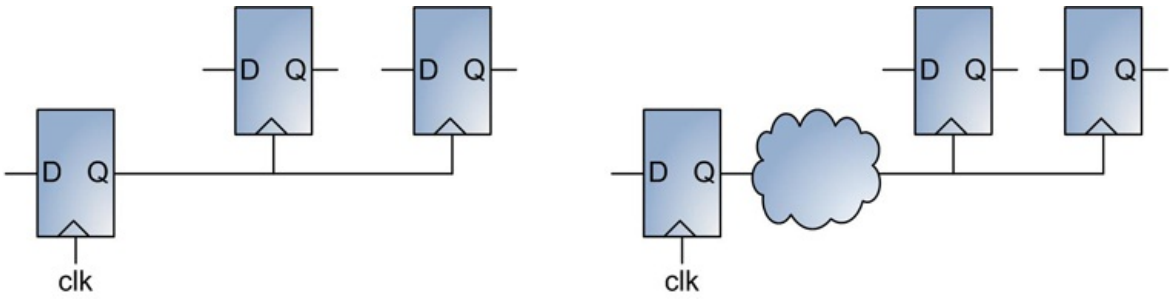

Figure 1: internally generated clocks

A clock generated by a combinatorial logic might have glitches, which can be falsely interpreted as a valid clock edge, and cause a functional problem in a design. Therefore, never use an output of combinatorial logic as a clock.

Internally generated clocks use general routing resources and, therefore, have longer delays compared with the dedicated clock routing. The consequence is additional clock skew, which makes the process of meeting timing more difficult. That becomes even more important if a lot of logic is using that internal clock.

As a general rule, avoid using internally generated clocks and use dedicated clocking resources whenever possible.

An overview of dedicated clocking resources provided by Xilinx FPGAs is provided in Tip #20.

Using a single clock edge

With the exception of few special circuits, such as capturing double data rate (DDR) data, always use an either positive or negative clock edge for registering data. One problem with using both edges is the duty cycle of the clock, which might not always be 50%.

Using differential clocks

It is recommended to use differential clocks for high frequencies. As a rule of thumb, high frequency is generally considered to be above 100MHz. The main advantage of differential clocks over single-ended counterparts is common-mode noise rejection, which provides better noise immunity. Differential clocks with PECL, LVPECL, and LVDS signal levels are popular choices to clock high-speed logic.

Xilinx FPGAs provide several dedicated primitives to use for differential clocking: IBUFDS, IBUFGDS, IBUFGDS_DIFF_OUT, OBUFDS, OBUFTDS, and others.

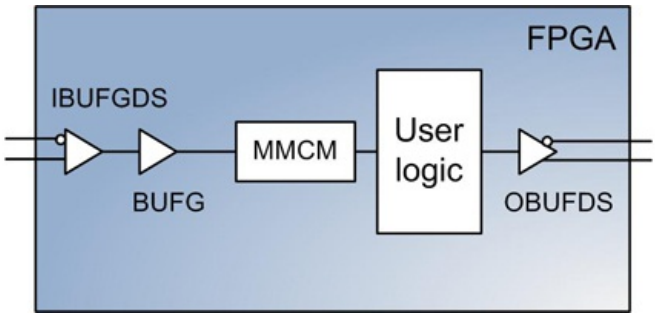

Figure 2: Using dedicated clocking resources

Using gated clocks

Clock gating is a technique that disables clock inputs to registers and other synchronous elements in the design using control signal. Clock gating is very effective in reducing power consumption and widely used in ASIC designs. However, using gated clocks in FPGA designs is discouraged. Tip #49 provides more information and examples on gated clocks and discusses some of the conversion techniques.

Using clock signal as a control, reset, or data input

It is not recommended to use clock signal as control, reset, or data input to general-purpose logic. The following are Verilog examples of such circuits.

```
module clock_schemes(  input clk1,clk2,clk3,clk4,clk5,
                       input data_in,
         output reg data_out1,data_out2,   data_out3, data_out4, data_out5, data_out6);
wire data_from_clock, reset_from_clock, control_from_clock;
// Clock used as data input
assign data_from_clock = clk1;
always @(posedge clk1)
   data_out1 <= ~data_out1;

always @(posedge clk2)
   data_out2 <= ~data_out2 & data_from_clock;
// Clock used as reset input
assign reset_from_clock = clk3;
always @(posedge clk3)
   data_out3 <= ~data_out2;

always @(posedge clk4, posedge reset_from_clock)
   if (reset_from_clock)
     data_out4 <= 0;
   else
     data_out4 <= data_in;

// Clock is used as control
assign control_from_clock = clk5;
always @(posedge clk5)
   data_out5 <= ~data_out5;

always @(*)
   data_out6 = control_from_clock ? data_in : data_out6;
endmodule // clock_schemes
```

Source synchronous clocks

Many external devices interfacing with FPGA use a source synchronous clock along with the data. If the interface is running at high speed, it might require the calibration of the clock edge to capture data at the center of the data window. To implement dynamic calibration, Xilinx MMCM primitive provides a dynamic reconfiguration port (DRP), which allows programmatic phase shifting of the clock. The following figure illustrates that the output clock from the MMCM is shifted such that the rising edge of the clock samples the data in the middle of the window.

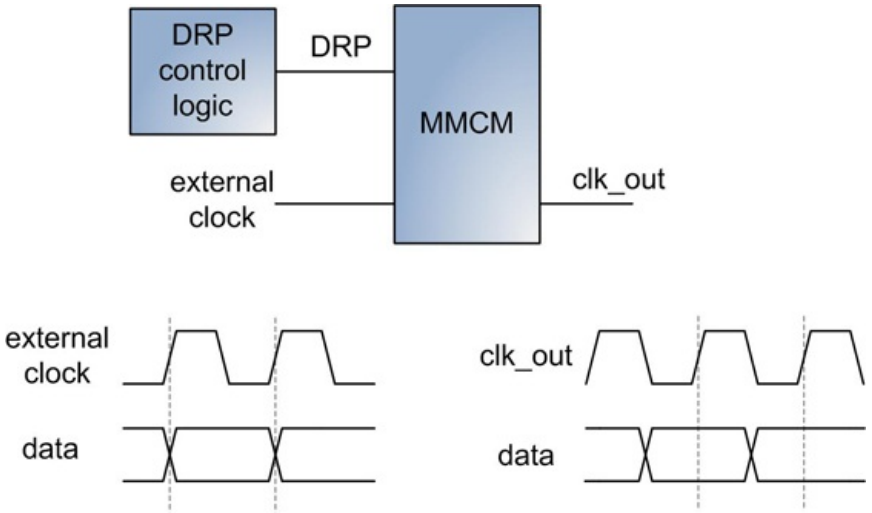

Figure 3: Using phase shifting of the clock

Clock multiplexing

Multiplexing clocks from several sources is required in designs that operate the same logic from different clock sources. One example is Ethernet MAC, which uses 2.5MHz, 25MHz, or 125MHz clocks depending on the negotiated speed of 10Mbs, 100Mbs, or 1Gbs, respectively. Another example is a power-on built-in self test (BIST) circuit, which uses a clock different from the one used during a normal operation.

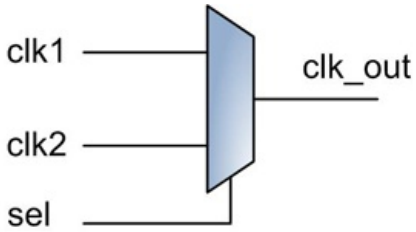

Figure 4: Clock multiplexing

It is recommended to use a dedicated clocking resource to implement clock multiplexing, to ensure that input and output clocks are using dedicated clock lines instead of general purpose logic. The input clock frequencies could be unrelated to each other. Using combinatorial logic to implement a multiplexor can generate a glitch on the clock line at the time of the switch. A glitch on the clock line is hazardous to the whole system because it could be interpreted as a valid clock edge by some registers but be missed by others.

Xilinx provides a BUFGMUX primitive that can multiplex between two global clock sources. It also ensures a glitch-free operation when the input clock selection is changed. Clock multiplexing requires careful timing constraints of all paths from an input to an output clock of the multiplexer.

Detecting clock absence

One method of detecting clock absence is to use an oversampling technique by another higher speed clock. The disadvantage of this method is the availability of another clock. An alternative method is to use the `locked` output from Xilinx MMCM primitive, as shown in the following figure.

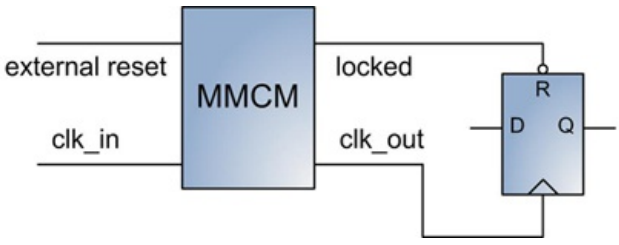

Figure 5: Using locked output to detect clock absence

22. Clock Domain Crossing

Most FPGA designs utilize more than one clock. An example of a multi-clock design is illustrated in the following figure.

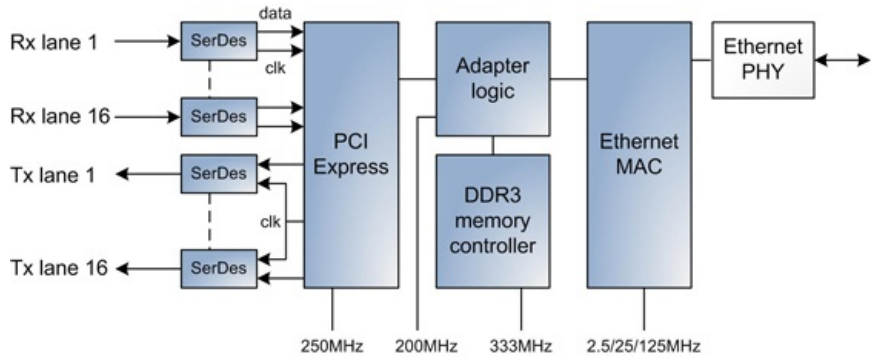

Figure 1: An example of a multi-clock design

The design implements a PCI Express to Ethernet adapter and is shown to illustrate the potential complexity of a clocking scheme. It has a 16-lane PCI Express, tri-mode Ethernet, DDR3 memory controller, and the bridge logic. 16 Serializer/Deserializer (SerDes) modules embedded in FPGA are used to receive PCI Express data, one for each lane. Each SerDes outputs a recovered clock synchronized to the data. A shared clock is used for all PCI Express transmit lanes. Tri-mode Ethernet MAC requires 2.5MHz, 25 MHz, and 125MHz clocks to operate at 10Mbs, 100Mbs, or 1Gbs speed, respectively.

The memory controller uses a 333MHz clock, and the bridge logic utilizes a 200MHz clock. In total, there are 23 clocks in the design. Each clock domain crossing – from PCI Express to bridge, bridge to Ethernet, and bridge to memory controller – requires using a different technique to ensure reliable operation of the design.

Metastability

Metastability is the main design problem to be considered for implementing data transmission between different clock domains.

Metastability is defined as a transitory state of a register which is neither logic '0' nor logic '1'. A register might enter a metastable state if the setup and hold timing requirements are not met. In a metastable state a register is set to an intermediate voltage level, which is neither a "zero" nor a "one" logic state. Small voltage and temperature perturbations can return the register to a valid state. The transition time and resulting logic level are indeterminate. In some cases, the register output can oscillate between the two valid states. Metastability conditions arise in designs with multiple clocks, or asynchronous inputs, and result in data corruption.

The following are some of the circuit examples that can cause metastability.

Example 1

A state machine may enter an illegal state if some of the inputs to the next state logic are driven by a register in a different clock domain. This is illustrated in the following figure.

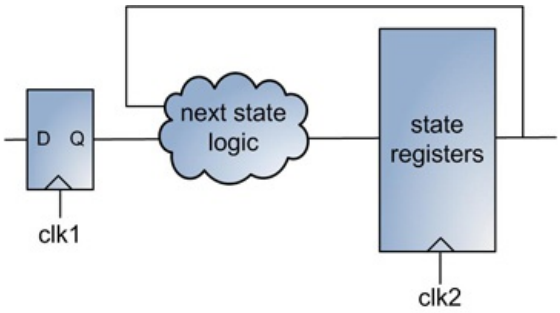

Figure 2: State machine enters an incorrect state

The exact problem that may occur due to metastability depends on the state machine implementation. If the state machine is implemented as one-hot – that is, there is exactly one register for each state – then the state machine may transition to a valid, but incorrect, state.

Example 2

An input data to a Xilinx BRAM primitive and the BRAM itself are in different clock domains.

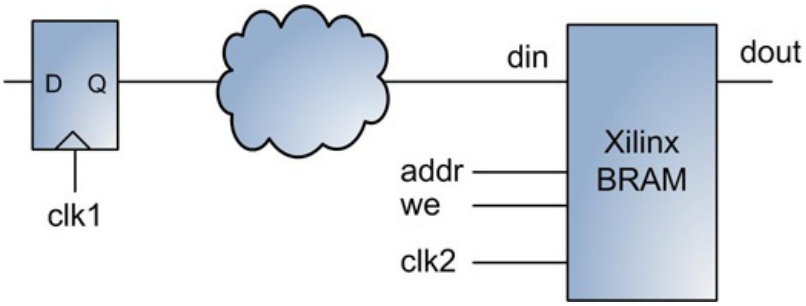

Figure 3: BRAM and its inputs are in different clock domains

If the input data violates the setup of hold requirements of a BRAM, that may result in data corruption. The same applies to other BRAM inputs, such as address and write enable.

Example 3

The output of a register in one clock domain is used as a synchronous reset to a register in another clock domain. The data output of the right register, shown

in the figure below, can be corrupted.

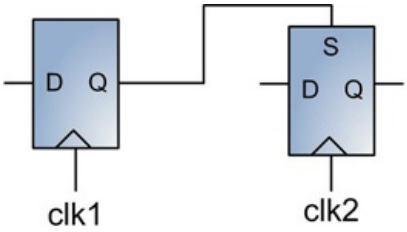

Figure 4: Metastability due to synchronous reset

Example 4

Data coherency problem may occur when a data bus is sampled by registers in different clock domains. This case is illustrated in the following figure.

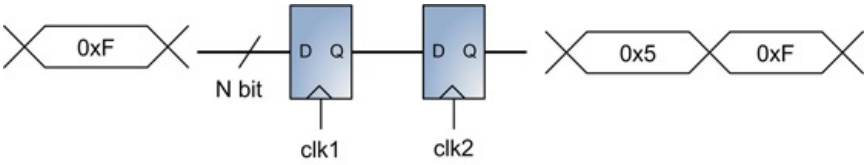

Figure 5: Data coherency

There is no guarantee that all the data outputs will be valid in the same clock. It might take several clocks for all the bits to settle.

Calculating Mean Time Between Failure (MTBF)

Using metastable signals can cause intermittent logic errors. Mean time between failure, or MTBF, is a metric that provides an estimate of the average time interval between two successive failures of a specific synchronous element. Synchronization circuits, such as using the two registers described in Tip #23, help increase the MTBF and reduce the probability of en error to practical levels, but they do not completely eliminate it.

There are several practical methods for measuring the metastability capture window described in the literature. The one applicable to Xilinx FPGA is Xilinx Application Note XAPP094 [1].

However, MTBF can be only determined using statistical methods. A commonly used MTBF equation is:

$$MTBF = \frac{exp(T * \tau)}{f1 * f2 * To}$$

f_1 and f_2 are the frequencies of two clock domains.

The product $T*\tau$ in the exponent describes the speed with which the metastable condition is resolved.

T_o is the duration of a critical time window during which the synchronous element is likely to become metastable.
T_o , T, and τ are circuit specific.
As an example, for f_1=1MHz, f_2=1KHz, T_0=30ps, $T*\tau = 10$,
MTBF = exp(10)/(1MHz * 1KHz * 30ps) = 734,216 sec = 204 hours.

Clock Domain Crossing (CDC) analysis

In complex multi-clock designs, the task of correctly detecting and verifying all clock domain crossing is not simple. Design problems due to CDC are typically not detected in a functional simulation. Unfortunately, there are only a few adequate tools from the functionality and cost perspective that perform automatic identification and verification of the CDC schemes used in FPGA designs.

Mentor Graphics Questa software provides a comprehensive CDC verification solution, including RTL analysis, identification of all clocks and clock domain crossings, and generation of assertions and metastability models.

The Xilinx XST synthesis tool provides a `-cross_clock_analysis` option to perform inter-clock domain analysis during timing optimization.

Resources

[1] Metastable Recovery in Virtex-II FPGAs, Xilinx Application Note XAPP094
http://www.xilinx.com/support/documentation/application_notes/xapp094.pdf

23. Clock Synchronization Circuits

Synchronization circuit captures an asynchronous input signal and produces an output, which is synchronized to a clock edge. Synchronization circuits are used to implement the proper clock domain crossing of a signal, and prevent setup and hold time violation. The choice of a circuit depends on different data and clock characteristics: the number of signals that cross the clock domains, how those signals are used – as a control or data, and the frequency relationship of the clocks. The following diagram shows an example of a system where a single-clock pulse in 250MHz clock domain needs to be transferred to a slower 100MHz domain.

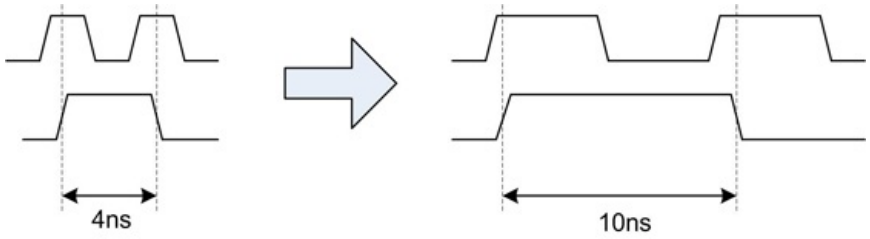

Figure 1: Clock domain crossing of a single-clock pulse

Synchronization using two registers

One approach to solving a metastability problem on individual signals is based on simple synchronizer comprising three registers, shown in the following figure.

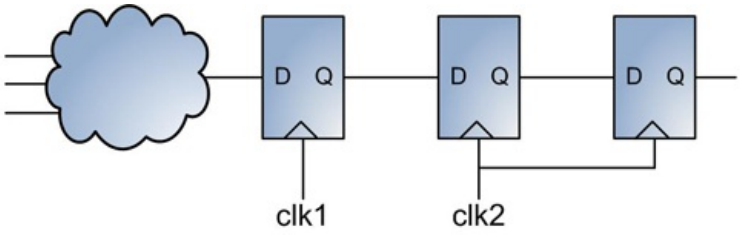

Figure 2: A simple synchronizer using two registers

The following is the implementation example.

```
module synchronizer(  input clk1,clk2,
         input reset,
         input data_in,
         output reg data_out);

   reg data_in_q, data_out_q;

   always @(posedge clk1, posedge reset)
      if(reset)
         data_in_q <= 1'b0;
      else
         data_in_q <= data_in;

   always @(posedge clk2, posedge reset)
      if(reset) begin
         data_out_q  <= 1'b0;
         data_out    <= 1'b0;
      end
      else begin
         data_out_q <= data_in_q;
         data_out   <= data_out_q;
      end

endmodule // synchronizer
```

It is important to add proper timing constraints to cover all the paths between two clock domains. In the above example the only path is `data_in_q`.

The following is the example of timing constraints in Xilinx UCF format.

```
# clk1 period constraint
NET "clk1" TNM_NET = clk1;
TIMESPEC TS_clk1 = PERIOD "clk1" 5 ns HIGH 50%;

# clk2 period constraint
NET "clk2" TNM_NET = clk2;
TIMESPEC TS_clk2 = PERIOD "clk2" 4.1 ns HIGH 50%;

# constrain the paths between clk1 and clk2 domains
TIMESPEC "TS_CDC_1" = FROM "clk1" TO "clk2" 5ns;

# another way to constrain the path between clk1 and clk2
# TIMESPEC "TS_CDC_1"  = FROM "clk1" TO "clk2"  TIG;
```

The following is the section from the timing analysis report, which covers `TS_CDC_1` constraint.

```
================================================================
Timing constraint: TS_CDC_1 = MAXDELAY FROM TIMEGRP "clk1" TO TIMEGRP "clk2" 5ns;

 1 path analyzed, 1 endpoint analyzed, 0 failing endpoints
 0 timing errors detected. (0 setup errors, 0 hold errors)
 Maximum delay is   3.094ns.
--------------------------------------------------------------
Slack (setup paths):  1.906ns (requirement - (data path - clock path skew + uncertainty))
  Source:            data_in_q (FF)
  Destination:        data_out_q (FF)
  Requirement:        5.000ns
  Data Path Delay:     2.248ns (Levels of Logic = 0)
  Clock Path Skew:     -0.811ns (2.532 - 3.343)
  Source Clock:       clk1_BUFGP rising
  Destination Clock:   clk2_BUFGP rising
  Clock Uncertainty:   0.035ns

Maximum Data Path at Slow Process Corner: data_in_q to data_out_q
Location      Delay type   Delay(ns)  Logical Resource(s)
----------------------------------------------------------
ILOGIC_X0Y7.Q4   Tickq       1.220   data_in_q
                             data_in_q
SLICE_X1Y7.AX    net (fanout=1) 0.914   data_in_q
SLICE_X1Y7.CLK   Tdick       0.114   data_out_q
```

```
--------------------------------------------------------
Total  2.248ns (1.334ns logic, 0.914ns route)
       (59.3% logic, 40.7% route)
--------------------------------------------------------------------
Hold Paths: TS_CDC_1 = MAXDELAY FROM TIMEGRP "clk1" TO TIMEGRP "clk2" 5 ns;
--------------------------------------------------------------------
Slack (hold path):  1.227ns (requirement - (clock path skew + uncertainty - data path))
 Source:            data_in_q (FF)
 Destination:        data_out_q (FF)
 Requirement:        0.000ns
 Data Path Delay:     1.162ns (Levels of Logic = 0)
 Positive Clock Path Skew: -0.100ns (0.986 - 1.086)
 Source Clock:       clk1_BUFGP rising
 Destination Clock:   clk2_BUFGP rising
 Clock Uncertainty:   0.035ns

Minimum Data Path at Fast Process Corner: data_in_q to data_out_q
Location      Delay type     Delay(ns)  Logical Resource(s)
-------------------------------------------------------
ILOGIC_X0Y7.Q4   Tickq        0.656    data_in_q
                                 data_in_q
SLICE_X1Y7.AX   net (fanout=1) 0.447    data_in_q
SLICE_X1Y7.CLK   Tckdi(-Th)   -0.059    data_out_q
-------------------------------------------------------
Total   1.162ns (0.715ns logic, 0.447ns route)
        (61.5% logic, 38.5% route)
-------------------------------------------------------
```

The figure below illustrates an incorrect way to implement the synchronizer.

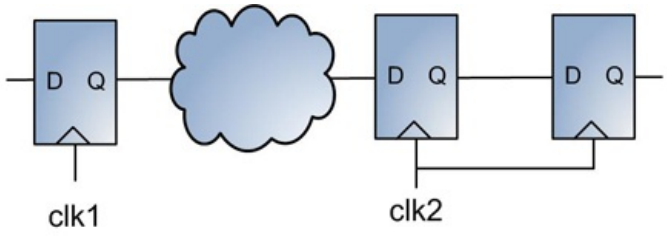

Figure 3: Incorrect synchronizer implementation

The problem with this implementation is combinational logic between adjacent registers in different clock domains. If combinational logic is located before the first synchronizing register, it becomes sensitive to glitches produced by the combinational logic. That increases the probability of propagating and invalid data across the synchronizer.

Using this synchronization circuit requires caution if it's used to synchronize busses. The problem is that the circuit cannot guarantee that all the outputs will be valid in the same clock. It might take several clocks for all the bits to settle.

This synchronizer will not work correctly if the frequency of clk1 is faster than clk2, such as in the example shown in the beginning of this Tip. If duration of the valid data is short, and it's sampled with a slower clock, the data can be missed. It should only be used when the input changes are slow relatively to the capturing `clk2`.

Synchronization using edge detection

The following figure shows a circuit that implements a synchronizer using edge detection technique.

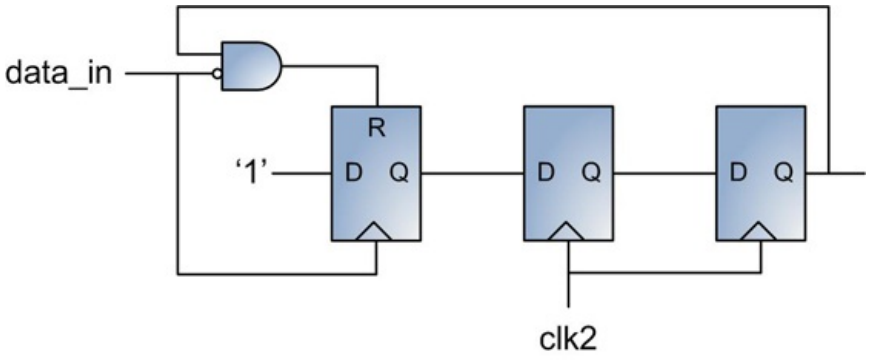

Figure 4: Synchronization using edge detection

The Verilog implementation of the circuit is given below.

```
module edge_detector(  input clk2,reset,
                  input data_in,
                  output reg data_out);

   reg data_in_q, data_out_q;
   wire reset_in;

   assign reset_in = (~data_in & data_out) | reset;

   always @(posedge data_in, posedge reset_in)
      if(reset_in)
         data_in_q <= 1'b0;
      else
         data_in_q <= 1'b1;

   always @(posedge clk2, posedge reset)
      if(reset) begin
         data_out_q  <= 1'b0;
         data_out   <= 1'b0;
      end
      else begin
         data_out_q <= data_in_q;
```

```
        data_out   <= data_out_q;
    end

endmodule // edge_detector
```

The principle of the operation of the circuit is illustrated in the following waveform.

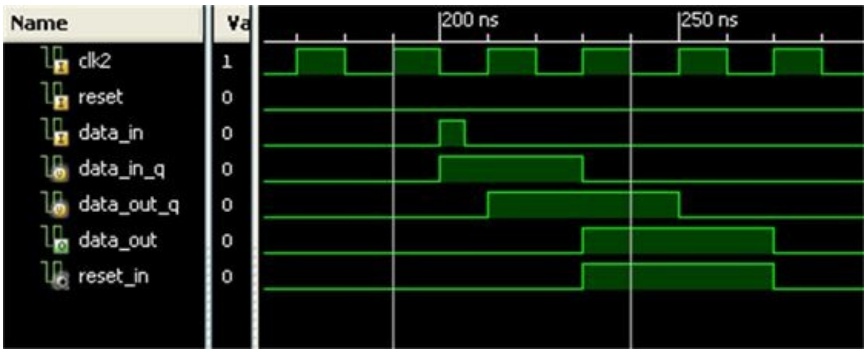

Figure 5: Edge synchronization waveform

The input data is used as a clock to the leftmost register. That allows it to capture pulses that are shorter than the `clk2` period. When the input data is deasserted, and the pulse propagated to the rightmost register, the leftmost register is cleared.

The disadvantage of this circuit is that is uses data signal as a clock.

Synchronization using an asynchronous FIFO

Using asynchronous FIFO is a robust approach to synchronize data busses. This method is easy to use, and works reliably for any data and clock frequencies.

The disadvantage is significant amount of logic and embedded memory resources required to implement a FIFO. In Xilinx FPGAs the smallest FIFO implemented with distributed memory is 16 entries. Another disadvantage is the latency added by the FIFO.

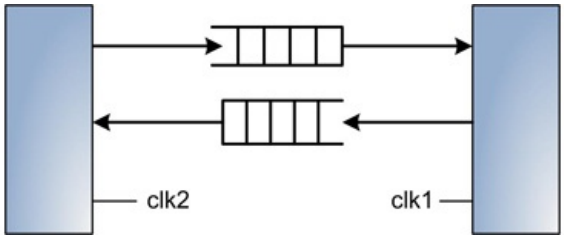

Figure 6: Using asynchronous FIFO for synchronization

Synchronization using Grey code

Grey code is a data encoding technique where only one bit at most changes every clock. Grey code can be used for data bus synchronization. This method is widely used for implementing counters, empty, full, and other control signals in asynchronous FIFOs, which cross FIFO read/write clock domains.

Synchronization using handshake

Handshake synchronizers rely on request-acknowledgement protocols to ensure delivery of the data. The following figure shows a top-level block diagram of a handshake synchronizer.

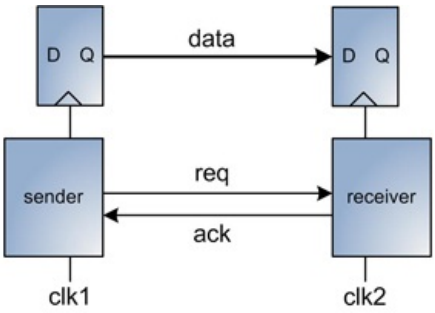

Figure 7: Handshake synchronizer

A request is sent from the source clock domain. The destination domain receives the request, performs its synchronization, captures the data, and sends back an acknowledgement. The source domain receives the acknowledge, and can prepare to send the next data. The advantage of the handshake synchronizer is that it doesn't require synchronizing the data, which can save a lot of logic resources. The disadvantage is the latency incurred during exchanging request/acknowledge for each data transfer.

Synchronization using asynchronous oversampling

Synchronization technique using asynchronous oversampling is described in [1]. The technique uses multiple phases of a mixed mode clock manager (MMCM) in Virtex-6 FPGAs to oversample the input data, and performing subsequent edge detection.

Resources

[1] Virtex-6 FPGA LVDS 4X Asynchronous Oversampling at 1.25 Gb/s, Xilinx Application Note XAPP881

http://www.xilinx.com/support/documentation/application_notes/xapp881_V6_4X_Asynch_OverSampling.pdf

[2] Clifford Cummings, "*Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs*", SNUG-2001, San Jost
http://www.sunburst-design.com/papers/CummingsSNUG2001SJ_AsyncClk.pdf

24. Using FIFOs

FIFO stands for first-in first-out. It's a very configurable component, and finds many uses in FPGA designs. Some usage examples are clock domain crossing, data buffering, bus width conversion, interfacing between producer/consumer modules.

The following figure shows an example of using FIFOs in a PCI Express to Ethernet adapter. The FIFOs perform data buffering between modules with different clock frequency and datapath width.

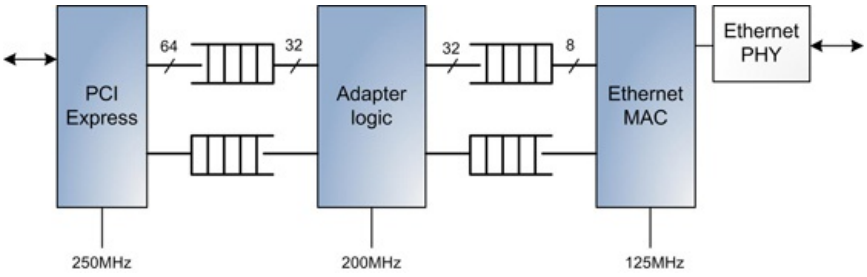

Figure 1: FIFO usage example

FIFO architecture

The following figure shows a simplified architecture of a FIFO.

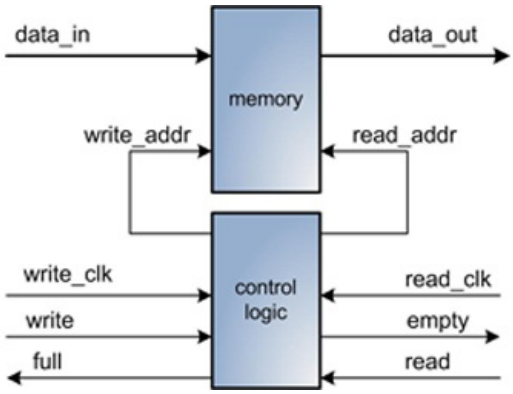

Figure 2: FIFO architecture

A FIFO has two main components: memory to store the data, and control logic. Control logic is responsible for maintaining address read and address write pointers to the memory, and updating status flags. There are several other status flags that can be exposed to the user: almost full, almost empty, FIFO data counter, read underflow, write overflow, and others.

A FIFO can be synchronous or asynchronous. Synchronous FIFO has the same read and write clocks. In asynchronous FIFO, read and write clocks are independent. Designing a reliable and high-performing asynchronous FIFO presents unique challenges. Memory module has to be dual-ported, and support concurrent read and write from two clock domains. Control logic has to implement proper clock domain crossing from the write pointer logic to empty flag and from the read pointer logic to full flag.

Several commercial solutions of clock domain crossing in FIFO control logic are using a Gray counter. An important property of a Gray counter is one bit change for every transition. Using the value of such a counter in unrelated clock domain for comparison is not going to generate a glitch.

FIFO configuration options

FPGA designers have several options of implementing a FIFO circuit. Xilinx provides several highly configurable FIFO cores. A custom FIFO can also be designed using embedded FPGA memory and logic.

Xilinx FIFO cores provide the following configuration options:

- FIFO depth: between 16 and 4 million entries.
- FIFO data width: from 1 to 1024 bit.
- Read to write port aspect ratio: from 1:8 to 8:1.
- One or separate FIFO read and write clock domains.
- Memory type: Block RAM, distributed RAM, built-in FIFO, shift register.
- Error checking (ECC)
- Data prefetch (first word fall-through).
- Status flags: full, empty, almost_full, almost_empty, read and write data count, overflow and underflow error.

Design challenges in using FIFOs

The following is a list of common challenges FPGA designers face when using FIFOs.

- Designers must take special precautions when using asynchronous FIFOs.
- Asynchronous FIFOs may operate in two unrelated read and write clock domains.
- Control and status signals are synchronous to either read or write clocks.

For example, full, almost full, write count, and write signals are synchronous to the write clock; empty, almost empty, read count, and read signals are

synchronous to read clock. Xilinx also recommends not relying on cycle accuracy of the status flags in asynchronous FIFOs. It is possible to have inconsistent cycle time behavior in different chips, due to process, voltage, and temperature variation. This can lead to arcane problems that are difficult to debug.

A frequent logic error is FIFO overflow and underflow. A FIFO overflow occurs when there is a write when the FIFO is already full. An underflow is a read from an empty FIFO. Overflow and underflow conditions are relatively simple to monitor. A designer can implement a circuit that latches an error bit when write and full, or read and empty flags are asserted in the same clock. Some Xilinx FIFO configurations provide overflow and underflow error flags. Designs can be set up to trigger on the FIFO access error bits by the integrated logic analyzer (ILA), and read the waveform by the ChipScope software.

Designers should take into account the FIFO read latency. Usually, the data is available on the next clock cycle after an FIFO read. If FIFO data outputs are registered, the latency is two clocks. Some Xilinx FIFO configurations provide prefetch option, which is also called first word fall-through, where read data is available on the same clock cycle as the read strobe. The data is prefetched on the FIFO read side as soon as it's written to an empty FIFO.

FIFOs have an option to use synchronous or asynchronous reset. Logic driving a synchronous reset should be in the same clock domain.

Calculating FIFO depth

One of the FIFO usage models is buffering the data between producer and consumer, as shown in the following figure.

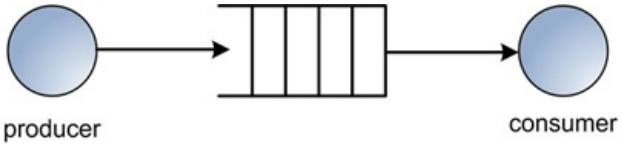

Figure 3: FIFO producer-consumer model

The overall producer data rate should not exceed the consumer rate of processing the data. A FIFO is not designed to overcome the rate difference. No matter how deep the FIFO is, it is going to eventually overflow if the producer's data rate is consistently higher than consumer's. An FIFO is intended to overcome temporary producer-consumer rate differences by buffering the excess data.

To accurately calculate the minimum required FIFO depth, designers need to have a good knowledge of the producer and consumer data characteristics such as arrival and departure rate, and burst duration. Underestimating FIFO depth will lead to the user having to exert significant effort to debugging and understanding the root cause of the problem, and consequent redesign. FPGA logic resources are expensive; therefore it is also important not to overestimate the FIFO depth.

Queuing theory defines theoretical models for various producer and consumer traffic characteristics. However, in most cases the models are too restrictive, and aren't applicable to the real-world situations.

Other analysis methods include computer simulations of the traffic conditions, or finding the FIFO size empirically by prototyping a similar system.

Designers can also use the following two simplified models: constant pace burst calculation, and variable pace length.

Constant pace burst calculation model

Let's denote P(t) the rate of data production, and C(t) the rate of data processing or consumption. This model assumes that P and C are constant for the time T – the duration of a burst. By the end of a burst, P*T data units were written to a FIFO, and C*T were read. The number of remaining data units is L = T*(P-C). L is the maximum size of the FIFO. This is a very simplistic model that assumes constant rate of data production and consumption. It also can't handle the situation when the next burst starts before all the data from the previous burst has being read.

The following example illustrates how to use this model.

1500 Byte Ethernet packets arrive every 50us and are written into an 8-bit FIFO. The arrival rate is 1Gbit/sec. The FIFO read rate is 100Mreads/sec.
P(t) = (1Gbit/sec)/8bit = 125Mwrites/sec
C(t) = 100Mreads/sec
T = 1500Byte/(1Gbit/sec) = 12us
L = T*(P-C) = 12*(125-100) =300

The FIFO has to be 300 entries deep to sustain the traffic. Intuitive interpretation of the results is as follows:

If the processing rate was close to zero, the entire packet will have to be stored in the FIFO, which will require 1,500 entries. If the processing rate is faster than the arrival rate, the FIFO depths will need to be 0. The closer the processing rate is to the arrival rate, the smaller the FIFO needs to be.

Variable pace length model

When the data arrival and processing rates are not constant, the previous model can be modified as follows:

$$L[Tx] = \int_{t1}^{Tx} [P(t) - C(t)]dt$$

$$L = max\ \{L[Tx]\}$$

t1 is burst start time, and *L[Tx]* is the FIFO length at any given time during the burst. *L[Tx]* will vary during the burst because of the changing arrival and processing rates. To find the maximum FIFO length, use the maximum value of the *L[Tx].*

The constant pace burst calculation model is a special case of the variable pace length model when the arrival and processing rates are constant.

25. Counters

Counters are used in virtually every FPGA design for different purposes: timers, utility counters, or even state machines. There are several different counter types: the most basic binary counter, prescaler, Johnson, LFSR, and others. Counters can count in the up or down direction, have a load input to start with predefined value, have synchronous or asynchronous reset, stop when reaching certain threshold or free-running, and many other configurations. Understanding the tradeoffs between different counter types and configurations and choosing the right one for a particular design can save a lot of logic resources, and improve performance.

Binary counter

Binary counter is the simplest and the most flexible counter type. It allows sequential count up or down, from 0 to 2^N -1, where N is the number of bits. The main disadvantage of a binary counter is significant amount of combinatorial feedback logic.
The following is an example implementation of a 32-bit binary counter.

```
module counter_binary(input clk,
                input reset,
                input counter_binary_en,
                output reg [31:0] counter_binary,
                output reg  counter_binary_match);

  always @(posedge clk)
   if(reset) begin
      counter_binary      <= 'b0;
      counter_binary_match <= 1'b0;
   end
   else begin
     counter_binary      <= counter_binary_en
        ? counter_binary + 1'b1
        : counter_binary;
     counter_binary_match <=
        counter_binary == `MATCH_PATTERN;
   end
endmodule // counter_binary
```

Johnson counter

The Johnson counter can be used to generate counters in which only one output changes on each clock cycle. The Johnson counter is made of a simple shift register with an inverted feedback. Johnson counters have 2N states, where N is the number of registers, compared to 2^N states for a binary counter. These counters are sometimes called "walking ring" counters, and are found in specialized applications, such as decade counters, or for generating a clock-like pattern with many different phases.

The following figure shows an example of a 3-bit Johnson counter.

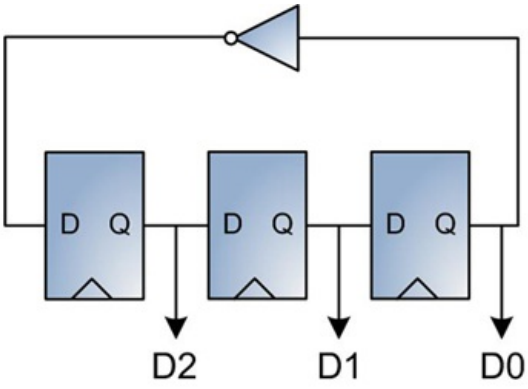

Figure 1: Johnson counter

The following is an example implementation of a 32-bit Johnson counter.

```
module counter_johnson(input clk,
               input reset,
               input counter_johnson_en,
               output reg [31:0] counter_johnson_q,
               output reg counter_johnson_match);

  always @(posedge clk, posedge reset)
   if(reset) begin
     counter_johnson_q    <= 'b0;
     counter_johnson_match <= 1'b0;
   end
   else begin
     counter_johnson_q    <= counter_johnson_en
        ? {counter_johnson_q[30:0], ~counter_johnson_q[31]}
        : counter_johnson_q;
     counter_johnson_match <=
        counter_johnson_q == `MATCH_PATTERN;
   end
endmodule // counter_binary
```

LFSR counter

Linear Feedback Shift Register (LFSR) is a shift register whose input is a linear function. Some of the outputs are combined using an XOR operation, and form a feedback. The LFSR operation is deterministic, and produces a repeating sequence of known values. LFSR counters are based on the LFSR registers. The main feature of LFSR counters is speed. The main disadvantage is that the output counting sequence is not incremental, like in a binary counter.

The following is an example implementation of a 32-bit LFSR counter that matches 32'h81234567 value. The code is generated using online LFSR generation tool offered by OutputLogic.com.

```
module lfsr_counter(
      input clk,
      input reset,
```

```
      input ce,
      output reg lfsr_done);

   reg [31:0] lfsr;
   wire d0,lfsr_equal;

   xnor(d0,lfsr[31],lfsr[21],lfsr[1],lfsr[0]);
   assign lfsr_equal = (lfsr == 32'h1565249);

   always @(posedge clk,posedge reset) begin
      if(reset) begin
         lfsr <= 32'h0;
         lfsr_done <= 0;
      end
      else begin
         if(ce)
            lfsr <= lfsr_equal ? 32'h0 : {lfsr[30:0],d0};
         lfsr_done <= lfsr_equal;
      end
   end
endmodule
```

Prescaler counter

The prescaler counter can be used to implement a faster binary counter by cascading smaller-size counters. Each cascade acts as a clock divider by downscaling the high-frequency clock. The main advantage is lower logic utilization. The disadvantage is less flexibility: such a counter has lower count granularity, and cannot count any value.

The following figure shows an example of a two-stage prescaler counter.

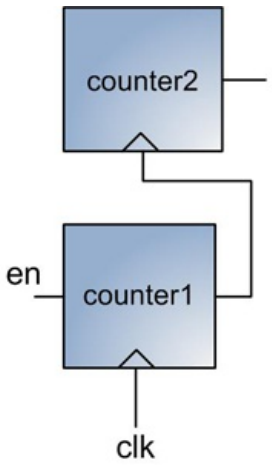

Figure 2: Prescaler counter

The following is an example implementation of a 32-bit prescaler counter. It consists of two cascaded 16-bit binary counters.

```
module counter_prescaler(input clk,
                  input reset,
                  input counter_prescale_en,
                  output [15:0]  counter_prescale,
                  output reg counter_prescale_match);

   reg [15:0] counter_prescale_low,counter_prescale;

   always @(posedge clk)
      if(reset) begin
         counter_prescale_low <= 'b0;
      end
      else begin
         counter_prescale_low <= counter_prescale_en
              ? counter_prescale_low + 1'b1
              : counter_prescale_low;
      end

   always @(posedge counter_prescale_low[15], posedge reset)
      if(reset) begin
         counter_prescale      <= 'b0;
         counter_prescale_match <= 1'b0;
      end
      else begin
         counter_prescale      <= counter_prescale + 1'b1;
         counter_prescale_match <=
            counter_prescale == `MATCH_PATTERN_PRESCALE;
      end

endmodule // counter_prescaler
```

Performance and logic utilization comparison

The following table summarizes performance and logic utilization of several 32- and 64-bit counter types. The first two are standard 32- and 64-bit binary counters. The next two counters are Xilinx counter cores generated using CoreGen application. Counter number five from the top is a 32-bit binary counter implemented using Xilinx DSP48 primitive. Prescaler, LFSR and Johnson are 32-bit counters.

All the counters were built for Xilinx Spartan-6 LX45 FPGA with -3 speed grade.

Table 1: performance and logic utilization of different counters

Counter	*Registers*	*LUTs*	*Slices*	*Max frequency, MHz*
Binary 32 bit	33	8	13	364

Binary 64 bit	65	12	19	310
Binary 32 bit using Xilinx core	33	8	14	257
Binary 64 bit using Xilinx core	65	12	20	216
Xilinx DSP48 32 bit	0	6	2	345
Prescaler 32 bit	33	5	10	440
LFSR 32 bit	33	15	9	303
Johnson 32 bit	33	8	9	375

In order to understand performance differences between different counters, it is useful to compare the number of logic levels, logic and routing delays of the worst path, and maximum fanout. The following table shows the information that was extracted from the project timing report.

Table 2: logic and routing delays of different counters

Counter	*Levels of logic*	*Logic delay, ns*	*Route delay, ns*	*Max fanout*
Binary 32 bit	3	1.1	1.6	3
Binary 64 bit	3	1.0	2.15	3
Binary 32 bit using Xilinx core	5	2.0	2.1	2
Binary 64 bit using Xilinx core	11	2.6	3.2	2
Xilinx DSP48 32 bit	N/A	N/A	N/A	N/A
Prescaler 32 bit	4	1.45	0.4	1
LFSR 32 bit	3	1.3	2.0	8
Johnson 32 bit	3	1.1	1.5	3

All the counters have expected register utilization: 32 or 64 bit data and one bit match.

The number of LUTs of and LFSR counter is higher than expected because this particular implementation uses additional LUTs as a pattern match.

A DSP48 counter utilizes 6 LUTs because of some logic external to DSP48 primitive.

For small and medium size counters, the slice utilization difference for the same counter width is not significant.

Among 32-bit counters, prescaler counter has the highest performance and the one generated by Xilinx CoreGen is the lowest. This is consistent with the logic level number and logic delay results.

The above performance and logic utilization results are only shown as an example, and should be interpreted judiciously. The results will vary significantly for different FPGA families, speed grades, tool options, and counter configuration.

All the source code, project files and the reports are available on the accompanying website.

26. Signed Arithmetic

Signed arithmetic is widely used in digital signal processing (DSP), image and video processing algorithms, communications, and many other FPGA designs. The first Verilog-95 standard didn't provide a good support for two's complement signed arithmetic. Developers needed to hand-code signed operations by using bit-level manipulations. That changed in Verilog-2001, which added explicit support for signed arithmetic, such as special keywords and built-in signed operators. To enable Verilog-2001, use `-verilog2001` command line option in XST, and "*set_option -vlog_std v2001*" command in Synplify.

Verilog support for signed expressions

Verilog standard defines `signed`, `unsigned` reserved words, and `$signed`, `$unsigned` system tasks.

The `signed` reserved word declares register and net data types as signed. It can also be placed on module port declarations. When either the date type or the port is declared signed, the other inherits the property of the signed data type or port.

The `unsigned` reserved word declares a register and net data types to be unsigned.

`$signed` and `$unsigned` system tasks perform type cast from signed to unsigned and vice versa.

The following code provides an example of using signed/unsigned reserved words and system tasks.

```
   integer val1; // 32-bit signed, 2's complement
   reg [31:0] val2; // 32-bit unsigned
   wire signed [15:0] val3; // 16-bit signed
   wire unsigned [15:0] val4; // unsigned keyword is optional
   assign val4 = $unsigned(val3); //type-cast to unsigned
   assign val3 = $signed(val4);   //type-cast to signed
reg[5:0] val5 = 6'shA; // sign specifier before the base,
                 // sign extended to binary 111010
```

Signed shift operators

There are two types of shift operators: logical, denoted as << and >>, and signed or arithmetic, denoted as <<< and >>>.

Both logical and signed left shift operators, << and <<<, shift the left operand to the left by the number of bit positions given by the right operand. The rightmost bit positions will be filled with zeroes.

The logical right shift operator, >>, shifts its left operand to the right by the number of bit positions given by the right operand. It will fill the vacated bit positions with zeros.

If the result type is unsigned, the signed right shift operator, >>>, will fill the vacated bit positions with zeroes. If the result is signed, the operator will perform sign extension of the leftmost, or sign, bit. The following examples illustrate different shift operators:

```
reg [7:0]      init1 = 8'h1;
reg signed [7:0] init2 = 8'h80;

reg [7:0]      shift1, shift2;
reg signed[7:0] shift3, shift4;

initial shift1 = init1 << 2;   // result is 8'h04
initial shift2 = init1 <<< 2;  // result is 8'h04

initial shift3 = init2 >> 2;   // result is 8'h02
initial shift4 = init2 >>> 2;  // result is 8'h0E
```

Arithmetic operations

Arithmetic operations are based on the data type of the operands. If all operands are signed, the result is calculated using signed arithmetic. If either operand is unsigned, the arithmetic is unsigned.

```
wire signed [15:0] x, y;
wire signed [16:0] sum_signed;
wire [16:0] sum_unsigned;

assign sum_signed = x + y; // 2's complement addition
assign sum_unsigned = $unsigned(x) + y; // unsigned addition
```

Absolute value

The following is an example of calculating an absolute value of a signed number.

```
reg signed [11:0] val = 12'h801; // -2047 in signed decimal
wire signed [11:0] abs_val;
assign abs_val = val[11] ? ~val + 1'b1 : val;
       // If the sign bit is set, perform 2's complement
       // The result is 12'h7FF or 2047 in signed decimal
```

Arithmetic negation

Verilog supports arithmetic negation operator, as shown in the following example:

```
assign b = invert ? -a : a;
```

Synthesis tools usually implement this code by subtracting *a* from zero, and adding a multiplexor to select either positive or negative value.

There are several caveats that FPGA designers should be aware of when working with the signed arithmetic. Circuits that implement using signed arithmetic might have logic utilization differences that can be compared to the unsigned implementation. Logic utilization may increase or decrease, depending on the circuit.

There might be a mismatch between simulation and synthesis results. Performing post-synthesis gate level simulation can help you to be more confident that the logic was implemented correctly.

Because FPGA synthesis tools provide different level of support for signed arithmetic, the portability of the code might become an issue. Designers should always consult synthesis tool user guide for support of signed arithmetic constructs.

27. State machines

A Finite State Machine (FSM), or simply a state machine, is defined as a model composed of a finite number of states, transitions between those states, and actions. State machines are widely used in FPGA designs to implement control logic, packet parsers, algorithms, and for many other purposes. State machines can differ in complexity from a simple two-state structure to implement a control circuit, to a very complex structure with over a hundred states to implement a communication protocol.

The following figure shows a block diagram of a Mealy state machine, whose outputs are determined by both current state and the value of its inputs.

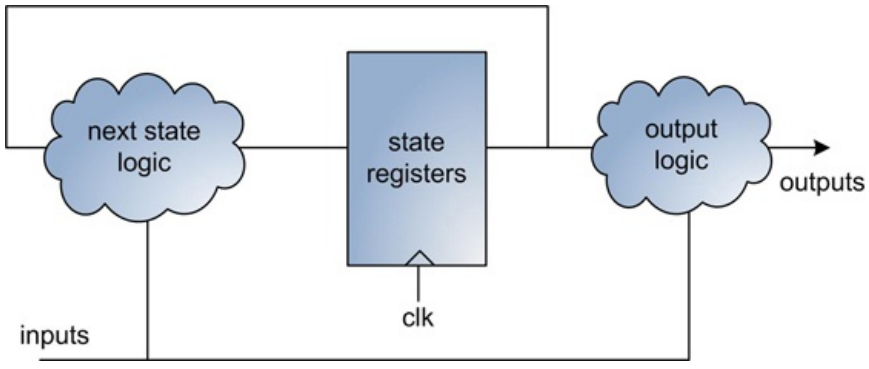

Figure 1: Mealy state machine

State machine coding techniques

Synthesis tools automatically infer state machine tools if the RTL follows the following coding guidelines (for Verilog HDL):

- State registers must be initialized using synchronous or asynchronous reset, or with register power-up value.
- State register type can be an integer or a set of parameters.
- Next-state equations can be described in a sequential or a separate combinatorial *always* block.

The following is an example of a state machine implementation with four states. It uses separate combinatorial *always* block for the next state logic. The circuit implemented in the example doesn't serve any practical purpose. The goal is to illustrate different state machine coding techniques and to show how synthesis tools implement them.

```
module state_machines( input clk,
                  input reset,
                  input [3:0] state_inputs,
                  output reg [3:0] state_outputs);

   localparam STATE_INIT   = 3'd0,
       STATE_ONE   = 3'd1,
       STATE_TWO   = 3'd2,
       STATE_THREE  = 3'd3;

   reg [2:0] state_cur,
           state_next;

   always @(posedge clk) begin
      if(reset) begin
         state_cur    <= STATE_INIT;
         state_outputs <= 4'b0;
  end
  else begin
   state_cur      <= state_next;
   state_outputs[0] <= state_cur == STATE_INIT;
   state_outputs[1] <= state_cur == STATE_ONE;
   state_outputs[2] <= state_cur == STATE_TWO;
   state_outputs[3] <= state_cur == STATE_THREE;
  end
end

always @(*) begin
  state_next = state_cur;

case(state_cur)
  STATE_INIT:  if(state_inputs[0]) state_next = STATE_ONE;
  STATE_ONE:   if(state_inputs[1]) state_next = STATE_TWO;
  STATE_TWO:   if(state_inputs[2]) state_next = STATE_THREE;
  STATE_THREE: if(state_inputs[3]) state_next = STATE_INIT;
endcase
end
endmodule // state_machines
```

XST produces the following synthesis log:

```
=========================================
*  HDL Synthesis
=========================================
Synthesizing Unit <state_machines>
   Found 4-bit register for signal <state_outputs>.
   Found 3-bit register for signal <state_cur>.
   Found finite state machine <FSM_0> for signal <state_cur>.
----------------------------------------
| States           | 4
| Transitions      | 8
| Inputs           | 4
| Outputs          | 4
| Clock            | clk (rising_edge)
| Reset            | reset (positive)
| Reset type       | synchronous
| Reset State      | 000
| Encoding         | auto
| Implementation   | LUT
----------------------------------------
Summary:
inferred   4 D-type flip-flop(s).
inferred   1 Finite State Machine(s).
=========================================
HDL Synthesis Report
Macro Statistics
# Registers                      : 1
 4-bit register                  : 1
# FSMs                 : 1
=========================================
Low Level Synthesis
=========================================
Analyzing FSM <MFsm> for best encoding.
Optimizing FSM <state_cur> on signal <state_cur[1:2]> with gray encoding.
-------------------
 State | Encoding
-------------------
 000   | 00
 001   | 01
 010   | 11
 011   | 10
-------------------
```

The above log shows that XST recognized four states in the state machine and changed the original state encoding to gray.

State machine synthesis options

The following is the list of XST synthesis options related to the state machine implementation.

FSM Encoding Algorithm (-fsm_encoding switch).

Auto: This is the default setting. XST will select the best encoding algorithm for each FSM. In the above example, XST has chosen gray.

One-hot: One register is associated with each state.

Gray: Encoding that guarantees only one bit change between two consecutive states.

Compact: Minimizes the number of bits in the state register.

Other encoding options are Johnson, Sequential, Speed1, and user-specific.

Other FPGA synthesis tools, such as Synopsys Synplify, provide similar FSM encoding options. However, the rules for automatic state extraction and optimization are different and, in most cases, proprietary.

RAM-Based FSM

XST provides an option to implement state machines using Block RAM resources. That is controlled by an -fsm_style switch. If XST cannot implement a state machine with BRAM, it issues a warning and implements it with LUTs. One case when a state machine cannot be implemented with BRAM is when it has an asynchronous reset.

The advantage of this option is more compact implementation of large state machines. A BRAM can be easily configured with any initial state values and can accommodate hundreds of states.

The following figure shows an implementation of a state machine with 256 states, 3-bit state inputs, and 10-bit outputs.

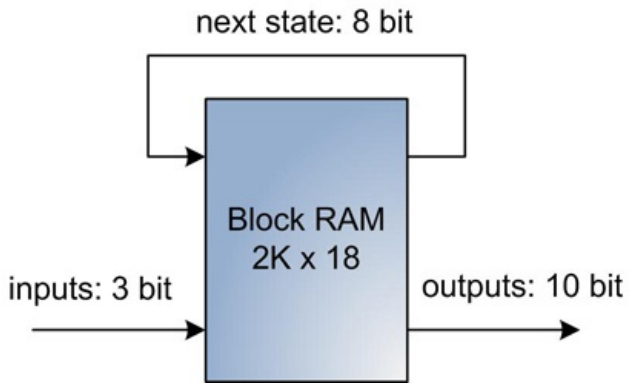

Figure 2: BRAM-based state machine

Safe FSM implementation

A state machine may enter an invalid state for various reasons. It can happen, for example, if state inputs and state registers are in different clock domains. If the state inputs aren't properly synchronized, state registers can enter a metastable state. When the state registers exit the metastable state, they can hold any value, including the one that is not part of the state machine's defined set of states.

In that event, the state machine will permanently stay in that state and will not react to state inputs. Other catalysts for entering an invalid state are single event upset (SEU) errors, which can occur in radioactive environments. They are caused by either a high-energy neutron or an alpha particle striking sections of the FPGA silicon that contain configuration data.

XST provides an option to recover from an invalid state by adding additional logic. That is controlled by the -safe_implementation switch. FPGA designers can implement other mechanisms to detect or correct an entry to an invalid state.

Safe FSM implementation is used in mission-critical systems, such as aerospace, medical, or automotive, where an FPGA must function in a harsh operating environment, gracefully recover from any error it encounters, and continue working.

In other cases, such as during the debug phase, designers do want the state machine to stay in an illegal state in order to be able to more easily find the root cause of the problem.

28. Using Xilinx DSP48 primitive

Xilinx DSP48 primitive is usually associated with digital signal processing applications. For example, high-definition video processing applications take advantage of the fixed precision multiplication. Floating-point processing is used in military radars. DSP48 primitives are cascaded to implement complex multiplications used in Fast Fourier Transforms (FFTs). Another commonly used function that takes advantage of DSP48 is the Finite Impulse Response Filter (FIR).

However, DSP48 primitive is highly configurable and can be used in a wide range of other applications, not only DSP-related ones. That has a benefit of saving a significant amount of FPGA logic resources.

DSP48 primitive is part of all Xilinx Virtex and Spartan FPGA families. The primitive has a different name and offers a slightly different feature set in each family. It's called DSP48E1 in Virtex-6, DSP48E in Virtex-5, DSP48A1 in Spartan-6, and DSP48A in Spartan-3.

DSP48 primitive supports a plethora of application and use cases. Some of them are adders and subtractors, multipliers and dividers, counters and timers, Single Instruction Multiple Data (SIMD) operations, pattern matching, and barrel shift registers. It supports signed and two's complement arithmetic and up to 48-bit operands. If needed, multiple DSP48 modules can be cascaded. DSP48 can operate at a speed of up to 600MHz in Virtex-6 and 390MHz in Spartan-6 FPGAs.

The following figure shows the block diagram of Virtex-6 DSP48E1 primitive.

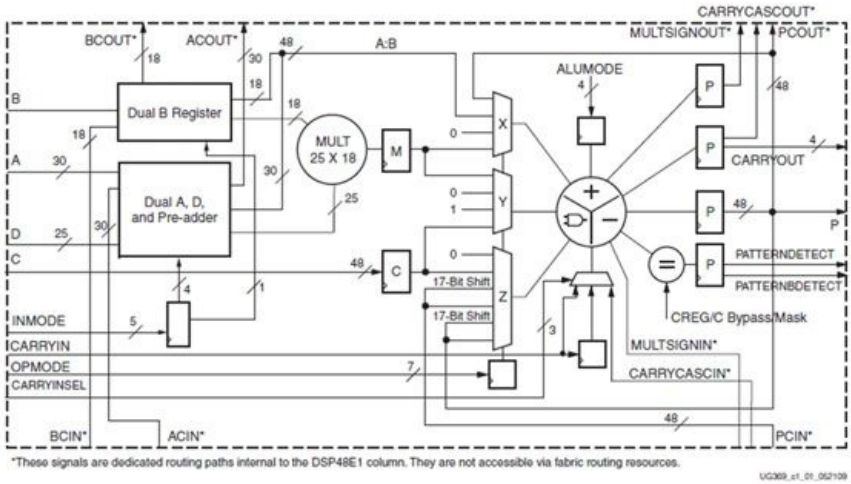

Figure 1: DSP48 block diagram (source: Virtex-6 FPGA DSP48E1 Slice, User Guide 369)

DSP48 consists of three main components: 25x18 bit multiplier, logic unit that performs three-input arithmetic and logic operations, input and output registers.

Implementation options

There are three main methods of using DSP48 primitives in the design.

A DSP48 primitive can be instantiated directly. This method enables a low-level access to all DSP48 features.

Xilinx CoreGen utility can be used to generate an IP core that takes advantage of the DSP48. The advantage of this method is that it is user-friendly and produces logic that is optimized for speed or area.

The most portable method is to infer DSP48 primitive by using certain coding styles and synthesis tool options. XST synthesis tool can automatically implement adders, subtractors, multipliers, and accumulators using DSP48. XST provides the following synthesis options that control the usage of DSP48 primitives.

Use DSP block (-use_dsp48 switch)

Auto: XST examines the benefits of placing each macro in DSP48 primitive, and then determines the most efficient implementation. Auto is the default setting.

Yes: XST places all macros in the DSP48 blocks whenever possible.

No: XST uses FPGA logic resources.

DSP Utilization Ratio (-dsp_utilization_ratio switch)

This option provides control over how XST takes advantage of available DSP48 resources. By default, XST attempts to use all available DSP48 resources.

The following are Verilog examples of a multiplier and an adder. They illustrate a coding style necessary to infer DSP48 primitive during the synthesis. The examples are built for Xilinx Spartan-6 LX45 FPGA with -3 speed grade and are available at the accompanying website.

Multipliers

The following is an example implementation of an 8-bit signed multiplier.

```
module mult(  input clk,
           input reset,
           input signed [7:0] a ,b,
           output reg signed [15:0] c);

   always @(posedge clk)
      if(reset)
         c <= 'b0;
      else
         c <= a * b;

endmodule // mult
```

XST synthesis produces the following report:

```
=========================================
* HDL Synthesis
=========================================
Synthesizing Unit <mult>.
   Found 16-bit register for signal <c>.
   Found 8x8-bit multiplier for signal <a[7]_b[7]_OUT>
   Summary:
      inferred   1 Multiplier(s).
      inferred  16 D-type flip-flop(s).
Unit <mult> synthesized.

=========================================================
*        Advanced HDL
=========================================================
Synthesizing (advanced) Unit <mult>.
Found pipelined multiplier on signal <a[7]_b[7]_OUT>:1 pipeline level(s) found in a register connected to the
multiplier macro output. Pushing register(s) into the multiplier macro.

INFO:Xst:2385 - HDL ADVISOR - You can improve the performance of the multiplier Mmult_a[7]_b[7]_OUT by adding 1
register level(s).
Unit <mult> synthesized (advanced).
=========================================================
Advanced HDL Synthesis Report

Macro Statistics
# Multipliers                          : 1
 8x8-bit registered multiplier             : 1

=========================================================
```

The synthesis report indicates that XST correctly inferred 8x8 bit multiplier as a multiplier macro and also pushed the 16-bit output register inside the macro.

Another multiplier type is multiplication by a constant. It can also be implemented by inferring a DSP48 primitive. An alternative implementation of multiplication by a constant is using an addition operation on a shifted multiplicand value. For instance, multiplication by 100 can be implemented as:

```
// c=a*100
assign c = a<<6 + a<<5 + a<<2;
```

Adder

The following is an example implementation of a 48-bit adder with a registered output.

```
module adder_sync_reset(input clk, reset,
                 input [47:0] a,b,
                 output reg [47:0] c);

  always @(posedge clk)
```

```
   if(reset)
     c <= 'b0;
   else
     c <= a + b;
endmodule // adder_sync_reset

=========================================================
*    HDL Synthesis
=========================================================
Synthesizing Unit <adder_sync_reset>.
   Found 48-bit register for signal <c>.
   Found 48-bit adder for signal <a[47]_b[47]_OUT>
   Summary:
 inferred   1 Adder/Subtractor(s).
 inferred  48 D-type flip-flop(s).
Unit <adder_sync_reset> synthesized.
=========================================================
Advanced HDL Synthesis Report

Synthesizing (advanced) Unit <adder_sync_reset>.
Found registered addsub on signal <a[47]_b[47]_OUT>:
1 register level(s) found in a register connected to the addsub macro output. Pushing register(s) into the addsub
macro.
Unit <adder_sync_reset> synthesized (advanced).
=========================================================
```

As in the previous example, the synthesis report indicates that XST correctly inferred 48-bit adder as a macro and also pushed the 48-bit output register inside the macro.

The following example shows the same implementation of a 48-bit adder with a registered output, but the register is reset asynchronously.

```
module adder_async_reset( input clk, reset,
                    input [47:0] a,b,
                    output reg [47:0] c);

   always @(posedge clk, posedge reset)
      if(reset)
         c <= 'b0;
      else
         c <= a + b;
endmodule // adder_async_reset
```

XST still correctly infers a 48-bit counter macro. However, it doesn’t push the output register inside the macro.

Double data rate technique

Typical FPGA designs run at much lower speeds than DSP48 primitive can support. In some cases designers can take advantage of the full computing performance of DSP48 by running the data through the primitive at the double data rate speed. An example use case is two low-speed video or protocol data streams that require the same processing.

The following figure illustrates the technique.

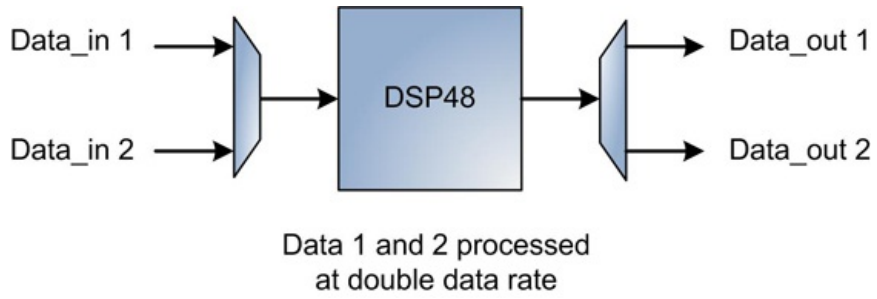

Figure 2: Processing the data at double data rate

Two datastreams are multiplexed into a single stream at twice the rate. The resulting datastream is processed by the DSP48 primitive and then demultiplexed back to two separate processed streams. For this technique to work, the data rate has to be at least twice as slow at the maximum supported rate of the DSP48. For example, if a DSP48 primitive can operate at 600MHz, the input data rate cannot exceed 300MHz.

There are several challenges that designers need to consider before using this technique. The amount of logic required to do multiplexing and demultiplexing of the datastreams can be substantial. Two datastreams have to have the same number of register delays. Single and double data rate clock domains have to be phase-aligned. And, finally, multiplexing and demultiplexing logic has to be placed in close proximity to the DSP48 primitive because it

might operate at a very high frequency.

29. Reset Scheme

The choice of a reset scheme has a profound effect on the performance, logic utilization, reliability, and robustness of the design. Different reset schemes have been discussed at length in magazines, white papers, online forums, and other technical publications. Reset scheme design is often a topic of heated debates among team members. Despite all that, there is no definitive answer on which reset scheme should be used in a given FPGA design.

Designing a good reset scheme that meets all the product requirements is a complex task. Factors that affect the decision making include previous experience of the designers, taking advantage of a working reset scheme from another similar design, using a third party IP core, and others.

In general, FPGA designers have the following options to choose from: asynchronous reset, synchronous reset, no reset, or a hybrid scheme that includes a combination of reset types. This Tip analyzes advantages and drawbacks of different reset schemes in order to enable FPGA designers to make the best decision applicable to a particular design.

Asynchronous reset

Asynchronous reset is defined as an input signal to a register or another synchronous element, which, when asserted, performs a reset of that element independent of the clock. Xilinx FPGA registers have the following configuration options: active high or low reset, and set or clear state upon reset assertion.

The following Verilog code is an example of implementing a register with an asynchronous reset.

```
reg [7:0] my_register;

always @(posedge clk, posedge rst) begin
  if(rst)
   my_register <= 8'h0;
  else
  my_register <= data_in;
end
```

Disadvantages of using asynchronous reset

Using asynchronous reset can cause intermittent design problems that are hard to troubleshoot. The following example illustrates the problem caused by delayed reset release. The circuit is a two-bit shift register. Upon reset, the left register is cleared and the right one is set.

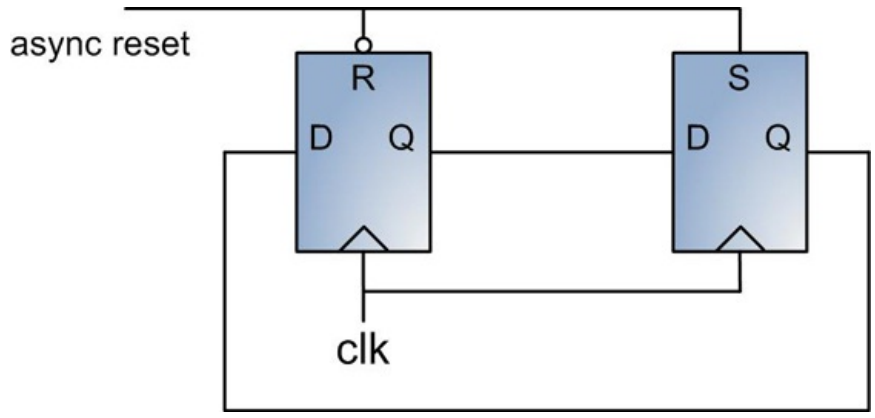

Figure 1: Shift register with asynchronous reset

After the reset is released, the contents of the right and left registers are toggled every rising clock edge. The following is a code example and a simulation waveform of the circuit.

```
   reg shift0, shift1;

   always @( posedge clk , posedge reset)
      if( reset ) begin
         shift0 <= 1'b0;
         shift1 <= 1'b1;
      end
```

```
else begin
   {shift1,shift0} <= #1 {shift0,shift1};
end
```

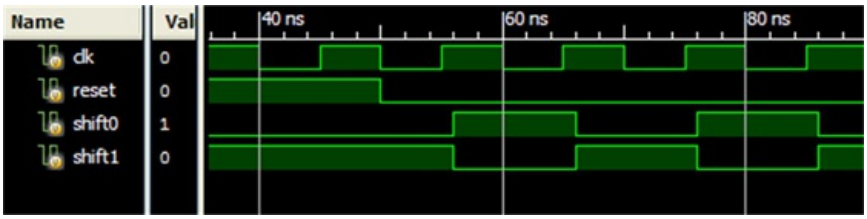

Figure 2: shift0 and shift1 toggle every clock

Because of the routing delays on the reset net, the reset of the left and right registers are not released at the same time. The functional problem with the circuit occurs when the reset of the left register is released before rising the edge of the clock, whereas the right register still stays at reset. The following simulation waveform illustrates the problem.

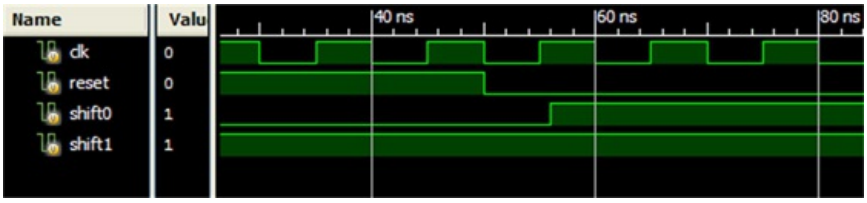

Figure 3: shift0 and shift1 are stuck at logic '1'

When the reset to the right register is finally released, both registers contain logic value of ‘1’.

The above example is not a hypothetical scenario. That can happen to state machines and other control logic in a real FPGA design with high logic utilization, and a high-fanout reset net. The reset release might not be seen at the same time by all state machine registers due to large delays. That can cause incorrect register initialization, and the state machine entering an invalid state. The problem debug is exacerbated by the fact that it is not reproducible in simulation, and it occurs intermittently in different FPGA builds.

The following figure illustrates another problem with using an asynchronous reset. It shows a circuit that consists of two registers. The data output of the left register is used as an input to a combinatorial logic “cloud”. The output of the “cloud” is driving an asynchronous reset of the right register.

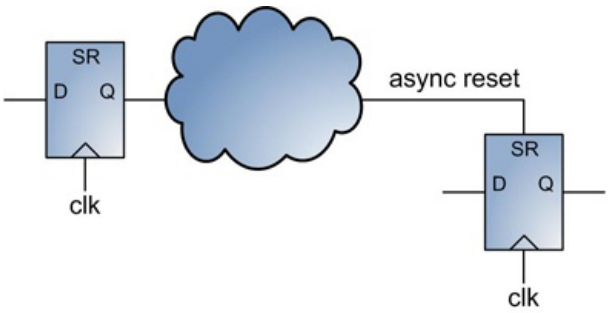

Figure 4: Asynchronous reset is driven by a combinatorial logic

One problem with this circuit is that there is a glitch on the asynchronous reset signal, which can be interpreted as a valid reset. Another problem is that if the asynchronous reset signal is not covered by the timing constraint, the right register might not be reset at the right time with respect to the clock, and as a consequence would be set to incorrect value.

By default, asynchronous reset nets are not included in the static timing analysis. Designers need to properly constrain all asynchronous reset nets, which is a more complex procedure than constraining synchronous elements.

Using asynchronous reset may result in sub-optimal logic utilization. Asynchronous resets prevent synthesis tools from performing certain logic optimizations, such as taking advantage of internal registers of DSP48 primitive (discussed in more detail in Tip #28). Several non-obvious logic optimization cases related to Control Sets when using synchronous resets are discussed in [2] and [3].

As shown in the first example, asynchronous reset nets rely on the propagation delays

within an FPGA. As a consequence, the design might behave differently for slightly different supply voltages or temperature conditions. That will also cause design portability issues to different FPGA families and speed grades.

Advantages of using asynchronous reset

Despite the numerous problems listed above, using asynchronous reset offers several advantages. Asynchronous reset doesn't require a clock to always be active. If a clock originates from an external device, and that device has periods with an anticipated loss of clock, for example during a power-saving mode, than synchronous reset cannot be used. Another advantage is faster physical implementation of a design. Asynchronous reset nets usually have more relaxed timing constraints comparing to the synchronous reset. It takes place and route tools less effort and time to meet those constraints. In large FPGA designs that utilize tens of thousands of registers and take several hours to build, any reduction in build time is a significant advantage.

The following figure shows an example of a simple circuit to generate an asynchronous reset.

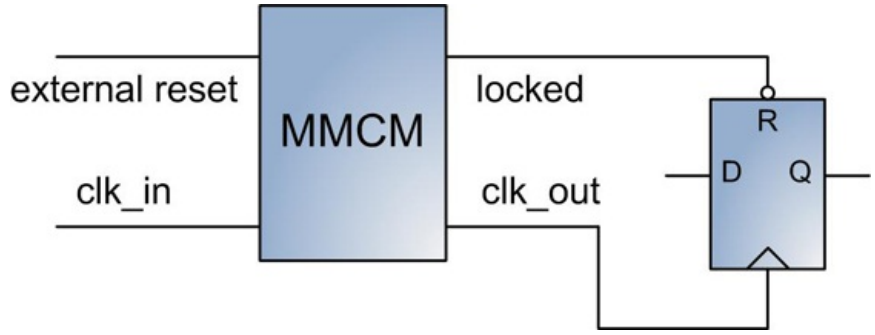

Figure 5: Using Xilinx MMCM locked output as a reset

The circuit uses inverted `locked` output of Virtex-6 MMCM primitive as a reset.

Synchronous reset

Synchronous reset is defined as an input signal to a register or another synchronous element, which, when asserted, performs a reset of that element. As the name suggests, synchronous reset takes place on the active edge of the clock. Xilinx FPGA registers with synchronous reset have the same configuration options as the ones with asynchronous reset. Moreover, the same hardware primitive can be configured as a register with either synchronous or asynchronous reset. The following is a Verilog example of implementing a register with a synchronous reset.

```
reg [7:0] my_register;

always @(posedge clk) begin
  if(rst)
   my_register <= 8'h0;
  else
   my_register <= data_in;
end
```

Xilinx recommends using synchronous reset scheme whenever possible. It adheres to synchronous design practices and avoids many problems discussed in the "Asynchronous reset" section.

As soon as the timing constraints are met, the design behavior is consistent for different voltages, temperature conditions, and process variations across different FPGA chips. That also makes the design more portable. Synchronous resets ensure that the reset can only occur at an active clock edge. Therefore, the clock works as a filter for small glitches on reset nets. Using synchronous reset is advantageous from the design logic utilization perspective. At the register level there is no difference in utilization: the same hardware primitive can be configured as a register with either synchronous or asynchronous reset. However, using synchronous reset helps synthesis and physical implementation tools perform different area optimizations.

No reset

Implementing synchronous logic without reset is yet another option that FPGA designers have at their disposal. The following is an example of implementing a register without a reset.

```
reg [7:0] my_register;
```

```
always @(posedge clk) begin
  my_register <= data_in;
end
```

By default, the register is initialized with '0' on FPGA power-up. Designers can initialize the register with any value during its declaration, as shown in the following example.

```
reg [7:0] my_register = 8'h55;
```

Implementing synchronous logic without reset offers some advantages. Reset nets, especially in large designs, consume a large amount of FPGA routing resources. Not using tens of thousands of reset signals helps improve design performance and decrease build time. Not using reset also enables replacing registers with dedicated FPGA primitives, such as Xilinx Shift Register LUTs (SRL), which can significantly lower overall logic utilization.

The disadvantage of not using reset is a less flexible design, which can only be initialized on FPGA power-up.

Hybrid reset scheme

In practice, reset scheme of a large FPGA design often contains a mix of reset types: synchronous, asynchronous, and no reset at all. The following figure depicts an example of such a hybrid reset scheme.

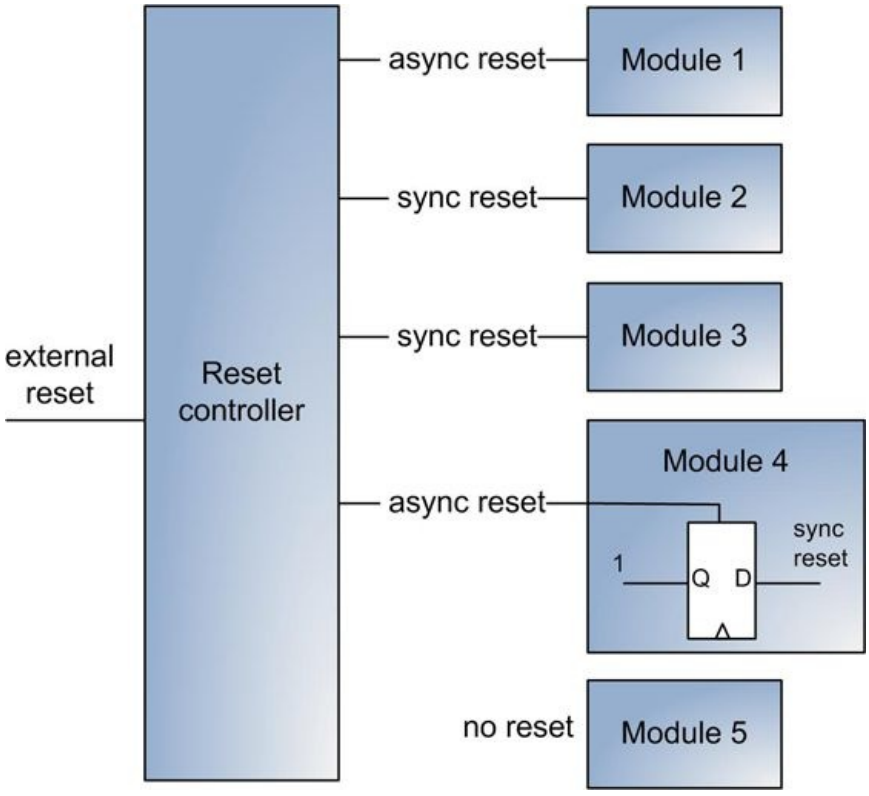

Figure 6: An example of a hybrid reset scheme

Modules 2 and 3 have synchronous reset in different clock domains. Module 1 uses asynchronous reset. Module 5 has no reset. Module 4 converts an asynchronous reset to synchronous internally.

An external asynchronous FPGA reset is connected to the Reset Controller. The Controller performs external reset synchronization, and sequences resets to Modules1-4 upon external reset release. Designing a robust reset synchronization circuit has unique challenges and pitfalls, which are discussed in [4].

The following figure is the FPGA die view of a hybrid reset scheme. Reset Controller is floor planned close to the center of the chip. The external reset input is coming from one of the IO pins. From the Reset Controller resets are distributed to different modules across the chip.

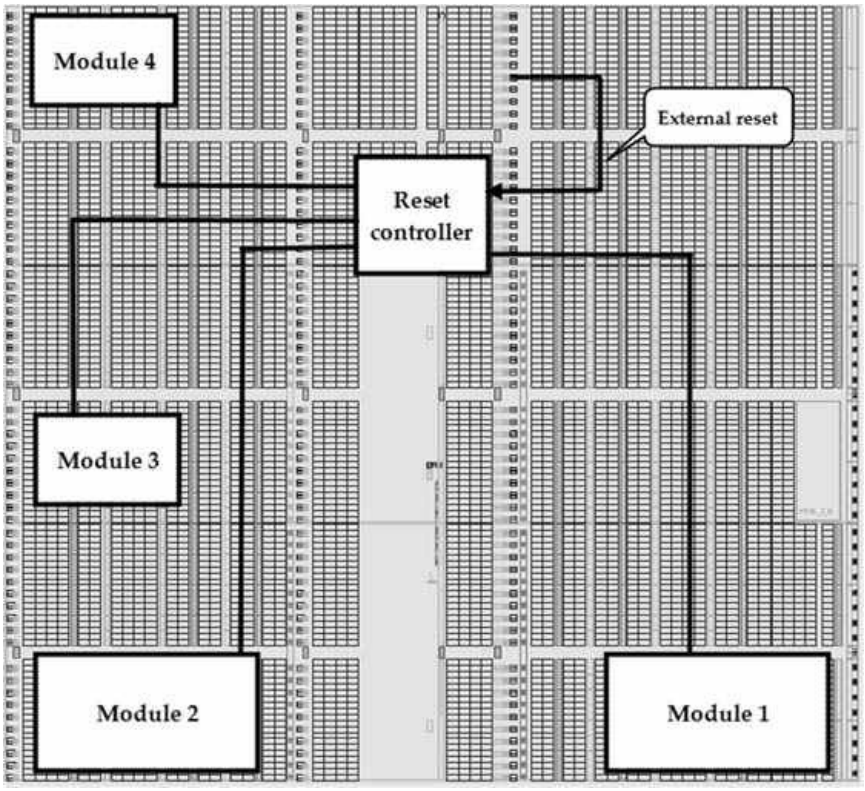

Figure 7: FPGA die view of a reset scheme

There are at least a couple of reasons for using a hybrid reset scheme. Large FPGA designs contain several IP cores and functional components with mixed reset schemes, which are often inflexible and cannot be changed.

A more subtle reason is related to the routing delays. A delay between two modules can exceed one or more clock periods. As an example, a routing delay across the mid-size Virtex-6 FPGA die is over 10ns. If the clock period is 3ns, it takes more than three clocks to cross the die. The consequence is that it's not feasible to design a pure synchronous reset scheme in a large FPGA.

Designs without external reset

Often a design doesn't have an external reset. One way to generate an explicit reset signal is to use a dedicated STARTUP_VIRTEX6 primitive, which Xilinx provides for Virtex-6 FPGAs. Designers can take advantage of the End Of Sequence (EOS) output of the STARTUP_VIRTEX6 primitive, which indicates the end of FPGA configuration. EOS signal is asynchronous, and will require a proper synchronization circuit to provide a clean synchronous reset.

However, this approach is not portable. For example, Xilinx Spartan FPGAs don't have an equivalent primitive.

Synthesis tool options

Synthesis tools provide several command-line options and attributes to control design reset.

XST has Asynchronous to Synchronous option (command line -async_to_sync), which treats all asynchronous reset signals as synchronous. It the option is enabled, it applies to all inferred sequential elements. The option will cause the mismatch between the original RTL and the post-synthesis netlist, and has to be used with caution.

"Use Synchronous Set" (use_sync_set) and "Use Synchronous Reset" (use_sync_reset) attributes control the use of dedicated synchronous set or reset inputs of a register. Those attributes have auto, yes, and no options. When the no option is specified, XST avoids using dedicated set or reset. In the auto mode, which is a default, XST attempts to estimate whether using dedicated synchronous set or reset will produce better results based on various metrics.

"Use Synchronous Set/Reset" attributes are applicable to the entire design or a particular instance.

Resources

[1] Ken Chapman, "*Get Smart About Reset: Think Local, Not Global*", Xilinx White Paper

WP272.
http://www.xilinx.com/support/documentation/white_papers/wp272.pdf

[2] Ken Chapman, "*Get your Priorities Right – Make your Design Up to 50% Smaller*", Xilinx White Paper WP275.
http://www.xilinx.com/support/documentation/white_papers/wp275.pdf

[3] Philippe Garrault, "*HDL Coding Practices to Accelerate Design Performance*", Xilinx Xcell Journal Issue 55.
http://www.xilinx.com/publications/archives/xcell/Xcell55.pdf

[4] Clifford E. Cummings, "*Get Asynchronous & Synchronous Reset Design Techniques - Part Deux*", SNUG 2003, Boston.
http://www.sunburst-design.com/papers/CummingsSNUG2003Boston_Resets.pdf

30. Designing Shift Registers

Shift registers find many uses in FPGA designs. Some of the shift register applications are converters between serial and parallel interfaces, logic delay circuits, pulse extenders, barrel shifters, and waveform generators. Linear Feedback Shift Register (LFSR) is a special type of shift register, whose input bit is a linear function of its previous state. LFSR is used in pseudo-random number generators, fast counters, Cyclic Redundancy Checkers (CRC), and many other applications.

Shift registers can have the following configuration options: shift register length, left or right data shift, clock enable, shift data multiple bits (barrel shifter), synchronous or asynchronous reset.

There are four main methods to implement shift registers in Xilinx FPGAs using flip-flops, SRLs, BRAMs, and DSP48 primitives. Each method offers advantages and drawbacks in terms of speed, logic utilization, and functionality.

Shift register implementation using flip flops

The following is a Verilog example of a shift register with clock enable, synchronous reset, one bit width, and 64-bit depth.

```
module shift_flops( input clk,reset,enable,
                input shift_din,
                output shift_dout );
  localparam SHIFT_REG_SIZE = 64;

  reg [SHIFT_REG_SIZE-1:0] shift_reg;

  assign shift_dout = shift_reg[SHIFT_REG_SIZE-1];

  always @(posedge clk)
   if(reset) begin
     shift_reg  <= 'b0;
   end
   else begin
     shift_reg  <= enable
         ? {shift_reg[SHIFT_REG_SIZE-2:0] , shift_din}
         : shift_reg;
   end
endmodule // shift_flops
```

An alternative implementation of the same shift register using Verilog for loop.

```
always @(posedge clk)
  if(reset) begin
   shift_reg  <= 'b0;
  end
   else begin
     for(ix=1; ix < SHIFT_REG_SIZE; ix = ix + 1)
      shift_reg[ix]  <= enable
        ? shift_reg[ix-1]
        : shift_reg[ix];
     shift_reg[0]  <= enable ? shift_din : shift_reg[0];
   end
```

Shift register implementation using SRL

Xilinx provides a RAM-based shift register IP core. It generates fast, compact FIFO-style shift registers using SRL primitives. User options allow creating either fixed-length or variable-length shift registers, specify width and depth, clock enable, and initialization.

The disadvantage of the SRL-based shift register is that it supports only left shift operation, and no reset. This is due to limitations of the SRL primitive.

Shift register implementation using Block RAMs

The following figure shows an example of a shift register implementation using a dual port BRAM.

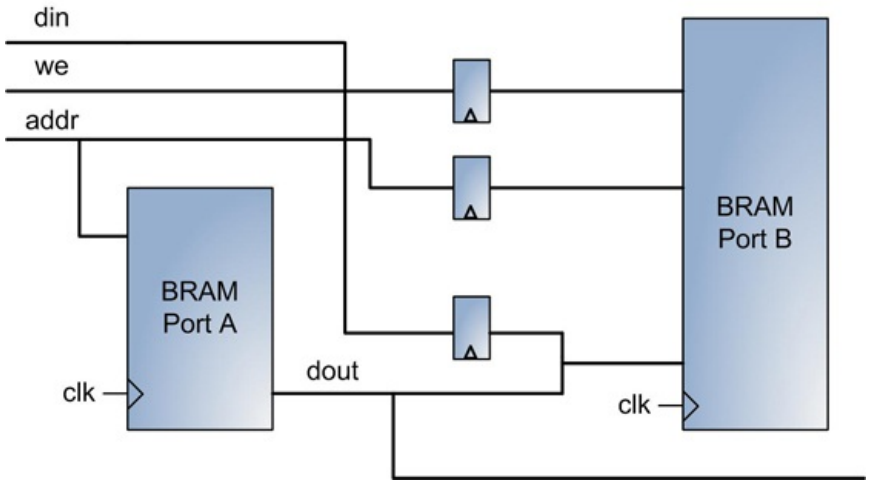

Figure 1: Shift Register Using Dual Port BRAM

The shift operation is performed by doing read from port A of the BRAM, concatenating delayed *din* with the data output from the port A, and writing the result to port B. The BRAM is configured in the write-first mode. That is, if both read and write operations to the same address are performed in the same clock, the write operation takes priority.

Another implementation option is a FIFO style, where each BRAM address contains a shift register value. This approach allows building very deep shift registers.

The shift register can operate at the full BRAM clock rate, which depends on the FPGA family and speed grade.

Shift register implementation using DSP48 primitive

Barrel shifter can be implemented using one or two DSP48 primitives. This implementation option is discussed in Tip #28.

Shift register synthesis options

XST provides Shift Register Extraction option to enable or disable shift register macro inference using SRL primitives.

-shreg_extract XST switch controls enabling shift register extraction for the entire project. Local control over shift register extraction can be achieved by placing `shreg_extract` attribute immediately before the module or signal declaration, as shown in the following Verilog example.

```
(* shreg_extract = "{yes | no}" *)
   reg [15:0] my_shift_register;
```

For the complete description of shift register implementation options and coding styles refer to the XST user guide documentation.

Logic utilization and performance results

The following table summarizes logic utilization and performance results for a 64-bit deep 1 bit data shift register using implementations discussed above.

All the shift registers were built for Xilinx Spartan-6 LX45 FPGA with 3-speed grade.

Table 1: Shift registers utilization and performance results

	Max freq	***Slices***	***LUTs***	***Regs***	***DSP48***	***BRAMs***
Flip flops	440	9	0	64	0	0
SRL	490	2	2	2	0	0

BRAM	275	1	0	2	0	1
DSP48	333	0	0	0	2	0

Shift register implementation using SRL has the highest performance. It is also more compact compared to the flip flop implementation.

The above performance and logic utilization results are only shown as an example, and should be interpreted judiciously. The results will vary significantly for different FPGA families, speed grades, tool options, and counter configuration.

All the source code, project files and the reports are available on the accompanying website.

31. Interfacing to external devices

FPGA designers have a plethora of options for designing an interface between FPGA and external devices. They can choose different bus type, electrical properties and speed.

Busses taxonomy

A bus is a design component used to connect two or more modules or devices. The following figure shows taxonomy of different busses.

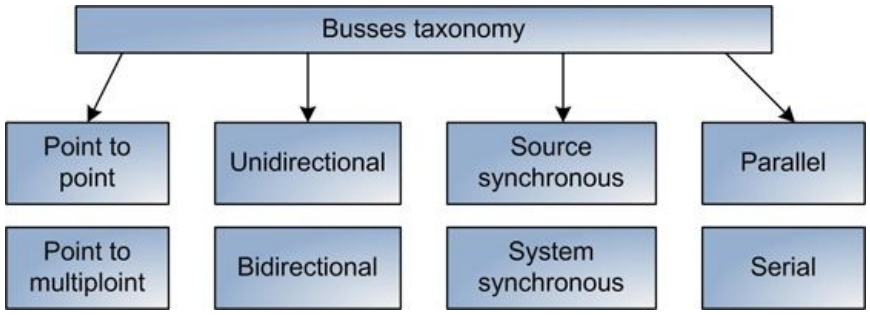

Figure 1: Busses Taxonomy

Point-to-point: a bus is connected to only two modules or devices. The advantage of a point-to-point bus is better signal integrity.

Point-to-multipoint: a bus is connected to more than two modules or devices. The advantage of a point-to-multipoint bus is lower pin count and simpler board design due to lower number of traces.

Unidirectional: the bus is driven by a single device or module connected to a bus. The advantage of a unidirectional bus is simpler implementation, and simpler termination scheme at the board level.

Bidirectional: the bus can be driven by any device or module connected to a bus. The advantage is lower pin count due to sharing of the same bus signals.

Source synchronous clock: is a technique of sourcing the clock along with the data on a bus. One advantage of this method is that it simplifies the clocking scheme of the system, and decouples bus driver from the recipients. This clocking scheme is used in several high-speed interfaces, for example SPI or PCI Express. The clock signal can be separate, or embedded into the data, and recovered on the receive side.

System synchronous clock: devices and modules connected to a bus are using a single clock. Unlike source synchronous method, system synchronous clocking scheme doesn't require a separate clock domain at the receiving device.

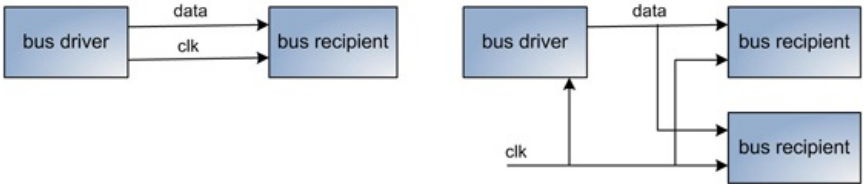

Figure 2: Source (on the left) and system (on the right) synchronous clocking scheme

Parallel busses

Parallel busses are typically used in low or medium frequency busses. There is no clear definition of what constitutes low or medium frequency, but the rule of thumb is below

100MHz is the low frequency, and between 100MHz and 300MHz is medium frequency.

Xilinx provides several primitives that can be used for designing parallel busses.

IDDR: a dedicated register that converts input dual data rate (DDR) data into a single data rate (SDR) output used in FPGA fabric.

ODDR: a dedicated register that converts input SDR data into a DDR external output.

IODELAY: design element used to provide a fixed or adjustable delay to the input data, and fixed delay to the output data. IODELAY is mainly used for data alignment of incoming and outgoing data.

IDELAYCTRL: used in conjunction with IODELAY for controlling the delay logic. One of the IDELAYCTRL inputs is a reference clock, which must be 200MHz to guarantee the tap delay accuracy in the IODELAY.

Adjusting input and output delays are required for calibrating a parallel bus operating at high frequencies. The adjustment process is dynamic and is performed after the board power up and periodically during the operation. There are several reasons for performing dynamic bus calibration. For instance, temperature and voltage fluctuations affect signal delay. In addition, trace delays on the PCB and inside the FPGA vary due to differences in manufacturing process. Because of the variable delays on different signals, parallel bus data may become skewed.

Serial busses

Parallel busses don't scale well to higher operating frequencies and bus width. Designing a parallel bus at high frequency is challenging because of the skew between multiple data signals, tight timing budget, and more complex board layout that requires to performing length-match of all bus signals. To overcome these challenges, many systems and several well-known communication protocols have migrated from a parallel to a serial interface. Two examples of this are Serial ATA, which is a serial version of the ATA or Parallel ATA, and PCI Express, which is the next generation of parallel PCI.

Another advantage of serial busses is a lower pin count. A disadvantage of a serial bus is that it has a more demanding PCB design. High speed serial links typically runs at multi-gigabit speed. As a result, they generate a lot more electromagnetic interference (EMI) and consume more power.

An external serial data stream is typically converted to the parallel data inside the FPGA. The module that does that is called SerDes (Serializer / Deserializer). A serializer takes in parallel data and converts it to a serial output at much higher rate. Conversely, a deserializer converts a high-speed serial input to a parallel output. Disadvantages of using SerDes are additional communication latency due to the extra steps required to perform serialization and deserialization; more complex initialization and periodic link training; and larger logic size.

The following figure shows a full duplex serial bus implemented as two SerDes modules.

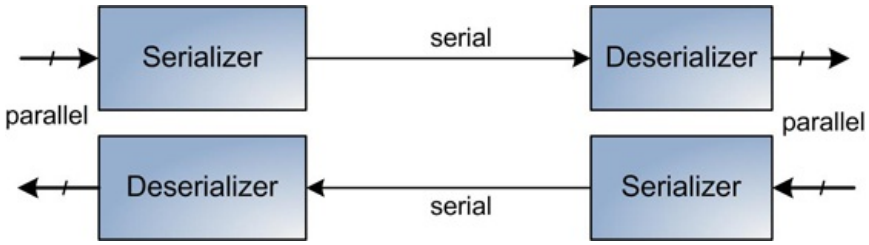

Figure 3: Full duplex serializer/deserializer

Xilinx provides several primitives that can be used for designing a serial bus.

ISERDES: a dedicated serial to parallel data converter to facilitate high-speed source synchronous data capturing. It has SDR and DDR data options and 2-to 6-bit data width.

GTP/GTX: an embedded transceiver module in some of the Virtex and Spartan FPGAs. It is a complex module, highly configurable, and tightly integrated with the FPGA logic resources. The following figure shows a block diagram of Virtex-6 GTX transceiver.

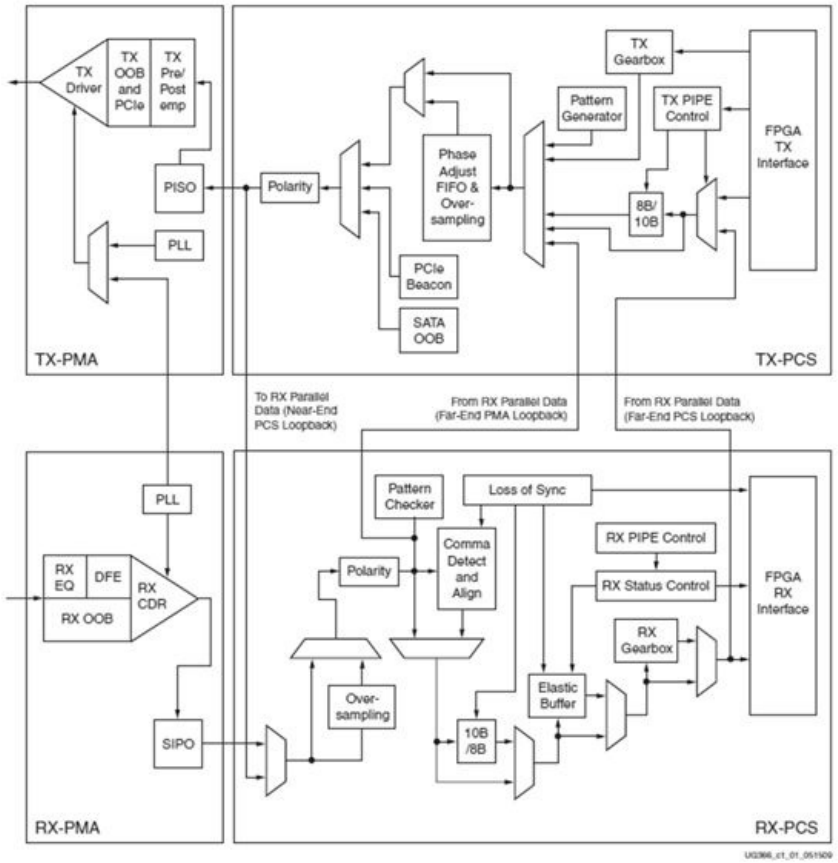

Figure 4: Transceiver block diagram (source: Virtex-6 FPGA GTX Transceivers, UG366)

Aurora

Aurora is a very efficient low-latency point-to-point serial protocol that utilizes GTP transceivers. It was designed to hide the interface details and overhead of the GTP. Xilinx provides an IP core with a user-friendly interface that implements Aurora protocol. Among the features that the core provides are different number of GTP channels, simplex and duplex operations, flow control, configurable line rate and user clock frequency.

The following figure illustrates point to point communication using Aurora core.

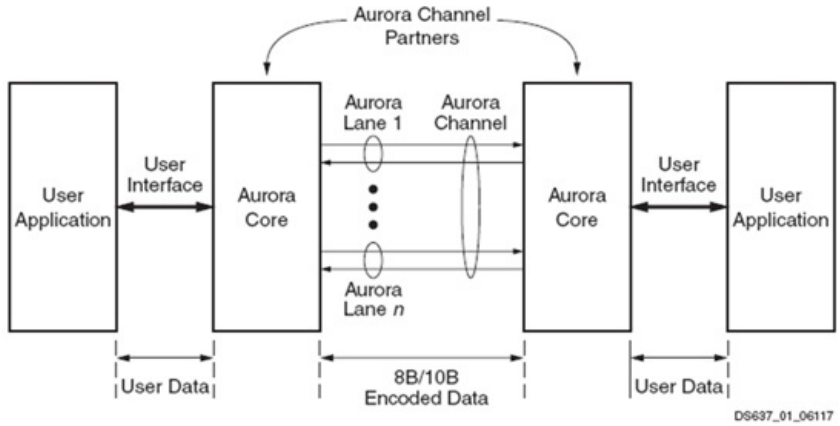

Figure 5: Using Aurora core (Source: Xilinx Aurora User Guide)

32. Using Look-up Tables and Carry Chains

Look-up tables and carry chains are combinatorial elements of a slice. In most cases, synthesis tools infer look-up tables and carry chains automatically from an HDL description. Sometimes designers want to use them explicitly to achieve higher performance. Another use is to gain low-level access for implementing custom circuits, for example priority trees, adders, or comparators, which contain carry chain structures and look-up tables.

Xilinx provides CARRY4 primitive to instantiate a carry chain structure, shown in the following figure.

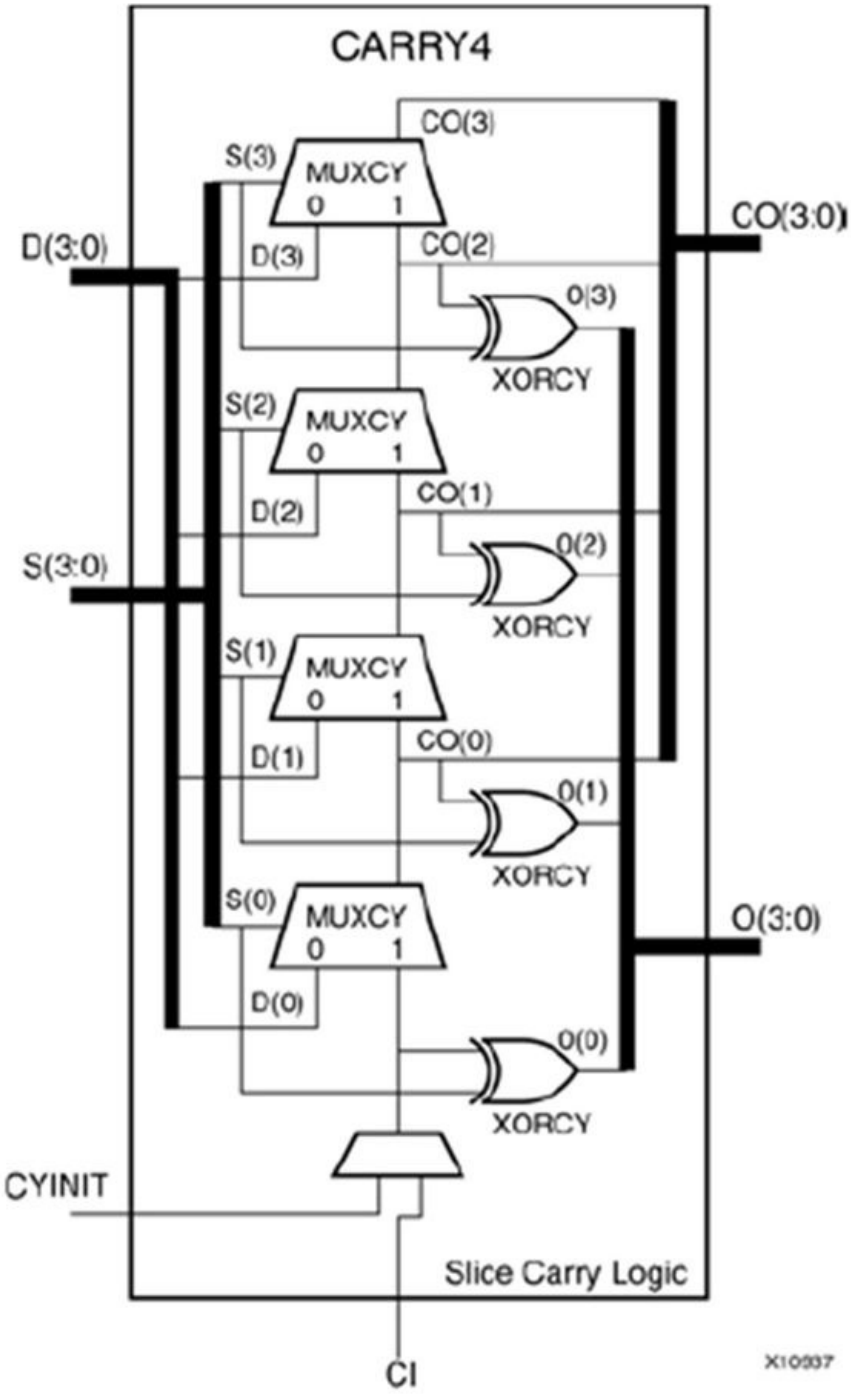

Figure 1: Carry chain logic (Source: Xilinx Virtex-6 Libraries Guide)

MUXCY and XORCY primitives that are part of the carry chain structure can also be instantiated directly.

Xilinx provides a variety of primitives for instantiating look-up tables. Some of the Virtex-6 family look-up tables are LUT5, LUT6, and LUT2. A detailed description of the look-up and carry chain primitives can be found in Xilinx Library Guide documentation for a specific FPGA family.

The following is a simple example of using three look-up tables to implement a 12-bit OR function.

```
// High-level Verilog implementation
assign user_out = |user_in[11:0];

// Implementation using explicit look-up tables instantiation
wire out, out1_14;

LUT6 #(
   .INIT ( 64'hFFFFFFFFFFFFFFFE ))
  out1 (
   .I0(user_in[3]),
```

```
   .I1(user_in[2]),
   .I2(user_in[5]),
   .I3(user_in[4]),
   .I4(user_in[7]),
   .I5(user_in[6]),
   .O(out));

  LUT6 #(
   .INIT ( 64'hFFFFFFFFFFFFFFFE ))
  out2 (
   .I0(user_in[9]),
   .I1(user_in[8]),
   .I2(user_in[11]),
   .I3(user_in[10]),
   .I4(user_in[1]),
   .I5(user_in[0]),
   .O(out1_14));

  LUT2 #(
   .INIT ( 4'hE ))
   out3 (
    .I0(out),
    .I1(out1_14),
    .O(user_out));
```

The INIT parameter to LUT2 and LUT6 primitives defines look-up table's logical function. It's calculated by assigning a 1 to corresponding INIT bit value when the associated inputs are applied. For example, the INIT value of 64'hFFFFFFFFFFFFFFFE in the above example indicates that the output of *out1* LUT6 instance is 0 for an all-zero input combination and 1 otherwise.

33. Designing Pipelines

Pipelines are used extensively in protocol and packet processing, DSP, image and video algorithms, CPU instruction pipelines, encryption/decryption, and many other designs.

Pipelining is defined as a process of splitting a logic function into multiple stages, such that each stage executes a small part of the function. The first pipeline stage can start processing a new input while the rest of the pipeline is still working on the previous input. This enables higher throughput and performance. However, pipelining also adds additional latency and logic overhead.

Latency, throughput, and the initiation interval are used to describe pipeline performance. Latency is defined as the number of clock cycles from the first valid input till the corresponding result available on the output of the pipeline. The initiation interval is the minimum number of clock cycles that must elapse between two consecutive valid inputs. Throughput is a measure of how often the pipeline produces the valid output.

To achieve the optimal performance of a pipeline, the amount of combinatorial logic between registered stages has to be balanced.

FPGA implementation of high performance pipelines presents unique challenges, and requires a good understanding of FPGA architecture, synthesis and physical implementation tool options in order to meet logic utilization and performance goals. A pipeline can be a complex logic structure that includes various synchronous components, such as DSPs, BRAMs, and Shift Register LUTs (SRLs). The pipeline can split into to several branches and merge back. It might include clock enable connected to every synchronous element to stall the pipeline when the input data is not available.

The following two figures show examples of complex pipelines. The first one implements a 32-bit AES encryption round. It includes registers, BRAMs, DSP, and four synchronized 8-bit dapathaths with feedback. The second one is the alpha-blending of two data streams.

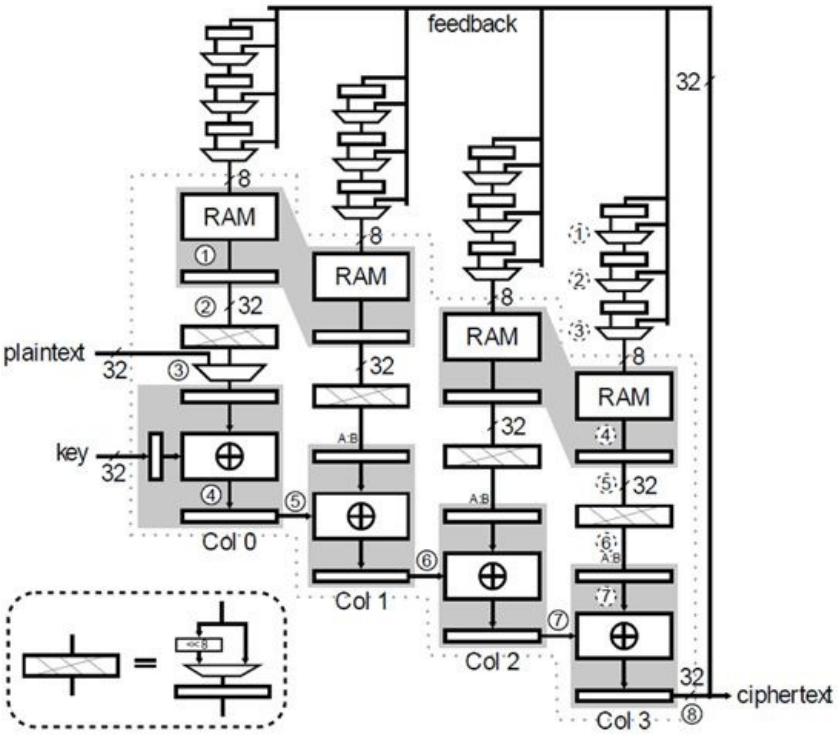

Figure 1: AES round (courtesy of Saar Drimer, PhD dissertation, 9/2009, University of Cambridge)

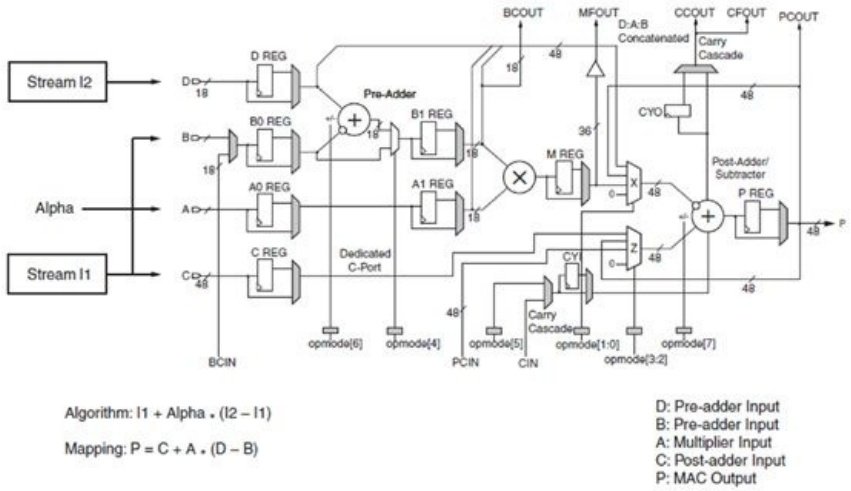

Figure 2: Alpha-blending pipeline (Source: Xilinx White Paper WP387)

Pipeline data width can vary from 1 to 256 bits, or even more than that. For example, the main datapath of a PCI Express Gen2 16-lane protocol processing engine can be 256 bit wide, and run at 250 MHz in order to process data at the bus speed. If the depth of such a pipeline is 8 stages, then the number of the pipeline registers is at least 256x8=2048. The pipeline also has to have clock enable signal connected to each synchronous element to stall when the data is not available. Therefore, the fanout of the clock enable signal in such a pipeline is 2048. Such a high fanout causes large routing delays, and decreased performance.

The following example of a simple pipeline is used to illustrate this problem.

```
module pipeline(input clk, reset, enable,
            input [127:0] data_in,
            output reg [127:0] data_out);
  reg [127:0] data_in_q,data_in_dq, data_in_2dq, data_in_3dq;

  always @(posedge clk)
   if(reset) begin
      data_in_q   <= 'b0;
      data_in_dq   <= 'b0;
      data_in_2dq  <= 'b0;
      data_in_3dq  <= 'b0;
      data_out    <= 'b0;
   end
   else begin
```

```
      data_in_q   <= enable ? data_in     : data_in_q;
      data_in_dq   <= enable ? data_in_q   : data_in_dq;
      data_in_2dq  <= enable ? data_in_dq   : data_in_2dq;
      data_in_3dq  <= enable ? data_in_2dq  : data_in_3dq;
      data_out    <= enable ? data_in_3dq  : data_out;
   end
endmodule // pipeline
```

The following figure shows a Xilinx FPGA Editor view of a high fanout enable net.

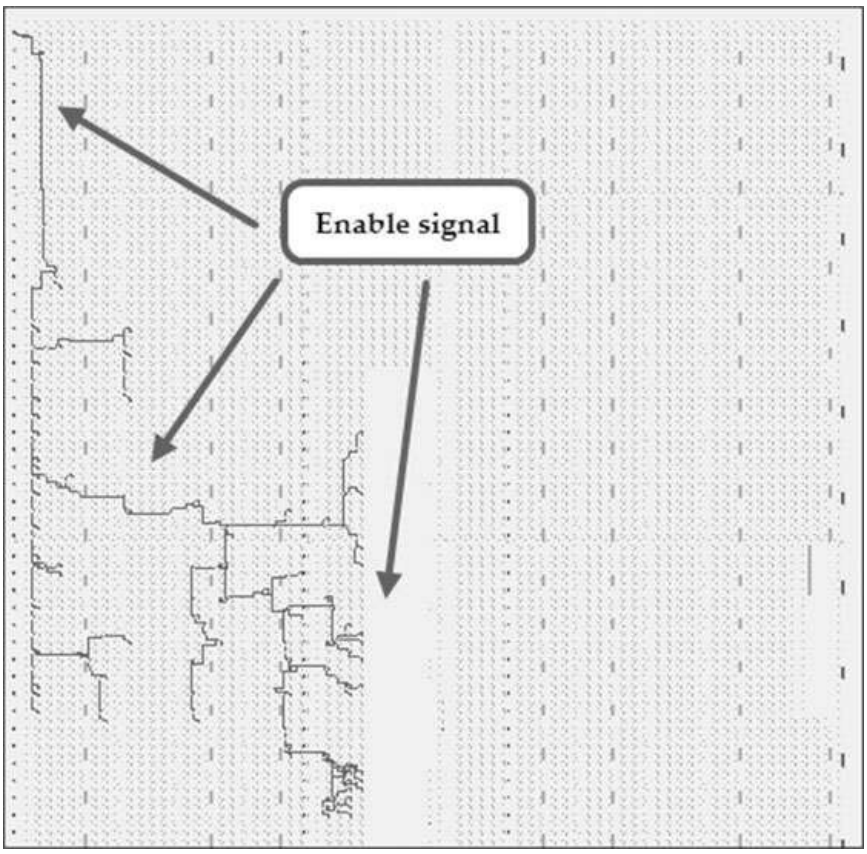

Figure 3: Distribution of *enable* signal

Executing FPGA Editor delay command on the enable net produces the following results (only part of the log is shown):

```
delay
Building the delay mediator...
Net "enable":
   driver - comp.pin "enable.I", site.pin "H21.I"
   1.987ns - comp.pin "data_out_212.CE", site.pin "SLICE_X0Y72.CE"
   1.893ns - comp.pin "data_out_30.CE", site.pin "SLICE_X0Y74.CE"
   1.799ns - comp.pin "data_out_211.CE", site.pin "SLICE_X0Y76.CE"
   1.425ns - comp.pin "data_out_17.CE", site.pin "SLICE_X1Y52.CE"
   1.799ns - comp.pin "data_out_26.CE", site.pin "SLICE_X5Y82.CE"
   2.625ns - comp.pin "data_out_35.CE", site.pin "SLICE_X7Y67.CE"
   5.815ns - comp.pin "data_out_751.CE", site.pin "SLICE_X40Y31.CE"
   5.900ns - comp.pin "data_out_7111.CE", site.pin "SLICE_X40Y32.CE"
   5.688ns - comp.pin "data_out_691.CE", site.pin "SLICE_X40Y33.CE"
   5.697ns - comp.pin "data_out_891.CE", site.pin "SLICE_X40Y34.CE"
   5.078ns - comp.pin "data_out_491.CE", site.pin "SLICE_X40Y42.CE"
   4.996ns - comp.pin "data_out_851.CE", site.pin "SLICE_X40Y50.CE"
   5.790ns - comp.pin "data_out_75.CE", site.pin "SLICE_X42Y31.CE"
   4.594ns - comp.pin "data_out_412.CE", site.pin "SLICE_X43Y57.CE"
```

The delay of the enable net from source to destination registers ranges between 1.276ns to 5.9ns.

There are several approaches to reduce the maximum delay. One option is to use dedicated low-skew routes. However, FPGAs have limited amount of dedicated routes, which are also used by the global clocks. Another method is to manually replicate the logic that drives clock enable in order to reduce the fanout. That can be also achieved by limiting the maximum fanout of a net. Xilinx XST provides -max_fanout global synthesis option, which sets the fanout limit on nets. If the limit is exceeded, the logic gates are replicated. Floorplanning can be used to constrain the area of the pipeline logic, which also reduces routing delays. Finally, the entire pipeline can be redesigned such that it doesn't require a global enable signal.

Limiting maximum fanout of the above example to 20 results in significant improvement, as shown in the following FPGA Editor log (only part of the log is shown):

```
delay
Net "enable_IBUF_33":
   1.038ns - comp.pin "enable_IBUF_13.D6", site.pin "SLICE_X33Y58.D1"
   0.934ns - comp.pin "enable_IBUF_11.A6", site.pin "SLICE_X34Y58.A3"
   0.581ns - comp.pin "enable_IBUF_9.A6", site.pin "SLICE_X34Y65.A4"
   0.930ns - comp.pin "enable_IBUF_12.A6", site.pin "SLICE_X35Y58.A3"
   0.722ns - comp.pin "enable_IBUF_10.C6", site.pin "SLICE_X35Y65.C2"
   0.380ns - comp.pin "enable_IBUF_8.A6", site.pin "SLICE_X40Y64.A3"
    driver - comp.pin "enable_IBUF_7.C", site.pin "SLICE_X40Y65.C"
   0.291ns - comp.pin "enable_IBUF_7.D6", site.pin "SLICE_X40Y65.D3"
Net "enable_IBUF_8":
   0.453ns - comp.pin "data_out_851.CE", site.pin "SLICE_X36Y62.CE"
   0.288ns - comp.pin "data_out_701.CE", site.pin "SLICE_X36Y64.CE"
   0.468ns - comp.pin "data_out_8111.CE", site.pin "SLICE_X36Y67.CE"
   0.365ns - comp.pin "data_out_881.CE", site.pin "SLICE_X40Y60.CE"
    driver - comp.pin "enable_IBUF_8.A", site.pin "SLICE_X40Y64.A"
   0.965ns - comp.pin "data_out_791.CE", site.pin "SLICE_X40Y78.CE"
   1.251ns - comp.pin "data_out_751.CE", site.pin "SLICE_X40Y85.CE"
```

Now, the maximum delay of the `enable` net is 1.251ns.

Another consideration is the efficient packing of pipelined designs into Slices. A Xilinx Virtex-6 Slice contains eight registers and four LUTs. In a register-dominated pipeline, in which the number of LUTs and registers is not balanced, Slice LUTs can be wasted. XST provides the `-reduce_control_set` option, which reduces the number of control sets and improves logic packing into Slices during MAP.

Retiming is a register-balancing technique that can be used for improving the timing performance of a pipeline. Retiming moves registers across combinatorial logic, but does not change the number of registers in a cycle or path from a primary input to a primary output of the pipeline. In Xilinx, build flow retiming is controlled by `-retiming` MAP option.

34. Using Embedded Memory

Xilinx FPGAs have two types of embedded memories: a dedicated Block RAM (BRAM) primitive, and a LUT configured as Distributed RAM. Each memory type can be further configured as a single- and dual-ported RAM or ROM.

The following figure shows the embedded memory taxonomy in Xilinx FPGAs.

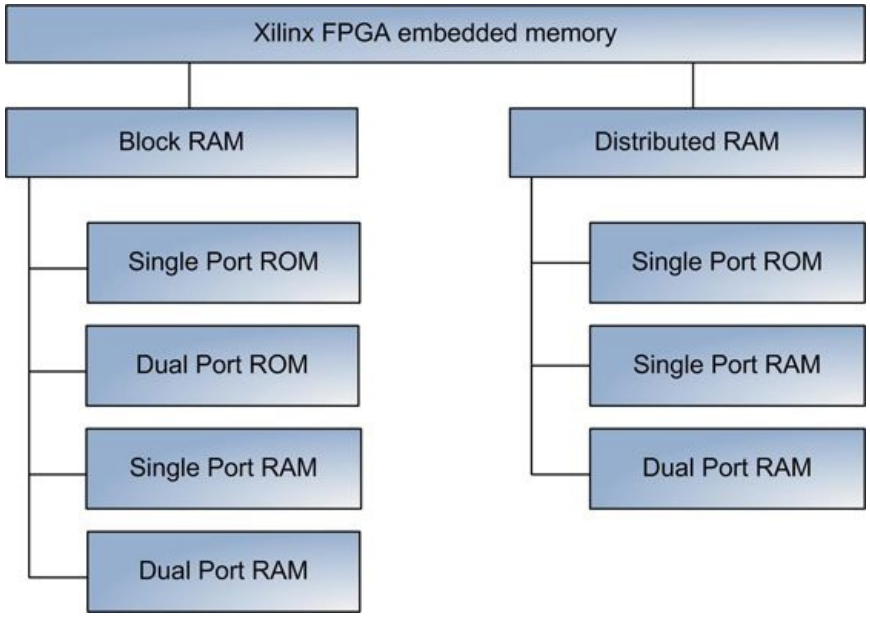

Figure 1: Xilinx FPGA embedded memory taxonomy

Block RAM

The following figure shows the IO ports of a Virtex-6 BRAM primitive.

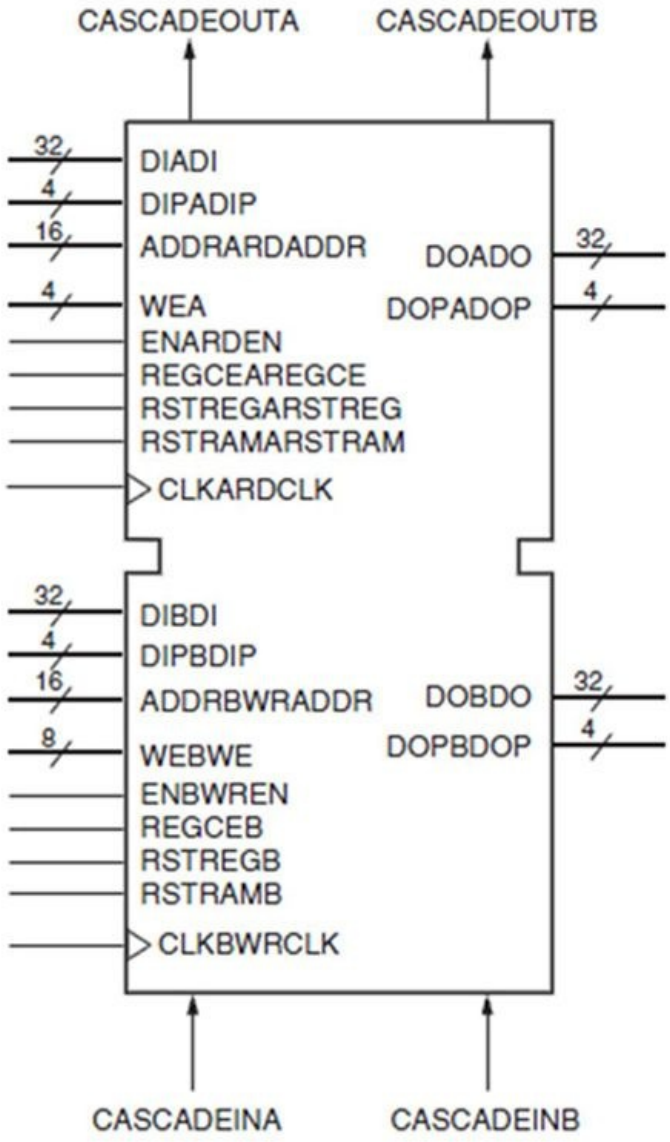

Figure 2: BRAM primitive (source: Xilinx User Guide UG363)

A BRAM in Virtex-6 FPGAs can store 36K bits, and can be configured as a single- or dual-ported RAM. Other configuration options include data width from 1 to 36 bit, memory depth up to 32K entries, enabling or disabling of data error detection and correction, register outputs, and cascading of multiple BRAM primitives.

There are three ways to incorporate a BRAM in the design. A BRAM can be directly instantiated using one of the primitives such as RAMB36E1 or RAMB18E1. This method enables low-level access to all BRAM features. The Xilinx CoreGen tool can be used to generate a customizable core, such as a FIFO, which includes BRAMs.

BRAMs can be inferred if the RTL follows the coding style defined by a synthesis tool. The main advantage of the inference method is code portability. The following is an example of a simple single-port RAM, which infers a BRAM during synthesis.

```
module bram_inference(   input clk,
                  input [15:0]     mem_din,
                  input [9:0]      mem_addr,
                  input          mem_we,
                  output reg [15:0] mem_dout);

   reg [15:0] ram [1:1024];

   always @(posedge clk) begin
     if(mem_we)
      ram[mem_addr] <= mem_din;

     mem_dout <= ram[mem_addr];
   end
endmodule // bram_inference
```

Synthesizing this example with XST produces the following log.

```
=========================================================
*  HDL Synthesis
=========================================================
Synthesizing Unit <bram_inference>.
Found 1024x16-bit single-port RAM <Mram_ram> for signal <ram>.
   Found 16-bit register for signal <mem_dout>.
   Summary:
     inferred   1 RAM(s).
     inferred  16 D-type flip-flop(s).
Unit <bram_inference> synthesized.
=========================================================
*  Advanced HDL Synthesis
=========================================================
Synthesizing (advanced) Unit <bram_inference>.
INFO:Xst:3040 - The RAM <Mram_ram> will be implemented as a BLOCK RAM, absorbing the following register(s):
<mem_dout>
   -----------------------------------------------------
   | ram_type         | Block
   -----------------------------------------------------
   | Port A
   |    aspect ratio   | 1024-word x 16-bit
   |    mode          | read-first
   |    clkA          | connected to signal <clk>
   |    weA           | connected to signal <mem_we>
   |    addrA         | connected to signal <mem_addr>
   |    diA           | connected to signal <mem_din>
   |    doA           | connected to signal <mem_dout>
   | optimization      | speed
   -----------------------------------------------------
Unit <bram_inference> synthesized (advanced).
=========================================================
Advanced HDL Synthesis Report
Macro Statistics
# RAMs                          : 1
 1024x16-bit single-port block RAM      : 1
=========================================================
```

The log indicates that XST inferred one BRAM, and also absorbed a 16-bit `mem_dout` register.

The inference rules and limitations are specific to a synthesis tool. For example, XST cannot infer dual-ported BRAMs with different port widths, read and write capability on both ports, or different clocks on each port.

BRAM initialization

It is often required to preload BRAM with predefined contents after FPGA configuration is complete. Designers have several options to perform BRAM initialization. If a memory is generated using CoreGen tool, BRAM contents can be initialized from a special memory coefficient (COE) file passed to C_INIT_FILE_NAME parameter. The following is an example of a COE file.

```
; 32-byte memory initialization
memory_initialization_radix = 16;
memory_initialization_vector =
6c600000,
6c800001,
6ca00002,
b8000006,
6c600007
```

If a BRAM is instantiated directly using RAMB36E1 or RAMB18E1 primitives, the contents can be initialized using INIT_A and INIT_B attributes directly in the code. The following is partial code to instantiate a `RAMB36E1` primitive.

```
RAMB36E1 #(
  .INIT_00(256'h1234),
  .INIT_01(256'h5678),
  .READ_WIDTH_A(18),
  .READ_WIDTH_B(9),
  //...other attributes
)
RAMB36E1_instance (
  .WEA(WEA),
  .WEBWE(WEBWE)
  //...other ports
);
```

The disadvantage of both methods is that it requires re-synthesis of the design after changing BRAM initialization values.

Xilinx FPGA Editor allows making modifications to BRAM contents. The following figure illustrates using FPGA

Editor to perform BRAM initialization.

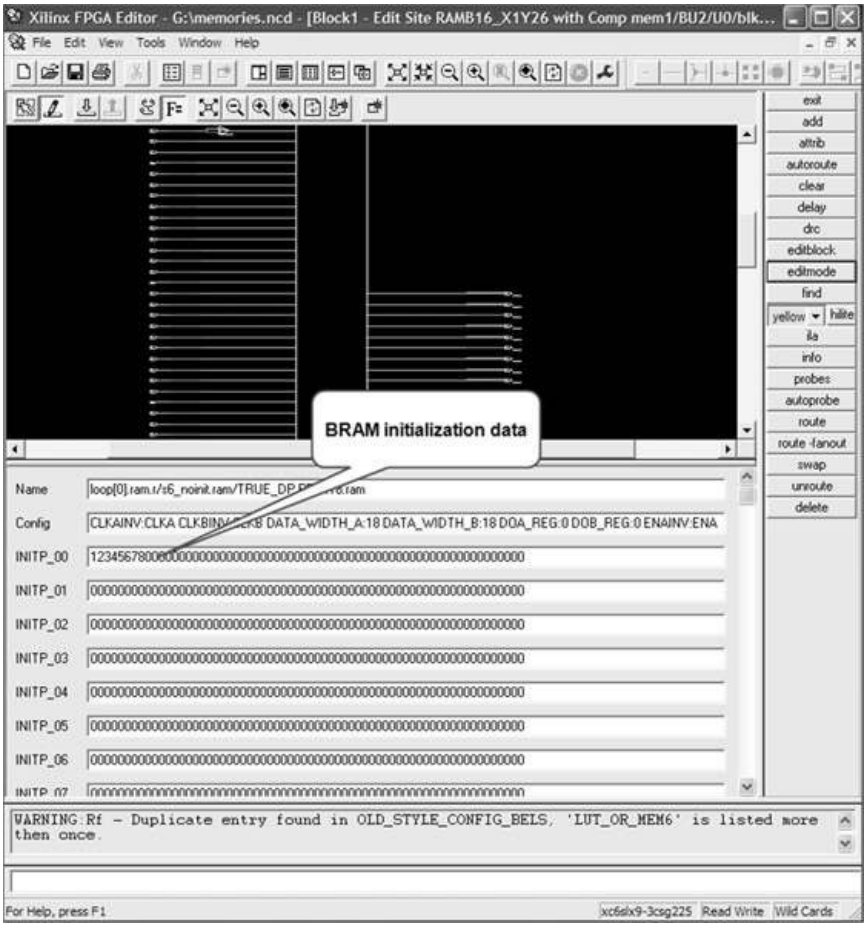

Figure 3: BRAM initialization using FPGA Editor

There are 128 INIT_A/B parameters 256 bit each, which cover the entire contents of a BRAM. Using FPGA Editor is convenient for making small BRAM modifications. It doesn't require performing re-synthesis or full physical implementation, only bitstream generation using Xilinx *bitgen* tool. The disadvantage of this method is the need to know physical locations of a BRAM module, which can change between builds.

The last method to do BRAM initialization is using Xilinx *data2mem* utility.

BRAM design challenges

There are several pitfalls and design challenges FPGA developers might encounter while using BRAMs.

All BRAM inputs, including reset, address, and data, are synchronous to the clock of the corresponding port. Not observing setup and hold timing requirements on the inputs will result in data corruption or other functional problems.

Asserting BRAM reset will cause the reset of flags and output registers. However, it has no impact on internal memory contents.

BRAM primitive has relatively slow clock-to-data time on the output data port. It ranges from 1.6ns to 3.73ns in Virtex-6 FPGAs depending on the speed grade and configuration. One implication of this is the inability to connect complex combinatorial logic to the data output in high speed designs. Enabling BRAM output register will reduce clock-to-data time to the value ranging between 0.6ns and 0.94ns, depending on the speed grade and configuration. The disadvantage of using an output register is increased read latency.

A BRAM has a minimum read latency of one clock. Designers have an option of using Distributed RAM in asynchronous read mode, which has no read latency.

Using BRAM in a dual-ported mode might result in data corruption in case of colliding writes from both ports.

Using BRAM primitives in high speed designs requires careful floorplanning, because routing delays to and from a BRAM might consume significant part of the timing budget. Distributed RAM, which is implemented using generic logic resources, doesn't have this problem.

Distributed RAM

Distributed RAMs are implemented used general purposed logic resources. Similarly to BRAM, there are three ways to incorporate a Distributed RAM in the design: directly instantiation of one of the Distributed RAM primitives, using CoreGen tool to generate a customizable core, and inferring in the RTL.

The following code example of a ROM, which infers Distributed RAM during synthesis.

```
module rom_inference(input clk,
                input [2:0]      mem_addr,
                output reg [15:0] mem_dout);
   reg [15:0] rom [1:8];
```

```
   always @(posedge clk) begin
      mem_dout <= rom[mem_addr];
   end
endmodule // rom_inference
```

Synthesizing this example with XST produces the following log.

```
=========================================================
*  HDL Synthesis
=========================================================
Synthesizing Unit <rom_inference>.
Found 8x16-bit single-port Read Only RAM <Mram_rom> for signal <rom>.
Found 16-bit register for signal <mem_dout>.
Summary:
  inferred   1 RAM(s).
  inferred  16 D-type flip-flop(s).
=========================================================
HDL Synthesis Report
Macro Statistics
# RAMs                                 : 1
 8x16-bit single-port Read Only RAM    : 1
# Registers                            : 1
 16-bit register                       : 1
=========================================================
*  Advanced HDL Synthesis
=========================================================
INFO:Xst:3048 - The small RAM <Mram_rom> will be implemented on LUTs in order to maximize performance and save block RAM resources. If you want to force
its implementation on block, use option/constraint ram_style.
   -----------------------------------------------------
   | ram_type         | Distributed
   -----------------------------------------------------
   | Port A
   |    aspect ratio   | 8-word x 16-bit
   |    weA         | connected to signal <GND>
   |    addrA        | connected to signal <mem_addr>
   |    diA         | connected to signal <GND>
   |    doA         | connected to internal node
   -----------------------------------------------------
Unit <rom_inference> synthesized (advanced).
=========================================================
Advanced HDL Synthesis Report
Macro Statistics
# RAMs                                 : 1
 8x16-bit single-port distributed Read Only RAM   : 1
# Registers                            : 16
 Flip-Flops                           : 16
=========================================================
```

Both BRAM and Distributed RAM require synchronous data write. However, unlike BRAM, Distributed RAM has an asynchronous read mode with zero latency. Distributed RAMs are typically used for small RAMs and ROMs.

Resources

[1] Virtex-6 FPGA Memory Resources, Xilinx User Guide UG363
http://www.xilinx.com/support/documentation/user_guides/ug363.pdf

[2] Xilinx Block Memory Generator
http://www.xilinx.com/products/ipcenter/Block_Memory_Generator.htm

[3] Xilinx Distributed Memory Generator
http://www.xilinx.com/support/documentation/ip_documentation/dist_mem_gen_ds322.pdf

35. Understanding FPGA Bitstream Structure

Bitstream is a commonly used term to describe a file that contains the full internal configuration state of an FPGA, including routing, logic resources, and IO settings. The majority of modern FPGAs are SRAM-based, including Xilinx Spartan and Virtex families. On each FPGA power-up, or during subsequent FPGA reconfiguration, a bitstream is read from the external non-volatile memory such as flash, processed by the FPGA configuration controller, and loaded to the internal configuration SRAM.

There are several situations when designers need to have a good understanding of internal structure of the FPGA bitstream. Some examples are low-level bitstream customizations that cannot be performed using parameters to the FPGA physical implementation tools, implementing complex configuration fallback schemes, creating short command sequences for partial FPGA reconfiguration via internal configuration port (ICAP), reading configuration status, and others. Unfortunately, reverse engineering and tampering with the bitstream to illegally obtain proprietary design information belongs to the list of use cases.

Bitstream format

Xilinx FPGA bitstream structure is shown in the following figure.

Figure 1: Xilinx FPGA bitstream structure

The bitstream consists of the following components: padding, synchronization word, commands for accessing configuration registers, memory frames, and desynchronization word.

Padding

Padding data is a sequence of all zeros or all ones and is ignored by the FPGA configuration controller. Padding data is required for separating bitstreams in non-volatile memory. It is convenient to use all ones for padding, because this is the state of flash memories after performing an erase operation.

Synchronization word

Synchronization word is a special value (0xAA995566) that tells the FPGA configuration controller to start

processing subsequent bitstream data.

Desynchronization word

Desynchronization word indicates to the FPGA configuration controller the end of the bitstream. After desynchronization, all bitstream data is ignored until the next synchronization word is encountered.

Commands

Commands are used to perform read and write access to FPGA configuration controller registers. Some of the commands that are present in every bitstream are IDCODE, which uniquely identifies the FPGA device the bitstream belongs to, Frame Address Register (FAR) and Frame Data Register (FDRI), and No Operation (NOOP) that is ignored.

Memory frames

Memory frame is a basic configuration unit in Xilinx FPGAs. A frame has a fixed size, which varies in different FPGA families. In Virtex-6 devices a frame has 2592 bits. Every Virtex-6 device has a different number of frames ranging from 7491 in the smallest LX75T to 55548 in the largest LX550T. A frame is used for independent configuration of several slices, IOs, BRAMs, and other FPGA components. Each frame has an address, which corresponds to a location in the FPGA configuration space. To configure a frame, bitstream uses a sequence of FAR and FDRI commands.

The Virtex-6 FPGA Configuration User Guide [1] contains sufficient documentation on a bitstream structure and commands to access registers in the FPGA configuration controller. However, the detailed documentation of memory frames is unavailable not only for Xilinx FPGAs but also for FPGAs from other vendors.

Xilinx bitgen utility

Bitgen is a Xilinx utility that takes a post place-and-route file in Native Circuit Description (NCD) format and creates a bitstream to be used for FPGA configuration. Bitgen is a highly configurable tool with more than 100 command-line options that are described in the Command Line Tools User Guide [2]. Some of the options are bitstream output format, enabling compression for reducing the bitstream size and FPGA configuration speed, using CRC to ensure data integrity, encrypting the bitstream, and many others.

Example

The following is an example of a short bitstream used for a difference-based partial reconfiguration. It is parsed by a script that annotates bitstream commands with a user friendly description. The script is written in Perl and is available on the accompanying website.

```
FFFFFFFF   Padding
FFFFFFFF   Padding
000000BB   Bus Width
11220044   8/16/32 BusWidth
FFFFFFFF   Padding
FFFFFFFF   Padding
AA995566   SYNC command
20000000   NO OP
30008001   write 1 word to CMD
00000007   RCRC command
20000000   NO OP
20000000   NO OP
30018001   write 1 word to ID
04250093   IDCODE
30008001   write 1 word to CMD
00000000   NULL command
30002001   write 1 word to FAR
00000000   FAR [Block 0 Top Row 0 Col 0 Minor 0]
30008001   write 1 word to CMD
00000001   WCFG command
20000000   NO OP
30002001   write 1 word to FAR
0000951A   FAR [Block 0 Top Row 1 Col 42 Minor 26]
20000000   NO OP
30004195   write 405 words to FDRI
   data words  0...404
30002001 write 1 word to FAR
001014A8 FAR [Block 0 Bot Row 0 Col 41 Minor 40]
20000000 NO OP
300040F3 write 243 words to FDRI
   data words  0...242
3000C001 write 1 word to MASK
00001000 MASK
30030001 write 1 word to CTL1
00000000 CTL1
30008001 write 1 word to CMD
00000003 LFRM command
20000000 NO OP
20000000 NO OP
20000000 NO OP
30002001 write 1 word to FAR
00EF8000 FAR [Block 7 Top Row 31 Col 0 Minor 0]
30000001 write 1 word to CRC
1C06EA42 CRC
30008001 write 1 word to CMD
0000000D DESYNCH command
20000000 NO OP
```

Examination of the bitstream reveals synchronization and desynchronization commands, IDCODE that belongs to Virtex-6 LX240T FPGA, and two frames of 405 and 243 words each.

Resources

[1] Virtex-6 FPGA Configuration User Guide UG360
http://www.xilinx.com/support/documentation/user_guides/ug360.pdf

[2] Xilinx Command Line Tools User Guide
http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_2/devref.pdf

36. FPGA Configuration

The configuration of all modern FPGAs falls into two categories: SRAM-based, which uses an external memory to program a configuration SRAM inside the FPGA, and non-volatile FPGAs, which can be configured only once.

Lattice and Actel FPGAs use a non-volatile configuration technology called Antifuse. The main advantages of a non-volatile FPGA configuration are simpler system design that doesn't require external memory and configuration controller, lower power consumption, lower cost, and faster FPGA configuration time. The biggest disadvantage is fixed configuration.

Most modern FPGAs are SRAM-based, including the Xilinx Spartan and Virtex families. On each FPGA power-up, or during subsequent FPGA reconfiguration, a bitstream is read from the external non-volatile memory, processed by the configuration controller, and loaded to the internal configuration SRAM. The configuration SRAM holds all the design information to configure logic, IO, embedded memories, routing, clocking, transceivers, and other FPGA primitives.

The following figure shows Xilinx Virtex-6 configuration architecture.

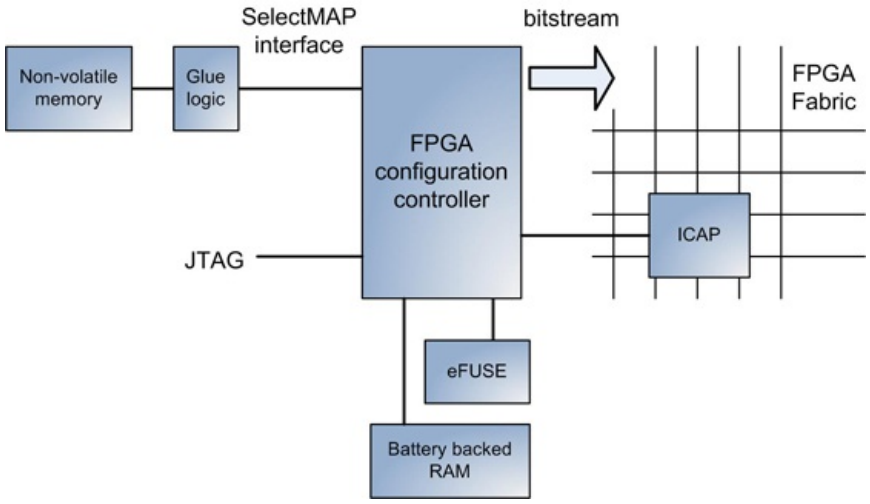

Figure 1: Xilinx Virtex-6 configuration architecture

The configuration is performed by the configuration controller inside the FPGA. The bitstream is stored in external non-volatile memory, such as flash. The external memory is connected to the configuration controller using the SelectMAP interface, which is Xilinx-specific. Additional glue logic might be required to bridge the SelectMAP and external memory interfaces. In addition, a bitstream can be loaded into the configuration controller via JTAG or ICAP. Bitstreams can be optionally encrypted to provide higher security. Internal Battery Backed RAM (BBR) and eFuse hold an encryption key required for bitstream decryption.

Each bit of the FPGA configuration memory, also called configuration memory cell, is initialized with the corresponding bit in the bitstream. The output of each memory cell is connected to a configurable functional block, such as LUT, register, BRAM, IO, routing, and others. The following figure shows a configuration memory cell connected to a multiplexer to set a specific routing path between components in the FPGA fabric. The logic state of zero or one is set during the FPGA configuration stage.

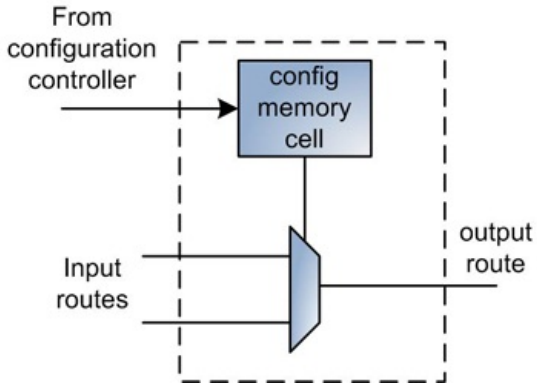

Figure 2: FPGA routing configuration

Xilinx FPGA configuration modes

There are several FPGA configuration modes that address different usage modes. The following figure shows taxonomy of Xilinx FPGA configuration modes.

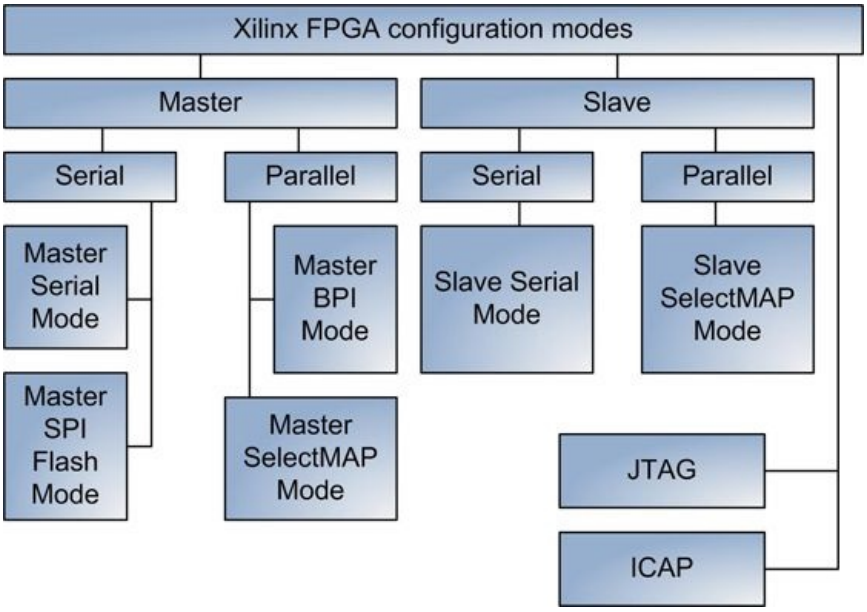

Figure 3: Taxonomy of FPGA configuration modes

Configuration modes are divided into two categories: master and slave. In master configuration modes, an FPGA controls configuration process. In slave modes, FPGA configuration is controlled by an external device, such as a microcontroller, CPLD, or another FPGA. In addition, there are two special configuration modes using JTAG and Internal Configuration Access Port (ICAP). There are four data width sizes to support different external memories: 32-bit, 16-bit, 8-bit, and 1-bit (serial).

The following is a brief overview of configuration modes.

JTAG

JTAG interface is mainly used during the debug. A special adapter is connected to dedicated FPGA pins to interface with Xilinx ChipScope and iMPACT software applications.

ICAP

A dedicated ICAP primitive interfaces with the user logic to perform configuration from within the FPGA fabric.

Master Serial Mode

In Master Serial Mode the FPGA is controlling the Xilinx Platform Flash to provide the configuration data. Xilinx Platform Flash is a special non-volatile flash memory designed to interface with Xilinx FPGAs directly using the SelectMAP interface.

Master SPI Flash Mode

In Master SPI Flash Mode, the FPGA is controlling serial SPI Flash to provide the configuration data.

Master SelectMAP Mode

In Master SelectMAP Mode, the FPGA is controlling Xilinx Platform Flash to provide 8- or 16-bit wide configuration data.

Master BPI Mode

In Master BPI Mode, the FPGA is controlling a parallel NOR Flash to provide 8- or 16-bit wide configuration data.

Slave Serial and SelectMAP Mode

In Slave Serial Mode, an external device, such as a microcontroller, CPLD, or another FPGA, controls the FPGA configuration process.

Designing FPGA configuration scheme

There are several design considerations for selecting the most appropriate FPGA configuration scheme for a particular design. The main selection criteria are:

- Choosing whether the configuration process is controlled by an external device (slave modes), or by FPGA itself (master modes). Master modes are the simplest from a system complexity standpoint but may not be suitable for all designs. The slave mode interface can be as simple as connecting the serial interface directly to the processor's IO pins and toggling them to read the bitstream data into the configuration controller.
- Choosing the type of external non-volatile memory and its size for storing one or several FPGA bitstreams. Although the cost of an external memory is relatively low compared with the cost of an FPGA, it still is not negligible. Designers can choose among SPI flash, parallel NOR flash, or Xilinx Platform Flash. In some designs an FPGA can be configured directly from the Host using an external processor connected to an FPGA configuration controller using one of the slave modes.
- The choice of the data width – serial, 8-bit, 16-bit, or 32-bit – affects the configuration speed and the

available number of FPGA IOs for the design.
- Field upgradability of the configuration bitstream can be an important requirement. The configuration scheme has to support cases in which the bitstream is getting corrupted while being programmed into a non-volatile memory.
- Xilinx FPGAs provides an option to encrypt a bitstream in cases in which higher design security is required. Decryption keys can be stored either in internal BBR or eFuse. BBR storage is volatile and requires an external battery. The advantage of using BBR is the relative ease of reprogramming the keys when compared with a non-volatile eFuse.

Calculating configuration time

In many applications FPGA configuration time is critical, and it is important to accurately estimate the time during configuration scheme selection. The configuration time depends on the bitstream size, clock frequency, and the data width of the configuration interface, and is defined by the following formula:

```
Config time = bitstream size * clock frequency * data width.
```

The following table provides the expected configuration times for the smallest and largest Xilinx Virtex-6 FPGAs using a 50MHz clock over different data widths of the configuration interface.

Table 1: FPGA configuration time

Virtex-6 FPGA	*Bitstream size*	*Config time over Serial interface*	*Config time over 8-bit interface*	*Config time over 32-bit interface*
LX75T	26,239,328	525ms	66ms	16.4ms
HX565T	160,655,264	3.2sec	402ms	100.4ms

Xilinx configuration-related primitives

The following table provides a list of configuration-related primitives supported by Xilinx Virtex-6 FPGAs.

Table 2: Xilinx Virtex-6 configuration-related primitives

Primitive	*Description*
USR_ACCESS_VIRTEX6	Enables access to a 32-bit configuration register within configuration logic
ICAP_VIRTEX6	Enables access to all configuration controller functions within FPGA fabric
CAPTURE_VIRTEX6	Enables readback register capture control
FRAME_ECC_VIRTEX6	Enables error detection and correction of configuration frames
EFUSE_USR	Provides internal access to 32-bit non-volatile fuses specific to the design
DNA_PORT	Provides access to a Device DNA port, which holds a unique device ID

Resources:

[1] Configuration for Xilinx Virtex-6 FPGA
http://www.xilinx.com/products/design_resources/config_sol/v6/config_v6.htm

37. FPGA Reconfiguration

The term reconfiguration refers to reprogramming an FPGA after it is already configured. There are two types of FPGA reconfiguration: full and partial. Full reconfiguration reprograms the entire FPGA, whereas partial reconfiguration replaces certain parts of the design while the rest of the design remains operational. Partial reconfiguration is not considered a special case of full reconfiguration, because both are fundamentally the same. Performing a partial FPGA reconfiguration is done using the same methods as full configuration: JTAG, ICAP, or SelectMAP interface, which is described in Tip #37. The bitstream structure is the same for both full and partial reconfiguration methods.

FPGA reconfiguration offers several benefits. It allows for the sharing of the same FPGA fabric across multiple designs. That in turn reduces FPGA size, cost, and system complexity. Full and partial reconfiguration opens the possibility for many innovative FPGA applications that would be cost prohibitive to implement otherwise. Some of the application examples that take advantage of the FPGA reconfiguration are DSP, audio, or video processors that change the processing algorithm based on the user input, communication controllers with integrated deep packet inspection engine, which switch a packet processor based on the protocol, and many others.

There is a lot of industry and academic research on the subject of FPGA reconfiguration, which produces a constant stream of interesting application notes, research papers, and dissertations.

Although partial reconfiguration is not a new functionality, it has not yet reached the mainstream. Design and implementation flows, tool support, and even the nomenclature are constantly evolving to become more user friendly. The ultimate goal is to provide a simple and transparent design flow to FPGA developers without requiring them to have detailed knowledge of configuration logic and bitstream structure.

Partial reconfiguration is a complex functionality riddled with multiple challenges during design implementation, tool flow, and the reconfiguration procedure itself. One challenge is to complete a smooth handoff during FPGA configuration change without disrupting the operation of the remaining design, or compromising its integrity. The FPGA fabric and IOs cannot be kept at reset, which is the case during full reconfiguration. Another challenge is

preventing the static part of the design from entering an invalid state while the dynamic part is being changed. Designers must properly define and constrain interfaces between static and dynamic parts so that FPGA physical implementation tools place logic and use exactly the same routing resources for both.

There are three partial reconfiguration flows that can be used for Xilinx FPGAs: difference-based, partition based, and using dynamic reconfiguration port.

Difference-based partial reconfiguration

Difference-based partial reconfiguration [2] is most suitable for making small design changes to LUT equations, IO characteristics, and BRAM contents. The following is a simple difference-based partial reconfiguration code and flow example. It can run on a Xilinx ML605 Virtex-6 development board.

```
// original module: turn on LED when both buttons are pressed
module top(input btn0,btn1,output led);
   assign led = btn0 & btn1;
endmodule // top

// partial reconfiguration module: turn on LED when either button is pressed
module top_pr(input btn0,btn1, output led);
   assign led = btn0 | btn1;

endmodule // top_pr

# Constraints file: the same for both designs
NET "btn0" LOC = "A18"  ;
NET "btn1" LOC = "H17"  ;
NET "led"  LOC = "AD21" ;

# LUT that implements LED function is locked into specific slice.
# For the original design LUT function is btn0 & btn1
# For the partial reconfig design LUT function is btn0 | btn1
INST "led" AREA_GROUP = "led";
AREA_GROUP "led" RANGE = SLICE_X65Y168:SLICE_X65Y168;

# bitgen command to generate a partial reconfiguration bitstream
# ActiveReconfig and Persist options prevent global reset
# during configuration change
$ bitgen -g ActiveReconfig:Yes -g Persist:Yes -r top_orig.bit top_pr.ncd top_pr.bit
```

`top_orig.bit` : the bitstream of the original design
`top_pr.ncd` : the post place-and-route output of the partial reconfiguration design
`top_pr.bit` : the resulting partial reconfiguration bitstream

The difference-based partial reconfiguration flow of the above example consists of the following steps:

1. Build `top` module. The result is `top_orig.bit` bitstream file
2. Build `top_pr` module. The result is `top_pr.ncd` post place-and-route file
3. Use `top_orig.bit` bitstream and `top_pr.ncd` to generate a bitstream that contains the difference in the LED LUT equation between the two designs.

Partition-based partial reconfiguration

Unlike the difference-based variation, partition-based partial reconfiguration flow supports reconfiguring large parts of the FPGA design. At the time of writing this book, it is supported by the PlanAhead tool for the Xilinx Virtex-6 family only (Spartan-6 FPGAs are not supported). PlanAhead provides an integrated environment to configure, implement, and manage partial reconfiguration projects using partitions. A very brief overview of the design and implementation flow is as follows:

- The FPGA developer designates parts of the design to be reconfigured.
- A region that contains necessary logic, embedded memory, IO, and other resources is allocated on FPGA die.
- The developer defines all possible design variants to occupy that region.
- The PlanAhead tool manages all the details of building such as design, including managing multiple netlists, static, and reconfigurable parts on the design, performing DRC, and producing appropriate bitstreams.

Xilinx application note XAPP883 [3] provides an example of using partial reconfiguration to enable fast configuration of an embedded PCI Express interface block.

Dynamic Reconfiguration

Another method of changing settings in Xilinx GTX transceivers, Mixed Mode Clock Manager (MMCM), and SystemMonitor primitives is by using the dynamic reconfiguration port (DRP). DRP provides a simple interface to the user logic and doesn't require detailed knowledge of configuration registers and the bitstream structure. As an example, DRP allows dynamic change of output clock frequency, phase shift, and duty cycle of MMCM [4].

Resources

[1] Xilinx FPGA partial reconfiguration portal
http://www.xilinx.com/tools/partial-reconfiguration.htm

[2] Difference-Based Partial Reconfiguration, Xilinx Application Note XAPP290
http://www.xilinx.com/support/documentation/application_notes/xapp290.pdf

[3] PCI Express Partial Reconfiguration, Application Note XAPP883
http://www.xilinx.com/support/documentation/application_
notes/xapp883_Fast_Config_PCIe.pdf

[4] MMCM Dynamic Reconfiguration, Xilinx Application Note XAPP878
http://www.xilinx.com/support/documentation/application_notes/xapp878.pdf

38. Estimating Design Size

There are several situations which call for an estimate of the amount of required FPGA resources needed. Estimating is one of the tasks to perform when starting a new FPGA-based project. Migration of an existing design to a new platform that uses a different FPGA family requires knowledge of whether the design is going to fit. Estimating the capacity of an ASIC design in terms of different FPGA resources is needed for porting the ASIC design to an FPGA-based prototyping platform.

Accurately estimating design size is important for several reasons. Firstly, it affects the selection of FPGA family and size. Use of a larger than necessary FPGA device increases the complexity and size of the PCB, power consumption, and the overall system cost. Overestimating the required capacity of an FPGA-based prototyping platform for ASIC designs affects the platform selection, and makes it more expensive. On the other hand, underestimating the design size may lead to significant project delays due to the PCB redesign needed to accommodate a larger FPGA.

There is no well-defined methodology to estimate FPGA design size that can apply to all cases and situations. Estimating the amount of FPGA resources of an existing design targeting a different FPGA family is easier than estimating for a new design, whose size is unknown. Methodologies like using the size of existing functional components and IP cores, and doing extrapolation can be employed for a new design. Using an open-source IP core with similar features can be helpful. Additionally, an analytical approach of estimating the size of individual state machines and pipeline stages can be applied as well, albeit experimental results usually produce far more accurate estimates and instill a higher confidence level.
An FPGA has different resources: LUTs and registers, DSP blocks, IO, clocking, routing, and specialized blocks such as PCI Express interface, Ethernet MAC, and transceivers. It is a common mistake to only estimate logic resources, which are LUTs and registers. For many designs, such as those using embedded processors, it is also important to estimate embedded memory. However, porting an ASIC design to FPGA requires estimating the amount of required clocking resources. Estimating each FPGA resource – logic, memory, clocking, and others – is a different task, which requires a different approach.

To help with these tasks, Xilinx PlanAhead has an option to estimate logic and memory resources without requiring design synthesis and physical implementation. Synopsys Certify can perform area estimates for different design blocks.

Estimating logic resources

The very first task to complete when estimating FPGA logic resources – LUTs and registers – is to decide which metrics to use. Designers have several options to choose from: logic cells, LUTs, registers, or ASIC gates. Performing an estimate using more than one metrics (for example: slices and LUTs) will produce more accurate results. The following table shows the logic capacity of the smallest and largest Xilinx Virtex-6 FPGAs expressed using different metrics.

Table 1: Xilinx Virtex-6 FPGA logic resources

	Virtex-6 LX75T (smallest)	***Virtex-6 LX760 (largest)***
Logic Cells	74,496	758,784
Slices	11,640	118,560
LUTs	46,560	474,240
Registers	93,120	948,480
ASIC gates*	1,117,440	11,381,760

* 1 Logic cell is approximately 15 ASIC gates

Using LUTs

LUT metrics are often used for estimating ASIC design size targeting FPGA. One reason for this is that ASIC designs ported to FPGA are usually LUT-dominated, meaning the ratio of registers to LUTs is heavily biased towards LUTs.

Note that earlier Xilinx FPGA families, such as Virtex-4 and Spartan-3, have 4-input LUTs, whereas Virtex-5, Virtex-6, and Spartan-6 have 6-input LUTs.

Using registers

In some cases, it is convenient to use registers as a metric. They are easier to count when performing size estimates for register-dominated designs, such as the ones that are heavily pipelined or contain large state machines.

Using logic cells

FPGA capacity is often measured in terms of logic cells. A logic cell is an equivalent of a four-input LUT and a register. The ratio between the number of logic cells and six-input LUTs is 1.6 to 1. Logic cells provide a convenient metrics that normalize the differences between FPGAs of the same family. The logic cell metric is a Xilinx specific, and should not be used to compare different FPGA families. The number of logic cells in Xilinx FPGA is reflected in the device name. For example, Virtex-6 LX75T FPGA contains 74,496 logic cells.

Using slices

Many FPGA IP vendors provide the size of the IP cores in slices. A Xilinx Virtex-6 slice contains four LUTs and eight registers. Other FPGA families contain different numbers. For example, a slice in Xilinx Virtex-4 FGPA contains two LUT and two registers. Xilinx Virtex-5 contains four LUTs and four registers. For that reason, using slices to compare logic utilization of different FPGA families will produce inaccurate results.

Slices can provide a more realistic and accurate estimate than LUTs and registers, because they take into account

packing efficiency, as described below.

Slice packing efficiency

Packing efficiency is calculated as the ratio of used LUTs or registers to their total number in all occupied slices.

Xilinx memory controller (MIG) design is used as an example to illustrate slice packing efficiency. The following table shows slice, LUT, and register utilization of the memory controller after floorplanning and physical implementation are done for Virtex-6 LX75T FPGA.

Table 2: Slice packing efficiency of LUTs and registers

	Used in the design	***Total in occupied slices***	***Packing efficiency, %***
Slices	978		
LUTs	2,579	978*4=3,912 (*)	66
Registers	2,343	978*8=7,824 (**)	30

* There are 4 LUTs in one Xilinx Virtex-6 slice

** There are 8 registers in one Xilinx Virtex-6 slice

Note that even though the number of LUTs and registers are balanced in this particular design, the slice packing efficiency is relatively low. Using registers as a metric for the size estimate of this design would be very inaccurate, and require adding a 70% margin.

Using ASIC gates

The capacity of an ASIC design is typically measured in terms of 2-input logic gates. This is a convenient metric to use when estimating the capacity of the FPGA-based prototyping platform, because it allows comparing the capacities of different FPGA families and vendors. However, because this metric is normalized to 2-input gates, it doesn't always provide an accurate measure of the ASIC design size. Xilinx provides the following formula to conversion ASIC gates to Virtex-6 logic cells:

```
1 logic cell = 15 ASIC gates
```

Using this conversion, a 100-million gate ASIC design will require the equivalent of 6.7 million logic cells, or nine Virtex-6 LX760 FPGAs.

Estimating memory resources

In is important to estimate the amount of embedded FPGA memory required by the design. Many designs, such embedded processors, use large amounts of FPGA memory to store data. The following table shows the amount of memory available in the smallest and largest Xilinx Virtex-6 FPGAs.

Table 3: Xilinx Virtex-6 FPGA memory resources

	Virtex-6 LX75T (smallest)	***Virtex-6 LX550T (largest)***
Block RAM	702 KByte	3,240 KByte
Distributed memory	130.6 KByte	1,035 KByte

Also, an additional memory required for debug instrumentation, such as adding ChipScope Integrated Logic Analyzer (ILA), should be taken into consideration.

Estimating clocking resources

An FPGA has limited amount of clocking resources: clock managers, global buffers, clock multiplexors, and dedicated clock routes. Even if the FPGA has sufficient logic and memory resources to accommodate a particular design, that might not be the case with the clocking resources. ASIC designs are known for having complex clock trees and wide use of gated clocks, which can be difficult to map into available FPGA clocking resources. The following table shows different available clocking resources in the smallest and largest Xilinx Virtex-6 FPGAs.

Table 4: Xilinx Virtex-6 FPGA clocking resources

Clocking resource	***Virtex-6 LX75T (smallest)***	***Virtex-6 LX550T (largest)***
Mixed Mode Clock Manager (MMCM)	6	18
Global clock routes	32	32
Regional clock routes	36	108

Estimating routing resources

Various blocks inside the FPGA are connected using routing resources. Designers have little control of how the routing resources are used. FPGA place-and-route tools use them "behind the scenes" to implement design connectivity in the most optimized way. However, it is impractical to quantify the amount of routing resources required for a particular design. Neither do FPGA vendors provide sufficient documentation on performance characteristics, implementation details, and quantity of the routing resources. Some routing performance characteristics can be obtained by analyzing timing reports. In addition, Xilinx FPGA Editor can be used to glean information about the routing structure.

The following figure shows Xilinx FPGA interconnect between slices and global routing as seen in the FPGA Editor.

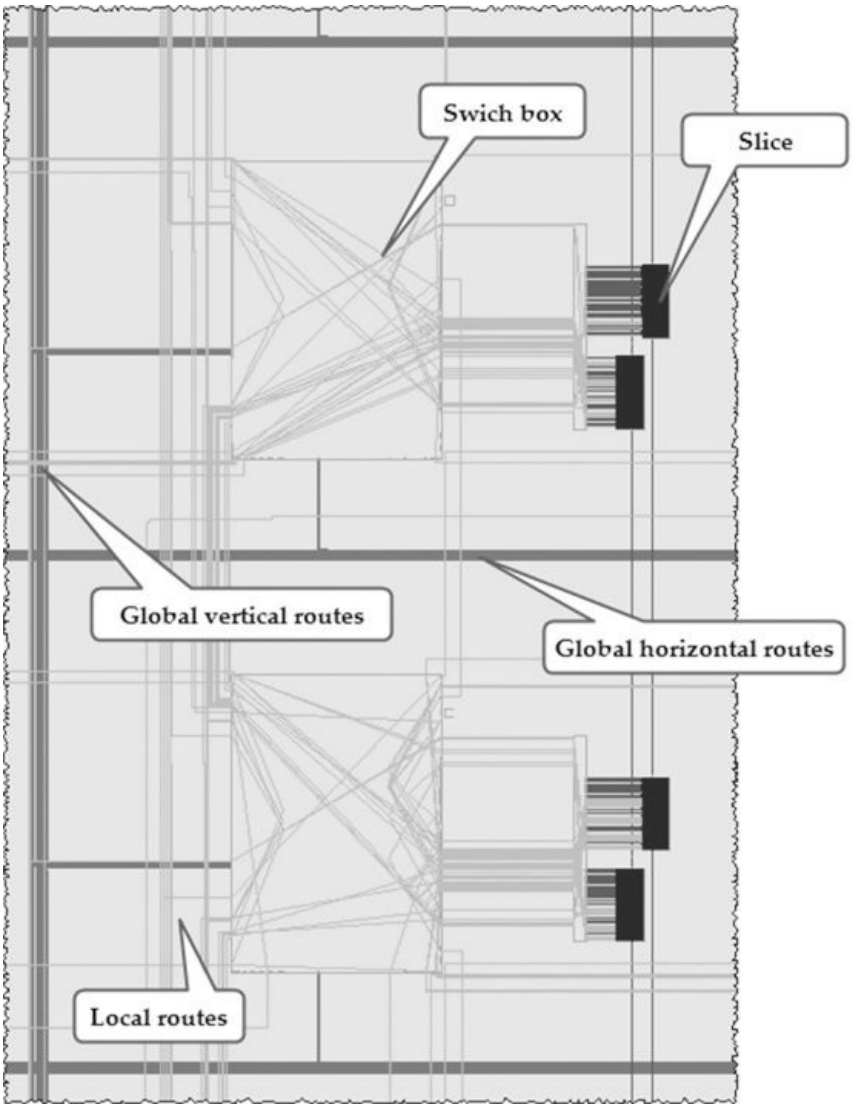

Figure 1: FPGA interconnect

The figure shows slices connected to the adjacent interconnect switch boxes using local routes. Switch boxes connect between each other using both local and global vertical and horizontal routes. A grid-like FPGA routing structure is shown in the following figure.

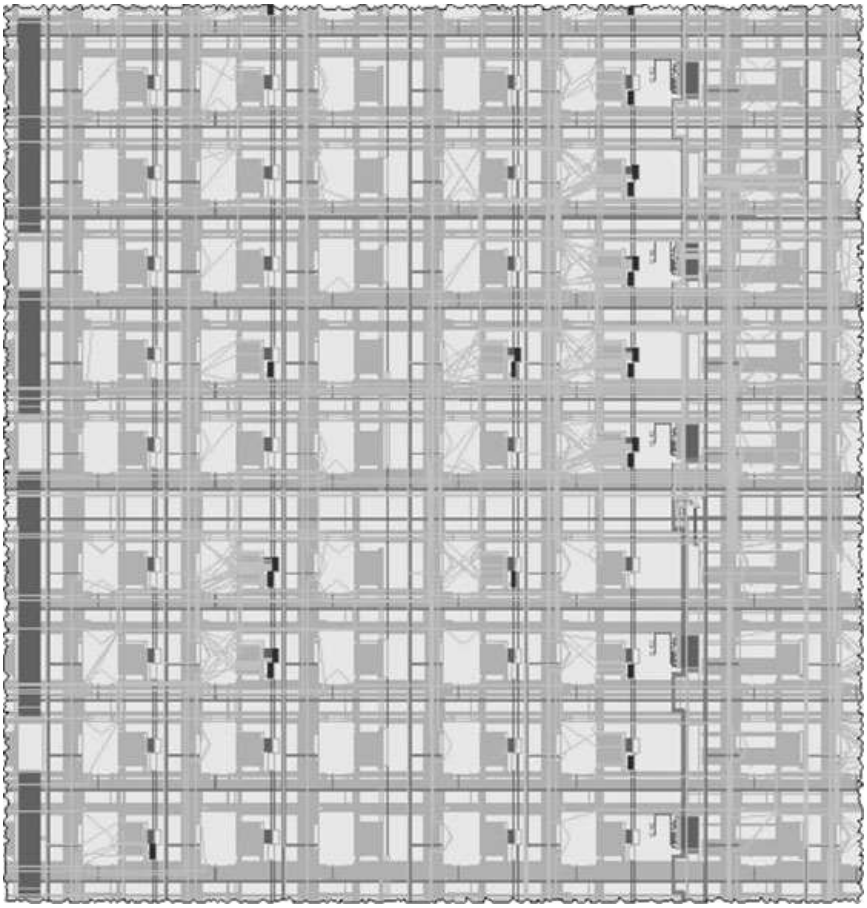

Figure 2: FPGA routing structure

Some FPGA designs make use of a lot of routing resources, which can cause the router tool to fail. A good example of such a design is a Graphics Processing Unit (GPU) ASIC ported to an FPGA-based prototyping platform. It consists of a large number of processors interconnected in a mesh structure.

Solutions to alleviate routing problems include performing logic replication in order to reduce the fanout, and changing the design floorplan. Some of these measures can significantly increase the logic utilization. PlanAhead can be used to visualize routing congestion in different areas of FPGA.

Estimating the overhead

It is important to account for the overhead and leave sufficient margin during the estimate of different FPGA resources. There are different overhead sources outlined below.

Using different synthesis and physical implementation tool options can result in significant variation of the logic and memory utilization. This is discussed in more detail in Tip #79. It is recommended to build the design using different tool options in order to establish a utilization range.

FPGA synthesis tools can produce different utilization results. It is recommended to use the same synthesis tool for both resource estimate and the actual design.

FPGA slice utilization should not exceed 75%. That number can be lower in order to provide enough space to accommodate additional design features.

Adding debug instrumentation, such as ChipScope ILA core, to the design may require a lot of memory resources.

Logic utilization may increase during the timing closure as a result of logic replication and other code changes.

39. Estimating Design Speed

Correctly estimating design speed affects FPGA speed grade selection, consumed power, and the system cost. Overestimating the speed unnecessarily increases power consumption, takes more effort to meet timing requirements, and increases the cost. Conversely, underestimating the speed may result in increased area utilization due to logic changes, or even require a redesign of the PCB to accommodate a higher performing FPGA from a different family.

The term speed grade indicates the maximum frequency and other performance ratings of a particular FPGA for a given family. Xilinx Virtex-6 FPGAs have three speed grades: -1 (slowest), -2, and -3 (fastest).

The following table provides performance characteristics of Xilinx Virtex-6 FPGA for different speed grades. The numbers indicate maximum performance.

Table 1: Performance of Xilinx Virtex-6 FPGA for different speed grades

	-3 (fastest)	*-2*	*-1*
Register clock-to-output delay, ns	0.29	0.33	0.39

Block RAM frequency, MHz	450-600	400-540	325-450
IO double data rate, Gbit/sec	1.4	1.3	1.25
IO single data rate, Mbit/sec	710	710	650
GTX/GTH transceivers, GHz	6.6/11.182	6.6/11.182	5.0/10.32

In practice, it is difficult to achieve maximum performance of logic and BRAM resources, because of the additional routing and logic delays.

There is no a single approach to estimate the design speed. In many designs, the speed requirement is driven by the external communication interface. For example, using PCI Express Gen2, 4-lane configuration will require running the user logic interfacing with the PCI Express at 250MHz. Gigabit Ethernet MAC providing an 8-bit interface to the user logic has to operate at 125MHz. In other designs the speed requirements are more flexible. FPGA implementations of a video processing algorithm can double the datapath width to lower the design speed. There is usually no hard requirement on the performance of an ASIC design running on an FPGA-based prototyping platform. In most cases the prototyped design runs at a fraction of the speed of the ASIC. It is beneficial to strive to improve the performance because it will decrease the validation time.

Design area and performance goals are usually interrelated. Often, measures taken to improve design area lead to decreased performance. This factor has to be taken into account when performing design speed estimates. Also, using different synthesis and physical implementation tool options can result in significant variation of the design performance. It is recommended to build the design using different tool options in order to establish an achievable performance range. It is up to the designer to decide what margin is sufficient.

It is important not to rely on the design performance results provided by the synthesis tool. Those are estimates, which can be very different from those done by a timing analyzer after the place-and-route.

The speed of an ASIC design ported to a multi-FPGA prototyping platform is dictated not only by the critical paths within an FPGA, but also by the communication speed between the FPGAs. The inter-FPGA communication speed can often become the performance bottleneck. Because FPGAs have a limited amount of user IOs, the interfaces between FPGAs are frequently implemented by multiplexing the signals, and using high-speed serializer/deserializer (SerDes) transceivers. The multiplexing ratio dictates the overall frequency.

The following example examines a simple case of determining the system performance. An ASIC design consists of a CPU connected to a peripheral controller via a 64-bit bus. The design is partitioned into two FPGAs using a SerDes.

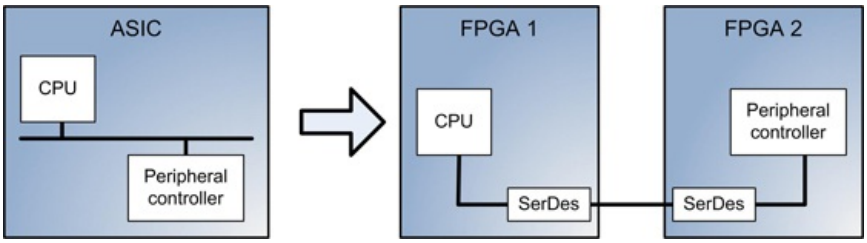

Figure 1: Partitioning an ASIC design into two FPGAs

Scenario 1

The SerDes is implemented using four Xilinx GTX transceivers. The transceivers are full-duplex, and running at speed of 5Gbit/sec. The transceivers use 8b/10b error correction scheme for the data, which adds 20% overhead.

The system performance is calculated as follow:

```
f = 5 Gbit/sec * 4 channels * 0.8 / 64 bit = 256 Mhz
```

Scenario 2

The SerDes is implemented using 16 differential pairs of regular FPGA IO pins. The IOs are double data rate (DDR), and running at speed of 320MHz. 8b/10b error correction scheme is used for the data.

The system performance is calculated as follows:

```
f = 320 MHz * 16 data bits * 2 (DDR) * 0.8 / 64 bit = 128 Mhz
```

This is, in fact, considered a very high system performance for the prototyping platform. Real designs require a lot more signals to be transferred between FPGAs. Also, prototyping platforms are more complex with more FPGAs and communication channels between them. These factors significantly increase the multiplexing ratio, and lower the maximum performance. ASIC designs running on FPGA-based prototyping platforms typically achieve speeds of 10-100MHz.

40. Estimating FPGA Power Consumption

Xilinx provides three main tools to perform FPGA power consumption estimate and analysis: PlanAhead Power Estimator, XPower Estimator (XPE), and XPower Analyzer (XPA). This Tip provides a brief overview of those tools. Several power optimization techniques are discussed in Tip #81.

PlanAhead Power Estimator

PlanAhead provides a tool to perform early power estimation based on the design resources. The power estimator uses RTL and signal toggle rates as input, and produces total and quiescent power estimates broken down by clocks, IOs, logic, and BRAM. It also provides Credibility Level metrics that can be High, Medium, or Low.

Figure 1: PlanAhead Power Estimator results

XPower Estimator Spreadsheet

XPower Estimator Spreadsheet (XPE) is an easy-to-use Microsoft Excel spreadsheet that allows experimentation with different design options to perform power estimates. Design parameters, such as clocks, IOs, logic and memory resources, and toggle rates can be either entered directly or imported from the existing MAP report. The following figure shows the XPE summary page, which provides the estimate of the total on-chip power broken down by dynamic and quiescent components, and junction temperature.

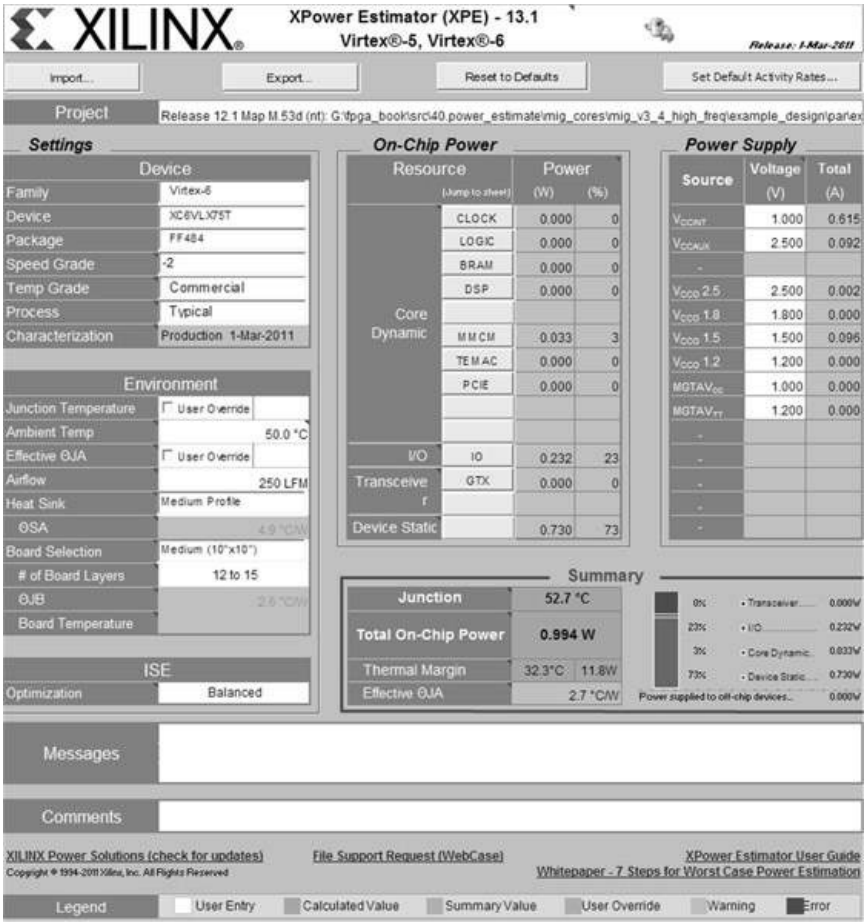

Figure 2: XPE summary page example

XPower Analyzer

Xilinx XPower Analyzer (XPA) is a utility that estimates power after the design implementation. XPA utilizes more design data, such as precise IO characteristics and toggle rates, and therefore is more accurate than XPE spreadsheet and PlanAhead Power Estimator.

The following is a command-line example of using XPower:

```
$ xpwr example.ncd example.pcf -v -o example.pwr -wx example.xpa
```

The required parameter is the NCD file produced by MAP or PAR. Optional files are PCF, produced by MAP, VCD, which includes signal activity rates, or XPA – power report settings file in XML format.

XPower produces power report in text format. It contains several sections: project settings, power consumption and thermal summaries, detailed power information broken down by clocks, logic, IO pins, and signals. It also provides a summary of dynamic and quiescent power, and power supply currents broken down by supply source.

Example

To showcase the XPower Analyzer and PlanAhead Power Estimator capabilities, Xilinx memory controller core (MIG) has been used as an example design. The core provides a balanced mix of features to demonstrate different components of the FPGA power consumption: high IO toggle rates, substantial logic utilization and the number of used IOs, and fast clock.

The core was built for Virtex-6 LX75T- FF484 FPGA with -2 speed grade. The following table provides main characteristics of the design relevant to the power consumption.

Table 1: Design characteristics

Frequency, MHz	***Memory technology***	***Slice utilization***	***Number of IOs***
533	DDR3	1495	44

The following table shows the activity rates used for the power estimate.

The activity rates are default. Users can provide more accurate activity rate information by performing functional simulation and passing the resulting NCD file as a parameter to the XPower tool.

Table 2: Design activity rates

Default Activity Rates (%)	
FF Toggle Rate	12.5
IO Toggle Rate	12.5

IO Enable Rate	100
BRAM Write Rate	50.0
BRAM Enable Rate	25.0
DSP Toggle Rate	12.5

The following table contains the on-chip power summary produced by XPA and broken down by individual resources.

Table 3: XPA on-chip power summary by resources

Resource	*Power (mW)*	*Used*	*Available*
Clocks	99	6	-
Logic	20.2	2579	46560
Signals	27.5	3788	-
IOs	421	40	240
MMCM	115	1	6
Quiescent	729	-	-
Total	1,412	-	-

It is interesting to note that the IO's power is a significant contributor to the overall power consumption. That is the consequence of using internal IO terminations by the memory controller design. Another interesting result is almost even quiescent and dynamic power consumption. Dynamic power in the table above is the sum of clocks, logic, signals, IOs, and MMCM components.

The following table contains the power summary of the same design produced by the PlanAhead Power Estimator.

Table 4: PlanAhead Power Estimator summary by resources

Resource	*Power (mW)*	*Credibility Level*
Clocks	279	High
Clock Manager	134	-
Logic	57	High
Block RAM	60	-
IOs	1,339	Medium
Quiescent	760	-
Total	2,629	High

PlanAhead Power Estimator and XPA results are close for quiescent power estimates. However, there is a significant difference between IO and total power estimates. One reason for that discrepancy is that physical constraints, such as IO characteristics, are taken into consideration in the XPA estimates.

Power usage numbers are very specific to FPGA device and design, and might differ by as much as an order of magnitude between smallest designs running on power-efficient Spartan-6 and the largest Virtex-6 FPGAs. The goal of this example is to show the capabilities of the tool, and illustrate what information it provides, rather than serve as a reference.

The example design and complete power reports used in this Tip are available on the accompanying website.

Resources

[1] Xilinx Power Solutions
http://xilinx.com/power

This is the most comprehensive website related to Xilinx FPGA power estimate and analysis.

[2] Seven Steps to an Accurate Power Estimation using XPE, Xilinx White Paper WP353
http://www.xilinx.com/support/documentation/white_papers/wp353.pdf

[3] XPower User Guide, Xilinx User Guide UG440
http://www.xilinx.com/support/documentation/user_guides/ug440.pdf

41. Pin Assignment

FPGA pin assignment is a collaborative process of PCB and FPGA logic design teams to assign all top-level ports of a design to FPGA IOs. At the end of the process PCB components are placed and routed, and the board passes SPICE and other simulations. FPGA design can be synthesized and implemented, and it passes DRC. Pin assignment completion is achieved as a result of series of trade-offs between PCB and FPGA logic design teams.

The following figure illustrates the pin assignment process.

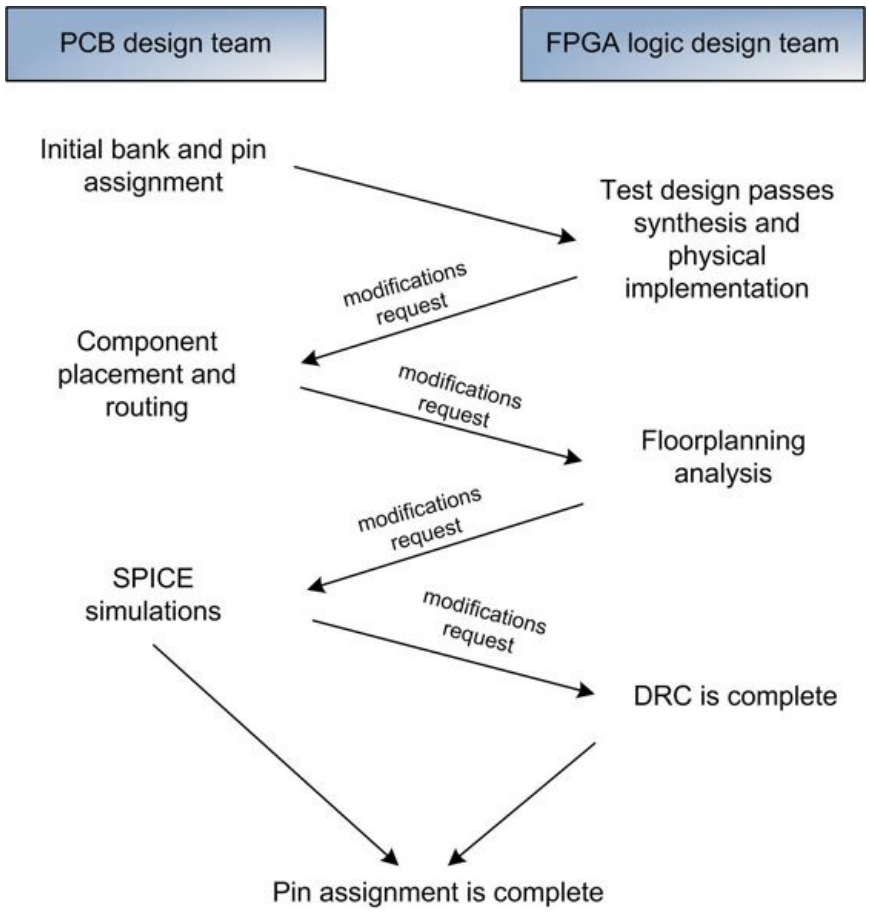

Figure 1: Pin assignment process

PCB design team usually initiates the pin assignment process. There are more pinout-related constraints associated with the board design comparing to FPGA logic design: board size, FPGA banking rules, power and clocking scheme, decoupling capacitors, terminations, signal integrity considerations, and many others. FPGA design team faces different sets of constraints: connecting clocks to the clock-capable IOs, assignment of pins in such a way that the design can meet timing, among others.

After electronic component selection and their initial placement, PCB designers perform initial pin assignment and pass it to the FPGA logic design team.

FPGA design team uses the pinout for synthesis and physical implementation. Typically, a complete RTL is not available that early in the design cycle. One approach is to create a test design that represents the final one in terms of pin characteristics: all pins have the same direction, are single-ended or differential, and include the same clocks. The test design should also include special signals, such as high-speed SerDes transceiver inputs/outputs. All IP cores with complex IO scheme, such as memory controller, PCI Express interface, Ethernet MAC, should be part of the test design.

Successfully passing physical implementation will prove feasibility of the pin assignment, and uncover a lot of other problems.

Unfortunately it’s a common mistake not to pay proper attention to this step, or even skipping it completely. Tight project deadlines, false sense of confidence, and other factors cause even the most experienced designers to ignore this step and move on. The consequences of this are extra cost due to board redesign, and lost time.

FPGA design team then passes all pin change requests and other recommendations back to the PCB design team.

PCB design team incorporates the changes, and continues with the component place and route, and board-level SPICE simulations. FPGA design team performs floorplanning analysis, and Design Rules Check (DRC). Floorplanning analysis is especially important for the high-speed design, where timing closure is difficult to achieve. Checks might include making sure that all the IOs belonging to the same bus or module are grouped together.

After exchanging several pinout modification requests, both PCB and FPGA design teams converge to an acceptable pin assignment and are ready to sign off.

The main challenge of the pin assignment process is that it occurs very early in the product design process. At that time, FPGA logic design is in the very early stages, or might not have even started yet. PCB designers have yet to finalize all the electronic component selection, board size, and the number of board layers.

Another challenge that can affect pin assignment is the need to define clocking scheme and speed of all interfaces. That leaves designers with fewer options for future changes. For example, the decision of running a bus at Single Data Rate (SDR), instead of Double Data Rate (DDR) because of the FPGA IO speed limitation, will require twice as many pins.

It’s a good idea not to use all the available pins in the FPGA, but leave some room for future product enhancements or potential changes of the peripheral components. One good example is the USB 2.0 PHY interface. USB 2.0 PHY chips have two interfaces with very different pin count: USB Transceiver Macrocell Interface (UTMI) and USB Low Pin Interface (ULPI). If the existing PHY chip with ULPI interface becomes obsolete, or requires replacement for any other reason to UTMI chip, that will create a problem.

Xilinx offers PlanAhead tool that helps designers improve pin assignment process. The following figure shows device and package views of a Spartan-6 LX25T FPGA.

The left side shows the device, or die, view, as seen by the FPGA logic designers. The right size is a package view as seen by the PCB designers.

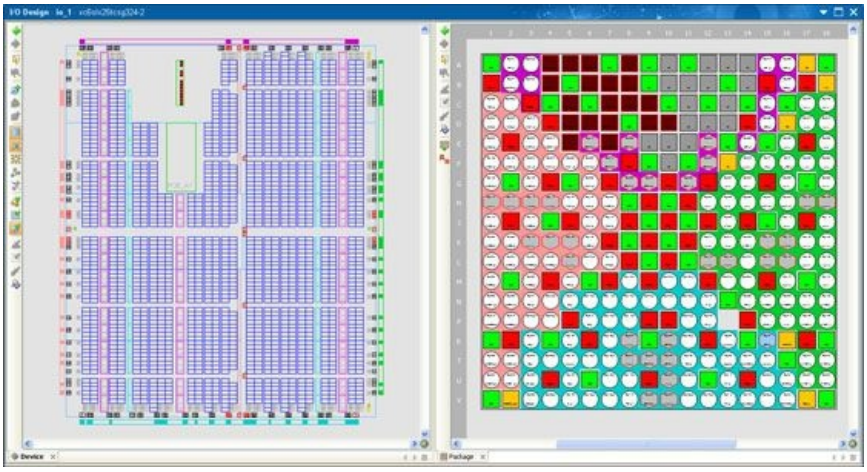

Figure 2: PlanAhead device and package views

Design Rules Check (DRC)

Design Rules Check is an essential step during the pin assignment. Pin assignment cannot be completed before a design passes all the design rules.

The following are some of the design rules that need to be checked:

IO port and clock logic rules: clocks are connected to the clock-capable IO, differential signals use correct polarity.

FPGA banking rules: IO standards legality and consistency, special power pins are correctly connected.

Simultaneously Switching Noise (SSN) rules: SSN is defined as the voltage fluctuation caused by the simultaneous switching of groups of IO signals. SSN causes dynamic voltage drop on power supply lines, which can affect performance of the output drivers and can lead to excessive jitter on critically timed signals. Xilinx PlanAhead includes SSN predictor tool to calculate estimated noise margin.

Performing manual DRC is impractical due to the complexity of design rules. Xilinx PlanAhead can perform DRC automatically.

42. Thermal Analysis

Performing an accurate thermal analysis is important for several reasons. It affects the mechanical design of a printed circuit board (PCB) and the enclosure. The analysis results dictate whether it is required to use heat sinks, fans, and other solutions to improve heat dissipation. In addition, using fans complicates mechanical design because they require taking measures to reduce the noise.

The goal is to design a system such that the junction temperature is kept within safe limits. Not doing so will dramatically decrease FPGA reliability, longevity, and in some cases, functionality. Thermal analysis has to cover all operating conditions of the FPGA: from a controlled lab environment with near constant temperature, to a space or other extreme hot/cold environment. Power analysis, discussed in Tip #40, is tightly intertwined with the thermal analysis.

The following figure shows mechanical components that affect thermal performance of the FPGA: top and bottom heat sinks, thermal pad and vias. Other factors that affect thermal performance are size and quality of direct thermal attachment pad, usage of thermal grease, amount of air flow, PCB area, and the number of copper layers.

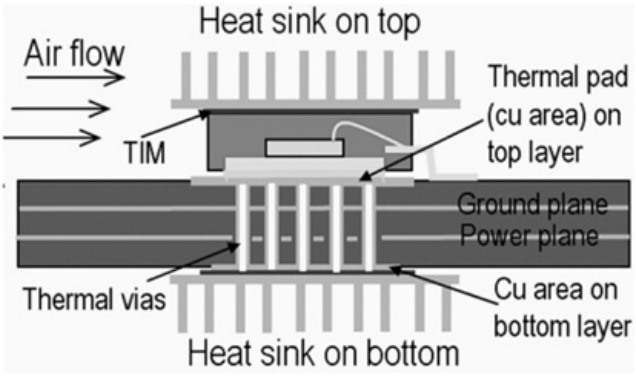

Figure 1: Thermal management (source: National Semiconductors Application Note 2026)

Modeling for thermal analysis

The following equation defines the relationship between ambient temperature, junction temperature, power

dissipation, and junction-ambient thermal resistance:

$T_J = (\Theta_{JA} \cdot P_D) + T_A$

T_J is a junction temperature. It is defined as the highest temperature of the actual semiconductor in an electronic device, and is usually specified in the device datasheet. For Virtex-6 devices, the maximum junction temperature T_J is +125°C. The nominal junction temperature is +85°C, and used for static timing analysis and power consumption calculations. DC and Switching Characteristics datasheet defines operating range, minimum, maximum, and nominal junction temperatures for different Xilinx FPGA families.

T_A is an ambient temperature and P_D is a power dissipation. Θ_{JA} is a junction-ambient thermal resistance, representing the ability of the material to conduct heat, and measured in units of °C/W.

The following figure shows a model frequently used for thermal analysis.

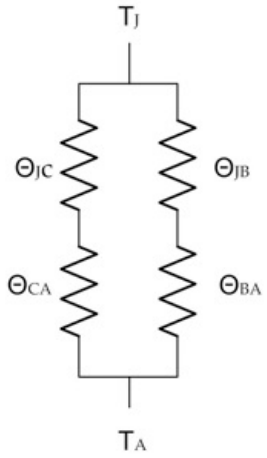

Figure 2: Thermal model

Θ_{JC} is a junction-case thermal resistance specific for a device. For Virtex-6 FPGA packages Θ_{JC} is typically less than 0.20°C/W.

Θ_{CA} is a case-ambient thermal resistance. It is calculated by adding all thermal resistance components on top of the device case: heatsinks, fans, thermal grease and other materials, and the airflow contribution.

There are two heat dissipation paths: junction-case-ambient, and junction-PCB-ambient. Because the two paths are in parallel, the overall Θ_{JA} can be expressed as:

$\Theta_{JA} = (\Theta_{JCA} \cdot \Theta_{JBA})/(\Theta_{JCA} + \Theta_{JBA})\Theta_{JCA}$ component is the thermal resistance through the package top surface to ambient, and equals to

$\Theta_{JCA} = \Theta_{JC} + \Theta_{CA}$.

Θ_{JBA} component is the thermal resistance through the PCB to ambient, and equals to

$\Theta_{JBA} = \Theta_{JB} + \Theta_{BA}$.

The model takes into account the PCB thermal resistance. For a small PCB with low number of layers, the model is reduced to Θ_{JCA}, because the board thermal resistance component Θ_{JBA} is large, and doesn't have significant contribution to the overall thermal resistance. The larger the PCB and the more layers, the smaller the Θ_{JBA} component is. The following table illustrates the impact of the PCB size and the number of layers on the thermal performance.

Table 1: Impact of the PCB on Θ_{JA} (source: Xilinx User Guide UG365)

Xilinx 35 x 35mm FF1148		θ_{JA} (°C/W) for Different Board Sizes		
		4 x 4 in	10 x 10 in	20 x 20 in
Layer Count of Mounted Board	4	9.1(1)	8.3	–
	8	8.0	5.5	4.9
	12	7.5	4.7	4.4
	16	7.2	4.5	4.2
	24	–	4.3	4.0

It is important to select the correct model for a thermal analysis. For example, not taking a PCB component into consideration may result in a very inaccurate estimate, as shown in the above table.

Thermal Management Options

Bare package

A bare package or a passive heatsink is recommended for low-end FPGAs with small packages, 1-6W power dissipation, and moderate air flow.

Passive heatsink

The purpose of a heatsink is to conduct heat away from a device. A heatsink is made of high thermal conductivity material, usually aluminum or copper. Increasing the surface area of a heatsink, for example fins, facilitates heat removal. An interface between a heatsink and a device, for example using thermal grease, applying strong contact pressure, and surface characteristics, are important for good thermal transfer.

Using a passive heatsink is recommended for mid-range FPGAs with 4-10W power dissipation in a system with good air flow.

Active heatsink

Using the combination of a heatsink and a fan is recommended for high-end FPGAs with large packages and 8-25W power dissipation.

Example

Find the maximum allowed power dissipation of a device given junction and ambient temperatures:

T_J = +85°C
T_A = +55°C
The device is Virtex-6 with Θ_{JC} = 0.20°C/W.
Thermal resistance of a heatsink and fan is Θ_{CA} = 1.80°C/W.
Θ_{JB} = 0.40°C/W, and Θ_{BA} = 2.60°C/W.

To find the maximum allowed power dissipation a PD requires solving the following equations:
$T_J = (\Theta_{JA} \cdot P_D) + T_A$
$\Theta_{JA} = (\Theta_{JCA} \cdot \Theta_{JBA})/(\Theta_{JCA} + \Theta_{JBA})$
$\Theta_{JCA} = \Theta_{JC} + \Theta_{CA} = 0.2 + 1.8 = 2.0$ °C/W
$\Theta_{JBA} = \Theta_{JB} + \Theta_{BA} = 0.4 + 2.6 = 3.0$ °C/W
$\Theta_{JA} = (2.0 \cdot 3.0)/(2.0 + 3.0) = 1.2$ °C/W
$P_D = (T_J - T_A)/\Theta_{JA} = (85 - 55)/1.2 = 25$ W
Therefore, for a given junction and ambient temperatures, and thermal resistances, the power dissipation of the FPGA should not exceed 25 Watt.

Resources

[1] Xilinx XPower Estimator (XPE) tool for performing thermal and power analysis
http://www.xilinx.com/products/design_resources/power_central

[2] Considerations for heatsink selection, Xilinx White Paper WP258
www.xilinx.com/support/documentation/white_papers/wp258.pdf

[3] Virtex-6 Packaging and Pinout Specifications, Xilinx User Guide UG365
www.xilinx.com/support/documentation/user_guides/ug365.pdf

[4] National Semiconductor Application Note 2026
www.national.com/an/AN/AN-2026.pdf

43. FPGA Cost Estimate

In many designs FPGA is the most expensive electronic component. High and medium capacity FPGAs can cost from several hundred to several thousand dollars. Therefore, minimizing FPGA cost is important and can significantly affect the total cost of the product.

Accurate FPGA cost estimation is not a simple task, and it depends on many factors. Technical characteristics that affect the FPGA cost are:

- FPGA family: lower cost Spartan-3, Spartan-6 to higher performance Virtex-5, Virtex-6
- Capacity: from 3,840 logic cells (smallest Spartan-6) to 758,784 logic cells (largest Virtex-6)
- Speed grade: -L1 (low power), -2, -3, -4 (fastest)

- Temperature range: Commercial (Tj = 0°C to +85°C), Industrial (Tj = -40°C to +100°C)
- Package type: smallest TQG144 to largest FF1924

Purchase quantity also has significant impact on the cost. Smaller quantities cost more per unit. The final cost also depends on the negotiation results with the FPGA distributor.

Also, as one FPGA family is getting older and needs replacement, the older FPGAs are getting more expensive, which makes the newer FPGA family more attractive in terms of cost. FPGA prices skyrocket as the family approaches the end of production date. Those are general pricing principles applicable to the entire electronic chip industry.

To make FPGA comparison data more meaningful, designers frequently use additional metrics, such as cost per logic cell and cost per user IO.

The goal of the FPGA cost estimate process is to take into consideration the above factors to find a sweet spot that meets the functional requirements of the engineering team as well as the cost requirements of the marketing and sales team.

The following tables show sample Xilinx FPGA prices.

Table 1: Lowest capacity Xilinx Virtex-6 LX75T prices (source: FindChips.com, February 2011)

Number user IOs	*Capacity, logic cells*	*Speed grade*	*Cost, $*	*Cost per logic cell, cents*	*Cost per user IO,$*
240	74,496	-1	625	0.8	2.6
240	74,496	-2	771	1.0	3.2
240	74,496	-3	1093	1.5	4.6
360	74,496	-1	718	1.0	2.0
360	74,496	-2	899	1.2	2.5
360	74,496	-3	1258	1.7	3.5

Table 2: Highest capacity Xilinx Virtex-6 LX760 prices (source: FindChips.com, February 2011)

Number user IOs	*Capacity, logic cells*	*Speed grade*	*Cost, $*	*Cost per logic cell, cents*	*Cost per user IO,$*
1200	758,784	-1	15,622	2.1	13.0
1200	758,784	-L1	19,528	2.6	16.3
1200	758,784	-2	20,714	2.7	17.3

Table 3: Lowest capacity Xilinx Spartan-6 LX4 prices (source: FindChips.com, February 2011)

Number user IOs	*Capacity, logic cells*	*Speed grade*	*Cost, $*	*Cost per logic cell, cents*	*Cost per user IO,$*
106	3,840	-L1	12	0.313	0.113
106	3,840	-2	10.4	0.271	0.098
106	3,840	-3	11.4	0.297	0.108
132	3,840	-L1	13	0.339	0.098
132	3,840	-2	11.4	0.297	0.086
132	3,840	-3	12.4	0.323	0.094

The prices are provided for illustration purpose only. The final price depends on the quantity and distributor, and is subject to significant change in time. However, a simple analysis of the price relationships between different families, capacities, speed grades, and user IOs yields some important results.

Price increase with capacity is non-linear, as indicated in the cost per logic cell ratio. It increases from 0.8 cents/logic cell for the smallest, Virtex-6 LX75, to 2.7 cents/logic cell for the largest, Virtex-6 LX760.

There is a non-linear cost increase for an FPGA with the same capacity and the number of user IOs and a faster speed grade. The exception is a low power -L1 speed grade, which is actually more expensive.

Cost per user IO decreases for an FPGA with the same capacity and speed grade.

The cost of a high-capacity FPGA is higher than the total cost of multiple smaller FPGAs with equivalent capacity. For example, it requires 10 Virtex-6 LX75 to reach the equivalent capacity of a single Virtex-6 LX760. However, the cost of 10 Virtex-6 LX75 is $6,250, less than half as much as a single Virtex-6 LX760, which costs $15,622. The rule of thumb is that low- and medium-capacity FPGAs are more price efficient than high-capacity ones using cost per logic cell and cost per user IO metrics.

44. GPGPU vs. FPGA

When starting a new design, system architects often ask themselves if they have to use an FPGA, or if there are alternatives, such as a General Purpose Graphics Processing Unit (General Purpose GPU, or GPGPU).

GPGPU has been gaining momentum and has been used in a variety of applications from solving partial differential equations in the finance industry, to accelerating Matlab simulations and high-resolution medical imaging. nVidia CUDA Zone, a web portal for GPGPU developers on the nVidia CUDA GPGPU platform, lists over 1,500 different applications. GPGPU-based applications became so widespread that Amazon Web Services (AWS) started supporting GPGPU as a cloud computing resource.

There is no clear answer as to when GPGPU can replace an FPGA, and vice versa. That question is application dependent. Typically, GPGPU is used to accelerate certain classes of compute-intensive software applications. The only interface current GPU cards provide for the Host connectivity is PCI Express. GPU has a limited amount of on-chip and on-board memory. FPGA, on the other hand, can be used in a broader range of applications, not just for software acceleration. FPGAs have customizable IOs, so it can interface with virtually any device. The disadvantage of FPGAs is cost.

System architects might want to consider criteria such as interface, speed, latency, and cost in order to make an informed decision of whether to use FPGA or GPGPU in their next design.

Interface

All modern GPU cards use PCI Express protocol as the interface to the Host. At the time of writing of this book, the fastest configuration is PCI Express Gen2 x16, which can deliver up to 6.4 GB/sec of useful payload. Even if it's sufficient for all video applications and most GPGPU acceleration applications, it still might not be enough for some. One example is multi-link network processing applications that are traditionally implemented using high-end FGPAs.

Another FPGA advantage is that it has highly configurable and flexible interfaces —hundreds of general-purpose IOs, high-speed embedded SerDes modules —that can be customized for different needs.

Throughput

Throughput, or speed, is highly dependent on the application. The main strength of both FPGA and GPGPU is their high level of parallelism. GPGPU excels in Single Instruction Multiple Data (SIMD) applications, such as DSP, image, and video processing. The performance degrades in applications that have more complex control flow and require larger memory, such as compilers. FPGA achieves peak performance if an application doesn't require external memory access, and uses only logic resources and embedded memory.

Latency

FPGA designs can have very low and consistent latency and are well-suited for real-time systems, which require a response time of 1ms or less. Some examples are TCP/IP checksum offload on high speed links (1GbE, 10GbE, or higher), fast encryption/decryption, or hard real-time weapons systems.

GPGPU can have much higher latency due to the architectural limitations. There are several factors that contribute to relatively high and inconsistent GPGPU latency. One is PCI Express interface between a GPU card and the Host, which is used in all modern GPU cards. Another is memory access latency both on the Host side and on the GPU card.

Cost

It is a common misconception that FPGA logic resources are much more expensive than those of GPGPUs. Indeed, high-end, high-capacity FPGAs are very expensive and can cost several thousand dollars (see Tip #43 for an FPGA cost analysis). However, there are several factors that determine the overall cost of the system. Application development time is a significant cost factor that is often overlooked. Developing an efficient application for a modern GPU is a complex task and requires specialized skills and experience. Also, it's not simple to find an experienced developer in both FPGA and GPGPU who can bridge the gap between the two.

It is difficult to perform a meaningful cost and performance comparison between FPGA and GPGPU in general terms. The best and

the most accurate way to compare the cost and performance is by prototyping. The basis can be an existing software application that runs on a GPGPU, and needs to be ported to FPGA. Tips #38 to #43 discuss different aspects of FPGA selection and process and can be helpful here. Conversely, an existing FPGA design can be ported to a GPGPU.

Resources

[1] GPGPU portal
http://gpgpu.org/about

[2] nVidia CUDA
http://www.nvidia.com/object/cuda_home_new.html

45. ASIC to FPGA Migration Tasks

The most common reason for migrating an ASIC design to FPGA is the prototyping of the ASIC design on an FPGA-based platform during its development. This is the focus of the discussion in Tips #45-55. Those Tips are the most useful for system engineers and architects who evaluate different commercial emulation solutions and custom-built alternatives, for logic designers tasked with porting an ASIC design to an FPGA-based prototyping platform, and for verification engineers who need to adapt existing testbenches to a new environment.

Implementing an FPGA-based prototype of an ASIC design requires overcoming a variety of development challenges, such as creating an optimal partition of the original design, meeting speed and capacity goals, porting RTL without compromising functionality, and making sure that ported design is functionally equivalent to the original.

There is no existing methodology, sometimes called Design-for-Prototyping, to migrate from an ASIC to FPGA that is applicable to every design. However, the overall flow and migration tasks remain the same: capacity and speed estimate, selection of emulation or prototyping platform, design partition, RTL modifications, synthesis and physical implementation, and, finally, verification of the ported design. This process is shown in the following figure.

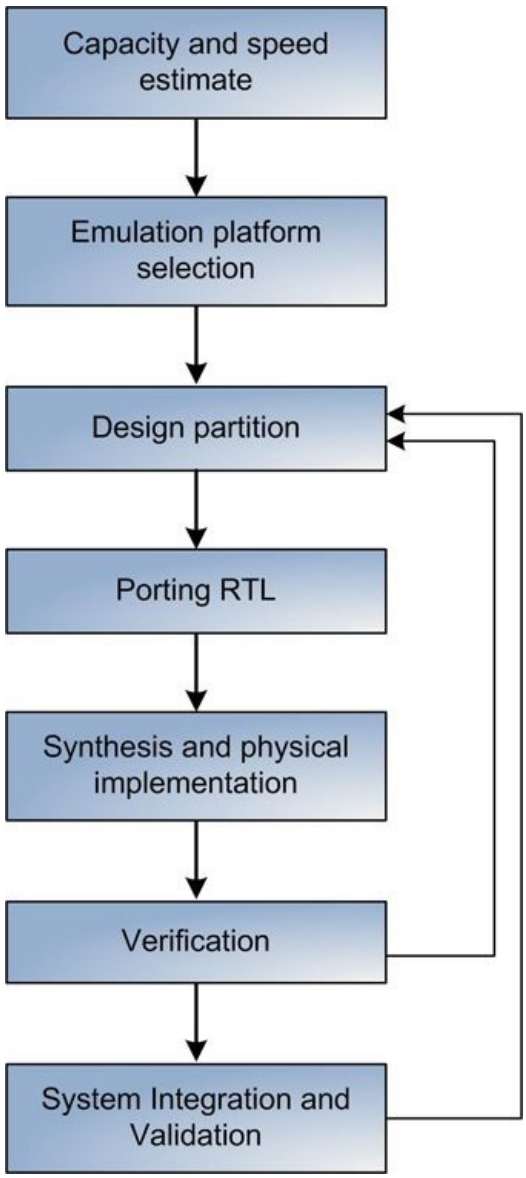

Figure 1: ASIC to FPGA migration tasks

The overall objective of the migration process is to modify the original RTL as little as possible so as to avoid introducing new bugs or increasing coverage holes.

Capacity estimate

Accurate capacity estimate of an ASIC design is important for several reasons. It directly affects selection of the emulation or prototyping platform, the architecture of the partitioned design, and cost. Underestimating the capacity will result in an inability to emulate the entire design. Incurring unnecessary costs is the result of overestimating capacity.
Different approaches to performing ASIC design capacity estimates are discussed in Tip #38.

Speed estimate

An accurate speed estimate affects the usage model of the emulation or prototyping platform. In most cases it is unrealistic to expect the ported FPGA-based design to run at the same speed as the ASIC design. FPGAs are inherently slower than ASICs, and there is an additional communication delay between FPGAs that constitute the emulation or prototyping platform. If the platform runs less than an order of magnitude slower than the original design—for example, at half or quarter speed—then most of the verification tasks can be performed on such a platform. It the speed is ten to one hundred times slower, some of the verification tasks will not be performed.

A good example is a system on chip (SoC) with multi-core processors. It typically takes up to five minutes to boot an embedded operating system on such an SoC. If the emulation platform speed is ten times slower, the boot time increases tenfold to an hour. Such a system can still be used for some of the verification tasks that require a running operating system. However, if the emulation speed is one hundred times slower and it takes half a day to boot an operating system, such a platform becomes completely unusable for many of the embedded software verification tasks.

Speed estimate is discussed in Tip #39.

Selecting emulation or prototyping platform

Selecting emulation or prototyping platforms can be a long and complex process, which includes evaluating existing platforms, making sure that all the requirements and goals such as emulation capacity, speed, cost, coverage, and tool support are met.
Platform selection process is discussed in Tip #47.

Partitioning of an ASIC design into multiple FPGAs

A typical ASIC design is too large to fit into a single FPGA and needs to be partitioned to run on a multi-FPGA platform. Design partition is the process of assigning parts of the ASIC design to individual FPGAs. Partition process might be a complex optimization task that includes meeting a platform's cost, speed, and capacity goals, while ensuring proper connectivity between FPGAs and reasonable complexity of the partition process.
Some of the design partition approaches are discussed in Tip #48.

Porting RTL

Developers need to overcome numerous challenges when porting an ASIC design RTL to FPGA. Clock trees of an ASIC design need to be ported to specific clocking resources of individual FPGAs and distributed across different FPGAs on an emulation platform.

ASIC design RTL might contain several design elements, such as transistors, transmission gates, bidirectional signals, and non-digital circuits that have no direct equivalent in FPGA architecture and need to be modeled.

Most of ASIC designs include intellectual property (IP) cores. The IP cores can range from relatively simple functional blocks from the Synopsys DesignWare library (which can be easily replaced with FPGA equivalents) to complex memory controllers, bus-connection subsystems such as PCI Express or USB, and CPU cores. Xilinx and other FPGA vendors provide many, but certainly not all, possible replaceable IP cores.

Tips #49-54 discuss porting of clocks, latches, memories, tri-state logic, combinatorial and non-synthesizable circuits in more detail.

Synthesis and physical implementation

The next step after porting the RTL is to perform design synthesis and physical implementation. The tasks included in this step are resolving ASIC and FPGA synthesis tool differences; specifying timing, area, and IO design constraints; choosing proper synthesis and physical implementation tool options; and, finally, performing timing closure of the design.

Although modern FPGA tools and design flows resemble the ones used in ASIC design, there are still several fundamental differences that developers need to address. ASIC synthesis tools, such as Synopsys Design Compiler, support a wider range of Verilog constructs and are less restrictive in general, compared with the FPGA synthesis tools. FPGA and ASIC tools might support different synthesis directives and options related to design optimization. ASIC tools are capable of handling much larger and more complex designs with denser routing and more logic levels. FPGA physical implementation tools might fail to build some designs due to placement or routing failure. In those cases, designers might need to make additional RTL changes, or repartition the design.

Ported design verification

Ported design verification is the remaining task in the migration process. Its goal is to make sure that the ported design is functionally equivalent to the original one and still complies with the specification. Verification tasks include software functional simulation, performing emulation or prototyping on a hardware platform, and equivalence checking. This step is described in Tip #55.

System integration and validation

Silicon validation and system integration are tasks performed after the chip is taped out. The main focus is on-board and system-level tasks to ensure that the chip is functioning correctly and can properly integrate into the system. The chip is plugged into a specially designed board. Next, a tester applies stimulus to chip's inputs and captures the outputs. Then, the outputs are compared to the expected values, and the pass/fail result is produced. The process is typically automated and performed concurrently on multiple boards operating at different conditions. There are many ways to generate the stimulus and compare the expected values, such as predefined sets of test vectors, self-generated pseudo-random patterns using Multiple Input Shift Register (MISR) signatures, and others. If one of the test sequences fails, the same sequence is run on the emulation platform in order to determine the root cause of the problem.

46. Differences Between ASIC and FPGA Designs

ASIC designs are different from the FPGA counterparts in several fundamental ways described below.

Programmability and customization

FPGAs are programmable devices by design that allow a high level of customization. A new design can be implemented and run on an FPGA in a matter of minutes or hours. In contrast, implementing a new ASIC or a full-custom design is very expensive, and takes months to create a new set of masks and fabricate a new silicon.

Design cycle time

A design cycle of an FPGA-based design can take as little as a week. A full design cycle of an ASIC or a full-custom design can easily take a year. The verification of an ASIC design is much longer and more complex that FPGA, and involves more engineering resources.

Cost

FPGAs have much higher cost than the equivalent ASIC. The cost difference can be compared using different metrics: per device, per equivalent capacity unit (typically ASIC gate), or cost per pin.

Design size

ASIC designs have a higher gate density, measured in transistors per silicon area. It also takes more transistors in FPGA to implement the same logic function as in ASIC because that logic is programmable and includes configuration capabilities and more routing. ASIC designs, measured in ASIC gates, can be one or two orders of magnitude larger than FPGA designs. Intel processors, nVidia GPUs, or Freescale SoC reached the size of hundreds of millions of ASIC gates, whereas the largest FPGA is less than ten million gates.

Power

FPGAs have a higher power consumption than ASICs. Advances in the CMOS process to reduce transistor size, and improvements in synthesis and physical implementation tools related to power optimizations techniques enabled FPGAs to be used in power-sensitive applications. Still, there is a significant power consumption gap between FPGAs and ASICs.

Performance

ASIC designs are faster. FPGA designs can typically run up to 400MHz. The peak performance of the fastest Xilinx-6 DSP48 embedded primitive is 600MHz. At the time of writing this book many multi-core processor SoCs exceeded 1GHz barrier, and the fastest Intel processor now runs at 3.2 GHz speed. Several conducted studies conclude that the performance difference is five times higher in favor of the ASIC compared to the frequency of the same FPGA design with the same fabrication process.

The main reason for such a big difference in performance is a different FPGA design architecture that includes configurable logic blocks, IO blocks which provide off-chip interface and routing that makes the connections between the logic and IO blocks.

Logic

FPGAs contain configurable logic blocks to implement the functionality of a circuit. In Xilinx FPGAs a logic block is called Slice, which contains several look-up tables (LUTs), registers, a carry chain, and some other logic depending on the FPGA family and slice type. This is fundamentally different than an ASIC standard cell, which is a non-configurable group of transistors that enables an abstract logic representation using a hardware description language (HDL).

The number of tional logic levels between registers in FPGA designs is several times lower than in ASIC designs. For example, in an FPGA design running at 300 MHz and using the fastest device, the number of logic levels typically doesn't exceed 5. An equivalent

ASIC circuit can have tens of logic levels.

Because of such a big difference in the number of logic levels, the overall logic composition of an ASIC design is different from the FPGA. A typical ASIC design ported to FPGA is LUT-dominant. That is, it contains much more LUTs than registers. That causes inefficient use of FPGA logic resources and suboptimal packing of LUT and registers into slices.

Routing

Routing refers to the interconnect between logic blocks and IO. An FPGA routing has two fundamental differences from ASIC. First, there is a fixed amount of FPGA routing resources, arranged as a horizontal and vertical grid. That routing layout makes the average connection length between two logic blocks longer, compared to a custom ASIC routing. Longer FPGA routes require additional buffers to overcome signal degradation. The second difference is that FPGA routing is programmable, which is the feature which enables FPGA configurability.

Clocking

Clock tree structure is custom-build for each ASIC design. FPGAs already include built-in clock management primitives that perform frequency synthesis and phase-locked loop (PLL), and dedicated low-skew clock routing resources. ASIC designs are using a clock gating technique to save more power than FPGA designs.

Design flow

There are many similarities between ASIC and FPGA design processes. Both include RTL synthesis, place and route, and timing closure steps. FPGA tools are sophisticated enough to support hierarchical design flow, low-level control over logic floorplanning, timing analysis, and support for script-based build flows.

However, there are a few fundamental differences between ASIC and FPGA design flow. FPGA flows don't require clock tree synthesis and scan chain insertion, because they are already part of the FPGA. Floorplanning FPGA design, and performing place and route are done by an FPGA design engineer, whereas those tasks might be performed at the ASIC foundry. Testing and verification of an ASIC design is an important part of the design process. Such tasks as a built-in self test (BIST), and automatic test pattern generation (ATPG) are non-existent in FPGA design process, because FPGA silicon has already been tested during manufacture.

Resources

[1] Ian Kuon and Jonathan Rose, *"Quantifying and Exploring the Gap Between FPGAs and ASICs"*, Springer, 2009, ISBN: 978-1-4419-0738-7

[2] Ian Kuon and Jonathan Rose, *"Measuring the Gap Between FPGAs and ASICs"*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) , Vol. 26, No. 2, pp 203-215, Feb. 2007

47. Selecting ASIC Emulation or Prototyping Platform

Verification of today's multi-million gate ASIC designs requires speed that in most cases cannot be provided using software-based simulation. ASIC emulation or prototyping on an FPGA-based platform is a cost-effective option available to design and verification engineers. It can speed up your software-based simulation exponentially, which will improve coverage and decrease the overall time-to-market availability of the product. Another role of prototyping is to provide a pre-silicon platform for early system integration of the design for firmware and applications software.

Selecting an emulation or prototyping platform that is capable of accelerating ASIC design verification is a complex process that can take you a long time before you make the final decision. The process includes a thorough evaluation of existing platforms and making sure that all of the technical and budgetary requirements like emulation capacity, speed, cost, coverage, and tool support are met. One of the platform evaluation goals is to uncover any hidden issues and perform a feasibility analysis. FPGA-based platforms are more restrictive than the ones based on custom processors. Some ASIC designs might have a fundamental problem of being partitioned, synthesized, or implemented for a specific FPGA-based platform.

Nomenclature

There is no consistent terminology that defines the different kinds of hardware acceleration platforms. Terms like emulation, prototyping, co-emulation, co-simulation, and in-circuit emulation are often used interchangeably, and the difference is not always clear.

There are several differences between emulation and prototyping platforms. Emulation platforms provide better debuggin capabilities close to the ones available in software simulators. That allows you to use emulation platforms in a wider range of project phases: from the block-level simulation to full-chip ICE (In-Circuit Emulation) and debug. Emulation platforms offer a more integrated solution such as a push-button build and debug flows, and more sophisticated software tools. Prototyping platforms lack good debug visibility, and are typically used during pre-silicon system integration and software debug. A prototyping platform might be a simple bare bones board with rudimentary software tools. The performance, or the emulation speed, of a prototyping platform is better: tens of MHz comparing to a few MHz in the emulation platform. Emulation platforms are usually more expensive than prototyping platforms.

Another class of acceleration platforms is simulation accelerators, or co-simulators. As the name suggests, co-simulators assist software simulators by running a portion of the design at a higher speed on a dedicated hardware platform. Typically, that part is synthesizable. The behavioral testbench is running on a host and communicates with the hardware platform using proprietary protocol over JTAG, Ethernet, PCI Express, or other high speed links. Xilinx ISIM simulator supports a co-simulation mode where part of the design runs on an FPGA.

Co-emulation is a more recent addition to the list of hardware acceleration tools. The primary distinction from a co-simulation is that the communications to the hardware platform is raised to the transaction level rather than being at the implementation level. In-circuit emulation (ICE) is not a platform, but instead is a method explaining how the hardware acceleration platform is used.

Architecture

Hardware acceleration platforms are designed using two main architectures based on FPGAs, or custom processors. FPGA-based platforms are usually cheaper, have faster emulation speed, and typically are used for ASIC prototyping. Emulation platforms are designed using hundreds, or even thousands of custom processors. They offer full design visibility, have higher capacity, and faster compile times.

Cost

Emulation platforms are usually more expensive than prototyping counterparts, and can easily reach several million dollars for the highest capacity configurations. Usually, the cost of a commercial hardware acceleration platform is proportional to its capacity. Additional components that add to the cost of ownership include maintenance cost, the cost of verification libraries, transaction models, tool add-ons, and device emulators.

Typically, platform vendors provide a lease option. Because the emulation or prototyping platform is required only for a relatively short period of time, typically a few months till the design tape-out, it is more cost effective to lease it rather than buy.

Frequently used metrics to compare different platforms in terms of their cost are cost-per-gate, and cost-per-emulation-cycle.

Capacity

The capacity of an ASIC emulation or prototyping platform is typically measured in terms of equivalent 2-input ASIC gates. Using these metrics allows the comparison of the capacity of very different platform architectures. Capacity can range from several million ASIC gates to over a billion for high-end emulation platforms. Different approaches to perform ASIC design capacity estimate are discussed in Tip #38.

On-chip memory

Many ASIC designs require a significant amount of on-chip memory for embedded processors, system caches, FIFOs, packet buffering, and other applications. Emulation and prototyping platforms provide a wide range of memory capacities from Megabyte to a Terabyte.

Emulation Speed

Different approaches to perform emulation speed estimates are discussed in Tip #39.

Partition

A typical ASIC design is too large to fit into a single FPGA, and needs to be partitioned to run on a multi-FPGA platform. The partition process might be a complex optimization task that includes meeting platform's cost, speed, and capacity goals, while ensuring proper connectivity between FPGAs, and reasonable complexity of the partition process. Different partitioning options are discussed in Tip #48.

Software tools

Hardware acceleration platform vendors provide very different software tools, ranging from automated partitioning software, to IDE, optimized synthesis tool, and debugger. Good software tools are the key to improving verification productivity.

Custom platforms

Intel Atom prototyping platform

One of the designs prototyped on a custom Xilinx FPGA-based platform is the Intel Atom processor. The following picture shows the prototyping platform.

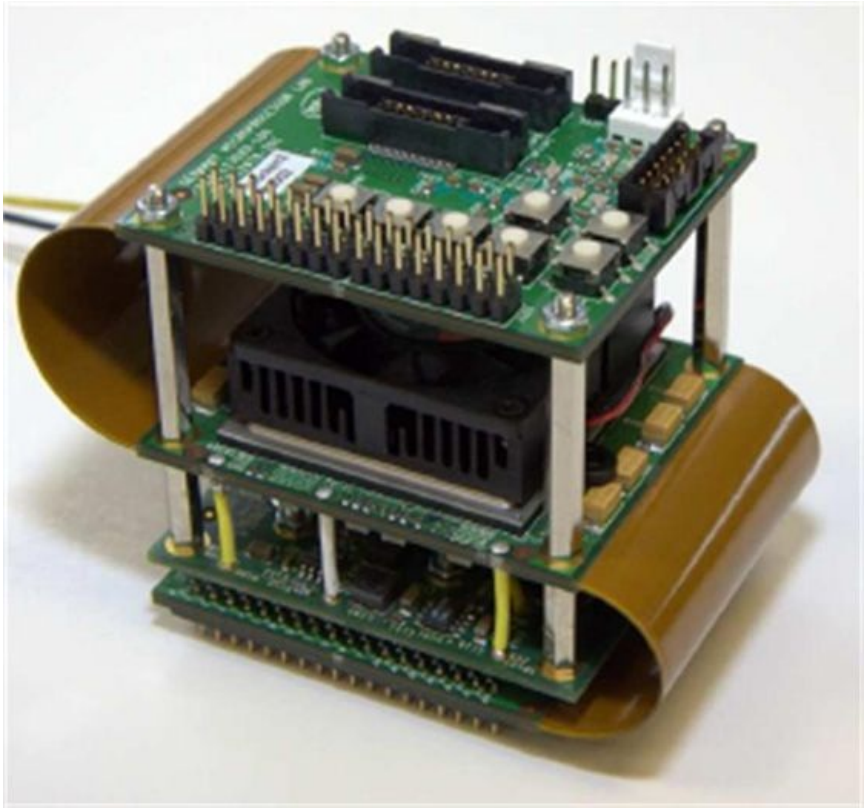

Figure 1: Intel Atom emulation platform (source: "*Intel Atom processor core made FPGA-synthesizable*", Microarchitecture Research Lab, Intel corporation)

This platform is not available for general use, and is shown as an example of a custom platform.

Build vs. Buy

Modern high-end FPGAs certainly make the task of designing a custom hardware acceleration platform realistic. Several high-end FPGAs have enough capacity, user IOs, and embedded memory to emulate small to medium-size ASIC designs.

Building a custom FPGA-based hardware acceleration platform can offer a significantly higher emulation speed approaching the maximum speed that an FPGA supports. In some cases running the emulation at 200-300MHz is a realistic expectation.

A custom-built platform can have features or capabilities that commercial platforms don't provide. For example, a custom-built platform can have a very high amount of external memory, a particular communication interface such as 10 Gigabit Ethernet, or a specific connectivity between FPGAs.

The main disadvantage of building a custom platform is significant design and bring-up time and effort. A hardware acceleration platform has a very short life cycle, from a few months to a year. By using an off-the-shelf platform, engineering teams can focus on ASIC design verification, rather than on building a short-lived platform.

Also, there is no definitive answer as to which approach is cheaper. There are several factors that affect the cost: the complexity of the custom-build platform, engineering effort, and the number of units to be manufactured.

Third party ASIC emulation and prototyping platforms

The following is a list of major ASIC emulation and prototyping platforms and tools.

Platform: Incisive Palladium

Company: Cadence

http://www.cadence.com/products/sd/palladium_series/pages/default.aspx

Figure 2: Palladium systems (source: Cadence Palladium datasheet)

Cadence Incisive Palladium Enterprise Simulator is a high-performance emulation platform. It is based on multiple custom processor-based compute engines designed for running verification applications. Palladium provides easy-to-use compiler optimized for fast runtime, and a broad ecosystem of verification libraries, software tools, and hardware add-ons to support emulation and system level hardware/software co-verification. Key features of the platform are (source: Cadence Palladium datasheet):

- Capacity: up to 2 Billion gates
- Dedicated user memory: up to 1 Terabyte
- IO pins: up to 60 thousand
- Emulation speed: up to 4 MHz
- Compile time: 10-30MGate/hour on a single workstation
- Transaction and assertion-based accelerations
- SpeedBridge adapters for connecting a wide range of peripheral devices using PCI Express, Ethernet, USB, AXI, and many other protocols.

Palladium is considered the market leader in providing hardware acceleration solutions for ASIC design verification.

Platform: Veloce
Company: Mentor Graphics
http://www.mentor.com/products/fv/emulation-systems/veloce

The Veloce product family is a high performance simulation acceleration and pre-silicon, emulation solution. It is based on multiple custom-designed processors. Veloce product line offers five scalable verification platforms with capacities from 8 million gates up to 512 million gates

Platform: ZeBu-Server
Company: EvE
http://www.eve-team.com

ZeBu-Server is an FPGA-based system emulation platform with capacity of up to 1 Billion gates and emulation speed of up to 30MHz. ZeBu-Server primarily aims at large-scale, multi-core chip and system emulation applications. ZeBu-Server is the highest performance emulator on the market. It uses Xilinx Virtex-5 LX330 FPGAs.

Products: RocketDrive, RocketVision
Company: GateRocket
http://www.gaterocket.com

GateRocket offers RocketDrive, a small FPGA-based peripheral, which acts as accelerator for the HDL simulator, and RocketVision, a companion software debugging option.

Platform: HES
Company: Aldec
http://www.aldec.com

Aldec Hardware Emulation System (HES) allows using off-the-shelve prototyping FPGA boards for simulation acceleration and emulation. The strength of the tool is in automated design compilation, standard testing interface, and integration with leading hardware/software debugging tools such as Synopsys Certify.

Company: Dini Group
http://www.dinigroup.com/new/index.php

Dini Group offers high-capacity FPGA boards for ASIC prototyping and emulation systems. The following figure shows an FPGA-based prototyping platform from Dini Group using 16 Xilinx Virtex-5 FPGAs.

Figure 3: FPGA-based prototyping platform (courtesy of Dini Group)

48. Partitioning an ASIC Design into Multiple FPGAs

A typical ASIC design is too large to fit into a single FPGA and needs to be partitioned to run on a multi-FPGA platform. Design partition is a process of assigning parts of the ASIC design to individual FPGAs. Partition process might be a complex optimization task that includes meeting platform's cost, speed, and capacity goals while ensuring proper connectivity between FPGAs and reasonable complexity of the partition process. There are several commercial partition tools that employ proprietary algorithms to perform partitioning of a complex ASIC design into multiple FPGAs. Designers can develop custom scripts that perform partition for simpler ASIC designs with clearly defined module boundaries.

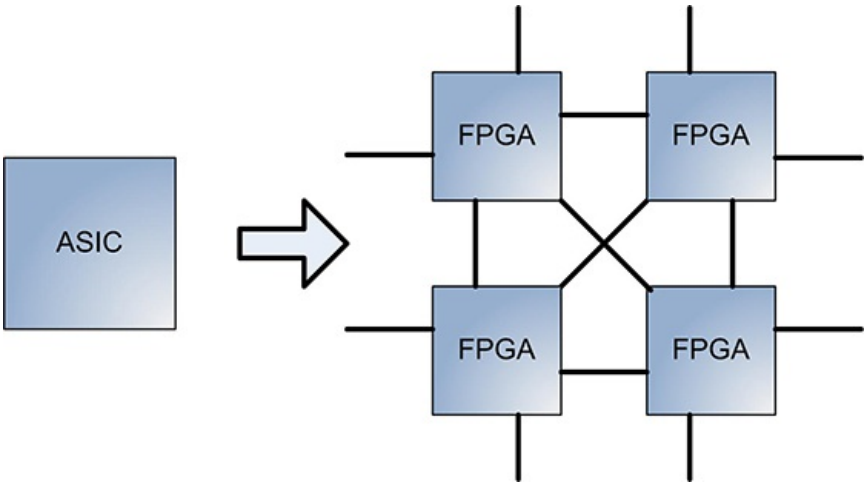

Figure 1: Partitioning an ASIC design into multiple FPGAs

The process of choosing and implementing a design partition includes several steps: requirements definition, feasibility analysis, identifying partition points, selecting FPGAs, defining interfaces between FPGAs, designing clocking and reset schemes, selecting partition tool, and defining partition flow. After the partition is done, designers might want to perform different optimizations.

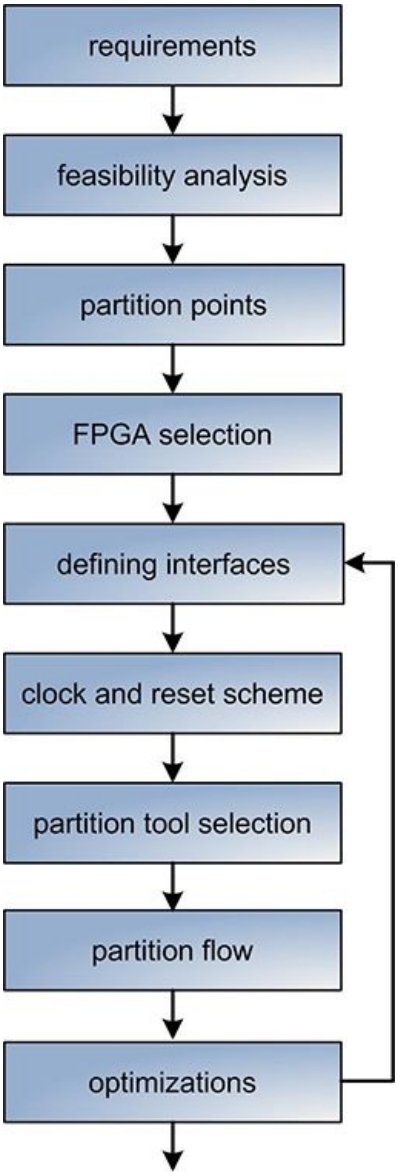

Figure 2: Partition process

Requirements

Requirements can be divided into two groups: soft and hard. Hard requirements are inflexible and must be met. Cost, platform capacity, the amount of on-board memory, the minimum number of user IOs, and specific peripheral devices (such as USB, PCI Express, and Ethernet) are hard requirements. Emulation speed and target FPGA utilization are soft requirements because they can be changed without significant impact on the platform functionality.

Feasibility analysis

The goal of the feasibility analysis is to determine whether the partition is required and, if it is, whether the design can be partitioned to fit into a specific FPGA-based prototyping platform. Even small ASIC designs that can fit into a single FPGA based on the capacity estimate might require partition if that FPGA has insufficient embedded memory, DSP, user IO, or clocking resources. At this point it is important to get a ballpark estimate of the number of required FPGAs, their capacity, and the interconnected structure. Another area of concern is whether there are hidden issues with synthesis and physical implementation flow: Partitioned design is too complex to be built, logic utilization of some of the FPGAs is too high, logic multiplexing ratio on the FPGA-to-FPGA interfaces exceeds the synthesis tool limit, routing is too congested and causes place-and-route tool errors, and other issues.

Identifying partition points

Most of the ASIC designs are modular and lend themselves to straightforward partitioning into multiple FPGAs. Examples are SoC with multi-core CPU subsystems, caches, memory controllers, and other peripherals, interconnected by busses with a low signal count. Other designs, such as graphics processors (GPU) with a more complex mesh interconnect structure between modules, might require automated partitioning tools to find the optimal partition points.

FPGA selection

FPGA selection includes choosing the minimum number of FPGAs required for partition, their capacity, and speed. The hardware platform doesn't have to consist of homogeneous FPGAs or be symmetrically interconnected. It can contain FPGAs with different

capacities, from different families and even different vendors, as long as the partitioned design meets all the requirements.

The following picture shows a block diagram of a simple 2-FPGA prototyping board from Dini Group with asymmetric connectivity.

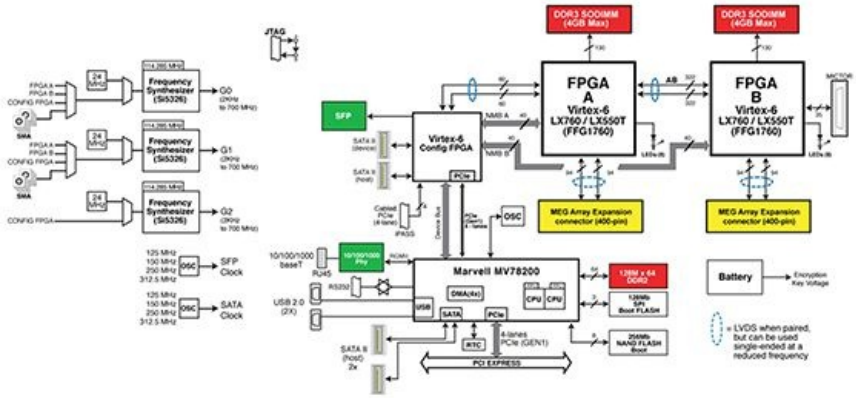

Figure 3: FPGA-based prototyping board (courtesy: Dini Group)

Interfaces

When a design is partitioned to multiple FPGAs, the total number of IOs needed for the entire design increases. This is known as Rent's rule and is expressed as:

$T = t \cdot g^p$

Where T is the total number of pins, g is the design size, p is "Rent's exponent" between 0.5 and 0.8, and t is a constant.

The implication of this rule is that the number of IOs in the partitioned design does not decrease as fast the size of partitioned modules. Splitting the original ASIC design into multiple FPGAs will require more pins than in the original design.

In general, a higher number of external pins available for signal interfaces between FPGAs allows for a lower multiplexing ration of the signals, which directly affects the overall emulation speed. This point is discussed in Tip #39.

Clocking scheme

The clock distribution scheme in a multi-FPGA system can be complex. One of the main challenges is tackling the clock skew. The overall clock skew becomes the combination of skews on the board and inside the FPGAs. When a clock is generated inside one FPGA and distributed to the others, the board skew component for each receiving FPGA has to be balanced. If not, it will cause hold-time violations. There are several clock distributions schemes to combat this issue. One is a loop-back clock structure, in which the clock-generation FPGA also receives the same clock. Another technique is replication of the clocking network to eliminate or avoid the board skew component.

The following figure illustrates the clock loop-back technique. An external clock enters FPGA 1, goes through a PLL, and is distributed to FPGAs 2 and 3 and back to FPGA 1. An important requirement is that the returning clock signal has the same delay to each of the FPGAs.

Figure 4: Clock loop-back technique

In large, multi-clock designs, several different clock distribution schemes may co-exist on the same prototyping board. For such designs, performing a manual partition that completes all necessary clock conversions is impractical. This is a task for partition tools that can do it automatically.

Reset scheme

One requirement for designing a reset scheme is that the logic in all FPGAs on the prototyping board comes out of reset at the same time. That imposes some limitations on reset line delays and the choice of a synchronous or asynchronous reset scheme.

Selecting partitioning tool

Partition options can be categorized into three main groups: manual partition, automatic partition, and semi-automatic partition that includes some manual changes.

Manual partition can be relatively simple to perform on a small modular design. It allows for the highest level of customization that automatic partition tools lack. Another reason is the high cost of partition tools. However, the cost of using the manual partitioning approach includes the cost of writing and maintaining the conversion scripts, performing verification of the partitioned design, and might exceed the cost of partition tools.

Partition can be performed at the RTL or post-synthesis netlist level, typically in Electronic Design Interchange Format (EDIF). One advantage of performing partition at the netlist level is that it's easier to perform accurate area estimates. Another advantage is that it's easier for an automated tool to find an optimal partition when working with a single flat netlist database. However, most of the

custom-developed scripts perform partition at the RTL level.

Automatic partitioning tools typically offer the following features:

- Perform coarse area estimation for a specific prototyping board.
- Perform feasibility analysis on a prototyping board without pin assignments and undefined connectivity between FPGAs.
- Including both pin and area requirements as part of the process of finding the optimized partition.
- Can perform partition at either RTL or post-synthesis netlist levels.
- Support for multiple third-party and custom prototyping board.

Even with using partition tools, it's difficult to achieve a full automation of the partition process. Every ASIC design has some unique features that require manual intervention. Also, partition tools might not perform some parts of the design optimization. In practice, the partition flow will start with some manual modifications using scripts and be completed by the automated tool.

Partition flows

There are two main partition flows: top-down and bottom-up.

In the top-down flow, the entire design is synthesized into a single flat netlist, usually in EDIF format, and then the netlist is partitioned using an automated tool. Some advantages of the top-down approach include a more compact netlist because the synthesis tool performs cross-module optimization on the entire design. A partitioning algorithm could be applied to the entire design with different constraints, resulting in a more optimal partition. The timing analysis can also be more accurate and cause higher performance.

In the bottom-up flow, the entire design is first partitioned into modules, such that each module fits into a single FPGA. Then each module is synthesized. Among the advantages of the bottom-up approach is that it can be more suitable for manual partition flow. Also, the runtime for very large designs can be faster than with the top-down approach because of the smaller netlist database.

Algorithms

Most of the partitioning algorithms used in commercial software tools are either patented or constitute company trade secret. An efficient partitioning algorithm is a significant competitive advantage of the product. It can result in faster build time, better performance, or lower logic utilization.

Many algorithms use an approach called simulated annealing to find the optimal partition. The algorithm represents a flattened design netlist as a graph. Then, the search space for finding the optimal solution is limited by applying constraints, such as speed and capacity goals, the number of IOs, and other metrics. The end result of the algorithm is the optimal cut in the graph, which represent the partition.

Optimizations

After successful partition, designers might want to perform different optimizations to improve emulation speed, decrease FPGA utilization by rebalancing the logic, or increase the speed of partition, synthesis, and physical implementation flow. Logic replication achieves good results for improving routability or design speed.

Frequently, decreasing the overall design build time is more important than performing other optimizations because it allows new designs to build quicker after making modifications, which speeds up the verification process.

Partitioning tools

Tool: Certify
Company: Synopsys
http://www.synopsys.com/Systems/FPGABasedPrototyping/Pages/Certify.aspx

Synopsys Certify is considered the best tool to perform ASIC design partitioning into multiple FPGAs. The tool can perform both automatic and manual partitions. It can resolve the board routability issues for custom and off-the-shelve prototyping boards. It reads-in the netlist of the target board and applies a routing algorithm to map the connections between partitions onto wires on the board.

Tool: ACE Compiler
Company: Auspy Development Inc.
http://www.auspy.com

The Auspy Custom Emulator (ACE) Compiler tool performs a multiple-FPGA design partition and integrates with commercial and custom prototyping platforms. The tool allows multi-language design import and full bottom-up synthesis for accurate gate-level estimation.

Many vendors of FPGA-based prototyping platforms also provide design partition tools specific for their platforms. One example is the EvE ZeBu-Server.

49. Porting Clocks

Clocking implementation is fundamentally different between ASIC and FPGA designs. ASIC designs require manual clock tree

synthesis, balancing clock delays, and other tasks. On the other hand, FPGAs have built-in clocking resources: PLLs, frequency synthesizers, phase shifters, and dedicated low-skew clock routing networks. ASIC designers have more freedom to implement fully custom clock networks. A predefined clock tree structure in FPGA makes the design much simpler. However, that becomes a disadvantage when porting an ASIC design to FPGA.

Clock trees of an ASIC design need to be ported to the quivalent clocking resources of individual FPGAs, and distributed across different FPGAs on the emulation platform.

An FGPA has a limited amount of clocking resources. For example, the largest Xilinx Virtex-6 LX760 FPGA has 18 Mixed Mode Clock Manager (MMCM) blocks. It's sufficient for most of the FPGA designs. However, that might not be enough for ASIC designs with a large number of custom clock trees. Another FPGA limitation is the limited number of global clocks that can enter one clock region.

Clock gating

ASIC designs are optimized for low power, and make extensive use of a clock gating technique for power optimization. The following figure shows an example of a gated clock, where the clock inputs of two registers are connected to a combinatorial logic.

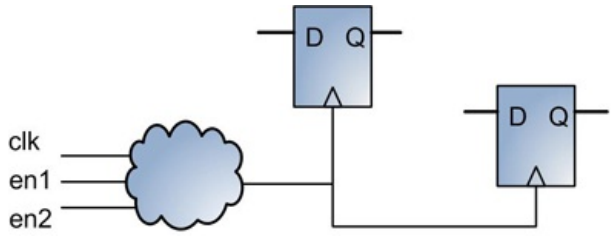

Figure 1: Gated clock

Under certain logic conditions based on `en1` and `en2` inputs, the clock coming to the registers is turned off, effectively both disabling the register, and conserving power.

Clock gating can lead to suboptimal performance results in FPGAs. The reason is that FPGAs are using dedicated high-speed and low-skew routing resources for clock nets. When the clock of a sequential element is gated, it is taken off the dedicated high-speed clock routes, and connected to a general-purpose routing instead. The resulting implementation introduces clock skew, and leads to poor performance and potential setup and hold-time violations. It is recommended that one performs conversion of all gated clocks to their functionally equivalent implementation.

Converting gated clocks

Gated clock conversion is a process of modifying the logic so that the clock inputs of the registers are connected directly to dedicated clock routes instead of combinatorial or sequential logic outputs.

One way to perform a gated clock conversion is to insert a multiplexer in front of the data input, and then connect the clock net directly to the clock input. The following figure shows the resulting circuit.

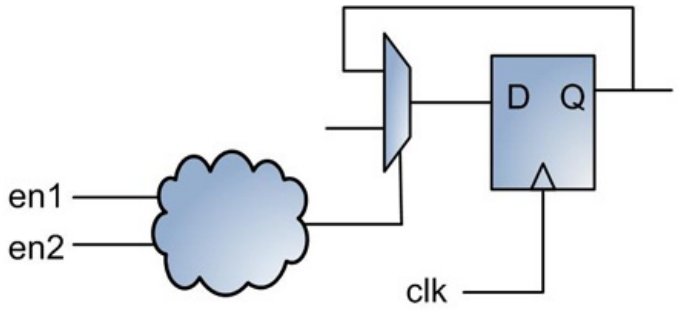

Figure 2: Converted gated clock using feedback and multiplexor

Another gated clock conversion method takes advantage of the enable input of the register, as shown in the following figure.

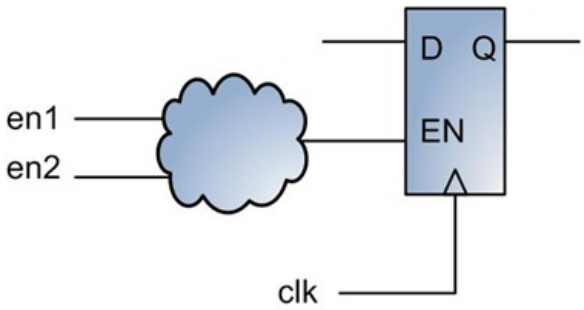

Figure 3: Converted gated clock using enable input

Tool options for automatic gated clock conversion

Some synthesis tools, such as Synopsys Synplify Pro and Synplify Premier, perform automatic gated clock conversion. To perform the conversion, Synopsys Synplify Pro and Synplify Premier require certain design conditions to be met: the gated clock logic must be combinatorial, clock signal has to be constrained, and the synchronous primitive driven by a gated clock must not be a black box.

Gated clocking schemes provided in the above examples are simple. In real designs, the gated clocking logic is more complex, and can drive not only registers, but memories, DSP blocks, and other primitives. A synthesis tool might not be able to perform automatic gated clock conversion in all cases.

Clock enables

Unlike gated clocks, ASIC designs that contain the logic with clock enables in the majority of cases do not require special porting techniques. FPGA synchronous primitives – registers, Block RAMs, DSP48 – already have dedicated clock enable input, which is automatically used by synthesis tools.

Porting clock management primitives

In most cases ASIC clock management modules, such as PLLs, can be directly ported into FPGA counterparts with a minor interface adaptation. Xilinx FPGAs provide a variety of configurable clock management modules such as DCM, PLL, and MMCM, clock buffers, clock multiplexors, and other clock-related resources. A more detailed discussion on Xilinx FPGA clocking resources is provided in Tip #20.

50. Porting Latches

Latches are used for different purposes in ASIC designs. Time borrowing is one of the latch usage models. That technique allows significant performance improvement in heavily pipelined designs and is briefly described below. Another motivation to use latches is that they use much less area. A register uses twice as many transistors as a latch. A common master-slave register implementation used in ASIC designs consists of two cascaded latches.

The figure below illustrates time borrowing technique. The technique takes advantage of latches that replace registers along the critical timing paths. They take up a portion of a clock period from adjacent registers and, by doing so, effectively increase the clock period. The theory behind the time borrowing technique is discussed in several papers, some of which are mentioned in [1]

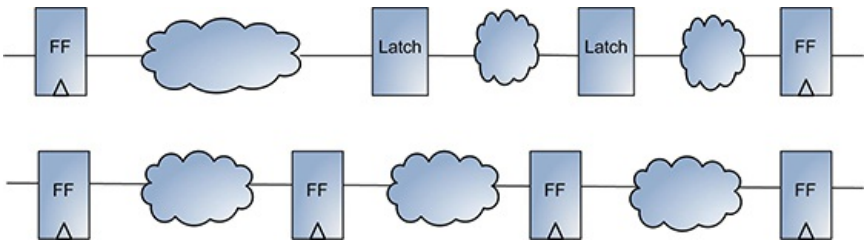

Figure 1: Time borrowing using latches

Both latches and registers are implemented using the same Xilinx FPGA hardware primitive. That hardware primitive shares the same input and output data, reset, and enable signals, and can be configured as a register with an edge-sensitive clock or as a latch with a transparent gate enable. Therefore, unlike ASIC implementation, latches use the same area in FPGA and offer no advantage.

Even though both latches and registers are storage elements, they are fundamentally different. A latch is level-sensitive and transparent whenever the gate enable input is active. I.e., as long is the gate enable is active, there is a direct path between data input and output. The data is stored in latch when the gate enable becomes inactive. On the other hand, a register is an edge-sensitive element. The data is stored in a register either on the rising or falling edge of the clock.

Using transparent latches in FPGA designs is not desirable for several reasons. When a latch is in the transparent state, glitches on the data input can pass to the output. Static timing analysis (STA) of a design with only synchronous registers is much simple than the one with transparent latches because the STA tool only needs to consider either the falling or rising edge of the clock. On the contrary, the STA of a design with latches needs to account for both the rising and falling edge of a gate enable and its duration for each latch. That makes latches subject to duty cycle jitter, as their behavior depends on the arrival times of both the rising and falling edge of a gate enable. Designs with edge-sensitive registers only need to consider the arrival time of one clock edge and are therefore immune to the duty cycle jitter. The STA of latch-based pipeline designs with transparent latches is even more challenging due to the characteristics of the time borrowing. A delay in one pipeline stage depends on the delays in all previous pipeline stages. Such high dependency across pipeline stages makes it very difficult to take into account all the factors that affect static timing analysis. The timing analysis complexity of such a design grows non-linearly with the number of pipeline stages.

Because of the above reasons, latch-based designs require more careful and elaborated timing constraints to ensure correct operation. Designers that perform synthesis and physical implementation of a latch-based design need to have a deep understanding of the full design intent and multiple timing relationships in order to add proper timing constraints.

It is recommended to convert all transparent latches in the ASIC design to equivalent logic circuits using edge-sensitive registers.

There isn't a universal conversion methodology applicable to all possible latch coding styles.

Most synthesis tools infer latches from incomplete conditional expressions, such as an *if* statement without a corresponding *else*, as shown in the following examples.

```
// no reset
always @(*)
  if(enable)
   d_out[0] <= d_in[0];

// a latch with asynchronous reset
always @(*)
  if(reset)
   d_out[1] = 1'b0;
  else if(enable)
   d_out[1] = d_in[1];

// a latch with asynchronous set


always @(*)
  if(set)
   d_out[2] = 1'b1;
  else if(enable)
   d_out[2] = d_in[2];
```

Xilinx XST produces the following log whenever it recognizes a latch.

```
Synthesizing Unit <latches>.
WARNING:Xst:737 - Found 1-bit latch for signal <d_out<2>>.
WARNING:Xst:737 - Found 1-bit latch for signal <d_out<0>>.
WARNING:Xst:737 - Found 1-bit latch for signal <d_out<1>>.
Latches may be generated from incomplete case or if statements. We do not recommend the use of latches in
FPGA/CPLD designs, as they may lead to timing problems.
Summary:
  inferred   3 Latch(s).
=========================================
HDL Synthesis Report

Macro Statistics
# Latches                              : 3
1-bit latch                             : 3
```

The conversion should be performed using recommended register coding style, which to some extent depends on a synthesis tool. A simple method is to convert gate enable input of a latch into a clock:

```
// a register with synchronous reset and positive edge clock
wire clk = enable;

always @(posedge clk)
  if(reset)
   d_out[1] <= 1'b0;
  else
   d_out[1] <= d_in[1];
```

The implementation variants of the above example can be a positive or negative edge clock (Xilinx FPGAs don't support dual-edge registers), synchronous or asynchronous reset, and register set or clear upon reset assertion.

Resources

[1] *Statistical Timing Analysis for Latch-based Pipeline Designs.* M. C. Chao, L. Wang, K. Cheng, S., and Kundu. s.l.: IEEE/ACM, 2004. Proceedings of the 2004 IEEE/ACM international Conference on Computer-Aided Design.

51. Porting Combinatorial Circuits

A combinatorial circuit is defined as logic that doesn't contain synchronous components, such as registers. This Tip discusses different challenges encountered during porting ASIC combinatorial circuits to FPGA.

Self-timed circuits

A good example of a self-timed circuit is the Synchronous RAM array controller. Memory read or write functions trigger a sequence of micro-transactions within a controller to access asynchronous RAM array: address decoding, precharge, accessing a memory cell, data demultiplexing, and other operations. What appears to the user as a single-clock SRAM read is actually a self-timed sequence of events.

Self-timed circuits rely on precise routing and logic delays that are impossible or very difficult to replicate in FPGA architecture. One

method of porting a self-timed circuit is to add an artificial high-frequency clock that runs at the speed of the fastest micro-transaction and synchronizes the logic to that clock. If it proves too difficult to implement or it introduces too many logic modifications, the entire circuit can be rewritten as a synthesizable model.

Latches and Flip-flops

Gate-level implementation of latches and flip-flops in most cases can be modeled as an FPGA register or latch. There are several types of flip-flops and latches that cannot be directly ported to FPGA and require custom implementation. Examples are flip-flops with both synchronous and asynchronous resets, set and clear, scan chain, double-edge clock, and several others.

Combinatorial loops

ASIC designs contain many circuits with combinatorial loops: gate-level implementation of latches and flip-flops, keepers, and many others.

The following figure provides some examples of circuits with combinatorial loops. The left circuit is a keeper built using two invertors connected back to back. This circuit is used in ASIC designs as a storage element, or a bus keeper (or also called bus-hold), which keeps the logic value on a bus in the absence of active drivers. The example in the middle is an SR latch. The example on the right is a register reset fed back from the logic driven by the register output.

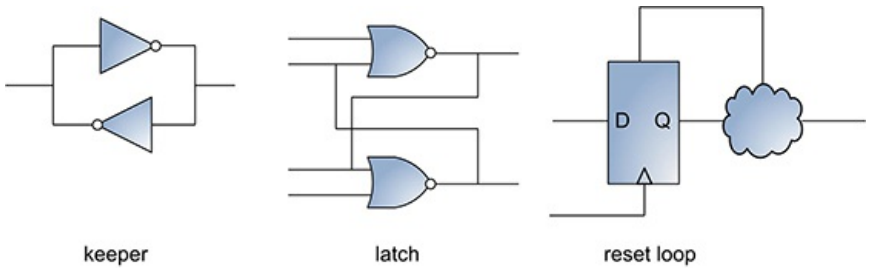

Figure 1: Example of circuits with combinatorial loops

The following is the Verilog code for the examples. The entire project can be downloaded from the book's website.

```
module combinatorial_loop_examples(
                     input clk,
                     input latch_s,latch_r,
                     output latch_q,latch_q_b,
                     inout keeper0, keeper1,
                     input  reset_loop_i,
                     output reg reset_loop_o);
   // latch
   nor nor1(latch_q,   latch_s,latch_q_b);
   nor nor2(latch_q_b, latch_r,latch_q);

   // keeper
   nor not1(keeper0, keeper1);
   not not2(keeper1, keeper0);

   // reset loop
   wire reset_loop_i_b = ~reset_loop_o & reset_loop_i;

   always @(posedge clk, posedge reset_loop_i_b)
      if(reset_loop_i_b)
         reset_loop_o <= 'b0;
      else
         reset_loop_o <= reset_loop_i;
 endmodule
```

Circuits that contain combinatorial loops can be synthesized for FPGA, but in most cases they will not function correctly. Combinatorial loop behavior depends on propagation delays through the logic in the loop and are therefore unpredictable.

The Xilinx XST synthesis tool will usually produce a combinatorial loop warning for simple (but not all) cases. Some loops might not synthesize at all by causing an infinite computation cycle that synthesis tools cannot handle.

For the above examples, XST produces warnings only for keeper and latch but not for the reset loop example. The Xilinx timing analyzer will also produce a warning that it cannot analyze some of the paths due to combinatorial loops.

The above examples are simple. In large real life designs, combinatorial loops can span multiple logic levels and might be very difficult to detect by simple code inspection or a simulation.

Software tools for commercial emulation platforms such as Cadence Palladium, Mentor Veloce, and EvE ZeBu employ various loop-breaking algorithms. The underlying idea behind the loop-breaking algorithms is to insert a synchronous element in the correct place in the loop without changing the circuit functionality. The inserted synchronous element can be clocked at the same or at a faster clock frequency, also known as oversampling technique.

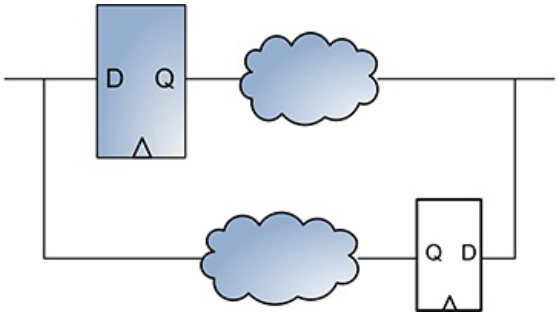

Figure 2: A register is inserted to break the combinatorial loop

Detecting and resolving combinatorial loops manually is especially challenging. It requires very good familiarity with design intent, circuit functionality, and implementation details. Typically, circuit porting to emulation platform is performed by a team of system or verification engineers and not by the original IC designers.

There are simple measures that designers can take to eliminate some of the combinatorial loops. They can remodel flip flops and latches by directly using an FPGA register and latch primitives. Many ASIC designs contain scan chains or other test logic that is not required in the emulated design. By tying off those circuits a lot of combinatorial loops are eliminated.

52. Porting Non-synthesizable Circuits

ASIC designs contain many circuits that are either not synthesizable at all or synthesized incorrectly for FPGAs. Some circuits cannot be synthesized due to the fundamental limitation of FPGA architecture. Examples include circuits that contain absolute delays, transistors and other switch-level primitives, 4-state Verilog values, signal strength, resistors, and capacitors.

FPGA synthesis tools provide different levels of Verilog language support. Xilinx XST is most strict in terms of what Verilog constructs can be synthesized. Synopsys Synplify provides a wider range of synthesizable Verilog constructs. Dedicated synthesis tools for ASIC emulation platforms, such as EvE ZeBu, provide the best Verilog language support.

The synthesizable subset of the Verilog language is usually well documented, and can be found in the synthesis tool user guide.

The rest of the Tip describes approaches and provides examples of porting different types of non-synthesizable circuits.

Absolute delays

Absolute delays are used to accurately model delays across delay lines and logic gates. Absolute delays are widely used in self-timed and asynchronous circuits. Delays can be implemented as a trace of specific length, or a chain of gates.

The following are Verilog delay examples using inertial delays, transport delays, and chains of *not* gates.

```
always @(*)
  #25 net_delayed = net_source;

not #(10) n1 (not_1, not_in);
not #(10) n2 (not_2, not_1);
not #(10) n3 (not_out, not_2);

always @(posedge clk)
   reg_out <= #5 reg_in;
```

Absolute delays don’t have an equivalent representation in the existing FPGA architectures. Most of the FPGA synthesis tools will produce a warning and ignore the delay.

There isn’t a good method of porting absolute delays without a major redesign of the circuit. One modeling approach is to convert an asynchronous circuit with a delay to a synchronous circuit by adding a high-speed oversampling clock.

Drive strength

Drive strength values are used in many ASIC circuits. Examples include RAM array controllers, power supply circuits, pull-up and pull-down resistors, IO circuits, and many others.

Verilog language defines the following signal strengths levels for each of the 0 and 1 logic level: *supply, strong, pull, weak,* and *highz.*

The following is an example of a one-bit storage unit that uses drive strength. The circuit is modeled as two back-to-back inverters and an access pass gate. Inverters are using *pull0* and *pull1* strength. When the pass gate is enabled, a signal with a *strong0* or *strong1* drive level overrides the *pull0* or *pull1* strength of the inverters.

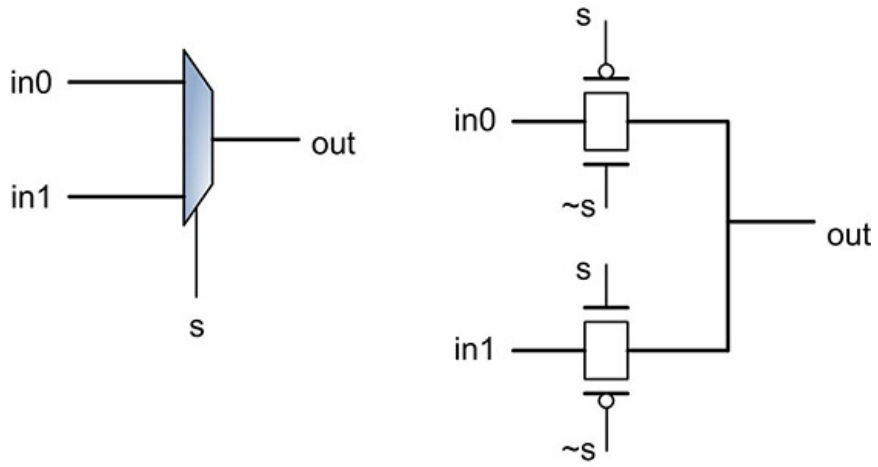

Figure 1: Bit storage circuit using drive strength

The Verilog model:

```
not (pull1,pull0) n1 (val_b, val);
not (pull1,pull0) n2 (val, val_b);
nmos n3 (in, val, en);
pmos n4 (in, val, ~en);
```

Drive strength doesn't have an equivalent representation in the existing FPGA architectures. Most of the FPGA synthesis tools will produce a warning and ignore the drive strength.

There isn't a good method of porting drive strength without a major redesign of the circuit. One modeling approach is to convert a single-bit net with assigned drive strength to a multi-bit bus. Each bus value is associated with a unique drive strength.

Logic strength

In addition to the drive strength, Verilog language provides four logic strength values that a signal can accept: 0, 1, z, and x. Most synthesis tools support only logic 0 and 1 values of the signals. Values of z and x are either ignored or produce a synthesis error.

In the technical literature, the logic strength range of a signal is often referred to as 4-state for 1,0,z, and x values, and 2-state for 1 and 0 values.

Many ASIC circuits are modeled using 4-state logic strength. A good example are tri-stated circuits, which use 0,1 to model logic levels and z to model a tri-state state. x is used to represent an unknown value. For example, after power-on a signal can be assigned x until the circuit comes out of reset and the signal is properly initialized. An unknown value x is very important for ASIC design verification, because it allows easy detection of error conditions.

Designers want to preserve the same 4-state behavior in the FPGA emulation model, which only supports 2-state. There isn't a generic solution that applies to all cases. One approach is to represent a 4-state logic strength using a two-bit bus.

Modeling 4-state values also includes porting all 4-state specific Verilog operators. Verilog has special equality and inequality operators that apply to 4-state signals: === and *!*==. Arithmetic, conditional, and relational operators produce different results if applied on 4-state and 2-state values.

The following table shows the results of evaluating 2-state and 4-state equality expressions.

Table 1: Evaluating equality expressions

{a,b}	*a == b*	*a === b*
00	1	1
01, 10	0	0
11	1	1
0x , x0	x	0
1x , x1	x	0
xx	x	1

Synthesis tools have different levels of support for 4-state logic and operators. For example, Xilinx XST will produce an error after encountering a === operator. Other synthesis tools will produce a warning and replace a 4-state === operator with a 2-state == equivalent.

Porting transistor-based circuits

ASIC designs frequently use custom-build circuits that contain CMOS switch-level primitives. Such circuits often achieve better performance and the most compact area utilization.

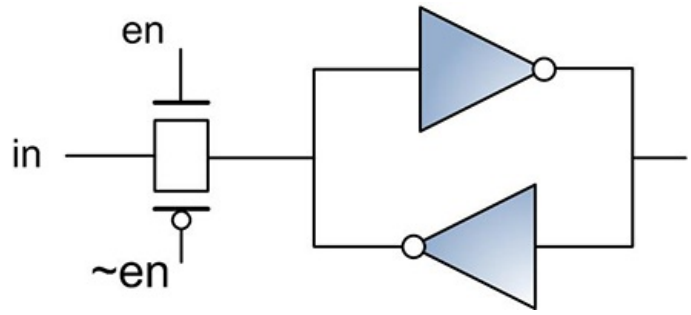

Figure 2: Two-to-one multiplexor implementation using pass gates

Xilinx XST doesn't support `cmos,pmos,nmos,tran,tranif0,tranif1` and other Verilog switch-level primitives. Synopsys Synplify provides limited synthesis support of `tran, tranif0,and tranif1` pass gates.

One way to port transistor-based circuits is to model them as a simple switch statement. The following are Verilog implementations of the two-to-one multiplexor using RTL model and with pass gates.

```
module mux2x1(input mux_in0, mux_in1, mux_sel,
          output mux_out_rtl,mux_out_passgate,
         mux_out_remodeled);

   assign mux_out_rtl = mux_sel ? mux_in1 : mux_in0;

   // non-synthesizable passgate implementation
   nmos p0 (mux_out_passgate,mux_in0, ~mux_sel);
   pmos p1 (mux_out_passgate,mux_in0, mux_sel);

   nmos p2 (mux_out_passgate,mux_in1, mux_sel);
   pmos p3 (mux_out_passgate,mux_in1, ~mux_sel);

   // implementation using remodelled transistors
   nmos_model p4 (mux_out_remodeled,mux_in0, ~mux_sel);
   pmos_model p5 (mux_out_remodeled,mux_in0, mux_sel);

   nmos_model p6 (mux_out_remodeled,mux_in1, mux_sel);
   pmos_model p7 (mux_out_remodeled,mux_in1, ~mux_sel);

endmodule // mux2x1

module nmos_model(output out,input in,ctrl);
   assign out = ctrl ? in : 1'bz;
endmodule

module pmos_model(output out,input in,ctrl);
   assign out = ~ctrl ? in : 1'bz;
endmodule
```

A two-to-one multiplexor is just a simple example of a transistor-based circuit. ASIC designs implement much more complex circuits using switch-level primitives: custom 256-1 multiplexors, latches and half-latches, flip-flops, custom state machines, memory arrays, and many others. A general approach to porting such circuits is to "divide and conquer." First, try to remodel individual switch elements. If that doesn't help solving all synthesis and physical implementation issues, remodel larger blocks.

Behavioral models

Non-digital components of an ASIC design, such as delay lines and transistors, are typically modeled using a behavioral subset of Verilog.

Behavioral models present a unique challenge. They are designed to represent circuit behavior and not to reflect internal design structure. Behavioral models are used in many cases; for example, by an IP core vendor, which delivers a netlist. A model might be as complex as necessary to represent the core but too complex for converting to a synthesizable design.

Small circuits are modeled using Verilog User Defined Primitives (UDP). UDP is non-synthesizable with most of the synthesis tools.

53. Modeling Memories

Memories are part of virtually every ASIC design. There are different types of memories: asynchronous RAM, synchronous RAM (SRAM), pipelined Zero Bus Turnaround (ZBT), Double Data Rate (DDR) DRAMs, Read Only Memory (ROM), and the list can go on. Memory sizes vary from a few-byte scratchpad, to several hundred kilobyte processor cache. Memory data widths vary from a single bit to 256-bit and even wider.

Typically, ASIC designs contain either on-chip SRAM, or a memory controller to interface with the external DDR DRAM. Xilinx provides memory controller (MIG) core with a full source code access that can be customized to model the one used in ASIC design.

In most cases asynchronous RAMs and SRAMs are foundry-specific and need to be converted to an FPGA-friendly functionally-equivalent Verilog model. Xilinx XST and other FPGA synthesis tools are capable of correctly infer and synthesize SRAMs that directly map into Xilinx BRAM or distributed RAM memory primitives.

In some cases, designers might need to modify the logic surrounding the memory in addition to modeling the memory itself. One example is asynchronous RAM. Xilinx embedded memories are synchronous, and have access latency of one or more clocks. Another example is write-enable for each of the data bits. Xilinx BRAM primitive has only byte-write enable. ASIC designs might use multi-port memories. An FPGA has a limited amount of embedded memory that might not be sufficient for some of the designs, such as large processor cashes. An ASIC memory might contain special features, such as error repair, redundancy, or built-in self test that don't have Xilinx equivalent.

Xilinx embedded memories have three access modes – read-first, write-first, and no-change – that can model most of the ASIC SRAM memories.

Tip #34 provides more details on Xilinx embedded memory features, and the rules on how to correctly infer BRAM or distributed RAM in HDL code.

There is a well-known technique to model bit-write enable using Xilinx BRAM. The technique takes advantage of a dual-port BRAM to perform read-modify-write operation in a single clock cycle. The following figure illustrates how it's done.

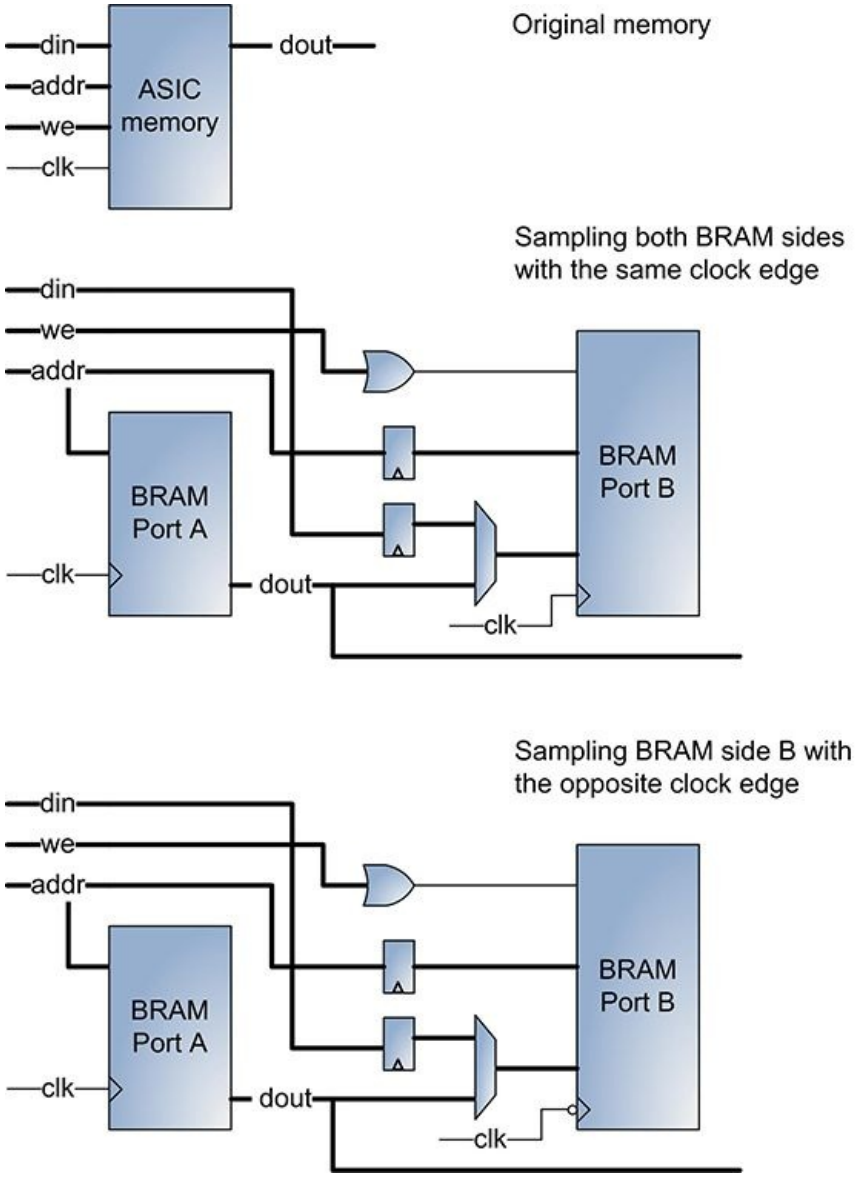

Figure 1: Implementing bit-write enable using BRAM

The dual-port BRAM can pipeline the write operation and achieve a throughput of one read-modify-write operation per clock cycle. To do so, the designer uses Port A as the read port, Port B as the write port, and uses one common clock for both ports. The read address is routed to Port A. A copy of the read address is delayed by one clock and routed to Port B. The data from Port A is multiplexed with the input data and write-enable bus, and used as the data input to Port B.

A similar solution is to implement read-modify-write operation on rising and falling edge of the clock. Rising clock edge is fed to Port A and delay registers. Port B is clocked by the falling clock edge.

The disadvantage of this method is the extra logic required to delay address and data busses, and multiplex the data input to Port B.

A simple design accompanied with the book provides an example of those two methods.

Xilinx BRAM and distributed memory primitives can be used to model a ROM. Methods of preloading ROM with a custom pattern are described in Tip #34. Small size ROMs can be modeled directly in the RTL using Verilog *case* statement.

Large ASIC SRAM and ROM memories can be modeled using off-the-shelf components. However, the disadvantage is decreased design performance and additional design complexity due to the need to interface with additional on-board chips.

54. Porting Tri-state Logic

Tri-state logic is used extensively in ASIC designs to implement bidirectional busses. Xilinx FPGA architecture doesn't support internal bidirectional signals. It only supports tri-stated output for the external IOs. However, in many cases XST and other Xilinx FPGA synthesis tools will correctly synthesize the original RTL by performing conversion of the tri-stated signals into combinatorial logic.

The following is a Verilog example that models a tri-stated output buffer.

```
assign tri_out = out_en ? data_out : 1'bz;
```

XST infers the tri-stated output buffer as OBUFT primitive.

```
OBUFT tri_out_OBUFT (
   .I(data_out),
   .T(out_en),
   .O(tri_out));
```

The XST synthesis report contains the following log:

```
=========================================================
*  HDL Synthesis
=========================================================
Synthesizing Unit <tristates>.
   Found 1-bit tristate buffer for signal <tri_out>
   Summary:
     inferred   1 Tristate(s).
=========================================================
HDL Synthesis Report

Macro Statistics
 # Tristates                      : 1
 1-bit tristate buffer            : 1
=========================================================
```

The following is Verilog example of an internal signal driven by two tri-state buffers.

```
module tristates( input clk,reset,
              input [1:0] d1_in,
              input d1_en,
              input [1:0] d2_in,
              input d2_en,
              output reg dout);

   tri tri_internal;

   assign tri_internal = d1_en ? |d1_in : 1'bz;
   assign tri_internal = d2_en ? ^d2_in : 1'bz;

   always @(posedge clk, posedge reset) begin
      if(reset)
         dout <= 0;
      else
         dout <= tri_internal;
   end
endmodule // Tristates
```

XST synthesizes the following circuit, which includes a register and a look-up table:

```
wire tri_internal;

  FDC dout (
      .C(clk),
      .CLR(reset),
      .D(tri_internal),
      .Q(dout));
```

```
  LUT6 #( .INIT ( 64'hFF7DFF7DFF7D287D ))
     tri_internalLogicTrst1 (
    .I0(d2_en),
    .I1(d2_in[0]),
    .I2(d2_in[1]),
    .I3(d1_en),
    .I4(d1_in[0]),
    .I5(d1_in[1]),
    .O(tri_internal));
```

The XST synthesis report contains the following log:

```
=========================================
* HDL Synthesis
=========================================
Synthesizing Unit <tristates>.
    Found 1-bit register for signal <dout>.
    Found 1-bit tristate buffer for signal <tri_internal>
    Summary:
 inferred   1 D-type flip-flop(s).
 inferred   2 Tristate(s).
Unit <tristates> synthesized.
=========================================
HDL Synthesis Report

Macro Statistics
# Registers                             : 1
 1-bit register                         : 1
# Tristates                             : 2
 1-bit tristate buffer                  : 2
# Xors                                  : 1
 1-bit xor2                             : 1
=========================================
*  Low Level Synthesis
=========================================
WARNING:Xst:2039 - Unit tristates: 1 multi-source signal is replaced by logic (pull-up yes): tri_internal.
```

XST has a *TRISTATE2LOGIC* constraint that enables or disables conversion of the tri-stated signals into logic. XST cannot perform the conversion if the tri-stated signal is connected to a black box or to a top-level output.

The Synopsys Synplify synthesis tool provides another option related to tri-stated signals, called “Push Tristates.” When that option is enabled, the tool pushes tri-stated signals through multiplexors, latches, registers, and buffers, and propagates the high-impedance state. The advantage of pushing tri-stated signals to the periphery is improved timing results. The following figure shows a circuit before and after applying “Push Tristates.”

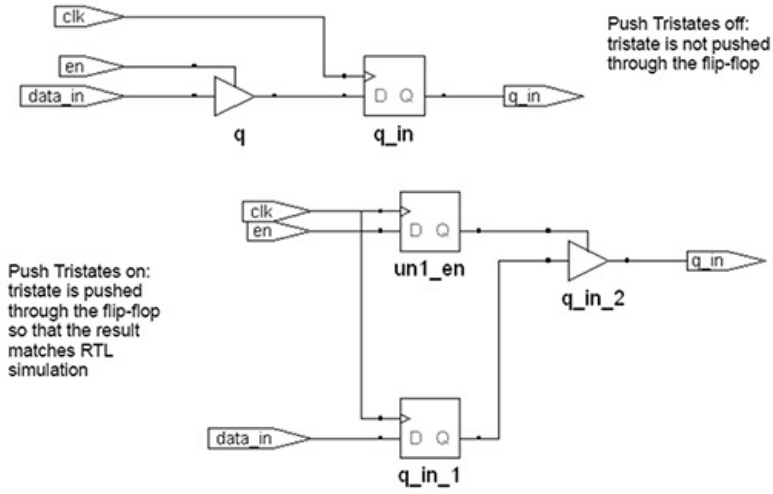

Figure 1: Synplify "Push Tristates" option (source: Synopsys Synplify User Guide)

When a synthesis tool cannot resolve a tri-stated signal, the designer has no choice but to manually change the circuit to the equivalent one without a tri-state.

55. Verification of a Ported Design

After completing all the ASIC migration tasks that require modifications of the original design – partitioning into multiple FPGAs, porting RTL, synthesis and physical implementation - the remaining task is to make sure that the ported design is functionally equivalent to the original one. This is the task of verification, which is intended to test that the design still functions according to the specifications.

A straightforward approach to verify ported design is to subject it to the same array of functional simulation and other tests as the original ASIC design.

The following figure illustrates the process.

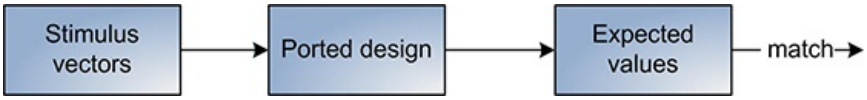

Figure 1: Replacing original ASIC design with the ported one

Simulation stimulus vectors are applied to the ported design instead of the original, and the same expected values are checked. Simulation runtime is expected to be similar to the original ASIC design, even though the ported design did undergo several transformations. This is the disadvantage of this method, because simulation runtime is significant, and can take several days for large ASIC designs.

Another approach shown in the figure below is a side-by-side simulation of the original ASIC and ported designs. The same stimulus vectors are fed into inputs of both designs. Instead of using the module that checks expected values, the outputs of the ported design are compared with the ones from the ASIC. The prerequisite is that the ASIC design is already verified, and produces correct outputs. This approach works well on small designs. It doesn't scale to larger designs in terms of simulation speed, because the size of the ported design is much higher that the module that performs expected value checking.

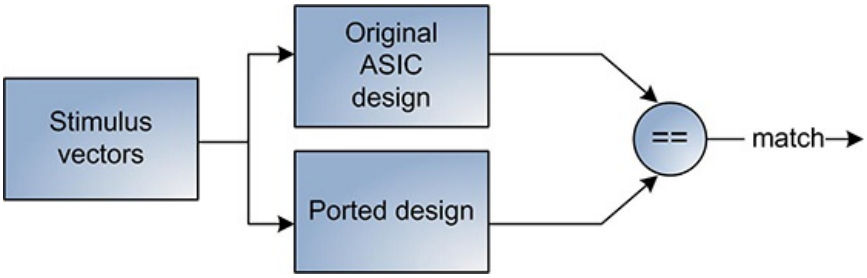

Figure 2: Comparing the results from original and ported designs

If the simulation testbench is synthesizable, the entire system that includes the ported design, synthesizable testbench, stimulus vector generator, and expected values checker, can be run on the emulation or prototyping platform. That approach, also illustrated in the following figure, is much faster than a software-based functional simulation.

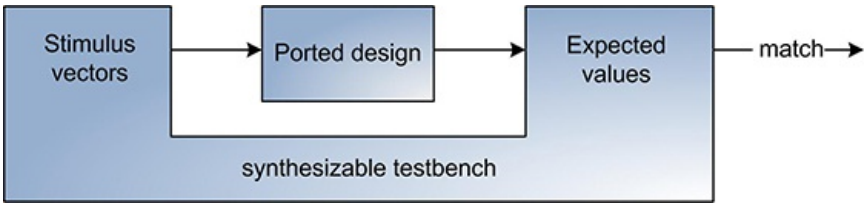

Figure 3: Performing tests on an emulation or prototyping platform

The disadvantage of this method is that it requires rebuilding the system for every code change in order to run it on the emulation or prototyping platform. That can take minutes or several hours, depending on the nature of the change.

Equivalence checking

Equivalence checking is the process of comparing an original, or "golden,"design with the modified one in order to determine that both are functionally equivalent and the modified design still complies with the original specification. The following figure depicts a simplified equivalence checking setup.

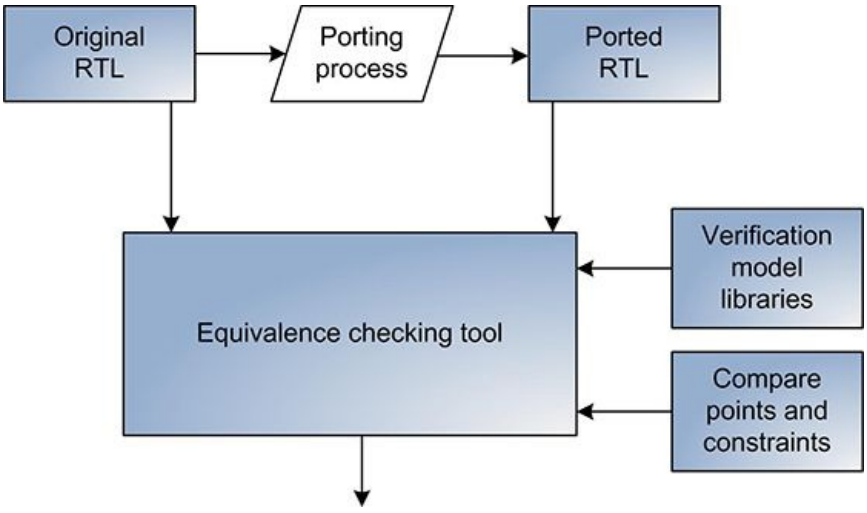

Figure 4: Equivalence checking process

The original ASIC design, the ported design, verification model libraries, compare points, and other constraints are inputs to the third party equivalence checking tool. Compare points describe the primary inputs and outputs, registers and latches, black boxes, and other design components. Constraints include design clock information. Equivalence checking tools typically require synthesizable RTL. Alternatively, the tools can work with a synthesized gate level netlist.

The main advantage of using equivalence checking is that it is faster than functional simulation. However, equivalence checking does not replace functional simulation. Instead, it serves as another method during the design verification process to improve coverage, and increase your confidence level that the ported design functions according to the specification.

There are several commercial equivalence checking tools, such as Cadence Conformal Logic Equivalence Checker (LEC), Mentor Graphics FormalPro, and Synopsys Formality. Each one provides a unique set of features, which might or might not be applicable to a particular design flow used in porting an ASIC design to FPGA.

Using synthesizable assertions

An assertion-based methodology is widely used during the verification process. Assertions augment the functional coverage by providing multiple observation points within the design. Usually, both RTL designers and verification engineers instrument the design with assertions. Assertions can be placed both inside the RTL code, which makes updates and management easier, or outside, to keep synthesizable and behavioral parts of the code separate. SystemVerilog provides assertion specification, a small subset of which is synthesizable assertions. Only a handful of FPGA design tools support synthesizable assertions. One of them is Synopsys Identify Pro that has assertion synthesis and debug capabilities.

56. FPGA Design Verification

Verification is the process of comparing a design's behavior against the designer's intent. Verification of an ASIC design is a broad and complex subject. There are many tools that perform formal and functional verification of an ASIC design. Various verification techniques are topic of academic and industrial research.

However, FPGA verification is still an emerging field. FPGA verification flows and methodologies are not as structured and defined as ASIC. Often FPGA design teams don't have a verification plan at all. Design verification is performed by doing haphazard and disorganized functional simulations without any clear goals.

The gap between ASIC and FPGA verification methodologies is caused by a few reasons.

The cost of an error in FPGA design is not as steep as in ASIC. Frequently, fixing an FPGA bug only requires a new build and reconfiguring the FPGA. "Trial and error" debug methodology lends itself well to being used in small FPGA designs. Unfortunately, that so-called methodology is also the result of a faster time-to-market requirement, at the expense of product quality. By contrast, ASIC teams spend up to 80% of the overall design time on verification.

ASIC design teams are larger than FPGA ones and therefore have more engineering resources to perform more stringent and thorough verification. Also, ASIC projects have bigger budgets and so can afford to buy or license more tools.

Verification is more rigorous in mission-critical FPGA designs, such as medical or military applications, but still it lags behind the ASIC.

Verification process is usually metric-driven, which enables measuring the progress, and being able to determine when the design is ready. Coverage is used as a metric for evaluating the verification progress, and directing verification efforts by identifying tested and untested portions of the design. It is defined as the percentage of verification objectives that have been met.

There are two main types of coverage metrics: (i) those that can be automatically extracted from the design code, such as code coverage; and (ii) those that are user-specified in order to tie the verification environment to the design intent or functionality. The latter is also known as functional coverage.The following figure shows an example of a FPGA system verification flow.

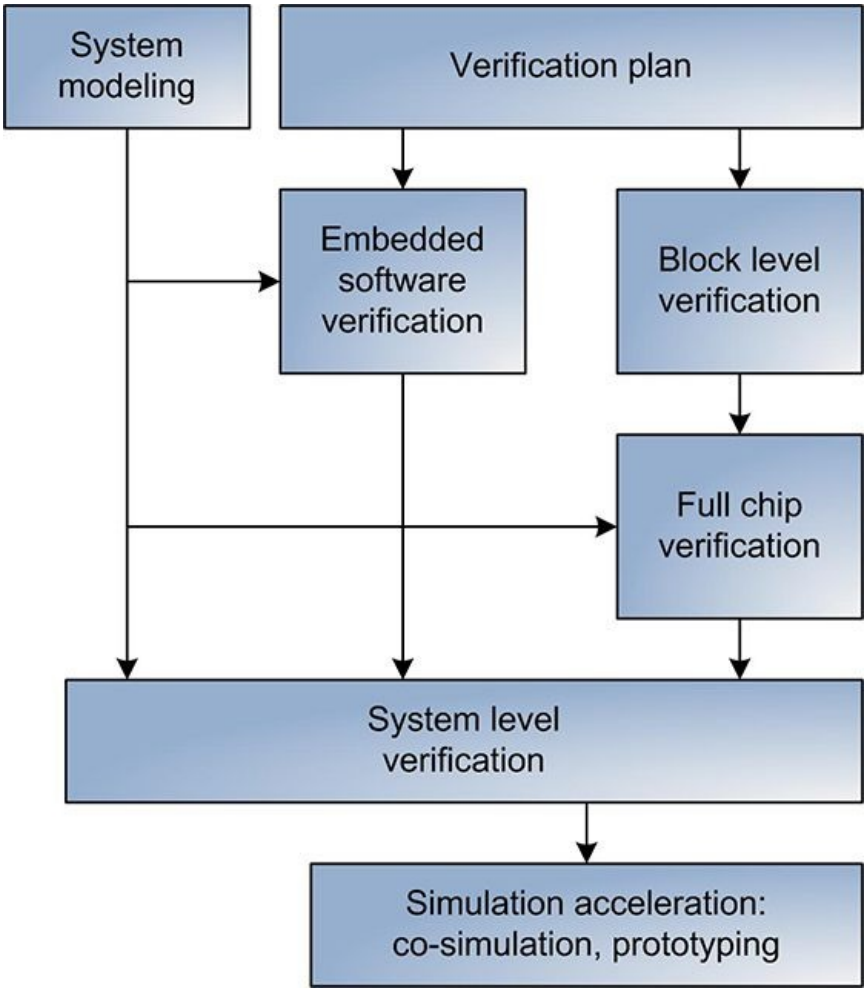

Figure 1: System verification flow plan

A large design can contain several FPGAs and embedded software. Verification of such a design is a system-level effort. The flow plan starts with the System Verification Plan, which defines the scope of work, specifies tasks, selects the tools, and establishes the verification coverage metrics. In parallel, the verification team develops different device and system-level models.

Verification of a digital logic is broken into block-level verification that focuses on simulating individual functional blocks. Next, verification is performed at the full-chip level for individual FPGAs. Finally, there is a system-level verification that combines all FPGAs, embedded software, and system and device models.

System-level verification can be followed by different acceleration methods, such as co-simulation, or system prototyping. By using dedicated prototyping boards and tools, the team can perform system-level verification using real peripheral devices instead of models.

The above flow is very generic, but it can still be applied to different designs: from a small single FPGA board and a single developer, to a board with 10+ FPGAs, embedded software running operating system, and a large cross-functional design team.

Functional Coverage

SystemVerilog provides extensive support for coverage-driven verification. It enables monitoring the coverage of variables and expressions, automatic and user-defined coverage bins with sets of values, transitions, or cross products, filtering conditions at multiple levels, events and sequences to automatically trigger coverage sampling, directives to activate, query, control, and regulate coverage. SystemVerilog includes functional coverage directly in the language by specifying a `covergroup` construct. A `covergroup` specification can include several components that encapsulate the coverage model: a clocking event, a set of coverage points, cross coverage between the coverage points, and various coverage options and arguments. The `covergroup` construct is a user-defined type. It is defined once and can be used in different contexts.

The following example defines coverage group `cover_group1` with three coverage points associated with a pixel position and color. The values of the variable position and color are sampled at the positive edge of signal `clk`.

```
enum { red, green, blue } color;
logic [1023:0] offset_x, offset_y;

covergroup cover_group1 @(posedge clk);
   c: coverpoint color;
   x: coverpoint offset_x;
   y: coverpoint offset_y;
endgroup
```

Code Coverage

Unlike functional coverage, code coverage provides no feedback on the code functional correctness. It provides a measure of completeness of testing and is complementary to functional coverage. Code coverage information is generated by a verification tool from design RTL or gate-level source. There are several types of code coverage described in the following table.

Table 1: Code Coverage Types

Code coverage type	*Description*
Statement coverage	Counts the execution of each statement on a line
Branch coverage	Counts the execution of each conditional “if/then/else” and “case” statement
Condition coverage	Analyzes the decision made in “if” and ternary statements
Expression coverage	Analyzes the expressions on the right hand side of assignment statements
Toggle coverage	Counts each time an object or a variable transitions from one state to another
State machine coverage	Counts the states, transitions, and paths within a state machine

Randomization

Stimulus randomization is an important verification technique. Its formal name is Constrained Random Testing Verification. This methodology allows you to achieve better results by focusing on corner cases when exhaustive coverage is not possible.

The following is an example of a simple 32-bit adder.

```
module adder(input [31:0] a,b,
          output [31:0] sum,
          input clk,reset);

always @(posedge clk)
  if(reset)
   sum <= 'b0;
  else
   sum = a + b;
endmodule // adder
```

Assuming the clock period is 100MHz, it’d take several thousand years to exhaustively simulate the adder:

$$\text{Time} = 2^{32+32} * 10\text{ns} / 31.56\ \text{ms/year} = 5845\ \text{years}$$

In most of the real-world FPGA designs, exhaustive coverage is unachievable. The best known verification approach that produces a good level of confidence is constraint-driven verification. One type of constraint-driven verification is directed method, where a set of known stimulus test vectors is applied to the DUT. Randomization of stimulus data can be more effective method.

There is a broad range of randomization techniques and tools available to verification engineers including broadly and narrowly constrained randomization, different random variable distributions, and others.

Verilog language defines `$random(seed)` system task to do randomization. SystemVerilog provides much richer set of features to perform randomization.

Every SystemVerilog class object has a built-in `randomize()` method. It supports two types of random properties: `rand` and `randc`. `rand` can repeat values without exhausting all of them. `randc` is exhausting all values before repeating any. Randomization is controlled using a `constraint` block. Constraint values can be weighted to follow a specific random distribution.

57. Simulation Types

Designers can perform different simulation types during FPGA development cycle: RTL, functional, gate-level, and timing.

RTL simulation is performed using the original Verilog or VHDL code.

Functional simulation, also known as post-synthesis simulation, is performed using a functional RTL model of the gate level netlist produced by a synthesis tool. Functional gate level RTL models contain simulation models of the FPGA-specific primitives, such as LUTs and registers. The reason for calling it a functional simulation is that it doesn’t contain any technology-specific timing, placement, or routing information about the design.

Gate-level timing simulation is performed on the design after the place and route stage. This is the most accurate simulation type, since it contains all the placement, routing and timing delay information.

RTL simulation is much faster than functional and gate-level timing simulation, because the simulation database is the smallest, and doesn’t contain placement, routing, and timing information of the final FPGA design. This is one reason why RTL simulation is