Verilog Coding Style for Efficient Digital Design

Kapil Batra
STMicroelectronics Ltd., India
Kapil.batra@st.com

Mohammad Suhaib Husain
msuhaib@hotmail.com

Abstract: In this paper, we discuss efficient coding and design styles using verilog. This can be immensely helpful for any digital designer initiating designs. Here, we address different problems ranging from RTL-Gate Level simulation mismatch to race conditions in writing behavioral models. All these problems are accompanied by an example to have a better idea, and these can be taken care off if these coding guidelines are followed. Discussion of all the techniques is beyond the scope of this paper, however, here we try to cover a few of them.

1. Reading a variable before assigning - Simulation and Synthesis mismatch

Non-Blocking statements within the *always* block are executed sequentially. This becomes an issue when variables are used in the *always* block. Variables may be used in some conditional expression or on the right hand side of an assignment statement besides being assigned some value. Now if they are used prior to being assigned, a mismatch may occur in simulation and synthesis. For pre-synthesis simulation, that variable will contain the value assigned to it in the previous pass but it may not happen in gate level simulation.

This can be illustrated by means of a very simple example. In module *example1*, "Z" is declared as a register. It is being used in the right hand side of an assignment statement before being assigned a value in the next statement. Register "Z" will hold the value of the previous pass until the assignment statement for "Z" is executed, thus, output "Q" will be assigned a stale value in the current pass. The module *example1* was simulated using VCS and the results are as shown in **Fig.1a**. It is clearly understood by seeing the waveform that by change in any input, output "Q" gets the stale value assigned to register "Z" in the previous pass.

```
module example1 (A, B, Q);
input A, B;
output Q;
reg Z;
always @(A, B, Z)
begin
    Q = Z;
    Z = A | B;
end
endmodule
```

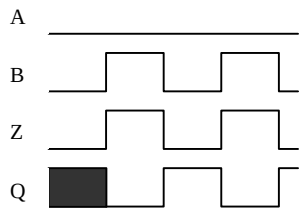

Fig. 1a

Now when this module example1 is synthesized, a simple OR gate is generated. And when we apply the same stimulus to the inputs "A" and "B", we get the waveform as shown in **Fig. 1b**, which is nothing but a simple OR gate.

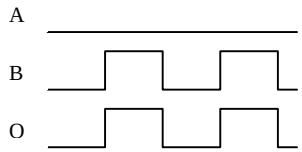

Fig. 1b

This results in a mismatch in pre and post-synthesis simulations that can be taken care off by correct ordering of assignment statements.

2. Using the sensitivity lists

Synthesis tools assume combinatorial logic from an *always* block without the keywords *posedge* or *negedge*. In a combinatorial *always* block, the resulting logic is derived from the statements inside the block only and not at all dependent on the sensitivity list. The synthesis tool will read and compare the sensitivity list with the statements in the *always* block, and report the part of the code that may cause a pre- and post-synthesis simulations mismatch. Putting in extra signals which are not at all used in the body of the *always* block, in the sensitivity list, would do nothing else but result in making the whole simulation slower as the *always* block will be evaluated more than it is required. On the other hand if the sensitivity list is not complete, the *always* block will not be evaluated for the signals missing in the sensitivity list though used in the *always* block. But the synthesis tool generates the logic as per the statements in the *always* block assuming a complete sensitivity list, whose functionality will be relatively different from the pre-synthesis simulation of the Verilog code having incomplete sensitivity lists.

In the module *test1*, the sensitivity list is complete; hence, both pre- and post-synthesis simulations match the synthesized logic as shown in **Fig. 2a**. In the next example, a module *test2* has an incomplete sensitivity list as it contains only one signal *"x"*. The *always* statement specifies that whenever an event occurs on *"x"*, the assignment statement is executed and *"z"* gets a value. If any event occurs on *"y"*, it will have no effect on the value of *"z"*. The synthesized logic for the module *test2* is a NAND gate as shown in **Fig. 2b,** whose output *"z"* changes if any of the two signals *"x"* or *"y"* change. As a result, there will be a mismatch between the pre- and the post- synthesis simulations.

```
module test1 (z,a,b,c,d);
input a, b, c, d;
output z;
wire a, b, c, d;
reg z, temp1, temp2;

always @(a or b or c or d)
begin
        temp1 = a | b;
        temp2 = c | d;
      z = temp1 ^ temp2;
end
endmodule

module test2 (z, x, y);
output z;
input x, y;
reg z;
wire x, y;
always @(x)
        z = ~(x & y);
endmodule
```

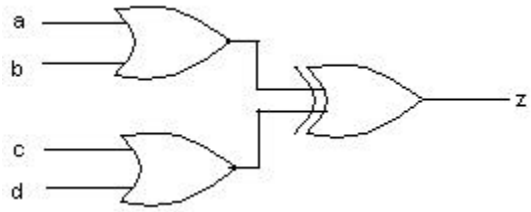

Fig. 2a

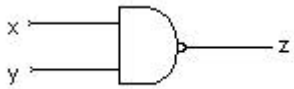

Fig. 2b

3. Full Case Parallel Case

The two directives '//synopsys full_case parallel_case' are the most over used and there is myth that the use of these make designs faster and smaller. This may not be true in all the cases, and instead they may cause a mismatch in the pre and post-synthesis simulation or may infer latches.

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.
Let us consider an example to illustrate synopsys full_case directive. In module *example2*, a full_case directive is used for a non-full case statement. Due to this synopsys assumes that inputs "A" and "B" will not be logic 0 at the same time and it does not optimizes the logic for that particular case. **Fig. 3** shows the inferred circuit for module *example2*.

```
module example2(A, B, C, D, Q)
input A, B, C, D;
output Q;
always @(A or B or C or D)
begin
        case(1'b1) // synopsys full_case parallel_case
        A : Q = C;
        B : Q = D;
        endcase
end
```

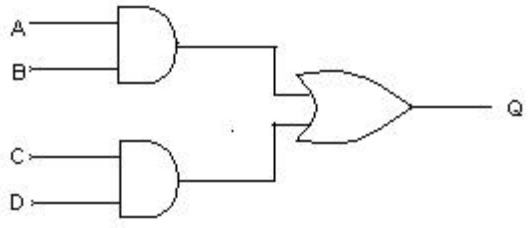

Fig. 3

In pre-synthesis simulation output "Q" will hold the value of the previous pass for "A"="B"=0. But in the post-synthesis simulation it will be logic 0 in this case.

When we remove the full_case directive in the same example, synopsys optimizes for the condition of "A"="B"=0 and infers a latch which is active when "A"="B"=0. This matches the pre-synthesis simulation results but a latch is not desirable most of the times.

So, unless we are sure that the particular condition would never occur ("A"="B"=0 in this example), a default case item must be used as shown below. This would ensure that there is no mismatch in pre and post-synthesis simulation results.

```
always @(A or B or C or D)
begin
        case(1'b1) // synopsys full_case parallel_case
        A : Q = C;
        B : Q = D;
        default : Q = 1'b0;
        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.

Let us consider an example to illustrate synopsys parallel_case directive. In module *example3*, if "parallel_case" directive is not used then the design will simulate like a priority encoder, both before and after synthesis. When a parallel_case directive is used, design will behave like a priority encoder in the pre-synthesis simulation but the synthesis tool will infer a non-priority encoder resulting in a mismatch in the pre and post-synthesis simulation.

```
module example3 (out0, out1, out2, in);

output out0, out1, out2;
input [2:0] in;
reg [2:0] out0, out1, out2;

always @(in)
begin
        {out2, out1, out0} = 3'b0;
        casez(in) // synopsys parallel_case
        3'b1?? : out0 = 1'b1;
        3'b?1? : out1 = 1'b1;
        3'b??1 : out2 = 1'b1;
        endcase
end

endmodule
```

4. Race Condition

Consider the circuit shown in **Fig. 4a**. A change in the output of flip-flop *FF1* may not meet the hold time requirement of the second flip-flop *FF2* . The condition may become even worse if there is a skew between the clock signals to the two flip-flops. If the clock signal to the flip-flop *FF2* lags that of flip-flop *FF1*, then it is also possible that the output of *FF2* may be same as that of *FF1*. In order to get rid of the problem, consider the circuit shown in **Fig.4b,** by adding a delay in the input path to the second flip-flop

FF2, the output of *FF2* will have sufficient time to change, well before the changes at the output of *FF1* travels to the flip-flop *FF2*.

Similarly a race condition may occur when two or more statements that are scheduled to execute in the same simulation time-step give different results when the order of statement execution is changed.

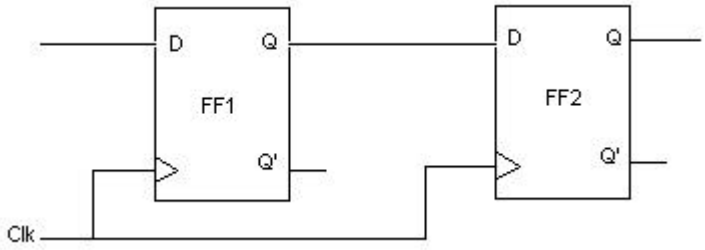

Fig. 4a

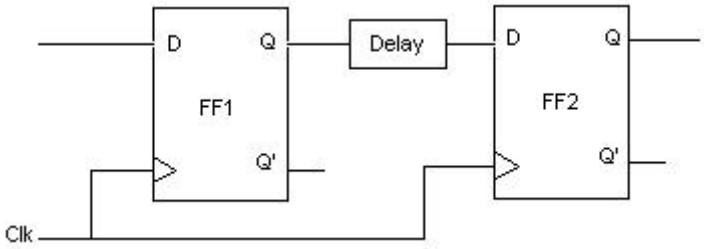

Fig. 4b.

The IEEE Verilog Standard defines which statements have guaranteed order of execution and which statements do not have a guaranteed order of execution. Let us consider an example to understand the problem and its solution.

```
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.

This uncertainty can be taken care off in different ways. In the solution shown below, by adding delay, right hand side is evaluated at the positive edge of clock and assignment takes place after the delay value. This makes sure that the output "Q" always gets the old value. But this solution is not generic as it explicitly uses delays.

```
always @(posedge clk)
begin
```

```
    Q = #1 A;       ……(1)
end
always @(posedge clk)
begin
    A = #1 B;       ……(2)
end
```

Instead of using delays, non-blocking statements can be used as shown below. By using non-blocking statements it is made sure that right hand side of the statement is evaluated and stored before assignments are done.

```
always @(posedge clk)
begin
    Q <= A;         ……(1)
end
always @(posedge clk)
begin
    A <= B;         ……(2)
end
```

5. Conclusion

In order to ensure the success of the design, a designer should be careful from the very start, each line of the verilog code must be understood completely and one should not wait for the bug report of the simulator. Keeping in mind the proper design techniques from the early stages of the design cycle will significantly reduce the time to debug.

References:

1) Bhasker J., A Verilog HDL Primer, second edition, BS Publications, Hyderabad, 1998.
2) Kurup P., Abbasi T., Logic Synthesis using Synopsys, second edition, Kluwer Academic Publication.