ECE120: Introduction to Computer Engineering

Notes Set 3.5 Finite State Machine Design Examples, Part II

This set of notes provides several additional examples of FSM design. We first design an FSM to control a vending machine, introducing encoders and decoders as components that help us to implement our design. We then design a game controller for a logic puzzle implemented as a children's game. Finally, we analyze a digital FSM designed to control the stoplights at the intersection of two roads.

3.5.1 Design of a Vending Machine

For the next example, we design an FSM to control a simple vending machine. The machine accepts U.S. coins[12] as payment and offers a choice of three items for sale.

What states does such an FSM need? The FSM needs to keep track of how much money has been inserted in order to decide whether a user can purchase one of the items. That information alone is enough for the simplest machine, but let's create a machine with adjustable item prices. We can use registers to hold the item prices, which we denote P_1, P_2, and P_3.

Technically, the item prices are also part of the internal state of the FSM. However, we leave out discussion (and, indeed, methods) for setting the item prices, so no state with a given combination of prices has any transition to a state with a different set of item prices. In other words, any given combination of item prices induces a subset of states that operate independently of the subset induced by a distinct combination of item prices. By abstracting away the prices in this way, we can focus on a general design that allows the owner of the machine to set the prices dynamically.

Our machine will not accept pennies, so let's have the FSM keep track of how much money has been inserted as a multiple of 5 cents (one nickel). The table to the right shows five types of coins, their value in dollars, and their value in terms of nickels.

coin type	value	# of nickels
nickel	\$0.05	1
dime	\$0.10	2
quarter	\$0.25	5
half dollar	\$0.50	10
dollar	\$1.00	20

The most expensive item in the machine might cost a dollar or two, so the FSM must track at least 20 or 40 nickels of value. Let's decide to use six bits to record the number of nickels, which allows the machine to keep track of up to \$3.15 (63 nickels). We call the abstract states **STATE00** through **STATE63**, and refer to a state with an inserted value of N nickels as **STATE**<N>.

Let's now create a next-state table, as shown at the top of the next page. The user can insert one of the five coin types, or can pick one of the three items. What should happen if the user inserts more money than the FSM can track? Let's make the FSM reject such coins. Similarly, if the user tries to buy an item without inserting enough money first, the FSM must reject the request. For each of the possible input events, we add a condition to separate the FSM states that allow the input event to be processed as the user desires from those states that do not. For example, if the user inserts a quarter, those states with $N < 59$ transition to states with value $N + 5$ and accept the quarter. Those states with $N \geq 59$ reject the coin and remain in **STATE**<N>.

[12] Most countries have small bills or coins in demoninations suitable for vending machine prices, so think about some other currency if you prefer.

initial state	input event	condition	final state: state	final state: accept coin	final state: release product
STATE$<N>$	no input	always	**STATE**$<N>$	—	none
STATE$<N>$	nickel inserted	$N < 63$	**STATE**$<N+1>$	yes	none
STATE$<N>$	nickel inserted	$N = 63$	**STATE**$<N>$	no	none
STATE$<N>$	dime inserted	$N < 62$	**STATE**$<N+2>$	yes	none
STATE$<N>$	dime inserted	$N \geq 62$	**STATE**$<N>$	no	none
STATE$<N>$	quarter inserted	$N < 59$	**STATE**$<N+5>$	yes	none
STATE$<N>$	quarter inserted	$N \geq 59$	**STATE**$<N>$	no	none
STATE$<N>$	half dollar inserted	$N < 54$	**STATE**$<N+10>$	yes	none
STATE$<N>$	half dollar inserted	$N \geq 54$	**STATE**$<N>$	no	none
STATE$<N>$	dollar inserted	$N < 44$	**STATE**$<N+20>$	yes	none
STATE$<N>$	dollar inserted	$N \geq 44$	**STATE**$<N>$	no	none
STATE$<N>$	item 1 selected	$N \geq P_1$	**STATE**$<N-P_1>$	—	1
STATE$<N>$	item 1 selected	$N < P_1$	**STATE**$<N>$	—	none
STATE$<N>$	item 2 selected	$N \geq P_2$	**STATE**$<N-P_2>$	—	2
STATE$<N>$	item 2 selected	$N < P_2$	**STATE**$<N>$	—	none
STATE$<N>$	item 3 selected	$N \geq P_3$	**STATE**$<N-P_3>$	—	3
STATE$<N>$	item 3 selected	$N < P_3$	**STATE**$<N>$	—	none

We can now begin to formalize the I/O for our machine. Inputs include insertion of coins and selection of items for purchase. Outputs include a signal to accept or reject an inserted coin as well as signals to release each of the three items.

For input to the FSM, we assume that a coin inserted in any given cycle is classified and delivered to our FSM using the three-bit representation shown to the right. For item selection, we assume that the user has access to three buttons, B_1, B_2, and B_3, that indicate a desire to purchase the corresponding item.

coin type	$C_2C_1C_0$
none	110
nickel	010
dime	000
quarter	011
half dollar	001
dollar	111

For output, the FSM must produce a signal A indicating whether a coin should be accepted. To control the release of items that have been purchased, the FSM must produce the signals R_1, R_2, and R_3, corresponding to the release of each item. Since outputs in our class depend only on state, we extend the internal state of the FSM to include bits for each of these output signals. The output signals go high in the cycle after the inputs that generate them. Thus, for example, the accept signal A corresponds to a coin inserted in the previous cycle, even if a second coin is inserted in the current cycle. This meaning must be made clear to whomever builds the mechanical system to return coins.

Now we are ready to complete the specification. How many states does the FSM have? With six bits to record money inserted and four bits to drive output signals, we have a total of 1,024 (2^{10}) states! Six different coin inputs are possible, and the selection buttons allow eight possible combinations, giving 48 transitions from each state. Fortunately, we can use the meaning of the bits to greatly simplify our analysis.

First, note that the current state of the coin accept bit and item release bits—the four bits of FSM state that control the outputs—have no effect on the next state of the FSM. Thus, we can consider only the current amount of money in a given state when thinking about the transitions from the state. As you have seen, we can further abstract the states using the number N, the number of nickels currently held by the vending machine.

We must still consider all 48 possible transitions from **STATE**$<N>$. Looking back at our abstract next-state table, notice that we had only eight types of input events (not counting "no input"). If we strictly prioritize these eight possible events, we can safely ignore combinations. Recall that we adopted a similar strategy for several earlier designs, including the ice cream dispenser in Notes Set 2.2 and the keyless entry system developed in Notes Set 3.1.3.

We choose to prioritize purchases over new coin insertions, and to prioritize item 3 over item 2 over item 1. These prioritizations are strict in the sense that if the user presses B_3, both other buttons are ignored, and any coin inserted is rejected, regardless of whether or not the user can actually purchase item 3 (the machine may not contain enough money to cover the item price). With the choice of strict prioritization, all transitions from all states become well-defined. We apply the transition rules in order of decreasing priority, with conditions, and with don't-cares for lower-priority inputs. For example, for any of the 16 **STATE50**'s (remember that the four current output bits do not affect transitions), the table below lists all possible transitions assuming that $P_3 = 60$, $P_2 = 10$, and $P_1 = 35$.

					next state				
initial state	B_3	B_2	B_1	$C_2C_1C_0$	state	A	R_3	R_2	R_1
STATE50	1	x	x	xxx	**STATE50**	0	0	0	0
STATE50	0	1	x	xxx	**STATE40**	0	0	1	0
STATE50	0	0	1	xxx	**STATE15**	0	0	0	1
STATE50	0	0	0	010	**STATE51**	1	0	0	0
STATE50	0	0	0	000	**STATE52**	1	0	0	0
STATE50	0	0	0	011	**STATE55**	1	0	0	0
STATE50	0	0	0	001	**STATE60**	1	0	0	0
STATE50	0	0	0	111	**STATE50**	0	0	0	0
STATE50	0	0	0	110	**STATE50**	0	0	0	0

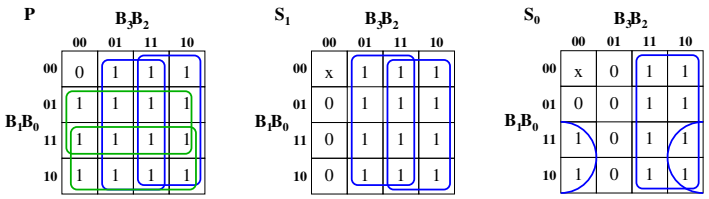
Next, we need to choose a state representation. But this task is essentially done: each output bit (A, R_1, R_2, and R_3) is represented with one bit in the internal representation, and the remaining six bits record the number of nickels held by the vending machine using an unsigned representation.

The choice of a numeric representation for the money held is important, as it allows us to use an adder to compute the money held in the next state.

3.5.2 Encoders and Decoders

Since we chose to prioritize purchases, let's begin by building logic to perform state transitions for purchases. Our first task is to implement prioritization among the three selection buttons. For this purpose, we construct a 4-input **priority encoder**, which generates a signal P whenever any of its four input lines is active and encodes the index of the highest active input as a two-bit unsigned number S. A truth table for our priority encoder appears on the left below, with K-maps for each of the output bits on the right.

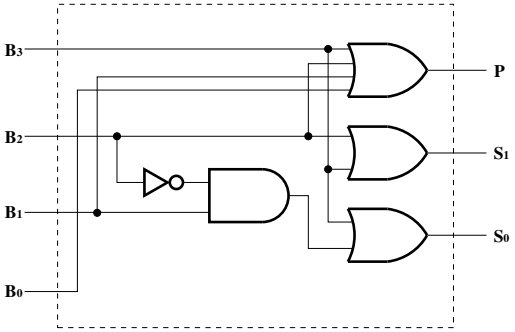
B_3	B_2	B_1	B_0	P	S
1	x	x	x	1	11
0	1	x	x	1	10
0	0	1	x	1	01
0	0	0	1	1	00
0	0	0	0	0	xx



From the K-maps, we extract the following equations:

$$\begin{aligned} P &= B_3 + B_2 + B_1 + B_0 \\ S_1 &= B_3 + B_2 \\ S_0 &= B_3 + \overline{B_2} B_1 \end{aligned}$$

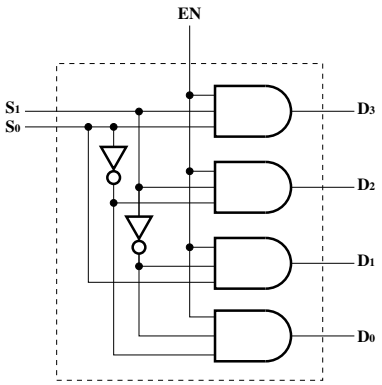
which allow us to implement our encoder as shown to the right.



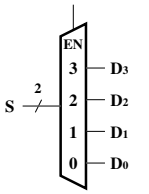
a 4–input priority encoder
(logic diagram)

If we connect our buttons B_1, B_2, and B_3 to the priority encoder (and feed 0 into the fourth input), it produces a signal P indicating that the user is trying to make a purchase and a two-bit signal S indicating which item the user wants.

We also need to build logic to control the item release outputs R_1, R_2, and R_3. An item should be released only when it has been selected (as indicated by the priority encoder signal S) and the vending machine has enough money. For now, let's leave aside calculation of the item release signal, which we call R, and focus on how we can produce the correct values of R_1, R_2, and R_3 from S and R.



a 2–to–4 decoder with enable
(logic diagram)



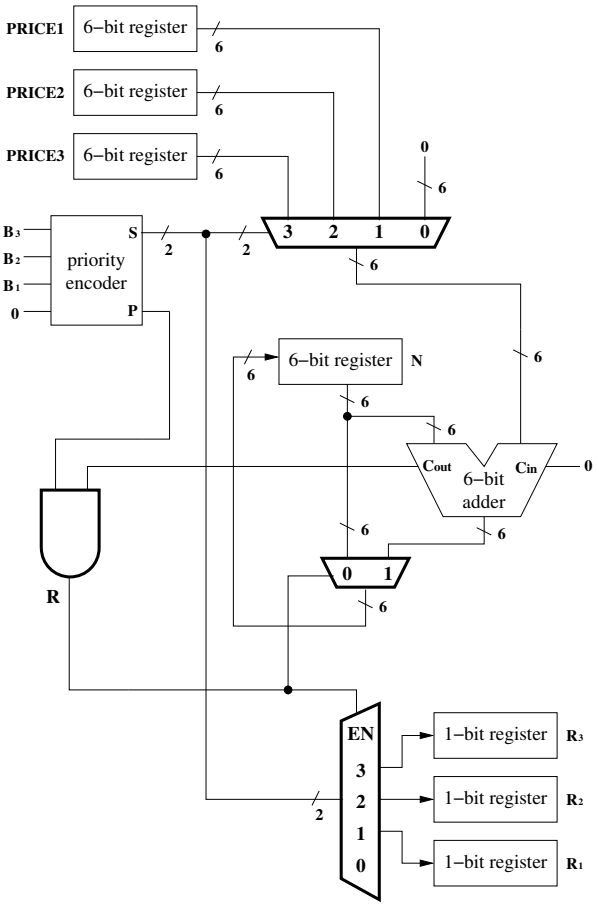
a 2–to–4 decoder
with enable
(symbolic form)

The component to the right is a **decoder** with an enable input. A decoder takes an input signal—typically one coded as a binary number—and produces one output for each possible value of the signal. You may notice the similarity with the structure of a mux: when the decoder is enabled ($EN = 1$), each of the AND gates produces one minterm of the input signal S. In the mux, each of the inputs is then included in one minterm's AND gate, and the outputs of all AND gates are ORd together. In the decoder, the AND gate outputs are the outputs of the decoder. Thus, when enabled, the decoder produces exactly one 1 bit on its outputs. When not enabled ($EN = 0$), the decoder produces all 0 bits.

We use a decoder to generate the release signals for the vending machine by connecting the signal S produced by the priority encoder to the decoder's S input and connecting the item release signal R to the decoder's EN input. The outputs D_1, D_2, and D_3 then correspond to the individual item release signals R_1, R_2, and R_3 for our vending machine.

3.5.3 Vending Machine Implementation

We are now ready to implement the FSM to handle purchases, as shown to the right. The current number of nickels, N, is stored in a register in the center of the diagram. Each cycle, N is fed into a 6-bit adder, which subtracts the price of any purchase requested in that cycle. Recall that we chose to record item prices in registers. We avoid the need to negate prices before adding them by storing the negated prices in our registers. Thus, the value of register PRICE1 is $-P_1$, the the value of register PRICE2 is $-P_2$, and the the value of register PRICE3 is $-P_3$. The priority encoder's S signal is then used to select the value of one of these three registers (using a 24-to-6 mux) as the second input to the adder.



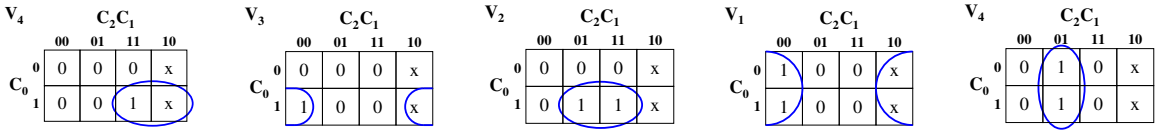
We use the adder to execute a subtraction, so the carry out C_{out} is 1 whenever the value of N is at least as great as the amount being subtracted. In that case, the purchase is successful. The AND gate on the left calculates the signal R indicating a successful purchase, which is then used to select the next value of N using the 12-to-6 mux below the adder. When no item selection buttons are pushed, P and thus R are both 0, and the mux below the adder keeps N unchanged in the next cycle. Similarly, if $P = 1$ but N is insufficient, C_{out} and thus R are both 0, and again N does not change. Only when $P = 1$ and $C_{out} = 1$ is the purchase successful, in which case the price is subtracted from N in the next cycle.

The signal R is also used to enable a decoder that generates the three individual item release outputs. The correct output is generated based on the decoded S signal from the priority encoder, and all three output bits are latched into registers to release the purchased item in the next cycle.

One minor note on the design so far: by hardwiring C_{in} to 0, we created a problem for items that cost nothing (0 nickels): in that case, C_{out} is always 0. We could instead store $-P_1 - 1$ in PRICE1 (and so forth) and feed P in to C_{in}, but maybe it's better not to allow free items.

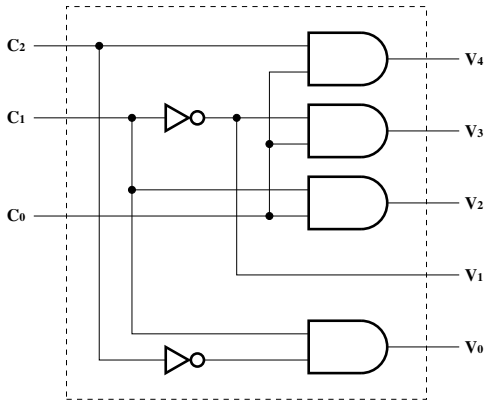
How can we support coin insertion? Let's use the same adder to add each inserted coin's value to N. The table at the right shows the value of each coin as a 5-bit unsigned number of nickels. Using this table, we can fill in K-maps for each bit of V, as shown below. Notice that we have marked the two undefined bit patterns for the coin type C as don't cares in the K-maps.

coin type	$C_2C_1C_0$	$V_4V_3V_2V_1V_0$
none	110	00000
nickel	010	00001
dime	000	00010
quarter	011	00101
half dollar	001	01010
dollar	111	10100



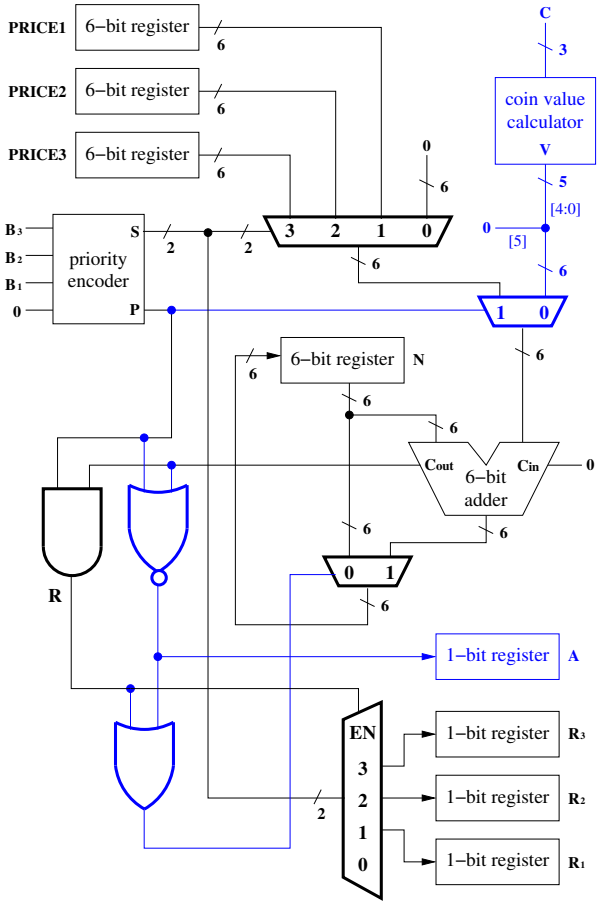
Solving the K-maps gives the following equations, which we implement as shown to the right.

$$\begin{aligned} V_4 &= C_2C_0 \\ V_3 &= \overline{C_1}C_0 \\ V_2 &= C_1C_0 \\ V_1 &= \overline{C_1} \\ V_0 &= \overline{C_2}C_1 \end{aligned}$$



coin value calculator

Now we can extend the design to handle coin insertion, as shown to the right with new elements highlighted in blue. The output of the coin value calculator is extended with a leading 0 and then fed into a 12-to-6 mux. When a purchase is requested, $P = 1$ and the mux forwards the item price to the adder—recall that we chose to give purchases priority over coin insertion. When no purchase is requested, the value of any coin inserted (or 0 when no coin is inserted) is passed to the adder.



Two new gates have been added on the lower left. First, let's verify that purchases work as before. When a purchase is requested, $P = 1$, so the NOR gate outputs 0, and the OR gate simple forwards R to control the mux that decides whether the purchase was successful, just as in our original design.

When no purchase is made ($P = 0$, and $R = 0$), the adder adds the value of any inserted coin to N. If the addition overflows, $C_{out} = 1$, and the output of the NOR gate is 0. Note that the NOR gate output is stored as the output A in the next cycle, so a coin that causes overflow in the amount of money stored is rejected. The OR gate also outputs 0, and N remains unchanged. If the addition does not overflow, the NOR gate outputs a 1, the coin is accepted ($A = 1$ in the next cycle), and the mux allows the sum $N+V$ to be written back as the new value of N.

The tables at the top of the next page define all of the state variables, inputs, outputs, and internal signals used in the design, and list the number of bits for each variable.

		FSM state
PRICE1	6	negated price of item 1 ($-P_1$)
PRICE2	6	negated price of item 2 ($-P_2$)
PRICE3	6	negated price of item 3 ($-P_3$)
N	6	value of money in machine (in nickels)
A	1	stored value of accept coin output
R_1	1	stored value of release item 1 output
R_2	1	stored value of release item 2 output
R_3	1	stored value of release item 3 output
		internal signals
V	5	inserted coin value in nickels
P	1	purchase requested (from priority encoder)
S	2	item # requested (from priority encoder)
R	1	release item (purchase approved)

		inputs
B_1	1	item 1 selected for purchase
B_2	1	item 2 selected for purchase
B_3	1	item 3 selected for purchase
C	3	coin inserted (see earlier table for meaning)
		outputs
A	1	accept inserted coin (last cycle)
R_1	1	release item 1
R_2	1	release item 2
R_3	1	release item 3
Note that outputs correspond one-to-one with four bits of FSM state.		

3.5.4 Design of a Game Controller

For the next example, imagine that you are part of a team building a game for children to play at Engineering Open House. The game revolves around an old logic problem in which a farmer must cross a river in order to reach the market. The farmer is traveling to the market to sell a fox, a goose, and some corn. The farmer has a boat, but the boat is only large enough to carry the fox, the goose, or the corn along with the farmer. The farmer knows that if he leaves the fox alone with the goose, the fox will eat the goose. Similarly, if the farmer leaves the goose alone with the corn, the goose will eat the corn. How can the farmer cross the river?

Your team decides to build a board illustrating the problem with a river from top to bottom and lights illustrating the positions of the farmer (always with the boat), the fox, the goose, and the corn on either the left bank or the right bank of the river. Everything starts on the left bank, and the children can play the game until they win by getting everything to the right bank or until they make a mistake. As the ECE major on your team, you get to design the FSM!

Since the four entities (farmer, fox, goose, and corn) can be only on one bank or the other, we can use one bit to represent the location of each entity. Rather than giving the states names, let's just call a state $FXGC$. The value of F represents the location of the farmer, either on the left bank ($F = 0$) or the right bank ($F = 1$). Using the same representation (0 for the left bank, 1 for the right bank), the value of X represents the location of the fox, G represents the location of the goose, and C represents the location of the corn.

We can now put together an abstract next-state table, as shown to the right. Once the player wins or loses, let's have the game indicate their final status and stop accepting requests to have the farmer cross the river. We can use a reset button to force the game back into the original state for the next player.

initial state	input event	condition	final state
$FXGC$	no input	always	$FXGC$
$FXGC$	reset	always	0000
$FXGC$	cross alone	always	$\bar{F}XGC$
$FXGC$	cross with fox	$F = X$	$\bar{F}\bar{X}GC$
$FXGC$	cross with goose	$F = G$	$\bar{F}X\bar{G}C$
$FXGC$	cross with corn	$F = C$	$\bar{F}XG\bar{C}$

Note that we have included conditions for some of the input events, as we did previously with the vending machine design. The conditions here require that the farmer be on the same bank as any entity that the player wants the farmer to carry across the river.

Next, we specify the I/O interface. For input, the game has five buttons. A reset button R forces the FSM back into the initial state. The other four buttons cause the farmer to cross the river: B_F crosses alone, B_X with the fox, B_G with the goose, and B_C with the corn.

For output, we need position indicators for the four entities, but let's assume that we can simply output the current state $FXGC$ and have appropriate images or lights appear on the correct banks of the river. We also need two more indicators: W for reaching the winning state, and L for reaching a losing state.

Now we are ready to complete the specification. We could use a strict prioritization of input events, as we did with earlier examples. Instead, in order to vary the designs a bit, we use a strict prioritization among allowed inputs. The reset button R has the highest priority, followed by B_F, B_C, B_G, and finally B_X. However, only those buttons that result in an allowed move are considered when selecting one button among several pressed in a single clock cycle.

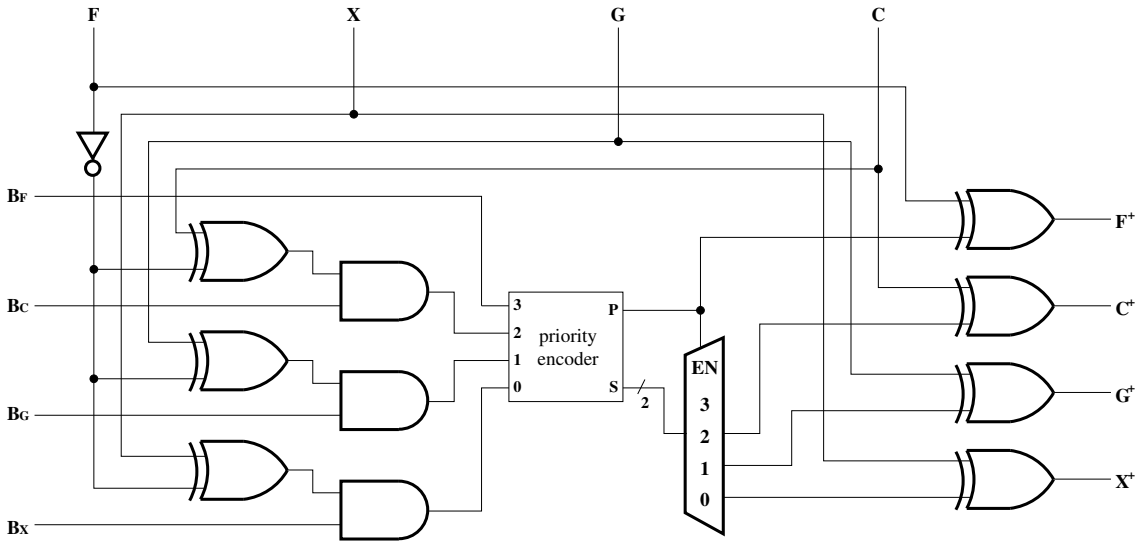
As an example, consider the state $FXGC = 0101$. The farmer is not on the same bank as the fox, nor as the corn, so the B_X and B_C buttons are ignored, leading to the next-state table to the right. Notice that B_G is accepted even if B_C is pressed because the farmer is not on the same bank as the corn. As shown later, this approach to prioritization of inputs is equally simple in terms of implementation.

$FXGC$	R	B_F	B_X	B_G	B_C	$F^+X^+G^+C^+$
0101	1	x	x	x	x	0000
0101	0	1	x	x	x	1101
0101	0	0	x	1	x	1111
0101	0	0	x	0	x	0101

Recall that we want to stop the game when the player wins or loses. In these states, only the reset button is accepted. For example, the state $FXGC = 0110$ is a losing state because the farmer has left the fox with the goose on the opposite side of the river. In this case, the player can reset the game, but other buttons are ignored.

$FXGC$	R	B_F	B_X	B_G	B_C	$F^+X^+G^+C^+$
0110	1	x	x	x	x	0000
0110	0	x	x	x	x	0110

As we have already chosen a representation, we now move on to implement the FSM. Let's begin by calculating the next state ignoring the reset button and the winning and losing states, as shown in the logic diagram below.



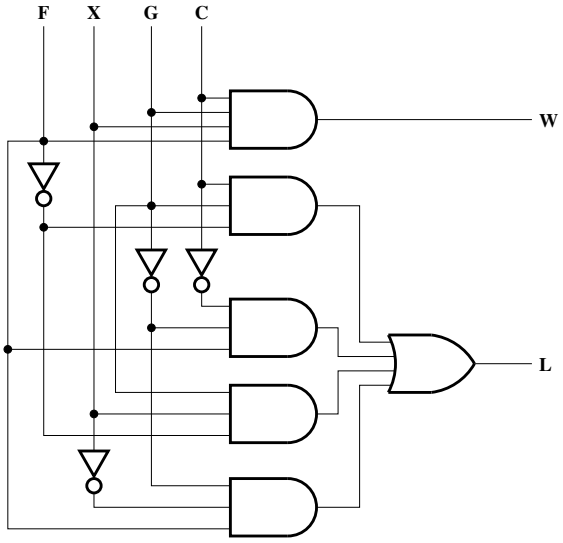
The left column of XOR gates determines whether the farmer is on the same bank as the corn (top gate), goose (middle gate), and fox (bottom gate). The output of each gate is then used to mask out the corresponding button: only when the farmer is on the same bank are these buttons considered. The adjusted button values are then fed into a priority encoder, which selects the highest priority input event according to the scheme that we outlined earlier (from highest to lowest, B_F, B_C, B_G, and B_X, ignoring the reset button).

The output of the priority encoder is then used to drive another column of XOR gates on the right in order to calculate the next state. If any of the allowed buttons is pressed, the priority encoder outputs $P = 1$, and the farmer's bank is changed. If B_C is allowed and selected by the priority encoder (only when B_F is not pressed), both the farmer and the corn's banks are flipped. The goose and the fox are handled in the same way.

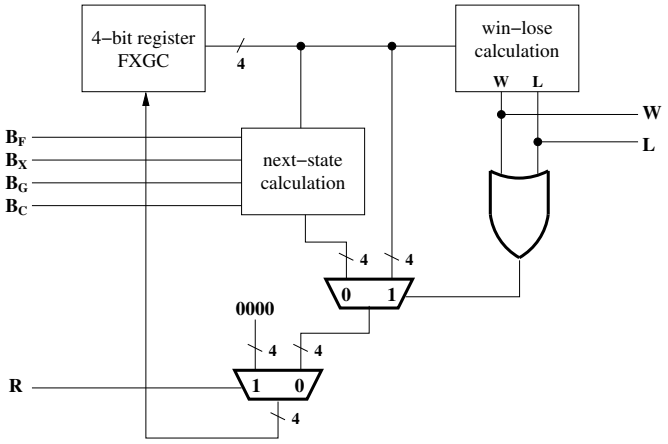
Next, let's build a component to produce the win and lose signals. The one winning state is $FXGC = 1111$, so we simply need an AND gate. For the lose signal L, we can fill in a K-map and derive an expression, as shown to the right, then implement as shown in the logic diagram to the far right. For the K-map, remember that the player loses whenever the fox and the goose are on the same side of the river, but opposite from the farmer, or whenever the goose and the corn are on the same side of the river, but opposite from the farmer.

L	FX 00	01	11	10
GC 00	0	0	1	1
01	0	0	0	1
11	1	1	0	0
10	0	1	0	0

$$L = F\bar{X}\bar{G} + \bar{F}XG + F\bar{G}\bar{C} + \bar{F}GC$$



Finally, we complete our design by integrating the next-state calculation and the win-lose calculation with a couple of muxes, as shown to the right. The lower mux controls the final value of the next state: note that it selects between the constant state $FXGC = 0000$ when the reset button R is pressed and the output of the upper mux when $R = 0$. The upper mux is controlled by $W + L$, and retains the current state whenever either signal is 1. In other words, once the player has won or lost, the upper mux prevents further state changes until the reset button is pressed. When R, W, and L are all 0, the next state is calculated according to whatever buttons have been pressed.



3.5.5 Analysis of a Stoplight Controller

In this example, we begin with a digital FSM design and analyze it to understand how it works and to verify that its behavior is appropriate. The FSM that we analyze has been designed to control the stoplights at the intersection of two roads. For naming purposes, we assume that one of the roads runs East and West (EW), and the second runs North and South (NS).

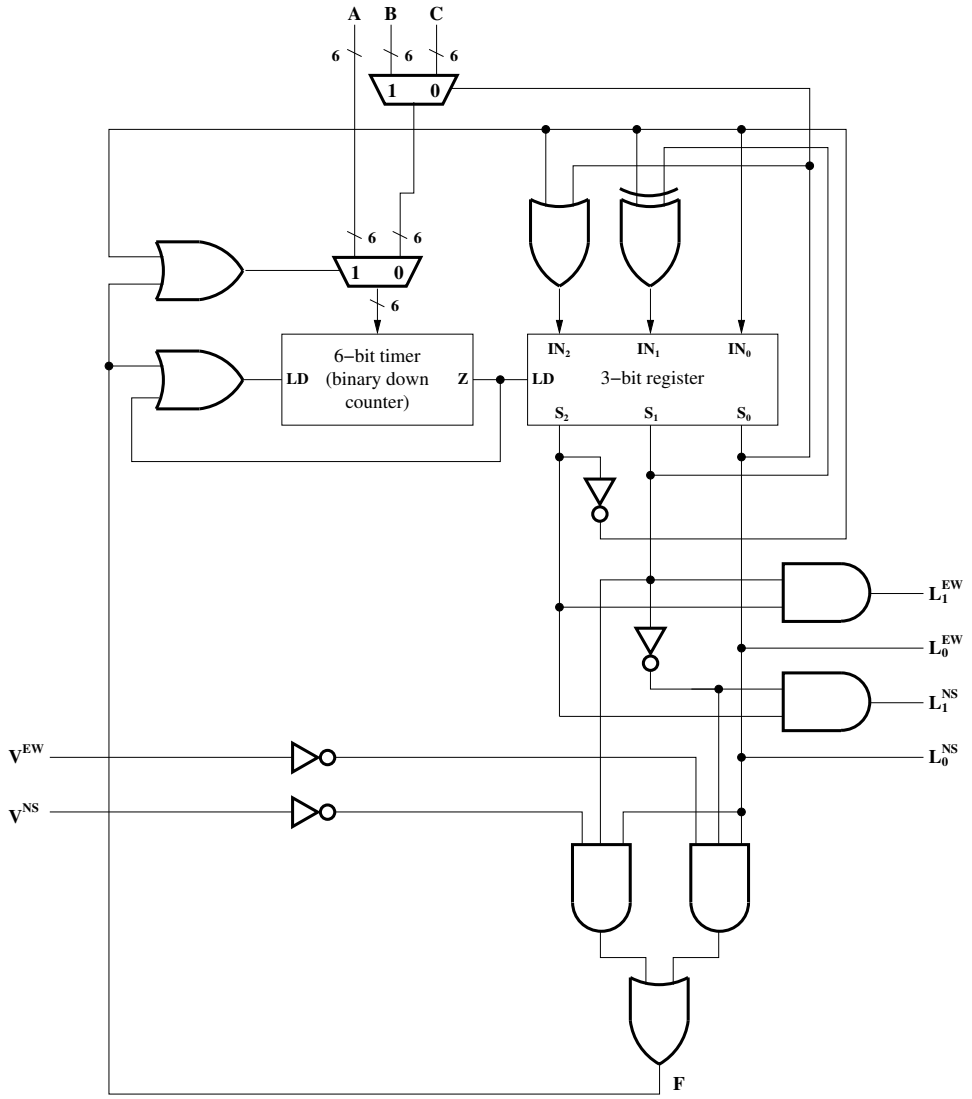
The stoplight controller has two inputs, each of which senses vehicles approaching from either direction on one of the two roads. The input $V^{EW} = 1$ when a vehicle approaches from either the East or the West, and the input $V^{NS} = 1$ when a vehicle approaches from either the North or the South. These inputs are also active when vehicles are stopped waiting at the corresponding lights. Another three inputs, A, B, and C, control the timing behavior of the system; we do not discuss them here except as variables.

The outputs of the controller consist of two 2-bit values, L^{EW} and L^{NS}, that specify the light colors for the two roads. In particular, L^{EW} controls the lights facing East and West, and L^{NS} controls the lights facing North and South. The meaning of these outputs is given in the table to the right.

L	light color
0x	red
10	yellow
11	green

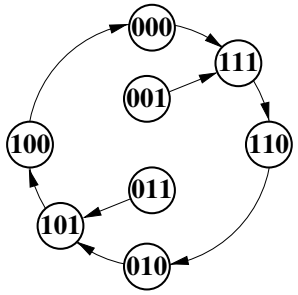
Let's think about the basic operation of the controller. For safety reasons, the controller must ensure that the lights on one or both roads are red at all times. Similarly, if a road has a green light, the controller should show a yellow light before showing a red light to give drivers some warning and allow them to slow down. Finally, for fairness, the controller should alternate green lights between the two roads.

Now take a look at the logic diagram below. The state of the FSM has been split into two pieces: a 3-bit register S and a 6-bit timer. The timer is simply a binary counter that counts downward and produces an output of $Z = 1$ when it reaches 0. Notice that the register S only takes a new value when the timer reaches 0, and that the Z signal from the timer also forces a new value to be loaded into the timer in the next cycle. We can thus think of transitions in the FSM on a cycle by cycle basis as consisting of two types. The first type simply counts downward for a number of cycles while holding the register S constant, while the second changes the value of S and sets the timer in order to maintain the new value of S for some number of cycles.



Let's look at the next-state logic for S, which feeds into the IN inputs on the 3-bit register ($S_2^+ = IN_2$ and so forth). Notice that none of the inputs to the FSM directly affect these values. The states of S thus act like a counter. By examining the connections, we can derive equations for the next state and draw a transition diagram, as shown to the right. As the figure shows, there are six states in the loop defined by the next-state logic, with the two remaining states converging into the loop after a single cycle.

$$\begin{aligned} S_2^+ &= \overline{S_2} + S_0 \\ S_1^+ &= \overline{S_2} \oplus S_1 \\ S_0^+ &= \overline{S_2} \end{aligned}$$



Let's now examine the outputs for each state in order to understand how the stoplight sequencing works. We derive equations for the outputs that control the lights, as shown to the right, then calculate values and colors for each state, as shown to the far right. For completeness, the table includes the states outside of the desired loop. The lights are all red in both of these states, which is necessary for safety.

$$\begin{aligned} L_1^{EW} &= S_2 S_1 \\ L_0^{EW} &= S_0 \\ L_1^{NS} &= S_2 \overline{S_1} \\ L_0^{NS} &= S_0 \end{aligned}$$

S	L^{EW}	L^{NS}	EW light color	NS light color
000	00	00	red	red
111	11	01	green	red
110	10	00	yellow	red
010	00	00	red	red
101	01	11	red	green
100	00	10	red	yellow
001	01	01	red	red
011	01	01	red	red

Now let's think about how the timer works. As we already noted, the timer value is set whenever S enters a new state, but it can also be set under other conditions—in particular, by the signal F calculated at the bottom of the FSM logic diagram.

For now, assume that $F = 0$. In this case, the timer is set only when the state S changes, and we can find the duration of each state by analyzing the muxes. The bottom mux selects A when $S_2 = 0$, and selects the output of the top mux when $S_2 = 1$. The top mux selects B when $S_0 = 1$, and selects C when $S_0 = 0$. Combining these results, we can calculate the duration of the next states of S when $F = 0$, as shown in the table to the right. We can then combine the next state duration with our previous calculation of the state sequencing (also the order in the table) to obtain the durations of each state, also shown in the rightmost column of the table.

S	EW light color	NS light color	next state duration	current state duration
000	red	red	A	C
111	green	red	B	A
110	yellow	red	C	B
010	red	red	A	C
101	red	green	B	A
100	red	yellow	C	B
001	red	red	A	—
011	red	red	A	—

What does F do? Analyzing the gates that produce it gives $F = S_1 S_0 \overline{V^{NS}} + \overline{S_1} S_0 \overline{V^{EW}}$. If we ignore the two states outside of the main loop for S, the first term is 1 only when the lights are green on the East and West roads and the detector for the North and South roads indicates that no vehicles are approaching. Similarly, the second term is 1 only when the lights are green on the North and South roads and the detector for the East and West roads indicates that no vehicles are approaching.

What happens when $F = 1$? First, the OR gate feeding into the timer's LD input produces a 1, meaning that the timer loads a new value instead of counting down. Second, the OR gate controlling the lower mux selects the A input. In other words, the timer is reset to A cycles, corresponding to the initial value for the green light states. In other words, the light stays green until vehicles approach on the other road, plus A more cycles.

Unfortunately, the signal F may also be 1 in the unused states of S, in which case the lights on both roads may remain red even though cars are waiting on one of the roads. To avoid this behavior, we must be sure to initialize the state S to one of the six states in the desired loop.