Xilinx Design Reuse Methodology for ASIC and FPGA Designers

SYSTEM-ON-A-CHIP DESIGNS REUSE SOLUTIONS

Xilinx

An Addendum to the:

REUSE METHODOLOGY MANUAL

FOR SYSTEM-ON-A-CHIP DESIGNS

Table of Contents

1 Introduction

FPGAs have changed dramatically since Xilinx first introduced them just 15 years ago. In the past, FPGA were primarily used for prototyping and lower volume applications; custom ASICs were used for high volume, cost sensitive designs. FPGAs had also been too expensive and too slow for many applications, let alone for System Level Integration (SLI). Plus, the development tools were often difficult to learn and lacked the features found in ASIC development systems. Now, this has all changed.

Silicon technology has progressed to allow chips with tens of millions of transistors. This not only promises new levels of integration onto a single chip, but also allows more features and capabilities in reprogrammable technology. With today's deep sub-micron technology, it is possible to deliver over 2 - million usable system gates in a FPGA. In addition, the average ASIC design operating at 30 – 50MHz can be implemented in a FPGA using the same RTL synthesis design methodology as ASICs. By the year 2004, the state-of-the-art FPGA will exceed 10 million system gates, allowing for multimillion gates FPGAs operating at speeds surpassing 300 MHz. Many designs, which previously could only achieve speed and cost-of-density goals in ASICs, are converting to much more flexible and productive reprogrammable solutions.

The availability of FPGAs in the 1-million system gate range has started a shift of SoC designs towards using Reprogrammable FPGAs, thereby starting a new era of System-on-a-Reprogrammable-Chip (SoRC). For a 1-million system gate SoRC design, an engineer designing 100 gates/day would require a hypothetical 42 years to complete, at a cost of $6 million. Clearly, immense productivity gains are needed to make million gate designs commercially viable, and SoRC based on today's million logic gate FPGAs, and tomorrow's 10-million logic gate FPGAs is a promising solution.

SoRC is no different from SoC in that it requires leveraging existing intellectual property (IP) to improve designer productivity. Reusable IP is essential to constructing bug-free multimillion-gate designs in a reasonable amount of time. Without reuse, the electronics industry will simply not be able to keep pace with the challenge of delivering the "better, faster, cheaper" devices that consumers expect.

With the availability of new FPGA architectures designed for system level integration and FPGA design tools that are compatible with ASIC design methodologies, it is now possible to employ a similar if not identical design reuse methodology for ASICs and FPGAs. For design teams that have longed to eliminate NRE costs and improve time-to-market without climbing the learning curve to become FPGA experts, this design reuse methodology adds a new level of freedom. However, this methodology is not without its limitations. This paper examines the designs that can take advantage of an identical ASIC and FPGA design reuse methodology and the simple modifications that can be made to the RTL code to enhance performance and reuse.

1.1 System-on-a-Reprogrammable Chip

To define SoRC, let us first start with a general definition of System-on-a-Chip (SoC). Most of the industry agrees that SoC is the incorporation of an entire system onto one chip. Dataquest's 1995 definition included a compute engine (microprocessor, microcontroller or digital signal processor), at least 100K of user gates and significant on-chip memory.

The definition of SoRC is just beginning to evolve and in this book is defined as system level integration implemented on a Reprogrammable FPGA device. The SoRC definition is similar to SoC since it generally includes a compute engine, 50K of user logic gates and on-chip memory. The definition of SoRC includes partial system integration into multiple chips as well as entire system level integration on a single chip. The challenges facing SoC and SoRC designers are similar even if the entire system is not integrated into one single chip. However SoRC is not defined as the gathering of glue logic, since SoRC designs contain system level integration issues that separate it from a general glue logic design.

1.2 Why Use an FPGA?

System-Level-Integration (SLI) using reprogrammable FPGA technology is made possible by advances in IC wafer technology especially in the area of deep submicron lithography. Today, state-of-the-art waferfabs find FPGAs an excellent mechanism for testing new wafer technology because of their reprogrammable nature. Incidentally, this trend in the wafer fabs means that FPGA companies have early access to the newest deep sub-micron technologies, dramatically increasing the number of gates available to designers as well as reducing the average gate cost sooner in the technology life-cycle than before. This trend, together with innovative system level architecture features, is leading FPGAs to become the preferred architecture for SLI.

1.2.1 ASIC vs. FPGA Design Flows

Figure 1 illustrates a typical ASIC design flow as compared to a typical FPGA design flow. The FPGA design flow has some noticeable advantages:

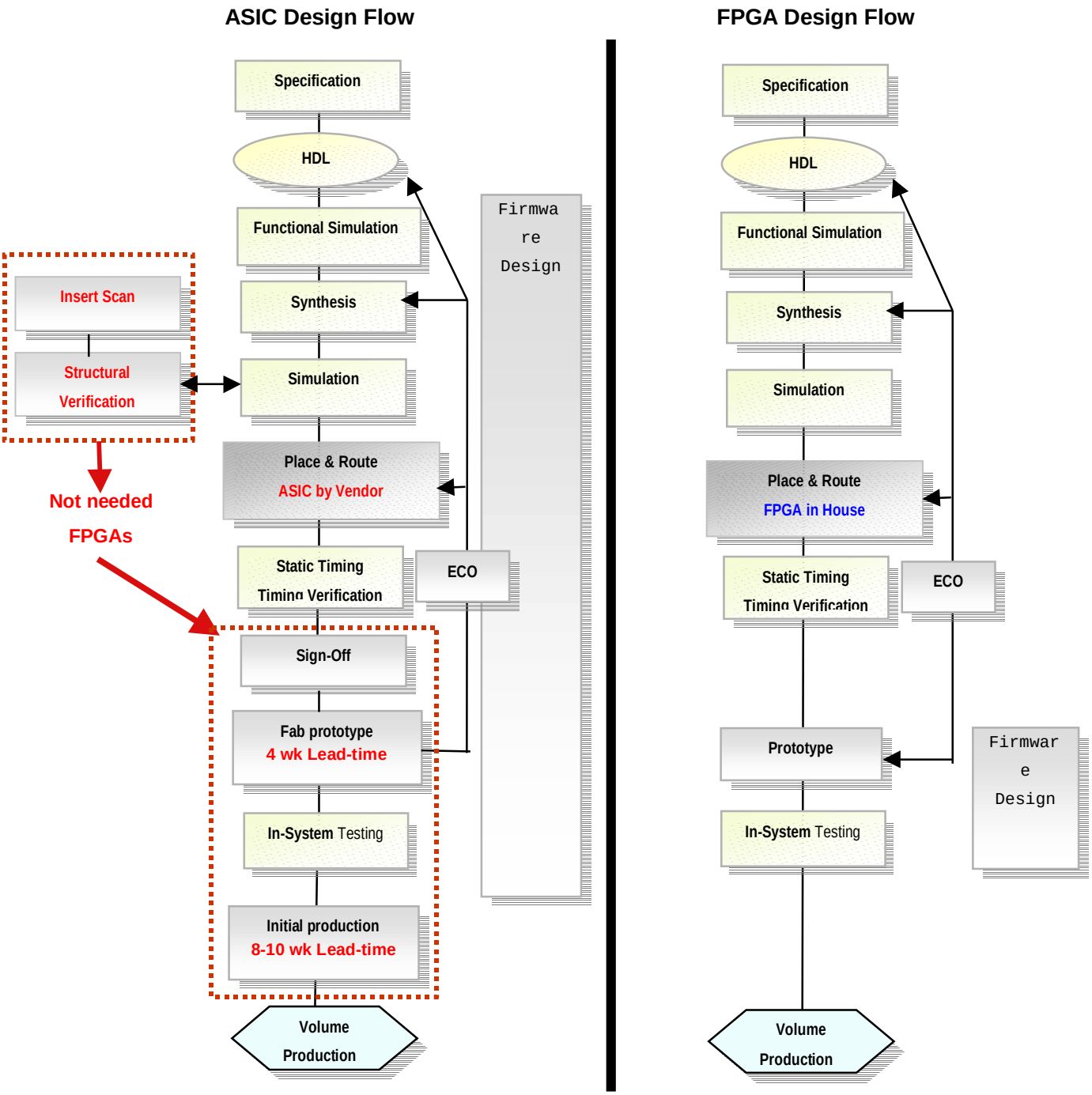


Figure 1 – ASIC Design Flow Compared to FPGA Design Flow

Reduced Risk – System-Level-Integration increases the complexity of implementation into the target device. Reprogrammable logic eliminates the risk and expense of semi-custom and custom IC development by providing a flexible design methodology. Systems designed in FPGAs can be prototyped in stages, allowing in-system testing of individual sub-modules. The design engineer can make design changes or ECOs in minutes avoiding multiple cycles through an ASIC manufacturing house, at 2 months per cycle.

The quality of IP for ASICs depends on whether it has been verified using a specific ASIC technology library. The ultimate proof of the quality of the IP is that it has been implemented in a desired ASIC technology. Implementing an IP or reuse module in more than one ASIC technology is costly. However implementing IP or reusable modules in a variety of FPGA devices is not.

Faster Testing and Manufacturing - ASICs require rigorous verification strategies to avoid multiple cycles through manufacturing. The manufacturing and test strategies must be well defined during the specification phase. Often different strategies must be used depending on the type of block. Memory blocks often use BIST or some form of direct memory access to detect and troubleshoot data retention problems. Other logic such as microprocessors requires custom test structures for full or partial scan or logic BIST.

FPGA designs, on the other hand, are implemented on reprogrammable devices that are 100% tested by the manufacture, before reaching the designer. In fact, FPGAs can be used to create test programs downloadable into other non-reprogrammable devices on the board. There is no non-recurring engineering (NRE) cost, no sign-off, and no delay while waiting for prototypes to be manufactured. The designer controls the entire design cycle, thereby shrinking the design cycle as well as the time to prototype. This allows essential steps such as firmware designing to occur at a stage late in the design flow but actually earlier in the actual design time.

Verification – ASIC technology requires strict verification before manufacturing. In large complex system level designs many more unexpected situations can occur. SoRC designers have a much more flexible in-system verification strategy. The designers can mix testbench verification strategies with in-circuit testing, thereby offering faster verification without the cost of accelerated simulators. Surveys of design engineers have found that the area of test vector generation and vector-based verification is the least favorite part of system design process.

Hardware and Software Co-Designing - Early versions of the systems prototype can be used to facilitate software and hardware co-design.

1.3 A Common Design Reuse Strategy

The dramatic improvement in FPGA architecture, pricing and design tools in the past few years has made it possible for ASIC and FPGA designers to share a common design methodology. Designs requiring performance in the 30 – 50 MHz range are usually implemented using a RTL synthesis design methodology making a common ASIC and FPGA design reuse strategy possible. However, designs requiring higher performance, will usually require additional techniques unique to the FPGA environment. A common design and design reuse strategy provides the flexibility to choose the best method to implement a system design without the overhead of retraining the design teams. In addition, one can take advantage of the design rules and guidelines found in reuse methodology manuals such as the *Reuse Design Methodology Manual* from Synopsys and Mentor or the web-based *Reuse Methodology Field Guide* from Qualis.

There are many challenges facing Soc/SoRC designers such as time-to-market pressures, quality of results, increasing chip complexity, varying levels of expertise, multi-site teams and management and implementation of a reuse strategy. FPGA designers are faced with the additional challenges of architectures with varying system features, meeting difficult performance goals and different implementation strategies. This paper addresses these unique challenges by showing how the guidelines found in the *Reuse Design Methodology Manual* can be applied

effectively on SoRC designs and by focusing on some additional guidelines that can further enhance performance and reusability for designers implementing a common reuse strategy.

- **Section 2** provides an overview of the system level features commonly found in the FPGA architectures designed for SLI. Xilinx's Virtex is a leading example.
- **Section 3** contains general RTL synthesis guidelines that apply to both ASIC and FPGA implementations, and have the greatest impact on improving system performance
- **Section 4** is a brief discussion of FPGA verification strategies and trends.

1.4 Definitions & Acronyms

This manual uses the following terms interchangeably: Macro, Module, Block, IP and Core. All of these terms refer to a design unit that can reasonably be viewed as a stand-alone sub-component of a complete System-on-a-Reconfigurable-Chip design.

Acronyms

- **CLB** – Combinatorial Logic Block
- **ESB** – Embedded Systems Block
- **FPGA** – Field Programmable Gate Array
- **HDL** – Hardware Description Language
- **LE** – Logic Element
- **OHE** – One Hot Encoded
- **RTL** – Register Transfer Level
- **SLI –** System-Level-Integration
- **SoC –** System-on-a-Chip
- **SoRC -** System-on-a-Reprogrammable-Chip System

2 System Level Reuse Issues for FPGAs

This section gives an overview of the system-level issues that are unique to FPGAs when designing for reuse. Generally these elements must be agreed upon or at least discussed to some level before starting to design the modules of the system.

2.1 System Synthesis and Timing Issues

Designers should follow the guidelines for synchronous design style, clocking and reset found in the “Reuse Methodology Manual” by Synopsys and Mentor, “Synthesis and Simulation Design Guide” by Xilinx or the Design Guides supplied by suppliers of EDA tools for the targeted FPGA.

2.1.1 Synchronous vs. Asynchronous Design Style

Rule – Avoid using latches. The system should be synchronous and register-based. Use D registers instead of latches. Exceptions to this rule should be made with great care and must be fully documented.

In the past latches have been popular for designs targeting ASICs. Although latches are a simpler circuit element then flip-flops, they add a level of complexity to the design such as ambiguous timing. Experienced designers maybe able to take advantage of the ambiguity to improve timing. Time borrowing is used to absorb some of the delay by guaranteeing that the data is set up before the leading clock edge at one stage or allowing the data to arrive as late as one setup time before the trailing clock edge at the next stage.

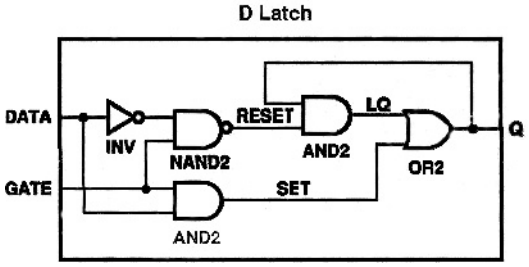
Example 1 D Latch Implemented with Gates

VHDL:

```
architecture BEHAV of d_latch is
begin
LATCH: process (GATE, DATA)
   begin
If (GATE= ‘1’) then
   Q <= DATA;
      end if;
end process; -- end latch
```

Verilog:

```
always @ (GATE or DATA)
begin
   if (GATE == 1’b1)
      Q<= DATA;
end
```



XSI page 2-28 – X4975
Figure 2 - D Latch Implemented with Gates

The problem caused by the ambiguity of latch timing, and exacerbated by time borrowing, is that it is impossible by inspection of the circuit to determine whether the designer intended to borrow time or the circuit is just slow. In the example of the D Latch implemented using gates, a combinatorial loop results in a hold-time requirement on DATA with respect to GATE. Since most synthesis tools for FPGAs do not process hold-time requirements because of the uncertainty of routing delays, it is not recommended to implement latches with combinatorial feedback loops. Whether the latch is implemented using the combinatorial gates or logic blocks, the timing analysis of each latch of the design is difficult. Over a large design, timing analysis becomes impossible. Only the original designer knows the full intent of the design. Thus, latch-based design is inherently not reusable.

In FPGA architectures another danger of inferring a latch from RTL code is that very few FPGA devices have latches available in the logic blocks internal to the device. In addition, in FPGA architectures that do have latches available in the internal logic blocks, the function of the latches can vary from one architecture to the next. HDL compilers infer latches from incomplete conditional expressions, such as an If statement without an Else clause. If the FPGA architecture does not have latch capabilities, the compiler may report an error or may implement the latch using gates in the logic blocks function generator. If the architecture has latches available in the device's input pin/pad and the latches are connected to an input port, the HDL compiler may implement the latch inside the input pad. Once again, only the original designer knows the full intent of the desired implementation.

2.1.3 System Clocking and Clock Distribution

In system level designing, the generation, synchronization and distribution of the clocks is essential. Special care has long been taken by ASIC designers in the layout of the clocking structure. FPGAs have an advantage in this area since the clocking mechanisms are designed into the device and pre-tested, balancing the clocking resources to the size and target applications of the device. This section takes a look at the clocking resources available to reduce and manage the impact of clock skew and clock delay.

SoC vs. SoRC – SoC clocking distribution methods such as building a balanced clock tree to distribute a single clock throughout the chip is not recommend for FPGAs designs. FPGAs have dedicated clocking distribution resources and methods that efficiently utilize resources to provide high fanout with low skew throughout the chip. These clocking resources are roughly equivalent to the high-power clock buffers found in SoC designs.

System Clocking

Rule – The design team must decide on the basic clock distribution architecture for the chip early in the design process. The most efficient clocking strategy and clock distribution will most likely be determined by the targeted architecture. This strategy must be well documented and conveyed to the entire design team.

FPGA architectures provide high-speed, low-skew clock distributions through dedicated global routing resources. FPGAs designed for SoRC on average provide 4 dedicated global clock resources and additional clocking resources through flexible low-skew routing resources. The dedicated resources for clocking distributions consist of dedicated clock pads located adjacent to global nets that in turn can drive any internal clocked resources. The input to the global buffer is selected either from the dedicated pads or from signals generated internal to the FPGA.

Rule – Keep the number of system clocks equal to or less than the number of dedicated global clock resources available in the targeted FPGA.

As an FPGAs design grows in size, the quality of on-chip clock distribution becomes more important. FPGA architectures designed for SoRC generally employ a clocking method to reduce and manage the impact of clock skew and clock delay. FPGA architectures for SoRC provide either a dedicated Phase-Locked Loop (PLL) or Delay-Locked Loop (DLL) circuit. These circuits not only remove clock delay but can also provide additional functionality such as frequency synthesis (clock multiplication and clock division) and clock conditioning (duty cycle correction and phase shifting). Designers can also use the multiple clock outputs, deskewed with respect to one another, to take advantage of multiple clock domains.

Rule – The number of clock domains and the clock frequency must be documented as well as the required frequency, associated PLL or DLL and the external timing requirements (setup/hold and output timing) needed to interface to the rest of the system.

Delay-Locked Loop (DLL) vs. Phase-Locked Loop (PLL)

Either a phase-locked-loop (PLL) or a delay-locked-loop (DLL) can be used to reduce the on-chip clock-distribution delay to zero. Both can be considered a servo-control system since they use a feedback loop.

A PLL uses a phase detector to drive a voltage-controlled oscillator (VCO) such that the VCO output has the desired frequency and phase. This generally involves an analog low-pass filter and an inherently analog VCO. A PLL can recover a stable clock from a noisy environment, but it is difficult to avoid generating random clock jitter.

A DLL uses a phase detector to increase the delay so much that the subsequent clock edge occurs at the desired moment. This generally involves a multi-tapped delay line, consisting of a large number of cascaded buffers. The adjustment is done by a digitally controlled multiplexer. This scheme lends itself to a totally digital implementation and is more compatible with standard circuit design methodology and IC processing. A DLL cannot suppress incoming clock jitter, passing the jitter straight through. There is no random output jitter, but there is a systematic output jitter of one delay-line increment, typically less than 50 picoseconds.

Neither a PLL nor a DLL can be used in PCI-designs that demand proper operation at instantly changing clock rates.

Delay-Locked Loop (DLL)

As shown in Figure 3, a DLL in its simplest form consists of a programmable delay line and some control logic. The delay line produces a delayed version of the input clock CLKIN. The clock distribution network routes the clock to all internal registers and to the clock feedback CLKFB pin. The control logic must sample the input clock as well as the feedback clock in order to adjust the delay line.

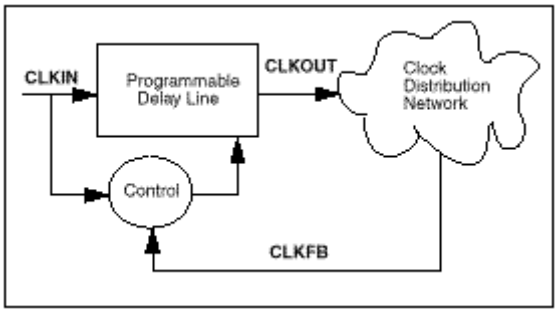


Figure 3 - A Delay-Locked Loop Block Diagram

In the example of a Xilinx Virtex architecture each global clock buffer is a fully digital DLL. Each DLL can drive two global clock networks. The DLL monitors the input clock and the distributed clock, and automatically adjusts a clock delay element.

A DLL works by inserting delay between the input clock and the feedback clock until the two rising edges align, putting the two clocks 360 degrees out of phase (effectively in phase). After the edges from the input clock line up with the edges from the feedback clock, the DLL “locks”. After the DLL locks, the two clocks have no discernible difference. Thus, the DLL output clock compensates for the delay in the clock distribution network, effectively removing the delay between the source clock and its loads. This ensures that clock edges arrive at internal flip-flops in synchronism with each clock edge arriving at the input.

FPGAs often uses multiple phases of a single clock to achieve higher clock frequencies. DLLs can provide control of multiple clock domains. In our example architecture, four phases of the source clock can be doubled or divided by 1.5, 2, 2.5, 3, 4, 5, 8 or 16.

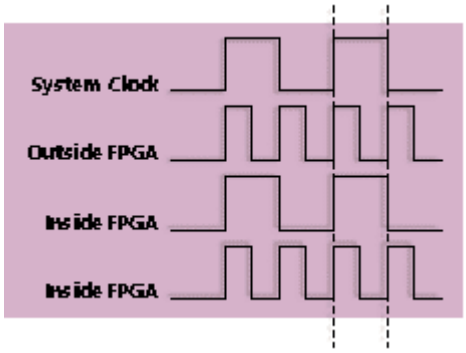


Figure 4 - Zero Delay Clock Management. Multiple DLLs facilitate precise generation of zero-delay clocks both inside and outside the FPGA for highest chip-to-chip speeds

Phase-Locked Loop

While designed for the same basic functions, the PLL uses a different architecture to accomplish the task. As shown in Figure 5, the fundamental difference between the PLL and DLL is that, instead of a delay line, the PLL uses a programmable oscillator to generate a clock signal that approximates the input clock CLKIN. The control logic, consisting of a phase detector and filter, adjusts the oscillator phase to compensate for the clock distribution delay.

The PLL control logic compares the input clock to the feedback clock CLKFB and adjusts the oscillator clock until the rising edge of the input clock aligns with the rising edge of the feedback clock. The PLL then “locks”. The Altera FLEX 20KE is an example of a FPGA architecture that contains a clock management system with phase-locked lock (PLL).

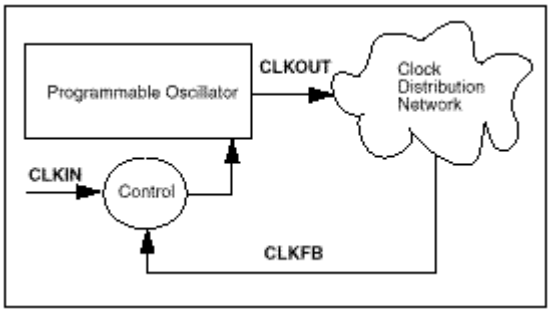


Figure 5 - Phase-Locked Loop Block Diagram

Guideline – If a phase-locked loop (PLL) is used for on-chip clock generation, then some means of disabling or bypassing he PLL should be provided. This bypass makes chip testing and debug easier.

2.2 Memory and Memory Interface

Memories present a special challenge when designing for reuse. In FPGA designs, memories are generally designed using vendor-supplied modules or module generators, making them very technology dependent. Memory compilers developed for ASICs are not currently designed to target FPGA architectures. However, some synthesis tools can recognize RAM from RTL code, making the design synthesis-tool dependent and FPGA-vendor independent.

2.2.1 On-Chip Memory

FPGAs architectures can accommodate small to medium blocks of memory on-chip. Smaller on-chip RAM and ROM can be distributed throughout the FPGA by configuring the logic function generators into bit-wide and byte-deep memory (i.e., 16x1, 16x2, 32x1, and 32x2). Distributed

RAM can be used for status registers, index registers, counter storage, constant-coefficient multipliers, distributed shift registers, FIFO or LIFO stacks, or any data storage operation. Dual port RAM simplifies the designs of FIFOs. The capabilities of these distributed blocks of memory are highly architecture dependent and must be documented to ensure a compatible architecture is chosen when the module is reused. Distributed memory generally supports level-sensitive, edge-triggered, dual and single port RAM. The edge-trigger capability simplifies system timing and provides better performance for distributed RAM-based design.

Medium size memory can utilize block memory structures of the FPGA architecture. These block memories complement the shallower distributed RAM structures and are generally organized into columns in the device. The columns extend the height of the chip. In the example of the Xilinx Virtex device, each block memory is four CLBs high. A Virtex device of 64 CLBs high will contain 16 memory blocks per column with a total of 32 blocks in two columns. The 1 million system gate (or ~350K logic gates) Virtex device has a total of 131,072 bits of block RAM available. The depth and width ratio are adjustable between 1 x 4096 to 16 x 256 (width x depth). Dedicated routing resources are provided to ensure routability and performance.

Implementing distributed or block memory can be performed in three different ways:

- RTL description
- Instantiation of primitives
- Vendor specific memory compiler

Guideline – A corporate reuse strategy, that standardizes on a synthesis tool or FPGA architecture. However, standardizing on a tool or architecture may hinder design reuse.

Guideline – If a corporate policy that standardizes on a synthesis tool, implementing distributed memory through the RTL description is generally recommended if the synthesis tool supports memory interfacing. The specific RTL coding style to infer a distributed or block memory is unique to each synthesis vendor and not all synthesis tools have memory inference capabilities for FPGAs devices.

Alternatively, distributed and block memory can be implemented using a vendor-specific memory compiler or through instantiation. Memory compilers and instantiation of memory primitives may provide access to features that can not be synthesized from a RTL description. If a memory compiler is used, it must be clearly specified in the script file and the compiler used must be document. Both memory compilers and instantiation generally require additional commands in the synthesis script file. Using a memory compiler requires that a “black box” be instantiated into the hierarchical design. Special commands are added to the synthesis script to ensure that the component is not compiled and that the design file can be located. Instantiation requires that commands be added to assign the ROM values and the initial RAM value.

Guideline – If a corporate policy is to standardize on a FPGAs device family or a family series that is backwards compatible, use of the FPGA vendor’s memory compiler is recommend.

2.2.2 Interfacing to Large Memory Blocks

Low-volume designs and prototypes can take advantage of customized solutions targeted at specific markets, such as Triscent’s CPSU family of devices. These solutions combine CPU, FPGA and larger blocks of SRAM for system-level integration targeting microcontroller-based

systems that contain from 16 to 64kbyets of SRAM with 8-bit processor cores and 40K FPGA system gates.

Standalone memories can provide designers with solutions when large blocks of storage are needed for caches, buffers, and large look-up tables, such as in networking applications. However the trend in SoC designs implemented as ASICs has been towards many smaller blocks of SRAM for local buffering, register files, and temporary storage.

SoRC vs. SoC - SoRC architectures are generally designed with small to medium memory capabilities. Standard cell devices can embed large blocks of high performance memory on-chip.

High-speed SRAM (e.g. 350MHz) with features such as double-data-rate (DDR) I/O capabilities and zero-bus latencies and very large, multiple Gigabyte memories are best left off-chip in both SoC and SoRC devices. Even half a megabit or more of SRAM or several megabytes of DRAM, is more cost-effective when implemented off-chip.

Most SoRC FPGA devices have banks of I/Os that can be configured to interface efficiently to high speed SRAM and synchronous DRAM. The market is shifting away from 5-V devices to chips that are operate from 3.3V supplies and offer 3.3V LVTTL (low-voltage TTL) interfaces rather than standard TTL or CMOS I/O levels. The voltage is continuing to drop as signal swings are reduced to improve access time and power dissipation. SRAMs are offering 2.5V I/O lines that meet the HSTL (high-speed subterminated logic) interface specifications. These higher-speed I/O lines will allow bus operations well beyond 300MHz.

SoC vs. SoRC - An advantage of designing a system using FPGA technology is that the FPGA vendor has already invested the resources to support various I/O standards. As a result, a designer can develop and prototype with a wide range of interface standards.

2.3 External Operability (I/O Standards)

Complex, system level chips require a variety of I/O interface standards. These different I/O standards provide higher levels of performance and/or lower power dissipation and are optimized for system-critical elements such as backplane, memory and communication systems. High-speed applications such as 66 MHz PCI require high-speed input and output capabilities. One obvious trend for system-level FPGA architectures is to add the necessary I/O buffers into the device to improve the overall system performance, reduce the board size, reduce cost, simplify the design and provide full high-speed access to other devices.

Rule – When designing with a reusable module, choose a SoRC device that supports the required I/O standards.

Rule – Any module design for reuse that contains I/O should take advantage of the variety of I/O standards provided in the selected FPGA architecture. It is important to document the I/O standards required and any specific feature of the SoRC device's I/O that was used in the initial implementation of the sub-module.

Verify that the selected architecture protects all pads from electrostatic discharge (ESD) and from over-voltage transients. Having IEEE 1149.1-compatible boundary scan test capabilities available in the I/O blocks can enhance board level testing.

Table 1 - Example of different I/O standards

Standard	Voh	Vref	Definition	Application
LVTTL	3.3	na	Low-voltage transistor-transistor logic	General purpose
LVCMOS2	2.5	na	Low-voltage complementary metal-oxide	General purpose
PCI 33MHz 3.3V	3.3	na	Personnel computer interface	PCI
PCI 33MHz 5.0V	3.3	na	Personnel computer interface	PCI
PCI 66MHz 3.3V	3.3	na	Personnel computer interface	PCI
GTL	na	0.80	Gunning transceiver logic	Backplane
GTL+	na	1.00	Gunning transceiver logic	Backplane
HSTL-I	1.5	0.75	High-speed transceiver logic	High Speed SRAM
HSTL-III	1.5	0.90	High-speed transceiver logic	High Speed SRAM
HSTL-IV	1.5	0.75	High-speed transceiver logic	High Speed SRAM
SST3-I	3.3	0.90	Stub-series terminated logic	Synchronous DRAM
SST3-II	3.3	1.50	Stub-series terminated logic	Synchronous DRAM
SST2-I/II	2.5	1.25	Stub-series terminated logic	Synchronous DRAM
AGP	3.3	1.32	Advanced graphics port	Graphics
CTT	3.3	1.5	Center tap terminated	High Speed Memory

I/Os on SoRC devices are often grouped in banks. The grouping of I/O's into these banks are generally placed into a module and can affect the floorplanning of the SoRC design.

Guidelines – Document the grouping of I/O into the device's banks and note any reason for constraining the module or I/O in the modules to a particular area or pin location for reasons such as Global Clock Buffers or DLL direct connections.

3 Coding and Synthesis Tips

Fine-grain ASIC architectures have the ability to tolerate a wide range of RTL coding styles while still allowing designers to meet their design goals. Course-grain FPGA architecture like Xilinx's Virtex and Altera's Apex are more sensitive to coding styles and design practices. In many cases, slight modifications in coding practices can improve the system performance anywhere from 10% to 100%. Design reuse methodologies already stress the importance of good coding practices to enhance reusability. Today, IP designers are utilizing many of these practices, as described in the *Reuse Methodology Manual*, resulting in modules that perform much faster in FPGAs than traditional ASIC designs converting to FPGAs.

The most common reason why a given design runs much slower in a FPGA compared to an ASIC is an excessive number of logic levels in the critical path. A logic level in a FPGA is considered to be one Combinatorial Logic Block (CLB) or Logic Element (LE) delay. In the example of a CLB, each CLB has a given throughput (alt. propagation?) delay and an associated routing delay. Once the amount of logic that can fit into one CLB is exceeded, another level of logic delay is added. For example, a module with 6 to 8 FPGA logic levels would operate at ~50MHz. This course-grain nature of FPGA may yield a higher penalty for added logic levels than with ASICs.

This section covers some of the most useful hints to enhance speed through reducing logic levels for FPGA SRAM architectures.

3.1 Abundance of Registers

FPGA architectures are generally register-rich. RTL coding styles that utilize registers can be employed to dramatically increase performance. This section contains several coding techniques that are known to be effective in increasing performance by utilizing registers.

3.1.1 Duplicating Registers

A technique commonly used to increase the speed of a critical path is to duplicate a register to reduce the fan-out of the critical path. Because FPGAs are register-rich, this is usually an advantageous structure since it can often be done at no extra expense to the design.

Example 2 – Verilog Example of Register with 64 Loads

```
module high_fanout(in, en, clk, out);
        input    [63:0]in;
        input    en, clk;
        output   [63:0] out;
        reg      [63:0] out;
        reg      tri_en;
        always @(posedge clk)  tri_en = en;
        always @(tri_en or in)  begin
                 if (tri_en)  out = in;
                 else  out = 64'bZ;
        end
endmodule
```

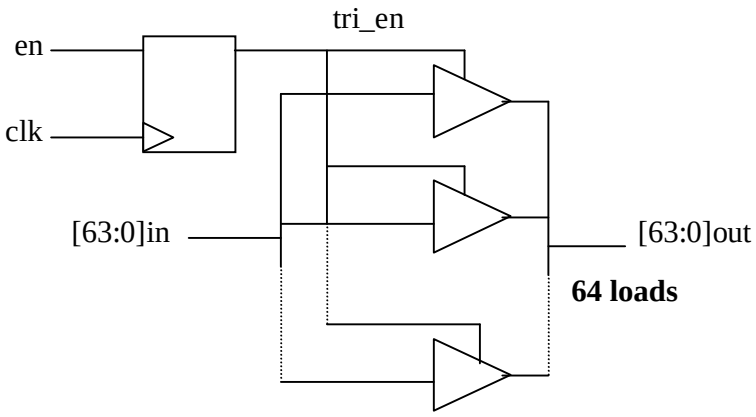


Figure 6 – Register with 64 Loads

Example 3 – Verilog Example of After Register Duplication to Reduce Fan-out

```
module low_fanout(in, en, clk, out);
        input    [63:0] in;
        input    en, clk;
        output   [63:0] out;
        reg      [63:0] out;
        reg      tri_en1, tri_en2;
        always @(posedge clk) begin
                 tri_en1 = en;  tri_en2 = en;
        end
        always @(tri_en1 or in)begin
                 if (tri_en1)  out[63:32] = in[63:32];
                 else  out[63:32] = 32'bZ;
        end
        always @(tri_en2 or in) begin
                 if (tri_en2) out[31:0] = in[31:0];
                 else out[31:0] = 32'bZ;
        end
endmodule
```

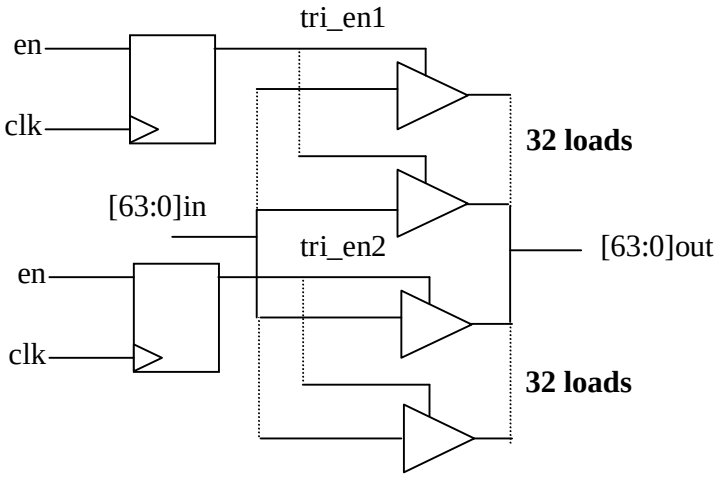


Figure 7 - Register Duplication to Reduce Fan-out

3.1.2 Partitioning at Register Boundary

Guideline - For large blocks, both inputs and outputs should be registered. For smaller modules either the input or the output of the module should be registered. Registering both the input and output makes timing closures within each block completely local. Internal timing has no effect on the timing of primary inputs and outputs of the block. The module gives a full clock cycle to propagate outputs from one module to the input of another.

Unlike ASICs, there is no need for buffers to be inserted at the top level to drive long wires since FPGA architectures designed for systems have an abundant amount of global routing with built in buffering.

This kind of defensive timing design is useful for large system level designs as well as reusable modules. In a reusable block, the module designer does not know the timing context in which the block will be used. Defensive timing design is the only way to assure that timing problems will not limit future use of the module.

3.1.3 One-Hot State Machines

State machines are one of the most commonly implemented functions in system level designs. Highly encoded state sequences will generally have many, wide-input logic functions to interpret the inputs and decode the states. When implemented in a FPGA this can result in several levels of logic between clock edges because multiple logic blocks are needed to decode the states.

Guideline - A better state-machine approach for FPGAs limits the amount of fan-in into one logic block. In some cases a binary encoding can be more efficient in smaller state machines.

The abundance of registers in FPGA architectures and the fan-in limitations of the CLB tend to favor a one-hot-encoding (OHE) style. The OHE scheme is named so because only one state register is asserted, or “hot”, at a time. One register is assigned to each state. Generally an OHE scheme will require two or fewer levels of logic between clock edges compared to binary encoding, translating into faster performance. In addition the logic circuit is simplified because OHE removes much of the state-decoding logic. An OHE state machine is essentially already fully decoded making verification simple. Many synthesis tools have the ability to convert state machines coded in one style to another.

3.1.4 Pipelining

Pipelining can dramatically improve device performance by restructuring long data paths with several levels of logic and breaking them up over multiple clocks. This method allows for a faster clock cycle and increased data throughput at small expense to latency from the extra latching overhead. Because FPGAs are register-rich, this is usually an advantageous structure for FPGA design since the pipeline is created at no cost in terms of device resources. However, since the data is now on a multi-cycle path, special considerations must be used for the rest of the design to account for the added path latency. Care must be taken when defining timing specifications for these paths. The ability to constrain multi-cycle paths with a synthesis tool varies based on the tool being used. Check the synthesis tool's documentation for information on multi-cycle paths.

Guideline – We recommend careful consideration before trying to pipeline a design. While pipelining can dramatically increase the clock speed, it can be difficult to do correctly. Also, since multicycle paths lend themselves to human error and tend to be more troublesome due to the difficulties in analyzing them correctly, they are not generally recommended for reusable modules.

In a design with multiple levels of logic between registers, the clock speed is limited by the clock-to-out time of the source flip-flop, plus the logic delay through the multiple levels of logic, plus the routing associated with the logic levels, plus the setup time of the destination register. Pipelining a design reduces the number of logic levels between the registers. The end result is a system clock that can run much faster.

Example 4 – Verilog Example before Pipelining

```
module no_pipeline (a, b, c, clk, out);
        input    a, b, c, clk;
        output   out;
        reg      out;
        reg      a_temp, b_temp, c_temp;
        always @(posedge clk) begin
                 out = (a_temp * b_temp) + c_temp;
                 a_temp = a; b_temp = b; c_temp = c;
        end
endmodule
```

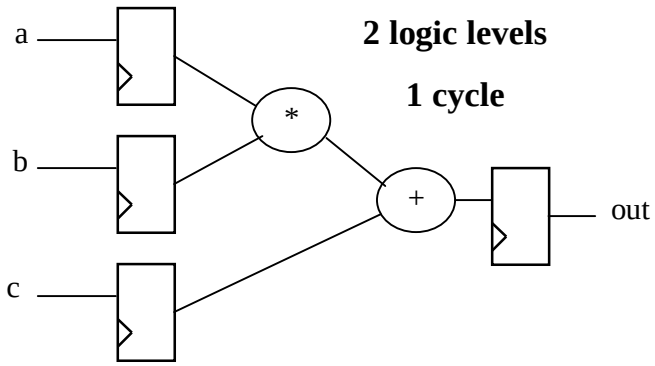


Figure 8 – Example before Pipelining

Example 5 – Verilog Example after Pipelining

```
module pipeline (a, b, c, clk, out);
        input    a, b, c, clk;
        output   out;
        reg      out;
        reg      a_temp, b_temp, c_temp1, c_temp2, mult_temp;
        always @(posedge clk) begin
                 mult_temp = a_temp * b_temp;
                 a_temp = a;  b_temp = b;
        end
        always @(posedge clk) begin
                 out = mult_temp + c_temp2;
```

```
                c_temp2 = c_temp1;
                c_temp1 = c;
        end
endmodule
```

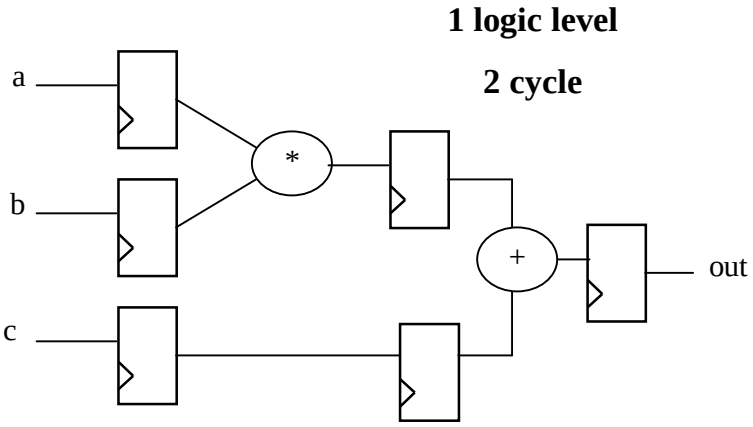


Figure 9 – Example after Pipelining

3.2 Case and IF-Then-Else

The goal in designing fast FPGA designs is to fit the most logic into one Combinatorial Logic Block (CLB). In the example of a Xilinx Virtex device, each CLB can implement any 6-input function and some functions of up to 13 variables. This means an 8-to-1 Mux can be implemented in 1 CLB delay and 1 local interconnect in 2.5ns (-6 device). In ASICs, the delay penalty for additional logic levels is much less than in FPGAs where each CLB logic level can be modeled as a step function increase in delay.

Improper use of the Nested If statement can result in an increase in area and longer delays in a design. Each If keyword specifies a priority-encoded logic whereas the Case statement generally creates balanced logic. An If statement can contain a set of different expressions while a Case statement is evaluated against a common controlling expression. Most synthesis tools can determine if the If-Elsif conditions are mutually exclusive, and will not create extra logic to build the priority tree.

Rule - To avoid long path delays, do not use extremely long Nested If constructs. In general, use the Case statement for complex decoding and use the If statement for speed-critical paths.

Guideline - In general, If-Else constructs are much slower unless the intention is to build a priority encoder. The If-Else statements are appropriate to use for priority encoders. In this case assign the highest priority to a late arriving critical signal.

Guideline - To quickly spot an inefficient nested if statement, scan code for deeply indented code.

Example 6 – VHDL example of inefficient nested If Statement

```
NESTED_IF: process (CLK)
begin
   if (CLK'event and CLK ='1') then
      if (RESET = '0') then
         if (ADDR_A = "00") then
            DEC_Q(5 downto 4) <=ADDR_D;
            DEC_Q(3 downto 2) <="01";
            DEC_Q(1 downto 0) <="00";
            if (ADDR_B = "01") then
               DEC_Q(3 downto 2) <= unsigned(ADDR_A) + '1';
               DEC_Q(1 downto 0) <= unsigned(ADDR_B) + '1';
               if (ADDR_C = "01") then
                  DEC_Q(5 downto 4) <=unsigned(ADDR_D) + '1';
                  if (ADDR_D = "11") then
                     DEC_Q(5 downto 4) <= "00";
                     end if;
                  else
                     DEC_Q(5 downto 4) <= ADDR_D;
                  end if;
               end if;
            else
               DEC_Q(5 downto 4) <= ADDR_D;
               DEC_Q(3 downto 2) <= ADDR_A;
               DEC_Q(1 downto 0) <= unsigned(ADDR_B) + '1';
            end if;
         else
            DEC_Q <= "000000";
         end if;
      end if;
end process;
```

7 Levels of Indentation →

In example 7 the nested If was modified to Use If-Case

Example 7 – VHDL Example of Case

```
IF_CASE: process (CLK)
begin
   if (CLK'event and CLK = '1') then
      if (RESET = '0') then
         case ADDR_ALL is
            when "00011011" =>
               DEC_Q(5 downto 4) <= "00";
               DEC_Q(3 downto 2) <= unsigned(ADDR_A) + '1';
               DEC_Q(1 downto 0) <= unsigned(ADDR_B) + '1';
            when "000110--" =>
               DEC_Q(5 downto 4) <= unsigned(ADDR_D) + '1';
               DEC_Q(3 downto 2) <= unsigned(ADDR_A) + '1';
               DEC_Q(1 downto 0) <= unsigned(ADDR_B) + '1';
            when "0001----" =>
               DEC_Q(5 downto 4) <= ADDR_D;
               DEC_Q(3 downto 2) <= unsigned(ADDR_A) + '1';
               DEC_Q(1 downto 0) <= unsigned(ADDR_B) + '1';
            when "00------" =>
               DEC_Q(5 downto 4) <= ADDR_D;
```

5 Levels of Indentation →

```
            DEC_Q(3 downto 2) <= “01”;
            DEC_Q(1 downto 0) <= “00”;
         when other =>
            DEC_Q(5 downto 4) <= ADDR_D;
            DEC_Q(3 downto 2) <= ADDR_A;
            DEC_Q(1 downto 0) <= unsigned(ADDR_B) + ‘1’;
         end case;
      else
         DEC_Q <= “000000”;
      end if;
   end if;
end process;
```

If-then-else statements are appropriate to use when you need a priority encoder. In this case you should assign the highest priority to a late arriving critical signal.

Example 8 – Verilog 8-to-1 MUX Example using IF-THEN-ELSE for Late Arriving Signals

```
always @(sel or in)
begin
 if (sel == 3'h0)
   out = in[0];
   else if (sel == 3'h1)
   out = in[1];
   else if (sel == 3'h2)
   out = in[2];
   else if (sel == 3'h3)
   out = in[3];
   else if (sel == 3'h4)
   out = in[4];
   else
   out = in[5];
end
```

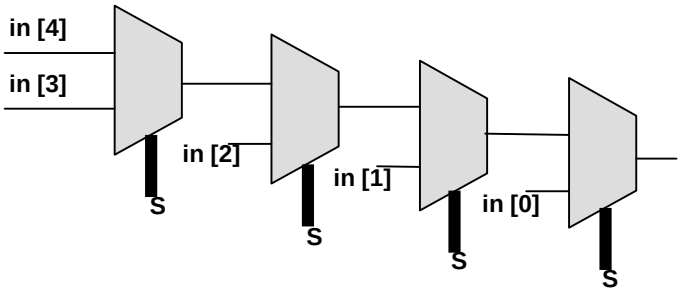


Figure 12 – 8-to-1 MUX Implementation

In the example of an 8-to-1 Multiplexer Design, using a Case statement yields a more compact design resulting in a faster implementation. In most FPGA architectures a 4-to-1 MUX can be implemented in a single CLB slice where it would take multiple CLB logic levels to implement using If-Else.

Example 9 – Verilog 8-to-1 MUX Example using Case

```
always @(C or D or E or F or S)
begin
  case (S)
```

```
      2'b000  : Z = C;
      2'b001  : Z = D;
      2'b010  : Z = E;
      2'b011  : Z = F;
      2'b100  : Z = G;
      2'b101  : Z = H;
      2'b110  : Z = I;
      default : Z = J;
  endcase
```

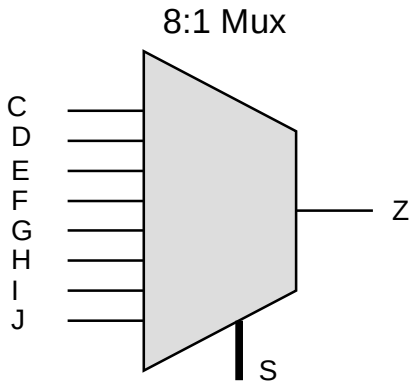


Figure 13 – 8-to-1 MUX Implementation

3.3 Critical Path Optimization

A common technique that is used to speed-up a critical path is to reduce the number of logic levels on the critical path by giving the late arriving signal the highest priority.

Example 10 – VHDL Example of Critical Path before Recoding

```
module critical_bad (in0, in1, in2, in3, critical, out);
   input in0, in1, in2, in3, critical;
   output out;

   assign out = (((in0&in1) & ~critical) | ~in2) & ~in3;

endmodule
```

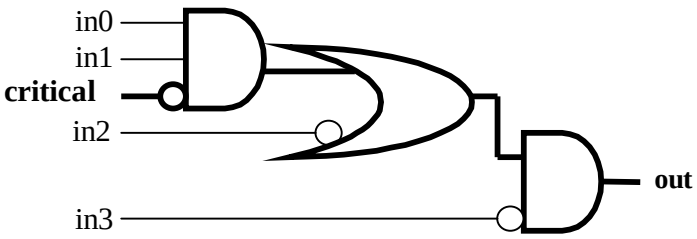


Figure 14 - Critical Path before Recoding

Example 11 – VHDL Example of Critical Path after Recoding

```
module critical_good (in0, in1, in2, in3, critical, out);
  input  in0, in1, in2, in3, critical;
  output out;

  assign out = ((in0&in1) | ~in2) & ~in3 & ~critical;

endmodule
```

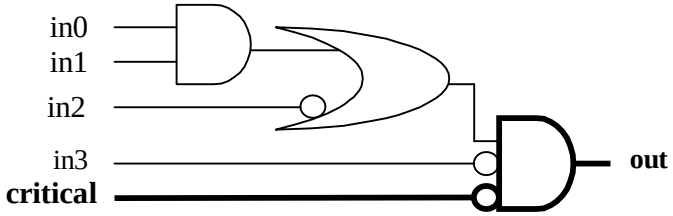


Figure 15 - Critical Path after Recoding

3.4 Tristate vs. Mux Buses

The first consideration in designing any on-chip bus is whether to use a tristate bus or a multiplexer-based bus. Tristate buses are popular for board-level designs and are also commonly found in FPGA-based designs, because they reduce the number of wires and are readily available on many FPGA devices. Tristate buses are problematic for on-chip interconnection since it is essential that only one driver is active on the bus at any one-time; any bus contention, with multiple drivers active at the same time, can increase power consumption and reduce the reliability of the chip. There are additional problems in ASICs that do not exist for FPGAs. For example, ASIC designers must make sure that tristate buses are never allowed to float. FPGA technologies provide weak keeper circuits that pull-up the floating bus to a known value.

Tristate buses are especially problematic for modules designed for reuse. There are a limited number of tristate resources (i.e., tristate buffers connected to interconnect) in each device family and device size within a family. The next designer may not have enough resources available, forcing a significant redesign.

Guidelines – We recommend using multiplexer-based buses when designing for reuse since they are technology-independent and more portable.

3.5 Arithmetic Functions

FPGA architectures designed for system level integration contains dedicated carry logic circuitry that provides fast arithmetic carry capabilities for high-speed arithmetic functions. The dedicated carry logic is generally inferred by the synthesis tools from an arithmetic operator (i.e., +, -, /**).** In the Xilinx Virtex architecture a 16x16 multiplier can effectively use the carry logic from the multiplier operand "*" and operate at 60MHz non-pipelined and 160MHz with pipeline stages. Many synthesis tools have libraries of pre-optimized functions, such as Synopsys DesignWare libraries, which can be inferred from RTL code as shown in the example following.

```
sum = a_in * b_in.
```

Guideline – Refer to the synthesis tools reference manual for the RTL coding style to effectively utilize the dedicated carry logic for fast arithmetic functions.

4 Verification Strategy

Design verification for ASIC system-level and reusable macros has consistently been one of the most difficult and challenging aspects for designers. FPGA design methodologies provide a flexible verification strategy resulting in a wide variety of verification methods and tools. Often, in smaller non system-level designs, functional simulation is bypassed and the designer proceeds directly to board level testing with probe points that can be easily added or removed. Timing verification in the form of simulation or static timing are used to test worst-case conditions or potential race conditions that may not be found during board level testing. The reprogrammability of the device allows the designer to easily probe or observe internal nodes. This methodology is very different from the traditional ASIC verification strategy, which requires rigorous testing to minimize the risk of manufacturing an incorrect design. Because of these differences in methodologies, widespread adoption of verification tools among FPGA users have slightly lagged ASIC users.

4.1 HDL Simulation and Testbench

It is recommended for multi-million gate FPGAs that an ASIC verification methodology be used that consists of a verification plan and strategy. The verification strategy generally consists of compliance, corner, random, real code and regression testing. Modules and sub-modules must be simulated and documented in order to ensure future usability. In surveys taken of digital designers, verification is often cited as the least favorite activity. A good testbench is more likely to be reused than the actual design code.

Guideline - A testbench methodology is recommended for both ASIC and FPGA modules designed for reuse. The same HDL simulators can be used to verify ASIC and FPGA designs.

4.2 Static Timing

For timing verification, static timing analysis is the most effective method of verifying a module's timing performance. As gate densities increase, gate-level simulators slow down, thereby limiting the number of test vectors that can be run and resulting in lower path coverage.

Guideline - Static timing provides a faster means to test all paths in the design. However, it is recommended to use a gate-simulator to check for misidentified false paths and to check blocks of asynchronous logic. .

A noticeable advantage of FPGAs is that multiple libraries and pre-layout statistical wireload models are not needed. Once the design is implemented, the layout is essentially determined and the timing numbers are real. Many FPGA vendors such as Xilinx and Actel also provide the ability to test bestcase and worstcase conditions and to vary the temperature and voltage. Varying the temperature and voltage in an ASIC device generally changes the delays. Since FPGA vendors usually publish worst case operating conditions for the various speedgrades of the devices. Reducing the maximum temperature and or increasing the minimum voltage causes faster operating condition or pro-rates the delays.

4.3 Formal Verification

Formal verification is beginning to emerge as a promising methodology for FPGA system level and design reuse methodology. Although FPGA designs do not go through the same physical transformations as ASICs, such as scan chain insertion, FPGA designs do go through less obtrusive transformation while being optimized and implemented into the FPGA's physical resources. Formal verification is a quick method to check that the functionality of the design remains as intended, providing additional peace of mind. More importantly for design reuse; formal verification can be used to check the functionality from one technology to another, providing maximum flexibility for the future.