FIFO

FIRST IN FIRST OUT

A FIFO (First-In, First-Out) is a **queue-based memory buffer** where data is written into the queue in the order it arrives and is read in the same order. The principle follows a **first-come, first-served** approach, making it ideal for **buffering, data transfer, and synchronization** in digital systems.

FIFO Depth Calculation

The **depth** of a FIFO refers to the number of storage locations available for holding data before the buffer is full. It is crucial for ensuring proper data flow without overflow or underflow conditions.

The depth (D) of a FIFO can be calculated based on system parameters:

FIFO Depth Calculation

$$\text{FIFO Depth} \geq (\text{Max Data Rate In} - \text{Min Data Rate Out}) \times \text{Latency} + \text{Margin}$$

Where:

- **Max_DataRate_In** = Maximum input data rate
- **Min_DataRate_Out** = Minimum output data rate
- **Burst_Size** = Maximum burst size of incoming data

For single -clock FIFO:

If the FIFO is operating in a **single clock domain**, the depth is determined by the throughput requirements.

$$FIFODepth \geq \frac{MaxDataThroughput(words/sec) * Latency(sec)}{ClockPeriod(sec)}$$

For Asynchronous (Dual-Clock) FIFO

If the FIFO is crossing **two clock domains (CDC)**, the depth is determined by the ratio of input and output clock frequencies:

$$FIFODepth \geq \frac{F_{write}}{F_{read}} * BurstSize + Margin$$

Where:

- **Fwrite** = Write clock frequency
- **Fread** = Read clock frequency
- **Burst Size** = Maximum burst of incoming data
- **Margin** = Safety margin (typically 10-20% of the calculated depth)

There are two types of FIFOs

1. **Synchronous FIFO**
2. **Asynchronous FIFO**

Synchronous FIFO

- Read and write operations occur under the same clock domain.
- Requires control logic for full, empty, and almost full/almost empty status.

FIFO Control Logic:

(a) FIFO Full Condition

A FIFO is **full** when the write pointer catches up to the read pointer after writing.

Full Condition: (Wptr+1)mod N = Rptr

(b) FIFO Empty Condition

A FIFO is **empty** when the read pointer reaches the write pointer (i.e., no unread data is available).

Empty Condition: Wptr=Rptr

These flags are used to prevent **overflow** (writing to a full FIFO) or **underflow** (reading from an empty FIFO).

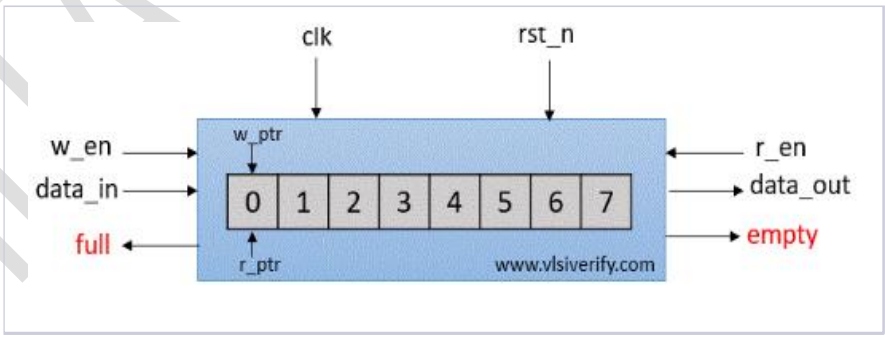
```
module synchronous_fifo
#(parameter DEPTH=8, DATA_WIDTH=8)
( input clk, rst_n,
  input w_en, r_en,
  input [DATA_WIDTH-1:0] data_in,
  output reg [DATA_WIDTH-1:0] data_out,
  output full, empty
);
  reg [$clog2(DEPTH)-1:0] w_ptr, r_ptr;
  reg [DATA_WIDTH-1:0] fifo[DEPTH];
// Set Default values on reset.
          always@(posedge clk) begin
            if(!rst_n) begin
            w_ptr <= 0; r_ptr <= 0;
           data_out <= 0;
           end
          end
 // To write data to FIFO
  always@(posedge clk) begin
    if(w_en & !full)begin
      fifo[w_ptr] <= data_in;
      w_ptr <= w_ptr + 1;
    end
  end
// To read data from FIFO
  always@(posedge clk) begin
    if(r_en & !empty) begin
      data_out <= fifo[r_ptr];
      r_ptr <= r_ptr + 1;
    end
  end
 assign full = ((w_ptr+1'b1) == r_ptr);
  assign empty = (w_ptr == r_ptr);
endmodule
```

Depth of FIFO: The number of slots or rows in FIFO is called the depth of the FIFO.

Width of FIFO: The number of bits that can be stored in each slot or row is called the width of the FIFO.

Working Principle

- Data is written into the FIFO **on a same clock edge till FIFO full**.
- Data is read from the FIFO **on a same clock edge till FIFO empty**.



Advantages of Synchronous FIFO

:: Simple control logic due to a single clock domain.
:: Faster and more power-efficient compared to asynchronous FIFOs.
:: Easier to implement in hardware (FPGAs, ASICs) using Block(BRAM).
:: Low latency and high-speed data transfer.

Applications of Synchronous FIFO

- Data buffering in processors and communication interfaces
- Pipeline processing in DSP (Digital Signal Processing)
- Networking devices for buffering
- Audio and video streaming systems
- Memory controllers for temporary data storage

Asynchronous FIFO

An Asynchronous FIFO is a type of FIFO memory used to transfer data between two different clock domains. It allows smooth data communication between two circuits that operate on different clock frequencies.

>> Many digital systems have **different clock domains** for different components.

>> A direct connection between these domains **causes metastability issues**.

>> **Asynchronous FIFOs** ensure reliable data transfer between these domains **without data loss or corruption**.

Pointer Synchronization Using Gray Code

- Directly passing binary pointers between different clock domains causes **metastability issues**.
- Solution: **Convert binary pointers to Gray code** before passing them to the other clock domain.

  Gray code ensures that only **one-bit changes at a time**, reducing errors.

VERILOG CODE

//FIFO full condition

```
wfull = (g_wptr_next ==
{~g_rptr_sync[PTR_WIDTH:PTR_WIDTH-1],
```

//FIFO empty conditionFIFO

```
rempty = (g_wptr_sync == g_rptr_next);
```

///FIFO WRITE POINTER HANDLER

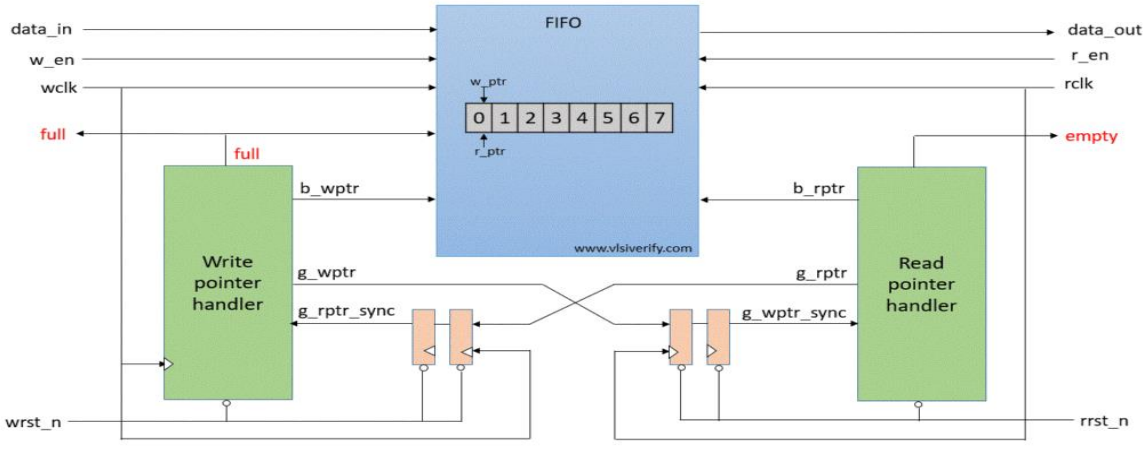
```
module wptr_handler #(parameter PTR_WIDTH=3) (
  input wclk, wrst_n, w_en,
  input [PTR_WIDTH:0] g_rptr_sync,
  output reg [PTR_WIDTH:0] b_wptr, g_wptr,
  output reg full
);
reg [PTR_WIDTH:0] b_wptr_next;
    reg [PTR_WIDTH:0] g_wptr_next;
    reg wrap_around;
    wire wfull;
  assign b_wptr_next = b_wptr+(w_en & !full);
  assign g_wptr_next = (b_wptr_next >>1)^b_wptr_next;
  always@(posedge wclk or negedge wrst_n) begin
       if(!wrst_n) begin
           b_wptr <= 0; // set default value
           g_wptr <= 0;  end
        else begin
         b_wptr <= b_wptr_next; // incr binary write pointer
         g_wptr <= g_wptr_next; // incr gray write pointer
       end
   end
  always@(posedge wclk or negedge wrst_n) begin
    if(!wrst_n)
        full <= 0;
    else
        full <= wfull;
  end
 assign wfull =
(g_wptr_next=={~g_rptr_sync[PTR_WIDTH:PTR_WIDTH-1],
g_rptr_sync[PTR_WIDTH-2:0]});
endmodule
```

//FIFO READ POINTER

```
module rptr_handler #(parameter PTR_WIDTH=3) (
  input rclk, rrst_n, r_en,
  input [PTR_WIDTH:0] g_wptr_sync,
  output reg [PTR_WIDTH:0] b_rptr, g_rptr,
  output reg empty
); reg [PTR_WIDTH:0] b_rptr_next;
   reg [PTR_WIDTH:0] g_rptr_next;
assign b_rptr_next = b_rptr+(r_en & !empty);
assign g_rptr_next = (b_rptr_next >>1)^b_rptr_next;
assign rempty = (g_wptr_sync == g_rptr_next);
always@(posedge rclk or negedge rrst_n) begin
    if(!rrst_n) begin
      b_rptr <= 0;
      g_rptr <= 0;
    end else begin
      b_rptr <= b_rptr_next;
      g_rptr <= g_rptr_next;
    end
  end
  always@(posedge rclk or negedge rrst_n) begin
    if(!rrst_n) empty <= 1;
    else        empty <= rempty;
  end
endmodule
```

///FIFO MEMORY

```
module fifo_mem #(parameter DEPTH=8,
DATA_WIDTH=8, PTR_WIDTH=3) (
  input wclk, w_en, rclk, r_en,
  input [PTR_WIDTH:0] b_wptr, b_rptr,
  input [DATA_WIDTH-1:0] data_in,
  input full, empty,
  output reg [DATA_WIDTH-1:0] data_out);
 reg [DATA_WIDTH-1:0] fifo[0:DEPTH-1];
 always@(posedge wclk) begin
    if(w_en & !full) begin
      fifo[b_wptr[PTR_WIDTH-1:0]] <= data_in;
    end  end
assign data_out = fifo[b_rptr[PTR_WIDTH-1:0]];
endmodule
```



Asynchronous FIFO

//TOP MODULE

```
`include "synchronizer.v"
`include "wptr_handler.v"
`include "rptr_handler.v"
`include "fifo_mem.v"

module asynchronous_fifo #(parameter DEPTH=8, DATA_WIDTH=8) (
  input wclk, wrst_n,
  input rclk, rrst_n,
  input w_en, r_en,
  input [DATA_WIDTH-1:0] data_in,
  output reg [DATA_WIDTH-1:0] data_out,
  output reg full, empty
);
  parameter PTR_WIDTH = $clog2(DEPTH);
  reg [PTR_WIDTH:0] g_wptr_sync, g_rptr_sync;
  reg [PTR_WIDTH:0] b_wptr, b_rptr;
  reg [PTR_WIDTH:0] g_wptr, g_rptr;
  wire [PTR_WIDTH-1:0] waddr, raddr;
synchronizer #(PTR_WIDTH) sync_wptr (rclk, rrst_n,g_wptr,g_wptr_sync);
//write pointer to read clock domain
synchronizer #(PTR_WIDTH) sync_rptr (wclk, wrst_n, g_rptr, g_rptr_sync);
//read pointer to write clock domain

wptr_handler #(PTR_WIDTH) wptr_h(wclk, wrst_n,
w_en,g_rptr_sync,b_wptr,g_wptr,full);
rptr_handler #(PTR_WIDTH) rptr_h(rclk, rrst_n,
r_en,g_wptr_sync,b_rptr,g_rptr,empty);
fifo_mem fifom(wclk, w_en, rclk, r_en,b_wptr, b_rptr, data_in,full,empty,
data_out);

endmodule
```

//2 FLIP-FLOP SYNCHRONIZER

```
module synchronizer #(parameter WIDTH=3) (input
clk, rst_n, [WIDTH:0] d_in, output reg [WIDTH:0]
d_out);
 reg [WIDTH:0] q1;
 always@(posedge clk) begin
  if(!rst_n) begin
    q1 <= 0;
    d_out <= 0;
   end
   else begin
     q1 <= d_in;
     d_out <= q1;
   end
  end
endmodule
```

Applications of Asynchronous FIFO

- Clock Domain Crossing (CDC)
- Processor to Peripheral Communication
- High-speed data Acquisition Systems
- Network Routers & Switches (Packet Buffers)
- Audio & Video Processing Systems
- FPGA-Based Designs (Bridging Different IP Blocks)
- USB & PCIe Communication
- Software-Defined Radio (SDR) & Wireless Communication
- Video Frame Buffers in GPUs