FIFO

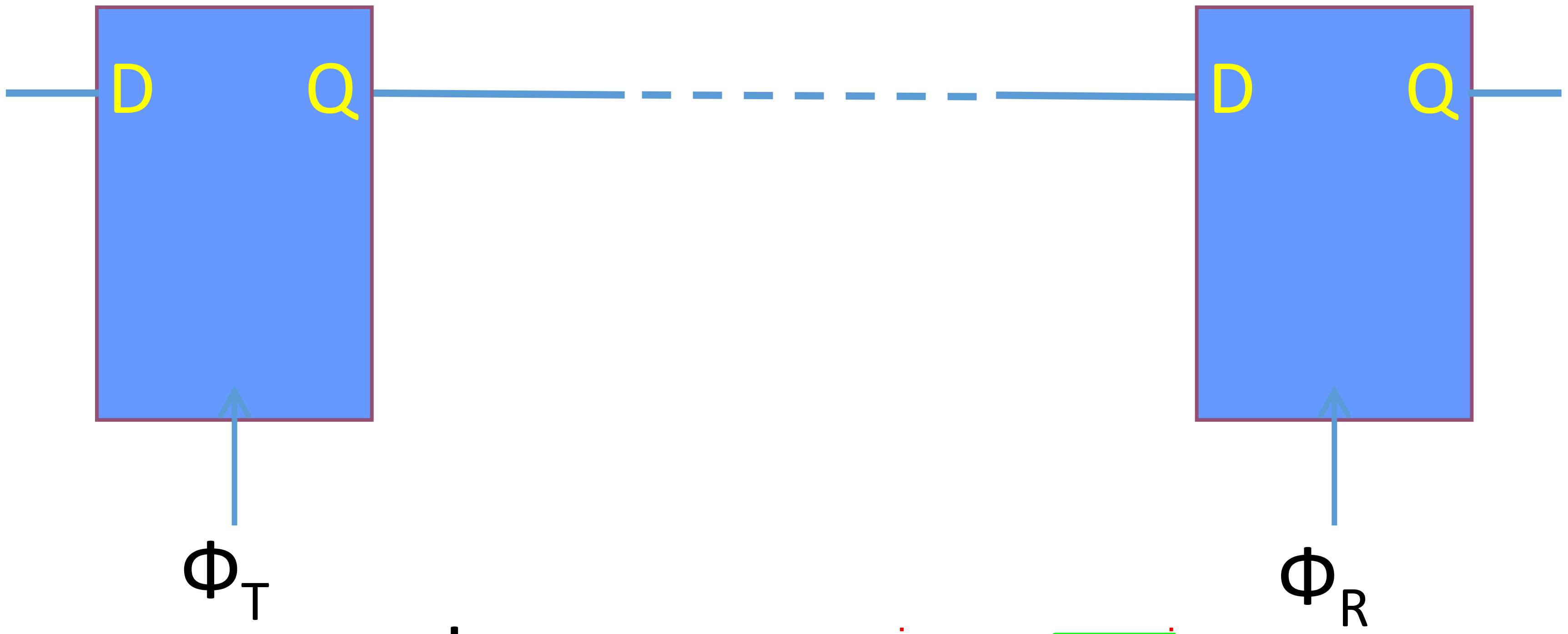
References

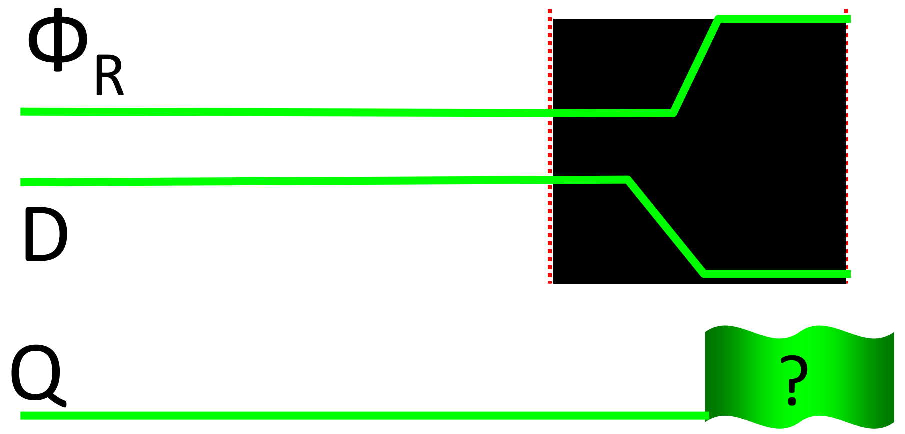
- FIFO Architecture, Functions, and Applications by Texas Instruments
- Understanding Metastability in FPGAs, White Paper, Altera
- Simulation and Synthesis Techniques for Asynchronous FIFO Design by Clifford E. Cummings, Sunburst Design, Inc.
- Case Study: FIFO Design by Vaibbhav T

Introduction

- System On Chip with multi clocks
- Possible metastability
- Metastability is a phenomenon that can cause system failure in digital devices

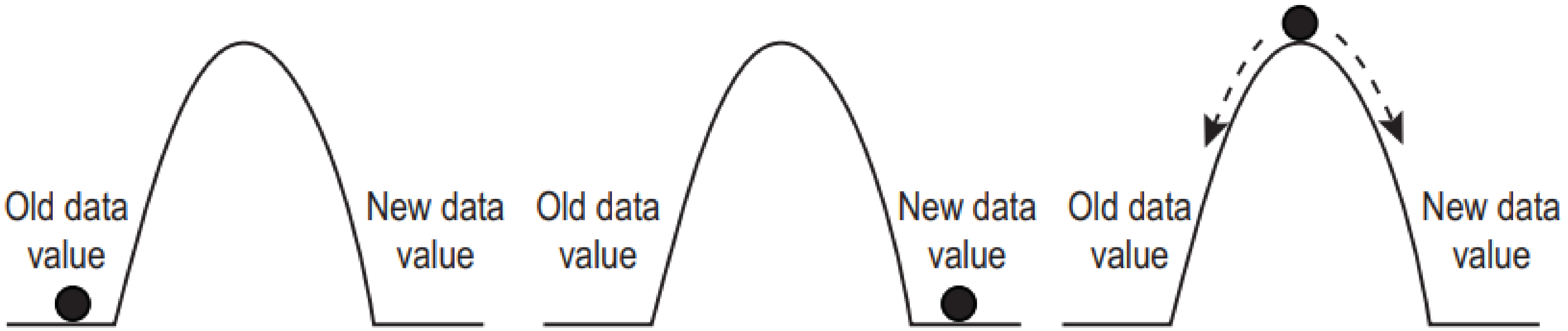
The problem: metastability





The problem: metastability

Figure 1. Metastability Illustrated as a Ball Dropped on a Hill



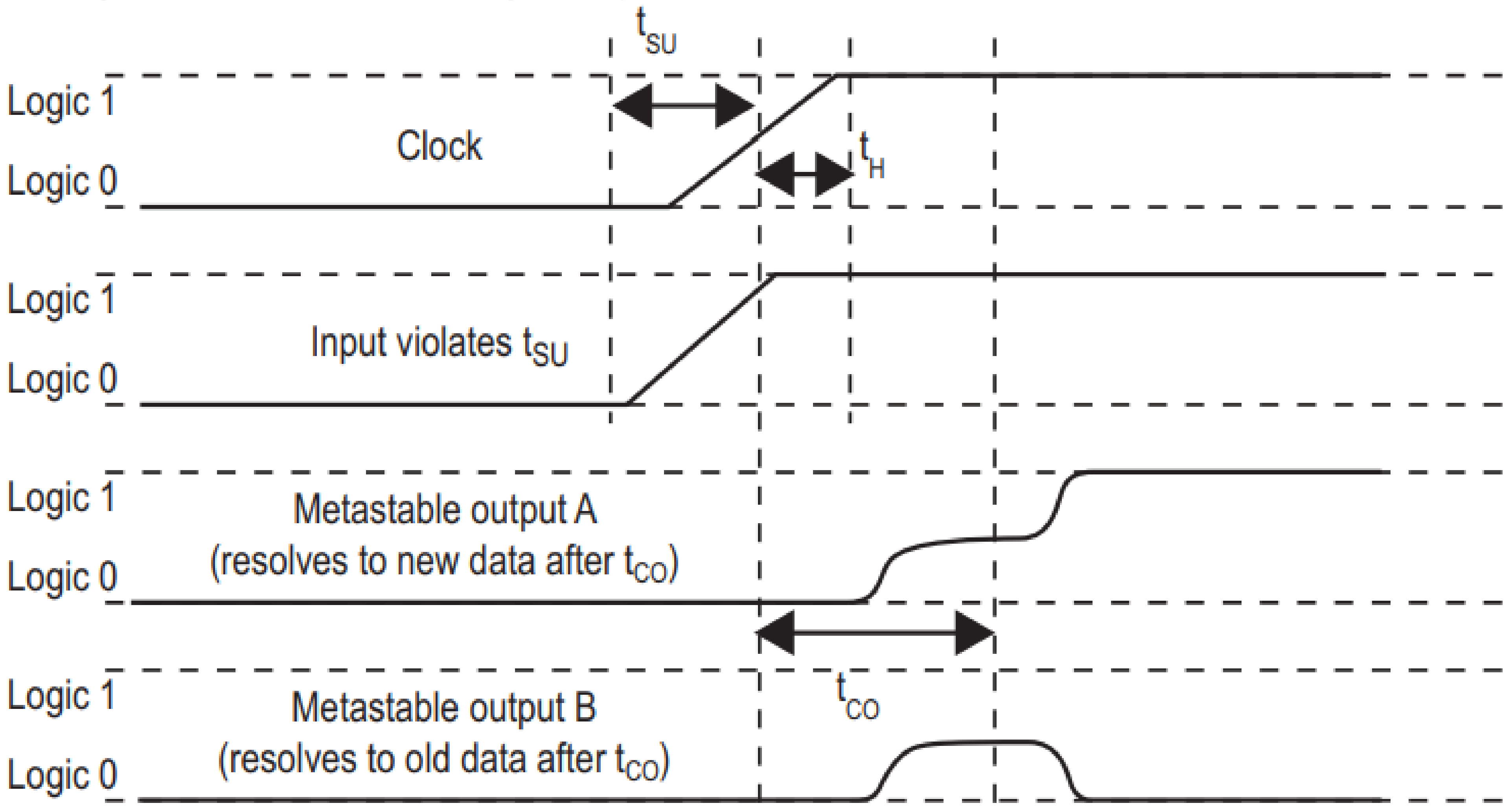
Signal transition occurs after clock edge and minimum t_H: Ball lands on the old data side.

Signal transition meets register t_{SU} and t_H: Ball lands on the new data side.

Signal violates register t_{SU} or t_H: Ball balances at top of hill or takes too long to reach the bottom. Output is metastable and violates t_{CO}.

The problem: metastability

Figure 2. Examples of Metastable Output Signals



When Does Metastability Cause Design Failures?

- If the data output signal resolves to a valid state before the next register captures the data, then the metastable signal does not negatively impact the system operation.

- But if the metastable signal does not resolve to a low or high state before it reaches the next design register, it can cause the system to fail.

- Continuing the ball and hill analogy, failure can occur when the time it takes for the ball to reach the bottom of the hill (a stable logic value 0 or 1) exceeds the allotted time, which is the register's t_{CO} plus any timing slack in the path from the register.

- When a metastable signal does not resolve in the allotted time, a logic failure can result if the destination logic observes inconsistent logic states, that is, different destination registers capture different values for the metastable signal.

Synchronization Registers

- When a signal is transferred between circuitry in unrelated or asynchronous clock domains, it is necessary to synchronize this signal to the new clock domain before it can be used.
- The first register in the new clock domain acts as a synchronization register
- To minimize the failures due to metastability in asynchronous signal transfers, circuit designers typically use a sequence of registers (a synchronization register chain or synchronizer) in the destination clock domain to resynchronize the signal to the new clock domain.
- These registers allow additional time for a potentially metastable signal to resolve to a known value before the signal is used in the rest of the design. .
- The timing slack available in the synchronizer register-to-register paths is the time available for a metastable signal to settle, and is known as the available metastability settling time

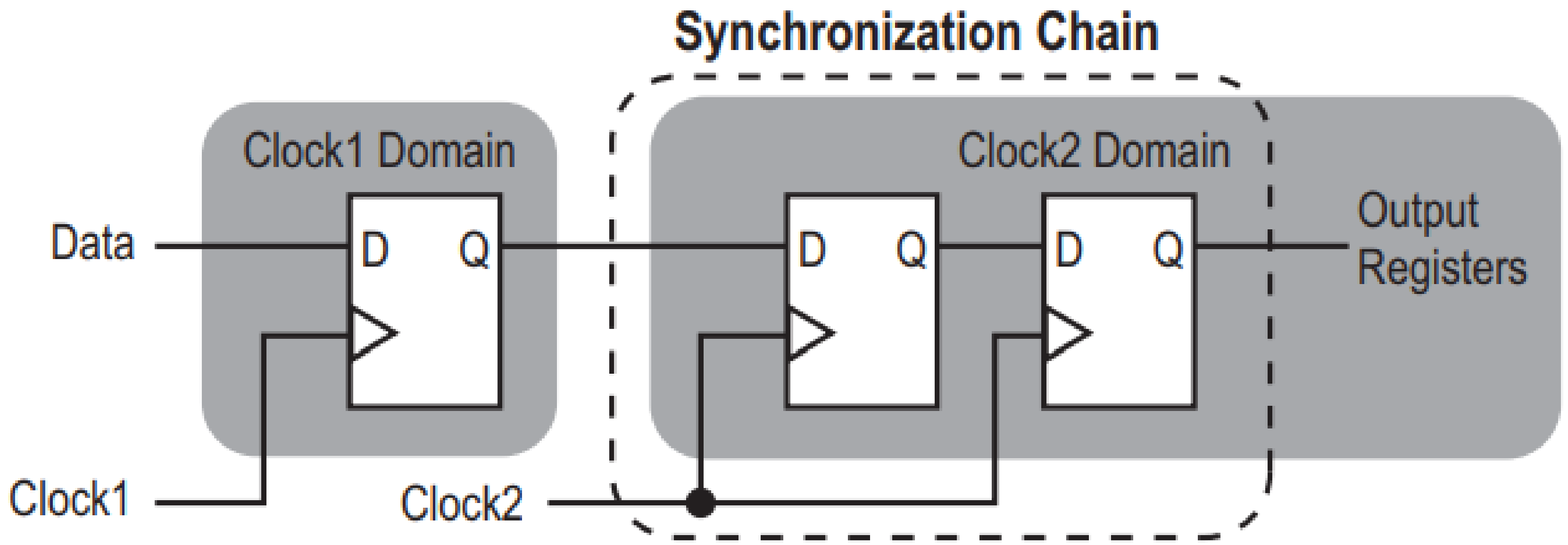
Synchronization Registers

- A synchronization register chain, or synchronizer, is defined as a sequence of registers that meets the following requirements:
- The registers in the chain are all clocked by the same or phase-related clocks
- The first register in the chain is driven from an unrelated or asynchronous clock domain
- Each register fans out to only one register, except the last register in the chain

Synchronization Registers

- The length of the synchronization register chain is the number of registers in the synchronizing clock domain that meet the above requirements

Figure 3. Sample Synchronization Register Chain



Synchronization Registers

- Signals that transfer between unrelated clock domains, can transition at any point relative to the clock edge of the capturing register.
- Therefore, the designer cannot predict the sequence of a signal’s transitions or the number of destination clock edges until the data transitions.
- FIFO logic uses synchronizers to transmit control signals between the two clock domains, then data is written and read with dual-port memory

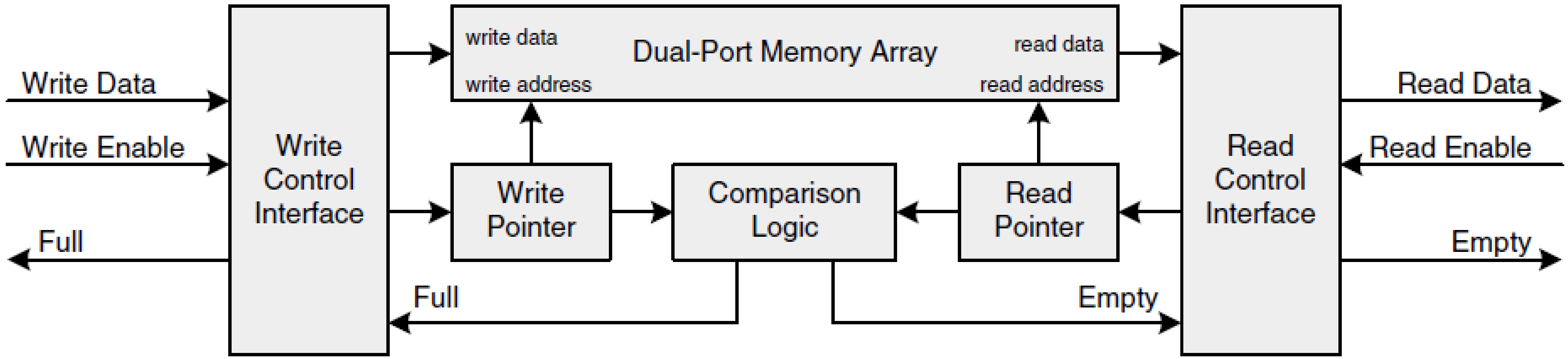
FIFO

- First-in-first-out (FIFO) memories are special-purpose devices that implement a basic queue structure
- Unlike other memory devices, a typical FIFO has two unidirectional ports without address inputs: one for writing and another for reading.
- As the name implies, the first data written is the first read, and the last data written is the last read.
- A FIFO is not a random access memory but a sequential access memory.
  - Unlike a conventional memory, once a data element has been read once, it cannot be read again, because the next read will return the next data element written to the FIFO.
  - By their nature, FIFOs are subject to overflow and underflow conditions.
  - Their finite size, often referred to as depth
  - An overflow occurs when an attempt is made to write new data to a full FIFO.
  - An empty FIFO has no data to provide on a read request, which results in an underflow.

Basic FIFO Architecture

- A FIFO is not addressed in a linear fashion; rather, it is made to form a continuous ring of memory that is addressed by the two internal pointers.
- The fullness of the FIFO is determined not by the absolute values of the pointers but by their relative values.
- An empty FIFO begins with its read and write pointers set to the same value.
- As entries are written, the write pointer increments.
- As entries are read, the read pointer increments as well.
- If the read pointer ever catches up to the write pointer such that the two match, the FIFO is empty again.
- If the read pointer fails to advance, the write pointer will eventually wraps around the end of the memory array and become equal to the read pointer.
- At this point, the FIFO is full and cannot accept any more data until reading resumes.
- Full and empty flags are generated by the FIFO to provide status to the writing and reading logic.
- Some FIFOs contain more detailed fullness status, such as signals that represent programmable fullness thresholds.

Basic FIFO Architecture

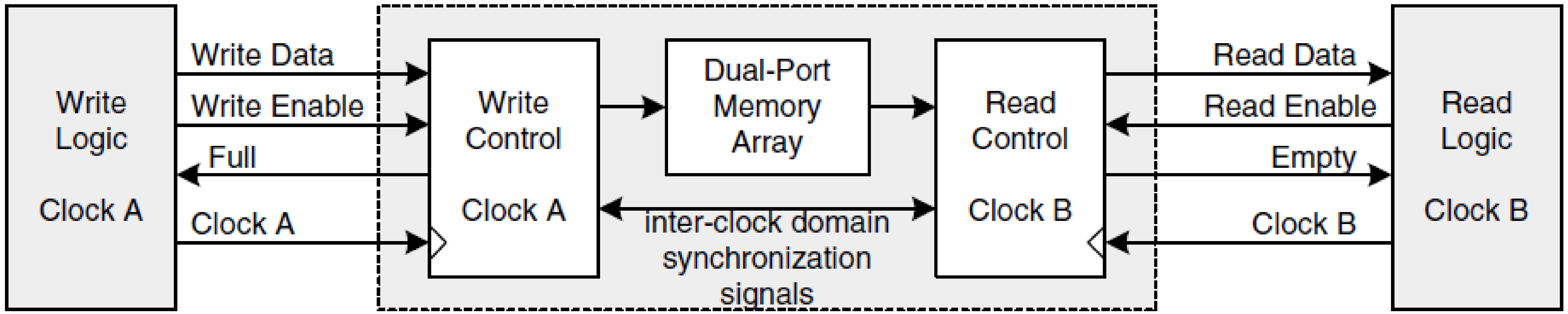


- A FIFO is created by surrounding a dual-port memory array—generally SRAM, but DRAM can also be used—with a write pointer, a read pointer, and control logic
- A FIFO can be synchronous or asynchronous

Basic FIFO Architecture

- One common role that a FIFO fills is in clock domain crossing.
- In such an application, there is a need to communicate a series of data values from a block of logic operating on one clock to another block operating on a different clock.
- Exchanging data between clock domains requires special attention, because there is normally no way to perform a conventional timing analysis across two different clocks to guarantee adequate setup and hold times at the destination flops.
- Either an asynchronous FIFO or a dual-clock synchronous FIFO can be used to solve this problem

Basic FIFO Architecture



Basic FIFO Architecture

- The dual-port memory at the heart of the FIFO is an asynchronous element that can be accessed by the logic operating in either clock domain.
- A dual-clock synchronous FIFO is designed to handle arbitrary differences in the clocks between the two halves of the device.
- When one or more bytes are written on clock A, the write-pointer information is carried across to the clock B domain within the FIFO via inter-clock domain synchronization logic.
- This enables the read-control interface to determine that there is data waiting to be read.
- Logic on clock B can read this data long after it has been safely written into the memory array and allowed to settle to a stable state.

Asynchronous FIFO Design

- Data values are written to a FIFO buffer from one clock domain and the data values are read from the same FIFO buffer from another clock domain, where the two clock domains are asynchronous to each other
- Used to safely pass data from one clock domain to another asynchronous clock domain

- **Synchronous FIFO Pointer Design**
  - Write and reads from the FIFO buffer are conducted in the same clock domain
  - The implementation counts,
  - o the number of writes to FIFO buffer to increment (on FIFO write but no read)
  - o the number of reads to FIFO buffer to Decrement (on FIFO read but no write)
  - Hold (no independent and simultaneous writes/reads) the current fill value of the FIFO buffer.
  - The FIFO is full when the FIFO counter reaches a predetermined full value and the FIFO is empty when the FIFO counter is zero.

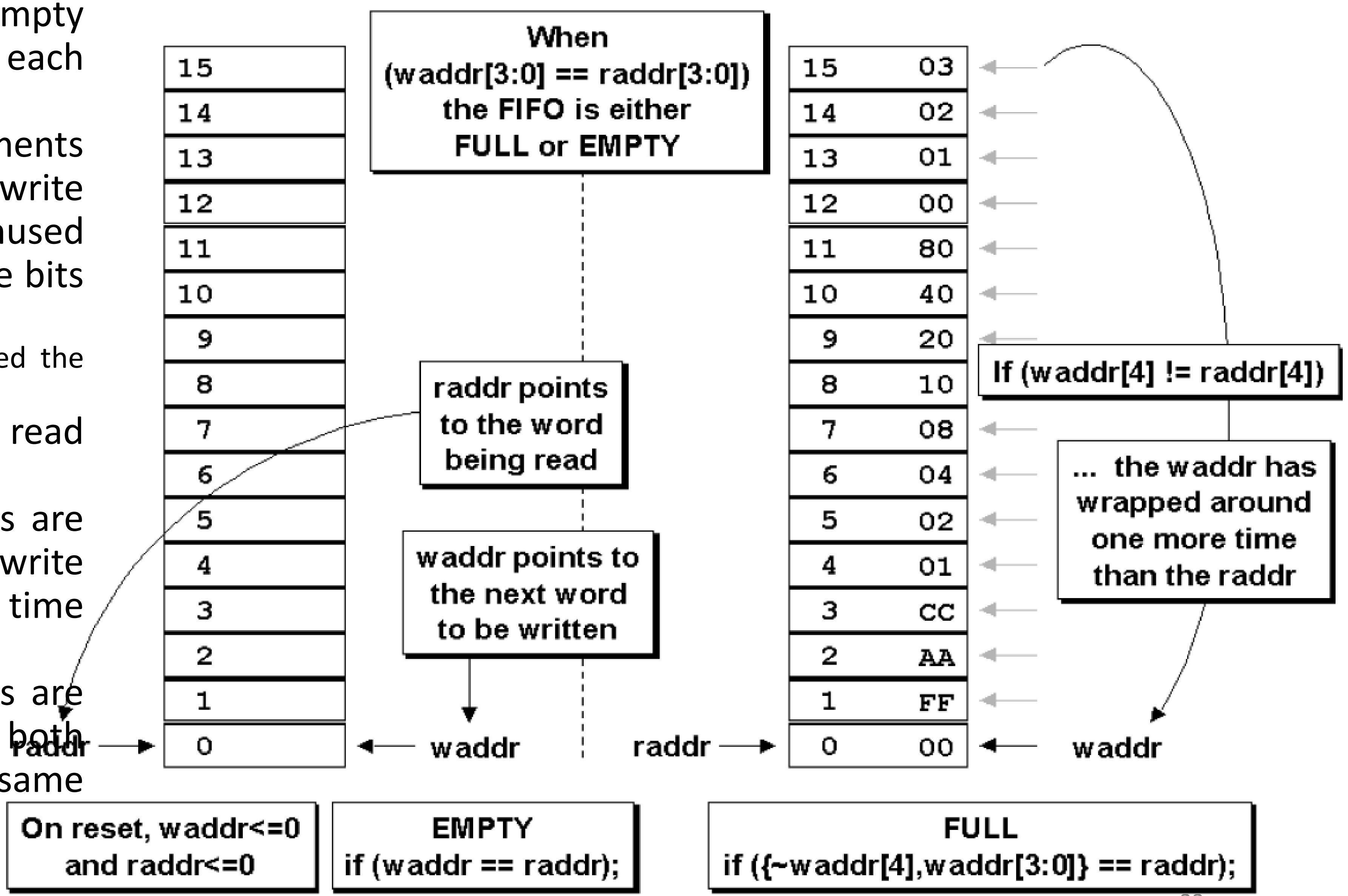
Asynchronous FIFO Design

Asynchronous FIFO pointers

- The write pointer always points to the next word to be written
- On a FIFO-write operation, the memory location that is pointed to by the write pointer is written, and then the write pointer is incremented to point to the next location to be written
- Read pointer always points to the current FIFO word to be read
- On reset, both pointers are reset to zero
- The FIFO is empty when the read and write pointers are both equal.
- This condition happens when both pointers are reset to zero during a reset operation, or when the read pointer catches up to the write pointer, having read the last word from the FIFO.
- A FIFO is full when the pointers are again equal, that is, when the write pointer has wrapped around and caught up to the read pointer.
- Problem! - The FIFO is either empty or full when the pointers are equal

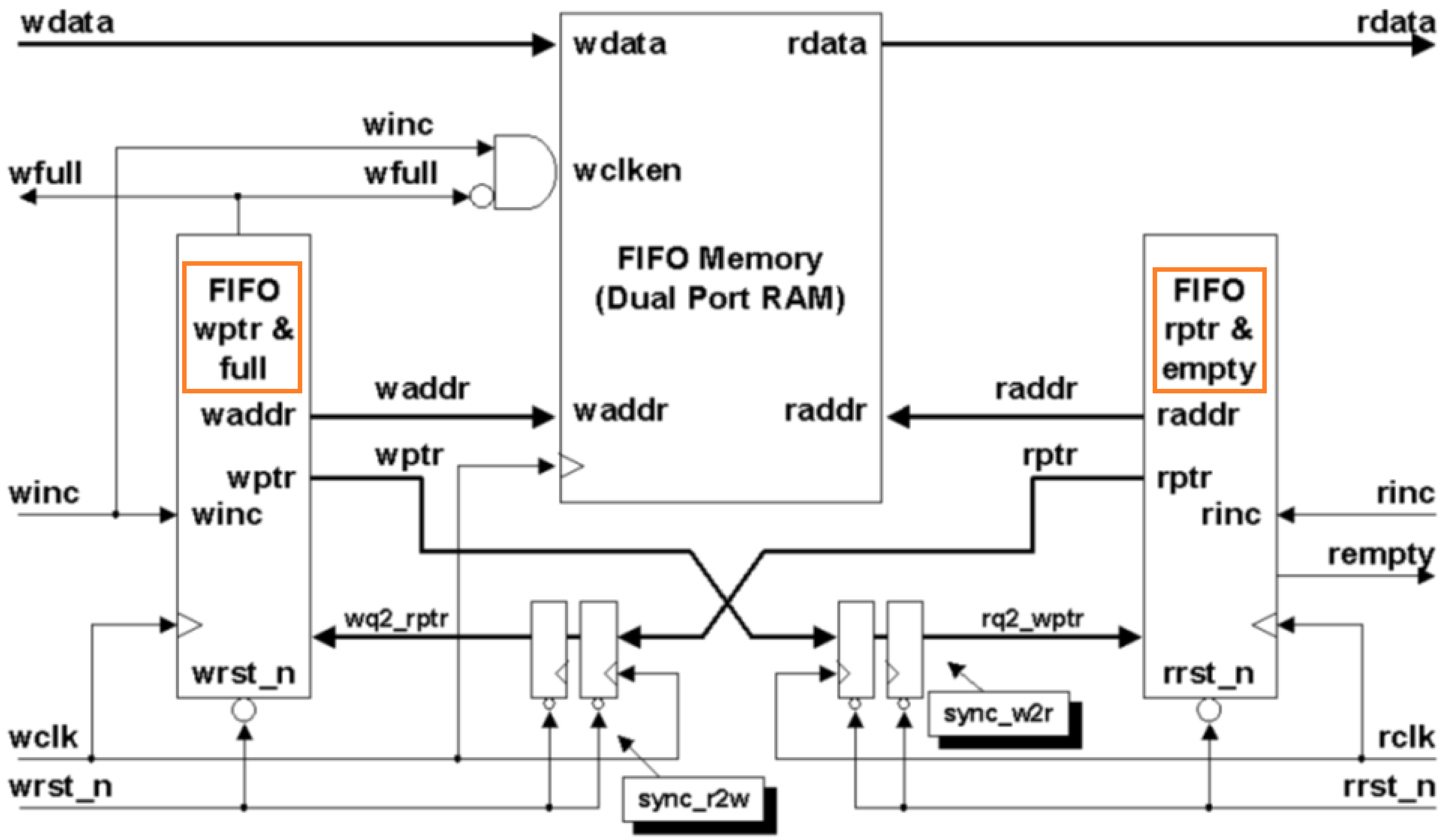
Asynchronous FIFO Design

- Distinguish between full and empty by adding an extra bit to each pointer.
- When the write pointer increments past the final FIFO address, the write pointer will increment the unused MSB while setting the rest of the bits back to zero
  - The FIFO has wrapped and toggled the pointer MSB.
- The same is done with the read pointer.
- If the MSBs of the two pointers are different, it means that the write pointer has wrapped one more time that the read pointer.
- If the MSBs of the two pointers are the same, it means that both pointers have wrapped the same number of times



When (waddr[3:0] == raddr[3:0]) the FIFO is either FULL or EMPTY
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
raddr
raddr points to the word being read
waddr points to the next word to be written
waddr
raddr
15 03
14 02
13 01
12 00
11 80
10 40
9 20
8 10
7 08
6 04
5 02
4 01
3 CC
2 AA
1 FF
0 00
waddr
If (waddr[4] != raddr[4])
... the waddr has wrapped around one more time than the raddr
On reset, waddr<=0 and raddr<=0
EMPTY if (waddr == raddr);
FULL if ({~waddr[4],waddr[3:0]} == raddr);

Asynchronous FIFO Design



Calculation of FIFO Depth

<u>Case – 1</u> : $f_A > f_B$ with no idle cycles in both write and read.

Writing frequency = f_A = 80MHz.
Reading Frequency = f_B = 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.

<u>Sol</u> :

- ✓ Time required to write one data item = $\frac{1}{80\ MHz}$ = 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 = $\frac{1}{50\ MHz}$ = 20 nSec.
- ✓ So, for every 20 nSec, the module B is going to read one data in the burst.
- ✓ So, 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 = $\left(\frac{1500\ nSec}{20\ nSec}\right) = 75$
- ✓ The remaining no. of bytes to be stored in the FIFO = 120 – 75 = 45.
- ✓ So, the FIFO which has to be in this scenario must be capable of storing 45 data items.

So, the minimum depth of the FIFO should be 45.

Calculation of FIFO Depth

Case – 2 : $f_A > f_B$ with one clk cycle delay between two successive reads and writes.

- Scenario is no way different from the previous scenario, because, there will be one clock cycle delay between two successive reads and writes.
- So, the approach is same as the earlier one

Calculation of FIFO Depth

Case – 3 : $f_A > f_B$ with idle cycles in both write and read.

Writing frequency = f_A = 80MHz.
Reading Frequency = f_B = 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 reads is = 3.

Sol :

- ✓ 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 **two** 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 **four** clock cycles, one data is read.
- ✓ Time required to write one data item = $2 * \frac{1}{80\,MHz}$ = 25 nSec.
- ✓ Time required to write all the data in the burst = 120 * 25 nSec. = 3000 nSec.
- ✓ Time required to read one data item = $4 * \frac{1}{50\,MHz}$ = 80 nSec.