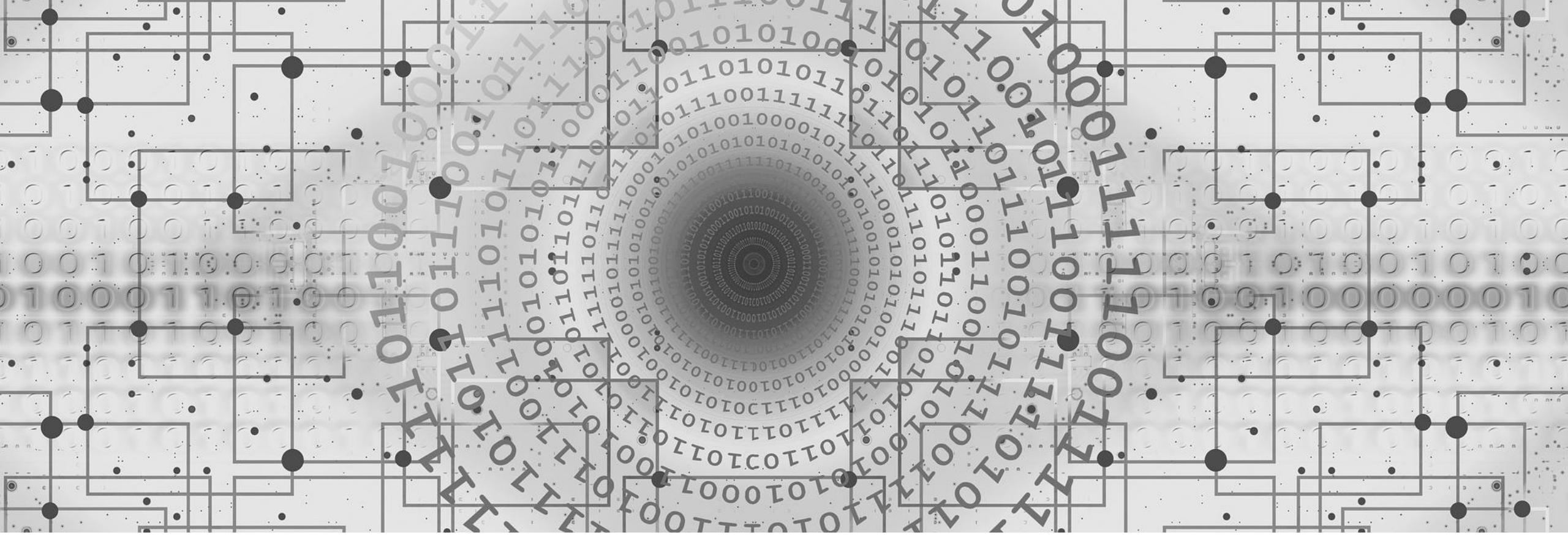


Lecture-15 & 16 : Timing and Synchronization

ECE-111

Vishal Karna

Winter 2022

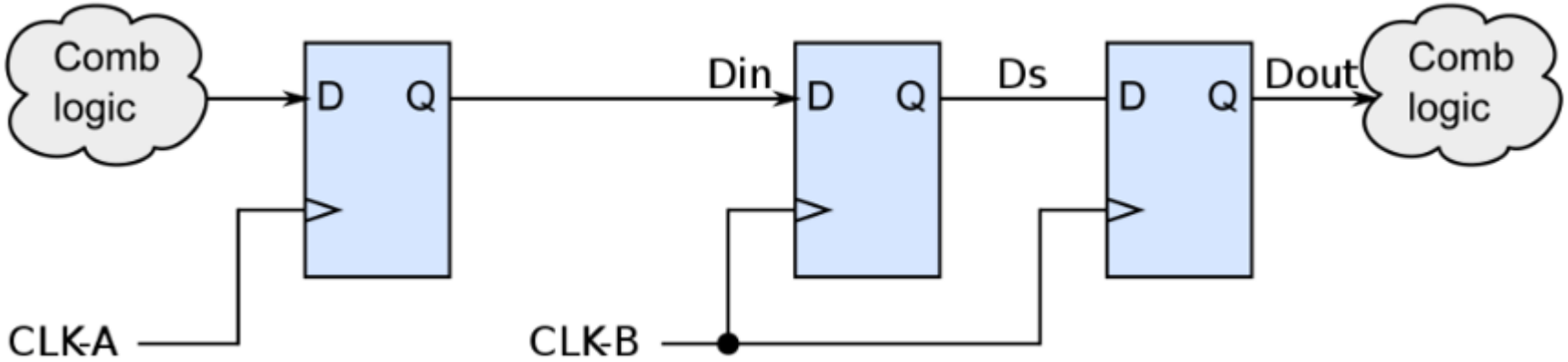
UC San Diego
JACOBS SCHOOL OF ENGINEERING
Electrical and Computer Engineering

Timing and Synchronization

Timing

- **What limits speed of a circuit ?**
  - Speed limited by the time data takes to get from one register (Flip Flop) to another
  - State machines and pipeline data stages

- **What is the bottleneck ?**
  - Combinational logic delay
  - Routing delay
  - Clock skew and delay
  - FlipFlop (FF) setup and hold time requirements



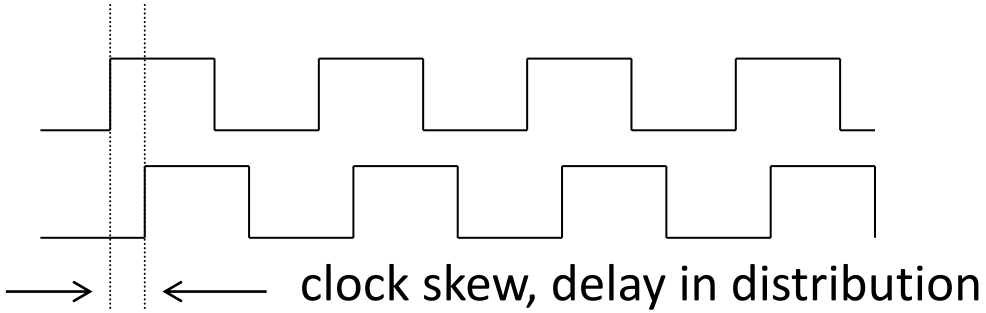
Timing

- **Combinational Delay**
  - Time taken by signal to propagate through gates
  - More gates, then higher combinational delay which will result in slower performance
  - FPGA doesn’t have “gates” instead it has N-input LUTs (look-up-tables)
  - All functions of 4 inputs or less are the same speed however with n >4 inputs there are more CLB’s hook up increasing routing delay

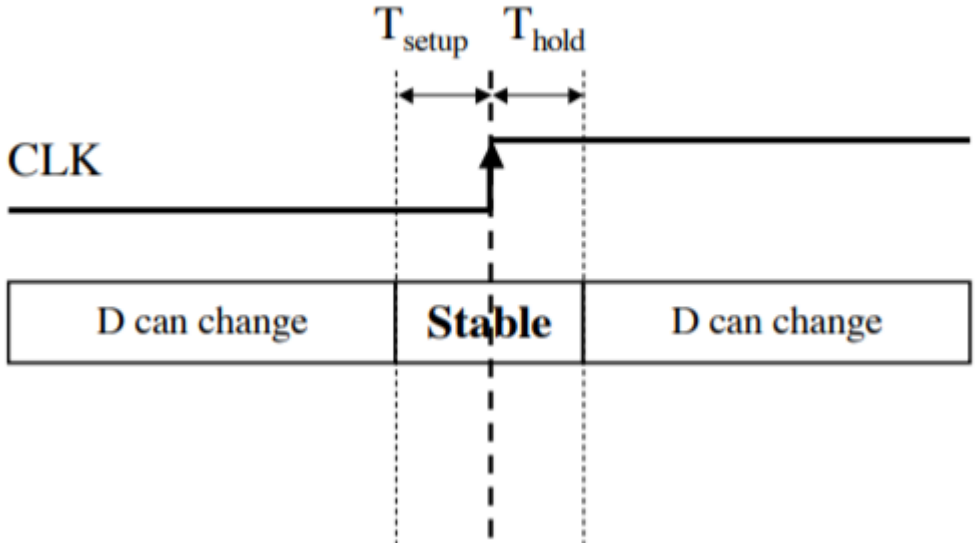
- **Routing Delay**
  - FPGA wiring slow compared to ASIC
  - Lots of switches to go through
  - Wires don’t go exactly where you want
  - Routing can be ~50% of total delay (requires Manual placement to reduce routing delay)

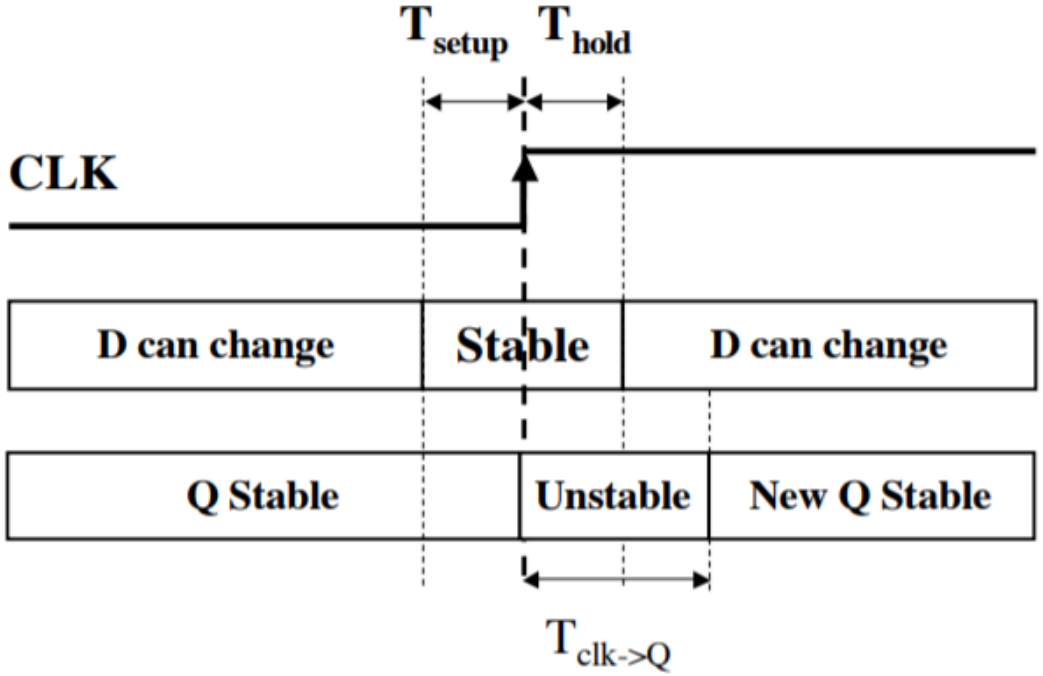
Clock Skews

- **Static variation in time of arrival of the clock edge at the different FlipFlops**
  - **If not accounted for, can lead to erroneous behavior.**
  - **Reason for clock skews :**
    - clock wires have delay
    - circuit is designed with a different number of clock buffers from the clock source to the various clock loads, or
    - buffers have unequal delay
  - **All synchronous circuits experience some clock skew:**
    - more of an issue for high-performance designs operating with very little extra time per clock cycle.
  - **To control clock skew**
    - Careful clock distribution. Equalize path delay from clock source to all clock loads by controlling wires delay and buffer delay.
    - Don't "gate" clocks.
  - **FPGA Structure:** very regular, "easy" distribution
  - **ASIC: Random logic,** different skews is common



FlipFlop Timing

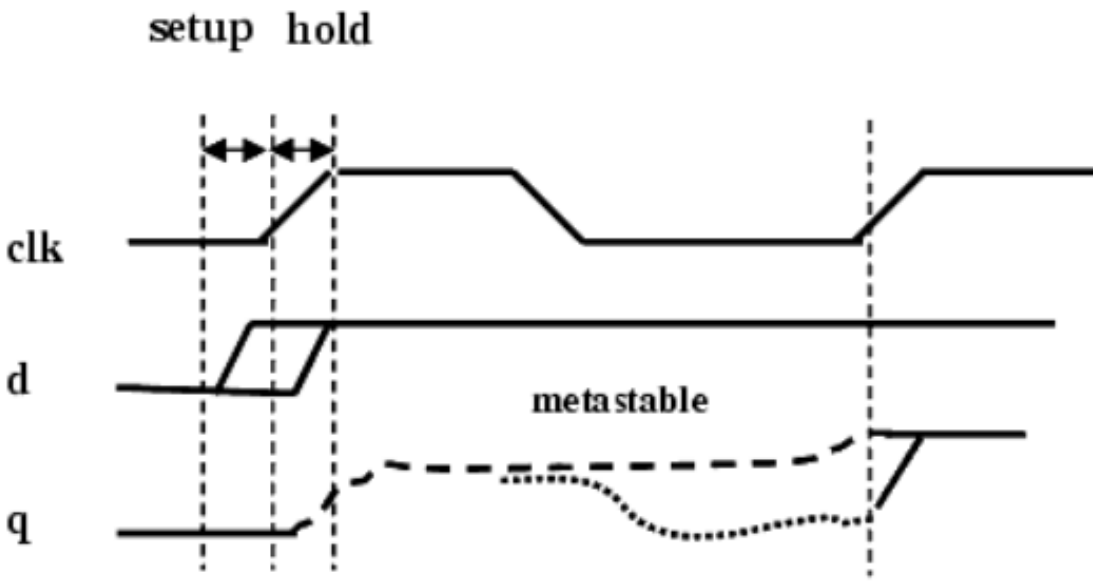


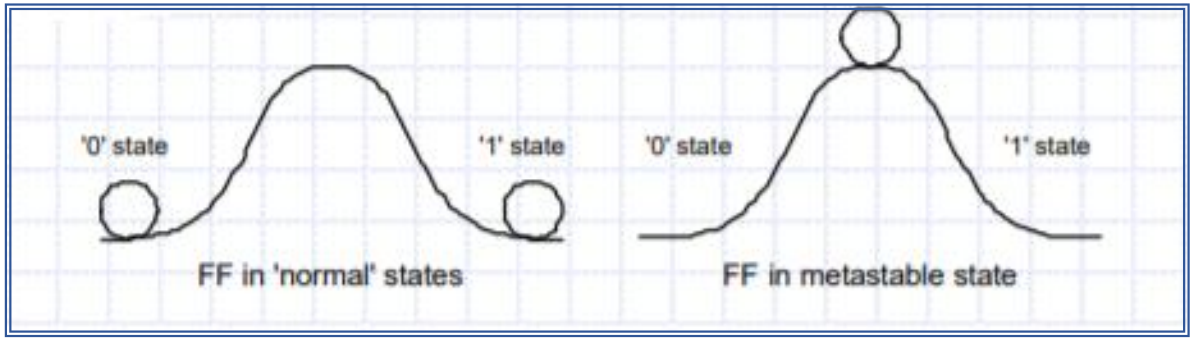


- **Setup Time :**
  - Minimum amount of time the synchronous input (D) must be stable before the active edge of the clock
- **Hold Time:**
  - Minimum amount of time the synchronous input (D) must be stable after the active edge of the clock
- **Note :** If Setup or Hold time is violated, the FF output will be **metastable** (**neither 1 or 0**)

- **Clock-to-Q (Tclk->Q)**
  - Time it takes for the Flipflop output to be in a stable state after a clock edge occurs
  - If output is sampled before Tclk->Q delay then Q is not guaranteed to be correct value

FlipFlop Metastability

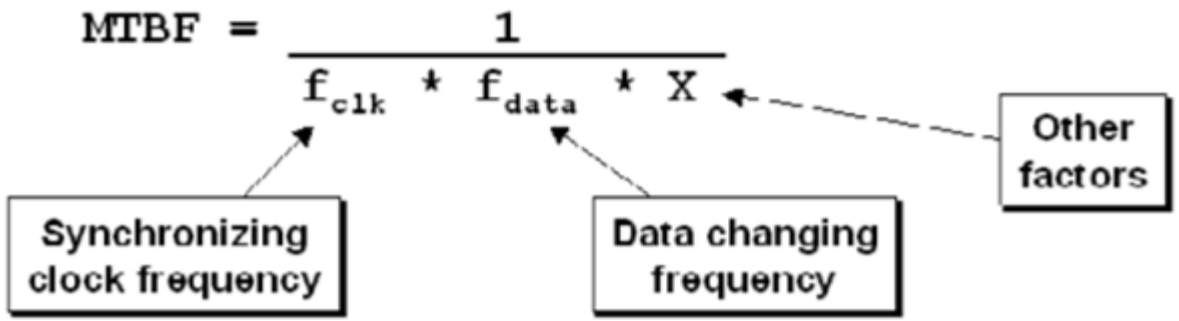




- Every flip-flop (FF) used in any design has a specified setup and hold time, or the time in which the data input is not legally permitted to change before and after a sampling clock edge.
- If some input (say d in Figure above) violates the setup and hold time of a FF, the output of the FF (q in Figure above) keeps oscillating for an indefinite amount of time.
- This unstable value may or may not non-deterministically converge to a stable value (either 0 or 1) before the next sampling clock edge arrive. Flip-Flop enters state metastability
- Meta-stability is a probabilistic phenomenon. **It cannot be completed eliminated !**

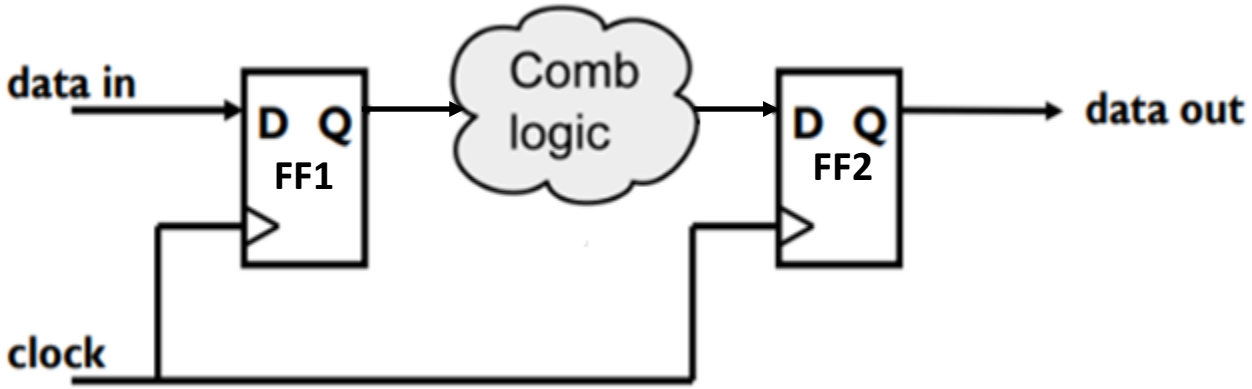
MTBF

- **MTBF** is Mean time between failure
- **MTBF** gives us information on how often a particular element will fail or in other words, it gives the average time interval between two successive failures.
- For a Flip Flop we can compute its **MTBF**, which is a figure of merit related to metastability
- **MTBF** of a flip-flop is shown below :
  - It is a function of source clock frequency and frequency of incoming data (data rate)



FlipFlop Timing

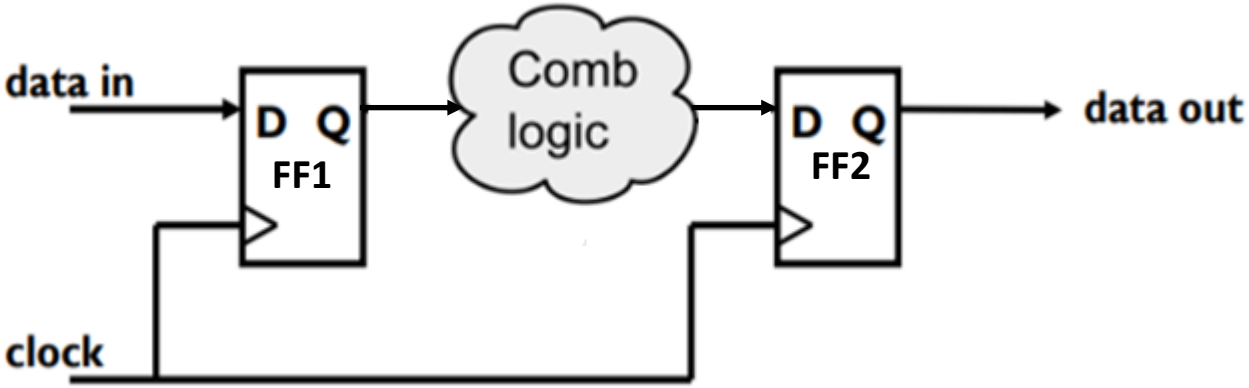
- Sample data using clock
- Hold data between clock cycles
- Computation (and delay) occurs between registers



- **Max Frequency of operation of above mentioned circuit is :** Fmax = 1 / T
  - $T >= T_{clk \rightarrow Q} + T_{CL\_max} + T_{setup}$ (Circuit will fail, need to consider clock skew, T_{skew})
  - $T >= T_{clk \rightarrow Q} + T_{CL\_max} + T_{setup} + T_{skew}$
  - Note: T_{setup} is for FF2, $T_{clk \rightarrow Q}$ is for FF1 and T_{CL_max} is max propagation delay of combinational logic

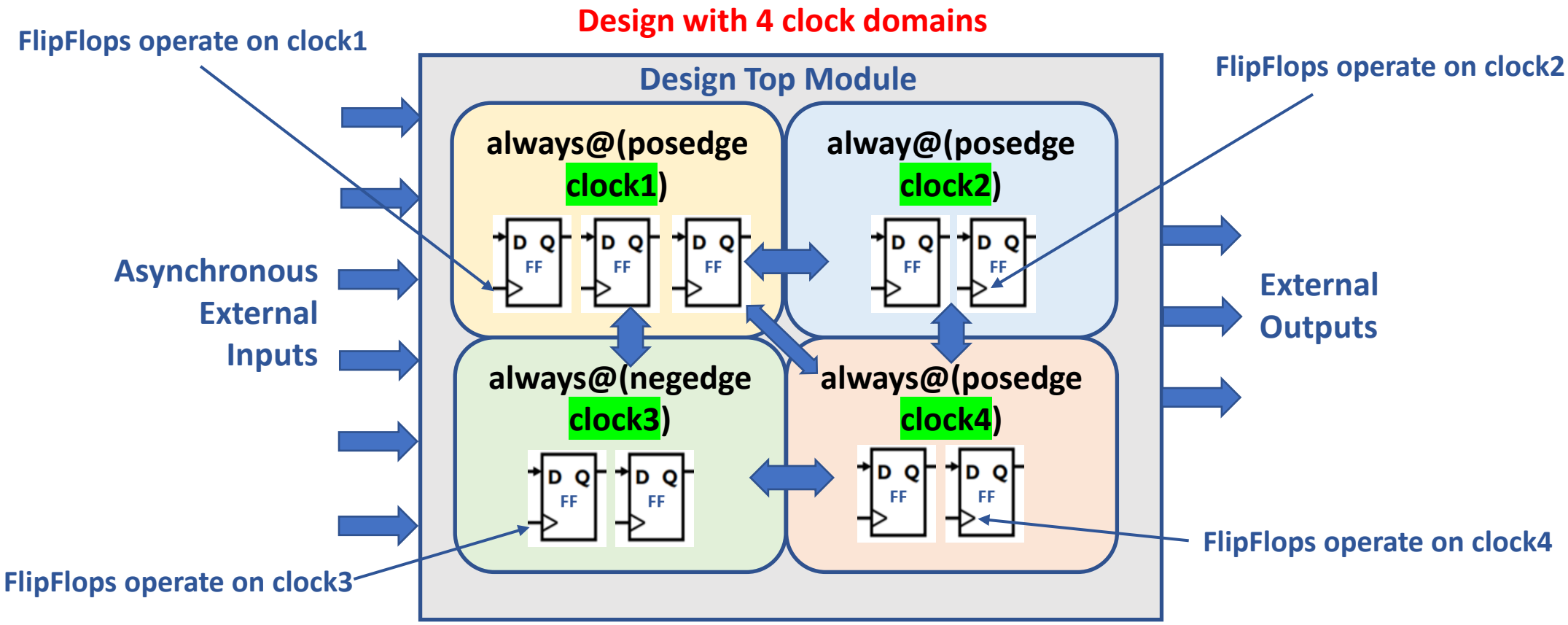
- **DFF values**: $T_{clk \rightarrow Q}$ =1ns, T_{setup} =1ns, T_{hold} =1ns, T_{skew} =max 2ns, T_{CL_max} =10ns, T_{CL_min} =1ns
  - 1 + 10 + 1 + 2 <= T (Clock Period) >= 1 + 10 + 1 + 2 = **14ns**, Fmax = **71Mhz** -**This is the max path timing constraint**
  - **Longest delay path is called as critical path**
  - MaxPath timing constraint violations can be addressed by slowing down the clock after the circuit is implemented

FlipFlop Timing



- **Min path timing constraint :** Now, what happens when the same clock edge is considered at the far DFF.
  - $T_{clk \rightarrow Q} + T_{CL\_min} >= T_{skew} + T_{hold}$ (1 + 1 >= 2 + 1)
  - The new value from FF1 can get there so fast that when the clock arrives the new value may change before it has been latched in to FF2. **This is known as Hold-Time Violation**.
  - Min path timing constraint cannot be fixed by slowing down the clock

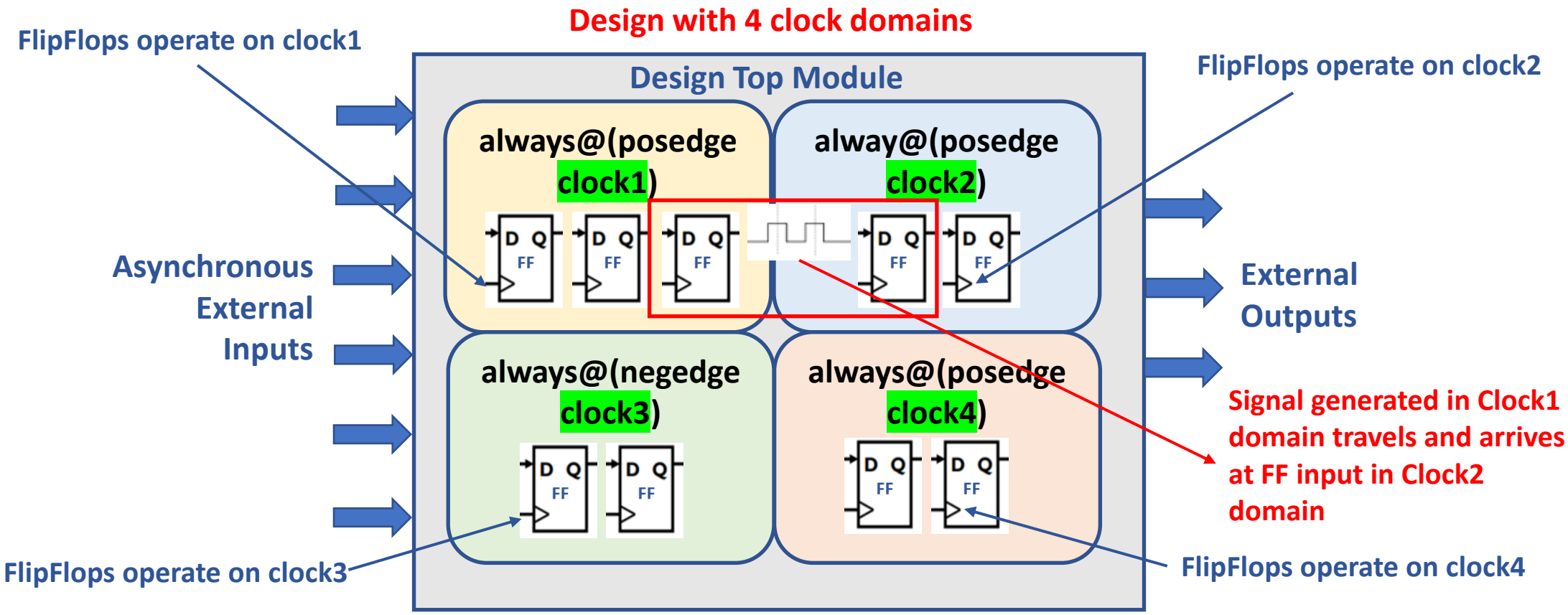
Clock Domain



❑ What is a Clock Domain ?

- **Clock Domain** is that portion of a circuit that is generated and processed by a single clock
- In diagram below, there are 4 clock domains in design top module
- Flipflops in each clock domain operate on a separate clock
- All 4 clocks can be of different frequency, out of phase and asynchronous to each other

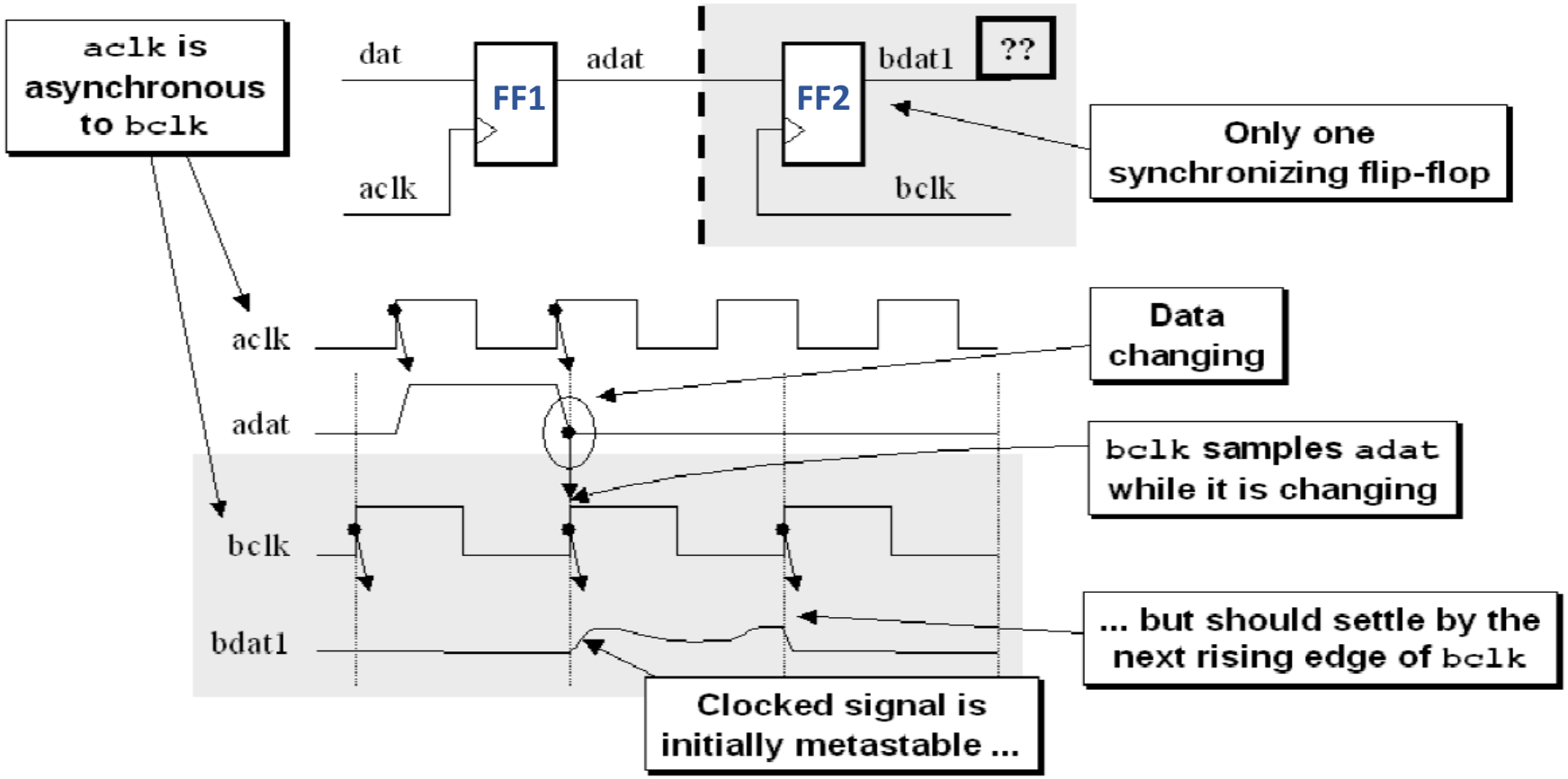
Clock Domain Crossing (CDC)



❑ What is Clock Domain Crossing (CDC) ?

- **Signal travels from one clock domain to another clock domain**
- **CDC** takes place anytime the inputs to a given flipflop were set based upon some other clock
- In example above signal generated from FF operating on clock1 travels and arrives at the flipflop inputs which is operating on clock2 → **Signal crossed clock domains**

What is the issue when signal crosses clock domains ?



- Synchronization failure that occurs when a signal (**adat**) generated in one clock domain (**aclk**) is sampled too close to the rising edge of a clock signal (**bclk**) from a second clock domain.
- **adat does not meet setup time requirement for FF2 hence output of FF2 (bdat1) goes metastable**
  - output (**bdat1**) going metastable and not converging to a legal stable state by the time the output must be sampled again.

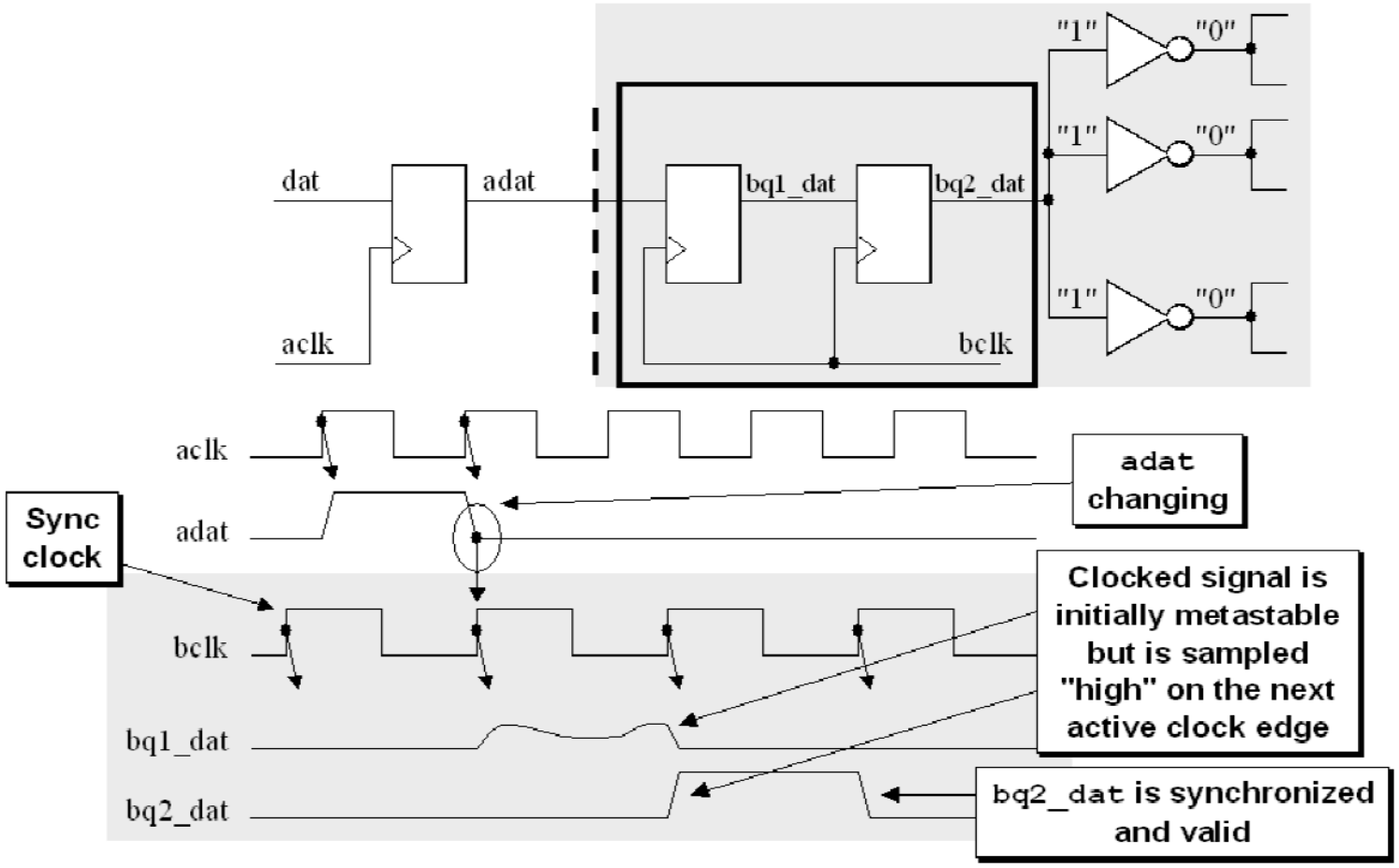
Synchronization Techniques

- **Open-loop and Close-loop Synchronization**

- **For Single-bit control signal synchronization :**
  - 2-FF or 3-FF Synchronizer (open-loop)

- **For Multi-bit data bus synchronization :**
  - Handshake Mechanism (closed loop)
  - Asynchronous FIFO (open loop)

2-FlipFlop Synchronizer to Address Metastability

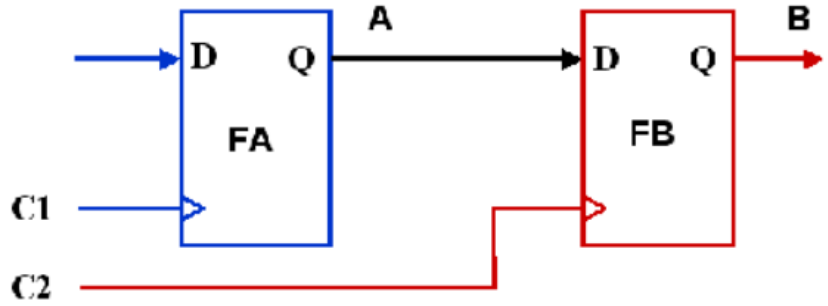


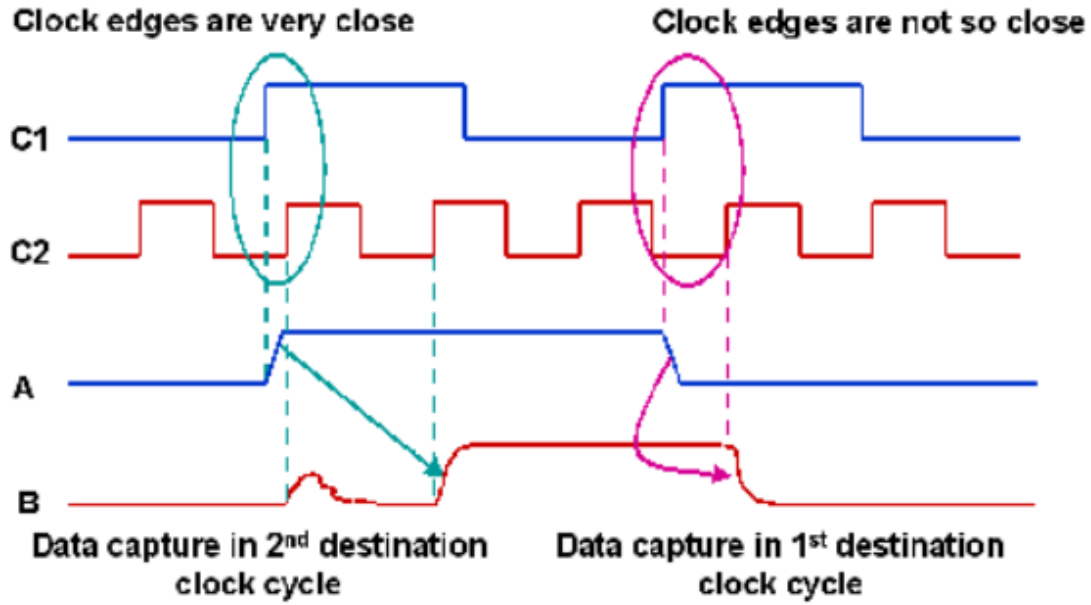
Note : Multi-Flop synchronizers allow sufficient time for the oscillations to settle down and ensure that a stable output is obtained in the destination domain

- First flipflop (**FF1**) samples the asynchronous input signal into the new clock domain (**bclk**)
- Waits for full clock cycle to permit any metastability on the synchronizer stage-1 output signal (**bq1_dat**) to decay, then the stage-1 signal is sampled by the same clock(**bclk**) into a second stage flipflop (**FF2**)
- The stage-2 signal (**bq2_dat**) is now a stable and valid signal synchronized and ready for distribution within the new clock domain

Signal Crossing Slow Clock Domain to Fast Clock Domain

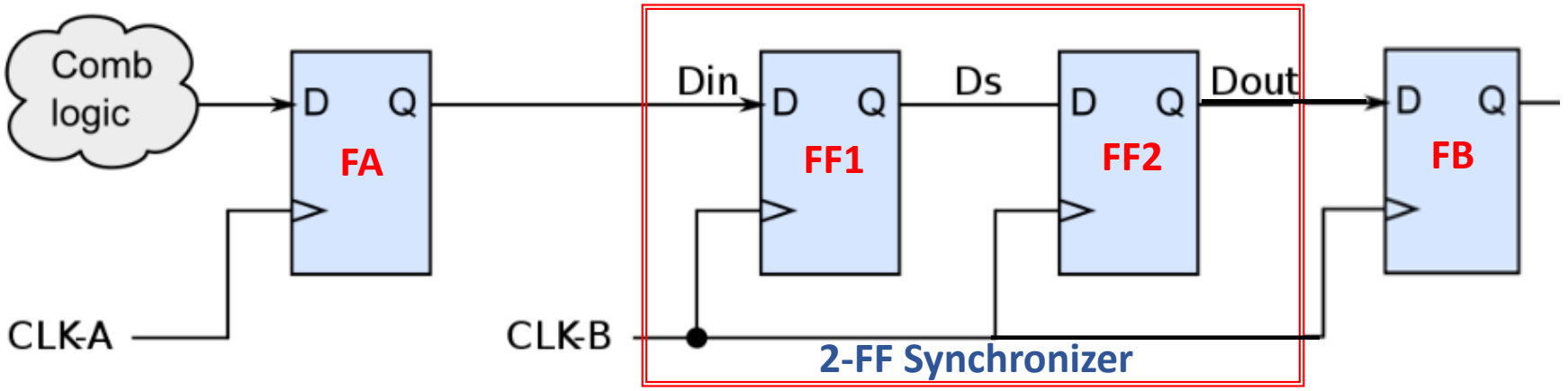
- **Scenario -1 : Flipflop-A clock frequency < Flipflop-B clock frequency**

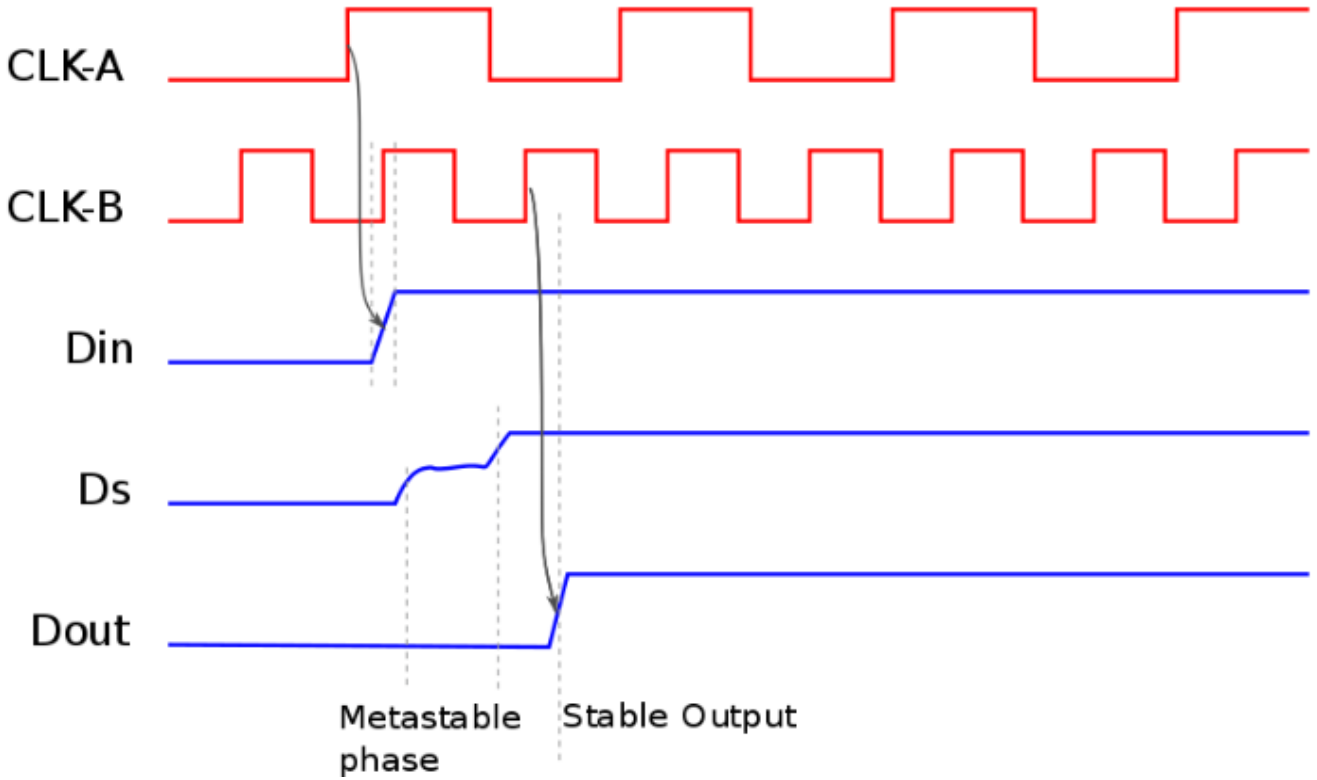




- If the transition on signal A happens very close to the active edge of clock C2, it could lead to setup or hold violation at the destination flop "FB“ and “FB” will enter metastable state.
- Output signal B from FA will be unstable and may or may not settle down to some stable value before the next clock edge of C2 arrives
- Unstable signal B fanout cones may read different values, and may cause the design to enter into an unknown functional state, leading to functional issues in the design
- **To address such slow to fast clock domain crossing scenario, two or three flipflop synchronizer can be used !**

2-FlipFlop Synchronizer For Slow to Fast Clock Domain

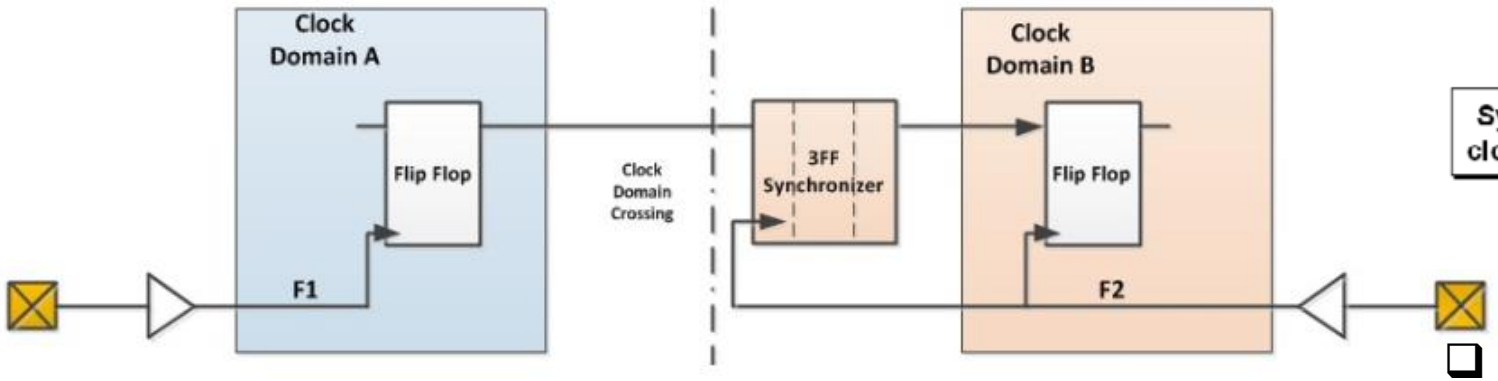




- CLK-A frequency is slower than CLK-B
- Din entering into CLK-B domain goes through 2-FF synchronizer stage (FF1, FF2)
- 2-FF synchronizer will operate on receiving clock which is CLK-B
- Even if Din changes close to CLK-B clock edge violating setup violation of FF1, causing metastable output Ds, output of second stage of synchronizer (FF2) will be stable.
- If Din changes fast and frequently, sometimes 2-FF synchronizer is not enough for metastable output to settle. In such cases 3rd FF can be added (i.e.3-FF synchronizer)

2-FlipFlop and 3-FlipFlop Synchronizer

3-FF Synchronizer



$$MTBF = \frac{1}{f_{clk} * f_{data} * X}$$

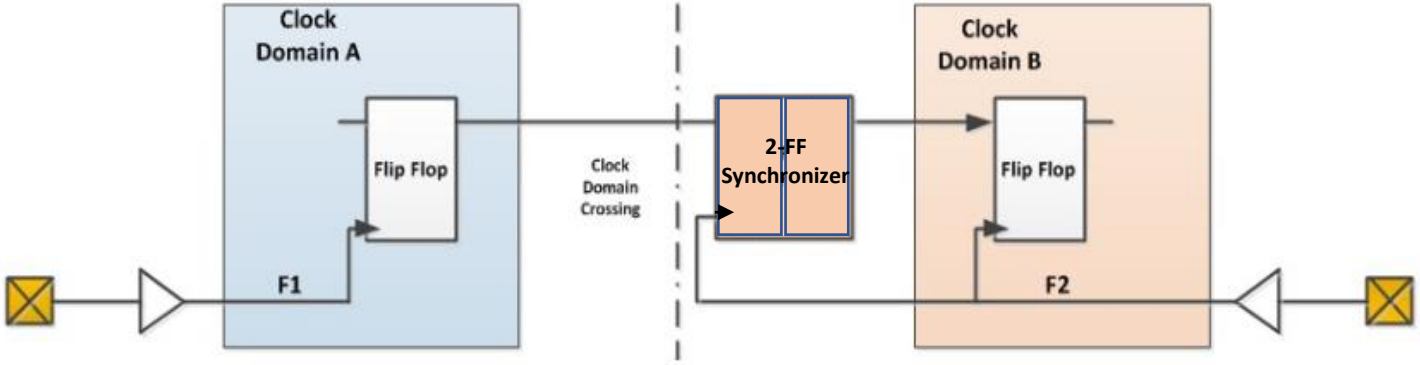
Other factors

Synchronizing clock frequency

Data changing frequency

Figure 4 - Primary contributing factors to short MTBF values

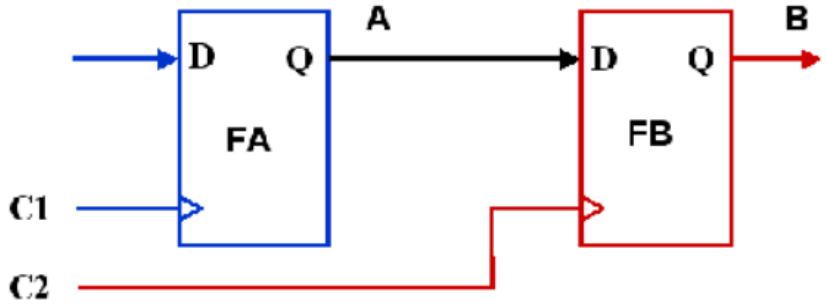
2-FF Synchronizer

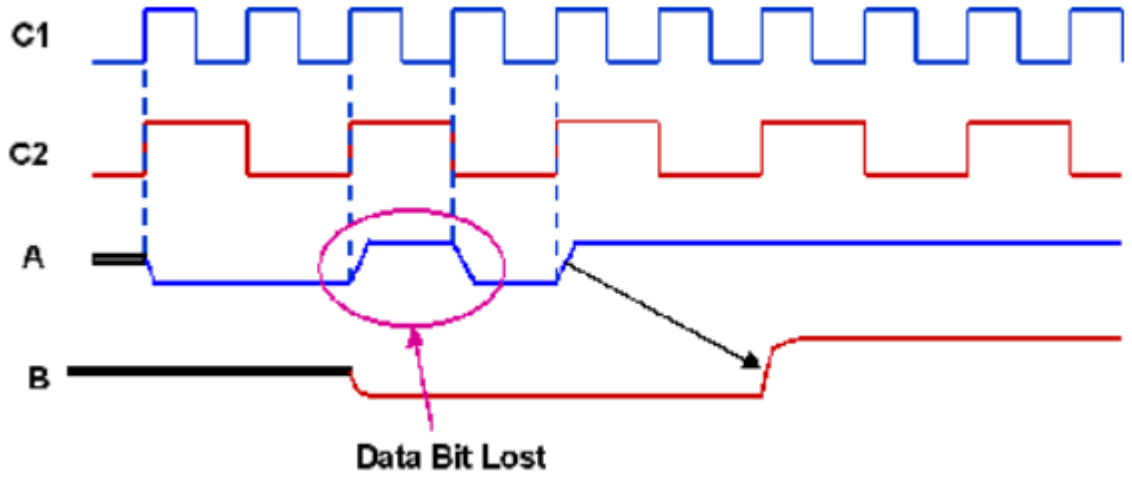


- **Synchronizers must be designed to reduce the chances system failure due to metastability**
- **Synchronizer requirements**
  - Reliable [high MTBF]
  - Low latency [works as quickly as possible]
  - Low power/area impact
- **For some designs 3-FF synchronizer might be required if source frequency is high and input data is changing fast**
- **Can increase MTBF by adding more series stages**
  - 3-FF synchronizer can have higher MTBF

Signal Crossing Fast Clock Domain to Slow Clock Domain

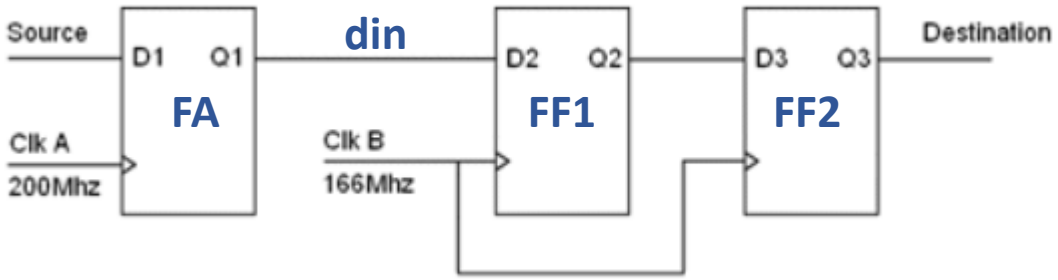
- **Scenario-2 : Flipflop-A clock frequency > Flipflop-B clock frequency**

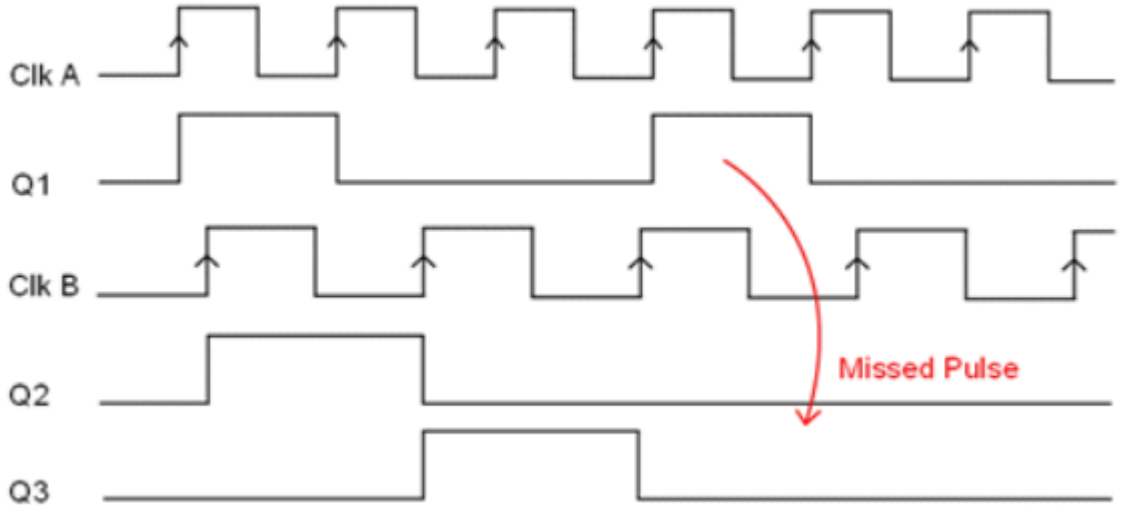




- C1 is two times faster than C2 and there is no phase difference between C1 and C2
- If signal A is changing rapidly, say for example signal A sequence is "00101111", output from FB in C2 clock domain will be "0011".
  - Here the third data value in the input sequence which is "1" is lost
  - For some circuits data loss is not acceptable
- **Two or three Flipflop synchronizer technique will not able to address data loss, instead it will require storage between FA and FB** !
  - Asynchronous FIFO (First in First Out) can be used between FA and FB with correct FIFO depth to address data loss

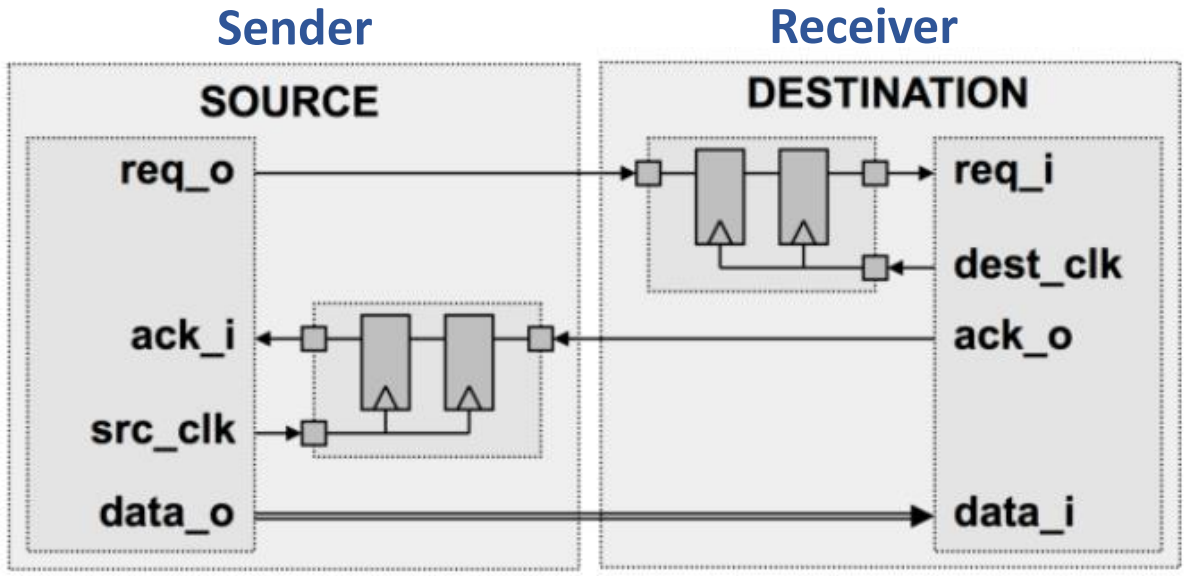
2-FF Synchronizer for Signal crossing Fast Clock to Slow Clock Domain

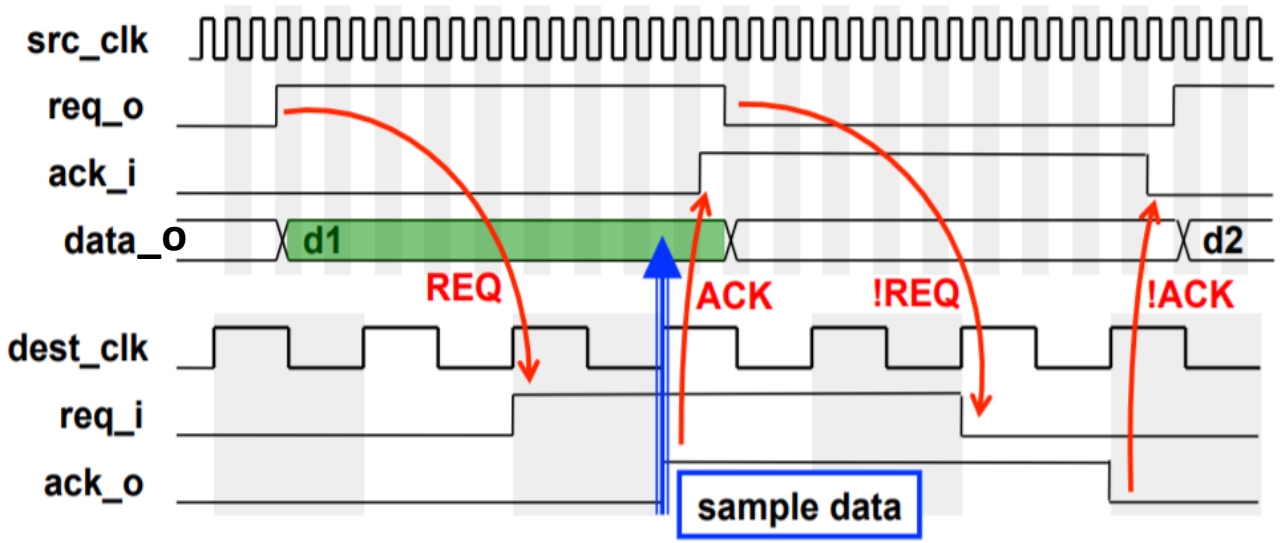




- Consider, signal crossing from source clock domain ClkA to Destination ClkB domain where ClkA is faster than ClkB
- ClkA is 200 Mhz and Clk B is 166 MHz
- Signal is Crossing Fast to Slow clock domain through 2-FF synchronizers (FF1 and FF2)
- Adding 2-FF or 3-FF synchronizer will still result in data loss when signal crossed fast to slow clock domain as seen in the waveform above
- If source signal “din” width entering FF1 can be guaranteed to be **1.5** times of receiving clock ClkB then 2-FF or 3-FF synchronizer still can be used for signals crossing fast to slow clock domain
- If source signal cannot be guaranteed to 1.5 times of receiving clock and if the data loss is not acceptable for subsequent circuit in destination clock domain **then 2-FF or 3-FF open ended synchronization technique should not be deployed !**

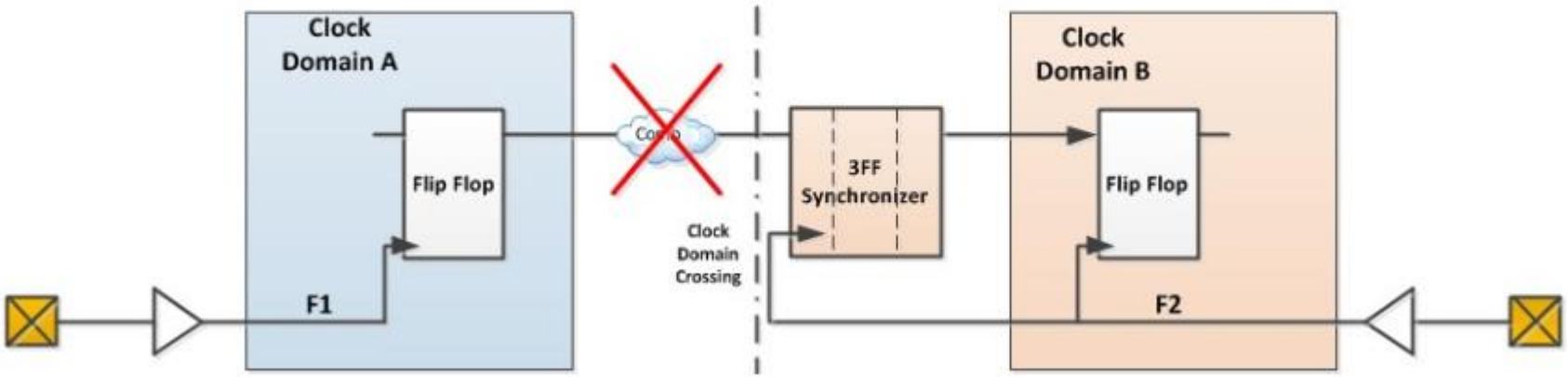
Handshake Synchronizer For Multi-bit Signal Crossing Fast to Slow Clock Domain





- For many open-ended data-passing applications, a simple two-line handshaking sequence is sufficient
- The sender places data onto a data bus (**data_o**) and then synchronizes a "**req_o**" signal (request) to the receiving clock domain.
- When the "**req_o**" signal is recognized in the destination clock domain, the receiver clocks the data_o into a register (the data should have been stable for at least two/three sampling clock edges in the destination clock domain)
- Receiver then passes an "**ack_o**" signal (acknowledgement) through a synchronizer to the sender.
- When the sender recognizes the synchronized "**ack_o**" signal, the sender can change the value being driven onto the data bus (**data_o**)

Register Output Before Signal Crosses Clock Domain



- **Avoid any** combinational logic between the launching flop and synchronizer, this will affect the MTBF of the synchronizer.
- It can also cause glitches if not handled carefully.

- Add a register after combinational logic if it can be avoided, to prevent glitch's entering another clock domain

2-FF and 3-FF Synchronizer Code

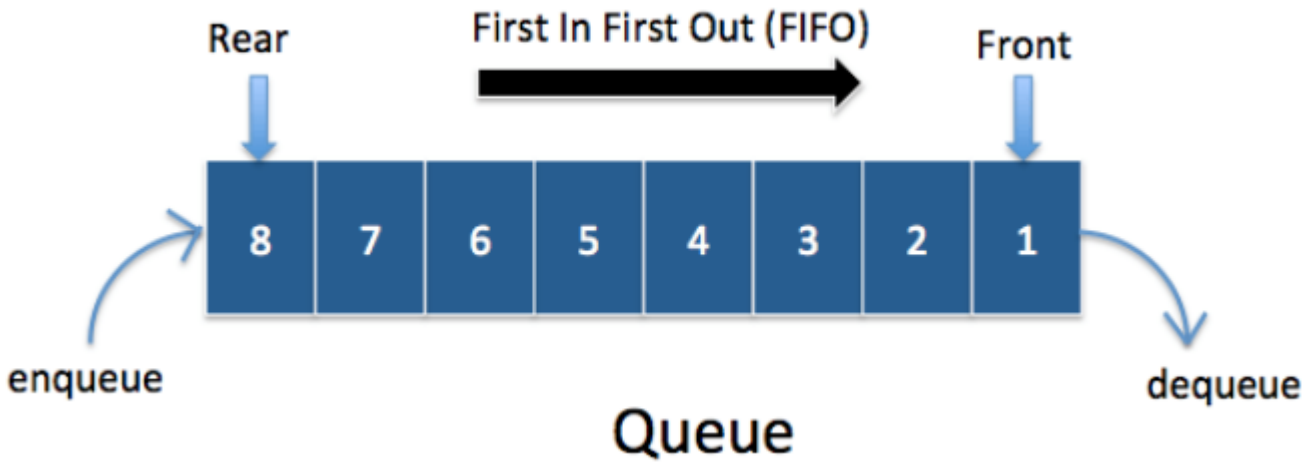
- 2FF and 3FF synchronizer can be built using shift register

```
module ff_sync_2(
  input logic clk, reset,
  input logic d,
  output logic q);

logic ff1;
always_ff@(posedge clk, posedge reset) begin
  if(reset == 1) begin
     q <= 0;
  end
  else begin
     ff1 <= d;
     q <= ff1;
  end
end
endmodule: ff_sync_2
```

```
module ff_sync_3(
  input logic clk, reset,
  input logic d,
  output logic q);

logic ff1, ff2;
always_ff@(posedge clk, posedge reset) begin
  if(reset == 1) begin
     q <= 0;
  end
  else begin
     ff1 <= d;
     ff2 <= ff1;
     q <= ff2;
  end
end
endmodule: ff_sync_3
```

What is a FIFO ?

- **FIFO means First-In First-Out**
  - A FIFO is a structure used in hardware or software application when one needs to buffer data
  - FIFO is a queue, with first data in comes out first
  - FIFO contains memory structure to store incoming data



FIFO Concepts

- FIFO is implemented using a **circular memory buffer** to store data element
- FIFO has a write pointer and read pointer in order to write data to the current memory address and read data from the memory address
- The maximum number of elements that can bed stored inside FIFO memory is known as the **FIFO depth**
- Each element inside FIFO memory can be 1 or more number of bits. It is known as **DATA WIDTH**
- In Figure-A, since write pointer (wr_ptr) and read pointer (rd_ptr) is pointing to same address location. This means FIFO is empty
- In Figure-B, write pointer has advanced and it is pointing to memory address location '3'. It means three data elements E1, E2, E2 were pushed into the FIFO. Since read pointer is pointing to 0^{th} address location, it means no data has been popped (read) out of FIFO yet.

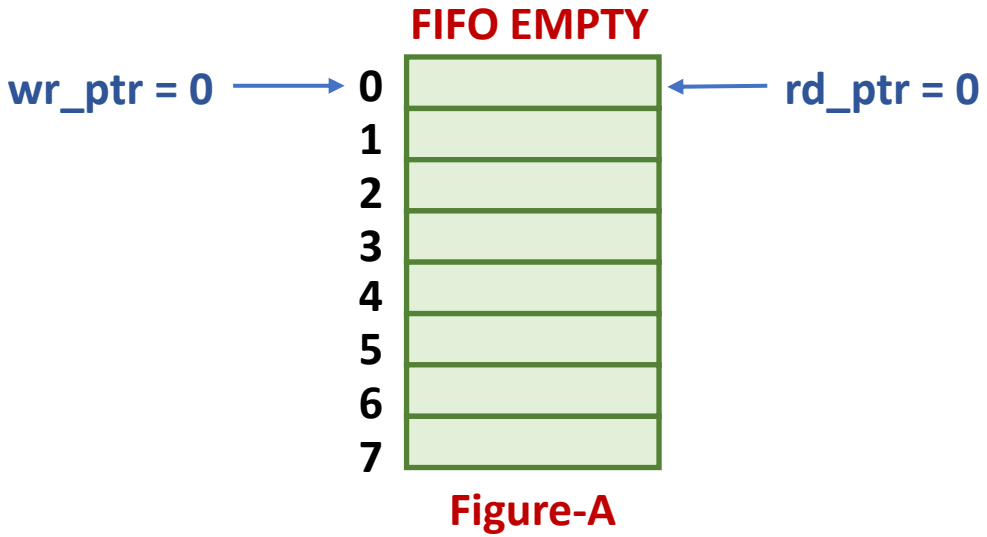
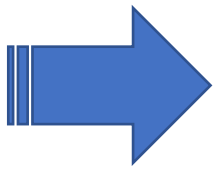


Figure-A



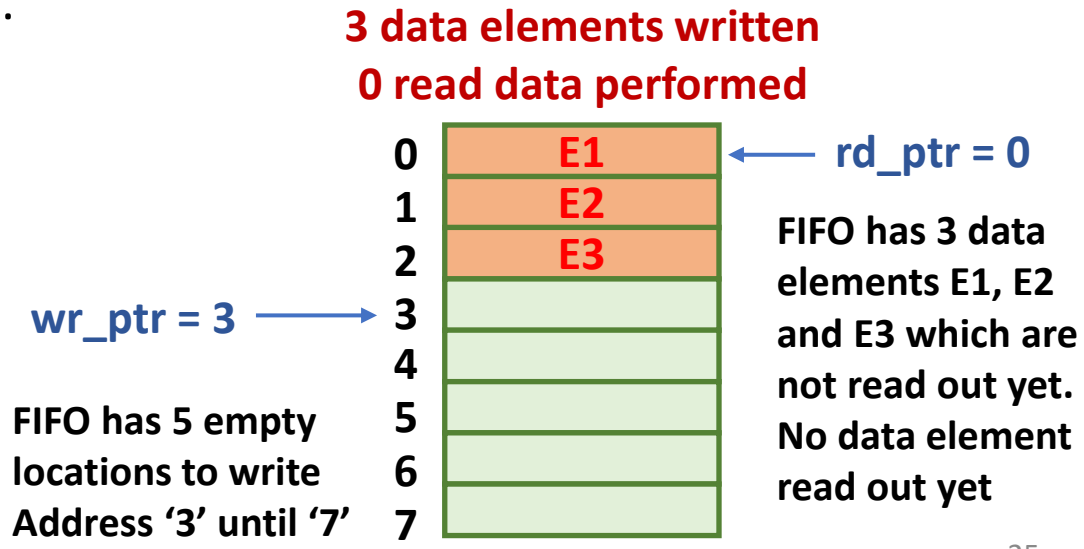
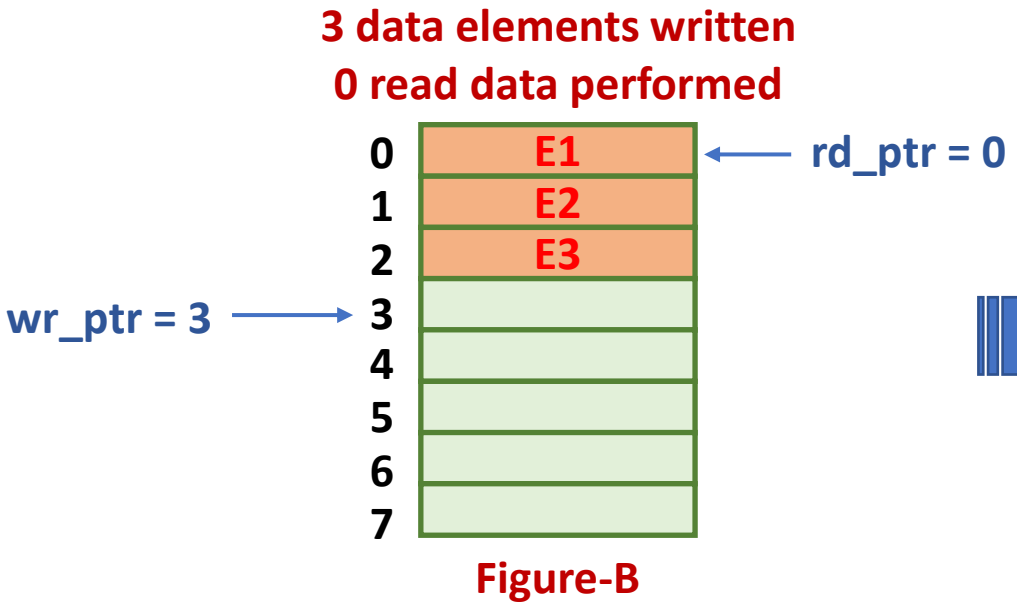
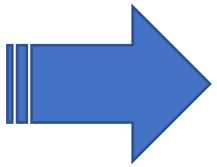


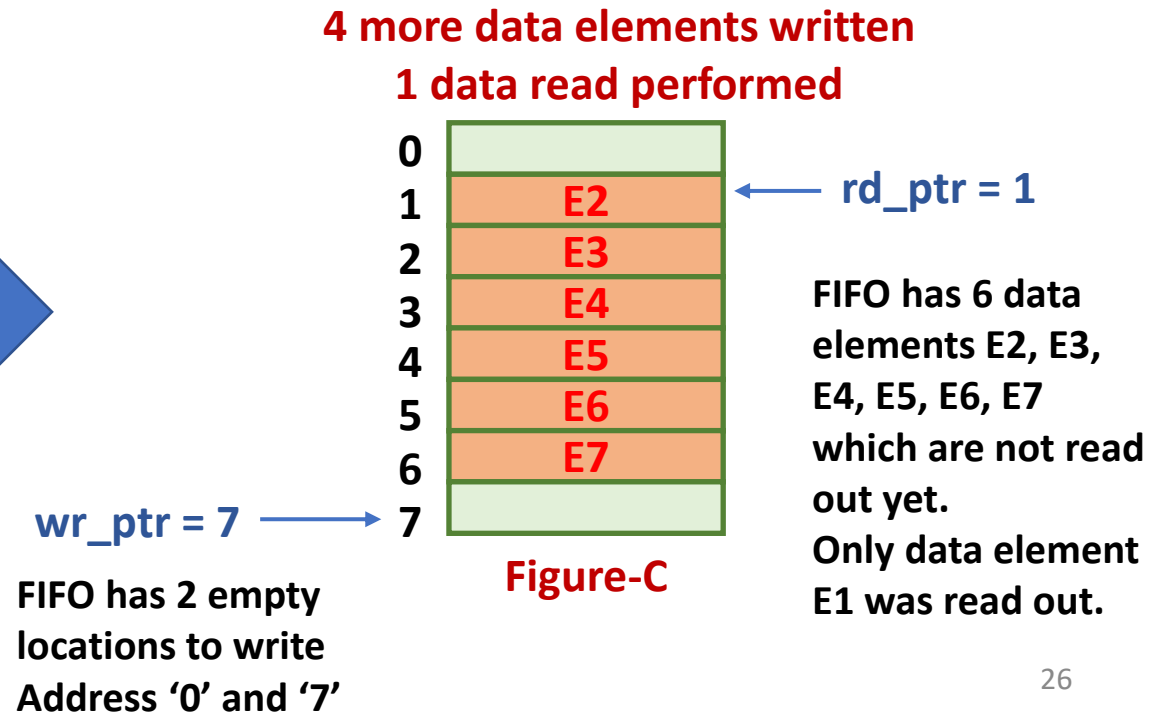
Figure-B

FIFO Concepts

- **In Figure-C, write pointer has advanced and it is pointing to memory address location '7'**
  - It means 4 new data elements E4, E5, E6, E7 were pushed into the FIFO.
- **In Figure-C, read pointer has advanced to address location '1'**
  - it means data from address location '0' which was E1, was read out from the FIFO memory
  - Remember, E1 was the first data entered into the FIFO and now the first data to exit

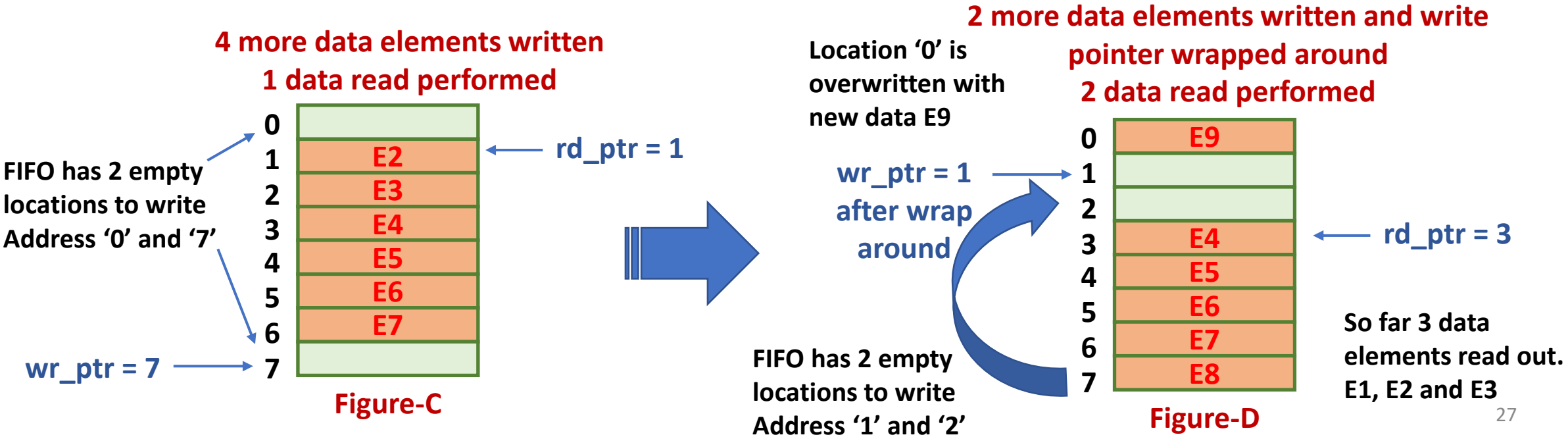






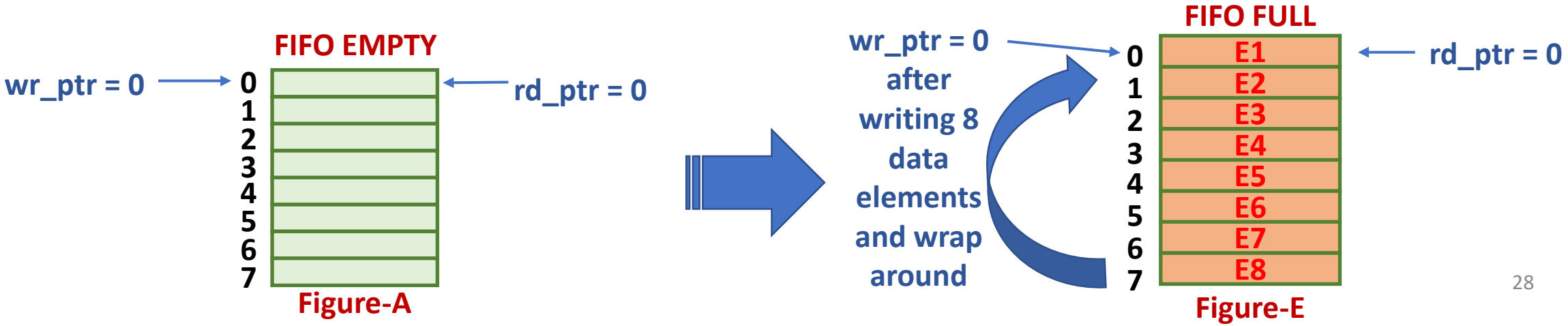
FIFO Concepts

- **In Figure-D, write pointer has advanced and it has wrapped around to point to location ‘1’**
  - It means 2 new data elements E8, E9 were pushed into the FIFO.
  - FIFO write pointer was able to wrap around since there were 2 new elements were to be pushed into FIFO. Location ‘7’ was available and location ‘0’ was freed up since E1 data from location ‘0’ was read out previously.
  - Since write and read pointers can wrap around, FIFO memory is known as **circular buffer**
- **In Figure-D, read pointer has advanced to address location ‘d’**
  - It means data from address location ‘1’ and ‘2’ which were E2 and E3 were read out



FIFO Concepts

- **FIFO Full and FIFO Empty can look the same !**
  - In Figure-A and Figure-E, both **wr_ptr** and **rd_ptr** are pointing to same location '0'
  - Pointing to same address location does not always mean that FIFO is empty
- **To determine if FIFO is empty or FIFO is full, additional information is required to differentiate**
  - Typically additional address bit is added to the MSB of write and read pointers
  - If FIFO_DEPTH = 8, then wr_ptr and rd_ptr width is 3 bit each
  - In such case, 4th bit is added to wr_ptr and rd_ptr. This is called as a "**wrap**" around bit
- **FIFO is Empty** if (wr_ptr[3:0] == rd_ptr[3:0])
  - Wrap around MSB bit of write and read pointer and remaining bits have same value
- **FIFO is Full** if ((wr_ptr[**3**] != rd_ptr[**3**]) && (wr_ptr[**2:0**] == rd_ptr[**2:0**]))
  - MSB wrap around bit's or write and read pointer are different and remaining bits are same



FIFO Concepts

- **Both Figure-E and Figure-F represents FIFO Full Condition**
  - In Figure-A and Figure-E, both **wr_ptr** and **rd_ptr** are pointing to same address locations
  - In Figure-A, 8 data elements were written and 0 were read out. Hence FIFO is full as max occupancy in FIFO memory is 8
  - In Figure-F, 10 data elements were written and 2 were read out. Hence FIFO is full as max occupancy in FIFO memory is 8
  - In Figure-E, **wr_ptr** = 4'b**1000 and rd_ptr** = 4'b**0000**
    - **FIFO_FULL = 1, since (wr_ptr[3] != rd_ptr[3] && wr_ptr[2:0] == rd_ptr[2:0])**
  - In Figure-F, **wr_ptr** = 4'b**1010 and rd_ptr** = 4'b**0010**
    - **FIFO_FULL = 1, since (wr_ptr[3] != rd_ptr[3] && wr_ptr[2:0] == rd_ptr[2:0])**

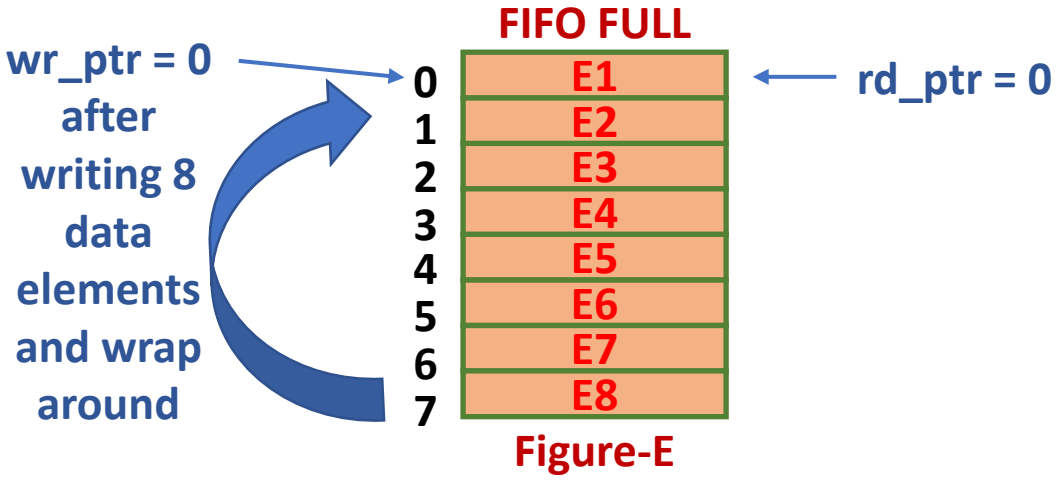


Figure-E

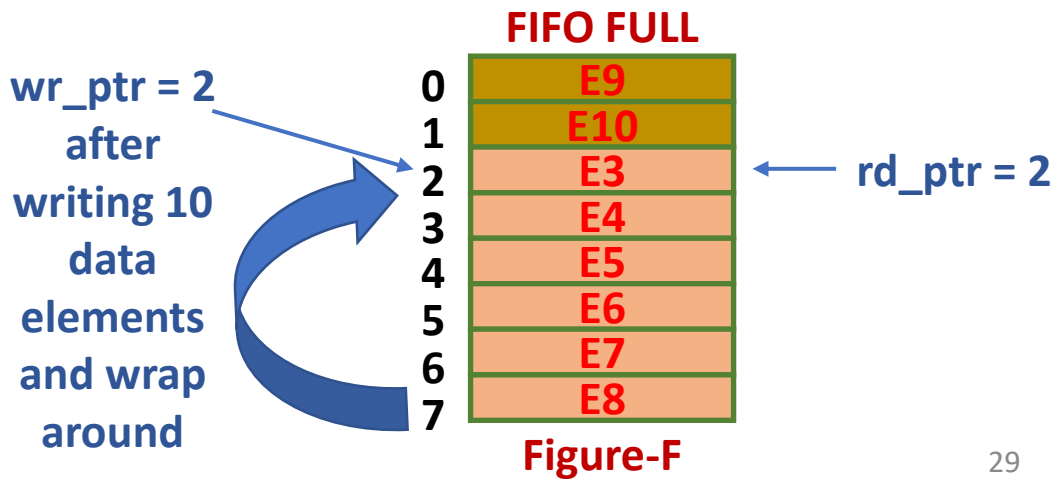
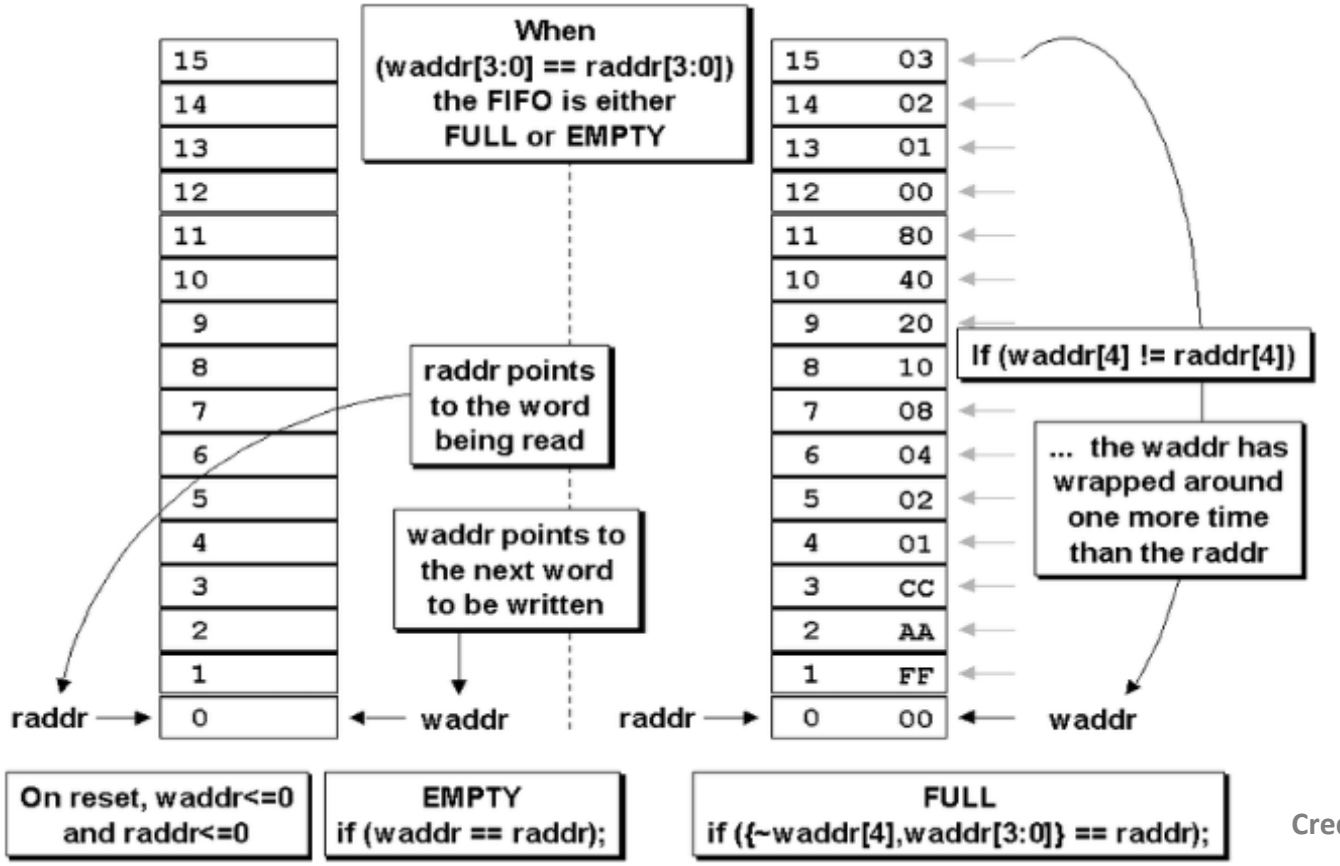


Figure-F

FIFO Concepts

❑ FIFO Full and FIFO Empty differentiation using MSB bit (wrap around Bit)

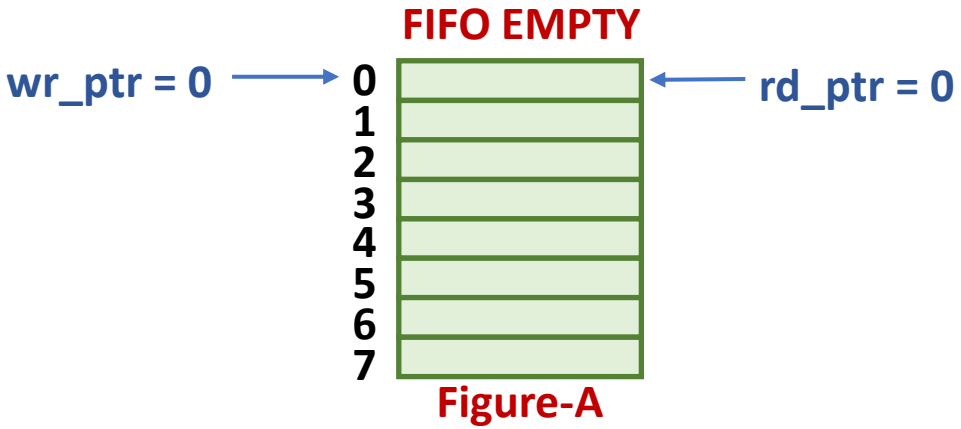
- Note : waddr and raddr in diagram is same as write and read pointers (wr_ptr and rd_ptr)
- Below mentioned example is with FIFO_DEPTH -16 with raddr and waddr have 5 bits each
  - MSB bit position waddr[4] and raddr[4] is the wrap around bit to differentiate between full and empty condition. Hence FIFO depth is still 2 ^ 4 = 16 and not 2 ^ 5 = 32.
  - In another words, only 16 data elements can be stored in FIFO Memory since MSB bit 4 is a wrap around bit

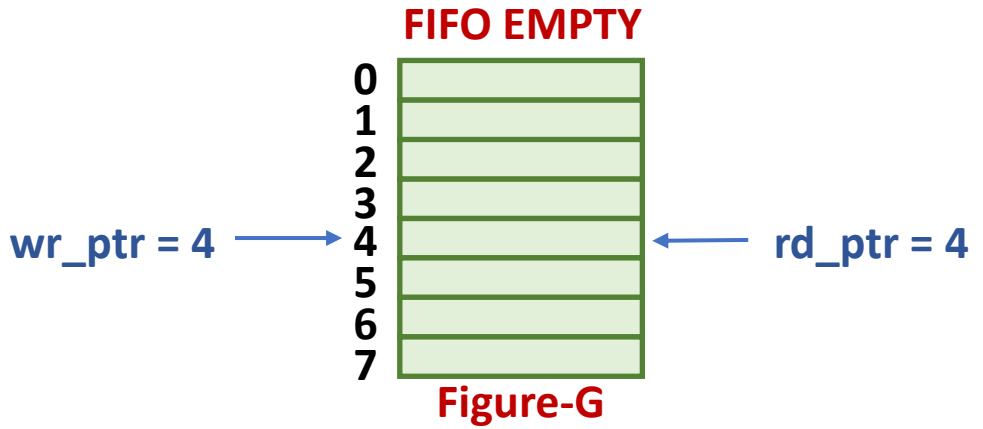


Credit : Clifford E. Cummings

FIFO Concepts

- **Both Figure-A and Figure-G represents FIFO Empty Condition**
  - In Figure-A and Figure-G, both **wr_ptr** and **rd_ptr** are pointing to same address locations
  - In Figure-A, 0 data elements were written and 0 were read out. Hence FIFO is empty since all 8 locations in FIFO memory are available to be written
  - In Figure-G, 4 data elements were written and 4 were read out. Hence FIFO is empty since all 8 locations in FIFO memory are available to be written
  - In Figure-A, **wr_ptr** = 4'b**0000 and rd_ptr** = 4'b**0000**
    - **FIFO_EMPTY = 1, since (wr_ptr[4:0] == rd_ptr[4:0])**
  - In Figure-G **wr_ptr** = 4'b**0100 and rd_ptr** = 4'b**0100**
    - **FIFO_EMPTY = 1, since (wr_ptr[4:0] == rd_ptr[4:0])**

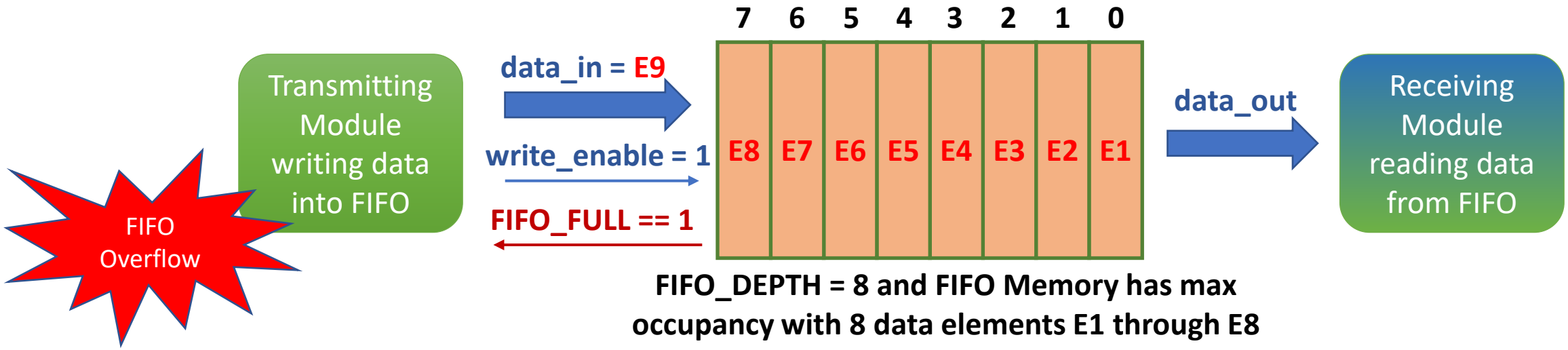




FIFO Concepts

❑ FIFO Overflow (also known as FIFO Overrun):

- When FIFO internal memory is full and any attempt to write a new data element to FIFO memory is called FIFO overflow condition
- Overflow expression is : (FIFO_FULL == 1) && (write_enable == 1). **See below mentioned Figure.**
- It is the responsibility of transmitting module to ensure it does not write to FIFO when it is full
- Typically FIFO designs provides FIFO_FULL signal which gives indication to transmitting module if FIFO has any room to write or not
- Additionally FIFO designs provides such more flags to indicate memory occupancy information
  - **FIFO_ALMOST_FULL :** FIFO internal memory has 1 location remaining to write before it is full. Acts like an early full indication
  - **FIFO_HALF_FULL** : FIFO internal memory has 50% occupancy
  - **FIFO_FULL** : FIFO is full and its internal memory has max occupancy

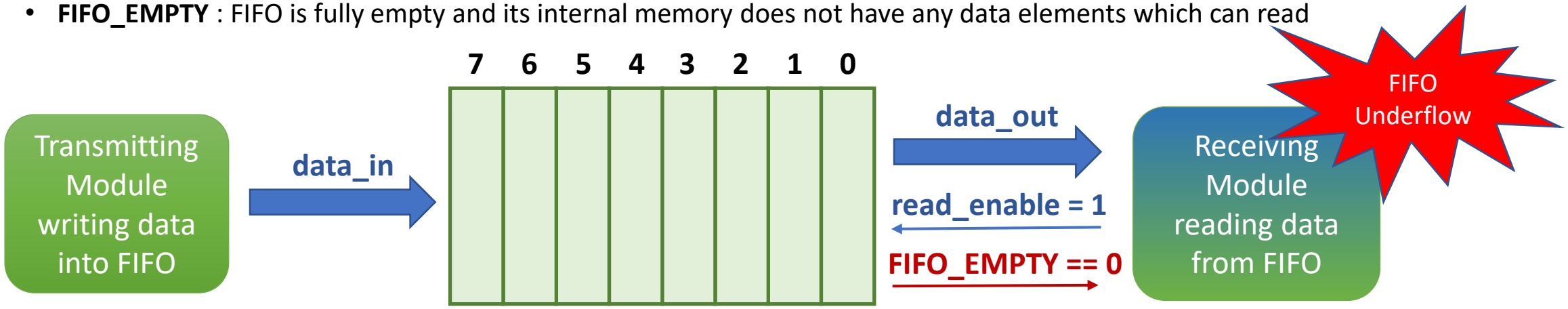


FIFO_DEPTH = 8 and FIFO Memory has max occupancy with 8 data elements E1 through E8

FIFO Concepts

❑ FIFO Underflow (FIFO Underrun) :

- When FIFO internal memory is fully empty and any attempt to read any location of FIFO memory is called FIFO underflow condition
- Underflow expression is : (FIFO_EMPTY == 0) && (read_enable == 1). **See below mentioned Figure.**
- It is the responsibility of receiving module to ensure it does not read FIFO memory when it is empty
- Typically FIFO designs provides FIFO_EMPTY signal which gives indication to receiving module if FIFO has any data element to read or not.
- Additionally FIFO designs provides such more flags to indicate memory occupancy information
  - **FIFO_ALMOST_EMPTY :** FIFO internal memory has 1 data element remaining to be read before it is fully empty. Acts like an early empty indication
  - **FIFO_HALF_EMPTY** : FIFO internal memory has 50% occupancy
  - **FIFO_EMPTY** : FIFO is fully empty and its internal memory does not have any data elements which can read

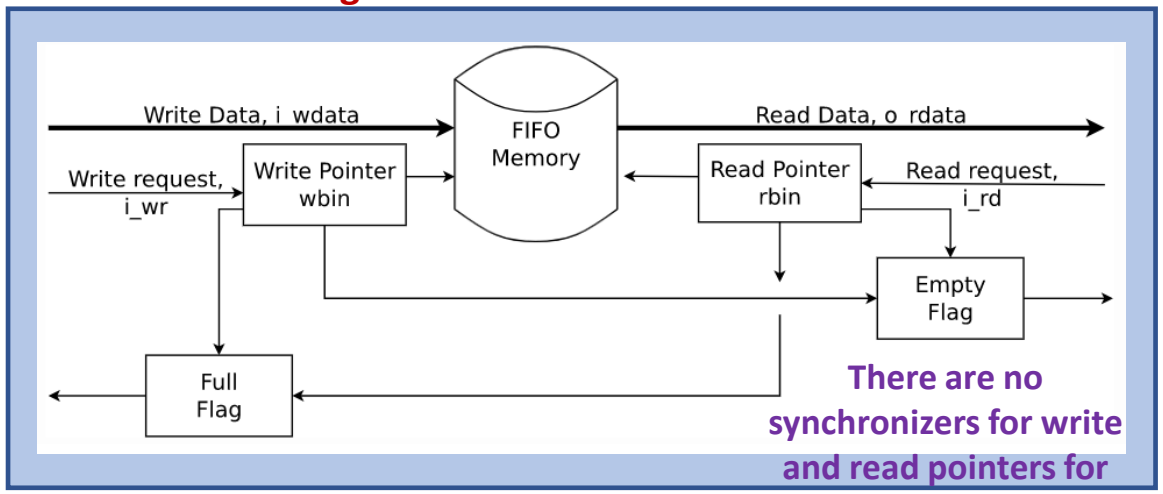


FIFO_DEPTH = 8 and FIFO Memory has all 8 locations which does not have any data

Types of FIFO

- **There are 4 kinds of FIFO**
  - Shift registers FIFO
  - Exclusive Read/Write FIFO's
  - Concurrent Read/Write FIFO's
    - **Synchronous and Asynchronous** FIFO
- **Basic difference between Synchronous and Asynchronous FIFO :**
  - In case of synchronous FIFO both write and read operation is performed on the same clock
  - In case of asynchronous FIFO write operation and read operation of asynchronous FIFO are performed on different clocks
    - **Write and read clock can run independently with same or different frequency !**

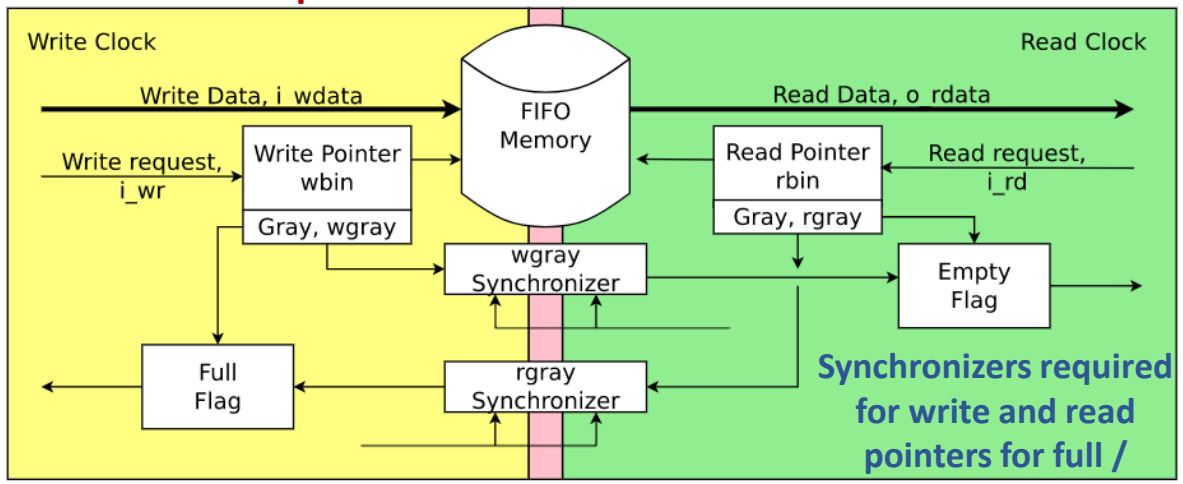
Single Clock for write and Read



There are no synchronizers for write and read pointers for full / empty generation

Synchronous FIFO

Separate Clock for write and read



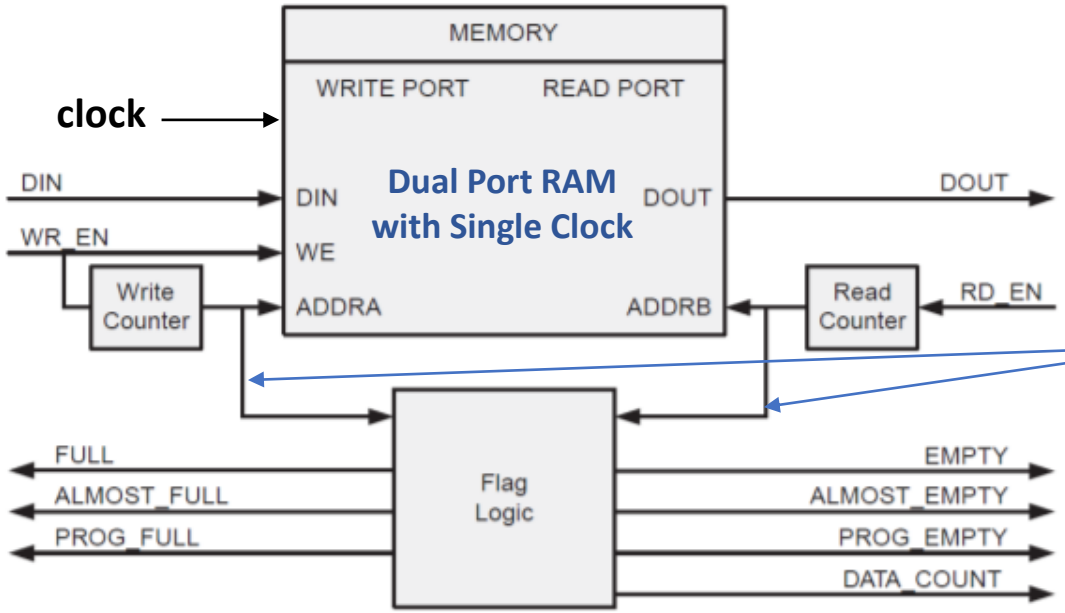
Synchronizers required for write and read pointers for full / empty generation

Asynchronous FIFO

Synchronous FIFO

Single clock used for both read and write operation !

Dual Port RAM supports simultaneous write and read operation !



There are no synchronizers for write and read address pointers for full / empty Flag generation !

FIFO Operation :

- Within FIFO there is a has dual port memory block for storage. **It allows simultaneous write and read operation !**
- FIFO has an input data in port (DIN) and an output read port (DOUT).
- Each data port has its own associated pointers which points to a location in the memory
- After a FIFO reset both the write and read pointers will be at the first memory location within the FIFO.
- Each write operation will cause the write pointer to increment to the next location in memory
- Each read operation will cause the read pointer to increment to the next location
- FIFO has full and empty signals indicating whether it has no empty location for any new data to be stored (full condition) or it has no data available for reading (empty condition)
- Full marker prevents overriding of existing data. This is known as fifo overrun or overflow condition
- Empty marker prevents reading junk data, when it is empty. This is known as fifo underrun or underflow condition

Synchronous FIFO RTL Model

```
`include "dual_port_ram.sv"
module sync_fifo#(
  parameter DATA_WIDTH = 32,
  parameter FIFO_DEPTH = 16)
(
  input  logic clk,
  input  logic reset,
  input  logic wr_en,
  input  logic rd_en,
  input  logic [DATA_WIDTH-1:0] data_in,
  output logic [DATA_WIDTH-1:0] data_out,
  output logic fifo_full,
  output logic fifo_empty
);

 // Local parameter to set address width based on FIFO
DEPTH
 localparam ADDR_WIDTH = $clog2(FIFO_DEPTH);

 // internal register declaration
 logic [ADDR_WIDTH:0]  wr_ptr;
 logic [ADDR_WIDTH:0]  rd_ptr;
```

```
// Step-1 : Increment write pointer each time wr_en is '1'
always_ff@(posedge clk, posedge reset) begin
  if(reset) begin
    wr_ptr <= 0;
  end
  else begin
   if(wr_en && !fifo_full) begin
     wr_ptr <= wr_ptr + 1;
                end
  end
end


// Step-2 : Increment read pointer each time rd_en is '1'
always_ff@(posedge clk,posedge reset) begin
  if(reset) begin
    rd_ptr <= 0;
  end
  else begin
   if(rd_en && !fifo_empty) begin
    rd_ptr <= rd_ptr + 1;
                end
  end
end
```

Synchronous FIFO RTL Model

```
// Step-3 : The FIFO is empty when both read and write pointers point to the same location
assign fifo_empty =  (wr_ptr == rd_ptr) ? 1 : 0;

// Step-4: Fifo is full when wr_ptr - rd_ptr = 2^address_width.
// In that case, the Lower address bits are identical, but the MSB address bit is different.
assign fifo_full  =  ((wr_ptr[ADDR_WIDTH] != rd_ptr[ADDR_WIDTH]) &&
                      (wr_ptr[ADDR_WIDTH-1:0] == rd_ptr[ADDR_WIDTH-1:0])) ? 1 : 0;

// Step-5 : Instantiate FIFO Memory
dual_port_ram #(
  .DATA_WIDTH(DATA_WIDTH),
  .ADDR_WIDTH(ADDR_WIDTH))
fifo_memory(
 .write_addr(wr_ptr),
 .read_addr(rd_ptr),
 .write_data(data_in),
 .read_data(data_out),
 .wr_en(wr_en && !fifo_full),        // And'ing wr_en with !fifo_full is required to avoid writing to fifo overflow
 .rd_en(rd_en && !fifo_empty),       // And'ing rd_en with !fifo_empty is required to avoid writing to fifo underflow
 .wr_clk(clk),
 .reset(reset)
);
endmodule:sync_fifo
```

Synchronous FIFO Resource Usage

```
// Simple Dual-Port Single Clock Distributed RAM with Asynchronous Read
module dual_port_ram #(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=32)
(  input logic wr_clk, reset,
   input logic wr_en,
   input logic[DATA_WIDTH-1:0] write_data,
   input logic[ADDR_WIDTH-1:0] write_addr,
   input logic rd_en,
   input logic[ADDR_WIDTH-1:0] read_addr,
   output logic[DATA_WIDTH-1:0] read_data);

   // Two dimensional memory array
   logic[DATA_WIDTH-1:0] mem[2**ADDR_WIDTH-1:0];

   // Synchronous write to FIFO Internal Memory
   always_ff@(posedge wr_clk, posedge reset) begin
    if(reset) begin
      for(int i=0; i<(2**ADDR_WIDTH); i++) begin
        mem[i] <= 0;
      end
    end
    else begin
      if(wr_en) mem[write_addr]  <= write_data;
    end
   end

   // Asynchronous read from FIFO Internal Memory
   assign read_data = (rd_en == 1) ? mem[read_addr] : 0;
endmodule:dual_port_ram
```

Due to asynchronous read implementation of dual port ram without registering address for read, Quartus will *not* infer internal block ram. Instead distributed ram will be inferred using logic registers

Analysis & Synthesis Resource Usage Summary

<<Filter>>

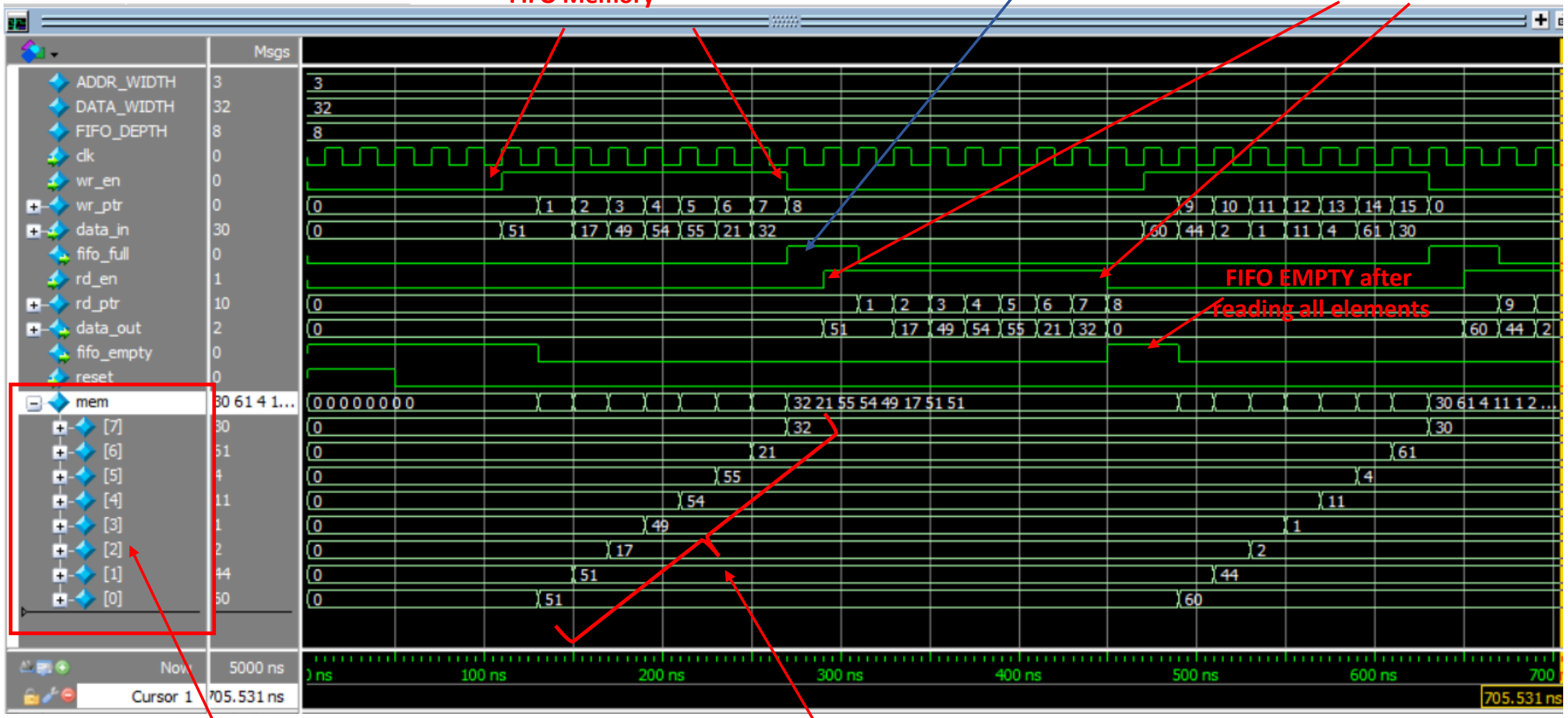
	Resource	Usage
3		
4	Estimated ALUTs Unavailable	0
1	-- Due to unpartnered combinational logic	0
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	231
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	0
2	-- 6 input functions	149
3	-- 5 input functions	72
4	-- 4 input functions	0
5	-- <=3 input functions	10
8		
9	Combinational ALUTs by mode	
1	-- normal mode	231
2	-- extended LUT mode	0
3	-- arithmetic mode	0
4	-- shared arithmetic mode	0
10		
11	Estimated ALUT/register pairs used	737
12		
13	Total registers	522
1	-- Dedicated logic registers	522
2	-- I/O registers	0

Synchronous FIFO Simulation Waveform

when wr_en==1 and fifo_full == 0
datain values : 51, 17, 49, 54, 55, 21,32 are stored in FIFO Memory

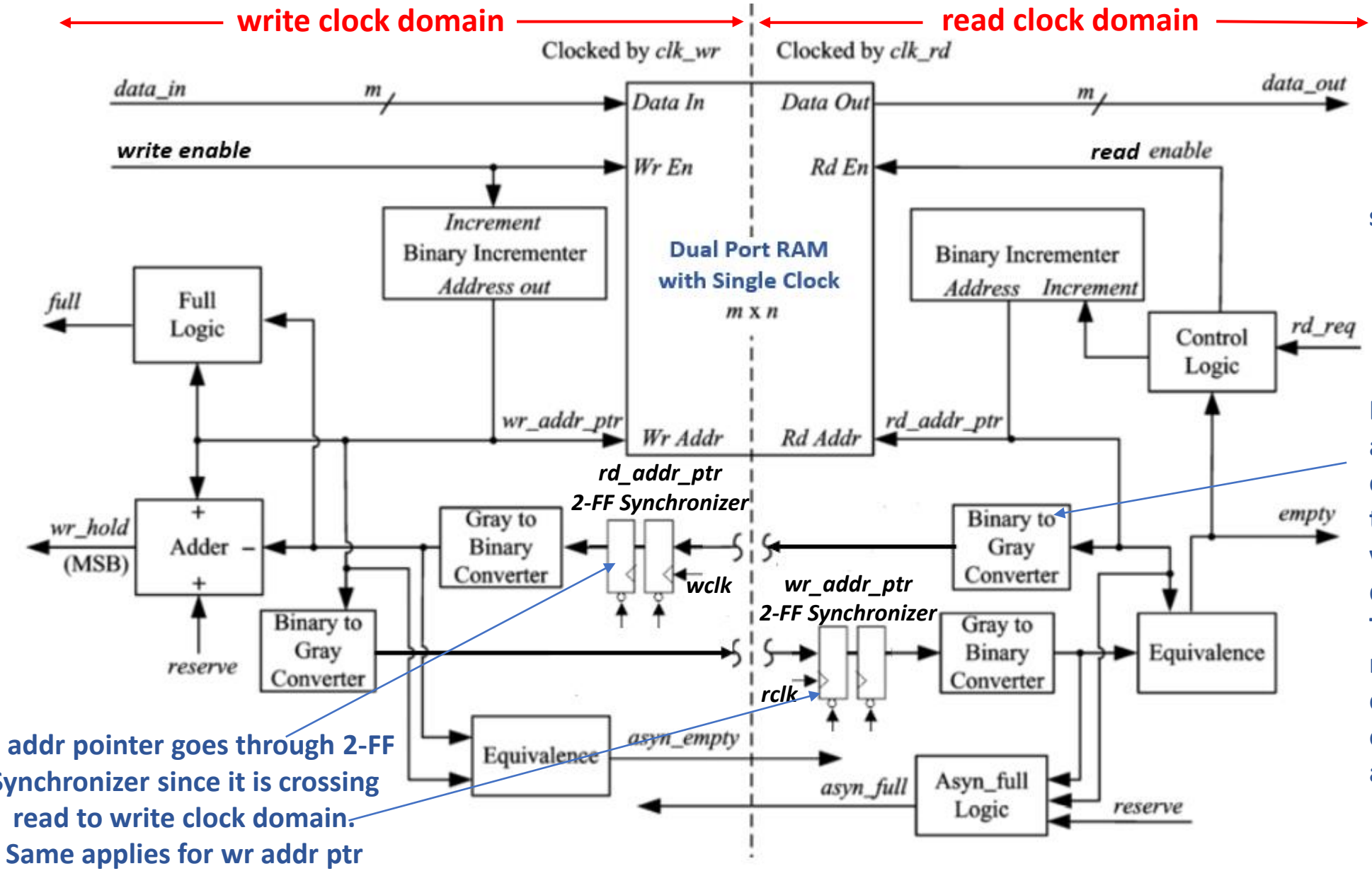
After writing 8 data elements fifo_full = 1 is set

when rd_en==1 and fifo_empty == 0
datain values : 51, 17, 49, 54, 55, 21,32 are read out from FIFO Memory



FIFO Memory (mem) has datain values : 51, 17, 49, 54, 55, 21,32. Each rd_en=1 caused data to push into FIFO memory

Asynchronous FIFO



Dual Port RAM with Single clock is used Since write is synchronous and read is asynchronous !

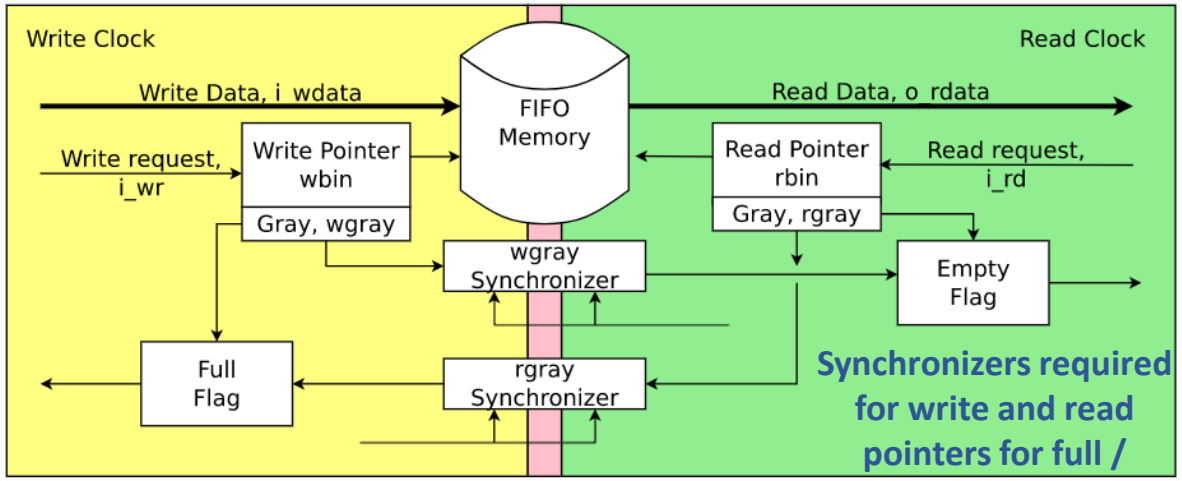
Before sending read pointer and write pointer to other clock domain, it is converted to gray coded value. In Gray value only 1 bit changes during counting.
This counter update strategy minimizes the error when counter value passes from one clock domain to another

rd addr pointer goes through 2-FF Synchronizer since it is crossing read to write clock domain.
Same applies for wr addr ptr

Asynchronous FIFO

- **Asynchronous FIFO write and read operation is similar to synchronous FIFO except :**
  - In case of asynchronous FIFO there are 2-FF synchronizers are used to synchronize write pointer coming from write clock domain to read clock domain. This is required for fifo empty flag generation.
  - Similarly, 2-FF synchronizers are used to synchronize read pointer coming from read clock domain to write clock domain. This is required for fifo full flag generation.
  - Additionally before write and read pointers are fed to the synchronizer, both write and read pointer binary count values are converted to gray encoded values.
    - In gray encoded value only 1 bit changes during counting. This counter update strategy minimizes the error when counter value passes from one clock domain to another

Separate Clock for write and read

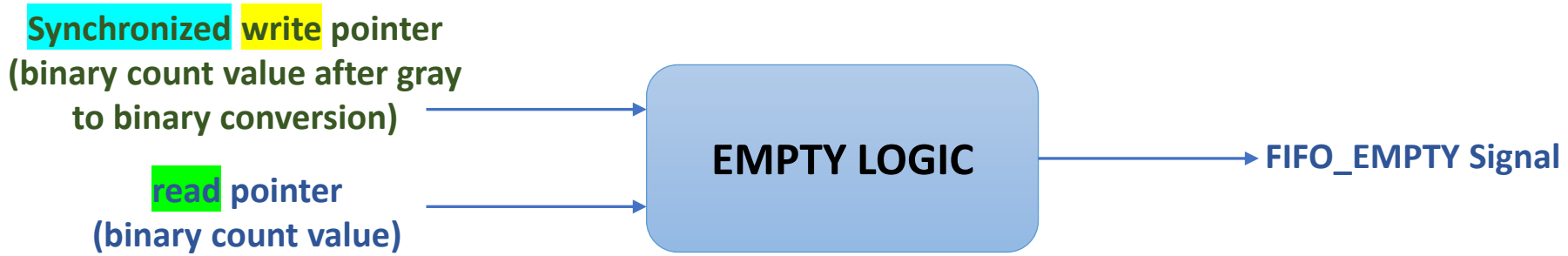


Asynchronous FIFO

Asynchronous FIFO Full and Empty Generation


```
assign fifo_full  =  ((wr_ptr[ADDR_WIDTH] != rd_ptr_binary2[ADDR_WIDTH])
                    && (wr_ptr[ADDR_WIDTH-1:0] == rd_ptr_binary2[ADDR_WIDTH-1:0])) ? 1 : 0;
```

Note : rd_ptr_binary2 is synchronized read pointer binary value



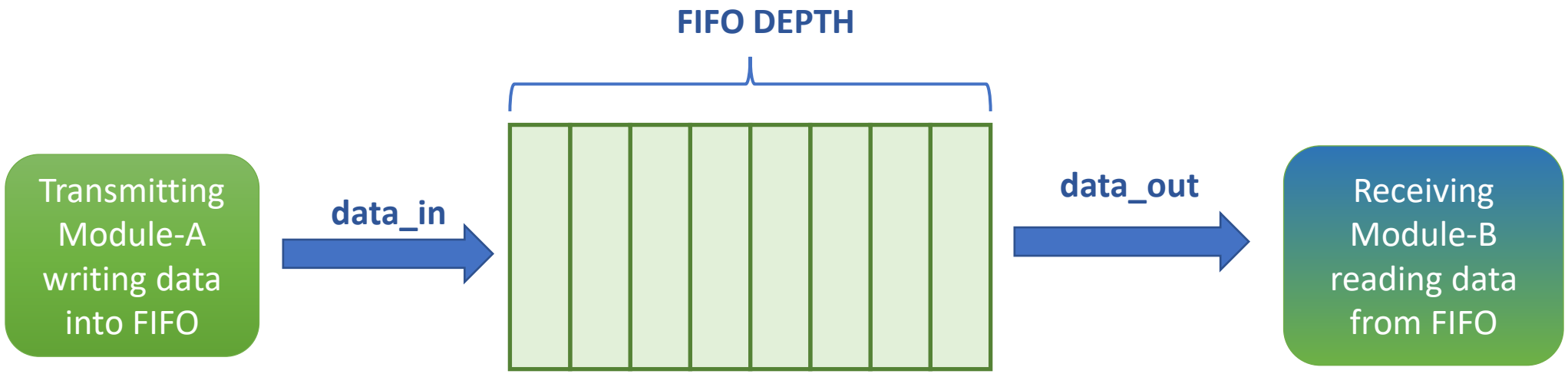

```
assign fifo_empty =  (wr_ptr_binary2 == rd_ptr) ? 1 : 0;
```

Note : wr_ptr_binary2 is synchronized write pointer binary value

Asynchronous FIFO Depth Calculation

- **To compute depth of asynchronous FIFO, following parameters are required to be known :**
  - Write and read clock frequency
  - Amount of data to be written (also know as Burst Length)
  - Rate at which data is written to the memory and rate at which data is read from the memory.

    Example :
    - There is 1 cycle of idle between two consecutive writes from module A to the FIFO
    - There is 3 cycle of idle between two consecutive read request from module B to the FIFO



Asynchronous FIFO Depth Calculation

- **Case-1:**
  - fA > fB with no idle cycles in both write and read.
  - Writing frequency = fA = 80MHz.
  - Reading Frequency = fB = 50MHz.
  - Burst Length = No. of data items to be transferred = 120.
  - There are no idle cycles in both reading and writing which means that, all the items in the burst will be written and read in consecutive clock cycles.

- **Solution:**
  - Time required to write one data item (1 / 80 Mhz) = 12.5 nSec
  - Time required to write all the data in the burst = 120 * 12.5 nSec. = 1500 nSec.
  - Time required to read one data item (1 / 50 Mhz) = 20 nSec.
  - For every 20 nSec, the module B is going to read one data in the burst.
  - In a period of 1500 nSec, 120 no. of data items can be written.
  - And the no. of data items can be read in a duration of 1500 nSec = (1500 nSec / 20 nSec) = 75
  - The remaining no. of bytes to be stored in the FIFO = 120 – 75 = 45.
  - Hence the FIFO which has to be in this scenario must be capable of storing **45** data items.
  - **Answer :** FIFO DEPTH = **45**

Asynchronous FIFO Depth Calculation

Case-2:

- fA > fB with no idle cycles in both write and read.
- Writing frequency = fA = 80MHz.
- Reading Frequency = fB = 50MHz.
- Burst Length = No. of data items to be transferred = 120.
- No. of idle cycles between two successive writes is = 1
- No. of idle cycles between two successive read is = 3

Solution:

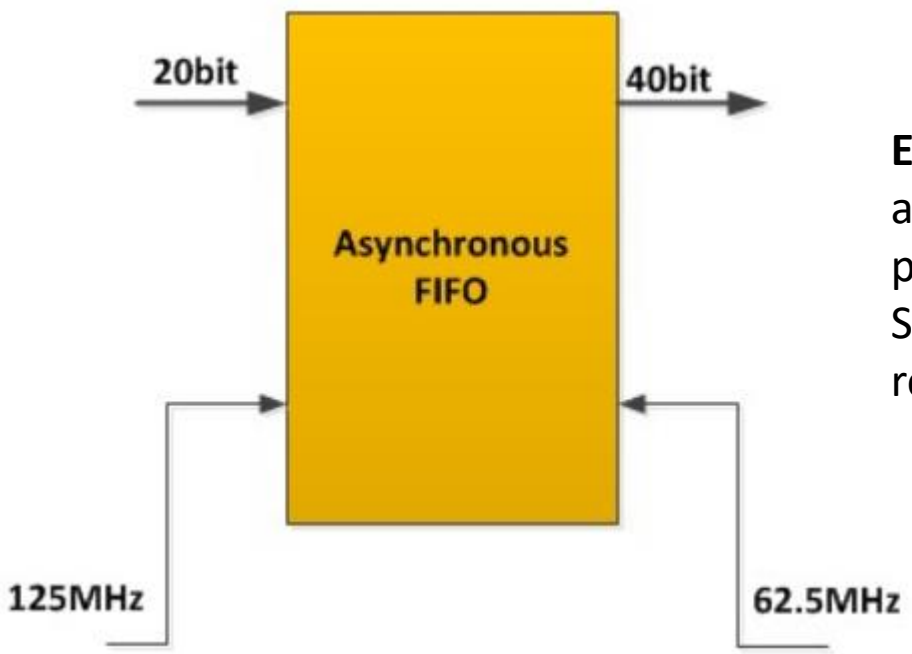
- The no. of idle cycles between two successive writes is **1** clock cycle. It means that, after writing one data, module A is waiting for one clock cycle, to initiate the next write. So, it can be understood that for every **2** clock cycles, one data is written.
- The no. of idle cycles between two successive reads is **3** clock cycles. It means that, after reading one data, module B is waiting for 3 clock cycles, to initiate the next read. So, it can be understood that for every **4** clock cycles, one data is read
- Time required to write one data item **2** * (1 / 80 Mhz) = 25 nSec
- Time required to write all the data in the burst = 120 * 25 nSec. = 3000 nSec.
- Time required to read one data item **4** * (1 / 50 Mhz) = 80 nSec.
- For every 80 nSec, the module B is going to read one data in the burst.
- In a period of 3000 nSec, 120 no. of data items can be written.
- And the no. of data items can be read in a duration of 3000 nSec = (3000 nSec / 80 nSec) = 37.5 (~37)
- The remaining no. of bytes to be stored in the FIFO = 120 – 37 = **83**. **Answer :** FIFO DEPTH = **83**

Asynchronous FIFO Depth Calculation

- **In homework Lab, FIFO DEPTH can be fixed to 8 in testbench based on below mentioned assumption** :
  - Write clock time period is 20ns
  - Read clock time period is 40ns
  - Number of packets to write is 16
  - No. of idle cycles between two successive writes is = 0
  - No. of idle cycles between two successive reads is = 0
  - FIFO Depth = 16 - ((16 * 20ns) / 40ns) = 8

Application of Asynchronous FIFO

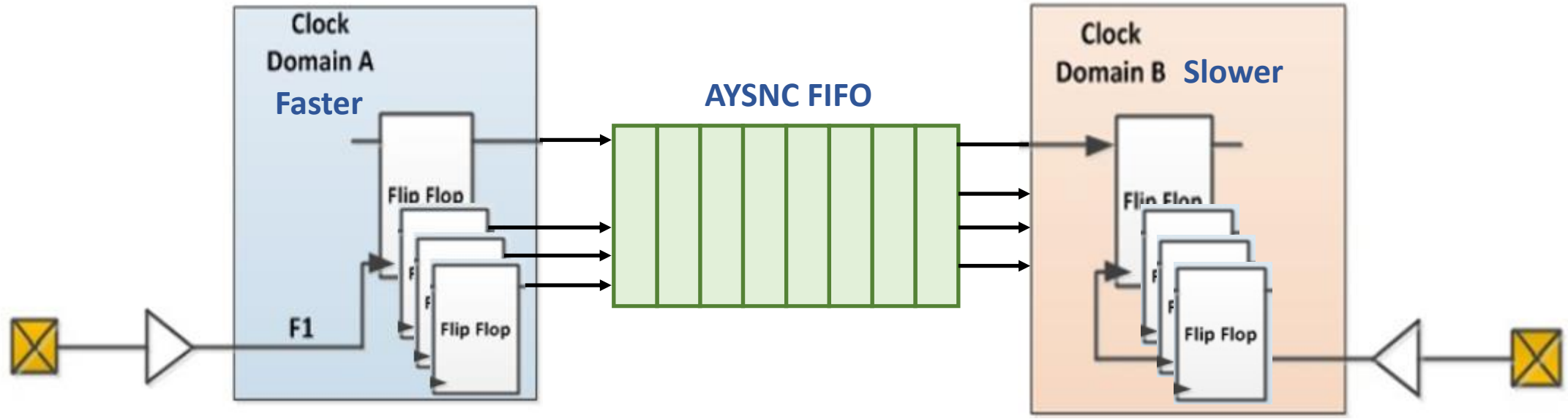
- **Multi-bit data buses are synchronized across clock domain using Asynchronous FIFOs, provided :**
  - Incoming data rate is same as outgoing data rate.
  - That means (**F1 x DW1) = (F2 xDW2)**.
    - Where F1- Frequency of launching flop,
    - F2- Frequency of capturing flop,
    - DW1- write data width to FIFO, DW2- Read data width from FIFO.



Example : 2.5Gigbps data path for PCIe Gen2. This data rate is achieved by using 20bit data path at 125MHz or 40bit data path at 62.5 MHz.
So, an asynchronous FIFO shown in in the diagram will operate reliably without any data loss or duplication

Application of Asynchronous FIFO

- **Asynchronous FIFO are used for multi-bit bus transmission from faster to slower clock domain**
  - Using 2-FF or 3-FF synchronizer for signals crossing fast to slow clock domain can lead into data loss.
  - Instead asynchronous FIFO is used in such scenarios to prevent data loss
  - Designs where data loss is not acceptable when crossing faster to slow clock domain then asynchronous fifo can be used as a synchronizer



Homework Assignment-9

- **Develop SystemVerilog RTL model for M-bit width and N-depth Asynchronous FIFO :**
  - Asynchronous FIFO RTL model should be configurable to set following mentioned parameters :
    - FIFO_DEPTH : This is number of data locations FIFO's internal memory can store. FIFO DEPTH value should be power of 2 (such as 2, 4, 8, 16, 32, and so on). Default value of FIFO DEPTH is set to 8 in testbench.
    - DATA_WIDTH : Width of each data element which can be stored in FIFO's internal memory. By default value is set to 32 data width in Testbench.
  - Support different clock frequency for write and read. Write clock should be faster than read clock
    - In testbench write clock is set to 20 ns (which is 50 Mhz clock) and read clock is set to 40 ns (which is 25 Mhz)
    - In tesbench for above mentioned default write and read clock frequency set NUM_OF_PACKETS=8
      - NUMBER_OF_PACKETS : Number of data elements which needs to be transmitted through FIFO. By default set to '8' in testbench. Use default values provided in testbench which is derived for wr_clk=20ns and rd_clk=40ns
  - Memory inside FIFO should be simple dual port memory with single clock.
    - Memory should support synchronous write and asynchronous read operation
    - Use dual_port_ram module provided in LAB folder which implements above mentioned requirement
  - Use M-bit wide and N-Stage deep shift register for write and read pointer synchronization
    - Use shift_register module provided in LAB folder which can take M-bit of data and generate N-cycles delayed version of input data
  - Synthesize async fifo RTL model and run simulation using testbench provided
  - Review synthesis results (resource usage and RTL netlist/schematic)
  - Review input and output signals in simulation waveform.
  - Describe simulation behavior, FIFO operation and provide simulation snapshot, RTL code, resource estimation in final report

Homework Assignment-9

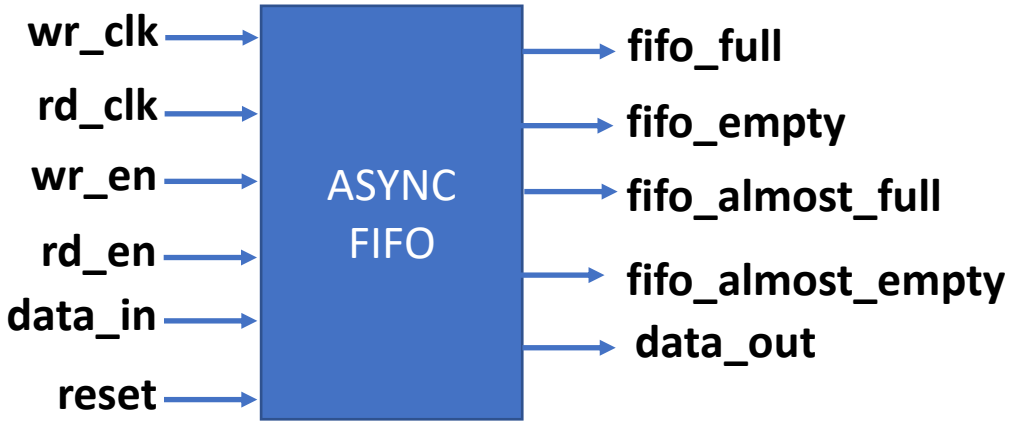
❑ Primary Ports for Asynchronous FIFO design :

- **input** logic **wr_clk, rd_clk** : Write and Read Clocks
- **input** logic **reset** : Asynchronous and active high reset
- **input** logic **wr_en** : write enable, if wr_en == 1, data gets written to FIFO Memory
- **input** logic **rd_en** : read_enable, if rd_en == 1, data gets read out from FIFO Memory
- **input** logic [DATA_WIDTH-1:0] **data_in** : input data to be written to FIFO Memory
- **output** logic [DATA_WIDTH-1:0] **data_out** : data read out from FIFO Memory
- **output** logic **fifo_full** : indicates FIFO is full and there are no locations inside FIFO memory for further writes
- **output** logic **fifo_empty** : indicates FIFO is empty and there are no data available inside FIFO memory for reading
- **output** logic **fifo_almost_full** : one cycle early indication of FIFO_FULL (fifo is not full yet, it will be next cycle)
- **output** logic **fifo_almost_empty** : one cycle early indication of FIFO_EMPTY (fifo is not empty yet, it will be next cycle)

❑ Name of the asynchronous fifo module : async_fifo

❑ Use below mentioned modules provided :

- **shift_register :**
  - To implement 2-FF synchronizer for write and read pointer
  - To implement 1 bit delayed version of early fifo empty and fifo full
- **dual_port_ram** :
  - For FIFO Memory Implementation



Homework Assignment-9

- **Note** : First simulate Synchronous FIFO design with testbench provided and review waveform and RTL code to understand Synchronous FIFO design

- And then, start implementing asynchronous fifo RTL model

- Lab folder has :
  - Synchronous FIFO full RTL model with testbench and
  - For asynchronous FIFO, template RTL model code with steps are provided including partial code to guide students on implementation.

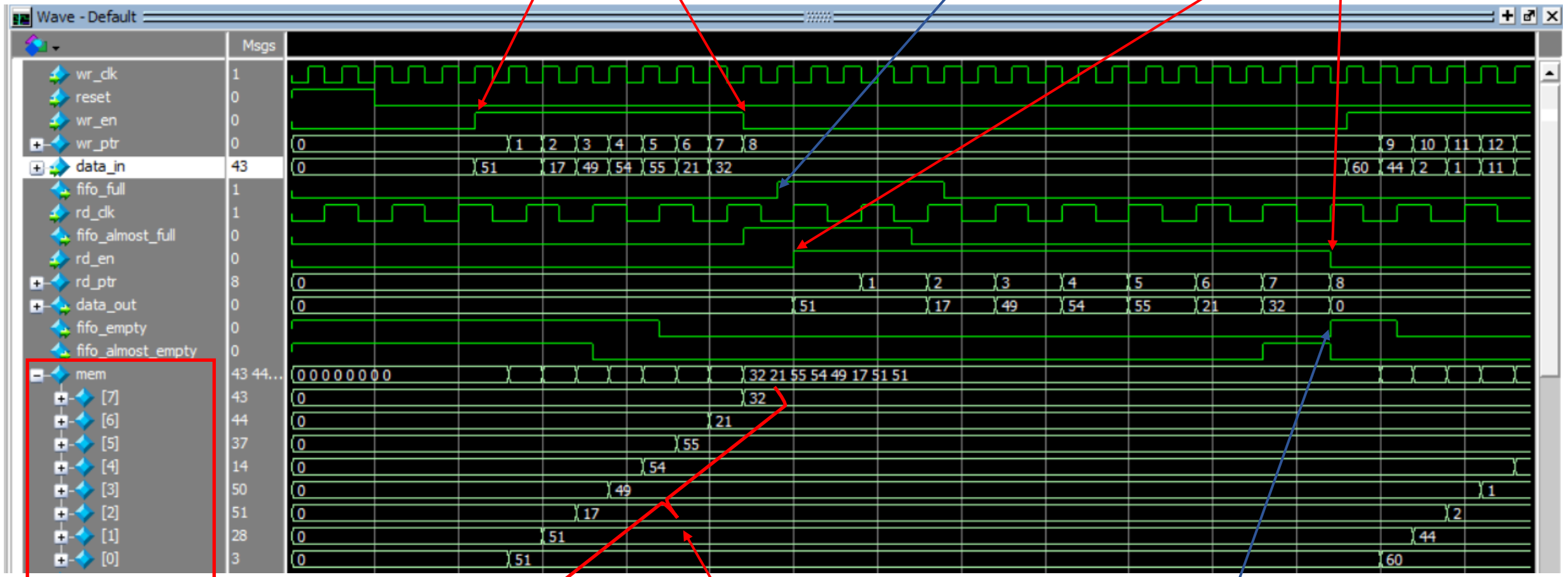
- **Note :**
  - **HW9 submission is optional and this will not have impact to final grade !**

Asynchronous FIFO Simulation Snapshot -1

when wr_en==1 and fifo_full == 0
datain values : 51, 17, 49, 54, 55, 21,32 are stored in FIFO Memory

After writing 8 data elements
fifo_full = 1 is set

when rd_en==1 and fifo_empty == 0
datain values : 51, 17, 49, 54, 55, 21,32 are read out from FIFO Memory



After reading all 8 data elements fifo_empty = 1 is set

FIFO Memory (mem) has datain values : 51, 17, 49, 54, 55, 21,32. Each rd_en=1 caused data to push into FIFO memory

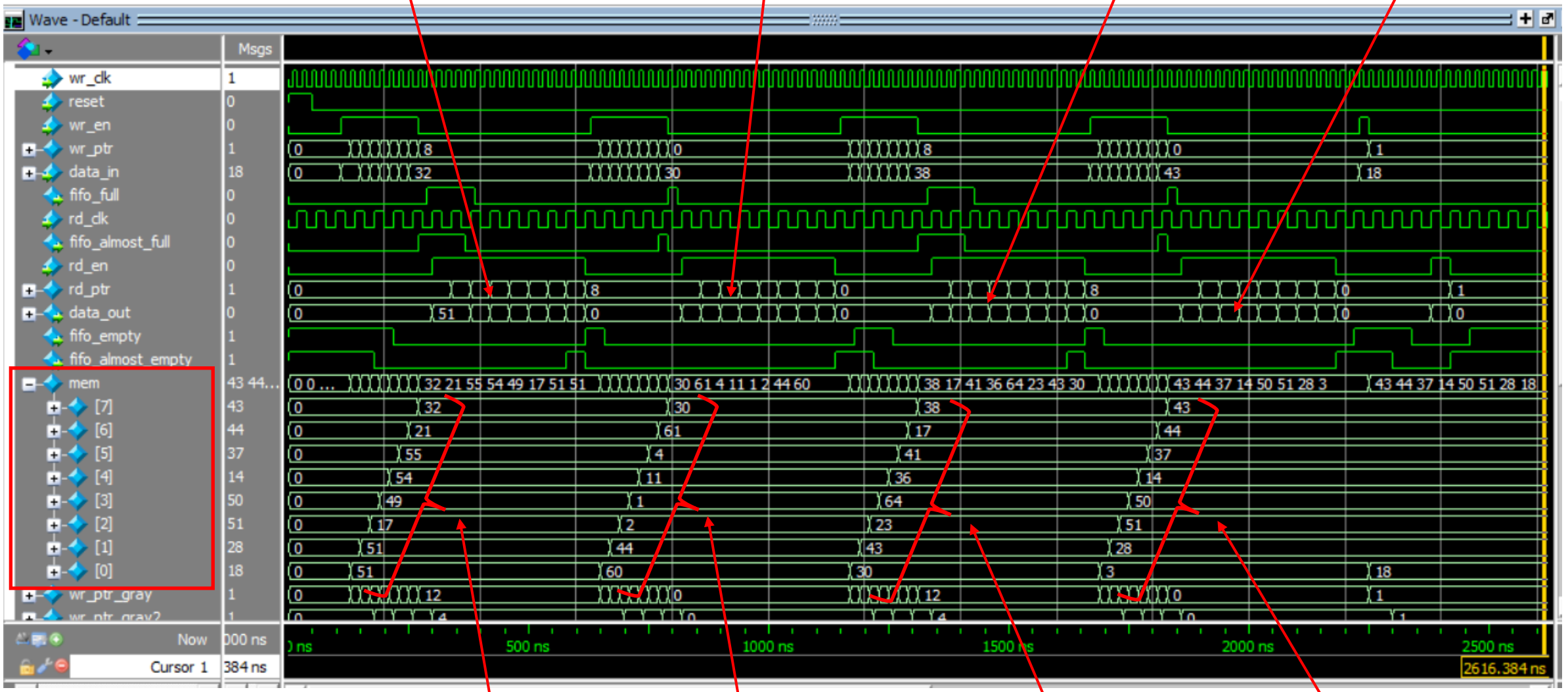
Asynchronous FIFO Simulation Snapshot -2

first set of 8 data values popped out of FIFO memory

second set of 8 data values popped out of FIFO memory

third set of 8 data values popped out of FIFO memory

Fourth set of 8 data values popped out of FIFO memory



first set of 8 datain values pushed in FIFO memory

Second set of 8 datain values pushed in FIFO memory

Thrid set of 8 datain values pushed in FIFO memory

Fourth set of 8 datain values pushed in FIFO memory

Steps to Model Asynchronous FIFO

- **Step-1** : Increment write pointer each time wr_en is '1' (binary counter)
- **Step-2 :** Convert write pointer to gray value from binary write pointer value before sending write pointer to rd_clk domain through 2-FlipFlip synchronizer
- **Step-3 :** Increment read pointer each time rd_en is '1' (binary counter)
- **Step-4** : Convert read pointer to gray value from binary read pointer value before sending read pointer to wr_clk domain through 2-FlipFlip synchronizer
- **Step-5 :** Generate fifo empty flag using synchronized write pointer and original read pointer value comparison. Before comparison convert back synchronized write pointer gray value to binary value.
  - The FIFO is empty when both read and write pointers point to the same location
  - assign t_fifo_empty = (wr_ptr_binary2 == rd_ptr) ? 1 : 0;
- **Step-6 :** Assert output signal fifo_almost_empty
  - FIFO Almost empty is generated simultaneously when the very last data available in fifo is read hence it is named as fifo almost empty. In another words, one cycle before fifo is actually empty
  - assign fifo_almost_empty = t_fifo_empty;

Steps to Model Asynchronous FIFO

- **Step-7 :** Generate fifo full flag. . Before comparison convert back synchronized read pointer gray value to binary value.
  - FIFO is full when wr_ptr - rd_ptr = 2^address_width.
  - In that case, the Lower address bits are identical, but the MSB address bit is different.
  - assign t_fifo_full = ((wr_ptr[ADDR_WIDTH] != rd_ptr_binary2[ADDR_WIDTH])
  - && (wr_ptr[ADDR_WIDTH-1:0] == rd_ptr_binary2[ADDR_WIDTH-1:0])) ? 1 :
- **Step-8 :** Assert almost full flag
  - FIFO Almost full is generated simultaneously when the very last location in fifo is written with data hence it is named as fifo almost full. In another words, one cycle before the fifo is actually full
  - assign fifo_almost_full = t_fifo_full;
- **Step-9 :** Instantiate or create dual port single clock FIFO Memory
  - Memory should support synchronous write and asynchronous read
  - when connecting below wr_en and rd_end remember to do logical-and with !fifo_full and !fifo_empty respectively. This is to prevent FIFO overflow and underflow conditions
  - Dual port ram should support perform simultaneous write and read operations.

Steps to Model Asynchronous FIFO

- **Step-10** : Synchronize write pointer gray value to read clock domain using 2-FF synchronizer.
  - This is done to synchronize wr_ptr to rd_clk domain, and in rd_clk domain, output of this synchronizer will be used to compute fifo empty flag.
- **Step-11 :** Convert synchronized write pointer gray value available from Step-10, back to binary value
  - Prior to generation for fifo empty flag, synchronized gray write pointer value is converted first to binary write pointer value
  - assign wr_ptr_binary2 = gray_to_binary(wr_ptr_gray2);
- **Step-12 :** Synchronize read pointer gray value to write clock domain using 2-FF synchronizer
  - This is done to synchronize rd_ptr to wr_clk domain, and in wr_clk domain, output of this synchronizer will be used to compute fifo full flag.
- **Step-13 :** Convert synchronized read pointer gray value available from Step-12, back to binary value
  - Prior to generation for fifo full flag, synchronized gray read pointer value is converted first to binary read pointer value
  - assign rd_ptr_binary2 = gray_to_binary(rd_ptr_gray2);

Steps to Model Asynchronous FIFO

- **Step-14 :** Delay fifo almost empty (t_fifo_empty) by 1 clock cycle to generate fifo_empty output signal
  - This is done to generate fifo_empty output signal after the last available data in FIFO is read out
- **Step-15 :** Delay fifo almost full (t_fifo_full) by 1 clock cycle to generate fifo_full output signal
  - This is done to generate fifo_full output signal after the last available location in FIFO is written with a data