A PROJECT REPORT ON

Design and Verification of Synchronous FIFO

SUBMITTED IN THE FULFILLMENT OF THE REQUIREMENTS FOR THE SYSTEM VERILOG COURSE (2 Credits)

PROJECT TEAM:

<table>
<thead>
<tr>
<th>NAMES</th>
<th>USN</th>
<th>EMAIL ID</th>
<th>CONTACT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dharshan BJ</td>
<td>1PI11EC034</td>
<td><a href="mailto:dharshanbj@gmail.com">dharshanbj@gmail.com</a></td>
<td>+91-9663239371</td>
</tr>
<tr>
<td>HK Sachin</td>
<td>1PI11EC039</td>
<td><a href="mailto:hhksachin5@gmail.com">hhksachin5@gmail.com</a></td>
<td>+91-9738554163</td>
</tr>
<tr>
<td>Mohammed Sahil Akbar</td>
<td>1PI11EC063</td>
<td><a href="mailto:sahilakbar94@gmail.com">sahilakbar94@gmail.com</a></td>
<td>+91-9538614406</td>
</tr>
</tbody>
</table>

MENTOR:

DEEPA YAGAIN
Associate Professor
Department of Electronics and Communication Engineering
P. E. S Institute of Technology
deepa.yagain@gmail.com
INTRODUCTION:

FIFO is a First-In-First-Out memory queue with control logic that manages the read and write operations, generates status flags, and provides optional handshake signals for interfacing with the user logic. It is often used to control the flow of data between source and destination. FIFO can be classified as synchronous or asynchronous depending on whether same clock or different (asynchronous) clocks control the read and write operations. In this project the objective is to design, verify and synthesize a synchronous FIFO using binary coded read and write pointers to address the memory array. FIFO full and empty flags are generated and passed on to source and destination logics, respectively, to pre-empt any overflow or underflow of data. In this way data integrity between source and destination is maintained.

In computer programming, FIFO (first-in, first-out) is an approach to handling program work requests from queues or stacks so that the oldest request is handled first. In hardware it is either an array of flops or Read/Write memory that store data given from one clock domain and on request supplies with the same data to other clock domain following the first in first out logic. The clock domain that supplies data to FIFO is often referred as WRITE OR INPUT LOGIC and the clock domain that reads data from the FIFO is often referred as READ OR OUTPUT LOGIC. FIFOs are used in designs to safely pass multi-bit data words from one clock domain to another or to control the flow of data between source and destination side sitting in the same clock domain. If read and write clock domains are governed by same clock signal the FIFO is said to be SYNCHRONOUS and if read and write clock domains are governed by different (asynchronous) clock signals FIFO is said to be ASYNCHRONOUS. FIFO full and FIFO empty flags are of great concern as no data should be written in full condition and no data should be read in empty condition, as it can lead to loss of data or generation of non relevant data. The full and empty conditions of FIFO are controlled using binary or gray pointers. In this report we deal with binary pointers only since we are designing SYNCHRONOUS FIFO. The gray pointers are used for generating full and empty conditions for ASYNCHRONOUS FIFO.
**SYNCHRONOUS FIFO**

A synchronous FIFO refers to a FIFO design where data values are written sequentially into a memory array using a clock signal, and the data values are sequentially read out from the memory array using the same clock signal.

**APPLICATIONS**

- FIFO’s are used to safely pass data between two asynchronous clock domains. In System-on-Chip designs there are components which often run on different clocks. So, to pass data from one such component to another we need ASYNCHRONOUS FIFO.

- Some times even if the Source and Requestor sides are controlled by same clock signal a FIFO is needed. This is to match the throughputs of the Source and the Requestor. For example in one case source may be supplying data at rate which the requestor can not handle or in other case requestor may be placing requests for data at a rate at which source can not supply. So, to bridge this gap between source and requestor capacities to supply and consume data a SYNCHRONOUS FIFO is used which acts as an elastic buffer.
BLACK BOX VIEW OF A SYNCHRONOUS FIFO  
(source: http://abramovbenjamin.net/labs/FIFO_design.pdf)

PORT LIST:

<table>
<thead>
<tr>
<th>NAME</th>
<th>I/O</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>I</td>
<td>Clock input to the FIFO. This is common input to both read and write sides of FIFO.</td>
</tr>
<tr>
<td>reset_n</td>
<td>I</td>
<td>active-low asynchronous reset input to FIFO read and write logic</td>
</tr>
<tr>
<td>flush</td>
<td>I</td>
<td>Active-high synchronous flush input to FIFO. A clock-wide pulse resets the FIFO read and write pointers</td>
</tr>
</tbody>
</table>

WRITE SIDE PORTS:

<table>
<thead>
<tr>
<th>NAME</th>
<th>I/O</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>write_data</td>
<td>I</td>
<td>Data input to FIFO</td>
</tr>
<tr>
<td>fifo_full</td>
<td>O</td>
<td>Qualifies the write data. Logic high indicates the data on write_data bus is valid and need to be sampled at next rising edge of the clock.</td>
</tr>
<tr>
<td>write_ack</td>
<td>O</td>
<td>Write acknowledgement to source.</td>
</tr>
</tbody>
</table>
READ SIDE PORTS:

<table>
<thead>
<tr>
<th>NAME</th>
<th>I/O</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_data</td>
<td>O</td>
<td>Read data in response to a read request.</td>
</tr>
<tr>
<td>fifo_empty</td>
<td>O</td>
<td>Indicates that FIFO’s internal memory is empty and therefore has no data to serve upon.</td>
</tr>
</tbody>
</table>

FUNCTIONAL DESCRIPTION:

The above figure depicts the basic building blocks of a synchronous FIFO which are: memory array, write control logic and read control logic. The memory array can be implemented either with array of flip-flops or with a dual-port read/write memory. Both of these implementations allow simultaneous read and write accesses. This simultaneous access gives the FIFO its inherent synchronization property. There are no restrictions regarding timing between accesses of the two ports. This means simply, that while one port is writing to the memory at one rate, the other port
can be reading at another rate totally independent of one another. The only restriction placed is that the simultaneous read and write access should not be from/to the same memory location. The Synchronous FIFO has a single clock port clk for both data-read and data-write operations. Data presented at the module's data-input port write_data is written into the next available empty memory location on a rising clock edge when the write-enable input write_enable is high. The full status output fifo_full indicates that no more empty locations remain in the module's internal memory. Data can be read out of the FIFO via the module's data-output port read_data in the order in which it was written by asserting read-enable signal read_enable prior to a rising clock edge. The memory-empty status output fifo_empty indicates that no more data resides in the module's internal memory. There are almost empty and almost full flags too viz. fifo_aempty and fifo_afull which can be used to control the read and write speeds of the requestor and the source.

WRITE CONTROL LOGIC:

Write Control Logic is used to control the write operation of the FIFO’s internal memory. It generates binary-coded write pointer which points to the memory location where the incoming data is to be written. Write pointer is incremented by one after every successful write operation. Additionally it generates FIFO full and almost full flags which in turn are used to prevent any data loss. For example if a write request comes when FIFO is full then Write Control Logic stalls the write into the memory till the time fifo_full flag gets de-asserted. It intimates the stalling of write to source by not sending any acknowledgement in response to the write request.

READ CONTROL LOGIC:

Read Control Logic is used to control the read operation of the FIFO’s internal memory. It generates binary-coded read pointer which points to the memory location from where the data is to be read. Read pointer is incremented by one after every successful read operation. Additionally it generates FIFO empty and almost empty flags which in turn are used to prevent any spurious data read. For example if a read request comes when FIFO is empty then Read Control Logic stalls the read from the memory till the time fifo_empty flag gets de-asserted. It intimates the stalling of read to the requestor by not asserting rdata_valid in response to the read request.

MEMORY ARRAY:

Memory Array is an array of flip-flops which stores data. Number of data words that the memory array can store is often referred as DEPTH of the FIFO. Length of the data word is referred as WIDTH of the FIFO. Besides flop-array it comprises read and write address decoding logic.
The functionality of Memory Array is relatively straightforward as described below:

1. If write_enable signal is high DATA present on write_data is written into the row pointed by write_addr on the next rising edge of the clock signal clk. Note that write_enable is asserted only when wdata_valid is high and FIFO is not full to avoid any data corruption.

2. If read_enable signal is high the DATA present in the row pointed by read_addr is sent onto the read_data bus on the next rising edge of the clock signal clk. Note that read_enable is asserted only when read_req is high and FIFO is not empty to avoid any spurious data being sent to the requestor.

3. It can handle simultaneous read and write enables as long as their addresses do not match.

FULL AND EMPTY FLAG GENERATION:

FIFO full and almost full flags are generated by Write Control Logic whereas empty and almost empty flags are generated by Read Control Logic. FIFO almost full and almost empty flags are generated to intimate the source and the requestor about impending full or empty conditions. The almost full and almost empty levels are parameterized. It is important to note that read and write pointers point to the same memory location at both full and empty conditions. Therefore, in order to differentiate between the two one extra bit is added to read and write pointers. For example if a FIFO has depth of 256 then to span it completely 8-bits will be needed. Therefore with one extra bit read and write pointers will be of 9-bits. When their lower 8-bits point to same memory location their MSBs are used to ascertain whether it is a full condition or empty condition. In empty conditions the MSBs are equal whereas in full condition MSBs are different.

OVERALL SEQUENCE OF OPERATIONS:

· At reset, both read and write pointers are 0. This is the empty condition of the FIFO, and fifo_empty is pulled high and fifo_full is low.

· At empty, reads are blocked and only operation possible is write.

· Since fifo_full is low, upon seeing a valid write data Write Control Logic will ensure the data be written into location 0 of memory array and write_ptr be incremented to 1. This causes the empty signal to go LOW.

· With fifo_empty pulled down, read operation can now be performed. Upon seeing read request at this state Read Control Logic will fetch data from location 0 and will increment read_ptr to 1.
· In this way read keeps following write until the FIFO gets empty again.

· If write operations are not matched by read soon FIFO will get full and any further write will get stalled until fifo_full is pulled down by a read.
· With the help of FIFO full and empty flags data integrity is maintained between the source and the requestor.

CODE:

I. DEVICE UNDER TEST:

PART A:

//-----------------------------------------------------
// Design Name : syn_fifo
// File Name   : syn_fifo.v
// Function    : Synchronous (single clock) FIFO
//-----------------------------------------------------
module syn_fifo (  
clk      , // Clock input  
rst      , // Active high reset  
wr_cs    , // Write chip select  
rd_cs    , // Read chip select  
data_in  , // Data input  
rd_en    , // Read enable  
wr_en    , // Write Enable  
data_out , // Data Output  
empty    , // FIFO empty  
full     , // FIFO full  
);

// FIFO constants
parameter DATA_WIDTH = 8;
parameter ADDR_WIDTH = 8;
parameter RAM_DEPTH = (1 << ADDR_WIDTH);
// Port Declarations
input clk;
input rst;
input wr_cs;
input rd_cs;
input rd_en;
input wr_en;
input [DATA_WIDTH-1:0] data_in;
output full;
output empty;
output [DATA_WIDTH-1:0] data_out;

//-----------Internal variables-------------------
reg [ADDR_WIDTH-1:0] wr_pointer;
reg [ADDR_WIDTH-1:0] rd_pointer;
reg [ADDR_WIDTH :0] status_cnt;
reg [DATA_WIDTH-1:0] data_out;
wire [DATA_WIDTH-1:0] data_ram;

//-----------Variable assignments--------------
assign full = (status_cnt == (RAM_DEPTH-1));
assign empty = (status_cnt == 0);

//-----------Code Start---------------------
always @ (posedge clk or posedge rst)
begin : WRITE_POINTER
  if (rst) begin
    wr_pointer <= 0;
  end else if (wr_cs && wr_en) begin
    wr_pointer <= wr_pointer + 1;
  end
end

always @ (posedge clk or posedge rst)
begin : READ_POINTER
  if (rst) begin
    rd_pointer <= 0;
  end else if (rd_cs && rd_en) begin
    rd_pointer <= rd_pointer + 1;
  end
end
always @(posedge clk or posedge rst)
begin : READ_DATA
  if (rst) begin
    data_out <= 0;
  end else if (rd_cs && rd_en) begin
    data_out <= data_ram;
  end
end

always @(posedge clk or posedge rst)
begin : STATUS_COUNTER
  if (rst) begin
    status_cnt <= 0;
    // Read but no write.
    end else if ((rd_cs && rd_en) && !(wr_cs && wr_en)
      && (status_cnt != 0)) begin
      status_cnt <= status_cnt - 1;
    // Write but no read.
    end else if ((wr_cs && wr_en) && !(rd_cs && rd_en)
      && (status_cnt != RAMDEPTH)) begin
        status_cnt <= status_cnt + 1;
    end
end

ram_dp_ar_aw #(DATA_WIDTH,ADDR_WIDTH)DP_RAM ( 
  .address_0 (wr_pointer), // address_0 input
  .data_0    (data_in),    // data_0 bi-directional
  .cs_0      (wr_cs),      // chip select
  .we_0      (wr_en),      // write enable
  .oe_0      (1'b0),       // output enable
  .address_1 (rd_pointer), // address_q input
  .data_1    (data_ram),   // data_1 bi-directional
  .cs_1      (rd_cs),      // chip select
  .we_1      (1'b0),       // Read enable
  .oe_1      (rd_en)       // output enable
);
endmodule
PART B:

//-----------------------------------------------------
// Design Name : ram_dp_ar_aw
// File Name   : ram_dp_ar_aw.v
// Function    : Asynchronous read write RAM
//-----------------------------------------------------

module ram_dp_ar_aw (  
address_0 , // address_0 Input  
data_0    , // data_0 bi-directional  
 cs_0    , // Chip Select  
we_0      , // Write Enable/Read Enable  
 oe_0      , // Output Enable  
address_1 , // address_1 Input  
data_1    , // data_1 bi-directional  
 cs_1    , // Chip Select  
we_1      , // Write Enable/Read Enable  
 oe_1      // Output Enable  
);

parameter DATA_WIDTH = 8 ;
parameter ADDR_WIDTH = 8 ;
parameter RAM_DEPTH = 1 << ADDR_WIDTH;

//------------------Input Ports--------------------------
in input [ADDR_WIDTH-1:0] address_0 ;
in input cs_0 ;
in input we_0 ;
in input oe_0 ;
in input [ADDR_WIDTH-1:0] address_1 ;
in input cs_1 ;
in input we_1 ;
in input oe_1 ;

//------------------Inout Ports--------------------------
inout [DATA_WIDTH-1:0] data_0 ;
inout [DATA_WIDTH-1:0] data_1 ;

//------------------Internal variables------------------
reg [DATA_WIDTH-1:0] data_0_out ;
reg [DATA_WIDTH-1:0] data_1_out;
reg [DATA_WIDTH-1:0] mem [0:RAM_DEPTH-1];

//--------------Code Starts Here------------------
// Memory Write Block
// Write Operation : When we_0 = 1, cs_0 = 1
always @ (address_0 or cs_0 or we_0 or data_0
or address_1 or cs_1 or we_1 or data_1)
begin : MEM_WRITE
  if ( cs_0 && we_0 ) begin
    mem[address_0] <= data_0;
  end else if (cs_1 && we_1) begin
    mem[address_1] <= data_1;
  end
end

// Tri-State Buffer control
// output : When we_0 = 0, oe_0 = 1, cs_0 = 1
assign data_0 = (cs_0 && oe_0 && !we_0) ? data_0_out : 8'bz;

// Memory Read Block
// Read Operation : When we_0 = 0, oe_0 = 1, cs_0 = 1
always @ (address_0 or cs_0 or we_1 or oe_0)
begin : MEM_READ_0
  if (cs_0 && !we_0 && oe_0) begin
    data_0_out <= mem[address_0];
  end else begin
    data_0_out <= 0;
  end
end

//Second Port of RAM
// Tri-State Buffer control
// output : When we_0 = 0, oe_0 = 1, cs_0 = 1
assign data_1 = (cs_1 && oe_1 && !we_1) ? data_1_out : 8'bz;
// Memory Read Block 1
// Read Operation : When we_1 = 0, oe_1 = 1, cs_1 = 1
always @ (address_1 or cs_1 or we_1 or oe_1)
begin : MEM_READ_1
  if (cs_1 && !we_1 && oe_1) begin
    data_1_out <= mem[address_1];
  end else begin
    data_1_out <= 0;
  end
end

dendmodule // End of Module ram_dp_ar_aw

II. SYSTEM VERILOG TEST BENCH :

`include "fifo_ports.sv"

program fifo_top (fifo_ports ports, fifo_monitor_ports mports);
  `include "fifo_sb.sv"
  `include "fifo_driver.sv"

  fifo_driver driver = new(ports, mports);

  initial begin
    driver.go();
  end

endprogram
III. FIFO SCORE BOARD

This section of the code basically detects the overflow and the underflow condition. Here if the FIFO buffer is full, it indicates by giving a status that the FIFO cannot handle extra request i.e no further push operation can be performed. Also if the FIFO buffer is empty and if we try to remove any data it indicated it as an error i.e no further pop operation can be performed.

class fifo_sb;
    mailbox fifo = new();
    integer size;

    function new();
    begin
        size = 0;
    end
    endfunction

    task addItem(bit [7:0] data);
    begin
        if (size == 7) begin
            $write("%dns : ERROR : Over flow detected, current occupancy %d\n",
                $time, size);
        end else begin
            fifo.put(data);
            size ++;
        end
    end
    endtask

    task compareItem (bit [7:0] data);
    begin
        bit [7:0] cdata  = 0;
        if (size == 0) begin
            $write("%dns : ERROR : Under flow detected\n",
                $time);
        end else begin


fifo.get (cdata);
if (data != cdata) begin
    $write("%dns : ERROR : Data mismatch, Expected %x
Got %x\n",
        $time, cdata, data);
    end
    size --;
end
endtask
endclass

IV. FIFO DRIVER

class fifo_driver;
    fifo_sb sb;
    virtual fifo_ports ports;
    virtual fifo_monitor_ports mports;

    bit rdDone;
    bit wrDone;

    integer wr_cmds;
    integer rd_cmds;

    function new (virtual fifo_ports ports, virtual
                      fifo_monitor_ports mports);
        begin
            this.ports = ports;
            this.mports = mports;
            sb = new();
            wr_cmds = 5;
            rd_cmds = 5;
            rdDone = 0;
            wrDone = 0;
            ports.wr_cs  = 0;
            ports.rd_cs  = 0;
            ports.wr_en  = 0;
        end
```
ports.rd_en  = 0;
ports.data_in  = 0;
end
endfunction

task monitorPush();
begin
    bit [7:0] data = 0;
    while (1) begin
        @ (posedge mports.clk);
        if (mports.wr_cs== 1 &&  mports.wr_en== 1) begin
            data = mports.data_in;
            sb.addItem(data);
            $write("%dns : Write posting to scoreboard data = %x

",$time, data);
        end
    end
end
tendtask

task monitorPop();
begin
    bit [7:0] data = 0;
    while (1) begin
        @ (posedge mports.clk);
        if (mports.rd_cs== 1 &&  mports.rd_en== 1) begin
            data = mports.data_out;
            $write("%dns : Read posting to scoreboard data = %x

",$time, data);
            sb.compareItem(data);
        end
    end
end
tendtask

task go();
begin
    // Assert reset first
    reset();
    // Start the monitors
```
repeat (5) @ (posedge ports.clk);
$write("%dns : Starting Pop and Push monitors\n",$time);
fork
  monitorPop();
  monitorPush();
join_none
$write("%dns : Starting Pop and Push generators\n",$time);
fork
  genPush();
  genPop();
join_none

while (!rdDone && !wrDone) begin
  @ (posedge ports.clk);
end
repeat (10) @ (posedge ports.clk);
$write("%dns : Terminating simulations\n",$time);
end
endtask

task reset();
begin
  repeat (5) @ (posedge ports.clk);
  $write("%dns : Asserting reset\n",$time);
  ports.rst= 1'b1;
  // Init all variables
  rdDone = 0;
  wrDone = 0;
  repeat (5) @ (posedge ports.clk);
  ports.rst= 1'b0;
  $write("%dns : Done asserting reset\n",$time);
end
endtask

task genPush();
begin
  bit [7:0] data = 0;
  integer i = 0;
for (i = 0; i < wr_cmds; i++) begin
  data = $random();
  @(posedge ports.clk);
  while (ports.full == 1'b1) begin
    ports.wr_cs = 1'b0;
    ports.wr_en = 1'b0;
    ports.data_in = 8'b0;
    @(posedge ports.clk);
  end
  ports.wr_cs = 1'b1;
  ports.wr_en = 1'b1;
  ports.data_in = data;
end
@(posedge ports.clk);
ports.wr_cs = 1'b0;
ports.wr_en = 1'b0;
ports.data_in = 8'b0;
repeat (10) @(posedge ports.clk);
wrDone = 1;
end
endtask
task genPop();
begin
  integer i = 0;
  for (i = 0; i < rd_cmds; i++) begin
    @(posedge ports.clk);
    while (ports.empty == 1'b1) begin
      ports.rd_cs = 1'b0;
      ports.rd_en = 1'b0;
      @(posedge ports.clk);
    end
    ports.rd_cs = 1'b1;
    ports.rd_en = 1'b1;
  end
  @(posedge ports.clk);
  ports.rd_cs = 1'b0;
  ports.rd_en = 1'b0;
  repeat (10) @(posedge ports.clk);
  rdDone = 1;
end
detask
class

V. PORTS FILE

ifndef FIFO_PORTS_SV
define FIFO_PORTS_SV

interface fifo_ports (  
    input wire  clk 
    output logic rst 
    input wire  full 
    input wire  empty 
    output logic wr_cs 
    output logic rd_cs 
    output logic rd_en 
    output logic wr_en 
    output logic [7:0] data_in 
    input wire [7:0] data_out 
);
endinterface

interface fifo_monitor_ports (  
    input wire  clk 
    input wire  rst 
    input wire  full 
    input wire  empty 
    input wire  wr_cs 
    input wire  rd_cs 
    input wire  rd_en 
    input wire  wr_en 
    input wire [7:0] data_in 
    input wire [7:0] data_out 
);
endinterface

endif
OUTPUT:

The above code has been verified using Questa Sim Simulator and the following output is obtained.

# 9ns : Asserting reset
# 19ns : Done asserting reset
# 29ns : Starting Pop and Push monitors
# 29ns : Starting Pop and Push generators
# 33ns : Write posting to scoreboard data = 24
# 35ns : Read posting to scoreboard data = 24
# 35ns : Write posting to scoreboard data = 81
# 37ns : Read posting to scoreboard data = 81
# 37ns : Write posting to scoreboard data = 09
# 39ns : Read posting to scoreboard data = 09
# 39ns : Write posting to scoreboard data = 63
# 41ns : Read posting to scoreboard data = 63
# 41ns : Write posting to scoreboard data = 0d
# 43ns : Read posting to scoreboard data = 0d
# 83ns : Terminating simulations
# 1
# Simulation stop requested
CONCLUSION:

The design of synchronous FIFO was studied and also verified using System Verilog. The following concepts of System Verilog were used in our project:

1. Class
2. Interface
3. Randomization
4. Ports
5. Program

The verification components in our project were split by the following blocks:

1. Push Generator
2. Pop Generator
3. Push Monitor
4. Pop Monitor
5. Scoreboard
6. System Verilog Testbench Top
7. System Verilog Interface File

REFERENCES:


