Verilog Coding Guidelines|

By

Verilog Course Team

Email:info@verilogcourseteam.com

www.vlsi-faqs.blogspot.com

www.verilogcourseteam.com

DISCLAIMER

Verilog Course Team does not warrant or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed. No warranty of any kind, implied, expressed or statutory, including to fitness for a particular purpose and freedom from computer virus, is given with respect to the contents of this document or its hyperlinks to other Internet resources. Reference in this document to any specific commercial products, processes, or services, or the use of any trade, firm or corporation name is for the information, and does not constitute endorsement, recommendation, or favoring.

About Verilog Course Team

Verilog Course Team is a Electronic Design Services (EDS) for **VLSI / EMBEDDED and MATLAB**, delivering a wide variety of end-to-end services , including design , development, & testing for customers around the world .With proven expertise across multiple domains such as Consumer Electronics Market ,Infotainment, Office Automation, Mobility and Equipment Controls. Verilog Course Team is managed by Engineers / Professionals possessing significant industrial experience across various application domains and engineering horizontals . Our engineers have expertise across a wide range of technologies, to the engineering efforts of our clients. Leveraging standards based components and investments in dedicated test lab infrastructure; we offer innovative, flexible and cost-effective Services and solutions.

Our Mission

Our mission is to provide cost effective, technology independent, good quality reusable Intellectual Property cores with quality and cost factor are our important constraints so as to satisfy our customers ultimately. We develop and continuously evaluate systems so as to pursue quality in all our deliverables. At our team, we are completely dedicated to customer's requirements. Our products are designed and devoted to empower their competitive edge and help them succeed.

1. Naming Conventions

1.1 Character set

Use only the characters [a-z][A-Z][0-9] $ and "_" in the identifiers used for naming module, ports, wires, regs, blocks etc. Do not use escaped identifiers to include special characters in identifiers.

Do not use the character "_" as the first or last character of an identifier. Do not use numerals as first character. Do not use capital letters for identifier except Parameter and define

Example:conventions.v

```
module conventions;
reg [7:0] a,a$b,b,c;
reg clk;

initial
 begin
       clk=0;
       c = 1;
       forever  #25 clk = !clk;
 end


always @ (posedge clk)
 begin
       a = b;
       b = c;
 end
endmodule
```

1.2 Case sensitive

Use lower case letters for all identifiers leaving the upper case letters for macros and parameters. Do not use the mixed case style. Also, ensure that all the identifiers in the design are unique even in a case insensitive environment.

Example:

module / / keyword
Module / / unique identifier but not keyword
MODULE / / unique identifier but not keyword

Identifier Name: fifoReadPointer.
Use: fifo_read_pointer- instead.

1.3 No keywords

Do not use Verilog keywords as identifiers. Avoid keywords from both the HDLs as RTL code of a re-usable design may have to be made available in both languages.

Example:

input –keyword
output –keyword

1.4 Use meaningful Names

Create identifiers by concatenating meaningful words or commonly used acronyms separated by character "_".

Example:

Use en_state_transition instead of est or en_st_trn.

1.5 Identifier length, and number of parameters

Do not to use very long identifiers. This is especially true for parameters. Design unit names of parameterized modules are created by concatenating instance names, values and parameter names during design elaboration. Limit the maximum number of characters in an identifier to 25.

1.6 Parameter/Define naming convention

Parameter and Define must be declared in Capital Letter.

Example:

```
Parameter DATA_WIDTH=3'b111  ;
`define EXAMPLE
```

1.7 Module names

Name the top level module of the design as <design_name>_top. Module name & file name must be identical This is typically the module containing IO buffers and other technology- dependent components in addition to module <design_name>_core. Module <design_name>_core should contain only technology independent portion of the design. Name the modules created as macro wrappers <macro_name>_wrap.

Example:

```
module test (port1,port2,…);
endmodule
```

The file should be saved as test.v

1.8 Instance names

If the module has single instance in that scope use inst_<modulename> as instance name. If there are more than one instance, then add meaningful suffixes to uniquify the names. Remember that the instance name in gate level netlist is a concatenation of RTL instance name and all the parameter ids and values in the instantiated module.

- **A module may be instantiated within another module**
- **There may be multiple instances of the same module**
- **Ports are either by order or by name**
- **Use by order unless there are lots of ports**
- **Can not mix the two syntax's in one instantiation**
- **Always use module name as instance name.**

Example:

memory memory_instance

syntax for instantiation with port order:

module_name instance_name (signal, signal...);

syntax for instantiation with port name:

module_name instance_name (.port_name(signal), .port_name (signal)...);

1.9 Blocks names

Label all the always blocks in the RTL code with meaningful names. This will be very useful for grouping/ungrouping of the design in synthesis tool and will result in better error/info messages. It is a standard practice to append the block labels with "_comb" or "_seq" depending on whether it is combinatorial or sequential.

Example:

```
module block_names;
reg [7:0] a,b,c; // register declarations
reg clk;

always@(b or c)
 begin:comb
       c<=a;
        b<=a;
 end
always @ (posedge clk):
 begin:seq
        a <= b;
        b <= c;
 end
endmodule
```

1.10 Global signals

Keep same names for global signals (rst, clk etc.) in all the hierarchies of the design. This should be true for any signal which are used in multiple design hierarchies. The actuals and formals in instantiation port maps should be the same IDs.

1.11 Clock signals

Name the clock signal as clk if there is only one clock in the design. In case of multiple clocks, use _clk as suffix.

Example:

pci_clk, vci_clk.

Never include the clock frequecy in clock signal name (40MHz_clk) since clock frequencies often change in the middle of the design cycle.

1.12 Reset signals

Name the reset signal as rst if there is only one reset in the design. In case of multiple resets, use _rst as suffix.

Example:

pci_rst, vci_rst.

1.13 Active low signals

All signals are lowercase alpha, numeric and underscore only.

Use _n as suffix.

Example:

intr_n, rst_n, irdy_n.

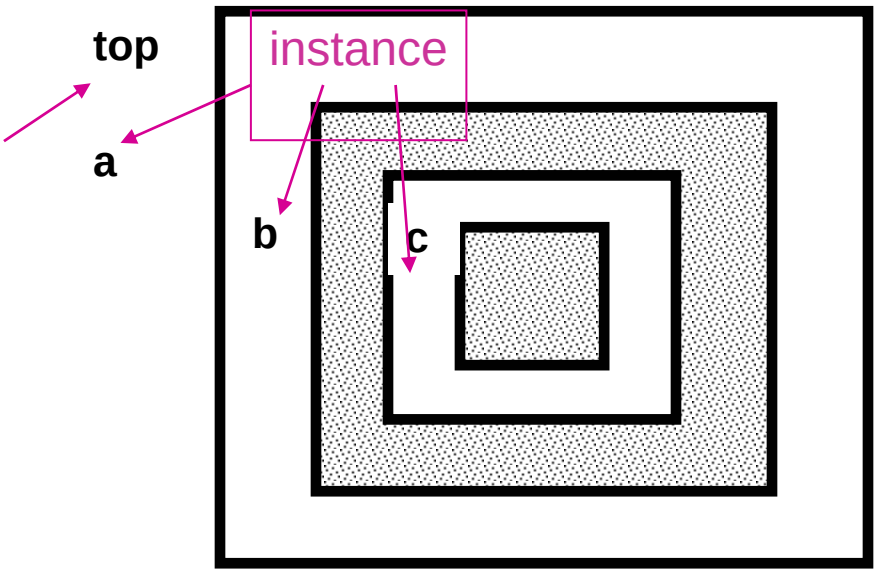
Avoid using characters '#' or 'N' as suffixes even in documents.

1.14 Module Hierarchy

A hierarchical path in Verilog is in form of:

module_name.instance_name.instance_name

top.a.b.c is the path for the hierarchy below.



1.15 Use of Macros

Macros are required to be used for any non-trivial constants, and for all bit-ranges. This rule is essential both for readability and maintainability of code. Having two inter-connected modules, each of which defines a bus as '17:0' is a recipe for disaster. Busses are preferably defined with a scheme such as the following:

```
`define   BUS_MSB   17
`define   BUS_LSB   0
`define   BUS_SIZE   (`BUS_MSB-`BUS_LSB+1)
`define   BUS_RANGE   `BUS_MSB:`BUS_LSB
```

This will minimize the number of places that have to be changed if the bus size must be changed.

1.16 MEMORY DECLARTION

Memories are declared as two-dimensional arrays of registers.

syntax:

reg [msb:lsb] identifier [first_addr:last_addr] ;

where

msb:lsb determine the width (word size) of the memory
first_addr:last_addr determine the depth (address range) of the memory

```
reg [15:0] mem [0:255];     // 256x16 memory
reg [0:31] mem_a [1023:0];      //1kx32 memory
```

1.17 Abbreviation

Use consistent abbreviation as shown:

Signal Naming Abbreviation

Signal/CellType	Abbreviation	Signal/Cell	Type Abbreviation
address	addr	shift	shift
IP address	ipaddr	chip select	cs
data	data	output enable	oe
control	ctrl	clock	clk
master	mst	comparator	comp
port	port	bank	bank
pipe	pip	adder/addition	add
packet	pack	subtractor/subtract	sub
counter	cnt	buffer	buff
count enable	cnten	fifo	fifo
load enable	lden	input	in
ready	rdy	output	out
early	erly	bidirectional	bidir
multiplexor	mux	main memory	mm
multiplexor select	muxsel	interface	if
register	reg (more stuff here...)		

2. STYLE

2.1 Page width: 75 characters

Considering the limited page width supported in many terminals and printers, restrict the maximum line length to 75 characters. For reuse macros reduce this number to 72 to comply with RMM.

2.2 No tabs

Do not use tabs for indentation. Tab settings are different in different environments and hence can spoil the indentation in some setup.

2.3 Port ordering

Arrange the port list and declarations in a cause and effect order. Group the list/declaration on the basis of functionality rather than port direction etc. Specify the reset and clock signals at the top of the list.

2.4 One statement per line

Limit the number of HDL statements per line to one. Do not include multiple statements, separated by semicolon, in the same line. This will improve readability and will make it is easy to process the code using scripts and utilities.

2.5 One declaration per line

Limit the number of port, wire or reg declaration per line to one. Do not include multiple declarations, separated by commas, in the same line. This will make it easy to comment, add, or delete the declared objects.

Example:

Wrong way:

```
input trdy_n, stop_n;
```

Right way:

```
input trdy_n;
input stop_n;
```

3. COMMENT

3.1 Comment blocks vs scattered comments

Describe a group of logic at the beginning of the file (in the header) or at the top of a block or group of blocks. Avoid scattering the comment for a related logic. Typically the reader would like to go through the comment and then understand the code itself. Scattered comment can make this exercise more tedious.

Example:

```
//File:      <filename>
//purpose:
//Project:
//Author:
```

3.2 Meaningful comments

Do not include what is obvious in the code in your comments. The comment should typically cover what is not expressed through the code itself.

Example:

History of a particular implementation, why a particular signal is used, any algorithm being implemented etc.

3.3 Single line comments

Use single line comments where ever possible. i.e. Use comments starting with '//' rather than '/* .. */' style. This makes it easy to cut-paste or move around the code and comments. It is also easy to follow the indentation with single line comments which makes the code more readable.

```
module pound_one;
reg [7:0] a,a$b,b,c; // register
declarations
reg clk;

initial
 begin
       clk=0;  // initialize the clock
       c = 1;
       forever  #25 clk = !clk;
 end
/*  This section of code implements
    a pipeline */
always @ (posedge clk)
 begin
       a = b;
       b = c;
 end
endmodule
```

4. DO'S AND DONT'S

4.1Use non-blocking assignments in sequential blocks

All registers assignments are concurrent. No combinatorial logic is allowed in sequential blocks. Always use non-blocking statements here.

```
module two_stage(Q, D, CLK);
input D, CLK;
output Q;

reg Q, P;

always @ (posedge CLK)
  begin
    P <= D;
    Q< = P;
end
```

4.2 Use blocking assignments in combinational blocks

Concurrency is not needed here. Often the combinatorial logic is implemented in multiple steps. Always use blocking statements for combinatorial blocks.

```
module two_stage(Q, D, data);
input D, data;
output Q;

reg Q, P;

always @ (data)
  begin
    Q = P;
    P = D;
 end
```

4.3 Ensure that there are no unused signals

Unused signals in the designs are often clear indication of incomplete or erroneous design. Check to make sure that design does not contain such signals.

4.3 Ensure that there are no un-driven signals

Un-driven signals in the designs are mostly clear indication of design errors. Check to make sure that design does not contain such signals.

5. FILE STRUCTURE

5.1 One file, one module

Create separate files for each modules. Name the file <module>.v. The only exceptions for this file naming convention shall be the technology-dependent modules (top module or macro wrapper modules). These files shall be appropriately named like design_name_fpga.v, design_name_tsmc.v, or design_name_virtex.v.

5.2 File header

Each source file should contain a header at the top of the file in the following format:

```
////////////////////////////////////////////////////////////////////
//////////
//(c) Copyright 2007 company name. All rights reserved
//
// File: <Filename>
// Project: <Projectname, Customer>
// Purpose: <One line description of the functionality>
// Author: <Author's name(s)>
//
// $Id: index.html,v 1.1 1999/03/09 01:55:57 george Exp $
//
// Detailed description of the module included in the file.
//Include relavant part of the spec
// Logical hierarchy tree
// Block diagrams
// Timing diagrams etc.
//
////////////////////////////////////////////////////////////////////
```

The above example is for verilog. Change the comment characters appropriately for other source types. Example: "#" in Tcl, Perl and CSH. The presence of variable Id in the header will capture the filename, user, version information every time the file is checked-in/committed.

5.3 Modification history

Each file should contain a log section at the bottom of the file in the following format:

```
//////////////////////////////////////////////////////////////////////
////
//
// Modification History:
//
// $Log$
//
//////////////////////////////////////////////////////////////////////
```

Listing the modification history at the top of the file can be annoying as one has to scroll down to reach the code every time the file is opened for reading. The variable Log will cause RCS/CVS to capture the user-comments entered during each check-in/commit as comments in footer section.

5.4 Include Files

Keep the `define statements and Parameters for a design in a single separate file and name the file DesignName_params.v

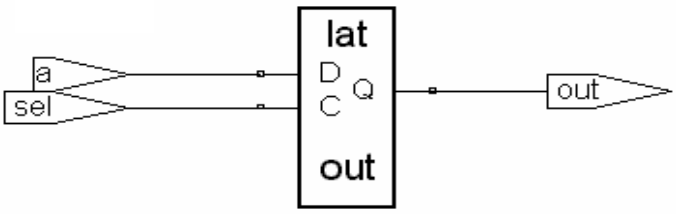
6. SIMULATION RELATED

6.1Incomplete sensitivity list

Synthesis tools issue only warnings for incomplete sensitivity lists in always (process) statements. However, such cases can result in simulation mismatch between RTL and Gate level. Ensure that all the signals accessed inside the blocks are included in the sensitivity list.

Example:

```
module default (out, sel, a);
input sel, a;
output out;
reg out;
always @(sel )
  begin
     if(sel)
      out = a;
  end
endmodule
```



6.2 Extra entries in sensitivity list

Extra entries in sensitivity list can slow down the simulation. Check the design to ensure that there are no unnecessary signals in the lists.

Example:

```
module default (out, sel, a);
input sel, a;
output out;
reg out;
always @(sel or a or out)
  begin
     if(sel)
       out = a;
  end
endmodule
```

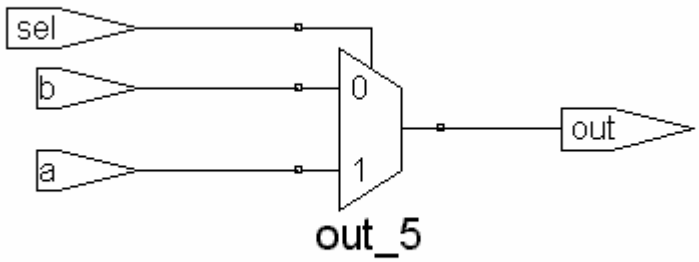
7. SYNTHESIS RELATED

7.2 if-else-Statement

When using if - else statement infers a priority – encoded logic, cascaded combination of multiplexers.

Example: Good Coding

```
The following generates a mux:
reg out, sel, a, b;
always @ (sel or a or b)
if(sel)
        out = a;
else out = b;
```

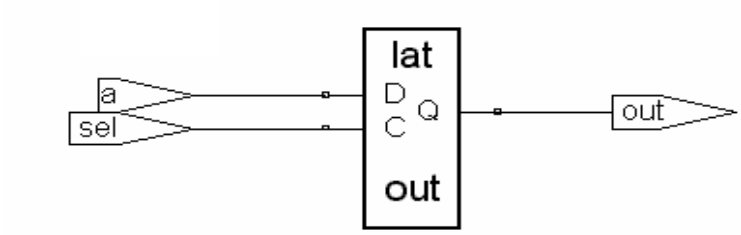


Example: Bad Coding

The following infers a latch:

```
always @ (sel or a or b)
if(sel)
        out = a;
```

Can you see why?



7.2 Case statement

A Case statement infers a single level multiplexer.

Synopsys full_case directive

From synthesis tool perspective, a full_case statement is the one in which every possible binary value is considered as a case item. When synopsys tool detects a full_case directive for a non-full case statement, it optimizes the logic in a way that outputs are don't care for unspecified case items.

Example: Does not infer latches

```
always @(SEL or A or B or C)
begin
case (SEL) //synopsys full_case
3'b001 : OUT <= A;
3'b010 : OUT <= B;
3'b011 : OUT <= C;
end
endcase
end
```

Infers latches for OUT because not all cases are specified

```
always @(SEL or A or B )
begin
case (SEL)
3'b001 : OUT <= A;
3'b010 : OUT <= B;
3'b011 : OUT <= C;
end
endcase
end
```

Synopsys Parallel_case directive

A parallel case statement is the one in which it is possible to match the case expression to only one case item. When synopsys detects the use of a "parallel_case" directive it optimizes the logic assuming that case expression would match only one case item thus preventing the synthesis tool to optimize for unnecessary logic leading to a reduction in area.

7.3 Infer Registers

Registers (flip-flop) are the preferred mechanism for sequential logic. To maintain consistency and to ensure correct synthesis, use the following templates to infer technology independent registers.

Example

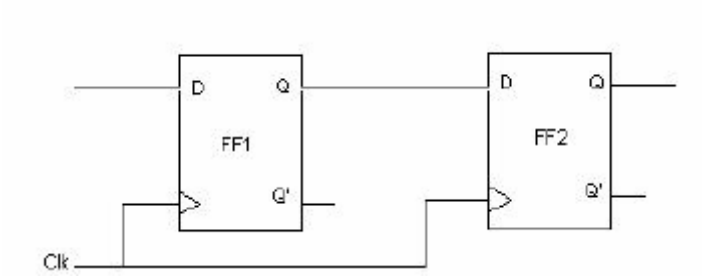
```
always @(posedge clk)
begin
if(rst)
begin
 ……
end
else
begin
…….
end
end
```

8. RACE CONDTION

```
always @(posedge clk)
begin
Q = A; ……(1)
End

always @(posedge clk)
begin
A = B; ……(2)
end
```

In this example, it is uncertain whether the output “Q” will get the old value of “A” or the newly assigned value of “A”. To be precise, the order of execution of the statements (1) and (2) is uncertain and may vary with different simulators.

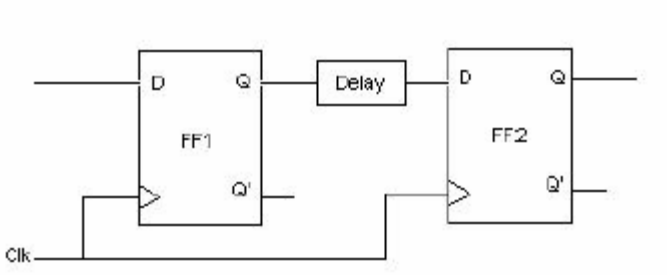


RACE SOLVED

```
always @(posedge clk)
begin
Q <= A; ……(1)
End

always @(posedge clk)
begin
A< = B; ……(2)
end
```

By using non-blocking statements it is made sure that right hand side of the statement is evaluated and stored before assignments are done.



9. Coding State Machines

- Separate the state machine HDL description into two process, one for the combinational logic and one for the sequential logic.
- Keep FSM logic and non-FSM logic in separate modules.
- Assign a default state for the state machine. This is useful to implement graceful entry into idle state if no other state is initiated.
- Always use One-Hot encoding method style.

Example

```
module example_a(clk, rst, input_sig_1, input_sig_2, a, b);
input clk, rst, input_sig_1, input_sig_2;
output a, b;
parameter S0 = 2'h0, S1 = 2'h1, S2 = 2'h2;
reg [1:0] state, next_state;
reg a,b;

always @ (posedge clk)
  if (rst)        // Fully synchronous reset
    state <= #1 S0;
  else
    state <= #1 next_state;

always @ (state or input_sig_1 or input_sig_2 ) // sensitive to all state
and all inputs
case (state)
     S0:  begin
          b = `FALSE;
          if (input_sig_1 || input_sig_2 )
            a = `TRUE;
          else
            a = `FALSE;
          if(input_sig_1 == `TRUE)
            next_state = S1;
          else
            next_state = S0;
          end
     S1: begin
          b = `TRUE;
          a = `FALSE;
          if(input_sig_2 == `TRUE)
            next_state = S2;
          else
            next_state = S0;
         end
     S2: begin
          b = `FALSE;
          a = `FALSE;
          next_state = S0;
         end
     default: begin
                a = 1'bx;
                      b = 1'bx;
               next_state = S0;
              end
   endcase
endmodule
```