**Implementation of FIFO (FIRST – IN FIRST – OUT)**

**using AXI-4 Stream protocol**

**What is FIFO ?**

In modern System-on-Chip (SoC) and FPGA-based designs, high-performance data transfer between different modules is a critical requirement. One of the most widely used methods to achieve efficient and reliable data movement is by using First-In-First-Out (FIFO) buffers.

A **FIFO** (First-In, First-Out) is a buffer that stores data temporarily and outputs it in the same order it was received.

**Why we need FIFO ?**

* It’s used to handle **rate mismatches** between producer and consumer modules, e.g., when one block generates data faster than another block can consume.
* It’s also used in **clock domain crossing (CDC)** to safely pass data between two clock regions.
* **Burst Handling** – If input comes in bursts but the output can only process steadily (or vice versa).
* **Decoupling** – It avoids stalling one module while another is busy.

**Basic architecture of a FIFO :**

A hardware FIFO has:

1. **Memory array** – stores the actual data (reg [W-1:0] mem[DEPTH-1:0]).
2. **Write pointer (wr\_ptr)** – It points to the next location to write data in.
3. **Read pointer (rd\_ptr)** – It points to the next location to read data from.
4. **Control logic** – generates flags (full, empty, sometimes almost\_full, almost\_empty).
5. **Optional counters** – track number of stored elements.

**AXI FIFO Concept:**

An **AXI FIFO** is generally a FIFO memory buffer that provides an **AXI4-Stream slave interface** on the input side and an **AXI4-Stream master interface** on the output side.

* **Write side (Slave interface):** Accepts data when s\_valid and s\_ready are high.
* **FIFO Memory:** Stores data sequentially.
* **Read side (Master interface):** Provides data when m\_valid and m\_ready are high.

**Wr\_ptr**

**Rd\_ptr**

**M\_ready**

**M\_valid**

**M\_data**

**S\_ready**

**S\_data**

**S\_valid**

SLAVE

FIFO

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

MASTER

Block diagram of AXI FIFO

**Applications of FIFO:**

FIFO is used in many digital systems:

* **AXI4-Stream Interfaces** – To buffer data between processing IP blocks.
* **UART, SPI, I2C Controllers** – To store received/transmitted data temporarily.
* **Network Routers / Switches** – Packets buffered before forwarding.
* **Multimedia (Video/Audio pipelines)** – Streaming pixels, audio samples, frames.
* **Data Converters (ADC/DAC)** – Handling mismatched rates between converters and processors.

**Types of FIFO:**

* Synchronous FIFO – It uses same clock for both read and write. Simpler logic (pointers can be compared directly).
* Asynchronous FIFO (CDC FIFO) - Read and write have different clocks.

**Implementation:**

1. Using pointer comparison: We compare wr\_ptr & rd\_ptr for both flags.

Full = (wr\_ptr +1 == rd\_ptr) or Full = (wr\_ptr +1 ) % Depth == rd\_ptr)

Empty = (wr\_ptr == rd\_ptr) or Empty = (wr\_ptr == rd\_ptr) && !Full

These conditions are for one-slot method where we reserve one memory to compare pointers.

1. Using counter : We count how many data-words are written into the memory and how many data-words are read from the memory.

Empty = (count == 0)

Full = (count == Depth)

**METHOD 1 – Using pointers comparison**

**Design code:**

module fifo\_axi#(parameter DATA\_WIDTH=32, FIFO\_DEPTH=16)(

//Global Signals

input aclk, aresetn,

//Input slave interface

input [DATA\_WIDTH-1:0] s\_data,

input s\_valid,

output s\_ready,

//Output master interface

output reg [DATA\_WIDTH-1:0] m\_data,

output m\_valid,

input m\_ready);

//fifo parameters

localparam ptr\_width = $clog2(FIFO\_DEPTH);

reg [DATA\_WIDTH-1:0] fifo\_mem [0:FIFO\_DEPTH-1]; // memory array

reg [ptr\_width-1:0] rd\_ptr,wr\_ptr; // read and write pointers

wire full = (wr\_ptr+1'b1)== rd\_ptr; // using one slot method

wire empty = (rd\_ptr == wr\_ptr) && !full ; // when both pointers are same, the fifo indicates

empty

assign s\_ready = !full;

assign m\_valid = !empty;

wire rd\_en = (m\_valid && m\_ready);

wire wr\_en = (s\_valid && s\_ready);

always@(\*) begin

if(empty) m\_data <= 32'hx;

else m\_data <= fifo\_mem[rd\_ptr];

end

always@(posedge aclk or negedge aresetn) begin

if(!aresetn) begin

wr\_ptr <= 0;

rd\_ptr <= 0;

m\_data <= 0;

end

else

begin

//Writing data into the FIFO

if (wr\_en)

begin

fifo\_mem[wr\_ptr] <= s\_data;

wr\_ptr <= wr\_ptr + 1 ;

end

//Reading data from FIFO

if (rd\_en)

begin

//m\_data <= fifo\_mem[rd\_ptr];

rd\_ptr <= rd\_ptr + 1;

end

end

end

endmodule

**Test bench:**

module axi\_fifo\_tb();

parameter DATA\_WIDTH = 32;

parameter FIFO\_DEPTH = 16;

reg aclk;

reg aresetn;

// Slave interface

reg [DATA\_WIDTH-1:0] s\_data;

reg s\_valid;

wire s\_ready;

// Master interface

wire [DATA\_WIDTH-1:0] m\_data;

wire m\_valid;

reg m\_ready;

integer i,j,wi,ri;

// DUT instantiation

fifo\_axi #(DATA\_WIDTH,FIFO\_DEPTH) dut

(aclk,aresetn,s\_data,s\_valid,s\_ready,m\_data,m\_valid,m\_ready);

// Clock generation

initial aclk=0;

always #5 aclk = ~aclk;

initial begin

aresetn = 0;

s\_valid = 0;

s\_data = 0;

m\_ready = 0;

#20 aresetn = 1;

end

task write\_fifo(input [DATA\_WIDTH-1:0] din);

begin

@(posedge aclk);

s\_valid=1;

s\_data=din;

@(posedge aclk);

s\_valid=0;

end

endtask

task read\_axififo();

begin

@(posedge aclk);

m\_ready=1;

@(posedge aclk);

m\_ready=0;

end

endtask

initial begin

@(posedge aresetn);

@(posedge aclk);

$display("Writing data into fifo");

for(i=0; i<FIFO\_DEPTH; i=i+1)

begin

write\_fifo($urandom);

$display(" %h data written in fifo",i);

End

$display("Reading data from fifo");

for(j=0; j<FIFO\_DEPTH; j=j+1)

begin

$display(" %h data read from fifo",j);

read\_axififo();

end

$display("Simultaneous write and read operation to fifo");

fork

begin

for(wi=0; wi<=30; wi=wi+1) begin

write\_fifo($urandom);

end

end

begin

for(ri=0; ri<=31; ri=ri+1)begin

read\_axififo();

end

end

join

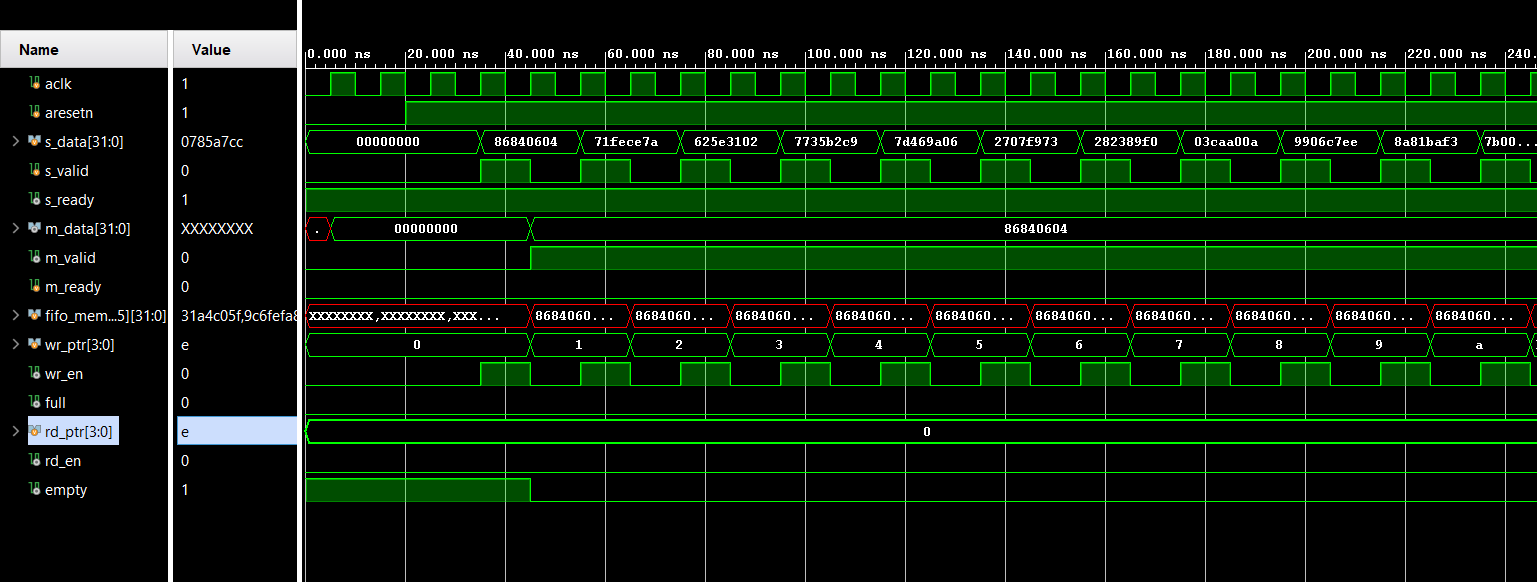
#50;

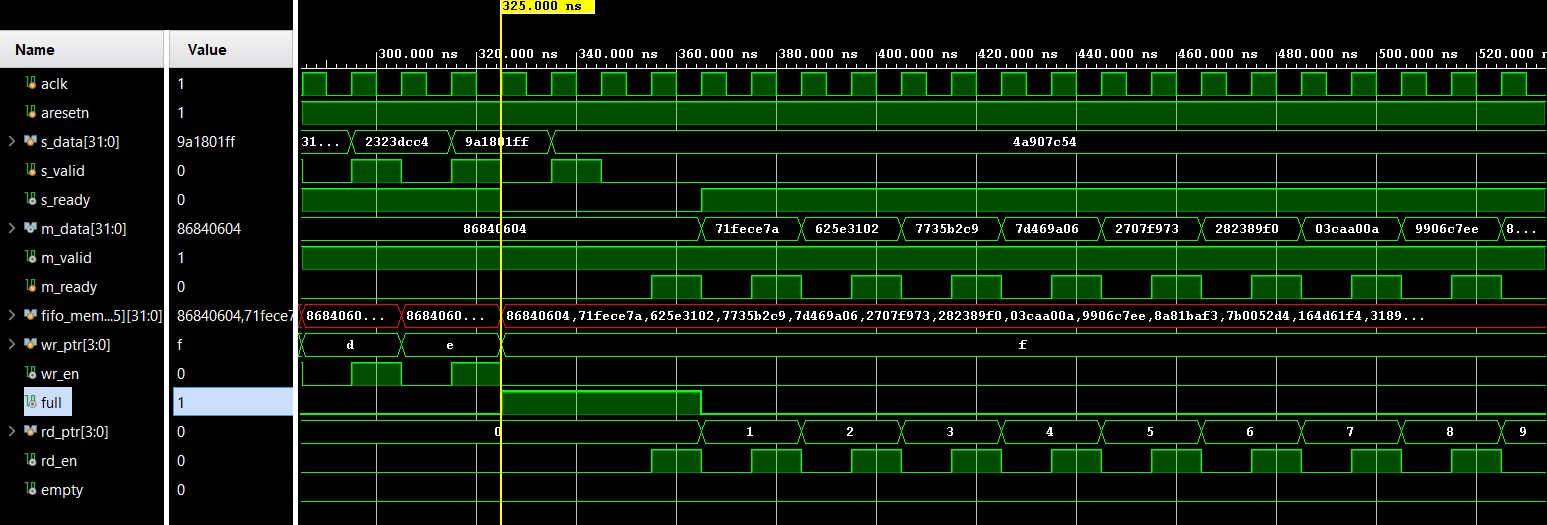
$finish;

end

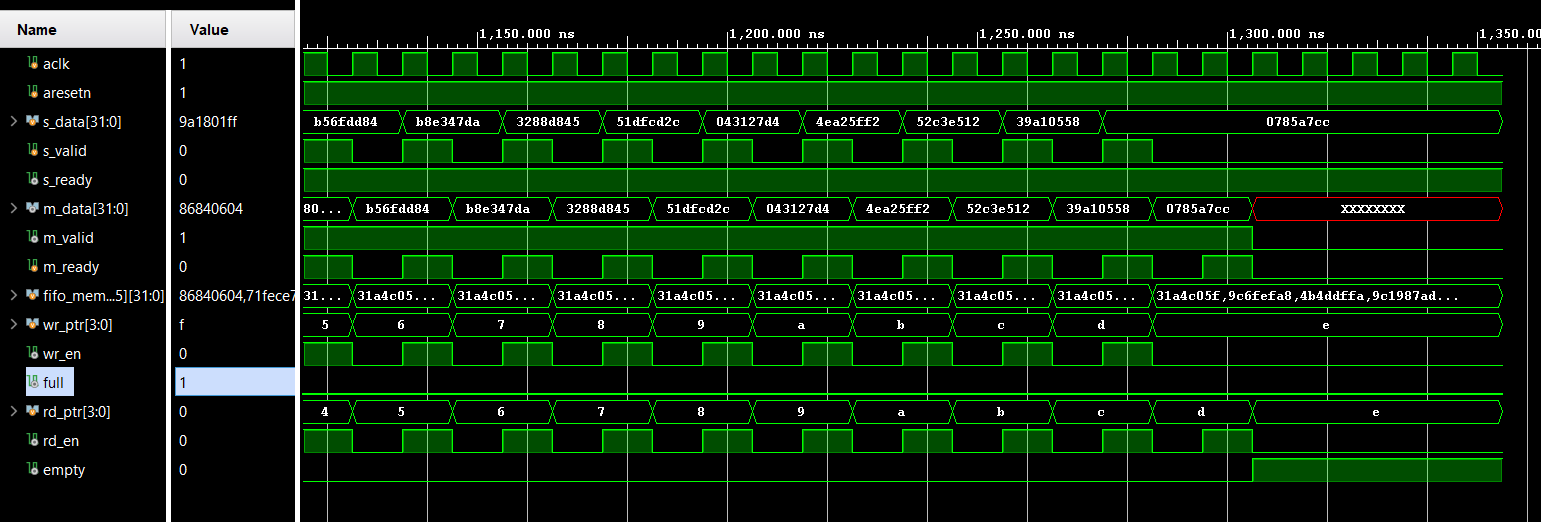
endmodule

**Simulation Results:**









**METHOD 2 – Using counter**

**Design code:**

module fifo\_axi #(parameter DATA\_WIDTH=32, FIFO\_DEPTH=16)(

//Global Signals

input aclk, aresetn,

//Input slave interface

input [DATA\_WIDTH-1:0] s\_data,

input s\_valid,

output s\_ready,

//Output master interface

output reg [DATA\_WIDTH-1:0] m\_data,

output m\_valid,

input m\_ready

);

//fifo parameters

localparam ptr\_width = $clog2(FIFO\_DEPTH);

reg [DATA\_WIDTH-1:0] fifo\_mem [0:FIFO\_DEPTH-1]; // memory array

reg [ptr\_width-1:0] rd\_ptr,wr\_ptr; // read and write pointers

reg [ptr\_width:0]count;

wire empty = (count == 0);

wire full = (count == FIFO\_DEPTH);

assign s\_ready = !full;

assign m\_valid = !empty;

wire rd\_en = (m\_valid && m\_ready);

wire wr\_en = (s\_valid && s\_ready);

always@(\*) begin

if(empty) m\_data = 32'hx;

else m\_data = fifo\_mem[rd\_ptr];

end

always@(posedge aclk or negedge aresetn)

begin

if(!aresetn) begin

wr\_ptr <= 0;

rd\_ptr <= 0;

count <= 0;

end

else begin

//Writing data into the FIFO

if (wr\_en)

begin

fifo\_mem[wr\_ptr] <= s\_data;

wr\_ptr <= wr\_ptr + 1 ;

end

//Reading data from FIFO

if (rd\_en)

begin

rd\_ptr <= rd\_ptr + 1;

end

case({wr\_en , rd\_en})

2'b10 : count <= count + 1;

2'b01 : count <= count - 1;

default : count <= count;

endcase

end

end

endmodule

**Test bench:**

module fifo\_axitb();

parameter DATA\_WIDTH = 32;

parameter FIFO\_DEPTH = 16;

reg aclk;

reg aresetn;

// Slave interface

reg [DATA\_WIDTH-1:0] s\_data;

reg s\_valid;

wire s\_ready;

// Master interface

wire [DATA\_WIDTH-1:0] m\_data;

wire m\_valid;

reg m\_ready;

integer i,j,w\_done,r\_done,x,y;

// DUT instantiation

fifo\_axi #(DATA\_WIDTH,FIFO\_DEPTH) dut

(aclk,aresetn,s\_data,s\_valid,s\_ready,m\_data,m\_valid,m\_ready);

// Clock generation

initial aclk=0;

always #5 aclk = ~aclk;

initial begin

aresetn = 0;

s\_valid = 0;

s\_data = 0;

m\_ready = 0;

#20 aresetn = 1;

end

task write(input [DATA\_WIDTH-1:0] in, output integer wf);

begin

@(posedge aclk);

wf=0;

s\_valid = $urandom\_range(0,1);

if(s\_valid) begin

s\_data = in;

wf=1;

// #10;

//s\_valid=0;

end

else s\_valid = 0;

@(posedge aclk);

end

endtask

task read(output integer rf);

begin

@(posedge aclk);

rf=0;

m\_ready=$urandom\_range(0,1);

if(m\_ready) begin

rf=1; //#10;

//m\_ready=0;

end

end

endtask

initial begin

@(posedge aresetn);

@(posedge aclk);

$display("Writing data into fifo until it's full ");

i=0;

while(i<=FIFO\_DEPTH)

begin

//@(posedge aclk);

write($random,w\_done);

if(w\_done) i=i+1;

#10;

s\_valid=0;

end

s\_valid=0;

$display("Reading data from fifo until it's empty ");

j=0;

while(j<=FIFO\_DEPTH) begin

read(r\_done);

if(r\_done) j=j+1;

#10;

end

m\_ready=0;

$display("Simultaneous writing and reading data for fifo ");

fork

begin

x=0;

while(x<=(FIFO\_DEPTH+FIFO\_DEPTH))

begin

@(posedge aclk);

write($random,w\_done);

if(w\_done) x=x+1;

#10;

s\_valid=0;

end

end

begin

y=0;

while(y<=(FIFO\_DEPTH+FIFO\_DEPTH)) begin

read(r\_done);

if(r\_done) y=y+1;

#10;

end

end

join

#20 $finish;

end

endmodule

**Simulation Results:**

