Verilog

Quick Reference Guide

Author:Susmita Nayak

Email:tech_support@maven-silicon.com

www.maven-silicon.com

Maven Silicon Confidential

All the presentations, books, documents [hard copies and soft copies] labs and projects [source code] that you are using and developing as part of the training course are the proprietary work of Maven Silicon and it is fully protected under copyright and trade secret laws. You may not view, use, disclose, copy, or distribute the materials or any information except pursuant to a valid written license from Maven Silicon

Table of Contents

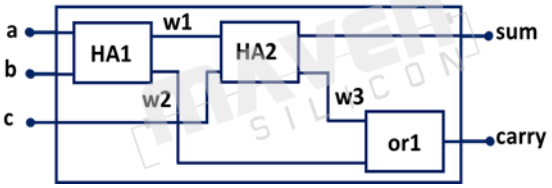
Data-types

1.1 Nets: wire

1.1.1 wire

wire is continuously driven by combinational logic. The default value of a wire is “z”.

```
/*The LHS of a continuous concurrent assignment
  should be a wire*/
wire a,b,y;
assign y = a && b;
```




```
/* w1,w2,w3 are internal wires
   which connects the instances HA1,HA2,Gate primitives*/
wire w1,w2,w3;
```

1.2 Registers/Variables: {reg, integer, real, time, string}

1.2.1 reg

Reg is a variable which retains value till it is updated. The default value of reg is “x”.

```
/*The LHS of an assignment inside a
  procedural block should be reg type*/
reg y;
wire a,b;

initial
  begin
    y = a || b;
  end
```

```
/*FFs are modelled using reg data-type*/
 reg y;
 always@(posedge clk)
  begin
    if(reset)
      y <= 0;
    else
      y <= a;
  end
```

1.2.2 integer

Integers are signed values. The default size is 32bits. The default value of an integer is “x”.

```
 /*Integer types are mostly used
   in loops for the iteration process*/

 integer i;

 initial
   begin
     for(i=0;i<10;i=i+1)
       begin
         #10;
         a = i;
       end
   end
```

1.2.3 real

Real numbers are expressed with a decimal point shall have atleast one digit on each side of the decimal point. The default value of real is 0.

```
 /*Real numbers in decimal & scientific notation*/
   real a,b;

   initial
     begin
       a = 3.9;
       b = 1.3e-2;
     end
```

1.2.4 time

Time variables shall behave same as reg of atleast 64 bits. It is unsigned with default value as "x".

```
/*Time data-type stores simulation time
  $time is a system function which returns
  simulation time*/
time t;

initial
  begin
    #10 t = $time; //t stores 10 time units
    #20 t = $time; //t stores 30 time units
  end
```

1.2.5 string

Strings are sequence of characters which are enclosed within double quotes " " and each character is stored as 8 bits ASCII value. They are stored as reg type variables.

```
/*8 bit ASCII value represents one character*/
reg [8*13:1]string_reg;
initial
  begin
    string_reg = "Maven Silicon";
    $display("Company Name -> %s",string_reg);
  end
```

1.3 Vectors

```
/*Vector represents buses*/
 reg  [3:0]a; //reg vector with msb:lsb {width of the bus}
 wire [2:0]b; //net vector with msb:lsb {width of the bus}
```

1.4 Arrays

```
reg [7:0]ram[0:255]; /* Declares "ram" as an array of
                        256 8 bit registers.
                        The depth is 256 and width
                        of each register is 8bits*/
```

1.5 Parameter constants

1.5.1 Parameter overriding

```
/*Parameter constants can't be changed
  during simulation run phase */
module ram(clk,rst,din,radd,wadd,dout);

  parameter WIDTH = 8;   // Defines WIDTH as a constant value 8
  parameter DEPTH = 256; // Defines DEPTH as a constant value 256
  reg [(WIDTH - 1):0]mem[(DEPTH - 1):0];//Declaring a 256X8 memory using parameters
  ……
endmodule
```

```
 /*Parameter overriding using order based*/

 module system(…);
   input   rst,clk;
   output  dout,done;
   ……..
   //Overriding width only
   ram #(16)MEM1(.clk(clk),.rst(rst),…);

   //Overriding depth & width
   ram #(16,512)MEM2(.clk(clk),.rst(rst),…);
   ……..
 endmodule
```

```
/*Parameter overriding using name based*/

module system(…);
  input   rst,clk;
  output  dout,done;
  ……..
  //Overriding width only
  ram #(.WIDTH(16))MEM1(.clk(clk),.rst(rst),…);

  //Overriding depth & width
  ram #(.WIDTH(16),.DEPTH(512))MEM2(.clk(clk),.rst(rst),…);
  ……..
endmodule
```

Operators

2.1 Logical operators

```
/*The result of the evaluation of a
  logical comparison shall be 1(defined as true),
  0(defined as false) & if the result is ambiguous, the
  unknown value (x).*/
reg [2:0]a,b;
reg c;
reg x,y,z;
initial
  begin
    a = 3'd5;
    b = 3'b111;
    c = 1'bx;
    x = a && b; //The logical value of a is true & b is true, hence x = 1
    y = a || c; //The logical value of a is true & c is 'bx, hence y = 1
    z = b && 0; //The logical value of b is true & 0 is false, hence z = 0
    x = !a;     //The logical value of a is true , hence x = 0
  end
```

2.2 Bitwise operators

```
/*The bit-wise operators shall perform bit-wise operations on the operands that is,
  the operator shall operate a bit in one operand with its corresponding bit in the other
  operand to calculate one bit for the result.*/

reg [2:0]a,b,c,x,y,z;
initial
  begin
    a = 3'd5;
    b = 3'b111;
    c = 3'bx;
    x = a & b;     //The value of x = 3'b101
    y = a | c;     //The value of y = 3'b1x1
    z = b ^ 3'b1; //The value of z = 3'b110
    x = ~a;        //The value of x = 3'b010
    y = a ~^ b;    //The value of y = 3'b101
  end
```

&			
a	1	0	1
c	x	x	x
y	x	0	x

2.3 Reduction operators

```
/*The unary reduction operators shall perform a bit-wise operation
on a single operand to produce a single bit result.*/

reg [3:0]a,b;
reg y,z;
initial
 begin
    a = 4'b0110;
    b = 4'b1000;
    y = ~&b;  //b reduces to 1
    z = ^a;   //a reduces to 0
    y = |a;   //a reduces to 1
    z = &b;   //b reduces to 0
    y = ~|a;  //b reduces to 0
    z = ~^a;  //a reduces to 1
 end
```

2.4 Shift operators

```
reg [3:0] a,b,x,y,z;
reg signed [3:0]c;
initial
  begin
    a = 4'b0110;
    b = 4'b1100;
    c = 4'b1101;
    x = a << 1;  //Logical left shift appends 0 in lsb i.e  x = 4'b1100
    y = b >> 2;  //Logical right shift appends 0 in msb i.e y = 4'b0011
    z = a >>> 1; /*Arithmetical right shift on unsigned operands appends
                   0 to msb i.e z = 4'b0011 */
    z = c >>> 1; /*Arithmetical right shift on signed operands appends the
                   sign bit to msb i.e z = 4'b1110*/
  end
```

2.5 Equality operators

```
/*Equality operators compare operands bit by bit, with zero filling
  if the two operands are of unequal bit length. */

  reg [3:0]a,b;
  reg y1,y2,y3,y4,y5,y6;

  initial
    begin
      a = 4'b0010;
      b = 4'b0011;
      y1 = (a == b); //Logical equality will return 0
      y2 = (a != b); //Logical inequality will return 1
      #10;
      a = 4'b101x;
      b = 4'b1010;
      y3 = (a === b); //Case equality will return 0
      y4 = (a !== b); //Case inequality will return 1
      y5 = (a == b);  //Logical equality will return x
      y6 = (a != b);  //Logical inequality will return x
    end
```

2.6 Relational operators

```
/* If either operand of a relational operator contains an unknown(x) or high impedance
   (z) value,then the result shall be a 1-bit unknown value (x). */
reg [3:0]a,b;
reg y1,y2,y3,y4;

initial
  begin
    a = 4'b0010;
    b = 4'b0011;
    y1 = (a > b); //The greater than operator returns 0
    y2 = (a < b); //The less than operator returns 1
    #10;
    y1 = (a <= b);//The greater than equal operator returns 1
    y2 = (a >= b);//The less than equal operator returns 0
    #10;
    a = 4'b101x;
    b = 4'b1010;
    y3 = (a > b); //The greater than operator returns x
    y4 = (a < b); //The greater than operator returns x
  end
```

2.7 Concatenation operators

```
/* A concatenation is the joining together of bits resulting
  from two or more expressions.*/
reg a;
reg [2:0] b, c;
reg [7:0] x;

initial
  begin
    a = 1'b1; b = 3'b100; c = 3'b110;
    x = {1'b0,a,b,c}; //Concatenates to 0_1_100_110
    #10;
    x = {{2{a}},b,{2{c}}}; //Replicates "a" to 2 times, "c" to 2 times
                           /* The value of x = {11_100_110}*/
  end
```

2.8 Conditional operators

```
/*conditional_expression ? true_expression : false_expression;*/

reg [3:0] a,b,c,y,z;

initial
  begin
    a = 4'b1010;
    b = 4'b0010;
    c = 4'b1110;
    y = (&c)?a:b; //&c reduces to 0, hence y = b i.e false_expression
    z = (c) ?a:b; //The logical value of c is true, hence y = a i.e true_expression
  end
```

2.9 Arithmetical operators

```
reg [3:0]a,b,c;
integer d,e;
reg [3:0]x,y,z;
integer k,l,m;

initial
    begin
      a = 4'b0010;
      b = 4'b0011;
      c = 4'b101x;
      d = 3;
      e = 8;
      x = a * b;   // evaluates to 0110
      y = a + b;   // evaluates to 0101
      z = b - a;   // evaluates to 0001
      k = c * a;   // evaluates to x
      l = e / d;   // evaluates to 2,fraction is truncated
      m = e % d;   // evaluates to 2
    end
```

Processes

3.1 Continuous process

3.1.1 Continuous concurrent process

```
/* A continuous concurrent process is sensitive
   to the source elements in the expression*/

wire x,y,z;
assign x = a & c;
assign z = b | c;
assign y = a ^ b;
```

3.2 Procedural process

3.2.1 initial, always

```
/* Initial process executes only once
   and starts at 0 simulation time*/
initial
  begin
    reset = 1'b1;
    #100 reset = 1'b0;
    #10 din = 1'b1;
    #10 din = 1'b0;
  end

/* always process executes repetitively
   and starts at 0 simulation time*/
always
  begin
    #10 clk = 0;
    #10 clk = 1;
  end
```

3.3 Events

```
/*always process with events gets
  triggerred whenever an event occurs*/
always@(posedge clk)
  begin
    if(reset)
      q <= 1'b0;
    else
      q <= d;
  end
```

Structured procedures

4.1 Blocking assignment

```
/*The value of D at time "t" will be
  assigned to the final output Q3 at the same time "t"*/
always@(posedge clock)
  begin
    Q1 = D;
    Q2 = Q1;
    Q3 = Q2;
  end
```

4.2 Non-blocking assignment

```
 /*The value of D at time "t" will be
   assigned to the final output Q3 at the time "t+3" cycles*/
 always@(posedge clock)
   begin
     Q1 <= D;
     Q2 <= Q1;
     Q3 <= Q2;
   end
```

4.3 Tasks

```
/*Tasks are procedures that
  passes values using output arguments*/
  input[31:0]address;
  output reg parity_reg;
  task parity_cal(input [31:0]data,
                  output reg parity);
    begin
      #1 parity = ~^data;
    end
  endtask

  always@(address)
    parity_cal(address,parity_reg);
```

4.4 Functions

```
/*Functions are procedures that
  returns a single value*/
input [31:0]address;
output reg parity_reg;

function parity_cal(input[31:0]data);
  begin
    parity_cal = ~^data;
  end
endfunction

always@(address)
  parity_reg = parity_cal(address);
```

4.5 Timing control statements

4.5.1 Delays: {Inertial, Regular, Intra-assignment}

```
/*Inertial delay models are simulation delay models that
  filter pulses that are shorter than the propagation delay
  of Verilog gate primitives*/

  wire c;
  reg a,b;
  assign #2 c = a&b;
```

```
 /*Regular delays for blocking assigments are additive*/
 reg a,b;

 initial
   begin
     a = 1'b0;
     #5 b = 1'b1;
     #10 a = 1'b1;
     #100; //The process ends at 115ns
   end
```

```
/*Regular delays for Non-blocking assigments are additive*/
reg a,b;

initial
  begin
    a <= 1'b0;
    #5 b <= 1'b1;
    #10 a <= 1'b1;
    #100; //The process ends at 115ns
  end
```

```
/*Intra-assign delays for blocking assigments are additive*/
reg a,b;

initial
  begin
    a = 1'b0;
    b = #5 1'b1;
    a = #10 1'b1;
    #100; //The process ends at 115ns
  end
```

```
/*Intra-assign delays for Non-blocking assigments are non-additive*/
reg a,b;

initial
  begin
    a <= 1'b0;
    b <= #5 1'b1;
    a <= #10 1'b1;
    #100; //The process ends at 100ns
  end
```

4.5.2 Wait

```
/*Wait is level sensitive & blocks the below statements
  until the condition is true*/

initial
  begin
    a=0;
    c=0;
    while(1)
      #10 a = ~a;
  end

always
  begin
    wait(a)
    #1 c = ~c;
  end
```

4.5.3 Event based timing

```
/*Event based timing control
  statements based on transitions*/
  initial
   begin
     @(negedge clk);
     reset = 1;
     @(negedge clk);
     reset = 0;
   end
```

4.6 Procedural blocks

4.6.1 Sequential blocks

```
/*begin-end being sequential will call the tasks
  sequentially i.e one after other*/
initial
  begin
    task1;//task1 is called at 0ns
    task2;//task2 is called only after task1 is completed
    task3;//task3 is called only after task2 is completed
  end
```

4.6.2 Parallel blocks

```
/*fork-join being parallel will invoke the tasks
  as 3 parallel threads*/
initial
  fork
    task1;//task1 is called at 0ns
    task2;//task2 is called at 0ns
    task3;//task3 is called at 0ns
  join
```

4.6.3 Named blocks

```
/*Named blocks allows encapsulation
  hence local variables can be declared
  within the block*/
always
  begin:block1
    integer i;
    for(i=0;i<32;i=i+1)
      begin
        -----
      end
  end
```

4.6.4 Disable

```
/*The disable statement can be used
  to disable a block itself*/
initial
  fork
    begin:B1
      reg [3:0]a;
      #30;
      disable B1;//The below statements within the block B1, will not execute after 30ns
      a = 7;
      $display($time,"Value of a =%d",a);
    end
    begin:B2 //This block will complete at 50ns
      reg [3:0]a;
      #50;
      a = 6;
      $display($time,"Value of a =%d",a);
    end
  join
```

4.6.5 Branching constructs: {if else, case}

```
/*if-elseif-else implies priority*/
always@(posedge clk)
  begin
    if(reset) //1st priority
      q <= 0;
    else if(load) //2nd priority
      q <= din;
    else  //3rd priority
      q <= q + 1'b1;
  end
```

```
 /*case implies parallel logic*/
 always@(*)
   begin
     case(sel)   //Any signal of the bus can be selected
       0 : y = din[0];
       1 : y = din[1];
       2 : y = din[2];
       3 : y = din[3];
       default: y = 0;
     endcase
  end
```

```
/*Full case will have all possible case items
  getting matched with the case expression*/
module full_case(input[2:0]ain,
                 output reg[7:0]y);
  always@(*)
    begin
      case(ain)
        3'b000  : y = 8'b00000001;
        3'b001  : y = 8'b00000010;
        3'b010  : y = 8'b00000100;
        3'b011  : y = 8'b00001000;
        default : y = 8'b00000000;
      endcase
    end
endmodule
```

```
/*Parallel case will have the case expression
  getting matched with only one case item at a time*/
module parallel_case(input[2:0]ain,
                     output reg[7:0]dout);

  always@(*)
    begin
      case(ain)
        3'b000 : dout = 8'b00000001;
        3'b001 : dout = 8'b00000010;
      endcase
    end

endmodule
```

```
/*Overlapping case will have the case expression
getting matched with more than one case item at a time*/
module overlapping_case(input [2:0]ain,
                        output reg [1:0]y);

  always@(*)
    begin
      casex(ain)
        3'b1?? : y = 0;//The first occurred case item will be having highest priority
        3'b?1? : y = 1;
        3'b??1 : y = 2;
      endcase
    end

endmodule
```

4.6.6 Looping constructs: {for, while, repeat, forever}

```
/*The for loop controls the execution
  of the below statements by a 3
  step process*/
  initial
   begin
     for(i=0;i<10;i=i+1)
       begin
         #10;
         y = i; //y gets value from 0 to 9 in every 10ns time-step
       end
   end
```

```
/*The while loop executes a statement
  until an expression becomes false*/
  initial
   begin
     #10;
     clk = 0;
     while(1) //The condition is always true hence it enters into infinite loop
       #10 clk = ~clk;
   end
```

```
/*repeat is a finite loop*/
initial
 begin
   repeat(10)
     write_t; //The task write_t is called for 10times
 end
```

```
 initial
   begin
     #10;
     clk = 0;
     forever //The loop is an infinite loop
       #10 clk = ~clk;
   end
```

System tasks & functions

5.1 Display system tasks

5.1.1 $display, $write, $strobe, $monitor

```
/*$strobe displays the last updated value in the current time-slot in which it is called
  $display displays the value at that instant of time
  $monitor continuously monitors the changes in the arguments*/
reg [1:0]s;
initial
  begin
    $strobe ("Value of strobed  s =%d at time =%t",s,$time); //Value of strobed  s = 2 at time =0
    $display("Value of displayed s =%d at time =%t",s,$time);//Value of displayed  s = x at time =0
    $write("Value of displayed s =%d at time =%t \n",s,$time);//Same as $display but needs a
                                                              //\n at the end
    $monitor("Value of s =%d at time =%t",s,$time);//Value of s = 2 at time =0
    s = 3;
    s = 2;
    #10;
    s = 0;//Value of s = 0 at time =10
  end
```

5.2 File operations

5.2.1 File write

```
/*File output operation*/
integer channel;
initial
  begin
    channel= $fopen("file.txt");
    $fmonitor(channel,$time,"a = %b,b = %b,z = %b",a,b,z);
    #100;
    $fclose(channel);
  end
```

```
integer chanel_1,chanel_2,comb;
initial
  begin
    chanel_1 = $fopen("file1.out");
    chanel_2 = $fopen("file2.out");
    comb = chanel_1 | chanel_2;
    $fmonitor(comb,$time,"a = %b,b = %b,z = %b",a,b,z);
    #100;
    $fclose(chanel_1);
    $fclose(chanel_2);
  end
```

5.2.2 File read

init8x8.txt

```
@2
11111111
01010101
00000000
@6
xxxxzzzz
1x1x1x1x
```

```
/*File input operation*/
reg [7:0]mem8x8[7:0];

initial
  begin
    $readmemb("init8x8.txt",mem8x8);
  end
```

5.3 Simulation control system tasks

5.3.1 $finish

```
initial
  begin
   /*Stimulus*/
   #100 $finish; //$finish will terminate the simulation after stimulus driven + 100ns
  end
```

5.3.2 $stop

```
initial
  begin
   /*Stimulus*/
   /*Debugging the DUT output in a self checking Testbench*/
   $stop; //$stop will suspend the simulation for debug interactive purpose
  end
```

5.4 Randomizing function

5.4.1 $random

```
/*$random is used to generate random integers*/
initial
  begin
    #1 y = {$random}%10;//It randomizes y between 0 to 9
  end
```

5.5 Command line input function

5.5.1 $test$plusargs

```
 /*$test$plusargs is a conditional simulator directive*/
 initial
   begin
     if ($test$plusargs("HELLO"))
       $display("Hello");
     if ($test$plusargs("WELCOME"))
       $display("Welcome");
   end

   Run simulator with command mode +HELLO
   Simulator output :
   Hello

   Run simulator with command mode +WELCOME
   Simulator output :
   Welcome
```

5.6 Simulation time functions

5.6.1 $time

```
 /*$time returns simulation time as unsigned integer format*/
reg set;
parameter q = 10;
initial
  begin
    $monitor($time,"set = %b",set);
    #q set = 0; //10 set = 0
    #q set = 1; //20 set = 0
  end
```

5.6.2 $realtime

```
/*$realtime returns simulation time in real format*/
reg set;
parameter p = 1.55;
initial
  begin
    $monitor($realtime,"set = %b",set);
    #p set = 0; //1.55 set = 0
    #p set = 1; //3.1 set = 1;
  end
```

$stime is same as $time but it returns an integer which is 32bits.

Compiler directives

6.1 `define

```
`define WIDTH 8
`define DEPTH 256
module ram(clk,reset,din,radd,wadd,dout);
input clk,reset;
.....
input [(`WIDTH - 1) : 0]din;
input [(`WIDTH - 1) : 0]dout;

reg [(`WIDTH - 1):0]mem[(`DEPTH - 1):0];
.........
.........
endmodule
```

6.2 `include

```
/*state_definition.v*/

`define HOLD   0
`define RESET  1
`define SET    2
`define TOGGLE 3

`include "state_definition.v"
module jk(input clk,rst,j,k,
          output reg q);
always@(posedge clk or posedge rst)
  if(rst)
    q <= 1'b0;
  else
    begin
      case({j,k})
        `HOLD   :  q <= q;
        `RESET  :  q <= 1'b0;
        `SET    :  q <= 1'b1;
        `TOGGLE :  q <= ~q;
      endcase
    end
endmodule
```

6.3 `timescale

```
`timescale  1ns/100ps
module timescale;

  reg clk ;
  parameter cycle = 15;
  always
    begin
      #(cycle/2.0);
      clk = 0;
      #(cycle/2.0);
      clk = 1;
    end

endmodule
```

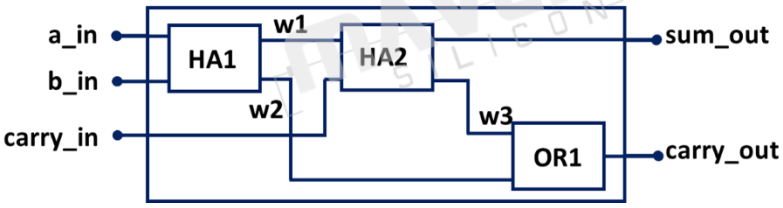
6.4 `ifdef

```
`define behavioral
module and_op (a, b, c);
  output a;
  input b, c;
  `ifdef behavioral
    wire a = b & c;
  `else
    and a1 (a,b,c);
  `endif
endmodule
```

Verilog examples

7.1 One bit full_adder using half_adder

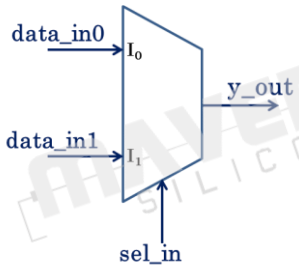
.




```
module half_adder(a_in, b_in,sum_out,carry_out);

  input  a_in,b_in;
  output sum_out,carry_out;

  assign sum_out = a_in ^ b_in;
  assign carry_out = a_in & b_in;

endmodule
```

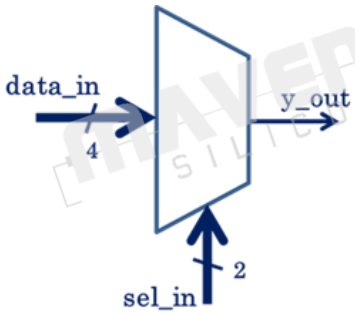
```
module  full_adder(a_in,b_in,carry_in,sum_out,carry_out);

  input  a_in,b_in,carry_in;
  output sum_out,carry_out;
  wire w1,w2,w3;

  half_adder HA1(.a_in(a_in),.b_in(b_in),.sum_out(w1),.carry_out(w2));

  half_adder HA2(.a_in(w1),.b_in(carry_in),.sum_out(sum_out),.carry_out(w3));

  or  OR1(carry_out, w2, w3) ;

endmodule
```

7.2 2:1 Multiplexer



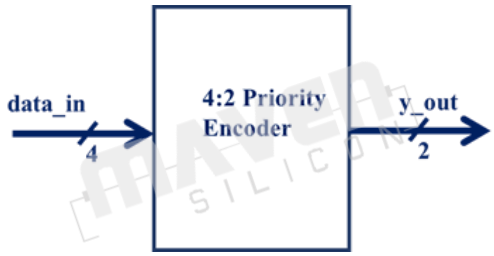

```
module mux_2_1(data_in0,data_in1,
               sel_in,y_out) ;
  input  data_in0,data_in1,sel_in;
  output  reg y_out ;

  always@( data_in0,data_in1,sel_in )
    begin
      if ( sel_in )
        y_out = data_in1;
      else
        y_out = data_in0;
    end
endmodule
```

7.3 4:1 Multiplexer



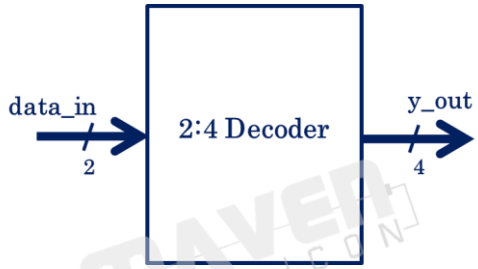

```
module mux_4_1(data_in,sel_in,y_out);

  input [3:0] data_in;
  input [1:0] sel_in;
  output reg y_out;

  always@( data_in,sel_in )
    begin
      case (sel_in)
        2'b00 : y_out = data_in[0];
        2'b01 : y_out = data_in[1];
        2'b10 : y_out = data_in[2];
        2'b11 : y_out = data_in[3];
        default : y_out = 0;
      endcase
    end
endmodule
```

7.4 4:2 Priority Encoder



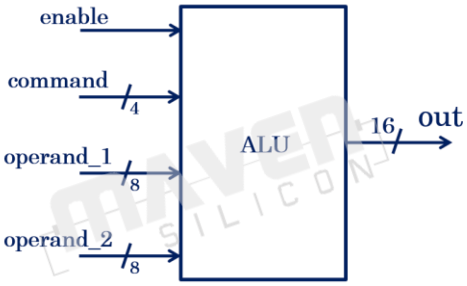

```
module pri_enc( data_in, y_out);

  input   [3:0]data_in;
  output  reg [1:0]y_out;

  always@(data_in)
    begin
      if (data_in[3])
        y_out = 2'd3;
      else if (data_in[2])
        y_out = 2'd2;
      else if (data_in[1])
        y_out = 2'd1;
      else if (data_in[0])
        y_out = 2'd0;
      else
        y_out = 2'd0;
    end
endmodule
```

7.5 2:4 decoder




```
module decoder( data_in, y_out) ;
  input [1:0] data_in;
  output reg [3:0] y_out ;
  always@(data_in)
    begin
      case (data_in)
        2'd0 : y_out = 4'b0001;
        2'd1 : y_out = 4'b0010;
        2'd2 : y_out = 4'b0100;
        2'd3 : y_out = 4'b1000;
        default : y_out = 4'b0000;
      endcase
    end
endmodule
```

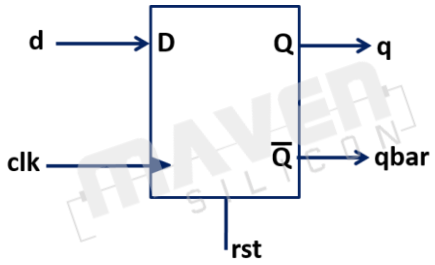
7.6 ALU




```
module alu (input [7:0] operand_1, operand_2,
            input enable,
            input [3:0] command,
            output [15:0] out);
  reg [15:0] tmp ;
  parameter ADD  = 4'b0000,
            INC  = 4'b0001,
            SUB  = 4'b0010,
            DEC  = 4'b0011,
            MUL  = 4'b0100,
            DIV  = 4'b0101,
            SHL  = 4'b0110,
            SHR  = 4'b0111,
            INV  = 4'b1000,
            AND  = 4'b1001,
            OR   = 4'b1010,
            NAND = 4'b1011,
            NOR  = 4'b1100,
            XOR  = 4'b1101,
            XNOR = 4'b1110,
            BUF  = 4'b1111;
```

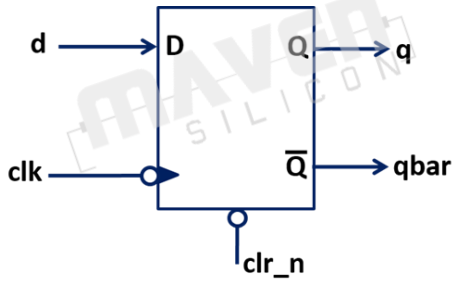
```
  always@(operand_1,operand_2,command)
    begin
      case (command)
        ADD : tmp = operand_1 + operand_2;
        INC : tmp = operand_1 + 1;
        SUB : tmp = operand_1 - operand_2;
        DEC : tmp = operand_1 - 1;
        MUL : tmp = operand_1 * operand_2;
        DIV : tmp = operand_1 / 2;
        SHL : tmp = operand_1 << 1'b1;
        SHR : tmp = operand_1 >> 1'b1;
        INV : tmp = ~operand_1;
        AND : tmp = operand_1 & operand_2;
        OR  : tmp = operand_1 | operand_2;
        NAND: tmp = ~(operand_1&operand_2);
        NOR : tmp = ~(operand_1|operand_2);
        XOR : tmp = operand_1 ^ operand_2;
        XNOR: tmp = ~(operand_1^operand_2);
        BUF : tmp = operand_1;
      endcase
    end
  assign out = (enable)? tmp : 16'dz ;
endmodule
```

7.7 DFF-Synchronous reset



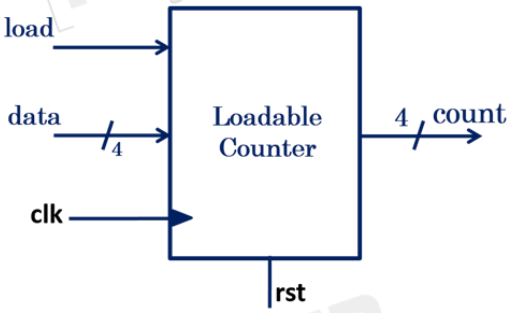

```
module dff_sync_rst (rst,clk,d,q,qbar);
  input rst,clk,d;
  output reg q;
  output qbar;

  always@(posedge clk)
    begin
      if (rst)
        q <= 1'b0;
      else
        q <= d;
    end
  assign qbar = ~q;
endmodule
```

7.8 DFF-Asynchronous clear



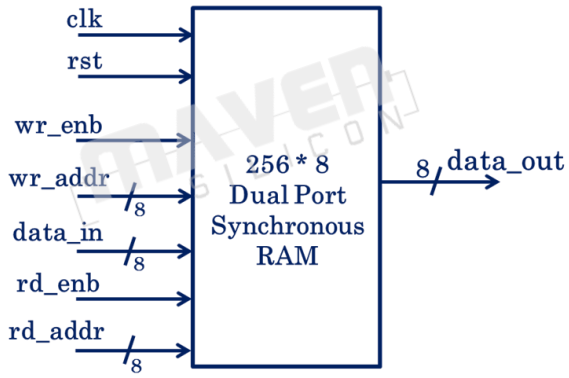

```
module dff_async_clr (clr_n,clk,d,q,qbar);
  input clr_n, clk,d;
  output reg q;
  output  qbar;

  always@(negedge clk,negedge clr_n)
    begin
      if (~clr_n)
        q <= 1'b0;
      else
        q <= d;
    end
  assign qbar = ~q;
endmodule
```

7.9 Modulo 12 – Counter




```
module  loadable_counter (input clk,rst,load,
                          input [3:0] data,
                          output reg [3:0]count);

  always@(posedge clk)
    begin
      if (rst)
        count <= 4'd0;
      else if (load)
        count <= data;
      else if (count == 4'd11)
        count <= 4'd0;
      else
        count <= count + 1'b1;
    end

endmodule
```

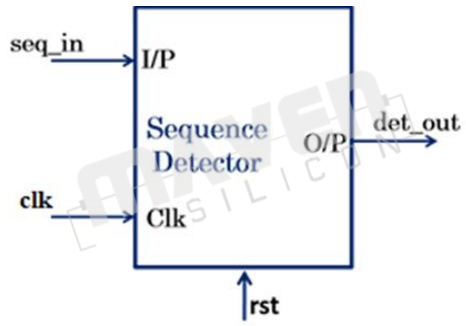
7.10 Dual port synchronous RAM – 256X8

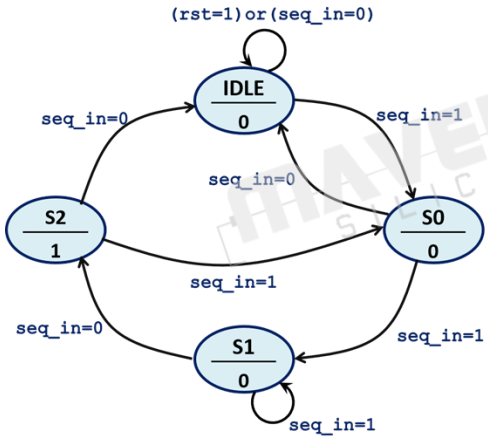



```
module dual_ram (clk,wr_enb,rd_enb,rst,
                 rd_addr,wr_addr,
                 data_in,data_out);

  parameter  RAM_WIDTH  = 8;
  parameter  RAM_DEPTH  = 256;
  parameter  ADDR_SIZE  = 8;

  input clk,wr_enb,rd_enb,rst;
  input[RAM_WIDTH-1:0] data_in;
  input [ADDR_SIZE-1:0] rd_addr, wr_addr ;
  output reg [RAM_WIDTH-1:0] data_out;

  reg [RAM_WIDTH-1:0]mem[RAM_DEPTH-1:0];
  integer i;
```

```
  always@(posedge clk)
    begin
      if( rst )
        begin
          for(i=0; i<RAM_DEPTH; i=i+1)
            mem[i] <= 0;

          data_out <= 0;
        end
      else
        begin
          if (wr_enb)
            mem[wr_addr] <= data_in;
          if (rd_enb)
            data_out <= mem[rd_addr] ;
        end
    end
endmodule
```

7.11 Sequence detector-110(Moore overlapping)






```
module  seq_det (clk,rst,seq_in,det_out);
  input clk,rst,seq_in;
  output det_out;
  parameter IDLE  = 4'b0001,
            S0    = 4'b0010,
            S1    = 4'b0100,
            S2    = 4'b1000;
  reg [3:0] present,next;
  always@(posedge clk) //Present state logic
    begin
      if(rst)
        present <= IDLE;
      else
        present <= next;
    end
```

```
  always@(present,seq_in) //Next state logic
    begin
      next = IDLE; //Default state
      case( present )
        IDLE :begin if(seq_in)  next = S0; end
        S0   :begin if(seq_in)  next = S1; end
        S1   :begin if(~seq_in) next = S2;
                    else        next = S1; end
        S2   :begin if(seq_in)  next = S0; end
      endcase
    end
  assign det = (present == S2) ? 1:0; //Output logic
endmodule
```