Module 4

FSM and memory modelling

Contents

Synchronous FIFO

First In First Out (FIFO) is a very popular and useful design block for purpose of synchronization and a handshaking mechanism between the modules.

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

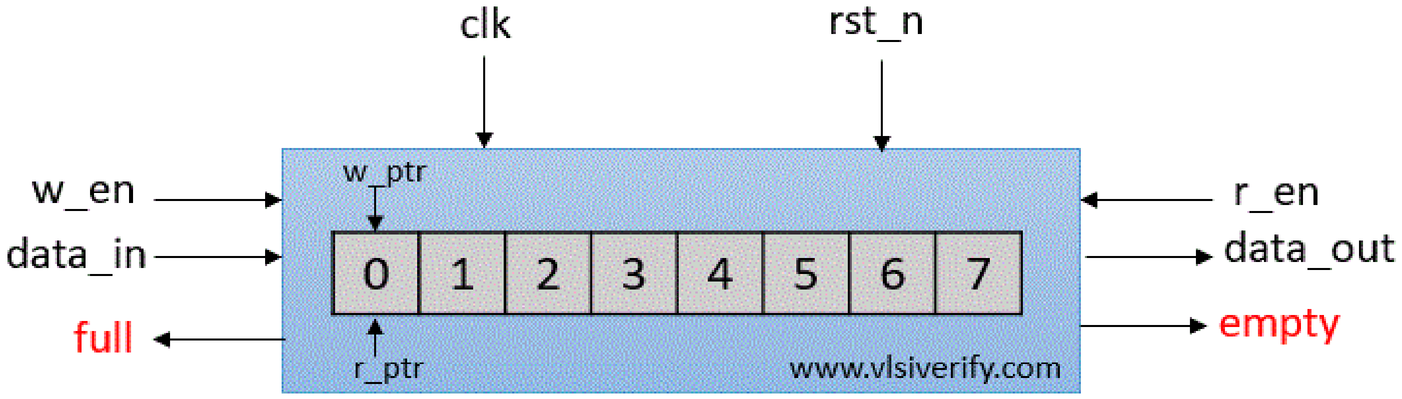
Width of FIFO: The number of bits that can be stored in each slot or row is called the width of the FIFO.

There are two types of FIFOs

1. Synchronous FIFO
2. Asynchronous FIFO

Synchronous FIFO

In Synchronous FIFO, data read and write operations use the same clock frequency. Usually, they are used with high clock frequency to support high-speed systems.



Synchronous FIFO

Signals:

wr_en: write enable

wr_data: write data

full: FIFO is full

empty: FIFO is empty

rd_en: read enable

rd_data: read data

w_ptr: write pointer

r_ptr: read pointer

FIFO write operation

FIFO can store/write the wr_data at every posedge of the clock based on wr_en signal till it is full. The write pointer gets incremented on every data write in FIFO memory.

FIFO read operation

The data can be taken out or read from FIFO at every posedge of the clock based on the rd_en signal till it is empty. The read pointer gets incremented on every data read from FIFO memory.

The width of the write and read pointer = log2(depth of FIFO). Code Link

The FIFO full and empty conditions can be determined as

Empty condition

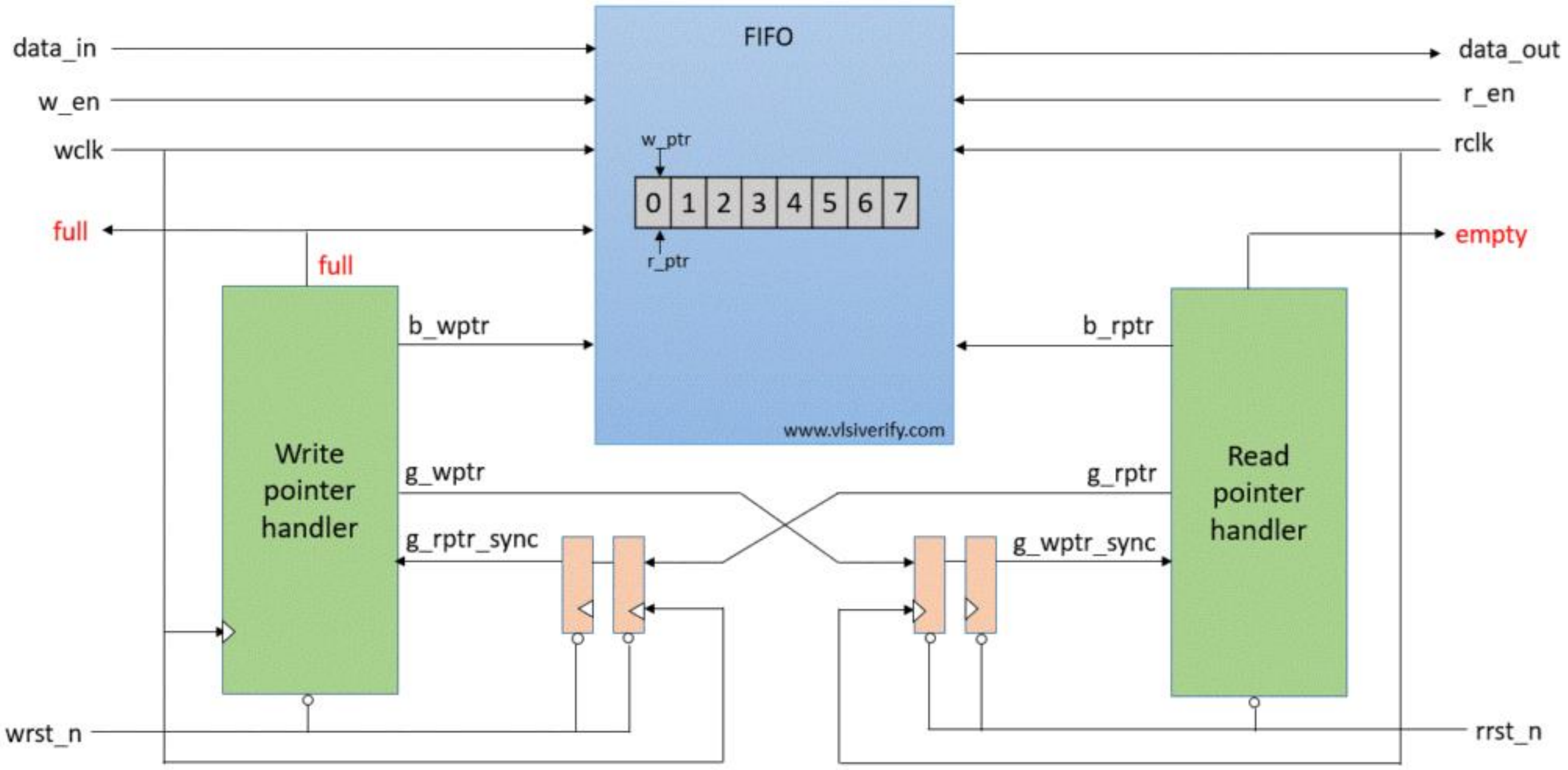
w_ptr == r_ptr i.e. write and read pointers has the same value.

Full condition

The full condition means every slot in the FIFO is occupied, but then w_ptr and r_ptr will again have the same value. Thus, it is not possible to determine whether it is a full or empty condition. Thus, the last slot of FIFO is intentionally kept empty, and the full condition can be written as (w_ptr+1'b1) == r_ptr)

Asynchronous FIFO

- In asynchronous FIFO, data read and write operations use different clock frequencies.
- 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.



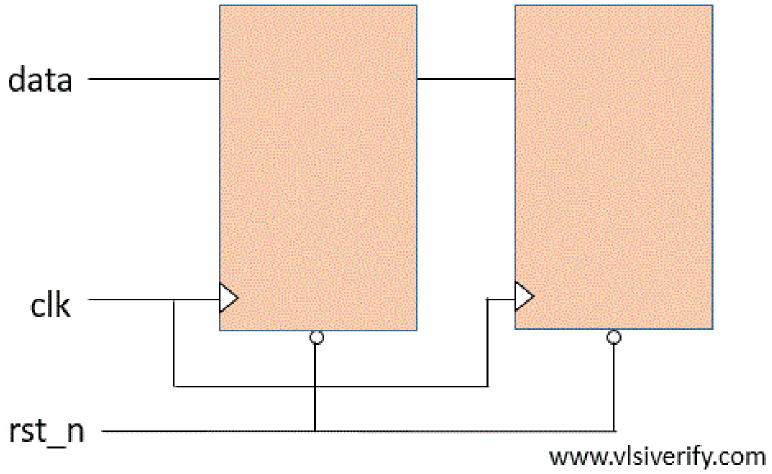
Asynchronous FIFO

Asynchronous FIFO Operation

- In the case of asynchronous FIFO write pointer is aligned to the write clock domain whereas the read pointer is aligned to the read clock domain.
- Hence, it requires domain crossing to calculate FIFO full and empty conditions.
- This causes metastability in the actual design.
- In order to resolve this metastability, 2 flip flops or 3 flip flops synchronizer can be used to pass write and read pointers.

A single “2 FF synchronizer” can resolve metastability for only one bit.

Hence, depending on write and read pointers multiple 2FF synchronizers are required.



2 flip-flop synchronizer

Usage of Gray codes

- Binary formatted write and read pointer values cannot be passed.
- Due to metastability, the overall write or read pointer value might be different.
- Both write and read pointers need to convert first to their equivalent gray code in their corresponding domain and then pass them to an opposite domain.

Memory

- Read-only memory (ROM)
- Random Access Memory (RAM)

- Single port RAM
- Dual port RAM

Single Port RAM

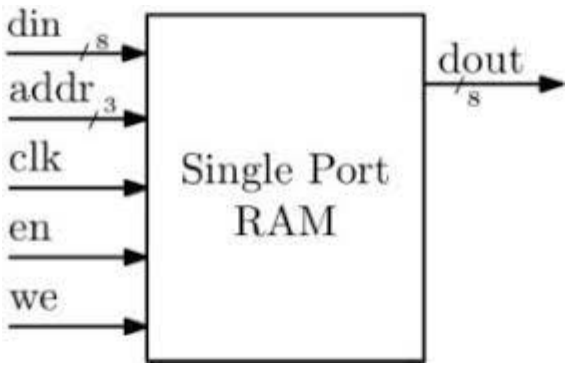
- A single-port RAM (Random Access Memory) is a type of digital memory component that allows data to be read from and written to a single memory location (address) at a time.
- It is a simple form of memory that provides a basic storage mechanism for digital systems.
- Each memory location in a single-port RAM can store a fixed number of bits (usually a power of 2, such as 8, 16, 32, etc.).

Single Port RAM

- During a read operation, the data stored at a specific address is retrieved.
- During a write operation, new data is stored at a specific address, replacing the previous data.
- Single-port RAMs are commonly used in various digital systems for tasks such as data storage, temporary buffering, and data manipulation.

Signals

- Single-port RAMs have address lines that are used to select the memory location to be accessed. The number of address lines determines the maximum number of memory locations that the RAM can hold.
- Data lines are used to carry the actual data to be read from or written to the memory location.
- Control signals, such as read enable (read request) and write enable (write request), are used to initiate specific memory operations.



Verilog Code

```
module single_port_ram(
  input [7:0] data,         //input data
  input [5:0] addr,         //address
  input we,         //write enable
  input clk,        //clk
  output [7:0] q //output data
);

  reg [7:0] ram [63:0];             //8*64 bit ram
  reg [5:0] addr_reg;
                                  //address register
```

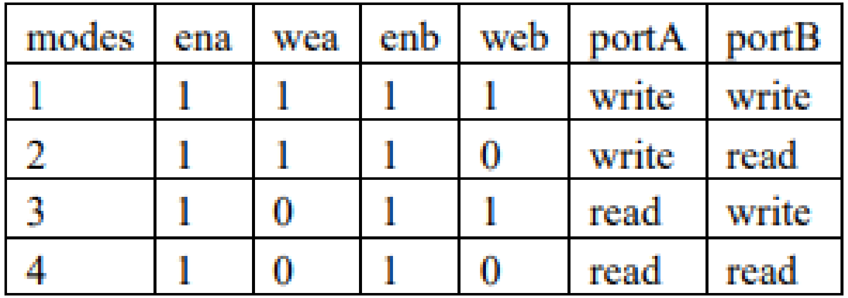
```
always @ (posedge clk)
   begin
     if(we)
       ram[addr] <= data;
     else
       addr_reg <= addr;
   end

  assign q = ram[addr_reg];

endmodule
```

```
module single_port_ram_tb;
  reg [7:0] data;                    //input data
  reg [5:0] addr;                   //address
  reg we;                                //write enable
  reg clk;                               //clk
  wire [7:0] q;                     //output data

  single_port_ram spr1(
    .data(data),
    .addr(addr),
    .we(we),
    .clk(clk),
    .q(q)
  );

  initial
    begin
      $dumpfile("dump.vcd");
      $dumpvars(1, single_port_ram_tb);

      clk=1'b1;
      forever #5 clk = ~clk;
    end
```

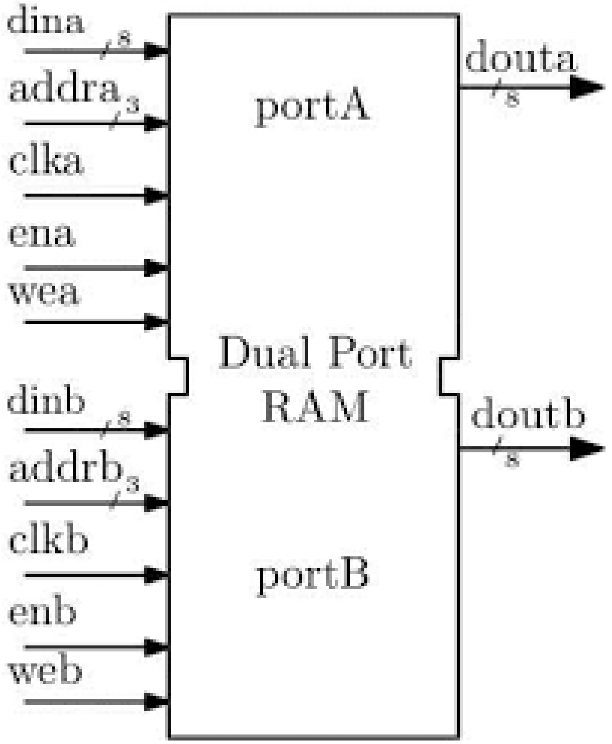
```
 initial
begin
                  we = 1’b1;

      data = 8'h01;
      addr = 5’d0;


      #10;
      data = 8'h02;
      addr = 5'd1;
      #10;

      data = 8'h03;
      addr = 5'd2;
      #10;

      addr = 5'd0;
      we = 1'b0;
      #10;

      addr = 5'd1;
      #10;

end
endmodule
```

Dual Port RAM

- Possible to access the same address locations through two ports.
- The various modes of a typical true dual port RAM is shown below.



modes	ena	wea	enb	web	portA	portB
1	1	1	1	1	write	write
2	1	1	1	0	write	read
3	1	0	1	1	read	write
4	1	0	1	0	read	read



```
module dp_ram (
clka,clkb,
ada,adb,
ina,inb,
ena,enb,
wea,web,
outa,outb);

input clka,clkb,ena,wea,enb,web;
input[2:0] ada,adb;
input[7:0] ina,inb;

output reg [7:0] outa,outb;

reg [7:0] mem [0:7];

initial begin
outa = 8'b00000000;
outb = 8'b00000000;
end
```

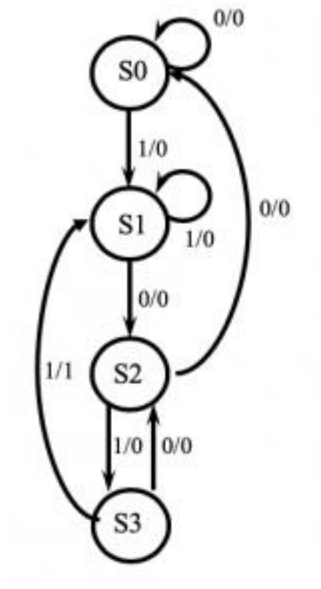
```
always@(posedge clka)
if(ena)
begin
if(wea)
mem[ada]=ina;
else
outa = mem[ada];
end
else
outa = outa;
always@(posedge clkb)
if(enb)
begin
if(web)
mem[adb]=inb;
else
outb = mem[adb];
end
else
outb = outb;
endmodule
```

SEQUENCE DETECTOR

- A sequence detector is a sequential state machine that takes an input string of bits and generates an output 1 whenever the target sequence has been detected.
- In a Mealy machine, output depends on the present state and the external input (x).
- Sequence detector is of two types:
  - Overlapping
  - Non-Overlapping
- In an overlapping sequence detector, the last bit of one sequence becomes the first bit of the next sequence.
- In a non-overlapping sequence detector, the last bit of one sequence does not become the first bit of the next sequence.

Example

- Design a FSM which detects the sequence “1011” from a given bit stream. Explain with neat state diagram for overlapping case.
- State Diagram



Verilog Code

```
module sd1011_mealy_over(
          input bit clk,
          input logic reset,
          input logic din,
          output logic dout);

  typedef enum logic [1:0]{S0, S1, S2, S3}
 state_t;
  state_t state;

  always @(posedge clk or posedge reset)
begin
    if(reset) begin
      dout <= 1'b0;
      state <= S0;
    end
    else begin
      case(state)
 S0: begin
         if(din) begin
           state <= S1;
           dout <=1'b0;
         end
         else
           dout <=1'b0;
       end
S1: begin
         if(~din) begin
           state <= S2;
           dout <=1'b0;
         end
         else begin
           dout <=1'b0;
         end
       end
S2: begin
         if(~din) begin
           state <= S0;
           dout <=1'b0;
         end
         else begin
           state <= S3;
           dout <=1'b0;
         end
       end
       S3: begin
         if(din) begin
           state <= S1;
           dout <=1'b1;
         end
         else begin
           state <= S2;
           dout <=1'b0;
         end
       end
      endcase
    end
  end   endmodule
```

Vending Machine

- Newspaper vending machine.
- In this wending machine, it accepts only two coins, 5 point and 10 point. Whenever total of coins equal to 15 points, then nw_pa signal will go high and user will get news paper. It will not return any coin, if total of points exceeds 15 points.
- System Specification:

Sr. No.	**Name of the Pin**	**Direction**	**Width**	**Description**
1	Nw_pa	Output	1	News Paper Signal
2	Coin	Input	2	Only two Coins, 5 =2'b01 10 =2'b10 0 =2'b00
3	Clk	Input	1	Clock Signal
4	Rst	Input	1	Reset Signal

```
modulevending_machine
                    (nw_pa,clk,coin,rst);
output reg nw_pa;
input [1:0] coin;
input clk,rst; reg [1:0] state;
reg [1:0] next_state;

parameter [1:0] s0=2'b00;
parameter [1:0] s5=2'b01;
parameter [1:0] s10=2'b10;
parameter [1:0] s15=2'b11;

always @(posedge clk)
begin
      if (rst)
      state=s0;
      else
      state=next_state;
end
always @(state,coin)
begin
case (state)
s0: begin
  if (coin==2'b00)
       next_state=s0;
  else if (coin==2'b01)
        next_state=s5;
  else if (coin==2'b10)
        next_state=s10;
end
s5: begin
   if (coin==2'b00)
        next_state=s5;
   else if (coin==2'b01)
       next_state=s10;
   else if (coin==2’b10)
       next_state=s15;
end
s10: begin
if (coin==2'b00)
next_state=s10;
else if (coin==2'b01)
next_state=s15;
else if (coin==2'b10)
next_state=s15;
end
s15: begin
next_state=s0;
end
default : next_state=s0;
endcase    end
always @(state)
begin
case (state)
s0 : nw_pa<=1'b0;
s5 : nw_pa<=1'b0;
s10: nw_pa<=1'b0;
s15: nw_pa<=1'b1;
default: nw_pa<=1'b0; endcase end
endmodule
```