

GUIDELINES FOR SAFE SYNTHESIS OF WHILE LOOPS IN IMPLICIT STYLE VERILOG HDL

Shahriyar M. Rizvi

Faculty of Engineering, American International University-Bangladesh.
House# 42, Block-H, Road# 7/B, Banani, Dhaka-1213, Bangladesh.
E-mail: shahriyar@aiub.edu

Jerry J. Cupal

Department of Electrical and Computer Engineering, University of Wyoming.
Room 5032, Engineering Building, 1000 E. University Avenue, Laramie WY 82071, USA.
E-mail: jcupal@uwyo.edu

ABSTRACT

Implicit style Verilog Hardware Description Language allows Register Transfer Level designers to model a Finite State Machine (FSM) with a single *always* structure. Here, multiple clock-edge-sensitive event-control expressions (ECE) model the operation of the FSM. Typical FSM algorithms contain looping structures that assert outputs for multiple clock cycles. Implicit coding style models these loops by making the ECEs recursive through the use of *while* loops. Post-synthesis operation of these FSMs reveals that these *while* loops assert the outputs for extra clock cycles. This paper describes some safe implicit coding styles for these FSMs that will deliver proper post-synthesis behavior.

1. INTRODUCTION

In Verilog Hardware Description Language (Verilog HDL), Finite State Machines (FSM) have been traditionally modeled and implemented in explicit coding style. In this coding style, combinational and sequential components of the FSM are modeled with separate *always* blocks. The concurrent operation of these two *always* blocks allows the code to ensure the overall synchronous nature of the machine and yet maintain asynchronous nature of a Mealy output. This feature of explicit style Verilog HDL ensures that the FSM maintains its functionality both before and after synthesis [1]-[6].

Implicit coding style has generated interest among the Register Transfer Level (RTL) designers over the years for its relatively higher level of abstraction. This coding style uses a single *always* block to model both the combinational and sequential components of the FSM. The *always* block contains multiple clock-edge-sensitive event-control expressions (ECE) that describe the operation of the FSM. The ECEs make the design fully synchronous. In other words, here, a Mealy output cannot have an asynchronous behavior [1]-[6].

Typical FSM algorithms contain looping structures that requires the FSM to stay in a particular state and assert an output for multiple clock cycles. Implicit coding style does this by positioning the relevant ECEs inside *while* loops. Post-synthesis behavior of these *while* loops reveals that they assert the outputs for extra clock cycles. This paper describes techniques for writing safe implicit style codes that will deliver proper post-synthesis behavior so that they can be implemented into hardware without any concern for pre and post-synthesis simulation mismatches.

The example FSMs discussed in this paper were simulated, synthesized and placed and routed in Xilinx 1.5 Foundation Series design environment. FPGA Express (from Synopsys) was used as the synthesis tool. Designs were targeted to Xilinx 95144-TQ100 CPLD.

2. EXPLICIT AND IMPLICIT CODING STYLES

One can illustrate the difference between explicit and implicit coding styles by analyzing an FSM that has been modeled in both styles. The example FSM that will be discussed in this section, initially resides in an idle state, where it evaluates the value of an input signal called Pb. When Pb is high near an active clock-edge, it proceeds to generate a signal called Pulse for three clock cycles. After that it returns to its idle state and continues with this same set of operation.

To implement the design specification, the FSM has to control a counter and a comparator. This FSM controls the counter through Clr and Inc signals, which are used to clear and increment it. The comparator compares the counter output (Count) with two (2) and indicates to the FSM whether the Count signal is less than two or not through feeding a signal (Count_LT_2) back to the FSM.

2.1 Explicit Style

An FSM algorithm that implements the above design specification can be described through an Algorithmic State Machine (ASM) chart. Fig. 1 shows this ASM chart. This FSM asserts Clr, when Pb is high in state 0 (idle state) and Inc, when Count_LT_2 is high in state 1. Both Clr and Inc are Mealy outputs. Pulse, however, is a Moore output.

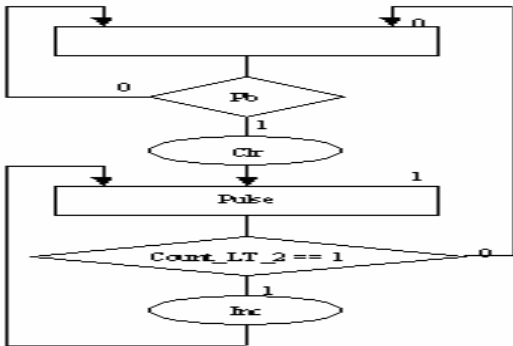


Fig. 1 ASM chart for example explicit FSM.

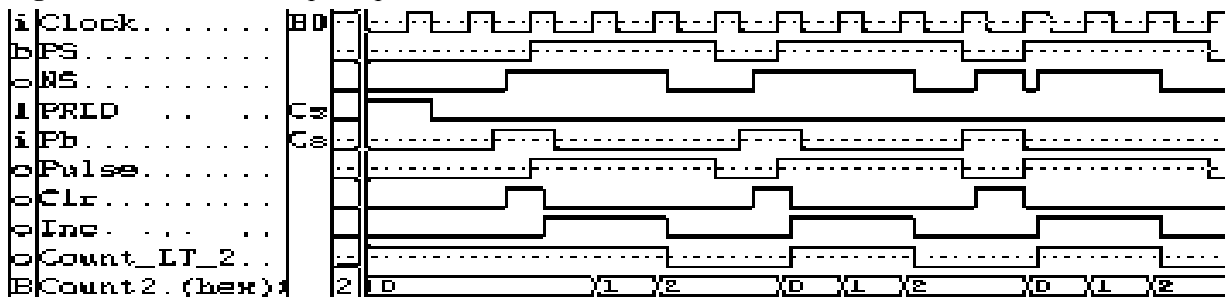


Fig. 3 Post-Route Simulation for the example explicit FSM

The explicit style code that corresponds to the ASM chart above is shown in Fig. 2^{*1}. It contains two *always* blocks. The one in the top contains a *case* structure to describe the combinational logic, which computes the values of next state (NS), Pulse, Clr and Inc whenever present state (PS), Pb or Count_LT_2 changes its value. The bottom *always* block contains a non-blocking statement that transfers the values of next state (NS) to present state (PS) at the rising edges of the clock.

```

always @(PS or Pb or Count_LT_2)
begin
Pulse = 0; Clr = 0; Inc = 0;
case (PS)
0: if (Pb == 1) begin Clr = 1; NS = 1; end
   else NS = 0;

1: begin Pulse = 1;
   if (Count_LT_2 == 1) begin Inc = 1; NS = 1; end
   else NS = 0; end

default: begin Pulse = 0; Clr = 0; Inc = 0; NS = 0; end
endcase
end

always @(posedge Clock)
PS <= NS;

```

Fig. 2 Explicit style Verilog HDL code for the example FSM.

The post-route simulation of the design shows that the synthesized and placed and routed circuit operates as per the code. For example, when Pb is high near the rising edge of Clock, Pulse is asserted for three clock cycles. Fig. 3 illustrates this.

2.2 Implicit Style

Implicit style FSM algorithms utilizes if structures and while loops to model conditions and looping structures. This can be seen in the ASM chart for the implicit version of the example FSM as displayed in Fig. 4.

^{*1} Module, wire and reg declarations are not shown in the figures that contain Verilog HDL codes for the example FSMs discussed in the paper. Only *always* blocks are shown. This is because the *always* blocks are the major focus of the paper and thus the remaining pieces of code was deemed unnecessary for the discussion by the authors.

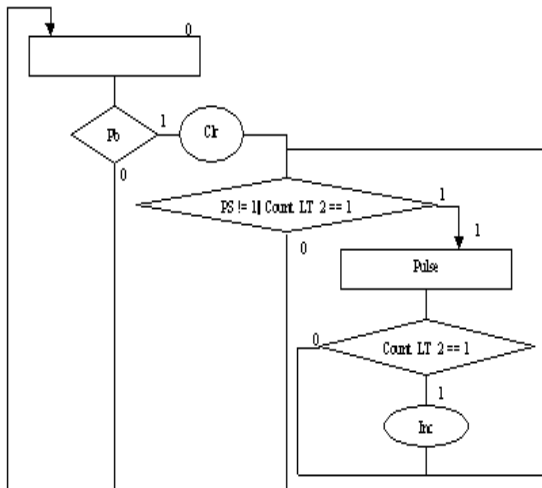


Fig. 4 ASM chart for the example implicit FSM.

One can see from Fig. 4 that, in contrast to the explicit version, the condition for asserting Pulse is evaluated (by the while loop) before entering state 1, where Pulse is actually asserted. Consequently, this input condition ($PS \neq 1 \parallel Count_LT_2$) is shared by multiple states---state 0 and state 1.

```

always
begin
@ (posedge Clock) PS <= 0; Pulse = 0; Clr = 0; Inc = 0;
if (Pb == 1)
begin
Clr = 1;
while (PS != 1 || Count_LT_2 == 1)
begin
@ (posedge Clock) PS <= 1; Pulse = 1; Clr = 0; Inc = 0;
if (Count_LT_2 == 1)
Inc = 1;
end
end
end
end

```

Fig. 5 Implicit style Verilog HDL code for the example FSM.

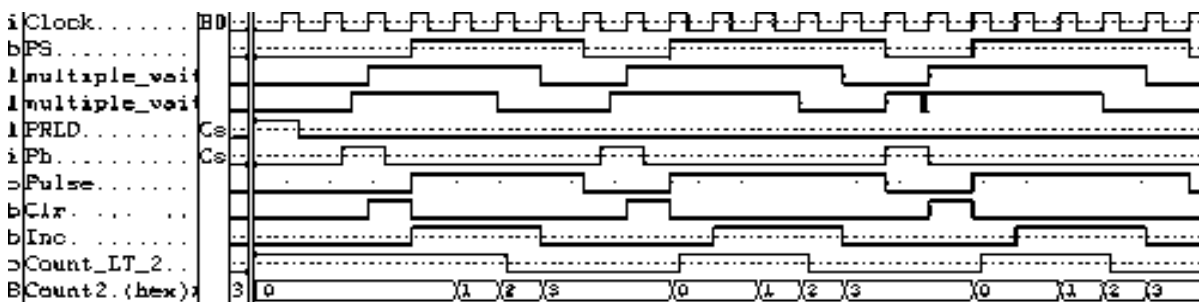


Fig. 6 Post-Route Simulation for the example implicit FSM

The implicit style code for modeling the ASM chart above, contains a single always block. It has two

clock-edge-sensitive event-control-expressions ($@(posedge Clock)$). The second ECE is positioned inside a while loop that makes it recursive. Fig. 5 shows this.

One can see from the post-route simulation, as shown in Fig. 6^{*2}, that Pb is high near three rising clock-edges and Pulse is scheduled to be asserted in all three. However, the duration of Pulse is not in accordance with the code. In the first such occurrence, Pulse is asserted for four clock cycles---one more than it should have. In the second and third such occurrences, it is asserted for five clock cycles---two more than it should have.

2.2.1 The First Type of Latency

One can notice that at these three clock edges the machine schedules transition from state 0 to state 1 and assertion of Pulse, since the condition $PS \neq 1$ is true and stable, which therefore, makes the whole condition $PS \neq 1 \parallel Count_LT_2$ true. However, the Inc signal, which is dependent only on Count_LT_2 (inside state 1), is asserted immediately only in the first case and is delayed by one clock cycle in the latter cases. However, just like the explicit version, Pulse is asserted one clock cycle longer than Inc. The latency of Inc signal occurs because the value of Count_LT_2, even though high, is not stable in the latter cases. This problem occurs because of the synchronous nature of the machine that requires inputs to be stable at the active clock-edges.

2.2.2 The Second Type of Latency

Regardless of the problem discussed above, Inc still should have been taken a low value when Count_LT_2 did the same. However, the simulation clearly shows that Inc is de-asserted one clock cycle later. This happens because the synthesis tool inserts flip-flops to all the outputs (as well as the output of the state register) including Inc. So in reality, the Inc

displayed in the simulation is not coming straight out of combinational logic. It is the “registered Inc” that

^{*2} The output and input signals of the state register are shown in Fig.6 and Fig. 8, below PS signal. The input of this register is driven by the combinational logic. The output of this register is registered again to generate PS.

is coming out of the FSM design. In the second and third case, Inc is not only “late” by one clock cycle; it also gets de-asserted one clock cycle later. As a consequence, Pulse is asserted for two extra clock cycles. This behavior is neither expected nor desired for implementing this FSM.

3. SAFE SYNTHESIS OF WHILE LOOPS

3.1 Post-Synthesis Processing

The improper post-route behavior of the implicit FSM due to the second type of latency occurs because of the output flip-flops. It is possible to remove these flip-flops by editing the EDIF files coming out of synthesis. This process will yield a post-route behavior that is identical to that of the explicit version [6]. However, it will remove the “implicitness” of the design. For example, the outputs will not have a guaranteed synchronous nature.

3.2 Safe Implicit Coding Styles

It is possible to address the incorrect post-synthesis behavior of the implicit FSM and still retain the implicit nature of the code through slight modification of the algorithm and the coding style. In this coding style, there is no guaranteed idle state. The FSM can enter any of the two states at the first rising edge, as shown in Fig. 7.

```

always
begin
  @(posedge Clock);
  Clr = 1;
  if (Pb == 0)
  begin
    PS <= 0; Pulse = 0; Inc = 0;
  end
  else
  begin
    while (PS != 1 || Count_LT_2 == 1)
    begin
      PS <= 1; Inc = 1;
      @(posedge Clock);
      Clr = 0; Pulse = 1;
    end
    PS <= 0; Pulse = 0; Clr = 1; Inc = 0;
  end
end

```

Fig. 7 “Safe” implicit style Verilog HDL code for the example FSM.

In this version of implicit code, the FSM clears the counter whether or not it enters state 0 or state 1 at the first rising edge. In state 1, both Clr and Inc signals are asserted. Even if the machine enters state 1 at the first rising edge, the counter will be cleared. This is because Clr has been assigned higher priority over Inc. This priority level is maintained in the code for the counter for all the FSM designs. Unlike the traditional implicit code, here, transition from state 0 to state 1 occurs at the very rising edge where Pb is high---it is not scheduled for the next rising edge. Pulse, however, is asserted one clock cycle later to allow the Clr signal to clear the counter before it is asserted. This is also done to avoid having unstable values for Count_LT_2 at rising edges where Inc is expected to be high. Inc still takes a low value one clock cycle after Count_LT_2 does, but the initial latency of Pulse ensures that they are both de-asserted together, but only after Pulse has been asserted for three clock cycles. This coding style essentially uses Clr and Inc signals as Moore signals.

This implicit code always asserts Pulse for three clock cycles. Thus it removes the problem encountered in traditional implicit code. Fig. 8 illustrates this^{*3}.

The key to safe coding style is essentially converting while loops to “do-while” type structures. In other words, the while loops should be located inside the ECEs. Therefore, input conditions are evaluated after entering a rising edge, not before it. The designer should, however, realize that it is not possible to model a Mealy output in this coding style.

4. CONCLUSION

Implicit coding style contains some elegant features that can be very attractive and useful for an RTL designer. It maintains a purely behavioral model of the FSM and thus frees the designer from devoting time to the details of the hardware architecture, like partitioning the design into combinational and sequential components. Stand-alone and recursive

*3 The PRLD signal, which is shown above Pb signals in Fig. 8, as well as Fig. 6 and Fig. 3, is a global asynchronous reset signal for Xilinx CPLDs. It resets all the flip-flops to zero-state at the beginning of the post-route simulation. Xilinx EDA tools also reset the flip-flops in actual hardware, after an HDL design is downloaded into a CPLD.

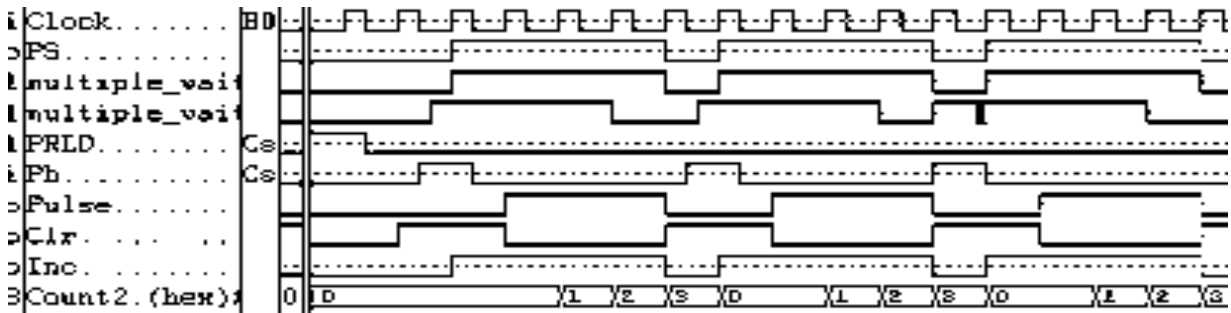


Fig. 8 Post-Route Simulation for the “safe” implicit FSM

ECEs (containing *while* loops) display the flow of the algorithm in a nice sequential manner. These clock-edge-sensitive expressions ensure that the outputs are all synchronous signals and thus, free from glitches.

The “safe” coding style that was described in this paper removes the problem of incorrect post-synthesis behavior of FSMs that are modeled in regular implicit coding style. The “safe” code is not a perfect equivalent of the initial implicit code. For example, Clr and Inc do not retain their Mealy characteristics. However, the “safe” coding style satisfies the design specifications, does not require any post-synthesis processing of netlist or any modification of the datapath components (counter and comparator). This paper shows that an implicit style Verilog HDL model of an FSM is not only meant for maintaining an elegant and abstract model of an FSM, but is fully suitable for actual hardware implementation.

REFERENCES

- [1] M. G. Arnold, Verilog Digital Computer Design: Algorithms to Hardware, Prentice Hall, 1999.
- [2] M. D. Ciletti, Modeling, Synthesis and Rapid Prototyping with Verilog HDL, Prentice Hall, 1999.
- [3] D. E. Thomas and P. R. Moorby, The Verilog Hardware Description Language (fifth edition), Kluwer Academic Publishers, 2002.
- [4] J. M. Lee, Verilog Quickstart; A Practical Guide to Simulation and Synthesis in Verilog (third edition), Kluwer Academic Publishers, 2002.
- [5] M. G. Arnold and N. J. Sample, “Guidelines for Safe Simulation and Synthesis of Implicit Style Verilog”, in Proc. International Verilog HDL Conference 1998, pp. 55-66, March 1998.
- [6] S. M. Rizvi and J. J. Cupal, “A Methodology to Remove Unwanted Delays in Outputs and Pre and Post-Synthesis Simulation Mismatches in Implicit State Machines”, in Notes of EDP Workshop 2004, pp. S5.3, April 2004.