INTRODUCTION TO VERILOG (Contd..)

EC39004 : VLSI LABORATORY

28 January 2019

Case statement

- case statement syntax

```
case (<expr>)
expr1:  seq_sta;
expr2:  seq_sta;
……..
exprn: seq_sta;
default: default  sta;
endcase
```

- A "case" statement can replace a complex "if…else" statement for multiway branching
- The expression "expr" is compared to the alternatives (expr1, expr2, etc.) in the order they are written
- If none of the alternatives match, the default statement is executed
- If "expr" matches with more than 1 condition, the 1st match is taken

Case statement

- Two variations: **"casez"** and **"casex".**
- The **"casez"** statement treats all **"z"** values in the case alternatives or the case expression as don't cares.
- The **"casex"** statement treats all **"x"** and **"z"** values in the case item as don't cares.

- If state is **"4'b01zx",** the second expression will give match, and next_state will be 1.

```
reg [3:0] state;
integer next_state;
 casex (state)
4'b1xxx : next_state = 0;
4'bx1xx : next_state = 1;
4'bxx1x : next_state = 2;
4'bxxx1 : next_state = 3;
default : next_state = 0;
endcase
```

Case statement (Example)

```
module case_ex (input a, b, c, d;
                input [1:0] sel,
                output reg mux_out);
```

```
always@(*)  begin
case (sel)
2'b00 : mux_out=a;
2'b01: mux_out=b;
2'b10: mux_out=c;
2'b11: mux_out=d;
default: mux_out=0;
end
endmodule
```

```
always @ (*)  begin
if (sel == 2'b00)  mux_out =a;
else if (sel == 2'b01)  mux_out = b;
else if (sel == 2'b10)  mux_out = c;
else if (sel == 2'b11)  mux_out = d;
else  mux_out =0;
end
endmodule
```

While Loop

```
while (<expression>)
    sequential_statement;
```

- The "while" loop executes until the expression is *not true*.
- The sequential_statement can be a single statement or a group of statements within "begin … end".
- Typically used in test benches; it is non-synthesizable

Example:

```
integer mycount;
initial
begin
    while (mycount <= 255)
        begin
          $display ("My count:%d",mycount);
          mycount = mycount + 1;
         end
end
```

For Loop

```
for (expr1; expr2; expr3)
    sequential_statement;
```

- The "for" loop consists of three parts:
  a) An initial condition (expr1).
  b) A check to see if the terminating condition is true (expr2).
  c) A procedural assignment to change the value of the control variable (expr3).
- The "for" loop can be conveniently used to initialize an array or memory.
- The "for" loop executes as long as the expression expr2 is true.
- The sequential_statement can be a single statement or a group of statements within "begin …end".

For Loop (Example)

Example:

```
integer mycount;  reg [100:1] data;  integer i;
initial
 for (mycount=0; mycount<=255; mycount=mycount+1)
  $display ("My count:%d", mycount);
initial
 for (i=1; i<=100; i=i+1)
  data[i] = 1'b0;
```

Repeat Loop

```
repeat (<expression>)
    sequential_statement;
```

- The expression in the "**repeat**" construct can be a constant, a variable or a signal value.
- If it is a variable or a signal value, it is evaluated only when the loop starts and not during execution of the loop.
- The sequential_statement can be a single statement or a group of statements within "begin …end".
- The "repeat" construct executes the loop a fixed number of times.
- It cannot be used to loop on a general logical expression like "while".

Repeat Loop (Example)

Example:

```
reg clock;
initial
 begin
  clock = 1'b0;
  repeat (100)
   #5 clock =
   ~clock;
 end
```

Forever Loop

```
forever
   sequential_statement;
```

- The "forever" construct does not use any expression and executes forever until $finish is encountered in the test bench.
- Equivalent to a "while" loop for which the expression is always true. while (1<2) is equivalent to "forever"
- The sequential_statement can be a single statement or a group of statements within "begin …end".
- The "forever" loop is typically used along with timing specifier.
- If delay is not specified, the simulator would execute this statement indefinitely without advancing $time.
- Rest of design will never be executed.

Forever Loop (Example)

```
// Clock generation using “forever” construct
reg clk;
initial
   begin
      clk = 1’b0;
      forever #5 clk = ~clk;   // Clock period of
 10 units
  end
```

@ (event_expression)

- The event expression specifies the event that is required to resume execution of the procedural block.
- The event can be any one of the following:
  1. Change of a signal value.
  2. Positive or negative edge occurring on signal (*posedge* or *negedge*).
  3. List of above-mentioned events, separated by "or" or comma.
- A "posedge" is any transition from {0, x, z} to 1, and from 0 to {z, x}.
- A "negedge" is any transition from {1, x, z} to 0, and from 1 to {z, x}.

Examples:

```
@ (in)                          // "in" changes
@ (a or b or c)                 // any of "a", "b", "c" changes
@ (a, b, c)                     // -- do --
@ (posedge clk)                 // positive edge of "clk"
@ (posedge clk or negedge reset) // positive edge of "clk" or negative edge of "reset"
@(*)                            // any variable changes
```

Procedural Assignment

- Procedural assignment statements can be used to update variables of types "reg", "integer", "real" or "time".

- The value assigned to a variable remains unchanged until another procedural assignment statement assigns a new value to the variable.

  – This is different from continuous assignment (using "assign") that results in the expression on the RHS to continuously drive the "net" type variable on the left.

- Two types of procedural assignment statements:

i. ***Blocking*** (denoted by "=")

ii. ***Non-blocking*** (denoted by "<=")

(i) Blocking Assignment

- General syntax:

  variable_name = [delay_or_event_control] expression;

- The "=" operator is used to specify blocking assignment.

- Blocking assignment statements are executed in the order they are specified in a procedural block.
  - The target of an assignments gets updated before the next sequential statement in the procedural block is executed.
  - They do not block execution of statements in other procedural blocks.

- This is the recommended style for modeling combinational logic.

(i) Blocking Assignment

- Blocking assignments can also generate sequential circuit elements during synthesis (e.g. incomplete specification in multi-way branching with "case").
- An example of blocking assignment:

```
integer a, b, c;  initial
  begin
    a = 10; b = 20; c = 15;
    a = b + c;  b = a + 5;  c = a – b;
  end
```

Initially, a=10, b=20, c=15
a becomes 35
b becomes 40
c becomes -5

Simulation of an Example (Blocking Statement)

```
module blocking_assgn;  integer
a, b, c, d;
Initially:
 a=30, b=20, c=15, d=5
always @ (*)  repeat (4)
begin
   #5 a = b + c;
   #5 d = a - 3;
   #5 b = d + 10;
   #5 c = c + 1;
 end
```

```
initial  begin
   $monitor ($time, "a=%4d, b=%4d,
      c=%4d, d=%4d", a, b, c, d);
   a = 30; b = 20; c = 15;     d = 5;
   #100 $finish;  end
endmodule
```

```
  0 a=   30, b=  20, c=   15, d=    5
  5 a=   35, b=  20, c=   15, d=    5
 10 a=   35, b=  20, c=   15, d=   32
 15 a=   35, b=  42, c=   15, d=   32
 20 a=   35, b=  42, c=   16, d=   32
 25 a=   58, b=  42, c=   16, d=   32
 30 a=   58, b=  42, c=   16, d=   55
 35 a=   58, b=  65, c=   16, d=   55
 40 a=   58, b=  65, c=   17, d=   55
 45 a=   82, b=  65, c=   17, d=   55
 50 a=   82, b=  65, c=   17, d=   79
 55 a=   82, b=  89, c=   17, d=   79
 60 a=   82, b=  89, c=   18, d=   79
 65 a=  107, b=  89, c=   18, d=   79
 70 a=  107, b=  89, c=   18, d=  104
 75 a=  107, b= 114, c=   18, d=  104
 80 a=  107, b= 114, c=   19, d=  104
```

(ii) Non-Blocking Assignment

- General syntax:

  variable_name <= [delay_or_event_control] expression;

- The "<="operator is used to specify non-blocking assignment.

- Non-blocking assignment statements allow scheduling of assignments without blocking execution of statements that follow within the procedural block.

  – The assignment to the target gets scheduled for the end of the simulation cycle (at the end of the procedural block).

  – Statements subsequent to the instruction under consideration are not blocked by the assignment.

  – Allows concurrent procedural assignment, suitable for sequential logic.

(ii) Non-Blocking Assignment

This is the recommended style for modeling sequential logic.

--Several "reg" type variables can be assigned synchronously, under the control of a common clock.

```
integer  a, b, c;
initial  begin
    a = 10; b = 20; c = 15;
  end
initial  begin
    a <= #5 b + c;  b <= #5 a
    + 5;
    c <= #5 a – b;  end
```

Initially, a=10, b=20, c=15

a becomes 35 at time =5

b becomes 15 at time =5

c becomes -10 at time =5

All the right hand side expressions are evaluated together based on the previous values of "a", "b" and "c".

They are assigned together at time 5.

Swapping values of two variables “a” and “b”

```
always @(posedge clk)
  a = b;
always @(posedge clk)
  b = a;
```

- Either a=b will execute before b=a, or vice versa, depending on simulator implementation.
- Both registers will get the same value (either “a” or “b”).
  - Race condition.

```
always @(posedge clk)
  a <= b;
always @(posedge clk)
  b <= a;
```

- Here the variables are correctly swapped.
- All RHS variables are read first, and assigned to LHS variables at the positive clock edge.

Trying to swap using blocking assignment

```
always @(posedge clk)
  begin
    a = b;  b = a;
  end
```

- Both “a” and “b” will be getting the value previously stored in “b”.

```
always @(posedge
  clk)  begin
    ta = a;  tb = b;
    a = tb;  b = ta;
  end
```

- Correct swapping will occur, but we need two temporary variables “ta” and “tb”

Simulation of an Example

```
module nonblocking_assgn;
integer a, b, c, d;
reg clock
always @ (posedge clock)
  begin
    a <= b + c;
    d <= a - 3;
     b <= d + 10;
     c <= c + 1;
  end
```

```
initial
begin
   $monitor ($time, "a=%4d, b=%4d,  c=%4d, d=%4d", a, b, c, d);
   a = 30;
   b = 20;
   c = 15;
   d = 5;
   clock = 0;
   forever  #5 clock = ~clock;
   end
initial
   #100 $finish;
   endmodule
```

Simulation Results

```
 0 a=   30, b=  20, c=   15, d=     5
 5 a=   35, b=  15, c=   16, d=    27
15 a=   31, b=  37, c=   17, d=    32
25 a=   54, b=  42, c=   18, d=    28
35 a=   60, b=  38, c=   19, d=    51
45 a=   57, b=  61, c=   20, d=    57
55 a=   81, b=  67, c=   21, d=    54
65 a=   88, b=  64, c=   22, d=    78
75 a=   86, b=  88, c=   23, d=    85
85 a=  111, b=  95, c=   24, d=    83
95 a=  119, b=  93, c=   25, d=   108
```

```
Initially:
  a=30, b=20, c=15, d=5
always @ (posedge clock)
  begin
    a <= b + c;
    d <= a - 3;  b <= d +
    10;  c <= c + 1;
  end
```

Some Rules to be Followed

- It is recommended that blocking and non-blocking assignments are not mixed in the same "always" block.
  - Simulator may allow, but this is not good design practice.
- Verilog synthesizer ignores the delays specified in a procedural assignment statement (blocking or non-blocking).
  - May lead to functional mismatch between the design model and the synthesized netlist.
- A variable cannot appear as the target of both a blocking and a non-blocking assignment.

– This is not permissible →

```
x = x + 5;
x <= y;
```

Assignment:Implement Signed (Baugh-Wooley)Multiplier