IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

NPTEL

Lecture 12: VERILOG DESCRIPTION STYLES

PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Description Styles in Verilog

- Two different styles of description:
  1. Data Flow
     - Continuous assignment

     Using assignment statements.
  2. Behavioral
     - Procedural assignment
       - Blocking
       - Non-blocking

     Using procedural statements similar to a program in high-level language.

Data Flow Style: Continuous Assignment

```
assign a = b + c;
assign sign = Z[15];
```

- Identified by the keyword "assign".
- Forms a static binding between:
  - The "net" being assigned on the left-hand side (LHS).
  - The expression on the right-hand side (RHS), which may consist of both "net" and "register" type variables.
- The assignment is continuously active:
  - Almost exclusively used to model combinational circuits.
  - We shall also see some examples of modeling sequential circuit elements.

- Some points to note:
  - A Verilog module can contain any number of "assign" statements.
  - Typically, the "assign" statements are followed by procedural descriptions.
  - The "assign" statements are used to model behavioral descriptions.
- We shall illustrate various usages of "assign" statements for modeling combinational and also some sequential logic blocks.

```
module   generate_MUX (data, select, out);
    input [15:0]  data;
    input [3:0]  select;
    output  out;
    assign  out = data[select];
endmodule
```

Non-constant index in expression on RHS generates a MUX

- Point to note:
  - Whenever there is an array reference on the RHS with a variable index, a MUX is generated by the synthesis tool.
  - If the index is a constant, just a wire will be generated.

    Example: `assign out = data[2];`

```
module  generate_set_of_MUX (a, b, f, sel);
    input [0:3] a, b;
    input sel;
    output [0:3] f;
    assign f = sel ? a : b;
endmodule
```

Conditional operator generates a MUX

- Point to note:
  - Whenever a conditional is encountered in the RHS of an expression, a 2-to-1 MUX is generated.
  - In the previous example, since the variables "a", "b" and "f" are vectors, an array of 2-to-1 MUX-es are generated.
  - What hardware will be generated by the following?

```
assign f = (a==0) ? (c+d) : (c–d);
```

```
module generate_decoder (out, in, select);
    input  in;
    input [0:1]  select;
    output [0:3]  out;
    assign  out[select] = in;
endmodule
```

Non-constant index in expression on LHS generates a decoder

- Point to note:
  - A constant index in the expression on the LHS will not generate a decoder.
  - Example: `assign out[5] = in;`

    This will simply generate a wire connection.
  - As a rule of thumb, whenever the synthesis tool detects a variable index in the LHS, a decoder is generated.

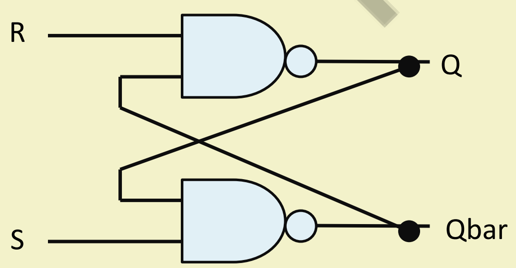
```
module level_sensitive_latch (D, Q, En);
    input  D, En;
    output Q;
    assign Q = En ? D : Q;
endmodule
```

En	D	Q_n
0	x	Q_{n-1}
1	0	0
1	1	1

Generates a D-type latch

Here is an example to describe a sequential logic element using "assign" statement.

- Modeling a simple S-R latch:



S	R	Q_n
1	1	Q_{n-1}
0	1	0
1	0	1
0	0	?

```
module sr_latch (Q, Q
  input S, R;
  output Q, Qbar;

  assign Q = ~(R & Qb
  assign Qbar = ~(S &
endmodule
```

```
module latchtest;
  reg S, R;   wire Q, Qbar;
  sr_latch LAT (Q, Qbar, S, R);
  initial
    begin
      $monitor ($time, "S=%b R=%b, Q=%b, Qbar=%b",
                       S, R, Q, Qbar);
         S = 1'b0;  R = 1'b1;
      #5 S = 1'b1;  R = 1'b1;
      #5 S = 1'b1;  R = 1'b0;
      #5 S = 1'b1;  R = 1'b1;
      #5 S = 1'b0;  R = 1'b0;
      #5 S = 1'b1;  R = 1'b1;
    end
endmodule
```


Simulation Output

```
 0  S=0, R=1, Q=0, Qbar=1
 5  S=1, R=1, Q=0, Qbar=1
10  S=1, R=0, Q=1, Qbar=0
15  S=1, R=1, Q=1, Qbar=0
20  S=0, R=0, Q=1, Qbar=1
and then the simulator hangs
```

END OF LECTURE 12

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog | 15

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Lecture 13: PROCEDURAL ASSIGNMENT

PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

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.

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.

```
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.

Some Short Cuts in Declarations

- "output" and "reg" can be declared together in the same statement.
  `output reg [7:0] data;`
  instead of `output [7:0] data;   reg [7:0] data;`
- A variable can be initialized when it is declared:
  `reg clock = 0;`
  instead of `reg clock;   initial clock = 0;`

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.


```
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.

- 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
```


- Only "reg" type variable can be assigned within an "initial" or 'always" block.
- Basic reason:
  - The sequential "always" block executes only when the event expression triggers.
  - At other times the block is doing nothing.
  - An object being assigned to must therefore remember the last value assigned (not continuously driven).
  - So, only "reg" type variables can be assigned within the "always" block.
  - Of course, any kind of variable may appear in the event expression (reg, wire, etc.).

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.

(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.

(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”.

(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.

- 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
```


END OF LECTURE 13

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

NPTEL

Lecture 14: PROCEDURAL ASSIGNMENT (CONTD.)

PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

(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
```

(e) “for” loop

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

- 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”.

- 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.

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;
```

(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".

- 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".

Example:

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

Exactly 100 clock pulses are generated.

(g) "forever" loop

```
forever
  sequential_statement;
```

- 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.

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


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

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

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..

@ (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:
  a) Change of a signal value.
  b) Positive or negative edge occurring on signal (*posedge* or *negedge*).
  c) 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


```
// D flip-flop with synchronous set and reset
module dff (q, qbar, d, set, reset, clk);
  input  d, set, reset, clk;
  output reg  q;     output qbar;
  assign  qbar = ~q;
  always @ (posedge clk)
    begin
      if (reset == 0)  q <= 0;
      else if (set == 0)  q <= 1;
      else q <= d;
    end
endmodule
```

```
// D flip-flop with asynchronous set and reset

module dff (q, qbar, d, set, reset, clk);
  input  d, set, reset, clk;
  output reg  q;    output qbar;

  assign  qbar = ~q;

  always @ (posedge clk or negedge set or negedge reset)
    begin
      if (reset == 0)  q <= 0;
      else if (set == 0)  q <= 1;
      else q <= d;
    end
endmodule
```



```
// Transparent latch with enable

module latch (q, qbar, din, enable);
  input  din, enable;
  output reg  q;    output  qbar;

  assign  qbar = ~q;

  always @ (din or enable)
    begin
      if (enable)  q = din;
    end
endmodule
```

END OF LECTURE 14

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Lecture 15: PROCEDURAL ASSIGNMENT (EXAMPLES)

PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

```
// 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.


```
// 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.

```
// 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.


```
// 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".

```
// 4-bit counter with asynchronous
reset
module counter (clk, rst, count);

  input clk, rst;
  output reg [3:0] count;

  always @(posedge clk or posedge rst)
    begin
      if (rst)
        count <= 0;
      else
        count <= count + 1;
    end
endmodule
```

The event condition triggers when either a positive edge of “clk” comes, or a positive edge of “rst”.


```
// 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”.

```
// 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.

- When a "case" statement is incompletely decoded, the synthesis tool will infer the need for a latch to hold the residual output when the select bits take the unspecified values.
  – It is up to the designer to code the design in such a way that latch can be avoided where possible.

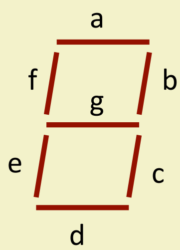
```
// A simple 4-function ALU
module ALU_4bit (f, a, b, op);
  input [1:0] op;     input [7:0] a, b;
  output reg [7:0] f;
  parameter  ADD=2'b00, SUB=2'b01, MUL=2'b10, DIV=2'b11;

  always  @(*)
      case (op)
        ADD : f = a + b;
        SUB : f = a – b;
        MUL : f = a * b;
        DIV : f = a / b;
      endcase
endmodule
```



```
module  priority_encoder  (in, code);
  input [7:0] in;
  output reg [2:0] code;
  always  @(in)
    begin
             if (in[0]) code = 3'b000;
        else if (in[1]) code = 3'b001;
        else if (in[2]) code = 3'b010;
        else if (in[3]) code = 3'b011;
        else if (in[4]) code = 3'b100;
        else if (in[5]) code = 3'b101;
        else if (in[6]) code = 3'b110;
        else if (in[7]) code = 3'b111;
        else            code = 3'bxxx;
    end
endmodule
```

- The inputs bits are checked sequentially one by one (in order of priority).
  - "in[0]" has the highest priority.
  - For simultaneously active inputs, the first active input encountered will be encoded.

```
module bcd_to_7seg (bcd, seg);
  input [3:0] bcd;
  output reg [6:0] seg;
  always @(bcd)
    case
              0: seg = 6'b0000001;
              1: seg = 6'b1001111;
              2: seg = 6'b0010010;
              3: seg = 6'b0000110;
              4: seg = 6'b1001100;
              5: seg = 6'b0100100;
              6: seg = 6'b0100000;
              7: seg = 6'b0001111;
              8: seg = 6'b0000000;
              9: seg = 6'b0000100;
      default : seg = 6'b1111111;
    endcase
endmodule
```



Segment bit assignment:
(a, b, c, d, e, f, g)

A segment glows when the corresponding bit of seg is 0.


```
// An n-bit comparator
module compare (A, B, lt, gt, eq);
  parameter word_size = 16;
  input [word_size-1:0]  A, B;
  output reg lt, gt, eq;

  always @ (*)
    begin
      gt = 0;  lt = 0;  eq = 0;
      if (A > B)  gt = 1;
      else if (A < B)  lt = 1;
      else  eq = 1
    end
endmodule
```

For actual synthesis, it is common to have a structured design representation of the comparator.

```
// A 2-bit comparator
module compare (A1, A0, B1, B0, lt, gt, eq);
  input  A1, A0, B1, B0;
  output reg  lt, gt, eq;

  always @ (A1, A0, B1, B0)
    begin
      lt = ({A1,A0) < {B1,B0});
      gt = ({A1,A0) > {B1,B0});
      eq = ({A1,A0) == {B1,B0});
    end
endmodule
```



```
module alu_example (alu_out, A, B, operation, en);
  input [2:0] operation;  input [7:0] A, B;
  input en;
  output [7:0] alu_out;   reg [7:0] alu_reg;

  assign alu_out = (en == 1) ? alu_reg : 4'bz;

  always @ (*)
    case (operation)
      3'b000 : alu_reg = A + B;
      3'b001 : alu_reg = A – B;
      3'b011 : alu_reg = ~ A;
      default : alu_reg = 4'b0;
    endcase
endmodule
```

END OF LECTURE 15