INTRODUCTION TO VERILOG (Contd..)

EC39004 : VLSI LABORATORY

21 January 2019

Verilog Description Styles

Two different styles of description :

- ❑ Data Flow
  - Continuous assignment — *Using assignment statements.*
- ❑ Behavioral
  - Procedural assignment — *Using procedural statements similar to a program in high-level language.*
    - ✓ Blocking
    - ✓ Non-blocking

Data Flow Assignment: Continuous Assignment

- Identified by the keyword "**assign**".

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

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

Data Flow Assignment: Continuous Assignment

Some points to note:

1. A Verilog module can contain any number of "assign" statements.
2. Typically, the "assign" statements are followed by procedural descriptions.
3. The "assign" statements are used to model behavioral descriptions.

- **Generic Form:**

module….
signal declarations;
assign statements;
Procedural descriptions;
endmodule

Code Examples

- Non-constant index on RHS of an expression generates a multiplexer logic → whenever there is an array reference on the RHS with a variable index, a MUX is generated by the synthesis tool

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

- If the index is a constant, just a wire will be generated eg. **assign out = data[2];**

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

Code Examples

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

- Point to note:
  1. Whenever a conditional operator is encountered in the RHS of an expression, a 2- to-1 MUX is generated.

  1. In the previous example, since the variables “a”, “b” and “f” are vectors, an array of 2-to-1 MUX-es are generated.

Code Examples

- As a rule of thumb, whenever the synthesis tool detects a variable index in the LHS, a decoder is generated.

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

  - A constant index in the expression on the LHS will not generate a decoder.
  - Example:

assign out[5] = in;

```
assign out[5] =in
```

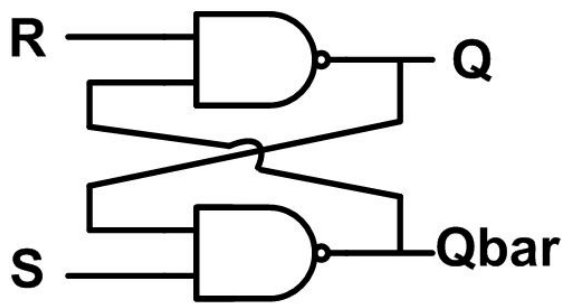
This will simply generate a wire connection.

Code Examples

```
module level_sensitive_latch (d, q, en);
input d, en;
output q;
assign q =  en ?d : q;
endmodule
```

- This is the description of a sequential logic element using "assign" statement is written so that previous value is memorized
- D-type latch is generated

Code Examples



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

Modeling a simple S-R Latch

```
module sr_latch (q, qbar, s, r);
input s, r;
output q, qbar;
assign q = ~(r & qbar); assign
qbar = ~(s & q); endmodule
```

S-R Latch (Testbench)

```
module srlatchtest;
        reg s, r; / / Inputs
        wire q, qbar; / / Outputs
 / /   Instantiate the Unit Under Test (UUT)  srlatch uut
    (.s(s), .r(r), .q(q), .qbar(qbar));  initial begin
/ /  Initialize Inputs
            $monitor("Time=%d, s=%b,r=%b.q=%b,qbar=%b", $time, s, r, q, qbar);
            s =  1'b0; r = 1'b1;
            #100; s =  1'b1; r =  1'b1; #100; s =  1'b1; r =  1'b0;  #100; s =  1'b1; r =  1'b1;
            #100; s =  1'b0; r =  1'b0;  #100; s =  1'b1; r =  1'b1; #100; s =  1'b1; r =  1'b0;
            #100; s =  1'b1; r = 1'b1;
            end  endmodule
```

SR Latch (Behavioral Simulation o/p)

Time= 0, s=0, r=1, q=0, qbar=1

Time= 100, s=1, r=1, q=0, qbar=1

Time= 200, s=1, r=0, q=1, qbar=0

Time= 300, s=1, r=1, q=1, qbar=0

Time= 400, s=0, r=0, q=1, qbar=1

Time= 500, s=1, r=1, q=1, qbar=0

Time= 600, s=1, r=0, q=1, qbar=0

Time= 700, s=1, r=1, q=1, qbar=0

→ The simulator hangs after Time=400

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.

The "initial" block

```
initial
   begin
   ……..
   ……..
   end
```

The 3 initial blocks execute concurrently:

- The 1st block executes at time 0
- The 3rd block terminates simulation at 25 time units

Example:

```
module testbench_example;
reg a,b,cin;
wire 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 “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.

"Initial" and "Always" block

Example:

```
initial
clk=1'b0; //initialized to 0 at time 0
always
#5  clk = ~clk;
initial
#500  $finish;
```

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

The “always” block Syntax

Syntax:

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

- 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.

The “always” block

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 or 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:

i. blocking (a = b + c);

ii. non-blocking (a <= b + c);

- The sequential statements are explained next.

begin … end

```
  begin…end
begin
        sequential_statement1;
        sequential_statement2;
        ……….
        ……….....
        sequential statement n;
  end
```

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

Syntax: 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”.

Example 1: If … else

```
module multiplex21 (in1, in0, s, f);
input in1, in0,s;
output reg f;
always @ (in1, in0,s)
    if    (s)
     f = in1;
     else
     f =  in0;
      endmodule
```

Example 2: If … else

```
module multiplex21 (in1, in0, s, f);
input in1, in0, s;
output reg f;
always @ (in1 or in0 or s)
// always @ (in1, in0, s)
if (s)
    f = in1;
else
    f = in0;
    endmodule
```

Example 3

```
module multiplex21 (in1, in0, s, f);
input in1, in0, s;
output reg f;
always @ (in1 or in0 or s)
f=(in1 & s)|(in0 & ~s);
endmodule
```

Example 4

```
module mux2to1 (s, a, b, y);
input s, a,b;
output y;
reg y, na,nb;
always@ (*)
begin
nb =  b & s;
na =  a &(~s);
y =  na |    nb;
end
endmodule
```

Assignment: 5*4unsigned Multiplier (Braun Array)