# A New Paradigm for Synchronous State Machine Design in Verilog

Randy Nuss



## *Introduction*

Synchronous State Machines are one of the most common building blocks in modern digital systems. They handle everything from communications handshaking protocols to microprocessor bus wait state insertion. State machines operate at hardware speeds where software cannot compete. All too often engineers take an ad-hoc approach to state machine design. Subtle and frustrating problems can arise from poorly designed state machines which typically manifest themselves as intermittent operation or lockup. Other problems such as glitches may appear in the outputs causing headaches for customers and service personnel long after a product is in production.

This article will first describe the basic architectures for synchronous state machines, then describe a method of State Machine implementation which leads to glitchless, minimum-delay operation. The Verilog Hardware Description Language will be utilized.

## *State Machine Architectures*

There are two generally accepted architectures for Synchronous State Machines. The first type considered is a state machine in which the outputs depend only on the current state. This is commonly known as a Moore machine. In the second type, the outputs depend on both the current state and the input variables. This is known as a Mealy Machine.

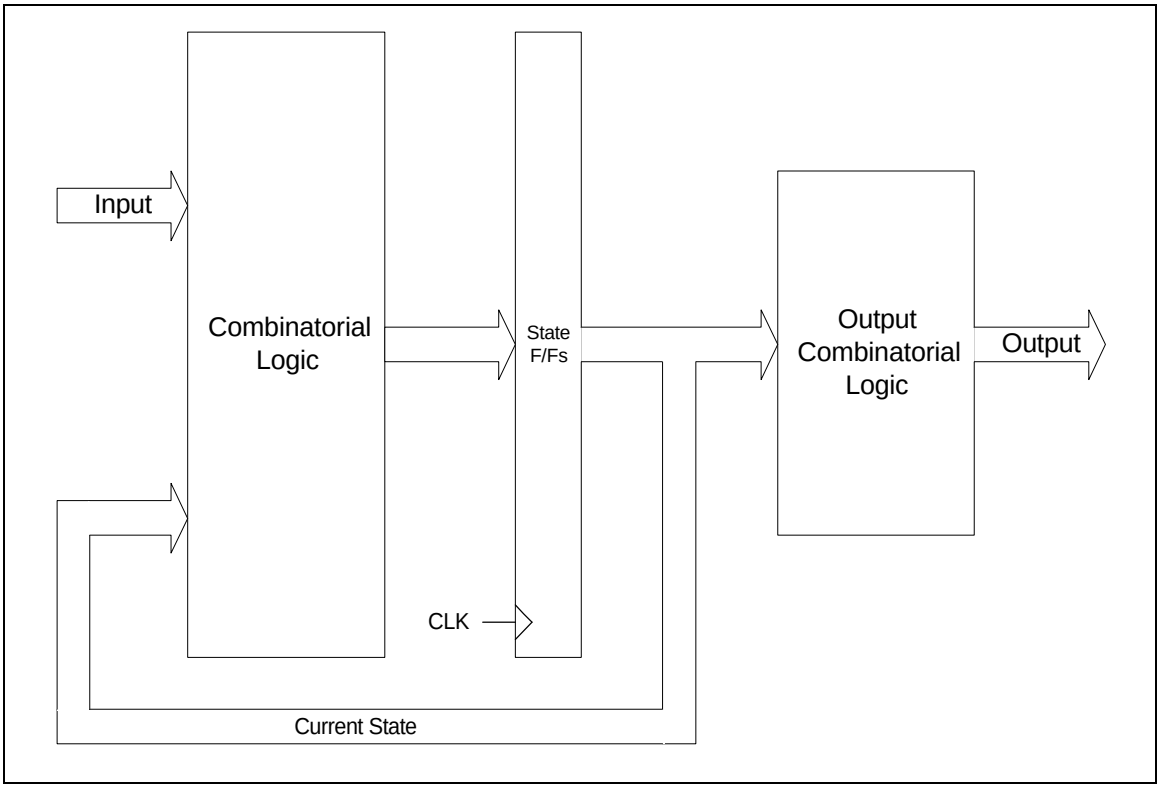


*Figure 1: Moore State Machine*

## Moore Machine

This is the simplest of the two state machine types. The outputs are combinatorial signals based solely on the current state. Unfortunately, this can lead to glitches on the output signals which can cause erratic operation of circuitry driven by the state machine.

The glitches are due to unequal delays in the Clock to Q path of the flip-flops which make up the state bits as well as unequal propagation delays in the combinatorial logic which derives the output from the state bits. Moore machine implementations are generally simpler than Mealy machines and may allow somewhat higher clock rates than a Mealy machine.

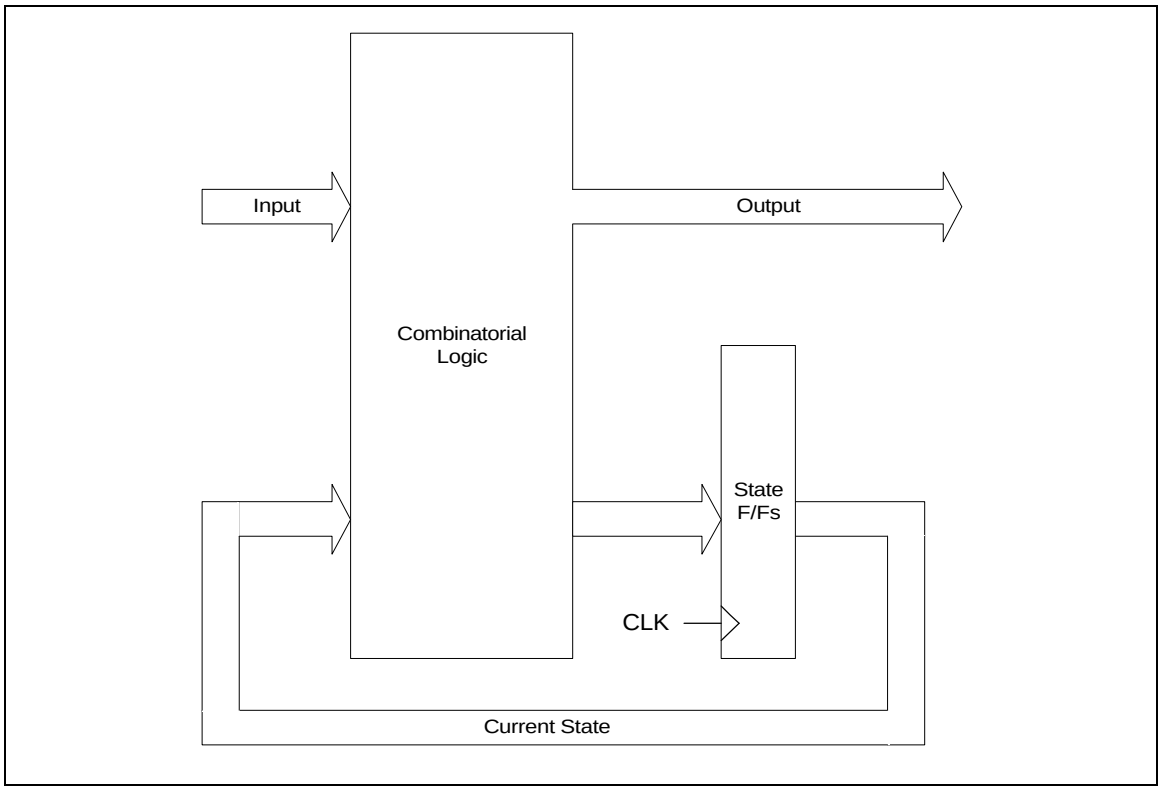


*Figure 2: Mealy State Machine*

## Mealy Machine

In a Mealy machine, the outputs are a function of not only the current state, but also the inputs. This implementation can lead to timing problems since the output timing is not simply a function of the clock, but of the input timing as well. For this reason, the Mealy architecture is generally a poor choice for synchronous devices or designs.

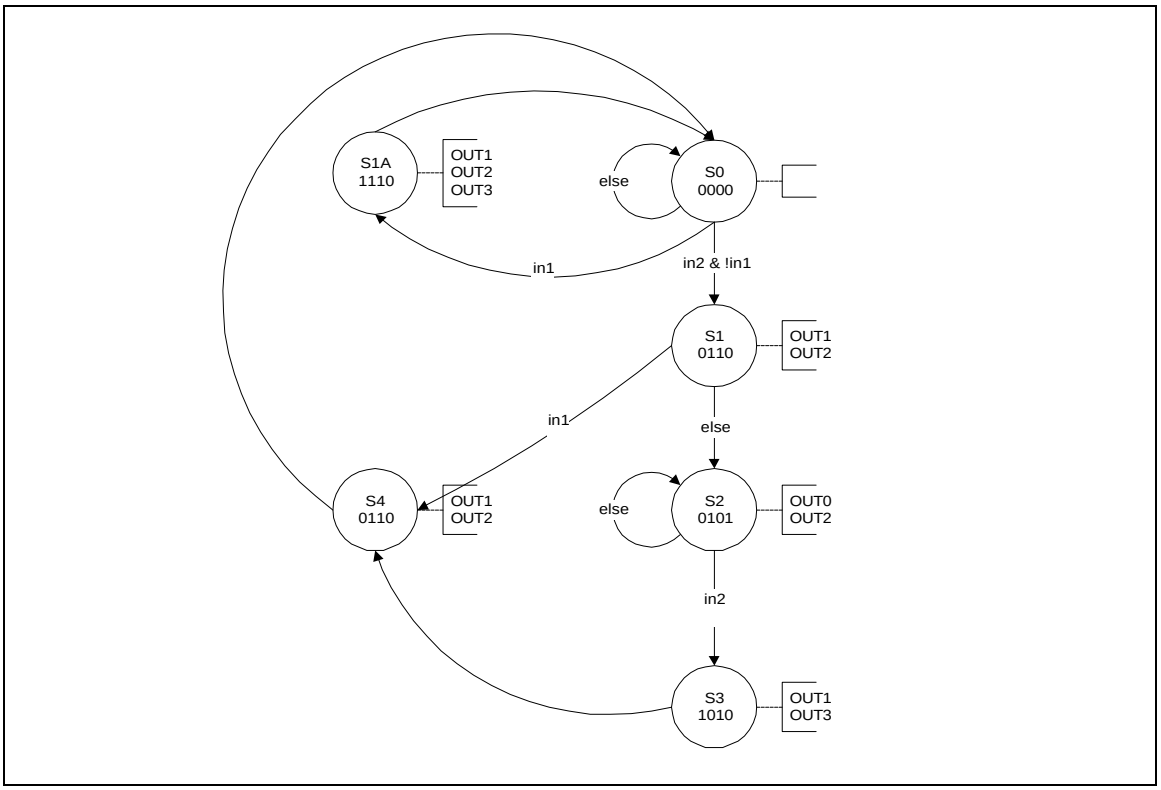


*Figure 3: Example State Diagram (Moore Machine)*

## *State Diagram*

A state diagram allows the designer to describe the desired state machine operation graphically. This helps him or her visualize the operation of the state machine prior to implementation. The diagram contains a wealth of information.

First, the diagram shows state transitions. The circles and arrows describe how the state machine moves from one state to another. The circles represent a particular value of the state variable. The arrowed lines describe how the state machine transitions from one state to the next state. The arrowed lines contain a boolean expression which shows the criteria for a transition from one state to another. If the boolean expression is TRUE and the current state is the state at the source of the arrowed line, the state machine will transition to the destination state on the next clock.

The diagram also shows the values of the output variables during each state. In a Moore Machine, the outputs are only a function of the current state; inputs have no effect after the clock. This makes the Moore Machine an ideal candidate for a fully synchronous design. Outputs of a Moore Machine have reasonably predictable timing and are always referenced to the clock. On the other hand, Mealy Machine outputs can be dependent on the value of inputs to the state machine which may be transitioning at less predictable times. It is for this reason, that Moore Machines generally provide for a smoother implementation in FPGA and Gate Array designs.

In a Moore Machine, the outputs can be represented just to the right of the circle which represents the state

as well as inside the circle. A useful system is to include just the outputs which are to be asserted during that cycle to the right of the circle. This highlights the fact that a particular signal is active during that state and nowhere else. For signals which do not have an asserted and deasserted state, the signals value can be shown for each cycle. An example of this might be a signal like “RDWR”. If the signal is logic 1, a read of some sort is implied while a logic 0 indicates a write is active.

Whenever possible, choose 0 to be the deasserted state and 1 to be the asserted state for State Machine inputs and outputs. Most FPGA and ASIC primitive libraries allow an I/O buffer to invert or not with no degradation in propagation delay. The signals may be inverted to the outside world, but inside the device, try to make all signals active high. Its less confusing, and when the complexity of the design ratchets up, you will be glad to have fewer mental inversions to make.

## *Input Variables*

**ALL** Input variables **MUST** be synchronous to the state machine clock**.** If they are not, strange things will begin to happen to the state machine in actual operation. Illegal states will be mysteriously entered. Total state machine lockup can result. Inevitably, the design will fail intermittently in the field. Why?

The reason lies in the fact that a physical state machine implementation uses physical gates which have a non-zero propagation delay. An input signal progagating through gates to the D input of one state flip flop will be slightly faster or slower than that same input signal travelling through a different set of gates to another flip flop’s D input. If the input signal changes at just the wrong time, the fast path will see the change, but the slow one won’t. The clock comes along and one of the flip-flops will now have an incorrect logic level. The overall state machine just made an illegal state transition; the circuit has failed. This type of design error generally goes unnoticed because the faulty behavior shows up only a small percentage of the time. Bewareof the asynchronous input!

The cure for the asynchronous input is a synchronizer. Generally a 2 stage synchronizer is adequate to prevent illegal state entry. It should be noted that there is a statistical probability, albeit very low, that an asynchronous input may still be able to propagate through a synchronizer. For each stage of synchronization employed, the probability is reduced considerably. Some space/military designs require the use of a 3 stage synchronizer, but 2 stages is generally considered the standard for commercial and industrial grade designs. To minimize latency, use the falling edge of the clock for the first stage and the rising edge as the second stage.

## *Output Variables*

Remember that in a Moore state machine, the value of output variables are solely a function of the current state. One of the valuable things about a Moore design is that the timing of the state transitions/outputs is dependent only upon the clock. But, we start to lose that timing simplicity when combinatorial logic is used to generate outputs which are functions of the state bits. Depending on the complexity of this “back-end” logic, the actual timing of the outputs can be seriously degraded.

Another problem with this popular approach is that narrow glitches can appear on these combinatorial outputs during state transitions. Due to clock skew, unequal clock to Q prop delay and unequal combinatorial logic prop delay, glitches will invariably occur. These glitches may be very narrow, and may not be visible at all on most logic analyzers and some scopes. You can be assured that sensitive edge based circuits which are driven by these outputs will see them and again circuit failure can result.

Worse still, glitch width and severity will change with temperature and power supply variations.

## A Better Way

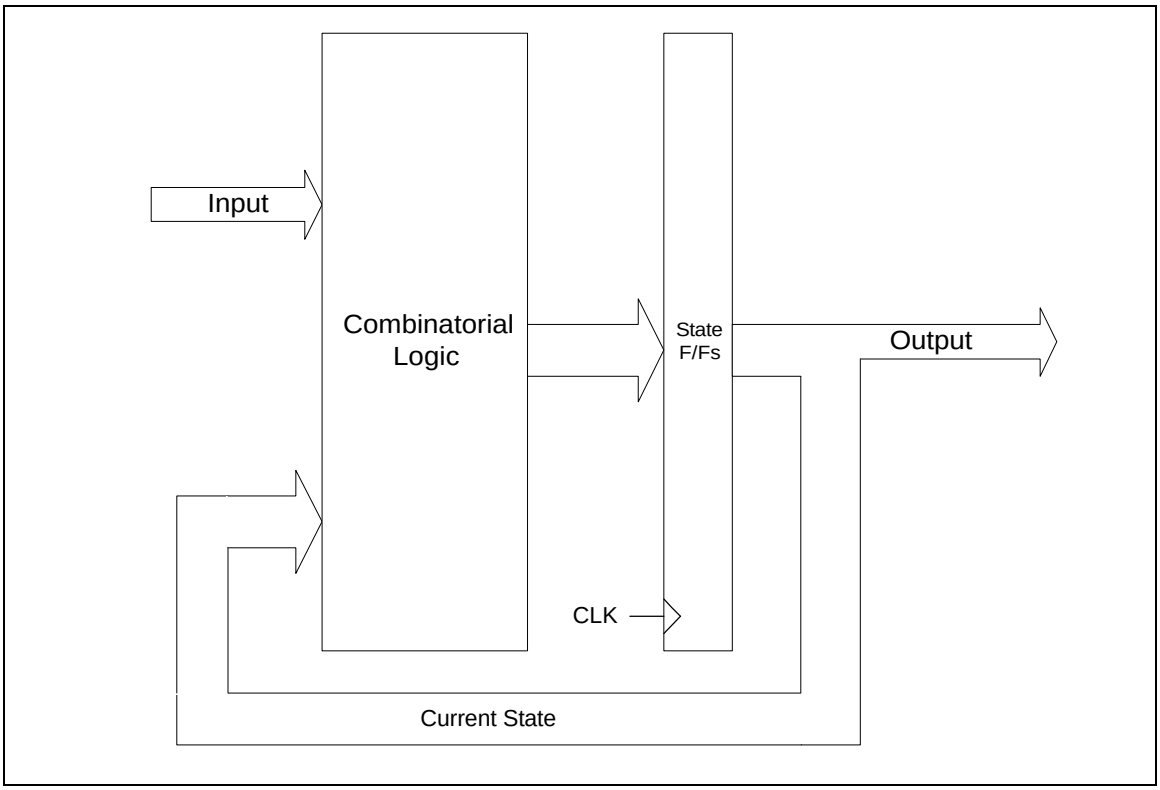


*Figure 4: Modified Moore Machine*

### *Registered Outputs*

A way to avoid both of the previous problems is to define the state machine such that the outputs ARE the state bits. Each output has predictable timing and minimal propagation delay. There are no glitches, because a single flip-flop changes at most once per clock assuming setup and hold timing on the input is observed. If an output is to be a logic 1 for two consecutive states, it is guaranteed that there will be no low going glitch if the inputs are properly synchronized as described above.

### *State Selection*

It is certainly possible, even likely, that the state machine requires that the output variables be at a unique logical level pattern in two different machine states. This presents a minor problem to the scheme described above in that we have decided to make the machine state equal to the state of the outputs.

To solve this problem, we must add 1 or more additional state bits which serve no purpose other than to differentiate two or more unique machine states which have the same output variable values. For instance, suppose that we are designing a state machine as shown in the diagram below:

Notice in the State Diagram example, that State S1 and State S4 require that the 4 outputs be at logic level 0110. To differentiate the two states, we add another state bit so that state S1 is now 00110 and S4 is now 10110. The added state bit is not wired to anything outside the state machine, but it allows each state to have

a unique output pattern.

## *Verilog Implementation*

### Output Variable Definition Section

In this section, we will use the verilog `‘define` statement to associate a state value to each state name.We simply transform the data directly from the State Diagram. The State Name and the state value are inside the circle that defines each state. Note the additional bit added to differentiate S1 and S4.

```
//
// Define Output states in terms of input variables
//                 +---------- OUT3
//                 |+--------- OUT2
//                 ||+-------- OUT1
//  extra bit ---+ |||+------- OUT0
//               | ||||
//               V VVVV
‘define S0    5’b0_0000     // Idle state, all bits set to 0
‘define S1    5’b0_0110     // extra bit == 0 here
‘define S1A   5’b0_1110
‘define S2    5’b0_0101
‘define S3    5’b0_1010
‘define S4    5’b1_0110     // extra bit == 1 here
```

### Output Assignment Section

In this section, we will first declare the state register. In this case it will be a 5 bit reg. Then we will use a continuous assignment to assign an output to its associated state bit. This continuous assignment will synthesize to a zero delay wire i.e. the output name and the register bit are synonyms. If there are asynchronous inputs to the state machine, this is a good place to instantiate synchronizers.

```
// Declare state flip flops
reg [4:0] state;

// Continuous assignment of state bits to outputs
assign {OUT3, OUT2, OUT1, OUT0} = state[3:0];

// Synchronize asynchronous inputs if needed
```

### State Transition Section

In this section, we use the State Diagram as a guide to explicitly define how the state machine transitions from one state to another. We also provide a means of initializing the state machine with a synchronous reset. The state machine will enter state S0 upon reset. Note that as reset is just another input to the state machine, care should be taken to ensure that it is synchronous as well.

```
//
// Define State Transitions
//
always @(posedge clk)
    if (reset)
        state = ‘S0;
    else
        case (state)
          ‘S0: if (in1 && !in2)
                   state = ‘S1;
               else if (in2)
                   state = ‘S1A;
               else
                   state = ‘S0;
          ‘S1: if (in1)
                   state = ‘S4;
               else
                   state = ‘S2;
          ‘S1A:
                   state = ‘S0;
          ‘S2: if (in2)
                   state = ‘S3;
               else
                   state = ‘S2;
          ‘S3:
                   state = ‘S4;
          ‘S4:
                   state = ‘S0;
          default:
                   state = ‘S0;
        endcase
```

### *Summary*

This paper first discussed some of the pitfalls which hardware designers can fall prey to in the design of synchronous state machines. Then a method avoiding these problems was discussed which has particular usefulness in FPGA and Gate Array implementations. Finally a complete example was presented in Verilog HDL to illustrate the simplicity of the problem solution.