A Short Tutorial
on
Hardware Modeling using Verilog

Part 2

1

Behavioral Style: Procedural Assignment

- Two kinds of procedural blocks are supported in Verilog:
  - The “initial” block
    - Executed once at the beginning of simulation.
    - Used only in test benches; cannot be used in synthesis.
  - The “always” block
    - A continuous loop that never terminates
- The procedural block defines:
  - A region of code containing *sequential* statements.
  - The statements execute in the order they are written.

2

2

The "initial" Block

- All statements inside an "*initial*" statement constitute an "*initial block*".
  - Grouped inside a "*begin … end*" structure for multiple statements.
  - The statements starts at time 0, and execute only once.
  - If there are multiple "*initial*" blocks, all the blocks will start to execute concurrently at time 0.
- The "initial" block is typically used to write *test benches* for simulation:
  - Specifies the stimulus to be applied to the design-under-test (DUT).
  - Specifies how the DUT outputs are to be displayed / handled.
  - Specifies the file where the waveform information is to be dumped.

3

3

```
module testbench_example;
  reg a, b, cin, sum, cout;

  initial
    cin = 1'b0;

  initial
    begin
      #5 a = 1'b1; b=1'b1;
      #5 b = 1'b0;
    end

  initial
    #25 $finish;

endmodule
```

- The three "initial" blocks execute concurrently.
- The first block executes at time 0.
- The third block terminates simulation at time 25 units.

4

4

The "always" Block

- All behavioral statements inside an "*always*" statement constitute an "*always block*".
  - Multiple statements are grouped using "begin … end".
- An "always" statement starts at time 0 and executes the statements inside the block *repeatedly, and never stops*.
  - Used to model a block of activity that is repeated indefinitely in a digital circuit.
  - For example, a clock signal that is generated continuously.
  - We can specify delays for simulation; however, for real circuits, the clock generator will be active as long as there is power supply.

5

5

```
module generating_clock;
  output reg clk;

  initial
    clk = 1'b0;  // initialized to 0 at time 0

  always
    #5 clk = ~clk;  // Toggle after time 5 units

  initial
    #500 $finish;
endmodule
```

- "initial" and "always blocks can coexist within the same Verilog module.
- They all execute concurrently; "initial" only once and "always" repeatedly.

6

6

- A module can contain any number of "always" blocks, all of which execute concurrently.
- The @(event_expression) part is required for both combinational and sequential circuit descriptions.

Basic syntax of "always" block:

```
always @(event_expression)
begin
  sequential_statement_1;
  sequential_statement_2;
  …
  sequential_statement_n;
end
```

7

7

- Only "reg" type variable can be assigned within an "initial" or 'always" block.

8

8

Sequential Statements in Verilog

- In Verilog, one of more sequential statements can be present inside an "initial" or "always" block.
  - The statements are executed sequentially.
  - Multiple assignment statements inside a "begin … end" block may either execute sequentially or concurrently depending upon on the type of assignment.
    - Two types of assignment statements: blocking (`a = b + c;`) or non-blocking (`a <= b + c;`).
- The sequential statements are explained next.

9

9

(a) begin … end

```
begin
  sequential_statement_1;
  sequential_statement_2;
  …
  sequential_statement_n;
end
```

- A number of sequential statements can be grouped together using "begin .. end".
- If n=1, "begin … end" is not required.

10

10

(b) if … else

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

```
if (<expression>)
  sequential_statement;
else
  sequential_statement;
```

```
if (<expression1>)
  sequential_statement;
else if (<expression2>)
  sequential_statement;
else if (<expression3>)
  sequential_statement;
else default_statement;
```

- Each sequential_statement can be a single statement or a group of statements within "begin … end".

11

11

(c) case

```
case (<expression>)
  expr1: sequential_statement;
  expr2: sequential_statement;
  ...
  exprn: sequential_statement;
  default: default_statement;
endcase
```

- Each sequential_statement can be a single statement or a group of statements within "begin … end".
- Can replace a complex "if … else" statement for multiway branching.
- The expression is compared to the alternatives (expr1, expr2, etc.) in the order they are written.
- If none of the alternatives matches, the default statement is executed.

12

12

- 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
```

13

13

(d) "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".

Example:

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

14

14

(e) "for" loop

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

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;
```

15

15

(f) "repeat" loop

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

- The "repeat" construct executes the loop a fixed number of times.
- It cannot be used to loop on a general logical expression like "while".

Example:

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

Exactly 100 clock pulses are generated.

16

16

(g) “forever” loop

```
forever
  sequential_statement;
```

```
// Clock generation using "forever" construct
reg clk;

initial
  begin
    clk = 1'b0;
    forever #5 clk = ~clk;  // Clock period of 10 units
  end
```

17

17

Other Constructs Available

```
# (time_value)
```

- Makes a block suspend for “time_value” units of time.
- The time unit can be specified using the `timescale command.

```
@ (event_expression)
```

- Makes a block suspend until “event_expression” triggers.
- Various keywords associated with “event_expression” shall be discussed with examples..

18

18

- 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

19

19

Examples

20

```
// A combinational logic example
module  mux21 (in1, in0, s, f);
    input  in1, in0, s;
    output reg f;

    always @(in1 or in0 or s)
      if  (s)
        f = in1;
      else
        f = in0;
endmodule
```

- The event expression in the “always” block triggers whenever at least one of “in1”, “in0” or “s” changes.
- The “or” keyword specifies the condition.

Using “assign”

```
module mux2to1 (in, sel, out);
  input [1:0] in;
  input sel;
  output out;

  assign out = in[sel];
endmodule
```

21

21

```
// A combinational logic example
module  mux21 (in1, in0, s, f);
    input  in1, in0, s;
    output reg f;

    always @(in1, in0, s)
      if  (s)
        f = in1;
      else
        f = in0;
endmodule
```

- An alternate way to specify the event condition by using comma instead of “or”.
- Supported in later versions of Verilog.

22

22

```
// A combinational logic example
module  mux21 (in1, in0, s, f);
    input  in1, in0, s;
    output reg f;

    always @(*)
      if  (s)
        f = in1;
      else
        f = in0;
endmodule
```

- An alternate way to specify the event condition by using a "*" instead of naming the variables.
- "*" is activated whenever *any* of the variables change.

23

23

```
// A sequential logic example

module dff_negedge (D, clock, Q, Qbar);
    input  D, clock;
    output reg Q, Qbar;

    always  @(negedge clock)
      begin
        Q = D;
        Qbar = ~D;
      end
endmodule
```

- The keyword "negedge" means at the negative going edge of the specified signal.
- Similarly, we can use "posedge".
- We can combine various triggering conditions by separating them by commas or "or".

24

24

```
// Another sequential logic example

module incomp_state_spec (curr_state, flag);
    input  [0:1] curr_state;
    output reg [0:1] flag;

    always @(curr_state)
      case (curr_state)
         0,1 : flag = 2;
         3   : flag = 0;
      endcase
endmodule
```

The variable “flag” is not assigned a value in all the branches of the “case” statement.

- A latch (2-bit) will be generated for “flag”.

25

25

```
// A small modification

module incomp_state_spec (curr_state, flag);
    input  [0:1] curr_state;
    output reg [0:1] flag;

    always @(curr_state)
    begin
      flag = 0;
      case (curr_state)
         0,1 : flag = 2;
         3   : flag = 0;
      endcase
    end
endmodule
```

Here the variable “flag” is defined for all the possible values of “curr_state”.

- A pure combinational circuit will be generated.
- The latch is avoided.

26

26

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:
  a) ***Blocking*** (denoted by “=“)
  b) ***Non-blocking*** (denoted by “<=“)

27

27

- 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

28

28

```
module blocking_example;
  reg X, Y, Z;
  reg [31:0] A, B;    integer sum;

  initial
    begin
      X = 1'b0;  Y = 1'b0;  Z = 1'b1;      // At time = 0
      sum = 1;                              // At time = 0
      A = 31'b0;  B = 31'hababababab;       // At time = 0
      #5  A[5] = 1'b1;                      // At time = 5
      #10 B[31:29] = {X, Y, Z};             // At time = 15
      sum = sum + 5;                        // At time = 15
    end
endmodule
```

29

29

(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.

30

30

- 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.

31

31

```
always @(posedge clk)
  begin
    a <= b & c;
    b <= a ^ d;
    c <= a | b;
  end
```

Recommended style for modeling synchronous circuits, where assignments take place in synchronism with clock.

All assignments take place synchronously at the rising edge of the clock.

32

32

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.

33

33

Examples

34

```
// 8-to-1 multiplexer: behavioral description
module  mux_8to1 (in, sel, out);
  input [7:0] in;   input [2:0] sel;
  output reg out;
  always  @(*)
    begin
      case (sel)
        3'b000: out = in[0];
        3'b001: out = in[1];
        3'b010: out = in[2];
        3'b011: out = in[3];
        3'b100: out = in[4];
        3'b101: out = in[5];
        3'b110: out = in[6];
        3'b111: out = in[7];
        default: out = 1'bx;
      endcase
    end
endmodule
```

35

35

```
// Up-down counter (synchronous clear)
module  counter (mode, clr, ld, d_in, clk, count);
   input mode, clr, ld, clk;
   input [0:7] d_in;
   output reg [0:7]  count;

   always  @ (posedge  clk)
     if (ld)         count <= d_in;
     else if (clr)   count <= 0;
     else if (mode)  count <= count + 1;
     else            count <= count - 1;
endmodule
```

36

36

```
// A ring counter (Modified version 1)
module  ring_counter  (clk, init, count);
  input  clk, init;
  output reg [7:0]  count;
  always  @ (posedge clk)
    begin
       if (init)  count = 8'b10000000;
        else  begin
                  count    <= count << 1;
                  count[0] <= count[7];
               end
    end
endmodule
```

- Since non-blocking assignments are used, rotation will take place correctly.

37

37