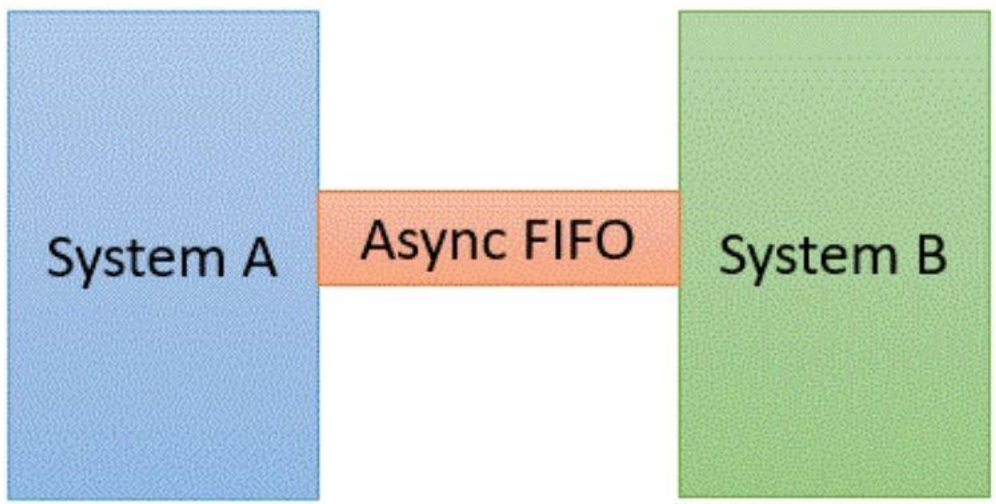
ASYNCHRONOUS FIFO

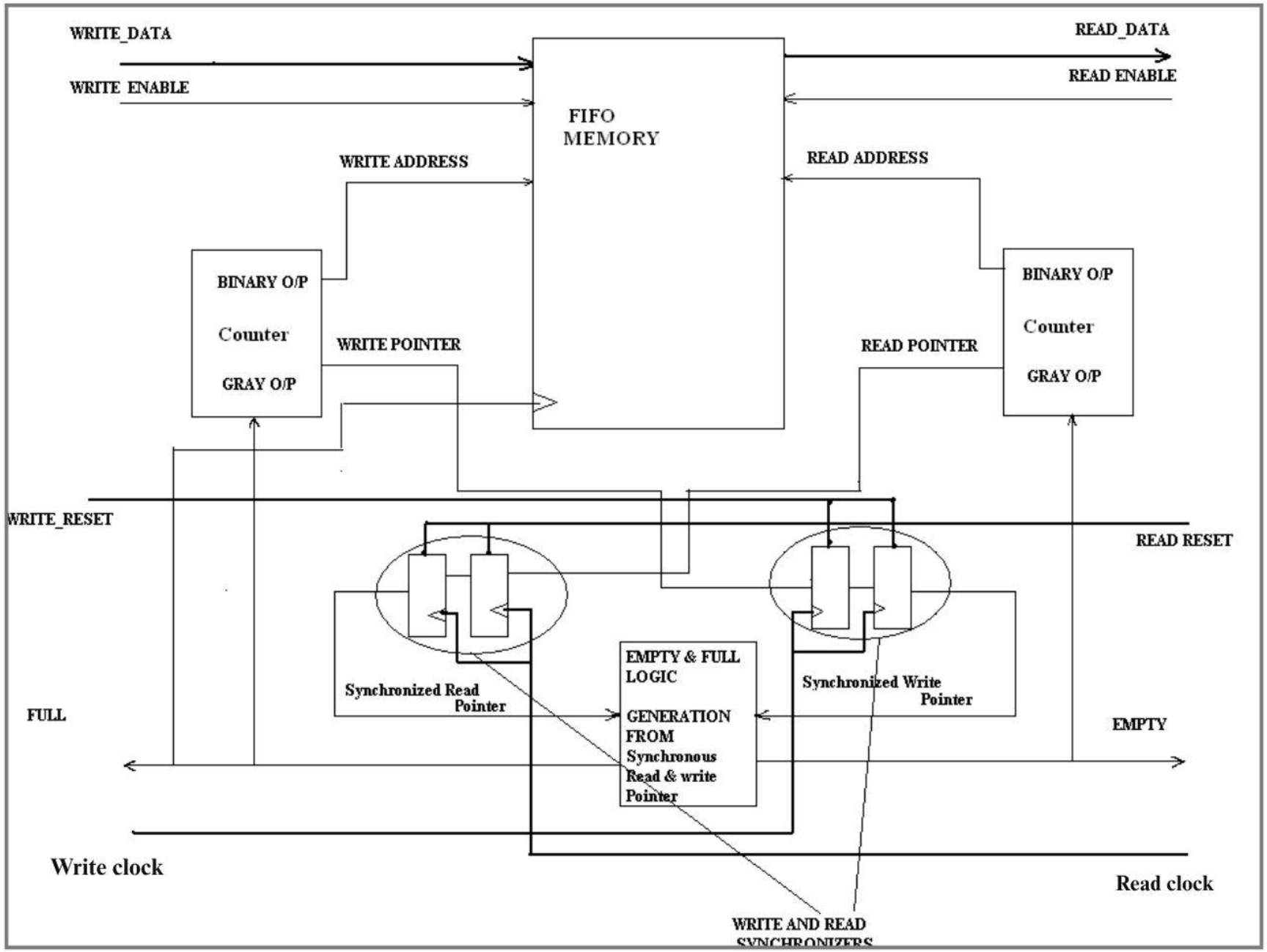
Aarti Kumari

What is Asynchronous FIFO ?

- In asynchronous FIFO, data read and write operations use different clock frequencies.
- Since write and read clocks are not synchronized, it is referred to as asynchronous FIFO
- Usually, these are used in systems where data need to pass from one clock domain to another which is generally termed as ‘clock domain crossing’.
- Thus, asynchronous FIFO helps to synchronize data flow between two systems working on different clocks.



Block diagram:



Main Modules:

- **FIFO Memory**: The heart of the system, storing the data.
- **Binary and Gray Counters**: Essential for tracking read and write pointers, with Gray code used to minimize metastability issues when crossing clock domains.
- **Synchronizers**: Critical circuits that safely transfer the read and write pointers between the asynchronous clock domains. They mitigate the risk of metastability.
- **Empty and Full Logic**: Detecting when the FIFO is empty or full is vital to prevent underflow and overflow errors. This logic is carefully designed considering the asynchronous nature of the signals.

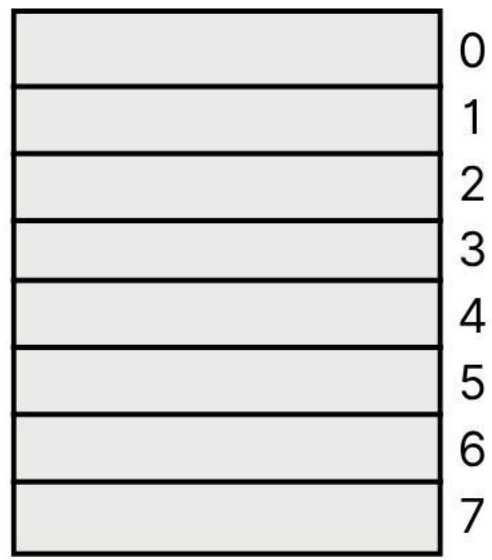
FIFO memory:

This is the heart of the FIFO.
The depth of memory is 8 bits and width is 8 bits.

It has following **inputs**:
Write Data (8 bit),
Write Enable,
Read Enable,
Write Clock,
Write address (4 bit),
Read Address (4 bit)

output : Read Data (8 bit)

DEPTH: Think of 8 memory location(rows)
WIDTH: Each location is 8 bits wide
Total memory capacity = Depth × Width
= 8bits × 8 bits = **64 bits** = 8 bytes



Binary and Gray pointers:

A binary pointer is used to address the FIFO memory, specifically for write and read operations.

- In the write and read logic, when you're accessing the memory (mem[addr]), you use a binary pointer (wptr, rptr).
- These are straightforward modulo pointers that increment with each successful write or read.

The Gray pointer is needed to address read and write pointers

- Synchronization and comparison for the EMPTY and FULL conditions

- In asynchronous FIFO, read and write are in separate clock domains.
- We need to synchronize pointers between these two domains.
- If we pass binary pointers across clock domains, there's a risk of:
    1. Metastability (unstable transitions)
    2. Glitches (e.g., 3 bits update at once → incorrect intermediate states)
- In Gray code, only one bit changes at a time during a count.
- This minimizes the chance of reading an invalid or unstable pointer value during synchronization.

Using Binary:

Decimal	Binary	After synch	Binary
5	1 0 1	101	5(d)
6	1 1 0	101 110 100 111	5(d) 6(d) 4(d) 7(d)

FULL

False condition

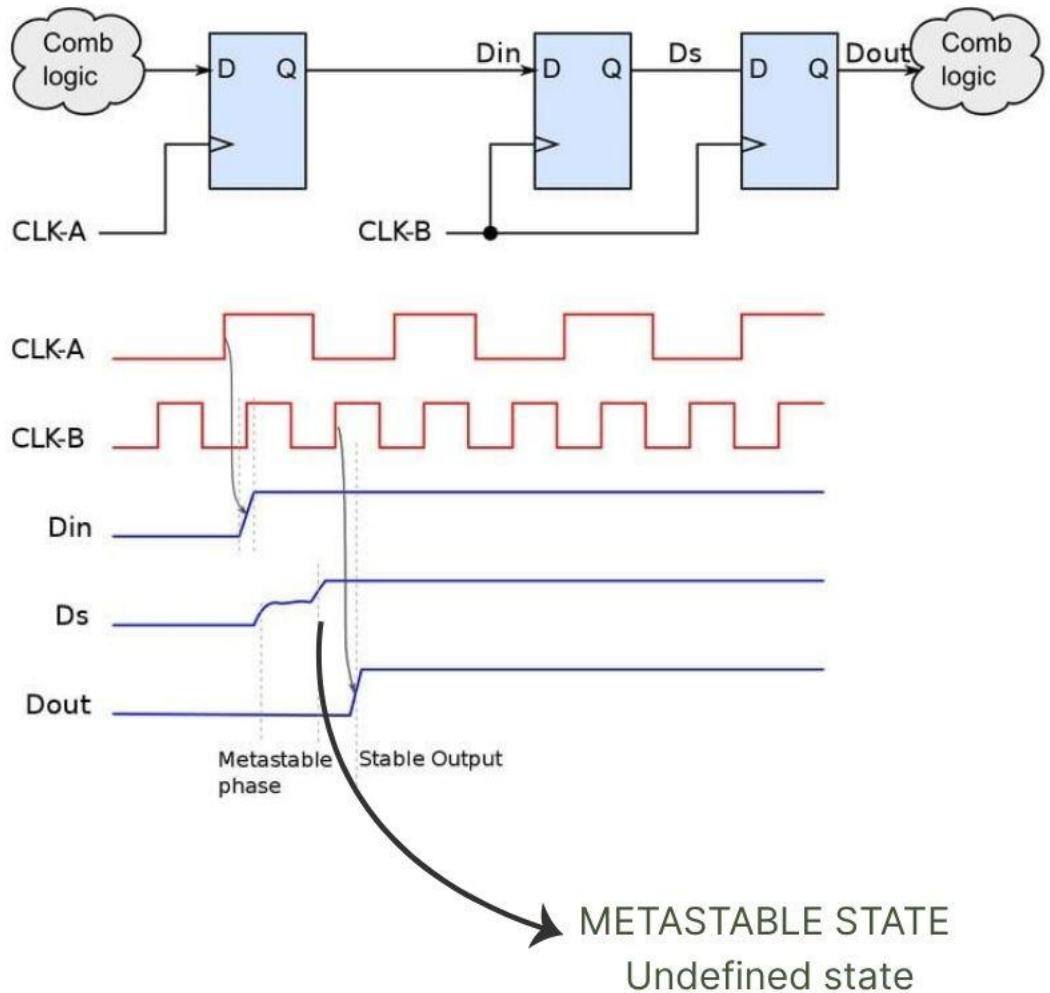
Using Gray:

Decimal	Binary	After synch	Binary
5	1 1 1	111	5(d)
6	1 0 1	111 101	5(d) 6(d)

No False condition

SYNCHRONIZER:

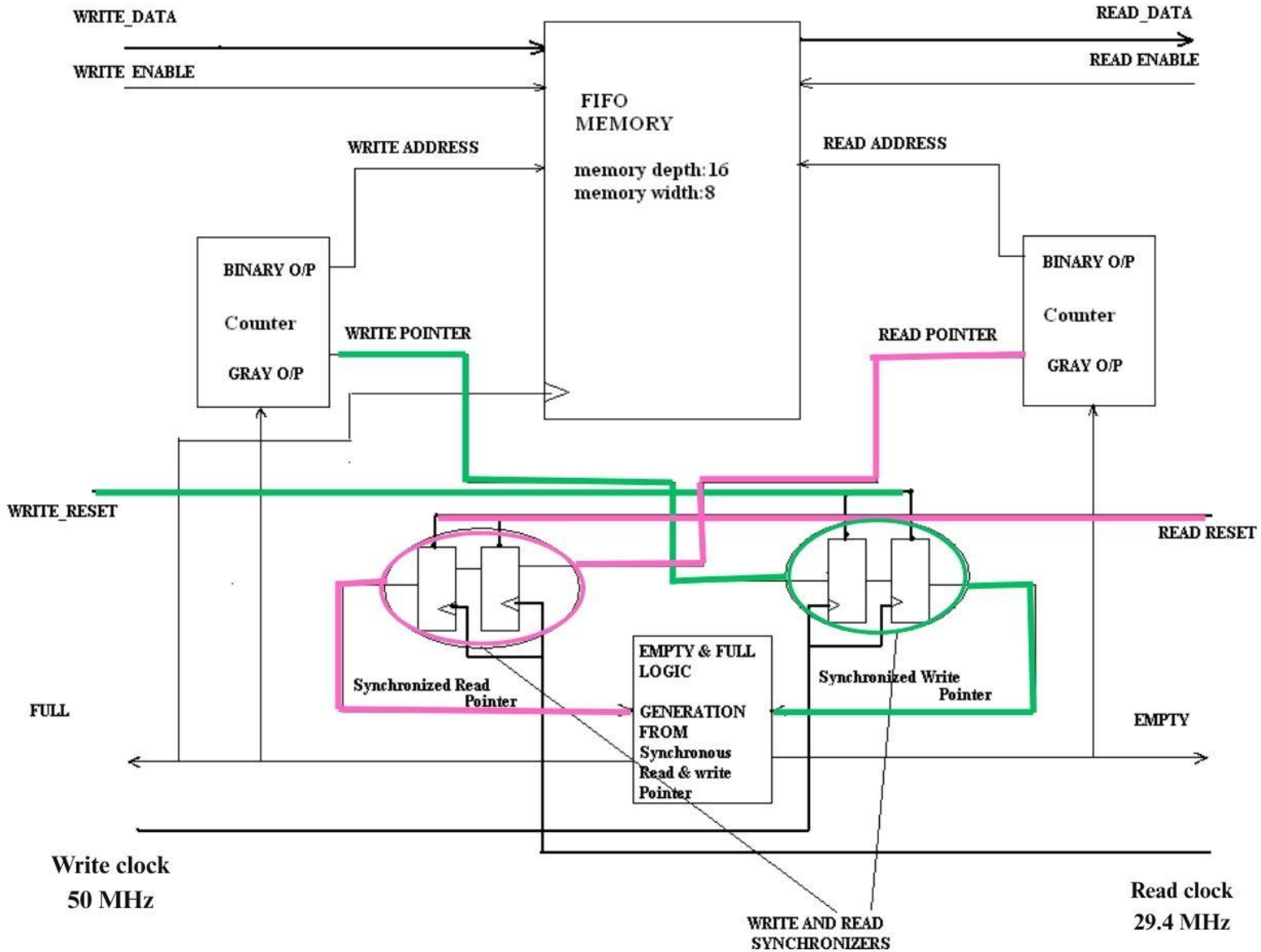
- Both flip-flops are clocked by the destination clock
- Reduces the chance of errors due to metastability



- 1st flip-flop might go into metastable state
- 2nd flip-flop likely gets clean, stable value

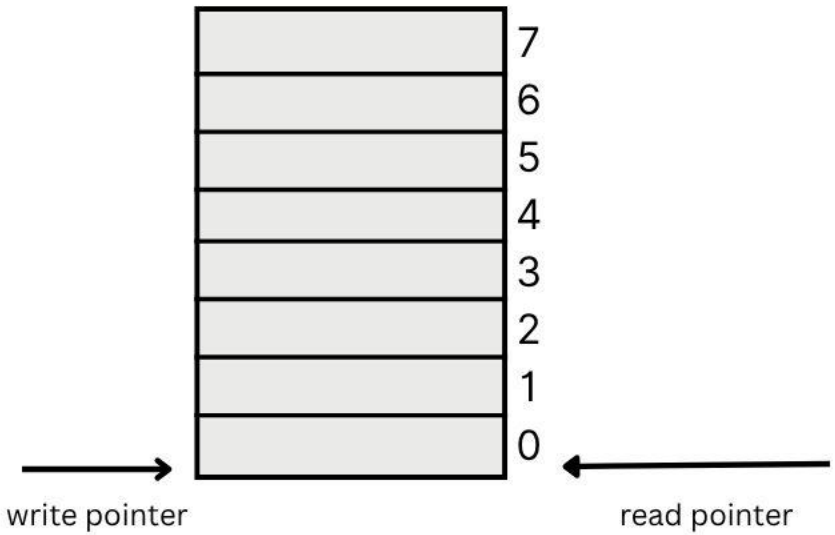
CROSS-CLOCK SYNCHRONIZATION

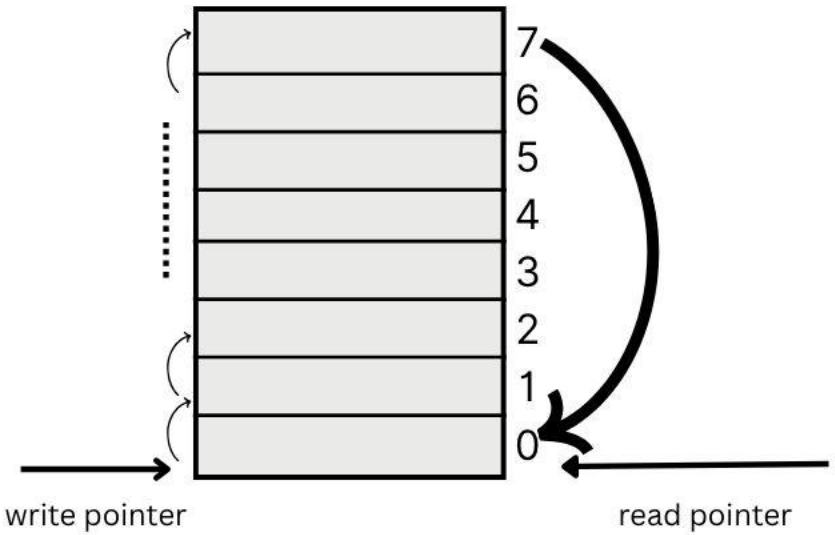
- Write clock domain receives synchronized Read Pointer (in Gray).
- Read clock domain receives synchronized Write Pointer (in Gray).
- This ensures safe comparison of pointers without metastability.
- These are used to generate:
  1. FULL flag in write domain.
  2. EMPTY flag in read domain.



EMPTY and FULL conditions:

For FIFO depth of 8, we need 3 bits for addressing





when

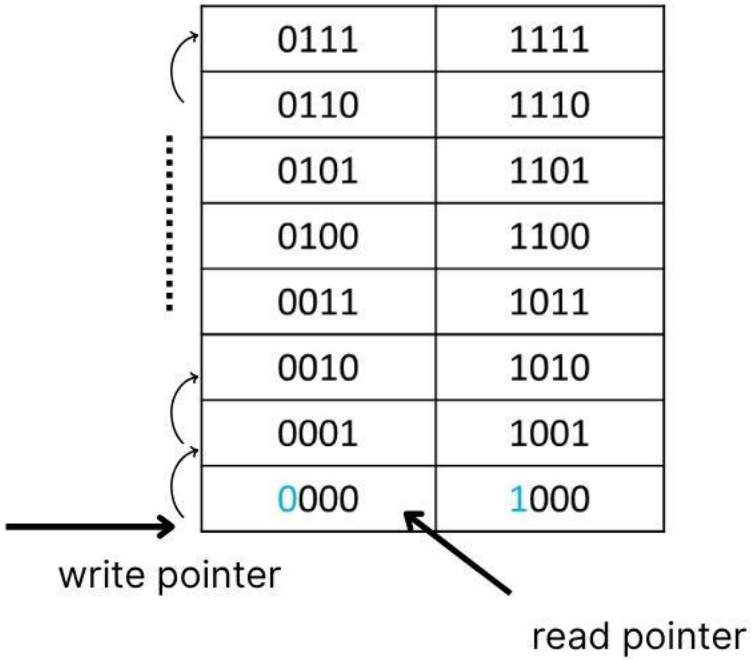
read pointer== write pointer

Memory can be **Full or Empty**

Add an extra bit for addressing to remove confusion

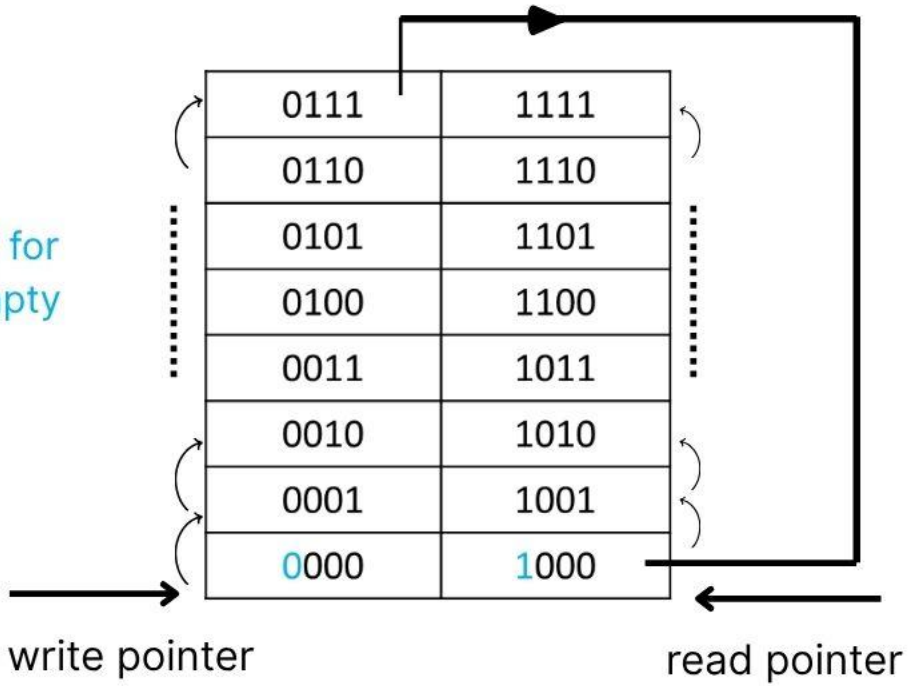
For FIFO depth of 8, 4 bits for addressing is used

Empty condition:



Observe the MSB bit for detecting full and empty logic

Full condition:



EMPTY logic:
If (read_en && ~EMPTY)
Increment Read Address
Else
No Increment

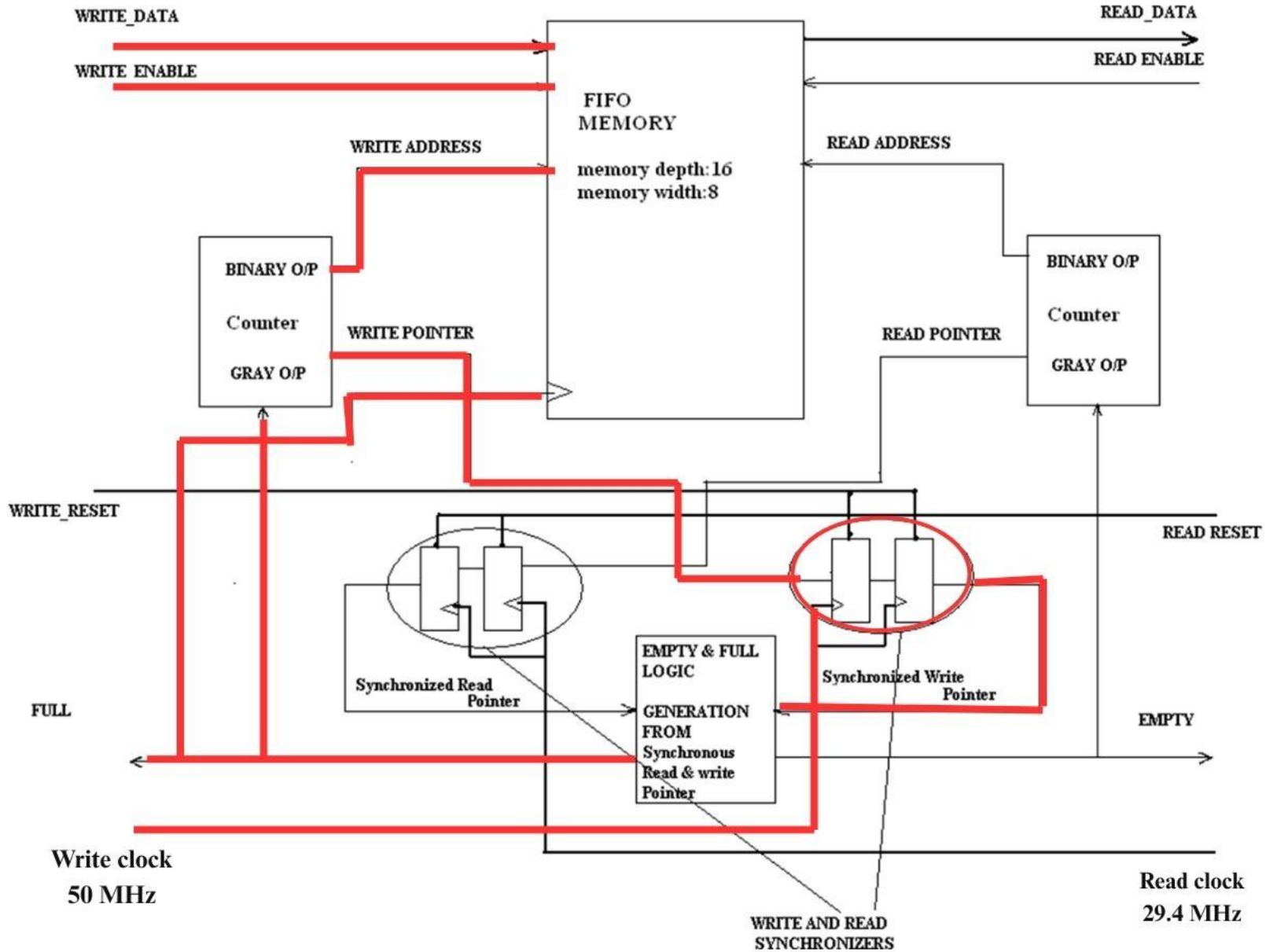
FULL logic:
If (write_en && ~FULL)
Increment Write Address
Else
No Increment

WRITE OPERATION FLOW

(left side, driven by WRITE_CLOCK

Step-by-step:

1. WRITE ENABLE is asserted & FIFO not FULL.
2. WRITE DATA goes to the FIFO input.
3. The Binary Counter (Write Pointer) gives:
   - Binary Output → used as the WRITE ADDRESS.
   - Gray Output → used for cross-clock synchronization.
4. WRITE ADDRESS selects the FIFO memory location to write.
5. Data is written into FIFO memory at WRITE_ADDRESS.
6. The Gray-coded Write Pointer is sent to the Read Clock Domain using double flip-flop synchronizers.
7. The synchronized write pointer is used to help the EMPTY/FULL LOGIC detect full condition.

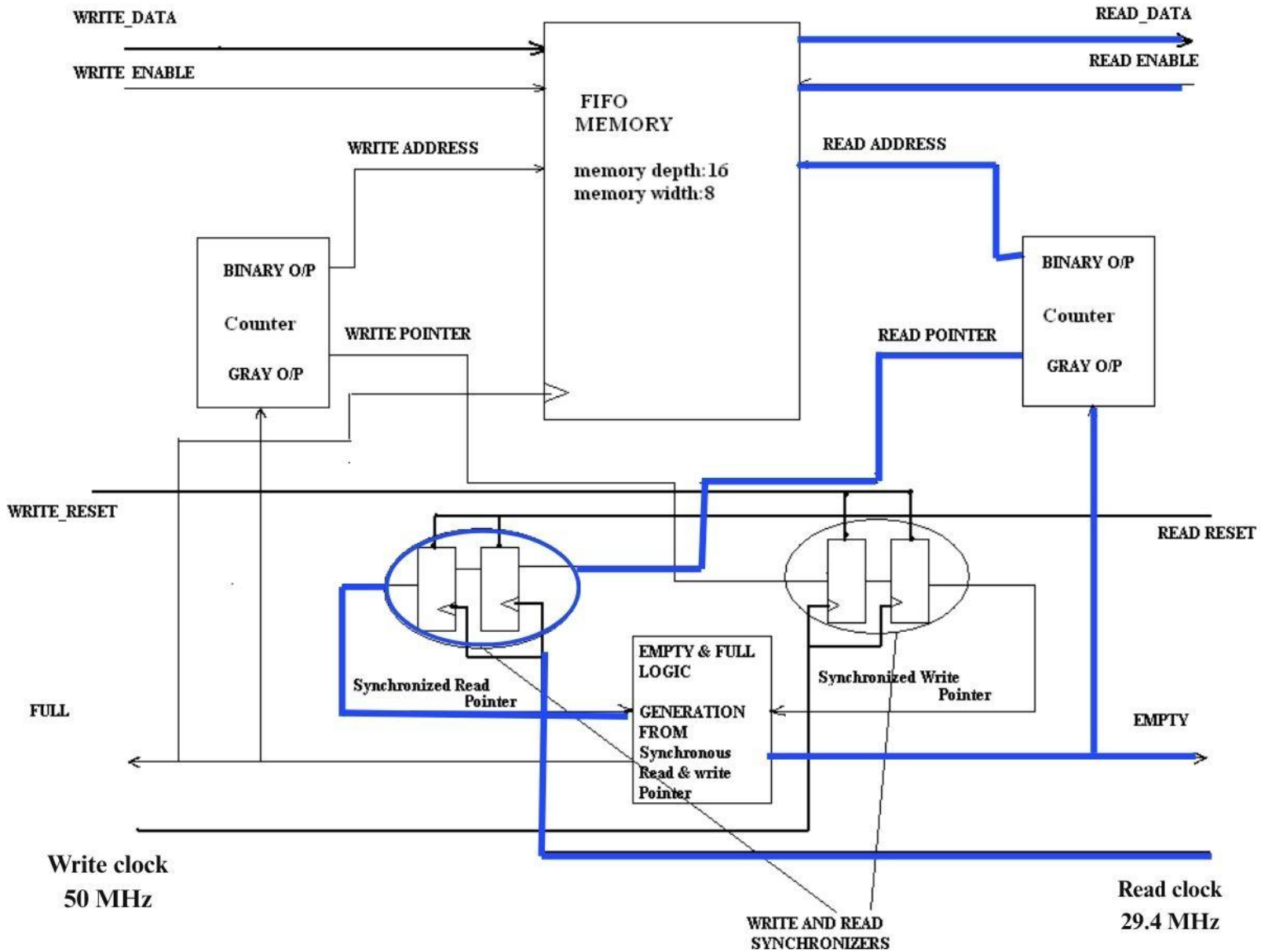


READ OPERATION FLOW

(right side, driven by READ_CLOCK)

Step-by-step:

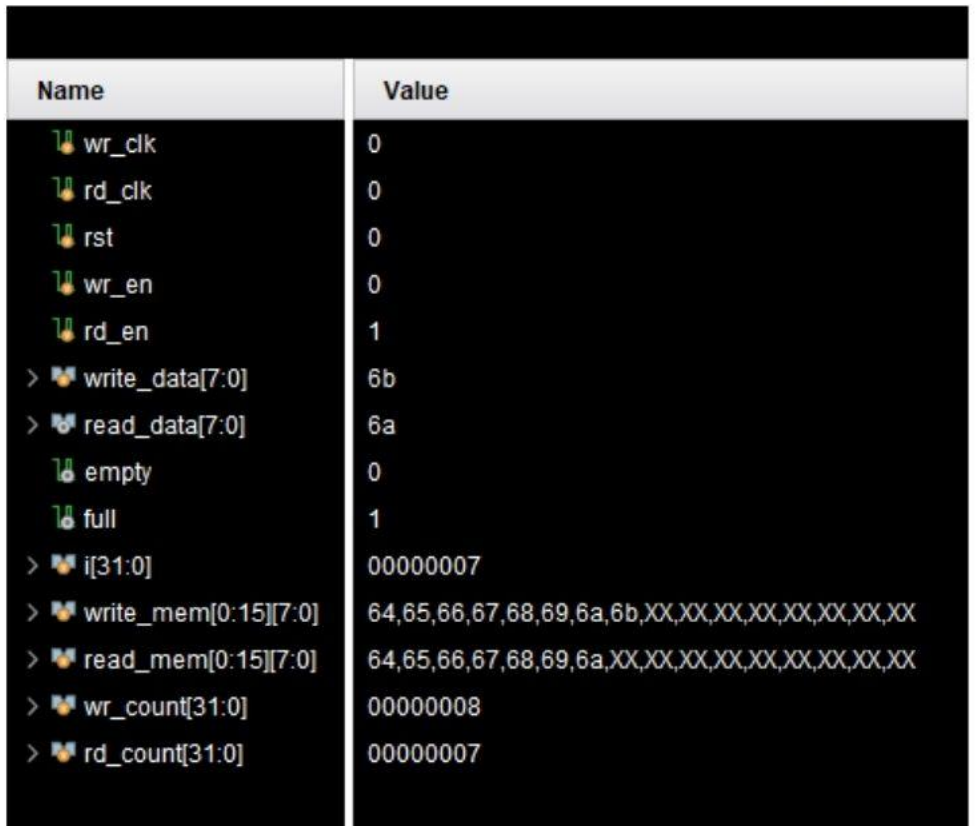
1. READ ENABLE is asserted & FIFO not EMPTY.
2. Binary Counter (Read Pointer) gives:
   - Binary Output → used as the READ ADDRESS.
   - Gray Output → for cross-clock synchronization.
3. READ ADDRESS selects the FIFO memory location to read.
4. Data is read from FIFO memory at READ_ADDRESS.
5. The Gray-coded Read Pointer is sent to the Write Clock Domain using synchronizers.
6. The synchronized read pointer is used to help the EMPTY/FULL LOGIC detect empty condition.



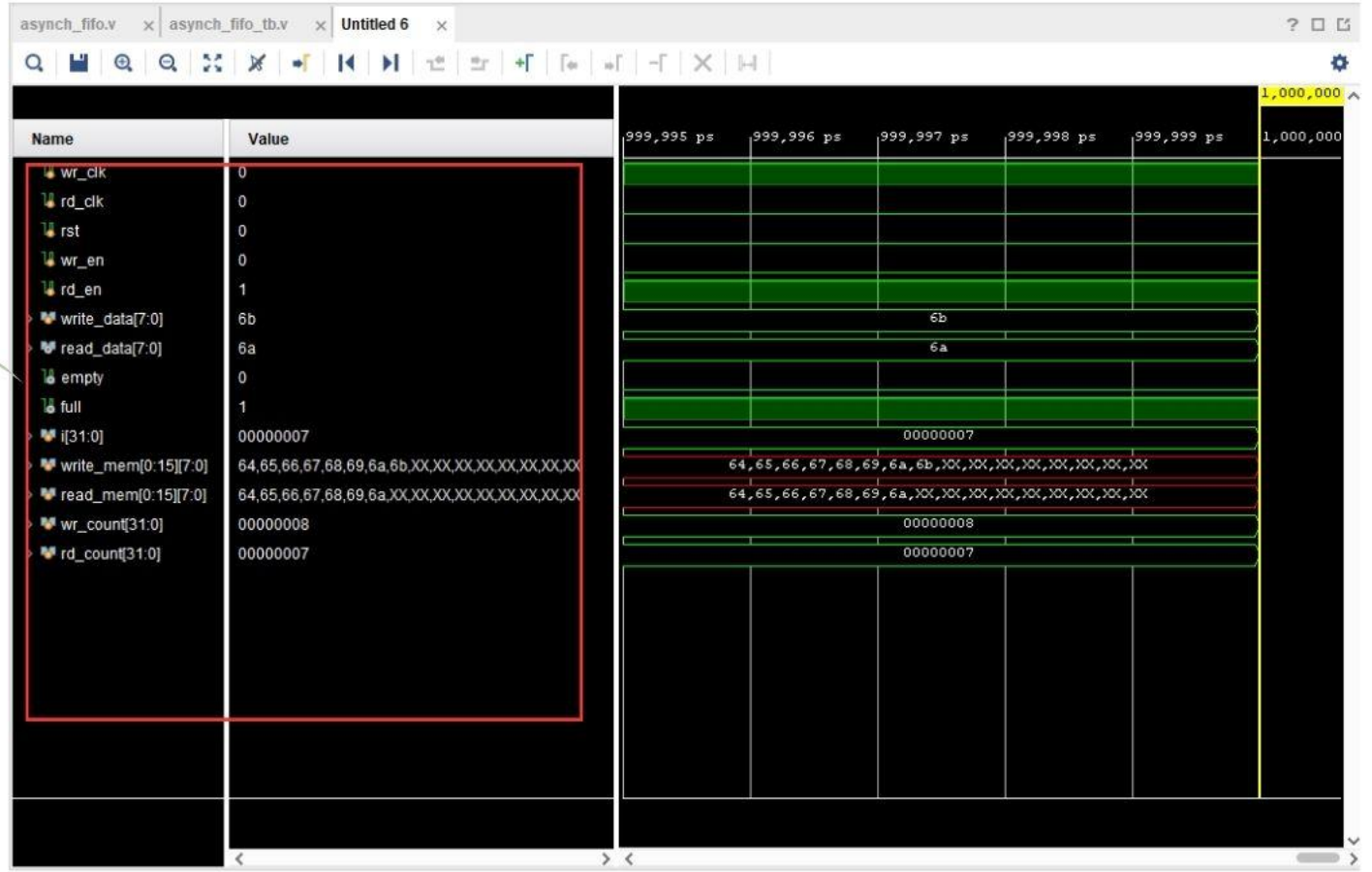
CODE :

```
module asynch_fifo (
    input wr_clk,              // Write clock
    input rd_clk,              // Read clock
    input rst,                 // Asynchronous reset
    input wr_en,               // Write enable signal
    input rd_en,               // Read enable signal
    input [7:0] write_data,    // Data input for writing to FIFO
    output reg [7:0] read_data, // Data output for reading from FIFO
    output wire empty,         // Flag to indicate FIFO is empty
    output wire full           // Flag to indicate FIFO is full
);

    parameter fifo_depth = 8; // FIFO depth is 8, so address pointer will be 3 bits wide
    parameter add_size = 4;   // Address size set to 4 bits to handle full/empty condition

    // Define the read and write pointers with 4 bits each
    reg [3:0] wptr, rptr;
    wire [3:0] wptr_gray, rptr_gray;

    // FIFO buffer memory of 8 entries, each 8 bits wide
    reg [7:0] mem [7:0];

    // Synchronized write and read pointers (Gray code)
    reg [3:0] wptr_gray_sync;
    reg [3:0] wptr_gray_ff1, wptr_gray_ff2;
    reg [3:0] rptr_gray_sync;
    reg [3:0] rptr_gray_ff1, rptr_gray_ff2;

    // Write data into FIFO buffer
    // This block is triggered by the write clock and checks for write enable and FIFO full c
    always @(posedge wr_clk) begin
        if (rst)
        begin
            // Reset write pointer to zero
            wptr <= 4'b0000;
        end
        else if (wr_en && !full) begin
            // Write data to the memory at the current write pointer position
            mem[wptr] <= write_data;
            // Increment the write pointer
            wptr <= wptr + 1;
        end
    end

    always @(posedge rd_clk) begin
        if (rst) begin
            // Reset read pointer to zero
            rptr <= 4'b0000;
        end else if (rd_en && !empty) begin
            // Read data from the memory at the current read pointer position
            read_data <= mem[rptr];
            // Increment the read pointer
            rptr <= rptr + 1;
        end
    end

    // Convert binary write and read pointers to Gray code
    // Gray code minimizes the risk of synchronization errors by ensuring only one bit changes at
    assign wptr_gray = wptr ^ (wptr >> 1);
    assign rptr_gray = rptr ^ (rptr >> 1);

    // Synchronizfe the write pointer to the read clock domain
    // This prevents metastability when comparing pointers across different clock domains
    always @(posedge rd_clk) begin
        if (rst) begin
            // Reset synchronization flip-flops
            wptr_gray_ff1 <= 0;
            wptr_gray_ff2 <= 0;
            wptr_gray_sync <= 0;
        end else begin
            // First stage of synchronization
            wptr_gray_ff1 <= wptr_gray;
            // Second stage of synchronization
            wptr_gray_ff2 <= wptr_gray_ff1;
            // Final synchronized write pointer in read clock domain
            wptr_gray_sync <= wptr_gray_ff2;
        end

    // Synchronize the read pointer to the write clock domain
    // This prevents metastability when comparing pointers across different clock domains
    always @(posedge wr_clk) begin
        if (rst) begin
            // Reset synchronization flip-flops
            rptr_gray_ff1 <= 0;
            rptr_gray_ff2 <= 0;
            rptr_gray_sync <= 0;
        end else begin
            // First stage of synchronization
            rptr_gray_ff1 <= rptr_gray;
            // Second stage of synchronization
            rptr_gray_ff2 <= rptr_gray_ff1;
            // Final synchronized read pointer in write clock domain
            rptr_gray_sync <= rptr_gray_ff2;
        end
    end

    // Calculate the empty condition
    // The FIFO is empty when the read pointer equals the synchronized write pointer
    assign empty = (rptr_gray == wptr_gray_sync);

    // Calculate the full condition
    // The FIFO is full when the write pointer is one position behind the read pointer, after wrapping around
    assign full = ((wptr_gray[3] != rptr_gray_sync[3]) &&  // MSB check to detect wrap-around
                   (wptr_gray[2] == rptr_gray_sync[2]) &&  // Ensure pointers are aligned except MSB
                   (wptr_gray[1:0] == rptr_gray_sync[1:0])); // Lower bits must match

endmodule
```

WAVEFORM:



Name	Value
wr_clk	0
rd_clk	0
rst	0
wr_en	0
rd_en	1
write_data[7:0]	6b
read_data[7:0]	6a
empty	0
full	1
i[31:0]	00000007
write_mem[0:15][7:0]	64,65,66,67,68,69,6a,6b,XX,XX,XX,XX,XX,XX,XX,XX
read_mem[0:15][7:0]	64,65,66,67,68,69,6a,XX,XX,XX,XX,XX,XX,XX,XX,XX
wr_count[31:0]	00000008
rd_count[31:0]	00000007



Conclusion: FIFO Working Summary

- Data is written correctly using wr_clk and wr_en when FIFO is not full.
- Data is read correctly using rd_clk and rd_en when FIFO is not empty.
- FIFO handles asynchronous read and write operations.
- FIFO status flags (full, empty) work properly.
- Internal memory shows correct and preserved data sequence.

Waveform analysis:

Write Operations (8 Writes):

- In the simulation, the FIFO performs 8 write operations starting shortly after reset. The write enable (wr_en) signal is asserted, and data values ranging from 0x64 to 0x6B are sequentially written into the FIFO on each rising edge of the write clock (wr_clk). As each value is written, the internal write pointer (wr_count) increments, and the write_mem array fills up to its maximum capacity of 8 entries. Once the FIFO reaches its limit, the full flag goes high, correctly indicating that no more data can be written until some is read out.

Read Operations (7 Reads):

- After the write phase, the read enable (rd_en) is asserted, and the FIFO begins reading data using the slower read clock (rd_clk). Only 7 read cycles occur by the 1µs mark, allowing 7 values—0x64 through 0x6A—to be read from the FIFO. The internal read pointer (rd_count) reflects this by reaching a count of 7. Since one data entry (0x6B) remains unread, the empty flag remains low, showing that the FIFO is not yet empty. This outcome is expected due to the slower read clock frequency compared to the write clock.

Conclusion:

•Developed FIFO enabling real data transfer between two unrelated clocks (50 MHz and 29.4 MHz) for a multi-clock FPGA system.

- Implemented gray code pointer-based synchronizer, reducing metastability risk to near-zero and ensuring data integrity in clock domain crossing environments.

- Delivered scalable, parameterized architecture adaptable for varied system depths, widths, and address sizes, cutting redesign time by 40%.

THANK YOU!