שפת תכנון חומרה
ורילוג - Verilog

Verilog – Memory Module

Dr. Avihai Aharon

Data Types: Memories

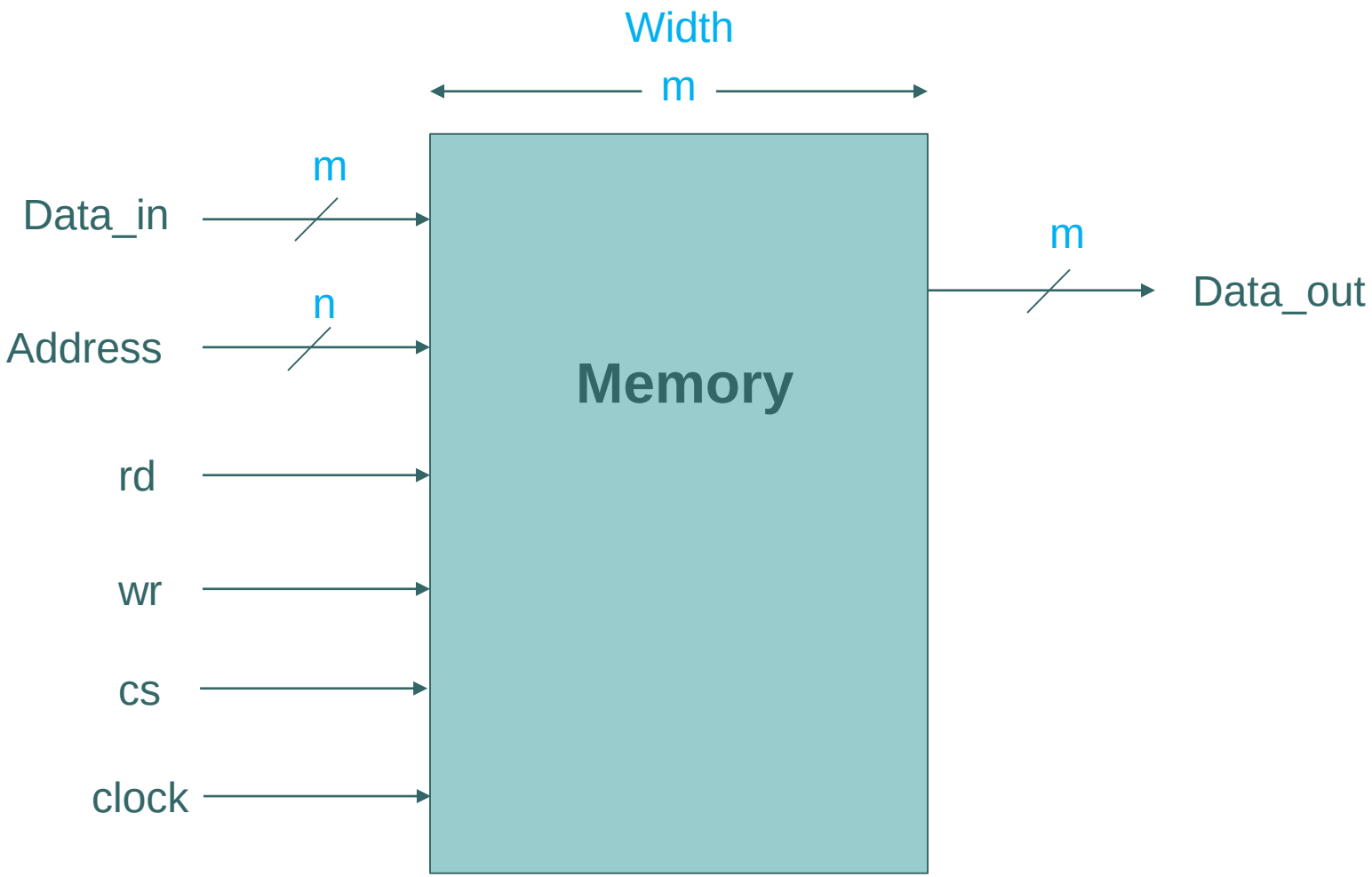
- **Memories are modeled in Verilog simply as an array of registers**
- **Each element of the array is known as a word. Each word can be one or more bits**
- **It is important not to confuse arrays with net or register vectors**
  - **vector: n-bits wide single element**
  - **array: 1-bit or n-bits wide multiple elements**
- **It is important to differentiate between n 1-bit registers and one n-bit register** (reg [7:0] regbyte)

```
reg mem1bit [0:1023];          // memory mem1bit with 1k 1-bit words

reg [7:0] membyte [0:1023];  // memory named membyte with 1k 8-bit words (bytes)
```

Here [7:0] is the memory width and [0:255] is the memory depth with the following parameters:
Width : 8 bits,
Depth : 256 , address 0 corresponds to location 0 in the array.

```
membyte[address] = data_in;        //Storing Values to memory
data_out = membyte[address];          //Reading Values from memory
```

Memory block



Depth = 2^n

Single Port RAM Synchronous Read/Write

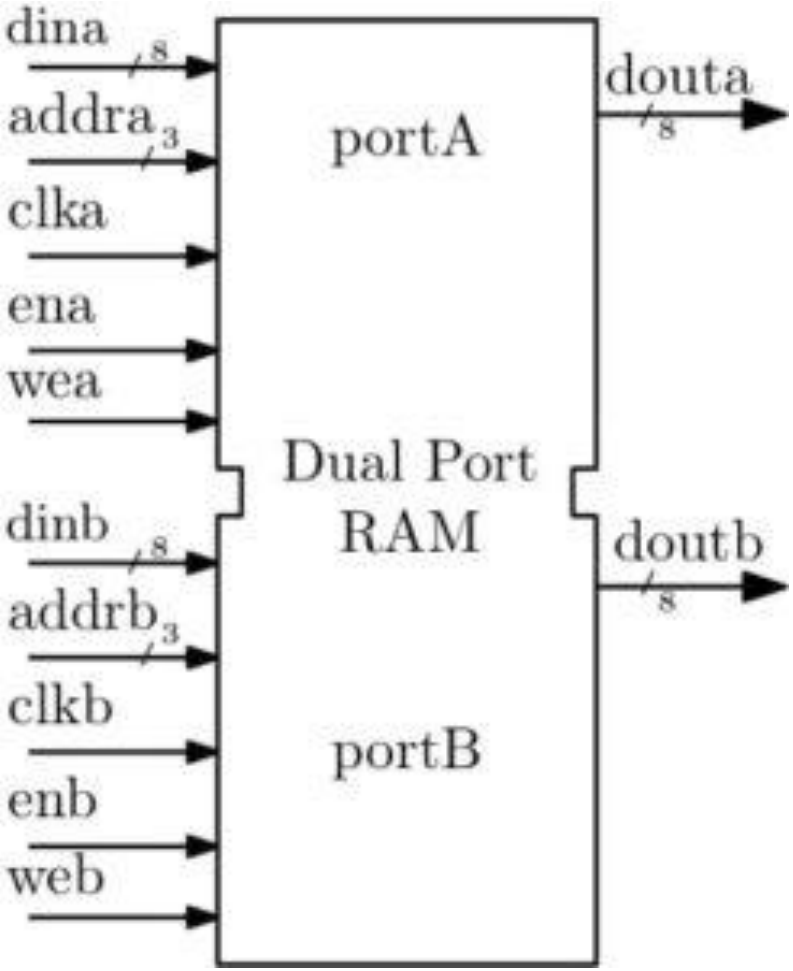
```
//----------------------------------------
module ram_sp_sr_sw (
clk          , // Clock Input
address      , // Address Input
data         , // Data bi-directional
cs           , // Chip Select
we           , // Write Enable/Read Enable
oe             // Output Enable
);

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

//--------------Input Ports--------------
input                    clk         ;
input [ADDR_WIDTH-1:0] address       ;
input                    cs          ;
input                    we          ;
input                    oe          ;

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

//--------------Internal variables--------
reg [DATA_WIDTH-1:0] data_out ;
reg [DATA_WIDTH-1:0] mem [0:RAM_DEPTH-1];
```

```
//--------------Code Starts Here------------------

// Tri-State Buffer control
// output : When we = 0, oe = 1, cs = 1
assign data = (cs && oe && !we) ? data_out : 8'bz;

// Memory Write Block
// Write Operation : When we = 1, cs = 1
always @ (posedge clk)
begin : MEM_WRITE
   if ( cs && we ) begin
        mem[address] = data;
   end
end

// Memory Read Block
// Read Operation : When we = 0, oe = 1, cs = 1
always @ (posedge clk)
begin : MEM_READ
  if (cs && !we && oe) begin
        data_out = mem[address];
  end
end

endmodule // End of Module ram_sp_sr_sw
```

Single Port RAM Asynch Read, Synch Write

```
//---------------Code Starts Here------------------

// Tri-State Buffer control
// output : When we = 0, oe = 1, cs = 1
assign data = (cs && oe && !we) ? data_out : 8'bz;

// Memory Write Block
// Write Operation : When we = 1, cs = 1
always @ (posedge clk)
begin : MEM_WRITE
   if ( cs && we ) begin
        mem[address] = data;
   end
end

// Memory Read Block
// Read Operation : When we = 0, oe = 1, cs = 1
always @ (address or cs or we or oe)
begin : MEM_READ
    if (cs && !we && oe) begin
        data_out = mem[address];
    end
end

endmodule // End of Module ram_sp_ar_sw
```

Dual Port RAM



ROM - Loading from File

```
//-----------------------------------------------------
module rom_using_file (
address , // Address input
data    , // Data output
read_en , // Read Enable
ce        // Chip Enable
);
input [7:0] address;
output [7:0] data;
input read_en;
input ce;

reg [7:0] mem [0:255] ;

assign data = (ce && read_en) ? mem[address] : 8'b0;

initial begin
  $readmemb("memory.list", mem); // memory_list is memory file
end

endmodule
```

```
//Comments are allowed
1100_1100   // This is first address i.e 8'h00
1010_1010   // This is second address i.e 8'h01
@ 55        // Jump to new address 8'h55
0101_1010   // This is address 8'h55
0110_1001   // This is address 8'h56
```

memory.list:

```
00000000
00000001
00000010
00000011
.
.
.
11111111
```

$**readmemb** is used for binary representation of memory content and $**readmemh** for hex representation.

Pay attention to change the file type to **list** by changing the extension (folder->view->extension √).

ROM - Loading from File - TB

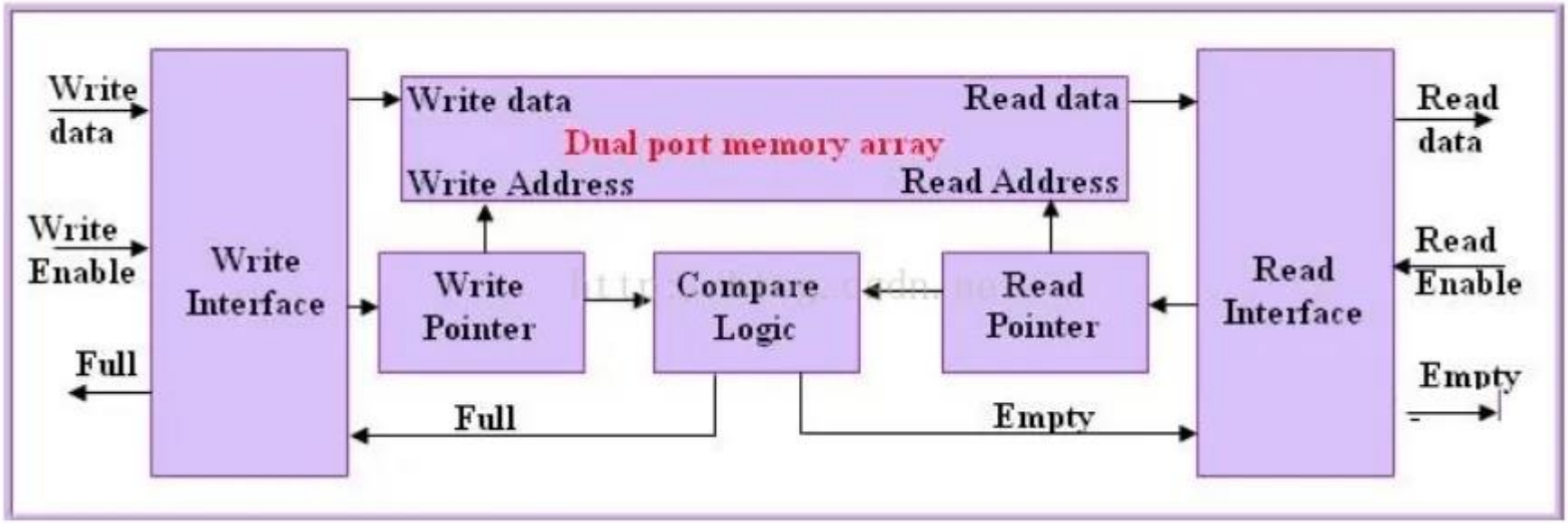
```
module rom_tb;
 reg [7:0] address;
 reg read_en, ce;
 wire [7:0] data;
 integer i;

 initial begin
   address = 0;
   read_en = 0;
   ce      = 0;
   for (i = 0; i <256; i = i +1 )begin
     #5 address = i;
     read_en = 1;
     ce = 1;
     #5 read_en = 0;
     ce = 0;
     address = 0;
   end
 end

rom DUT(
address , // Address input
data    , // Data output
read_en , // Read Enable
ce        // Chip Enable
);
```

ROM – using case

```
//--------------------------------
module rom_using_case (
address , // Address input
data    , // Data output
read_en , // Read Enable
ce        // Chip Enable
);
input [3:0] address;
output [7:0] data;
input read_en;
input ce;

reg [7:0] data ;
```

```
always @ (ce or read_en or address)
begin
  case (address)
    0 : data = 10;
    1 : data = 55;
    2 : data = 244;
    3 : data = 0;
    4 : data = 1;
    5 : data = 8'hff;
    6 : data = 8'h11;
    7 : data = 8'h1;
    8 : data = 8'h10;
    9 : data = 8'h0;
    10 : data = 8'h10;
    11 : data = 8'h15;
    12 : data = 8'h60;
    13 : data = 8'h90;
    14 : data = 8'h70;
    15 : data = 8'h90;
  endcase
end

endmodule
```

Synchronous FIFO



Name of the Pin	Direction	Width	Description
Rst_a	Input	1	Reset Input
Clk	Input	1	Clock Input
wr_en	Input	1	when high write into fifo
rd_en	input	1	when high read from memory
Data_in	Input	4	Data Input
Data_out	Output	4	Data output
Full	Output	1	Fifo status 1 if fifo is full
Empty	Output	1	Fifo status 1 if fifo is empty

Synchronous FIFO

```
//----------------------------------------
module syn_fifo (
clk       , // Clock input
rst       , // Active high reset
wr_cs     , // Write chip select
rd_cs     , // Read chipe 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);
```

Synchronous FIFO

```
//-----------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
```

Synchronous FIFO

```
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 != RAM_DEPTH)) begin
    status_cnt <= status_cnt + 1;
  end
end

//Dual Port RAM
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
```

Class exercises

1. להכין קובץ נתונים לקוד rom_using_file ולקרוא אותו (ניתן להקטין את עומק הזיכרון – יש לשים לב לרוחב הכתובת שישתנה בהתאם).
   - יש לשים לב לשנות את הסיומת ל-List ע"פ ההנחיות.
2. טסטבנץ' למערכת rom_using_file כאשר בודקים תוכן של כמה כתובות שהוגדרו מראש (לדוגמא נתונים מכתובות 0,3,6,8,11,14 בזיכרון)
3. טסטבנץ' למערכת ה-FIFO, יש לבדוק את מצבים הקיימים בFIFO כמוכן להציג בדיאגרמת הזמנים את הסיגנלים הפנימיים (כגון: פוינטרים, CNT וכו')