FPGA DESIGN RELIABILITY I

This paper addresses how to build a robust FPGA based design. What does robust mean? It means designing an FPGA that works predictably and reliably. All the logic, all the clever algorithms, all the computational engines and data communication paths must build upon an FPGA that works reliably due to good design practices. This paper introduces the architectural aspects of FPGAs (as opposed to ASICs), and provides an introduction to synchronous design techniques.

1. FPGA Architectural Issues

FPGAs and ASICs share many of the same architectural issues, some of which are listed below:

- Selecting an appropriately-sized part
- Partitioning logic
- Considering external interfaces
- Power dissipation

However, FPGAs differ substantially from ASICs in one architectural area. This warrants further discussion.

The structure of a typical FPGA places a small amount of combinatorial logic upstream from a flip-flop. These two elements are tightly coupled in close proximity within a structure called a logic cell, logic element, or logic block. While there is very little delay between the flip-flop and its associated logic, that logic is quite limited. So, if a more complex combinatorial function is required, the combinatorial logic from multiple logic cells must be interconnected.

The FPGA's logic cells are interconnected through configuration of the FPGA's routing resources. Because this routing passes through active switches, it is much slower than the metal interconnect paths in ASICs. Therefore, the propagation time through a large combinatorial circuit is longer in an FPGA than in an ASIC.

Fortunately, FPGAs are register-rich; there is typically one flip flop for each small combinatorial logic function. This allows FPGA designers to break up large combinatorial functions, placing registers in between stages. This is called pipelining, a technique which goes hand-in-hand with synchronous design (discussed later in this document). The goal of pipelining is to reduce the propagation delay associated with combinatorial logic, allowing the entire circuit to be clocked faster. The frequent registers allow intermediate results from each combinatorial stage to be synchronized before they are passed on to the next stage.

Pipelining allows FPGAs to approach or match the clock speed of custom ASICs. However, while the clock rate may be the same, more register stages will be required than used for an equivalent circuit in an ASIC. Therefore, it will take more clock periods to move a single piece of data through the circuit. In other words, while the data *throughput* is the same as that of the ASIC, the data *latency* increases. This is usually not a problem as long as the designer plans for the situation at the start of the design process – in the architectural phase.

An experienced FPGA designer never loses sight of the register-rich, slow-combinatorial-logic aspects of these devices. This mental bias should affect architectural planning, latency budgets, performance estimates, and design entry for the FPGA.

A special comment should be made about CPLDs. CPLDs are not nearly so register-rich as FPGAs, while they are structured in a way that allows faster results from some complex combinatorial functions. However, their performance is slower to that of ASICs, because their signal routing uses techniques that are slower than the metal interconnections of ASICs. To best utilize both FPGAs and CPLDs a designer must understand their internal architectures by thoroughly reading the available data sheets and design guides.

2. FPGA Design-Related Issues

Although many of the topics discussed in this section are common between ASIC and FPGA design, they are mentioned in some detail because FPGA designers seem to frequently overlook these considerations. This may be due to the gradual transition many designers have made from PCB design, to use of small programmable parts (PALs, PLDs, etc.), and finally to large FPGAs.

2.1.1. Synchronous vs. Asynchronous Design Practices

A fundamental requirement of reliable FPGA design is the use of synchronous logic design. Conversely, the most common errors fall into the category of asynchronous design practices.

Asynchronous circuitry often results in the occurrence of a clock transition at a synchronous element (i.e. a flip-flop) just as the data input is changing. This hazard results in the output of the flip-flop becoming unknown. The output may keep its previous value, take on the input value, for a time go to an intermediate voltage level between a logic high and logic low, or oscillate between logic high and logic low levels. These symptoms, collectively, are known as metastable events, or metastability. The lack of a deterministic output state propagates to downstream logic, creating occasional data errors observable at the outputs of the FPGA.

Errors produced by metastability occur irregularly. Their frequency is subject to process characteristics of the FPGA, as well as device voltage and temperature conditions. Observable errors are also data-dependent. These factors often make it difficult and frustrating to identify the source of the problem. Making matters worse, an engineer who is unfamiliar with the conditions causing metastability or design techniques to present them is unlikely to even suspect the root cause of the observable errors.

Asynchronous design practices include improper clock routing, use of derived clocks, use of more clock domains than necessary, improper crossing of clock domains, and improper reset techniques. These issues will each be discussed. Following each issue discussion is a paragraph or set of paragraphs providing design guidance for improving reliability.

2.1.1.1. Improper Clock Routing

In any digital circuit, clock routing must be carefully designed to ensure that the data input to a flip-flop is stable during a window of time surrounding a clock transition. The time interval just preceding the clock transition is known as “data setup time,” or simply, “setup time.” The interval following the clock transition, during which the data must remain stable, is known as “data hold time” or “hold time.” The device manufacturer publishes these times, which vary with the device type, and may vary from signal to signal. However, if either setup or hold times are violated, the behavior of the flip-flop is unpredictable. Metastability may result.

Signal delay is a combination of the propagation delay through active circuitry and the transmission delay of interconnect paths on the FPGA die. In an FPGA, both data signals and clock signals experience delays from both sources. Improper routing of the clock leads to excessive clock signal delay, which can result in violation of data hold time.

As an example, imagine flip-flops A and B, clocked by the same clock signal. The output of flip-flop A is routed to the input of flip-flop B. When A is clocked, the transition of its output data takes a finite amount of time to propagate to B. However, what happens if the routing of the clock signal between flip-flops A and B is such that the clock transition gets to B well after it gets to A? What will happen if this clock "skew" is equal to, or greater, than the propagation of the data signal? The result is that the data hold time will be violated, and metastable events will occur.

This illustrates the problem of using slow or high-skew routing resources (which are intended to route data signals) to route a clock signal. Although it is possible to use this "general interconnect" for clock routing, doing so safely requires special considerations. In general, dedicated global clock networks should be used for clock routing, which will eliminate the skew that produces metastability

2.1.1.1.1. Design Guidance: Proper use of FPGA Clock Resources

FPGA manufacturers have gone to great lengths to ensure that circuits will run reliably, as long as synchronous design practices are followed. A number of techniques that comprise synchronous design are discussed in this section.

As previously stated, clock skew is minimized when using global clock nets. Thus, it is imperative to use the dedicated clock buffer and clock nets. Besides reducing skew, use of a global network is designed to "cooperate" with the manufacturer's place and route software. Signals produced by flip-flops driven by the same global clock net will be routed in a way that guarantees that the clock-to-output delay (t_{co}) of a source flip-flop, plus the minimum propagation delay of the routing path to any destination flip-flop, will equal or exceed the worst case clock skew plus the hold time required at all destination flip-flops. Another way of saying this is that the routing software guarantees that metastability will <u>not</u> occur – as long as a global clock net is used to route the clock, and as long as signals stay within one clock domain. Therefore, the designer should use global clock nets exclusively, if possible.

In addition to global clock buffers and clock nets, modern FPGAs provide a host of specialized clock resources, including Phase-Locked Loops (PLLs), Delay-Locked Loops (DLLs), Frequency Synthesizers, Clock Multiplexers, Phase Shifters, Rational-Rate Multipliers, and similar mechanisms to multiply, divide, delay, or smoothly switch the clock. It is imperative that the designer thoroughly read the manufacturer's literature on the correct use of these features.

If two clock domains are created by means of one of these devices, and if these domains have a known phase relationship, then it is safe to pass signals from one domain to another without fear of metastability, provided the relationship between clock edges in the different domains are taken into consideration. As an example, if two clock domains have a phase locked frequency ratio of 1:2, and they are phase aligned, then data can be passed between the two domains on any clock edge, **as long as the signals crossing domains reach their destination flip-flops in the time required by the clock period of the faster domain.** In this case, the routing software is guided by a timing constraint written by the designer. This constraint will specify, as a delay, the period of the faster clock domain and those particular signals to which that delay applies.

In general, to determine the constraints that must be placed on signals crossing domains, analysis must be done on a case-by-case basis. Likewise, when a clock signal is intentionally phase-shifted in one domain relative to another, the designer should determine the appropriate constraints on domain-crossing signals by a careful timing analysis.

2.1.1.1.2. Design Guidance: When the Clock Resources Run Out

Whenever possible, a designer should plan to use a device with sufficient clock resources (especially clock nets) to accommodate all the clocks in the design. In spite of best efforts, situations arise in which there aren't enough independent clock nets to handle the required number of clock domains, and there is no way to eliminate one or more domains.

In such cases, it is possible to use general (non-clock) FPGA routing resources and still avoid metastability if certain techniques are applied:

- A metastability hazard occurs when a signal is passed between flip-flops clocked by the same clock, but having that clock severely skewed due to the use of general routing resources. Therefore, if flip-flops in a clock domain using general routing do not interact with each other, hold time hazards will not occur. An example of this is a read/write register used as a data store by an external microprocessor. (Note that if this register data is used within the FPGA, clock domain crossing logic must be inserted between the register and the other FPGA logic.)
- When flip-flops within the clock domain do interact, metastability can still be avoided by clocking each flip-flop that receives data on the opposite edge of the clock as the flip-flop that produced that data. One technique that effectively implements this scheme is to build master-slave flip-flops, consuming two FPGA flip-flops for each bit of storage.
- In some cases, it's possible to build synchronous elements, such as counters and shift registers, with fewer than two flip-flops per bit while still using the opposite-edge-of-the-clock scheme. These are difficult and must be analyzed on a case-by-case basis. This document will not provide a tutorial on such unusual logic construction.

2.1.1.2. Use of Derived Clocks

Inexperienced designers will sometimes attempt to create a clock with special characteristics – a different frequency, a different duty cycle, or one that can be disabled, for example – by "deriving" the secondary clock from the primary clock. This is accomplished by combining the primary clock with other logic signals using combinatorial logic, by using the output of a sequential circuit (e.g. a binary counter clocked by the primary clock) as the derived clock, or by a combination of the two approaches.

All of these methods to derive a clock result in the secondary clock being delayed relative to the primary clock. This creates two separate clock domains with a special relationship. The primary domain is "ahead of" the secondary domain. That is, the data transitions of the primary domain will occur sooner than those of the secondary domain, and may in fact occur at roughly the same time as the secondary domain's clock. This means that data clocked first through the primary domain, and then the secondary domain, is likely to produce a metastable event at the flip-flops

of the secondary domain. Data traveling from the secondary domain back to the primary domain does not have this problem.

However, designers often fail to recognize the separate domains and realize that proper mitigating techniques must be used. Mitigating techniques that might have worked when designing with MSI logic on PC boards, such as trying to interpose combinatorial logic in the data paths in order to create a minimum delay, will fail in an FPGA. The internal logic of FPGAs provides no mechanism to create a guaranteed minimum delay. Even if a designer succeeds in making one FPGA work as desired, a process speed-up by the manufacturer may result in devices that consistently fail.

2.1.1.2.1. Design Guidance: Using Clock Enables

Clock enables should be used as the alternative to derived clocks. Most FPGA flip-flops are equipped with dedicated clock enable pins. Designers sometimes make the mistake of thinking of a clock enable as a control input to a clock gate (a simple 2-input AND gate). Instead, the clock enable is actually the select input of a 2-input multiplexer feeding the flip-flop. One input of the multiplexer is the actual data (D) input. The other input is the flip-flop's output (Q). Each clock period, the flip-flop either samples new data or remains unchanged, depending on whether the clock enable is asserted or not, respectively. However, note that the flip-flop is clocked each clock period, and that the clock is not in any way gated or delayed.

Using the clock enable, it's possible to control the sampling activity of a flip-flop to any degree, without using a derived clock. For instance, in order to create a flip-flop that samples a signal once every four clocks, a designer needs to create a clock enable that is asserted every fourth clock. This clock enable can be the output of any combination of combinatorial and synchronous logic running in the same clock domain as the flip-flop being enabled. Glitches on clock enables are irrelevant, since they will settle out by the time the enabled flip-flop samples its D input. It is only necessary that the clock enable meet setup and hold times to the flip-flop, just as the data input must.

2.1.1.3. Excessive Number of Clock Domains

As the complexity of circuitry held within a single FPGA increases, so does the likelihood that multiple clock domains will exist within the FPGA. If complex enough, these may exceed the available clock resources (clock buffers, clock networks, and clock management circuits) of the device.

The use of multiple domains also requires that extra care be taken when moving data from one domain to another. The rules for doing so are moderately complex. Clock-domain-crossing logic can prove a source of errors for even an experienced and careful designer. For all these reasons, it is important to minimize the number of clock domains when possible.

Inexperienced designers may not recognize opportunities to eliminate a clock domain. As an example, consider a video compression engine in which compressed video frames are time stamped. A large part of the circuitry runs at the pixel clock rate. An external source supplies a

"tick" 1000 times per second to increment a counter in the FPGA. This counter is the source of the timestamp. The tick rate is independent of and asynchronous to the pixel clock.

An inexperienced designer might build the counter in a separate clock domain, operating at 1000 Hz. He is now faced with two less than ideal situations. First, an additional set of FPGA clock resources has been consumed. Second, the timestamp value might be changing at the exact instant it needs to be copied into the data of the compressed frame. Successfully moving the entire timestamp between domains without incurring metastability, and without stalling the compression pipeline, is a non-trivial problem.

For this example, there is a simple and elegant way to move the tick, which is relatively slow, into the pixel clock domain. Likewise, there are other techniques to reduce the number of clock domains in an FPGA.

2.1.1.3.1. Design Guidance: Moving Signals into a Different Clock Domain

In the example of the time-stamped compressed video frame, the tick occurs at a much slower rate than the pixel clock, providing an opportunity to use the pixel clock to sample the tick. The duration of the high state of the tick must be at least one pixel clock period long, assuming the high-going transition of the tick is important. The tick need not have a symmetrical waveform. To move the tick into the pixel clock domain, we must prevent the tick from corrupting pixel-clock domain logic due to metastability and we must produce one event for every high-going tick transition. The first requirement is accomplished by passing the tick through a two flip-flop synchronizer. Adding a rising edge detector satisfies the second requirement. The entire circuit is shown in Figure 1. All flip-flops are clocked at the pixel clock rate.

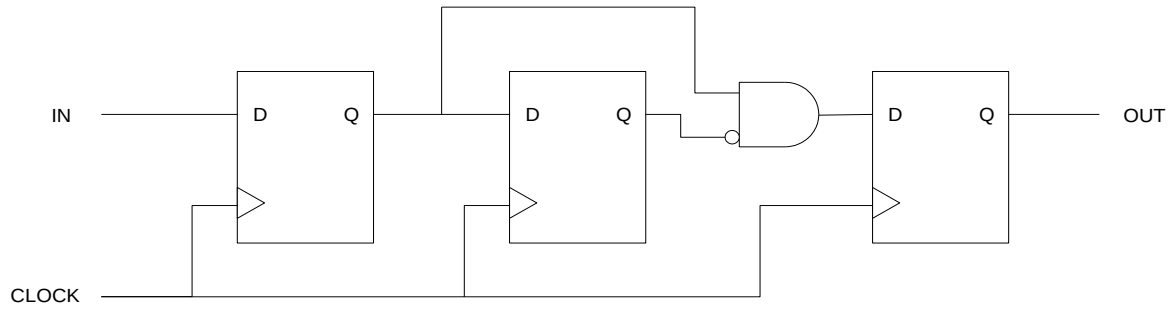

Figure 1: Synchronizer with Rising-Edge Detector

The synchronizer is comprised of the first and third flip-flops. The synchronizer is a probabilistic method of resolving metastability. The first flip-flop, responding to the tick, outputs a high, low, or indeterminate (somewhere between logic high and low, or oscillating) signal. The indeterminate (metastable) state is likely to be resolved, to either a logic high or logic low, with a very high degree of probability, before being clocked by either of the other two flip-flops. This prevents metastable effects from reaching circuitry in the pixel clock domain. The middle flip-flop and AND gate make up the rising edge detector. It should be noted that

there is a latency associate with moving the tick into the pixel clock domain. In this particular case, that latency is between 2 and 3 pixel clocks.

2.1.1.3.2. Design Guidance: Opportunities to Eliminate Clock Domains

In seeking to reduce the number of clock domains, a designer should look for the following opportunities:

- Sample an input signal into another domain, as discussed in the previous section.
- Combine two domains in which the clock rate of one is an integral multiple of the clock rate in the other. Use the faster clock rate in both, with a clock enable, running at the slower rate, for those flip-flops that had been in the slower domain.
- Architect the system with a goal of reducing the number of different clocks. For instance, consider running a 33 MHz-capable microprocessor at 30 MHz if the FPGA already requires a 30 MHz clock. Note that this will also work if the FPGA has a multiple or ratio of 30MHz., e.g. 60 MHz, 10 MHz, etc.

2.1.1.4. Improper Crossing of Clock Domains

Properly passing signals and data from one clock domain to another is difficult because there are so many variations: It may be necessary to move a signal from a slow domain to a faster domain, or the reverse may be true. We may be looking for an edge event, or we may be monitoring the level of a signal. Moving a serial data stream or parallel data from one domain to another further increases complexity. Failure to handle each of these situations correctly will result in metastability and data corruption.

2.1.1.4.1. Design Guidance: Signal Crossing from a Slower to a Faster Domain

Moving a signal from a slower to a faster clock domain is straightforward. We've already discussed the problem of recognizing that a transition (in the previous example, from low to high) occurred in the slower domain. The second case to consider is that of moving a signal level to the faster domain. In other words, as long as the signal in the slower domain remains low, we want the equivalent signal in the faster domain to remain low. Likewise, we want the high duration of the input signal to be reflected by a high signal of similar duration in the fast domain. However, all transitions of the output signal should occur on pixel clock edges.

Accomplishing this is easier than detecting the signal transition. All that is required is to pass the signal through two flip-flops, each clocked by the higher speed domain's clock. This prevents any metastability occurring in the first flip-flop from propagating beyond the second flip-flop.

2.1.1.4.2. Design Guidance: Signal Crossing from a Faster to a Slower Domain

Moving events, such as pulses or edges, from a faster to a slower clock domain can be done only if the events occur infrequently. When changes to the input signal occur too frequently, some will be missed when sampled in the output domain. It's easy to visualize the problem if we

consider the faster domain signal changing on every clock edge. The rapid transitions are impossible to reproduce in the slower domain, where the information rate is limited to the slower clock frequency.

There are two general types of events that can be moved from a faster to a slower domain. In the first case, we wish only to know the current level of the input signal. For this, a simple 2 flip-flop synchronizer, as described in 2.1.1.4.1 is sufficient to prevent metastability. However, note that this circuit may miss events in the faster clock domain that are shorter than the slower domain's clock period.

The second case involves detecting either edges or short pulses occurring in the faster domain. For this, a more sophisticated circuit is required, best described by the schematic in Figure 2.

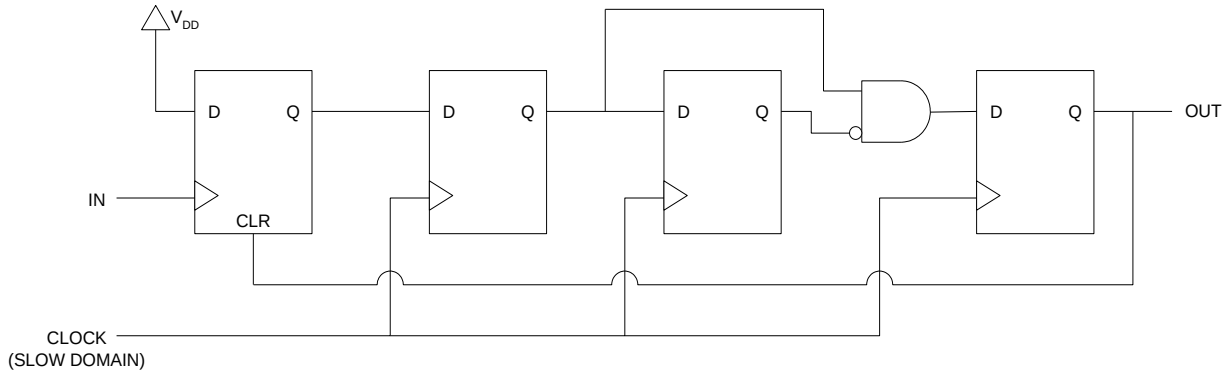

Figure 2: Asynchronous Rising Edge Detector

This particular circuit can successfully capture rising edge events or short, high-going pulses, and move them into the slower domain. Keep in mind that when back-to-back events occur over too short a span of time, one or more will be missed due to the limited information rate in the slower domain. The circuit shown also has a recovery time, during which it cannot respond to a new low-to-high transition on its input. Therefore, this circuit will pass all transitions or pulses only if they occur infrequently with respect to the slower domain's clock period.

A similar circuit can be constructed to capture falling edge events or low-going pulses. Both circuits may be used simultaneously to give the slower domain information about both high and low input transitions. Depending on the activity of the input signal, and the period of the slower domain's clock, it is possible for the slower domain's logic to see a high transition and low transition occur at the same time! Of course, these transitions must occur sequentially in the faster domain, but both may be reported in the same clock period of the slower domain.

2.1.1.4.3. Design Guidance: Serial Stream Crossing from Faster to Slower Domain

Up to this point, we've considered moving signals that represent a status or event across domains. Now we will look at moving a data stream across domains. As explained in the previous section, events occurring continuously on a signal in the faster domain cannot be

transferred serially into the slower domain intact, because some data will be lost, which is unacceptable.

However, perhaps the data in the faster domain is "bursty" in nature; periods of rapid transitions are separated by long periods of inactivity. An example of this is a bit-serial transmission in which 8 data bits are received at, say, 38.4 kilobits per second, but there are long "mark state" times (times during which no valid data is transmitted) between 8-bit strings. In this case, if the average data rate is equal to or below the clock period of the slower domain, data can be moved across the interface by using a serial FIFO (that is, a 1-bit wide FIFO). The basic characteristics of a FIFO are discussed in section 2.1.1.4.5.

If the serial data is continuous rather than bursty, a serial FIFO will overflow. The only way to move this data to the faster domain is to widen the data path between domains. For instance, the serial input can be "de-serialized" into 8-bit words while still in the faster domain, and then transferred into the slower domain. This degree of widening the data is sufficient for the example, as long as the clock rate of the faster domain is not more than 8 times as fast as that of the slower domain. Moving the data in parallel into the slower domain essentially means building a parallel FIFO between the two domains (even though the FIFO may need be only one or two words deep). Parallel FIFOs are considered in section 2.1.1.4.5.

2.1.1.4.4. Design Guidance: Serial Stream Crossing from Slower to Faster Domain

In this situation, there is adequate bandwidth in the faster domain to absorb all possible transitions that occur in the slower domain. Since there will be more clock periods in the faster domain for the same data, the difficulty is in identifying which periods contain valid data and which do not, once the data has been moved to the faster domain.

To accomplish this, the designer must create a "valid" flag in the faster domain. The valid flag is true only during those clock periods in which the faster data stream contains valid information from the slower domain. The valid flag is false for those periods in which the data stream contains "filler." Obviously, the valid flag must be synchronized with the faster domain data stream.

There are many ways of accomplishing this. The most general-purpose, and therefore perhaps the surest, way is to construct a one-bit-wide FIFO between the two clock domains. Data is written to the FIFO input on every slow-domain clock. Data is read out of the FIFO on every fast-domain clock in which the FIFO is not empty. The valid bit is constructed by inverting the FIFO's empty flag and registering it using the faster domain's clock.

2.1.1.4.5. Design Guidance: Moving Parallel Data between Domains.

When a single-bit-wide signal moves through a synchronizer, the output will eventually resolve, providing an accurate, although delayed, copy of the input signal. This is not the case with parallel data. If parallel data is moved across domains using multiple serial synchronizers in parallel, the output data will be flawed. This happens when input data fails to meet setup or hold time of the synchronizer's flip-flops. As a result, some flip-flops pass the previous data value while some pass the new data value. The result is a parallel output that has a mixture of old and new data – a completely erroneous value.

Parallel data that either changes or need be sampled infrequently can be moved across domains safely by identifying a window during which it is safe for the destination domain to read the data. For instance, if there is sufficient time between changes in the input data, a designer could create a separate signal which transitions just as the parallel data in the source domain changes. When this event has been passed through a synchronizer and recognized by the destination domain, the destination domain reads the parallel data. This technique works as long as the parallel data remains unchanged until the destination domain can read it. If the data changes too rapidly for this technique to work, it's best to use a parallel FIFO.

A FIFO is essentially a dual-port memory in which data is written at the rate of the one domain (when there is data available to write) and read out at the rate of the other domain. Constructing such a FIFO will necessitate flags that indicate full, almost full, empty, almost empty, or any other signals used to prevent overflow or underflow. Details on FIFO construction will not be provided here; modern FPGA vendor-supplied software often provides customizable, pre-built FIFOs that can be inserted into a design. It should be mentioned that signals like "full" and "empty" must be ported to the proper clock domains, and be free from metastability.

On both the input and output of a FIFO, the parallel data may be accompanied by a valid flag, which identifies clock periods containing actual data words, as opposed to "filler." On the input, only valid data phases are written to the FIFO. On the output, all words read from a non-empty FIFO are valid. When the FIFO is empty, clock periods containing invalid data will result. It is necessary to distinguish between these two cases by creating a valid flag in the FIFO's output domain.

If valid data words occur less frequently in the source domain than the clock rate of the destination domain, there is adequate bandwidth in the destination domain to absorb the data. A simple valid bit at the FIFO output is sufficient. On the other hand, if source domain data occurs more frequently than the destination domain clock rate, it will be necessary to increase the data width (typically by doubling, tripling, quadrupling, etc. the data width) until the effective source domain word rate is equal to or below the clock rate in the destination domain.

2.1.1.4.6. Design Guidance: Moving Signals and Data between Phase-Related Domains

If two clock domains have a known and repetitive phase relationship, it may be possible to move data between them without concern for metastable events. Some examples of this are:

- With domains of identical frequency, but phase shifted, data can be transferred without synchronizers as long as analysis guarantees that setup and hold time are met at the destination flip-flop.
- One domain's frequency is an integral multiple, X, of the other's, and the slower domain's clock transition lines up closely with every X^{th} clock transition of the faster domain (this is the case when using Xilinx DLLs to create one clock domain from another). In this case, data can be transferred on any of the faster domain's clock transitions providing the propagation delay between domains, plus setup time, is less than the faster domain's clock period.
- The relationship between the two domains' clocks is a constant ratio, e.g. 2/3, and clock transitions closely line up periodically (this is the case when using Xilinx digital

frequency synthesizer – part of the DCM -- to create one clock domain from another). By recognizing the phase relationship between these clocks, it is possible to pass data on specific clock edges, but careful timing analysis and circuit design, beyond the scope of this discussion, is required. In an unusual case such as this, it is simpler to pass data between domains using FIFOs, as previously described.

Even when synchronizers can be eliminated in the situations just described, a designer must remain aware of data bandwidth considerations, and the need to adjust data widths as necessary.

Epilog: Meeting FPGA Timing Requirements

Achieving timing goals in an FPGA requires primarily good architecture and logic design and secondarily skillful use of the implementation tools. The former can be accomplished by adhering to the design practices discussed in this, and other papers by the author, including:

- Synchronous design
- Pipelining
- Partitioning of very high speed logic into a separate clock domain
- Avoidance of latches and asynchronous resets
- Encoding of FSMs to achieve timing goals

One other excellent technique is to register signal outputs from leaf-level modules. This has several advantages:

- No optimization is needed across hierarchical boundaries
- Enables the ability to preserve the hierarchy
- Allows bottom-up compilation
- Allows re-compilation of only those levels that have changed
- Enables hierarchical floorplanning
- Increases the capability of a guided implementation
- Forces the designer to keep like-logic together
- Allows selective timing constraints to be written