Crossing the Abyss

Asynchronous Signals in a Synchronous World

Table of Contents

List of Figures

List of Tables

1 Introduction

Logic circuits having a single clock are the most elementary type of digital design. The reality is that modern digital designs are increasingly sophisticated; having multiple clocks driving different circuits and circuits that must reliably communicate with each other. Most data movement applications such as disk drive controllers, CDROM/DVD controllers, modems, network interfaces and network processors, have multiple clock domains and bear inherent challenges moving data across clock domains.

In modern IC, ASIC and FPGA designs, the engineer has many software programs to help create million gate circuits, but these programs cannot solve the problem of signal synchronization. When signals travel from one clock domain to another, the signal appears to be asynchronous in the new clock domain. Since the engineer's toolbox does not have the tools to handle this situation, it is up to the designer to know reliable design techniques that reduce the risk of failure for circuits communicating across clock domains.

This paper explores the fundamentals of signal synchronization and demonstrates circuits a designer can use to handle signals that cross clock domains. It examines design methodologies for synchronizing single signals and ways of handling groups of signals including data busses that cross clock domains.

1.1 Fundamentals

The first step in managing multi-clock designs is to understand the problem of signal stability. When a signal crosses a clock domain, it appears to be an asynchronous signal to the circuitry in the new clock domain. The circuit that receives this signal needs to synchronize it. Synchronization prevents the metastable state of the first storage element (flip-flop) in the new clock domain from propagating through the circuit.

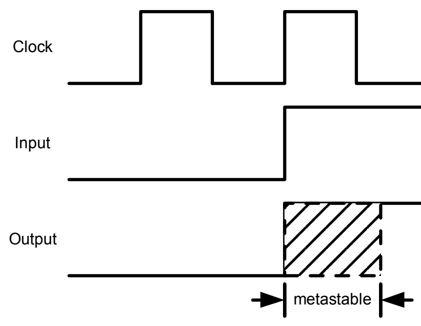

Figure 1-1: Metastable Output

Metastability is the inability of a flip-flop to arrive at a known state in a specific amount of time. When a flip-flop enters a metastable state, a designer cannot predict the element's output voltage level nor when the output will settle to a correct voltage level (see Figure 1-1: Metastable Output). During this settling time, the flip-flop's output is at some intermediate voltage level or may oscillate and can cause a cascade of failures when the flip-flops further down the signal path capture the invalid output level.

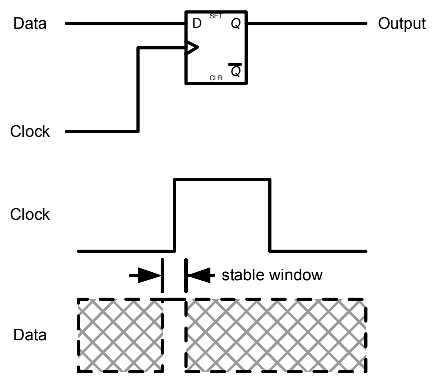

Figure 1-2: Stable Window

For any flip-flop, there is a small window of time where the input must be stable (see Figure 1-2: Stable Window). This window of time is a function of the design of the flip-flop, the implementation technology, operating conditions and the load on the output for outputs not buffered. Also sharp edge rates on the input signal minimize the window of time. The probability of a flip-flop entering a metastable state is also a function of the data and clock frequencies. There are more windows of vulnerability as the clock frequency goes up and there is greater probability of hitting the window as the data frequency goes up.

FPGA manufacturers and IC foundries qualify their flip-flops and determine their characteristics. Mean Time Between Failures (MTBF) describes the metastability characteristic of a flip-flop using statistics to determine the probability of a flip-flop failure. The MTBF is based in part on the length of the time window during which a change in the input signal causes the flip-flop to become unstable. In addition, MTBF calculation uses the frequency of the input signal and the frequency of the clock driving the flip-flop.

Each different type of flip-flop in an ASIC library or in a type of FPGA has timing requirements that help the designer determine the window of vulnerability. Setup time describes the time an input signal to a flip-flop must be stable before the clock edge and the hold time is the time the signal must remain stable after the clock edge. These are very conservative times to account for all the possible variations in supply voltage, operating temperature, signal quality and fabrication variations. If a design meets these timing requirements, the possibility of the flip-flop failing is so small it is negligible.

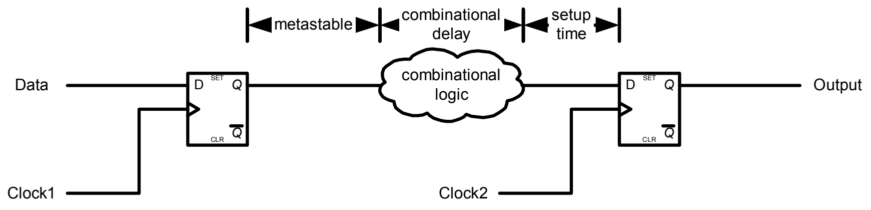

Figure 1-3: Synthesis Timing Calculation

Synthesis programs that engineers use in modern IC and FPGA designs, ensure digital circuits meet the setup and hold requirements for each flip-flop in the design, but asynchronous signals pose problems for the software. A signal crossing a clock domain appears to be asynchronous to the logic in the new clock domain. Most synthesis programs have trouble solving the problem of determining if asynchronous signals meet the timing requirements for flip-flops. Synthesis programs cannot determine the time the flip-flop is unstable, and so they cannot determine the total delay from the flip-flop, through the combinational logic to the next flip-flop (see Figure 1-3: Synthesis Timing Calculation). Since synthesis software cannot handle signal synchronization, the designer uses circuits that mitigate the impact of asynchronous signaling.

1.2 Signal Synchronization

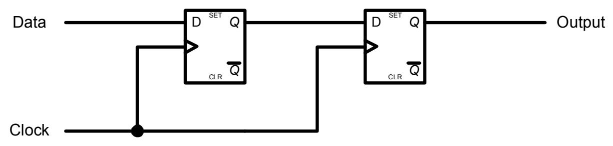

Figure 1-4: A Simple Synchronizer

Synchronizing signals begins by protecting downstream logic from the metastable state of the first flip-flop in a new clock domain. A simple synchronizer consists of two flip-flops in series without combinational circuitry between them (see Figure 1-4: A Simple Synchronizer). This design ensures the first flip-flop exits its metastable state and its output settles before the second flip-flop samples the first one's output.

Besides the circuit design, there is another requirement to make a successful synchronizer. The layout engineer needs to place the flip-flops close to each other. This guarantees the shortest signal wire between the output of the first flip-flop and the input of the second one and ensures the smallest possible clock skew between the flip-flops.

IC foundries help with signal synchronization by providing specially designed synchronizer cells. Usually this synchronizer cells consists of a flip-flop with a very high gain that uses more power and is larger than a standard flip-flop. This flip-flop has reduced setup and hold time requirements for the input signal and is resistant to oscillating when input signal causes a metastable condition. Another type of synchronizer cell contains two flip-flops, which eases the layout engineer's job by meeting the requirement of placing the flip-flops close to each other and prevents the designer from placing any combinational logic between the flip-flops.

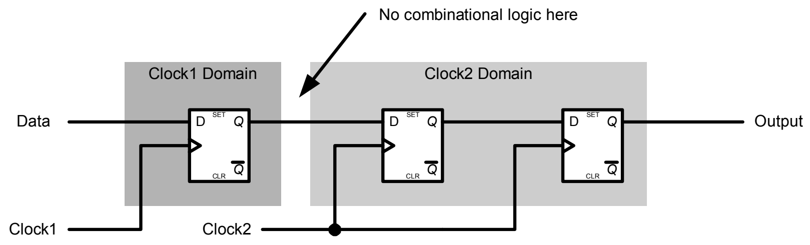

Figure 1-5: Full Synchronizer Circuit

For synchronization to work properly, the signal crossing clock domains comes from a flip-flop in the original clock domain. It does not pass through any combinational logic between the originating flip-flop and the first flip-flop of the synchronizer (see Figure 1-5: Full Synchronizer Circuit). This is important because the first stage of a synchronizer is sensitive to glitches that combination logic produces. If a glitch is long enough and occurs at the correct time, it could meet the setup and hold requirements of the first flip-flop in the synchronizer. This leads to the synchronizer passing a false valid indication to the rest of the logic in the new clock domain. Another consideration when using synchronizers is signal delay.

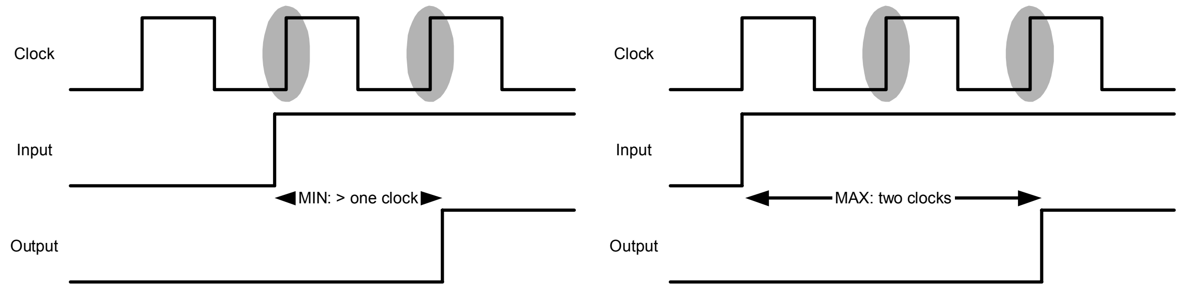

Figure 1-6: Synchronizer Timing

A synchronized signal is valid in the new clock domain after two clock edges. The signal delay is between one and two clock periods in the new clock domain (see Figure 1-6: Synchronizer Timing). A rule of thumb is a synchronizer circuit causes two clock cycles of delay in the new clock domain and a designer needs to consider how synchronization delay impacts timing of signals crossing clock domains.

1.3 Synchronizer Circuits

There are many different designs for synchronizers and each has specific uses because one type does not work well in all applications. All synchronizers use the basic circuit shown in Figure 1-4 and fall into three basic categories: level, edge-detect and pulse.

1.3.1 Level Synchronizer

The circuit in Figure 1-4 is a level synchronizer where the signal crossing clock domains stays high and low for more than a two clock cycles in the new clock domain. A requirement of this circuit is that the signal needs to transition to its invalid state before it can become valid again. Each time the signal goes valid, the receiving logic considers it a single event no matter how long the signal remains valid. This circuit is the heart of all other synchronizers as in the edge-detect synchronizer.

1.3.2 Edge-Detect Synchronizer

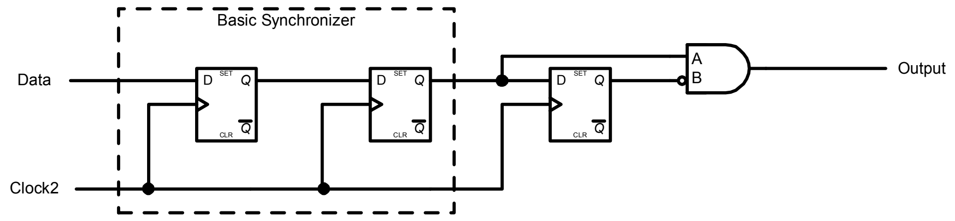

Figure 1-7: Edge-Detect Synchronizer

Figure 1-7 shows the edge-detect synchronizer circuit, which adds a flip-flop to the output of the level synchronizer. The output of the additional flip-flop is inverted and "AND"-ed with the output of the level synchronizer. This circuit detects the rising edge of the input to the synchronizer and generates a single clock wide active high pulse. Switching the inverter on the "AND" gate inputs creates a synchronizer that detects the falling edge of the input signal. In addition, changing the AND gate to a NAND gate results in a circuit that generates an active low pulse.

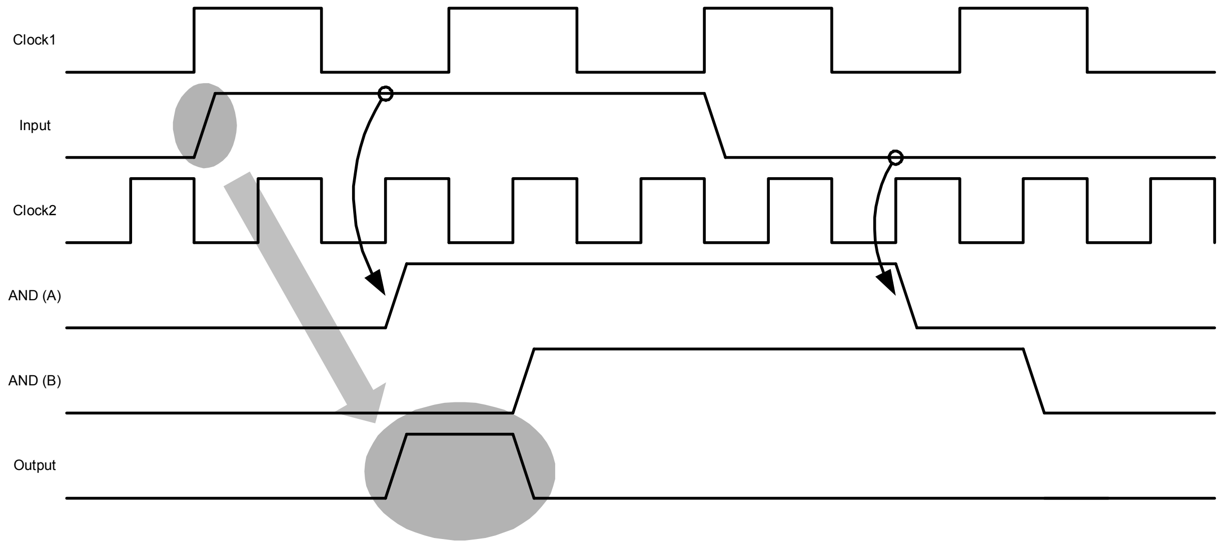

Figure 1-8: Rising Edge-Detect Synchronizer Timing

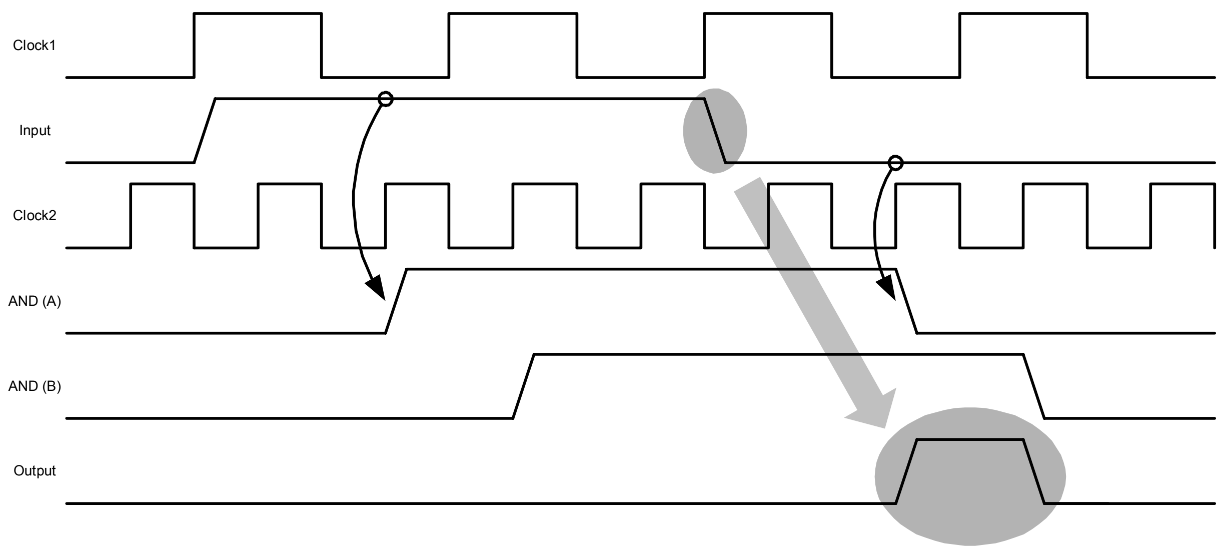

Figure 1-9: Falling Edge-Detect Synchronizer Timing

The edge-detect synchronizer's main application is synchronizing a pulse going to a faster clock domain. This circuit produces a pulse that indicates the rising (or falling) edge of the input signal. Figure 1-8 and Figure 1-9 show the circuit's timing for rising edge and falling edge detection respectively.

A restriction on the application of this synchronizer is the width of the input pulse must be greater than the period of the synchronizer clock plus the required hold time of the first synchronizer flip-flop. The safest pulse width is twice the synchronizer clock period. This synchronizer does not work if the input is a single clock-wide pulse going to a slower clock domain; however, the pulse synchronizer solves this problem.

1.3.3 Pulse Synchronizer

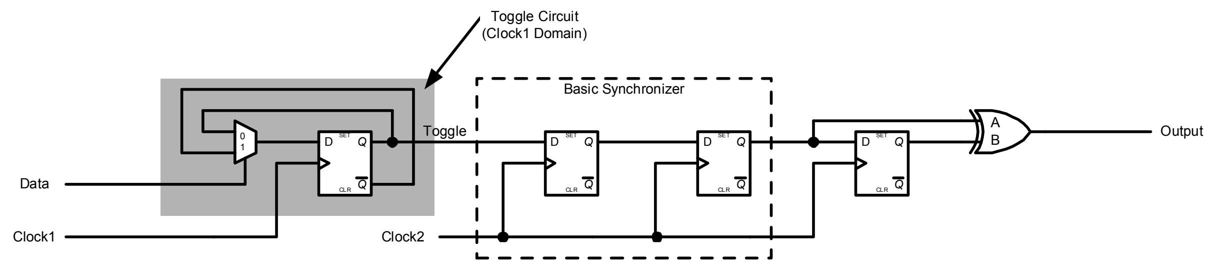

Figure 1-10: Pulse Synchronizer

Figure 1-10 shows the pulse synchronizer. The input signal is a single clock cycle wide pulse that triggers a toggle circuit in the originating clock domain. The output of the toggle circuit is a signal that switches from high to low and vice versa each time it receives a pulse. This signal passes through the level synchronizer and arrives at one input of the "XOR" gate while a one clock cycle delayed version goes to the other input of the "XOR". For one clock cycle, each time the toggle circuit changes state the output of this synchronizer goes high.

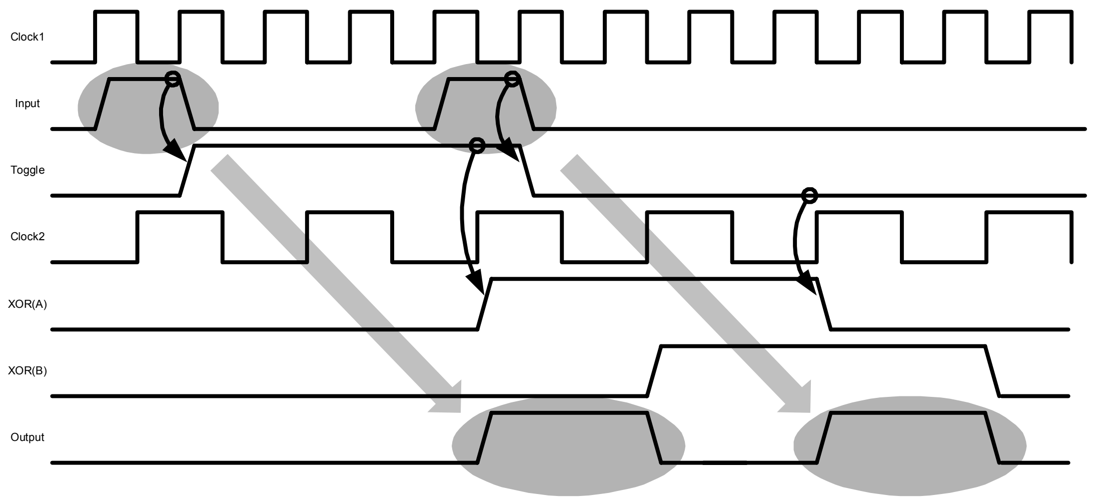

Figure 1-11: Pulse Synchronizer Timing

The basic function of a pulse synchronizer is to take a single clock wide pulse from one clock domain and create a single clock wide pulse in the new domain. Figure 1-11 shows the synchronizer's timing.

One restriction on this synchronizer design is the input pulses must have a minimum time between them. This minimum spacing between the pulses is equal to two synchronizer clock periods. If the input pulses are closer, the output pulses in the new clock domain are adjacent to each other resulting in an output pulse that is wider than one clock cycle. This is a more severe problem when the clock period of input pulse is greater than twice the synchronizer clock period. In this case, if the input pulses are too close, the synchronizer does not detect every one.

Type	Application	Input	Output	Restriction
Level	Synchronizes level signals	Level	Level	Input must be valid for at least two clock periods in the new domain. Each time output goes valid, counts as a single event.
Edge-detect	Detects rising or falling edge of input	Level or Pulse	Pulse	Input must be valid for at least two clock periods in the new domain.
Pulse	Synchronizes single clock-wide pulses	Pulse	Pulse	Input pulses must be spaced by at least two clock periods in the new domain.

Table 1-1: Synchronizer Summary

Table 1-1 show the synchronizers described above, their application, the type of output and the restriction on how a designer uses them. There are other synchronizer designs but these serve most applications a designer encounters.

2 Design Methodologies

Synchronizers are the most basic tools that an engineer uses to handle signals crossing clock domains. However, an engineer also needs to know protocols that circuits use when they communicate with each other asynchronously. In many applications, simple signals are not the only information crossing clock domains; data and control busses also travel together across domains. Engineers have additional tools at their disposal that can handle these situations, such as handshaking protocols and FIFOs.

2.1 Handshaking

Handshaking allows digital circuits to effectively communicate with each other when the response time of one or both circuits is not predictable. An example that most designers have encountered is an arbitrated bus. Here more than one circuit requests access to a single bus (i.e., PCI, AMBA), and arbitration determines which circuit gains access to the bus. Each circuit signals a "request" and the arbitration logic determines which of the requesters is the "winner". The winning requester receives an "acknowledge" that indicates it has access to the bus, it discontinues its request and begins the bus transaction.

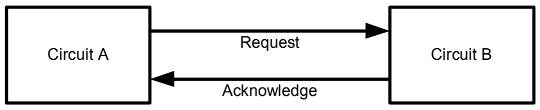

Figure 2-1: Handshake Circuit

The response time of circuits in different clock domains is not predictable because of synchronization so handshake signaling is a very effective means of communication between them. Full and partial handshake signaling are the two fundamental types of handshake protocol. Each type of handshake uses the synchronizers described above, and each has its own set of design trade-offs. To illustrate the various handshake protocols, the diagram in Figure 2-1 shows two circuits and the signals between them. Circuit A controls the Request signal, Circuit B controls the Acknowledge signal, and each handshake protocol is examined using this diagram.

2.1.1 Full Handshaking

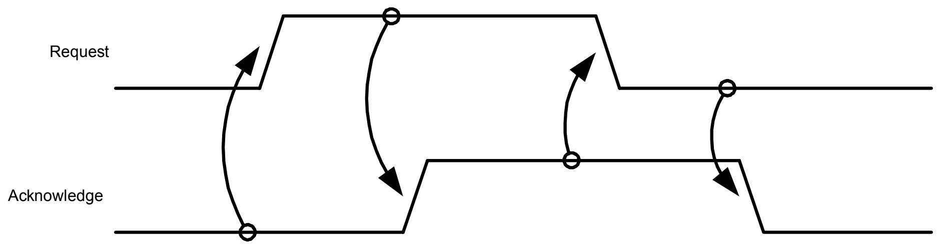

Figure 2-2: Full Handshake Signaling

Full handshake signaling means the two circuits wait for each other before asserting or dropping their respective handshake signal. The way this protocol works is first Circuit A asserts its request signal. Next, Circuit B detects that the request signal is valid and asserts its acknowledge signal.

When Circuit A detects that the acknowledge signal is valid, it drops its request signal. Finally when Circuit B detects that the request is invalid, it drops its acknowledge signal (see Figure 2-2: Full Handshake Signaling). Circuit A does not make a new request until it detects that the acknowledge signal is invalid.

This type of handshake uses level synchronizers. A designer uses this technique when Circuit B (the acknowledging circuit) needs to inform Circuit A (the requesting circuit) that it is actively processing the request. This handshake requires that the requesting circuit hold off its next request until it detects that the acknowledge signal is invalid. The following is a detailed description and timing for full handshake signaling.

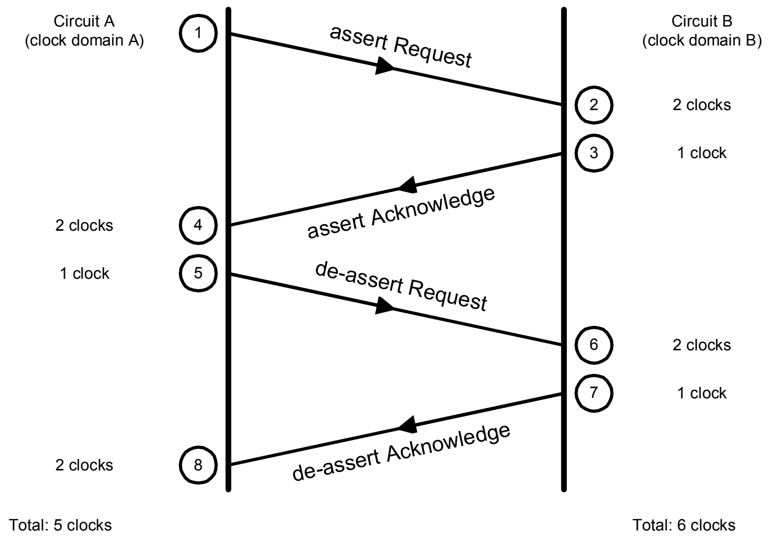

Figure 2-3: Full Handshake Flow

Use the rules of thumb that signals take two clock cycles to cross a clock domain and circuits register signals before they cross clock domains. The sequence and timing for this type of handshake is (see Figure 2-3: Full Handshake Flow):

1. Circuit A asserts Request (in clock domain A).
2. Circuit B detects Request two clocks later (in clock domain B).
3. In the next clock cycle (domain B), Circuit B asserts Acknowledge.
4. Circuit A detects Acknowledge two clocks later (in clock domain A).
5. In the next clock cycle (domain A), Circuit A de-asserts Request.
6. Circuit B detects Request two clocks later (in clock domain B).
7. In the next clock cycle (domain B), Circuit B de-asserts Acknowledge.
8. Circuit A detects Acknowledge two clocks later (in clock domain A).

Here, the complete sequence takes a maximum of 5 cycles in the A clock domain plus a maximum of 6 cycles in the B clock domain.

With full handshake, each circuit explicitly knows the state of the other by examining the request and acknowledge signals, making full handshake signaling very robust. The drawback with this scheme is the entire process takes time. Partial handshaking is another signaling technique that shortens this sequence of events.

2.1.2 Partial Handshaking

With partial handshake signaling, the two circuits communicating with each other do not wait for the other one before dropping their respective signal and continuing with the handshake sequence. Partial handshaking is less robust than full handshaking but the complete handshake cycle is faster. It is less robust because the handshake signals do not indicate the state of the both circuits so the designer needs ensure the circuit saves state information normally present in full handshake signals. By not waiting until the other circuit drops it handshake signal, the whole sequence of events is shorter.

When using partial handshake signaling, the acknowledging circuit must generate its signal at the correct time. If the acknowledging circuit needs to complete processing the request before it can handle another, then the timing of the acknowledge signal is important. The circuit uses its acknowledge signal to indicate when it completed any processing. There two partial handshake schemes, one that mixes level and pulse signaling and the other that uses pulse signaling only.

2.1.2.1 Partial Handshake Technique I

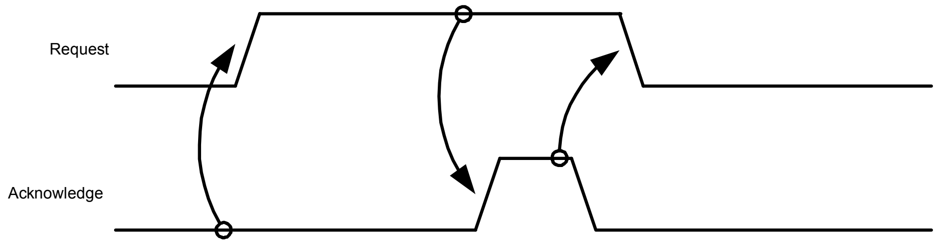

Figure 2-4: Partial Handshake I Signaling

In the first partial handshake scheme, Circuit A asserts its request signal and the Circuit B acknowledges it with a single clock-wide pulse. In this case, Circuit B does not care when Circuit A drops its request signal. However, to make this technique work, Circuit A must drop its request signal for at least one clock cycle otherwise Circuit B cannot distinguish between the previous and the new request.

With this handshake, Circuit B uses a level synchronizer for the request signal and Circuit A uses a pulse synchronizer for the acknowledge signal. In this handshake protocol the acknowledge pulses only occur when Circuit B detects the request signal. This allows Circuit A to control the spacing of the pulses into the synchronizer by controlling the timing of its request signal. The following is a detailed description and timing for this type of partial handshake signaling.

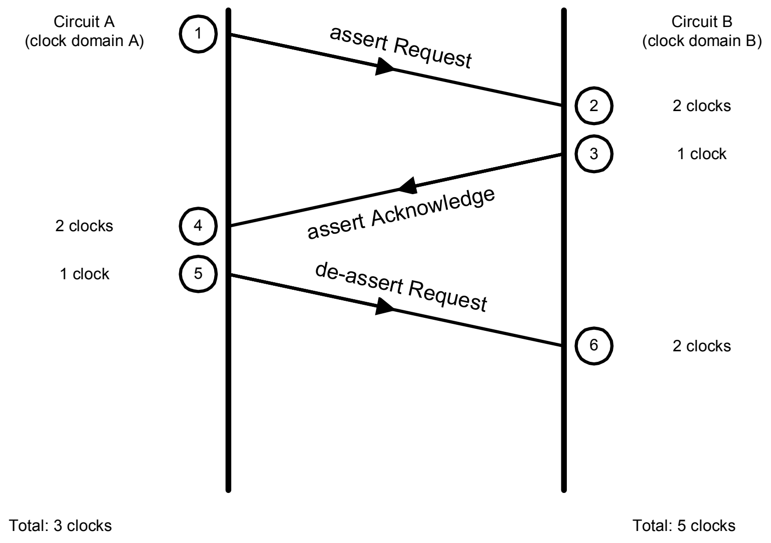

Figure 2-5: Partial Handshake I Flow

Once again, use the rules of thumb that signals take two clock cycles to cross a clock domain and circuits register signals before they cross clock domains. The sequence and timing for this type of handshake is (see Figure 2-5: Partial Handshake I Flow):

1. Circuit A asserts Request (in clock domain A).
2. Circuit B detects Request two clocks later (in clock domain B).
3. In the next clock cycle (domain B), Circuit B asserts Acknowledge.
4. Circuit A detects Acknowledge two clocks later (in clock domain A).
5. In the next clock cycle (domain A), Circuit A de-asserts Request.
6. Circuit B detects Request two clocks later (in clock domain B).

Here, the complete sequence takes a maximum of 3 cycles in the A clock domain plus a maximum of 5 cycles in the B clock domain. This partial handshake signaling uses 2 less clock cycles in the A clock domain and 1 less clock cycle in the B clock domain than full handshake signaling. The complete sequence can be shorter by a few more clock cycles by using the following partial handshake technique.

2.1.2.2 Partial Handshake Technique II

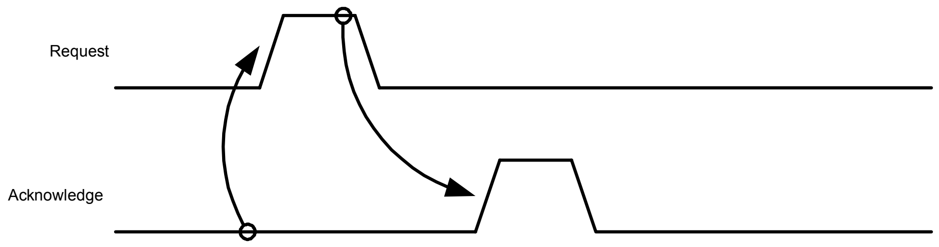

Figure 2-6: Partial Handshake II Signaling

In this second partial handshake scheme, Circuit A asserts it request with a single clock-wide pulse and Circuit B acknowledges it with a single clock-wide pulse. In this case, both circuits need to save state to indicate that the request is pending.

This type of handshake uses pulse synchronizers but if one circuit has a clock that is twice as fast as the other, that circuit can use an edge-detect synchronizer instead. The following is a detailed description and timing for this type of partial handshake signaling.

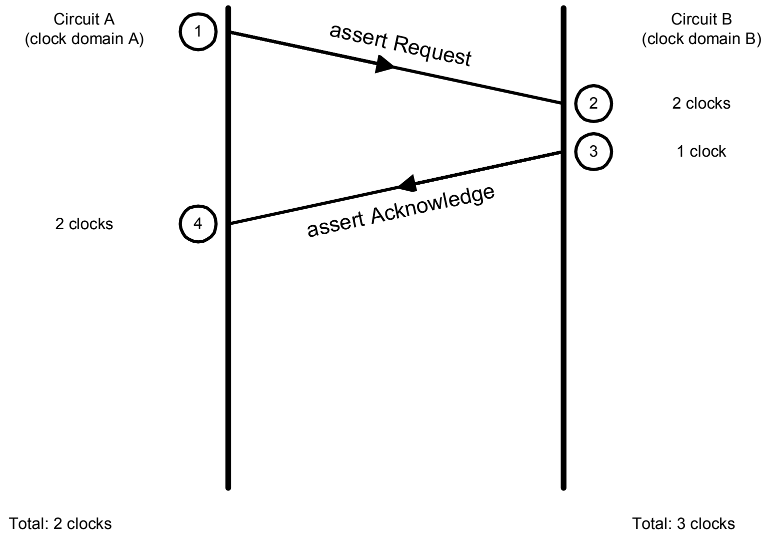

Figure 2-7: Partial Handshake II Flow

Once again, use the rules of thumb that signals take two clock cycles to cross a clock domain and circuits register signals before they cross clock domains. The sequence and timing for this type of handshake is (see Figure 2-7: Partial Handshake II Flow):

1. Circuit A asserts Request (in clock domain A).
2. Circuit B detects Request two clocks later (in clock domain B).
3. In the next clock cycle (domain B), Circuit B asserts Acknowledge.
4. Circuit A detects Acknowledge two clocks later (in clock domain A).

Here, the complete sequence takes a maximum of 2 cycles in the A clock domain plus a maximum of 3 cycles in the B clock domain. This partial handshaking technique uses 3 less clock cycles in the A clock domain and 3 less clock cycle in the B clock domain than full handshake signaling. This technique is also faster than the first partial handshake signaling by one cycle in the A clock domain and 2 cycles in the B clock domain.

Handshake Type	Circuits	Signaling Type	Sequence Length	Synchronizer	Restrictions
Full	Circuit A (Request)	level	5 clocks	level	▪ Sequence is long ▪ Request must be invalid for at least two of the Circuit B clock periods ▪ Acknowledge must be invalid for at least two of the Circuit A clock periods
	Circuit B (Acknowledge)	level	6 clocks	level	
Partial I	Circuit A (Request)	level	3 clocks	pulse or edge-detect	▪ Must control rate of Acknowledge pulses ▪ Request must be invalid for at least two of the Circuit B clock periods
	Circuit B (Acknowledge)	pulse	5 clocks	level	
Partial II	Circuit A (Request)	pulse	2 clocks	pulse or edge-detect	▪ Must save pending request information ▪ Must register Request and Acknowledge signals
	Circuit B (Acknowledge)	pulse	3 clocks	pulse or edge-detect	

Table 2-1: Handshake Summary

Table 2-1 shows each handshake protocol, the type of signaling, the length of the sequence, the synchronizers used and some considerations for the designer. These handshake protocols involve single signals that cross clock domains, when groups of signals cross clock domains, the designer needs to use more complex signaling schemes.

2.2 Data Path Design

One import rule when synchronizing signals is a design should not synchronize the same signal in more than one place. Since synchronization takes between one and two clock cycles, a designer cannot reliably predict when a signal arrives across a clock domain. In addition, the timing of the synchronized signals in the new clock domain can vary between each synchronizer where one has one clock cycle of delay and the other two cycles. This is a “race condition” between the individually synchronized signals. This race condition also applies to groups of signals such as data, address and control busses that need to travel together across clock domains.

A design should not use individual synchronizers on each signal in the group or on each bit of a data or address bus. The synchronization problem for a signal bus is more complex than for individual signals.

2.2.1 Basic Data Path Design

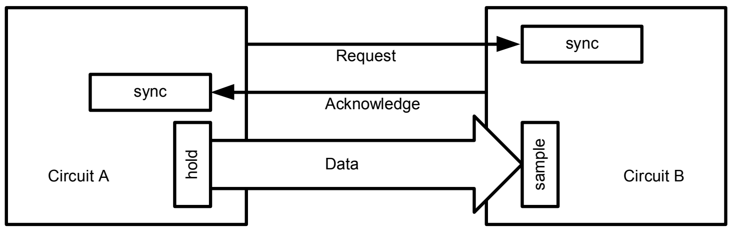

Figure 2-8: Data Path Synchronizer Diagram

The most basic way to solve the problem of bus synchronization is a holding register and handshake signaling. The circuit consists of a register that holds the signal bus, combined with one of the handshake schemes described earlier (see Figure 2-8: Data Path Synchronizer Diagram). The handshake signals indicate when the circuit in the new clock domain can sample the bus and when the originating circuit can replace the current contents of the holding register.

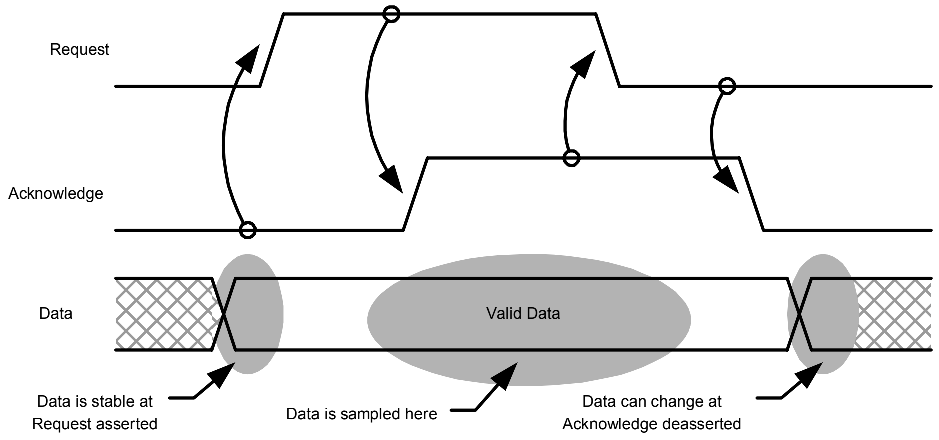
Request
Acknowledge
Data
Valid Data
Data is stable at Request asserted
Data is sampled here
Data can change at Acknowledge deasserted

Figure 2-9: Data Path Timing Using Full Handshake

In this design, the transmitting circuit stores the Data (signal bus) in the holding register as it asserts the Request signal. These two actions can happen at the same time since the Request signal takes at least one clock cycle before the receiving circuit detects it. When the receiving circuit samples the Data (signal bus), it asserts the Acknowledge signal (see Figure 2-9: Data Path Timing Using Full Handshake). This design uses full handshake and takes a long time to complete the transfer.

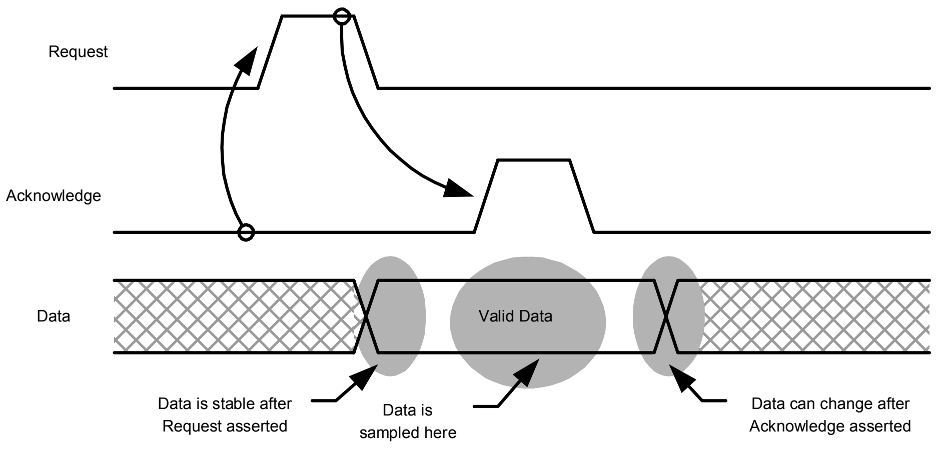
Request
Acknowledge
Data
Valid Data
Data is stable after Request asserted
Data is sampled here
Data can change after Acknowledge asserted

Figure 2-10: Data Path Timing Using Partial Handshake

A design using full handshake signaling has a large window of time for the receiving circuit to sample the signal bus and is not very efficient. The same design can use a partial handshake instead of the full handshake to speed up the transfer (see Figure 2-10: Data Path Timing Using Partial Handshake).

With this type of bus synchronization, a design synchronizes the handshake signals but not the signal bus. The signal bus originates from the holding register and remains stable until after the receiving circuit samples it. This bus synchronization may not work in applications where the transmitting circuit presents data too fast for the receiving circuit to handle.

2.2.2 Advanced Data Path Design

There are many cases where data needs to "pile up" as it crosses clock domains, so designs using a single holding register would not work. One case is a transmitting circuit that presents data in bursts, too fast for a receiving circuit to sample. Another case is a receiving circuit that samples data faster than the transmitting circuit but in a narrower data width. These situations call for the designer to use a FIFO.

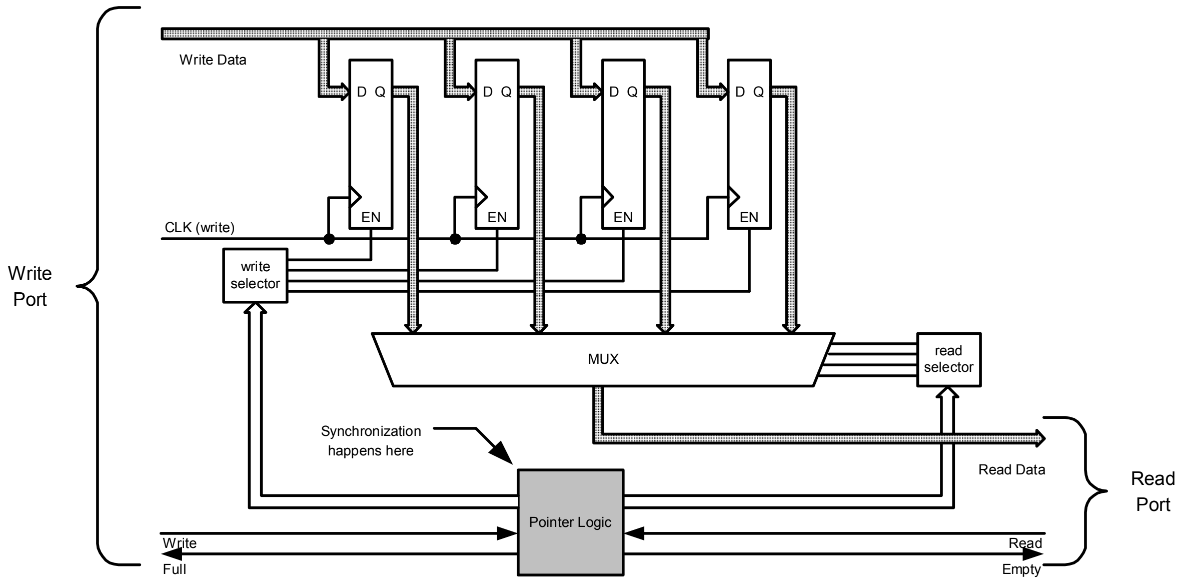

Figure 2-11: Four-Entry FIFO

Fundamentally, a designer uses a FIFO for speed matching, data width matching or both. For speed matching, the faster port on the FIFO handles burst transfers while the slower port handles constant rate transfers. However, with these different access types and speeds, the average data rates into and out of the FIFO have to be the same otherwise the FIFO overflows or underflows. Like the single register design above, the FIFO holds data in registers while it synchronizes status signals (see Figure 2-11: Four-Entry FIFO) that determine when data changes or is sampled.

In speed matching applications, each port has a different clock. The registers in the FIFO use the Write Port clock just as the holding register used the clock of the circuit changing the register's contents. The signal synchronization happens in the Pointer Logic and is more complex than handshake signaling. There are several approaches to designing the Pointer Logic. The first method is to synchronize the read and write strobes while using counters in each clock domain to track the available entries in the FIFO.

2.2.2.1 Counter Based FIFO Status

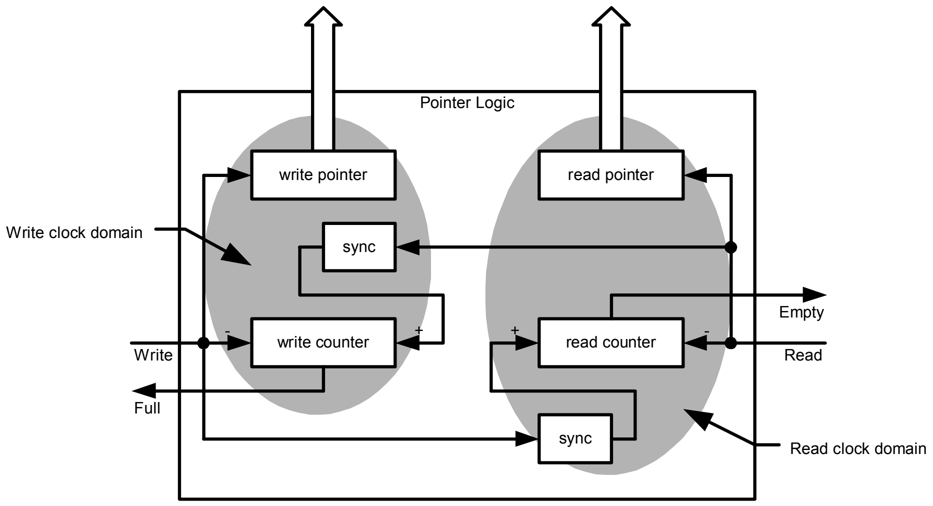

Figure 2-12: Dual Counter FIFO Status

In this design, the counters reflect the number of FIFO entries available for reading or writing and the counters are synchronous to their respect port. The read counter tracks the number of entries that contain valid data and the write counter tracks the number of entries available to store data. When the Pointer Logic is reset the read counter starts at zero since no data is available to read. The write counter starts at the number of entries in the FIFO, meaning all the entries are available for storing data.

The Read signal decrements the read counter and is synchronized to the write clock domain before it increments the write counter. The Write signal decrements the write counter and is synchronized to the read clock domain before it increments the read counter (see Figure 2-12: Dual Counter FIFO Status).

This design requires pulses and pulse type synchronizers for the Read and Write signals. If a level signal crosses from one clock domain to a faster one, it remains valid for more clock cycles in the faster domain than in the slow one. If a counter changes whenever the read or write signal is valid, then the faster clock domain detects more reads or writes than actually happened in the slower clock domain. Pulse synchronizers translate a single clock wide pulse in one clock domain to the single clock wide pulse in the new clock domain and each pulse represents a single read or write of the FIFO.

This FIFO status technique gives pessimistic status for both reads and writes. The status on the write side indicates full when the FIFO has all entries filled and continues to indicate full after the read strobe triggers since synchronization delays the strobe to the write counter. This is also true for the empty status on the read side since synchronization delays the write strobe to the read counter.

Another consideration for this design is detecting full or empty at the right time. If the FIFO has one entry remaining and the write strobe triggers the full status must be set at that time. This gives the full indication one clock sooner to allow the circuit writing into the FIFO enough time to stop the next write from overflowing the FIFO. This is also true to the read side of the FIFO. In this case, if the FIFO has only one entry and the read strobe triggers, the empty status must be set giving the read circuitry time to prevent a read of an empty FIFO.

This Pointer Logic restricts the circuits using the FIFO from accessing it on every clock cycle, even in the slow clock domain. The advantage of this is that the circuits accessing the FIFO have at least one clock cycle to evaluate the FIFO status. The FIFO can have every entry filled with data without overwriting valid data or can be empty without reading invalid data.

Another advantage of this design is that each side can read their respective counters and determine how many entries are available. A designer can use this FIFO design for circuits that perform multiple reads or writes of data without causing an underflow or overflow condition.

The draw back with this design is counters determine the status rather than directly comparing the read and write pointers and for large FIFOs, these counters can be very large. Also the average data rates are half the slowest clock frequency since the pulses (read or write) from the faster clock domain must be spaced by at least two clock periods in the slow clock domain when using pulse synchronizers (see Pulse Synchronizer). Another implementation of FIFO Pointer Logic that eliminates some of these problems uses direct pointer comparison.

2.2.2.2 Pointer Compare FIFO Status

In a synchronous FIFO design, comparing the read and write pointers determines FIFO status. Pointer comparison in asynchronous designs is more challenging since each pointer exists in a different clock domain and synchronizing a signal bus requires the bus not change while synchronizing handshake signals (see Basic Data Path Design). A FIFO design using this technique for pointer synchronization would be very slow. To solve this problem the FIFO Pointer Logic uses Gray Code instead of binary coding for the pointers.

Decimal	Binary	Grey
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Table 2-2: Binary to Gray Code

Gray Code only changes one bit at a time for each increase in the count (see Table 2-2: Binary to Gray Code). It is possible to use synchronizers on a Gray Code busses since it has only one signal changing each time the bus changes. This eliminates the race condition between the bits of the Gray Coded bus passing through separate synchronizers (see Basic Data Path Design).

To convert Gray Code to binary use:

$$\begin{gathered} b_n = g_n \\ b_{n-1} = b_n \oplus g_{n-1} \\ b_{n-2} = b_{n-1} \oplus g_{n-2} \\ \vdots \\ b_1 = b_2 \oplus g_1 \\ b_0 = b_1 \oplus g_0 \end{gathered}$$

and to convert binary to Gray Code use:

$$\begin{gathered} g_n = b_n \\ g_{n-1} = b_n \oplus b_{n-1} \\ g_{n-2} = b_{n-1} \oplus b_{n-2} \\ \vdots \\ g_1 = b_2 \oplus b_1 \\ g_0 = b_1 \oplus b_0 \end{gathered}$$

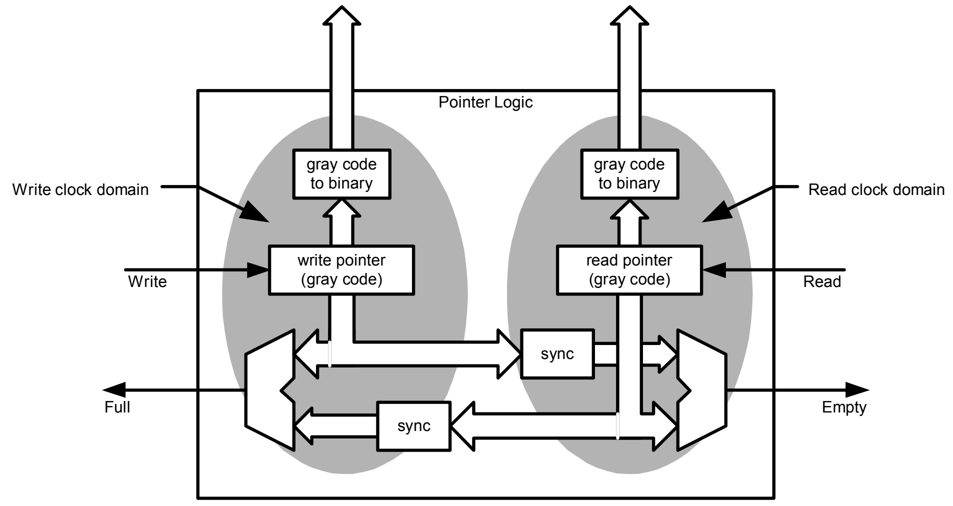

Figure 2-13: Gray Code Pointer Compare FIFO Status

The pointers for this design are Gray Code counters (see Figure 2-13: Gray Code Pointer Compare FIFO Status). Using binary pointers instead requires synchronizing the pointer values after converting them to Gray Code and violates the restriction that synchronized signals originate from flip-flops before crossing a clock domain (see Fundamentals). A Gray Code counter is easy to implement.

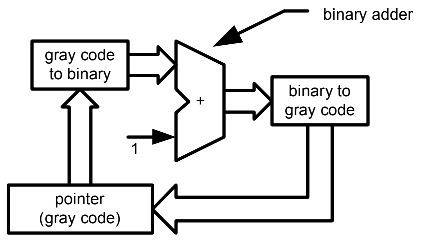

Figure 2-14: Gray Code Counter

The Gray Code counter is a binary adder with converters from and to Gray Code before and after the adder (see Figure 2-14: Gray Code Counter). Since converting to and from Gray Code is a XOR operation, this counter design has only two more levels of logic than a binary counter. A design can use the same technique to compare Gray Code pointer values by adding converters between the pointers and binary comparison logic.

A FIFO with this Pointer Logic is very fast and circuits can read or write the FIFO on every clock cycle. However, accessing the FIFO on every cycle means the FIFO status has to include an almost full and an almost empty indication so the circuits accessing the FIFO have enough time to stop. Almost full indicates that one entry is available to write and almost empty indicates one entry remains unread. These are the least number of status signals and a design needs more if the circuits accessing the FIFO use a burst access with a fixed minimum size.

This FIFO status technique gives pessimistic status for both reads and writes. The status on the write side indicates full when the FIFO fills and continues to indicate full after it is read since synchronization delays the read pointer to the write-side comparison logic. This is also true for the empty status on the read side since synchronization delays the write pointer to the read-side comparison logic.

3 Conclusion

To prevent metastability of flip-flops receiving signals that cross clock domains from causing unpredictable behavior in circuits, use synchronization. For single signals, there are three basic types of synchronizers, level, edge and pulse. Use the level synchronizer for signals that remain valid for many clock cycles. Use the edge-detect synchronizer for level signals in the slower clock domain that change to pulses in the new clock domain. Finally, use pulse synchronizers for pulses crossing clock domains. Remember when a signal bus crosses clock domains, it needs to arrive in the new clock domain at the same time (i.e., in the same clock cycle). Do not synchronize each individual signal but use a holding register and handshaking.

Handshaking indicates when signals in the holding register are valid and when to sample them. There are two basic types of handshake protocol, full and partial. Full handshake is the most robust but also uses the most time (clock cycles) to complete. Partial handshake is faster but requires more care when designing circuits using it. Using handshake and a holding register is useful for data busses but does not provide for passing more than one data word at a time to the new clock domain.

When data needs to pile up such as when one or both circuit(s) are transmitting or receiving data in bursts, use a FIFO. The flip-flops of a FIFO can be registers, latches or memory. There are two basic types of FIFO status logic, counter and pointer comparison. Counter based FIFO status is simple to implement but restricts access speed to half that of the slowest clock in the FIFO status logic. Pointer comparison is more complex and involves Gray Code counters and comparison logic but allows FIFO access on every clock cycle. However this type of access may require advanced warning to the circuits accessing the FIFO through additional status signals to prevent overflow or underflow conditions.

Using these techniques ensures that any designs with multiple clock domains have reliable and predictable performance.

4 References

Leroy Davis, "Logic Metastability", September 2002.
Available at www.interfacebus.com/Design_MetaStable.html

Luke Seed, "Introduction to VLSI/Clocked CMOS Circuits", The University of Sheffield, UK, February 2002.
Available at www.shef.ac.uk/eee/teach/resources/eee310/documents/VLSI_Clocked_CMOS.pdf

Eilhard Haseloff, "Metastable Response in 5-V Logic Circuits", February 1997, (Texas Instruments Application Note SDYA006)
Available at www-s.ti.com/sc/psheets/sdya006/sdya006.pdf

Clifford E. Cummings, "Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs", SNUG 2001 (Synopsys Users Group Conference, San Jose, CA, 2001) User Papers, March 2001, Section MC1, 3rd paper.
Available at www.sunburst-design.com/papers

Luke Seed, "Introduction to VLSI/Clocked CMOS Circuits", The University of Sheffield, UK, February 2002.
Available at www.shef.ac.uk/eee/teach/resources/eee310/documents/VLSI_Clocked_CMOS.pdf

Anon, "A Metastability Primer", November 15, 1989, (Philips Semiconductors Application Note AN219).
Available at www.semiconductors.philips.com/acrobat/applicationnotes/AN219_1.pdf

John F. Wakerly, Digital Design: Principles and Practices, Prentice Hall, 1990.