

Handling Multiple Clocks

(Problems & Remedies in Designs involving Multiple Clocks)

Mohit Arora, Prashant Bhargava
Shivraj Gupta



125, Udyog Vihar,
Phase I,
Gurgaon, India
(www.dcmtech.com)

ABSTRACT

The scope of this paper deals with issues regarding multiple clock designs and provides short but comprehensive information on the same. Designs involving single clock are like a walk in the park... but the real challenge comes when one has to face more than one clock. Designers are faced with problems of metastability, phase or frequency difference among the clocks involved, performing asynchronous data transfer, etc. This paper covers the issue of multiple clock domains & its problems, by starting with a simple design of a single clock FIFO and later expanding it to dual clock domain and separately detailing on the problems involving more than single clock domains. In short, this paper covers what all a designer needs to make a robust and efficient design involving multiple clock domains.

Navigation Aids

Contents

1. Introduction	1
2. Multiple Clock Domains	1
3. Problems in design with multiple clock domains	2
Setup Time & Hold Time Violation:	3
Metastability:	4
4. We shall Overcome.....	7
Design Tips:	7
Transferring signals between clock domains:	8
(a) Transfer of Control Signals (Synchronization)	9
(b) Transfer of Data Signals	11
5. Handshake Signaling Method.....	12
6. Data transfer through Synchronous FIFO	14
Synchronous FIFO Architecture	15
Working of Synchronous FIFO	16
FIFO full and FIFO empty generation	16
7. Asynchronous FIFO (Dual Clock FIFO)	18
Gray Code Fundamentals.....	20
Effect of synchronization on pointers	21
Gray code implementation of FIFO pointers	24
FIFO Full and FIFO Empty generation	28
An Alternative approach for FIFO full and FIFO empty generation	31
FIFO Design	33
FIFO empty condition generation	33
FIFO full condition generation	34
8. Conclusions	36
9. Acknowledgements.....	37
10. References.....	37
11. Author & Contact information	37

List of Figures

Figure 1: Single clock Domain.....	1
Figure 2: Multiple clock domain system	1
Figure 3: Relationship between clocks	2
Figure 4: Setup Time & Hold Time.....	3
Figure 5: Violation of Setup & Hold Times.....	4
Figure 6: Metastability Timing Parameters	5
Figure 7: Behavior of a FF.....	5
Figure 8: Metastability in Flip Flops.....	6
Figure 9: Design Partitioning	8
Figure 10: Two-stage synchronizer circuit	9
Figure 11: Timings for the two-stage synchronizer circuit	10
Figure 12: Three-stage synchronizer circuit	10
Figure 13: Timings for the three-stage synchronizer circuit	11
Figure 14: Two-clock system divided into two separate systems	12
Figure 15: Timings for Handshaking Method of Data Transfer.....	13
Figure 16: Constraint on xreq signal	14
Figure 17: Synchronous FIFO Architecture.....	15
Figure 18: FIFO Full Condition.....	16
Figure 19: Asynchronous FIFO (Dual Clock FIFO).....	18
Figure 20: Effect of synchronization on FIFO full logic.....	21
Figure 21: FIFO Full Illusion	22
Figure 22: Effect of synchronization on FIFO empty logic.....	23
Figure 23: FIFO Empty Illusion.....	23
Figure 24: Gray counter using binary adder.....	24
Figure 25: Gray Counter Hardware.....	28
Figure 26: FIFO full condition.....	29
Figure 27: FIFO empty condition.....	29
Figure 28: FIFO full & empty signals generation hardware.....	30
Figure 29: 4-bit Gray Counter.....	31
Figure 30: Conversion of 4-bit Gray code to 3-bit Gray code.....	32
Figure 31: Dual Clock FIFO Design	33
Figure 32: FIFO full & FIFO empty conditions	34
Figure 33: FIFO full condition.....	35

1. Introduction

Designs involving single clocks are very easy, simple and implemented without much effort. But in actual practice, there are few designs that function on just one clock. This paper deals with multiple clock designs, problems faced therein and how should an individual progress in order to get a robust design that works on multiple clock.

A single clock design is shown in [Figure 1](#). In a single clock domain, there is a single clock that goes through the entire design. Such designs are easy to implement and as compared to multiple clock designs, pose less problems of Metastability, Setup & Hold time violations.

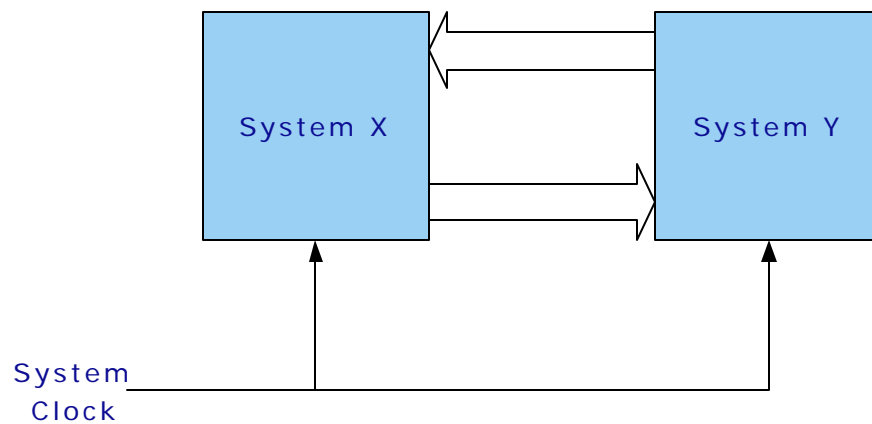


Figure 1: Single clock Domain

2. Multiple Clock Domains

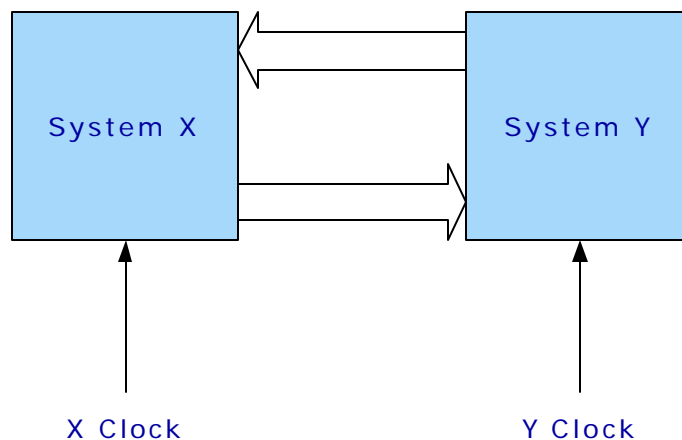
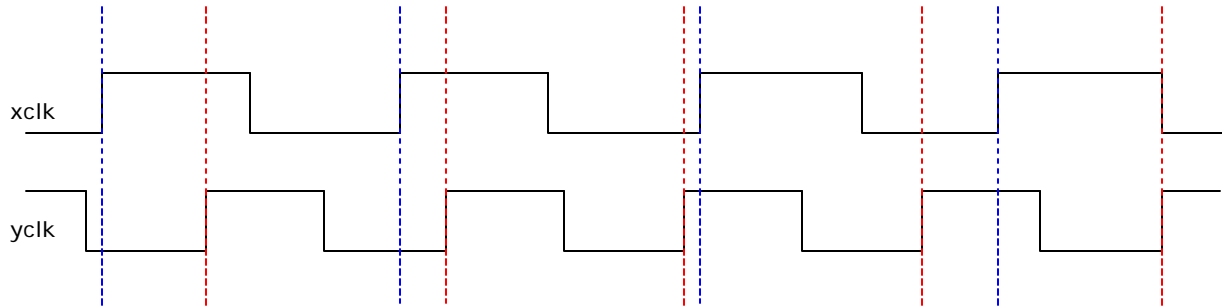


Figure 2: Multiple clock domain system

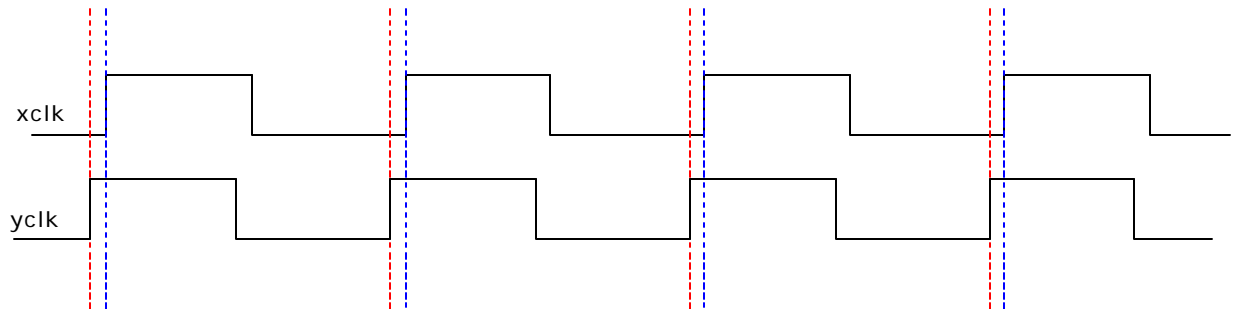
One of the challenges faced by an engineer is developing a design having multiple clocks. Designs with multiple clocks ([Figure 2](#)) have the following type of clock relationships:

- Clocks with different frequencies
- Clocks with same frequency but different phases between them

The two relationships between clocks are shown below in [Figure 3](#).



Clocks with different frequencies



Clocks having same frequency but different phase

Figure 3: Relationship between clocks

3. Problems in design with multiple clock domains

It is well known that “There is no gain without pain”. So in order to make a robust design working with multiple clocks, one should know the problems that are faced in such a design. The problems generally faced in such designs are:

- Setup Time & Hold Time Violation
- Metastability

Setup Time & Hold Time Violation:

Setup Time: The time required for data input to remain stable prior to arrival of clock pulse

Hold Time: The time required for data input to remain stable after the arrival of clock pulse

[Figure 4](#) shows these times with respect to the rising edge of clock.

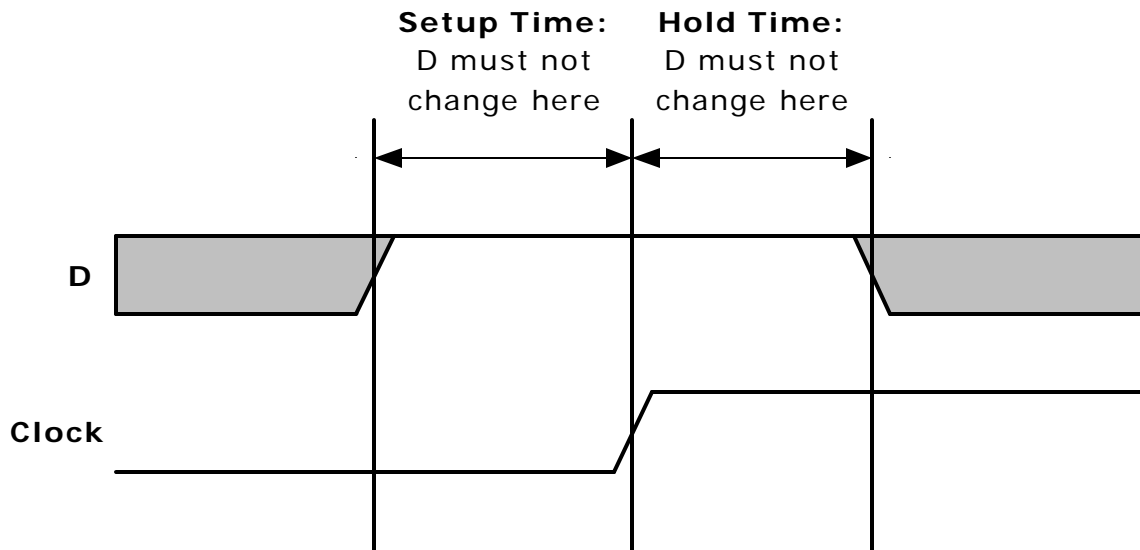


Figure 4: Setup Time & Hold Time

Setup Time requires that input should become stable before the rising edge of the clock while, Hold Time requires that input remains stable after the arrival of clock pulse. This can be easily achieved in single clock domains. However, in multiple clock domains, it may happen that the output from one clock domain may be changing when the rising edge of the second clock domain comes. This will cause the output of the second clock domain to become metastable, thereby leading to wrong results.

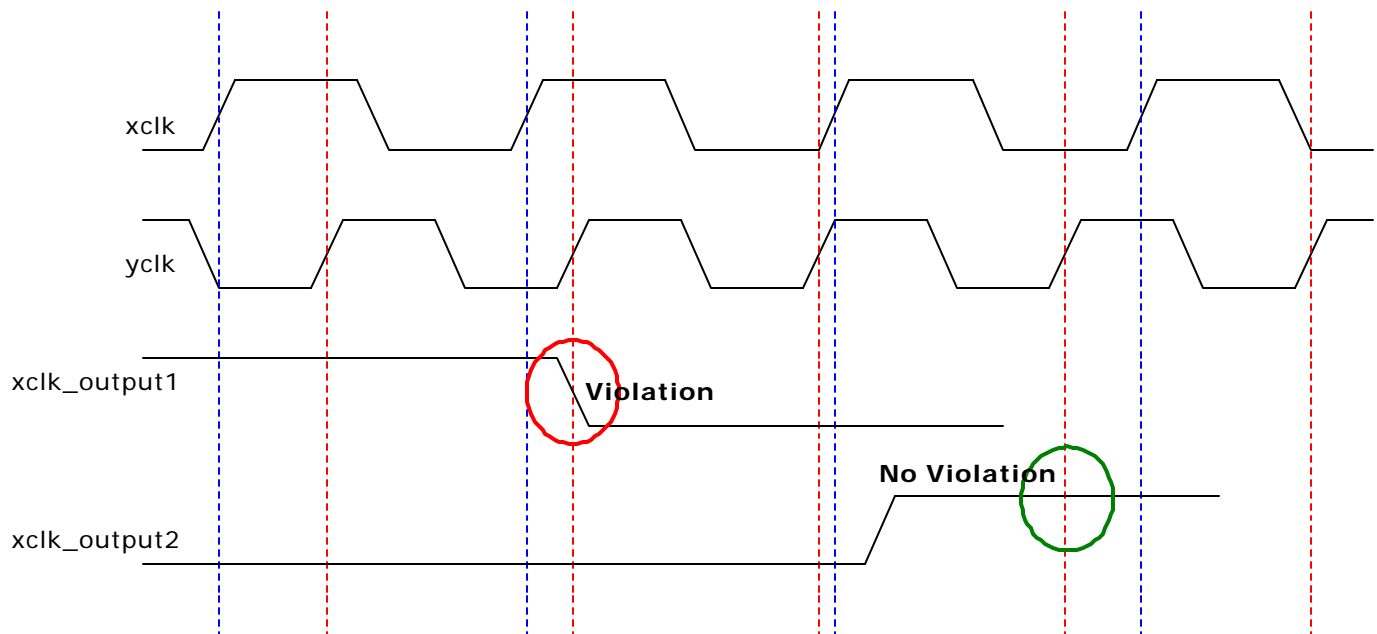


Figure 5: Violation of Setup & Hold Times

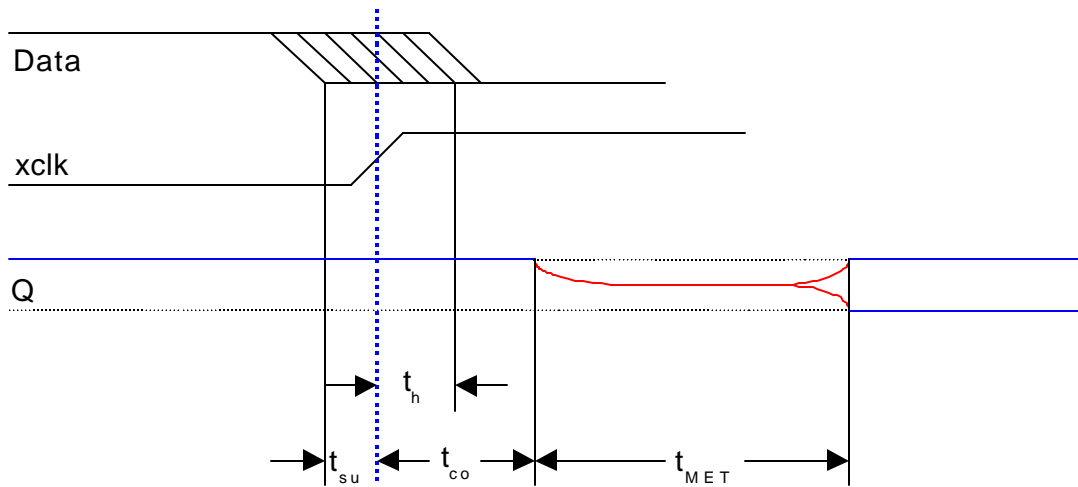
Consider a dual clock system shown in [Figure 2](#). Its timing for transfer of signals between domains is shown in [Figure 5](#). By looking at [Figure 5](#), we see that xclk_output1 (belonging to the xclk domain) changes near the rising edge of yclk. When this signal is sampled by yclk, the xclk_output1 signal is in transition stage called “Metastable” state. This causes violation of setup & hold times w.r.t. to yclk. Thus the signals in yclk domain that depend on xclk_output1 will go into metastable state and give wrong results. However, xclk_output2 (belonging to the xclk domain) is stable at the rising edge of yclk. There are no violation of setup & hold times. Thus the signals in yclk domain that depend on the xclk_output2 signal will give correct output.

Metastability:

This problem arises as a result of violation of setup and hold times. Every flip-flop that is used in any design has a specified setup and hold time, or the time in which the data input is not legally permitted to change before and after a rising clock edge. If the signal does change during this time window, the output will be unknown or “metastable”. This propagation of unwanted information is called Metastability. As a result the output of a flip-flop can produce a glitch or remain temporarily in metastable state, thus taking longer to return to stable state.

When a flip-flop is in a metastable (“in between”) state, the output hovers at a voltage level between high and low, causing the output transition to be delayed beyond the specified clock-to-output delay (t_{co}). The additional time

beyond t_{co} that a metastable output takes to resolve to a stable state is called the settling time (t_{MET}). This has been shown in [Figure 6](#). Not every transition that violates the setup or hold times results in a metastable output. The likelihood that a flip-flop enters a metastable state and the time required to return to stable state depends on the process technology used to manufacture the device and on the ambient conditions. Generally, flip-flops will quickly return to a stable state.



Legend:

- ⋮ Rising Edge of Clock
- Metastability Region
- Stable Region

Figure 6: Metastability Timing Parameters

The operation of a flip-flop is analogous to a ball rolling over a frictionless hill, as shown in [Figure 7](#).

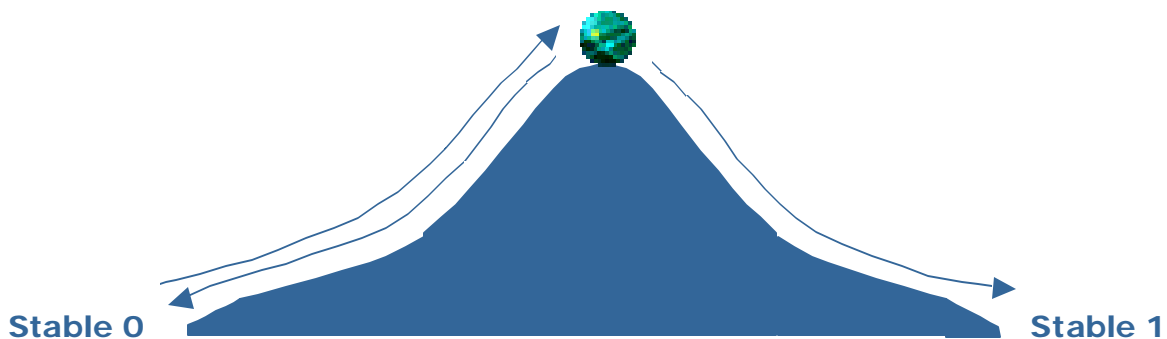


Figure 7: Behavior of a FF

Each side of the hill represents a stable state (i.e. high or low) and the top represents a metastable state. Suppose the ball is in a stable state (i.e. either 1 or 0) and a push (state transition) is given to the ball that is sufficient (no setup or hold time violations) enough to make the ball cross over to the other stable state. The ball crosses to the other stable state within the specified time.

However, if the push is less (i.e. violation of setup & hold time), the ball shall travel to the top of the hill (i.e. output metastable), stay there for sometime and return to either stable state (i.e. output becomes stable eventually). It may also happen that the ball may rise partially and come back (i.e. output may produce some glitches). Either condition increases the delay from clock transition to a stable output.

Thus, in simple words, when a signal is changing in one clock domain (src_data_out) and is sampled in another clock domain (dest_data_in), then this causes the output to become metastable. This is known as Synchronization Failure.

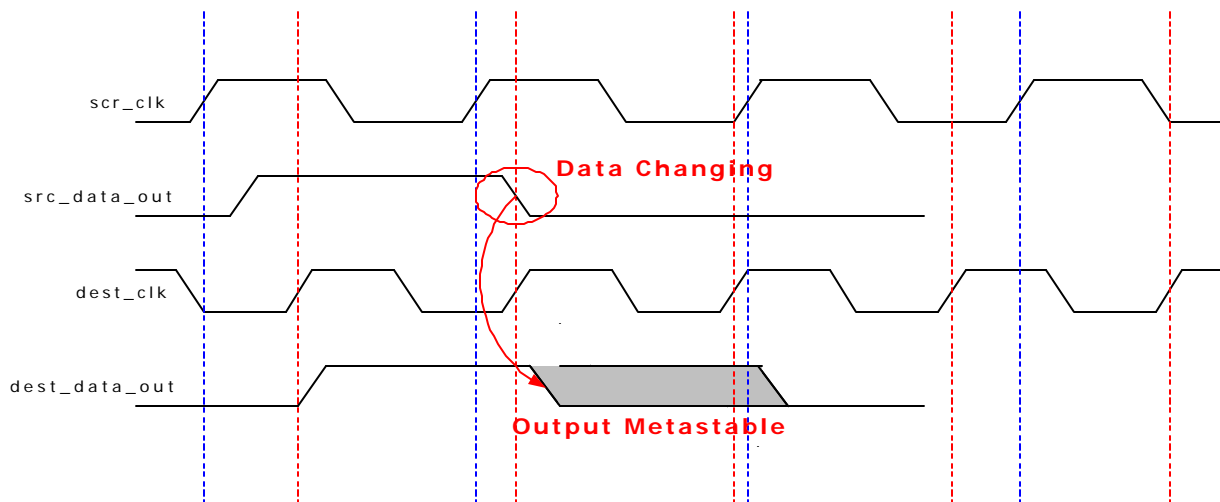


Figure 8: Metastability in Flip Flops

Again, if we look at [Figure 5](#), we see that signal xclk_output1 is violating the setup and hold times, i.e. it is changing at the time the yclk samples it. This will cause the signals (that depend on xclk_output1, say yclk_input) in yclk clock domain to become metastable for sometime. However, the duration of this metastability condition may persist over more than one clock, thereby making internal signals (those signals that depend on yclk_input) of yclk domain to become metastable.

4. We shall Overcome...

Every cloud has a silver lining. And so do these dark clouds discussed in [Section \(3\)](#) have a silver lining. In this section solutions and tips have been provided that will help a designer make a robust multiple clock design. Overcoming the problems in multiple clock designs can be done in the following ways:

- Design Tips for efficient handling of a design with multiple clocks
 - Clock Nomenclature
 - Design Partitioning
- Avoiding metastability & violations of setup & hold times while transferring control & data signals

Design Tips:

When working on a design with multiple clocks, it is beneficial that one follows certain guidelines to help during simulation & synthesis. Some common guidelines are:

- Clock Nomenclature
- Design Partitioning

Clock Nomenclature

For easy handling of clock signals by synthesis scripts, it is necessary that there should be a certain clock naming procedure that is used all over the design. For example, system clock be named as sys_clk, transmitter clock be named as tx_clk and receiver clock as rx_clk, etc. This would help during scripting to refer to all clock signals using wildcards. Similarly, signals belonging to a particular clock domain can be prefixed by its clock name. For example, signals clocked by system clock can start as sys_rom_addr, sys_rom_data.

Using this nomenclature any engineer on the team can identify to which domain a particular signal belongs. Thus the engineer can easily decide whether to use the signal directly or through a synchronizer.

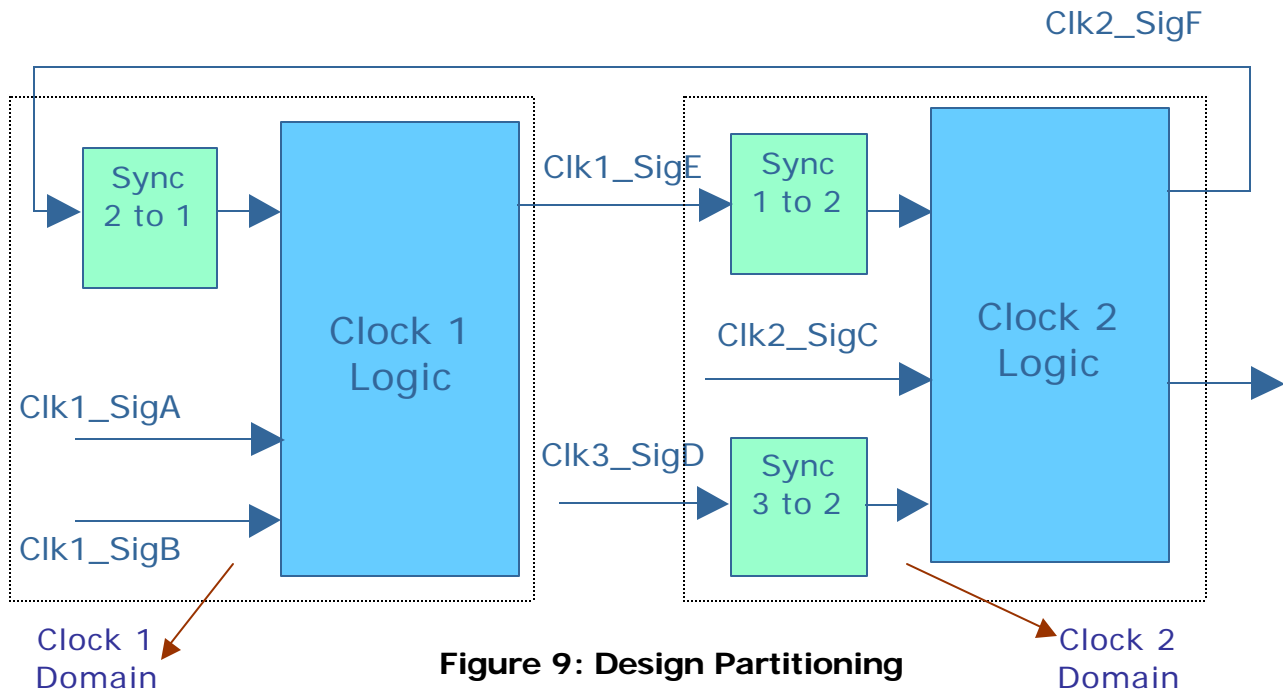
This naming procedure can significantly reduce confusion and provide easy interfacing among modules thereby increasing the efficiency of the team.

Design Partitioning

This is another technique for efficient designing of modules having multiple clocks. According to this guideline, (1) a module should work on one clock only, (2) a synchronizer module (module that performs the function of transferring signals from one domain to another) be made for all signals that

cross from one clock domain to another, so that all inputs are synchronized to the same clock before entering the module and (3) the synchronizer module should be as small as possible.

The advantage of partitioning a design is that static timing analysis becomes easy if all signals entering or leaving a clock domain are made synchronous to the clock used in that module. The design becomes completely synchronous. Also no timing analysis is required on the synchronization modules. However, it should be ensured that hold time requirements are met.



As shown in [Figure 9](#), we have separated the entire logic into three clock domains viz Clock1, Clock2 and Clock3 domains. The names to the signal have been given as per the nomenclature explained in Section [Clock Nomenclature](#). Now any signal going from one clock domain to another passes through an external synchronization module. This synchronization module converts the clock domain of the signal to the clock domain used by the module. Thus, in the above figure, Sync 1 to 2 module converts the signals coming from clock1 domain to clock2 domain.

The process of synchronization is explained later in this document.

Transferring signals between clock domains:

Setup or Hold time violation and metastability are major problems in multiple clock designs that are faced during transferring signals between different clock domains. The transfer of signals can be categorized into two groups, namely:

- Transfer of Control Signals
- Transfer of Data Signals

(a) Transfer of Control Signals (Synchronization)

If an asynchronous signal is directly fed to several flip-flops, the probability that a metastable event will occur greatly increases because there are more flip-flops that could become metastable. To avoid such a condition of metastability, the output of the synchronizing flip-flop is used rather than the asynchronous signal.

To reduce the effects of metastability, designers most commonly use a multiple stage synchronizer in which two or more flip-flops are cascaded to form a synchronizing circuit shown in [Figure 10](#).

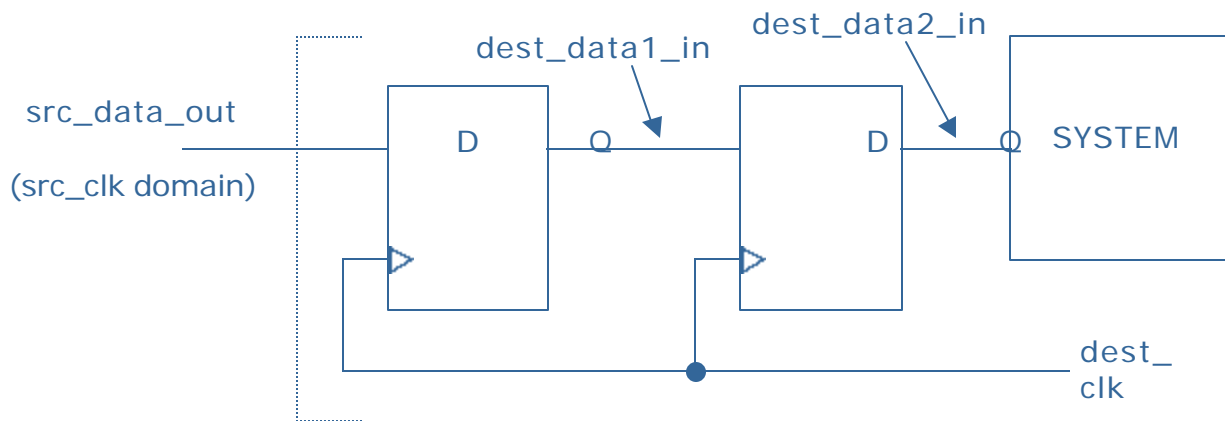


Figure 10: Two-stage synchronizer circuit

If the synchronizing flip-flop produces a metastable output, the metastability may get resolved before it is sampled by the second flip flop. This method does not guarantee that the output of the second flip-flop will go metastable but it surely decreases the probability of metastability.

One drawback or more aptly called “necessary evil” of the synchronizer circuit is that it adds up clocks to the total latency of the circuit.

The timings of the synchronizer circuit have been shown in [Figure 11](#). The asynchronous output (src_data_out) from the source module working on src_clk is fed to the first synchronizer flip-flop. The signal dest_data1_in (output of first synchronizing flip-flop) goes metastable but resolves to a stable state before it is sampled by the second flip-flop in the synchronizer circuit. Thus the signal dest_data2_in (output of the second synchronizer flip-flop) is synchronized to the dest_clk used by the destination module.

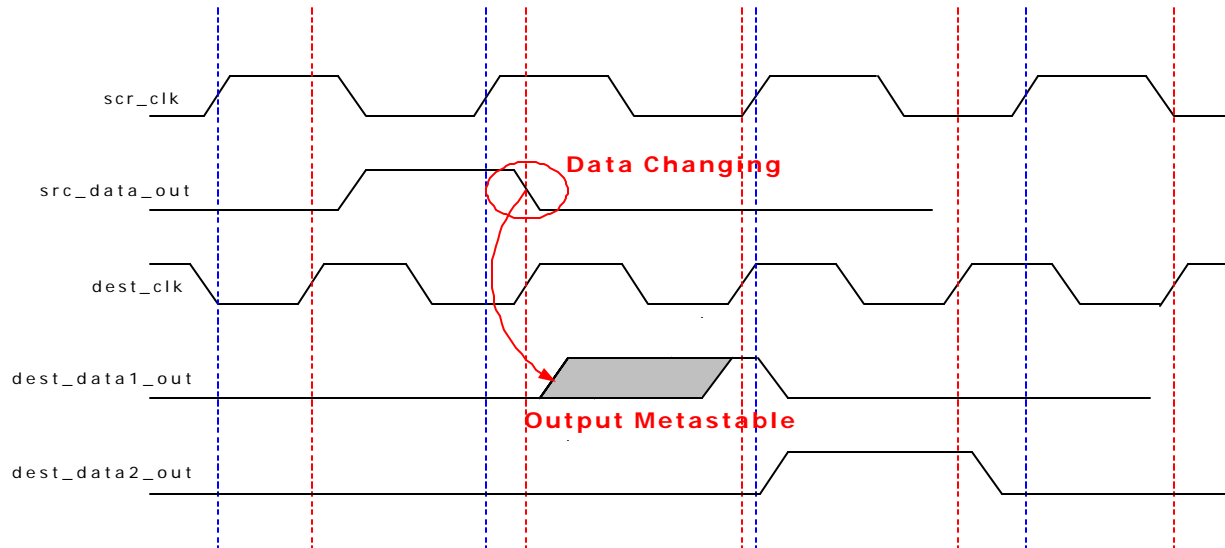


Figure 11: Timings for the two-stage synchronizer circuit

In some cases, it may happen that the output of the first synchronizer flip-flop takes longer than one clock to resolve from a metastable state to a stable state. If a two-stage synchronizer circuit is used here, the output of the second synchronizer flip-flop will also go metastable. Thus, we are back to where we started. The sole purpose of the synchronizer goes in vain. In such cases, it is safe to use a three-stage synchronizer circuit.

A three-stage synchronizer circuit is shown in [Figure 12](#).

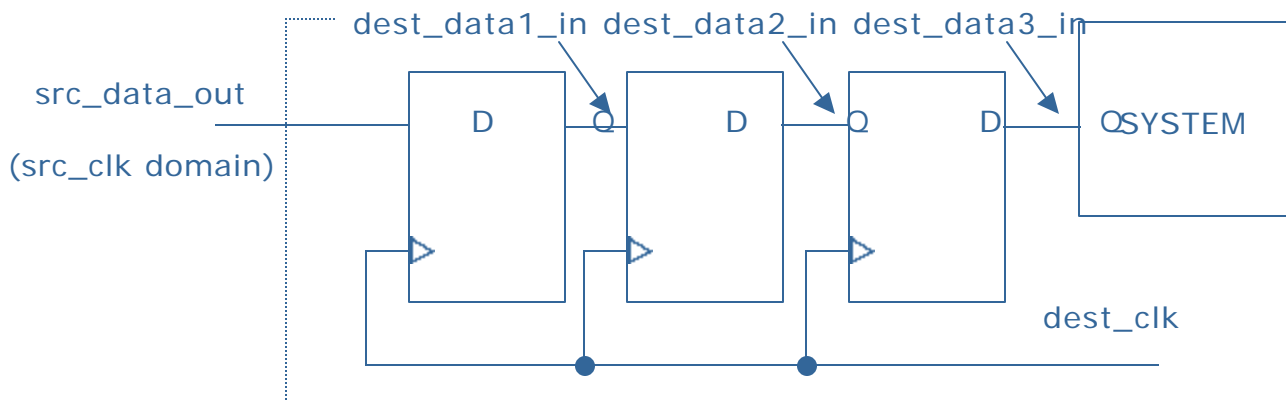


Figure 12: Three-stage synchronizer circuit

A three-stage synchronizer comprises of three cascaded flip-flops. As the second flip-flop output goes metastable, it resolves to a stable state before it is sampled by the third flip-flop.

Timings of the three-stage synchronizer circuit have been shown in [Figure 13](#). The asynchronous output (src_data_out) from the source module working on src_clk is fed to the first synchronizer flip-flop. The signal dest_data1_in (output of first synchronizing flip-flop) goes metastable but resolves to a stable state after more than one clock duration. When the second flip-flop samples the output of the first flip-flop, the signal dest_data2_in (output of the second synchronizer flip-flop) also becomes metastable. But this signal resolves to a stable state before it is sampled by the third flip-flop. Thus the asynchronous output of the src_clk domain is now synchronized to the dest_clk domain.

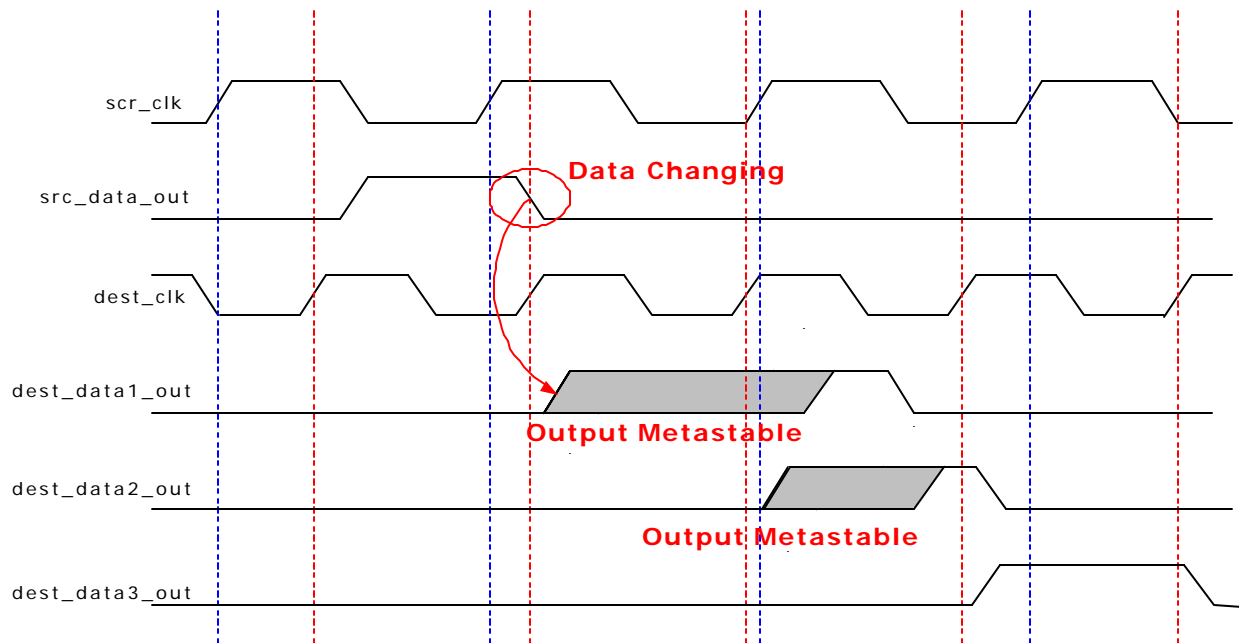


Figure 13: Timings for the three-stage synchronizer circuit

However, a two-stage synchronizer circuit is sufficient to avoid metastability in multiple clock designs. A three-stage synchronizer is required in designs where the clock frequencies are very high.

(b) Transfer of Data Signals

Multiple clock design often requires data transfer from one clock domain to other clock domain. Below are the two commonly used methods for Data Synchronization between two clock domains.

- Handshake signaling method.
- Asynchronous FIFO

5. Handshake Signaling Method

This is one of the oldest methods used to pass on the data from one domain to other.

Treating two-clock domain as two different systems as shown below: -

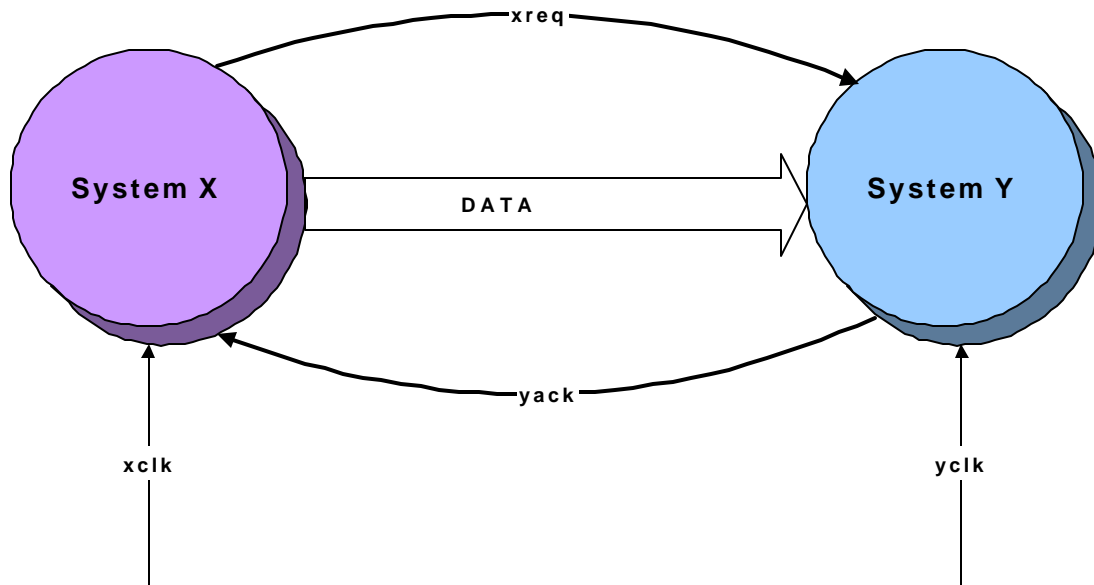


Figure 14: Two-clock system divided into two separate systems

System X sends data to system Y based on the handshake signals "**xack**" and "**yreq**" signals.

Below is the sequence of transfer of data for handshake signaling: -

- Transmitter (System X above) places the data on the data bus and asserts a "**xreq**" (request) signal indicating valid data on the data bus of the receiver (System Y above).
- "**xreq**" signal is synchronized with the receiver clock domain (system Y clock above "**yclk**")
- Receiver (System Y above) latches the data on the data bus on recognition of synchronized "**xreq**" signal: "**yreq2**".
- Receiver (system Y above) asserts the "**yack**" (acknowledge) signal, asserting that it has accepted the data.
- "**yack**" is synchronized to the transmitter clock (System X clock above "**xclk**").
- When the transmitter (system X above) recognizes the synchronized acknowledge signal ("**xack2**"), the transmitter can change the value driven on the data bus.

Following is the timing waveform for the above sequence of operation:-

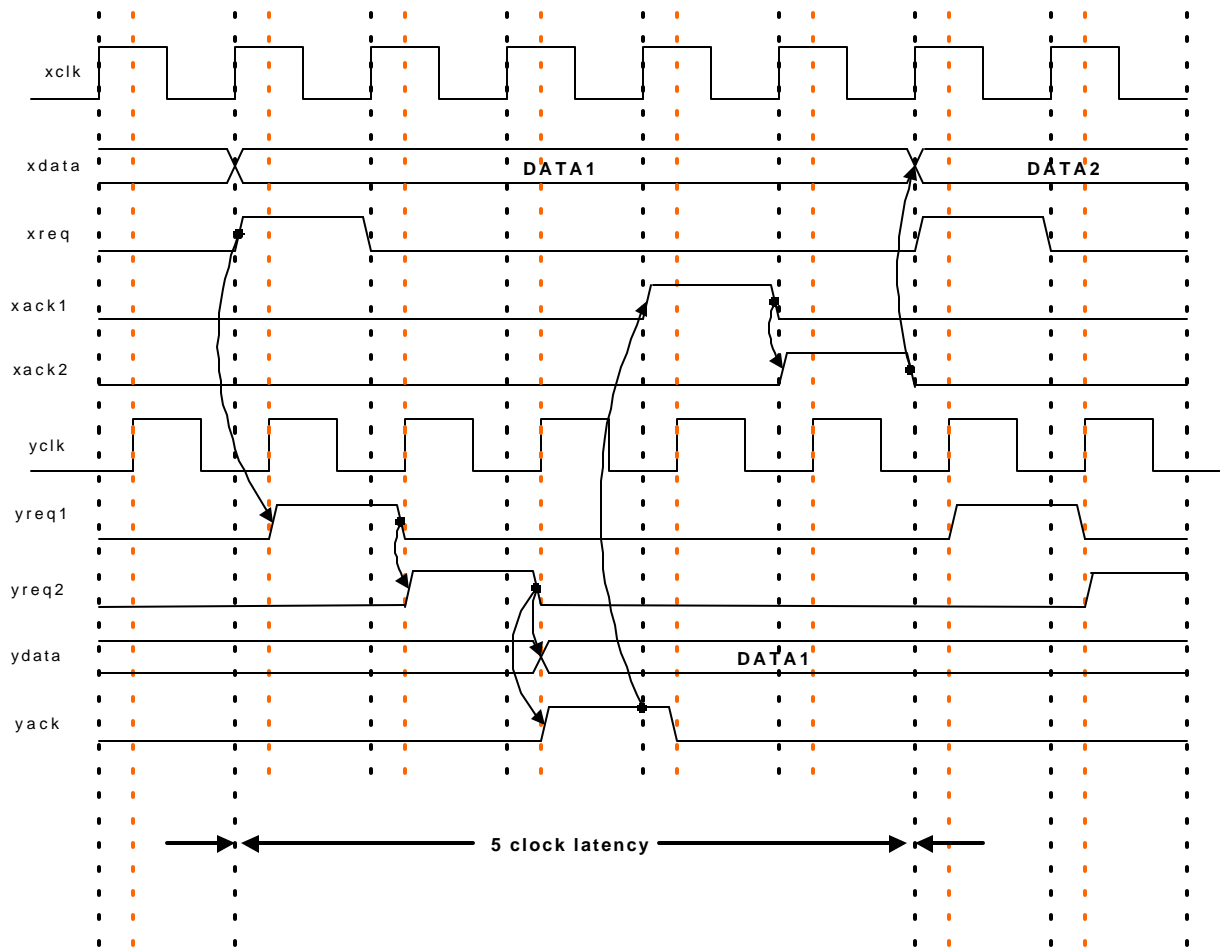


Figure 15: Timings for Handshaking Method of Data Transfer

As shown above it takes 5 clocks to transfer single data from transmitter to receiver safely.

Requirements for the above operation

1. Data should be stable for atleast two rising clock edges in the sending clock domain.
2. Request signal ("xreq" above) should be of more than 2 rising edge clock else this signal won't get captured if passed from faster clock domain to slower clock domain as shown in [Figure 16](#).

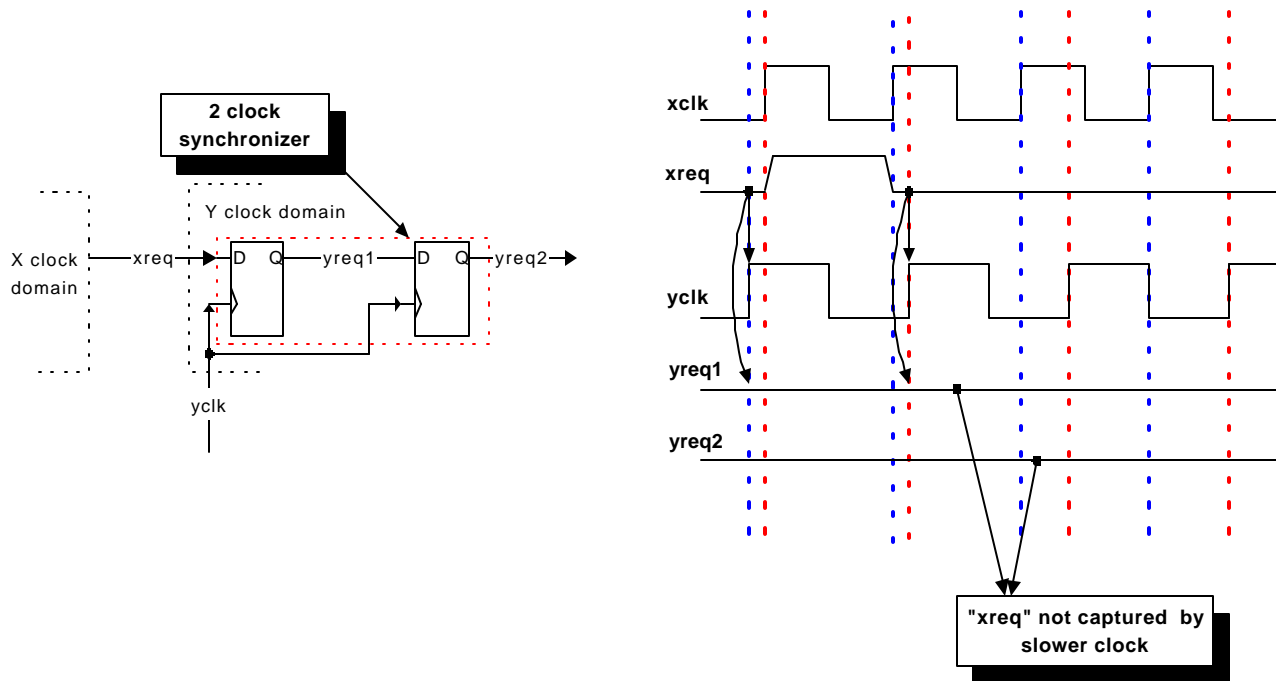


Figure 16: Constraint on xreq signal

Disadvantages of data transfer through handshaking signaling

"Latency" for a single data transfer from one clock domain to other is much more than that of a FIFO used for the same data transfer.

6. Data transfer through Synchronous FIFO

During a system design, there are many components that work on different frequencies like processor, memory etc. and they might have their own clock crystal. First-In-First-Out queues play an important role in the exchange of data between such devices. FIFOs are simple memories that are used to queue up data transmitted over communication buses.

Thus FIFOs are usually used for domain crossing, and therefore dual clock design. Lets start up with a simple synchronous FIFO with reading and writing done on the same clock and later on we will move to asynchronous FIFO with reading and writing done on different clock frequencies.

Synchronous FIFO Architecture

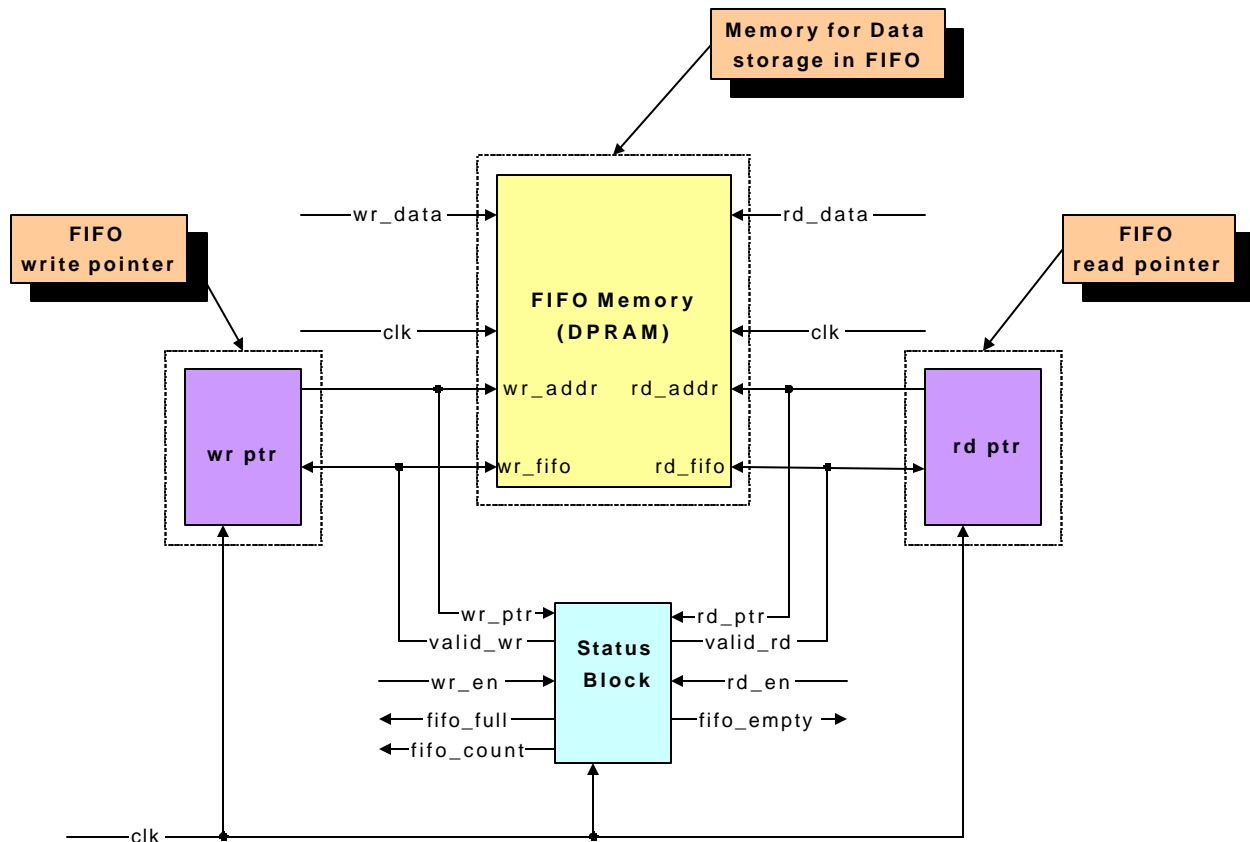


Figure 17: Synchronous FIFO Architecture

Figure above shows a general architecture of a Synchronized FIFO. DPRAM (Dual port RAM) is used as a FIFO memory since we require reading and writing to be independent.

The read and write ports have separate read and write addresses generated by two read and write pointers. The write pointer points to the location that will be written next and the read pointer to the location that will be read next. A valid write enable increments the write pointer and a valid read enable increments the read pointer.

Shown above in the figure, is a *status* block that generates the "fifo_empty" and "fifo_full" signal. If "fifo_full" is asserted it means that there is no room for more data to be written into the FIFO. Similarly "fifo_empty" indicates that there is no data available in the FIFO to be read by the external world. This block also indicates the number of empty or full locations in the FIFO at any point of time by performing some logic on the two pointers.

The *DPRAM* above can have either synchronous reads or asynchronous reads. For synchronous read, an explicit read signal is supposed to be provided before the data at the output of the FIFO is valid. For asynchronous reads, *DPRAM* does not have registered outputs; valid data is available as soon as it is written (data is read first and then the pointer is incremented).

Working of Synchronous FIFO

On reset, both the read and write pointer are '0'. "fifo_empty" signal is asserted and "fifo_full" remains "low" during this time. Further reads from FIFO are blocked when FIFO is empty so only write operation is possible. Now subsequent writes on the FIFO increments the write pointer and deasserts the FIFO empty signal. A point is reached where there is no room for more data when the write pointer equal to $\text{RAM_SIZE} - 1$. At this time a write causes the write pointer to again roll back to '0' and "fifo_full" signal is asserted.

Looking above we have the same condition when the FIFO is empty or full that is when read pointer equals to the write pointer. It is necessary to distinguish between these two conditions.

FIFO full and FIFO empty generation

Lets first look at the FIFO full generation by taking an example.

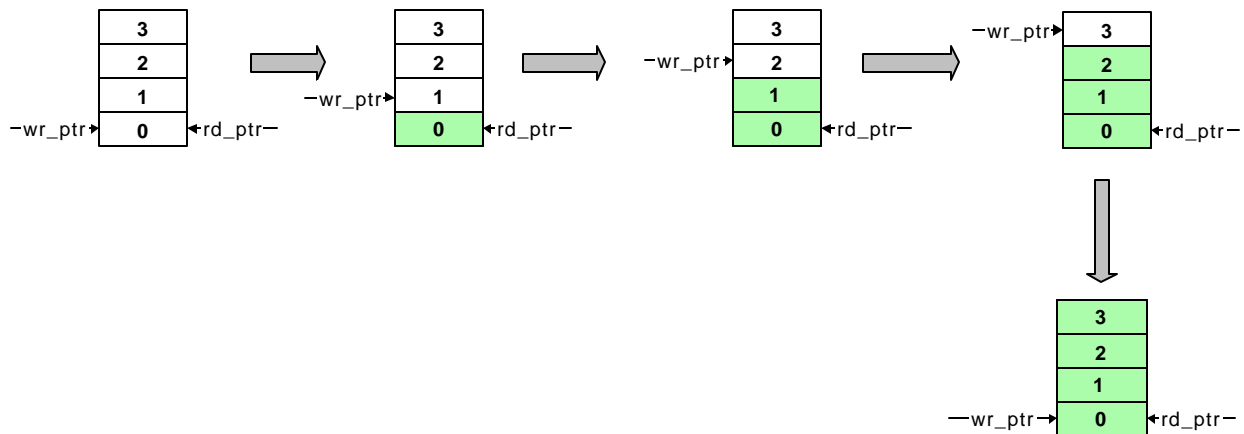


Figure 18: FIFO Full Condition

All the transitions shown in above figure are in subsequent clocks. As shown above in the figure if a write causes both the pointers to become equal in the next clock, then FIFO becomes full. So we have the following condition for assertion of fifo_full signal.

When $(\text{read_pointer} = \text{write_pointer} + 1)$ and "write" \rightarrow asserts the fifo_full signal

Below is the verilog code for the above logic: -

```
always @ (posedge clk or negedge reset_n)
begin: fifo_full_gen
    if (~reset_n)
        fifo_full <= 1'b0;
    else if (wr_fifo && rd_fifo)
        ; //do nothing
    else if (rd_fifo)
        fifo_full <= 1'b0;
    else if (wr_fifo && (rd_ptr = wr_ptr + 1'b1 ))
        fifo_full <= 1'b1;
end
```

Similarly if a read causes both the pointers to become equal in the next clock, then FIFO becomes empty. So we have the following condition for assertion of fifo_empty signal.

When (write_pointer = read_pointer + 1) and "read" → asserts the fifo_empty signal

Below is the verilog code for the above logic: -

```
always @ (posedge clk or negedge reset_n)
begin: full_gen
    if (~reset_n)
        fifo_empty <= 1'b1;
    else if ( wr_fifo && rd_fifo)
        ; //do nothing
    else if (wr_fifo)
        fifo_empty <= 1'b0;
    else if (rd_fifo && (wr_ptr = rd_ptr + 1'b1 ))
        fifo_empty <= 1'b1;
end
```

An alternative approach

An alternative way of generation the full and empty conditions are by maintaining a counter that constantly indicates the number of full/empty locations left on the FIFO.

Width of the counter is equal to the depth of the FIFO so that it can store the maximum count value. Now initially on reset the counter value is zero. On subsequent writes the counter is incremented by one and on subsequent reads the counter is decremented by one.

Now FIFO empty condition can easily be generated when the counter values reaches ZERO and FIFO full condition when counter's values equals the size of the FIFO. But this implementation is not efficient as compared to our previous implementation since it requires additional hardware (comparators) for the generation of FIFO empty and FIFO full conditions. As the depth of the FIFO increases so is the width of the counter, thus requiring higher order comparators for FIFO empty and FIFO full condition generation that finally lowers the maximum frequency of operation of the FIFO.

7. Asynchronous FIFO (Dual Clock FIFO)

Unlike handshake signaling, Asynchronous FIFO is used in case of performance critical designs where clock latency is a factor rather than system resources (Area, no of LCs).

Figure below shows a case of two systems X and Y respectively where data from system X is supposed to be transferred to system B both systems working at different clocks.

System X writes the data on "**xclk**" clock into FIFO and is read out by System B on "**yclk**" clock.

To take care of the underflow and overflow conditions, we have FIFO full and FIFO empty signals.

Overflow condition is taken care by FIFO full signal: Data is not written into the FIFO if FIFO Full signal is asserted else Data will be overwritten.

Underflow condition is taken care by FIFO empty signal: Data is not read from the FIFO if FIFO empty signal is asserted else junk data would be read.

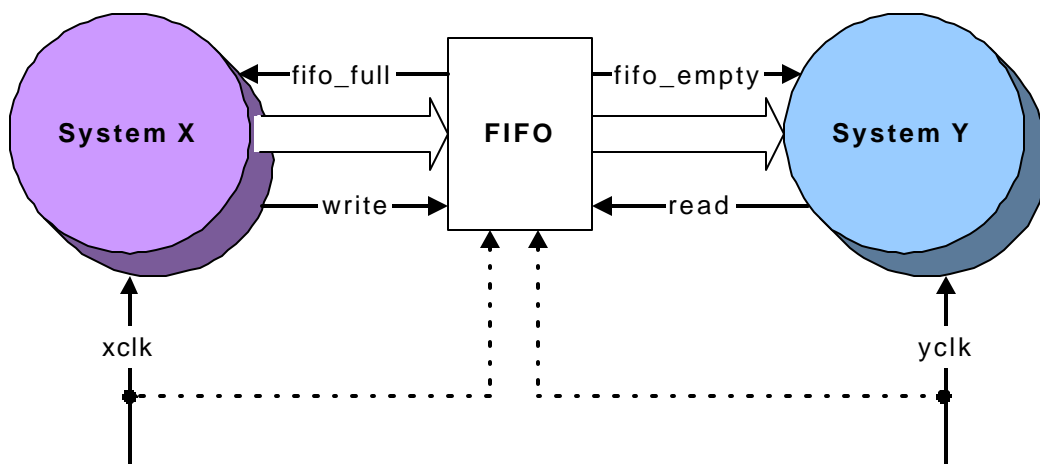


Figure 19: Asynchronous FIFO (Dual Clock FIFO)

A simple [Synchronous FIFO](#) can be implemented using Dual Port RAM with separate ports for read and write operations where reading and writing done on the same clock.

The same concept can be extended for designing Asynchronous FIFO with special care taken for FIFO empty and FIFO Full signal generation to avoid metastability conditions.

Avoid using Binary counters for Pointer implementation of Asynchronous FIFO

Take the case of write pointer. Write pointer is always incremented on write clock whenever there is a valid write request to a FIFO. Read pointer is incremented on read clock whenever there is a valid read request. Now for FIFO Full signal generation, write pointer needs to be compared with read pointer and since both the pointers are synchronous to their respective clock but asynchronous to each other will result in wrong sampling of the pointer values for comparison if binary counters are used for the pointer implementation.

This is illustrated as shown below.

Say, the binary counter is changing from FFF to 000. In this case all the bits will change at the same time. Metastability can be avoided by synchronizing the counter, but you may still get sampled values that are widely off the mark, so synchronizing the counters is not the final solution.

Possible transitions from FFF to 000

FFF → 000
FFF → 001
FFF → 010
FFF → 011
FFF → 100
FFF → 101
FFF → 110
FFF → 111

If the synchronizing clock edge comes in the middle of the transition from FFF to 000, it is possible that any of the 3 bit binary value be sampled and synchronized in new clock domain.

Since the generation of FIFO full and FIFO empty flags depends on these pointer value, incorrect value of these pointers will result in wrong triggering of the flags. There might be a case where FIFO full flag not getting triggered

even when actually FIFO is full resulting in data getting lost or FIFO empty flag not getting triggered resulting in junk data being read.

NOTE: Looking at the above case, it's highly unrecommended to use binary counter for read and write pointers implementation.

Gray Code Fundamentals

One way of implementing FIFO pointers is to make them count in Gray-code.

The advantage of the Gray code over pure binary numbers is that a number in Gray code changes by one bit as it proceeds from one number to the next.

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

The Gray code is shown in the table above.

To obtain a different Gray code, one can start with any bit combination and proceed to obtain the next bit combination by changing only one bit from 0 to 1 or 1 to 0 in any desired random fashion, as long as two numbers do not have identical code assignments. The Gray code is also known as *reflected* code. [4]

Since gray code is a unit distance code, every next value differs from previous in one bit position, will result in a maximum of a single bit error/transition.

Now for example if the counter's actual value is "1010" and the counter changes to "1011", then you will read either "1010"(old value) or "1011"(new incremented value) but no other value.

Note: Synchronizing gray counter will rarely result in sampled counter value getting metastable and secondly the value sampled will have at most one bit error.

Effect of synchronization on pointers

Now suppose we wish to know whether or not the FIFO is full. If FIFO is full, we should block further accesses. For the FIFO full condition we compare the read and write pointer that are incremented on their respective clocks.

We synchronize the read pointer (Gray coded) to the write clock. Let us see how this works by taking an example.

As shown in [Figure 20](#), initially read and write pointers are zero at t_0 and FIFO is empty. As subsequent write takes place on FIFO, write pointer is incremented. A stage is reached when write pointer equals read pointer and FIFO becomes FULL. This happens at t_5 as shown in [Figure 20](#).

Suppose a read takes place at t_6 , now since a typical synchronizer circuit consists of at least 2 flip flops, synchronizing read pointer on write clock will result in changed read pointer reflected after 2 write clocks. This results in blocking additional writes on the FIFO and is harmless. It would have been a problem if we did not block writes when the FIFO was actually full.

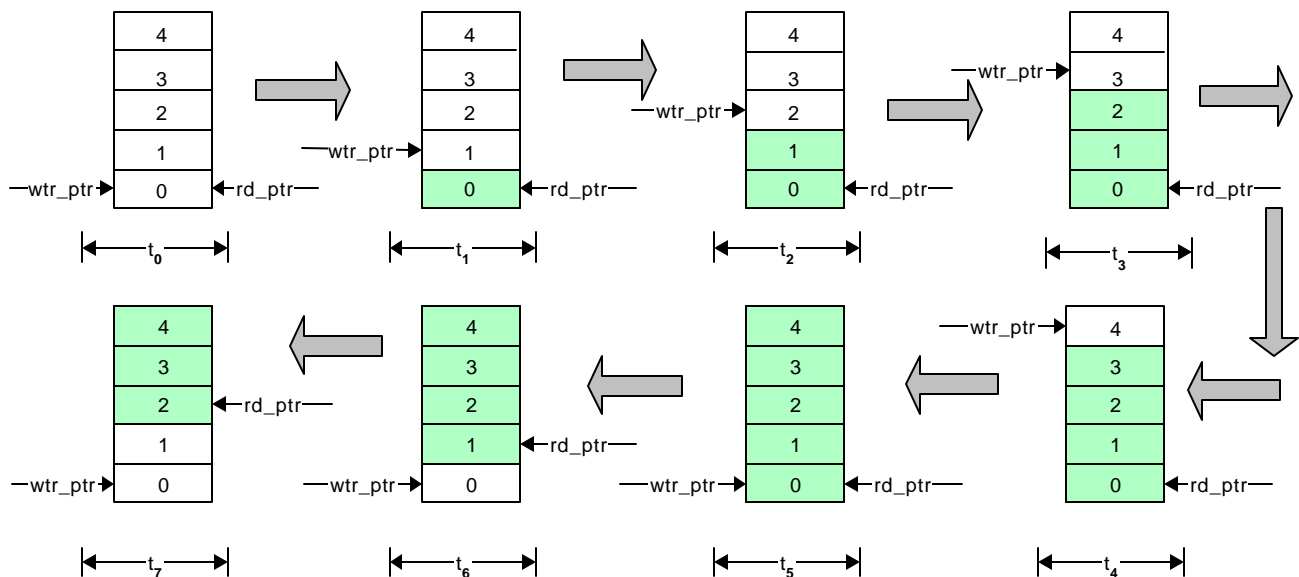


Figure 20: Effect of synchronization on FIFO full logic

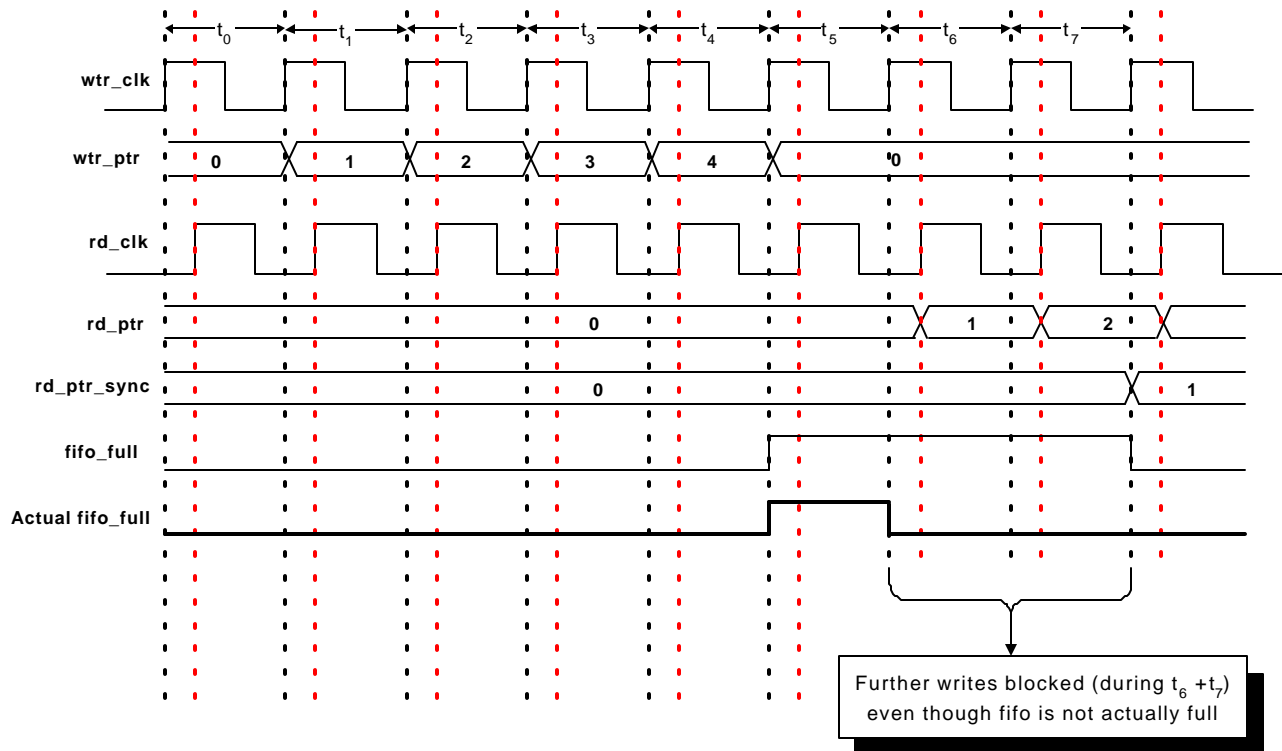


Figure 21: FIFO Full Illusion

Now suppose take the case when one wants to know whether or not FIFO is empty. In that case, further read access to FIFO should be blocked.

Now Read and Write pointers are supposed to be compared for FIFO empty calculation. We synchronize the write pointer to the read clock. So read side sees delayed writes (2 clock delayed signal), and may decide that the FIFO is empty when it actually has some data. This will result in reading getting blocked till the writes becomes visible to the read side.

As shown in [Figure 21](#), initially read and write pointers are zero at t_0 and FIFO is empty, as subsequent write takes place on FIFO, write pointer is incremented. A stage is reached when write pointer equals to read pointer and FIFO becomes FULL. This happens at t_3 as shown in [Figure 21](#).

Suppose subsequent reading starts at t_4 and again FIFO becomes empty at t_6 . Assume that the FIFO is again written at t_7 and t_8 . Now since a typical synchronizer circuit consists of at least 2 flip flops, synchronizing write pointer on read clock will result in changed write pointer reflected after 2 read clocks. This results in blocking additional reads on FIFO and is harmless. It would have been a problem if we did not block reads when the FIFO was actually empty.

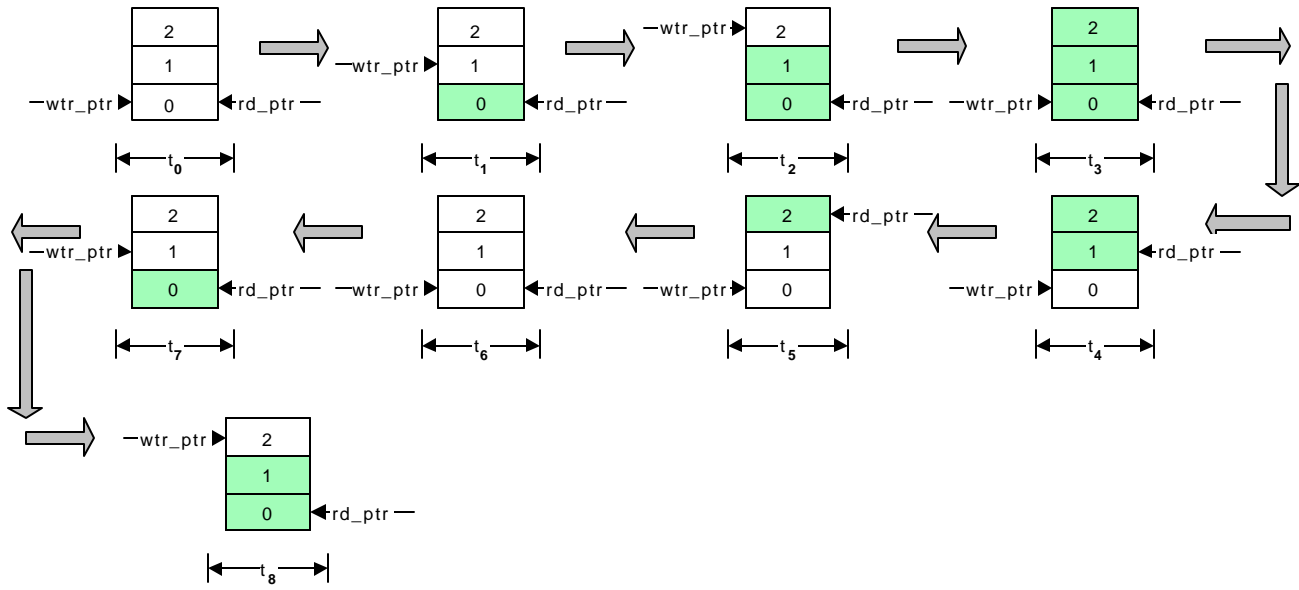


Figure 22: Effect of synchronization on FIFO empty logic

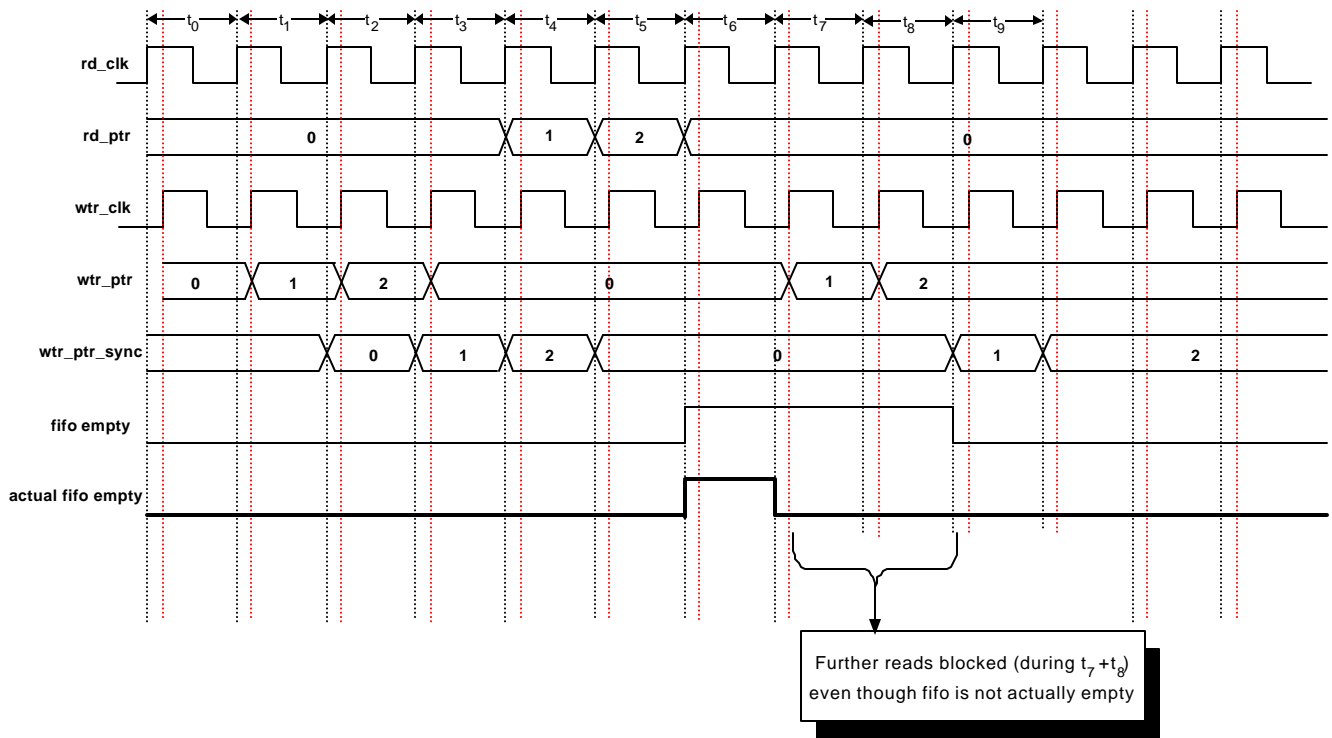


Figure 23: FIFO Empty Illusion

Note: Reporting to the write side that FIFO is full when it is not is OK, and so is reporting to the read side that the FIFO is empty when it is not. Even if the

synchronized values of the pointer (synchronized read pointer during write and synchronized write pointer during read) remains metastable for a small period of time, the effect would be to block writes/reads causing the FIFO to hand for a while, but not causing any errors.

Gray code implementation of FIFO pointers

We need to correctly sample Read and Write pointer values for perfect generation of FIFO empty and FIFO full conditions. So the best way for passing pointers between clock domains is to use a gray code counter for pointers implementation, since they would eliminate most of the errors if the synchronized clock signal comes in the middle of the counter transition.

Designing a gray code counter seems quite complex but is indeed simple. All that is supposed to be done is the following: -

- STEP I : Convert the Gray value to Binary value.
- STEP II : Increment the Binary value depending on some condition.
- STEP III : Convert the Binary value back to Gray.
- STEP IV : Store the final Gray value of the counter in a registers.

[Figure 24](#) shows the generalized Gray counter.

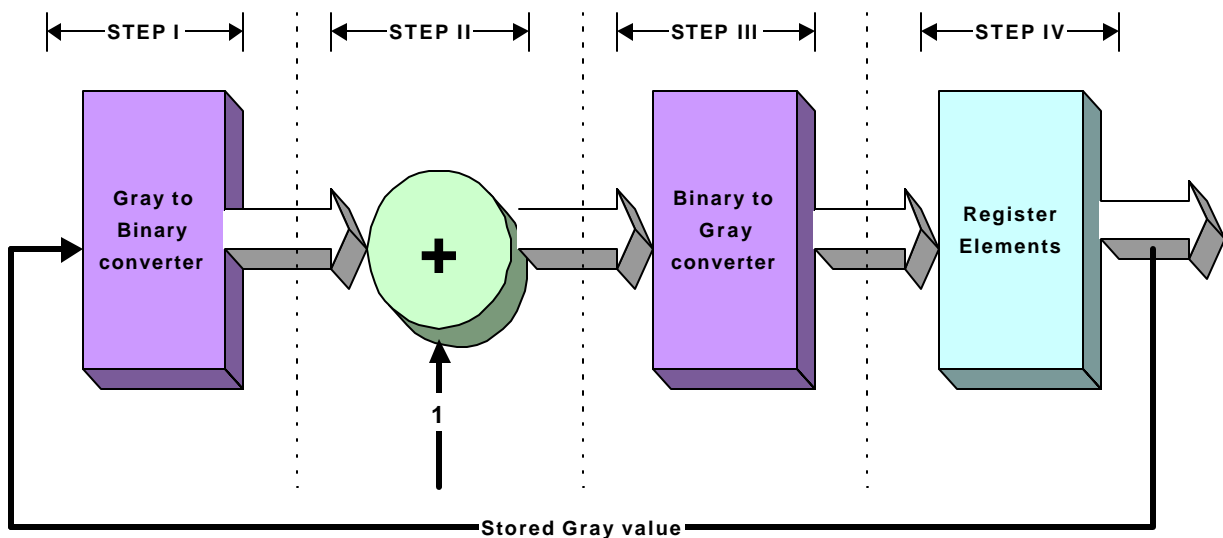


Figure 24: Gray counter using binary adder

Gray to binary converter

Table below shows the 4 bit counter values when counted in Gray and binary. Subsequent rows in a particular column show the transition values of the counter when incremented on clock.

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

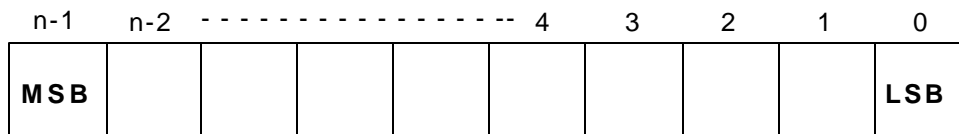
The equation for the Gray to Binary conversion is

$$\text{bin}_{n-1} = \text{gray}_{n-1} \quad \text{eq.(1)}$$

$$\text{bin}_i = \text{gray}_i \oplus \text{bin}_{i+1} \text{ where } i < n-1 \quad \text{eq.(2)}$$

in an n bit counter value.

Figure below shows the bit numbering of the counter.



Let us take a simple example of converting gray value "1010" into its binary equivalent.

Here above $n-1 = 3$

Substituting the value of $i = 3$ in the eq.(1) above we have

$$\text{bin}_3 = \text{gray}_3 = \text{gray}[3] = 1$$

Substituting the value of $i = 2$ in the eq.(2) above we have

$$\text{bin}_2 = \text{gray}_2 \oplus \text{bin}_3 = \text{gray}_2 \oplus \text{gray}_3 = \text{gray}[2] \oplus \text{gray}[3] = 1$$

Substituting the value of $i = 1$ in the eq.(2) above we have

$$\begin{aligned} \text{bin}_1 &= \text{gray}_1 \oplus \text{bin}_2 = \text{gray}_1 \oplus \text{gray}_2 \oplus \text{gray}_3 \\ &= \text{gray}[1] \oplus \text{gray}[2] \oplus \text{gray}[3] = 0 \end{aligned}$$

Substituting the value of $i = 0$ in the eq.(2) above we have

$$\begin{aligned} \text{bin}_0 &= \text{gray}_0 \oplus \text{bin}_1 = \text{gray}_0 \oplus \text{gray}_1 \oplus \text{gray}_2 \oplus \text{gray}_3 \\ &= \text{gray}[0] \oplus \text{gray}[1] \oplus \text{gray}[2] \oplus \text{gray}[3] = 0 \end{aligned}$$

so we have the following four equations:

$$\text{bin}[0] = \text{gray}[0] \oplus \text{gray}[1] \oplus \text{gray}[2] \oplus \text{gray}[3] \quad \text{eq.(3)}$$

$$\text{bin}[1] = \text{gray}[1] \oplus \text{gray}[2] \oplus \text{gray}[3] \quad \text{eq.(4)}$$

$$\text{bin}[2] = \text{gray}[2] \oplus \text{gray}[3] \quad \text{eq.(5)}$$

$$\text{bin}[3] = \text{gray}[3] \quad \text{eq.(6)}$$

Finally we have "1100" as the binary value of the given gray value "1010".

So from above equations its clear that bin[3] can be generated by right shifting gray value by 3, bin[2] by right shifting gray value by 2, bin[1] by right shifting gray value by 1 and bin[0] by right shifting gray value by 0.

Below is the verilog code of the above gray to binary converter

```
module gray_to_bin (bin , gray);

parameter SIZE = 4;
input  [SIZE - 1:0] bin;
output [SIZE - 1:0] gray;
reg    [SIZE - 1:0] bin;
integer i;
always @ (gray)
    for ( i = 0; i <= SIZE; i = i + 1)
        bin[i] = ^(gray >> i);
endmodule
```

Binary to Gray converter

Following are the equations for Binary to Gray conversion: -

$$\text{gray}_{n-1} = \text{bin}_{n-1} \quad \text{eq.(1)}$$

$$\text{gray}_i = \text{bin}_i \oplus \text{bin}_{i+1} \text{ where } i < n - 1 \quad \text{eq.(2)}$$

Let us take a simple example of converting binary value "1100" back into its gray equivalent.

Here above $n-1 = 3$

Substituting the value of $i = 3$ in the eq.(1) above we have

$$\text{gray}_3 = \text{bin}_3 = \text{bin}[3] = 1$$

Substituting the value of $i = 2$ in the eq.(2) above we have

$$\text{gray}_2 = \text{bin}_2 \oplus \text{bin}_3 = \text{bin}[2] \oplus \text{bin}[3] = 0$$

Substituting the value of $i = 1$ in the eq.(2) above we have

$$\text{gray}_1 = \text{bin}_1 \oplus \text{bin}_2 = \text{bin}[1] \oplus \text{bin}[2] = 1$$

Substituting the value of $i = 0$ in the eq.(2) above we have

$$\text{gray}_0 = \text{bin}_0 \oplus \text{bin}_1 = \text{bin}[0] \oplus \text{bin}[1] = 0$$

Finally we got back the same gray value "1010" of the given binary value "1100".

From the above, we have the following four equations:

$$\text{gray}[0] = \text{bin}[0] \oplus \text{bin}[1] \quad \text{eq.(3)}$$

$$\text{gray}[1] = \text{bin}[1] \oplus \text{bin}[2] \quad \text{eq.(4)}$$

$$\text{gray}[2] = \text{bin}[2] \oplus \text{bin}[3] \quad \text{eq.(5)}$$

$$\text{gray}[3] = \text{bin}[3] \quad \text{eq.(6)}$$

As inferred from eq.(3), eq.(4), eq.(5) and eq.(6), we can obtain the equivalent gray value by performing bit wise exclusive or operation between the binary value and its right shift version as shown below:-

bin[3]	bin[2]	bin[1]	bin[0]	→binary value: <i>bin</i>
0	bin[3]	bin[2]	bin[1]	→right shift (<i>bin</i>)

gray[3]	gray[2]	gray[1]	gray[0]	→equivalent gray value

Below is the Verilog code of the above binary to gray converter

```
module bin_to_gray (bin, gray);
  parameter SIZE = 4;
  input  [SIZE-1:0] bin;
  output [SIZE-1:0] gray;

  assign gray = (bin >> 1) ^ bin;

endmodule
```

Gray code counter implementation

It is a combination of all the four steps shown in the [Figure 25](#) (gray to binary converter, adder, binary to gray converter and finally sets of register elements to store the gray value).

Below is the Verilog code of the Gray code counter: -

```
module gray_counter (clk, gray, inr, reset_n)
  parameter SIZE = 4;
  input      clk, inr, reset_n;
  output [SIZE - 1] gray;
  reg      [SIZE - 1] gray_temp, gray, bin_temp, bin;
  integer i;
```

```

always @(gray or inr)
begin:gray_bin_gray
  for (i = 0; i<SIZE ; i = i +1)
    bin[i] = ^ (gray >> i);           // gray to binary conversion
    bin_temp = bin  + inr;             // addition in binary
    gray_temp = (bin_temp >> 1) ^ bin_temp; // binary to gray conversion
  end

```

The always block below registers the converted gray value.

```

always @(posedge clk or negedge reset_n)
begin:gray_registered
  if (~reset_n)
    gray <= {SIZE{1'b0}};
  else
    gray <= gray_temp;
  end

```

Shown below is the logical diagram for the above gray counter code:-

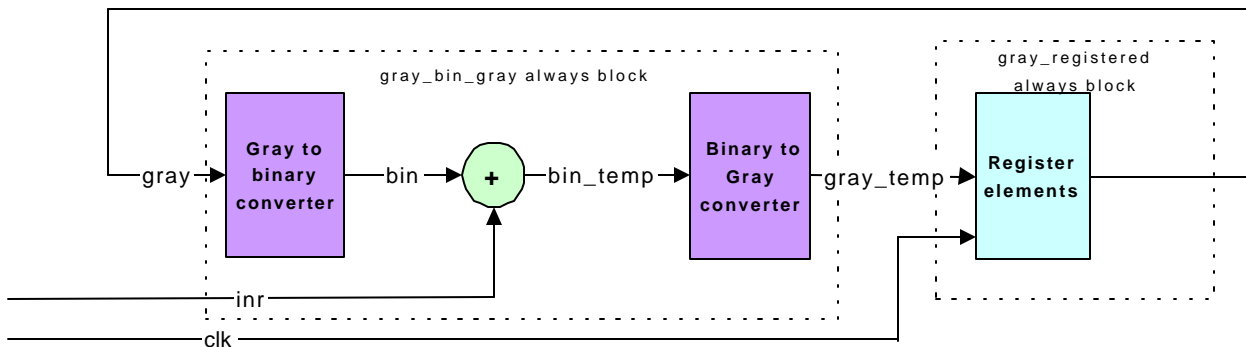


Figure 25: Gray Counter Hardware

FIFO Full and FIFO Empty generation

We need an N bit pointer to address 2^N locations in a FIFO. Now the FIFO is empty when both pointers are equal and FIFO is full also when both the pointers are equal, so we need an extra bit to keep track whether the FIFO is full or empty when both the pointers are equal.

The FIFO is full when the most significant bits of the binary versions of the pointers differ and the remaining N bits are equal.

The FIFO is empty when the binary versions of the pointers are exactly equal in all bit positions. Let us see how it works with an example: -

Consider an 8 deep FIFO. Since we need just 3 bits pointer to address all its 8 locations but we take an extra bit to distinguish between FIFO full and FIFO empty condition. Initially both rd_ptr_bin and wr_ptr_bin are "0000" and the FIFO is empty. Now after 8 subsequent writes to FIFO we have the following values of read and write pointer: -

```
rd_ptr_bin  = "0000"
wr_ptr_bin  = "1000"
```

Which is the FIFO full condition as shown in the [Figure 26](#).

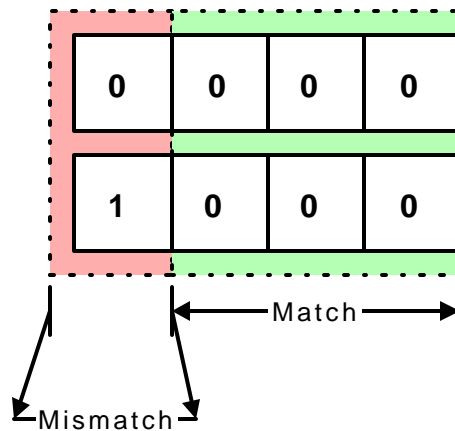


Figure 26: FIFO full condition

Now suppose the user performs subsequent eight reads, we have the following values of the read and write pointer: -

```
rd_ptr_bin  = "1000"
wr_ptr_bin  = "1000"
```

Which is the FIFO empty condition as shown in the [Figure 27](#).

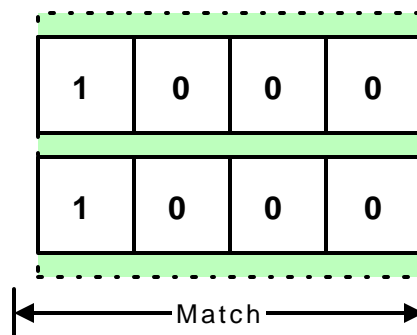


Figure 27: FIFO empty condition

Below shows the block diagram showing FIFO empty and FIFO full generation:-

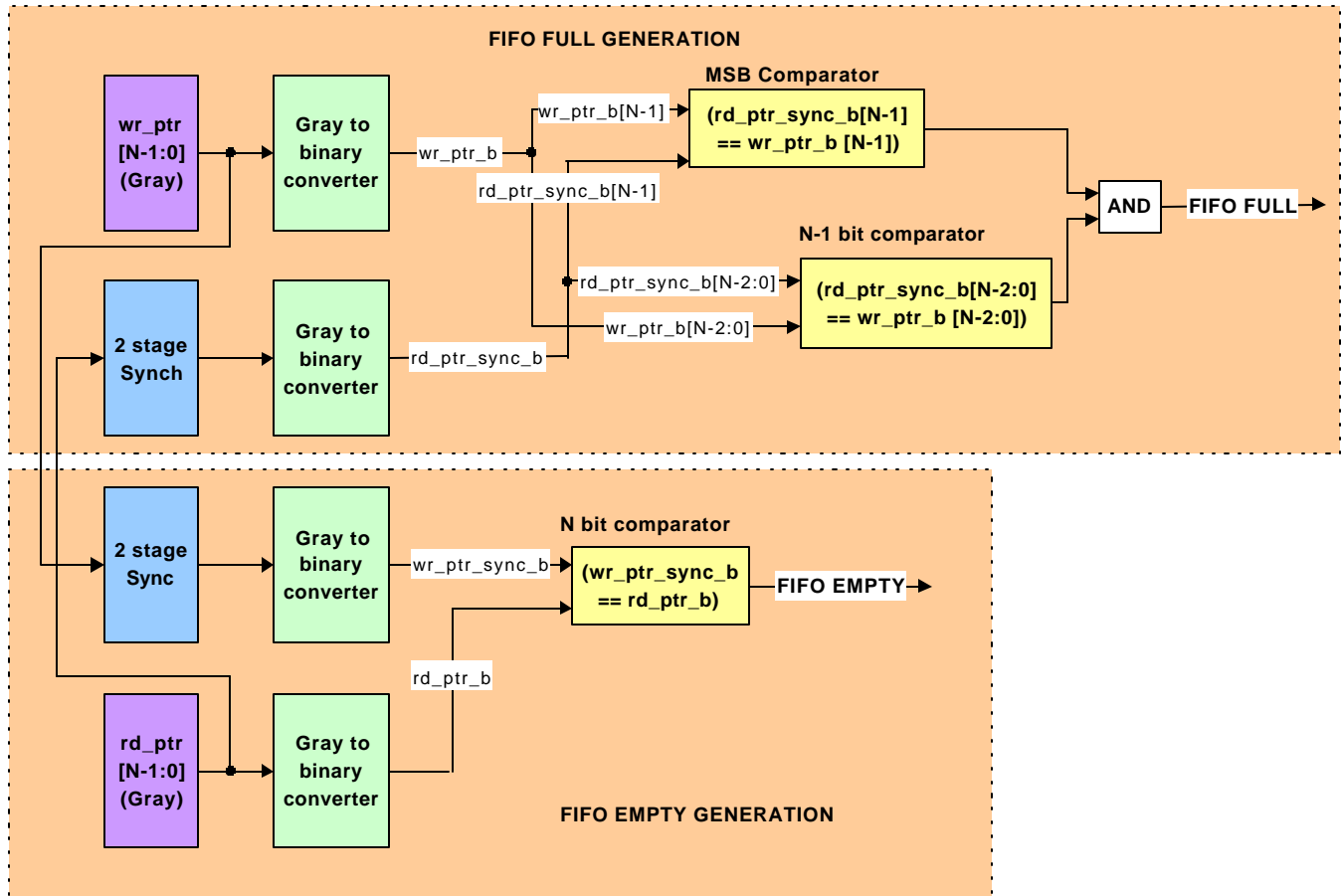


Figure 28: FIFO full & empty signals generation hardware

In this case, your maximum frequency of operation will depend on how fast your gray code counters work since it requires a chain of XOR gates.

Here in above pointer's value is stored in gray and all the comparisons, incrementing of pointers etc. is done in binary that makes the implementation and debugging quite simpler. As shown in the figure above, it requires four gray to binary converters, which can be avoided if the comparison etc. for calculation of FIFO empty and FIFO full generation are done directly in gray. This is somewhat complicated and it requires some additional logic. Let's see how this alternative approach works in our next section.

An Alternative approach for FIFO full and FIFO empty generation

This approach requires creating two gray code counters, one of n bit and the other of $n-1$ bit. The two counters can be created by a single n bit counter and then modifying its 2^{nd} MSB to generate $(n-1)$ bit gray code counter with the same LSBs as of the n bit counter.

Before we start up with the main logic lets look up some more about Gray code counters.

Lets look below at the following 4-bit Gray code counter

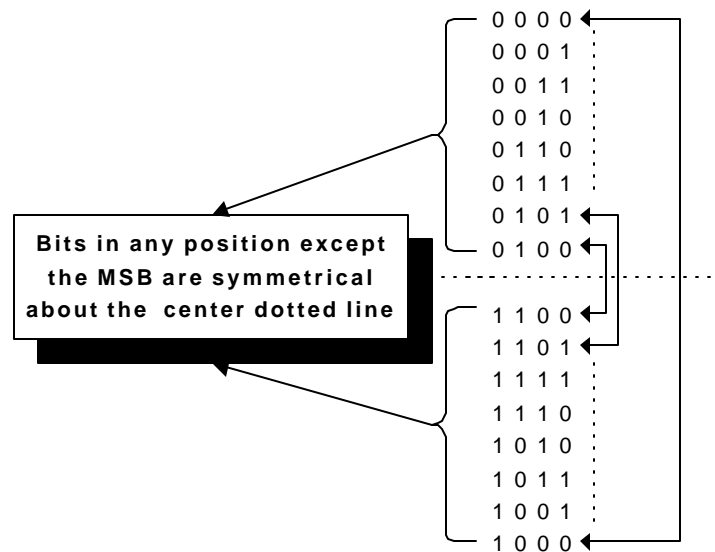


Figure 29: 4-bit Gray Counter

As shown above in the figure, bits in any column except the MSB are symmetrical about the sequence mid-point. Thus the second half of the 4-bit Gray code is a mirror image of the first half with the MSB inverted.

Now $(n-1)$ bit Gray code can easily be generated by XORing the two MSBs of the n -bit Gray code to generate the MSB for the $(n-1)$ bit gray code. Rest of the $(n-2)$ bits can be simply using the $(n-2)$ bits of the n -bit counter. [Figure 30](#) below shows the conversion of 4-bit to 3-bit Gray code.

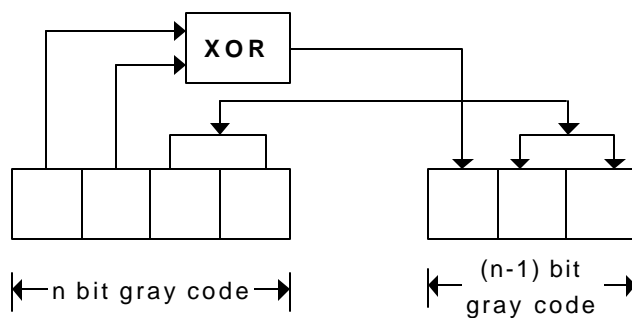


Figure 30: Conversion of 4-bit Gray code to 3-bit Gray code

4 bit Gray Code	3 bit Converted Gray Code
0000	000
0001	001
0011	011
0010	010
0110	110
0111	111
0101	101
0100	100
1100	000
1101	001
1111	011
1110	010
1010	110
1011	111
1001	101
1000	100

Now let's come back to the FIFO design. We will use this dual n -bit Gray code counter later when we will come to FIFO empty and FIFO full generation logic.

FIFO Design

[Figure 31](#) below shows the block diagram for the FIFO. Dual port Memory is used as storage elements inside a FIFO.

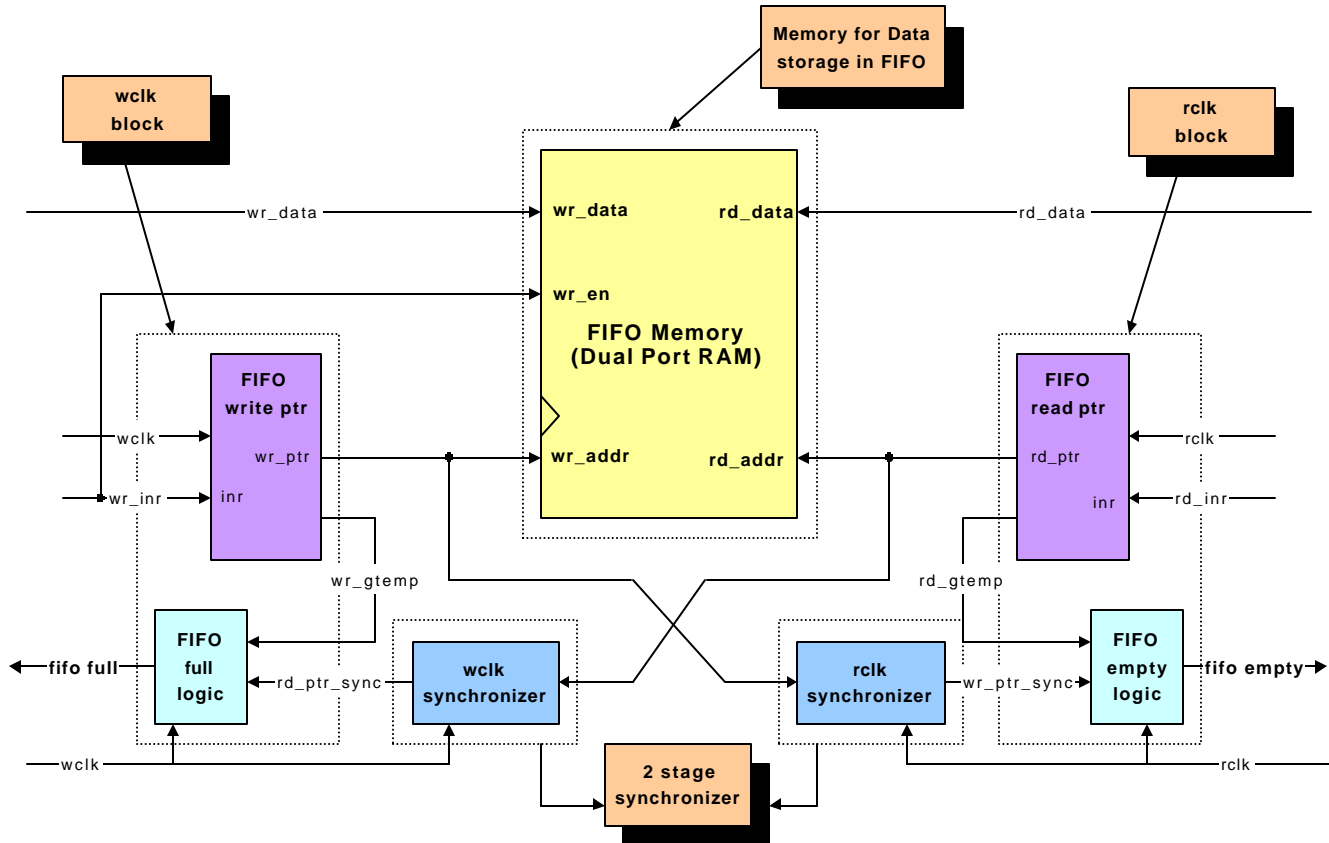


Figure 31: Dual Clock FIFO Design

FIFO empty condition generation

FIFO empty flag would be generated in the read clock immediately when the FIFO becomes empty that is when the read pointer matches up with the synchronized write pointer.

Here both the read pointer and the synchronized write pointer are directly compared in gray and are not converted to their binary equivalent before the comparison unlike what we had in our previous implementation. This saves our four gray to binary converters as previously shown in [Figure 28](#).

Here again the pointers are one bit larger than needed to address the FIFO memory. The synchronized write pointer (*wr_ptr_sync*) is compared against the *rd_gtemp* (the next gray code that will be registered in the *rd_ptr*).

Note: FIFO empty output generated is registered.

Below is the Verilog code for the above logic.

```
always @ (posedge rclk or negedge reset_n)
begin: fifo_empty_gen
  if (~reset_n)
    fifo_empty <= 1'b1;
  else
    fifo_empty <= (rd_gtemp == wr_ptr_sync);
end
```

FIFO full condition generation

FIFO full flag would be generated in the write clock immediately when the FIFO becomes full. Again both the write and synchronized read pointers are compared but this comparison is not that simple as for FIFO empty flag generation.

Here again we have to take pointer with one extra bit than required to address the FIFO memory. We cannot use the same logic that we used in our previous implementation since here we have our comparisons done in gray instead of binary. Lets see by taking an example.

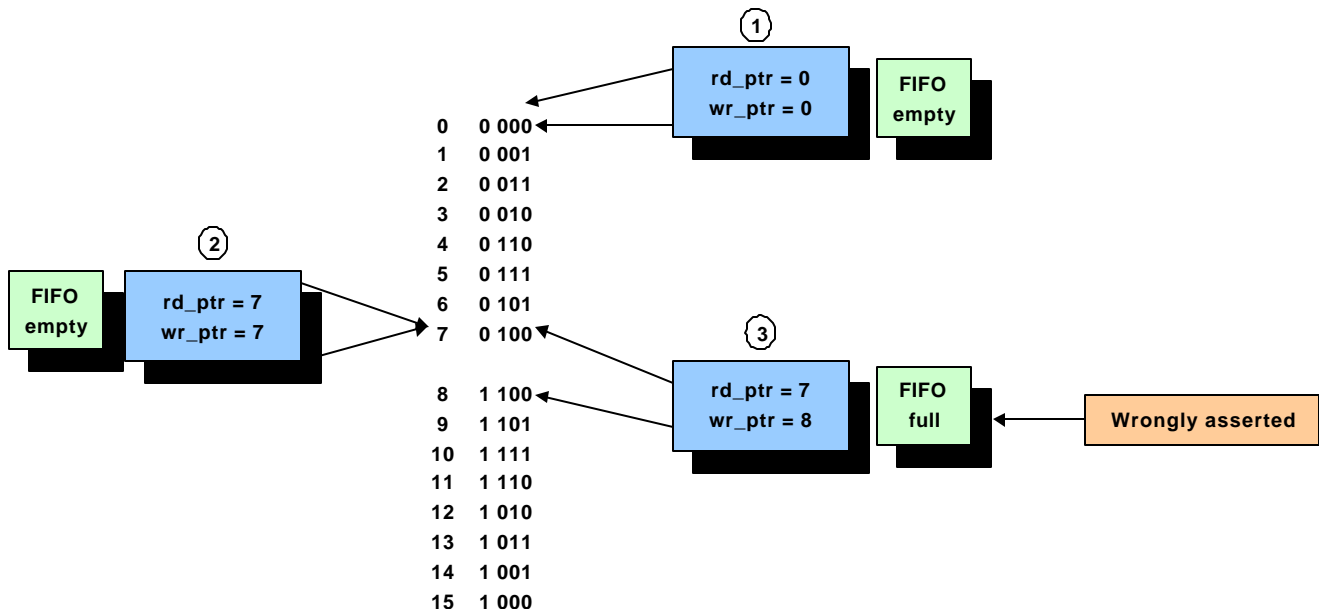


Figure 32: FIFO full & FIFO empty conditions

Figure above shows the following steps performed on a 8 depth FIFO.

STEP 1: Initially FIFO is empty with $rd\ ptr = wr\ ptr = 0$ shown as 1. in the figure above.

STEP 2: Subsequent writes takes places on FIFO till the FIFO becomes full with $rd\ ptr = 0$ and $wr\ ptr = 7$. Now subsequent 8 reads takes place with finally $rd\ ptr = wr\ ptr = 7$ and FIFO becomes empty (since all the bits of read and write pointer are equal) as shown in figure above.

STEP 3: Now with a single write $rd\ ptr = 7$ and write $ptr = 8$ and using the same logic as we used for our previous binary implementation, FIFO again becomes full which is not true.

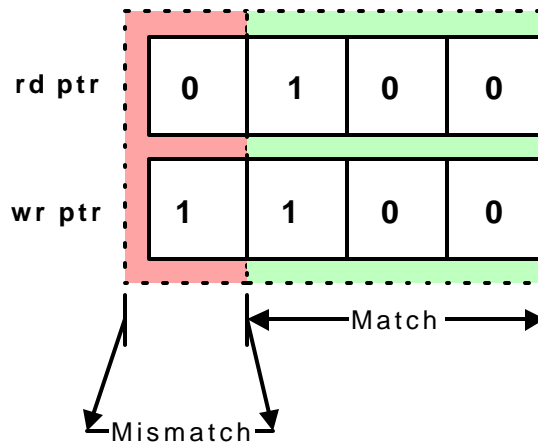


Figure 33: FIFO full condition

This condition can be easily take care by using dual n-bit gray code counter.

The correct method to perform the full comparison is accomplished by synchronizing the $rd\ ptr$ into the write clock domain. The MSBs are compared and should differ in case the write pointer has wrapped one more time than the synchronized read pointer. Then if the synchronized read pointer's MSB is high, the 2nd MSB of the synchronized read pointer (rd_ptr_sync) is inverted before doing a comparison against a (n-1) bit write pointer.

So the FIFO full flag is asserted when all the 3 condition below are true: -

(1) MSB of the synchronized read pointer (rd_ptr_sync) should differ from the MSB of the next gray code value of the write pointer (wr_gtemp) that will be registered in the wr_ptr .

(2) 2nd MSB of the next gray code count in the write clock domain (wr_gtemp) should be equal to 2nd MSB of the read pointer that has been synchronized into the write clock domain (rd_ptr_sync).

(3) All the left out LSB's of the two pointers should match.

Note: The 2nd MSB in (2) above is calculated by XORing the first two MSBs of the pointer. (Doing an exclusive-or operation of the two MSBs causes the 2nd MSB to be inverted if the MSB is high)

Below is the Verilog code for the above logic.

```

wire rd_2nd_msb = rd_ptr_sync[SIZE] ^ rd_ptr_sync[SIZE - 1];
wire wr_2nd_msb = wr_gtemp[SIZE] ^ wr_gtemp[SIZE - 1];

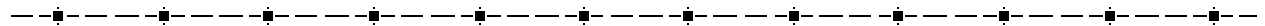
always @ (posedge wclk or negedge reset_n)
begin: fifo_full_gen
    if (~reset_n)
        fifo_full <= 1'b0;
    else
        fifo_full <= ((wr_gtemp[SIZE] != rd_ptr_sync[SIZE]) &&
            (rd_2nd_msb == wr_2nd_msb) &&
            (wr_gtemp[SIZE-2:0] == rd_ptr_sync[SIZE-2:0]));
    end
end

```

8. Conclusions

Summarizing the contents, it is seen that multiple clock designs always pose problems of metastability, setup & hold time violation. Using proper design guidelines and partitioning designs is the first step towards handling multiple clock designs effectively. Transfer of control signals requires synchronization of the signal coming from different clock domain. Data transfer can be done via handshaking signal or through use of FIFOs. Using FIFOs is preferred as it reduces the latency of data transfer. Using an async FIFO with gray code read & write pointers is the safe and efficient way of data transfer.

Multi-clock designs have always been a major hurdle to cross by designers. Using the techniques presented in this paper, a designer will be able to handle such designs with ease.



9. Acknowledgements

We would like to thank one and all who helped in the successful writing of this paper. We are indeed grateful to Mr. Amit Srivastava for helping us in reviewing the document and providing valuable feedback in a short time.

10. References

- [1]. 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.
- [2]. Vijay A. Nebhrajani, "Asynchronous FIFO Architectures"
http://www.parmita.com/chipguru/issue3/files/async_fifo2_corrected.pdf
- [3]. Samir Palnitkar, Verilog HDL, A Guide to Digital Design and Synthesis, Sunsoft Press A Prentice Hall.
- [4]. "Digital Design" by Morris M. Mano
- [5]. "Metastability in Altera Devices" by Altera Corp., USA

11. Author & Contact information

For any further information please contact:

Mohit Arora (mohit.arora@dcmtech.co.in)

Design Engineer

Prashant Bhargava (prashant.bhargava@dcmtech.co.in)

Design Engineer

Shivraj Gupta (shivraj.gupta@dcmtech.co.in)

Project Manager