


EE461 Verilog-HDL

Week 5 RTL Level Design

Alex Yang, Engineering School, NPU

Week 5 Outlines

- RTL Models in Verilog (Synthesizable)
  - Continuous Assignment
  - “always” blocks
    - Conditional Statement if-else
    - Case Statements
    - Looping Statements

Week 5 Outlines

- RTL Models in Verilog (Synthesizable)

```
module RTLDesgin(a_i, b_i, en_i,
    c_o,clk);
    input       a_i, b_i, en_i, clk;
    output      c_o;
    reg         c_o;
assign  b_i = (en)? 1'bx: 1'bz;
assign  b_i = (en==0)?   1'b0:
              (en==1)?   1'b1:
              (en==2)?   1'bx: 1'bz;
// Call functions; not tasks
assign  b_i = foo(... ...)
function .. ...
// Don't include non-syn statements
endfunction
task ... ...
// Don't include non-syn statements
endtask
```

```
always@(a _i or b_i) begin
// if-else, if-else if- else, case, (or
    casez, casez: don't recommand)
// blocking(=)
// (for-loop:  don't recommand)
//functions/tasks(w/o timing control)
end
always@(posedge clk)begin
// if-else, if-else if- else, case, (or
    casez, casez: don't recommand)
// blocking(=)
// (for-loop:  don't recommand)
//functions/tasks(w/o timing control)
end
endmodule
```

Note: In RTL level design, please don't include "initial" block

Continuous Assignment

- Syntax & Usage

  e.g. assign #10 {carry, sum} = a+b; (Synthesizable)

  Note: "+" is arithmetic operator, not logic or (|).

  - Variable data type in LHS must be wire, and RHS could be reg or wire
  - Modeling combinational logic(circuit with logic gates).
  - Must be outside of "always" blocks in RTL Level.
  - Can't have "begin-end" block inside.
  - The continuous assign overrides any procedural assignments.
  - Delay time is just for simulation, not real delay time from device.

Continuous Assignment

- Syntax & Usage

  Other formats:

  e.g. assign #10 a = (en)? b: c; // Transition Delay

  //Transition Delay-> min : typical : max

  // Delay time can't be zero

  assign #(1:2:3) a = (en)? b: c;

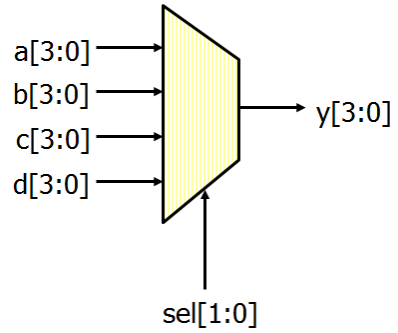
  assign #10 a = (en==0)? b:
  (en==1)? c:
  (en==2)? d: e;

  Synthesized Hardware from continuous assignment:
  Mux w/o priority except multi-layer conditional assignment

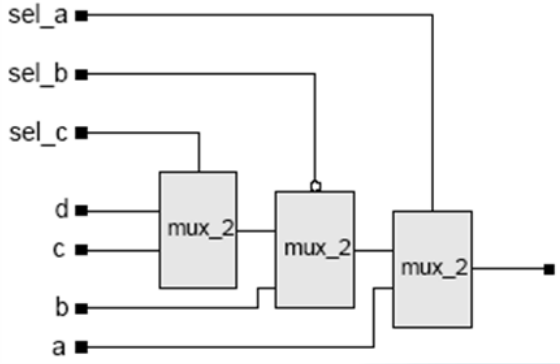
Continuous Assignment

- Syntax & Usage

Synthesized Hardware: Mux w/o priority



Synthesized Hardware: Mux with priority



Continuous Assignment

- Syntax & Usage

```
module mux4bits(y_o, a_i, b_i, c_i, d_i, sel_i);
    input [3:0] a_i, b_i, c_i, d_i;
    input [1:0] sel_i;
    output [3:0] y_o;      //By default, data type of input and output port is wire
    assign   y_o = (sel == 0) ? a_i :          // y_o = ? If sel = 2'b1z,
                   (sel == 1) ? b_i :          // Ans: y_o = 4'bxxxx
                   (sel == 2) ? c_i :
                   (sel == 3) ? d_i : 4'bx;
endmodule
```

You may write a testbench to observe the outputs

```
`include "mux4bits.v"
module mux4bitsTB;
    … …
endmodule
```

Continuous Assignment

- Syntax & Usage

  assign out = s1 ? (s0 ? i3 : i2) : (s0 ? i1 : i0) ;

  Questions:

  1. Design a detector to detect if all bits are 0s or all 1s.

  2. Design 2's complement number converter.

Continuous Assignment

- Examples:

```
module CondOp();
        reg [3:0] X_r;
        reg [3:0] Y_r;
        wire Z_w;
        assign Z_w=(X_r==Y_r)? 1'b1:1'b0;
         //assign Z_w=(X_r!=Y_r)? 1'b1:1'bz;
        //assign Z_w=(X_r===Y_r)? 1'b1:1'b0;
        //assign Z_w=(X_r !==Y_r)? 1'b1:1'b0;
        initial begin
                X_r=4'b101x;
                // X_r = 4'b1010;
                // X_r = 4'b101z;
                Y_r=4'b101z;
                $monitor("Z_w's values: %d\n", Z_w);
        end
endmodule

Ans:  ?
```

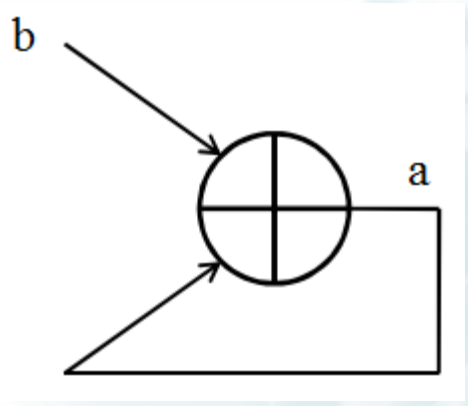
```
module CondOp1();
        reg s0_r, s1_r, s2_r;
        reg i0_r, i1_r, i2_r, i3_r;
        wire out_w;

        assign out_w = s1_r ? ( s0_r ? i3_r : i2_r) :
                                (s0_r ? i1_r : i0_r);
        initial begin
                i3_r=1'b0; i2_r=1'b1; i1_r=1'bx; i0_r=1'bz;
                #1 s1_r = 1'bx;
                #1 s1_r = 1'bz;
          $monitor("out_w's values: %d\n", out_w);
        end
endmodule

Ans:  out_w's values: x
```

Continuous Assignment

- Avoid logic loop
  - HDL Compiler and Design Compiler will automatically open up asynchronous logic loops
  - Without disabling the combinational feedback loop,the static timing analyzer can't resolve

Example:

```
wire [3:0] a;
wire [3:0] b;
assign a = b + a;
```



always blocks

- **"always" For Combinational Logic**
  (Synthesizable)
  - Function: always blocks loop to execute over and over again
  - Syntax Requirements
  - Synthesized Hardware: Logic Gates

- **"always" For Sequential Logic**
  (Synthesizable)
  - Syntax Requirements
  - Synthesized Hardware: Flip Flops

always blocks

- "always" For Combinational Logic

```
module twoCompFA( x_i, y_i, cin_i, sum_o, co_o);
        input wire[3:0]         x_i,    y_i;
        input wire              cin_i;
        output reg[3:0]         sum_o;
        output reg              co_o;
        reg[3:0]                t_r;
        always @(x_i, y_i, cin_i) begin // OR always@(x_i or y_i or cin_i)
                t_r = y_i ^ {4{cin_i}};
                {co_o, sum_o} = x_i + t_r + cin_i;
        end
endmodule
```

Note: module as above combines adder and subtractor

always blocks

- "always" For Combinational Logic
  - Syntax Requirements:

```
always @(x_i, y_i, cin_i) begin        // always@(*)
        t_r = y_i ^ {4{cin_i}};
        {co_o, sum_o} = x_i + t_r + cin_i;
end
```

1. Sensitivity list includes all variables in RHS except those from LHS, or simply use single star.
2. Variables in LHS must be reg data type.
3. "="(blocking) should be used as assignment
4. Sequence of assignment statements can't be changed in "always" block. Otherwise, logic function will be changed.
5. Once one of variables in the sensitivity list of "always" changes, values of variables in LHS will be recalculated.

always blocks

- "always" For Combinational Logic
  - Difference between assignments in two blocks with different sequence

```
always @(a, b, c) begin          // always@(*)
        d = a + b;               // a = 1, b=1; c=1 => d=2;e = 3
        e = d + c;               // If a => 2, => d=3; e = 4
end

always @(a, b, c) begin          // always@(*)
        e = d + c;               // a = 1, b=1; c=1 =>d=2; e = x
        d = a + b;               // If a => 2;=> d=3; e = 3
end
```

Note: In “always” block combinational logic, assignments sequence can’t be changed.

always blocks

- "always" can't trig itself

```
module trigger_itself();
        reg clk;
        always @ (clk)          // Trig itself
        //always begin          // It should be like this
                #5 clk = !clk;
        end
        // Testbench code here
        initial begin
                $monitor("TIME = %d  CLK = %b",$time,clk);
                clk = 0;
                #100 $display("TIME = %d  CLK = %b",$time,clk);
                $finish;
        end
endmodule
```

Note: What are you going to get? Ans: time 0 CLK=0; time 5 CLK = 1
no more.

always blocks

- "always" For Sequential Logic

```
module seqLogic(
                        d_i,
                        clk,
                        q_o
);
        input           d_i, clk;
        output  q_o;
        always @ (posedge clk) begin
                // Or (negedge clk)
                q_o <= #1 d_i;      //#1 delay is nonsythesizable
        end
endmodule
```

```
input           clk;
input           din_i;
output reg [3:0]    qout_o;

always @(posedge clk)
   qout_o  <= #1 {din_i, qout_o[3:1]};
```

Modeling: Shift data in

always blocks

- "always" For Sequential Logic
  - Syntax Requirements:

1. Clock signal should be in the sensitivity list such as (posedge or negedge)
2. Variables in LHS must be reg data type in assignment.
3. “<="(nonblocking) assignment should be used
4. Sequence of nonblocking assignment statements doesn’t matter.
5. Once rising/falling edge of clock signal comes, values of variables in LHS will be recalculated.
6. Synthesized hardware: D flip flop

always blocks

- "always" For Sequential Logic
  - Difference between assignments in two blocks with different sequence

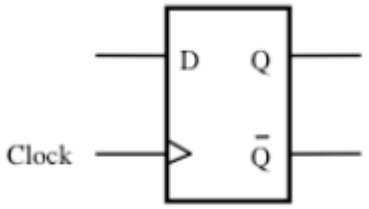
```
always @(posedge clk) begin //nonblocking: simultaneous assignment
    b <= a;        // a = 1, b=2, c=3, d =4(initial values  before assignment)
    c <= b ;       // New value  s: a =1, b=1, c=2, d = 3 like shift registers
    d <= c;
end

always @(posedge clk) begin     // blocking: procedural assignment
    b = a;         // a = 1, b=2, c=3, d =4(initial values  before assignment)
    c = b ;        // New values: a =1, b=1, c=1, d=1
    d = c;
end
```

Note: In “always” block sequential logic, assignments must be nonblocking.

always blocks

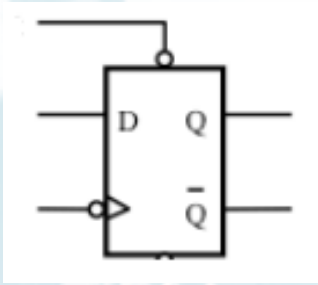
- "always" For Sequential Logic
  - Synchronous D Flip Flop With Rest

```
always@(posege clk)begin
    if (rst) begin
    … …
    end
end
```



  - Asynchronous D Flip Flop With Reset

```
always@(negedge clk or negedge rst)begin
    if(!rst) begin … … end
end //always@(negedge clk or rst): wrong
```

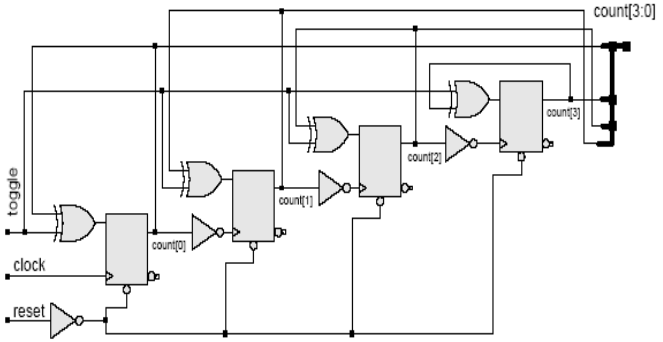


Note: Two internal hardware circuits are different.

always blocks

- Asynchronous Sequential Logic

```
assign c0 = count[0], c1 = count[1], c2 = count[2];
    always @ (posedge reset or posedge clock)
     if (reset == 1'b1) count[0] <= 1'b0;
     else if (toggle == 1'b1)
            count[0] <= ~count[0];
   always @ (posedge reset or negedge c0)
     if (reset == 1'b1) count[1] <= 1'b0;
     else if (toggle == 1'b1)
             count[1] <= ~count[1];
   always @ (posedge reset or negedge c1)
     if (reset == 1'b1) count[2] <= 1'b0;
     else if (toggle == 1'b1) count[2] <= ~count[2];
   always @ (posedge reset or negedge c2)
     if (reset == 1'b1) count[3] <= 1'b0;
     else if (toggle == 1'b1) count[3] <= ~count[3];
```



always blocks

- Synchronous & Asynchronous Sequential Logic
  - For reset, both have the pros and cons. Read academic paper for this special topic
  - In above example, we don't recommend to design counter in asynchronous sequenctial logic

- Some Designs We Don't Recommend
  - Synchronous & Asynchronous Mix logic
  - "posedge" & "negedge" Mix Trigger
  - Multiple Clock Source Drives
  - Synchronous & Asynchronous Reset Mix
  - Reset in Combinational Logic

always blocks

- Some Designs We Don't Recommend
  - Synchronous & Asynchronous Mix logic

BAD Design

```
reg[1:0]  sig;

always @(posedge clk or negedge rst)
    if (~rst ) sig <= #1 2'b0;
    else if (condition1)
      sig <= #1 2'b10;
always@(posedge clk)
  if(condition2) sig <= #1 2'b11;
 ... ...
```

Good Design

```
reg[1:0]  sig;

always @(posedge clk or negedge rst)
    if (~rst ) sig <= #1 2'b0;
    else if (condition1)
      sig <= #1 2'b10;
    else if(condition2)
      sig <= #1 2'b11;
  ... ...
```

always blocks

- Some Designs We Don't Recommend
  - "posedge" & "negedge" Mix Trigger

BAD Design

```
reg[2:0]  data;

always @(posedge clk or negedge rst)
    if (~rst ) data <= #1 2'b0;
    else       data <= #1 dinA;
always @(negedge clk or negedge rst)
   if (~rst )  d <= #1 2'b0;
    else       d <= #1 dinB;
 ... ...
```

Good Design

```
reg[2:0]  data;

always @(posedge clk or negedge rst)
    if (~rst ) data <= #1 2'b0;
    else       data <= #1 dinA;
always @(posedge clk or negedge rst)
   if (~rst )  d <= #1 2'b0;
    else       d <= #1 dinB;
 ... ...
```

always blocks

- Some Designs We Don't Recommend
  - Multiple Clock Source Drives

BAD Design

```
reg[2:0]  data;

always @(posedge clkA or posedge clkB)
    if (~rst ) data <= #1 2'b0;
    else       data <= #1 dinA;
```

Or more clock sources in one module

```
always @(posedge clkA)
 ... ...
always @(posedge clkB)
 ... ...
```

Good Design

```
reg[2:0]  data;

always @(posedge clk or negedge rst)
    if (~rst ) data <= #1 2'b0;
    else       data <= #1 dinA;
always @(posedge clk or negedge rst)

  ... ...
```

In one module, one clock source and one reset

always blocks

- Some Designs We Don't Recommend
  - Synchronous & Asynchronous Reset Mix

BAD Design:

```
reg       a;
reg       b;
reg       c;


always @(a or b or c or negedge rst)
    if (~rst )
            dout = 0;
  else
            dout = a & (b | c);
```

Good Design:

```
reg       a;
reg       b;
reg       c;

always @(posedge clk or negedge rst)
    if (~rst )
            a <= 0; b <= 0; c <= 0;
  always @(a or b or c)
            dout = a & (b | c);
```

always blocks

- Some Designs We Don't Recommend
  - Reset in Combinational

BAD Design: Reset has combo logic

```
reg[3:0]  cnt;
wire synRst;

assign synRst = (cnt >= 10);

always @(posedge clk or negedge rst)
     if (~rst ) cnt <= #1 4'b0;
     else       cnt <= #1 cnt + 1;
always @(posedge clk or negedge rst)
    if(~rst | synRst) dout <=#1 0;
   else               dout <= #1 din;
```

Good Design: DFF reset trigger: only reset

```
reg[3:0]  cnt;
wire synRst;

assign synRst = (cnt >= 10);

always @(posedge clk or negedge rst)
     if (~rst ) cnt <= #1 4'b0;
     else       cnt <= #1 cnt + 1;
always @(posedge clk or negedge rst)
    if(~rst)              dout <=#1 0;
   else if(~synRst) dout <= #1 0;
   else                  dout <= #1 din;
```

always blocks

- Multi-always blocks in Design

```
module addPipe(cout,sum,ina,inb,cin,clk);
        input[7:0] ina,inb; input cin,clk; output[7:0] sum;
        output cout;reg cout; reg[7:0] sum;
        reg[3:0] tempa,tempb,firsts; reg firstc;

        always @(posedge clk) begin
        {firstc,firsts} <= #1 ina[3:0]+inb[3:0]+cin;
                tempa <= #1 ina[7:4];
                tempb <= #1 inb[7:4];
        end
        always @(posedge clk) begin
        {cout,sum[7:4]} <= #1 tempa+tempb+firstc;
                sum[3:0] <= #1 firsts;
        end
endmodule
```

If ina = 8'hA5 & inb=8'h5A, cin=1'b0 calculate results at each clock cycles

Conditional Statement if-else

- It must be within “always” combinational logic/ sequential logic blocks or "initial" blocks.
- Syntax:
  if or if-else or if-else if or if- “else if”-else or nested if-else

```
module testIf();

reg latch;
wire enable,din;

always @ (enable or din)
    if (enable) begin
            latch = din;
    end
endmodule
```

```
module  testIfElse();

    reg dff;
   wire clk,din,reset;

   always @ (posedge clk)
      if (reset) begin
            dff <= #1 0;
      end else  begin
            dff <= #1 din;
      end
endmodule
```

Conditional Statement if-else

- Synthesized Hardware of “if-else”: Mux with priority

```
module nested_if();

reg [3:0] counter;
reg clk,reset,enable, up_en, down_en;

always @ (posedge clk)
   if (reset == 1'b0) begin
            counter <= #1 4'b0000;
  end else if (enable == 1'b1 && up_en == 1'b1)
  begin
            counter <= #1 counter + 1'b1;
  end else if (enable == 1'b1 && down_en == 1'b1)
  begin
            counter <= #1 counter - 1'b1;
  end else begin
            counter <= #1 counter; // Redundant
code
  end

endmodule
```

```
always @(negedge clk)
// always @(negedge clk or posedge rst)
   if (rst)
            qout <= #1 4'd0;
   else
            qout <= #1 (qout + 1) ;
```

```
always @(*) begin
   if (s1) begin
      if (s0) out = i3; else out = i2;
   end
   else begin
      if (s0) out = i1; else out = i0;
   end
end
```

Conditional Statement if-else

- Example 1:

```
module testIf1();
    reg [3:0] X, Y;
    initial begin
            X=4'b101x;
            Y=4'b101z;
            if(X == 4'b101z)begin
                    $display("Statement 1 has been selected!");
            end
            if(X == Y)begin
                    $display("Statement 2 has been selected!");
            end
```

Conditional Statement if-else

```
        if(X != Y)begin
                $display("Statement 3 has been selected!");
        end
        if(X == X)begin
                $display("Statement 4 has been selected!");
        end
        $display("No one has been selected!");
    end
endmodule
```

Ans: "No one has been selected!"

Conditional Statement if-else

- Example 2:

```
module testIf2();
    reg [3:0] X;
    reg [3:0] Y;
    initial begin
        X=4'b101x;
        Y=4'b101z;
        if(X===4'b101z)begin
            $display("Statement 1 has been selected!");
        end
        if(X===Y)begin
            $display("Statement 2 has been selected!");
        end
```

Conditional Statement if-else

```
        if(X!==Y)begin
                $display("Statement 3 has been selected!");
         end
         if(X===X)begin
                $display("Statement 4 has been selected!");
         end
         $display("No one has been selected!");
    end
endmodule
```

Ans: "Statement 3 has been selected!"
"Statement 4 has been selected!"

Conditional Statement if-else

- Synthesized Hardware of “if”: Latch

  - Transparent latches will appear if you write a combinational process or always block where an output is not assigned under all possible input conditions known as incomplete assignment.

  - Sensitive list is not complete. It behaves just like latch in simulation. But after synthesize, it is still combinational logic. it's a difference between simulation and synthesize.

```
module simpleIf();
    reg latch;
    wire enable,din;

   always @ (enable or din) begin
       if (enable) begin
           latch = din;
      end  // It doesn't cover 0 or more
    end
endmodule
```

```
always @ ( A ) begin
            C = A & B ;
end
```

```
always @ ( B ) begin
            C = A & B ;
end
```

Conditional Statement if-else

- Inferred latch issues in Verilog
  - Timing issue in latch will occur. In an FPGA, all storage elements (like FF's) should be synchronized with one clock. Anything that is asynchronous to that clock needs to be treated very carefully.
  - A latch is an asynchronous storage element without clock involvement.
  - Sometimes, you may find glitches from latch's output.
  - Of course, sometimes latches are required. You just have to use them very rarely, only when absolutely required, and then you must design the logic right so there are no glitches possible.

Conditional Statement if-else

- How to avoid latch in combinational logic always block
  - Complete if-else assignment
  - Add default values. However, will sometimes break the logic in a design.
  - In sequential always block, there is not such issue.

```
wire Trigger , Pass ;
reg A , C ;

always @ ( * ) begin
            A = 1 ' b0 ;
            C = 1 ' b1 ;
            if ( Trigger ) begin
                        A = Pass ;
                        C = Pass ;
            end
end
```

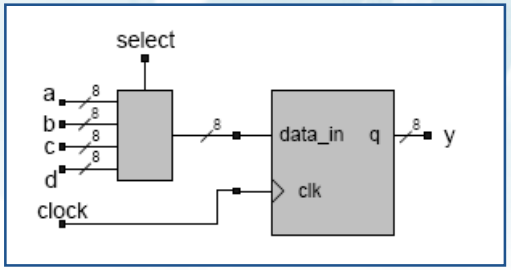
```
module testIf();
   reg latch;
   wire enable,din;
   always @ (enable or din)begin
      if (enable) begin
            latch = din;
      end else begin
            latch = 1'b0;
      end
   end
endmodule
```

Case Statement

- It must be within “always” combinational logic , sequential logic block or initial block(Non-syn).
- Match exactly 0, 1, x and z like "===". ?=X no more
- Synthsized hardware: non-priority Mux (Synthesizable)
- No break keyword in case

```
case ()
   < case1 > : < statement >
   < case2 > : < statement >
   .....
   default : < statement >
endcase
always @ (a or b or c or d or sel)
        case (sel)
                0 : y = a;
                1 : y = b;
                2 : y = c;
                3 : y = d;
        default : $display("Error in SEL");
endcase
```

```
always @ (posedge clock)
        case (select)
          0: y <= a; // non-blocking
          1: y <= b;
          2: y <= c;
          3: y <= d;
          default y <= 8'bx;
        endcase
```



Case Statement

- Different format in case statement

```
module testCase();
    reg[3:0]            Y;
    reg[3:0]            A;
    initial begin
        A = 4'b01xz;
        case(1'b1)
                A[3]: begin
                        Y<=4'b1000;
                        break;    // Error, automatically execute "break"
                      end
                A[2]: Y<=4'b0100;
                A[1]: Y<=4'b0010;
                A[0]: Y<=4'b0001;
                default:Y<=4'b0000;
      endcase
   end
endmodule
```

Case Statement

- Example:

```
module caseTest();
    reg [3:0]     X;
    initial begin
                X=4'b101x;
            case(X)     // Note: no break keyword in Verilog
                4'b1010: $display("Statement 1 has been selected!");
                4'b101x: $display("Statement 2 has been selected!");
                4'b101z: $display("Statement 3 has been selected!");
                4'bxxxx:  $display("Statement 4 has been selected!");
                4'bzzzz:  $display("Statement 5 has been selected!");
                default: $display("Default has been selected!");
             endcase
   end
endmodule
```

Ans: Statement 2 has been selected!

Case Statement

- Inferred latch issues in incomplete case statement

```
module case2 (a_i, b_i, c_i, sel_i,     d_o);
    input[7:0]        a_i,  b_i,  c_i;
    input[1:0]        sel_i;
    output[7:0]       d_o;

    reg[1:0]          sel_i;
    always@(*) begin
      case (sel_i)
            2'b00 : d_o = a_i;
            2'b01 : d_o = b_i;
            2'b10 : d_o = c_i;
            //default: d_o = 8'bx; w/o default, it infers latch
       endcase
    end
endmodule
```

Casex Statement

- It must be within “always” combinational logic or sequential logic block or initial block(**non-synth.**).
- Match: x = z and x/z = 0,1,x & z
- Synthsized hardware: non-priority Mux (Synthesizable)

```
module testCasex1();
        reg [3:0] X;
        initial begin
            X=4'b101x;
            casex(X)
                    4'b100z: $display("Statement 1 has been selected!");
                    4'b10xx: $display("Statement 2 has been selected!");
                    4'b11xz: $display("Statement 3 has been selected!");
                    4'bxxxx: $display("Statement 4 has been selected!");
                    4'bzzzz: $display("Statement 5 has been selected!");
                    default: $display("Default has been selected!");
            endcase
    end
endmodule
```

Ans: Statement 2 has been selected! But 101x = 4'bxxxx, 101x = 4'bzzzz

Notice that once first match is found and then execution jumps out.

Casex Statement

- In casex, ? = 0/1/x/z
- Don't recommend to use casex

```
module testCasex2();
        reg [3:0] X;

        initial begin
            X=4'b101x;
            casex(X)
                4'b???1: $display("Statement 1 has been selected!");
                4'b??1?: $display("Statement 2 has been selected!");
                4'b?1??: $display("Statement 3 has been selected!");
                4'b1???: $display("Statement 4 has been selected!");
                default: $display("Default has been selected!");
            endcase
        end
endmodule
```

Ans: Statement 1 has been selected! But 101x = 4'b??1?, 101x = 4'b1???

Casez Statement

- In casez, ? = 0/1/x/z, z = 0,1,z and x = 0,1,x; **x≠z**
- Use casez carefully

```
module testCasez1();
        reg [3:0] X;
        initial begin
            X=4'b1x00;
            casez(X)
                    4'b100z: $display("Statement 1 has been selected!");
                    4'b10xx: $display("Statement 2 has been selected!");
                    4'b11xz: $display("Statement 3 has been selected!");
                    4'bxxxx: $display("Statement 4 has been selected!");
                    4'b0zzz: $display("Statement 5 has been selected!");
                    default: $display("Default has been selected!");
             endcase
        end
endmodule
```

Ans: Statement 1 has been selected! But 1x00 = 4'b10xx,
1x00 = 4'b10xx, 1x00 = 4'b11xz, 1x00 = 4'bxxxx

Casez Statement

```
module testCasez2();
    reg [3:0] X;
    initial begin
       X=4'b1x0z;
       casez(X)
              4'b???1: $display("Statement 1 has been selected!");
              4'b??1?: $display("Statement 2 has been selected!");
              4'b?1??: $display("Statement 3 has been selected!");
              4'b1???: $display("Statement 4 has been selected!");
              default: $display("Default has been selected!");
        endcase
    end
endmodule
```

Ans:Statement 1 has been selected! But1x0z = 4'b?1??,
1x0z = 4'b1???,

Looping Statements

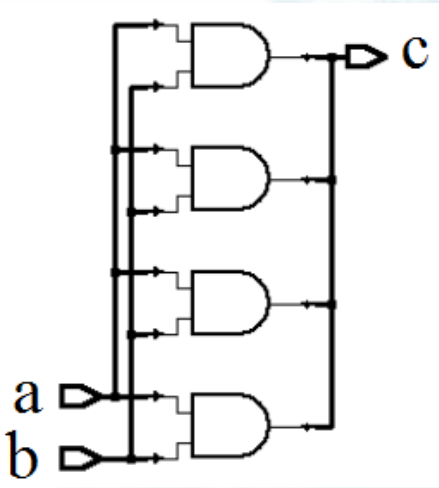
- for-loop Statement (Synthesizable)
  - In “always” combinational / sequential logic block (Synthesizable)
  - In “initial” block (Non-synthesizable)
- while-loop Statement
  - In “always” sequential logic block (Non-Synthesizable)
  - In “initial” block (Non-synthesizable)
- forever-loop Statement(Non-synthesizable)
- repeat-loop Statement(Non-synthesizable)

for-loop

- In “always” combinational logic block(Synthesizable)

```
wire[3:0]     a, b;
reg[3:0]      c;
integer        i;
always@( a or b ) begin
      for( i=0; i<4; i=i+1 ) begin
            c[i] = a[i] & b[i];
      end
end
-----------------------------------------------
wire[3:0]     a, b;
reg[3:0]      c;
always@( a or b ) begin
            c[0] = a[0] & b[0];
            c[1] = a[1] & b[1];
            c[2] = a[2] & b[2];
            c[3] = a[3] & b[3];
end
```

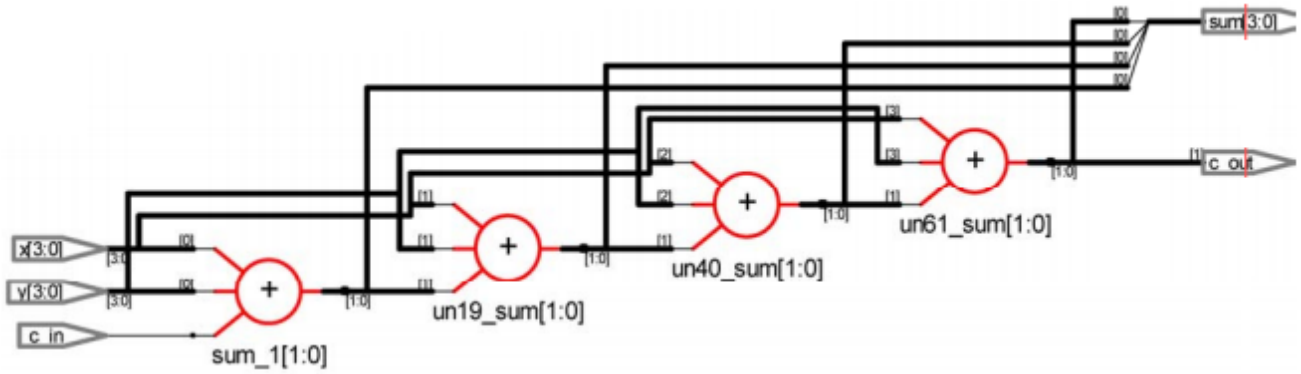
1. Verilog for-loop syntax:
   for(i=0; i<4; i = i+1)
   not like this: for(i=0; i<4; i ++)
2. Index must be integer
3. Start, step & end value must be constant
4. for-loop loops are“unrolled” in synthesis



for-loop

- In “always” combinational logic block(Synthesizable)

```
module nbitAdderFor( x, y, c_in, sum, c_out);
        parameter N = 4;
        input [N-1:0] x, y;  ... ...;
        integer i;
        always @(x or y or c_in) begin
                co = c_in;
                for (i = 0; i < N; i = i + 1)
                        {co, sum[i]} = x[i] + y[i] + co;
                        c_out = co;
        end
endmodule
```

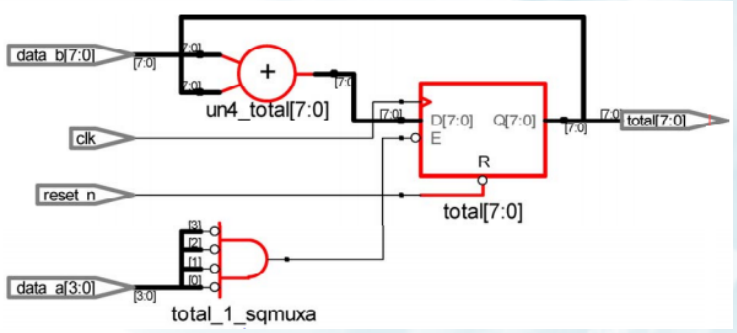


for-loop

- In “always” sequential logic block (Synthesizable)

```
module for_test(clk,n,m);
//programe for testing the for loop
input clk;
output [7:0] m;
output n;

reg [7:0] m;
integer n;
//What do you get?
 always @(posedge clk)begin
   for(n=2;n>=0;n=n-1)begin
      if(n==2) m <=8'h02;
      else if(n==1) m <=8'h03;
      else if(n==0) m <=8'h04;
   end
 end
endmodule
```

```
integer i;
// what does the following statement do?
always @(posedge clk or negedge reset_n)begin
    if (!reset_n) total <= 0;
    else for (i = 0; i < M; i = i + 1)
            if (data_a[i] == 1)
            total <= total + data_b;
end
```



for-loop

- In “initial” block (Non-synthesizable)

```
`timescale 1ns/1ps
module stimulus(xa, xb, xci);
        output [3:0] xa, xb;
        output xci;
        reg [3:0] xa, xb;
        reg xci;
        integer i, j, k;
        // Stimulus – generate all input combinations
        initial begin
                for (i=0; i<16; i=i+1)
                        for (j=0; j<16; j=j+1)
                                for (k=0; k<2; k=k+1) begin
                                        xa = i; xb = j; xci = k;
                                        #5;
                                end
        end // initial begin
endmodule // stimulus
```

while-loop

- In “always” block (Non-synthesizable, used in Testbench)

```
module while_example();
reg [5:0] loc;
reg [7:0] data;

always @ (data or loc) begin
  loc = 0;
  // If Data is 0, then loc is 32 (invalid value)
  if (data == 0) begin
    loc = 32;
  end else begin
    while (data[0] == 0) begin
      loc = loc + 1;
      data = data >> 1;
    end
  end
  $display ("DATA = %b   LOCATION =
%d",data,loc);
end
```

```
initial begin
  #1 data = 8'b11;
  #1 data = 8'b100;
  #1 data = 8'b1000;
  #1 data = 8'b1000_0000;
  #1 data = 8'b0;
  #1 $finish;
end

endmodule
```

Function: It is to detect the bit location where 1st one is.

Note: If condition is a x or z in "while" loop, it is treated as 0.

while-loop

- In “initial” block (Non-synthesizable)

```
module count_mod;

        integer count;

        initial begin
                count = 0;
                while (count < 16) begin
                        $display(" Count = %d", count);
                        count = count + 1;
                end
        end
endmodule
```

- forever/while can be synthesized when it contains @(posedge clk). But be careful to use it.