Chapter 17

Factoring Finite State Machines

Factoring a state machine is the process of splitting the machine into two or more simpler machines. Factoring can greatly simplify the design of a state machine by separating orthogonal aspects of the machine into separate FSMs where they can be handled independently. The separate FSMs communicate via logic signals. One FSM provides input control signals to another FSM and senses its output status signals. Such factoring, if done properly, makes the machine simpler and also makes it easier to understand and maintain — by separating issues.

In a factored FSM, the state of each sub-machine represents one dimension of a multi-dimensional state space. Collectively the states of all of the sub-machines define the state of the overall machine — a single point in this state space. The combined machine has a number of states that is equal to the product of the number of states of the individual sub-machines — the number of points in the state space.\(^1\) With individual sub-machines having a few 10s of states, it is not unusual for the overall machine to have thousands to millions of states. It would be impractical to handle such a large number of states without factoring.

We have already seen one form of factoring in Section 16.3 where we developed a state machine with a data path component and a control component. In effect we factored the total state of the machine into a datapath portion and a control portion. Here we generalize this concept by showing how the control portion itself can be factored.

In this chapter we illustrate factoring by working two examples. In the first example, we start with a flat FSM and factor it into multiple simpler FSMs. In the second example we derive a factored FSM directly from the specification, without bothering with the flat FSM. Most real FSMs are designed using the latter method. A factoring is usually a natural outgrowth of the specification.

\(^1\)For most machines, not all points in the state space are reachable.
of a machine. It is rarely applied to an already flat machine.

### 17.1 A Light Flasher

Suppose you have been asked to design a light flasher. The flasher has a single input \( \text{in} \) and a single output \( \text{out} \). When \( \text{in} \) goes high (for one cycle) it initiates the flashing sequence. During this sequence output \( \text{out} \), which drives a light-emitting diode (LED) flashes three times. For each flash \( \text{out} \) goes high (LED on) for six cycles. Output \( \text{out} \) goes low for four cycles between flashes. After the third flash your FSM returns to the off state awaiting the next pulse on \( \text{in} \).

A state diagram for this light flasher is shown in Figure 17.1. The FSM contains 27 states six for each of the three flashes, four for each of the two periods between flashes, and one OFF state. We can implement this FSM with a big case statement, with 27 cases. However, suppose the specifications change to require 12 cycles for each flash, 4 flashes, and 7 cycles between flashes. Because this machine is flat any one of these changes would require completely changing the case statement.

We will illustrate the process of factoring with our light flasher. This machine
can be factored in two ways. First, we can factor out the counting of on and off intervals into a timer. This allows each of the six or four state sequences for a given flash or interval to be reduced to a single state. This factoring of timing both simplifies the machine and makes it considerably simpler to change the on and off intervals. Second, we can factor out the three flashes (even though the last is slightly different). This again simplifies the machine and also allows us to change the number of flashes.

Figure 17.2 shows how the flasher is factored to separate the process of counting time intervals into a separate machine. As shown in Figure 17.2(b), the factored machine consists of a master FSM that accepts in and generates out and a timer FSM. The timer FSM receives two control signals from the master and returns a single status signal. The tload control signal instructs the timer to load a value into a count-down timer (i.e., to start the timer), and tsel selects the value to be loaded. When tsel is high, the timer is set to count six cycles. The counter counts four cycles when tsel is low. When the counter has completed its count down the done signal is asserted and remains asserted until the counter is reloaded. A state diagram for this timer is shown
The timer is implemented as a datapath FSM as discussed in Section 16.1.3.

A state diagram of the master FSM is shown in Figure 17.2(a). The states correspond exactly to the states of Figure 17.1 except that each sequence of repeated states (e.g., A1 to A6) is replaced by a single state (e.g., A). The machine starts in the OFF state. This state repeatedly loads the timer with the count-down value for the on sequence ($tsel = 1$). When $in$ goes true, the FSM enters state A, $out$ goes high, and the timer begins counting down. The master FSM stays in state A waiting for the timer to finish counting and signal done. It will remain in this state for six cycles. During the last cycle in state A, done is true, so $tload$ is asserted ($tload = done$) and the counter is loaded with the count-down value for the off sequence ($tsel = 0$). Note that $tload$ is asserted only during this final cycle of state A, when done is true. If it was asserted during each cycle of state A the timer would continually reset and never reach done. Since done is true in this final cycle, the machine enters state B on the next cycle and out goes low. The process repeats in states B through E with each state waiting for done before moving to the next state. Each of these states (except E) loads the counter for the next state during its last cycle by specifying $tload = done$.

If we compare the FSM of Figure 17.2 to the flat FSM of Figure 17.1 we realize that the state of the flat machine has been separated. The portion of the state that represents the cycle count during the current flash or space (the numeric part of the state name, or the horizontal position in Figure 17.1) is held in the counter, while the portion of state that reflects which flash or space we are currently on (the alpha part of the state name, or the vertical position in Figure 17.1) is held in the master FSM. By decomposing the state into horizontal and vertical in this manner, we are able to represent all 27 states of the original machine with just two six-state machines.

Verilog code for the master FSM of Figure 17.2(a) is shown in Figure 17.3. The Flash module instantiates a state register and a timer (see Figure ??) and then uses a case statement to describe the combinational next state and output logic. For each of the six states a single concatenated assignment is used to set the output signal, $out$, the two timer controls, $tload$ and $tsel$, and the next state $next1$. Note that the values assigned to timer control $tload$ depend on the value of the timer status done. This is an example of a state machine where an input done directly affects an output $tload$ with no delay. In each state, the next state is determined with a $?:$ statement that waits for either $in$ or done before advancing to the next state. A final assign statement resets the machine to the OFF state.

For completeness the verilog code for the timer is shown in Figure 17.4. The approach taken is similar to that described in Section 16.1.3.

The waveform display from a simulation of the factored flasher of Figure 17.2 is shown in Figure 17.5. The output, the fourth line from the top, shows the

---

$^2$This state diagram only shows $tload$ being active in state DONE. In fact, our timer can be loaded in any state. The additional edges are omitted for clarity.
// define states for flash1
#define SWIDTH 3
#define S_OFF 3'b000 // off state
#define S_A 3'b001 // first flash
#define S_B 3'b010 // first space
#define S_C 3'b011 // second flash
#define S_D 3'b100 // second space
#define S_E 3'b101 // third and final flash

// Flash - flashes out three times 6 cycles on, 4 cycles off
// each time in is asserted.
module Flash(clk, rst, in, out);
input clk, rst, in; // in triggers start of flash sequence
output out; // out drives LED
reg out; // output
wire ['SWIDTH-1:0] state, next; // current state
reg ['SWIDTH-1:0] next1; // next state without reset
reg tload, tsel; // timer inputs
wire done; // timer output

// instantiate state register
DFF #('SWIDTH) state_reg(clk, next, state);

// instantiate timer
Timer1 timer(clk, rst, tload, tsel, done);

// next state and output logic
always @(state or rst or in or done) begin
  case(state)
    'S_OFF: {out, tload, tsel, next1} =
      {1'b0, 1'b1, 1'b1, in ? 'S_A : 'S_OFF };
    'S_A: {out, tload, tsel, next1} =
      {1'b1, done, 1'b0, done ? 'S_B : 'S_A };
    'S_B: {out, tload, tsel, next1} =
      {1'b0, done, 1'b1, done ? 'S_C : 'S_B };
    'S_C: {out, tload, tsel, next1} =
      {1'b1, done, 1'b0, done ? 'S_D : 'S_C };
    'S_D: {out, tload, tsel, next1} =
      {1'b0, done, 1'b1, done ? 'S_E : 'S_D };
    'S_E: {out, tload, tsel, next1} =
      {1'b1, done, 1'b1, done ? 'S_OFF : 'S_E };
    default: {out, tload, tsel, next1} =
      {1'b1, done, 1'b1, done ? 'S_OFF : 'S_E };
  endcase
end

// reset to off state
assign next = rst ? 'S_OFF : next1;
endmodule

Figure 17.3: Verilog description of the master FSM from Figure 17.2(a).
// define time intervals
// load 5 for 6-cycle interval 5 to 0.
#define T_WIDTH 3
#define T_ON 3'd5
#define T_OFF 3'd3

// Timer 1 - reset to done state. Load time when tload is asserted
// Load with T_ON if tsel, otherwise T_OFF. If not being loaded or
// reset, timer counts down each cycle. Done is asserted and timing
// stops when counter reaches 0.
module Timer1(clk, rst, tload, tsel, done);
  parameter n='T_WIDTH;
  input clk, rst, tload, tsel;
  output done;
  wire [n-1:0] count;
  reg [n-1:0] next_count;
  wire done;

  // state register
  DFF #(n) state(clk, next_count, count);

  // signal done
  assign done = !(|count);

  // next count logic
  always@(rst or tload or tsel or done or count) begin
    casex({rst, tload, tsel, done})
      4'b1xxx: next_count = 'T_WIDTH'b0;
      4'b011x: next_count = 'T_ON;
      4'b010x: next_count = 'T_OFF;
      4'b00x0: next_count = count - 1'b1;
      4'b00x1: next_count = count;
      default: next_count = count;
    endcase
  end
endmodule

Figure 17.4: Verilog description for the timer FSM used by the light flasher of Figure 17.2.
Figure 17.5: A waveform display showing a simulation of the factored light flasher of Figure 17.2.
desired three pulses with each pulse six clocks wide and spaces four clocks wide. The state of the master FSM is directly below the output, and the state of the timer is on the bottom line. The waveforms show how the master FSM stays in one state while the timer counts down — from 5 to 0 for flashes, and from 3 to 0 for spaces. The timer control lines (directly above the timer state) show how in states A-E (1-5) tload follows done.

We can factor our flasher further by recognizing that states A, C, and E are repeating the same function. The only difference is the number of remaining flashes. We can factor the number of remaining flashes out into a second counter as shown in Figure 17.6. Here the master FSM has only three states that determine whether the machine is off, in a flash, or in a space. The state that determines the position within a flash or space is held in a timer (just as in Figure 17.2. Finally, the state that determines the number of remaining flashes is held in a counter. Collectively the three FSMs, the master, the timer, and the counter, determine the total state of the factored machine. Each of the three sub-machines determines the state along one axis of a three-dimensional state space.

The state diagram of the master FSM for the doubly factored machine is shown in Figure 17.6(b). The machine has only three states. It starts in the OFF state. In the off state, both the timer and the counter are loaded. The timer is loaded with the count-down value for a flash. The counter is loaded with one less than the number of flashes required (i.e. the counter is loaded with a 3 for 4 flashes). Having input in go high causes a transition to the FLASH state. In the FLASH state the output out is true, the timer counts down, and the counter is idle. During the last cycle of the FLASH state the timer has reached its zero state and tdone is true. During this cycle the timer is reloaded with the count-down value for a space. With tdone true, the FSM proceeds from the FLASH state to the SPACE state if the counter is not done (cdone false). Otherwise, if this was the last flash (cdone true) the machine returns to the OFF state. In the SPACE state, the output is false, the timer counts down, and the counter is idle. In the final cycle of the SPACE state, tdone is true. This causes the counter to decrement, reducing the count of the number of remaining flashes, and the timer to reload with the count-down value for a flash.

The Verilog code for the doubly factored flasher of Figure 17.6 is shown in Figure 17.7 and the Verilog description of the counter module is shown in Figure 17.8. The next-state and output function for the master FSM again uses a concatenated assignment to set the next state, the output, and the four control signals with a single assignment. A nested ?: statement is used to compute the next state for the FLASH state to test both tdone and cdone. In several states, status signal tdone is passed directly through to control signals tload and cdec. The counter module is nearly identical to the timer module but with slightly different control because the counter only decrements when cdec is asserted while the timer always decrements. With some generalization a single parameterized module could be used for both functions.

The waveforms from a simulation of the doubly factored light flasher are shown in Figure 17.9. For this simulation, the counter was initialized with a
Figure 17.6: The light flasher of Figure 17.1 factored twice. The position within the current flash is held in a timer. The number of flashes remaining is held in a counter. Finally, whether we are off, in a flash, or a space is determined by the master FSM. (a) Block diagram of the twice factored machine. (b) State diagram of the master FSM.
// defines for doubly factored states
'define XWIDTH 2
'define X_OFF 2'b00
'define X_FLASH 2'b01
'define X_SPACE 2'b10

module Flash2(clk, rst, in, out);
    input clk, rst, in; // in triggers start of flash sequence
    output out; // out drives LED
    reg out; // output
    wire ['XWIDTH-1:0] state, next; // current state
    reg ['XWIDTH-1:0] next1; // next state without reset
    reg tload, tsel, cload, cdec; // timer and counter inputs
    wire tdone, cdone; // timer and counter outputs

    // instantiate state register
    DFF #(‘XWIDTH) state_reg(clk, next, state);

    // instantiate timer and counter
    Timer1 timer(clk, rst, tload, tsel, tdone);
    Counter1 counter(clk, rst, cload, cdec, cdone);

    always @(state or rst or in or tdone or cdone) begin
        case(state)
            'X_OFF: {out, tload, tsel, cload, cdec, next1} =
                {1'b0, 1'b1, 1'b1, 1'b1, 1'b0,
                in ? 'X_FLASH : 'X_OFF } ;
            'X_FLASH:{out, tload, tsel, cload, cdec, next1} =
                {1'b1, tdone, 1'b0, 1'b0, 1'b0,
                tdone ? (cdone ? 'X_OFF : 'X_SPACE) : 'X_FLASH } ;
            'X_SPACE:{out, tload, tsel, cload, cdec, next1} =
                {1'b0, tdone, 1'b1, 1'b0, tdone,
                tdone ? 'X_FLASH : 'X_SPACE } ;
            default:{out, tload, tsel, cload, cdec, next1} =
                {1'b0, tdone, 1'b1, 1'b0, tdone,
                tdone ? 'X_FLASH : 'X_SPACE } ;
        endcase
    end

    assign next = rst ? 'X_OFF : next1;
endmodule

Figure 17.7: Verilog description of the master FSM from Figure 17.6.
// defines for pulse counter
// load with 3 for four pulses
'thedefine C_WIDTH 2
'thedefine C_COUNT 3

// Counter1 - pulse counter
// cload - loads counter with C_COUNT
// cdec - decrements counter by one if not already zero
// cdone - signals when count has reached zero

module Counter1(clk, rst, cload, cdec, cdone) ;
    parameter n='C_WIDTH ;
    input clk, rst, cload, cdec ;
    output cdone ;
    wire [n-1:0] count ;
    reg [n-1:0] next_count ;
    wire cdone ;

    // state register
    DFF #(n) state(clk, next_count, count) ;

    // signal done
    assign cdone = !(|count) ;

    // next count logic
    always @(rst or cload or cdec or cdone or count) begin
        case({rst, cload, cdec, cdone})
            4'b1xxx: next_count = 'C_WIDTH'b0 ;
            4'b01xx: next_count = 'C_COUNT ;
            4'b0010: next_count = count - 1'b1 ;
            4'b00x1: next_count = count ;
            default: next_count = count ;
        endcase
    end
endmodule

Figure 17.8: Verilog description of the counter from Figure 17.6.
Figure 17.9: A waveform display showing a simulation of the doubly factored light flasher of Figure 17.6.
3 so that the master FSM produces 4 flashes. The different time scales of the three state variables are clearly visible. The counter (bottom line) moves most slowly, counting down from 3 to 0 over the four-flash sequence. It decrements after the last cycle of each space. At each point in time, it represents the number of flashes left to do after the current flash is complete. The master FSM state moves the next most slowly. After starting on the 0 (OFF) state, it alternates between the 1 (FLASH) state and the 2 (SPACE) state until the four flashes are complete. Finally, the timer state moves most rapidly counting down from 5 to 0 for flashes, or 3 to 0 for spaces.

17.2 Traffic Light Controller

As a second example of factoring, we consider a more sophisticated version of the traffic-light controller we introduced in Section 14.3. This machine has two inputs car_{ew} and car_{lt} that indicate that cars are waiting on the east-west road (ew) and that cars are waiting in a left-turn lane (lt). The machine has nine output lines that drive three sets of three lights each one set each for the north-south road, the east-west road, and the left-turn lane (from the north-south road). Each set of lights consists of a red light, a yellow light, and a green light.

Normally the light will be green for the north-south road. However, if a car is detected on the east-west road or the left-turn lane, we wish to switch the lights so that the east-west or left-turn light is green (with priority going to the left-turn lane). Once the lights are switched to east-west or left-turn, we leave them switched until either no more cars are detected in that direction, or until a timer expires. The lights then return to green in the north-south direction.

Each time we switch the lights we switch the lights in the active direction from green to yellow. Then, after a time interval, we switch them to red. Then, after a second time interval, we switch them to green. The lights are not allowed to change again until they have been green for a third time interval.

Given this specification we decide to factor the finite-state machine that implements this traffic-light controller into five modules as shown in Figure 17.10. A master FSM accepts the inputs and decides which direction should be green dir. To time when it is time to force the lights back to north-south, it uses a timer, Timer1. It also receives a signal ok back from the combiner that indicates that the sequence from the last direction change is complete and the direction is allowed to change again.

The combiner module maintains a current direction state, and combines this current direction state with the light signal from the light FSM to generate the 9 light outputs lights. The combiner also receives direction requests from the master FSM on the dir signal and sequences the light FSM in response to these requests. When a new direction request occurs, the combiner deasserts on (sets it low) to ask the light FSM to switch the lights to red. Once the lights are red, the current direction is set equal to the direction request and the on signal is asserted to request that the lights be sequenced to green. Only after
the done signal from the light FSM is asserted, signaling that the sequence is complete and the lights have been green for the required time interval is the ok signal asserted to allow another direction change.

The modules here illustrate two types of relationships. The master FSM and combiner modules form a pipeline. Requests flow down this pipeline from left to right. A request is input to the Master FSM in the form of a transition on the car_ew and car_lt inputs. The Master processes this request and in turn issues a request to the combiner on the dir lines. The combiner in turn processes the request and outputs the appropriate sequence on the lights output in response. We will discuss pipelines in more depth in Chapter 21.

The ok signal here is an example of a flow control signal. It provides back-pressure on the master FSM to prevent it from getting ahead of the combiner and light FSMs. The master FSM makes a request and it is not allowed to make another request until the ok signal indicates that the rest of the circuit is finished processing the first request.

The other relationships in Figure 17.10 are master-slave relationships. The Master FSM acts as a master to Timer1, giving it commands, and Timer1 acts as a slave, receiving the commands and carrying them out. In a similar manner the combiner is the master of the Light FSM which in turn is the master of Timer2.

The light FSM sequences the traffic lights from green to red and then back to green. It receives requests from the combiner on the on signal and responds to these requests with the done signal. It also generates the 3-bit light signal which indicates which light in the current direction should be illuminated. When
the on signal is set high, it requests that the light signal switch the light signal to green. When this is completed, and the minimum green time interval has elapsed, the light FSM responds by asserting done. When the on signal is set low, it requests that the light FSM sequence the light signal to red — via a yellow state and observing the required time intervals. When this is completed done is set low. The light FSM uses its own timer (Timer2) to count out the time intervals required for light sequencing.

For the interface between the light FSM and the combiner, the done signal provides flow control. The combiner can only toggle on high when done is low and can only toggle on low when done is high. After toggling on, it must wait for done to switch to the same state as on before switching on again.

A state diagram for the master FSM is shown in Figure 17.11. The machine starts in the NS state. In this state Timer1 is loaded so it can count down in the LT and EW states, and the requested direction is NS. State NS is exited when the ok signal indicates that a new direction can be requested and when one of the car signals indicates that there is a car waiting in another direction. In the EW and LT states, the new direction is requested and the state is exited with ok is true and either there is no longer a car in that direction or the timer has expired, as signaled by tdone.

The light FSM is a simple light sequencer similar to our light flasher of Section 17.1. A state diagram for the light FSM is shown in Figure 17.12. Like the light flasher, during the last cycle in each state the timer is loaded with the timeout for the next state. The machine starts in the RED state. It transitions to GREEN when the timer is done and the on signal from the combiner indicates...
that a green light is requested. The done signal is asserted after the timer in the GREEN state has completed its count down. The transition to YELLOW is triggered by the timer being complete and the on signal being low. The transition from YELLOW to RED occurs on the timer alone. The done signal is held high during the YELLOW state. It is allowed to go low, signaling that the transition to RED is complete, only after the timer has completed its countdown in the RED state.

A Verilog description of the master FSM for the factored traffic-light controller is shown in Figure 17.13. This module instantiates a timer and then uses a case statement to realize a three-state FSM that exactly follows the state diagram of Figure 17.11. Verilog select, "?:", statements are used for the next-state logic. A triply-nested select statement is used in state M_NS to test in turn signals ok, car_lt, and car_ew.

The master FSM does not instantiate the combiner. Both the master and combiner are instantiated as peer modules at the top level.

A Verilog description of the combiner module is shown in Figure 17.14. This module accepts a dir input from the master FSM, responds to the master FSM with flow-control signal ok, and generates the lights output. The key piece of state in the combiner module is the current direction register cur_dir. This holds the direction that the light FSM is currently sequencing. It is updated with the requested direction, dir, when on and done are both low. This occurs when the combiner has requested that the lights be sequence to red (on low), and the light FSM has completed this requested action (done low).

The on command, which requests that the light FSM to sequence the lights to green, is asserted any time the current direction and the requested direction match. When the master FSM requests a new direction, this causes on to go low - requesting the light FSM to sequence the lights red in preparation for the new direction.

The ok response to the master FSM is asserted when on and done are both true. This occurs when the light FSM has completed sequencing the requested direction to green.

The lights output is computed by a case statement with the current direction as the case variable. This case statement inserts the light output from
// master FSM states
// these also serve as direction values
'define MWIDTH 2
'define M_NS 2'b00
'define M_EW 2'b01
'define M_LT 2'b10

// Master FSM
// car_ew - car waiting on east-west road
// car_lt - car waiting in left-turn lane
// ok    - signal that it is ok to request a new direction
// dir   - output signaling new requested direction

module TLC_Master(clk, rst, car_ew, car_lt, ok, dir);
  input clk, rst, car_ew, car_lt, ok;
  output [1:0] dir;

  wire ['MWIDTH-1:0] state, next; // current state and next state
  reg ['MWIDTH-1:0] next1; // next state without reset
  reg tload; // timer load
  reg [1:0] dir; // direction output
  wire tdone; // timer completion

  // instantiate state register
  DFF #(MWIDTH) state_reg(clk, next, state);

  // instantiate timer
  Timer #(TWIDTH) timer(clk, rst, tload, 'T_EXP, tdone);

  always @(state or rst or car_ew or car_lt or ok or tdone) begin
    case(state)
      'M_NS: {dir, tload, next1} =
        ('M_NS, 1'b1, ok ? (car_lt ? 'M_LT :
          (car_ew ? 'M_EW : 'M_NS)) :
          'M_NS);
      'M_EW: {dir, tload, next1} =
        ('M_EW, 1'b0, (ok & (!car_ew | tdone)) ? 'M_NS : 'M_EW);
      'M_LT: {dir, tload, next1} =
        ('M_LT, 1'b0, (ok & (!car_ew | tdone)) ? 'M_NS : 'M_LT);
      default: {dir, tload, next1} =
        ('M_NS, 1'b0, 'M_NS);
    endcase
  end
  assign next = rst ? 'M_NS : next1;
endmodule

Figure 17.13: Verilog description of the master FSM for the traffic-light controller.
module TLC_Combiner(clk, rst, dir, ok, lights);
    input clk, rst;
    input [1:0] dir;
    output ok;
    output [8:0] lights;
    wire done;
    wire [2:0] light;
    reg [8:0] lights;
    wire [1:0] cur_dir;

    // request green from light FSM until direction changes
    wire on = (cur_dir == dir);

    // update direction when light FSM has made lights red
    wire [1:0] next_dir = rst ? 2'b0 : (!on & !done) ? dir : cur_dir;

    // ok to take another change when light FSM is done
    wire ok = on & done;

    // current direction register
    DFF #(2) dir_reg(clk, next_dir, cur_dir);

    // combine cur_dir and light to get lights
    always @(cur_dir or light) begin
        case(cur_dir)
            'M_NS: lights = {light, 'RED, 'RED};
            'M_EW: lights = {'RED, light, 'RED};
            'M_LT: lights = {'RED, 'RED, light};
            default: lights = {'RED, 'RED, 'RED};
        endcase
    end

    // light FSM
    TLC_Light lt(clk, rst, on, done, light);
endmodule

Figure 17.14: Verilog description of the combiner for the traffic-light controller.
the light FSM into the position corresponding to the current direction and sets the other positions to be red.

A Verilog description of the light FSM is shown in Figure 17.15. The module instantiates a timer and then uses a case statement to implement a FSM with state transitions as shown in Figure 17.12. Verilog select statements are used for the next-state logic.

Waveforms from a simulation of the factored traffic-light controller are shown in Figure 17.16. The machine is initially reset with the Master FSM in the NS state (00). Output dir is also NS (00). The ok line is initially low because the light FSM has not yet finished its sequencing to make the lights green in the NS direction. The lights are initially all red (444) but change after one cycle to be green in the north-south direction (144).

The light FSM is initialized to the RED (00) state and advances to the green state because on and tdone are both asserted. In the green state it starts a timer and waits until tdone is again asserted before signaling done to the combiner which in turn causes the combiner to signal ok to the Master FSM.

Because car_ew is asserted, once the ok signal goes high, the Master FSM requests a change in direction to east-west by setting dir to 01. The combiner responds by setting on low to request the light FSM to sequence the current direction’s light to red. The light FSM in turn responds by transitioning to the YELLOW state (10) which causes light to go to yellow (2) and lights to go to 244 - yellow in the north south direction. When the light timer completes its countdown, the light machine enters the RED state (00), light becomes 4 (RED), and lights becomes 444 - red in all directions. Once the light timer counts down the minimum time in the all-red state, the light FSM sets done low signaling that it has completed the transition.

With on and done both low, the combiner updates cur_dir to east-west (01) and sets on high to request the light FSM to sequence the lights green in the east-west direction. When the light FSM completes this action - including timing the green state - it sets done true. This in turn causes the combiner to set ok true, signaling the Master FSM that it is ready to accept a new direction.

When ok is asserted the second time, the Mater FSM requests the north-south direction again (dir = 00). This decision is driven by the car_ew line being low and the master timer being done. This new direction causes on to go low, sequencing the lights to the all red state. Then, after cur_dir is updated, setting on high to sequence them back to the green state. When this is all complete, ok is asserted again.

On this third assertion of ok, signal car_lt is true, so a left-turn is requested (dir = 10) and the lights are sequenced to red and back to green again. When the sequencing is completed, ok is asserted for a fourth time. This time car_lt is still asserted, but the master timer is done, so a north-south direction is again requested.
module TLC_Light(clk, rst, on, done, light);
input clk, rst, on;
output done;
output [2:0] light;
reg [2:0] light;
reg done;
wire [‘LWIDTH-1:0] state, next ; // current state, next state
reg [‘LWIDTH-1:0] next1 ; // next state w/o reset
reg tload;
reg [‘TWIDTH-1:0] tin;
wire tdone;

// instantiate state register
DFF #('LWIDTH) state_reg(clk, next, state);

// instantiate timer
Timer timer(clk, rst, tload, tin, tdone);

always @(state or rst or on or tdone) begin
  case(state)
    ‘L_RED: {tload, tin, light, done, next1} =
      {tdone & on, ‘T_GREEN, ‘RED, !tdone,
       (tdone & on) ? ‘L_GREEN : ‘L_RED};
    ‘L_GREEN: {tload, tin, light, done, next1} =
      {tdone & !on, ‘T_YELLOW, ‘GREEN, tdone,
       (tdone & !on) ? ‘L_YELLOW : ‘L_GREEN};
    ‘L_YELLOW: {tload, tin, light, done, next1} =
    default: {tload, tin, light, done, next1} =
  endcase
end

assign next = rst ? ‘L_RED : next1;
endmodule

Figure 17.15: Verilog description of the light FSM for the traffic-light controller.
Figure 17.16: A waveform display showing a simulation of the factored traffic-light controller.
17.3 Exercises

17–1 Flasher. Modify the flasher FSM to flash an SOS sequence - three short flashes (one clock each) followed by three long flashes (four clocks each) followed by three short flashes again.

17–2 Traffic Light Controller. Modify the traffic light controller so that north-south and east-west have equal priority. Add an additional input, car_ns that indicates when a car is waiting in the north-south direction. In either the NS or EW states, change to the other if there is a car waiting in the other direction and the master timer is done.

17–3 Traffic Light Controller. Modify the traffic light controller so that a switch from a red light to a green light is preceded by both red and yellow being on for three clocks.
Chapter 18

Microcode

Realizing the next-state and output logic of a finite-state machine using a memory array gives a flexible way of realizing a FSM. The function of the FSM can be altered by changing the contents of the memory. We refer to the contents of the memory array as microcode and a machine realized in this manner is called a microcoded FSM. Each word of the memory array determines the behavior of the machine for a particular state and input combination and is referred to as a microinstruction.

We can reduce the required size of a microcode memory by augmenting the memory with special logic to compute the next state, and by selectively updating infrequently changing outputs. A single microinstruction can be provided for each state, rather than one for each state × input combination, by providing an instruction sequencer and a branch microinstruction to cause changes in control flow. Bits of the microinstruction can be shared by different functions by defining different microinstruction types for control, output, and other functions.

Microcode was originated by Maurice Wilkes at Cambridge University in 1951 to implement the control logic for the EDSAC computer [?]. It has been widely used since that time in many different types of digital systems.

18.1 A Simple Microcoded FSM

Figure 18.1 shows a block diagram of a simple microcoded FSM. A memory array holds the next-state and output functions. Each word of the array holds the next state and output for a particular combination of input and next state. The array is addressed by the concatenation of the current state and the inputs. A pair of registers holds the current state and current output.

In practice the memory could be realized as a RAM or EEPROM allowing software to reprogram the microcode. Alternatively the memory could be a ROM. With a ROM, a new mask set is required to reprogram the microcode. However, this is still advantageous as changing the program of the ROM does not otherwise alter the layout of the chip. Some ROM designs even allow the
program to be changed by altering only a single metal-level mask — reducing
the cost of the change. Some designs take a hybrid approach putting most of
the microcode into ROM (to reduce cost) but keeping a small portion of microcode
in RAM. A method is provided to redirect an arbitrary state sequence into the
RAM portion of the microcode to allow any state to be patched using the RAM.

A verilog description of this FSM is shown in Figure 18.2. It follows the
schematic exactly with the addition of some logic to reset the state when rst
is asserted. The module ROM is a read-only-memory that takes an address
\{state, in\} and returns a microinstruction uinst. The microinstruction is
then split into its next-state and output components. The FSM of Figures 18.1
and 18.2 is very simple because it has no function until the ROM is programmed.

To see how to program the ROM to realize a finite state machine, consider
our simple traffic-light controller from Section 14.3. The state diagram of this
controller is repeated in Figure 18.3. To fill our microcode ROM, we simply
write down the next state and output for each current state/input combination
as shown in Table 18.1. Consider the first line of the table. Address 0000
corresponds to state GNS (green north-south) with input car_ew = 0. For this
state, the output is 100001 (green north-south, red east-west) and the next state
is GNS (000). Thus, the contents of ROM location 0000 is the 000100001, the
concatenation of the next state 000 with the output. For the second line of the
table, address 0001 corresponds to GNS with car_ew = 1. The output here is
the same as for the first line, but the next state is now YNS (001) hence the
contents of this ROM location is 001100001. The remaining rows of the table
are derived in a similar manner. The ROM itself is loaded with the contents of
the column labeled Data.

The results of simulating the microcoded FSM of Figure 18.2 using the
ROM contents from Table 18.1 are shown in Figure 18.4. The output, state,
and microcode ROM address, and microcode ROM data (microinstruction) (the
bottom four signals) are displayed in octal (base 8). The system initializes to
state 0 (GNS) with output 41 (green (4) north-south, red (1) east-west). Then
module ucode1(clk,rst,in,out) ;  
  parameter n = 1 ; // input width  
  parameter m = 6 ; // output width  
  parameter k = 3 ; // bits of state  

  input  clk, rst ;  
  input  [n-1:0] in ;  
  output [m-1:0] out ;  

  wire  [k-1:0] next, state ;  
  wire  [k+m-1:0] uinst ;  

  DFF #(k) state_reg(clk, next, state) ; // state register  
  DFF #(m) out_reg(clk, uinst[m-1:0], out) ; // output register  
  ROM #(n+k,m+k) uc({state, in}, uinst) ; // microcode store  
  assign next = rst ? {k{1'b0}} : uinst[m+k-1:m] ; // reset state  
endmodule

Figure 18.2: Verilog description of a simple microcoded FSM

Figure 18.3: State diagram of simple traffic-light controller.
the input line (car_ew) goes high switching the ROM address from 00 to 01. This causes the microinstruction to switch from 041 to 141 which selects a next state of 1 (YNS) on the next clock. The machine then proceeds through states 2 (GEW) and 3 (YEW) before returning to state 0.

The beauty of microcode is that we can change the function of our FSM by changing only the contents of the ROM. Suppose for example that we would like to modify the FSM so that:

1. The light stays green in the east-west direction as long as car_ew is true.

2. The light stays green in the north south direction for a minimum of three cycles (states GNS1, GNS2, and GNS3).

3. After a yellow light, the lights should go red in both directions for one cycle before turning the new light green.

Table 18.2 shows the state table that implements these modifications. We split state GNS into three states and add two new states (RNS and REW). Note that the GEW state now tests car_ew and stays in GEW as long as it is true. The results of simulating our FSM of Figure 18.2 with this new microcode is shown in the waveform display of Figure 18.5.

### 18.2 Instruction Sequencing

A microcoded FSM can be made considerably more efficient by using a *sequencer* to generate the address of the next instruction. Performing instruction sequencing has two big advantages. First, for microinstructions that simply proceed to the next instruction, this instruction address can be generated with a counter, eliminating the need to store the address in the microcode memory. Second, by using logic to select or combine the different inputs, the microcode store can store a single microinstruction for each state, rather than having to store separate (and nearly identical) instructions for each possible combination of inputs.
Figure 18.4: A waveform display showing a simulation of the microcoded FSM of Figure 18.2 using the microcode from Table 18.1.
Figure 18.5: A waveform display showing a simulation of the microcoded FSM of Figure 18.2 using the microcode from Table 18.2.
Table 18.2: State table for simple microcoded traffic-light controller

<table>
<thead>
<tr>
<th>Address</th>
<th>State</th>
<th>car_ew</th>
<th>Next State</th>
<th>Output</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>GNS1(000)</td>
<td>0</td>
<td>GNS2(001)</td>
<td>100001</td>
<td>001100001</td>
</tr>
<tr>
<td>0001</td>
<td>GNS1(000)</td>
<td>1</td>
<td>GNS2(001)</td>
<td>100001</td>
<td>001100001</td>
</tr>
<tr>
<td>0010</td>
<td>GNS2(001)</td>
<td>0</td>
<td>GNS3(010)</td>
<td>100001</td>
<td>010100001</td>
</tr>
<tr>
<td>0011</td>
<td>GNS2(001)</td>
<td>1</td>
<td>GNS3(010)</td>
<td>100001</td>
<td>010100001</td>
</tr>
<tr>
<td>0100</td>
<td>GNS3(010)</td>
<td>0</td>
<td>GNS3(010)</td>
<td>100001</td>
<td>010100001</td>
</tr>
<tr>
<td>0101</td>
<td>GNS3(010)</td>
<td>1</td>
<td>YNS (011)</td>
<td>1000001</td>
<td>011100001</td>
</tr>
<tr>
<td>0110</td>
<td>YNS (011)</td>
<td>0</td>
<td>RNS (100)</td>
<td>010001</td>
<td>100010001</td>
</tr>
<tr>
<td>0111</td>
<td>YNS (011)</td>
<td>1</td>
<td>RNS (100)</td>
<td>010001</td>
<td>100010001</td>
</tr>
<tr>
<td>1000</td>
<td>RNS (100)</td>
<td>0</td>
<td>GEW (101)</td>
<td>001001</td>
<td>101001001</td>
</tr>
<tr>
<td>1001</td>
<td>RNS (100)</td>
<td>1</td>
<td>GEW (101)</td>
<td>001001</td>
<td>101001001</td>
</tr>
<tr>
<td>1010</td>
<td>GEW (101)</td>
<td>0</td>
<td>YEW (110)</td>
<td>001100</td>
<td>110001100</td>
</tr>
<tr>
<td>1011</td>
<td>GEW (101)</td>
<td>1</td>
<td>GEW (101)</td>
<td>001100</td>
<td>110001100</td>
</tr>
<tr>
<td>1100</td>
<td>YEW (110)</td>
<td>0</td>
<td>REW (111)</td>
<td>001010</td>
<td>111001010</td>
</tr>
<tr>
<td>1101</td>
<td>YEW (110)</td>
<td>1</td>
<td>REW (111)</td>
<td>001010</td>
<td>111001010</td>
</tr>
<tr>
<td>1110</td>
<td>REW (111)</td>
<td>0</td>
<td>GNS (000)</td>
<td>001001</td>
<td>000001001</td>
</tr>
<tr>
<td>1111</td>
<td>REW (111)</td>
<td>1</td>
<td>GNS (000)</td>
<td>001001</td>
<td>000001001</td>
</tr>
</tbody>
</table>

A review of the microcode from Table 18.2 shows the redundancy a sequencer can eliminate. Each instruction includes an explicit next-state field while all instructions either select themselves or the next instruction in sequence for the next state. Also, all instructions are duplicated for both input states, with only two having a small difference in the next state field. This overhead would be even higher if there were more than one input signal.

Adding a sequencer to our microcoded FSM is the first step from an FSM (where the next state is determined by a logic function) to a stored-program computer, where the next instruction is determined by interpreting the current instruction. With a sequencer, like a stored program computer, our microcoded machine executes microinstructions in sequence until a branch instruction redirects execution to a new address.

Figure 18.6 shows a microcoded FSM that uses an instruction sequencer. The state register here is replaced with a microprogram counter (μPC or uPC) register. At any point in time this register represents the current state by selecting the current microinstruction. With this design we have reduced the number of microinstructions in the microcode memory from $2^{s+1}$ to $2^s$, that is from $2^s$ per state to 1 per state. The cost of this reduction is increasing the width of each microinstruction from $s+o$ bits to $s+o+b$ bits. Each microinstruction consists of three fields as shown in Figure 18.7: an $o$-bit field that specifies the current output, an $s$-bit field that specifies the address to branch to (the branch target), and a $b$-bit field that specifies a branch instruction.

With the instruction sequencer, inputs are tested by the branch logic as directed by a branch instruction from the current microinstruction. As a result
Figure 18.6: A microcoded FSM using an instruction sequencer. A multiplexer and incrementer compute the next microinstruction address (next state) based on a branch instruction and the input conditions.

Figure 18.7: Microinstruction format for the microcoded FSM with instruction sequencer of Figure 18.6.
Table 18.3: Branch instruction encodings.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>OpCode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>NOP</td>
<td>Never branch, always proceed to uPC+1</td>
</tr>
<tr>
<td>001</td>
<td>B0</td>
<td>Branch on input 0. If \text{in[0]} branch to \text{br_upc} otherwise continue to uPC+1</td>
</tr>
<tr>
<td>010</td>
<td>B1</td>
<td>Branch on input 1.</td>
</tr>
<tr>
<td>011</td>
<td>BA</td>
<td>Branch any. Branch if either input is true.</td>
</tr>
<tr>
<td>100</td>
<td>BR</td>
<td>Always branch. Select \text{br_upc} as the next uPC regardless of the inputs.</td>
</tr>
<tr>
<td>101</td>
<td>BN0</td>
<td>Branch on not input 0. If \text{in[0]} is false, branch, otherwise continue to uPC+1.</td>
</tr>
<tr>
<td>110</td>
<td>BNI</td>
<td>Branch on not input 1.</td>
</tr>
<tr>
<td>111</td>
<td>BNA</td>
<td>Branch only if inputs 0 and 1 are both false.</td>
</tr>
</tbody>
</table>

of this test, the sequencer either branches (by selecting the branch target field as the next uPC) or doesn’t (by selecting uPC+1 as the next UPC).

Consider an example with a two-bit input field. We can define a three-bit branch instruction \text{brinst} as follows.

\[
\text{branch} = (\text{brinst}[0] \& \text{in}[0] | \text{brinst}[1] \& \text{in}[1]) \oplus \text{brinst}[2] ;
\]

Branch instruction bits 0 and 1 select whether we test input bits 0 or 1 (or either). Branch instruction bit 2 controls the polarity of the test. If \text{brinst}[2] is low, we branch if the selected bit(s) is high. Otherwise we branch if the selected bit(s) is low. Using this encoding of the branch instruction, we can perform the branches shown in Table 18.3.

Other encodings of the branch instruction are possible. A common \(n\)-bit encoding uses \(n-1\)-bits to select one of \(2^{n-1}\) inputs to test and the remaining bit to select whether to branch on the selected input high or low. One of the inputs is set always high to allow the NOP and BR instructions to be created. For this encoding the branch signal would be:

\[
\text{branch} = \text{brinst}[n-1] \oplus \text{in[brinst[n-2:0]]} ;
\]

The branch instructions created by this alternate encoding (for a 3-bit \text{brinst} and three inputs are listed in Table 18.4. To provide the NOP and BR instructions, we use a constant 1 for the fourth input. Here each branch instruction tests exactly one input, while in the encoding of Table 18.3 the instructions may test zero, one, or two inputs. Many other possible encodings beyond to the two presented here are possible.

A Verilog description of the microcoded FSM with an instruction sequencer is shown in Figure 18.8. The Verilog follows the block diagram of Figure 18.6 closely. One assign statement calculates signal \text{branch} which is true if the sequencer is to branch on the next cycle. A second assign statement then calculates the next micro program counter (\text{nupc}) based on \text{branch} and \text{rst}.
<table>
<thead>
<tr>
<th>Encoding</th>
<th>OpCode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>B0</td>
<td>Branch on input 0.</td>
</tr>
<tr>
<td>001</td>
<td>B1</td>
<td>Branch on input 1.</td>
</tr>
<tr>
<td>010</td>
<td>B2</td>
<td>Branch on input 2.</td>
</tr>
<tr>
<td>011</td>
<td>BR</td>
<td>Always branch. (Input 3 is the constant &quot;1&quot;).</td>
</tr>
<tr>
<td>100</td>
<td>BN0</td>
<td>Branch on not input 0.</td>
</tr>
<tr>
<td>101</td>
<td>BN1</td>
<td>Branch on not input 1.</td>
</tr>
<tr>
<td>110</td>
<td>BN2</td>
<td>Branch on not input 2.</td>
</tr>
<tr>
<td>111</td>
<td>NOP</td>
<td>Never branch.</td>
</tr>
</tbody>
</table>

Table 18.4: Alternate branch instruction encodings.

```verilog
module ucode2(clk,rst,in,out) ;
    parameter n = 2 ; // input width
    parameter m = 9 ; // output width
    parameter k = 4 ; // bits of state
    parameter j = 3 ; // bits of instruction

    input clk, rst ;
    input [n-1:0] in ;
    output [m-1:0] out ;

    wire [k-1:0] nupc, upc ; // microprogram counter
    wire [j+k+m-1:0] uinst ; // microinstruction word

    // split off fields of microinstruction
    wire [m-1:0] nxt_out ; // = uinst[m-1:0] ;
    wire [k-1:0] br_upc ; // = uinst[m+k-1:m] ;
    wire [j-1:0] brinst ; // = uinst[m+j+k-1:m+k] ;
    assign {brinst, br_upc, nxt_out} = uinst ;

    DFF #(k) upc_reg(clk, nupc, upc) ; // microprogram counter
    DFF #(m) out_reg(clk, nxt_out, out) ; // output register
    ROM #(k,m+k+j) uc(upc, uinst) ; // microcode store

    // branch instruction decode

    // sequencer
    assign nupc = rst ? {k{1'b0}} : branch ? br_upc : upc + 1'b1 ;
endmodule
```

Figure 18.8: Verilog description of a microcoded FSM with an instruction sequencer
Table 18.5: Microcode for traffic light controller with sequencer of Figure 18.6.

To make the code more readable we use an assign statement to split out the three fields of the microinstruction into the output (nxt_out), the branch target (br_upc), and the branch instruction (brinst). Using these mnemonic names, rather than indexing fields of uinst makes the remainder of the code easier to follow.

Consider a slightly more involved version of our traffic-light controller that includes a left-turn signal as well as north-south and east-west signals. Table 18.5 shows the microcode. Here input 0 is car_lt and input 1 is car_ew so we rename our branches BLT (branch if car_lt), BNEW (branch if not car_ew) and so on.

The microcode of Table 18.5 starts in state NS1 where the light is green in the north-south direction. In this state the left-turn sensor is checked with a BLT to LT1. If car_lt is true, control transfers to LT1. Otherwise the uPC proceeds to the next state, NS2. In NS2 the microcode branches back to NS1 if car_ew is false (BNEW NS1). Otherwise control falls through to EW1 where the north-south light goes yellow. EW1 is always followed by EW2 where the east-west light is green. A BEW EW2 keeps the uPC in state EW2 as long as car_ew is true. When car_ew goes false, the uPC proceeds to state EW3 where the east-west light is yellow and a BR NS1 transfers control back to NS1. The left turn sequence (LT1, LT2, LT3) operates in a similar manner.

Waveforms from a simulation of the microcoded sequencer of Figure 18.8 running the microcode of Table 18.5 is shown in Figure 18.9. The fifth row from the top shows the microprogram counter upc. The machine is reset to upc = 0 (NS1) advances to 1 (NS2), then back to 0 (NS1) before branching to 5 (LT1). It proceeds from 5 (LT1) to 6 (LT2) and remains in 6 until car_lt goes low. At that point it advances to 7 (LT3) and back to 0 (NS1). At this point car_ew=1, and the sequence followed is 0, 1, 2, 3 (NS1, NS2, EW1, EW2). The machine stays in 3 (EW2) until car_ew goes low and then proceeds to 4 (EW3) and back to 0 (NS1). The machine cycles between NS1 and NS2 a few times until both car_ew and car_lt go high at the same time. As the machine is in NS1 when this happens, car_lt is checked first and the uPC is directed to LT1.

Because the microcoded FSM of Figure 18.6 can only branch one way in each microinstruction, it takes two states (NS1 and NS2) to perform a three-
Figure 18.9: A waveform display showing a simulation of the microcoded FSM of Figure 18.2 using the microcode from Table 18.2.
way branch between staying with north-south, going to east-west, or going to left-turn. This results in two states with the lights green in the north-south direction (NS1 and NS2) and two states with the lights yellow in the north-south direction (EW1 and LT1). The real solution to this problem is to support a multi-way branch (which we will discuss below). However we can partially solve the problem in software by using the alternate microcode shown in Table 18.6.

In the alternate microcode of Table 18.6, the uPC stays in state NS1 as long as \texttt{car\_ew} and \texttt{car\_lt} are both false by using a BNA NS1 (branch on not any inputs to NS1). NS1 is now the only state with the lights green in the north-south direction. If any inputs are true, the uPC proceeds to state NS2 which is the single state with the lights yellow in the north-south direction. State NS2 tests the \texttt{car\_lt} input and branches to state LT1 if true (BLT LT1). If \texttt{car\_lt} is false, the uPC proceeds to EW1. The remainder of the machine is similar to that of Table 18.5 except that the EW and LT states have been renumbered.

Simulation waveforms for this alternate microcode are shown in Figure 18.10.


table
<table>
<thead>
<tr>
<th>Address</th>
<th>State</th>
<th>Br Inst</th>
<th>Target</th>
<th>NS LT EW</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>NS1</td>
<td>BNA (111)</td>
<td>NS1(0000) 100001001</td>
<td>1110000100010001001</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>NS2</td>
<td>BLT (001)</td>
<td>LT1(0100) 010001001</td>
<td>0010100010001001001</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>EW1</td>
<td>BEW (010)</td>
<td>EW1(0010) 001001100</td>
<td>0100010001001001001</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>EW2</td>
<td>BR (100)</td>
<td>NS1(0000) 001001010</td>
<td>0010100010001001001</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>LT1</td>
<td>BLT (001)</td>
<td>LT1(0100) 001100001</td>
<td>0010100011000001001</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>LT2</td>
<td>BR (100)</td>
<td>NS1(0000) 001010001</td>
<td>10000000001010001001</td>
<td></td>
</tr>
</tbody>
</table>

Table 18.6: Alternate microcode for traffic light controller with sequencer of Figure 18.6.

18.3 Multi-way Branches

As we saw in the previous section, using an instruction sequencer greatly reduces the size of our microcode memory but at the expense of restricting each state to have at most two next states (upc+1 and br_upc). This restriction can be a problem if we have an FSM that has a large number of exits from a particular state. For example, in a microcoded processor, it is typical to branch to one of 10s to 100s of next states based on the \texttt{opcode} of the current instruction. Another multi-way branch is then needed on the addressing mode of the instruction. Implementing such a multi-way dispatch using the sequencer of Section 18.2 would result in very poor efficiency as \( n \) cycles would be required to test for \( n \) different opcodes.

We can overcome this limitation of two-way branches by using an instruction sequencer that supports multi-way branches as shown in Figure 18.11. This sequencer is similar to that of Figure 18.6 except that the branch target, \texttt{br_upc}, is generated from the branch instruction, \texttt{brinst}, and inputs rather than being provided directly from the microinstruction. With this approach, we can branch
Figure 18.10: A waveform display showing a simulation of the microcoded FSM of Figure 18.2 using the microcode from Table 18.2.
Figure 18.11: A microcoded FSM with an instruction sequencer that supports multi-way branches.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRx</td>
<td>00xx</td>
<td>Branch on condition x. (As in Table 18.4 includes BR)</td>
</tr>
<tr>
<td>BRNx</td>
<td>01xx</td>
<td>Branch on not condition x. (As in Table 18.4 includes NOP)</td>
</tr>
<tr>
<td>BR4</td>
<td>1000</td>
<td>Branch four ways. nupc = br_upc + in.</td>
</tr>
</tbody>
</table>

Table 18.7: Branch instructions for microcoded FSM supporting multi-way branches.

To up to $2^i$ next states (one for each input combination) from each state.

The branch instruction encodes not just the condition to test, but also how to determine the branch target. Table 18.7 shows one possible method of encoding multi-way branch instructions. The BRx and BRNx instructions are two-way branch instructions identical to those specified in Table 18.4. The BR4 instruction is a four-way branch that selects one of four adjacent states (from br_upc to br_upc+3) depending on the input.

To use the BR4 instruction requires some care in mapping states to microinstruction addresses and may require that some states are duplicated. Consider, for example, the state diagram of Figure 18.12. A mapping of this state diagram to microcode addresses for a machine with a four-way branch instruction that adds the input to the branch target is shown in Table 18.8. The four-way branch from X targets address 000, so we must lay out branch targets A1, B1, C1, and X in locations 000, 001, 010, and 011 respectively. In a similar manner we locate states so that the four-way branch from C1 targets address 100. Thus we must place states C2, C3, X, and C1 at locations 100, 101, 110, and 111 respectively. To make this work we need two copies of X, one at 011 and one at
18.4 Multiple Instruction Types

So far we have considered microcoded FSMs that update all output bits in every microinstruction. In general, most FSMs need to update only a subset of the outputs in a given state. Our traffic-light controller FSMs, for example, change at most one light on each state change. We can save bits of the microinstruction by modifying our FSM to update only a single output register in any given state. We waste other microinstruction bits by specifying a branch instruction and branch target in each microinstruction even though many microinstructions...
always proceed to the next state and don’t branch. We can save these redundant branch bits by only branching in some instructions and updating outputs in other instructions.

Figure 18.13 shows the instruction formats for a microcoded FSM with two microinstruction types: a branch instruction and a store (output) instruction. Each microinstruction is one or the other. A branch instruction, identified by a 1 in the left-most bit, specifies a branch condition and a branch target. When the FSM encounters a branch microinstruction, it branches (or doesn’t) as specified by the condition and target. No outputs are updated. A store instruction, identified by a 0 in the left-most bit, specifies an output register and a value. When the FSM encounters a store microinstruction it stores the value to the specified output register and then proceeds to the next microinstruction in sequence. No branching takes place with a store instruction.

Figure 18.14 shows a block diagram of a microcoded FSM that supports the two instruction types of Figure 18.13. Each microinstruction is split into an x-bit instruction field and a s-bit value field. The instruction field holds the operation-code (opcode) bit (the left-most bit that distinguishes between branch and store) and either the condition (for a branch) or the destination (for
The value field holds either the branch target (for a branch) or the new output value (for a store).

The instruction sequencer of Figure 18.14 is identical to that of Figure 18.6 except that the branch logic always selects the next microinstruction \( \text{upc} + 1 \) when the current microinstruction is a store instruction.\(^1\) The major difference is with the output logic. Here a decoder enables at most one output register to receive the value field on a store instruction.

A Verilog description for the microcode engine with two instruction types of Figure 18.14 is shown in Figure 18.15. Here the microinstruction is split into an opcode (0=store, 1=branch), an instruction (destination for store, condition for branch), and a value. For a store, the destination is decoded into a one-hot enable vector, \( e \), that is used to enable the value to be stored into one of the three output registers (ns=0, ew=1, lt=2) or to load the timer (dest = 3). For a branch, \( \text{inst}[2] \) determines the polarity of the branch, and the low two bits, \( \text{inst}[1:0] \), determine the condition to be tested (lt=0, ew=1, lt|ew=2, timer=3).

Table 18.9 shows the microcode for a more sophisticated traffic-light controller programmed on the microcode engine of Figure 18.15. The first three states load the three output registers with RED in the east-west and left-turn registers and GREEN in the north-south register. Next states NS1 and NS2 wait for eight cycles by loading the timer and waiting for it to signal done. State NS4 then waits for either input. The north-south light is set to YELLOW in NS5. NS6 and NS7 then set the timer and wait for it to be done before advancing to NS8 where the north-south light is set to RED. If the left-turn input is true NS9 branches to LT1 to sequence the left-turn light. Otherwise the east-west lights are sequenced in states EW1 through EW9. Waveforms from a simulation of this microcode are shown in Figure 18.16.

### 18.5 Microcode Subroutines

The state sequences in Table 18.9 are very repetitive. The NS, EW, and LT sequences perform largely the same actions. The only salient difference is the output register being written. Just as we shared common state sequences by factoring FSMs in Chapter 17, we can share common state sequences in a microcoded FSM by supporting subroutines. A subroutine is a sequence of instructions that can be called from several different points and after exiting returns control to the point from which it was called.

Figure 18.17 shows the block diagram of a microcode engine that supports one level of subroutines. This machine is identical to that of Figure 18.14 except for two differences: (a) a return uPC register, \( r\text{upc} \) and associated logic have been added to the sequencer, and (b) a select register and associated logic have been added to the output section.

---

\(^1\)One can just as easily add multiple instruction types and output registers to a FSM that supports multi-way branches (Figure 18.11).
module ucodeMI(clk,rst,in,out) ;
  parameter n = 2 ; // input width
  parameter m = 9 ; // output width
  parameter o = 3 ; // output sub-width
  parameter k = 5 ; // bits of state
  parameter j = 4 ; // bits of instruction
  input clk, rst ;
  input [n-1:0] in ;
  output [m-1:0] out ;

  wire [k-1:0] nupc, upc ; // microprogram counter
  wire [j+k-1:0] uinst ; // microinstruction word
  wire done ; // timer done signal

  // split off fields of microinstruction
  wire opcode ; // opcode bit
  wire [j-2:0] inst ; // condition for branch, dest for store
  wire [k-1:0] value ; // target for branch, value for store
  assign {opcode, inst, value} = uinst ;

  DFF #(k) upc_reg(clk, nupc, upc) ; // microprogram counter
  ROM #(k,k+j) uc(upc, uinst) ; // microcode store

  // output registers and timer
  DFFE #(o) or0(clk, e[0], value[o-1:0], out[o-1:0]) ; // NS
  DFFE #(o) or1(clk, e[1], value[o-1:0], out[2*o-1:o]) ; // EW
  DFFE #(o) or2(clk, e[2], value[o-1:0], out[3*o-1:2*o]) ; // LT
  Timer #(k) tim(clk, rst, e[3], value, done) ; // timer

  // enable for output registers and timer
  wire [3:0] e = opcode ? 4'b0 : 1<<inst ;

  // branch instruction decode
  wire branch = opcode ? (inst[2] & (inst[1:0] == 0) & in[0]) | // BLT
    ((inst[1:0] == 1) & in[1]) | // BEW
    ((inst[1:0] == 2) & (in[0]|in[1])) | // BLE
    ((inst[1:0] == 3) & done)) // BTD
    : 1'b0 ; // for a store opcode

  // microprogram counter
  assign nupc = rst ? {k{1'b0}} : branch ? value : upc + 1'b1 ;
endmodule

Figure 18.15: Verilog description of a microcoded FSM with two instruction types.
<table>
<thead>
<tr>
<th>Address</th>
<th>State</th>
<th>Instruction</th>
<th>Value</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>RST1</td>
<td>STT (0010)</td>
<td>RED 001</td>
<td>0010000001</td>
</tr>
<tr>
<td>00001</td>
<td>RST2</td>
<td>SEW (0001)</td>
<td>RED 001</td>
<td>0010000001</td>
</tr>
<tr>
<td>00010</td>
<td>NS1</td>
<td>SNS (0000)</td>
<td>GREEN 100</td>
<td>0000000100</td>
</tr>
<tr>
<td>00011</td>
<td>NS2</td>
<td>STIM(0011)</td>
<td>TGRN 01000</td>
<td>0011010000</td>
</tr>
<tr>
<td>00010</td>
<td>NS3</td>
<td>BNTD(1111)</td>
<td>NS3 00100</td>
<td>1111001000</td>
</tr>
<tr>
<td>00101</td>
<td>NS4</td>
<td>BNLE(1110)</td>
<td>NS4 00101</td>
<td>1110001001</td>
</tr>
<tr>
<td>00110</td>
<td>NS5</td>
<td>SNS (0000)</td>
<td>YELLOW 010</td>
<td>0000000101</td>
</tr>
<tr>
<td>00111</td>
<td>NS6</td>
<td>STIM(0011)</td>
<td>TYEL 00011</td>
<td>0011000111</td>
</tr>
<tr>
<td>01000</td>
<td>NS7</td>
<td>BNTD(1111)</td>
<td>NS7 01000</td>
<td>1111010000</td>
</tr>
<tr>
<td>01001</td>
<td>NS8</td>
<td>SNS (0000)</td>
<td>RED 0001</td>
<td>0000000011</td>
</tr>
<tr>
<td>01010</td>
<td>NS9</td>
<td>BLT (1000)</td>
<td>LT1 10100</td>
<td>1000101000</td>
</tr>
<tr>
<td>01101</td>
<td>EW1</td>
<td>STIM(0011)</td>
<td>TRED 00010</td>
<td>0011000100</td>
</tr>
<tr>
<td>01100</td>
<td>EW2</td>
<td>BNTD(1111)</td>
<td>EW2 01100</td>
<td>1111011000</td>
</tr>
<tr>
<td>01101</td>
<td>EW3</td>
<td>SEW (0001)</td>
<td>GREEN 100</td>
<td>0001000100</td>
</tr>
<tr>
<td>01110</td>
<td>EW4</td>
<td>STIM(0011)</td>
<td>TGRN 01000</td>
<td>0011010000</td>
</tr>
<tr>
<td>01111</td>
<td>EW5</td>
<td>BNTD(1111)</td>
<td>EW5 01111</td>
<td>1111011111</td>
</tr>
<tr>
<td>10000</td>
<td>EW6</td>
<td>SEW (0001)</td>
<td>YELLOW 010</td>
<td>0010000101</td>
</tr>
<tr>
<td>10001</td>
<td>EW7</td>
<td>STIM(0011)</td>
<td>TYEL 00011</td>
<td>0011000111</td>
</tr>
<tr>
<td>10010</td>
<td>EW8</td>
<td>BNTD(1111)</td>
<td>EW8 10010</td>
<td>1111100100</td>
</tr>
<tr>
<td>10011</td>
<td>EW9</td>
<td>BTD (1011)</td>
<td>RST2 00001</td>
<td>1011000010</td>
</tr>
<tr>
<td>10100</td>
<td>LT1</td>
<td>STIM(0011)</td>
<td>TRED 00010</td>
<td>0011000101</td>
</tr>
<tr>
<td>10101</td>
<td>LT2</td>
<td>BNTD(1111)</td>
<td>LT2 10101</td>
<td>1111101010</td>
</tr>
<tr>
<td>10110</td>
<td>LT3</td>
<td>SLT (0010)</td>
<td>GREEN 100</td>
<td>0010000100</td>
</tr>
<tr>
<td>10111</td>
<td>LT4</td>
<td>STIM(0011)</td>
<td>TGRN 01000</td>
<td>0011010000</td>
</tr>
<tr>
<td>11000</td>
<td>LT5</td>
<td>BNTD(1111)</td>
<td>LT5 11000</td>
<td>1111100000</td>
</tr>
<tr>
<td>11001</td>
<td>LT6</td>
<td>SLT (0010)</td>
<td>YELLOW 010</td>
<td>0010000100</td>
</tr>
<tr>
<td>11010</td>
<td>LT7</td>
<td>STIM(0011)</td>
<td>TYEL 00011</td>
<td>0011000111</td>
</tr>
<tr>
<td>11011</td>
<td>LT8</td>
<td>BNTD(1111)</td>
<td>LT8 10010</td>
<td>1111111111</td>
</tr>
<tr>
<td>11100</td>
<td>LT9</td>
<td>BTD (1011)</td>
<td>RST1 00000</td>
<td>1011000000</td>
</tr>
</tbody>
</table>

Table 18.9: Microcode to implement traffic-light controller on FSM with two instruction types.
Figure 18.16: A waveform display showing a simulation of the microcoded FSM of Figure 18.15 using the microcode from Table 18.9.
The $rupc$ register is used to hold the $upc$ to which a subroutine should branch when it completes. When a subroutine is called, the branch target is selected as the next $upc$ and $upc+1$, the address of the next instruction in sequence is saved in the $rupc$ register. A special branch instruction CALL is used to cause the enable line to the $rupc$ register, $erpc$, to be asserted. When the subroutine is complete, it returns control to the saved location using another special branch instruction RET to select the $rupc$ as the source of the next $upc$.

The select register is used to allow the same state sequence to write to different output registers when called from different places. A two-bit register identifier ($ns=0$, $ew=1$, $lt=2$) can be stored in the select register. A special store instruction $SSEL$ can then be used to store to the register specified by the select register (rather than by the destination bits of the instruction). Thus, the main program can store 0 (ns) into the select register and then call a subroutine to sequence the north-south lights on and off. The program can then store 1 (ew) into the select register and call the same subroutine to sequence the east-west lights on and off. The same subroutine can sequence different lights because it performs all of its output using the SSEL instruction.

EXAMPLE CODE HERE

18.6 A Simple Computer

18.7 Bibliographic Notes

18.8 Exercises

microcode sequence detector