Verilog Behavioral Modeling Part-V

Feb-9-2014

Procedural Block Control

Procedural blocks become active at simulation time zero. Use level sensitive event controls to control the execution of a procedure.

```
module dlatch_using_always();
reg q;

reg d, enable;

always @ (d or enable)
if (enable) begin
  q = d;
end

initial begin
  $monitor (" ENABLE = %b D = %b  Q = %b",enable,d,q);
   #1  enable = 0;
   #1  d = 1;
   #1  enable = 1;
   #1  d = 0;
   #1  d = 1;
   #1  d = 0;
   #1  enable = 0;
   #10  $finish;
end

endmodule
```

You could download file dlatch_using_always.v here

Any change in either d or enable satisfies the event control and allows the execution of the statements in the procedure. The procedure is sensitive to any change in d or enable.

Combo Logic using Procedural Coding

To model combinational logic, a procedure block must be sensitive to any change on the input. There is one important rule that needs to be followed while modeling combinational logic. If you use conditional checking using "if", then you need to mention the "else" part. Missing the else part results in a latch. If you don't like typing the else part, then you must initialize all the variables of that combo block as soon as it enters.

Example - One bit Adder

```
module adder_using_always ();
reg a, b;
reg sum, carry;

always @ (a or b)
begin
  {carry,sum} = a + b;
end

initial begin
```

```
  $monitor (" A = %b B = %b CARRY = %b SUM = %b",a,b,carry,sum);
  #10  a = 0;
   b = 0;
   #10  a = 1;
   #10  b = 1;
   #10  a = 0;
   #10  b = 0;
   #10  $finish;
end

endmodule
```

You could download file adder_using_always.v here

The statements within the procedural block work with entire vectors at a time.

Example - 4-bit Adder

```
module adder_4_bit_using_always ();
reg[3:0] a, b;
reg [3:0] sum;
reg carry;

always @ (a or b)
begin
  {carry,sum} = a + b;
end

initial begin
  $monitor (" A = %b B = %b CARRY = %b SUM = %b",a,b,carry,sum);
  #10  a = 8;
   b = 7;
   #10  a = 10;
   #10  b = 15;
   #10  a = 0;
   #10  b = 0;
   #10  $finish;
end

endmodule
```

You could download file adder_4_bit_using_always.v here

Example - Ways to avoid Latches - Cover all conditions

```
module avoid_latch_else ();

reg q;
reg enable, d;

always @ (enable or d)
if (enable) begin
  q = d;
end else begin
  q = 0;
end

initial begin
  $monitor (" ENABLE = %b  D = %b Q = %b",enable,d,q);
  #1  enable = 0;
  #1  d = 0;
  #1  enable = 1;
  #1  d = 1;
  #1  d = 0;
  #1  d = 1;
  #1  d = 0;
  #1  d = 1;
  #1  enable = 0;
  #1  $finish;
end

endmodule
```

You could download file avoid_latch_else.v here

Example - Ways to avoid Latches - Snit the variables to zero

```
module avoid_latch_init ();

reg q;
reg enable, d;

always @ (enable or d)
begin
  q = 0;
  if (enable) begin
    q = d;
  end
end

initial begin
  $monitor (" ENABLE = %b  D = %b Q = %b",enable,d,q);
  #1  enable = 0;
  #1  d = 0;
  #1  enable = 1;
  #1  d = 1;
  #1  d = 0;
  #1  d = 1;
  #1  d = 0;
  #1  d = 1;
  #1  enable = 0;
  #1  $finish;
end

endmodule
```

You could download file avoid_latch_init.v here

Sequential Logic using Procedural Coding

To model sequential logic, a procedure block must be sensitive to positive edge or negative edge of clock. To model asynchronous reset, procedure block must be sensitive to both clock and reset. All the assignments to sequential logic should be made through nonblocking assignments.

Sometimes it's tempting to have multiple edge triggering variables in the sensitive list: this is fine for simulation. But for synthesis this does not make sense, as in real life, flip-flop can have only one clock, one reset and one preset (i.e. posedge clk or posedge reset or posedge preset).

One common mistake the new beginner makes is using clock as the enable input to flip-flop. This is fine for simulation, but for synthesis, this is not right.

Example - Bad coding - Using two clocks

```
module wrong_seq();

reg q;
reg clk1, clk2, d1, d2;

always @ (posedge clk1 or posedge clk2)
if (clk1) begin
  q <= d1;
end else if (clk2) begin
  q <= d2;
end
```

```

initial begin
  $monitor ("CLK1 = %b CLK2 = %b D1 = %b D2 %b Q = %b",
    clk1, clk2, d1, d2, q);
  clk1 = 0;
  clk2 = 0;
  d1 = 0;
  d2 = 1;
  #10  $finish;
end

always
 #1  clk1 = ~clk1;

always
 #1.9 clk2 = ~clk2;

endmodule
```

You could download file wrong_seq.v here

Example - D Flip-flop with async reset and async preset

```
module dff_async_reset_async_preset();

reg clk,reset,preset,d;
reg  q;

always @ (posedge clk or posedge reset or posedge preset)
if (reset) begin
  q <= 0;
end else if (preset) begin
  q <= 1;
end else begin
  q <= d;
end

// Testbench code here
initial begin
  $monitor("CLK = %b RESET = %b PRESET = %b D = %b Q = %b",
    clk,reset,preset,d,q);
  clk    = 0;
  #1  reset  = 0;
  preset = 0;
  d      = 0;
  #1  reset = 1;
  #2  reset = 0;
  #2  preset = 1;
  #2  preset = 0;
  repeat (4) begin
    #2  d      = ~d;
  end
  #2  $finish;
end

always
 #1  clk = ~clk;

endmodule
```

You could download file dff_async_reset_async_preset.v here

Example - D Flip-flop with sync reset and sync preset

```
module dff_sync_reset_sync_preset();

reg clk,reset,preset,d;
reg  q;

```

```
always @ (posedge clk)
if (reset) begin
  q <= 0;
end else if (preset) begin
  q <= 1;
end else begin
  q <= d;
end

// Testbench code here
initial begin
  $monitor("CLK = %b RESET = %b PRESET = %b D = %b Q = %b",
    clk,reset,preset,d,q);
  clk    = 0;
  #1  reset  = 0;
  preset = 0;
  d      = 0;
  #1  reset = 1;
  #2  reset = 0;
  #2  preset = 1;
  #2  preset = 0;
  repeat (4) begin
    #2  d      = ~d;
  end
  #2  $finish;
end

always
 #1  clk = ~clk;

endmodule
```

You could download file dff_sync_reset_sync_preset.v here

A procedure can't trigger itself

One cannot trigger the block with a variable that block assigns value or drives.

```
module trigger_itself();

reg clk;

always @ (clk)
  #5  clk = ! clk;

// Testbench code here
initial begin
  $monitor("TIME = %d  CLK = %b",$time,clk);
  clk = 0;
  #500  $display("TIME = %d  CLK = %b",$time,clk);
  $finish;
end

endmodule
```

You could download file trigger_itself.v here

Procedural Block Concurrency

If we have multiple always blocks inside one module, then all the blocks (i.e. all the always blocks and initial blocks) will start executing at time 0 and will continue to execute concurrently. Sometimes this leads to race conditions, if coding is not done properly.

```
module multiple_blocks ();
reg a,b;
reg c,d;
reg clk,reset;
// Combo Logic
```

```
always @ ( c)
begin
  a = c;
end
// Seq Logic
always @ (posedge clk)
if (reset) begin
  b <= 0;
end else begin
  b <= a & d;
end

// Testbench code here
initial begin
  $monitor("TIME = %d CLK = %b C = %b D = %b A = %b B = %b",
    $time, clk,c,d,a,b);
  clk = 0;
  reset = 0;
  c = 0;
  d = 0;
  #2 reset = 1;
  #2 reset = 0;
  #2 c = 1;
  #2 d = 1;
  #2 c = 0;
  #5 $finish;
end
// Clock generator
always
 #1 clk = ~clk;

endmodule
```

You could download file multiple_blocks.v here

Race condition

```
module race_condition();
reg b;

initial begin
  b = 0;
end

initial begin
  b = 1;
end

endmodule
```

You could download file race_condition.v here

In the code above it is difficult to say the value of b, as both blocks are supposed to execute at same time. In Verilog, if care is not taken, a race condition is something that occurs very often.

Named Blocks

Blocks can be named by adding : block_name after the keyword begin. Named blocks can be disabled using the 'disable' statement.

Example - Named Blocks

```
// This code find the lowest bit set
module named_block_disable();

reg [31:0] bit_detect;
reg [5:0]  bit_position;
integer i;

always @ (bit_detect)
begin : BIT_DETECT
  for (i = 0; i < 32 ; i = i + 1) begin
     // If bit is set, latch the bit position
     // Disable the execution of the block
     if (bit_detect[i] == 1) begin
        bit_position = i;
        disable BIT_DETECT;
     end  else begin
        bit_position = 32;
     end
  end
end

// Testbench code here
initial begin
  $monitor(" INPUT = %b  MIN_POSITION = %d", bit_detect, bit_position);
  #1  bit_detect = 32'h1000_1000;
  #1  bit_detect = 32'h1100_0000;
  #1  bit_detect = 32'h1000_1010;
  #10  $finish;
end

endmodule
```

You could download file named_block_disable.v here

In the example above, BIT_DETECT is the named block and it is disabled whenever the bit position is detected.