A Methodology to Remove Unwanted Delays in Outputs and Pre and Post-Synthesis Simulation Mismatches in Implicit State Machines

Shahriyar M. Rizvi

American International University-Bangladesh, Dhaka, Dhaka-1213, Bangladesh,
shahriyar@aiub.edu

Jerry J. Cupal

University of Wyoming, Laramie, WY 82071, USA, jcupal@uwyo.edu

Abstract

When a Finite State Machine (FSM) is modeled in implicit style Verilog HDL, values of inputs just after active edges of the system clock determine the next states of the machine. Synthesis tools interpret this feature by inserting D-type flip-flops at the present state signals and the outputs. This interpretation delays these signals by one clock cycle as compared to an FSM synthesized from explicit style Verilog HDL code. In synchronous design, this delay is both unwanted and unnecessary. In certain cases, this delay creates pre and post-synthesis simulation mismatches. A technique has been developed to remove these flip-flops after synthesis, and prior to place and route. This allows designers to produce implicit FSMs that deliver correct functionality both before and after synthesis and operate with the same speed as an explicit FSM. Since implicit style has a higher level of abstraction, this modification process can provide a designer a more abstract and perhaps, a better method to model and implement an FSM.

1 Introduction

In digital hardware design, designers often partition hardware into FSMs and datapath components. It is possible to write synthesizable Verilog HDL code for an FSM in two ways: explicit style and implicit style [1], [2], [3], [4], [5].

Explicit style of coding models the operation of an FSM as the combined operation of its combinational and sequential components. That is why it requires modeling these two components separately. In this coding style, and in actual hardware, values of inputs just prior to active edges of the system clock determine the next states of the machine. Here, a Mealy output can glitch. These features allow explicit style to accurately model the architecture and the operation of the synthesized FSM.

In contrast, implicit style closely models the behavior of an FSM. Here, there is no separation between combinational and sequential logic. In fact, implicit code hides the details of the hardware architecture from the designer's view [1], [2], [3], [4], [5].

In implicit style, values of inputs just after active edges of the system clock determine the next states of the machine. Synthesis tools interpret this feature through adding D type flip-flops to present state signals and outputs. This delay is neither necessary nor desired for realizing many design specifications. Since all the outputs are registered, a Mealy output cannot glitch, in this case. Furthermore, when an implicit FSM has to control certain datapath elements (i.e. counter), this delay can cause pre and post-synthesis simulation mismatches.

A technique has been developed to remove these output flip-flops after synthesis, and prior to place and route, in Xilinx 3.1i ISE design environment. This paper gives the details of this modification process. Basically it consists of identifying the flip-flops in a text file coming out of the synthesis tool (FPGA Express) and replacing them with simple buffers. Post-route simulations show that a "modified" implicit FSM preserves the correct functionality even after synthesis and operates just like an explicit FSM---without the delays in the outputs.

2 The Behavior of an Example Mealy FSM Coded In Explicit Style

A Design Specifications and Algorithm

A simple FSM can be designed to illustrate the behavior of explicit style Verilog HDL. This FSM waits for an external input Pb to go high in an idle "state" and when it does, the FSM proceeds to output Red, Yellow and Green signals for six, one and six clock cycles respectively. It returns to the "idle" state and repeats the same sequence forever. Figure 1 describes the algorithm in an Algorithmic State Machine (ASM) chart.

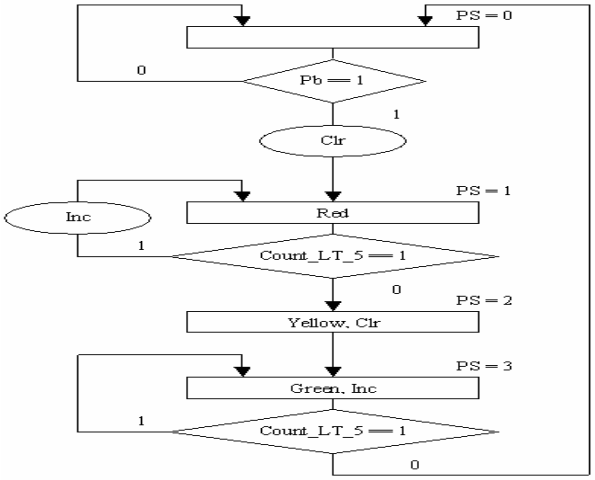

Figure 1: ASM chart for the explicit FSM

To realize the design specifications, this FSM has to control a counter and a comparator in its datapath. The FSM can control this counter through output signals (Clr and Inc) that can be used to clear and increment the counter. The comparator can output a signal (Count_LT_5) that can be fed back to the FSM to indicate to it whether the value of count is less than five or not. In this case a low value in Count_LT_5 signal signifies that the value of count is less than five.

B Verilog HDL Description, Logic Synthesis and Timing Verification

Explicit style Verilog HDL, as shown in Figure 2, uses separate *always* blocks to model combinational and sequential logic. The first *always* block is level-sensitive. It models the combinational logic that generates next state signals (NS) and outputs (Red, Yellow, Green, Clr and Inc). This logic is modeled with a *case statement*. Inside this *case statement*, conditional state transitions are handled with *if-else* structures. These *if-else* structures are also utilized to assert Mealy outputs. The second *always* block is edge-sensitive. It models the sequential logic that contains a *present state register*, which performs state transitions [1], [5].

The two *always* blocks, running in parallel, ensure that explicit style code models the hardware exactly. However, this feature makes the code harder to read and maintain. Here, the designer always has to take into account the operation of both these blocks simultaneously to understand the operation of the FSM. This makes it harder to follow the algorithm. Furthermore, when the number of states is large, the *case statement*, which has a "go-to" structure, can become very long and clumsy and thus even harder to read and maintain.

```
module MixedExplicit_Controller ( PRLD, Pb, Clock, Count_LT_5, Red, Yellow, Green,
                                  Clr, Inc, PS);
//input, output, wire, reg declarations

        always @ (Pb or PS or Count_LT_5)
        begin
        Red = 0; Yellow = 0; Green = 0; Clr = 0; Inc = 0;

        case (PS)
        0: if (Pb == 0) NS =0;
           else begin Clr =1; NS =1; end

        1: begin Red =1;
           if (Count_LT_5 == 1) begin Inc = 1; NS = 1; end
           else NS = 2; end

        2: begin Yellow =1; Clr = 1; NS = 3; end

        3: begin Green = 1; Inc = 1;
           if (Count_LT_5 == 1) NS = 3;
           else NS = 0; end

        default: begin Red = 0; Yellow = 0; Green = 0; Clr = 0; Inc = 0; NS = 0; end
        endcase

        end

        always @ (posedge Clock)
        PS <= NS;

endmodule
```

Figure 2: Verilog HDL code for the explicit FSM

The behavior of the FSM, as shown in the post-route timing simulation (Figure 3), is what a designer would expect. For example, values of present state (PS) and the inputs (Pb, Count_LT_5) just prior to the positive edges of the system clock (Clock) determine the next state (NS) of the machine. The values of NS are transferred to PS at every positive edge of the Clock by the *present state register*. Here, Mealy Clr signal (in state 0) and the Mealy Inc signal (in state 1) exactly follow the inputs they are sensitive to (Pb and Count_LT_5).

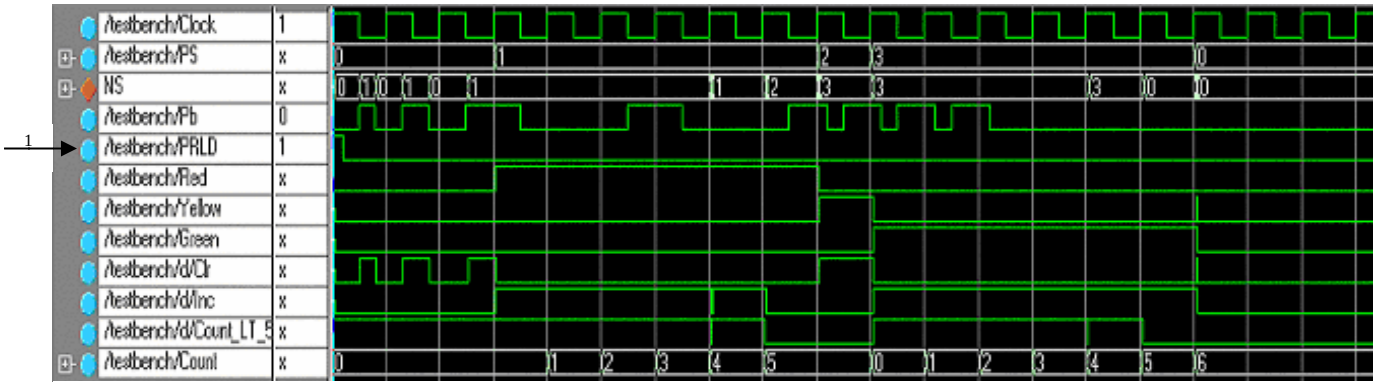

Figure 3: Post-Route Timing Simulation for the explicit FSM

[1]PRLD is an asynchronous global reset signal meant for initializing flip-flops for designs that are targeted to Xilinx CPLDs.

The FSM inferred two flip-flops and thirty-six gates in the logic synthesis process. The flip-flop count was indeed expected for an FSM with four states.

3 The Behavior of an Example Mealy FSM Coded In implicit style

A Design Specifications and Algorithm

A simple FSM can be designed that is coded in implicit style Verilog HDL and satisfies the same design specifications as the example explicit FSM. The implicit algorithm for this FSM, as shown in figure 4, models its operation as an "evolution of activity within a cyclic behavior" [2].

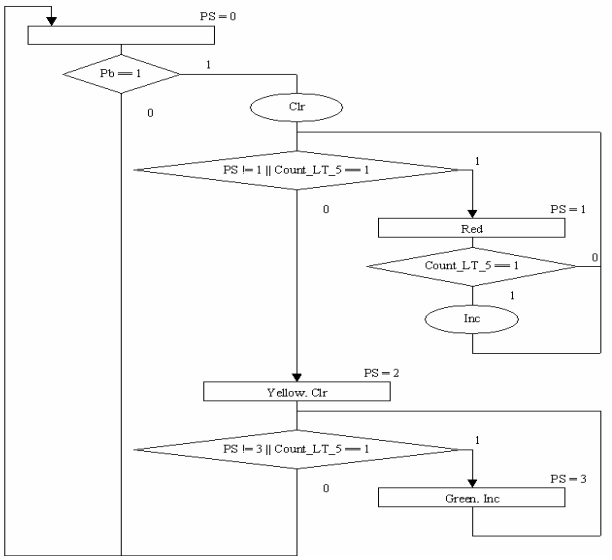

Figure 4: ASM chart for the implicit FSM

In implicit style, the designer models the conditional state transitions---which are inherent in most FSM algorithms---with *if* structures and *while* loops. It utilizes top-testing *while* loops, to model the bottom-testing "do-while" looping structures, which in this case describe the machine's operation in state 1 and state 3. Here, unlike explicit style, conditions are evaluated before entering a state rather than just after it. However, properly written conditions for these *while* loops can preserve the "do-while-ness" of these transitions (i.e. the FSM staying in a state at least for one clock cycle). For example the first while loop evaluates a condition (PS != 1 || Count_LT_5 == 1) that is true in state 0 at least once [1], [5].

The "do-while" structure that describes the operation of the explicit version in state 0, is realized here by an *if* structure. By utilizing the cyclic nature of algorithms, this *if* structure is made recursive like a *while* loop. For example, when Pb has a low value in state 0, the algorithm forces the machine to exit the algorithm and enter state 0 upon reentry. This is exactly what happens when the machine has to transition from state 3 to state 0 (when the condition for the second *while* loop becomes false) [1], [5].

In implicit style, Mealy outputs can be handled inside *if* or *if-else* structures. Here, the Mealy Clr signal in state 0 and Mealy Inc signals in state 1 are asserted within *if* structures [1], [5].

B Verilog HDL Description, Logic Synthesis and Timing Verification

The Verilog HDL description for the implicit FSM as shown in Figure 5, illustrates that it follows the above algorithm exactly. In implicit style, there is a single edge-sensitive *always* block. It contains multiple event control expressions (i.e. *@(posedge Clock).* They describe the progression of states in a nice sequential manner. One can easily see the flow of the algorithm. There are no references to next states in the code. The *always* block models both combinational and sequential logic. Here, the designer can avoid concentrating on details of the hardware architecture. All these features make an implicit code easier to read and maintain [1], [2], [5].

```
module MixedImplicit_Controller ( PRLD, Pb, Clock, Count_LT_5, Red, Yellow, Green,
                Clr, Inc, PS);
//input, output, wire, reg declarations

        always
        begin

        @ (posedge Clock) #1 PS <= 0;
        Red = 0; Yellow = 0; Green = 0; Clr = 0; Inc = 0;

                if (Pb == 1)
                begin
                        Clr = 1;
                        while (PS != 1  ||  Count_LT_5 == 1)
                        begin
                                @ (posedge Clock) #1 PS <= 1;
                                Red = 1; Yellow = 0; Green = 0; Clr = 0; Inc = 0;
                                if (Count_LT_5 == 1)
                                        Inc = 1;
                        end

                        @ (posedge Clock) #1 PS < = 2;
                        Red = 0; Yellow = 1; Green = 0; Clr = 1; Inc = 0;

                        while (PS != 3  ||  Count_LT_5 == 1)
                        begin
                                @ (posedge Clock) #1 PS <= 3;
                                Red = 0; Yellow = 0; Green = 1; Clr = 0; Inc = 1;
                        end
                end

        end

endmodule
```

Figure 5: Verilog HDL code for the implicit FSM

Despite the nice features of implicit style, the nature of the code generates a one-clock cycle delay in the outputs (Red, Yellow, Green, Clr, Inc) and the present state signals (PS) as compared to explicit style. The intent of implicit style is to evaluate the values of inputs just after positive edges of the Clock for determining the next states of the machine. As a consequence, the machine transitions from state 0 to state 1 one-clock-cycle later than its explicit version. This is evident in the functional simulation that is shown in figure 6. Here, both Moore

and Mealy outputs are delayed by one clock cycle. In addition to that, here, a Mealy output (i.e. Clr in state 0) cannot follow the input it is sensitive to if the input is an asynchronous signal (like Pb) and it glitches.

The FSM inferred nine flip-flops and fifty-three gates in the logic synthesis process[2]. The flip-flop count can be accounted for by analyzing the architecture of the synthesized hardware. Here, two flip-flops are inferred for the *multiple wait state register*, two for the *present state register* and five for the *output register*. In implicit style the *multiple wait state register* is the register that performs state transitions. It transfers the value of next state signals (NS) to multiple wait state signals (MWS) at every positive edge of the Clock. The multiple wait state signals (MWS) are registered by the *present state register* to generate the present state signals (PS).

Here the signals coming out of the combinational logic are referred to as NRed, NYellow, NGreen, NClr and NInc for convenience since the names assigned by the synthesis tool are very long. They are registered by the *output register* to generate the actual outputs (Red, Yellow, Green, Clr, Inc).

The reader should note that the operation of the synthesized hardware is realized by the collective operation of the combinational logic and the above-mentioned registers. However, this architecture of the synthesized hardware or the operation of its various components is implicit in the code.

The post-route timing simulation showed that the functionality of the FSM did not remain intact. For example, the FSM stayed in state 1 and state 3 for one extra clock cycle, as shown in Figure 7. This happens because the counter is not sensitive to NInc; it is sensitive to Inc, which is always one-clock-cycle "late". Therefore, when Count_LT_5 takes a low value, say in state 1, NInc is de-asserted immediately but it cannot instantly stop the counter from incrementing. The Inc signal is de-asserted one clock cycle later than NInc. By that time, the counter has counted up to six, when it should have stopped at five.

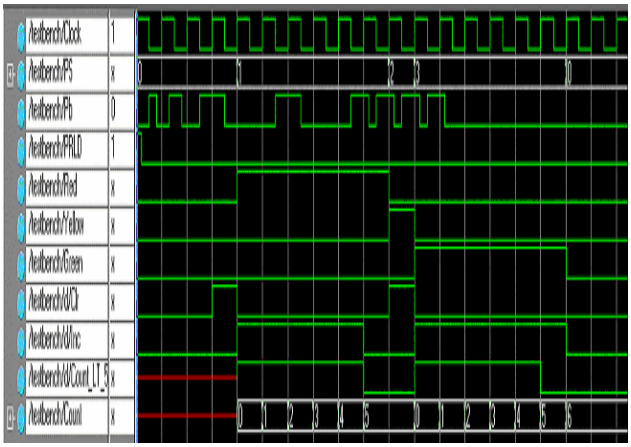

Figure 6: Functional Simulation for the explicit FSM

4 A Process To Remove the Delay Problem In Implicit FSMs

A Top-Down Design Flow in Xilinx 3.1i ISE EDA Tool

The FSM designs discussed in the paper were verified and implemented with Xilinx 3.1i ISE Electronic Design Automation (EDA) tool. This tool uses FPGA Express logic synthesis engine from Synopsys and ModelSim XE 5.3d simulator from Model Technology.

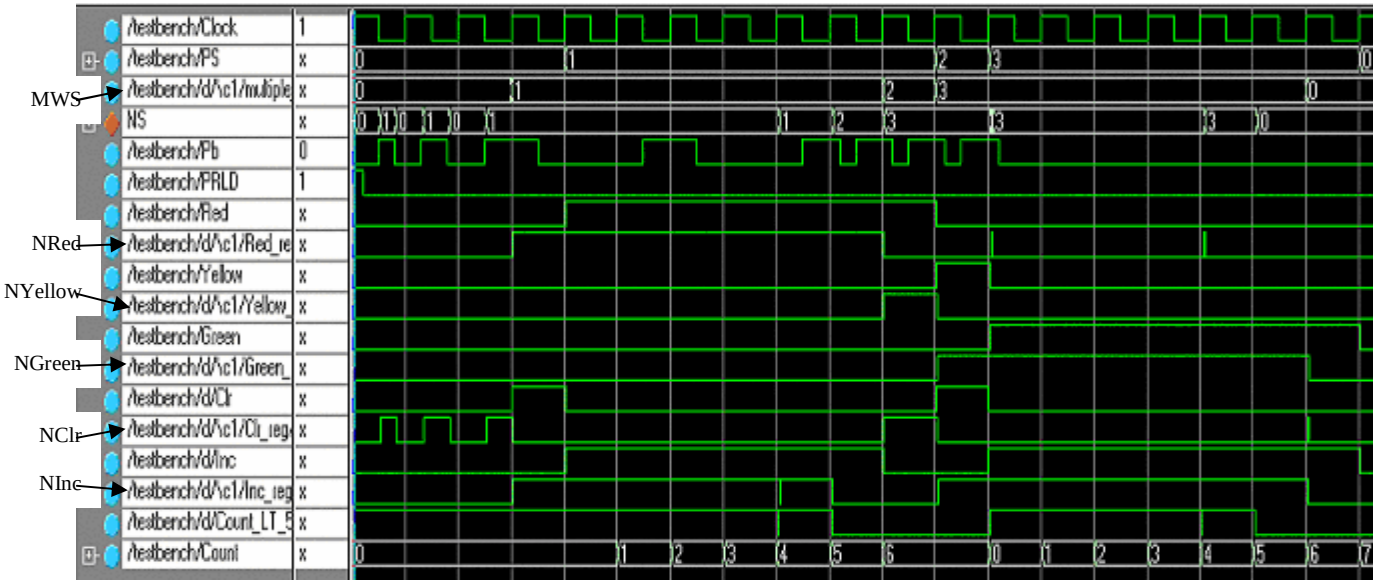

Figure7: Post-Route Timing Simulation for the explicit FSM

[2]Before synthesizing an implicit code, the *#1* delays contained in the event control expressions [@(posedge Clock)..] have to be commented out [5].

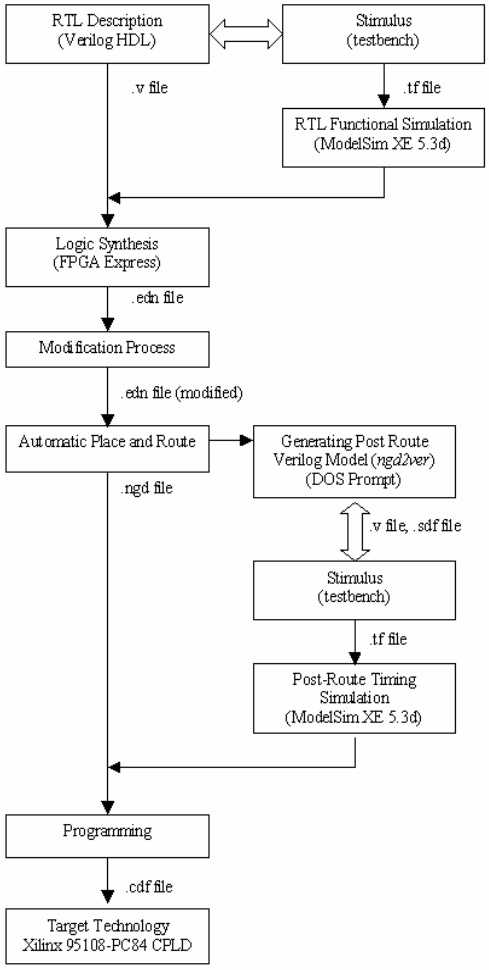

Figure 8: Top-down Design Flow in Xilinx 3.1i ISE Design Environment

In a typical design project in Xilinx 3.1i ISE design environment, the RTL description of a design and its test vectors are contained in files with *.v* and *.tf* extensions respectively. A functional simulation can be performed on this *.v* file. The logic synthesis process takes this *.v* file and outputs a text file with *.edn* extension. This file serves as the input to the place and route tool, which outputs a binary file with *.ngd* extension. At this point the designer has to perform a process called *ngd2ver* in a DOS prompt to generate post-route model of the design. This process takes in the *.ngd* file and outputs a *.v* file containing the post-route Verilog model, and a file with *.sdf* extension that contains the timing delay information. A post-route timing simulation can be performed on this new *.v* file in a new design project. Finally the programming process takes the *.ngd* file, residing in the original design project, and creates a programming file with *.cdf* extension. Now the programming tool can download the design into a target technology. This top-down design flow is shown in figure 8.

B The Modification Process for Implicit FSMs

A modification process has been developed to removes the delay problem in implicit FSMs. It basically involves removing the flip-flops in the *output register* and the *present state register* by modifying the netlist (*.edn* file) coming out of the logic synthesis process. The behavior of this "modified" implicit design can be verified by post-route timing simulations. The modification process consists of some simple steps as shown in figure 9.

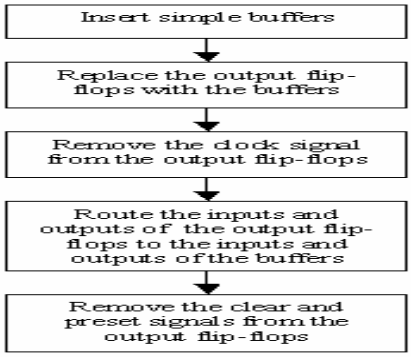

Figure 9: Flow Chart for the Modification Process for Implicit FSMs

I Step One

First, the designer has to add the library cell of a simple buffer called *BUF*[3]. To do that, the designer has to detect the library cell of an output buffer called *OBUF* in the netlist (.edn file), copy and paste it below the *OBUF* cell and change the cell name to *BUF*.

II Step Two

The next step is to identify the flip-flops in the *output register* and the *present state register*. In the example FSM, the designer would find these flip-flops listed as *Red_reg*, *Yellow_reg*, *Green_reg*, *Clr_reg*, *Inc_reg*, *PS_reg_1 "PS_reg<1>"* and *PS_reg_0 "PS_reg<0>"*. He can rename them as *Red_buf*, *Yellow_buf* and so on. In each case, the flip-flop type is listed beside the cell reference (*cellRef*). In this case, the cell reference would be *FDCP*. The designer should change the cell reference from *FDCP* to *BUF*[3]. Here, *FDCP* refers to a D type flip-flop.

III Step Three

In this step the designer should identify the wires carrying the Clock signal to the output flip-flops mentioned above, and remove them. Typically these wires would be described under the lines that read *(net Clock..(joined..(portRef Clock)*.

IV Step Four

The designer should route the input and output signals of the output flip-flops to the input and output signals of the buffers that replaced them. So the *D* and *Q* ports of the flip-flops should now be the *I* and *O* ports of the buffers. For example, the Q port of the flip-flop driving the Red signal should read *(net Red..(portRef O)* instead of *(net Red..(portRef Q)*, after this step.

[3] It was noticed that, in CPLD implementations, the netlist (*.edn* file) would typically have *OBUF* cells since the synthesis tool uses *OBUF* type buffers to drive the outputs from the *FDCP* type flip-flops.

V Step Five

Finally, the wires carrying clear (CLR) and preset (PRE) signals to the output flip-flops should be identified, and then removed. For example, the netlist fragment that describes the wires carrying clear and preset signals to the *Red_reg* flip-flop reads *(net N61...joined..(portRef..CLR..(instanceRefRed_reg)..(port RefPRE….(instanceRef Red_reg)).* So the modified netlist should not have it.

C The Results of the Modification Process

The post-route simulation[4] reveals that there are no delays in the present state signals or the outputs. There are glitches in some signals but they have no effect on the overall operation of the machine. MWS, NRed, NYellow, NGreen, NClr and NInc signals are identical to PS, Red, Yellow, Green, Clr and Inc signals respectively. Mealy Clr (in state 0) and Mealy Inc signals (in state 1) are subject to glitches now. Figure 10 illustrates this.

5 Conclusion

The modification process described here removes some of the major drawbacks in implicit style. First, this process removes the delays in the present state signals and the outputs. Second, it maintains the functionality of an implicit design after synthesis. Third, it removes the excess sequential hardware that is normally generated when implicit designs are synthesized.

When a designer applies this methodology to implicit FSMs, the difference between explicit and implicit styles is essentially eliminated after synthesis. The designer now has total freedom in choosing a particular style for modeling and implementing an FSM.

A comparative analysis of some of the inherent features of these two coding styles reveals that implicit style has many advantages over explicit style. The most important advantage of implicit style lies in its higher level of abstraction, which primarily manifests itself in the implicitness of the next states, the capability of modeling both combinational and sequential logic in the same *always* block and the use of high-level looping structures to realize an algorithm. Since the code reflects, primarily, the behavior of the FSM, it allows the designer to concentrate more on algorithm---where typically real design problems lie--than on architectural details. It is the belief of the authors that this modification process is an important step in making implicit style the preferred FSM coding style of the future.

6 REFERENCES

[1] Arnold, M. G. Verilog Digital Computer Design: Algorithms to Hardware. Upper Saddle River NJ, Prentice Hall, 1999, 7-59,100-106,177-196.

[2] Ciletti, M. D. Modeling, Synthesis and Rapid Prototyping with Verilog HDL. Upper Saddle River NJ, Prentice Hall, 1999, 238-260, 378-409, 463-464.

[3] Thomas, D. E., and Moorby, P. R. The Verilog Hardware Description Language (fifth edition). Boston MA, Kluwer Academic Publishers, 2002, 53-57, 82-88, 195-208.

[4] Lee, J. M. Verilog Quickstart: A Practical Guide to Simulation and Synthesis in Verilog (third edition). Boston MA, Kluwer Academic Publishers, 2002, 116-117, 169-186.

[5] Arnold, M. G. and Sample, N. J. (eds.). Guidelines for Safe Simulation and Synthesis of Implicit Style Verilog. Proc. 7th International Verilog HDL Conference (Santa Clara, California, March 15-17, 1998), 55-66.

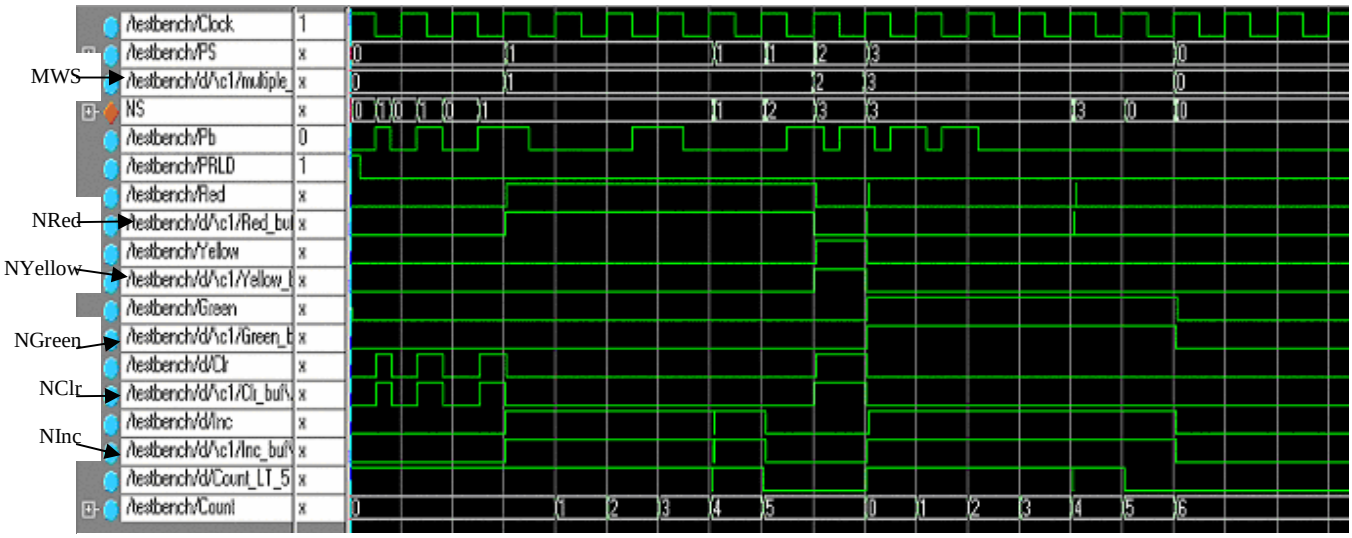

Figure 10: Post-Route Timing Simulation for the Modified Implicit FSM

[4] The authors captured the operation of all the FSMs discussed here, after implementing them in Xilinx 95108-PC84 CPLDs, with a logic analyzer. The snaps confirmed the results and can be provided for the reviewers if needed.