EECS 270 Verilog Reference: Sequential Logic

1 Introduction

In the first few EECS 270 labs, your designs were based solely on combinational logic, which is logic that depends only on its current inputs. However, there are many cases in which we would like to implement functions that store some state—information about past inputs—and use that state to compute the functions' outputs. Logic that can store state is known as sequential logic. The major use of sequential logic, at least in the EECS 270 lab, is in the design of state machines. This document discusses general principles of state machine design and also takes you through an example to show the various features of state machine design in the Verilog HDL. It has two appendices—Appendix A shows the code for the arbiter, the example we use in our Verilog discussion, in full. Appendix B provides a quick reference to the techniques discussed within and the Verilog constructs you need.

2 State machine design

The state machines you must create in EECS 270 all follow the same general design methodology. As shown in the block diagram in Figure 1, you can break the design into three parts: the next state logic, the state memory, and the output logic. The next state logic is a combinational block responsible for implementing the transitions in the state diagram—based on the current state and the inputs, it determines what the next state should be. The state memory contains the flip-flops that store the current state of the system. At each clock edge, this block updates the state bits with the new values calculated in the next state logic. The output logic determines the output of the system. Depending on the type of state machine—Moore or Mealy—this logic may depend either on the current state alone or on the current state and the inputs. The block diagram shows a system designed in this manner with *N+1* bits of state, *M+1* input bits, and *P+1* output bits.

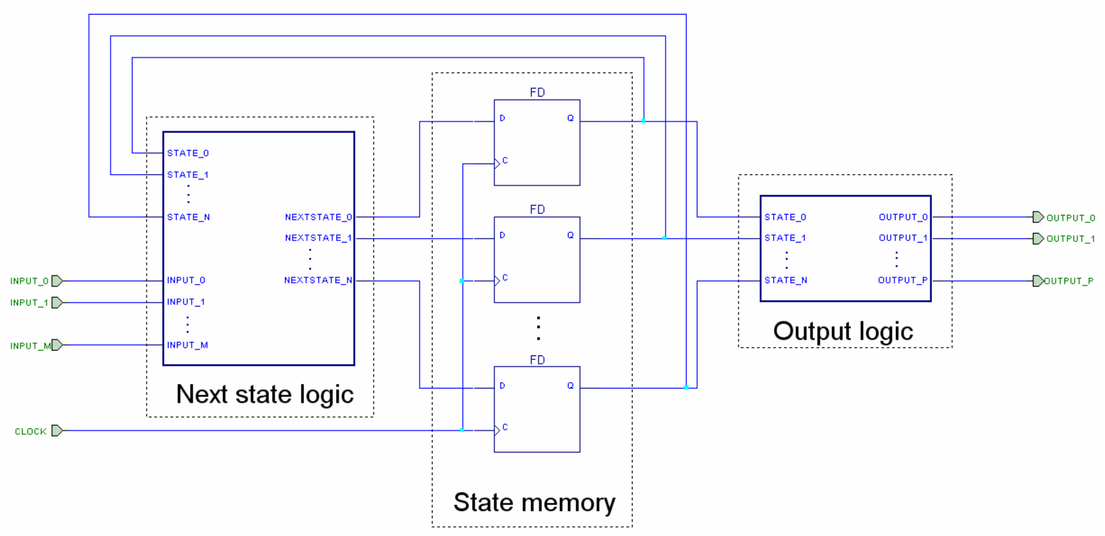


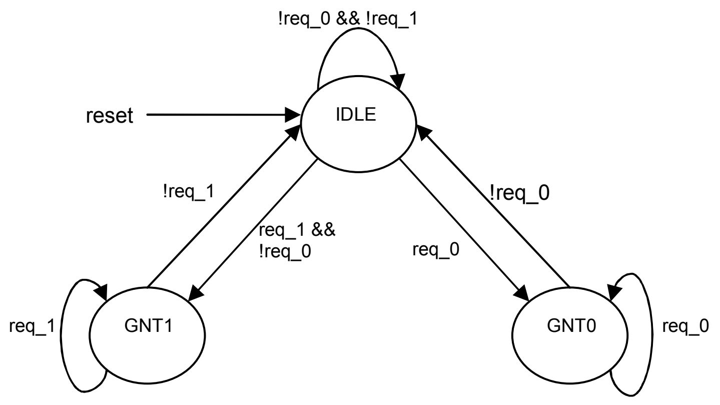
Figure 1: Block diagram for general state machine design

3 Implementing state machines in Verilog

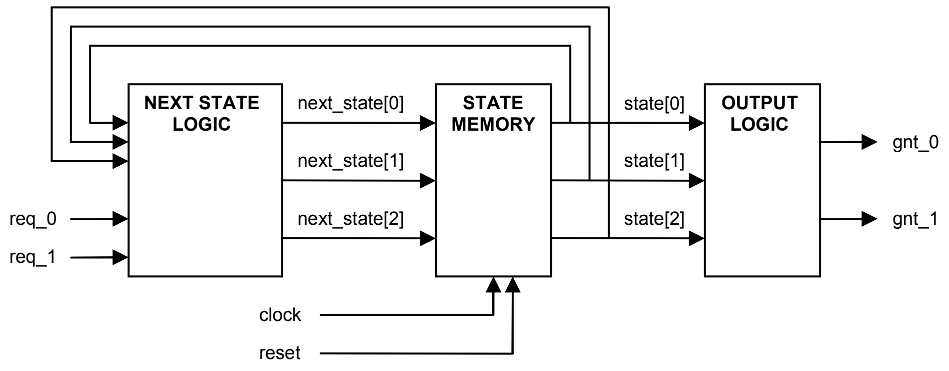
We will base most of our Verilog discussion on the example of an arbiter, which is adapted from Deepak Kumar Tala's site (http://www.asic-world.com/tidbits/fsm.html). The arbiter has four inputs—a clock, a reset input, and two request inputs, req_0 and req_1—and two grant outputs, gnt_0 and gnt_1. The circuit is designed to function as follows:

- When req_0 is asserted and req_1 is not asserted, gnt_0 is asserted
- When req_1 is asserted and req_0 is not asserted, gnt_1 is asserted
- When both req_0 and req_1 are asserted then gnt_0 is asserted; in other words, priority is given to req_0 over req_1.

A state diagram and block diagram for the arbiter are shown below. The code starts on the following page. Rather than discussing the whole arbiter module at once, we break it into the major sections—declarations, next state logic, sequential logic, and output logic—and highlight the features and design of each. Each block of code is provided with a short explanation of its functionality and is followed by a more detailed discussion. The code is shown in its entirety at the end of this document.



(a)



(b)

Figure 2: (a) State diagram and (b) block diagram for arbiter

3.1 Declarations

```
module arbiter (clock, reset, req_0, req_1, gnt_0, gnt_1);

input    clock, reset, req_0, req_1;
output   gnt_0, gnt_1;
reg gnt_0, gnt_1;

parameter   IDLE=3'b001;
parameter   GNT0=3'b010;
parameter   GNT1=3'b100;  .

reg [2:0] state;
reg [2:0] next_state;
```

The module declaration and input/output declarations are the same as in any other Verilog module.

We need to declare anything to which a value will be assigned inside of an always block as “reg” not “wire”.

This section is the place where you define a unique encoding for each state in your machine; the example uses a one-hot encoding for its three states.

You must also declare intermediate variables to hold the values of the current and next state—the inputs and outputs of the flip-flops that make up the state memory of each state machine

Figure 3: Declarations section of the arbiter module

As with all Verilog modules, your state machines require a declarations section at the beginning. This section of the code is where you declare your inputs, outputs, and intermediate variables; it will also be the area where you define your state encoding.

Notice that gnt_0 and gnt_1 are both “declared” twice. In this case they are listed as outputs, but also declared as type reg. Anything which will be assigned a value in an always block must be declared to be of type reg. Anything which will be assigned a value in an assign statement must be declared to be of type wire. As a default, outputs are assumed to be of type wire unless redeclared otherwise. As we will not be using assign statements in this example, everything will be of type reg.

One of the most important aspects of any state machine is the definition of the various states and the manner in which they are encoded. You can specify these state definitions by using parameters as shown in the code above. The example above uses a one-hot encoding for its state definitions; note that we could save some hardware by specifying the three states in 2 bits, as follows:

```
parameter    IDLE  =2'b00
parameter    GNT0 =2’b01
parameter    GNT1 =2’b10
```

This code uses two separate state variables, state and next_state. *Although it is declared as type reg, next_state is the output of the combinational logic that derives the next state from the current state and the inputs.* state is used in the actual sequential part of the circuit, holding the value of the current state for each cycle. You should follow this same practice—have one variable that represents the result of the next state logic, and one variable that actually stores the state. Note that the bit width of each of these signals matches the bit width of the encoded states.

3.2 Next state logic

```
always @*
begin
 case(state)
 IDLE :     if (req_0 == 1'b1)
                next_state = GNT0;
            else if (req_1 == 1'b1)
                next_state= GNT1;
            else
                next_state = IDLE;
 GNT0 :     if (req_0 == 1'b1)
                next_state = GNT0;
            else
                next_state = IDLE;
 GNT1 :     if (req_1 == 1'b1)
                next_state = GNT1;
            else
                next_state = IDLE;
 default:   next_state = IDLE;
 endcase
end
```

This example uses conditional statements (if and case) to implement combinational logic; each of these statements must be enclosed in an always block. An always @* block is used as combinational logic.

See below for an explanation of the syntax.

The next state logic is a combinational block used to derive the next values of the state bits from the current state and the inputs—essentially, this logic implements the transitions in the state diagram. For example, the first set of lines handle the cases where the arbiter is in state IDLE—if req_0 is 1, it transitions to GNT0, if req_1 is 1, it transitions to GNT1, and if neither input is 1, it remains in the same state, just as shown in Figure 2.

Figure 4: Next state logic section of arbiter module

The next state logic is a combinational block that determines the new values of the state bits from their current values and the values of the inputs. This section removes most of the complexity from the state memory, a region of code which should be as simple as possible. This implementation uses if and case statements to handle the state transitions; these statements and the details of their use are explained in the sub-sections below.

Note that you could implement the next state logic using assign statements because it is strictly combinational logic. This practice may be particularly useful in cases where it is relatively easy to derive the next state bits. The arbiter example is one such case, especially given the one-hot encoding of the states. We leave it up to you to verify the correctness of these equations, but the code above could be written as follows:

```
assign next_state[0] = (~req_0 & ~req_1 & state[0]) |
                              (~req_0 & state[1]) | (~req_1 & state[2]);
assign next_state[1] = req_0 & (state[0] | state[1]);
assign next_state[2] = req_1 & (state[0] | state[2]);
```

If we did rewrite the next state logic using assign statements, we would also have to change the declaration section and declare the variable next_state as a wire, not a reg. <u>Again, variables used on the left-hand side of assign statements must be wires, while variables used on the left-hand side of assignments within always blocks must be regs.</u>

3.2.1 Combinational logic via always blocks

Although combinational logic can always be written using assign statements, there are more powerful constructs—specifically, the if and case statements—that can be used to implement conditional assignments in a manner that may be more familiar to software programmers. We use both of these statements in the next state logic. There are some subtleties involved in using if and case statements because they must be enclosed in always blocks, as shown in the arbiter. The format of an always block is as follows:

```
always @ (<sensitivity list>)
begin
    <body of block>
end
```

We always will use "@(posedge clock)" for sequential logic and "@*" for combinational logic. There are other options. See the section on sensitivity lists (below).

Separate always blocks are evaluated, in parallel, throughout the execution of a simulation; they will synthesize to parallel blocks on the FPGA. As you will see in later sections, the sequential logic and output logic are also implemented in always blocks. Therefore, these three blocks will execute simultaneously. Within each block itself, however, execution will proceed sequentially, as in a typical C program, provided you use the blocking assignment operator (=). The non-blocking assignment operator (<=) is used strictly in sequential logic and is discussed in the next section.

The begin and end keywords act in a manner similar to the curly brackets, { and }, in many high-level programming languages. They are used to group multiple statements together and should be present for every always block. As noted above, variables assigned to inside always blocks should be declared as type reg, even if the block is used for combinational logic; these variables will synthesize to wires if they store no state.

3.2.2 Sensitivity lists

The sensitivity list is the tricky part of an always block. It must contain all of the variables to which the block is "sensitive;" in other words, if any of the variables in the sensitivity list change, the always block is re-evaluated. The "@*" notation simply says that if anything changes which could impact any result, the whole block should be re-evaluated. *For combinational logic you should always use @*.*

3.2.3 if and case statements

Within the body of an always block, you may use if and case statements. Their syntax is demonstrated in the arbiter example, but we give general forms for each below, beginning with the if statement:

```
if (<condition1>)
    <statement1>
else if (<condition2>)
    <statement2>
…
else
    <statement3>
```

The if statement functions almost exactly as in typical software programming languages. One major difference is that if a statement like <statement1> contains multiple assignments, those assignments must be enclosed by the begin and end keywords. An example is shown in the code for the output logic; see section 3.4. The case statement is shown next:

```
case (<variable>)
<case1>: <statement1>
<case2>: <statement2>
…
default: <statement_default>
endcase
```

Each of the cases (<case1>, <case2>, etc.) must be a literal or macro reference, as in the example. The keyword endcase denotes the end of the case statement. In a combinational always block, it is important that **any** variable you **ever** assign to is **always** assigned to. You should do this by specifying default behavior.[1] There are a number of ways to do this. One way to do this is to simply assign a value to every output before you do anything else. You can then override it later. (The last assignment

[1] Why this is important is a bit subtle. In a combinational circuit the output is, by definition, determined solely by the input. If you can go through a combinational always block without assigning a value to a certain output, it is unclear what that output should be. If your intent was to have it keep it's old value, you've just created a sequential logic block. Quartus will generate an "Always Construct Warning" if you do this.

encountered in the always block is the one that actually occurs.) Another way is to be sure that every explicitly list (as a “case” or an “if” condition) assigns values to all variables and that you have a default case that covers every possibility. In an if statement, this is done with an else statement after all if/else if statements. in a case statement, this is done with a default case.

If there is a path through your combinational always block that doesn’t assign to all outputs, you should get a warning that looks something like this:

```
Warning (10240): Verilog HDL Always Construct warning at test2.v(30): variable
"next_state" may not be assigned a new value in every possible path through the
Always Construct.  Variable "next_state" holds its previous value in every path with
no new value assignment, which may create a combinational loop in the current
design.
```

3.3 State memory

```
always @ (posedge clock)
begin
  if (reset == 1'b1)
     state <= IDLE;
  else
     state <= next_state;
end
```

The state memory block implements the flip-flops at the core of every state machine. Since we want the state memory to change only at the rising edge of the clock signal, we write the sensitivity list as shown here—the keyword posedge denotes a rising edge of the given signal. This block simply assigns the values of the next_state bits (calculated above) to the current state bits unless reset is 1, in which case the machine returns to its initial state.

Note the <= assignment operator used here. This operator is a blocking assignment and should be used in every sequential logic block. If you are implementing combinational logic in an always block, use the standard = operator for assignments.

This particular state memory block has a synchronous reset—the reset is evaluated only on the rising edge of the clock. Asynchronous resets are discussed below.

Figure 5: State memory section of arbiter module

The state memory block stores the state bits for your system. It ensures that your state transitions occur at regular intervals—usually at rising clock edges—and allows the state bits to remain stable while their next values are calculated in the next state logic.

One of the key differences between combinational logic in always blocks and sequential logic in always blocks is in the sensitivity list. When designing combinational logic, these lists effectively contain every variable that can possibly affect the assignments inside the given block. If any of those variables change, the whole block is evaluated.

For sequential logic, however, the timing of new assignments should almost always depend solely on the clock driving that logic. Since typical state machines store their state in D flip-flops, the logic updating that state should only change on rising clock edges. If you wanted to implement this block using negative edge-triggered flip-flops, you would change the posedge keyword to negedge.

3.3.1 Blocking assignments

Note that the assignments in this always block use <= instead of =. The <= operator is a non-blocking assignment operator; it should always be used in sequential logic blocks. Rather than executing sequentially, non-blocking assignments behave as follows:

- Within a given block, all right-hand sides for all assignments are evaluated.
- The new values are assigned simultaneously, not sequentially.

This behavior means that the following two pairs of assignments are not equivalent if they are each inside an always block:

```
a = 1;                        a <= 1;
b = a;                        b <= a;
```

Assume that a initially holds the value 0. After the left pair of assignments, b will have the value 1, because blocking assignments are used—first 1 is assigned to a, then a is assigned to b. In the right pair of assignments, b will have the value 0—when the right-hand sides are evaluated, a still has the value 0, so 0 is assigned to b and 1 is assigned to a.

3.3.2 Synchronous vs. asynchronous reset

In the arbiter example, the reset signal used in the sequential logic block is synchronous—since that block is only evaluated at each rising edge of the clock, the reset can only affect the output at the clock edge. If you want to implement an asynchronous reset signal, a reset that changes the output regardless of when the next clock edge is, you should change the first line of the sequential always block to read:

```
always @ (posedge clock or posedge reset)
```

Now, the block will be evaluated every time the reset changes from 0 to 1, as well as at the rising clock edges. Note that if the reset signal is still 1 at a subsequent rising clock edge, it will continue to hold the arbiter in the idle state. In general, you shouldn't use asynchronous resets in this class. You have to be very careful that there are no glitches on the reset line if you do chose to use it.

3.4 Output logic

```
always @*
begin
 case(state)
 IDLE :     begin
            gnt_0 = 1'b0;
            gnt_1 = 1'b0;
            end
 GNT0 :     begin
            gnt_0 = 1'b1;
            gnt_1 = 1'b0;
          end
 GNT1 :     begin
            gnt_0 = 1'b0;
            gnt_1 = 1'b1;
          end
 default :  begin
            gnt_0 = 1'b0;
            gnt_1 = 1'b0;
         end
 endcase
end
```

The output logic block is also combinational logic, but its structure depends on the type of state machine you implement. If you design a Moore machine, then the outputs depend solely on the current state. If you instead design a Mealy machine, then the outputs depend on the current state and the inputs.

Note the use of the begin and end keywords in each of the cases in the case statement. If you want to group multiple assignments together in a case or if statement, begin must precede the first assignment and end must follow the last one.

Figure 6: Output logic section of arbiter module

The output logic does exactly what its name implies—calculates the output of the system. In a Moore machine, this logic depends only on the current state. In a Mealy machine, this logic depends on both the current state and the inputs. Mealy machines may need to store additional state for their outputs (see the text for more details).

Since the arbiter is a Moore machine and its outputs are combinational, we can once again use assign statements to implement this logic just as we can with the next state logic. The equations for the outputs gnt_0 and gnt_1 are even simpler than the next state equations—gnt_0 is only 1 if the system is in state GNT0, and gnt_1 is only 1 if the system is in state GNT1. Once again, the one-hot encoding of the states makes the assign statements easy to write:

```
assign gnt_0 = state[1];
assign gnt_1 = state[2];
```

References

The original arbiter example can be found at http://www.asic-world.com/tidbits/fsm.html

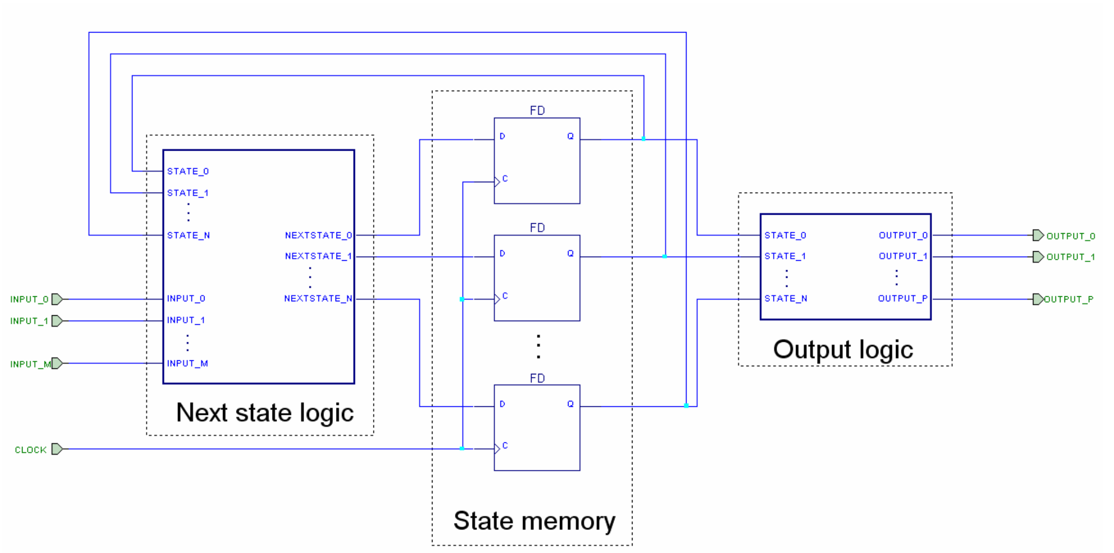
As with the combinational logic documentation, this document borrows from the EECS 470 Verilog reference material, particularly “Synthesizable Verilog Guidelines for EECS 470,” found at http://www.eecs.umich.edu/courses/eecs470/synth.pdf

Appendix A: Complete code example

```
module arbiter (clock, reset, req_0, req_1, gnt_0, gnt_1);

input clock, reset, req_0, req_1;         // Input declarations
output      gnt_0, gnt_1;                 // Output declarations
reg         gnt_0, gnt_1;

parameter  IDLE=3'b001;          // State definitions
parameter  GNT0=3'b010;          // This example uses a one-hot encoding for its three
parameter  GNT1=3'b100;          // states.  You can use a different encoding scheme.

reg [2:0] state;                 // Sequential variable to store the current state
reg [2:0] next_state;            // Combinational variable used to calculate the next state

always @*                        // Combinational logic block
begin
 case(state)
 IDLE :     if (req_0 == 1'b1)              // Implements transitions in state diagram
                  next_state = GNT0;        // (see Fig. 2 for state diagram)
            else if (req_1 == 1'b1)
            next_state= GNT1;
            else
            next_state = IDLE;
 GNT0 :     if (req_0 == 1'b1)
            next_state = GNT0;
            else
            next_state = IDLE;
 GNT1 :     if (req_1 == 1'b1)
                  next_state = GNT1;
            else
                  next_state = IDLE;
 default:   next_state = IDLE;
 endcase
end         // end of always @*

always @ (posedge clock)         // Sequential logic-implements flip-flops (with
begin                            // reset) to store value of current state; reset
 if (reset == 1'b1)              // returns machine to initial state
       state <= IDLE;
 else
       state <= next_state;
end         // end of always @ (posedge clock)

always @*        // Output logic-determines outputs from current state
begin
 case(state)
 IDLE :     begin
                  gnt_0 = 1'b0;
                  gnt_1 = 1'b0;
            end
 GNT0 :     begin
                  gnt_0 = 1'b1;
                  gnt_1 = 1'b0;
           end
 GNT1 :     begin
                  gnt_0 = 1'b0;
                  gnt_1 = 1'b1;
           end
 default :  begin
            gnt_0 = 1'b0;
                  gnt_1 = 1'b0;
            end
 endcase
end         // end of always @ (state)
endmodule
```

Appendix B: Quick reference guide

General state machine design



- Break Verilog code into next state logic, sequential logic, and output logic
- Next state logic
  - Determines next state of system from current state and inputs
  - Strictly combinational logic
  - Can be implemented with assign statements or with if/case statements inside always blocks
- Sequential logic
  - Flip-flops that store state of system
  - May incorporate reset signal that returns machine to initial state
- Output logic
  - Calculates system outputs
  - Moore machine → output logic depends solely on current state (shown above)
  - Mealy machine → output logic depends on current state and input bits

always blocks

- Format:

```
always @ (<sensitivity list>)
begin
   <body of block>
end
```

- Variables assigned within must be declared as type reg
- Sensitivity list
  - Variables in sensitivity list separated by or (not logical operator |)
  - In combinational logic should use @*
  - In sequential logic, sensitivity list should contain clock edge at which flip-flops are expected to change
    - posedge clock → positive edge-triggered flip-flops
    - negedge clock → negative edge-triggered flip-flops

        - If an asynchronous reset is used, sensitivity list should contain clock edge and posedge reset to ensure state changes as soon as reset is asserted. No such change is necessary for a synchronous reset.
- Assignments
    - Combinational always blocks should use blocking assignments: =
    - Sequential always blocks should use non-blocking assignments: <=

if statements
- Format

```
if (<condition1>)
     <statement1>
else if (<condition2>)
     <statement2>
…
else
     <statement3>
```

- Must be inside always blocks
- <condition1> … <condition*x*> must cover all possible values of variables checked in conditions to avoid "implying a latch"
    - Can cover remaining cases using else statement
- If any of the statements contain multiple assignments, surround assignments with begin and end keywords

case statements
- Format

```
case (<variable>)
<case1>: <statement1>
<case2>: <statement2>
…
default: <statement_default>
endcase
```

- Must be inside always blocks
- <case1> … <case*x*> must cover all possible values of <variable> to avoid "implying a latch"
    - Can cover remaining values using default case
- If any of the statements contain multiple assignments, surround assignments with begin and end keywords