ASYNCHRONOUS FIFO DESIGN

(Dual Clock FIFO)

PRIYANSH KASYAP
(213070114)

Department of
Electrical Engineering
IIT-Bombay

Contents

Objective:

Introduction:

Asynchronous FIFO : FIFOs are often used to safely pass data from one clock domain to another clock domain. An asynchronous FIFO refers to a FIFO design where data is written to a FIFO buffer from one clock domain and the same data is read from the FIFO buffer from another clock domain, where the two clock domains are asynchronous to each other.

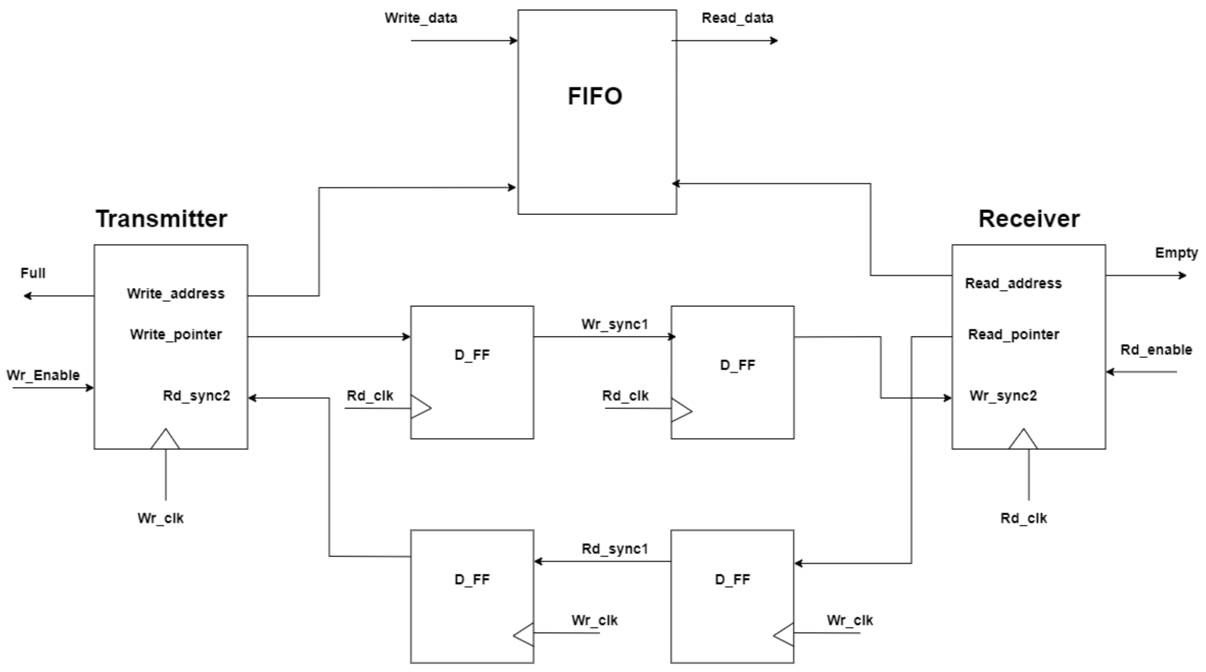


Fig_1: Asynchronous FIFO design

Blocks in FIFO design:

Read clock domain to write clockdomain Synchronizer:

This module contains flip-flops that are synchronized to the write clock. Synchronize the write pointer into the read clock domain. This is because the write pointer is having metastability due to setup and hold time in the read clock. So, there is a synchronizer for this problem. The synchronized read pointer will be used by the Write control module to generate the **Write_Full** condition.

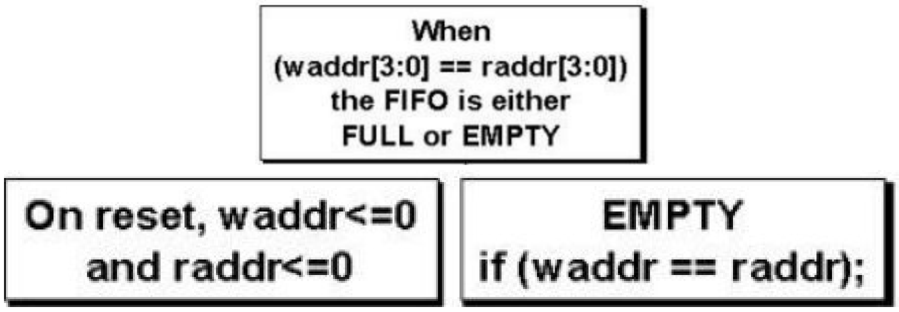
Write clock domain to read clockdomain Synchronizer:

This module also contains flip-flops that are synchronized to the read clock , Similarly we will use one synchronizer to synchronize read pointer into the write clock domain, because read pointer, due to setup and hold time violation in the write clock will give the metastability. Further, this synchronized write pointer will be used to generate the **Read_empty** condition.

Logic to generate Read_empty signal:

This module is completely synchronous to the read- clock domain and contains the FIFO read pointer and empty-flag logic.

In our case we have used FIFO of depth 16 and address is of 4 bits, let us suppose that we have written some data at these 16 places. Now, write pointer will wrap up, again comes to 0th location and will start writing values. So, as to get to know whether the FIFO is full or empty, we will add one extra bit such that when FIFO is empty both the pointers will point to same location



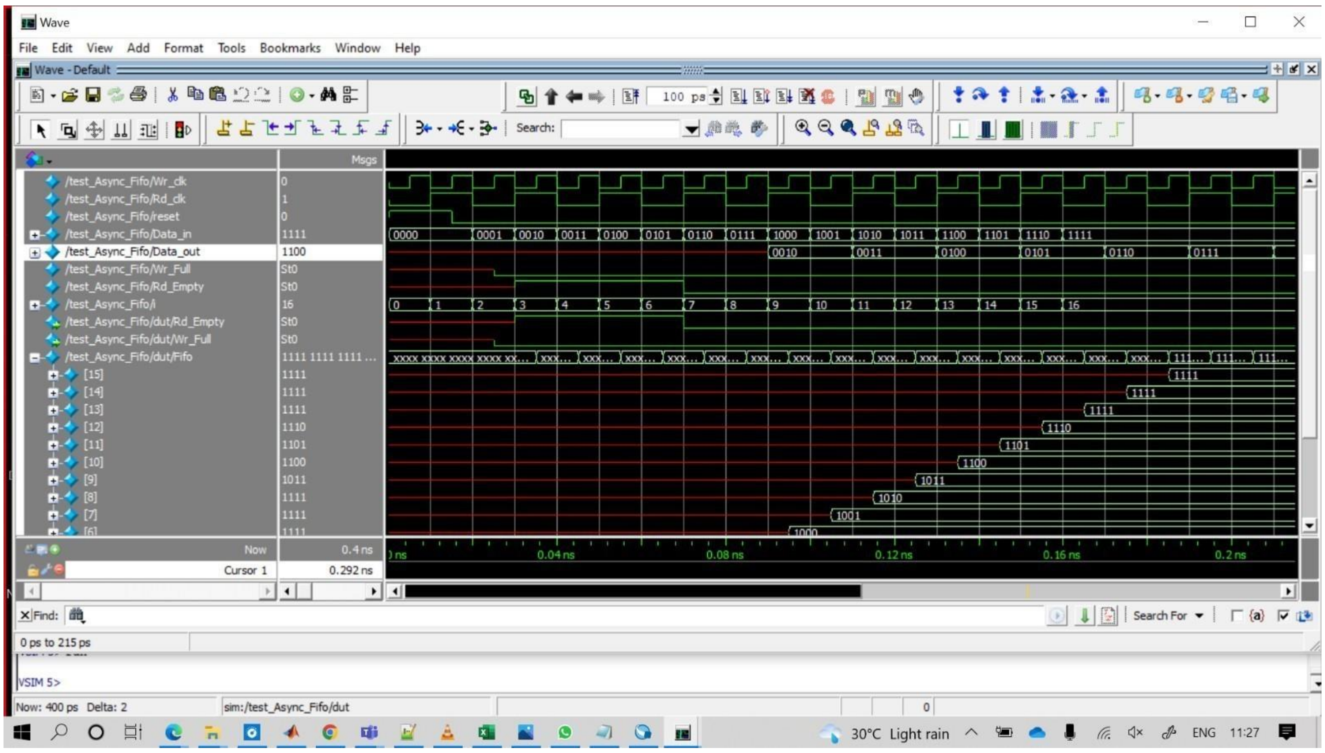
Logic to generate Write_Full signal:

This module is completely synchronous to the write- clock domain and contains the FIFO write pointer and full-flag logic.

When the complement of MSB of write pointer equals the MSB of read pointer, and rest all the bits are same, then we can say that write pointer has traversed the FIFO once, i.e. it has already wrapped up and hence can generate the FULL condition logic.

```
FULL
if ({~waddr[4],waddr[3:0]} == raddr);
```

Simulation Results:



```
/test_Async_Fifo/Wr_clk          0
/test_Async_Fifo/Rd_clk          1
/test_Async_Fifo/reset           0
/test_Async_Fifo/Data_in         1111
/test_Async_Fifo/Data_out        1100
/test_Async_Fifo/Wr_Full         St0
/test_Async_Fifo/Rd_Empty        St0
/test_Async_Fifo/i               16
/test_Async_Fifo/dut/Rd_Empty    St0
/test_Async_Fifo/dut/Wr_Full     St0
/test_Async_Fifo/dut/Fifo        1111 1111 1111 ...
  [15]                           1111
  [14]                           1111
  [13]                           1111
  [12]                           1110
  [11]                           1101
  [10]                           1100
  [9]                            1011
  [8]                            1111
  [7]                            1111
  [6]                            1111
Now                              0.4 ns
Cursor 1                         0.292 ns

0 ps to 215 ps
VSIM 5>
Now: 400 ps  Delta: 2    sim:/test_Async_Fifo/dut
```

- Verilog code is written for Asynchronous FIFO and Testbench in Quartus Prime.
- Simulations are done in Modelsim using the written testbench.

CODE :-

```
module Async_Fifo(Wr_clk,Rd_clk,reset,Data_in,Data_out,Rd_Empty,Wr_Full);
parameter Depth = 16;
parameter Width = 4;
input Wr_clk,Rd_clk,reset;
input [Width-1:0] Data_in;
output [Width-1:0] Data_out;
output Rd_Empty,Wr_Full;

reg [Width-1:0] Fifo [Depth-1:0];

reg [Width-1:0] Data_out_reg;
reg [Width:0] Wr_ptr_binary,Wr_sync_1,Wr_ptr_sync_2;
reg [Width:0] Rd_ptr_binary,Rd_sync_1,Rd_ptr_sync_2;
wire [Width:0] Rd_ptr_gray,Wr_ptr_gray;
wire [Width:0] Rd_ptr_binary_sync,Wr_ptr_binary_sync;

//--Write_data_into_Fifo--//

assign Rd_Empty = (Wr_ptr_binary_sync == Rd_ptr_binary) ? 1 : 0;
assign Wr_Full = ({~Wr_ptr_binary[Width],Wr_ptr_binary[Width-1:0]} == Rd_ptr_binary_sync) ? 1:0;

assign Data_out = Data_out_reg;
always @(posedge Wr_clk)
begin
        if(reset)
                        Wr_ptr_binary <= 4'd0;
        else if(Wr_Full == 0 ) begin
                        Wr_ptr_binary <= Wr_ptr_binary + 1;
                        Fifo[Wr_ptr_binary[Width-1:0]] <= Data_in;
                        end
end


//--Read_data_out_of_Fifo--//

always @(posedge Rd_clk)
begin
        if(reset)
                Rd_ptr_binary <= 4'd0;
                else if(Rd_Empty == 0) begin

                Rd_ptr_binary <= Rd_ptr_binary + 1;
                Data_out_reg <= Fifo[Rd_ptr_binary[Width-1:0]];
                end
end

//--Read_and_Write_synchronizers--//
always @(posedge Rd_clk)
begin
```

```
Wr_sync_1 <= Wr_ptr_gray;
Wr_ptr_sync_2 <= Wr_sync_1;
end

always @(posedge Wr_clk)
begin
Rd_sync_1 <= Rd_ptr_gray;
Rd_ptr_sync_2 <= Rd_sync_1;
end

//-- Binary_to_gray_and_gray_to_binary--//

assign Wr_ptr_gray = Wr_ptr_binary ^ (Wr_ptr_binary >> 1);
assign Rd_ptr_gray = Rd_ptr_binary ^ (Rd_ptr_binary >> 1);

assign Rd_ptr_binary_sync[4] = Rd_ptr_sync_2[4];
assign Rd_ptr_binary_sync[3] = Rd_ptr_binary_sync[4]^Rd_ptr_sync_2[3];
assign Rd_ptr_binary_sync[2] = Rd_ptr_binary_sync[3]^Rd_ptr_sync_2[2];
assign Rd_ptr_binary_sync[1] = Rd_ptr_binary_sync[2]^Rd_ptr_sync_2[1];
assign Rd_ptr_binary_sync[0] = Rd_ptr_binary_sync[1]^Rd_ptr_sync_2[0];

assign Wr_ptr_binary_sync[4] = Wr_ptr_sync_2[4];
assign Wr_ptr_binary_sync[3] = Wr_ptr_binary_sync[4]^Wr_ptr_sync_2[3];
assign Wr_ptr_binary_sync[2] = Wr_ptr_binary_sync[3]^Wr_ptr_sync_2[2];
assign Wr_ptr_binary_sync[1] = Wr_ptr_binary_sync[2]^Wr_ptr_sync_2[1];
assign Wr_ptr_binary_sync[0] = Wr_ptr_binary_sync[1]^Wr_ptr_sync_2[0];



endmodule
```

TEST BENCH

```
module test_Async_Fifo;
reg Wr_clk,Rd_clk,reset;
reg [3:0] Data_in;
wire [3:0]Data_out;
wire Wr_Full,Rd_Empty;
integer i;
Async_Fifo
dut(.Rd_clk(Rd_clk),.Wr_clk(Wr_clk),.reset(reset),.Data_in(Data_in),.Data_out(Data_out),.Wr_Full(Wr_Full
),.Rd_Empty(Rd_Empty));

initial
begin
Wr_clk=0;
Rd_clk=0;
reset=1;
#15 reset=0;
end


always
#5 Wr_clk=~Wr_clk;

always
#10 Rd_clk=~Rd_clk;



initial
#1000 $finish;


initial begin
Data_in<=4'd0;

for(i=0;i<30;i=i+1)
#10 Data_in <= i;
end

initial
#350 Data_in <= 4'd0;

endmodule
```

7