# RTL CODING FOR LOGIC SYNTHESIS

## 1.1. Synthesizable and Non-Synthesizable Verilog constructs

<table>
<tr><th></th><th>Synthesizable</th><th>Non-Synthesizable</th></tr>
<tr><td>Basic</td><td>Identifiers, escaped identifiers, Sized constants (b, o, d, h), Unsized constants (2'b11, 3'07, 32'd123, 8'hff), Signed constants (s) 3'bs101, module, endmodule, macromodule, ANSI-style module, task, and function port lists</td><td>system tasks, real constants</td></tr>
<tr><td>Data types</td><td>wire, wand, wor, tri, triand, trior, supply0, supply1, trireg (treated as wire), reg, integer, parameter, input, output, inout, memory(reg [7:0] x [3:0];), N-dimensional arrays,</td><td>real, time, event, tri0, tri1</td></tr>
<tr><td>Module instances</td><td>Connect port by name, order, Override parameter by order, Override parameter by name, Constants connected to ports, Unconnected ports, Expressions connected to ports,</td><td>Delay on built-in gates</td></tr>
<tr><td>Generate statements</td><td>if,case,for generate, concurrent begin end blocks, genvar,</td><td></td></tr>
<tr><td>Primitives</td><td>and, or, nand, nor, xor, xnor,not, notif0, notif1, buf, bufif0, bufif1, tran,</td><td>User defined primitives (UDPs), table, pullup, pulldown, pmos, nmos, cmos, rpmos, rnmos, rcmos, tranif0, tranif1, rtran, rtranif0, rtranif1,</td></tr>
<tr><td>Operators and expressions</td><td>+, - (binary and unary)</td><td></td></tr>
<tr><td>Bitwise operations</td><td>&, |, ^, ~^, ^~</td><td></td></tr>
<tr><td>Reduction operations</td><td>&, |, ^, ~&, ~|, ~^, ^~, !, &&, || , ==, !=, <, <=, >, >=, <<, >>, <<< >>>, {}, {n{}}, ?:, function call</td><td>===, !==</td></tr>
</table>

| | | |
|---|---|---|
| **Event control** | event or, @ (partial), event or using comma syntax, posedge, negedge (partial), | Event trigger (->), delay and wait (#) |
| **Bit and part selects** | Bit select, Bit select of array element, Constant part select, Variable part select ( +:, -:), Variable bit-select on left side of an assignment | |
| **Continuous assignments** | net and wire declaration, assign | Using delay |
| **Procedural blocks** | always (exactly one @ required), | initial |
| **Procedural statements** | ;, begin-end, if-else, repeat, case, casex, casez, default, for-while-forever-disable(partial), | fork, join |
| **Procedural assignments** | blocking (=), non-blocking (<=) | force, release |
| **Functions and tasks** | Functions, tasks | |
| **Compiler directives** | `define, `undef, `resetall, `ifndef, `elsif, `line, `ifdef, `else, `endif, `include | |

## 1.2. How hardware is infered?

### 1.2.1 Register inference

Whenever there is a 'posedge' or 'negedge' construct synthesis tool infers a flip flop.

```
always @(posedge clk)
        output_reg <= data;
```

Above code infers D-flip flop.

**Asynchronous reset :**

```
module async_rst(clk,rst,data,out);
	input clk, rst, data;
	output out;
	reg out;

	always @(posedge clk or negedge rst)
	begin
	if(!rst)
	out<=1’b0;
	else
	out<=data;
	end
endmodule
```

In above case the sensitivity list includes both clock and the rst and hence it infers a asynchronous reset flip flop. rst has negedge in sensitivity list and hence same should be checked in the code.

**Synchronous Reset:**

```
module sync_rst(clk,rst,data,out);
	input clk, rst, data;
	output out;
	reg out;

	always @(posedge clk)
	begin
```

```
        if(!rst)
        out<=1'b0;
        else
        out<=data;
        end
endmodule
```

In above case the sensitivity list doesn't include 'rst' and hence it infers a synchronous reset flip flop.

### 1.2.2 Mux Inference

"if else" loop infers a mux.

eg.:

```
if(sel) z=a; else z=b;
```

General case statement infers a mux. If case statement is a overlapping structure then priority encoder in infered. Case statements only works with true values of 0 or 1.

### 1.2.3. Priority Encoder Inference

Multiple if statements with multiple branches result in the creation of priority encoder structure.

"if else if" infers priority encoder.

### 1.2.4. Combo Logics

If unknown 'x' or 'z' is assigned then it will be realized into tristate buffer. So avoid using 'x' and 'z'. usage of these may mislead synthesis.

Eg.:

```
assign tri_out=en ? tri_in : 1b'z;
```

### 1.2.5. if vs case

Multiflexer is faster circuit. Therefore is priority encoding structure is not required then use 'case' staements instead of 'if-else' statement.

Use late arriving signal early in an 'if-else' loop to keep these late arriving signals with critical timing closest to the output of a logic block.

### 1.2.6. Proper partitioning for synthesis

Properly partition the top level design based on functionality. Keep related combinational logic in same module. It is not recommended to add glue logic at top level of the module. Heirarchical designs are good but unnecessary heirachies may limit the optimizations across the heirarchies. It is pracically observed that deeper heirarchies cause miserably failing boundary optimizations due to incresed number of either setup or hold fixing buffer insertion. In such cases ungrouping or flattening heirachy command can be used to flatten the unwanted heirarchies before cmpiling the design to achieve better results.

### 1.2.7 FSM synthesis guidelines

If you are using state machine for coding then take care to separate it from other logic. This helps synthesis tools to synthesize and optimize FSM logic much better. Use "parameter" in Verilog to describe state names. A "always" block should have all the combinational logic for computing the next state.

### 1.2.8. Blocking vs non-blocking-race condition

- Never mix a description of combinational (blocking) construct with sequential (nonblocking).
- Blocking: combinational →racing

Since the final outputs depends on the order in which the assignments are evaluated, blocking assignments within sequential block may cause race condition.

- Nonblocking: sequential →No race condition

Nonblockng assignments closely resemble hardware as they are order independent.

- Most of the applications which require data tranfer within module required to be written using non-blocking assignment statement.

### 1.2.9. Technology independent RTL coding

Write HDL code in technology independent fasion. This helps reusage of the HDL code for any technology node. Do not hard code logic gates from the technology library unless it is necessary to meet critical timing issues.

### 1.2.10. Pads separate from core logic

Pads are instantiated like any other module instantiation. If design has large number of I/O pads it is recommended to keep the pad instantiations in a separate file. Note that pads are technology dependant and hence the above recommendation!

### 1.2.11. Clock logic guidelines

In case of multiple clocks in the design, make sure that clock generation and reset logics are written in one module for better handling in synthesis. If a clock is used in different modules of different heirarchy then keep clock names common across all the modules. This makes constraining that clock easier and also supports better handling of synthesis scripts.

- **Don't use mixed clock edges**

mixing of edge sensitive and level sensitive lists are not allowed. Below code is a wrong one.

```
always @(posedge clk or posedge rst)
```

- **Avoid clock buffers or any other logic**

If any signal crosses multiple clock domains having different clock frequencies then those signals must be properly synchronised with synchronous logic. Synthesis tools can't optimize any timing paths between asynchronous clock domains.

### 1.2.12. Reset logic guidelines

**Synchronous Reset:**

**Advantages:**

- Easy to synthesize, just another synchronous input to the design.

**Disadvantages:**

- Require a free running clock. At power-up clock is must for reset.

**Asynchronous Reset:**

**Advantages:**

- Doesn’t require a free running clock.
- Uses separate input on flip flop, so it doesn’t affect flop data timing.

**Disadvantages:**

- Harder to implement. Considered as high fanout net
- STA, simulation, DFT becomes difficult

### 1.2.13. Registered outputs

All outputs should be registered and combinational logic should be either at the input section or in between two registered stages of a module.

### 1.2.14. Incomplete sensitivity list

Sensitive list should contain all inputs. If inputs are missed in the sensitivity list, then the changes of that inputs will not be recognised by simulator. Synthesised logic in most cases may correct for the blocks containing incomplete sensitivity list. But this may cause simulation mismatches between source RTL and synthesised netlist. Generally synthesis tools issue a warning for the “always” block having incomplete sensitivity list. Registers can also be added in the sensitive list.

### 1.2.15. Avoid latch inference

- “if-else” statements must be end with ‘else’ statements. Else ‘unintentional latches’ will be realized (at output) due to the missing ‘else’ statement at the end.
- Same is true for ‘case’ statement. ‘default’ statement must be added.

**Work Around:**

Either include all possible combination of inputs or initialise the value before the loop starts.

Eg.:

```
if(z)   a=b;
```

Above code will infer a latch. Because if z=1, value of 'a' is defined. But if z=0 value of 'a' is not specified. Hence it is assumed that previous value has to be retained and hence latch is infered.

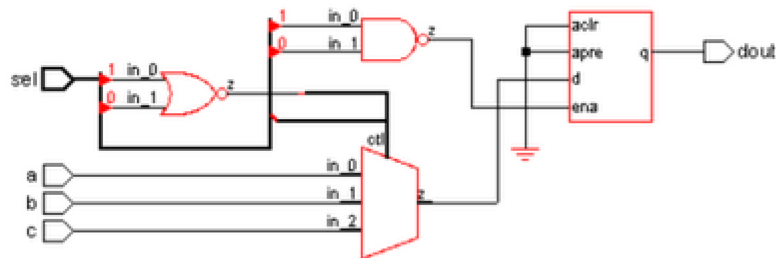
Eg.:

```
module latch_inf_test(a, x, y, t, out);
      input [2:0] a;
      input x, y, t;
      output out; reg out;

      always @(a or x or y or t)
      begin
      case(a)
                      3'b001:out=x;
                      3'b010:out=y;
                      3'b100:out=t;
      endcase
      end
endmodule
```

Eg.:

```
module case_latch(dout,sel,a,b,c);
      input [1:0] sel;
      input a,b,c;
```

```
    output dout;
    reg dout;

    always @(a or b or c or sel)
    begin
    case (sel)
    2'b00 : dout = a;
    2'b01 : dout = b;
    2'b10 : dout = c;
    endcase
    end
endmodule
```

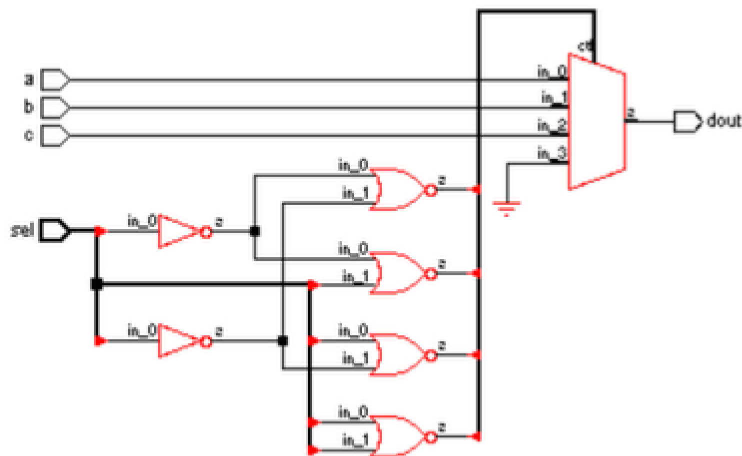



(Above code and figure are Courtesy of Cadence Manuals)

### Preventing a Latch by Assigning a Default Value

```
module case_default(dout,sel,a,b,c);
    input [1:0] sel;
    input a,b,c;
    output dout;
    reg dout;

    always @(a or b or c or sel)
```

```
      begin
      case (sel)
      2'b00 : dout = a;
      2'b01 : dout = b;
      2'b10 : dout = c;
      default : dout = 1'b0;
      endcase
      end
endmodule
```




(Above code and figure are courtesy of Cadence Manuals)

### 1.2.16. Use Constants

Use constants instead of hard coded numeric values.

Below coding style is not recommended:

```
wire [15:0] input_bus;
reg [15:0] output bus;
```

Recommended coding style:

```
`define INPUT_BUS_WIDTH 16
```

```
`define OUTPUT_BUS_WIDTH 16
wire [INPUT_BUS_WIDTH-1:0] input_bus;
reg [OUTPUT_BUS_WIDTH-1:0] output_bus;
```

Keep constants and parameters definitions in separate file with naming convention such as design_name.constants.v and design_name.parameters.v

### 1.2.17. General Coding guidelines for ASIC synthesis

- "Inference" of the logic should be given higher priority compared to instantiation of the logic.
- File name and module name should be same.
- A file should have only one module.
- Use lowercase letters for ports, variables and signal names.
- Use uppercase for constants, user defined types.

Source : http://asic-soc.blogspot.in/2012/06/rtl-coding-for-logic-synthesis.html