CONDITIONAL STATEMENTS

(BEHAVIORAL MODELING)

Contents

Conditional Statements

- Conditional statements are used for making decisions based upon certain conditions.
- These conditions are used to decide whether or not a statement should be executed.
- Keywords **if and else** are used for conditional statements. The syntax of an **if statement** is:

```
if ( condition_1 )
  procedural_statement_1
{ else if ( condition_2 )

  procedural_statement_2 }
[ else
  procedural_statement_3 ]
```

If conditional evaluates to a non-zero known value, then the procedural_statement_1 is executed. If conditional evaluates to a value 0, x or z, the procedural_statement_1 is not executed, and an else branch, if it exists, is executed.

```
if (Sum < 60)
  begin
    Grade = C;
    Total_C = Total_C + 1;
  end
else if (Sum < 75)
  begin
    Grade = B;
    Total_B = Total_B + 1;
  end
else
  begin
    Grade = A;
    Total_A = Total_A + 1;
  end
```

Note:

- ***The condition expression must always be within parenthesis.***
- ***Also there is a possibility for an ambiguity if an if-if-else form is use***

```
if (Clk)
  if (Reset)
    Q = 0;
  else
    Q = D;
```

Note: In the above case, the else associates with the closest if that does not have an else.
In this case else associates with the innermost if.

Case Statement

- A case statement is a multi-way conditional branch. It has the following syntax:

```
case ( case_expr )
  case_item_expr {, case_item_expr } : procedural_statement
  . . .
  . . .

 [ default: procedural_statement ]
endcase
```

- The case expression is evaluated first.
- Next the case item expressions are evaluated and compared in the order given.
- The set of statements that match the first true condition is executed.
- Multiple case item expressions can be specified in one branch; these values need not be mutually-exclusive.
- The default case covers all values that are not covered by the case item expressions.
- Neither the case expression nor the case item expressions need to be constant expressions.
- In a case statement, the x and z values are compared as their literal values.

```
//Execute statements based on the ALU control signal
reg [1:0] alu_control;
...
...
case (alu_control)
   2'd0 : y = x + z;
   2'd1 : y = x - z;
   2'd2 : y = x * z;
   default : $display("Invalid ALU control signal");
endcase
```

4:1 Multiplexer with Case Statement

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
reg out;

always @(s1 or s0 or i0 or i1 or i2 or i3)
case ({s1, s0}) //Switch based on concatenation of control signals
        2'd0 : out = i0;
        2'd1 : out = i1;
        2'd2 : out = i2;
        2'd3 : out = i3;
        default: $display("Invalid control signals");
endcase

endmodule
```

Don't-cares in Case

- In the case statement, the values x and z are interpreted literally, that is, as x and z values.
- There are two other forms of case statements', casex and casez, that use a different interpretation for x and z values.
- The syntax is exactly identical to that of a case statement except for the keywords casex and casez.

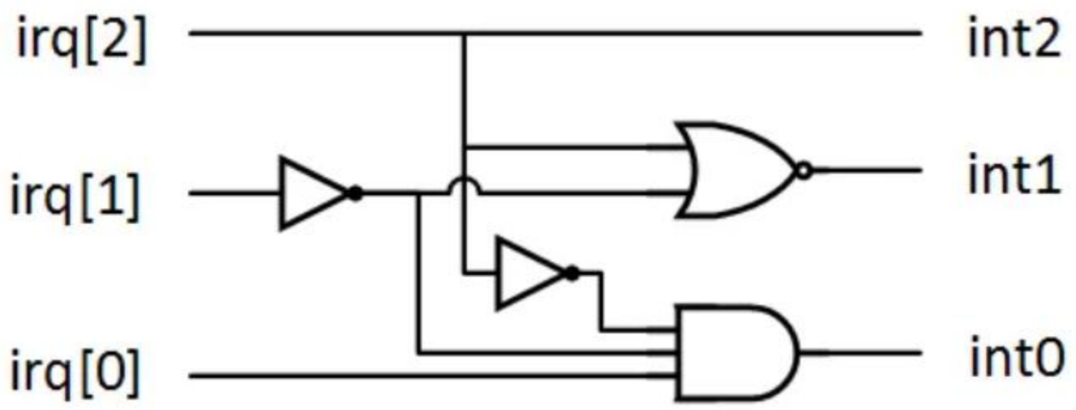
1. **casez:** In a casez statement, the value z that appears in the case expression and in any case item expression is considered as a don't-care, that is, that bit is ignored (not compared).
2. **casex:** In a casex statement, both the values x and z are considered as don't-cares.

Example of a casez statement.

```
casez (Mask)
  4'b1??? :  Dbus[4] = 0;
  4'b01?? :  Dbus[3] = 0;
  4'b001? :  Dbus[2] = 0;
  4'b0001 :  Dbus[1] = 0;
endcase
```

- The ? Character can be used instead of the character z to imply a don't-care.
- The casez statement example implies that if the first bit of Mask is 1 (other bits of Mask are ignored), then 0 is assigned to Dbus[4], if first bit of Mask is 0 and the second bit is 1 (other bits are ignored),then Dbus[3] gets assigned the value 0, and so on.

Write the code using casez



```
always @(irq) begin
  {int2, int1, int0} = 3'b000;
  casez (irq)
    3'b1?? : int2 = 1'b1;
    3'b?1? : int1 = 1'b1;
    3'b??1 : int0 = 1'b1;
    default: {int2, int1, int0} = 3'b000;
  endcase
end
```

Mux using casez

```
module mux4to1(sel, a0, a1, a2, a3, out);
        input [2:0] sel;
        input [3:0] a0, a1, a2, a3;
        output reg[3:0] c;

        always @(sel or a0 or a1 or a2 or a3)
        begin
                casez(sel)
                        3'b000: out = a0;
                        3'b001: out = a1;
                        3'b010: out = a2;
                        3'b011: out = a3;
                        3'b1??: out = 4'b0000;
                endcase
        end
endmodule
```

Example of casex Use

```
reg [3:0] encoding;
integer state;

casex (encoding) //logic value x represents a don't care bit.
4'b1xxx : next_state = 3;
4'bx1xx : next_state = 2;
4'bxx1x : next_state = 1;
4'bxxx1 : next_state = 0;
default : next_state = 0;
endcase
```

Thus, an input encoding = 4'b10xz would cause next_state = 3 to be executed.

Behavioral Modeling of 4:2 Priority Encoder

Y3	Y2	Y1	Y0	A1	A0
0	0	0	1	0	0
0	0	1	x	0	1
0	1	x	x	1	0
1	x	x	x	1	1

```
module priority_encoderbehave(A, Y);

input [3:0]Y;
output reg [1:0]A;

always@(Y)
begin

casex(Y)

4'b0001:A = 2'b00;
4'b001x:A = 2'b01;
4'b01xx:A = 2'b10;
4'b1xxx:A = 2'b11;
default:$display("Error!");
endcase
end
endmodule
```

Loop Statement

- There are four kinds of loop statements. These are:

1. Forever-loop
2. Repeat-loop
3. While-loop
4. For-loop

1. Forever-loop Statement

- The keyword forever is used to express this loop.
- The loop does not contain any expression and executes forever until the $finish task is encountered.
- The loop is equivalent to a while loop with an expression that always evaluates to true, e.g., while (1).
- This loop statement continuously executes the procedural statement. Thus to get out of such a loop, a disable statement may be used with the procedural statement.
- Also, some form of timing controls must be used in the procedure
- If timing control constructs are not used, the Verilog simulator would execute this statement infinitely without advancing simulation time and the rest of the design would never be executed.

The syntax for this form of loop statement is:

```
forever
  procedural_statement
```

Example 1: Clock generation

```
//Use forever loop instead of always block
reg clock;

initial
begin
        clock = 1'b0;
        forever #10 clock = ~clock; //Clock with period of 20 units
end
```

OR,

```
initial
  begin
    Clock = 0;
    #5 forever
      # 10 Clock = ~ Clock;
  end
```

2. Repeat-loop Statement

- The keyword repeat is used for this loop.
- The repeat construct executes the loop a fixed number of times.
- A repeat construct cannot be used to loop on a general logical expression. A while loop is used for that purpose.
- A repeat construct must contain a number, which can be a constant, a variable or a signal value.
- However, if the number is a variable or signal value, it is evaluated only when the loop starts and not during the loop execution.
- If loop count expression is an x or a z, then the loop count is treated as a 0

This form of loop statement has the form:

```
repeat ( loop_count )
  procedural_statement
```

Q. Increment and display count from 0 to 127

```
integer count;

initial
begin
    count = 0;
    repeat(128)
    begin
          $display("Count = %d", count);
          count = count + 1;
    end
end
```

Q. Model a data buffer that latches data at the positive edge of clock for the next eight cycles after it receives a data start signal.

```
module data_buffer(data_start, data, clock);

parameter cycles = 8;
input data_start;
input [15:0] data;
input clock;

reg [15:0] buffer [0:7];
integer i;

 always @(posedge clock)
 begin
   if(data_start) //data start signal is true
   begin
     i = 0;
```

```
    repeat(cycles) //Store data at the posedge of next 8 clock
                   //cycles
    begin
      @(posedge clock) buffer[i] = data; //waits till next
                                         // posedge to latch data
      i = i + 1;
    end
  end
end

endmodule
```

3. While-loop Statement

- This loop executes the procedural statement until the specified condition becomes false.
- If the expression is false to begin with, then the procedural statement is never executed.
- If the condition is an x or a z, it is treated as a 0 (false).

The syntax of this form of loop statement is:

```
while ( condition )
  procedural_statement
```

Q. Increment count from 0 to 127. Exit at count 128

```
integer count;

initial
begin
        count = 0;

        while (count < 128) //Execute loop till count is 127.
                            //exit at count 128
        begin
                $display("Count = %d", count);
               count = count + 1;
        end
end
```

Q. Find the first bit with a value 1 in flag (vector variable)

```
'define TRUE 1'b1';
'define FALSE 1'b0;
reg [15:0] flag;
integer i; //integer to keep count
reg continue;

initial
begin
   flag = 16'b 0010_0000_0000_0000;
   i = 0;
   continue = 'TRUE;

  while((i < 16) && continue )    //Multiple conditions using operators
  begin
    if (flag[i])
```

```
      begin
        $display("Encountered a TRUE bit at element number %d", i);
        continue = 'FALSE;
      end
      i = i + 1;
    end
end
```

4. For-loop Statement

- A for-loop statement repeats the execution of the procedural statement a certain number of times.
- The initial_assignment specifies the initial value of the loop index.
- The condition specifies the condition when loop execution must stop.
- As long as the condition is true, the statements in the loop are executed.
- The step_assignment specifies the assignment to modify, typically to increment or decrement, the step count.

This loop statement is of the form:

```
for ( initial_assignment ; condition ; step_assignment )
  procedural_statement
```

E.g. 1

```
integer count;

initial
    for ( count=0; count < 128; count = count + 1)
           $display("Count = %d", count);
```

E.g. 2

```
integer K;

for (K = 0; K < MAX_RANGE; K = K + 1)
  begin
    if (Abus[K] == 0)
      Abus[K] = 1;
    else if (Abus[K] == 1)
      Abus[K] = 0;
    else
      $display ("Abus[K] is an x or a z");
  end
```

Procedural Continuous Assignment

- A procedural continuous assignment is a procedural assignment, that is, it can appear within an always statement or an initial statement.
- This assignment can override all other assignments to a net or a register.
- It allows the expression in the assignment to be driven continuously into a register or a net.
- There are two kinds of procedural continuous assignments.

1. **Assign and deassign** procedural statements: these assign to registers
2. **Force and release** procedural statements: these assign primarily to nets, though they can also be used for registers.

- Note, this is not a continuous assignment; a continuous assignment occurs outside of an initial or an always statement.
- The assign and force statements are "continuous" in the sense that any change of operand in the right-hand side expression causes the assignment to be redone while the assign or force is in effect.
- The target of a procedural continuous assignment cannot be a part-select or a bit-select of a register.

Assign - deassign

```
module DFF (D, Clr, Clk, Q);
  input D, Clr, Clk;
  output Q;
  reg Q;

  always
    @(Clr) begin
      if (! Clr)        // D has no effect on Q
        assign Q = 0;         //
      else
        deassign Q;
  end


  always
    @(negedge Clk) Q = D;
endmodule
```

- An assign procedural statement overrides all procedural assignments to a register.
- The deassign procedural statement ends the continuous assignment to a register. The value in the register is retained until assigned again.

  If Clr s 0, the assign procedural statement causes Q to be stuck at 0 irrespective of any clock edges, that is, Clk and D have no effect on Q.
- If Clr becomes 1, the deassign statement is executed; this causes the override to be removed, so that in the future Clk can effect Q.

Force - release

- The force and release procedural statements are very similar to assign and deassign, except that force and release can be applied to nets as well as to registers.
- The force statement, when applied to a register, causes the current value of the register to be overridden by the value of the force expression.
- When a release on the register is executed, the current value is held in the register unless a procedural continuous assignment was already in effect (at the time the force statement was executed)in which case, the continuous assignment establishes the new value of the register.
- A force procedural statement on a net overrides all the drivers for the net until a release procedural statement is executed on that net.

```
  wire Prt;
  . . .
  or #1 (Prt, Std, Dzx);

initial
  begin
    force Prt = Dzx & Std;
    #5;    // Wait for 5 time units.
    release Prt;
  end
```

The execution of the force statement causes the value of Prt to override the value from the or gate primitive until the release statement is executed, upon which the driver of Prt from the or gate primitive takes back its effect. While the force assignment is in effect (first 5 time units), any changes on Dzx and Std cause the assignment to be executed again

Write behavioral model to Implement a 8-bit binary counter

```
module ( count, reset, clk );

output [7:0] count;
reg [7:0] count;

input reset, clk;

// consider reset as active low signal

always @( posedge clk, negedge reset)
begin
 if(reset == 1'b0)
     count <= 8'h00;
 else
     count <= count + 8'h01;
end

endmodule
```

THANK YOU