Verilog RTL Coding Guideline for Synthesis and Simulation

May 2014

Ando KI

Contents

What for

- Define a set of modeling rules for writing Verilog HDL description for synthesis
- Define how the semantics of Verilog HDL are used
- Describe the syntax of the language with reference to what shall be supported and what shall not be supported for interoperability
- Enhance the portability of Verilog-HDL-based design across synthesis tools conforming to this standard
- Minimize the potential for functional mismatch that may occur between the RTL model and the synthesized netlist

Terminologies (1/2)

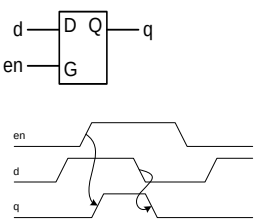
- Synchronous & asynchronous
  - Synchronous: Data that changes only on a clock edge
  - Asynchronous: Data that changes value independent of the clock edge
- Edge-sensitive & level-sensitive storage device
  - Edge-sensitive storage device: Any device mapped to by a synthesis tool that is edge-sensitive to a clock, e.g., a flip-flop.
  - Level-sensitive storage device: Any device mapped to by a synthesis tool that is level-sensitive to a clock, e.g., a latch.
- Combinational & sequential logic
  - Combinational logic: Logic that does not have any storage device, either edge-sensitive or level-sensitive.
  - Sequential logic: Logic that includes any kind of storage device, either level-sensitive or edge-sensitive.

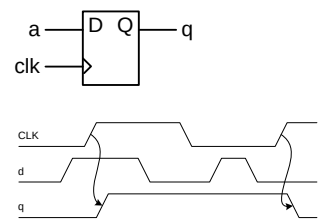
Terminologies (2/2)

- Instantiation & inference
  - Instantiation: explicitly select a technology-specific element in the target library.
  - Inference: direct the HDL Compiler to infer latches or flip-flops or other resource blocks from your Verilog or VHDL description.
- Technology-independent & technology-dependent
  - Keeping the pre-synthesis design source code technology-independent allows you to re-target to other technologies at a later time, with a minimum of re-design effort.
  - Technology-dependent design uses dedicated functions or design techniques optimized for speed or area
    - The designer may require detailed understanding on device architectures

Latch and flip-flop

- Latch
  - When en = 1, latch is *transparent*
    - D flows through to Q like a buffer
  - When en = 0, the latch is *opaque*
    - Q holds its old value independent of D
  - a.k.a. *transparent latch* or *level-sensitive latch*
- Flip-flop
  - When CLK rises, D is copied to Q
  - At all other times, Q holds its value
  - a.k.a. *positive edge-triggered flip-flop*, *master-slave flip-flop*





Modeling hardware elements

- Modeling combinational logic
- Modeling edge-sensitive sequential logic
- Modeling level-sensitive storage devices
- Modeling tri-state drivers
- Modeling read-only memories (ROM)
- Modeling random access memories (RAM)

Modeling combinational logic

- Continuous assignment
- Net declaration assignment
- Always statement
  - Event list shall not contain an edge event (posedge or negedge)
  - Event list must include all the variables read in the always statement
    - If not, mismatch can occur between simulation and synthesized logic.
  - Shall not use both a blocking assignment (=) and a non-blocking assignment (<=) in the same always statement

```
wire my_signal = A + B;

wire   your_signal;
assign your_signal = A + B;

always @ (in1 or in2) out = in1 + in2;

always @ (*) begin
    tmp1 = a & b;
    tmp2 = c & d;
    z    = tmp1 | tmp2;
end
// implicit event list expression yields comb. logic
```


```
always @ (posedge a or b) … …
// not supported; does not model comb. Logic

always @ (in) if (ena) out  = in;
                  else     out = in2;
 // should be always @ (in or ena) …

always @ (in1 or in2 or ena) if (ena) out = in1;
                                    else     out <= in2;
// don't mix = and <= in an always statement
```

Modeling edge-sensitive seq. logic

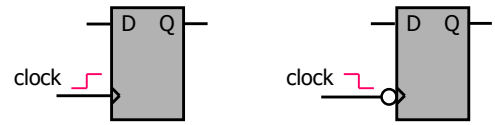
- Sequential logic shall be modeled using an always statement that has one or more edge event in the event list.

```
// positive edge
always @ (posedge <clock_name>) … …

// negative edge
always @ (negedge <clock_name>) … …
```


Modeling edge-sensitive storage devices

- An edge-sensitive storage shall not use multiple event lists
- Nonblocking procedural assignment should be used
- Multiple event lists are not supported within an always statement

```
reg out;
…
always @ (posedge clock) out <= in;
// out: pos-edge triggered storage

reg [3:0] out;
always @ (negedge clock) out <= in;
// out: neg-edge triggered storage with 4 elements
```




```
always @ (posedge clock) begin
        out <= 0;
        @ (posedge clock);
        out <= 1;
end
// not legal: multiple-edge event list
```

Edge-sensitive with synchronous set/reset

- An edge-sensitive storage device with a synchronous set/reset is modeled using an always statement whose event list only contains edge events representing the clock.

```
reg out;
…

always (posedge clock) if (reset) out <= 1'b0;
                       else       out <= in;
// synchronous reset

always @ (posedge clock) if (set) out <= 1'b1;
                         else     out <= in;
// synchronous set
```


```
always @ (posedge clock or reset) begin
        if (reset) out <= 1'b0;
        else       out <= in;
end
// do not mix edge and level sensitive event
```

Edge-sensitive with asynchronous set/reset (1/2)

- An edge-sensitive storage device with an asynchronous set/reset is modeled using an always statement whose event list contains edge events representing the clock and asynchronous control variables.
- Level-sensitive events shall not be allowed in the event list of an edge-sensitive storage device model.
- The first if and optional else if statements model asynchronous control and a final else statement specifies the synchronous logic portion.
  - Andy sequence of edge events can be in event list.

```
// generic form
always @ (posedge <condA> or negedge <condB>
          or negedge <condC> or …
          or posedge <clock>)
  if (condA) // positive polarity
    … … // asynchronous logic
  else if (~ <condB>) // negative polarity
    … … // asynchronous logic
  else if (~ <condC>)
    … … // asynchronous logic
  else if … …
  else // implicit posedge <clock>
    … … // synchronous logic
```

  - The final else statement
    - If there are N edge events in the event list, the else following (N-1) if's, at the same level as the top-level if statement, determines the final else.

Edge-sensitive with asynchronous set/reset (2/2)

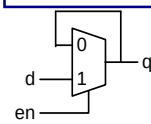
```
always @ (posedge clock or posedge set)
        if (set) out <= 1'b1;
        else     out <= din;
// edge-sensitive storage with asynchronous set


always @ (posedge clock or negedge reset)
         if (~reset) out <= 1'b0;
         else        out <= din;
// edge-sensitive storage with asynchronous reset

always @ (posedge clock or negedge clear)
         if (~clear) out <= 0;
         else
         if (ping)         out <= in;
         else if (pong) out <= 8'bFF;
         else              out <= din;
 // synchronous logic starts after the first else
```

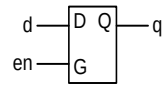

```
always @ (posedge clock or posedge reset)
        out <= in;
 // not legal: the if statement is missing

always @ (posedge clock or negedge clear)
        if (clear) out <= 0;
        else       out <= in;
// not legal: if condition does not match the polarity
// of the edge event
```

Modeling level-sensitive storage devices

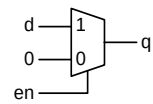
- The event list does not contain a edge events (posedge or negedge).
  - The same as the combinational logic.
- The event list of the always statement should list all variables read within the always statement.
  - The same as the combinational logic.
- There is no explicit assignment to the variable.
  - Implicitly value keeping semantics.
- Nonblocking procedural assignment should be used for variable.
- Note: it is latch inference intentionally.

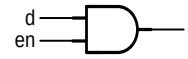
```
always @ (en or d) if (en) q <= d;
// If en is deasserted, q will hold its value.
```





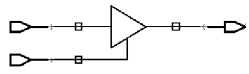

```
always @ (en or d)
         if (en) q <= d;
         else    q <= 'b0;
// Latch is not inferred, but a combinational logic.
// q is assigned on every execution of the always
// statement → meaning combinational logic.
```





Modeling tri-state drivers (1/3)

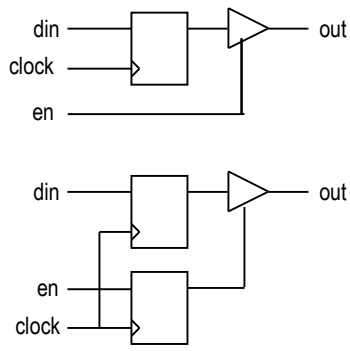
- Tri-state logic shall be modeled when a variable is assigned the value z.



```
always @ (q or en)
if (!en) out <= 'bz;
else     out <= q;
// out is a three-state driver.
```

```
module ztest (test2, test1, test3, en);
input    [0:1] en;
input    [7:0] test1, test3;
output [7:0] test2;
wire    [7:0] test2;
assign test2 = (en == 2'b01) ? test1 : 8'bz;
assign test2 = (en == 2'b10) ? test3 : 8'bz;
// test2 is three-state when en is 2'b00 or 2'b11.
endmodule
```


Modeling tri-state drivers (2/3)




```
always @ (posedge clock) q <= din;
assign out = en ? q : 1'bz;
// one edge-sen. storage with a tri-state driver
```

```
always @ (posedge clock)
        if (!en) out <= 1'bz;
        else     out <= din;
// two edge-sen. storages. One for din, one for en,
// with a tri-state driver for din-out.
```

Modeling tri-state drivers (3/3)

- Z assignment shall not propagate across variable assignment.

```
module ztest;
wire test1, test2, test3;
input test2;
output test3;
assign test1 = 1'bz;
assign test3 = test1 & test2;
// test3 will never receive a z assignment
// test2 always determines test3.
endmodule
```

```
always @ (in)
begin
tmp = 'bz;
out = tmp;
// out shall not be driven by three state drivers
// because the value 'bz does not propagate
// across the variable assignment.
end
```

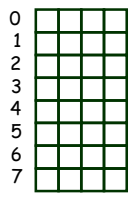

Modeling one-dimensional ROM

```
module rom_case(
output reg [3:0] z,
input wire [2:0] a; // Address - 8 deep memory.
always @* // @(a)
case (a)
3'b000: z = 4'b1011;
3'b001: z = 4'b0001;
3'b100: z = 4'b0011;
3'b110: z = 4'b0010;
3'b111: z = 4'b1110;
default: z = 4'b0000;
endcase
endmodule // rom_case
// z is the ROM, and its address size is
// determined by a.
```

Modeling two-dimensional ROM

- With fixed contents

```
module rom_2dimarray_initial (
output wire [3:0] z,
input wire [2:0] a; // address- 8 deep memory
// Declare a memory rom of 8 4-bit registers.
// The indices are 0 to 7:
reg [3:0] rom[0:7];
initial begin
rom[0] = 4'b1011;
rom[1] = 4'b0001;
rom[2] = 4'b0011;
rom[3] = 4'b0010;
rom[4] = 4'b1110;
rom[5] = 4'b0111;
rom[6] = 4'b0101;
rom[7] = 4'b0100;
end
assign z = rom[a];
endmodule
```



- With data in text file

```
module rom_2dimarray_initial_readmem (
output wire [3:0] z,
input wire [2:0] a);
// Declare a memory rom of 8 4-bit registers.
// The indices are 0 to 7:
reg [3:0] rom[0:7];
initial $readmemb("rom.data", rom);
assign z = rom[a];
endmodule
// with data in text file
```

```
// Example of content "rom.data" file:
1011 // addr=0
1000 // addr=1
0000 // addr=2
1000 // addr=3
0010 // addr=4
0101 // addr=5
1111 // addr=6
1001 // addr=7
```


Modeling RAM

- RAM element with edge-sensitive

```
// A RAM element is an edge-sensitive storage
// element:
module ram_test(
output wire [7:0] q,
input wire [7:0] d,
input wire [6:0] a,
input wire clk, we);
reg [7:0] mem [127:0];
always @(posedge clk) if (we) mem[a] <= d;
assign q = mem[a];
endmodule
```

- RAM element with level-sensitive

```
// A RAM element is a level-sensitive storage
// element:
module ramlatch (
output wire [7:0] q, // output
input wire [7:0] d, // data input
input wire [6:0] a, // address
input wire we; // clock and write enable
// Memory 128 deep, 8 wide:
reg [7:0] mem [127:0];
always @* if (we) mem[a] <= d;
assign q = mem[a];
endmodule
```

Contents

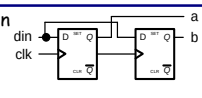
- Coding guideline for synthesis & simulation
  - Blocking and non-blocking
  - Race problem
  - Use parentheses
  - Resource sharing
  - Pre-schedule RTL
  - Full case for if statement
  - Full/parallel case for case statement
  - Pseudo comment for full case
  - Pseudo comment for parallel case
  - Casex
  - Casez

Blocking and non-blocking

- Blocking assignments (=) execute in sequential order.
- Non-blocking assignments (<=) execute concurrently.

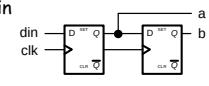
Blocking assignments (1/2)

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



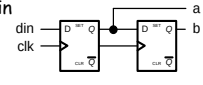
Non-blocking assignments (1/2)

```
always @ (posedge clk) begin
  a <= din;
  b <= a;
end
```



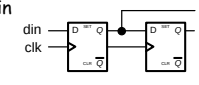
Blocking assignments (2/2)

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



Non-blocking assignments (2/2)

```
always @ (posedge clk) begin
  b <= a;
  a <= din;
end
```

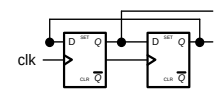


Race condition

- Blocking assignments follow order of statements.
- Parallel procedures have no order.
- Thus, blocking assignments across parallel procedures can cause race condition.
  - The result depends on execution order, which is not explicitly determined.

```
always @(posedge clk) a = b;
always @(posedge clk) b = a;
// With Blocking both start at same time
// a could transfer to b first and b also could
// transfer to a first. The order is not fixed.
```

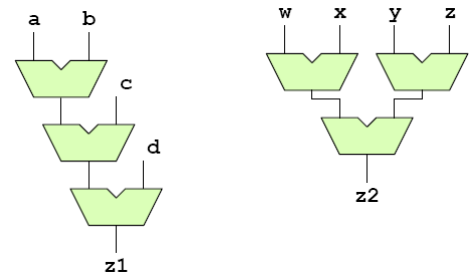
```
always @(posedge clk) a <= b;
always @(posedge clk) b <= a;
```



Use parentheses

- Use parentheses to control the structure of a complex design
- Use parentheses to guide synthesis

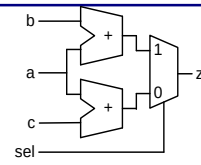
```
 assign z1 = a + b + c + d;
assign z2 = (w + x)+ (y + z);
// the same hardware blocks used,
// but z2 gives better delay-time characteristics than z1.
```



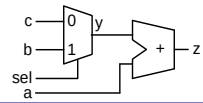
Resource sharing

- Operators can be shared within an always block by default.
- The resource sharing can help to save hardware resources, but may fail to make hardware smaller

```
always @(a or b or c or sel)
    if (sel) z = a + b;
    else     z = a + c;
// without sharing
// larger
```

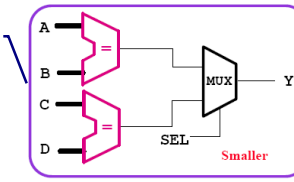



```
always @(a or b or c or sel) begin
    if (sel) y = b;
    else     y = c;
    z = a + y;
end
// with sharing
// smaller
```

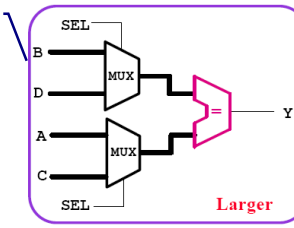



```
always @(SEL or A or B or C or D)
    if (SEL) Y = (A == B);
    else     Y = (C == D);
end
```

Without sharing, but smaller



With sharing, but larger



Pre-schedule RTL code

- Think if you can reduce logic from the architecture view before starting coding, pre-schedule your RTL design if possible.

```
[What wanted to be hardware]
assume there is A[31:0] and shiftcnt that is N
if a[31] == 0, result = {N{1'b0}, A[31], ...,A[N]}
if a[31] == 1, result = {N{1'b1}, A[31], ...,A[N]}
```

Be complex and requires more hardware

```
module shift (A, shiftcnt, z);
input [31:0] A;
input [4:0] shiftcnt;
output [31:0] z;
assign z = A[31] ? ((A >>shiftcnt)
                    |(((32'b1 <<shiftcnt) -1))
                      << (32 -shiftcnt))
                  : (A >>shiftcnt);
endmodule
```

Be simple and requires less hardware

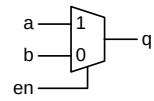
```
module shift (A, shiftcnt, z);
input [31:0] A;
input [4:0] shiftcnt;
output [31:0] z;
assign z = {{31{A[31]}},A} >>shiftcnt;
endmodule
```

Full case for if construct

- Be called full when all possible outcomes are accounted for.

It is full case.

```
always @ (en or a or b)
        if (en) q <= a;
        else    q <= b;
```

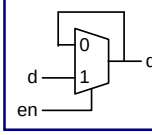


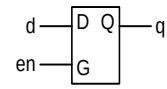
It is full case.

```
always @ (en or a or b) begin
        q = b;
        if (en) q = a;
end
```

It is not full case.

```
always @ (en or d) if (en) q <= d;
// If en is false, there may be need to preserve
// a previous value set onto the wire q.
```

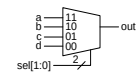




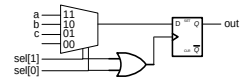
Full/parallel case for case construct (1/2)

- A case statement is called full if all possible outcomes are accounted for.
- A case statement is called parallel if the stated alternatives are mutually exclusive.

```
module full_par (sel, a, b, c, d, out);
input [1:0] sel;
input       a, b, c, d;
output      out; reg out;
always @ (sel or a or b or c or d)
  case (sel)
  2'b11:    out <= a;
  2'b10:    out <= b;
  2'b01:    out <= c;
  default: out <= d; // 2'b00
  endcase
endmodule
```

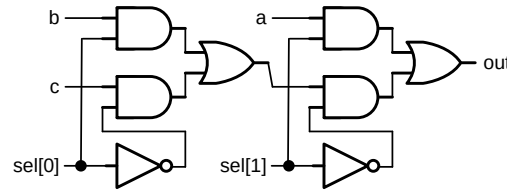



```
module par_not_full (sel, a, b, c, out);
input [1:0] sel;
input       a, b, c;
output      out; reg out;
always @ (sel or a or b or c)
  case (sel)
  2'b11:    out <= a;
  2'b10:    out <= b;
  2'b01:    out <= c;
  endcase
endmodule
```

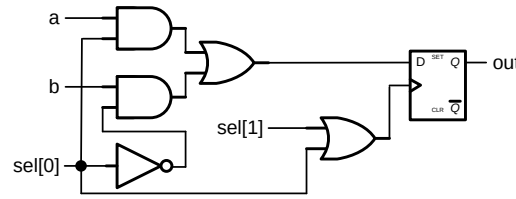


Full/parallel case for case construct (2/2)

```
module full_not_par (sel, a, b, c, out);
input [1:0] sel;
input       a, b, c;
output      out; reg out;
always @ (sel or a or b or c)
  case (sel)
  2'b1?:   out <= a; // 2'b10, 2'b11
  2'b?1:   out <= b; // 2'b01, 2'b11
  default: out <= c; // 2'b00
  endcase
endmodule
// If the case is 2'b11 occurs, the first outcome gets higher priority because it is closer to the output.
```




```
module not_full_not_par (sel, a, b, out);
input [1:0] sel;
input       a, b;
output      out; reg out;
always @ (sel or a or b)
  case (sel)
  2'b1?:   out <= a; // 2'b10, 2'b11
  2'b?1:   out <= b; // 2'b01, 2'b11
  endcase
endmodule
```



Pseudo-comment for full case

- The user may know that it is impossible for the missing case(s) to occur. Then, the user can use pseudo-comment for full case.
  - // synopsys full_case
  - (* synthesis, full_case *)
- Full_case indicates that all user-desired cases have been specified.
  - Other cases never occurs.
  - Let compiler think the cases left out be don't care.
- You can attach the full_case directive to tell the synthesis tool the Verilog case item expressions cover all the possible cases.

```
always @(SEL or A or B or C) begin
case (SEL)
3'b001 : OUT <=  A;
3'b010 : OUT <= B;
3'b100 : OUT <= C;
endcase
end
// Infers latches for OUT
// because not all cases are specified
// which make the logic keep its value
```

```
always @(SEL or A or B or C)
case (SEL) //synopsys full_case
3'b001 : OUT <= A;
3'b010 : OUT <= B;
3'b100 : OUT <= C;
default: OUT <= 'b0;
endcase
```

```
always @(SEL or A or B or C)
case (SEL) // synopsys full_case
3'b001 : OUT <=  A;
3'b010 : OUT <= B;
3'b100 : OUT <= C;
endcase
```

Pseudo comment for parallel case

- The user may know that the cases are really mutually exclusive. Then the user can use pseudo comment for parallel case.
  - // synopsys parallel_cases
  - (* synthesis, parallel_case *)
- Attach the parallel_case directive to improve the efficiency if no two case item expressions ever match the test expression at the same time.
  - E.g., state of FSM – cannot be multiple state at the same time.

```
always @(SEL or A or B or C)
case (SEL)
A : OUT <= 3'b001
B : OUT <= 3'b010
C : OUT <= 3'b100
endcase
// infers priority encoder
```

```
always @(SEL or A or B or C)
case (SEL) // synopsys parallel_case
A : OUT <= 3'b001
B : OUT <= 3'b010
C : OUT <= 3'b100
endcase
// infers multiplexer
```

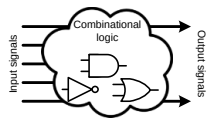

Casex

- The casex treats z or x or ? symbols in the expression or case items as don't – cares.

Casez

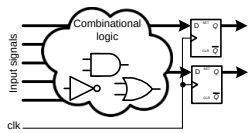
- The casez treats z or ? symbols in the expression or case items as don't-cares.
  - The syntax of literal numbers allows the use of the question mark (?) in place of z in the casez statement.

Coding



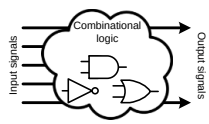

```
wire sig_in_a;
wire sig_in_b;
wire sig_out_c;
wire sig_out_d = sig_in_a & sig_in_b;
assign sig_out_c = sig_in_a | sig_in_b;
```

Combinational logic inference by continuous assignment (not wire and assign)




```
wire sig_in_a;
wire sig_in_b;
reg sig_out_c;
reg sig_out_d;

always @ (posedge clk) begin
    sig_out_d <= sig_in_a & sig_in_b;
    sig_out_c <= sig_in_a | sig_in_b;
end
```

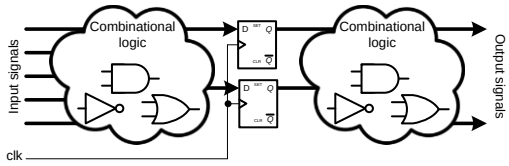
Register or Flip-Flop inference by always block (note "posedge clk")




```
wire sig_in_a;
wire sig_in_b;
reg sig_out_c;
reg sig_out_d;

always @ ( * ) begin
    sig_out_d <= sig_in_a & sig_in_b;
    sig_out_c <= sig_in_a | sig_in_b;
end
```

Combinational logic inference by always block (note event list or sensitivity list)

Coding




```
wire sig_in_a;
wire sig_in_b;
wire sig_out_c;
wire sig_out_d;
reg sig_reg_x;
reg sig_reg_y

always @ (posedge clk) begin
   sig_reg_x <= sig_in_a & sig_in_b;
   sig_reg_y <= sig_in_a | sig_in_b;
end

assign sig_out_c = sig_reg_x & sig_reg_y;
assign sig_out_d = sig_reg_x | sig_reg_y;
```


References

- IEEE Standard for Verilog Register Transfer Level Synthesis, IEEE Std. 1364.1, IEEE Computer Society, Dec. 18, 2002.
- IEEE Standard Verilog Language Reference Manual, IEEE Std. 1364, IEEE Computer Society, 2001.