EECS150: Finite State Machines in Verilog

UC Berkeley College of Engineering
Department of Electrical Engineering and Computer Science

1 Introduction

This document describes how to write a finite state machine (FSM) in Verilog. Specifically, in EECS150, you will be designing Moore machines for your project. This document only discusses how to describe Moore machines.

Moore machines are very useful because their output signals are synchronized with the clock. No matter when input signals reach the Moore Machine, its output signals will not change until the rising edge of the next clock cycle. This is very important to avoid setup timing violations. For example, if a Mealy machines input signal(s) changes sometime in the middle of a clock cycle, one or more of its outputs and next state signals may change some time later. "Some time later" might come after the setup time threshold for the next rising edge. If this happens, the registers that will hold the FSMs next state may receive garbage, or just incorrect inputs. Obviously, this amounts to a bug(s) in your FSM. A very painful and difficult-to-find bug at that.

The tradeoff in using the Moore machine is that sometimes the Moore machine will require more states to specify its function than the Mealy machine. This is because in a Moore machine, output signals are only dependent on the current state. In a Mealy machine, outputs are dependent on both the current state *and* the inputs. The Mealy machine allows you to specify different output behavior for a single state. In EECS150, however, the FSMs that you will be designing do not typically have enough states for this to create a significant problem. We will err on the side of caution, and vie for a safe but sometimes more verbose FSM implementation, in this course.

2 Motivation

EECS150 is concerned with circuit design. You will be using Verilog to describe your circuits. Unfortunately, Verilog, being originally designed to support circuit simulation rather than circuit synthesis, is chalked full of syntactical idiosyncrasies that, if not properly understood, will create odd bugs in your designs. This document will show you how to write a Moore FSM in a template-based fashion. This "cookie-cutter" approach is designed to avoid Verilog's bug-prone areas, while keeping your code as non-verbose as possible. Verilog is a means to an end. This document will show you how to get to the point: designing circuits; while fighting Verilog as little as possible.

3 A Basic FSM

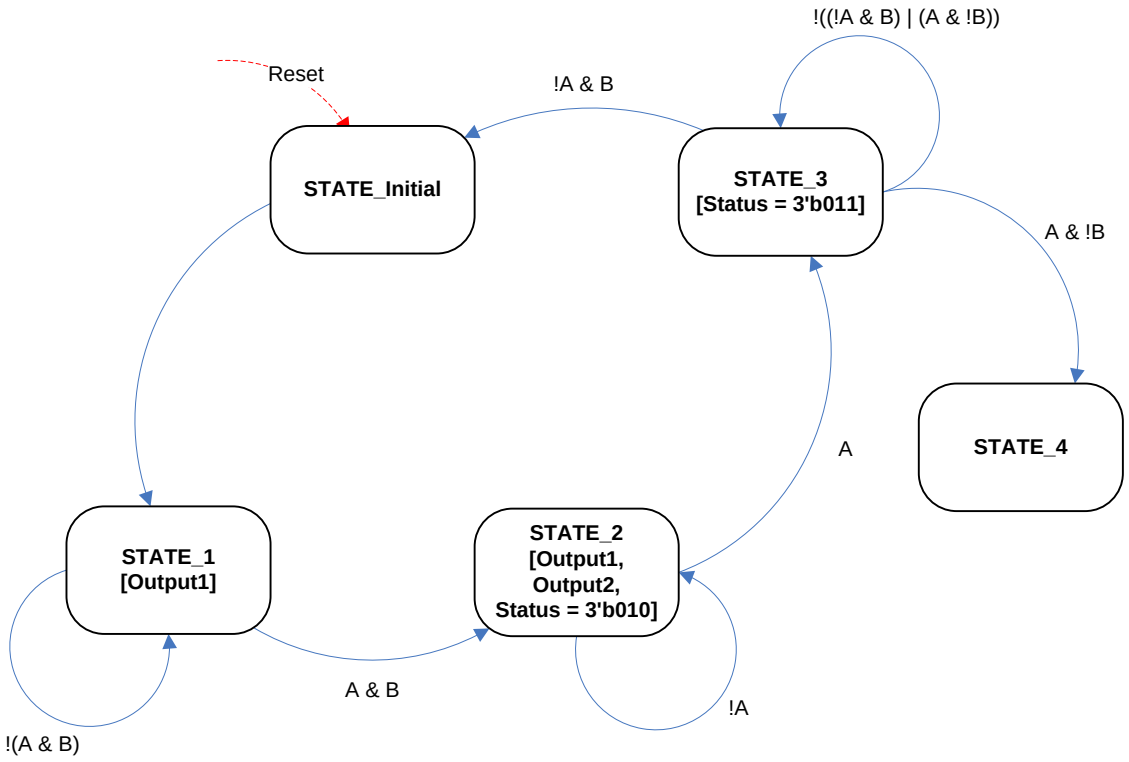
Figure 1 depicts an example Moore FSM. You can tell that this is a Moore machine because the outputs are shown inside [...]s instead of on state transition arcs. The following sections will refer to Figure 1 as an example use-case for the Moore machine FSM template.

The FSM shown in Figure 1 is useful because it exemplifies the following:

1. The concept of an initial state.[1]
2. States with non-conditional outward transitions.
3. States with conditional outward transitions.

[1] There must always be an initial state for the FSM to start at after a `Reset`.

Figure 1 A basic FSM



`&`	logical `and`
`|`	logical `or`
`!`	logical `not`

4. States that loop back onto themselves.
5. States with no outward transitions.

We would like to be able to express this type of behavior in a Verilog-written FSM.

4 The FSM in Verilog

In looking at Figure 1, we will need a way to express the following in Verilog:

1. A state encoding for each state.
2. A mechanism for keeping track of the current state.
3. Transitions from state to state.
4. Output values based on the current state.

We will construct the FSM one step at a time.

4.1 1: Creating a State Encoding

We will create our state encoding with Verilog parameters. Parameters are symbolic constants with either global (given by the Verilog keyword `parameter`) or module (`localparam`) scope. Because we only

Program 1 The state encoding (in decimal)

```
localparam STATE_Initial = 3'd0,
           STATE_1 = 3'd1,
           STATE_2 = 3'd2,
           STATE_3 = 3'd3,
           STATE_4 = 3'd4;
```

want our state encoding to be visible to the module in which we will write the FSM, we will use the latter: `localparam`. With this in mind, we can specify Program 1.

In Program 1, the `3'd` notation indicates that the number specified is in the decimal radix. If we were to use `3'b`, the encoding would look like that shown in Program 2. Both implementations are equivelent. Base 10, or `3'd`, is typically easier to read.

Because this FSM has 5 total states, we must allocate 3 bits to specify the encoding (hence `3'd` as opposed to `2'd` or `4'd`. **This is extremely important.** If you specify too few bits for your state encoding, Verilog will not warn you. In fact, when synthesized, each state will only get as many bits as you provide. For example, if `STATE_4` was specified like this: `STATE_4 = 2'd4`, `STATE_4` would be specified as `00`, the bottem 2 bits of what was intended, namely `100`.

Program 2 The state encoding (in binary)

```
localparam STATE_Initial = 3'b000,
           STATE_1 = 3'b001,
           STATE_2 = 3'b010,
           STATE_3 = 3'b011,
           STATE_4 = 3'b100;
```

As 3 bits can specify a total of 8 states (0-7), our encoding specifies 3 potential states not specified as being actual states. There are several ways of dealing with this problem:

1. Ignore it, and always press `Reset` as a way of initializing the FSM.
2. Specify these states, and make non-conditional transitions from them to the `STATE_Initial`.

To reduce ambiguity, we will choose the second option, which makes our final state encoding that shown in Program 3.

Program 3 The state encoding with place-holder states (in decimal)

```
localparam STATE_Initial = 3'd0,
           STATE_1 = 3'd1,
           STATE_2 = 3'd2,
           STATE_3 = 3'd3,
           STATE_4 = 3'd4,
           STATE_5_PlaceHolder = 3'd5,
           STATE_6_PlaceHolder = 3'd6,
           STATE_7_PlaceHolder = 3'd7;
```

This is a simple encoding: `STATE_Initial` is assigned 0, `STATE_1` is assigned 1, etc. This is not optimal if state minimization can be performed on the FSM (taught at the end of EECS150). We do not recommend applying state minimization techniques by hand, however. They have the tendancy to introduce bugs and create cryptic FSMs that cannot be easily understood by human readers. This defeats one of the large pros of Verilog: human readability. Furthermore, the Synthesis tools that 'compile' an FSM, written in Verilog, perform state minimization automatically. Only perform state minimization manually to the extent that the function of the FSM remains clear.

4.2 2: Keeping Track of the Current State

We have several options in how to store the current state of our FSM. The first option is to instantiate a module that acts as a register and use its output value as our current state. Alternatively, we can create a `reg` element of the appropriate width and use its value as our current state. We will use the second method for the remainder of this tutorial, out of personal preference. As such, we will store the current state as depicted in Program 4.

Program 4 Storing the current state in a `reg`

```
reg [2:0] CurrentState;
```

If this material seems unfamilar, read Section 4.2.1, which explains the difference betweeen `wire` and `reg` elements in Verilog. **If this material is familiar**, feel free to skip to Section 4.3.

4.2.1 `wire` and `reg` Elements in Verilog

Sections 4.2.2 to 4.2.4 discuss the difference between `wire` and `reg` in Verilog, and when to use each of them.

4.2.2 `wire` Elements (Combinational logic)

`wire` elements are simple wires (or busses of arbitrary width) in Verilog designs. The following are syntax rules when using `wire`s:

1. `wire` elements are used to connect `input` and `output` **p**orts of a module **i**nstantiation together with some other element in your design.
2. `wire` elements are used as `input`s and `output`s within an actual module declaration.
3. `wire` elements must be driven by something, and cannot store a value without being driven.
4. `wire` elements cannot be used as the left-hand side of an `=` or `<=` sign in an `always@` block.
5. `wire` elements are the only legal type on the left-hand side of an `assign` statement.
6. `wire` elements are a stateless way of connecting two peices in a Verilog-based design.
7. `wire` elements can only be used to model combinational logic.

Program 5 shows various legal uses of the `wire` element.

Program 5 Legal uses of the `wire` element

```
wire        A, B, C, D, E; // simple 1-bit wide wires
wire [8:0] Wide;     // a 9-bit wide wire
reg I;

assign A = B & C;    // using a wire with an assign statement

always @(B or C) begin
    I = B | C;       // using wires on the right-hand side of an always@
                     // assignment
end

mymodule mymodule_instance(.In (D),
                           .Out(E)); // using a wire as the output of a module
```

4.2.3 `reg` Elements (Combinational and Sequential logic)

`reg` are similar to wires, but can be used to store information ('state') like registers. The following are syntax rules when using `reg` elements.

1. `reg` elements **c**an be connected to the input **p**ort of a module **i**nstantiation.
2. `reg` elements **c**annot be connected to the output **p**ort of a module **i**nstantiation.
3. `reg` elements **c**an be used as `output`s within an actual module declaration.
4. `reg` elements **c**annot be used as `input`s within an actual module declaration.
5. `reg` is the only legal type on the left-hand side of an `always@` block `=` or `<=` sign.
6. `reg` is the only legal type on the left-hand side of an `initial` block `=` sign (used in Test Benches).
7. `reg` cannot be used on the left-hand side of an `assign` statement.
8. `reg` can be used to create registers when used in conjunction with `always@(posedge Clock)` blocks.
9. `reg` can, therefore, be used to create both combinational and sequential logic.

Program 6 shows various legal uses of the `reg` element.

Program 6 Legal uses of the `reg` element

```
wire      A, B;
reg       I, J, K;    // simple 1-bit wide reg elements
reg[8:0]  Wide;       // a 9-bit wide reg element

always @(A or B) begin
    I = A | B;        // using a reg as the left-hand side of an always@
                      // assignment
end

initial begin         // using a reg in an initial block
    J = 1'b1;
    #1
    J = 1'b0;
end

always @(posedge Clock) begin
    K <= I;           // using a reg to create a positive-edge-triggered register
end
```

4.2.4 When `wire` and `reg` Elements are Interchangable

`wire` and `reg` elements can be used interchangably in certain situations:

1. Both can appear on the right-hand side of `assign` statements and `always@` block `=` or `<=` signs.
2. Both can be connected to the input **p**orts of module instantiations.

4.3 3: Transitioning from State to State

After we have established our state encoding and a means of storing the current state value (which will henceforth be referred to as `CurrentState`), our next task is to create a way for the FSM to actually change state, and for it to choose *how* to change state. This material requires that you be comfortable with `always@` blocks. **If the `always@` block is unfamilar**, read Section 4.3.1, which explains `always@` block in Verilog. **If the `always@` block is familiar**, feel free to skip to Section 4.3.8.

4.3.1 `always@` Blocks in Verilog

Sections 4.3.2 to 4.3.7 discuss `always@` blocks in Verilog, and when to use the two major flavors of `always@` block, namely the `always@( * )` and `always@(posedge Clock)` block.

4.3.2 `always@` Blocks

`always@` blocks are used to describe events that should happen under certain conditions. `always@` blocks are always followed by a set of parentheses, a `begin`, some code, and an `end`. Program 7 shows a skeleton `always@` block.

Program 7 The skeleton of an `always@` block

```
always @( ... sensitivity list ... ) begin
    ... elements ...
end
```

In Program 7, the `sensitivity list` is discussed in greater detail in Section 4.3.6. The contents of the `always@` block, namely `elements` describe elements that should be set when the sensitivity list is "satisfied." For now, just know that when the sensitivity list is "satisfied," the elements inside the `always@` block are set/updated. They are not otherwise.

Elements in an `always@` block are set/updated in sequentially and in parallel, depending on the type of assignment used. There are two types of assignments: `<=` (non-blocking) and `=` (blocking).

4.3.3 `<=` (non-blocking) Assignments

Non-blocking assignments happen in parallel. In other words, if an `always@` block contains multiple `<=` assignments, which are literally written in Verilog sequentially, you should think of all of the assignments being set at **exactly** the same time. For example, consider Program 8.

Program 8 `<=` assignments inside of an `always@` block

```
always @( ... sensitivity list ... ) begin
    B <= A;
    C <= B;
    D <= C;
end
```

Program 8 specifies a circuit that reads "when the sensitivity list is satisfied, `B` gets `A`'s value, `C` gets `B`'s **old** value, and `D` gets `C`'s **old** value." The key here is that `C` gets `B`'s **old** value, etc (read: **think OLD value!**. This ensures that `C` is not set to `A`, as `A` is `B`'s **new** value, as of the `always@` block's execution. Non-blocking assignments are used when specifying sequential[2] logic (see Section 4.3.5).

4.3.4 `=` (blocking) Assignments

Blocking assignments happen sequentially. In other words, if an `always@` block contains multiple `=` assignments, you should think of the assignments being set one after another. For example, consider Program 9.

Program 9 specifies a circuit that reads "when the sensitivity list is satisfied, `B` gets `A`, `C` gets `B`, and `D` gets `C`." But, by the time `C` gets `B`, `B` has been set to `A`. Likewise, by the time `D` gets `C`, `C` has been set to `B`, which, as we stated above, has been set to `A`. This `always@` block turns `B`, `C`, and `D` into `A`. Blocking assignments are used when specifying combinational logic (see Section 4.3.6).

[2]This point might be confusing. We said that non-blocking statements happen in parallel. Yet, they are useful for specifying *sequential* logic? In digital design, sequential logic doesn't refer to things happening in parallel or a sequence, as we have been discussing, but rather to logic that has *state*.

Program 9 `=` assignments inside of an `always@` block

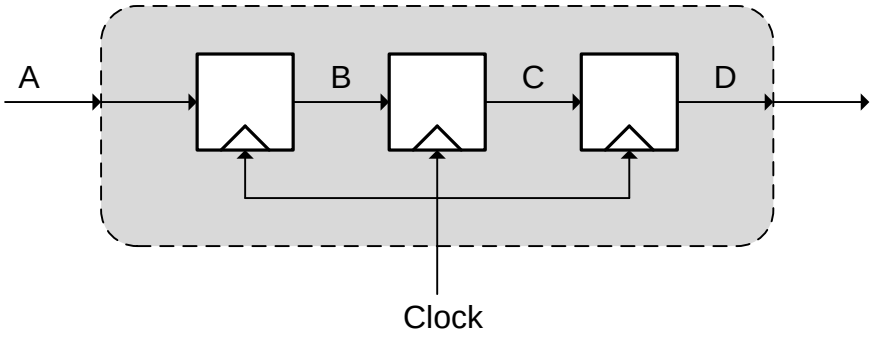
```
always @( ... sensitivity list ... ) begin
    B = A;
    C = B;
    D = C;
end
```

4.3.5 `always@(posedge Clock)` Blocks

`always@(posedge Clock)` ("always at the **positive edge** of the clock") or `always@(negedge Clock)` ("always at the **negative edge** of the clock") blocks are used to describe **Sequential Logic**, or Registers. Only `<=` (non-blocking) assignments should be used in an `always@(posedge Clock)` block. Never use `=` (blocking) assignments in `always@(posedge Clock)` blocks. Only use `always@(posedge Clock)` blocks when you want to infer an element(s) that changes its value at the positive or negative edge of the clock.

For example, consider Figure 2, a recreation of Program 8 that uses `posedge Clock` as its sensitivity list. Figure 2 is also known as a shift register. The completed `always@` block is shown in Program 10.

Figure 2 A shift register



Program 10 A shift register, using `<=` assignments inside of an `always@(posedge Clock)` block

```
always @(posedge Clock) begin
    B <= A;
    C <= B;
    D <= C;
end
```

4.3.6 `always@( * )` Blocks

`always@( * )` blocks are used to describe **Combinational Logic**, or Logic Gates. Only `=` (blocking) assignments should be used in an `always@( * )` block. Never use `<=` (non-blocking) assignments in `always@( * )` blocks. Only use `always@( * )` block when you want to infer an element(s) that changes its value as soon as one or more of its inputs change.

Always use '`*`' (star) for your **sensitivity list** in `always@( * )` blocks. The sensitivity list specifies which signals should trigger the elements inside the `always@` block to be updated. For example, given 3 wires `A`, `B` and `C`, we can create an `and` gate through Program 11, and shown graphically in Figure 3.

Program 11 specifies that "when A or B change values, update the value of every element inside the `always@( * )` block. In this case, the only element inside the `always@( * )` block is `C`, which in this case is assigned the `and` of `A` and `B`. A very common bug is to introduce an **incomplete sensitivity list**. See Program 12 for two examples of incomplete sensitivity lists.

Program 11 An `and` gate inside of an `always@( * )` block

```
always @(A or B) begin
    C = A & B;
end
```

Figure 3 The `and` gate produced by Program 11 (this is a normal `and` gate!)

In Program 12, the first example produces an `and` gate that only updates its output `C` when `A` changes. If `B` changes, but `A` does not change, `C` does not change because the `always@(A)` block isn't executed. Likewise, the second example produces an `and` gate that doesn't react to a change in `A`. **Incomplete sensitivity lists are almost NEVER what you want!** They introduce very hard-to-find bugs. As such, we use `always@( * )`. The '`*`' is shorthand for `always@(A or B)` in our examples. In other words, '`*`' sets the sensitivity list to any values that can have an impact on a value(s) determined by the `always@( * )` block. '`*`' provides a bug-free shorthand for creating complete sensitivity lists.

4.3.7 Pitfalls

You might be wondering what happens if you don't follow the conventions set forth in Sections 4.3.5 and 4.3.6. The following are some easy-to-make mistakes in Verilog that can have a dramatic [and undesired] effect on a circuit.

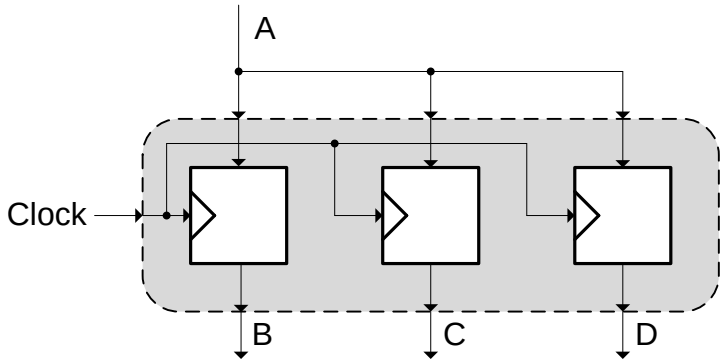
1. Consider the shift register from Figure 2. If you place `=` assignments inside of an `always@(posedge Clock)` block to produce the shift register, you instead get the parallel registers shown in Figure 4 and Program 13. You might also get one register, whose output is tied to `B`, `C` and `D`. Both possible outcomes are equivelent. These circuit make sense, but don't create shift registers! (As shift registers are common construct, we assume that you wanted to create a shift register)

Program 12 An `and` gate with an incomplete sensitivity list (**this is incorrect!**)

```
always @(A) begin
    C = A & B;
end
```

```
always @(B) begin
    C = A & B;
end
```

Figure 4 Parallel registers



Program 13 Parallel registers, using `=` assignments inside of an `always@(posedge Clock)` block

```
always @(posedge Clock) begin
    B = A;
    C = B;
    D = C;
end
```

2. The opposite example (shown in Program 14), where we place `<=` assignments inside of `always@( * )` is less pronounced. In this case, just consider what type of circuit you want to create: do you want all statements to be executed in parallel or in 'sequence' (see Section 4.3.3 and 4.3.4)? In the `always@( * )`, the distinction between `<=` and `=` is sometimes very subtle, as the point of `always@( * )` is to trigger at indefinite times (unlike the very definite `posedge Clock`). We recommend `=` in conjunction with `always@( * )` to establish good convention (as `=` was originally meant to be associated with combinational logic).

Program 14 `<=` assignments inside of `always@( * )` blocks

```
always @( * ) begin
    B <= A;
    C <= B;
    D <= C;
end
```

3. Consider the case of incompletely specified sensitivity lists. An incompletely specified sensitivity list, as discussed in Section 4.3.6, will create an `always@` block that doesn't always set/update its elements when it should. In truth, synthesis tools will often know what you mean if you provide an incomplete sensitivity list, and pretend that your sensitivity list was complete. This is **not** the case with simulation tools (like ModelSim), however. ModelSim will not correct your sensitivity list bugs, and your simulations will be plagued with odd errors. Furthermore, the synthesis tools catching your errors is not guarenteed. An easy way to avoid these potential problems is to use `always@( * )` as opposed to `always@(Input1 or Input 2 or ...)`.

4. Lastly, a very subtle point which perhaps has the potential to cause the most frustration is **latch generation**. If you don't assign **every** element that **can** be assigned inside an `always@( * )` block **every** time that `always@( * )` block is executed, a latch (similar to a register but much harder to work with in FPGAs) will be inferred for that element. This is **never** what you want and is a terrible place for bugs. As this is subtle, it is somewhat hard to visualize. Consider Program 15.

Program 15 An `always@( * )` block that will generate a latch for `C`

```
wire Trigger, Pass;
reg A, C;

always @( * ) begin
    A = 1'b0;
    if (Trigger) begin
        A = Pass;
        C = Pass;
    end
end
```

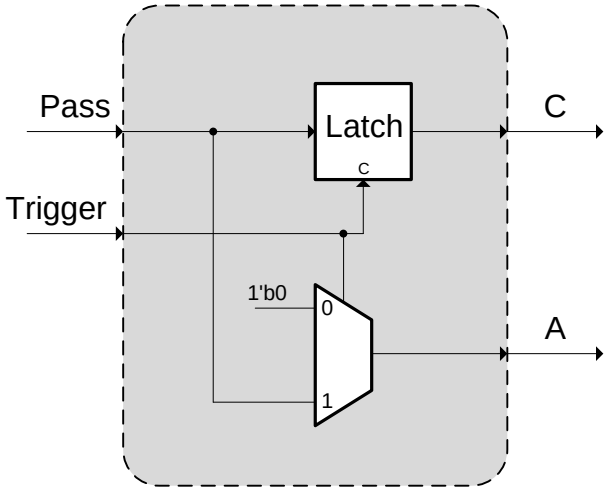
In Program 15, `A` and `C` are both assigned in at least one place inside the `always@` block.

`A` is always assigned at least once. This is because the first line of the `always@` block specifies a

default value for `A`. This is a perfectly valid assignment. It ensures that `A` is always assigned with each execution of the `always@` block.

`C` on the other hand is **not** always assigned. When `Trigger = 1'b1`, the `if` statement 'executes' and both `A` and `C` get set. If `Trigger = 1'b0`, however, the `if` is skipped. `A` is safe, as it was given a default value on the first line of the `always@` block. `C` on the other hand, doesn't get assigned at all when this happens. As such, a latch is inferred for `C`. The erroneous circuit depicted in Program 15 is shown in Figure 5.

Figure 5 The circuit generated by Program 15 (this is an erroneous circuit!)



To fix this problem, we must make sure that `C` gets set every time the `always@` block is 'executed.' A simple way to force this is to add another default value, depicted in Program 16 and shown in Figure 6.

Program 16 An `always@( * )` block that will not generate latches

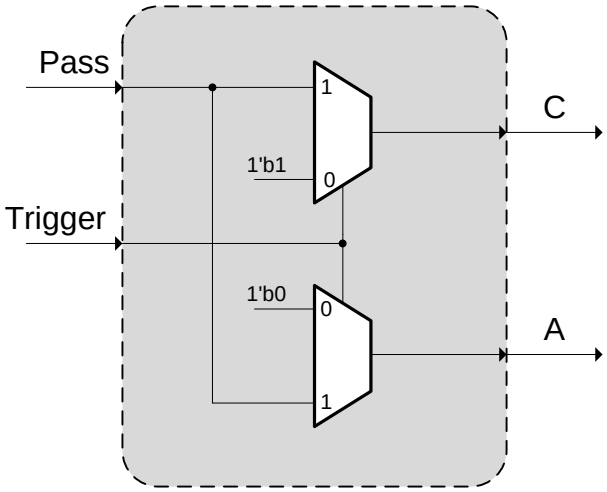
```
wire Trigger, Pass;
reg A, C;

always @( * ) begin
    A = 1'b0;
    C = 1'b1;
    if (Trigger) begin
        A = Pass;
        C = Pass;
    end
end
```

Default values are an easy way to avoid latch generation, however, will sometimes break the logic in a design. As such, other ways of ensuring that each value always gets set are going to be worth looking into. Typically, they involve proper use of the Verilog `else` statement, and other flow constructs.

Know that setting a `reg` to itself is not an acceptable way to ensure that the `reg` always gets set. For example, `C = C;` enjected into the top of the `always@( * )` block in Program 15 will not supress latch generation. In every 'execution' of an `always@( * )` block, each value that is assigned in **at least one place** must be assigned to a non-trivial value during **every** 'execution' of the `always@( * )` block.

Figure 6 The circuit generated by Program 16 (this is correct!)



4.3.8 Post-`always@`: Specifying our FSM's Transition Behavior

At this point, the tutorial assumes that you are familiar with the material starting at Section 4.3.1.

Specifying an FSM's transition behavior is done in 3 steps. First, we must choose how to store the information that will tell the FSM what the next state should be on the next rising edge. Second, we must create a physical means of transitioning from the `CurrentState` to the next state. Third, we must implement the conditional-transitioning mechanism that will choose what the next state should be and under what conditions a transition should be made.

We will store the next state information in a `reg` of the same width as the `CurrentState` `reg`. This is because we will assign the next state in an `always@( * )` block. The details of this process will be specified shortly. Our `CurrentState` and next state (which will henceforth be called `NextState`) are shown in Program 17.

Program 17 Storing the current state and next state in `reg` elements

```
reg [2:0] CurrentState;
reg [2:0] NextState;
```

Once we have established `CurrentState` and `NextState`, we can create a means of transitioning at the positive-edge of the clock through an `always@(posedge Clock)` block, as shown in Program 18. The `always@(posedge Clock)` block in Program 18 is identical, and must be present, in **all FSMs**. It returns the FSM to the initial state after a reset, and forces `CurrentState` to take on the value stored in `NextState` at the positive-edge of the clock.

Program 18 The `always@(posedge Clock)` block

```
always@(posedge Clock) begin
    if (Reset) CurrentState <= STATE_Initial;
    else CurrentState <= NextState;
end
```

Once we have established a means of transitioning at the positive-edge of the clock, we are ready to specify the FSM's conditional-transitioning behavior, or how it chooses what `NextState` should be. We will use an `always( * )` block in conjunction with a `case` statement to accomplish this. Unlike the `always@(posedge Clock)` block above, which is the same for **every** FSM, the `always@( * )` block is very

dependant on which FSM it is written for. This is because while every FSM will transition from state to state on the positive-edge of the clock, different FSMs will have different states and different arcs from state to state. Again, for this tutorial, we will specify an example `always@( * )` block using the FSM shown in Figure 1.

Program 19 The `always@( * )` block

```
1 always@( * ) begin
2     NextState = CurrentState;
3     case (CurrentState)
4         STATE_Initial: begin
5             NextState = STATE_1;
6         end
7         STATE_1: begin
8             if (A & B) NextState = STATE_2;
9         end
10        STATE_2: begin
11            if (A) NextState = STATE_3;
12        end
13        STATE_3: begin
14            if (!A & B) NextState = STATE_Initial;
15            else if (A & !B) NextState = STATE_4;
16        end
17        STATE_4: begin
18        end
19        // ---------------------------------------
20        // Place-holder transitions
21        // ---------------------------------------
22        STATE_5_PlaceHolder: begin
23            NextState = STATE_Initial;
24        end
25        STATE_6_PlaceHolder: begin
26            NextState = STATE_Initial;
27        end
28        STATE_7_PlaceHolder: begin
29            NextState = STATE_Initial;
30        end
31        // ---------------------------------------
32    endcase
33 end
```

We will now discuss Program 19 line by line.

On line 2, we establish a default value for the `NextState` `reg`. The default value reads: "if `NextState` does not get assigned anywhere in the `case` statement (below), set `NextState` to the `CurrentState` (which means that no state change will occur). This avoids a potential Verilog-introduced bug that we will explain shortly. It also greatly reduces the amount of code that the `always@( * )` block will take to specify (another perk that will be explained shortly).

Line 3 specifies the beginning of a Verilog `case` statement. For all FSMs, this `case` statement is of the form: `case(CurrentState)`. This means that the branch of the `case` statement that is chosen based on the `CurrentState`. This makes sense because state transition behavior for Moore machines is based solely on the `CurrentState`.

Line 4 specifies the case when the `CurrentState` is `STATE_Initial`. This statement is self-explanatory: "when we are in the initial state, **always** transition to `STATE_1` at the next rising edge of the clock." This matches Figure 1, as it should.

Line 7 specifies the case when `CurrentState` is `STATE_1`. This case is more interesting because of the implied transition loop back onto `STATE_1`. So, if `A & B` is true, transition to `STATE_2`. However, if `A & B` is **not** true, stay in `STATE_1`. This last statement isn't written in Program 19 anywhere, however, is an implied `else` statement because of the default we established in Line 2! **This is where the Line**

2 default value saves space: whenever a state loops back on itself, we need not specify an `else` statement. In fact, we need not write anything at all.

Lines 10-12, describing `STATE_2` takes advantage of the Line 2 default value in the same way.

Lines 13-16 also take advantage of the default, but are worth mentioning as they show how a state (`STATE_4` can make different transitions to different states through an `else if` statement.

Lines 22-30 specify the actions to take if the FSM ever enters one of the unused states described in Section 3. Specifically, the FSM should return to the initial state so that it can restart its normal operation. If a state machine has unused states, because its state encoding does not take up every value in its binary specification, for a given number of bits, and it accidentally enters an unused state, it might get stuck in an infinite loop of garbage transitions.

As a closing note on state transitions, **there is another very good reason to establish the default value as was done on Line 2**. The latch problem discussed in the tutorial on `always@` blocks (starting at Section 4.3.1) is very common in FSMs because FSMs liberally assign values in potentially large and complex `always@` blocks. Using default values can greatly decrease the chance of generating latches in FSMs. If not for all of the other convenience-related reasons, you should use them for this reason.

4.4 4: Outputting Values Based on the `CurrentState`

The final step in specifying a Moore FSM is assigning output values based on `CurrentState`. Fortunately, this is simple with `assign` statements. See Program 20 for the FSM's (from Figure 1) output specification.

Program 20 The outputs from Figure 1

```
1 wire      Output1, Output2;
2 reg[2:0] Status;
3
4 assign Output1 = (CurrentState == STATE_1) | (CurrentState == STATE_2);
5 assign Output2 = (CurrentState == STATE_2);
6
7 always@( * ) begin
8     Status = 3'b000;
9     case (CurrentState)
10        STATE_2: begin
11            Status = 3'b010;
12        end
13        STATE_3: begin
14            Status = 3'b011;
15        end
16    endcase
17 end
```

Alternatively, the output assignment for `Status` can be combined into the `always@( * )` block that chooses what the next state should be (see Program 19). It is seperated here for clarity.

4.5 A Complete FSM

In this tutorial we hasve discussed why the Moore machine FSM is useful in digital design and how to create it in the Verilog HDL. We approached the problem in four different steps, namely defining an encoding (Section 4.1), establishing a way to store state (Section 4.2), creating a means of choosing between (possibly) more than one '`NextState`' (Section 4.3), and outputting both 1-bit and multi-bit output signals (all synchronous with the clock, Section 4.4). The final product, namely the FSM shown in Figure 3, is reproduced in its entirety in Program 21 (part 1) and Program 22 (part 2).[3] Again, you can alternatively combine the `case` statement used to assign the `Status` output into the `always@( * )` used to choose between possible `NextState` values. Seperating the two is personal preference.

[3]The final FSM in Verilog is split into two parts due to its size.

Now that we have constructed the final FSM, notice that it is packaged into a Verilog `module`. Each FSM in a design should have its own `module` for composition and organization purposes. An FSM `module` will always have `Clock` and `Reset` input signals, almost always have other input signals that manipulate the `NextState`, and always have `Output` signals based on the output. Besides these convention-imposed constraints, an FSM `module` is a normal Verilog `module`.

Rev.	Name	Date	Description
B	Chris Fletcher	9/4/08	Added Section 4.5 on the complete FSM.
A	Chris Fletcher	8/31/08	Wrote new tutorial.

Program 21 The complete FSM (part 1) from Figure 1

```
module BasicFsm(
        // ------------------------------------------------------------
        // Inputs
        // ------------------------------------------------------------
        input wire      Clock,
        input wire      Reset,
        input wire      A,
        input wire      B,
        // ------------------------------------------------------------

        // ------------------------------------------------------------
        // Outputs
        // ------------------------------------------------------------
        output wire      Output1,
        output wire      Output2,
        output reg[2:0]  Status
        // ------------------------------------------------------------
        );

// ------------------------------------------------------------------------
// State Encoding
// ------------------------------------------------------------------------
localparam STATE_Initial = 3'd0,
           STATE_1 = 3'd1,
           STATE_2 = 3'd2,
           STATE_3 = 3'd3,
           STATE_4 = 3'd4,
           STATE_5_PlaceHolder = 3'd5,
           STATE_6_PlaceHolder = 3'd6,
           STATE_7_PlaceHolder = 3'd7;
// ------------------------------------------------------------------------

// ------------------------------------------------------------------------
// State reg Declarations
// ------------------------------------------------------------------------
reg[2:0] CurrentState;
reg[2:0] NextState;
// ------------------------------------------------------------------------

// ------------------------------------------------------------------------
// Outputs
// ------------------------------------------------------------------------
// 1-bit outputs
assign Output1 = (CurrentState == STATE_1) | (CurrentState == STATE_2);
assign Output2 = (CurrentState == STATE_2);

// multi-bit outputs
always@( * ) begin
    Status = 3'b000;
    case (CurrentState)
        STATE_2: begin
            Status = 3'b010;
        end
        STATE_3: begin
            Status = 3'b011;
        end
    endcase
end
// ------------------------------------------------------------------------
```

Program 22 The complete FSM (part 2) from Figure 1

```
// ------------------------------------------------------------------------
// Synchronous State-Transition always@(posedge Clock) block
// ------------------------------------------------------------------------
always@(posedge Clock) begin
    if (Reset) CurrentState <= STATE_Initial;
    else CurrentState <= NextState;
end
// ------------------------------------------------------------------------

// ------------------------------------------------------------------------
// Conditional State-Transition always@( * ) block
// ------------------------------------------------------------------------
always@( * ) begin
    NextState = CurrentState;
    case (CurrentState)
        STATE_Initial: begin
            NextState = STATE_1;
        end
        STATE_1: begin
            if (A & B) NextState = STATE_2;
        end
        STATE_2: begin
                        if (A) NextState = STATE_3;
        end
        STATE_3: begin
            if (!A & B) NextState = STATE_Initial;
            else if (A & !B) NextState = STATE_4;
        end
        STATE_4: begin
        end
        STATE_5_PlaceHolder: begin
            NextState = STATE_Initial;
        end
        STATE_6_PlaceHolder: begin
            NextState = STATE_Initial;
        end
        STATE_7_PlaceHolder: begin
            NextState = STATE_Initial;
        end
    endcase
end
// ------------------------------------------------------------------------

endmodule
// ------------------------------------------------------------------------
```