



EECS 598 VLSI for Wireless Comm. and Machine Learning

Hardware Description Language (HDL): Verilog

Prof. Hun-Seok Kim

hunseok@umich.edu

Verilog tutorial from EECS 470 Lab. Credit: Prof. Dreslinski, GSI Versteeg, GSI Zekany

What is HDL, Verilog?

What is Verilog?

- ▶ Hardware Description Language - IEEE 1364-2005
 - ▶ Superseded by SystemVerilog - IEEE 1800-2009
- ▶ Two Forms
 1. Behavioral
 2. Structural
- ▶ It can be built into hardware. If you can't think of at least one (inefficient) way to build it, it might not be good.

Why do I care?

- ▶ We use Behavioral Verilog to do computer architecture here.
- ▶ Semiconductor Industry Standard (VHDL is also common, more so in Europe)

Behavioral vs. Structural

Behavioral Verilog

- ▶ Describes function of design
- ▶ Abstractions
 - ▶ Arithmetic operations (+, -, *, /)
 - ▶ Logical operations (&, |, ^, ~)

Structural Verilog

- ▶ Describes construction of design
- ▶ No abstraction
- ▶ Uses modules, corresponding to physical devices, for everything

Suppose we want to build an adder?

Structural Verilog

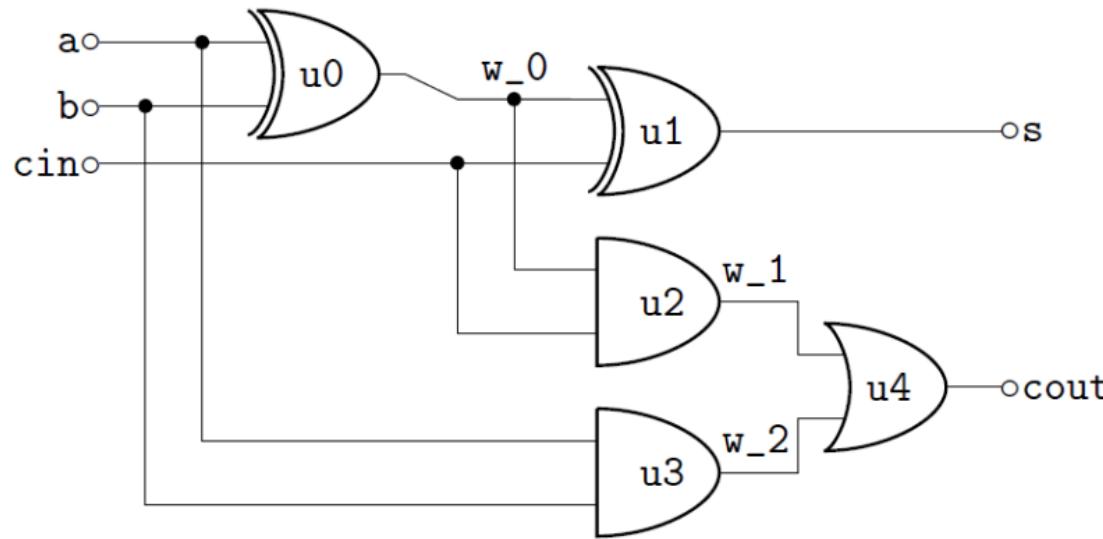


Figure: 1-bit Full Adder

```
module one_bit_adder(
    input wire a,b,cin,
    output wire sum,cout);
    wire w_0,w_1,w_2;
    xor u0(w_0,a,b);
    xor u1(sum,w_0,(cin));
    and u2(w_1,w_0,(cin));
    and u3(w_2,a,b);
    or u4(cout,w_1,w_2);
endmodule
```

Behavioral Verilog

```
module one_bit_adder(
    input wire a,b,cin,
    output wire sum,cout);
    assign sum = a ^ b ^ cin;
    assign cout = ((a ^ b) & cin) | a & b;
endmodule
```

OR ...

```
module one_bit_adder(
    input logic a,b,cin,
    output logic sum,cout);

    always_comb
    begin
        sum = a ^ b ^ cin;
        cout = 1'b0;
        if ((a ^ b) & cin) | (a & b))
            cout = 1'b1;
    end
endmodule
```



Verilog Semantics

Lexical

- ▶ Everything is case sensitive.
- ▶ Type instances must start with A-Z, a-z, _. They may contain A-Z, a-z, 0-9, _, \$.
- ▶ Comments begin with // or are enclosed with /* and */.

Data Types, Values

Synthesizable Data Types

wires Also called nets

```
wire a_wire;  
wire [3:0] another_4bit_wire;
```

- ▶ Cannot hold state

logic Replaced reg in SystemVerilog

```
logic [7:0] an_8bit_register;  
reg a_register;
```

- ▶ Holds state, might turn into flip-flops
- ▶ Less confusing than using reg with combinational logic
(coming up...)



Data Types, Values

Unsynthesizable Data Types

`integer` Signed 32-bit variable

`time` Unsigned 64-bit variable

`real` Double-precision floating point variable

Four State Logic

`0` False, low

`1` True, high

`Z` High-impedance, unconnected net

`X` Unknown, invalid, don't care

Data Values

Literals/Constants

- ▶ Written in the format <bitwidth>'<base><constant>
- ▶ Options for <base> are...
 - b Binary
 - o Octal
 - d Decimal
 - h Hexadecimal

```
assign an_8bit_register = 8'b10101111;
assign a_32bit_wire = 32'hABCD_EF01;
assign a_4bit_logic = 4'hE;
```

Operators

Arithmetic	
*	Multiplication
/	Division
+	Addition
-	Subtraction
%	Modulus
**	Exponentiation
Bitwise	
~	Complement
&	And
	Or
~	Nor
^	Xor
~~	Xnor
Logical	
!	Complement
&&	And
	Or
Shift	
>>	Logical right shift
<<	Logical left shift
>>>	Arithmetic right shift
<<<	Arithmetic left shift
Relational	
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
!=	Inequality
!==	4-state inequality
==	Equality
====	4-state equality
Special	
{,}	Concatenation
{n{m}}	Replication
:?	Ternary

Setting Values

assign Statements

- ▶ One line descriptions of combinational logic
- ▶ Left hand side must be a wire (SystemVerilog allows assign statements on logic type)
- ▶ Right hand side can be any one line verilog expression
- ▶ Including (possibly nested) ternary (?:)

Example

```
module one_bit_adder(
    input wire a,b,cin,
    output wire sum,cout);
    assign sum = a ^ b ^ cin;
    assign cout = ((a ^ b) & cin) | a & b;
endmodule
```

Setting Values

always Blocks

- ▶ Contents of `always` blocks are executed whenever anything in the sensitivity list happens
- ▶ Two main types in this class...
 - ▶ `always_comb`
 - ▶ implied sensitivity lists of every signal inside the block
 - ▶ Used for combinational logic. Replaced `always @*`
 - ▶ `always_ff @ (posedge clk)`
 - ▶ sensitivity list containing only the positive transition of the `clk` signal
 - ▶ Used for sequential logic
- ▶ All left hand side signals need to be logic type.

Examples

Combinational Block

```
always_comb
begin
    x = a + b;
    y = x + 8'h5;
end
```

Sequential Block

```
always_ff @ (posedge clk)
begin
    x <= #1 next_x;
    y <= #1 next_y;
end
```

Blocking vs. Non-blocking statement

Blocking Assignment

- ▶ Combinational Blocks
- ▶ Each assignment is processed in order, earlier assignments block later ones
- ▶ Uses the = operator

vs.

Nonblocking Assignment

- ▶ Sequential Blocks
- ▶ All assignments occur “simultaneously,” delays are necessary for accurate simulation
- ▶ Uses the \leq operator

Blocking Example

```
always_comb
begin
    x = new_val1;
    y = new_val2;
    sum = x + y;
end
```

- ▶ Behave exactly as expected
- ▶ Standard combinational logic

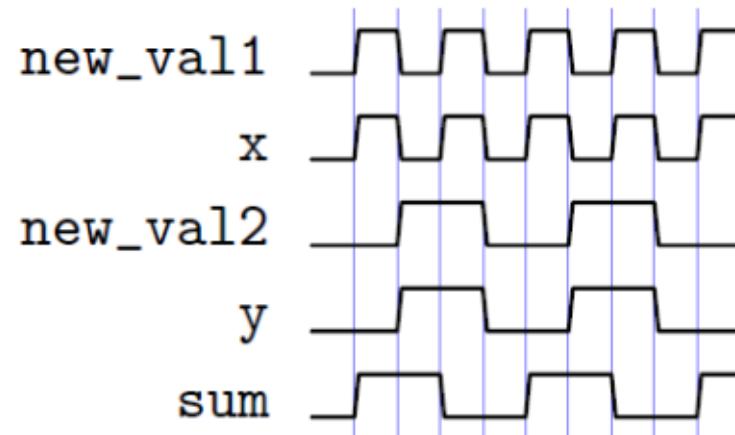


Figure: Timing diagram for the above example.

Nonblocking Example

```
always_ff @ (posedge clock)
begin
    x <= #1 new_val1;
    y <= #1 new_val2;
    sum <= #1 x + y;
end
```

- ▶ What changes between these two examples?
- ▶ Nonblocking means that `sum` lags a cycle behind the other two signals

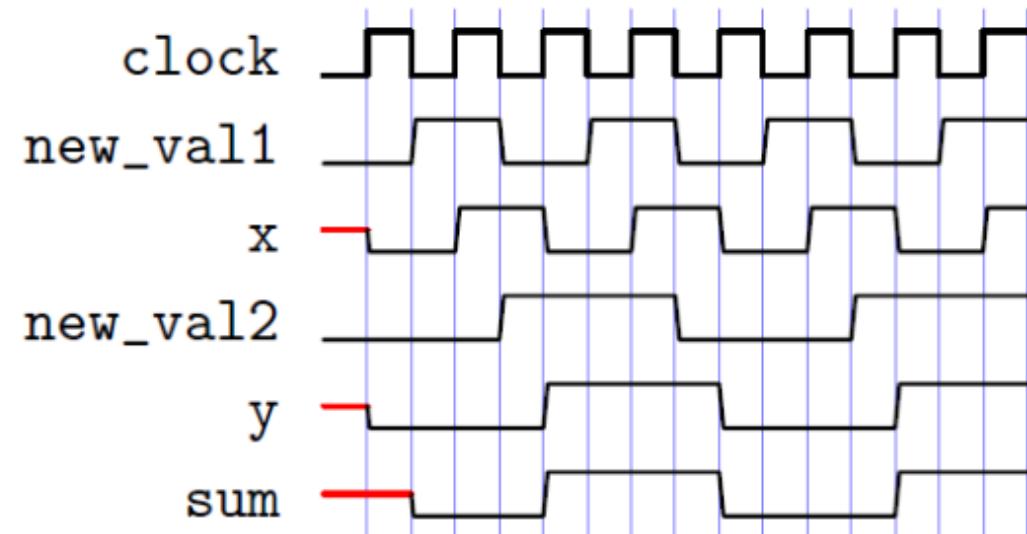


Figure: Timing diagram for the above example.

Bad Example

```
always_ff @(posedge clock)
begin
    x <= #1 y;
    z = x;
end
```

- ▶ z is updated after x
- ▶ z updates on negedge clock

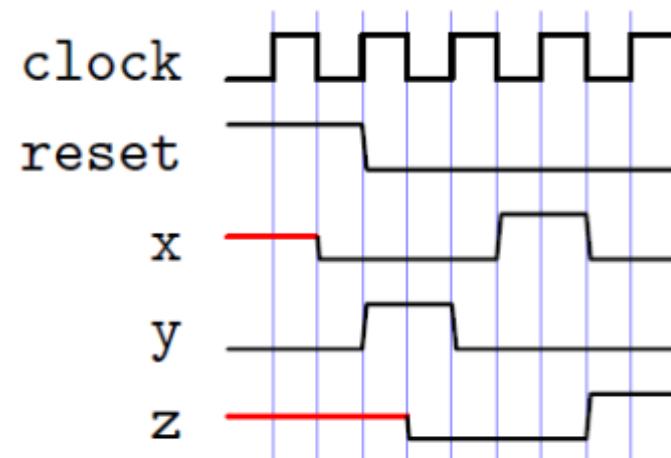


Figure: Timing diagram for the above example.

Recommended tips: avoid latches

Latches

- ▶ What is a latch?
 - ▶ Memory device without a clock
- ▶ Generated by a synthesis tool when a net needs to hold state without being clocked (combinational logic)
- ▶ Generally bad, unless designed in intentionally
- ▶ Unnecessary in this class

Latches

- ▶ Always assign every variable on every path
- ▶ This code generates a latch
- ▶ Why does this happen?

```
always_comb
begin
    if (cond)
        next_x = y;
end
```

To avoid unintentional latches

Possible Solutions to Latches

```
always_comb
begin
    next_x = x;
    if (cond)
        next_x = y;
end
```

```
always_comb
begin
    if (cond)
        next_x = y;
    else
        next_x = x;
end
```

Modules

Intro to Modules

- ▶ Basic organizational unit in Verilog
- ▶ Can be reused

Module Example

```
module my_simple_mux(
    input wire select_in, a_in, b_in; //inputs listed
    output wire muxed_out); //outputs listed
    assign muxed_out = select_in ? b_in : a_in;
endmodule
```

Writing Modules

- ▶ Inputs and outputs must be listed, including size and type format: <dir> <type> <[WIDTH-1:0]> <name>;
e.g. output logic [31:0] addr;
- ▶ In module declaration line or after it, inside the module

Instantiating Modules

- ▶ Two methods of instantiation
 1. e.g. my_simple_mux m1(.a_in(a), .b_in(b),
.select_in(s), .muxed_out(m));
 2. e.g. my_simple_mux m1(a, b, s, m);
- ▶ The former is much safer...
- ▶ Introspection (in testbenches): module.submodule.signal

Keys to Synthesizability

- ▶ Remember – Behavioral Verilog implies no specific hardware design
- ▶ But, it has to be synthesizable
- ▶ Better be able to build it somehow

Combinational Logic

- ▶ Avoid feedback (combinatorial loops)
- ▶ Always blocks should
 - ▶ Be `always_comb` blocks
 - ▶ Use the blocking assignment operator =
- ▶ All variables assigned on all paths
 - ▶ Default values
 - ▶ `if(...)` paired with an `else`

Sequential Logic

Sequential Logic

- ▶ Avoid clock- and reset-gating
- ▶ Always blocks should
 - ▶ Be `always_ff @(posedge clock)` blocks
 - ▶ Use the nonblocking assignment operator, with a delay $\leq \#1$
- ▶ No path should set a variable more than once
- ▶ Reset all variables used in the block
- ▶ `//synopsys sync_set_reset "reset"`

Flow Control

All Flow Control

- ▶ Can only be used inside procedural blocks (`always`, `initial`, `task`, `function`)
- ▶ Encapsulate multiline assignments with `begin...end`
- ▶ Remember to assign on all paths

Synthesizable Flow Control

- ▶ `if/else`
- ▶ `case`

Unsynthesizable Flow Control

- ▶ Useful in testbenches
- ▶ For example...
 - ▶ `for`
 - ▶ `while`
 - ▶ `repeat`
 - ▶ `forever`

Synthesizable Flow Control Example

```
always_comb
begin
    if (muxy == 1'b0)
        y = a;
    else
        y = b;
end
```

The Ternary Alternative

```
wire y;
assign y = muxy ? b : a;
```

Casez Example

```
always_comb
begin
    casez(alu_op)
        3'b000: r = a + b;
        3'b001: r = a - b;
        3'b010: r = a * b;
        ...
        3'b1???: r = a ^ b;
    endcase
end
```



Testing

What is a test bench?

- ▶ Provides inputs to one or more modules
- ▶ Checks that corresponding output makes sense
- ▶ Basic building block of Verilog testing

Why do I care?

- ▶ Finding bugs in a single module is hard...
- ▶ But not as hard as finding bugs after combining many modules
- ▶ Better test benches tend to result in higher project scores

Intro to Testbench

Features of the Test Bench

- ▶ Unsynthesized
 - ▶ Remember unsynthesizable constructs? This is where they're used.
 - ▶ In particular, unsynthesizable flow control is useful in testbenches (e.g. `for`, `while`)
- ▶ Programmatic
 - ▶ Many programmatic, rather than hardware design, features are available e.g. functions, tasks, classes (in SystemVerilog)

Testbench

A good test bench should, in order...

1. Declare inputs and outputs for the module(s) being tested
2. Instantiate the module (possibly under the name DUT for Device Under Test)
3. Setup a clock driver (if necessary)
4. Setup a correctness checking function (if necessary/possible)
5. Inside an initial block...
 - 5.1 Assign default values to all inputs, including asserting any available reset signal
 - 5.2 \$monitor or \$display important signals
 - 5.3 Describe changes in input, using good testing practice



Unsynthesizable Procedural Blocks

initial Blocks

- ▶ Procedural blocks, just like `always`
- ▶ Contents are simulated once at the beginning of a simulation
- ▶ Used to set values inside a test bench
- ▶ Should only be used in test benches

initial Block Example

```
initial
begin
    @(negedge clock);
    reset = 1'b1;
    in0 = 1'b0;
    in1 = 1'b1;
    @(negedge clock);
    reset = 1'b0;
    @(negedge clock);
    in0 = 1'b1;
    ...
end
```

Testbench tasks and functions

task

- ▶ Reuse commonly repeated code
- ▶ Can have delays (e.g. #5)
- ▶ Can have timing information (e.g. @(negedge clock))
- ▶ Might be synthesizable (difficult, not recommended)

function

- ▶ Reuse commonly repeated code
- ▶ No delays, no timing
- ▶ Can return values, unlike a task
- ▶ Basically combinational logic

task Example

```
task exit_on_error;
    input [63:0] A, B, SUM;
    input C_IN, C_OUT;
    begin
$display("@@@ Incorrect at time %4.0f", $time);
$display("@@@ Time:%4.0f clock:%b A:%h B:%h CIN:%b SUM:%h"
        "COUT:%b", $time, clock, A, B, C_IN, SUM, C_OUT);
$display("@@@ expected sum=%b", (A+B+C_IN) );
$finish;
    end
endtask
```

function Example

```
function check_addition;
    input wire [31:0] a, b;
    begin
        check_addition = a + b;
    end
endfunction

assign c = check_addition(a,b);
```



System tasks and functions

- ▶ Just like regular tasks and functions
- ▶ But they introspect the simulation
- ▶ Mostly these are used to print information
- ▶ Behave just like printf from C

System tasks and functions

\$monitor Used in test benches. Prints every time an argument changes. Very bad for large projects.

e.g. `$monitor("format",signal,...)`

\$display Can be used in either test benches or design, but not after synthesis. Prints once. Not the best debugging technique for significant projects.

e.g. `$display("format",signal,...)`

\$strobe Like display, but prints at the end of the current simulation time unit.

e.g. `$strobe("format",signal,...)`

\$time The current simulation time as a 64 bit integer.

\$reset Resets the simulation to the beginning.

\$finish Exit the simulator, return to terminal.

More available at ASIC World.



Testbench examples

Test Bench Setup

```
module testbench;
    logic clock, reset, taken, transition, prediction;

    two_bit_predictor(
        .clock(clock),
        .reset(reset),
        .taken(taken),
        .transition(transition),
        .prediction(prediction));

    always
    begin
        clock = #5 ~clock;
    end
```

Test Bench Test Cases

```
initial
begin
    $monitor("Time:%4.0f clock:%b reset:%b taken:%b trans:%b"
             "pred:%b", $time, clock, reset, taken,
             transition, prediction);
    clock = 1'b1;
    reset = 1'b1;
    taken = 1'b1;
    transition = 1'b1;
    @(negedge clock);
    @(negedge clock);
    reset = 1'b0;
    @(negedge clock);
    taken = 1'b1;
    @(negedge clock);
    ...
    $finish;
end
```



Testbench Tips

Remember to...

- ▶ Initialize all module inputs
- ▶ Then assert reset
- ▶ Use @(negedge clock) when changing inputs to avoid race conditions



FIR example (might contain bugs...)

```
module my_fir (
    input          clk ,
    input          reset_n,
    input signed [7:0] fir_in,
    input          fir_val,
    output reg      fir_out_val,
    output reg signed [6:0]   fir_out  );

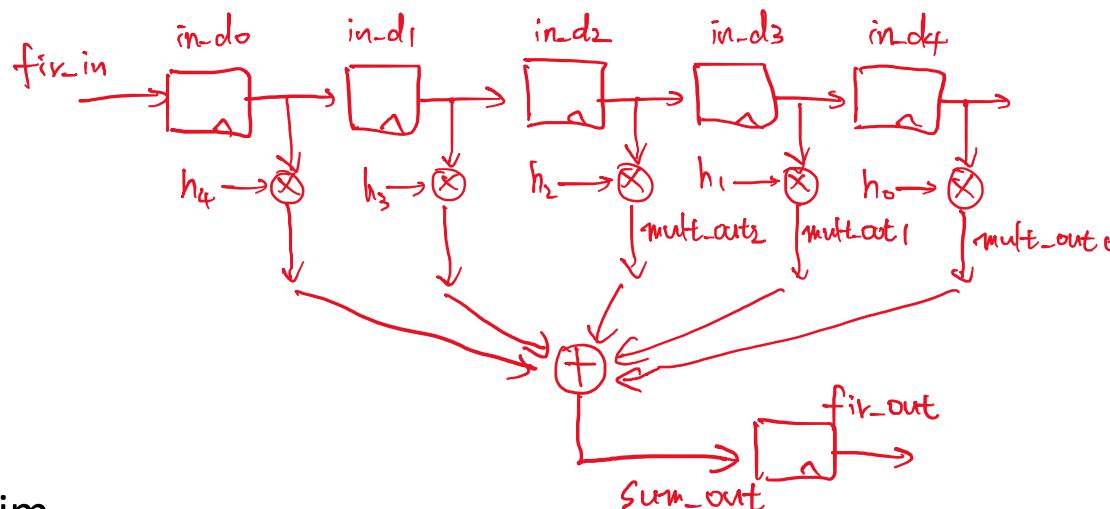
//=====
// Internal signals (wires and FFs)
//=====

localparam      h0      = -7;
localparam      h1      = 13;
...
reg           signed [7:0]   in_d0;
reg           signed [7:0]   in_d1;
...
reg           signed [7:0]   in_d4;
Reg            in_val_reg;
wire          signed [11:0]  mult_out0;
wire          signed [11:0]  mult_out1;
...
wire          signed [14:0]  sum_out;
...
assign        mult_out0 = in_d4*h0;      // combinational logic
assign        mult_out1 = in_d3*h1;
...
assign        sum_out = {{3, mult_out0[11]}, mult_out0} + {{3, mult_out1[11]}, mult_out1} + {{3, mult_out2[11]}, mult_out2} + {{3, mult_out3[11]}, mult_out3} + {{3, mult_out4[11]}, mult_out4};      // sign extension and then add.
```

```

always @(posedge clk or negedge reset_n)
begin
  if (~reset_n)
    begin
      in_d0    <= 0;
      in_d1    <= 0;
      ...
      in_val_reg      <= 1'b0;
      fir_out_val     <= 1'b0;
      fir_out        <= 0;
    end
  else
    begin
      in_val_reg      <= #1 fir_val;
      in_d0            <= #1 fir_in;
      in_d1            <= #1 in_d0;
      ...
      fir_out_val     <= #1 in_val_reg;
      fir_out          <= #1 sum_out[9:3];      // truncation without checking saturation
    end
  end
endmodule

```



FIR Testbench example

```

`timescale 1 ns / 1 ps

module my_fir_tb;
parameter CLOCK_PERIOD      = 125;      // 8MHz
reg                  clk;
reg                  reset_n;
reg signed [7:0]      fir_in;
reg                  fir_val;
reg                  fir_out_val;
reg signed [6:0]      fir_out;
integer               finput, in_read_cnt;

my_fir UUT(
    .clk      (clk      ),
    .reset_n  (reset_n  ),
    .fir_in   (fir_in   ),
    .fir_val   (fir_val  ),
    .fir_out   (fir_out  ),
    .fir_out_val(fir_out_val)
);
initial
begin
    finput      = $fopen("input.txt", "r");
    clk         = 1'b0;
    reset_n    = 1'b1;
    in_read_cnt = -1;
    #(CLOCK_PERIOD)
    reset_n    = 1'b0;
    #(CLOCK_PERIOD)
    reset_n    = 1'b1;
    ...
end
always #(CLOCK_PERIOD/2.0) clk = ~clk;

```

```

always @ (negedge clk)
begin
#1
begin
if (in_read_cnt < 1)
    fir_val      = 1'b0;
else
begin
in_read_cnt      = $fscanf(finput,"%d %d\n", fir_in, fir_val);

if (in_read_cnt < 1)
begin
fclose(finput);
fir_val      = 1'b0;
end
else
begin
end
end
end

```