Chapter 14

Sequential Logic

The output of sequential logic depends not only on its input, but also on its *state* which may reflect the history of the input. We form a sequential logic circuit via feedback - feeding state variables computed by a block of combinational logic back to its input. General sequential logic, with asynchronous feedback, can become complex to design and analyze due to multiple state bits changing at different times. We simplify our design and analysis tasks in this chapter by restricting ourselves to *synchronous* sequential logic, in which the state variables are held in a register and updated on reach rising edge of a clock signal.[1]

The behavior of a synchronous sequential logic circuit or *finite-state machine* (FSM) is completely described by two logic functions: one which computes its next state as a function of its input and present state, and one that computes its output - also as a function of input and present state. We describe these two functions by means of a *state table*, or graphically with a *state diagram*. If states are specified symbolically, a state assignment maps the symbolic states onto a set of bit vectors - both binary and one-hot state assignments are commonly used.

Given a state table (or state diagram) and a state assignment, the task of implementing a finite-state machine is a simple one of synthesizing the next-state and output logic functions. For a one-hot state encoding, the synthesis is particularly simple as each state maps to a separate flip-flop and all edges in the state diagram leading to a state map into a logic function on the input of that flip-flop. For binary encodings, Karnaugh maps for each bit of the state vector are written and reduced to logic equations.

Finite-state machines are implemented in Verilog by declaring a state register to hold the current state, and describing the next-state and output functions with combinational logic descriptions, such as `case` statements as described in Chapter 7. State assignments should be specified with `` `define `` statements to allow them to be changed without altering the machine description itself. Special attention should be given to resetting the FSM.

[1]We revisit asynchronous sequential circuits in Chapter 22.

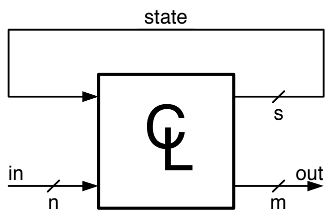


Figure 14.1: Sequential circuits are formed when feedback paths carrying state information are added to combinational circuits. The output of a sequential circuit depends on both the current input and on the state, which is a function of previous inputs.

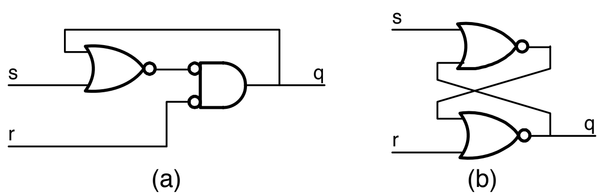


Figure 14.2: An RS flip-flop is a simple example of a sequential circuit: (a) schematic (b) alternate schematic that doesn't obey the *bubble rule*.

14.1 Sequential Circuits

Recall that a combinational circuit produces an output that depends only on the current state of its input. Recall also that combinational circuits must be acyclic. If we add feedback to a combinational circuit, creating a cycle as shown in Figure 14.1, the circuit becomes sequential. The output of a sequential circuit depends not only on its current input, but also on the history of previous inputs. The cycle created by the feedback allows the circuit to store information about its previous input. We refer to the information stored on the feedback signals as the state of the circuit.

A sequential circuit generates an output that is a function of both its input and its present state. It also generates its next state, also as a function of its input and its present state.

Figure 14.2 shows a reset-set (RS) flip-flop, a very simple sequential logic circuit that is composed of two NOR gates.[2] The output q is fed back to the input as a state variable. The circuit's behavior is described by the equation: $q = \overline{r} \wedge (s \vee q)$. The state variable q appears on both sides of the equation. To make the dynamics clearer we rewrite this as $q_{\text{new}} = \overline{r} \wedge (s \vee q_{\text{old}})$. That is, the

[2] We draw the circuit as shown in Figure 14.2(a). Many people (who do not obey the bubble rule) draw it as shown in Figure 14.2(b).

r	s	q_{old}	q_{new}
0	0	0	0
0	0	1	1
0	1	X	1
1	X	X	0

Table 14.1: State table for RS flip-flop.

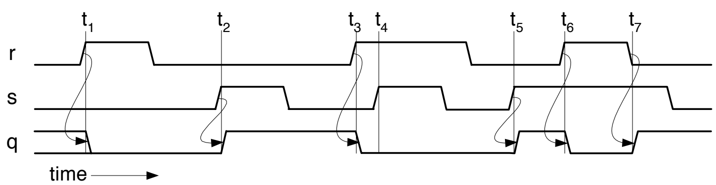


Figure 14.3: Timing diagram showing operation of the RS flip-flop. The value of signals is shown as time advances from left to right.

equation tells us how to derive the *new* state of q as a function of the inputs and the *old* state of q.

From the equation (and the schematic) it is easy to see that if $r = 1$, $q = 0$ and the flip-flop is reset, if $s = 1$ and $r = 0$, $q = 1$ and the flip-flop is set, and if $s = 0$ and $r = 0$, the output q stays in whatever state it was in. The output q reflects the last input to be high. If r was high last, $q = 0$. If s was high last, $q = 1$. We summarize this behavior in the state table of Table 14.1.

Because the function of sequential circuits depends on the evolution of signals over time, we often describe their behavior using a *timing diagram*. A timing diagram illustrating operation of the RS flip-flop is shown in Figure 14.3. The figure shows the waveforms, signal level as a function of time, for the signals r, s, and q. Time advances from left to right. Arrows from one signal to another show cause and effect.

Initially, q is in an unknown state, it could be either high or low, denoted by both high and low lines. At time t_1, r goes high causing q to fall - resetting the flip-flop. Signal s goes high at t_2 causing q to rise - setting the filp-flop The flip-flop is reset again at t_3. Signal s goes high at t_4, but this has no effect on the output, since r is also high. Signal s going high with r low at t_5 does set the flip-flop. It is reset again at t_6 when r goes high - even though s is still high. The flip-flop is set for a final time at t_7 when r goes low.

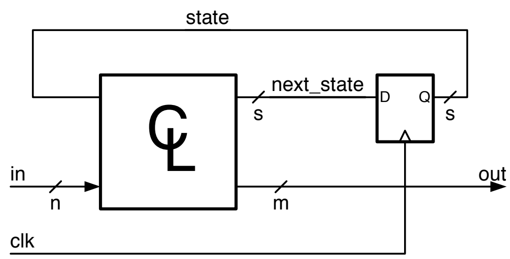


Figure 14.4: A *synchronous* sequential circuit breaks the state feedback loop with a clocked storage element (in this case a D-type flip-flop). The flip-flop ensures that all state variables change value at the same time - when the clock signal rises.

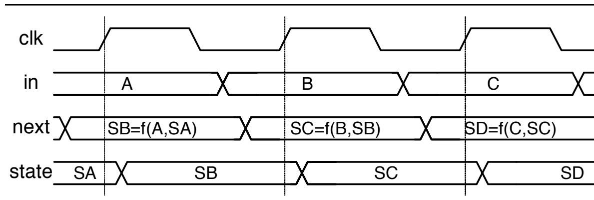


Figure 14.5: Timing diagram showing operation of a synchronous sequential circuit. The state advances on each rising edge of the clock signal, clk.

14.2 Synchronous Sequential Circuits

While the RS flip-flop of Figure 14.2 is simple enough to understand, arbitrary sequential circuits, with many bits of state feedback, can give complex behavior. Part of the complexity is due to the fact that the different bits of the next-state signal may change at different times. Such *races* can lead to a next-state output that depends on circuit delay. We defer discussion of general *asynchronous* sequential circuits until Chapter 22. Until then, we restrict our attention to *synchronous* sequential circuits in which clocked storage elements are used to ensure that all state variables change state at the same time - synchronized to a clock signal. Synchronous sequential circuits are sometimes called *finite-state machines* or FSMs.

A block diagram of a synchronous sequential logic circuit is shown in Figure 14.4. The circuit is synchronous because the state feedback loop is broken by an s-bit wide D flip-flop (where s is the number of state bits). This flip-flop circuit, described in the next section, updates its output with the value of its input on the rising edge of the clock signal. At all other times the output remains stable. Inserting the D flip-flop into the feedback loop constrains all of the state bits to change at the same time - eliminating the possiblity of races.

state	next state		out	
	in=0	in=1	in=0	in=1
00	00	01	0	0
01	00	11	0	0
11	01	10	0	0
10	11	00	0	1

Table 14.2: State table for example synchronous logic circuit.

cycle	state	in	out
0	00	0	0
1	00	1	0
2	01	1	0
3	11	0	0
4	01	1	0
5	11	1	0
6	10	0	0
7	11	1	0
8	10	1	1
9	00		0

Table 14.3: State sequence for the sequential logic circuit described by Table 14.2 on the input sequence 011011011.

Operation of a synchronous sequential circuit is illustrated in the timing diagram of Figure 14.5. During each clock cycle, the time from one rising edge of the clock to the next, a block of combinational logic computes the next state and output (not shown) as a combinational function of the input and current state. At each rising edge of the clock the current state (state) is updated with the next state computed during the previous clock cycle.

During the first clock cycle of Figure 14.5 for example, the current state is SB and the input changes to B before the end of the cycle. The combinational logic then computes the next state $SC = f(B, SB)$. At the end of the cycle the clock rises again updating the current state to be SC. The state will be SC until the clock rises again.

We can analyze synchronous sequential circuits on a clock-by-clock basis. The next state and output during the a given clock cycle depend only on the current state and input during that clock cycle. At each clock, the current state advances to the next state.

For example, suppose our next state and output logic are as given by Table 14.2. If our circuit starts in state 00 and has an input sequence of 011011011 for the first 9 cycles, what will its state and output be each cycle?

Operation of our example circuit is shown in Table 14.3. In cycle 0 we start

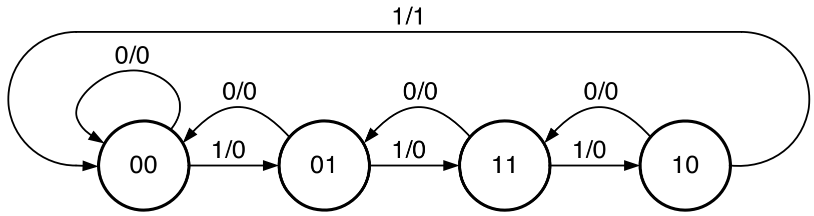


Figure 14.6: State diagram for finite-state machine of Table 14.2. The circles represent the four states. Each arrow represents a *state transition* from a current state to a next state and is labeled *input*/em output - the input that causes the transition, and the output in the current state for that input.

in state 00. The input and output are both 0 this cycle. With an input of 0 in state 00, the next state is also 00, so in cycle 1 we remain in state 00, but now with an input of 1. A state of 00 and an input of 1 gives us a next state of 01 for cycle 2. With the input high in state 01, we get a next state of 11 for cycle 3. The input goes low in state 11 in cycle 3, taking us back to state 01 for cycle 4. The input is high for the next two cycles taking us to 11 and 10 in cycles 5 and 6 respectively. A low input in cycle 6 takes us back to 11 for cycle 7. High inputs for cycles 7 and 8 takes us to states 10 and 00 for cycles 8 and 9. The output goes high in cycle 8 since the state is 10 and the input is 1.

One representation of a finite-state machine is a *state table* such as Table 14.2 that gives the next-state and output functions in tabular form. An equivalent graphical representation is a state diagram as shown in Figure 14.6.

Each circle in Figure 14.6 represents one state. Its labeled with the name of the state. Here we use the value of the state variables as the state name. Later we will introduce symbolic state names, independent of the state coding. The next state function is shown by the arrows. Each arrow represents a *state transition* and is labeled with the input and output values during that transition. For example, the arrow from state 00 to state 01, labeled 1/0, implies that in state 00 when the input is 0 the next state is 01 and the output is 0. Note that an arrow may go from a state to itself, as in the case of the input 0 transition from state 00 to state 00. Also, a transition may go quite a distance in the diagram, as with the input 1 transition from state 10 to state 00.

14.3 Traffic Light Controller

As a second example of a FSM, consider the problem of controlling the traffic lights at an intersection of a north-south road with an east-west road. There are six lights to control green, yellow, and red for the north-south road, and for the east-west road. Our FSM will take as input signal carew that indicates that a car is waiting on the east-west road. A second input, rst, resets the FSM to

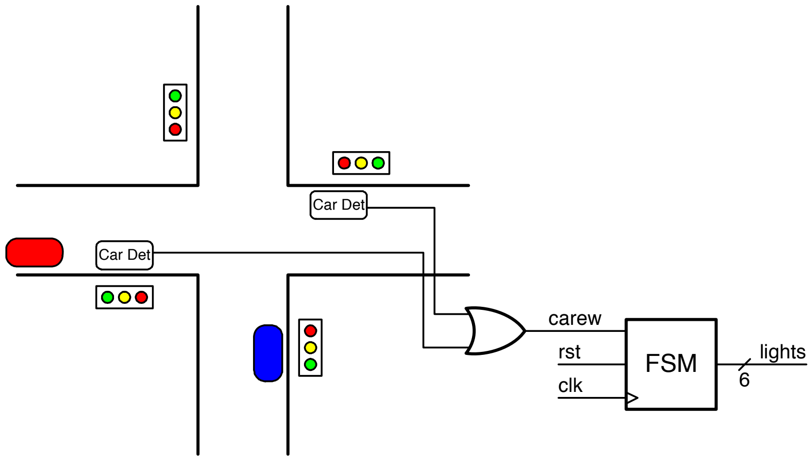


Figure 14.7: Controlling traffic lights at a road intersection with a FSM. Our FSM has two inputs, a reset (rst), and a signal that indicates that a car is waiting on the east-west road (carew). The FSM has six outputs that control the three north-south lights (green, yellow, red), and the three east-west lights.

a known state.

We start with an English-language description of our FSM:

1. Reset to a state where the light is green in the north-south direction and red in the east-west direction.

2. When a car is detected in the east-west direction (carew = 1) go through a sequence that makes the light green in the east-west direction and then return to green in the north-south direction.

3. A direction with a green light must first transition to a state where the light is yellow before going to a state where the light goes red.

4. A direction can have a green light only if the light in the other direction is red.

A state diagram for a FSM that meets our specification is shown in Figure 14.8. Compared to the state diagram of Figure 14.6 there are two major differences First, the states are labeled with symbolic names. the output values are placed under the states rather than on the transitions. This is because the output is a function only of the state, not of the input. Second, the output values are placed under the states rather than on the transitions. This is because the output is a function only of the state, not of the input.[3]

[3]A FSM where the output depends only on the current state and not the input is sometimes called a *Moore machine*. A FSM where the output depends on both the current state and the input is sometimes called a *Mealy machine*.

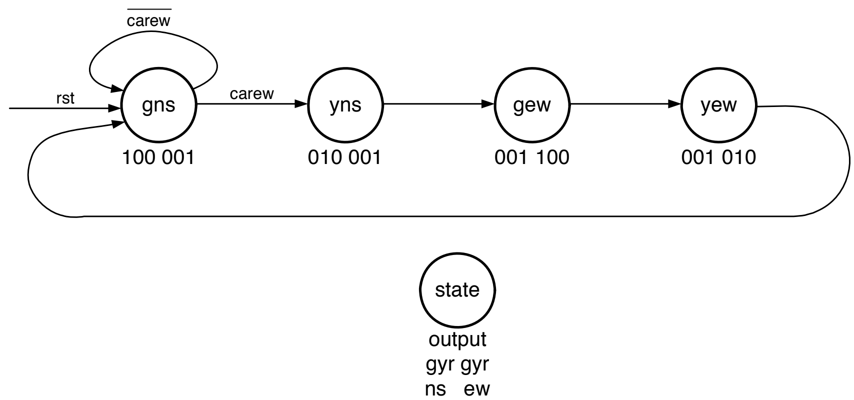


Figure 14.8: State diagram for a traffic-light controller FSM. The states are labeled with symbolic names. The outputs are given under each state (green-yellow-red (3-bits) for north-south followed by green-yellow-red for east-west). The reset arrows are omitted. The FSM resets to state gns.

The FSM resets to state gns (for green-north-south). In this state the output is 100 001. The first 100 represents the north-south lights (green-yellow-red). The 001 represents the east-west lights (also green-yellow-red). Hence the light is green in the north-south direction and red in the east-west direction. Resetting to this state satisfies specification number 1. The arrow labeled $\overline{\text{carew}}$ keeps us in state gns until a car is detected in the east-west direction.

When a car is detected in the east-west direction, signal carew becomes true, and the next rising edge of the clock causes the machine to enter state yns (yellow north-south). In this state the output is 010 001. The 010 implies yellow in the north-south direction, and 001 implies red in the east-west direction. Transitioning to this state before makign east-west green satisfies specification 3 on the transition from gns to gew. The arrow out of state yns has no label. This implies that this state transition always occurs (unless the FSM is reset).

State yns is always followed by state gew (green east-west). In this state the output is 001 100 - red in the north-south direction and green in the east-west direction. This state, and the sequence it is part of, satisfies specification 2. State gew is always followed by state yew. State yew (yellow east west) has output 001 010 - red in the north-south direction and yellow in the east-west direction. This state statisfies specification 3 on the transition between gew and gns.

A state table for the traffic-light controller is shown in Table 14.4. Reset is not shown.

state	next state		out
	carew=0	carew=1	
gns	gns	yns	100 001
yns	gew	gew	010 001
gew	yew	yew	001 100
yew	gns	gns	001 010

Table 14.4: State table for the traffic light controller FSM. The FSM resets to state gns.

state	encoding
gns	0001
yns	0010
gew	0100
yew	1000

Table 14.5: A one-hot state assignment for the traffic light controller FSM.

14.4 State Assignment

When a FSM is specified with symbolic state names as in Figure 14.8 or Table 14.4 we need to assign actual binary values to the states before we can implement the FSM. This process of assigning values to states is called *state assignment*.

With a synchronous machine, we can assign the states to any set of values, as long as the value for each state is unique. [4] It takes at least $s_{\min} = \log_2(N)$ bits to represent N states; however, the best state assignment is not always one with the fewest bits. We refer to each bit of a state vector as a *state variable*.

A *one-hot* state assignment uses N bits to represent N states. Each state gets its own bit. When the machine is in the i^{th} state, the corresponding bit b_i of the state variable is 1. In all other states, b_i is 0. A one-hot state assignment for the traffic-light controller FSM is shown in Table 14.5. It takes four bits to represent the four states. In any state, only one bit is set. A one-hot assignment makes the logic design of a finite state machine particularly simple as we shall see below.

A binary state assignment uses the minimum number of bits, $s_{\min} \log_2(N)$, to represent N states. Of the $N!$ possible binary state assignments (24 for 4 states), it doesn't really matter which one you choose. While lots of academic papers have been written about choosing state assignments to minimize implementation logic, in practice it doesn't really matter. Except in very rare cases the number of gates saved by optimizing the state assignment is unimportant. Don't waste

[4] This is not true of an asynchronous machine where careful state assignment is required to avoid races.

state	encoding
gns	00
yns	01
gew	11
yew	10

Table 14.6: A binary state assignment for the traffic light controller FSM.

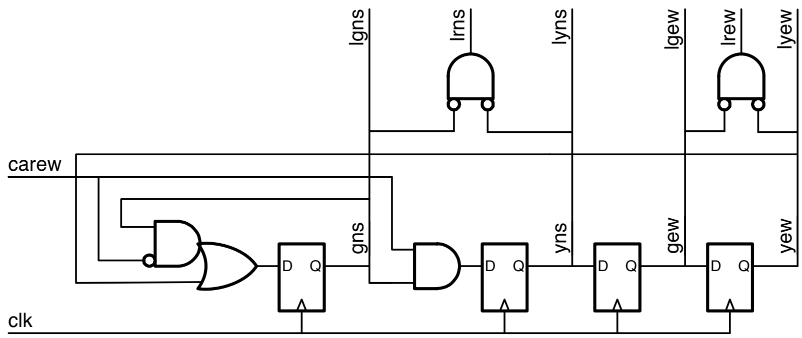


Figure 14.9: Implementation of the traffic light controller FSM using a one-hot state encoding. Four flip-flops are used, one corresponding to each state. The state transition arrows into a state directly translate into the logic preceding the corresponding flip-flop.

much time on state assignment. Design time is more important than a few gates.

One possible binary state assignment for the traffic light controller FSM is shown in Table 14.6. This particular state assignment uses a Gray code so that only one-bit of state changes on each state transition. This sometimes reduces power and minimizes logic. We could just as easily have chosen a straight binary count (gew = 10, yew = 11) and it wouldn't make much difference.

14.5 Implementation of Finite State Machines

Given a state table (or state diagram) and a state assignment the implementation of a FSM is reduced the problem of designing two combinational logic circuits, one for the next-state and one for the output. These combinational logic circuits are combined with a s-bit wide D-flip-flop to update the current state from the next state on each rising edge of the clock. A multi-bit D-flip-flop like this is often called a *register* and when it is used to hold the state of an FSM it is called the *state register*.

With a one-hot state assignment, the implementation of the next-state logic

state	carew	next state	comment
00	0	00	green north/south carew = 0
00	1	01	green north/south carew = 1
01	0	11	yellow north/south carew = 0
01	1	11	yellow north/south carew = 1
11	0	10	green east/west carew = 0
11	1	10	green east/west carew = 1
10	0	00	yellow east/west carew = 0
10	1	00	yellow east/west carew = 1

Table 14.7: Truth table for the next state function of the traffic light controller FSM with the state assignment of Table 14.6.

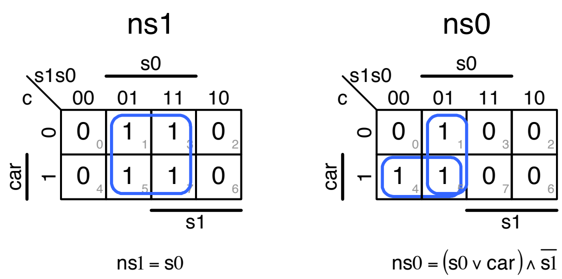
is a direct translation of the state diagram as shown in Figure 14.9 for our traffic light controller FSM. Four flip-flops correspond to the four states: gns, yns, gew, and yew. When the first flip flop is set, the FSM is in state gns. The logic feeding the D input of each flip-flop is an OR of the transition arrows feeding the corresponding state in the state diagram. For states gew and yew this is just a wire. These states always follow the preceding state. For state yns, the input logic is an AND gate that ANDs the previous state (gns) with the condition for a transition from gns to yns (carew). State gns is the target of two destination arrows and hence requires an OR gate to combine them. Its input logic is the OR of a wire from state yew, and an AND gate combining state gns and condition $\overline{\text{carew}}$ - this corresponds to the back edge from state gns to itself.

It is always possible to directly implement a one-hot FSM in this manner. This made design and maintenance of FSMs very easy in the era before logic synthesis. One would simply instantiate a flip-flop for each state, and approprate input gates for each transition arrow. The function is immediately apparent from the logic. Adding, deleting, or changing the condition on a transition arrow was straightforward and affected only only the part of the logic associated with the transition arrow. With modern logic synthesis, the advantage of this approach is greatly diminished.

The output logic for the circuit of Figure 14.9 consists of two NOR gates. States gns, yns, gew, and yew drive the green and yellow light outputs directly. The red light outputs are generated by observing that for each direction, if the yellow and green lights are off, the red light should be on: $r = \overline{y} \vee \overline{g}$.

To implement our FSM with a binary state encoding, we proceed with logic synthesis of each of the state variables. First, we convert our state table into a truth table - showing each next state variables as a function of the current state variables and all inputs. For example, a truth table for the traffic light controller FSM with the state encoding of Table 14.6 is shown in Table 14.7.

From this state table we draw the two Kargaugh maps shown in Figure 14.5. The left K-map shows the truth-table for the MSB of the next-state (ns1) and



caption

state	output
00	100 001
01	010 001
11	001 100
10	001 010

Table 14.8: Truth table for the output function of the traffic light controller FSM with the state assignment of Table 14.6.

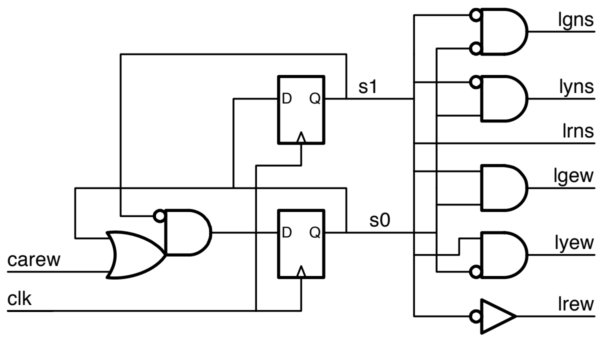
the right K-map shows the truth table for the LSB of the next-stage (ns0). The next state logic here is very simple. The ns1 function has a single prime implicant, and the ns0 function has 2. All three are essential. The logic here is very simple:

$$n_1 = s_0 \tag{14.1}$$
$$n_0 = (s_0 \vee \text{carew}) \wedge \overline{s_1} \tag{14.2}$$

Where n_1, n_0 are the next state variables and s_1, s_0 are the current state variables.

Now that we have the next-state function, what remains is to derive the output function. To do this we write down the truth table for the outputs - a function of only the current state. This is shown in Table 14.8. The logic functions for the output variables can be derived directly from this table - a K-map is not necessary, we get:

$$g_{\text{ns}} = \overline{s_1} \wedge \overline{s_0} \tag{14.3}$$
$$y_{\text{ns}} = \overline{s_1} \wedge s_0 \tag{14.4}$$
$$r_{\text{ns}} = s_1 \tag{14.5}$$
$$g_{\text{ew}} = s_1 \wedge s_0 \tag{14.6}$$
$$y_{\text{ew}} = s_1 \wedge \overline{s_0} \tag{14.7}$$
$$r_{\text{ew}} = \overline{s_1} \tag{14.8}$$



caption

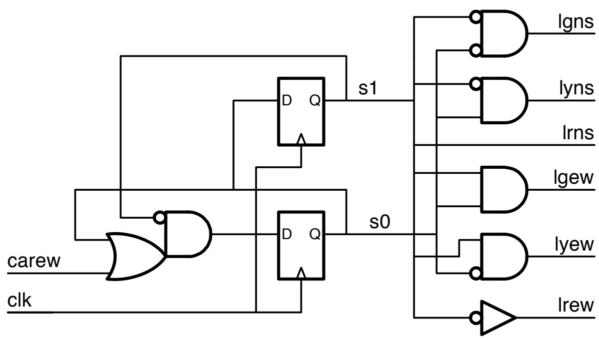


Figure 14.10: Logic diagram for traffic light controller FSM implemented with state assignmet of Table 14.6.

Combining the next-state and logic equations we get the logic diagram of Figure 14.10.

14.6 Verilog Implementation of Finite State Machines

With Verilog, designing a FSM is a simple matter of specifying the next-state and output functions and selecting a state assignment. Logic synthesis does all the work of generating the next-state and output logic. A verilog description of the traffic light controller FSM is shown in Figure 14.11. The bulk of the logic is in a single `case` statement that defines both the next-state and output functions.

There are three key points to be made about this code.

1. In implementing all sequential logic, all state variables should be explicitly

```
//----------------------------------------------------
// Traffic_Light
// Inputs:
//   clk - system clock
//   rst - reset - high true
//   carew - car east/west - true when car is waiting in east-west direction
// Outputs:
//   lights - (6 bits) {gns, yns, rns, gew, yew, rew}
// Waits in state GNS until carew is true, then sequences YNS, GEW, YEW
// and back to GNS.
//----------------------------------------------------
module Traffic_Light(clk, rst, carew, lights) ;
  input clk ;
  input rst ;            // reset
  input carew ;          // car present on east-west road
  output [5:0] lights ;  // {gns, yns, rns, gew, yew, rew}
  wire [`SWIDTH-1:0] state, next ; // current and next state
  reg [`SWIDTH-1:0] next1 ;        // next state w/o reset
  reg [5:0] lights ;               // output - six lights 1=on

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

  // next state and output equations - this is combinational logic
  always @(state or carew) begin
    case(state)
      `GNS: {next1, lights} = {(carew ? `YNS : `GNS), `GNSL} ;
      `YNS: {next1, lights} = {`GEW, `YNSL} ;
      `GEW: {next1, lights} = {`YEW, `GEWL} ;
      `YEW: {next1, lights} = {`GNS, `YEWL} ;
    endcase
  end
  // add reset
  assign next = rst ? `GNS : next1 ;
endmodule
```

Figure 14.11: Verilog description of traffic light controller FSM.

declared as D-flip-flops. **DO NOT** let the Verilog compiler infer flip-flops for you. In this code, the state flip-flops are explicitly instantiated in the code:

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

This code instantiates an `` `SWIDTH ``-wide D-flip-flop that is clocked by `clk`, has input `next` and output `state`.

2. In designing finite state machines, use Verilog `` `define `` statements to define all constants. Do not hard-code any constants in the code. Constants that should be declared this way include the width of the state vector `` `SWIDTH ``, the state encodings (e.g., `` `GNS ``), and input and output encodings (e.g., `` `GNSL ``). In particular defining symbolic names for the state encodings enables you to change a state assignment by just changing the definitions. We'll see an example of this below.

3. Make sure you reset your FSM. Here we declare two next-state vectors, `next1`, and `next`. The `case` statement computes `next1` as the next state ignoring reset `rst`. A final assign statement overrides this next state with the reset state `` `GNS `` if `rst` is asserted:

```
// add reset
assign next = rst ? `GNS : next1 ;
```

Factoring the reset out of the next state function in this manner greatly improves readability of the code. If we didn't do this, we'd have to repeat the reset logic in every state - rather than doing it just once.

The Verilog definitions for the traffic light controller FSM are shown in Figure 14.12. Using `` `define `` in this manner enables us to use symbolic names in our code, improving readability, and also makes it easy to change encodings. For example, substituting the one-hot state encodings of Figure 14.13 changes our FSM from a binary to a one-hot state assignment without changing any other lines of code.

The Verilog code for the `DFF` module is shown in Figure 14.14. The behavior is described by the `always` block:

```
always @(posedge clk)
  out = in ;
```

This block performs the update of the output `out = in` on every rising edge (`posedge`) of `clk`.

Our test bench for the traffic-light controller is shown in Figure 14.15. To thoroughly test an FSM, we would like to both visit every state, and traverse every edge of the state diagram. Achieving this coverage is not particularly difficult for our traffic light controller FSM.

```
//----------------------------------------------
// define state assignment - binary
//----------------------------------------------
`define SWIDTH 2
`define GNS 2'b00
`define YNS 2'b01
`define GEW 2'b11
`define YEW 2'b10
//----------------------------------------------
// define output codes
//----------------------------------------------
`define GNSL 6'b100001
`define YNSL 6'b010001
`define GEWL 6'b001100
`define YEWL 6'b001010
```

Figure 14.12: Verilog definitions for traffic-light controller state variables and output encodings.

```
//---------------------------------------------------------------------
// define state assignment - one hot
//---------------------------------------------------------------------
`define SWIDTH 4
`define GNS 4'b1000
`define YNS 4'b0100
`define GEW 4'b0010
`define YEW 4'b0001
```

Figure 14.13: Verilog definitions for a one-hot state assignment for the traffic light controller FSM.

```
module DFF(clk, in, out) ;
  parameter n = 1;  // width
  input clk ;
  input  [n-1:0] in ;
  output [n-1:0] out ;
  reg    [n-1:0] out ;

  always @(posedge clk)
    out = in ;
endmodule
```

Figure 14.14: Verilog description of a D-flip-flop.

```
module Test_Fsm1 ;
  reg clk, rst, carew ;
  wire [5:0] lights ;

  Traffic_Light tl(clk, rst, carew, lights) ;

  // clock with period of 10 units
  initial begin
    clk = 1 ; #5 clk = 0 ;
    forever
      begin
        $display("%b %b %b %b", rst, carew, tl.state, lights ) ;
        #5 clk = 1 ; #5 clk = 0 ;
      end
    end

  // input stimuli
  initial begin
    rst = 0 ; carew=0 ;        // start w/o reset to show x state
    #15 rst = 1 ; carew = 0 ;  // reset
    #10 rst = 0 ;              // remove reset
    #20 carew = 1 ;            // wait 2 cycles, then car arrives
    #30 carew = 0 ;            // car leaves after 3 cycles (green)
    #20 carew = 1 ;            // wait 2 cycles then car comes and stays
    #60                        // 6 more cycles
    $stop ;
  end
endmodule
```

Figure 14.15: Verilog test bench for the traffic light controller FSM.

```
0 0 xx xxxxxx
1 0 xx xxxxxx
0 0 00 100001
0 0 00 100001
0 1 00 100001
0 1 01 010001
0 1 11 001100
0 0 10 001010
0 0 00 100001
0 1 00 100001
0 1 01 010001
0 1 11 001100
0 1 10 001010
0 1 00 100001
0 1 01 010001
```

Figure 14.16: Results of simulating the traffic light controller FSM of Figure 14.11 using the test bench of Figure 14.15. Each line shows the values of `rst`, `carew`, `state`, and `lights` on a falling edge of the clock.

The test bench has three components. First, it instantiates a `Traffic_Light` module - the unit being tested. The second component is an `initial` block that performs clock generation and output. The `forever` block repeats its body until the simulation terminates. In this case the repeated code displays some variables and generates a clock with a 10 delay-unit period. The display is done in the middle of the clock cycle - just after `clk` goes low. The final component generates the inputs for the module being tested.

The inputs, and the response of the module are shown in textual form in Figure 14.16, which shows the signals on each falling edge of the clock, and as waveforms in Figure **??**. Initially, `state` and `next` are both in an unknown state (`x` in the text output, and a line midway between a 1 and 0 in the waveform). Signal `rst` is asserted in the second clock cycle to reset to a known state. Next-state signal `next` responds immediately, and `state` follows on the rising edge of the clock. The FSM stays in state 00 until `carew` rises on clock 5, causing the FSM to go to state 01 on clock 6. This starts a sequence through states 01, 11, 10, and back to 00 on clock 9. It stays in state 00 for two cycles this time, until `carew` going high in clock 10 causes the FSM to start the sequence again in clock 11. This time `carew` stays high and the sequence repeats until the simulation ends.

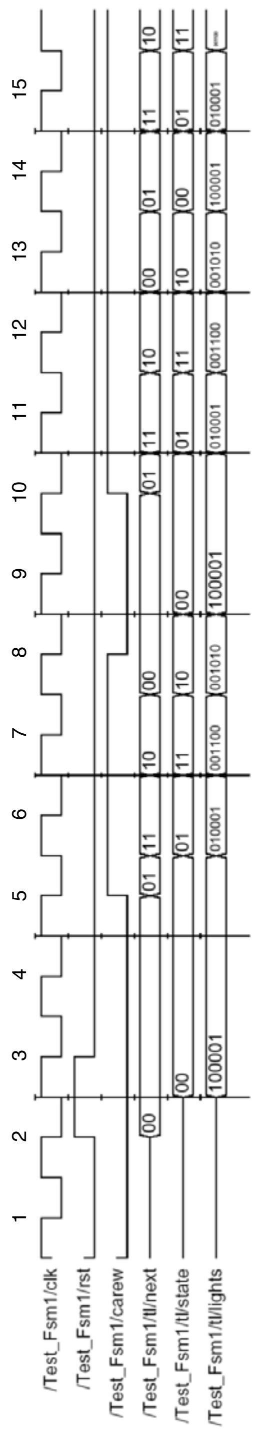


Figure 14.17: Waveforms from simulation of the traffic light controller of Figure 14.11 using test bench of Figure 14.15.

14.7 Bibliographic Notes

14.8 Exercises

14–1 *Homing sequences.* The finite-state machine described by Table 14.2 does not have a reset input. Explain how you can get the machine in a known state regardless of its initial starting state by providing a fixed input sequence. An input sequence that always takes an FSM to the same state is called a *homing sequence.*

14–2 *Homing sequences.* Suppose the traffic light controller FSM of Table 14.4 did not reset to state gns. Find a homing sequence for the machine that will get it to state gns.

14–3 *Finite State Machines.* Modify the traffic light controller FSM of Table 14.4 so that it makes lights go red in both directions for one cycle before turning a light green. Show a state table and state diagram for your new FSM.

14–4 *Finite State Machines.* Modify the traffic light controller FSM of Table 14.4 so that it takes an additional input, carns, that indicates when there is a car waiting in the north-south direction. Change the logic so that once the light changes to east-west, it stays with east-west green until a car is detected waiting in the north-south direction. Show a state table and state diagram for your new FSM.

14–5 *FSM Implementation.* Implement the traffic light controller FSM with a state encoding where gns = 00, yns = 01, gew = 10, and yew = 11. Show Karnaugh maps for the next state variables and output variables and a gate-level schematic for the FSM.

14–6 *A Digital Lock* Draw a state diagram and a state table for a digital lock. The lock has two inputs a and b and one output, *unlock*. The output is assered only if the sequence a, b, a, a is observed. Each element of the sequence must last for one or more cycles, and there may be zero or more cycles of both inputs low between the sequence elements. After unlocking, either input going high causes unlock to go low.

14–7 *Anti-Lock Brakes*A FSM for an anti-lock brake system accepts two inputs - *wheel* and *time*, and generates a single output - *unlock*. The *wheel* input pulses high for one clock cycle each time the wheel rotates a small amount. The *time* input pulses high for one clock cycle every 10ms. If the machine detects two *time* pulses since the last *wheel* pulse, it concludes that the wheel is locked and *unlock* is asserted for one clock cycle to “pump” the brakes. After *unlock* goes high, the machine waits for two *time* pulses before resuming normal operation. Thus, there are a minimum of four *time* pulses between *unlock* pulses. Draw a state diagram (bubble diagram) for this state machine.

14–8 *Direction Sensor.* Draw a state diagram and state table for a direction sensor machine. This machine has two inputs *il* and *ir* and two outputs *ol* and *or*. The machine outputs a one cycle pulse on *ol* any time a high level on *il* for one or more cycles is followed, after zero or more cycles, by a high level on *ir* for zero or more cycles. Similarly a one cycle pulse on *or* is output if a high level on *ir* is followed by a high level on *il*.

Chapter 15

Timing Constraints

How fast will a FSM run? Could making our logic too fast cause our FSM to fail? In this chapter, we will see how to answer these questions by analyzing the timing of our finite state machines and the flip-flops used to build them.

Finite state machines are governed by two timing constraints - a maximum delay constraint and a minimum delay constraint. The maximum speed at which we can operate a FSM depends on two flip-flop parameters (the setup time and propagation delay) along with the maximum propagation delay of the next-state logic. On the other hand, the minimum delay constraint depends on the other two flip-flop parameters (hold time and contamination delay) and the minimum contamination delay of the next-state logic. We will see that if the minimum delay constraint is not met, our FSM may indeed fail to operate at any clock speed due to hold-time violations. Clock skew, the delay between the clocks arriving at different flip-flops, affects both maximum and minimum delay constraints.

15.1 Propagation and Contamination Delay

In a synchronous system, logic signals advance from the stable state at the end of one clock cycle to a new stable state at the end of the next clock cycle. Between these two stable states, they may go through an arbitrary number of transitions.

In analyzing timing of a logic block we are concerned with two times. First, we would like to know how long the output retains its initial stable value (from the last clock cycle) after an input first changes (in the new clock cycle). We refer to this time as the *contamination delay* of the block- the time it takes for the old stable value to become contaminated by an input transition. Note that this first change in the output value does not in general leave the output in its new stable state.

The second time we would like to know is how long it takes the output to reach its new stable state after the input stops changing. We refer to this time

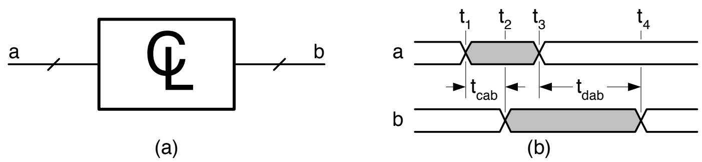


Figure 15.1: Propagation delay t_{dab} and contamination delay t_{cab}. The *contamination delay* of a logic block is the time from when the **first** input signal **first** changes to when the **first** output signal **first** changes. The *propagation delay* of a logic block is the time from when the **last** input signal **last** changes to when the **last** output signal **last** changes.

as the *propagation delay* of the block - the time it takes for the stable value of the input to propagate to a stable value at the output.

Propagation delay and contamination delay are illustrated in Figure 15.1. Figure 15.1(a) shows a combinational logic block with input a and output b. Figure 15.1(b) shows how the output b responds when input a changes state. Up to time t_1, both input a and output b are in their stable state from the last clock cycle. At time t_1 input a first changes. If a is a multi-bit signal, this is the time when the first bit of a to change state toggles - other bits may change at later times. Whether single-bit or multi-bit, t_1 is the time of the first transition on a. A given bit of a may toggle more than once before reaching its new stable state. At time t_2, a contamination delay t_{dab} after t_1 this first change on a may affect output b, and b may change state. Up until t_2, output b was guaranteed to have the steady-state value from the previous clock cycle. The first bit of b to change toggles for the first time at time t_2 as with a at t_1, this bit of b may toggle again before reaching a steady state, and other bits of b may change state later.

At time t_3 input a stops changing state. From t_3 until at least the end of the current clock cycle, signal a is guaranteed to be in its stable state for this clock cycle. Time t_3 represents the time at which the last bit of a to toggle toggles for the last time. At time t_4, a propagation delay t_{dab} after t_3, the last change of input a has its final affect on output b. From this point to at least the end of the clock cycle, output b is guaranteed to be in its stable state for this clock cycle.

We denote a propagation (contamination) delay from a signal a to a signal b as t_{dab} (t_{cab}). The d or c in the subscript denotes propagation or contamination. The rest of the subscript gives the source signal and destination signal of the delay. That is t_{dxy} is the delay starting with a transition on signal x to a transition on signal y.

Propagation and contamination delays sum up over linear paths as shown in Figure 15.2. The timing diagram in the figure shows that when two modules are composed in series, their delays sum:

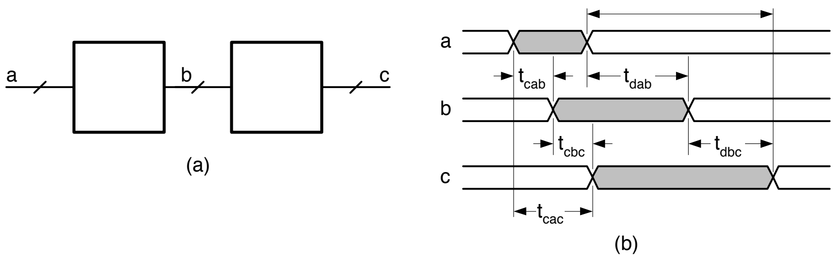


Figure 15.2: Propagation and contamination delay sum over a linear path. (a) Two modules in series with input a, intermediate signal b, and output c. (b) Timing diagram showing that $t_{cac} = t_{cab} + t_{cbc}$ and similarly for propagation delay.

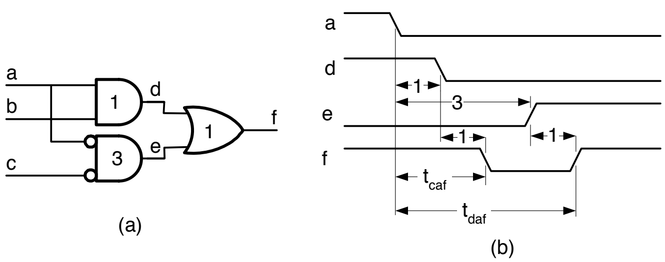


Figure 15.3: A circuit with a hazard illustrating propagation and contamination delay.

$$t_{cac} = t_{cab} + t_{cac}, \quad (15.1)$$
$$t_{dac} = t_{dab} + t_{dac}. \quad (15.2)$$

To handle circuits with parallel paths, we simply enumerate all possible single-bit paths. The overall contamination delay is the *minimum* over all paths, and the overall propagation delay is the *maximum* over all paths.

Figure 15.3(a) shows a circuit with a static-1 hazard (recall Section 6.10). The value in each gate symbol is the delay of the gate in arbitrary time units. (Here we assume the contamination and propagation delay of the basic gates are the same). The timing diagram in Figure 15.3(b) illustrates the timing when signal a falls while $b = 1$ and $c = 0$. The output changes for the first time after two time units and for the last time after four time units. Hence, $t_{caf} = 2$ and $t_{daf} = 4$.

We can get the same result by enumerating paths. The minimum delay path is a-d-f with a contamination delay of 2 while the maximum path is a-e-f with

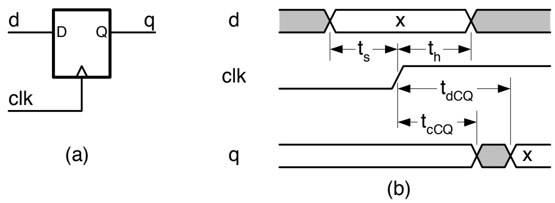


Figure 15.4: A D-flip-flop. (a) Schematic symbol, (b) Timing diagram. The D-flip-flop samples its input on the rising edge of the clock and updates the output with the value sampled. For correct sampling the input must be *stable* from t_s before the clock rises to t_h after the clock rises. The output may change as soon as t_{cCQ} after the clock. The output takes on the correct value no later than t_{pCQ} after the clock.

a propagation delay of 4.

Contamination and propagation delay are independent of input state. The contamination delay of the circuit in Figure 15.3(a) from a to f is 2 regardless of the state of signals b and c. This delay represents the possibility that the output *may* change 2 time units after a transition on a, not a guarantee that it will change.

May people confuse contamination delay with minimum propagation delay. They are not the same thing. The minimum propagation delay is the minimum value (over some range of parameters: voltage, temperature, process variation, input combinations) of the time for the correct steady-state value to appear on the output of a circuit after a transition on the input. In contrast, the contamination delay is the time until the output first changes from its old steady state value after a transition on its input. These are not the same thing. The transition that sets the contamination delay is not in general a change of the output to its steady state value, but rather a change to some intermediate value - for example the transition of a to 0 in the hazard of Figure 15.3.

15.2 The D Flip-Flop

The timing constraints that determine whether an FSM will operate or not, and at what speed it will operate, are governed by the clocked storage elements used to construct the FSM - in our case, the D flip-flop. A schematic symbol for a D-flip-flop is shown in Figure 15.4. A multi-bit D-flip-flop is sometimes called a *register*.

A D flip-flop samples its input on the rising edge of the clock signal and updates its output with the value sampled. This sampling and update is illustrated in the timing diagram of Figure 15.4(b). For the sampling to take place correctly, the input data (shown in the top waveform of the timing diagram) must be stable for a period before and after the rising edge of the clock. Specifically, the data must have reached its correct value (labeled x in the figure) at least a *setup time* t_s before the clock reaches its 50% point, and the data must be held stable at this value until a *hold time* t_h after the clock reaches its 50% point.[1] During the gray areas in the data waveform, D can take on any value. However it must remain stable with a value of x during the setup and hold intervals for the flip-flop to correctly sample the value x.

If the input meets its setup and hold time constraints, the flip-flop will update the output with the sampled value x as shown on the bottom waveform of Figure 15.4(b). The old value (which was sampled on the previous rising edge of the clock) will remain stable on the output until a *contamination delay* t_{cCQ} after the rising edge of the clock. The circuit may continue to rely on the old value being stable up until this point in time. After the contamination delay the output of the flip-flop may change, but not necessarily to the correct value. This period where the output value is not guaranteed is labeled gray in the figure. A *propagation delay* t_{dCQ} after the rising edge of the clock, the output is guaranteed to have the value x sampled from the input. It will then hold this value stable until t_{cCQ} after the next rising edge of the clock.

15.3 Setup and Hold Time Constraint

Now that we have introduced the nomenclature, the timing constraints on a finite-state machine are quite simple. To ensure that the clock cycle t_{cy} is long enough for the longest path to satisfy the setup time of the D-flip-flop we must satisfy:

$$t_{\text{cy}} \geq t_{dCQ} + t_{d\text{Max}} + t_s. \tag{15.3}$$

Where $t_{d\text{Max}}$ is the maximum propagation delay from the output of a D-flip-flop to the input of a D-flip-flop.

We also must ensure that no signal is contaminated so quickly as to violate the hold-time constraint on the input of a D-flip-flop by satisfying:

$$t_h \leq t_{cCQ} + t_{c\text{Min}}. \tag{15.4}$$

Where $t_{c\text{Min}}$ is the minimum contamination delay from the output of a D-flip-flop to the input of a D-flip-flop.

The two constraints (15.3) and (15.4) govern system timing. Setup-time constraint (15.3) determines performance by giving the minimum cycle time t_{cy} at which the circuit will operate. The hold-time constraint, on the other hand, is a correctness constraint. If (15.4) is violated, the circuit may not meet its hold time constraint regardless of the cycle time.

Figure 15.5 shows a simple finite-state machine that we shall use to illustrate setup and hold constraints below. The FSM consits of two flip-flops. The

[1] Note that t_s or t_h may be negative, but $t_s + t_h$ will always be positive.

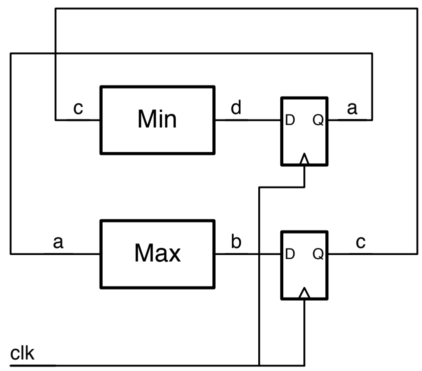


Figure 15.5: A simple FSM to illustrate setup and hold constraints

upper flip-flop generates state-bit a that propagates through a maximum-length (maximum propagation delay) logic path (Max) to generate signal b. Signal b is in turn sampled by the lower flip-flop. The lower flip-flop in turn generates signal c which propagates through a minimum-length (minimum contamination delay) block to generate signal d which is sampled by the upper flip-flop. Note that the destination flip-flop of a minimum-delay path is not necessarily the source flip-flop of a maximum-delay path (and vice versa). In general we need to test all possible paths from all flip-flops to all flip-flops to find the minimum and maximum paths.

The maximum-delay path from the upper flip-flop to the lower flip-flop of Figure 15.5 stresses the setup-time of the lower flip-flop. If this path is too slow, the next clock edge may arrive at the lower flip-flop before its input signal b has settled at its final value for the cycle. Figure 15.6 repeats Figure 15.5 with this path highlighed. A timing diagram corresponding to this path is shown in Figure 15.7. Suppose the rising edge of the clock samples value x on signal d then after a flip-flop propagation delay t_{dCQ} flip-flop output a will take on value x and hold this value through the remainder of the clock cycle. Signal a is input to combinational block Max which generates signal b. After an additional propagation delay from a to b t_{dab} (which corresponds to $t_{d\mathrm{Max}}$ in Inequality (15.3)) signal b takes on its final value for the clock cycle $f(x)$. Signal b must settle at this final value at least t_s before the rising edge of the next clock for Constraint (15.3) to be satisfied. The sum of the propagation delays along the maximum path and the setup time must be less than the cycle time. In the timing diagram, signal b settles slightly early leaving a timing margin or *slack* time of t_{slack}. The clock cycle t_{cy} could be reduced by t_{slack} and the setup constraint would still be met.

The minimum-delay path from the lower flip-flop to the upper flip-flop of Figure 15.5 stresses the hold time of the upper flip-flop. If this path is too fast, signal d might change before a hold time after the rising edge of the clock. Fig-

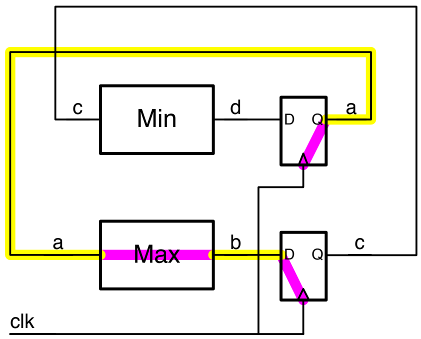


Figure 15.6: Setup-time constraint. The maximum path from the clock on a source flip-flop to the clock on a destination flip-flop is shaded. From the rising edge of the clock, the signal must propagate to the Q output of the flip-flop (t_{dCQ}) and propagate through the maximum-delay logic path (t_{dab}) at least a setup time (t_s) before the next clock edge. In this case the signal arrives slightly early (t_{slack}).

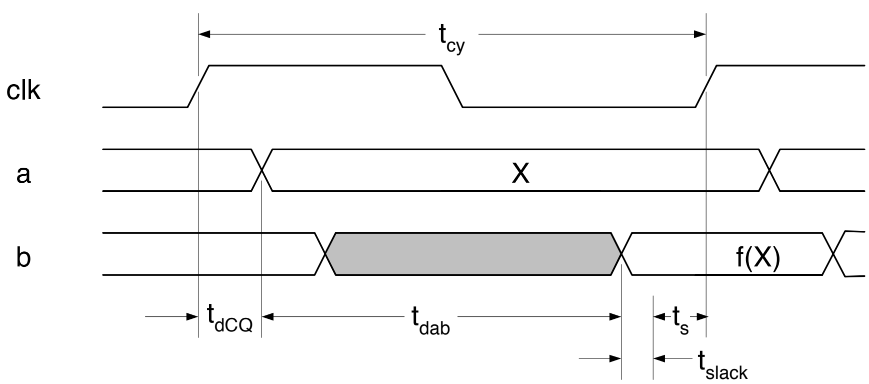


Figure 15.7: Timing diagram illustrating the setup-time constraint.

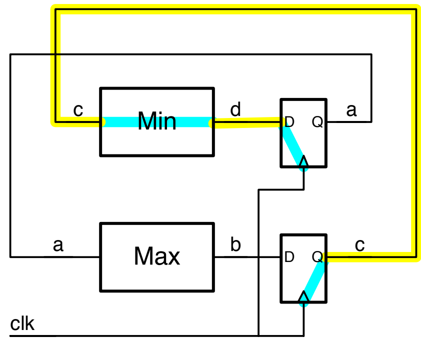


Figure 15.8:

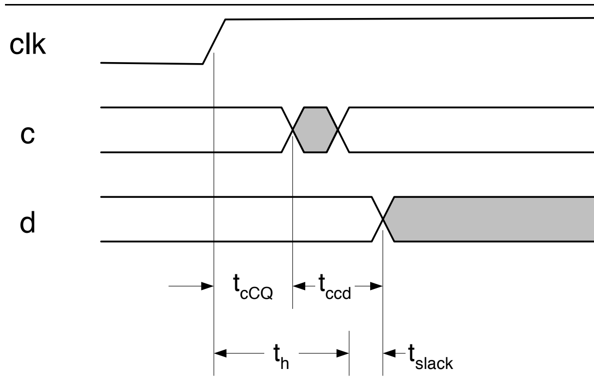


Figure 15.9:

ure 15.8 shows this timing path highlighted. A timing diagram illustrating the signals along this path is shown in Figure 15.9. A flip-flop contamination delay t_{cCQ} after the rising edge of the clock, signal c may first change. A contamination delay of the logic block t_{ccd} (which corresponds to $t_{c\mathrm{Min}}$ in Constraint (15.4)) signal d may change. For the hold time constraint to be satisfied, this first change on signal d is not allowed to occur until t_h after the rising edge of the clock. The sum of the contamination delays along the minimum path must be larger than the hold time. In the figure, the contamination delays exceed the hold time by a considerable timing margin or slack time t_{slack}.

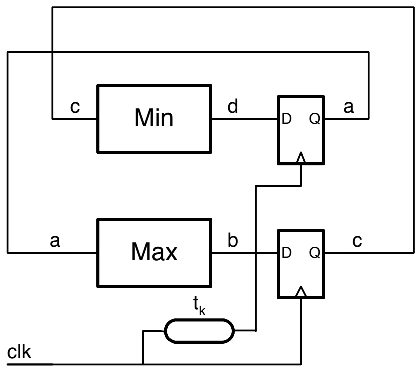


Figure 15.10: FSM of Figure 15.5 with clock skew.

15.4 The Effect of Clock Skew

On an ideal chip, the clock signal would change at the input of all flip-flops at the same time. In practice, device variations and wire delays in the clock distribution network cause the timing of the clock signal to vary slightly from flip-flop to flip-flop. We refer to this spatial variation in clock timing as *clock skew*. Clock skew adversely affects both the setup and hold timing constraints. With a skew of t_k, these two constraints become:

$$t_{\text{cy}} \geq t_{dCQ} + t_{d\text{Max}} + t_s + t_k. \tag{15.5}$$

$$t_h \leq t_{cCQ} + t_{c\text{Min}} - t_k. \tag{15.6}$$

Figure 15.10 shows the FSM of Figure 15.5 with clock skew added. A delay line (the oval shaped block) with a delay of t_k (the magnitude of the skew) is connected between the clock input and the clock to the upper flip-flop. Hence each edge of the clock arrives at the upper flip-flop t_k later than it arrives at the lower flip-flop. Delaying the clock to the source of the maximum-length path causes this path to effectively get longer. In a similar manner, delaying the clock to the destination of the minimum-length path effectively makes this path shorter.

The effect of skew on the minimum-length path, and hence on the hold-time constraint, is shown in Figure 15.11. The contamination delays from the clock to c to d add as above. However, now signal d must stay stable until t_h after the delayed clock *clkd* or $t_h + t_k$ after the original clock *clk*. The effect is the same as increasing the hold time by t_k.

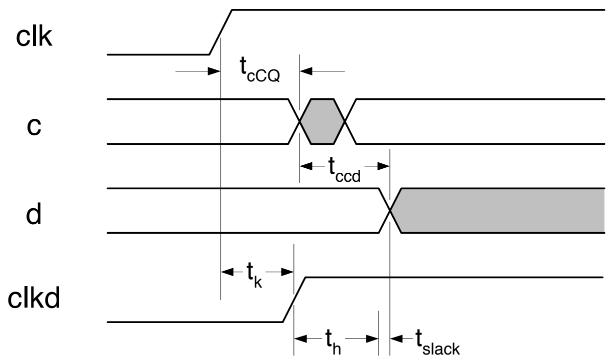


Figure 15.11: Timing diagram showing the effect of clock skew on the hold-time constraint.

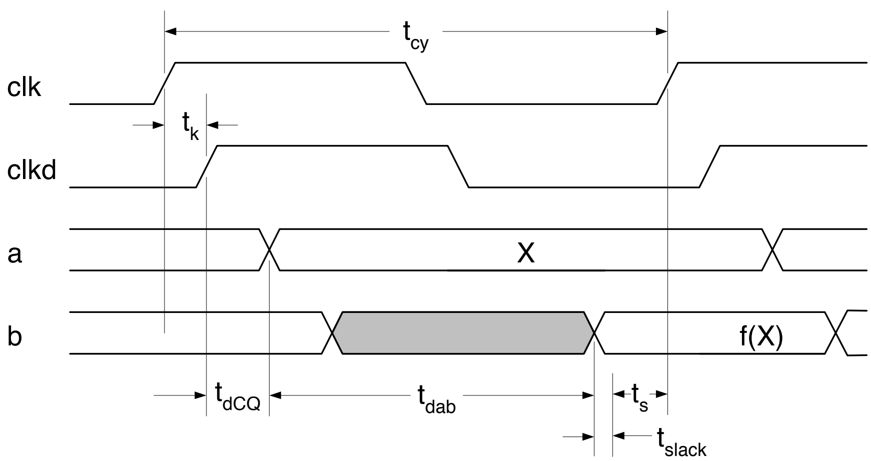


Figure 15.12: Timing diagram showing the effect of clock skew on the setup time constraint.

The timing diagram of Figure 15.12 illustrates the effect of clock skew on the setup time constraint. Delaying the clock to the upper flip-flop delays the transition of a by t_k, effectively adding t_k to the maximum path.

15.5 Timing Examples

example setup and hold calculations with and without skew
find max frequency
check hold constraint
fix hold constraint

15.6 Timing and Logic Synthesis

15.7 Bibliographic Notes

15.8 Exercises

15–1 Setup time.

15–2 Hold time.

15–3 Clock skew.

Chapter 16

Data Path Sequential Logic

In the last chapter we saw how a finite state machine can be synthesized from a state diagram by writing down a table for the next state function and synthesizing the logic that realizes this table. For many sequential functions, however, the next state function can be more simply described by an expression rather than by a table. Such functions are more efficiently described and realized as *datapaths* where the next state is computed as a logical function, often involving arithmetic circuits, multiplexers, and other building block circuits.

16.1 Counters

16.1.1 A Simpler Counter

Suppose you want to build a finite-state machine with the state diagram shown in Figure 16.1. This circuit is forced to state 0 whenever input `r` is true. Whenever input `r` is false, the machine counts through the states from 0 to 31 and then cycles back to 0. Because of this counting behavior, we refer to this finite-state machine as a *counter*.

We could design the counter employing the methodology developed in Chapter 14. A Verilog description taking this approach for a three-bit counter (8 states) is shown in Figure 16.2.[1] A 3-bit wide bank of flip-flops holds the current state `count` and updates it from the next state `next` on each rising edge

[1] A 5-bit counter (32 states) using this approach would require 32 lines in the state table.

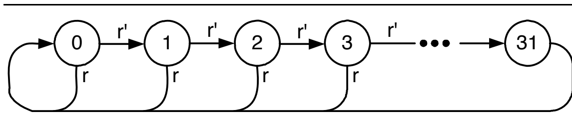


Figure 16.1: State diagram for a 5-bit counter.

```
module Counter1(rst,clk,count) ;
  input rst, clk ; // reset and clock
  output [2:0] count ;
  reg    [2:0] next ;

  DFF #(3) count(clk, next, count) ;

  always@(rst, count) begin
    casex({rst,count})
      4'b1xxxxx: next = 0 ;
      4'd0: next = 1 ;
      4'd1: next = 2 ;
      4'd2: next = 3 ;
      4'd3: next = 4 ;
      4'd4: next = 5 ;
      4'd5: next = 6 ;
      4'd6: next = 7 ;
      4'd7: next = 0 ;
      default: next = 0 ;
    endcase
  end
endmodule
```

Figure 16.2: A three-bit counter FSM specified as a state table.

```
module Counter(clk, rst, count) ;
  parameter n=5 ;
  input rst, clk ; // reset and clock
  output [n-1:0] count ;

  wire   [n-1:0] next = rst? 0 : count+1 ;

  DFF #(n) count(clk, next, count) ;
endmodule
```

Figure 16.3: An n-bit counter FSM specified with a single assign statement.

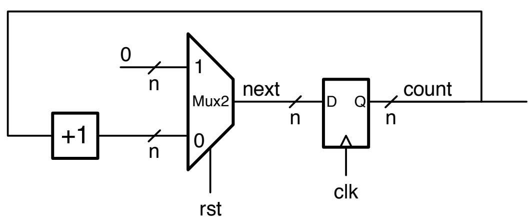


Figure 16.4: Block diagram of a simple counter. The counter state is held in a state register (flip-flops). The next state is selected by a multiplexer to be zero if `rst` is asserted, or the output of an incrementer (`count+1`) otherwise.

of the clock. The `casex` statement directly captures the state table, specifying the next state for each input and current state combination.

While this method for generating counters works, it is verbose and inefficient. The lines of the state table are repetitive. The behavior of the machine can be captured entirely by the single line:

```
assign next = rst ? 0 : count + 1 ;
```

This is a *datapath* description of the finite state machine in which we specify the next state as a function of the current state and inputs.

A verilog description of an n-bit counter module using such a datapath description is shown in Figure 16.3. An n-bit wide bank of flip-flops holds the current state, `count`, and updates it from the next state `next` on each rising edge of the clock. A single assign statement describes the next-state function. The next state is zero if `rst` is high, or `count+1` otherwise.

A block diagram of this counter is shown in Figure 16.4. The figure illustrates the datapath nature of this implementation. The next state is computed by data flowing through paths involving combinational building blocks — including arithmetic blocks and multiplexers. In this case, the two building blocks are a

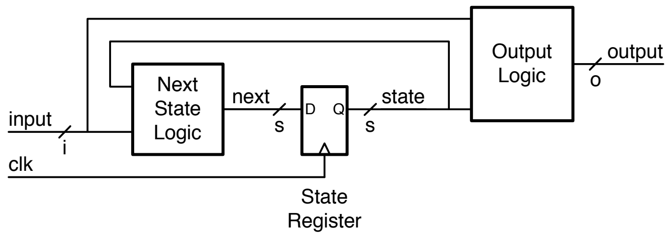


Figure 16.5: In general, a sequential datapath circuit consists of a state register, next-state logic, and output logic. The next-state and output modules are described by functions rather than tables.

multiplexer, which selects between reset (0) and increment (`count+1`), and an incrementer, which increments `count` to give `count+1`.

In general, a sequential datapath circuit takes the form shown in Figure 16.5. As with any synchronous sequential logic module, the state is held in a *state register*. An output logic module computes the output signals as a function of inputs and current state. The next state is computed by a next-state module as a function of inputs and current state. What distinguishes a datapath is that the next-state logic and output logic are described by functions rather than tables. For our simple counter, the output logic is just the current state, and the next state logic is a selection between incrementing or setting to zero. We will see more complex examples of sequential datapath circuits below. However, they all retain this functional description of the next-state and output functions.

16.1.2 An Up/Down/Load (UDL) Counter

Our simple counter had only two choices for next state: reset or increment. Often we require a counter with more options for the next state. A counter may need to count down (decrement) as well as count up, there may be cases where it needs to hold its value, and on occasion we may need to load an arbitrary value into the counter.

Figure 16.6 shows the Verilog description of such an up/down/load (UDL) counter. The counter has four control inputs (`rst, up, down,` and `load`, and one n-bit wide data input `in`. The next-state function is described by a `casex` statement. If reset (`rst`) is asserted ,the next state is 0. Otherwise, if `up` is asserted, the next state is `out+1`. If `down` is asserted (and not `up` or `rst`) the counter decrements by setting the next state to `out-1`. The counter is loaded when `load` is asserted by setting the next state to `in`. Finally, if none of the control inputs are asserted, the counter holds its present value by setting next to `out`.

This description accurately captures the function of the UDL counter and is adequate for most purposes. However, it is somewhat inefficient in that it will result in the instantiation of both an incrementer (to compute `out+1`) and a

```
module UDL_Count1(clk, rst, up, down, load, in, out) ;
  parameter n = 4 ;
  input clk, rst, up, down, load ;
  input [n-1:0] in ;
  output [n-1:0] out ;
  wire [n-1:0] out ;
  reg  [n-1:0] next ;

  DFF #(n) count(clk, next, out) ;

  always@(rst, up, down, load, in, out) begin
    casex({rst, up, down, load})
      4'b1xxx: next = {n{1'b0}} ;
      4'b01xx: next = out + 1'b1 ;
      4'b001x: next = out - 1'b1 ;
      4'b0001: next = in ;
      default: next = out ;
    endcase
  end
endmodule
```

Figure 16.6: Verilog description of an up/down/load (UDL) counter.

```
module UDL_Count2(clk, rst, up, down, load, in, out) ;
  parameter n = 4 ;
  input clk, rst, up, down, load ;
  input [n-1:0] in ;
  output [n-1:0] out ;
  wire [n-1:0] out, outpm1 ;
  reg  [n-1:0] next ;

  DFF #(n) count(clk, next, out) ;

  assign outpm1 = out + {{n-1{down}},1'b1} ;

  always@(rst, up, down, load, in, out, outpm1) begin
    casex({rst, up, down, load})
      4'b1xxx: next = {n{1'b0}} ;
      4'b01xx: next = outpm1 ;
      4'b001x: next = outpm1 ;
      4'b0001: next = in ;
      default: next = out ;
    endcase
  end
endmodule
```

Figure 16.7: Verilog description of an up/down/load (UDL) counter with a shared incrementer/decrementer.

decrementer (to compute `out-1`). From our brief study of computer arithmetic (Chapter 10) we know that these two circuits could be combined.

If we are in an operating mode where saving a few gates matters (which is unlikely) we can describe a more economical counter circuit as shown in Figure 16.7. This circuit factors the increment and decrement operations out of the `casex` statement. Instead, it realizes them in signal `outpm1` (out plus or minus 1). This signal is generated by an assign statement that adds 1 to `out` if `down` is false, and adds -1 to `out` if `down` is true. The code is otherwise identical to that of Figure 16.6.

A block diagram of the UDL counter is shown in Figure 16.8. Like the simple counter of Figure 16.4, and like most datapath circuits, the UDL counter consists of a multiplexer that selects different options for the next state. Some of the options are created by function units. Here the multiplexer has four inputs - to select the input (load), the output of the incrementer/decrementer (up or down), 0 (reset), and count (hold). The single function unit is an incrementer/decrementer that can either add or subtract 1 from the current count. The `down` line controls whether to increment or decrement. A block of combinational logic generates the select signals for the multiplexer by decoding the

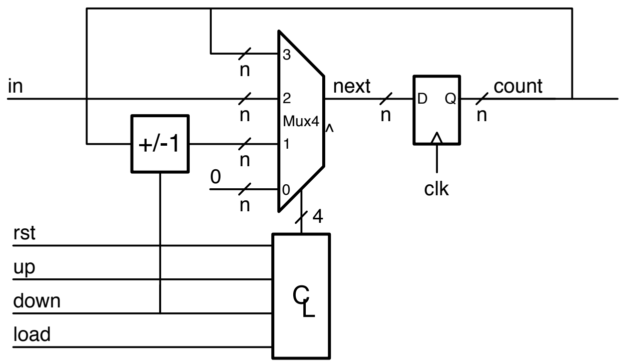


Figure 16.8: Block diagram of an up/down/load counter.

control inputs. Verilog code corresponding to this block diagram is shown in Figure 16.9.

16.1.3 A Timer

In many applications, like a more involved version of our traffic-light controller, we would like a timer that we can set with an initial time t and after t cycles have passed, it signals us that the time is complete. This is analogous to your kitchen timer that you set with an interval, and it signals you audibly when the interval is complete.

A block diagram of an FSM timer is shown in Figure 16.10. It follows our familiar theme of using a multiplexer to select the next state from among constants, inputs, and the outputs of function units (in this case a decrementer). What is different about this block diagram is that it includes an output function unit. A zero checker asserts signal `done` when the count has reached zero.

To operate the timer, the time interval is applied to input `in` and control signal `load` is asserted to load the interval. Each cycle after this load, the internal state `count` counts down. When `count` reaches zero, output `done` is asserted and counting stops. The reset input `rst` is only used to initialize the timer on power-up.

A Verilog description of the timer is shown in Figure 16.11. The style is similar to our simple counter and structural UDL counter with a multiplexer and a state register. The decrementer is implemented in the argument list of the multiplexer. To keep the timer from continuing to decrement, we select the zero input of the multiplexer when `done` is asserted and `load` is not. The final assign statement implements the zero checker.

```
module UDL_Count3(clk, rst, up, down, load, in, out) ;
  parameter n = 4 ;
  input clk, rst, up, down, load ;
  input [n-1:0] in ;
  output [n-1:0] out ;
  wire [n-1:0] out, next, outpm1 ;

  DFF #(n) count(clk, next, out) ;

  assign outpm1 = out + {{n-1{down}},1'b1} ;

  Mux4 #(n) mux(out, in, outpm1, {n{1'b0}},
                {(!rst & !up & !down & !load),
                 (!rst & !up & !down & load),
                 (!rst & (up | down)),
                 rst},
                next) ;
endmodule
```

Figure 16.9: Verilog description of an up/down/load (UDL) counter using a shared incrementer/decrementer and an explicit multiplexer.

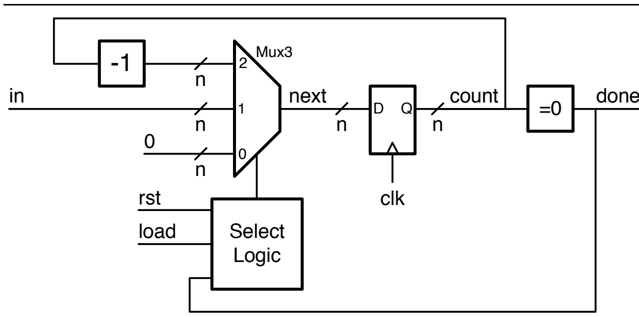


Figure 16.10: Block diagram of a timer FSM.

```
//----------------------------------------------------------------------
// Timer module
// Reset sets count to zero
// Load sets count to in
// Otherwise count decrements and saturates at zero (doesn't wrap)
// Done is asserted when count is zero
//----------------------------------------------------------------------
module Timer(clk, rst, load, in, done) ;
  parameter n=4 ;
  input clk, rst, load ;
  input [n-1:0] in ;
  output done ;
  wire [n-1:0] count, next_count ;

  DFF #(n) cnt(clk, next_count, count) ;
  Mux3 #(n) mux({n{1'b0}}, in, count-1'b1,
                {!rst & !load & !done,
                 load & !rst,
                 rst | (done & !load)},
                next_count) ;
  wire done = !(|count) ;
endmodule
//----------------------------------------------------------------------
```

Figure 16.11: Verilog description of a timer.

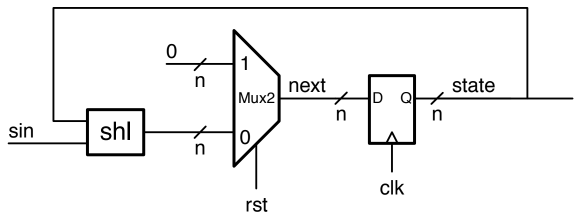


Figure 16.12: Block diagram of a simple shift register.

16.2 Shift Registers

In addition to incrementing, decrementing, and comparing, another popular datapath function is shifting. Shifters are particularly useful in serializers and deserializers that convert data from parallel to serial form and back again.

16.2.1 A Simple Shift Register

A block diagram of a simple shift register is shown in Figure 16.12 and a Verilog description of this module is given in Figure 16.13. Unless the shift register is reset, the next state is the current state shifted one-bit to the left with the `sin` input providing the rightmost bit (LSB). In the Verilog implementation, the 2:1 multiplexer is performed using the select "`? :` " construct, and the left shift is performed by the expression:

```
{out[n-2:0], sin}
```

Here the concatenate operation is used to concatenate the rightmost $n-1$ bits of `out` with `sin`, effectively shifting the bits of out one position to the left and inserting `sin` into the LSB.

This simple shift register could be used for example as a deserializer. A serial input is received on input `sin` and every n clocks the parallel output is read from `out`. A framing protocol of some type is necessary to determine where each parallel symbol starts and stops - i.e., during which clock the parallel output should be read.

16.2.2 Left/Right/Load (LRL) Shift Register

Analogous to our complex counter, we can make a complex shift register that can be loaded shifts in either direction. The block diagram of this left/right/load (LRL) shift register is shown in Figure 16.14 and the Verilog description is shown in Figure **??**. The module uses a five-input multiplexer to select between zero, a left shift (done with concatenation), a right shift (also done with concatenation), the input, and the output. Note that the shift expressions:

```
//----------------------------------------------------------------------
// Basic shift register
// rst - sets out to zero, otherwise out shifts left - sin becomes lsb
//----------------------------------------------------------------------
module Shift_Register1(clk, rst, sin, out) ;
  parameter n = 4 ;
  input clk, rst, sin ;
  output [n-1:0] out ;
  wire [n-1:0] out, next ;

  DFF #(n) cnt(clk, next, out) ;
  assign next = rst ? {n{1'b0}} : {out[n-2:0], sin} ;
endmodule
```

Figure 16.13: Verilog description of simple shift register. If `rst` is asserted the register is set to all 0s, otherwise it shifts left with input `sin` filling the LSB.

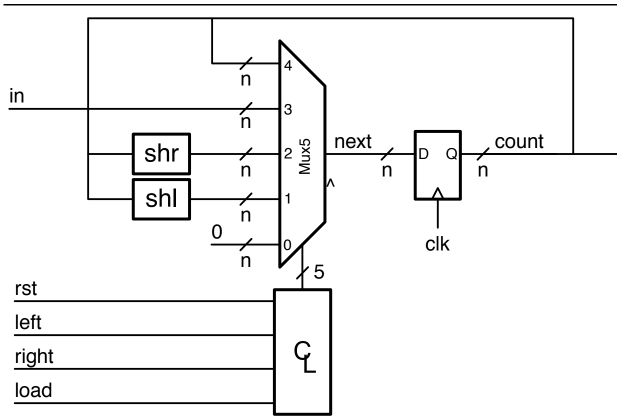


Figure 16.14: Block diagram of a left-right-load shift register.

```
//----------------------------------------------------------------------
// Left/Right SR with Load
//----------------------------------------------------------------------
module LRL_Shift_Register(clk, rst, left, right, load, sin, in, out) ;
  parameter n = 4 ;
  input clk, rst, left, right, load, sin ;
  input [n-1:0] in ;
  output [n-1:0] out ;
  wire [n-1:0] out, next ;

  DFF #(n) cnt(clk, next, out) ;
  Mux5 #(n) mux({n{1'b0}}, {out[n-2:0],sin}, {sin, out[n-1:1]}, in, out,
                {(!rst & !left & !right & !load),
                 (!rst & !left & !right & load),
                 (!rst & !left & right),
                 (!rst & left),
                 rst}, next) ;
endmodule
```

Figure 16.15: Verilog description of a left/right/load (LRL) shift register.

```
{out[n-2:0],sin}
{sin, out[n-1:1]}
```

do not actually generate any logic. This shift operation is just wiring - from `sin` and the selected bits of `out` to the appropriate input bits of the multiplexer.

16.2.3 A Universal Shifter/Counter

If we made the LRL shift register module also increment, decrement, and check for zero, it could serve in place of any of the modules we have discussed so far in this section. This idea is not as far-fetched as it may seem. At first you may think that using a full featured module when all we need is a simple counter is wasteful. However, most synthesis systems given constant inputs will eliminate logic that is never used (i.e., if `up` and `down` are always zero the incrementer/decrementer will be eliminated). Thus, the unused logic shouldn't cost us anything in practice.[2] The advantage of the universal shifter/counter is that its a single module to remember and maintain.

Verilog for a universal shifter/counter module is shown in Figure 16.16. This code largely combines the code of the individual modules above. It uses a seven-input multiplexer to select between input, current state, increment, decrement, left shift, right shift, and zero. The increment/decrement is done with an assign

[2]Check that this is true with your synthesis system before using this approach.

(as with the UDL counter) and the shifts are done with concatenation (as with the LRL shifter).

One thing that is new with the universal module is the use of an arbiter `RArb` to make sure that no two select inputs to the multiplexer are asserted at the same time. This arbitration was explicitly coded in the modules above. With seven select inputs its easier to instantiate the arbiter module to perform this logic. In many uses of the universal shifter/counter (or any of our other datapath modules) we can be certain that two command inputs will not be asserted at the same time. In these cases, we don't need the arbitration (whether done with a module or explicitly). However, to verify that the command inputs are in fact one-hot, it is useful to code an *assertion* into the module. The assertion is a logical expression that generates no gates but flags an error during simulation if the expression is violated.

16.3 Control and Data Partitioning

A common theme in digital design is the separation of a module into a control finite-state machine and a datapath as shown in Figure 16.17. The data path computes its next state via multiplexers and function units - like the counters and shift registers in this chapter. The control FSM, on the other hand, computes its next state via a state table. The inputs to the module are separated into control inputs - that affect the state of the control FSM, and data inputs, that supply values to the datapath. Module outputs are partitioned in a similar manner. The control FSM controls the operation of the datapath via a set of command signals. The datapath communicates back to the control FSM via a set of status signals.

The counter and shifter examples we have looked at so far in this chapter are degenerate examples of this organization. They each consist of a datapath - with multiplexers and function units (shifters, incrementers, and/or adders) - and a control unit. However, their control units have been strictly combinational. The datapath commands (e.g, multiplexer selects and add/subtract controls) and control outputs have been functions soley of the current control inputs and datapath status. In this section we will examine two examples of modules where the control section includes internal state.

16.3.1 Example: Vending Machine FSM

Consider the problem of designing the controller for a soft-drink vending machine. The specification is as follows: The vending machine accepts nickels, dimes, and quarters. Whenever a coin is deposited into the coin slot, a pulse appears for one clock cycle on one of three lines indicating the type of coin: `nickel`, `dime`, or `quarter`). The price of the item is set on an n-bit switch internal to the machine (in units of nickels) and is input to the controller on the n-bit signal `price`. When sufficient coins have been deposited to purchase a soft drink, an status signal `enough` is asserted. Any time `enough` is asserted and the

```
//----------------------------------------------------------------------
// Universal Shifter/Counter
// inputs take priority in order listed
// rst - resets state to zero
// left - shifts state to the left, sin fills LSB
// right - shifts state to the right, sin fills MSB
// up - increments state
// down - decrements state - will not decrement through zero.
// load - load from in
//
// Output done indicates when state is all zeros.
//----------------------------------------------------------------------
module UnivShCnt(clk, rst, left, right, up, down, load, sin, in, out, done) ;
  parameter n = 4 ;
  input clk, rst, left, right, up, down, load, sin ;
  input [n-1:0] in ;
  output [n-1:0] out ;
  output done ;
  wire [6:0] sel ; // multiplexer select, formed by arbiting the commands
  wire [n-1:0] out, next, outpm1 ;

  assign outpm1 = out + {{n-1{down}},1'b1} ; // incr or decr out
  DFF #(n) cnt(clk, next, out) ;
  RArb #(7) arb({rst, left, right, up, down & ~done, load, 1'b1}, sel) ;
  Mux7 #(n) mux(out, in, outpm1, outpm1,
              {sin,out[n-1:1]},{out[n-2:0],sin},{n{1'b0}}, sel, next) ;
  wire done = !(|out) ;
endmodule
```

Figure 16.16: Verilog description of an up/down/load (UDL) counter using a shared incrementer/decrementer and an explicit multiplexer.

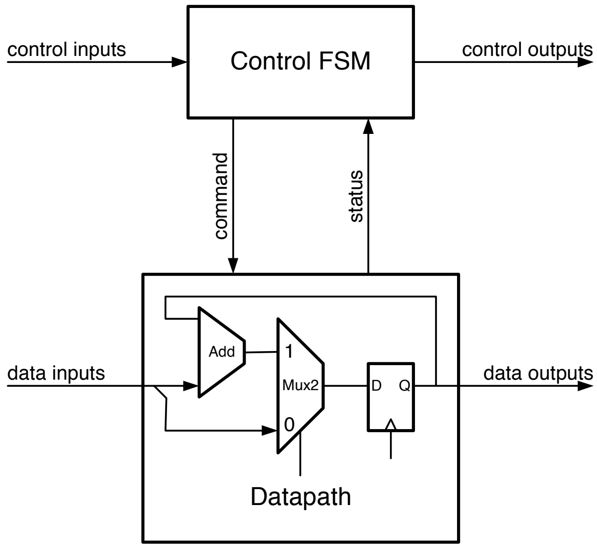


Figure 16.17: Systems are often partitioned into datapath, where the next state is determined by function units, and control, where the next state is determined by state tables.

user presses a `dispense` button, signal `serve` is asserted for exactly one cycle to serve the soft-drink. After asserting `serve`, the FSM must wait until signal `done` is asserted indicating that the mechanism has finished serving the soft drink. After serving is done, the machine returns change (if any) to the user. It does this one nickel at a time, asserting the signal `change`, for exactly one cycle and waiting for signal `done` to indicate that a nickel has been dispensed before dispensing the next nickel or returning to its original state. Any time the signal `done` is asserted, we must wait for `done` to go low before proceeding.

To design this machine, we start by considering the control part of the machine. First, lets look at the inputs and outputs. All of the inputs except `price` (`rst`, `nickel`, `dime`, `quarter`, `dispense`, and `done`) are control inputs, and all of the outputs `serve` and `change` are control outputs. Status that we need from the datapath includes a signal `enough` that indicates that enough money has been deposited, and a signal `zero` that indicates that no more change is owed. Commands to the datapath we will consider below when we look at the data state.

Now lets consider the states of the control portion of the machine. As illustrated in the state diagram of Figure 16.18 operates in three main phases. [3] First, during the deposit phase (state deposit), the user deposits coins and then

[3]In this diagram, edges from a state to itself are omitted. If the conditions on all edges leading out of the current state are not satisfied, the FSM stays in that state.

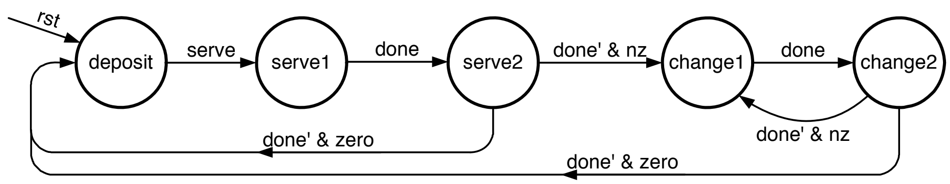


Figure 16.18: State diagram for control part of a vending machine controller. Edges from a state to itself are omitted for clarity. If none of the logic conditions on edges out of a state are true, the machine will remain in that state.

presses `dispense`. When signal `dispense` and `enough` are both asserted the machine advances to the serving phase (states serve1 and serve2). In state serve1 it waits for signal `done` to indicate that the soft drink has been served, and in state serve2, it waits for `done` to be deasserted. We assert the output `serve` during the first cycle only of state serve1. If there is no change to be dispensed, `zero` is true, the FSM returns to the deposit state from serve2. However, if there is change to be dispensed, the FSM enters the change phase (states change1 and change2). The machine cycles through these states, asserting `change` and waiting for `done` in change1 and waiting for `done` to go low in `change2`. We assert output `change` during the first cycle of each visit to state change1. Only when all change has been dispensed do we return to the deposit state.

Now that we have the control states defined, we turn our attention to the data state. This FSM has a single piece of data state - the amount of money the machine currently owes the user - in units of nickels. We'll call this state variable `amount`. The different actions that affect `amount` are:

Reset: amount $\leftarrow$ 0.

Deposit a coin: amount $\leftarrow$ amount + value, where value = 1, 2, or 5 for a nickel, dime, or quarter respectively.

Serve a drink: amount $\leftarrow$ amount - price.

Return one nickel of change: amount $\leftarrow$ amount - 1.

Otherwise: No change in amount.

We can now design a datapath that supports these operations. State variable `amount` can either be zeroed, added to or subtracted from, or hold its value. From these *register transfers* we see that we need the datapath of Figure 16.19. The next state for amount `next` is selected by a 3:1 multiplexer that selects between 0, `amount`, or `sum`, the output of an add/subtract unit that adds or subtracts `value` from `amount`. The `value` is selected by a 4:1 multiplexer to be 1, 2, 5, or `price`. We see from the figure that the command signals needed to

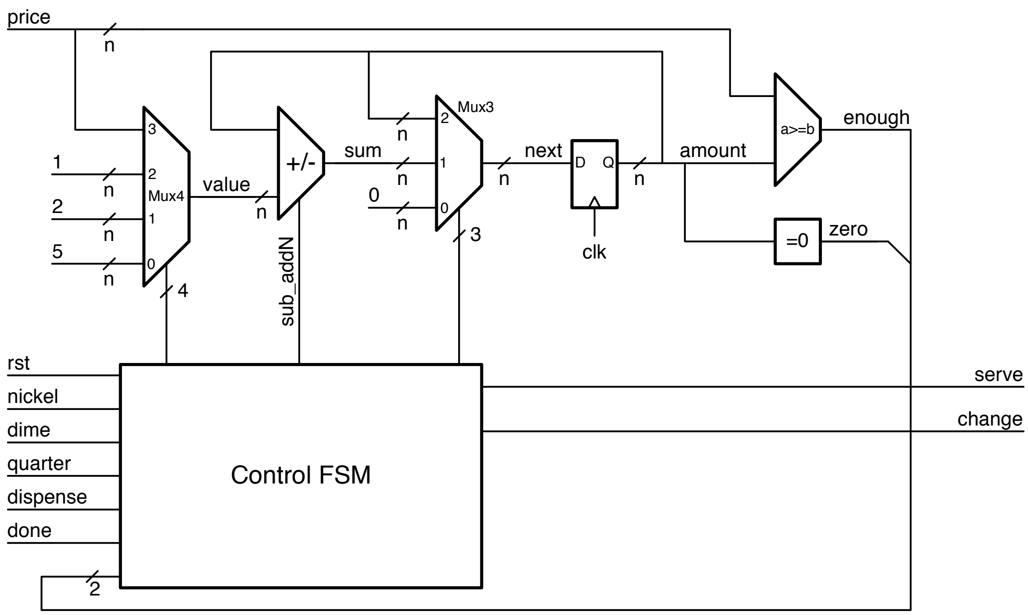


Figure 16.19: Block diagram of a vending machine controller.

```
//----------------------------------------------------------------------
// VendingMachine - Top level module
// Just hooks together control and datapath
//----------------------------------------------------------------------
module VendingMachine(clk, rst, nickel, dime, quarter, dispense, done, price,
                      serve, change) ;
  parameter n = `DWIDTH ;
  input clk, rst, nickel, dime, quarter, dispense, done ;
  input [n-1:0] price ;
  output serve, change ;

  wire enough, zero, sub ;
  wire [3:0] selval ;
  wire [2:0] selnext ;

  VendingMachineControl vmc(clk, rst, nickel, dime, quarter, dispense, done,
  enough, zero, serve, change, selval, selnext, sub) ;

  VendingMachineData #(n) vmd(clk, selval, selnext, sub, price, enough, zero) ;
endmodule
```

Figure 16.20: Verilog top-level module for vending-machine controller just declares the control and data modules and connects them together.

control the datapath are the two multiplexer select signals and the add/subtract control.

Verilog code for our vending machine controller is shown in Figures 16.20 through 16.23. The top-level module is shown in Figure 16.20. This module just instantiates the control (`VendingMachineControl`) and data (`VendingMachineData`) modules and connects them up. The command signals `sub`, `selval` and `selnext` are declared at this level as are the status signals `enough` and `zero`.

The Verilog for the control module of our vending machine controller is split over two figures. Figure 16.21 shows the first half of the module which includes the logic for generating the output and command variables. A variable `first` is defined which is used to distinguish the first cycle of states serve1 and change1. After one cycle in either of these states, `first` goes low. This variable enables us to only assert the outputs for one cycle on each visit to these states and to only decrement amount once on each visit to the change1 state. Without the `first` variable, we would need to expand both serve1 and change1 into two states. The outputs `serve` and `change` are generated by ANDing `first` with signals that are true in states serve1 and change1 respectively.

The datapath control signals are determined from input and state variables. The two select signals are one-hot variables. Each bit is determined by a logic expression. Signal `selval` (which selects the value input to the add/subtract unit) selects price when in the deposit state (`dep` true) and dispense is pressed. A 1 is selected if a nickel is input in the deposit state or if we are in the change state. Inputs of 2 and 5 are selected in the deposit state if a dime or a quarter are input respectively. Signal `selnext` selects the next state for the amount variable. Zero is selected if `rst` is true, and the output of the add/subtract unit is selected if variable `selv` is true. Otherwise the current value of amount is selected. Variable `selv` is true in the deposit state if a coin is entered or if dispense and enough are both true and in the change state if first is true. Note that we can select `price` to be the value whenever `dispense` is asserted in the deposit state because we only select the add/subtract output if `enough` is also asserted. Finally, the add/subtract control signal is set to subtract on `dispense` and `change` actions. Otherwise the adder adds.

The second half of the control module in Figure 16.22 shows the next state logic. This logic is implemented as a `casex` statement where the case is on the concatenation of four input bits and the current state. Most transitions encode to a single case. For example, when the machine is in the deposit state, and `dispense` and `enough` are both true, the first case `4'b11xx,'DEPOSIT` is active. It takes the next two cases to encode when the machine stays in the deposit state. A separate assignment statement is used reset the FSM to the deposit state.

A Verilog description of the vending machine controller datapath is shown in Figure 16.23. This code follows the datapath part of Figure **??** very closely. A state register holds the current amount. A three-input multiplexer feeds the state register with zero, the `sum` output of the add/subtract unit, or the current value of amount. An add subtract unit adds or subtracts `value` to or from `amount`. A four-input multiplexer selects the `value` to be added or subtracted.

```
module VendingMachineControl(clk, rst, nickel, dime, quarter, dispense, done,
  enough, zero, serve, change, selval, selnext, sub) ;
  input clk, rst, nickel, dime, quarter, dispense, done, enough, zero ;
  output serve, change, sub ;
  output [3:0] selval ;
  output [2:0] selnext ;
  wire [`SWIDTH-1:0] state, next ; // current and next state
  reg [`SWIDTH-1:0] next1 ;        // next state w/o reset

  // outputs
  wire first ; // true during first cycle of serve1 or change1
  wire serve1 = (state == `SERVE1) ;
  wire change1 = (state == `CHANGE1) ;
  wire serve = serve1 & first ;
  wire change = change1 & first ;

  // datapath controls
  wire dep = (state == `DEPOSIT) ;
  // price, 1, 2, 5
  wire [3:0] selval = {(dep & dispense),
                       ((dep & nickel)| change),
                       (dep & dime),
                       (dep & quarter)} ;
  // amount, sum, 0
  wire selv = (dep & (nickel | dime | quarter | (dispense & enough))) |
             (change & first) ;
  wire [2:0] selnext = {!(selv | rst),selv,rst} ;

  // subtract
  wire sub = (dep & dispense) | change ;

  // only do actions on first cycle of serve1 or change1
  wire nfirst = !(serve1 | change1) ;
  DFF #(1) first_reg(clk, nfirst, first) ;
```

Figure 16.21: Verilog description of control module for vending machine controller (part 1 of 2). This first half of the control module shows the output and command variables implemented with assign statements.

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

  // next state logic
  always @(state or zero or dispense or done or enough) begin
    casex({dispense, enough, done, zero, state})
      {4'b11xx,`DEPOSIT}:  next1 = `SERVE1 ;  // dispense & enough
      {4'b0xxx,`DEPOSIT}:  next1 = `DEPOSIT ;
      {4'bx0xx,`DEPOSIT}:  next1 = `DEPOSIT ;
      {4'bxx1x,`SERVE1}:   next1 = `SERVE2 ;  // done
      {4'bxx0x,`SERVE1}:   next1 = `SERVE1 ;
      {4'bxx01,`SERVE2}:   next1 = `DEPOSIT ;  // ~done & zero
      {4'bxx00,`SERVE2}:   next1 = `CHANGE1 ;  // ~done & ~zero
      {4'bxx1x,`SERVE2}:   next1 = `SERVE2  ;  // done
      {4'bxx1x,`CHANGE1}:  next1 = `CHANGE2 ;  // done
      {4'bxx0x,`CHANGE1}:  next1 = `CHANGE1 ;  // done
      {4'bxx00,`CHANGE2}:  next1 = `CHANGE1 ;  // ~done & ~zero
      {4'bxx01,`CHANGE2}:  next1 = `DEPOSIT ;  // ~done & zero
      {4'bxx1x,`CHANGE2}:  next1 = `CHANGE2 ;  // ~done & zero
    endcase
  end

  // reset next state
  assign next = rst ? `DEPOSIT : next1 ;
endmodule
```

Figure 16.22: Verilog description of control module for vending machine controller (part 2 of 2). This second half of the control module shows the next-state function implemented with a `casex` statement.

```
module VendingMachineData(clk, selval, selnext, sub, price, enough, zero) ;
  parameter n = 6 ;
  input clk, sub ;
  input [3:0] selval ;  // price, 1, 2, 5
  input [2:0] selnext ; // amount, sum, 0
  input [n-1:0] price ; // price of soft drink - in nickels
  output enough ;       // amount > price
  output zero ;         // amount = zero

  wire [n-1:0] sum ;  // output of add/subtract unit
  wire [n-1:0] amount ; // current amount
  wire [n-1:0] next ;   // next amount
  wire [n-1:0] value ;  // value to add or subtract from amount
  wire ovf ;            // overflow - ignore for now

  // state register holds current amount
  DFF #(n) amt(clk, next, amount) ;

  // select next state from 0, sum, or hold
  Mux3 #(n) nsmux({n{1'b0}}, sum, amount, selnext, next) ;

  // add or subtract a value from current amount
  AddSub #(n) add(amount, value, sub, sum, ovf) ;

  // select the value to add or subtract
  Mux4 #(n) vmux(`QUARTER, `DIME, `NICKEL, price, selval, value) ;

  // comparators
  wire enough = (amount >= price) ;
  wire zero = (amount == 0) ;
endmodule
```

Figure 16.23: Datapath for vending machine controller.

Finally, two assign statements generate the status signals `enough` and `zero`.

A testbench for the vending machine controller is shown in Figure 16.24 and waveforms from simulating the controller with this test bench are shown in Figure 16.25. The test begins by resetting the machine. A nickel is then deposited followed by a dime - bringing `amount` to 3 (15 cents). At this point, we go one cycle with no input to make sure `amount` stays at 3. We then assert `dispense` to make sure that trying to dispense a soft drink before enough money has been deposited doesn't work. Next we deposit two quarters in back-to-back cycles bringing `amount` to 8 and then 13. When `amount` reaches 13 signal `enough` goes high since we have exceeded the price (which is 11). After an idle cycle, we again assert `dispense`. This time it works. The state advances to 001 (serve1) and `amount` is reduced to 2 (price of 11 deducted).

In the first cycle of the serve1 state (state = 001) `serve` is asserted. The machine remains in this state for one more cycle waiting for `done` to go high. It spends just one cycle in state serve2 (state = 011) since `done` is already low and continues into state change1 (state = 010). On the first cycle of change1, output `change` is asserted (to return an nickel to the user) and `amount` is decremented to 1. The machine remains in the change1 state for one more cycle waiting for `done`. It then spends just one cycle in state change2 (state = 100) before returning to change1 - because zero is not true. Again `change` is asserted and `amount` is decremented on the first cycle of change1. This time, however `amount` is decremented to zero and signal `zero` is asserted. After waiting a second cycle in the change1 state, the machine transitions through change2 and back to the deposit state - since `zero` is set - ready to start again.

16.3.2 Example: Combination Lock

As our second control and datapath example, consider an electronic combination lock that accepts input from a decimal keypad. The user must enters the code as a sequence of decimal digits and then presses an `enter` key. If the user entered the correct sequence, the `unlock` output is asserted (presumably to actuate a large bolt that unlocks a door). To relock the machine, the user presses `enter` a second time. If the user entered an incorrect sequence, a `busy` output is asserted and a timer is activated to wait a predefined period before allowing the user to try again. The busy light should not come on until the user had entered the entire sequence and pressed `enter`. If the light were to come on at the first incorrect keypress, this gives the user information that can be used to discover the code one digit at a time.

There are three inputs to our lock system: `key`, `key_valid`, and `enter`. A four-bit code `key` indicates the current key being pressed and is accompanied by a signal `key_valid` that indicates when a key is valid. The keyboard is preprocessed so that every key press asserts key and valid for exactly one cycle. In a similar manner, the `enter` signal is preprocessed so that it is for exactly one cycle each time the user presses the enter key to unlock or relock the bolt. The length of the sequence is set by an internal variable `length`, and the sequence itself is stored in an internal memory.

```
module testVend ;
  reg clk, rst, nickel, dime, quarter, dispense, done ;
  reg [3:0] price ;
  wire serve, change ;

  VendingMachine #(4) vm(clk, rst, nickel, dime, quarter, dispense, done, price,
                         serve, change) ;

  // clock with period of 10 units
  initial begin
    clk = 1 ; #5 clk = 0 ;
    forever
      begin
        $display("%b %h %h %b %b",
        {nickel,dime,quarter,dispense}, vm.vmc.state, vm.vmd.amount, serve, change) ;
        #5 clk = 1 ; #5 clk = 0 ;
      end
    end

  // give prompt feedback
  always @(posedge clk) begin
    done = (serve | change) ;
  end

  initial begin
    rst = 1 ; {nickel, dime, quarter, dispense} = 4'b0 ; price = `PRICE ;
    #25 rst = 0 ;
    #10 {nickel, dime, quarter, dispense} = 4'b1000 ; // nickel 1
    #10 {nickel, dime, quarter, dispense} = 4'b0100 ; // dime 3
    #10 {nickel, dime, quarter, dispense} = 4'b0000 ; // nothing
    #10 {nickel, dime, quarter, dispense} = 4'b0001 ; // try to dispense early
    #10 {nickel, dime, quarter, dispense} = 4'b0010 ; // quarter 8
    #10 {nickel, dime, quarter, dispense} = 4'b0010 ; // quarter 13
    #10 {nickel, dime, quarter, dispense} = 4'b0000 ; // nothing
    #10 {nickel, dime, quarter, dispense} = 4'b0001 ; // dispense 2
    #10 dispense = 0 ;
    #100 $stop ;
  end
endmodule
```

Figure 16.24: Verilog testbench for vending machine controller.

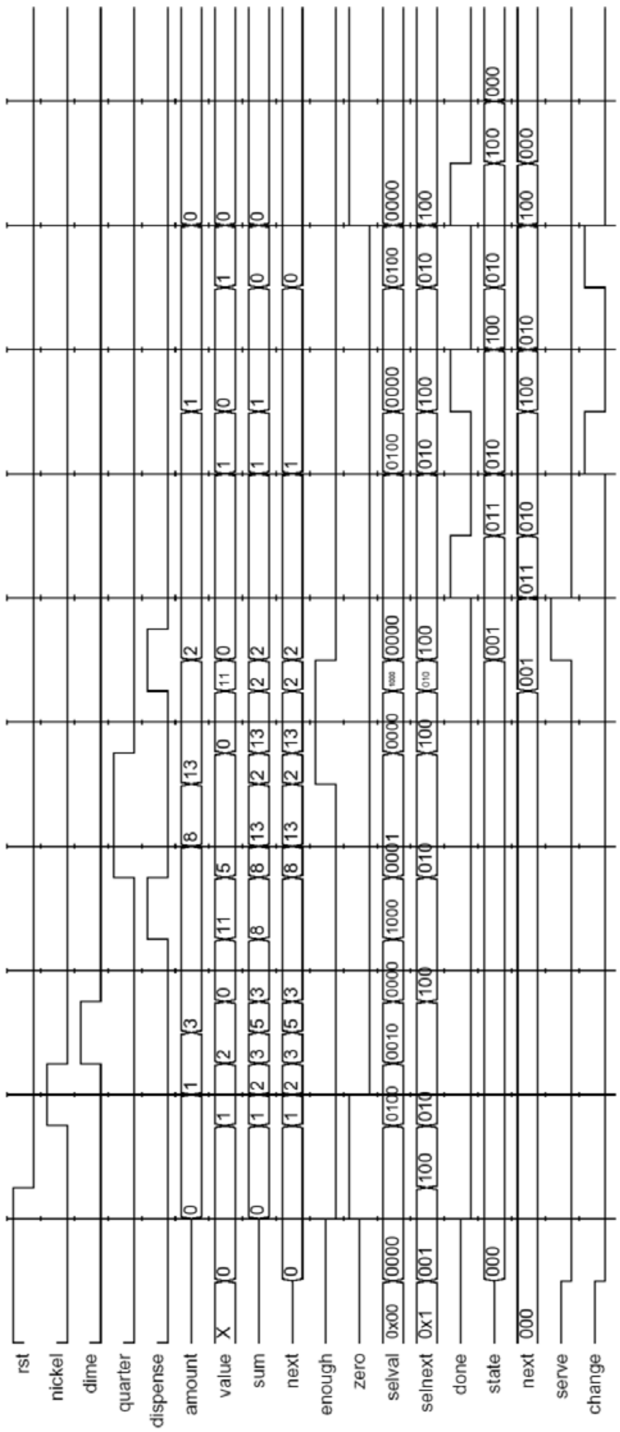


Figure 16.25: Waveforms from simulating the vending machine controller with the test bench of Figure 16.24

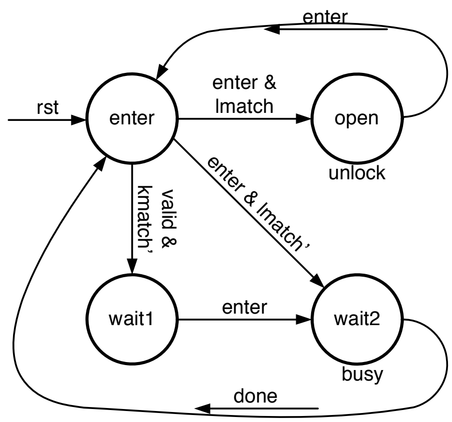


Figure 16.26: State diagram for control portion of combination lock.

A state diagram describing the control portion of our combination lock machine is shown in Figure 16.26. The machine resets to the enter state. In this state the machine accepts input. Each time a key is pressed, it is checked against the expected digit. If its correct (kmatch) the machine stays in the enter state, if not ($\texttt{valid} \wedge \overline{\texttt{kmatch}}$) the machine transitions to the wait1 state. The machine waits in the wait1 state until `enter` is pressed, and then enters the wait2 state. The machine starts a timer in the wait2 state and remains in this state, asserting `busy` until the timer signals `done`.

When the entire code has been entered correctly, the machine will still be in the enter state - an incorrect digit would have taken it to wait1 - and `lmatch` will be true - the length of the code entered matches the internal `length` variable. If `enter` is pressed at this point, the machine transitions to the open state ($\texttt{enter} \wedge \texttt{lmatch}$) and the bolt is unlocked. A second `enter` in the open state returns to the reset. If in the enter state, the `enter` key is pressed when the length does not match (either too few or too many digits in the codeword) ($\texttt{enter} \wedge \overline{\texttt{lmatch}}$) the machine goes to the wait2 state.

A block diagram of the datapath portion of our combination lock is shown in Figure 16.27. The datapath has two distinct sections. The upper section compares the length and value of the code entered. The lower section times the wait period when an incorrect code is entered. This section consists of a single timer module that loads the timer interval `twait` when control signal `load` is asserted. The timer then counts down. When it reaches zero, status signal `done` is asserted.

The upper half of the datapath consists of a counter, a ROM, and two comparators. The counter keeps track of which digit of the code we are on. It is reset to zero before entering the enter state and then counts up as each key is pressed. The counter output, index, selects the digit to be compared next.

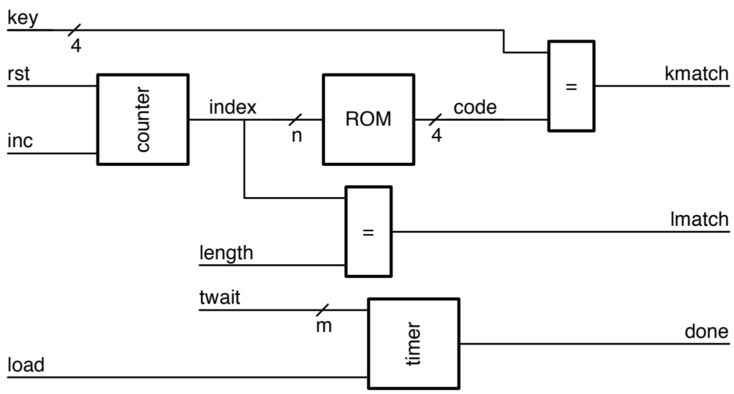


Figure 16.27: Block diagram of datapath portion of combination lock.

Signal index is compared to length to generate status signal `lmatch`. It is also used as an address for the ROM containing the code. The current digit of the code is read from the ROM, signal code, and compared to the key entered by the user to generate status signal `kmatch`.

Figures 16.28 and 16.29 show a Verilog implementation of our combination lock. Here we have put both the control and datapath in a single module. This eliminates the code that is otherwise required to wire together the control and data sections, but for a large module can result in an unweildy piece of code.

The datapath portion of the module is shown in Figure 16.28. The Verilog follows directly from the block diagram of Figure 16.27. An up/down/load counter (Section 16.1.2) is used for the counter. Only the `rst` and `up` control inputs are used. The `load` and `down` are zero. The synthesizer will take advantage of these zero control inputs when synthesizing the counter and eliminate the unused logic. The timer module of Section 16.1.3 is used to count down in the wait2 state. The two comparators are implemented with assignment statements.

Figure 16.29 shows the control portion of the Verilog description of the combination lock module. The top portion of the code generates the output and next state variables. Outputs `busy` and `unlock` are generated by decoding the state variable `state` since these outputs are true whenever the machine is in the wait2 and open states respectively. The enter and wait1 states are decoded as well for use in the command equations. The `rstctr` command signal is set to reset the digit counter on reset or in states that transition to the enter state (wait2 and open) so the counter is zeroed and ready to sequence digits in the enter state. Similarly the `load` signal loads the timer in states that transition to the wait2 state (enter and wait1) so it can count down in wait2. The `inc` signal increments the counter in the enter state each time a key is pressed.

The next-state function is implemented with a `casex` statement with a sep-

```
//----------------------------------------------------------------------
// CombLock
// Inputs:
//   key - (4-bit) accepts a code digit each time key_valid is true
//   key_valid - signals when a new code digits is on key
//   enter - signals when entire code has been entered
// Outputs:
//   busy - asserted after incorrect code word entered during timeout
//   unlock - asserted after correct codeword is entered until enter
//            is pressed again.
//----------------------------------------------------------------------
module CombLock(clk, rst, key, key_valid, enter, busy, unlock) ;
  parameter n = 4 ; // bits of code length
  parameter m = 4 ; // bits of timer
  input clk, rst, key_valid, enter ;
  input [3:0] key ;
  output busy, unlock ;

  //--- datapath ----------------------------------------
  wire rstctr ; // reset the digit counter
  wire inc ;    // increment the digit counter
  wire load ;   // load the timer
  wire done ;   // timer done
  wire [n-1:0] index ;
  wire [3:0] code ;
  UDL_Count #(n) ctr(clk, rstctr, inc, 1'b0, 1'b0, 4'b0, index) ; // counter
  ROM #(n,4) rom(index, code) ; // ROM storing the code
  Timer #(m) tim(clk, rst, load, `TWAIT, done) ; // wait timer
  wire kmatch = (code == key) ;       // key comparator
  wire lmatch = (index == `LENGTH) ;  // length comparator
```

Figure 16.28: Verilog description of the combination lock (part 1 of 2). This section describes the datapath and closely follows Figure 16.27.

```
  //--- control ---------------------------------------
  wire [`SWIDTH-1:0] state, next ;       // current and next state
  reg  [`SWIDTH-1:0] next1 ;             // for reset
  wire senter = (state == `ENTER) ;      // decode state
  wire unlock = (state == `OPEN) ;
  wire busy = (state == `WAIT2) ;
  wire swait1 = (state == `WAIT1) ;
  assign rstctr = rst | unlock | busy ;  // reset before returning to enter
  assign inc = senter & key_valid ;      // increment on each key entry
  assign load = senter | swait1 ;        // load before entering wait2

  DFF #(`SWIDTH) sr(clk, next, state) ; // state register

  always @(enter or lmatch or key_valid or kmatch or done or state) begin
    casex({enter, lmatch, key_valid, kmatch, done,state})
      {5'bxx10x,`ENTER}: next1 = `WAIT1 ;  // valid & ~kmatch
      {5'b0x11x,`ENTER}: next1 = `ENTER  ; // valid & kmatch
      {5'b110xx,`ENTER}: next1 = `OPEN  ;  // enter & lmatch
      {5'b10xxx,`ENTER}: next1 = `WAIT2  ; // enter & ~lmatch
      {5'b0x0xx,`ENTER}: next1 = `ENTER  ; // ~enter & ~valid

      {5'b1xxxx,`OPEN}: next1 = `ENTER  ; // enter
      {5'b0xxxx,`OPEN}: next1 = `OPEN   ; // ~enter

      {5'b1xxxx,`WAIT1}: next1 = `WAIT2   ; // enter
      {5'b0xxxx,`WAIT1}: next1 = `WAIT1   ; // ~enter

      {5'bxxxx1,`WAIT2}: next1 = `ENTER   ; // done
      {5'bxxxx0,`WAIT2}: next1 = `WAIT2   ; // ~done
    endcase
  end
  assign next = rst ? `ENTER : next1 ; // reset
endmodule
```

Figure 16.29: Verilog description of the combination lock (part 2 of 2). This section describes the control. Assign statements generate the command and output signals. A `casex` statement computes the next-state function.

arate `assign` statement to reset to the enter state. The case variable is a concatenation of the five input and status signals that affect the next state function with the current state.

Waveforms from simulating the combination lock module on a sequence of test inputs are shown in Figure 16.30. The test visits all states and traverses all edges of the state diagram of Figure 16.26. This is done with three attempts to unlock the lock. One correct attempt and two failures. After reset, the test first enters the correct sequence with a pause after the 1 and after the 7. On the cycle after entering the final 8, `enter` is pressed and the next cycle `unlock` goes high and the machine enters the open state (`state = 1`. After one cycle low, `enter` goes high again returning the machine to the enter state.

The second attempt to unlock the lock involves entering an invalid code. As soon as the first key is entered incorrectly (7 instead of 8) the machine transits to the wait1 state (`state` = 2). It stays in this state until `enter` is pressed at which time it goes to the wait2 state (`state` = 3). In the wait2 state the `busy` output is asserted and the timer starts counting down from 4.[4]

The third attempt to unlock the lock involves entering a code of the wrong length. After the first digit is entered correctly, `enter` is asserted. Because the correct length code has not yet been entered (`lmatch` =0), the machine transits to the wait2 state and starts counting down the timer.

16.4 Bibliographic Notes

16.5 Exercises

16–1 - design a counter that can count up, count down, load and that saturates counting up (at a programmable max count) and counting down (at zero).

16–2 - design a datapath circuit to compute Fibonacci numbers. Each cycle the circuit should output the next Fibonacci number (starting with 0 on the first cycle). The circuit should have parameterized width and should signal when the next number is too large to be represented at this width.

16–3 - redesign the fibonacci number circuit of Exercise 16–2 to be 32-bits wide but use only an 8-bit adder.

16–4 - vending machine with extended coin pulse.

16–5 - vending machine with saturating add

16–6 - combination lock with multiple users - first digit selects one of 8 codes.

16–7 - combination lock that forgives one error. optional, calculate probability of guessing

[4] In practice we would use a much longer timeout. However using a short timeout greatly reduces simulation time and results in an easier to read set of waveforms.

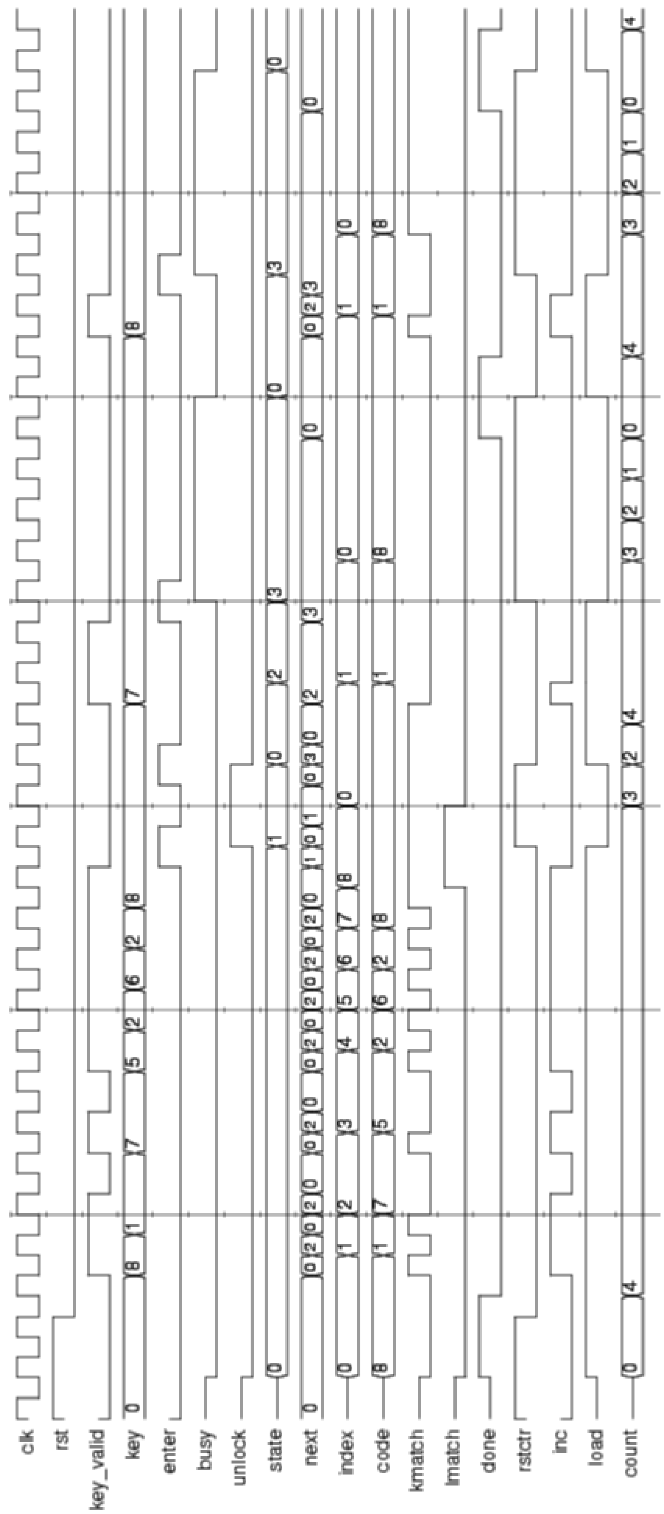


Figure 16.30: Waveforms from simulating the combination lock module of Figures 16.28 and 16.29.