

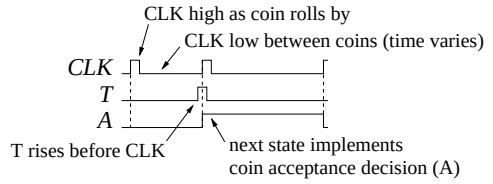
ECE199JL: Introduction to Computer Engineering **Fall 2012**
Notes Set 3.3

Design of the Finite State Machine for the Lab

This set of notes explains the process that Prof. Jones used to develop the FSM for the lab. The lab simulates a vending machine mechanism for automatically identifying coins (dimes and quarters only), tracking the amount of money entered by the user, accepting or rejecting coins, and emitting a signal when a total of 35 cents has been accepted. In the lab, we will only drive a light with the "paid in full" signal. Sorry, no candy nor Dew will be distributed!

Physical Design, Sensors, and Timing

The physical elements of the lab were designed by Prof. Chris Schmitz and constructed with some help from the ECE shop. A user inserts a coin into a slot at one end of the device. The coin then rolls down a slope towards a gate controlled by a servo. The gate can be raised or lowered, and determines whether the coin exits from the other side or the bottom of the device. As the coin rolls, it passes two optical sensors.[1] One of these sensors is positioned high enough above the slope that a dime passes beneath the sesnor, allowing the signal T produced by the sensor to tell us whether the coin is a dime or a quarter. The second sensor is positioned so that all coins pass in front of it. The sensor positions are chosen carefully to ensure that, in the case of a quarter, the coin is still blocking the first sensor when it reaches the second sensor. Blocked sensors give a signal of 1 in this design, so the rising edge the signal from the second sensor can be used as a "clock" for our FSM. When the rising edge occurs, the signal T from the first sensor indicates whether the coin is a quarter ($T = 1$) or a dime ($T = 0$).



A sample timing diagram for the lab appears to the right. The clock signal generated by the lab is not only not a square wave—in other words, the high and low portions are not equal—but is also unlikely to be periodic. Instead, the "cycle" is defined by the time between coin insertions. The T signal serves as the single input to our FSM. In the timing diagram, T is shown as rising and falling before the clock edge. We use positive edge-triggered flip-flops to implement our FSM, thus the aspect of the relative timing that matters to our design is that, when the clock rises, the value of T is stable and indicates the type of coin entered. The signal T may fall before or after the clock does—the two are equivalent for our FSM's needs.

The signal A in the timing diagram is an output from the FSM, and indicates whether or not the coin should be accepted. This signal controls the servo that drives the gate, and thus determines whether the coin is accepted ($A = 1$) as payment or rejected ($A = 0$) and returned to the user.

Looking at the timing diagram, you should note that our FSM makes a decision based on its current state and the input T and enters a new state at the rising clock edge. The value of A in the next cycle thus determines the position of the gate when the coin eventually rolls to the end of the slope. As we said earlier, our FSM is thus a Moore machine: the output A does not depend on the input T, but only on the current internal state bits of the the FSM. However, you should also now realize that making A depend on T is not adequate for this lab. If A were to rise with T and fall with the rising clock edge (on entry to the next state), or even fall with the falling edge of T, the gate would return to the reject position by the time the coin reached the gate, regardless of our FSM's decision!

[1] The full system actually allows four sensors to differentiate four types of coins, but our lab uses only two of these sensors.

An Abstract Model

We start by writing down states for a user's expected behavior. Given the fairly tight constraints that we have placed on our lab, few combinations are possible. For a total of 35 cents, a user should either insert a dime followed by a quarter, or a quarter followed by a dime. We begin in a START state, which transitions to states DIME or QUARTER when the user inserts the first coin. With no previous coin, we need not specify a value for A. No money has been deposited, so we set output $P = 0$ in the START state. We next create DIME and QUARTER states corresponding to the user having entered one coin. The first coin should be accepted, but more money is needed, so both of these states output $A = 1$ and $P = 0$. When a coin of the opposite type is entered, each state moves to a state called PAID, which we use for the case in which a total of 35 cents has been received. For now, we ignore the possibility that the same type of coin is deposited more than once. Finally, the PAID state accepts the second coin ($A = 1$) and indicates that the user has paid the full price of 35 cents ($P = 1$).

state	dime ($T = 0$)	quarter ($T = 1$)	accept? (A)	paid? (P)
START	DIME	QUARTER		no
DIME		PAID	yes	no
QUARTER	PAID		yes	no
PAID			yes	yes

We next extend our design to handle user mistakes. If a user enters a second dime in the DIME state, our FSM should reject the coin. We create a REJECTD state and add it as the next state from DIME when a dime is entered. The REJECTD state rejects the dime ($A = 0$) and continues to wait for a quarter ($P = 0$). What should we use as next states from REJECTD? If the user enters a third dime (or a fourth, or a fifth, and so on), we want to reject the new dime as well. If the user enters a quarter, we want to accept the coin, at which point we have received 35 cents (counting the first dime). We use this reasoning to complete the description of REJECTD. We also create an analogous state, REJECTQ, to handle a user who inserts more than one quarter.

state	dime ($T = 0$)	quarter ($T = 1$)	accept? (A)	paid? (P)
START	DIME	QUARTER		no
DIME	REJECTD	PAID	yes	no
REJECTD	REJECTD	PAID	no	no
QUARTER	PAID	REJECTQ	yes	no
REJECTQ	PAID	REJECTQ	no	no
PAID			yes	yes

What should happen after a user has paid 35 cents and bought one item? The FSM at that point is in the PAID state, which delivers the item by setting $P = 1$. Given that we want the FSM to allow the user to purchase another item, how should we choose the next states from PAID? The behavior that we want from PAID is identical to the behavior that we defined from START. The 35 cents already deposited was used to pay for the item delivered, so the machine is no longer holding any of the user's money. We can thus simply set the next states from PAID to be DIME when a dime is inserted and QUARTER when a quarter is inserted.

At this point, we make a decision intended primarily to simplify the logic needed to build the lab. Without a physical item delivery mechanism with a specification for how its input must be driven, the behavior of the output signal P can be fairly flexible. For example, we could build a delivery mechanism that used the rising edge of P to open a chute. In this case, the output $P = 0$ in the start state is not relevant, and we can merge the state START with the state PAID. The way that we handle P in the lab, we might find it strange to have a "paid" light turn on before inserting any money, but keeping the design simple enough for a first lab exercise is more important. Our final abstract state table appears above.

state	dime ($T = 0$)	quarter ($T = 1$)	accept? (A)	paid? (P)
PAID	DIME	QUARTER	yes	yes
DIME	REJECTD	PAID	yes	no
REJECTD	REJECTD	PAID	no	no
QUARTER	PAID	REJECTQ	yes	no
REJECTQ	PAID	REJECTQ	no	no

Picking the Representation

We are now ready to choose the state representation for the lab FSM. With five states, we need three bits of internal state. Prof. Jones decided to leverage human meaning in assigning the bit patterns, as follows:

S_2	type of last coin inserted (0 for dime, 1 for quarter)
S_1	more than one quarter inserted? (1 for yes, 0 for no)
S_0	more than one dime inserted? (1 for yes, 0 for no)

These meanings are not easy to apply to all of our states. For example, in the PAID state, the last coin inserted may have been of either type, or of no type at all, since we decided to start our FSM in that state as well. However, for the other four states, the meanings provide a clear and unique set of bit pattern assignments, as shown to the right. We can choose any of the remaining four bit patterns (010, 011, 101, or 111) for the PAID state. In fact, *we can choose all of the remaining patterns* for the PAID state. We can always represent any state with more than one pattern if we have spare patterns available. Prof. Jones used this freedom to simplify the logic design.

state	$S_2S_1S_0$
PAID	???
DIME	000
REJECTD	001
QUARTER	100
REJECTQ	110

This particular example is slightly tricky. The four free patterns do not share any single bit in common, so we cannot simply insert x's into all K-map entries for which the next state is PAID. For example, if we insert an x into the K-map for S_2^+, and then choose a function for S_2^+ that produces a value of 1 in place of the don't care, we must also produce a 1 in the corresponding entry of the K-map for S_0^+. Our options for PAID include 101 and 111, but not 100 nor 110. These latter two states have other meanings.

Let's begin by writing a next-state table consisting mostly of bits, as shown to the right. We use this table to write out a K-map for S_2^+ as follows: any of the patterns that may be used for the PAID state obey the next-state rules for PAID. Any next-state marked as PAID is marked as don't care in the K-map, since we can choose patterns starting with either or both values to represent our PAID state. The resulting K-map appears to the far right. As shown, we simply set $S_2^+ = T$, which matches our original "meaning" for S_2. That is, S_2 is the type of the last coin inserted.

state	$S_2S_1S_0$	$S_2^+S_1^+S_0^+$ $T=0$	$T=1$
PAID	PAID	000	100
DIME	000	001	PAID
REJECTD	001	001	PAID
QUARTER	100	PAID	110
REJECTQ	110	PAID	110

S_2^+

S_0T \ S_2S_1	00	01	11	10
00	0	0	x	x
01	x	1	1	1
11	x	1	1	1
10	0	0	0	0

Based on our choice for S_2^+, we can rewrite the K-map as shown to the right, with green italics and shading marking the values produced for the x's in the specification. Each of these boxes corresponds to one transition into the PAID state. By specifying the S_2 value, we cut the number of possible choices from four to two in each case. For those combinations in which the implementation produces $S_2^+ = 0$, we must choose $S_1^+ = 1$, but are still free to leave S_0^+ marked as a don't care. Similarly, for those combinations in which the implementation produces $S_2^+ = 1$, we must choose $S_0^+ = 1$, but are still free to leave S_1^+ marked as a don't care.

S_2^+

S_0T \ S_2S_1	00	01	11	10
00	0	0	*0*	*0*
01	*1*	1	1	1
11	*1*	1	1	1
10	0	0	0	0

The K-maps for S_1^+ and S_0^+ are shown to the right. We have not given algebraic expressions for either, but have indicated our choices by highlighting the resulting replacements of don't care entries with the values produced by our expressions. At this point, we can review the state patterns actually produced by each of the four next-state transitions into the PAID state. From the DIME state, we move into the 101 state when the user inserts a quarter. The result is the same from the REJECTD state. From the QUARTER state, however, we move into the 010 state when the user inserts a dime. The result is the same from the REJECTQ state. We must thus classify both patterns, 101 and 010, as PAID states. The remaining two patterns, 011 and 111, cannot

S_1^+

S_0T \ S_2S_1	00	01	11	10
00	0	0	1	1
01	*0*	0	1	1
11	*0*	0	0	0
10	0	0	0	0

S_0^+

S_0T \ S_2S_1	00	01	11	10
00	1	0	*0*	*0*
01	1	0	0	0
11	1	0	0	0
10	1	0	0	0

be reached from any of the states in our design. We might then try to leverage the fact that the next-state patterns from these two states are not relevant (recall that we fixed the next-state patterns for all four of the possible PAID states) to further simplify our logic, but doing so does not provide any advantage (you may want to check our claim).

		$S_2^+S_1^+S_0^+$			
state	$S_2S_1S_0$	$T=0$	$T=1$	A	P
PAID1	010	000	100	1	1
PAID2	101	000	100	1	1
DIME	000	001	101	1	0
REJECTD	001	001	101	0	0
QUARTER	100	010	110	1	0
REJECTQ	110	010	110	0	0
EXTRA1	011	000	100	x	x
EXTRA2	111	000	100	x	x

The final state table is shown to the right. We have included the extra states at the bottom of the table. We have specified the next-state logic for these states, but left the output bits as don't cares. A state transition diagram appears at the bottom of this page.

Testing the Design

Having a complete design on paper is a good step forward, but humans make mistakes at all stages. How can we know that a circuit that we build in the lab correctly implements the FSM that we have outlined in these notes?

For the lab design, we have two problems to solve. First, we have not specified an initialization scheme for the FSM. We may want the FSM to start in one of the PAID states, but adding initialization logic to the design may mean requiring you to wire together significantly more chips. Second, we need a sequence of inputs that manages to test that all of the next-state and output logic implementations are correct.

Testing sequential logic, including FSMs, is in general extremely difficult. In fact, large sequential systems today are generally converted into combinational logic by using shift registers to fill the flip-flops with a particular pattern, executing the logic for one clock cycle, and checking that the resulting pattern of bits in the flip-flops is correct. This approach is called **scan-based testing**, and is discussed in ECE 543. You will make use of a similar approach when you test your combinational logic in the second week of the lab, before wiring up the flip-flops.

We have designed our FSM to be easy to test (even small FSMs may be challenging) with a brute force approach. In particular, we identify two input sequences that together serve both to initialize and to test a correctly implemented variant of our FSM. Our initialization sequence forces the FSM into a specific state regardless of its initial state. And our test sequence crosses every transition arc leaving the six valid states.

In terms of T, the coin type, we initialize the FSM with the input sequence 001. Notice that such a sequence takes any initial state into PAID2.

For testing, we use the input sequence 111010010001. You should trace this sequence, starting from PAID2, on the diagram below to see how the test sequence covers all of the possible arcs. As we test, we need also to observe the A and P outputs in each state to check the output logic.