CS 207 Digital Logic - Spring 2019
Lab 4 - Behavioral Modeling

Monday, Mar. 11, 2019

1 Experiment A

Behavioral models in Verilog contain procedural statements, which control the simulation and manipulate variables of the data types. These all statements are contained within the procedures. Each of the procedure has an activity flow associated with it.

During simulation of behavioral model, all the flows defined by the `always` and `initial` statements start together at simulation time 0. The `initial` statements are executed once, and the `always` statements are executed repetitively. For example, in the following model,

initial_example.v

```verilog
module initial_example();
reg clk,reset,enable,data;

initial begin
 clk = 0;
 reset = 0;
 enable = 0;
 data = 0;
end

endmodule
```

the `initial` block execution starts at time 0, which just executed all the statements within begin and end statement, without waiting.

always_example.v

```verilog
module always_example();
reg clk,reset,enable,q_in,data;

always @ (posedge clk)
if (reset)  begin
   data <= 0;
end else if (enable) begin
```

```
  data <= q_in;
end

endmodule
```

In an always block, when the trigger event occurs (here positive edge of clock), the code inside begin and end is executed; then once again the always block waits for next event triggering. This process of waiting and executing on event is repeated till simulation stops.

If a procedure block contains more than one statement, those statements must be enclosed within either a sequential `begin-end` block or a parallel `fork-join` block. Try the following code and see what's the difference:

`initial_begin_end.v`

```
module initial_begin_end();
reg clk,reset,enable,data;

initial begin
 $monitor(
   "#%g clk=%b reset=%b enable=%b data=%b",
   $time, clk, reset, enable, data);
 #1  clk = 0;
 #10 reset = 0;
 #5  enable = 0;
 #3  data = 0;
 #1 $finish;
end

endmodule
```

`initial_fork_join.v`

```
module initial_fork_join();
reg clk,reset,enable,data;

initial begin
 $monitor("#%g clk=%b reset=%b enable=%b data=%b",
   $time, clk, reset, enable, data);
 fork
   #1  clk = 0;
   #10 reset = 0;
   #5  enable = 0;
   #3  data = 0;
 join
 #1 $display ("#%g Terminating simulation", $time);
 $finish;
end

endmodule
```

To summarize, the `begin-end` keywords:

- Group several statements together.
- Cause the statements to be evaluated sequentially (one at a time)
  - Any timing within the sequential groups is relative to the previous statement.
  - Delays in the sequence accumulate (each delay is added to the previous delay)
  - Block finishes after the last statement in the block.

The `fork-join` keywords:

- Group several statements together.
- Cause the statements to be evaluated in parallel (all at the same time).
  - Timing within parallel group is absolute to the beginning of the group.
  - Block finishes after the last statement completes (Statement with highest delay, it can be the first statement in the block).

Assignment
Save the command line output in `initial_begin_end.log` and `initial_fork_join.log`, respectively. Assignment 1 requires the output files.

2 Experiment B

Verilog has two ways of value assignment, blocking and nonblocking. Blocking assignments are executed in the order they are coded, hence they are sequential. Since they block the execution of next statement, till the current statement is executed, they are called blocking assignments. Assignment are made with "`=`" symbol. Example: `a = b`.

Nonblocking assignments are executed in parallel. Since the execution of next statement is not blocked due to execution of current statement, they are called nonblocking statement. Assignments are made with "`<=`" symbol. Example `a <= b`.

`blocking_nonblocking.v`

```
module blocking_nonblocking();

reg a,b,c,d;
// Blocking Assignment
initial begin
  #10 a = 0;
  #11 a = 1;
  #12 a = 0;
  #13 a = 1;
end

initial begin
  #10 b <= 0;
  #11 b <= 1;
  #12 b <= 0;
  #13 b <= 1;
end

initial begin
   c = #10 0;
   c = #11 1;
   c = #12 0;
   c = #13 1;
end

initial begin
   d <= #10 0;
   d <= #11 1;
   d <= #12 0;
   d <= #13 1;
end

initial begin
  $monitor("#%g A = %b B = %b C = %b D = %b",$time, a, b, c,
   ↪  d);
  #50 $finish;
end

endmodule
```

Assignment
Save the command line output in `blocking_nonblocking.log`. Assignment 1 requires the output file.

3 Experiment C

3.1 The Conditional Statement `if-else`

The `if-else` statement controls the execution of other statements. In programming language like c, `if-else` controls the flow of program. When more than one statement needs to be executed for an `if` condition, we need to use `begin` and `end` as seen in earlier examples.

nested_if.v

```verilog
module nested_if();

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

always @ (posedge clk)
// If reset is asserted
if (reset == 1'b0) begin
   counter <= 4'b0000;
// If counter is enable and up count is asserted
end else if (enable == 1'b1 && up_en == 1'b1) begin
   counter <= counter + 1'b1;
// If counter is enable and down count is asserted
end else if (enable == 1'b1 && down_en == 1'b1) begin
   counter <= counter - 1'b1;
// If counting is disabled
end else begin
   counter <= counter; // Redundant code
end

// Testbench code
initial begin
  $monitor ("#%g reset=%b enable=%b up=%b down=%b count=%b",
             $time, reset, enable, up_en, down_en,counter);
  $display("#%g Driving all inputs to know state",$time);
  clk = 0;
  reset = 0;
  enable = 0;
  up_en = 0;
  down_en = 0;
  #3 reset = 1;
  $display("#%g De-Asserting reset",$time);
  #4 enable = 1;
  $display("#%g De-Asserting reset",$time);
  #4 up_en = 1;
  $display("#%g Putting counter in up count mode",$time);
  #10 up_en = 0;
  down_en = 1;
  $display("#%g Putting counter in down count mode",$time);
  #8 $finish;
end

always #1 clk = ~clk;

endmodule
```

Assignment
Save the command line output in `nested_if.log`. Assignment 1 requires the output file.

3.2 The Case Statement

The `case` statement compares an expression to a series of cases and executes the statement or statement group associated with the first matching case. Case statement supports single or multiple statements, and group multiple statements using begin and end keywords.

mux.v

```
module mux (a,b,c,d,sel,y);
input a, b, c, d;
input [1:0] sel;
output y;

reg y;

always @ (a or b or c or d or sel)
case (sel)
  0 : y = a;
  1 : y = b;
  2 : y = c;
  3 : y = d;
  2'bxx,2'bzz : $display("Error in SEL");
  default : $display("Error in SEL");
endcase

endmodule
```

Special versions of the case statement allow the x ad z logic values to be used as “don’t care”:

- `casez`: Treats z as don’t care.
- `casex`: Treats x and z as don’t care.

casez_example.v

```
module casez_example();
reg [3:0] opcode;
reg [1:0] a,b,c;
reg [1:0] out;

always @ (opcode or a or b or c)
casez(opcode)
  4'b1zzx : begin // Don't care about lower 2:1 bit, bit 0
    ↪ match with x
```

```
              out = a;
              $display("#%g 4'b1zzx is selected, opcode %b",
                 ↪ $time,opcode);
            end
  4'b01?? : begin
              out = b; // bit 1:0 is don't care
              $display("#%g 4'b01?? is selected, opcode %b",
                 ↪ $time,opcode);
            end
  4'b001? : begin  // bit 0 is don't care
              out = c;
              $display("#%g 4'b001? is selected, opcode %b",
                 ↪ $time,opcode);
            end
  default : begin
              $display("#%g default is selected, opcode %b",
                 ↪ $time,opcode);
            end
endcase

// Testbench code goes here
always #2 a = $random;
always #2 b = $random;
always #2 c = $random;

initial begin
  opcode = 0;
  #2 opcode = 4'b101x;
  #2 opcode = 4'b0101;
  #2 opcode = 4'b0010;
  #2 opcode = 4'b0000;
  #2 $finish;
end

endmodule
```

Assignment
Save the command line output in `casez_example.log`. Assignment 1 requires the output file.

The following example shows the difference among `case`, `casex`, and `casez`:

`case_compare.v`

```
module case_compare;

reg sel;

initial begin
```

```
  #1 $display ("\n     Driving 0");
  sel = 0;
  #1 $display ("\n     Driving 1");
  sel = 1;
  #1 $display ("\n     Driving x");
  sel = 1'bx;
  #1 $display ("\n     Driving z");
  sel = 1'bz;
  #1 $finish;
end

always @ (sel)
case (sel)
  1'b0 : $display("Normal : Logic 0 on sel");
  1'b1 : $display("Normal : Logic 1 on sel");
  1'bx : $display("Normal : Logic x on sel");
  1'bz : $display("Normal : Logic z on sel");
endcase

always @ (sel)
casex (sel)
  1'b0 : $display("CASEX  : Logic 0 on sel");
  1'b1 : $display("CASEX  : Logic 1 on sel");
  1'bx : $display("CASEX  : Logic x on sel");
  1'bz : $display("CASEX  : Logic z on sel");
endcase

always @ (sel)
casez (sel)
  1'b0 : $display("CASEZ  : Logic 0 on sel");
  1'b1 : $display("CASEZ  : Logic 1 on sel");
  1'bx : $display("CASEZ  : Logic x on sel");
  1'bz : $display("CASEZ  : Logic z on sel");
endcase

endmodule
```

4 Experiment D

In Verilog, looping statements appear inside procedural blocks only. It has four looping statements, i.e., `forever`, `repeat`, `while`, `for`.

4.1 The `forever` Statement

The `forever` loop executes continually, the loop never ends. Normally we use forever statements in initial blocks.

One should be very careful in using a forever statement: if no timing construct is present in the forever statement, simulation could hang. The code below is one such application, where a timing construct is included inside a forever statement.

forever_example.v

```
module forever_example ();

reg clk;

initial begin
  #1 clk = 0;
  forever begin
    #5 clk = !clk;
  end
end

initial begin
  $monitor ("#%g  clk = %b",$time, clk);
  #100 $finish;
end

endmodule
```

4.2 The `repeat` Statement

The `repeat` loop executes statements for a fixed number of times.

repeat_example.v

```
module repeat_example();
reg  [3:0] opcode;
reg  [15:0] data;
reg         temp;

always @ (opcode or data)
begin
  if (opcode == 10) begin
    // Perform rotate
    repeat (8) begin
      #1 temp = data[15];
      data = data << 1;
      data[0] = temp;
    end
  end
end
// Simple test code
initial begin
   $display (" TEMP  DATA");
   $monitor (" %b     %b ",temp, data);
   #1 data = 18'hF0;
   #1 opcode = 10;
```

```
    #10 opcode = 0;
    #1 $finish;
end

endmodule
```

4.3 The `while` loop Statement

The `while` loop executes as long as an expression evaluates as true. This is the same as in any other programming language.

while_example.v

```
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
```

4.4 The `for` loop Statement

The `for` loop is the same as in any other programming language:

- Executes an initial assignment once at the start of the loop.

- Executes the loop as long as an expression evaluates as true.
- Executes a step assignment at the end of each pass through the loop.

for_example.v

```
module for_example();

integer i;
reg [7:0] ram [0:255];

initial begin
  for (i = 0; i < 256; i = i + 1) begin
    #1 $display(" Address = %g  Data = %h",i,ram[i]);
    ram[i] <= 0; // Initialize the RAM with 0
    #1 $display(" Address = %g  Data = %h",i,ram[i]);
  end
  #1 $finish;
end

endmodule
```