6

Practicing Design

And now, let's do some design. In this chapter we present several detailed examples of digital design, from the small, yet important, to the substantial project. The goals are two: to illustrate good design methodology and to introduce important design problems and their treatment. There is repetition of some aspects of design, particularly the use of ASM charts, yet each example has fresh material. We will indulge in some excursions into interesting topics as they occur in the context of design. We hope that in this way the concepts will be more meaningful to you than they would be as separate, isolated subjects. These examples will be more meaningful if constructed and debugged using a simulator and we urge you to do so, especially if you don't do the laboratory that accompanies this book.

Examples 1 and 2 illustrate some basic design issues related to human interaction with a digital device. In example 3 we develop circuits to support the conversion of information from a serial bit stream to a sequence of bytes and vice versa, common operations in data communications. The traffic-light controller in example 4 provides more practice in design. In examples 5 and 6 we bring together several design problems and techniques.

DESIGN EXAMPLE 1: A SINGLE PULSER

Our first illustration of design is the development of a circuit for handling a common situation involving a human operator of a machine. Most real systems involve humans, usually at switches, pushbuttons, and lights. Digital systems generally (but not always) run at speeds many thousands of times faster than human reactions. When a machine operator presses a button to initiate some action, the digital device must detect this signal and perform the appropriate steps. In the typical case, the machine completes the actions in a flash, and is back interrogating the pushbutton signal again long before the operator can release the button. We must develop a scheme so that the machine processes a particular button depression only once. A circuit for this is called *a single pulser;* it delivers a pulse only a single clock cycle long when a button is pressed. If we have such a circuit, our digital machine may test the single-pulser output instead of dealing directly with the pushbutton signal and may thereby detect only one event as long as the button is down.

We will study this problem and its solution in several ways. First, we develop the most fundamental solution.

Algorithmic Solution of the Single Pulser

Statement of the problem. We have a debounced pushbutton, on (true) in the down position, off (false) in the up position. Devise a circuit to sense the depression of the button and assert an output signal for one clock cycle. The system should not allow additional assertions of the output until after the operator has released the button.

Approach. We will develop an ASM solution to the problem. Since the problem is stated in terms of clock pulses, we know we are dealing with a synchronous system (for which we are grateful). Let us name the important inputs and outputs. Clearly, the position of the pushbutton is of great importance to the algorithm. The pushbutton signal can change at any time, independently of the state of the system clock, so the signal is asynchronous, and our name for it should have a terminal *: for instance, PB*. Call the output of the single-pulser circuit PB.PULSE.

We know to be cautious about allowing asynchronous test inputs to creep into our ASM charts. To avoid testing PB*, we should synchronize the signal using a clocked D flip-flop. This flip-flop, with input PB* and output PB.SYNC, becomes part of the architecture of our solution.

Control algorithm. Now we may write an ASM chart to describe the algorithm. The algorithm will test PB.SYNC and produce an output PB.PULSE. The algorithm has two states: one for detecting the first moment that PB.SYNC becomes true (button goes down), and the other to wait until PB.SYNC becomes false (button goes up). Figure 6–1 is the ASM.

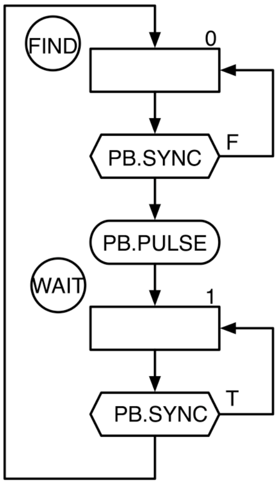


Figure 6-1 A single pulser ASM

Implementing the design. The equation for the output PB.PULSE is

$$\text{PB.PULSE} = \text{FIND} \bullet \text{PB.SYNC}$$

Once we have a way of producing the variable FIND, our problem is solved. To obtain the value of FIND, we must implement a state generator for our ASM. This is one of the few times we will depart from our favored one-hot implementation and use an encoding to point to one of the two states; this is natural for two-state machines since a one-bit pointer, a flip-flop, is all we need as a bi-valued pointer. The pointer value is placed to the upper right of each ASM state in fig. 6-1.

The two-state ASM will require one flip-flop to record the state, and logic to develop the next-state input to the flip-flop. In our present design, the output of one flip-flop has two distinct states, and this is sufficient to produce signals for the two states WAIT and FIND directly, without decoding.

Look at Fig. 6-1. Formally, the flip-flop output is a state variable A, A = 0 representing the FIND state and A = 1 the WAIT state. Thus

$$\text{WAIT} = \text{A}$$

$$\text{FIND} = \overline{\text{A}}$$

So we may just call the flip-flop output WAIT, and then we have $\text{FIND} = \overline{\text{WAIT}}$ and the transition table becomes:

A	PB.SYNC	A(D)
0	0	0
0	1	1
1	0	0
1	1	1

We see that the D input to the state flip-flop is simply PB.SYNC! The resulting trivially simple hardware is shown in Fig. 6-2. Lest you think the hardware would be obvious without the guidance of an ASM, there is a humorous, obtuse, and complex, circuit in the literature attempting to solve the single-pulser problem, (that circuit clearly resulted from a bottom up approach; the reference is mercifully omitted).

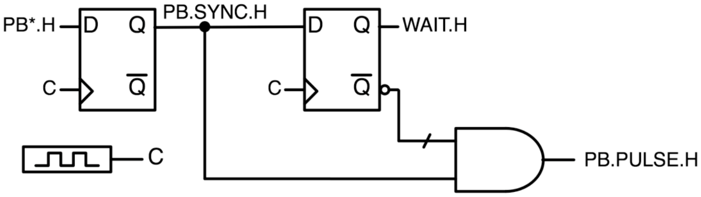


Figure 6-2. An ASM guided synthesis of a single-pulser circuit

We have purposely ignored initializing the single-pulser. On power-up the flip-flops could have any of four values: 00, 01, 10, and 11. What will the circuit do in each of these states? What should you do to make the circuit well behaved under all circumstances? (assume the PB*.H signal is false at power-up)

A Combined Architecture-Algorithm Solution

Although the foregoing derivation is our most fundamental solution of the single-pulser problem, there is another approach that illustrates an important connection between the architecture and the algorithm in digital design. Suppose we reason as follows. Our architecture consists of a synchronizing flip-flop for PB*, with output PB.SYNC, and another D flip-flop whose function is to delay the PB.SYNC signal by one clock time. The second flip-flop has input PB.SYNC and output PB.DELAYED. Then our single-pulser output signal PB.PULSE should be true only when PB.SYNC is true and PB.DELAYED is still false. As soon as PB.DELAYED becomes true, we must stop asserting PB.PULSE. Can we formalize these thoughts with an ASM? Of course, Fig. 6–3 is a one-state solution. This ASM requires no flip-flops for state generation, since there is only one state. The equation for PB.PULSE is

$$\text{PB.PULSE} = \text{PB.SYNC} \bullet \text{PB.DELAYED}$$

The implementation looks identical to our original circuit in Fig. 6–2 except for a trivial change in the names in the interior of the circuit.

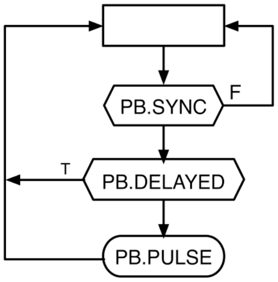


Figure 6-3 A single-pulser ASM in one state

Implications. This is an interesting phenomenon. We first had a two-state system with minimal architecture (one flip-flop), and now we have a one-state system with more inputs and more architecture (two flip-flops). Both solutions yield the same hardware.

This reflects an important concept in digital design. The distinction between architecture and algorithm is arbitrary. In general, by enlarging the architecture, we may convert any ASM into one with fewer but more complex states. At the limit, it is always possible to describe any algorithm in a single state. For complex systems, this yields such a messy ASM that it is not useful; nevertheless, it is technically correct. The vital point is that architectures and ASMs are tools to assist us to understand our problem and to produce a clear, correct implementation. The tools are to serve us, not control us!

Whereas we prefer the first solution to our example as more fundamental, there is merit in the second solution also. Both result in equivalent (in this case, identical) hardware.

A Single-Pulser Building Block

Having developed a circuit for producing a single pulse from a long input signal, we may package the circuit in a black box and treat it as one of our design building blocks. The single-pulser black box has an input PB* from an asynchronous source, and produces a one-clock-cycle true output PB.PULSE when the input becomes true. Whenever we need this type of behavior, we may mentally plug in our black box, and when we build our circuit, we use hardware similar to Fig. 6-2 in the box. It would be nice if the single-pulser box were available as a standard, but usually it is not. However, we still treat the single-pulser operation as a building block in our work.

Generalizing the Single-Pulser

Think about the human-machine interaction implied by our single-pulser problem. When the operator presses the button, the single pulse promptly appears (and disappears). The machine must have been ready to act on the pulse signal. How did the operator know when it was okay to press the button? Somehow, the operator must infer the correct time from the condition of the machine. Usually this would mean a light or some combination of lights on the control panel. The circuit controls these lights, indicating its readiness to process a button depression. The single-pulser works well in this common situation.

Now suppose we treat the operator-machine interaction differently. Let the operator press a button at will but require the operator to hold the button down until there is some indication that the machine has received the signal. Then the machine may be in any state when the button is depressed, and only when the circuit is ready to respond to the button will the light come on.

The single-pulser will not work in this case, but we can handle the situation with an ASM structure that looks for the pushbutton depression, then lights the light until the operator releases the button, and then performs the desired operation. Figure 6-4 is a typical form of this ASM. (In Design Example 5 in this chapter, we discuss an additional variation of the single-pulser algorithm.)

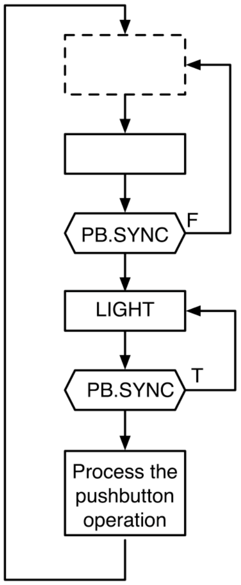


Figure 6-4. Diagram of an ASM if the operator may press a pushbutton at any time. In this algorithm, the machine waits until the operator releases the button before responding

DESIGN EXAMPLE 2: A SYSTEM CLOCK

The clock is the master pacer in digital systems, and we must design it carefully. A good clock is perhaps the most potent debugging tool a designer can put into a system. At the least, the clock must have an automatic mode whose frequency may be fixed at some high value and a manual mode that generates one hazard-free active clock edge when a pushbutton is depressed. To do this, there must be a mode switch to select automatic or manual status. Throwing the mode switch must not shorten a clock cycle in the automatic mode. If the mode switch were flipped 10 percent of the way into a clock cycle, thereby truncating the cycle at that point, our circuit could suffer from timing problems. We must let the last cycle run its normal course before the clock system enters the manual mode.

Our approach to design requires that we move difficult, universal concepts up front so that we can solve them once and then apply them to all projects. The system clock is an important concept so let's design a circuit for it.

Statement of the Problem

(a) Design a hazard-free system clock that runs in two modes, automatic and manual. The automatic mode is a fixed-frequency mode derived from a continuously running clock or oscillator. (Crystal controlled oscillators are a standard commercial module) The manual mode should produce a true clock output when a pushbutton is depressed, and a false output otherwise.

(b) Activating the mode switch must never cause truncation of a clock cycle.

(c) In the automatic mode, the clock circuit should ignore the manual pushbutton.

Digesting the Problem

There are two inputs, from a debounced mode switch and from a debounced pushbutton. We may formalize the variables in the problem. Let the mode switch output be MAN, and let MAN = T in the manual mode and MAN = F in the automatic mode. Let the manual pushbutton's output be PB, and let PB = T when the button is down and PB = F when the button is up. Let the clock output be CLK. Since we are dealing with a clock for synchronous edge-triggered systems, it is natural to let CLK = T be the high voltage level and CLK = F be the low voltage level, although this choice is not essential to the design.

In digesting the problem, we uncover a crucial point: the clock output must be free of hazards. In Chapter 4, you learned about the catastrophes that will occur when clocks deliver spurious edges. The most general way to avoid hazards is to avoid gates on the lines that must be hazard-free. How can we build something without gates? Commercial flip-flops are designed to have hazard-free outputs, *provided the set-up and hold time constraints are met*; violating these conditions can lead to meta-stability which is a disaster leading to system failures at unpredictable times. Errors every 10 minutes are easy to find, a failure every 10 years probably exceeds the useful life of most digital systems; it’s the ones that fail every 10 hours or 10 days that give a designer fits.

We will produce the clock output directly from a flip-flop, paying particular attention to stay within its design constraints. With hardware, don’t take chances!

Now we are ready to derive an ASM chart for our system clock circuit.

Algorithm for the System Clock

Our system is a synchronous circuit clocked by a continuously running oscillator that will drive the system clock ASM. The output CLK has two levels, T and F, so we might use two ASM states, one for CLK = T and the other for CLK = F. In the automatic mode, we would expect to flip back and forth between the states constantly. Our design problem is to fit the manual mode operations into this framework. In the manual mode, the ASM moves to whichever state reflects the PB position. Figure 6–5 is the ASM chart and the stars on MAN* and PB* signals alert us to potential difficulties that must be resolved before we rush to an implementation.

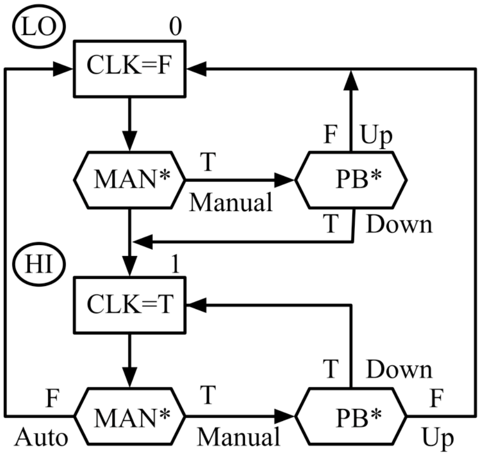
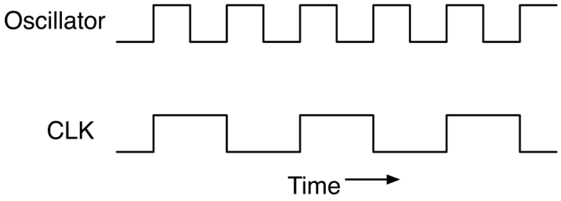


Figure 6-5. A system clock ASM

This ASM ignores the pushbutton unless the mode switch is in the manual mode. In the automatic mode, the active state alternates between **LO** and **HI,** producing a CLK output of half the ASM clock frequency, as shown in the following timing diagram:



The ASM formulation clearly demonstrates that the automatic mode overrides the pushbutton and that, whenever we switch from the automatic to the manual mode, the last automatic clock phase exists for its full duration, without any shortening.

What about the *'s on MAN and PB? The usual engineering analysis argues that they can be ignored in this case. The machine is always either in one state or in the other. When the mode switch is changed from automatic to manual, the ASM either detects the change during the present ASM state, or it doesn't. If it does, at the next state transition it moves into the manual mode; if the ASM misses the change, it remains in the automatic mode for one more clock cycle before entering manual mode. In either event, in this simple ASM there is nothing to go wrong as a result of the asynchronous nature of the MAN signal. A similar argument applies to PB. So we can drop the "*" on these signals in the ASM chart of figure 6-5.

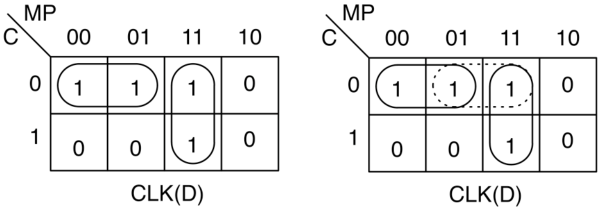
A deeper analysis should inject extreme caution into this simplistic rationalization. Meta-stability is an ever-present danger when a flip flop's data inputs change at the same time as the clock, (this is discussed in appendix ***).** At all costs you must avoid this, so without further ado lets dispose of this potential problem by synchronizing MAN* and PB*

Implementing the Circuit

The transition table for the ASM is:

Present state	Next State	Condition
LO	LO	$\mathrm{MAN} \bullet \overline{\mathrm{PB}}$
	HI	$\overline{\mathrm{MAN}} + \mathrm{MAN} \bullet \mathrm{PB} = \overline{\mathrm{MAN}} + \mathrm{PB}$
HI	LO	$\overline{\mathrm{MAN}} + \mathrm{MAN} \bullet \overline{\mathrm{PB}} = \overline{\mathrm{MAN}} + \overline{\mathrm{PB}}$
	HI	$\mathrm{MAN} \bullet \mathrm{PB}$

From this the equations for CLK(D) are readily derived by plotting on a K-map:



(a) (b)

You may be tempted to use the circling in (a) but we know from a discussion of hazards that it may contain a potential glitch, so, without thinking about it, use the circling at (b). Gates are cheap, errors expensive.

$$\mathrm{CLK(D)} = \overline{\mathrm{C}} \bullet (\overline{\mathrm{M}} + \mathrm{P}) + \mathrm{M} \bullet \mathrm{P}$$

Let's use the traditional gate method for implementing the state generator; one flip-flop will encode the two states in our ASM. The purpose of the design is to produce the system clock output CLK. We have already decided to produce the system clock signal CLK as the output of a flip-flop. According to the ASM, CLK is true in state **HI** and false in state **LO**. In Fig. 6-5, we chose to represent **LO** by 0 and **HI** by 1. This is the same behavior as the CLK output: false in **LO** and true in **HI.** In this case, we get our desired CLK output from the same flip-flop used in the state generator. Very convenient.

Both voltage signal forms will be available for the debounced switch and pushbutton variables. This almost completes our solution to the design problem except for the dangling direct set and clear flip-flop inputs. What should be done about them?

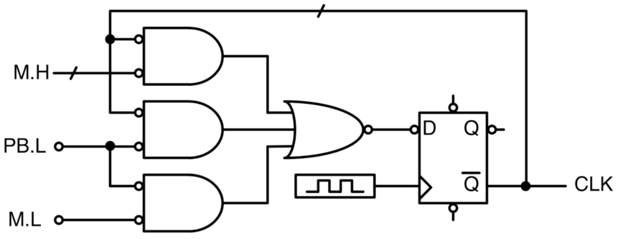


Figure 6-6. A traditional gate implementation of the Clock ASM

In our own practice we never use JK versions of state generators because of the obscure process leading from an ASM to a final circuit. However, in this one case, it does lead to the somewhat simpler circuit of Fig. 6-7 and you may use it if you wish and are willing to accept it on faith. (It does have the interesting property that the external feedback of Fig. 6-6 is subsumed into the internal feedback of the JK flip-flop. (As a pedagogical exercise, JK synthesis is covered in appendix *)

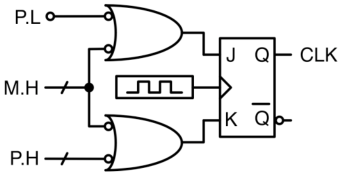


Figure 6-7. JK implementation of the clock ASM

As a designer, the first thing you should do after deriving a hardware implementation of an ASM is to check the logical correctness of your derivation by simulation.

There are many levels of simulation, ranging from detailed modeling of the analog behavior of gates that account for loading and give accurate timings, to simple logic simulations, which ignore many fine grain details. At this stage we want a simulation at the level of *ASM transitions* that is system clock transitions. Herein lies a problem: real circuit simulators perforce work at the *gate* level and there will ordinarily be many gate level transitions after leaving one ASM state and arriving at the successor state. At the ASM level we are not interested in these intermediate transitions; that comes later after you have verified that your design is logically correct and when you want to find out how fast it runs. At the logic level we are concerned only with correctness, not speed. Not only that, when working at this level you ordinarily want to work with a manual clock so intermediate gate transitions can ripple to completion, so you can inspect the final results for that ASM state. After verification of all ASM transitions, you will then want to run your simulation at full clock speed to verify correctness at that level.

Even if your final hardware circuit runs with a fixed clock you can still debug a simulated version using a manual clock, a powerful debugging tool and one we highly recommend.

The circuits of Figure 6-6 or 6-7 are a neat way to trick a gate level simulator into being an ASM level simulator. All gate level transitions will still be properly sequenced but only the final result will show after each depression of the manual clock pushbutton.

As a matter of course, always use the output of circuit of fig. 6-6, or the JK version in figure 6-7, to drive all clocked modules when doing logic-level simulations of digital systems.

DESIGN EXAMPLE 3. Serial Data Transmission.

As a designer, you will sooner or later have to manage movement of data and you have a choice of moving it in parallel chunks, (byte or word), or breaking chunks into bit streams at the sender and re-assembling bit streams into chunks at the receiver.

Intuitively one would assume parallel transmission would be faster since a chunk, as opposed to a bit, is transferred every *clock* cycle. There's a reason *clock* is italicized here. Whose clock?

As long as your data is localized to a single Silicon chip you will have tight control of clock skew and you can rely on data and clocks being tightly locked together at both sender and receiver. In this case wider chunks are better and this drives designers to ever-wider bus widths.

As soon as data moves off chip an entirely different design constraint rears its ugly head. With a parallel data bus you must guarantee two things:

a) bits must arrive at the destination with negligible inter-bit skew.

b) more importantly, a separate wire must carry a clock locked to the data. While that's trivial at the sender it is very difficult to guarantee it at the receiver; as clocks get faster it can become impossible.

In modern systems we bypass these difficulties by breaking data chunks into serial bits, thereby elimination skew, and more importantly, using transmission protocols that allow the serial bit stream *itself* to encode a clock locked to bit intervals. No matter how long the wires, we can then recover a bit clock at the receiver.

SERIAL-PARALLEL DATA CONVERSIONS

In this example, we design circuits to convert parallel data into serial form and at the receiver convert serial data back into parallel form. Our example has two independent circuits: a parallel-to-serial converter and a serial-to-parallel converter. Both converters deal with 8-bit parallel bytes, the conventional unit for data transfer.

Specifying the Problem

Parallel-to-serial conversion. The parallel-to-serial (P→S) converter accepts 8-bit bytes from some source and transmits the bits serially on a serial-out line (SO) at a specified bit rate. As long as the system is running, the serial bit rate is constant, and the P→S process never stops. Bit rates range from 56k (phone modems) to 460 mega bits/second in USB devices.

The need to maintain a valid serial stream of data at this fixed rate places a severe constraint on the device that supplies the 8-bit bytes: whenever the P→S converter needs a new byte, the new byte must be present. But the byte supplier is running at its own speed, engaged in its own duties, only one of which is to supply bytes to the P→S converter. A simple way to handle such a situation is to provide *a one-byte buffer* register in the converter to hold the incoming byte whenever the sending device supplies it. Then our converter can move the data to another spot when it is ready to process the byte, thus freeing the buffer to hold another byte.

(The term *buffer* has two meanings in digital design. Previously in this book, the word has meant a source of power for a logic signal. Here, we use the term in the software sense of a temporary storage area to accommodate differences in the operating characteristics of a source and a destination.)

We need some way to notify the supplier of bytes when it is time to fill the converter's buffer with a new byte. The byte supplier must not provide a new byte too soon, lest the new byte destroy the previous byte in the buffer before the P→S converter has processed it. On the other hand, a basic presumption of this system is that the byte supplier *must not fail* to supply a byte on time; otherwise, we cannot supply the continuous stream of bits required by the serial-out line. We may use a simple one-way signal to tell the byte supplier to fill up the buffer. No response from the byte supplier (other than filling the buffer!) is necessary, since we cannot tolerate any slippage and our converter could do nothing about a failure if one occurred. Let's use a variable FILLIT as a signal to the byte supplier; whenever the converter accepts a new byte from its buffer, the converter will toggle (complement) FILLIT. At a time safely before the converter needs the next byte, the byte supplier must sense this change in FILLIT, and provide a new byte into the buffer. FILLIT behaves like a modulo-2 counter.

Building the P→S Converter

Design. The architecture will have an 8-bit buffer register to hold a byte from the byte supplier, and an 8-bit shift register to hold a byte while it is being disassembled and shipped out bit by bit over the serial-out line. Also, there will be a controlled flip-flop FILLIT to tell the byte supplier when to deliver another byte.

The basic timing element for parallel-to-serial conversion is the serial bit time, so a clock operating at the P→S bit rate is a natural system clock for our synchronous design.

There are two approaches to the design of the control algorithm. We could draw a nine-state ASM chart that produces a serial SO bit in states 1-8. This ASM could load a byte from the buffer into the shift register in state 0 and shift the byte one position to the right in each of the other eight states.

Alternatively, we can view the ASM as having a single state, with the architecture containing a binary counter capable of counting from 0 through 8, to distinguish the nine activities per byte. Let's adopt this latter view and add a counter to our architecture. The ASM in Fig. 6-8 is then self-explaining; Fig. 6-9 shows the supporting architecture.

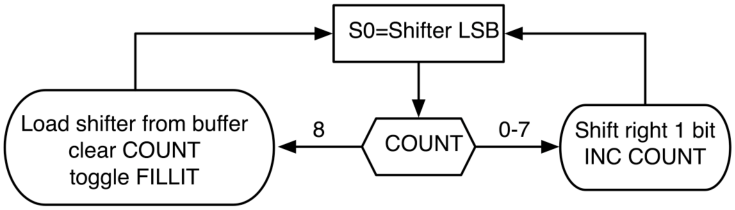


Fig. 6-8. ASM for a single buffer P→S converter

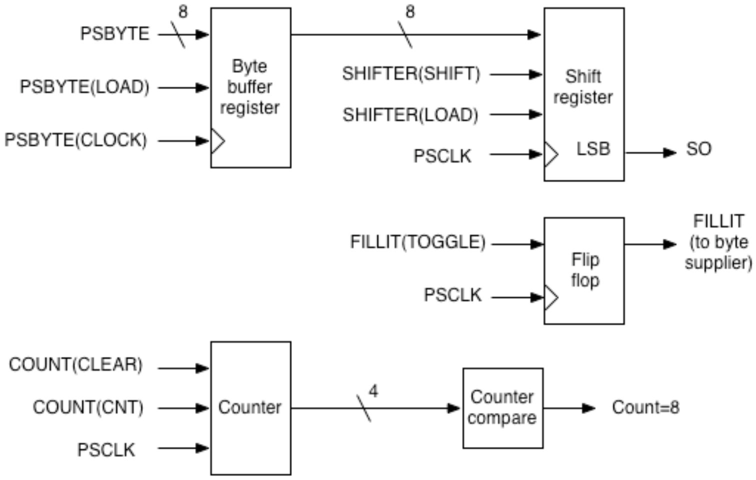


Fig. 6-9. Architecture for the single buffered P→S converter

An alternate ASM and architecture would be *double buffering.* Two shift registers feed a multiplexer, while one is shifting the other is loading with roles exchanged when the shifter becomes empty. The relative merit of this ASM and architecture is left as a problem.

Both architectures are straightforward, fundamentally because the P→S clock belongs to the sender and is fixed. Things are not so simple for the receiver.

Building the S→P Converter

Clocks are the problem here. Consider a phone or cable modem that carries a data signal only; even if you had two separate signal paths, one for data and one for clock, there is no chance the relative cable delays would be equal. Phone signals often travel miles before reaching a destination and once a signal enters the phone network, who knows what insults it will encounter in transit.

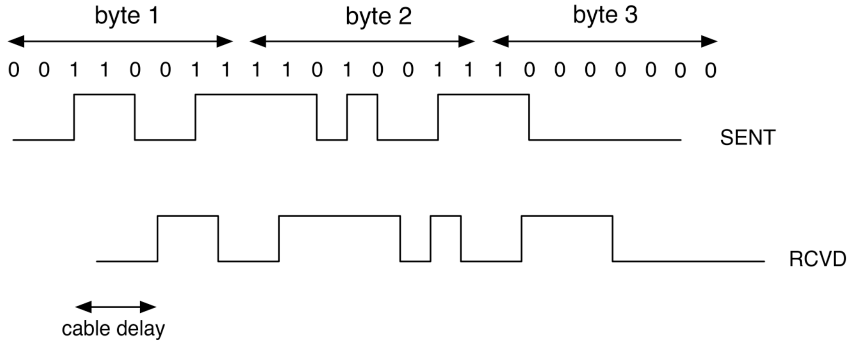


Figure 6-10. Input and output signals in a serial data path

Somehow, we must be able to take the received signal, in isolation, and recover a clock from its data-*without reference to the sender's clock.* Fortunately, a remarkable analog circuit called a Phase Locked Loop (PLL) will do just that. (see appendix *)

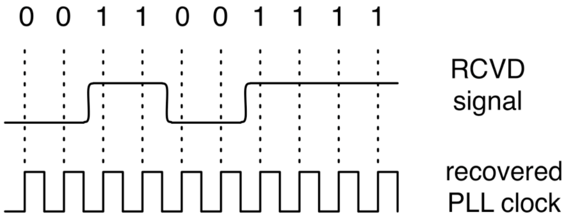


Fig. 6-11. Using a PLL to clock a received analog signal

Now that we have a clock locked to the data we are back in familiar synchronous design territory. All we need to do is make sure we sample the received analog signal after each transition has had time to stabilize before clocking it into the receivers shift register.

One caveat remains: PLL's require time to lock onto an incoming analog signal. Assume the incoming line is quiescent with no L→H or H→L transitions the PLL can use for lock. Before sending data we must preface it with a special SYNC pattern, distinct from *any* standard data byte, to allow the PLL to acquire lock and start producing clock edges for downstream data manipulation. The number of bits and bit pattern in the SYNC preamble are protocol dependent but in any case will be known to the receiver.

Now that we are back on familiar turf, the ASM and architecture for a single buffered S→P converter are straightforward exercises.

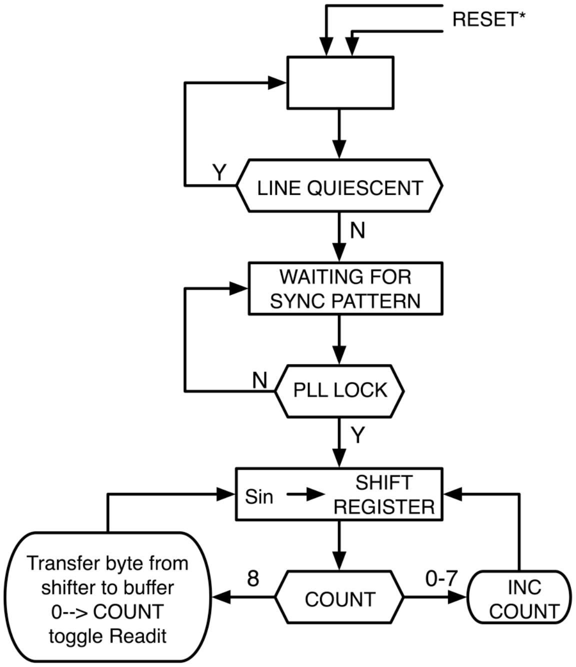


Figure 6-12. Serial to Parallel ASM

We leave the design of a double buffered ASM and architecture as a straightforward embellishment of the single buffered protocol.

A more complete discussion of a popular serial protocol, USB 2.0, is covered in appendix* for those who wish to delve deeper into clock recovery and serial protocols.

DESIGN EXAMPLE 4: A TRAFFIC-LIGHT CONTROLLER

This example was inspired by a similar problem in Carver Mead and Lynn Conway's pioneering book, *Introduction to VLSI Systems.* We will solve the problem with our structured design techniques.

Statement of the Problem

A busy highway is intersected by a little-used farm road, as shown in Fig. 6-13. The farm road contains sensors that cause the signal CARS to go true when one or more cars are on the farm road at the positions labeled `C.' We wish to control the traffic signals at the intersection so that, in the absence of cars waiting on the farm road, the highway light will be green. If a car activates the sensor at either position C, we wish the highway light to cycle through yellow to red and the farm-road light then to turn green. The farm-road light is to remain green only while the sensors indicate the presence of one or more cars, but never longer than some fraction of a minute, after which it is to cycle through yellow to red and the highway light is to turn green. The highway signal is not to be interrupted again for farm-road traffic until some fraction of a minute has elapsed.

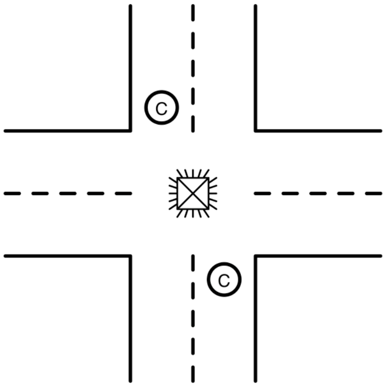


Figure 6-13. The location of the traffic signal and sensors.

Preliminary Considerations

The highway traffic is given priority, but not to the extent that the farm-road traffic can be stalled indefinitely. Since the default condition is a green light on the highway, we do not need any sensors in the highway lanes. To keep the example uncluttered, we will assume that the outputs of the sensors are combined external to our design to produce the single signal CARS, and that this signal satisfactorily indicates the presence of cars desiring to enter or cross the highway. We might ask what signals the traffic lights must receive to activate their three colors, but we defer such inquiries because we would like our solution to be independent of any particular brand of traffic signal until we are ready to specify one.

The control of the traffic signals involves four intervals: the minimum time the highway light will be green, the maximum time the farm-road light will be green, the duration of the highway's yellow signal, and the duration of the farm-road's yellow signal. For simplicity, we assume that the first two intervals are the same and that both yellow-light intervals are the same. To generate signals representing these two intervals, we plan to have a timer, driven by a (high-speed) master clock and initiated by a starting signal START.TIMER. Whenever the timer is started, two output signals, THOLD and TYEL, are negated. If the timer is not interrupted, the two signals will be asserted after their respective intervals have elapsed; the signals will remain asserted until the timer is restarted with START.TIMER. The same timing unit and the same enabling signal will suffice for both timings, since the two intervals do not overlap.

Our preliminary architecture is shown in Fig. 6-14.

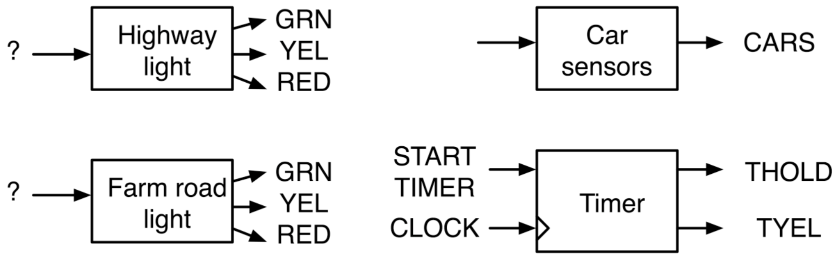


Figure 6-14. Preliminary architecture for traffic light controller

The Control Algorithm

The control of the traffic signals breaks naturally into four events, which will result in four ASM states:

State **HG:** Highway light green (and farm-road light red).

State **HY**: Highway light yellow (and farm-road light red).

State **FG**: Farm-road light green (and highway light red).

State **FY**: Farm-road light yellow (and highway light red).

When the highway light is green (state **HG**), the controller must be alert for farm-road traffic, and, if cars are on the farm road and sufficient time has elapsed, must cycle the lights through state **HY** to state **FG**. In state **FG**, when the farm-road light is green, the controller must be prepared to cycle through state **FY** to state **HG** whenever no cars remain on the farm road or if the stipulated time has elapsed. These observations lead quickly to the ASM in Fig. 6-15. Each state tests one of the two intervals THOLD or TYEL, and so, as we enter each state, we must start the timer unit.

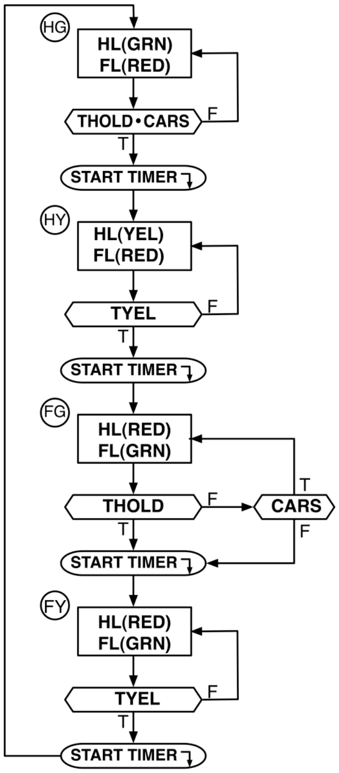
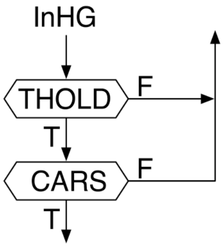
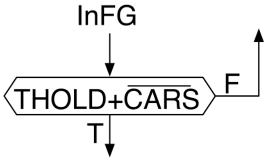


Figure 6-15. ASM for the traffic light controller

In state **HG**, the ASM describes the mutual requirement of farm-road cars and a sufficiently long interval, using a single test of the product of THOLD and CARS. In state **FG**, the ASM uses separate tests of THOLD and CARS to describe the logic resulting in the escape to the next state. These constructions seem natural to us, so we used them. Other ways of representing the test conditions will lead to exactly the same output equations and state generator; for instance, follow the synthesis below, and then repeat the synthesis with these tests:





Realizing the ASM

From the ASM chart, we derive equations for the outputs and we tabulate information that will lead to the construction of a state generator. The natural parameters to describe the condition of a traffic signal are, of course, the colors of the signal. We derive:

HL.GRN = HG
HL.YEL = HY
HL.RED = FG + FY

FL.GRN = FG
FL.YEL = FY
FL.RED = HG + HY

The signal for starting the timer unit is

$$\text{START.TIMER} = \text{HG}\bullet\text{THOLD}\bullet\text{CARS} + \text{HY}\bullet\text{TYEL} + \text{FG}\bullet(\text{THOLD} + \text{CARS}) + \text{FY}\bullet\text{TYEL}$$

(the conditional outputs with the down arrows in Figure 6-15 mean the timer should be started at the end of the conditional time).

While the MUX method for next state generation is largely of historical interest, we will use it here for pedagogical purposes. All finite state machines use the present state and status inputs to generate the next state; *we care not how*, as long as the hardware comes up with the proper next sate. Using mathematical function notation:

NEXT STATE =**f** (PRESENT STATE, STATUS INPUTS)

Lets use table lookup, (think MUX!), to implement the state generator function, "**f**".

We assume an encoded state generator, which will require two D flip-flops. The encoding is arbitrary: we use B and A as the state variables, and the following assignment:

State	B	A
HG	0	0
HY	0	1
FY	1	0
FG	1	1

For such a simple ASM, we could write down the state generator by inspection. Nevertheless, in almost every design we find it useful to tabulate the conditions for changes in state. Table 6-1 shows the next-state conditions for our traffic light controller. Figure 6–16 shows the state generator.

Present state	Code	Next state	BA	Condition
HG	0	**HG**	00	$\overline{\text{THOLD}\bullet\text{CARS}}$
		HY	01	$\text{THOLD}\bullet\text{CARS}$
HY	1	**HY**	01	$\overline{\text{TYEL}}$
		FG	11	TYEL
FY	2	**FY**	10	$\overline{\text{TYEL}}$
		HG	00	TYEL
FG	3	**FG**	11	$\overline{\text{THOLD}}\bullet\text{CARS}$
		FY	10	$\text{THOLD}+\overline{\text{CARS}}$

TABLE 6-1. Conditions for state transitions in the traffic light controller

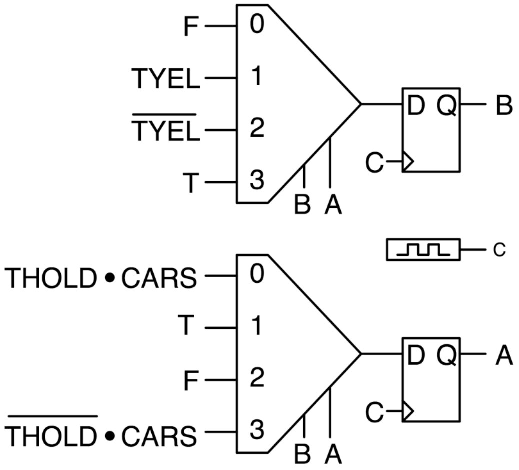


Figure 6-16. A Multiplexer state generator for the traffic light controller

Choosing a Particular Traffic Signal

The ASM's outputs that are to control the traffic signals are expressed in terms of the active color of the lights. Now suppose we select a particular traffic signal and, reading the instructions, find that each signal is controlled by a 2-bit code that specifies which of the three colors is active. In Fig. 6–17 we show this particular choice of a traffic light, adopting T = H for the code. Now we may derive logic equations that express the bits of the manufacturer's code in terms of our traffic-light variables:

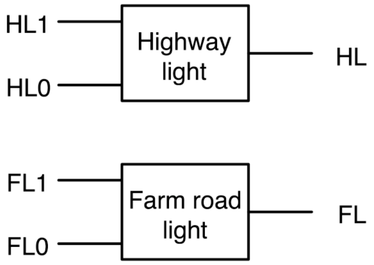
HL1=HL.YEL HL0=HL.RED

FL1=FL.YEL FL0=FL.RED

Now that the behavior of the particular traffic signal has been expressed in our nomenclature, there remains only to plug in the particular expressions for each light's colors to complete the implementation. Using the ASM output equations, we get:

HL1=HY
HL0=FG+FY
FL1=FY
FL0=HG+HY

Our refusal to commit ourselves to a particular traffic signal left us with an incomplete early statement of the architecture in Fig. 6–14 but allowed us to form a solution independent of the brand. When we finally settled on a brand, in Fig. 6–17, we completed the solution *without altering our original work.* This is an important technique, and represents good top-down design.



HL1	HL0	HL
0	0	GRN
1	0	YEL
0	1	RED

FL1	FL0	FL
0	0	GRN
1	0	YEL
0	1	RED

Figure 6-17. A particular traffic light for Design Example*

DESIGN EXAMPLE 5: THE BLACK JACK DEALER

As a slightly more complex example we will design a machine to simulate the dealer’s actions in a black jack game. Black jack is a familiar card game involving a dealer and one or more players. The players can exercise their judgment but the rules specify what the dealer will do with each new card he receives.

In the supplementary laboratory that accompanies this course you will use a simulator to build a gate level implementation a real computer and, for the adventuresome, go on to implement that design real hardware. Absent this exposure, you can get some feeling for real world design by simulating the Black Jack Dealer on a commercially available simulator, and we strongly urge you to do so.

The Rules of Play for the Dealer

The cards have values of 1 (ace) to 10 (10 and face cards). An ace may have the value of 1 or 11 during the play of the hand, whichever is advantageous. The dealer deals himself cards one at a time, counting ace as 11, until his score is greater than 16. If the dealer's score does not exceed 21, he "stands," and his play of the hand is finished. If the dealer's score is greater than 21, he is "broke" and loses the hand. The dealer must revalue an ace from 11 to 1 to avoid going broke but must then continue accepting cards ("hits") until the count exceeds 16.

Stating the Problem

These rules understood, we may state our hardware design problem. With a human operator to present cards to the Black Jack Dealer machine, play the dealer's hand to produce Stand or Broke.

This is an algorithm, and as such, it is always helpful to think about it in as many ways as possible before casting it into ASM formalism. For readers who come from a software background, we could also state the problem in the form of a program written in pseudo code. This suppresses most of the machine details, including the detailed state timings that we must eventually specify. If we don't understand the problem at an operational level, we surely cannot build a correct machine! Here is a pseudo code statement of the dealer's algorithm. The variables have obvious meanings, except for *ace11flag,* which remembers if the algorithm has valued an ace as 11 points.

```
new_game:      {score=0;
                ace11flag=false;
                stand=false;
                broke=false;}

       hit:    score=score+card_value; /* accept a card*/
               if (card=ace and ace11flag=false)
                    then {score=score+10;
                           ace11flag=true;}

another_hit?:if (score <= 16) goto hit;
               if (score <21)
                    then { stand=true;
                           display score;
                           goto new_game;}
               if (ace11flag=true)
                    then { score=score-10;
                           ace11flag=false;
                           goto another_hit?;}

                    else {broke=true; goto new_game;}
```

Since this is pseudo code we are free to develop our own program style. If you prefer stating the algorithm in your favorite high-level language that's OK too. Do whatever it takes to get a rock solid feel for any algorithm before even thinking of hardware implementations.

You will note the liberal presence of **goto** statements in our algorithm, something usually discouraged in high level languages; but state machines are essentially **goto** constructs, where state generation hardware computes the next state as a function of present state and status variables. Hence we depart from standard high level programming conventions by explicitly coding with **goto**'s to more closely conform to state machine hardware. Further, notice that we use the {…} notation to indicate operations that hardware can carry out in parallel—in one state—something that software must do sequentially.

Digesting the Problem

We will call the hardware device Dealer. The basic questions we might ask in our first stab at architecture are: How will the operator present cards to Dealer? How will the operator know what the result of each card is? We can see that the operator interacts closely with the machine:

a) Dealer must signal the operator when to deal a new card.
b) The operator must signal Dealer when the new card's value is ready for processing.
c) Dealer must tell the operator if Dealer stands, went broke, or if the hand is still in progress.
d) When Dealer stands, the operator must be able to see the point value of Dealer's hand.

These thoughts suggest that an important aspect of Dealer's design is the interaction between the machine and the human. Assuming that the operator knows the binary number system, we choose a set of four toggle switches to hold a card value of from 1 to 10 in binary. These toggle switches will act as a register for the input data. A set of five lights is a simple way to display Dealer's score, which cannot exceed 21. To control the interaction, we can give the machine a set of status lights: **Hit**, to tell the operator when to enter a new card for Dealer; **Stand** and **Broke**, to inform the operator of the final results of the hand. We could provide a **New.Game** status light to show when to begin a new hand, but this is unnecessary, since either **Stand** or **Broke** must be signaled when the previous hand is complete, and thus the operator knows when a new game may begin without a special **New.Game** light.

We must not forget to give the operator a way to tell Dealer when to process a card; therefore, we will specify a pushbutton switch **Card.Ready** that the operator can press.

At this point we have a fairly good idea of the interface between Dealer and the human operator. Figure 6-18 shows the general plan. This is an important step in our design, for at this point we may go to our potential operators and describe how they will use the Dealer machine. Presumably, the operators don't much care what Dealer is like inside, but they will be interested in the operating instructions—the user's manual. Talking to users at this stage in the design can help avoid agonizing redesign later, in case we have misunderstood the problem.

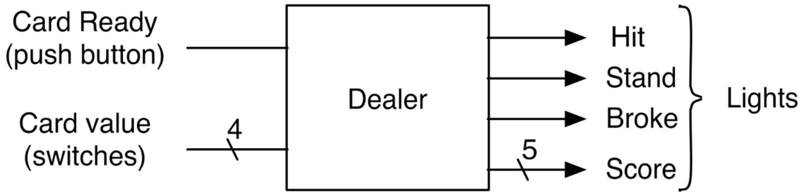


Figure 6-18. The Black Jack Dealer's operator interface

Most of the architecture is not yet specified—only the interface signals. Hopefully, by now that we have a clear enough understanding of the problem to begin work on the control algorithm.

Initial ASM for the Black Jack Dealer

With the operational algorithm as a guide, we may propose a hardware algorithm, using the ASM notation. As usual, our design will be synchronous, running from its own internal clock at a speed independent of the human operator's actions. As we develop the ASM, we will gain insight into the internal architecture required by Dealer. Figure 6–19 is a first attempt to describe an ASM for Dealer. The ASM assumes the following architectural elements:

a) Memory flip-flops for the HIT, STAND, and BROKE signals.
b) A register to hold the current card value.
c) A register to hold SCORE.
d) A flip-flop for ACE11FLAG.
e) A black box to add, subtract, and clear the SCORE.
f) A black box to report if CARD = ACE, SCORE > 16, and SCORE > 21.

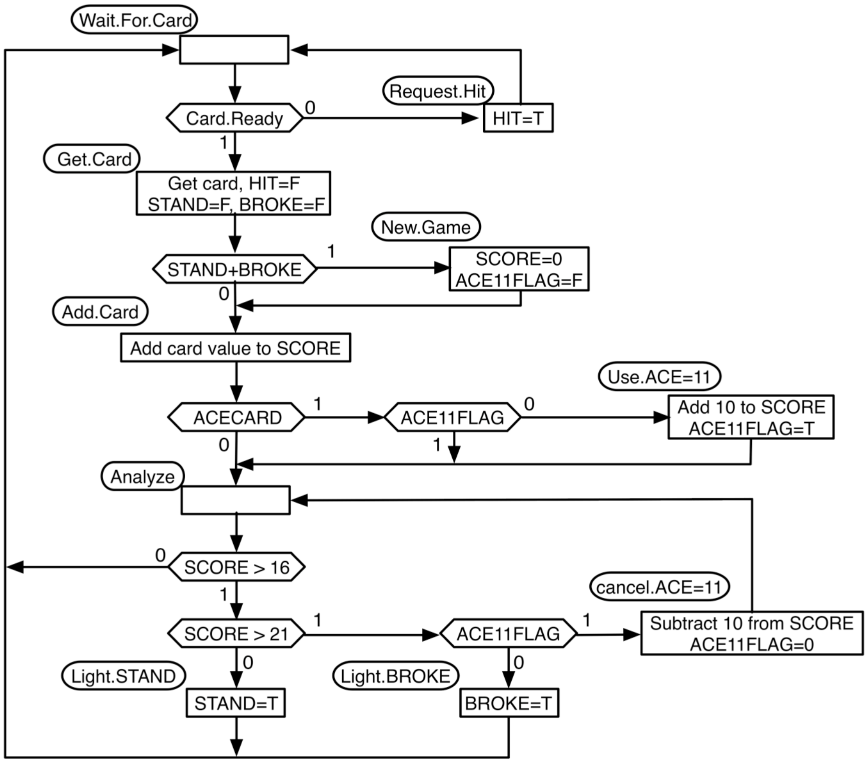


Figure 6-19. A primitive ASM for the Black Jack Dealer with too many states and an error

Reducing the Number of States (general discussion)

Our ASM has quite a few states, and you may wonder if this is desirable. States are the fundamental element of a hardware control algorithm, but they may have two undesirable side effects:

(a) Each state requires a clock cycle to execute.
(b) Overall complexity may increase and lead to more hardware.

Complexity issues are case by case specific and it is difficult to formulate general rules. For one-hot designs, reducing states clearly causes a corresponding reduction of state flip-flops and corresponding reduction of clock cycles. However, next state logic usually gets more complicated and this may increase the total number of circuit elements. The trade off will ultimately reduce to a 3-way consideration of speed vs. complexity vs. algorithm transparency. Our strong inclination is to always favor algorithm transparency and recommend this path until you become an experienced designer where circumstances may sometimes dictate ultimate speed or minimum hardware.

For encoded state machines the issues are a good deal murkier. Consider the two state machines in figure 6-20, one of 4 states and the other of just a single state.

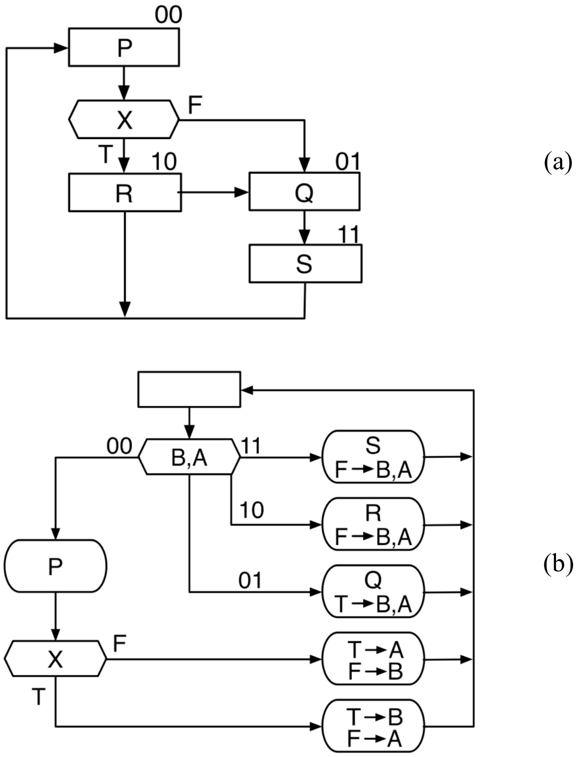
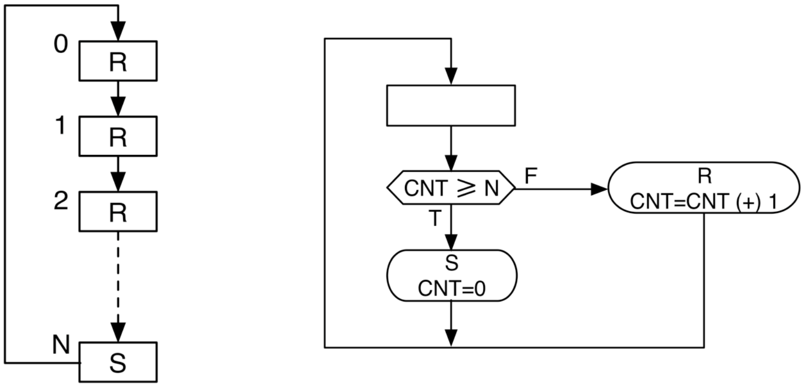


Figure 6-20 Converting a four-state ASM (a), into an equivalent one-state ASM (b)

This extreme example reduces the entire ASM into a single state, technically this is always possible by using auxiliary flip flops (here B and A) to remember the last path followed; knowing that path provides enough history to allow calculation of the next path. Obviously, this recasting of the algorithm, although technically equivalent to the original four-state version, is neither as clear nor as meaningful. The test inputs B and A are artificial and have no real meaning in the algorithm. Since we want maximum clarity, we reject this particular single-state ASM, and in general take a dim view of such esoteric tricks. (We leave it as a problem to compare the speed and hardware complexity of the two approaches in Figure 6-20).

There is one case where using auxiliary variables makes sense: a purely sequential ASM—a cycle of states with no tests—where the natural auxiliary variables are simply bits of a counter used to cycle through the states; the state generator is then that binary counter, which counts up to the maximum state value and then resets to 0. On occasion, we may increase the clarity by making the counter a part of the architecture, as in Fig. 6-21b. In both designs, the hardware is the same. Choose the algorithm that seems clearer for your problem. Design Examples 3 and 4 in this chapter also illustrate this point.



(a) (N+1) state cyclic ASM (b) One-state ASM with counter architecture

Figure 6-21 Equivalent formulations of a cyclic process of N+1 elements

Reducing the Number of States (black jack machine)

In general, use moderation in eliminating states, having increased clarity as your goal. In the Black Jack Dealer ASM, we may easily incorporate the outputs of states LIGHT.STAND, LIGHT.BROKE, and CANCEL.ACE=11 into conditional outputs within state ANALYZE. We may also eliminate the NEW.GAME state.

In the Black Jack Dealer machine, neither saving states, or complexity of this simple ASM, is apt to be serious. It is likely that the human operator is much slower than the clock; so superfluous clock cycles will not cause difficulty. Also, we have straightforward methods of developing state generators from ASM charts of any reasonable size.

Nevertheless, as practice in dealing with more complex and demanding designs, we will attempt to reduce the 10 states to a smaller number. Our basic technique is to introduce conditional outputs into an existing state, to replace the unconditional outputs of a separate state. States are candidates for collapse into the previous state if the state's activities (particularly its outputs) do not need to follow the activities of its predecessor state sequentially. Such state-saving moves do not necessarily save hardware. Although the number of flip-flop memory elements for states may decrease, the command output logic usually becomes more complex. Experience shows that a moderate effort to save states is worthwhile, as long as we maintain clarity in our design.

Errors in the Algorithm

The ASM in Fig. 6–19 contains two errors. Can you find them? They are both in the interface with the operator, and you have seen both earlier in this chapter. Consider the Card.Ready button and its use. First, we assume by now that you will have debounced the CARD.READY signal. (We always debounce mechanical switch signals that are used to provide test inputs in our ASM.) But CARD.READY changes with the operator's actions and is not synchronized with the ASM's clock. In our ASM, we should therefore label this signal CARD.READY* (* for asynchronous). We hope you noticed this error, since asynchronous inputs are a common problem and you must be alert to them. Our treatment of this asynchronous test input is immediate and ruthless. Without pausing to investigate whether this asynchronous signal will cause problems, we eliminate it. We synchronize CARD.READY* with a D flip-flop operating synchronously with our ASM. In so doing, we add an element to our architecture. Call the output of the flip-flop CARD.RDY.SYNC. It is this new synchronized signal that the ASM tests in the WAIT.FOR.CARD state.

Races, Again (for encoded state machines)

Let's again explore the problems introduced when an ASM tests an asynchronous input. Assume that we have allowed the CARD.READY* signal to remain in the ASM and that we have made an encoded state assignment.

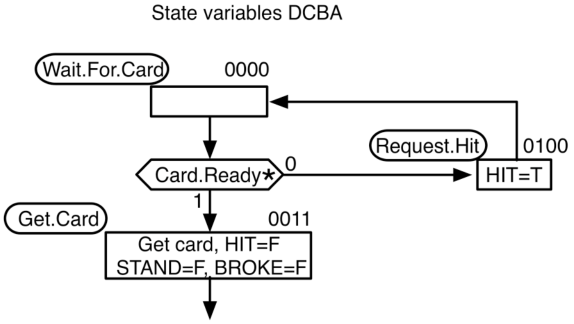


Figure 6-22 A segment of an encoded ASM with a transition race

Figure 6-22 is the relevant part of the ASM. In state 0000, when CARD.READY* = F, the state generator logic will be preparing to change the state variable C from 0 to 1. If CARD.READY* becomes true in state 0000, the state generator logic will switch C back to 0, and B and A to 1. If CARD.READY* changes to true very close to the transition point for leaving state 0000, the inputs to the state flip-flops will be changing when the clock edge occurs, and the resulting outputs of flip-flops C, B, and A are unpredictable. The next state might be any of eight possibilities: 0000, 0001, 0010, 0011, 0100, 0101, 0110, or 0111 ! In this particular example, states 0000, 0011, and 0100 would be tolerable, but the rest are clearly erroneous. The situation is a transition race, and you must eliminate it.

By modifying our ASM chart in a straightforward way (but retaining the asynchronous CARD.READY* signal), we can illustrate another type of race, the output race. If we manipulate HIT in conditional outputs in state WAIT.FOR.CARD instead of in separate states, we have the partial ASM in Fig. 6-23. Should CARD.READY* change from F to T near the clock edge, not only is the next state in doubt, but since the input to the HIT flip-flop is changing from T to F, the HIT output is also uncertain. We may reach the GET.CARD state successfully, but the Hit light may still be on! The output race is characteristic of outputs that are conditional on asynchronous test inputs. You must eliminate these races.

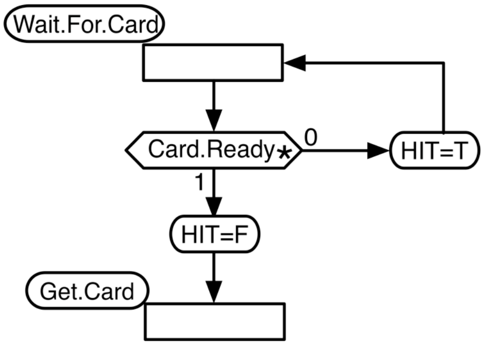


Figure 6-23. A segment of an ASM with an output race

There is a welter of traditional and complex techniques for dealing with the problem of races by manipulating state assignments. The straightforward way, as you know, is to eliminate the cause of the problem! If all test inputs are synchronous with the ASM clock, there can be no races. It is tempting to study your particular ASM to see if a particular asynchronous input can cause trouble. Usually, this is wasteful, mind-cluttering activity. Synchronize the input and move ahead.

This sweeping simplification works because the ASM's action depends on only one (synchronized) input at a time. If the control of the ASM requires the simultaneous synchronization of more than one asynchronous input, no method will produce reliable results. This is bad design of the external interface. Modern practice requires that all relevant inputs be stable (unchanging) at the time the single status signal announces that an event is to occur. For our Black Jack Dealer, we require that the card switches be set prior to the single CARD.READY* announcement.

Races, Again (for one-hot state machines)

Let's contrast one-hot state machines with encoded state machines. Encoded controllers use a set of flip-flops as a *pointer* to states; one-hot controllers are pointer-less, using one flip-flop per state, and thus are inherently *decoded* state representations. Figure 6-22 will now have three separate D-flip-flops: WAIT.FOR.CARD, REQUEST.HIT, and GET.CARD.

All clocked flip-flops have a time window surrounding the clock edge when inputs must not change: set-up-time before the clock edge, hold-time after the clock edge. If D, JK, or T inputs change in this window the flip-flop can settle unpredictably into either a 0 or 1 state. If CARD.READY* changes during these time windows then REQUEST.HIT and GET.CARD could wind up with all 4 possible values: 00, 01, 10, and 11. The 00 configuration would correspond to a no-hot state and the 11 configuration to a two-hot state, either would be disastrous.

The discussion for output races is unchanged from that in Figure 6-23. Why?

So, synchronize all asynchronous test inputs as a matter of course!

Process Synchronization

Our problems are not quite over. Suppose the operator presses the Card.Ready button. What happens if, as is likely, the ASM completes its actions in response to the CARD.RDY.SYNC signal and returns to the WAIT.FOR.CARD state before the operator has released the pushbutton? Dealer will process the same card again, since CARD.RDY.SYNC is still true. This is the problem studied in this chapter's first example, the single-pulser. Because of its importance and the subtlety of some of the implications, we will discuss the subject again, from a different perspective.

Handshakes. The problem is a failure to complete *a full handshake* between the operator and the dealer. *A full handshake* is an important mechanism for controlling the activities of two independent but cooperating processes. Frequently one device (A) must issue a request for action to another device (B). Since the speed and state of device B are unknown to device A (and vice versa), we need a general method by which device A can request action of device B and can be certain that device B has recognized the request. The sequence of events in the full handshake is:

(1) Device A senses that device B is not still acknowledging a previous request and requests an action (device A extends its hand).

(2) Device B senses the request and acknowledges receipt of the request (device B extends its hand).

(3) Device A senses device B's acknowledgment and drops its request (device A drops its hand).

(4) Device B senses that device A has recognized device B's acknowledgment and drops its acknowledgment (device B drops its hand).

The success of the handshake depends heavily on the sequence of events but does not depend at all on the duration of any step. In our Black Jack Dealer, the operator and Dealer must shake hands in the process of requesting and entering a new card. HIT requests a new card; CARD.READY* is the operator's acknowledgment of the request. The desired sequence is:

(1) Dealer senses that the (synchronized) CARD.READY signal is false, and asserts HIT, keeping HIT true at least until CARD.READY goes true.

(2) Operator sees the Hit light on, and (after preparing a new card) presses the Card.Ready button, keeping it on at least until the Hit light goes off.

(3) Dealer detects that the (synchronized) CARD.READY signal is true, and drops HIT (and proceeds to process the new card), keeping HIT false at least until CARD.READY goes false.

(4) Operator sees the Hit light off, and releases the Card.Ready button, keeping it released (and hands off the card switches) at least until the Hit light goes on.

The Single-Pulser Revisited

What is wrong with our preliminary design? The operator's role appears correct (we would, of course, fully describe the rules in the user's operating instructions). The Dealer ASM performs step (1) properly, and most of step (3), but fails to keep HIT false until it detects that the operator's button is released. Dealer is failing to observe the dropping of the operator's hand. We can recast this requirement by saying that Dealer must respond once and only once to each action of the operator. Earlier in this chapter you studied several ways of describing and handling this common phenomenon of "once and only once." Here, let us express the solution yet another way. We incorporate the one-state single-pulser algorithm (Fig. 6-3) directly into the ASM, so that we may add our own specialized conditional outputs to the test branches. You have already seen that we will have a CARD.RDY.SYNC flip-flop in the architecture; to accommodate the single-pulser we will include another flip-flop with output CARD.RDY.DELAYED. Then in the control algorithm, we explicitly test the values of these two flip-flops in order to isolate one and only one recognition of the operator's button push. The handshake signal HIT arises from a conditional output whenever the pushbutton is up. The relevant part of this ASM is in Fig. 6-24.

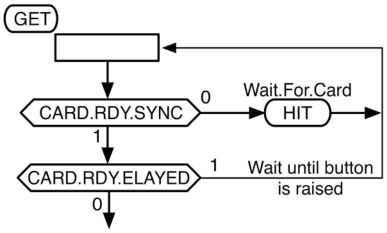


Figure 6-24. A specialized single-pulser for the Black Jack Dealer

This algorithm solves the difficulty of accomplishing the full handshake: HIT will only be asserted in the Wait.For.Card branch of state GET. If the button is up, Dealer is happy to request a new card in state GET. Whenever GET detects that the button is pressed, HIT will go false. The first time GET sees the button pressed, the ASM will process a new card. If control returns to GET while the button is still down, the algorithm simply waits (with HIT still false) until the operator releases the button.

The Final ASM for the Black Jack Dealer

The algorithm for the Black Jack Dealer now seems to be developing nicely. We have found that we may eliminate some of the states in our original proposal without jeopardizing clarity. We have explored in detail the synchronization requirements of certain inputs and the larger handshaking requirements between the operator and Dealer. Figure 6-25 is the improved ASM for Dealer. In this figure, as in Fig. 6-13, the labels on the conditional output boxes describe conditional output terms—logic terms useful in developing a systematic implementation of a complex ASM. The conditional output terms are not state names; they merely represent positions within their parent state. For instance, the circled label .1 on the conditional output in state GET is a shorthand for the label GET.1. During the implementation of the control algorithm, we will derive equations for the logic variable GET.1 as well as each of the other terms.

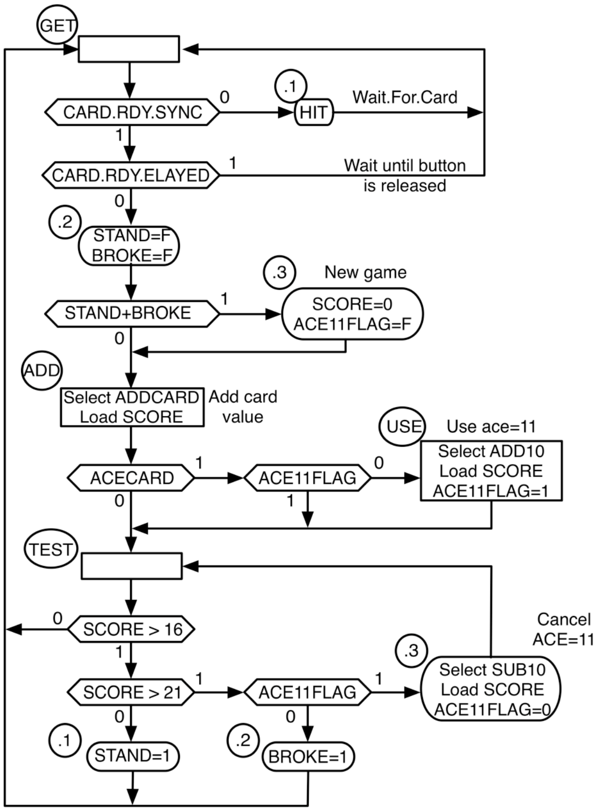


Figure 6-25. The final ASM for the Black Jack Dealer

The Final Architecture of the Black Jack Dealer

Figure 6-26 is the functional architecture of Dealer. Along the way, as we developed a more detailed understanding of the problem, we made certain modifications to the original architecture; these are reflected in the figure.

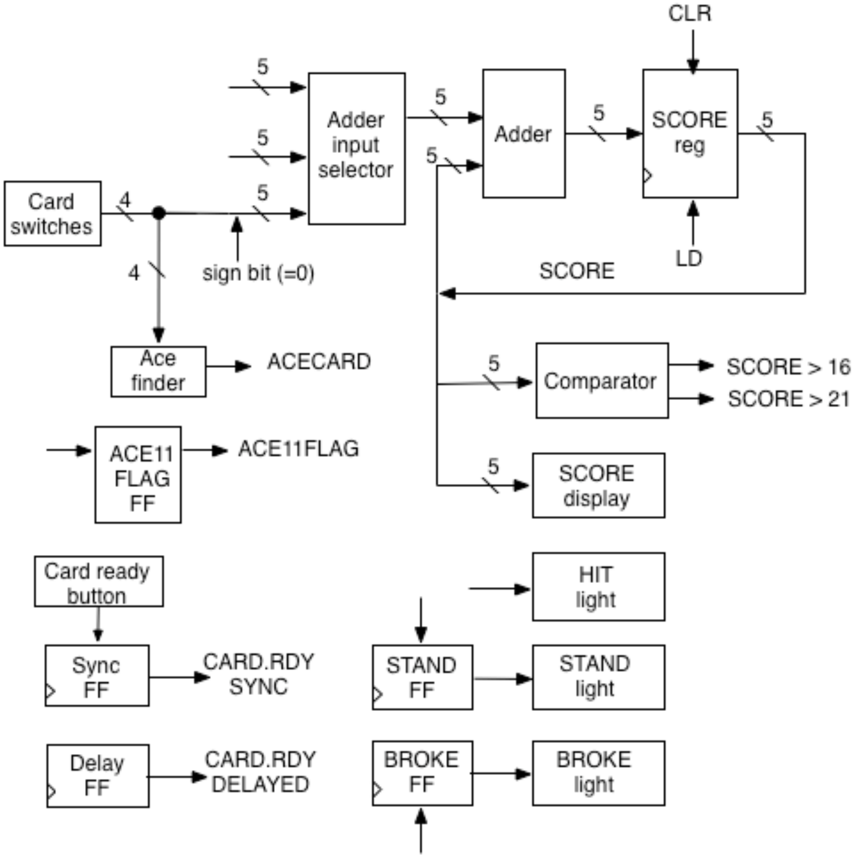


Figure 6-26. The functional architecture of the black jack dealer

The significant modifications or elaborations include the following:

a) Since the operating procedures stipulate that the card switches are stable throughout the time that Dealer processes the card (in other words, as long as HIT is false), we do not need a separate register to hold the current card value; the card switches themselves will suffice.

b) The ASM manages the HIT signal directly, without a flip-flop to preserve its value across states.

c) Our treatment of CARD.READY* introduces two new flip-flops, the components of the single-pulser. We have elaborated on the black box for preparing the input to the SCORE register. There are four operations that modify SCORE: clearing SCORE to 0, adding CARD to SCORE, adding +10 to SCORE, and subtracting +10 (adding -10) from SCORE. A 5-bit adder can add the appropriate value to SCORE if we can select the proper value. You know how to manage selection, so you will probably guess that we will use multiplexers. The important point to realize at this stage is that we can select one input from several, without worrying about the exact library modules. With SCORE as one input to the 5-bit adder and the other input selected by a selector black box, the circuit is nearly specified. A judicious choice of the modules for SCORE should allow us to clear this register separately from the register-load operation. On this basis, the original nebulous black box has separated into three components: an adder building block, a selector building block, and a control input for clearing the SCORE register building block.

Surely now we will sit down and define the exact modules to support the dealer architecture. This would be a reasonable step to take at this time, but as a further illustration of the power of the methods you are learning, let's see how far we can carry the implementation of the control algorithm without specifying the exact hardware in the architecture.

Implementing the Control Algorithm

It is the task of the ASM to generate the necessary commands (outputs) to control the architecture and to provide the outputs to the external world. In Fig. 6-27 we show how the ASM controls the architecture. The inputs and outputs of the ASM are still specified at a somewhat abstract level. We will develop the logic equations for each signal. Such a development depends on our understanding of how to convert building blocks into logic primitives, just as implementing a software flowchart requires knowledge of the programming language. As we have stressed repeatedly, the goal is to think of the problem, not the hardware, for as long as possible.

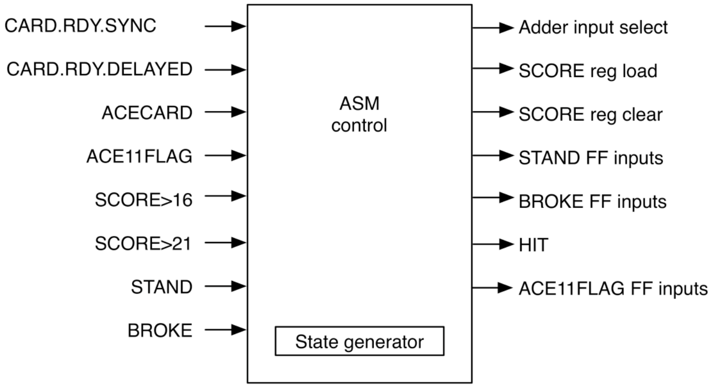


Figure 6-27. The functional control of the Black Jack Dealer

Our design now involves two tasks: Implementing the flow of the ASM (the state generator), and implementing the outputs (commands). First, we will define the conditional output terms, which will be useful parameters for many of the remaining equations. Reading directly from the ASM, we have the following logic equations.

$$\text{GET.1} = \text{GET} \bullet \overline{\text{CARD.RDY.SYNC}}$$

$$\text{GET.2} = \text{GET} \bullet \text{CARD.RDY.DYNC} \bullet \overline{\text{CARD.RDY.DELAYED}}$$

$$\text{GET.3} = \text{GET.2} \bullet (\text{STAND} + \text{BROKE})$$

$$\text{TEST.1} = \text{TEST} \bullet \text{SCORE} > 16 \bullet \overline{\text{SCORE} > 21}$$

$$\begin{aligned}\mathrm{TEST.2} &= \mathrm{TEST} \bullet \mathrm{SCORE} > 16 \bullet \mathrm{SCORE} > 21 \bullet \overline{\mathrm{ACE11FLAG}} \\ &= \mathrm{TEST} \bullet \mathrm{CORE} > 21 \bullet \overline{\mathrm{ACE11FLAG}}\end{aligned}$$

$$\begin{aligned}\mathrm{TEST.3} &= \mathrm{TEST} \bullet \mathrm{SCORE} > 16 \bullet \mathrm{SCORE} > 21 \bullet \mathrm{ACE11FLAG} \\ &= \mathrm{TEST} \bullet \mathrm{SCORE} > 21 \bullet \mathrm{ACE11FLAG}\end{aligned}$$

Next, we implement the state generator. Table 6–3 is a summary of the ASM state transitions.

TABLE 6-3 STATE TRANSITIONS IN THE BLACK JACK DEALER ASM

Present state	Next state	Condition from the ASM
GET	GET	$\overline{\mathrm{GET.2}}$
	ADD	GET.2
ADD	USE	$\mathrm{ACECARD} \bullet \overline{\mathrm{ACE11FLAG}}$
	TEST	$\overline{\mathrm{ACCARD}} + \mathrm{ACE11FLAG}$
USE	TEST	T
TEST	GET	$\overline{\mathrm{TEST.3}}$
	TEST	TEST.3

Notice the handy use of the conditional output terms. For example, the full condition for moving from state GET to state ADD is

$$\mathrm{FROM.GET.TO.ADD} = \mathrm{CARD.RDY.SYNC} \bullet \mathrm{CARD.RDY.DELAYED}$$

But conditional output term GET.2 is just this expression ANDed with GET. Producing the conditional output in the oval at position GET.2 will require that we implement the conditional output term GET.2, so we know that this term is available in the final circuit. Appending GET to the expression for FROM.GET.TO.ADD has no logical effect, since the expression already implies that the ASM is in state GET. It is therefore permissible, and convenient, to use the conditional output terms as required to develop the state generator logic.

As usual, it is convenient to decode the state flip-flop outputs, producing in this case the four state terms GET, ADD, USE, and TEST. These terms complete the requirements of the logic equations developed thus far.

As a last step in the ASM synthesis, we derive the output signals. Reading almost directly from the ASM, we have:

HIT = GET.1
STAND(SET) = TEST.1
STAND(CLR) = GET.2
BROKE(SET) = TEST.2
BROKE(CLR) = GET.2
ACE11FLAG(SET_ = USE
ACE11FLAG(CLR) = GET.3 + TEST.3
SCORE(LD) = ADD + USE + TEST.3
SCORE(CLR = GET.3

Deriving the controls for the adder selector requires one further set of parameters. We need 3 data inputs to the selector: CARD, +10, and −10. Each data path is 5 bits wide, so five 4-input multiplexers will serve nicely as the basis for this selector, each mux being controlled by the same select signals, S1 and S0. We shall assign +10 as the input to the 0-position of the multiplexer −10 to the 1-position, and CARD to the 2-position. Our task is to derive logic equations for the selector controls S1 and S0. With these specifications and the ASM chart, we develop Table 6-4.

TABLE 6-4 ADDER INPUT SELECTION FOR BLACK JACK DEALER

MUX position	Select inputs		Data inputs	ASM notation	ASM condition
	S1	S0			
0	0	0	+10	ADD10	USE
1	0	1	−10	SUB10	TEST.3
2	1	0	CARD	ADDCARD	ADD
3	1	1	—		—

From this tabulation we easily derive the select input equations:

S1 = ADD
S2 = TEST.3

We now have equations for all the control signals, still without a commitment to specific hardware. The last step, hopefully straightforward and bug-free, is to select gates and flip flops, then draft the final circuit diagrams for the architecture, state generator, and output signals. Here is the first point at which voltage enters our design! Using mixed-logic drafting conventions, you can implement the logic equations with whatever gates are handy. JK flip-flops are an obvious choice for the storage elements for those signals requiring controlled setting and clearing: STAND, BROKE, and ACE11FLAG.

This completes the Black Jack Dealer problem. The methodology is important and well worth your close study and emulation using a simulator. The documentation of this design should include most of the figures and tables developed in this exercise that relate to the final design. Think what a help this high-level documentation would be to you if you were presented with a real, nonfunctioning circuit and told to get it running or to modify it in some way.

DESIGN EXAMPLE 6: A GARAGE DOOR OPENER

For our last design example we sketch the logic for a garage door opener and its hardware implementation. Here the human interface is more important than the hardware and we revisit this topic after discussing an implementation biased by our exposure to conventional openers.

Conventional garage door openers have most of their logic concentrated in the motor unit that receives commands by radio from a user keypad. Since we are only interested in the overall logic of the complete system, we will adopt the convenient fiction that all logic is located in the keypad which in turn issues commands over wires to the motor unit. This allows us to abstract off the irrelevant issues of radio communication and concentrate on system logic.

Since the overall architecture of a garage door system is an industry standard, we can violate our dictum of algorithm first and architecture second, and show a high level view of the system, with the proviso that a detailed consideration of the algorithm may force some slight change in our preliminary architecture. In this particular case this is not a disturbing inversion of procedure, but in general you should be wary of thinking about implementation details before algorithm design.

A keypad situated on the building's exterior issues control signals to a motor that raises and lowers the door. Mechanical limit switches sense when the door is fully opened or closed. Keypad logic is an example of a module that is largely architecture oriented and therefore relegated to appendix ***.** For the moment you may assume a standard keyboard interface that reports a row and column address on key depression, with key depression announced by an asynchronous signal, DA*.

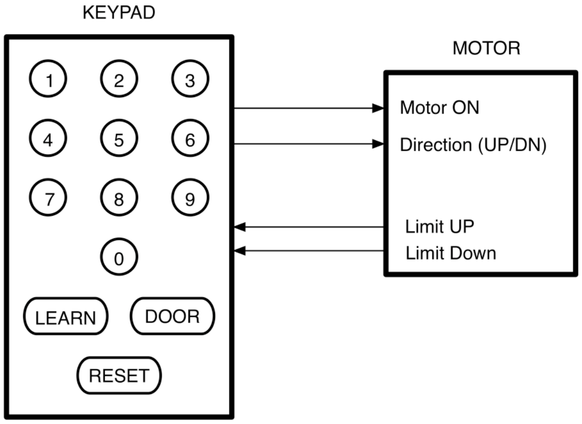


Figure 6-28. An Overview of the Garage Door System

Before even considering algorithm design we must make sure we understand the overall problem. Try to write down all situations the algorithm must address even if some of them are redundant; they are easily dealt with now but critical design omissions are much harder to rectify when you are far down the design path. Typical questions might be:

1) Should there be a master reset, if so, what should it do?

What will the doors state be after installation? Carpenters will do whatever is convenient and will probably find it easier to install the door in the closed state, but can we be sure of that? Maybe it was easier for them to work with the door in the fully retracted state, or perhaps in-between. When the keypad is installed and first powered up it can't assume anything about the door's position. It seems natural that a master reset should be present that will force the ASM into an initial state that is responsible for driving the door into a full down condition and then move into a quiescent ASM state waiting for some input.

2) What's the controller's quiescent state?

   In the quiescent state we expect the user to press 4 digits to initiate a door operation or perhaps initiate a new learn PIN sequence. Provisionally, we might also use this state to clear out old PIN entries

3) How is the keypad interfaced with the ASM?
   a) Digit keys
   b) LEARN, DOOR, and RESET keys

   Referring to figure GD-1, we will assume the digit keys 0-9 are arranged in a 4x3 array and scanned with the keyboard algorithm discussed in appendix *, which reports the row/column indices of a depressed key. Row 00_2 is at the top and contains the "1,2,3" keys, row 11_2 contains the "0" key; column 00_2 contains the "1,4,7" keys and column 10_2 contains the "3,6,9" keys. The asynchronous signal, DA*, announces a key depression and availability of the corresponding 2-bit row code and 2-bit column code. LEARN*, DOOR*, and RESET* will be debounced asynchronous push button signals.

4) A sequence of 4 numeric key depressions that matches an internal stored PIN, followed by the ENTER key, should actuate the door motor; but this leaves some questions unanswered:
   a) How do we get into the door actuate cycle in the first place?

      The ENTER key will actuate the door after valid PIN entry

   b) What happens if fewer than 4 digits are pressed followed by the ENTER key?

      This is obviously an error and should cause a branch to an error handling portion of the ASM

   c) What happens if more than 4 digits are entered followed by the ENTER key?

      We could treat this as an error, or alternately we could just accept the last 4 digits, and if they match the master PIN, treat it as a normal sequence. If they don't match the master PIN, treat it as an error. Let's adopt the second alternative.

   d) What happens if the correct PIN is entered, but left hanging when not followed by the ENTRY key?

      This is obviously an error and should cause a branch to an error handling portion of the ASM

   e) How does the controller know to signal up or down?

Alternate door sequences should reverse direction.

   f) How does the controller know where the door is?
   g) What constitutes an error condition?
   h) What should the system do on an error?

5) The system must be able to learn a new PIN. Many of the same questions need to be addressed
   d) How does the system prevent an unauthorized user from setting his own PIN?
      A user must first match the reference pin before being allowed to advance to the LEARN sequence
   e) What sequence of keystrokes leads to the learn portion of the ASM?
   f) How does the system signal success in learning a new PIN
   g) What constitutes an error condition?
   h) What should the system do on an error?
6) And lastly the "chicken and egg problem". What happens the very first time a user tries to generate a new PIN?
   On installation there must be a pre-assigned PIN otherwise a new user would never be able to store a new private PIN. So the manufacturer must load a standard PIN, say 0123, into the PIN EEPROM memory, with instructions to a new user to immediately change it to a new personal PIN.

We invite you look at your own door opener and see how it address these questions as well as looking for missing conditions that must be incorporated into your ASM. For now, we will assume this is a complete list. As a preliminary step, let's formulate the problem as a standard flow chart. This gives us a good overview of the complete problem and results in sub task partitions, something that comes naturally to good algorithm designers.

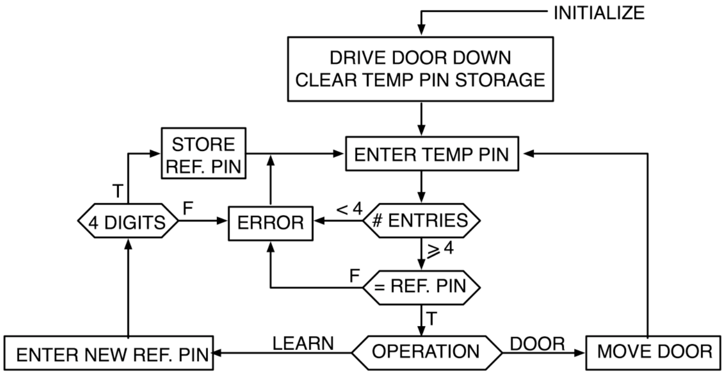


Figure 6-29. A high level view of the garage door algorithm, (not an ASM).

We can now begin to expand these high level operations into architecture and algorithm, (ASM states). Even before casting the algorithm into an ASM the software flowchart points to probable architectural elements. The reference PIN must maintain its value until changed by a new LEARN PIN sequence; EEPROM is a natural candidate for reference pin storage.

Each operation requires entry of a temporary PIN to validate a user and once compared against the reference pin it can be discarded. A number of possibilities fit these requirements: flip flops, RAM, or a shift register. At this stage our algorithm is insensitive to our choice but we know that it must be temporary storage, capable of holding 4, 4-bit numbers, and be able to report when less than 4 numbers have been entered. Even at this early stage our problem formulation points to an architecture to handle this problem. Two solutions come to mind: a 4x4 RAM or a 4x4 shift register. Since we have chosen to accept only the last 4 entries of an entry sequence the shift register solution seems natural since a 4-bit register will only retain the last 4. The addition of a 5th bit is a clever way to flag the case of less than 4 entries as shown in Fig. 6-30. If we first clear all register bits, a "0" will appear at the lower right "Q" terminal if less than 4 keypad codes have been entered. Note that we came to this solution *after* considering the algorithm, not before.

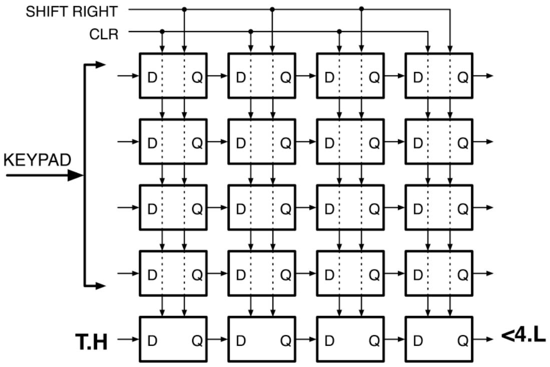


Figure 6-30. Temporary PIN storage architecture

Lets start by considering ASM's for likely fragments of our final algorithm, starting with the initialize fragment:

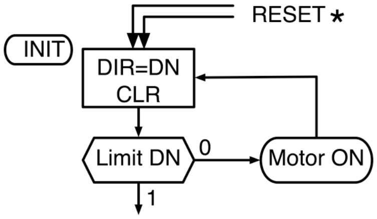


Figure 6-31. Initialize ASM

We should shift in a new KEY code for every assertion of DA*, making sure to synchronize DA* and run it through a single-pulser to give DA(SP), something you should automatically do when processing slow manual signals on a fast state machine. Entries will happen repeatedly until terminated by LEARN* or DOOR* signals, which should also be synchronized and single-pulsed.

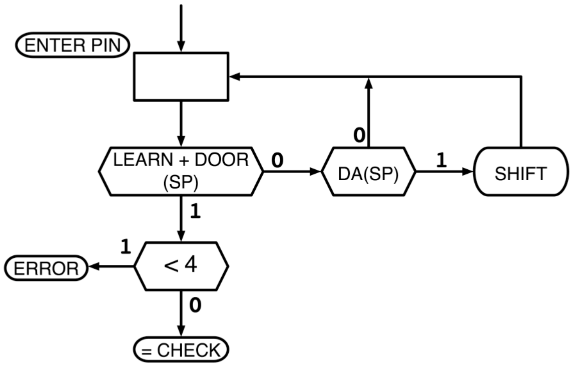


Figure 6-32. Enter New Pin ASM

Equality checking can be done “in line” with 4 states or in one state with an auxiliary loop counter. We leave the loop version as an exercise and present the in line version.

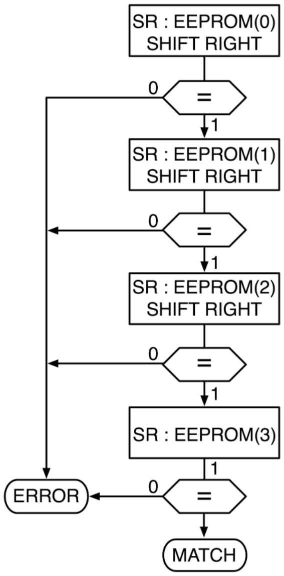
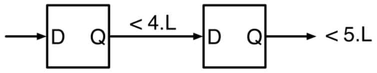


Figure 6-33. Valid PIN ASM

Looking at the LEARN portion of Figure 6-35 we see that the architecture of 6-30 will also serve for entry of a new PIN for storage in the reference EEPROM, with one minor difference; we must now check for exactly 4 entries. Adding a 1-bit shift register to the 5th row detects this condition.



Exactly 4 entries $= \overline{(<4)}\bullet(<5)$

Figure 6-34. Modified 6-30 To detect 4 entry condition

We now have a good feel for partial ASM components so lets try to put them together into a provisional, complete, ASM. This will give us the opportunity to view the problem as a whole; hopefully there will be no flaws. A possible final ASM is shown in Figure 6-35.

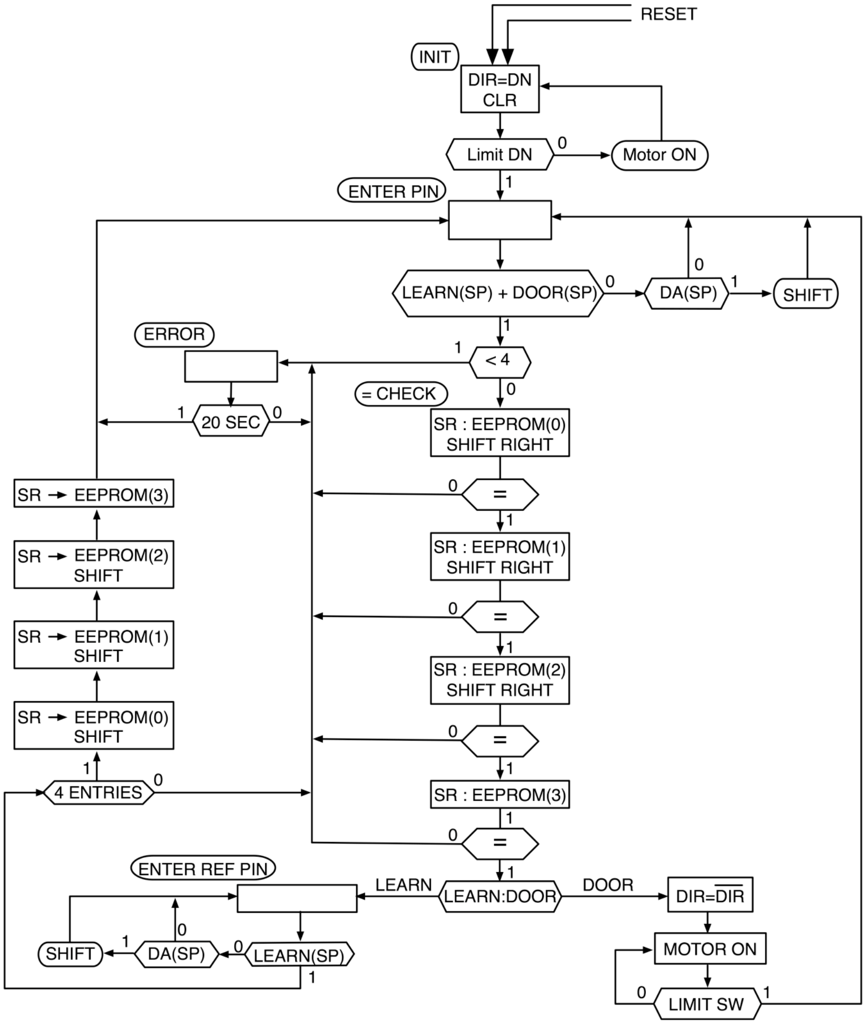


Figure 6-35. Garage Door ASM (with an error)

Unfortunately, we have been premature and consideration of the ASM reveals a subtle flaw. The algorithm for entering keystrokes into the temporary PIN shift register is straightforward. Enter data, *once*, on every assertion of DA*, this warns us that we must single pulse DA* to give DA(SP), well and good; the entry process is terminated by depression of either the LEARN* or DOOR* keys and again they must be single pulsed, and the ASM correctly handles that. We now have a PIN loaded and must check it for a match to the reference PIN, again, this is correctly handled by the ASM.

After verifying that we have matched our entered PIN against the reference PIN we must either do a DOOR cycle or a LEARN cycle, but both DOOR(SP), and LEARN(SP) have gone away! (This assumes we are using our standard single pulser module that outputs a one-clock cycle pulse at the beginning of a key depression).

We no longer know which way to go. Ahh, but our finger is much slower than our ASM clock so DOOR* is still asserted and we could safely use a synchronized version, DOOR, to at least tell us to actuate a DOOR cycle as follows:

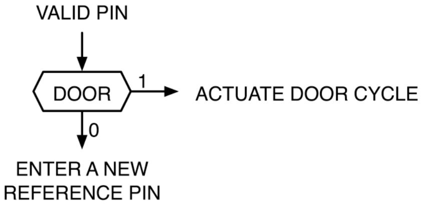


Figure 6-36. A fix(?) to the Garage Door ASM

Do you have a bad taste in your mouth? *You should!* Why? We are using hardware to rescue a bad ASM, even worse, it's an *obscure* hardware trick. Any logical difficulties should be resolved at the algorithm level where they are exposed for all to see. We are faced with a standard problem: recovering knowledge of a past event. Every algorithm designer has a standard approach to this problem: set a flag, and later query the flag. Of course, we must also properly handle that flag during initialization.

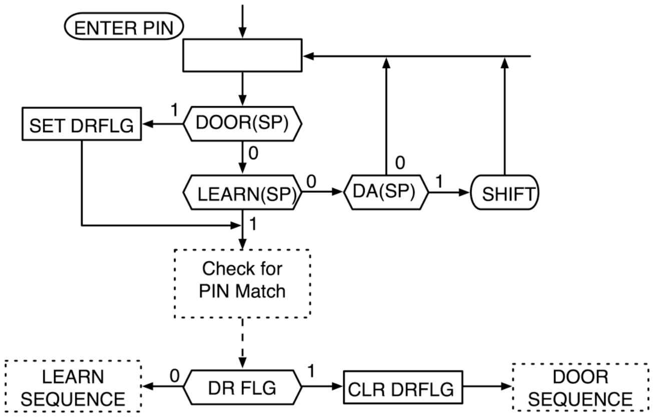


Figure 6-37. Properly handling prior ASM events.

The Garage Door Opener Revisited.

This presentation was inevitably biased by what we already know about garage door openers, and more subtly by our own technophile's viewpoint. If you can, try to put yourself in the position of introducing the very first opener to the waiting world. You must now look at the entire problem from a *technophobe's* viewpoint, usually a difficult thing for an engineer. A good way to proceed would be to write a simulation, in a high level language, that portrays the keypad on a touch sensitive CRT with a visual door image on the same screen. Then invite a selection of technophobes to play with the simulation until you find a keypad layout and key sequence that is as intuitive as possible. No doubt you would find that our keypad layout and algorithm of Figures 6-28 and 6-29 would be slightly altered; technophobes don't think like engineers! That's not important; engineers will *not* be your customer base.

Think about you VCR remote and you will see that many designs simply bypassed this obvious first step. Hardware can't be elegant unless the human interface is intuitive!

READINGS AND SOURCES

EXERCISES

6-1. Show that any ASM may be expressed as an ASM with only one state. Why do we not do away with state generators by always designing with single-state ASMs? Discuss the advantages and disadvantages of this approach.

6-2. Although we find that we must debounce manual control switches, we usually do not need to debounce manual data entry switches. Why?

6-3. Build a four-state ASM that emits one of two BCD number sequences, depending on the value of a control variable NINESCOMP. Each sequence has a cycle of four digits:

When NINESCOMP = F, the sequence is 0, 1, 2, 3, 0, 1, ...

When NINESCOMP = T, the sequence is 9, 8, 7, 6, 9, 8, ...

6-4. Exercises 1–36 and 2–32 deal with the seven-segment numeric display. Assume that the display integrated circuit requires discrete signals for each segment *a* through *g*. Build a four-state ASM to repeatedly display the first four prime numbers in proper sequence: 2, 3, 5, 7, 2, 3, 5, ...

6-5. Using a 60-Hz periodic logic signal, produce a signal that can serve as a 1-Hz clock.

6-6. If your simulator has 7-segment modules, design the two-digit "second" display of a digital clock. The clock must cycle continuously from 00 through 59, except when a signal from a pushbutton forces the display to 00. Assume that a 1-Hz synchronous clock signal is available.

6-7. Extend the previous exercise to produce a full 24-hour clock display. How will you set the correct time? *(Hint:* Commercial digital clock units often provide a speed-up mode, in which the displayed time goes through a complete cycle in less than 1 minute.)

6-8. Add an alarm feature to the digital clock.

6-9. Design an implementation of Fig. 6–1 using the formal one-hot method with no simplifications. Show how this implementation may be rigorously transformed into the circuit of Fig. 6–2.

6-10. Add a blinking-light feature to the traffic-light controller in

Design Example 5. Assume that a new *BLINK* signal is available and that, when *BLINK is* asserted, the highway lights blink yellow and the farm-road lights blink red.

6-11. Finish the detailed design of the garage in Design Example 6. Produce circuit diagrams suitable for actual construction of the door's electronics.

6-12. Complete the detailed design of the Black Jack Dealer of Design Example 5. Produce circuit diagrams suitable for construction of the electronics.

6-13. For the Black Jack Dealer, the initial ASM in Fig. 6–26 required a flip-flop for the output *HIT*. Show how the further development of the control algorithm led to the elimination of the *HIT* flip-flop from the architecture. This is a typical example of the algorithm modifying the architecture.

6-14. In Fig. 6–22, state variables C, B, and A are all involved in transition races. Why is state variable *D* not similarly involved?

6-15. Binary patterns that differ in exactly 1 bit are said to be *a unit distance* apart. Consider an ASM state (the "predecessor") that has branch paths to several "successor" states. (Note that one possible successor is the predecessor state itself.

a) If all successor states have encoded state assignments at most a unit distance from the predecessor, show that no transition races will arise, even if asynchronous test inputs are present in the predecessor state.

b) Show that the condition in part (a) is not sufficient to preclude output races.

6-16. In the Black Jack Dealer, the architecture contains black boxes that produce such signals as ACECARD, SCOREGTI6, and SCOREGT21. For example, see Fig. 6-26. These black boxes involve only simple combinational circuits. Why do we choose to use the black boxes during the design process, instead of showing the circuits directly?

6-17. The ASM for the Black Jack Dealer (Fig. 6–25) tests several signals (ACECARD, SCOREGTI6, SCOREGT21) that are generated by combinational logic within architectural black boxes. Since these black boxes are not clocked, how do we know that their outputs are synchronous and therefore suitable for testing in the ASM?

6-18. Define carefully the use of a full handshake to synchronize two independent processes. Show how the need for process synchronization arises whenever a human operator interacts with a machine. Illustrate these concepts with the Black Jack Dealer.

6-19. To synchronize events in two cooperating but independent processes, designers have used a variety of techniques, many of which we may describe as "incomplete handshakes." For instance, consider the following incomplete handshake:

a) Device A requests an action (device A extends its hand).

b) Device B senses the request and acknowledges receipt of the request (device B extends its hand).

c) Device A senses device B's acknowledgment and drops its request (device A drops its hand).

d) Device B drops its acknowledgment at any time after asserting its acknowledgment (device B drops its hand).

This looks like a "complete" handshake, but there are circumstances in which the handshake may be incomplete. Draw timing diagrams for the possible behaviors of the request and acknowledge signals. Discuss the effectiveness of the above protocol for the following conditions:

i) Device A is a machine; device B is a human being.

ii) Device A is a human being; device B is a machine.

iii) Both devices are machines.

iv) Both devices are human beings.

6-20. Implement a one-hot state generator for the Black Jack Dealer.

6-21. In the Black Jack Dealer example, the signals SCOREGT16 and SCOREGT21 arise from a comparator architectural element. Write logic equations for these two variables. Show by Boolean algebraic manipulation that the term SCOREGT16•SCOREGT21 reduces to SCOREGT2I, thus rigorously demonstrating the validity of the simplifications of TEST.2 and TEST.3 performed "by observation" in the text.

6-22. Design a synchronous digital circuit with the following properties:

Inputs:

a) Two 4-bit binary numbers, *A* and *B,* in signed magnitude notation (1 sign bit, 3 magnitude bits).

b) A GO signal from a manual pushbutton.

Outputs:

a) A 4-bit binary number C in signed magnitude notation.

b) A signal EVEN for a display lamp.

Task:

a) Wait for the GO signal to be asserted.

b) When GO appears, clear the signal EVEN to false, and load the values on the input lines A and B into two 4-bit registers RA and RB. [Hereafter, (RA) means "contents of RA," etc.]

c) Then, produce an output C in register RC, as follows:

If (RA) > (RB), then transfer the quotient of (RA)/2 to RC.

If (RA) ≤ (RB), then transfer (RB) to RC.

d) If (RA) > (RB) and the remainder of (RA)/2 is 0, then assert EVEN;

otherwise, EVEN remains false.

e) Return to step (a) to await another GO signal.

6-23. Construct an ASM that will turn on a light as the first person enters a room, and turn off the light as the last person leaves. Assume that there is a single door fitted with two photocells that generate suitable voltage outputs. One photocell is on the inner side of the door and the other is on the outer side. Light beams shine on each photocell, producing a false output from the cell; a true output from a photocell arises when the light beam is interrupted. Assume that once a person starts through the door, the process is completed, and that only one person enters or leaves at a time.

6-24. Design a versatile timer circuit. The circuit has two input codes:

a) A 3-bit code describing the unit of counting: code 0 = 100 nsec, code 1 = 1 μsec, code 2 = 10 μsec, ... , code 7 = 1 sec.

b) An 8-bit code describing the number of counts.

Asserting an input signal START will cause the timer to begin counting the specified number of counts, each count being of the specified duration. The only output from the timer is a signal TIMESUP, which becomes false when timing begins and becomes true when the specified interval has elapsed. The timer can time an interval from 100 nsec to 255 sec. Use a 100-nsec clock to drive the timer. Use good synchronous design techniques in determining how to clock and advance the decade counters.

6-25. The *stack is* a software data structure often implemented in hardware. A stack is an ordered set of elements analogous to a stack of plates in a cafeteria. Only the top element (plate) is accessible. Removal of the top element exposes the next-to-top element, which then becomes the top. Addition of an element to the stack causes the former top element to become next-to-top; the added element becomes the top. The operation of adding an element to a stack is called *push*. The removal operation is called *pop*. These are the only allowed stack operations. A stack is sometimes called a *LIFO* (last in, first out) memory. In hardware, we may implement a stack in RAM or with an array of discrete registers.

a) Using RAM, design a stack that accepts push and pop operations and properly adds or removes an element from the top of the stack.

b) Design a small five-element stack using registers.

6-26. Repeat Exercise 6–25 for a stack that also provides two status signals:

EMPTY: Asserted when the stack contains no elements. Your stack should ignore a command to pop an empty stack.

FULL: Asserted when the stack contains a predetermined maximum number of elements. Your stack should ignore an attempt to push a full stack.

6-27. The *queue is* a software data structure that is sometimes implemented in hardware. The queue has a front and a rear, like a line for tickets at a theater. *A write* operation adds an element to the rear of the queue; a *read* operation removes the element at the front of the queue. No other operations are allowed. The queue is also called *a FIFO* (first in, first out) memory. A queue has two status indicators:

EMPTY: Asserted when the queue contains no elements. The circuit should ignore an attempt to read an empty queue.

FULL: Asserted when the last available memory location is occupied with a queue element. The circuit should ignore an attempt to write into a full queue.

(a) One approach to implementing a queue in RAM is to maintain two pointers *FRONT* and *REAR* as RAM addresses to the extremities of the queue. *WRITE* increments *REAR* and adds an element to the rear of the queue; *READ* extracts the front queue element and increments *FRONT. FULL* becomes true when *REAR* points to the highest memory location in the RAM. Design such a queue. You may alter the foregoing suggestions as long as you still implement a queue.

Why is this project more difficult than designing the stack of Exercises 6–25 and 6–26? In this implementation, when *FULL* is true, is all of the memory filled with queue elements? If *EMPTY* is true, what should be the values of *FRONT* and *REAR?*

(b) Another approach to a RAM implementation of a queue is similar to that in part (a) but allows the queue to go "around the corner," so that *REAR* and *FRONT* may advance from the highest memory address to address zero. Design such a queue. When does *FULL* become true?

6-28. Design a controller for an elevator in a six-story building. Your controller must respond to call switches on each floor and floor-select switches within the car.

6-29. Design a four-way traffic-light controller that will keep traffic moving efficiently along two busy streets that intersect. In this exercise, consider only straight-through traffic.

6-30. Extend Exercise 6–29 to include left-turn signals at each approach to the intersection.

6-31. Extend Exercise 6–29 to include pedestrian crosswalk signals.