Introduction to CPLD and FPGA Design

ESC-306, ESC-326

Bob Zeidman
President
Zeidman Consulting

Bob@ZeidmanConsulting.com
www. ZeidmanConsulting.com

1. INTRODUCTION

Complex Programmable Logic Devices (CPLDs) and Field Programmable Gate Arrays (FPGAs) are becoming a critical part of every system design. Many vendors offer many different architectures and processes. Which one is right for your design? How do you design one of these so that it works correctly and functions as you expect in your entire system? These are the questions that this paper sets out to answer.

The first sections of this paper deals with the internal architecture and characteristics of these devices. Simple programmable logic devices are described in an overview, leading up to a detailed description of the Complex Programmable Logic Device and the Field Programmable Gate Array. The various architectures of these devices are examined in detail along with their tradeoffs, which allow you to decide which particular device is right for your design.

The next sections of this paper discuss in detail, the design, simulation, and testing issues that arise when designing a CPLD or FPGA. The final sections of this paper examine new architectures of programmable devices and the software needed to support them.

Understanding these issues will allow you to design a chip that functions correctly in your system and will be reliable throughout the lifetime of your product.

2. THE MASKED GATE ARRAY ASIC

An Application Specific Integrated Circuit, or ASIC, is a chip that can be designed by an engineer with no particular knowledge of semiconductor physics or semiconductor processes. The ASIC vendor has created a library of cells and functions that the designer can use without needing to know precisely how these functions are implemented in silicon. The ASIC vendor also typically supports software tools that automate such processes as synthesis and circuit layout. The ASIC vendor may even supply application engineers to assist the ASIC design engineer with the task. The vendor then lays out the chip, creates the masks, and manufactures the ASICs.

The gate array is an ASIC with a particular architecture that consists of rows and columns of regular transistor structures. Each basic cell, or gate, consists of the same small number of transistors that are not connected. In fact, none of the transistors on the gate array are initially connected at all. The reason for this is that the connection is determined completely by the design that you implement. Once you have your design, the layout software figures out which transistors to connect. First, your low level functions are connected together. For example, six transistors could be connected to create

a D flip-flop. These six transistors would be located physically very close to each other. After your low level functions have been routed, these would in turn be connected together. The software would continue this process until the entire design is complete. This row and column structure is illustrated in Figure 1.

The ASIC vendor manufactures many unrouted die which contain the arrays of gates and which it can use for any gate array customer. An integrated circuit consists of many layers of materials including semiconductor material (e.g., silicon), insulators (e.g., oxides), and conductors (e.g., metal). An unrouted die is processed with all of the layers except for the final metal layers that connect the gates together. Once your design is complete, the vendor simply needs to add the last metal layers to the die to create your chip, using photomasks for each metal layer. For this reason, it is sometimes referred to as a Masked Gate Array to differentiate it from a Field Programmable Gate Array.

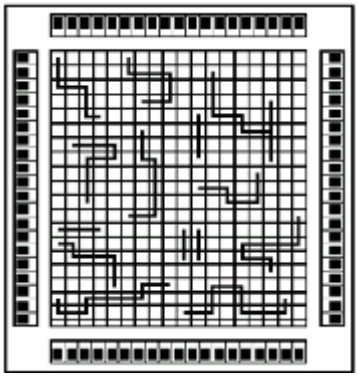

Figure 1 Masked Gate Array Architecture

3. THE EVOLUTION OF PROGRAMMABLE DEVICES

Programmable devices have gone through a long evolution to reach the complexity that they have today. The following sections give an approximately chronological discussion of these devices from least complex to most complex.

3.1 Programmable Read Only Memories (PROMs)

Programmable Read Only Memories, or PROMs, are simply memories that can be inexpensively programmed by the user to contain a specific pattern. This pattern can be used to represent a microprocessor program, a simple algorithm, or a state machine. Some PROMs can be programmed once only. Other PROMs, such as EPROMs or EEPROMs can be erased and programmed multiple times.

PROMs are excellent for implementing any kind of combinatorial logic with a limited number of inputs and outputs. For sequential logic, external

clocked devices such as flip-flops or microprocessors must be added. Also, PROMs tend to be extremely slow, so they are not useful for applications where speed is an issue.

3.2 Programmable Logic Arrays (PLAs)

Programmable Logic Arrays (PLAs) were a solution to the speed and input limitations of PROMs. PLAs consist of a large number of inputs connected to an AND plane, where different combinations of signals can be logically ANDed together according to how the part is programmed. The outputs of the AND plane go into an OR plane, where the terms are ORed together in different combinations and finally outputs are produced. At the inputs and outputs there are typically inverters so that logical NOTs can be obtained. These devices can implement a large number of combinatorial functions, though not all possible combinations like a PROM can. However, they generally have many more inputs and are much faster.

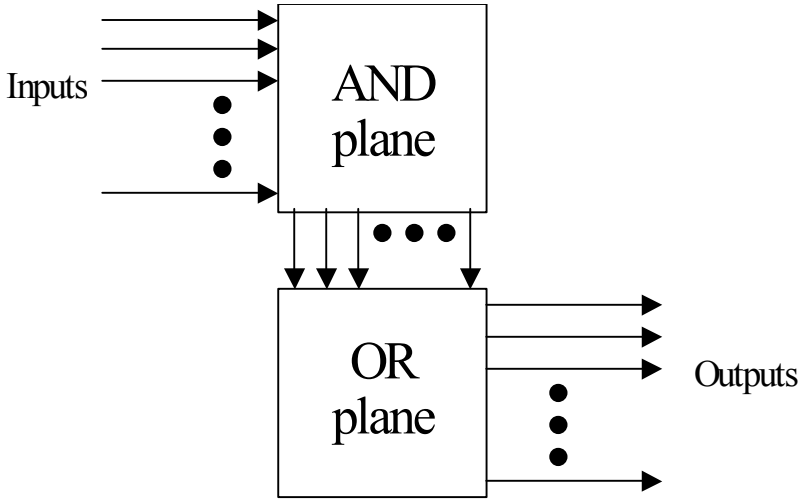

Figure 2 PLA Architecture

3.3 Programmable Array Logic (PALs)

The Programmable Array Logic (PAL) is a variation of the PLA. Like the PLA, it has a wide, programmable AND plane for ANDing inputs together. However, the OR plane is fixed, limiting the number of terms that can be ORed together. Other basic logic devices, such as multiplexers, exclusive ORs, and latches are added to the inputs and outputs. Most importantly, clocked elements, typically flip-flops, are included. These devices are now able to implement a large number of logic functions including clocked sequential logic need for state machines. This was an important development that allowed PALs to replace much of the standard logic in many designs. PALs are also extremely fast.

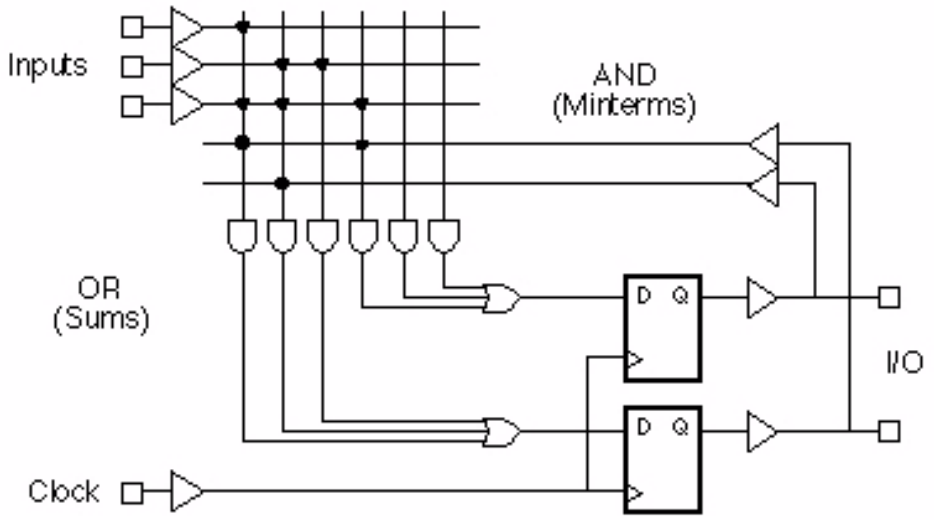

Figure 3 PAL Architecture

3.4 CPLDs and FPGAs

Ideally, though, the hardware designer wanted something that gave him or her the flexibility and complexity of an ASIC but with the shorter turn-around time of a programmable device. The solution came in the form of two new devices - the Complex Programmable Logic Device (CPLD) and the Field Programmable Gate Array. As can be seen in Figure 4, CPLDs and FPGAs bridge the gap between PALs and Gate Arrays. CPLDs are as fast as PALs but more complex. FPGAs approach the complexity of Gate Arrays but are still programmable.

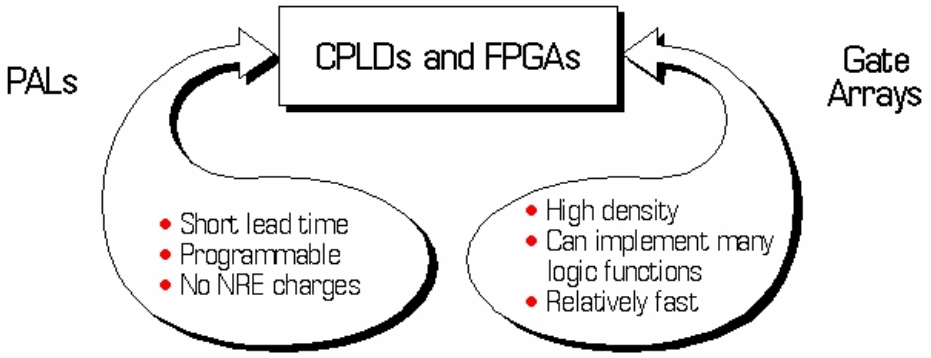

Figure 4 Comparison of CPLDs and FPGAs

3.5 Complex Programmable Logic Devices (CPLDs)

Complex Programmable Logic Devices (CPLDs) are exactly what they claim to be. Essentially they are designed to appear just like a large number of PALs in a single chip, connected to each other through a crosspoint switch They use the same development tools and programmers, and are based on the same technologies, but they can handle much more complex logic and more of it.

3.5.1 CPLD Architectures

The diagram in Figure 5 shows the internal architecture of a typical CPLD. While each manufacturer has a different variation, in general they are all similar in that they consist of function blocks, input/output block, and an interconnect matrix. The devices are programmed using programmable elements that, depending on the technology of the manufacturer, can be EPROM cells, EEPROM cells, or Flash EPROM cells.

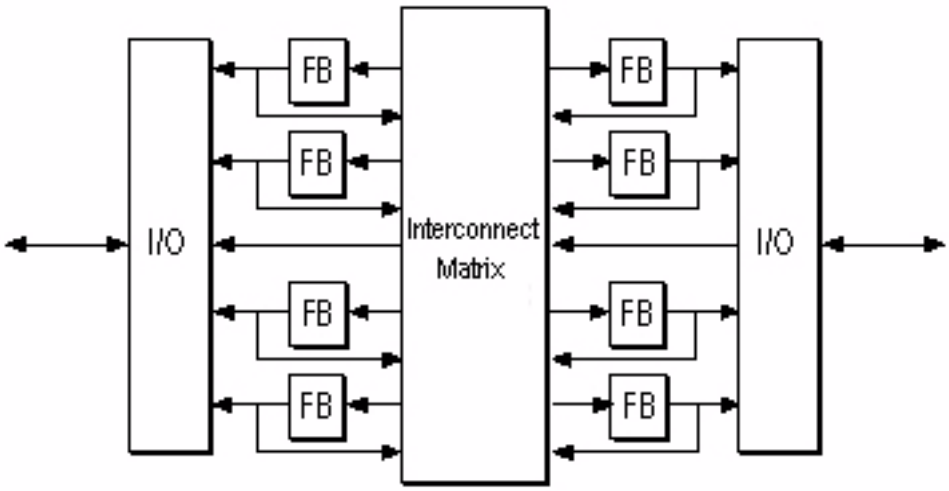

Figure 5 CPLD Architecture

3.5.1.1 Function Blocks

A typical function block is shown in Figure 6. The AND plane still exists as shown by the crossing wires. The AND plane can accept inputs from the I/O blocks, other function blocks, or feedback from the same function block. The terms and then ORed together using a fixed number of OR gates, and terms are selected via a large multiplexer. The outputs of the mux can then be sent straight out of the block, or through a clocked flip-flop. This particular block includes additional logic such as a selectable exclusive OR and a master reset signal, in addition to being able to program the polarity at different stages.

Usually, the function blocks are designed to be similar to existing PAL architectures, such as the 22V10, so that the designer can use familiar tools or even older designs without changing them.

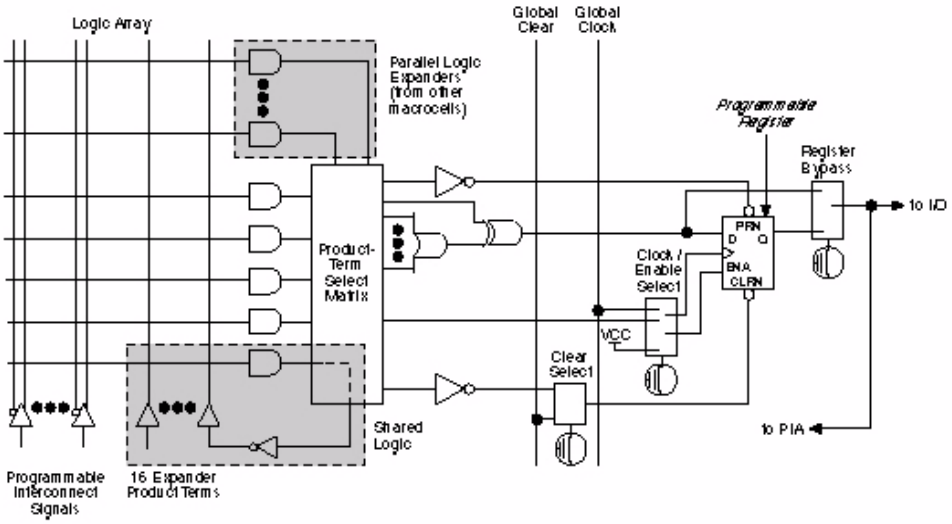

Figure 6 CPLD Function Block

3.5.1.2 I/O Blocks

Figure 7 shows a typical I/O block of a CPLD. The I/O block is used to drive signals to the pins of the CPLD device at the appropriate voltage levels with the appropriate current. Usually, a flip-flop is included, as shown in the figure. This is done on outputs so that clocked signals can be output directly to the pins without encountering significant delay. It is done for inputs so that there is not much delay on a signal before reaching a flip-flop, which would increase the device hold time requirement. Also, some small amount of logic is included in the I/O block simply to add some more resources to the device.

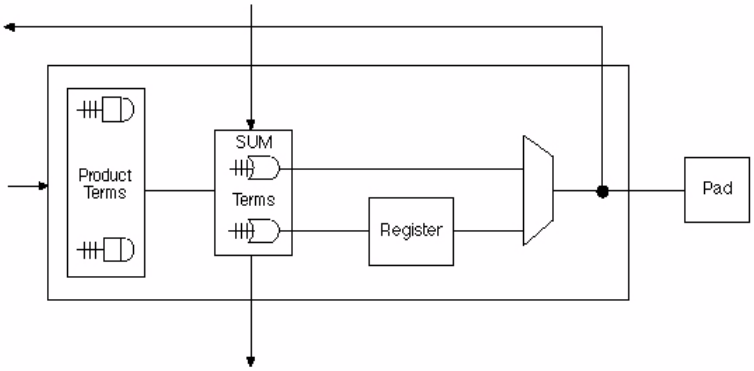

Figure 7 CPLD Input/Output Block

3.5.1.3 Interconnect

The CPLD interconnect is a very large programmable switch matrix that allows signals from all parts of the device go to all other parts of the device. While no switch can connect all internal function blocks to all other function blocks, there is enough flexibility to allow many combinations of connections.

3.5.1.4 Programmable Elements

Different manufacturers use different technologies to implement the programmable elements of a CPLD. The common technologies are Electrically Programmable Read Only Memory (EPROM), Electrically Erasable PROM (EEPROM) and Flash EPROM. These technologies are similar to, or next generation versions of, the technologies that were used for the simplest programmable devices, PROMs.

3.5.2 CPLD Architecture Issues

When considering a CPLD for use in a design, the following issues should be taken into account:

1. The programming technology
    - EPROM, EEPROM, or Flash EPROM? This will determine the equipment needed to program the devices and whether they came be programmed only once or many times.
2. The function block capability
    - How many function blocks are there in the device?
    - How many product and sum terms can be used?
    - What are the minimum and maximum delays through the logic?
    - What additional logic resources are there such as XNORs, ALUs, etc.?
    - What kinds of register controls are available (e.g., clock enable, reset, preset, polarity control)? How many are local inputs to the function block and how many are global, chip-wide inputs?
    - What kind of clock drivers are in the device and what is the worst-case skew of the clock signal on the chip. This will help determine the maximum frequency at which the device can run.
3. The I/O capability
    - How many I/O are independent, used for any function, and how many are dedicated for clock input, master reset, etc.?
    - What is the output drive capability in terms of voltage levels

and current?
- What kind of logic is included in an I/O block that can be used to increase the functionality of the design?

3.5.3 Example CPLD Families

Some CPLD families from different vendors are listed below:

- Altera MAX 7000 and MAX 9000 families
- Atmel ATF and ATV families
- Lattice ispLSI family
- Lattice (Vantis) MACH family
- Xilinx XC9500 family

3.6 Field Programmable Gate Arrays (FPGAs)

Field Programmable Gate Arrays are called this because rather than having a structure similar to a PAL or other programmable device, they are structured very much like a gate array ASIC. This makes FPGAs very nice for use in prototyping ASICs, or in places where and ASIC will eventually be used. For example, an FPGA maybe used in a design that needs to get to market quickly regardless of cost. Later an ASIC can be used in place of the FPGA when the production volume increases, in order to reduce cost.

3.6.1 FPGA Architectures

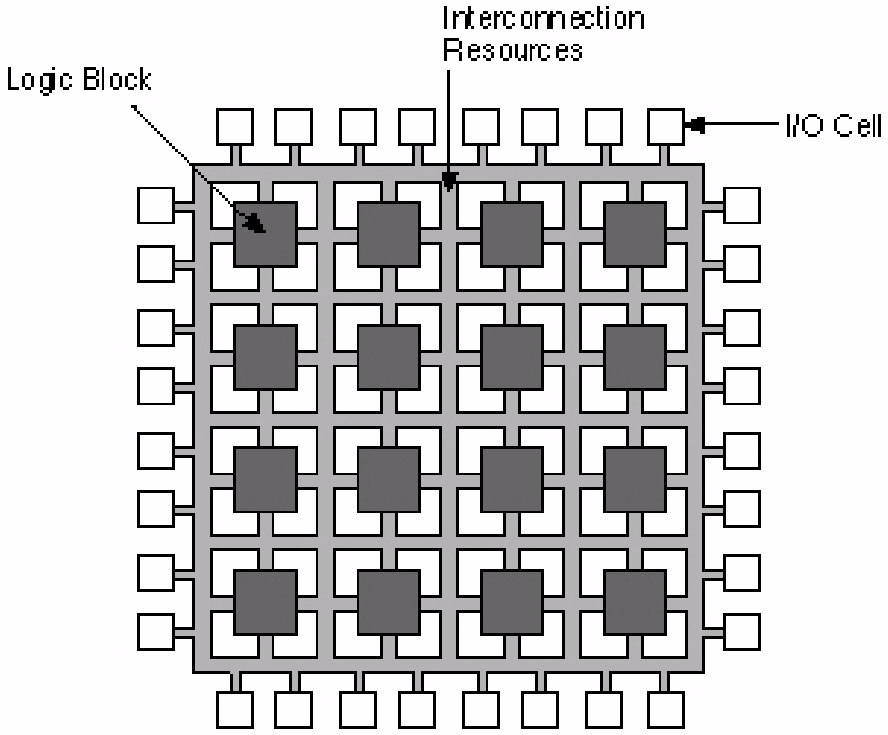

Figure 8 FPGA Architecture

Each FPGA vendor has its own FPGA architecture, but in general terms they are all a variation of that shown in Figure 8. The architecture consists of configurable logic blocks, configurable I/O blocks, and programmable interconnect. Also, there will be clock circuitry for driving the clock signals to each logic block, and additional logic resources such as ALUs, memory, and decoders may be available. The two basic types of programmable elements for an FPGA are Static RAM and anti-fuses.

3.6.1.1 Configurable Logic Blocks

Configurable Logic Blocks contain the logic for the FPGA. In a large-grain architecture, these CLBs will contain enough logic to create a small state machine. In a fine-grain architecture, more like a true gate array ASIC, the CLB will contain only very basic logic. The diagram in Figure 9 would be considered a large grain block. It contains RAM for creating arbitrary combinatorial logic functions, also known as lookup tables (LUTs). It also contains flip-flops for clocked storage elements, and multiplexers in order to route the logic within the block and to and from external resources. The muxes also allow polarity selection and reset and clear input selection.

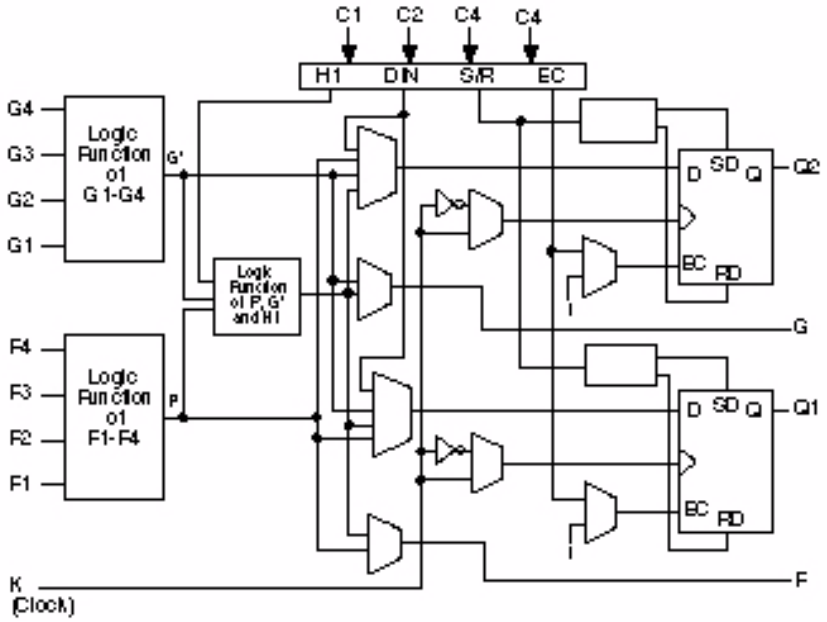

Figure 9 FPGA Configurable Logic Block

3.6.1.2 Configurable I/O Blocks

A Configurable I/O Block, shown in Figure 10, is used to bring signals onto the chip and send them back off again. It consists of an input buffer and an output buffer with three state and open collector output controls. Typically there are pull up resistors on the outputs and sometimes pull down resistors. The polarity of the output can usually be programmed for active high or active low output and often the slew rate of the output can be programmed for fast or slow rise and fall times. In addition, there is often a flip-flop on outputs so that clocked signals can be output directly to the pins without encountering significant delay. It is done for inputs so that there is not much delay on a signal before reaching a flip-flop, which would increase the device hold time requirement.

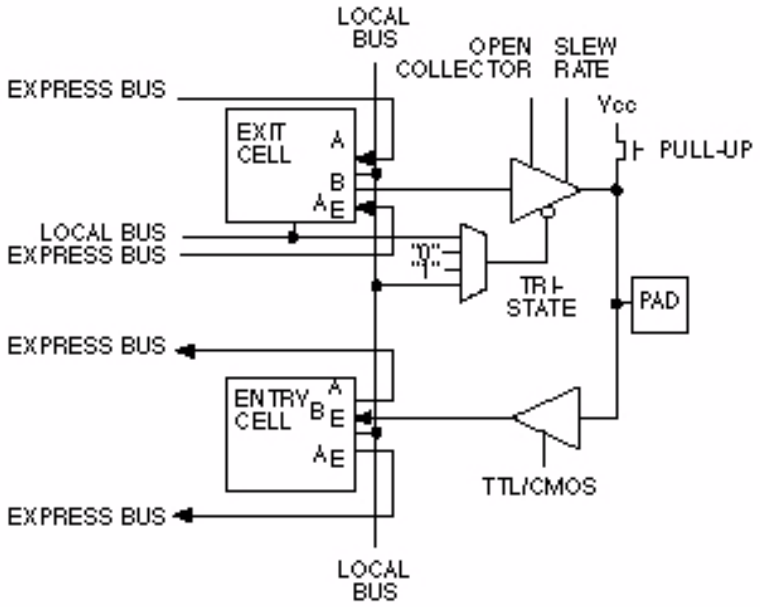

Figure 10 FPGA Configurable I/O Block

3.6.1.3 Programmable Interconnect

The interconnect of an FPGA is very different than that of a CPLD, but is rather similar to that of a gate array ASIC. In Figure 11, a hierarchy of interconnect resources can be seen. There are long lines that can be used to connect critical CLBs that are physically far from each other on the chip without inducing much delay. They can also be used as buses within the chip. There are also short lines that are used to connect individual CLBs that are located physically close to each other. There are often one or several switch matrices, like that in a CPLD, to connect these long and short lines together in specific ways. Programmable switches inside the chip allow the connection of CLBs to interconnect lines and interconnect lines to each other and to the switch matrix. Three-state buffers are used to connect many CLBs to a long line, creating a bus. Special long lines, called global clock lines, are specially designed for low impedance and thus fast propagation times. These are connected to the clock buffers and to each clocked element in each CLB. This is how the clocks are distributed throughout the FPGA.

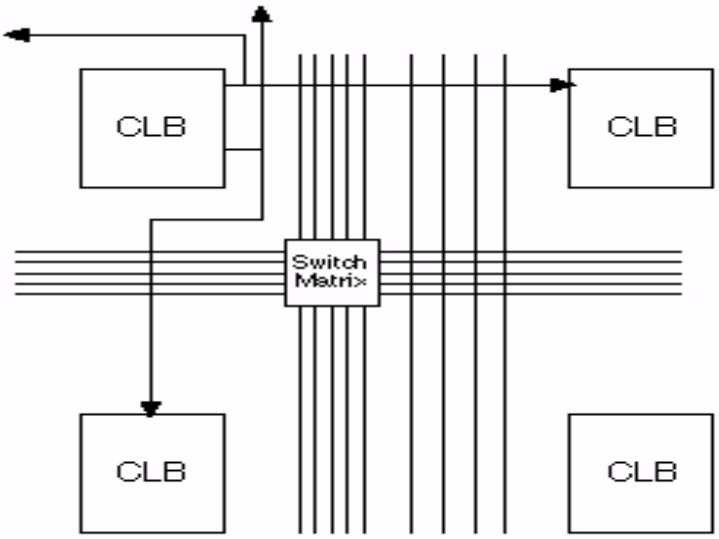

Figure 11 FPGA Programmable Interconnect

3.6.1.4 Clock Circuitry

Special I/O blocks with special high drive clock buffers, known as clock drivers, are distributed around the chip. These buffers connect to clock input pads and drive the clock signals onto the global clock lines described above. These clock lines are designed for low skew times and fast propagation times. As we will discuss later, synchronous design is a must with FPGAs, since absolute skew and delay cannot be guaranteed. Only when using clock signals from clock buffers can the relative delays and skew times be guaranteed.

3.6.2 Small vs. Large Granularity

Small grain FPGAs resemble ASIC gate arrays in that the CLBs contain only small, very basic elements such as NAND gates, NOR gates, etc. The philosophy is that small elements can be connected to make larger functions without wasting too much logic. In a large-grain FPGA, where the CLB can contain two or more flip-flops, a design that does not need many flip-flops will leave many of them unused. Unfortunately, small grain architectures require much more routing resources, which take up space and insert a large amount of delay which can more than compensate for the better utilization.

Small Granularity	**Large Granularity**
better utilization	fewer levels of logic
direct conversion to ASIC	less interconnect delay

Table 1 Small vs. Large Grain FPGAs

A comparison of advantages of each type of architecture is shown in Table 1 above. The choice of which architecture to use is dependent on your specific application.

3.6.3 SRAM vs. Anti-fuse Programming

There are two competing methods of programming FPGAs. The first, SRAM programming, involves small Static RAM bits for each programming element. Writing the bit with a zero turns off a switch, while writing with a one turns on a switch. The other method involves anti-fuses, which consist of microscopic structures that, unlike a regular fuse, normally make no connection. A certain amount of current during programming of the device causes the two sides of the anti-fuse to connect.

The advantages of SRAM based FPGAs is that they use a standard fabrication process that chip fabrication plants are familiar with and are always optimizing for better performance. Since the SRAMs are reprogrammable, the FPGAs can be reprogrammed any number of times, even while they are in the system, just like writing to a normal SRAM. SRAM based devices can easily use the internal SRAMs as small memories in the design. The disadvantages are that they are volatile, which means a power glitch could potentially change it. Also, SRAM-based devices have large routing delays.

The advantages of Anti-fuse based FPGAs are that they are non-volatile and the delays due to routing are very small, so they tend to be faster. Antifuse based FPGAs tend to require lower power and they are better for keeping your design information out of the hands of competitors because they do not require an external device to program them upon power-up as SRAM based devices do. The disadvantages are that they require a complex fabrication process, they require an external programmer to program them, and once they are programmed, they cannot be changed.

3.6.4 Example FPGA Families

Examples of SRAM based FPGA families include the following:

- Altera FLEX family
- Atmel AT6000 and AT40K families
- Lucent Technologies ORCA family
- Xilinx XC4000 and Virtex families

Examples of Anti-fuse based FPGA families include the following:

- Actel SX and MX families
- Quicklogic pASIC family

3.7 Choosing Between CPLDs and FPGAs

Choosing between a CPLD and an FPGA will depend on the characteristics and requirements of your project. A summary of the characteristics of each is show in Figure 12 below.

	CPLD	FPGA
Architecture	PAL-like	Gate Array-like
Density	Low to medium 12 22V10s or more	Medium to high up to 1 million gates
Speed	Fast, predictable	Application dependent
Interconnect	Crossbar	Routing
Power Consumption	High	Medium

Figure 12 CPLDs vs. FPGAs

4. DESIGN ISSUES

In the next sections of this paper, we will discuss those areas that are unique to FPGA design or that are particularly critical to these devices.

4.1 Top-Down Design

Top-down design is the design method whereby high level functions are defined first, and the lower level implementation details are filled in later. A schematic can be viewed as a hierarchical tree as shown in Figure 13. The top level block represents the entire chip. Each lower level block represents major functions of the chip. Intermediate level blocks may contain smaller functionality blocks combined with gate-level logic. The bottom level contains only gates and macrofunctions, which are vendor-supplied high-level functions. Fortunately, schematic capture software and hardware description languages used for chip design easily allow use of the top-down design methodology.

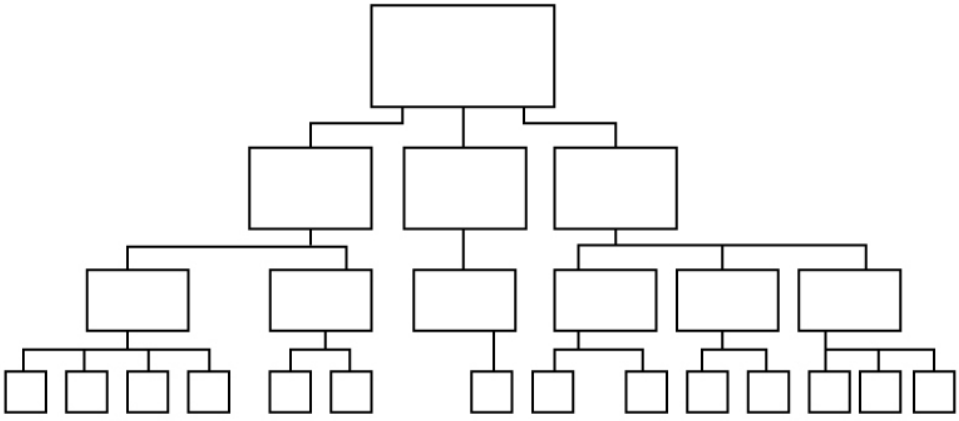

Figure 13 Top-Down Design

Top-down design is the preferred methodology for chip design for several reasons. First, chips often incorporate a large number of gates and a very high level of functionality. This methodology simplifies the design task and allows more than one engineer, when necessary, to design the chip. Second, it allows flexibility in the design. Sections can be removed and replaced with higher-performance or optimized designs without affecting other sections of the chip.

Also important is the fact that simulation is much simplified using this design methodology. Simulation is an extremely important consideration in chip design since a chip cannot be blue-wired after production. For this reason, simulation must be done extensively before the chip is sent for fabrication. A top-down design approach allows each module to be simulated independently from the rest of the design. This is important for complex designs where an entire design can take weeks to simulate and days to debug. Simulation is discussed in more detail later in this paper.

4.2 Keep the Architecture in Mind

Look at the particular architecture to determine which logic devices fit best into it. The vendor may be able to offer advice about this. Many synthesis packages can target their results to a specific FPGA or CPLD family from a specific vendor, taking advantage of the architecture to provide you with faster, more optimal designs.

4.3 Synchronous Design

One of the most important concepts in chip design, and one of the hardest to enforce on novice chip designers, is that of synchronous design. Once a chip designer uncovers a problem due to asynchronous design and attempts to fix it, he or she usually becomes an evangelical convert to synchronous design. This is because asynchronous design problems are due to marginal timing problems that may appear intermittently, or may appear only when the vendor changes its semiconductor process. Asynchronous designs that work for years in one process may suddenly fail when the chip is manufactured using a newer process.

Synchronous design simply means that all data is passed through combinatorial logic and flip-flops that are synchronized to a single clock. Delay is always controlled by flip-flops, not combinatorial logic. No signal that is generated by combinatorial logic can be fed back to the same group of combinatorial logic without first going through a synchronizing flip-flop. Clocks cannot be gated - in other words, clocks must go directly to the clock inputs of the flip-flops without going through any combinatorial logic.

The following sections cover common asynchronous design problems and how to fix them using synchronous logic.

4.3.1 Race conditions

Figure 14 shows an asynchronous race condition where a clock signal is used to reset a flip-flop. When SIG2 is low, the flip-flop is reset to a low state. On the rising edge of SIG2, the designer wants the output to change to the high state of SIG1. Unfortunately, since we don't know the exact internal timing of the flip-flop or the routing delay of the signal to the clock versus the reset input, we cannot know which signal will arrive first - the clock or the reset. This is a race condition. If the clock rising edge appears first, the output will remain low. If the reset signal appears first, the output will go high. A slight change in temperature, voltage, or process may cause a chip that works correctly to suddenly work incorrectly. A more reliable synchronous solution is shown in Figure 15. Here a faster clock is used, and the flip-flop is reset on the rising edge of the clock. This circuit performs the same function, but as long as SIG1 and SIG2 are produced synchronously - they change only after the rising edge of CLK - there is no race condition.

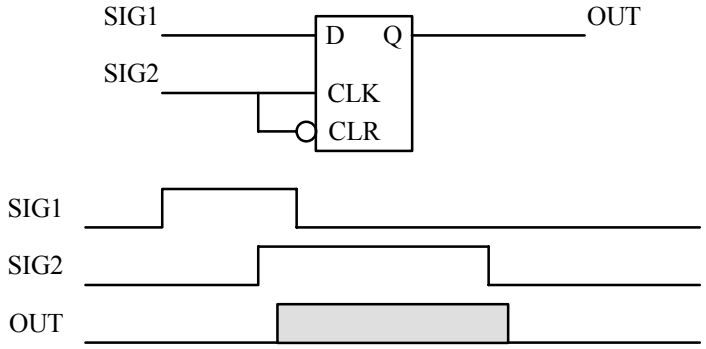

Figure 14 Asynchronous: Race Condition

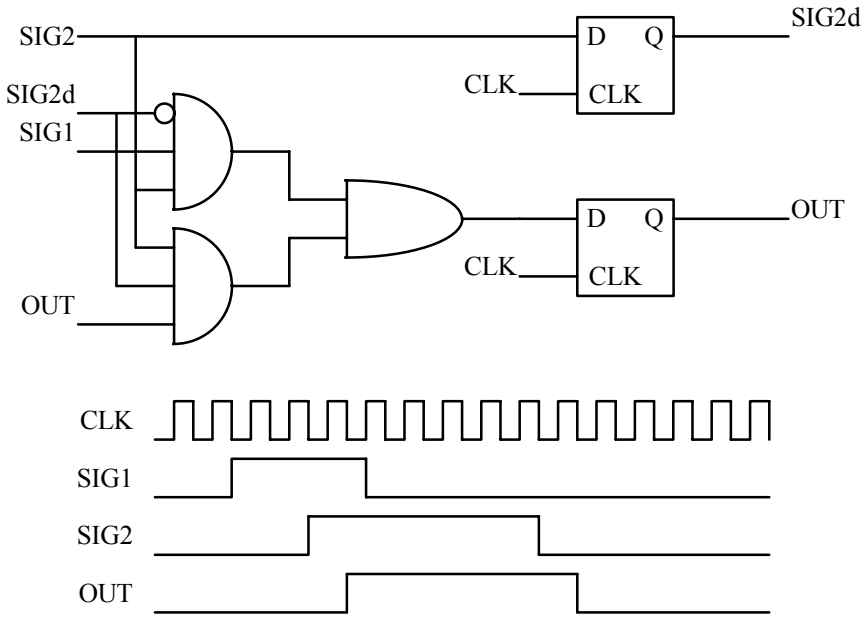

Figure 15 Synchronous: No Race Condition

4.3.2 Delay dependent logic

Figure 16 shows logic used to create a pulse. The pulse width depends very explicitly on the delay of the individual logic gates. If the process should change, making the delay shorter, the pulse width will shorten also, to the point where the logic that it feeds may not recognize it at all. A synchronous pulse generator is shown in Figure 17. This pulse depends only on the clock period. Changes to the process will not cause any significant change in the pulse width.

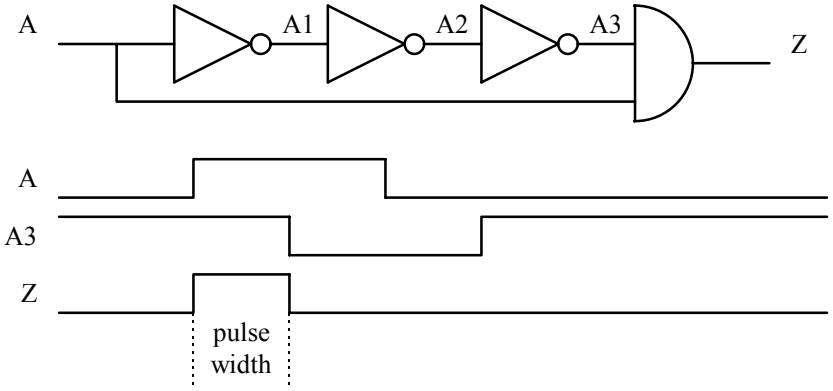

Figure 16 Asynchronous: Delay Dependent Logic

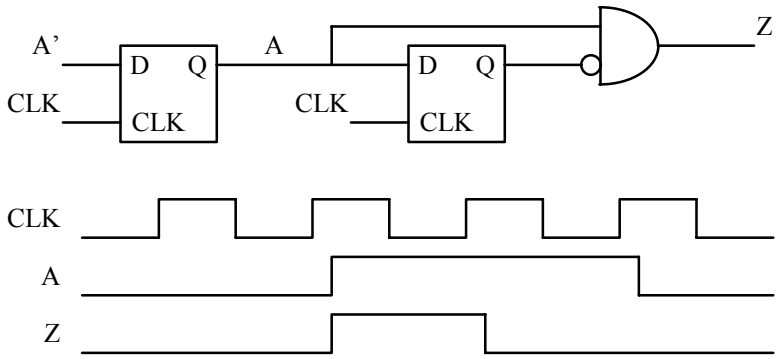

Figure 17 Synchronous: Delay Independent Logic

4.3.3 Hold time violations

Figure 18 shows an asynchronous circuit with a hold time violation. Hold time violations occur when data changes around the same time as the clock edge. It is uncertain which value will be registered by the clock. The circuit in Figure 19 fixes this problem by putting both flip-flops on the same clock and using a flip-flop with an enable input. A pulse generator creates a pulse that enables the flip-flop.

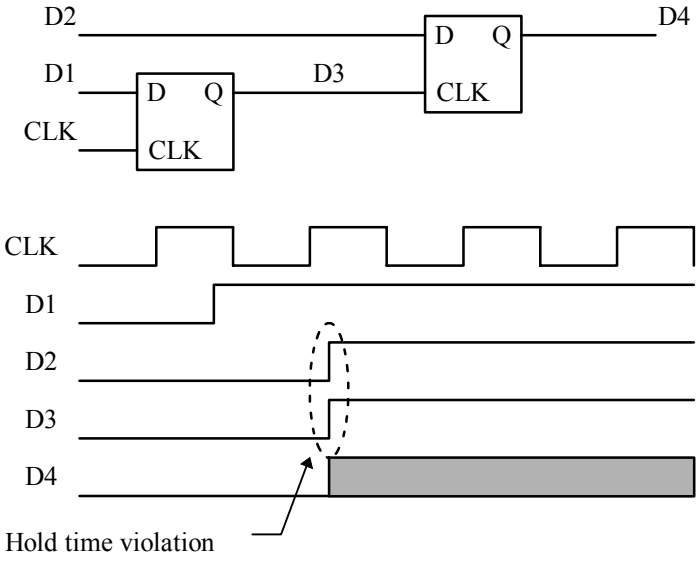

Figure 18 Asynchronous: Hold Time Violation

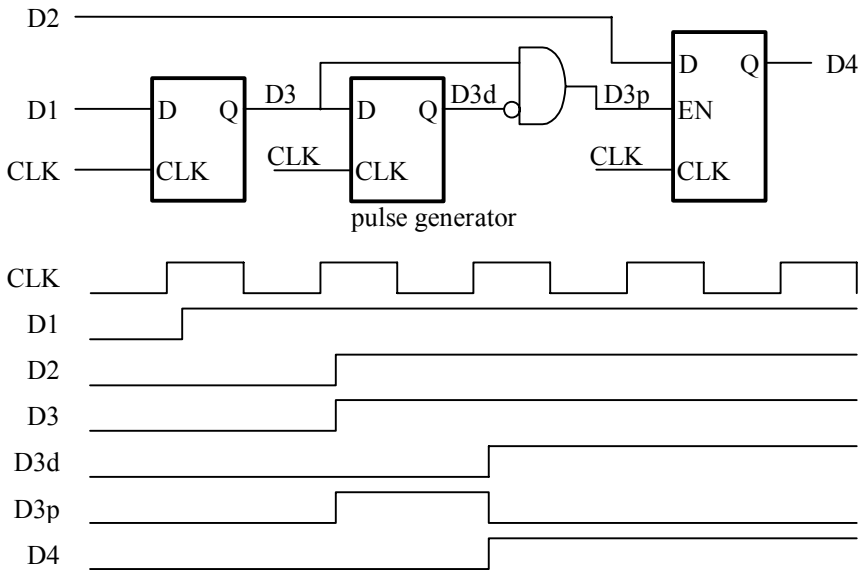

Figure 19 Synchronous: No Hold Time Violation

4.3.4 Glitches

A glitch can occur due to small delays in a circuit such as that shown in Figure 20. An inverting multiplexer contains a glitch when switching between two signals, both of which are high. Yet due to the delay in the inverter, the output goes high for a very short time. Synchronizing this output by sending it through a flip-flop as shown in Figure 21, ensures that this glitch will not appear on the output and will not affect logic further downstream.

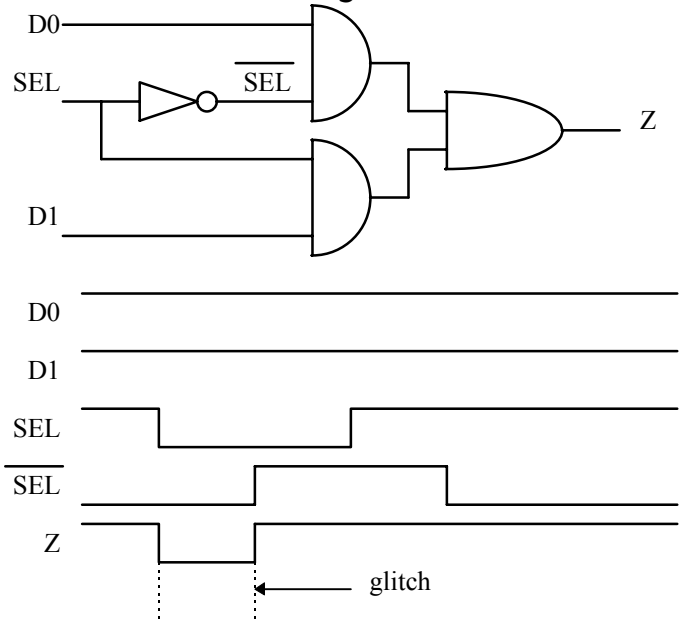

Figure 20 Asynchronous: Glitch

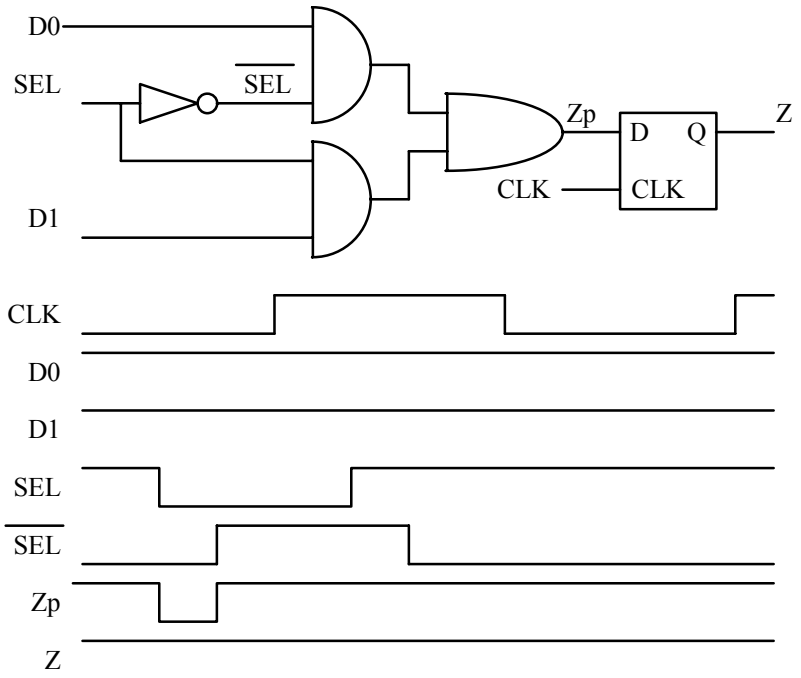

Figure 21 Synchronous: No Glitch

4.3.5 Bad clocking

Figure 22 shows an example of asynchronous clocking. This kind of clocking will produce problems of the type discussed previously. The correct way to enable and disable outputs is not by putting logic on the clock input, but by putting logic on the data input as shown in Figure 23.

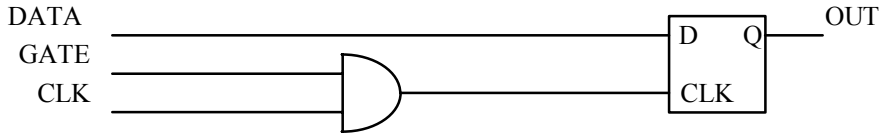

Figure 22 Asynchronous: Bad Clocking

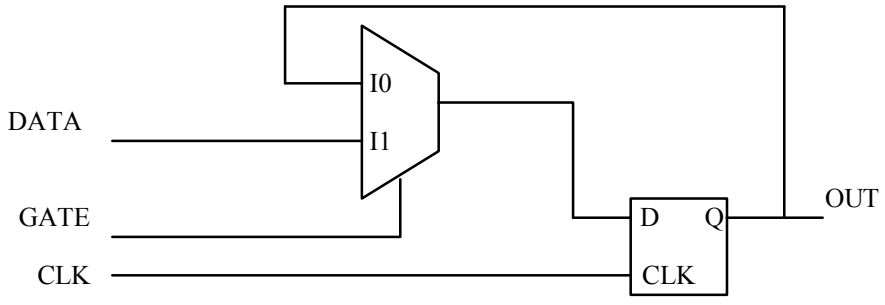

Figure 23 Synchronous: Good Clocking

4.3.6 Metastability

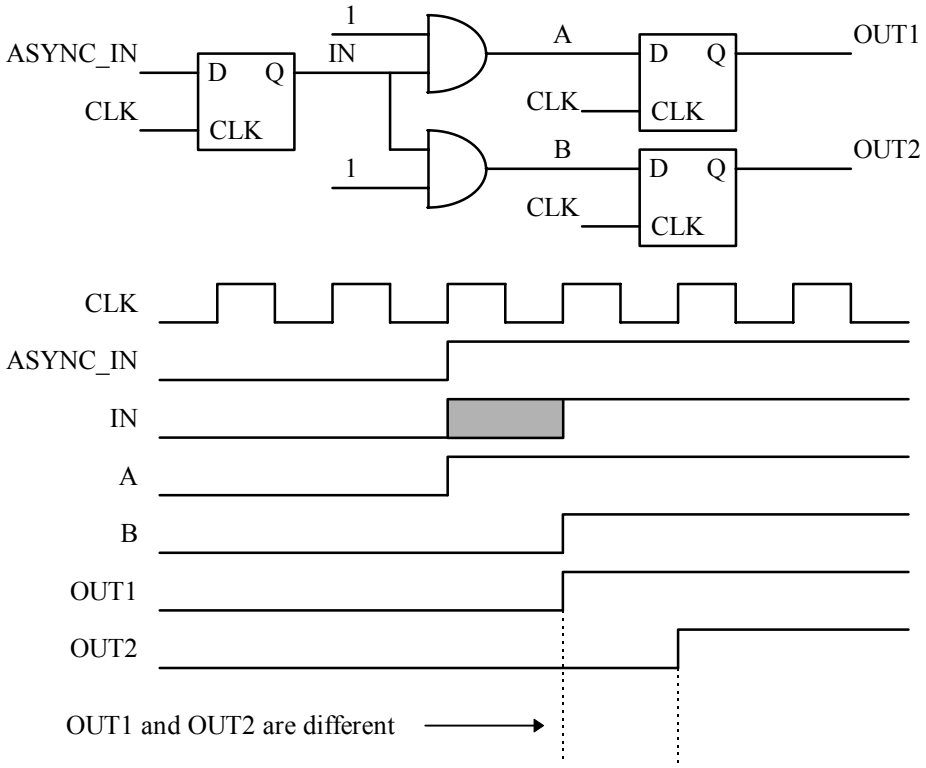

Figure 24 Metastability - The Problem

One of the great buzzwords, and often-misunderstood concepts, of synchronous design is metastability. Metastability refers to a condition that arises when an asynchronous signal is clocked into a synchronous flip-flop. While chip designers would prefer a completely synchronous world, the unfortunate fact is that signals coming into a chip will depend on a user pushing a button or an interrupt from a processor, or will be generated by a clock that is different from the one used by the chip. In these cases, the asynchronous signal must be synchronized to the chip clock so that it can be used by the internal circuitry. The designer must be careful how to do this in order to avoid metastability problems as shown in Figure 24. If the ASYNC_IN signal goes high around the same time as the clock, we have an unavoidable race condition. The output of the flip-flop can actually go to an undefined voltage level that is somewhere between a logic 0 and logic 1. This is because an internal transistor did not have enough time to fully charge to the correct level. This metalevel may remain until the transistor voltage leaks off or “decays”, or until the next clock cycle. During the clock cycle, the gates that are connected to the output of the flip-flop may interpret this level differently. In the figure, the upper gate sees the level as a logic 1 whereas the lower gate sees it as a logic 0. In normal operation, OUT1 and OUT2 should always be the same value. In this case, they are not and this could send the

logic into an unexpected state from which it may never return. This metastability can permanently lock up your chip.

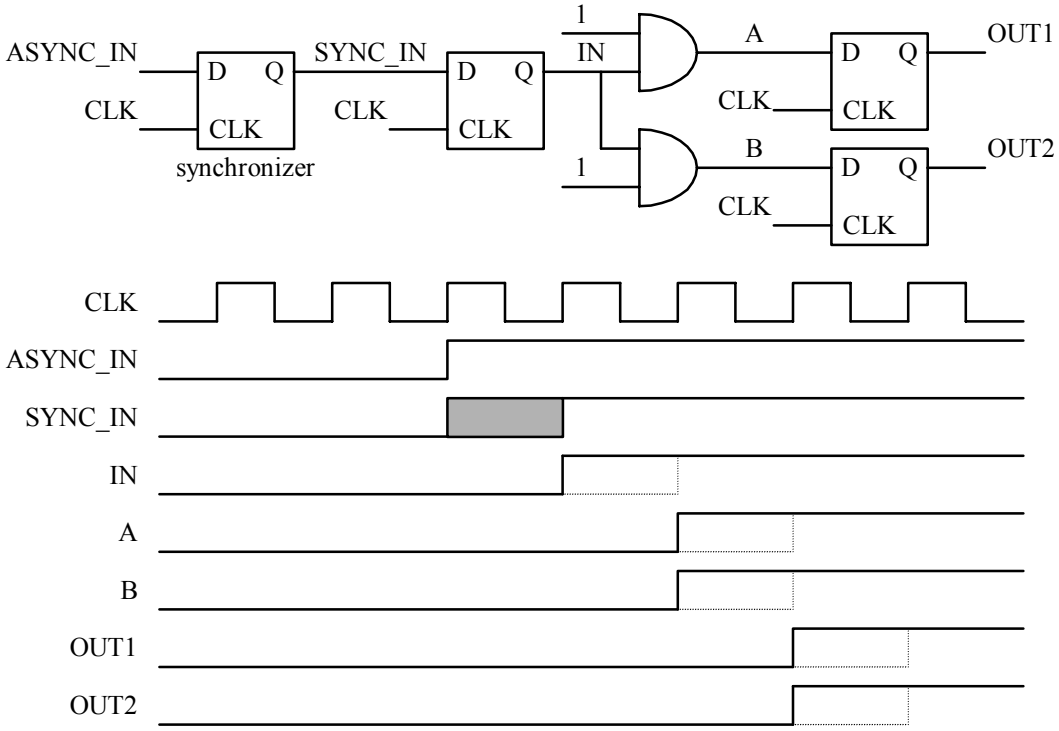

Figure 25 Metastability - The "Solution"

The “solution” to this metastability problem is shown in Figure 25. By placing a synchronizer flip-flop in front of the logic, the synchronized input will be sampled by only one device, the second flip-flop, and be interpreted only as a logic 0 or 1. The upper and lower gates will both sample the same logic level, and the metastability problem is avoided. Or is it? The word solution is in quotation marks for a very good reason. There is a very small but non-zero probability that the output of the synchronizer flip-flop will not decay to a valid logic level within one clock period. In this case, the next flip-flop will sample an indeterminate value, and there is again a possibility that the output of that flip-flop will be indeterminate. At higher frequencies, this possibility is greater. Unfortunately, there is no certain solution to this problem. Some vendors provide special synchronizer flip-flops whose output transistors decay very quickly. Also, inserting more synchronizer flip-flops reduces the probability of metastability but it will never reduce it to zero. The correct action involves discussing metastability problems with the vendor, and including enough synchronizing flip-flops to reduce the probability so that it is unlikely to occur within the lifetime of the product.

Notice that each synchronizer flip-flop may delay the logic level change

on the input by one clock cycle before it is recognized by the internal circuitry of the chip. Given that the external signal is asynchronous, by definition this is not a problem since the exact time that it is asserted will not be deterministic. If this delay is a problem, then most likely the entire system will need to be synchronized to a single clock.

4.3.7 Allowable uses of asynchronous logic

Now that I've gone through a long argument against asynchronous design, I will tell you the few exceptions that I have found to this rule. These exceptions, however, must be designed with extreme caution and only as a last resort when a synchronous solution cannot be found.

4.3.7.1 Asynchronous reset

There are times when an asynchronous reset is acceptable, or even preferred. If the vendor's library includes asynchronously reset-able flip-flops, the reset input can be tied to a master reset in order to reduce the routing congestion and to reduce the logic required for a synchronous reset. FPGAs and CPLDs will typically have master reset signals built into the architecture. Using these signals to reset state machines frees up interconnect for other uses.

Asynchronous reset should be used only for resetting the entire chip and should not occur during normal functioning of the chip. After reset, you must ensure that the chip is in a stable state such that no flip-flops will change until an input changes. You must also ensure that the inputs to the chip are stable and will not change for at least one clock cycle after the reset is removed.

4.3.7.2 Asynchronous latches on inputs

Some buses, such as the VME bus, are designed to be asynchronous. In order to interface with these buses, it is necessary to use asynchronous latches to capture addresses or data. Once the data is captured, it must be synchronized to the internal clock. However, it is usually much more efficient to use asynchronous latches to capture the data initially. Unless your chip uses a clock that has a frequency much higher than that of the bus, attempting to synchronously latch these signals will cause a large amount of overhead and may actually produce timing problems rather than reduce them.

4.4 Floating Nodes

Floating nodes, or internal nodes of a circuit which are not continually driven, should be avoided. An example of a potential floating node is shown in Figure 26. If signals SEL_A and SEL_B are both not asserted, signal OUT will float to an unknown level. Downstream logic may interpret OUT as a logic 1, a logic 0, or it may produce a metastable state. In addition, any CMOS circuitry

that has OUT as an input will use up power since CMOS uses power when the input is in the threshold region.

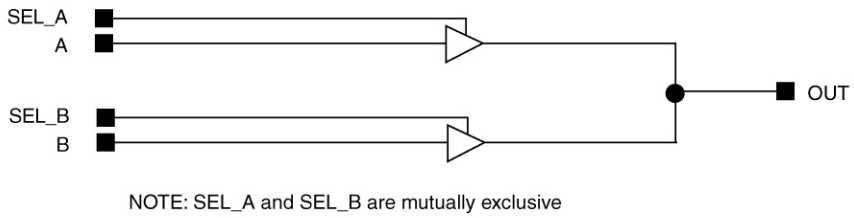

Figure 26 Floating Nodes - The Problem

Two solutions to the floating node problem are shown in Figure 27. At the top, signal OUT is pulled up using an internal pull-up resistor. This ensures that when both select signals are not asserted, OUT will be pulled to a good logic level. The other solution, shown at the bottom of the figure, is to make sure that something is driving the output at all times. A third select is generated which drives the output to a good level when neither of the select signals is asserted.

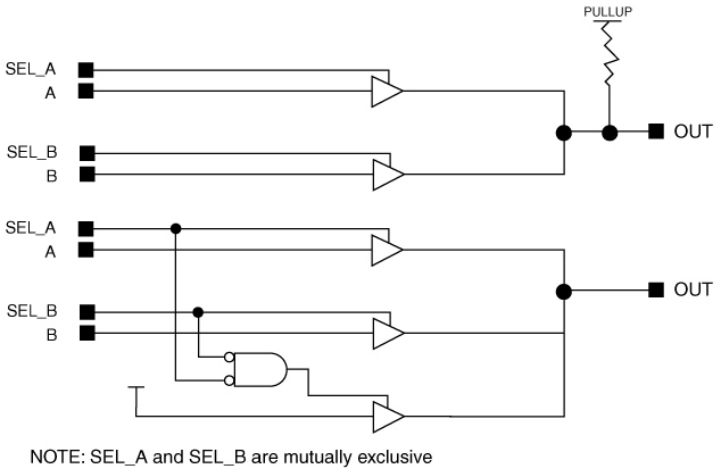

Figure 27 Floating Nodes - Solutions

4.5 Bus Contention

Bus contention occurs when two outputs drive the same signal at the same time as shown in Figure 28. For obvious reasons, this is bad and reduces the reliability of the chip. If bus contention occurs even for short times during a clock cycle, after many clock cycles the possibility of damage to one of the drivers increases. The solution is to ensure that both drivers cannot be

asserted simultaneously. This can be accomplished by inserting additional logic as shown in Figure 29. The ideal solution is to avoid tri-state drivers altogether, and use muxes whenever possible.

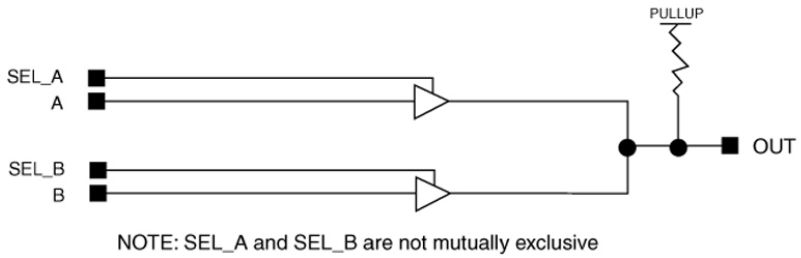

Figure 28 Bus Contention - The Problem

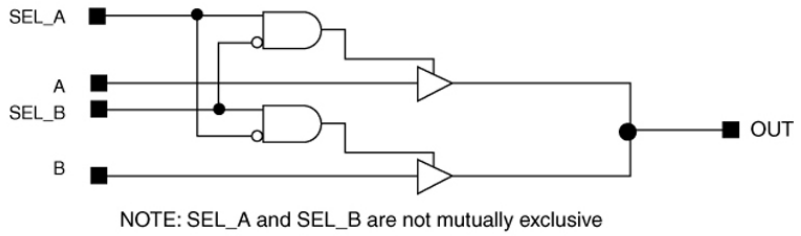

Figure 29 Bus Contention - The Solution

4.6 One-Hot State Encoding

For large grain FPGAs, which are the majority of architectures available, the normal method of designing state machines is not optimal. This is because the each CLB in an FPGA has one or more flip-flops, making for an abundance of flip-flops. For large combinatorial logic terms, however, many CLBs are often involved which means connecting these CLBs through slow interconnect. A typical state machine design, like the one shown in Figure 30, uses few flip-flops and much combinatorial logic. This is good for ASICs, bad for FPGAs.

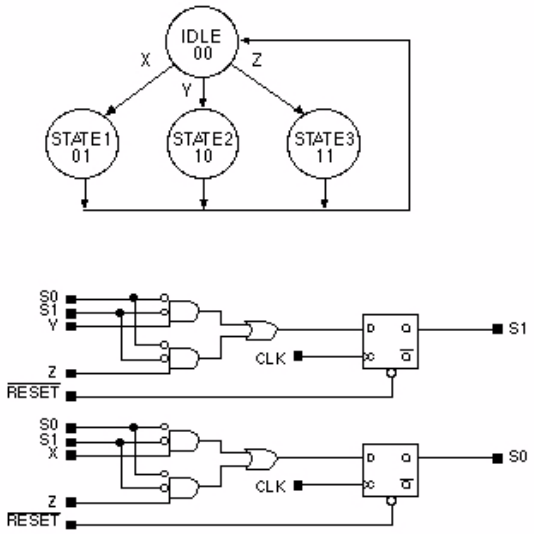

Figure 30 State Machine: Usual Method

The better method of designing state machines for FPGAs is known as one-hot encoding, seen in Figure 31. Using this method, each state is represented by a single flip-flop, rather than encoded from several flip-flop outputs. This greatly reduces the combinatorial logic, since only one bit needs to be checked to see if the state machine is in a particular state. It is important to note that each state bit flip-flop needs to be reset when initialized, except for the IDLE state flip-flop that needs to be set so that the state machine begins in the IDLE state.

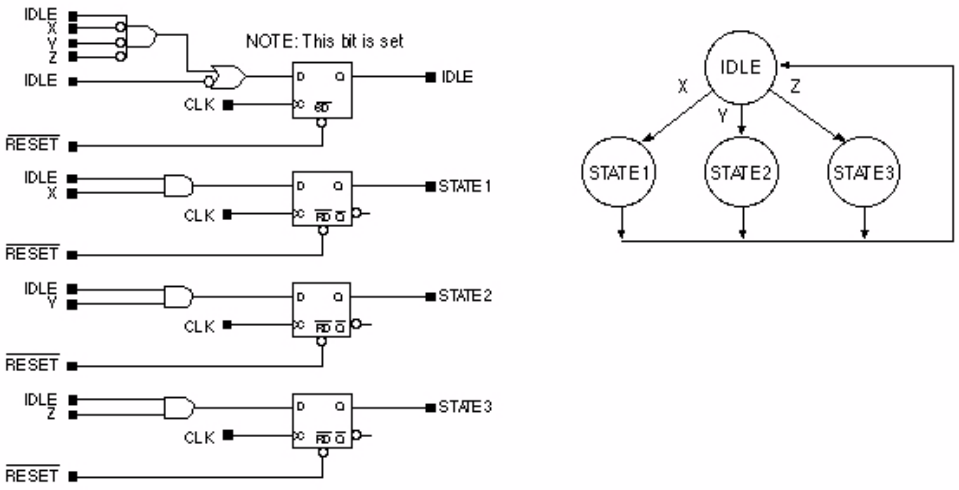

Figure 31 State Machine: One-Hot Encoding

5. DESIGN FOR TEST (DFT)

"Design for test" is a concept that means your chip is designed in such a way that testing it is easy. Test logic plays two roles. First, it helps debug a chip that has design flaws. Second, it can catch manufacturing problems. Both are particularly important for ASIC design because of the black box nature of ASICs where internal nodes are simply not accessible to you when there is a problem. These techniques are also applicable to CPLDs and FPGAs, many of which already have built-in test features. The following DFT techniques allow for better testing of a chip. While not all of these techniques need to be included in your design, those that are needed should be included at design time. DFT techniques should be taken into account during the design process rather than afterwards. Otherwise, circuits can be designed that are later found to be difficult, if not impossible, to test.

One important consideration that can be overlooked is that test logic is intended to increase the testability and reliability of your chip. If test logic becomes too large, it can actually decrease reliability because the test logic can itself have problems that cause the chip to malfunction. A rule of thumb is that test circuitry should not make up more than 10% of the logic of the entire chip. Similarly, if you spend more than 10% of your time designing and simulating your test logic independently of the functionality of the chip, then you have more test circuitry than you need.

5.1 Testing Redundant Logic

The top of Figure 32 shows a circuit that has duplicated logic in order to increase the reliability of the design. However, since the circuit is not testable, the effect is not as useful as it could be. The circuit on the bottom shows how test lines can be added to allow the entire circuit to be tested.

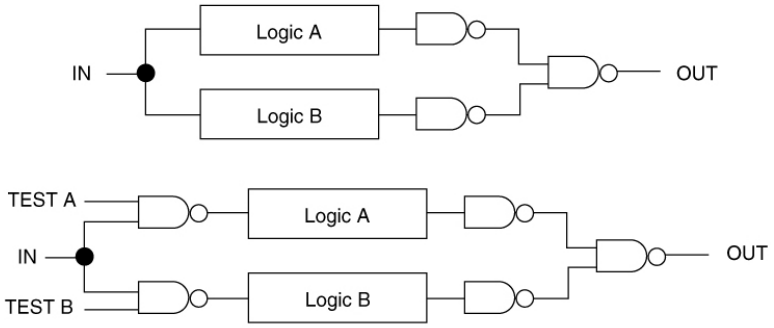

Figure 32 Testing Redundant Logic

5.2 Initializing State Machines

It is important that all state machines, and in fact all registers in your design be able to be initialized. This ensures that if a problem arises, the chip can be put into a known state from which to begin debugging. Also, for simulation purposes, simulation software needs to start out from a known state before useful results can be obtained.

5.3 Observable Nodes

As many nodes as possible in your chip design should be observable. In other words, it should be possible to determine the values of these nodes using the I/O pins of the chip. On the left side of Figure 33, an unobservable state machine is shown. On the right side, the state machine has been made observable by taking each state machine through a mux to an external pin. Test signals can be used to select which output is being observed. If no pins are available, the state bits can be muxed onto an existing pin that, during testing, is used to observe the state machine. This allows for much easier debugging of internal state machines.

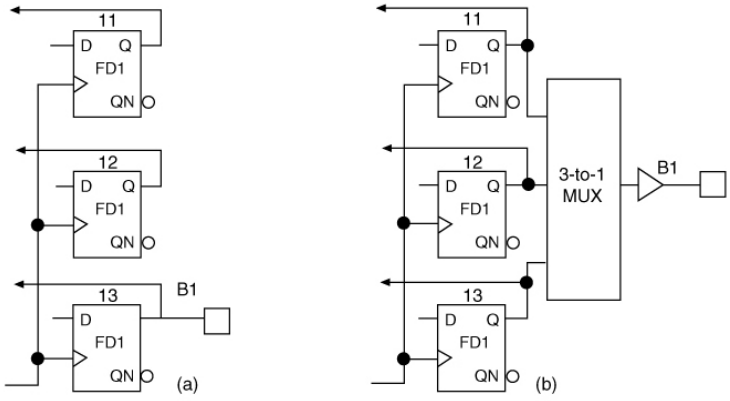

Figure 33 Observable Nodes

5.4 Scan Techniques

Scan techniques, shown in Figure 34, allow the nodes of the chip to be scanned out so that they can be observed externally. There are two main scan techniques - full scan and boundary scan. Full scan is extremely flexible, especially since it can also allow values to be scanned into the chip so that you can start it from a known state. This is particularly useful if a problem occurs only after the chip has been operating for a long time. A state can be quickly scanned into the chip that corresponds to the state that would normally be reached after a long time in operation. The drawback of scan techniques is that they require a lot of software development to support. Also, if states are scanned into the chip, you must be careful not to scan in illegal states. It is

possible to turn on multiple drivers to a single net internally which would normally not happen, but which would burn out the chip. Similarly, outputs must be disabled while the chip is being scanned since dangerous combinations of outputs may be asserted that can harm your system. There are other considerations, also, such as what to do with the clock and what to do with the rest of the system while the chip is being scanned.

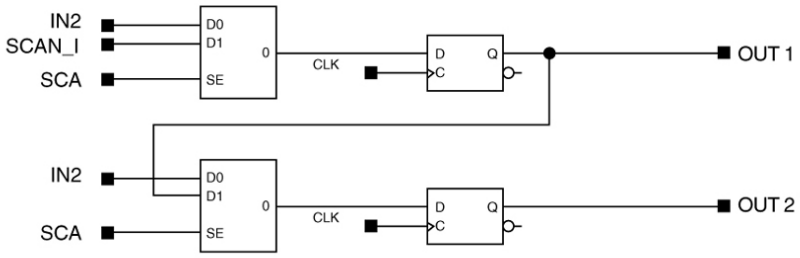

Figure 34 Scan Methodology

Boundary scan is somewhat easier to implement and does not add as much logic to the entire chip design. Boundary scan only scans nodes around the boundary of the chip, but not internal nodes. In this way, internal contention problems are avoided, although contention problems with the rest of the system still need to be considered. Boundary scan is also useful for testing the rest of your system, since the outputs can be toggled and the effect on the rest of the system observed.

5.5 Built-In Self Test

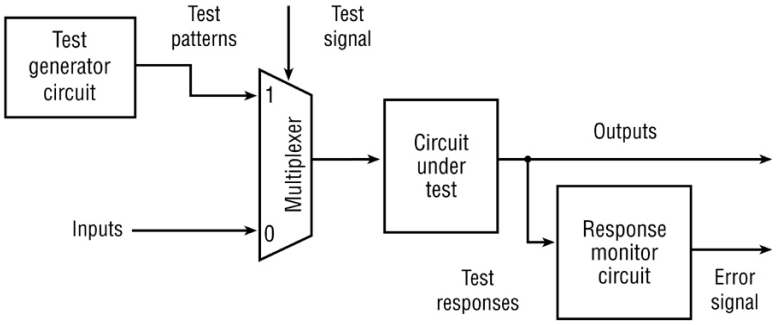

Figure 35 Built-In Self Test

Another method of testing your chip is to put all of the test circuitry on

the chip in such a way that the chip tests itself. This is called built-in self test or BIST. In this case, some circuitry inside the chip can be activated by asserting a special input or combination of inputs. This circuitry then runs a series of test on the chip. If the result of the tests does not match the expected result, the chip signals that there is a problem. The details of what type of tests to run and how to signal a good or bad chip is left up to the designer.

5.6 Signature Analysis

Signature analysis involves putting a pseudo-random sequence of ones and zeroes into the chip and noting the ones and zeroes that come out. This output sequence is referred to as the chip's signature. This type of testing can be accomplished with the chip in a normal mode of operation, but is usually performed in scan mode as described above. By repeating the same pseudo-random series of bits, the resulting signature should be the same for each chip. Any chip that produces an incorrect signature is a bad chip. This type of testing is probabilistic and assumes that a pseudo-random sequence of events has a good chance of catching errors, which may not be true. However, it requires very little hardware to implement and can be used as a simple form of BIST.

6. SIMULATION ISSUES

Perhaps the most important phase of chip design, and the most often overlooked phase, is that of simulation. Simulation can save many frustrating hours debugging a chip in your system. Doing a good job at simulation uncovers errors before they are set in silicon, and can help determine that your chip will function correctly in your system.

There are two main aspects of your design for which simulation is used to determine correctness - functionality and timing. Functionality refers to how the chip functions as a whole, and how it functions in your system. A chip that is designed to function as an Ethernet controller may function correctly on its own. In a system that requires an ATM controller, for example, it will not work at all. It is important to look not only at the functionality of the chip as an independent design, but also to test its functionality within the system in which it will be incorporated.

The second aspect of your design which simulation examines is timing. Will your chip meet all of its timing requirements under all possible conditions? Are there any race conditions? Are the setup and hold time requirements met for each flip-flop? Do the I/O signals of the chip meet the timing requirements of the system? The following sections discuss ways of using timing to determine both correct functionality and correct timing.

6.1.1 Functional Simulation

Functional simulation involves simulating the functionality of a device without taking the timing of the device into account. This type of simulation is important initially in order to get as many bugs out of the device as possible and to determine that the chip will work correctly in your system. During the first phases of simulation, you shouldn't be very concerned about timing because it will change as the design changes. In fact, the final timing will not be known precisely until the layout is complete. Of course you need to know initially that, in general, the timing of the chip process can support the speed and the I/O requirements of your design.

When performing functional simulation, a rough estimate of the amount of simulation to perform is called toggle coverage, which measures the percentage of flip-flops in the chip that change state during simulation from 0 to 1 and 1 to 0. Many simulation packages will give you a number for the toggle coverage, and you should have 100 percent coverage before feeling good about the amount of simulation. This coverage can still leave many potential faults uncovered, but it signifies that each state machine has been simulated and no part of the circuit has gone unexamined.

Toggle coverage is primarily used for schematic based designs, which are rare these days. The equivalent check for designs using HDLs is called code coverage, which measures the percentage of possible code statement branches that have been executed. In other words, an assignment statement is completely covered if it is executed at all, while a branch statement is completely covered if all possible branches are taken during the simulation.

6.1.2 Static Timing Analysis

Static timing analysis is a process that looks at a synchronous design and determines the highest operating frequency of the design that does not violate any setup and hold times. You can also use the static timing analysis software to specify a specific frequency, and the tool will list all paths that violate the timing requirements. These paths can then be adjusted to meet your requirements. Any asynchronous parts of your design (they should be few, if any) must be examined by hand.

Static timing analysis, or some sort of timing analysis must be performed immediately before layout of your chip. At this point, the timing numbers will be estimates that take expected trace lengths into account. After layout, timing analysis must be performed again to determine that the real chip, with real trace lengths and delays, still meets you timing requirements.

6.1.3 Timing Simulation

This method of timing analysis is growing less and less popular. It

involves including timing information in a functional simulation so that the real behavior of the chip is simulated. The advantage of this kind of simulation is that timing and functional problems can be examined and corrected. Also, asynchronous designs must use this type of analysis because static timing analysis only works for synchronous designs. This is another reason for designing synchronous chips only.

As chips become larger, though, this type of compute-intensive simulation takes longer and longer to run. Also, simulations can miss particular transitions that result in worst-case results. This means that certain long delay paths never get evaluated and a chip with timing problems can pass timing simulation. If you do need to perform timing simulation, it is important to do both worst-case simulation and best-case simulation. The term “best-case” can be misleading. It refers to a chip that, due to voltage, temperature, and process variations, is operating faster than the typical chip. However, hold time problems become apparent only during the best-case conditions.

7. EMERGING TECHNOLOGIES

7.1 Cores

By a “core” we are simply referring to the basic function, excluding any extraneous circuits like I/O buffers that would be found on a processor chip. There are two types of cores. The soft core, known as an IP core, is a function that is described by its logic function rather than by any physical implementation. Cores usually consist of HDL code. Hard cores, on the other hand, consist of physical implementations of a function. With respect to CPLDs and FPGAs, these hard cores are known as embedded cores because they are physically embedded onto the die and surrounded by programmable logic.

Many of the FPGA and CPLD vendors have begun offering cores. As the density of programmable devices increases, it is enabling what is called a System on a Programmable Chip (SOPC). In other words, whereas programmable devices were initially developed to replace glue logic, entire systems can now be placed on a single programmable device. Systems consist of all kinds of complicated devices like processors. In order to place these complex functions within a programmable device, there are three options — design the function yourself, purchase the HDL code for the function and incorporate it into your HDL code, or get the vendor to include the function as a cell in the programmable device. The second option is the IP core while the third option is the embedded core.

7.1.1 IP Cores

IP cores are often sold by third party vendors that specialize in creating

these functions. Recently, CPLD and FPGA vendors have begun offering their own soft cores. IP cores reduce the time and manpower requirements for the FPGA designer. IP cores have already been designed, characterized, and verified. Also, IP cores can often be modifiable, meaning that you can add or subtract functionality to suit your needs.

But IP cores may also be expensive. IP cores can be optimized to a certain degree, but the complete optimization depends on its use in a particular device and also depends on the logic to which it is connected. Such IP purchased from a third party may not be optimized for your particular CPLD or FPGA vendor. You may not be able to meet your speed or power requirements, especially after you have placed and routed it.

7.1.2 Embedded Cores

The embedded core is in many ways ideal for users, which is one reason why programmable device vendors are now offering embedded cores in their devices. The embedded core will be optimized for the vendor's process to give you good timing and power consumption numbers. The function will be placed as a single cell on the silicon die and so the performance of the function will not depend on the rest of your design since it will not need to be placed and routed.

Some embedded cores are analog devices that cannot be designed into an ordinary CPLD or FPGA. By integrating these functions into the device, you can avoid the difficult process of designing analog devices, and you save the chips and components that would otherwise be required outside the programmable device.

Of course there is a drawback to embedded cores. By using an embedded core in your programmable device, you tie your design into a single vendor. Unless another vendor offers the same embedded core, which is unlikely, switching to another vendor will require a large effort and will not be pleasant.

Another reason for offering embedded cores is a business reason. There are essentially two major players in the CPLD and FPGA markets — Xilinx and Altera. The smaller players have tried for years to compete with the result, generally, that their market share has remained flat or shrunk. In order for the smaller vendors to differentiate themselves from the big two, they need to find a niche market that they can dominate. These niche markets support those designs that need a very specific function. I should say that these niche markets might turn out to be very big. However, it is a bet-the-house risk, especially for the smaller companies. If a small company puts a lot of resources into developing and marketing a programmable device that includes a specific processor that ends up being designed into every personal computer, then that

vendor can see a significant amount of sales. But if the vendor bets on the wrong processor, they could lose a huge amount of R&D money and get little revenue in return. This isn't as big a risk for the large vendors because they have more resources, more sales channels, and more cash to quickly change directions and develop new families of devices.

7.1.3 Processor cores

Processor cores are one of the types of cores commonly available as IP cores or embedded cores. These processors tend to be those that are designed for embedded systems since, almost by definition, programmable devices are embedded systems.

If the processor core is embedded, you will be using a processor that has been optimized and has predictable timing and power consumption numbers. For either type of core, tools will be readily available for software development. Off-the-shelf cross compilers and simulators can be used to debug code before the design has been completed and the programmable device is available.

An example of an FPGA with an embedded processor, along with other embedded cores, is shown in Figure 36.

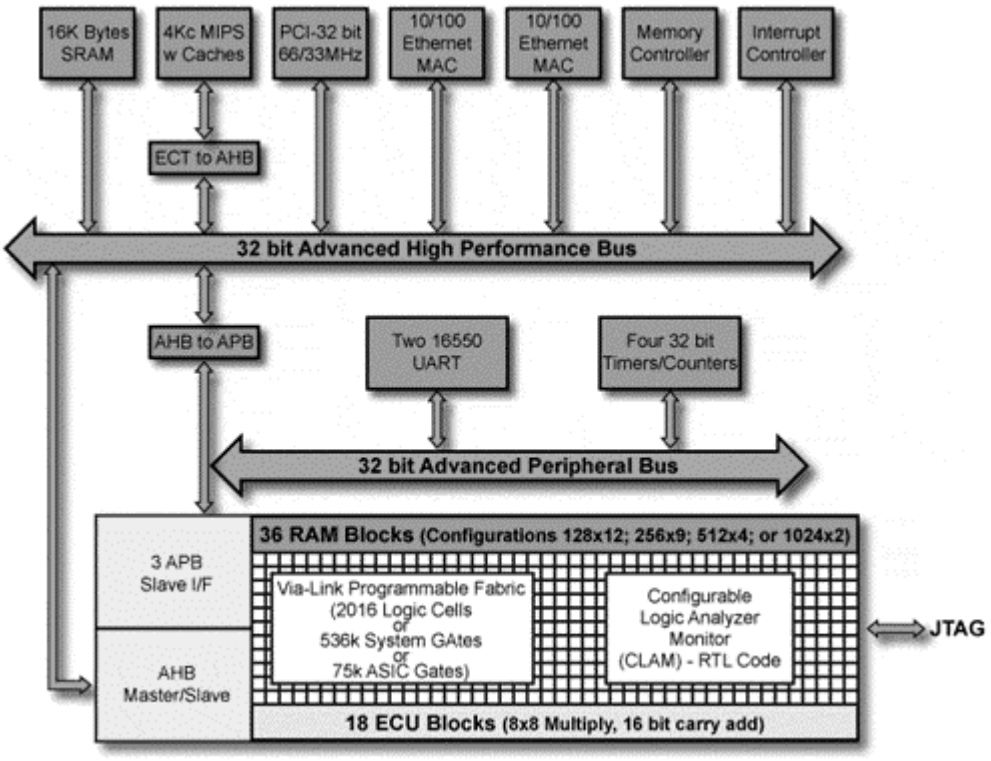

Figure 36 FPGA with embedded processor core

7.1.4 DSP cores

Digital Signal Processors (DSPs) are another common type of core that is offered as an IP core or an embedded core. These are essentially specialized processors that are used for manipulating analog signals. They are commonly used for filtering and compression of video or audio signals. Many engineers have argued that as general processors become faster, DSPs will be less useful because the same functions can be accomplished on the generic processors. However video and audio digitization, compression, and filtering requirements have increased in recent years as millions of users connect to the Internet and regularly upload and download all kinds of information over relatively limited bandwidth connections. So far, DSP demand for use in networking and graphics devices has been increasing, not decreasing.

7.1.5 Embedded PHY cores

PHY cores are the analog circuitry that drives networks. Many companies are now integrating this functionality onto their devices. Because these devices include specialized analog circuitry, they are available only as embedded cores.

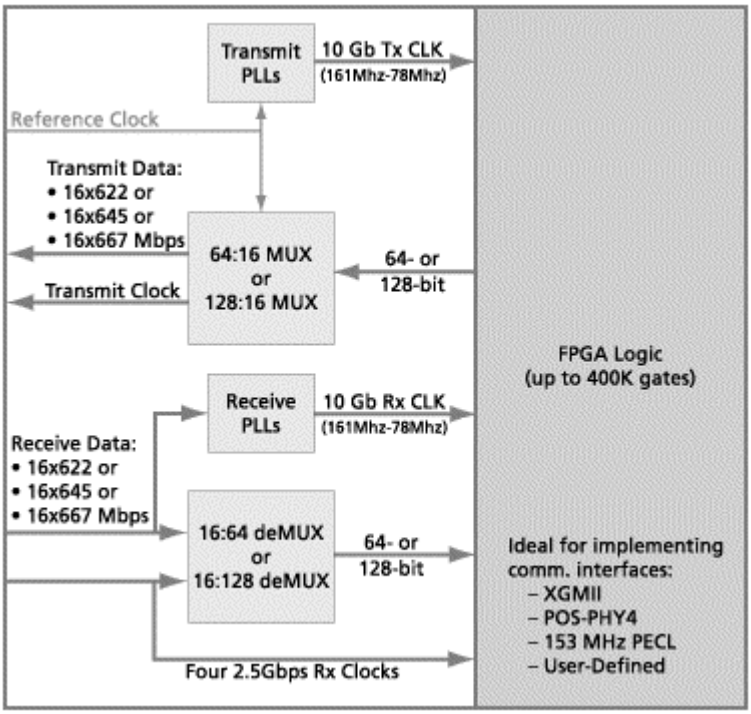

Figure 37 FPGA with embedded PHY core

In the late nineties, during the heyday of the Internet, networking companies were springing up all over. In order to save design time, these

companies could use FPGAs with PHY cores built in. Unfortunately, this boom didn't last and some networking technologies did not find the mass acceptance that was predicted. For engineers designing an interface to a specific type of network, an FPGA with the appropriate PHY core can be a very good resource. For the programmable device vendor, it can be something of a risk to support a particular PHY core that may not end up being the standard that they expect or have the mass-market acceptance that they are counting on.

Figure 37 shows an example of an FPGA with an embedded PHY core that can be programmed to interface to a variety of different networks.

7.2 Special I/O drivers

Special I/O drivers are now being embedded into programmable devices. The newer buses inside personal computers need to be driven by special high-drive, impedance-matched circuits. They need to have inputs with very specific voltage threshold values. Many vendors now offer programmable devices with I/O that meet these special requirements. Many times, this is the only way to design a programmable device that can interface with these devices.

7.3 New Architectures

New architectures are being developed for CPLDs and FPGAs. There are still occasional attempts to create a fine grain architecture where the logic blocks consist of small logic functions. Most of these attempts, I believe, are doomed to failure because routing is still the main constraint in any FPGA. Fine grain architectures require more routing than large grain architectures.

One type of architecture that is being developed for FPGAs has a logic block that it based on a DSP. In Figure 38, we see such a logic block. This type of FPGA will be better for use in chips that need a significant amount of signal processing. I have certain doubts about this future path, though. First, the majority of programmable devices do not perform any DSP, so this architecture targets a relatively small market. Second, special tools will be needed to convert digital signaling algorithms for use in such a specialized FPGA. These tools will need to optimize the algorithm very well so that performance in this specialized FPGA can actually perform better than a standard DSP, or a generic processor, running code that has been optimized using tools and compilers that have been available for years.

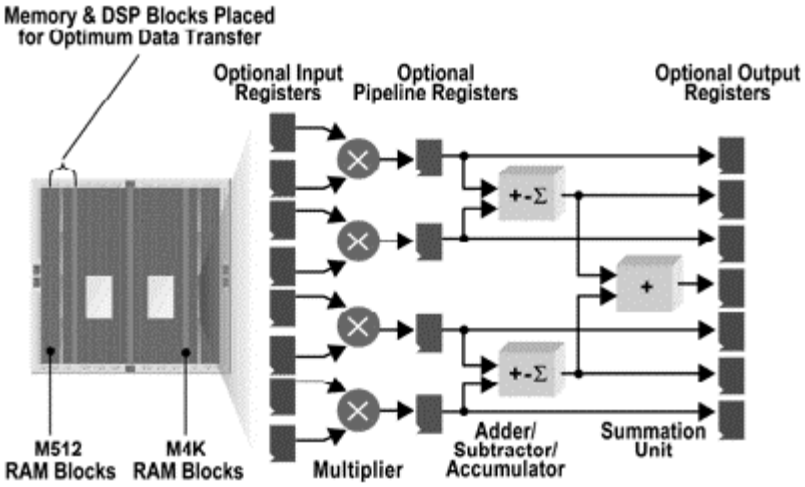

Figure 38 DSP core cell in an FPGA

7.4 ASICs with embedded FPGA cells

A relatively new concept that has taken hold in the imaginations of some established FPGA vendors and some new startup companies is to embed FPGAs into ASICs. There are two ways of doing this. One way is to create small programmable cells of logic that can be used in an ASIC. These cells would be similar to the configurable logic blocks of an FPGA and could be placed, along with hard logic cells, anywhere on an ASIC. The other way is to embed an FPGA core into an ASIC and allow logic to be placed around this core.

The technology of providing FPGA cells for ASIC designs is an interesting one. I don't have a good feel for the size of this market, though I feel that there definitely is a market. There are several specific areas where I see potential.

1. **Cost reduction.** For engineers who are already designing systems that include both ASICs and FPGAs, putting FPGA cells inside the ASIC combines multiple chips into one hybrid chip. This will result in a significant cost savings by eliminating chips. For engineers who are considering a design that includes ASIC technology and FPGA technology, this solution saves PC board space, and the resulting hybrid chip will generally require fewer external pins because the ASIC/FPGA interface is now inside the chip. Smaller PC boards results in lower cost. More importantly, lower pin count on a chip results in significantly lower costs because package size is a large percentage of the overall per-piece cost of an ASIC.
2. **Changing communication protocols.** We've already seen flash memory technology used extensively in modem designs so that the

modems could be released before a communication protocol was finalized. This gave modem manufacturers that used this technology a head start in the market. When the protocol was finalized, the user simply needed to update the modem firmware. This technology can be used in switches and routers and other complicated communication devices in the same way. Network device manufacturers can ship devices before a protocol is fully defined. Of course, they can do that now using discrete FPGAs in their design, but this technology offers cost advantages by placing all logic, both fixed and flexible, onto a single chip.

3. **Bus interfaces and memory interfaces.** These are other areas that are good candidates for this technology. The FPGA functionality allows the engineer to fine tune the logic while it is in the field. I believe that the opportunity for this kind of market exists for very new interfaces that may not be well defined or for which accurate simulation models don't yet exist. However, I also believe that accurate simulation models exist for older, well-defined interfaces and so the technology will not be applied as much for supporting these legacy interfaces.
4. **Architecture enhancements.** One interesting idea that this technology further enables is the ability to make architectural changes after a product has been manufactured and shipped. In my experience, very little analysis of complex equipment is performed to locate performance bottlenecks. This technology enables changes to a system's architecture to be tested in the field. Those changes that resulted in better operation can be incorporated into the design. It may also be that different uses of a device may require different designs. A device can be customized for particular customers based on their environment and requirements.
5. **Reconfigurable computing.** The concept of using FPGA devices to perform some of the algorithmic work of a general-purpose computer has excited researchers for several years. Currently, the work is mostly confined to universities and R&D labs because of the complexity and challenges from the design of the software and the hardware. In particular, it has been difficult to develop compilers or interpreters that can take general algorithms, written in general programming languages like C, and map the functionality onto reconfigurable hardware. Should these issues be resolved, and reconfigurable computing becomes successful, this technology could

be an ideal platform for it because it enables the tight integration of high-speed logic and reconfigurable logic on the same chip.

The example in Figure 39 shows a block diagram of an implementation of a 32-tap FIR filter. The shaded blocks are implemented in FPGA cells while the unshaded blocks are implemented in ASIC cells. The RAM is much easier to implement, and more efficient to implement, as a RAM cell than in an FPGA. By implementing the Address Generator and ROM in FPGA cells, the algorithm can easily be reprogrammed.

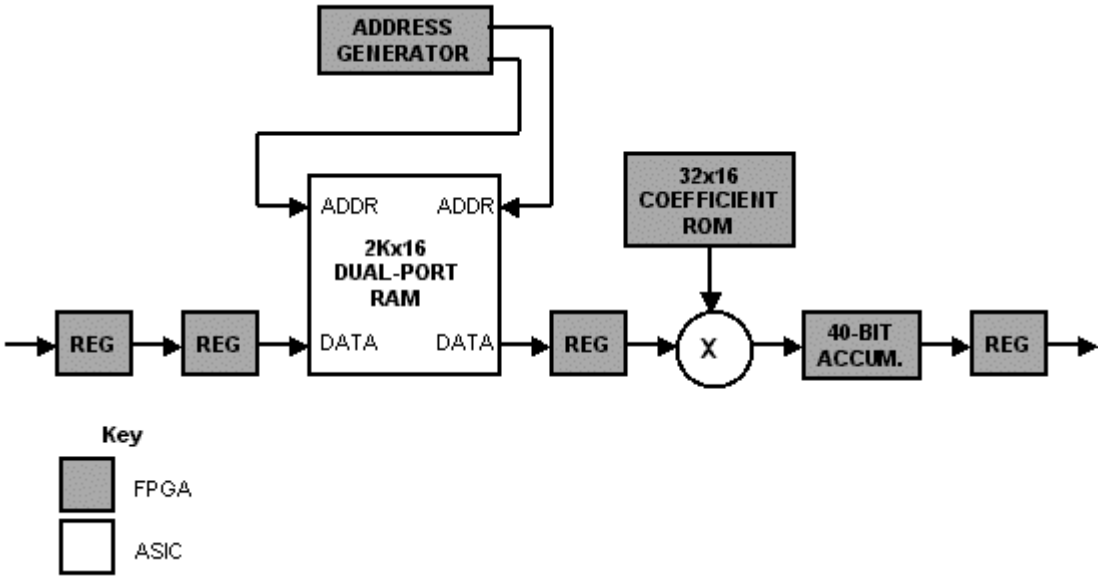

Figure 39 Mixed ASIC/FPGA design

8. NEW TOOLS

The most significant area for the future, I believe, lies in the creation of new development tools for FPGAs. As programmable devices become larger, more complex, and include one or more processors, there is a huge need for tools to take advantage of these features and optimize the designs.

Hardware designers can use hardware description languages (HDLs) like Verilog to design their chips at a very high level. They then run their synthesis and layout tools that optimize the design.

As FPGAs come to incorporate processors, the development tools need to take software into account and need to optimize at a higher level of abstraction. Hardware/software codesign tools will be a necessity.

Ultimately, there will have to be a melding of hardware and software expertise in an FPGA designer. System level issues must be understood and addressed, though perhaps not the particulars of FPGA routing resources or

operating system task switching. Intelligent tools will be needed to synthesize and optimize software just as it is now used to synthesize and optimize hardware. These intelligent tools will work with libraries of pre-tested hardware objects and software functions, leaving “low-level” C and Verilog design necessary only for unique, specialized sections of hardware or software.

Software developers and their tools will be affected by this integration too. To take full advantage of the hardware components in the programmable arrays around them, compilers and RTOSes will need to make such integration more seamless. If dynamic reconfigurability ever becomes commonplace, a future RTOS may even get into the business of scheduling, placement, and routing of hardware objects—perhaps treating them as distinct tasks with communication mechanisms not unlike software tasks.

Essentially, platform FPGAs with embedded processors will take market share away from ASICs, will become the dominant platform for embedded system design, and will finally allow the fulfillment of the promise of and force further development of hardware/software codesign tools.

9. CONCLUSION

This paper has presented an overview of CPLD and FPGA technologies, and given guidelines for developing a chip based on my experience designing for a large number of companies and a large number of applications. If all of these guidelines are followed, the chances of creating a working chip in a short time at minimum expense are excellent.

Bob Zeidman is the president of Zeidman Consulting (www.ZeidmanConsulting.com), a contract research and development firm. Since 1983, he has designed ASICs, FPGAs, and PC boards for RISC-based parallel processor systems, laser printers, network switches and routers, and other real time systems. His clients have included Apple Computer, Cisco Systems, Intel, Quickturn Design Systems, and Texas Instruments. Among his publications are technical papers on hardware and software design methods as well as three textbooks — *Designing with FPGAs and CPLDs*, *Verilog Designer's Library*, and *Introduction to Verilog*. He has taught courses at engineering conferences throughout the world. Bob earned bachelor's degrees in physics and electrical engineering at Cornell University and a master's degree in electrical engineering at Stanford University.