5

Design Methods

In Part I we presented basic design tools and introduced components used to build digital systems at the modular level of complexity. The fascinating part of digital design lies before us. We now consider how we may assemble a complete system from building blocks. It is in this area that the designer can create elegance and beauty, or chaos and headaches.

To some extent, digital design is an art form. Most designers have had to develop a style of digital design by trial and error, and their efforts often have not converged to an efficient and aesthetic style. On the other hand, there are underlying principles that can be immensely useful to designers. Our goal in this book is to start you down the right path, and to take you far enough so that you can develop your own designs with a solid sense of good style.

Design style is a curiously neglected subject, perhaps because a traditional study of logic design emphasizes the microscopic transistor and gate aspects of the subject. We emphasize a macroscopic view of digital systems by starting from the original problem. The result is a top-down approach to design.

ELEMENTS OF DESIGN STYLE

Here are the guidelines for good design style:

(a) Design from the top down.
(b) Maintain a clear distinction between the controller and the controlled hardware (the architecture).
(c) Develop a clearly defined architecture and control algorithm before making detailed decisions about hardware.

Top-Down Design

A design starts with a careful study of the overall problem. At this stage, we deliberately ignore details and ask such questions as:

(a) Is the problem clearly stated?
(b) Could we restate the problem more clearly or more simply?
(c) If we are working with a subsystem of a larger system, what is its relationship to its host? Would a different partitioning of the entire system yield a simpler structure?

At this stage our concerns are global, and we must stay at that level until we have hammered out a sensible statement of the problem and have digested the problem to the point where we understand what we must solve. This is essential, since any difficulties at this level are serious. No amount of wizardry with components can remedy errors in the understanding of the problem.

After we have clearly specified the problem at the global level, we seek a rational way to partition the problem into smaller pieces with clearly defined interrelationships. Our goal is to choose "natural" pieces in such a way that we can comprehend each piece as a unit and understand the interaction of the units. This partitioning process proceeds to lower levels until finally we choose the actual library modules.

Unfortunately, many designers reverse this process by rushing to circuit data libraries to find the unit that will "solve" their problem. Often they find a module that solves a slightly different problem. Thus enters the infamous "patch" to force the problem, which is itself not well defined in the designer's mind, to the library part. This process proceeds from the bottom up, often in a divergent manner. Many commercial designs bear unmistakable traces of this method of design.

Separation of Controller and Architecture

One of the first steps in a top-down design is to partition the design into (a) a *control algorithm* and (b) an *architecture* that will be controlled by this algorithm. The top-down analysis will suggest a rough preliminary version of the system's architecture, involving abstract building blocks such as registers, memories, and data paths. Since the architecture is specific to the particular design, there is no general prescription for writing down this preliminary architecture. The main guidelines are to make the architecture natural to the problem and to design with high-level units rather than with modules and voltages. The examples of design in Chapter 6 will illustrate the art of specifying the rough architecture.

Next, we work out the details of the control algorithm at an *abstract level.* The control algorithm is often surprisingly independent of hardware. For example, if you were designing a computer, what operations would you expect your control algorithm to accomplish?

(a) Get the next instruction.

(b) Test to see if operands are needed and get the operands if required, making any necessary indirect memory references to indirectly addressed operands.

(c) Execute the individual instruction.

As you will see in Chapter 6, we may specify a complete flowchart (algorithm) for operations like this with almost no knowledge of the specific hardware.

You should explore the construction of the control algorithm until you have a clear understanding of your approach to the solution. The exploration may go through several iterations, but eventually you will complete the process, at which time you should turn your attention to the hardware for the architecture that is suggested by the control algorithm. The algorithm will guide you to the hardware. Note how powerful this concept is. We have a tool that allows us to solve a problem in a rational way instead of randomly looking at modules and wondering if they will fit into the design.

We can formalize the controller-architecture separation with the diagram in Fig. 5-1. The controller issues properly sequenced commands to the controlled device. These commands make the architecture perform the actions dictated by the control algorithm. Usually, the controller will need status information from the architecture that serves as decision variables for the control algorithm. As the design matures, the controller's command outputs and status inputs go from abstract concepts of control to Boolean variables, and finally to voltage representations of the Boolean variables.

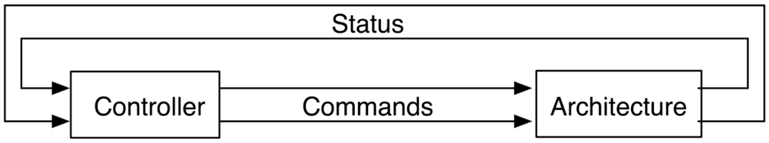


Figure 5-1. The structure of a state machine

Consider the following example. A problem requires that a word be written into a memory. The preliminary architecture for the problem is just a black box for the memory, the details of its inner construction being deferred until later. The memory will require four items of input: a memory address MA to tell where to write the data, a word of DATA for input, a line R/W to tell whether to read or write, and a GO signal to start the read or write operation. The only status returned by the memory will be memory cycle complete CC.

Figure 5-2 is the functional diagram corresponding to this analysis. The command lines are:

MA (n lines)
DATA (m lines)
R/ W (1 line)
GO (1 line)

The parameters n and m are determined by the characteristics of the memory needed by the problem. For instance, a 1,024-word by 8-bit memory would have $n = 10$ ($1,024 = 2^{10}$) and $m = 8$ (each word of memory has eight bits). The status line is CC.

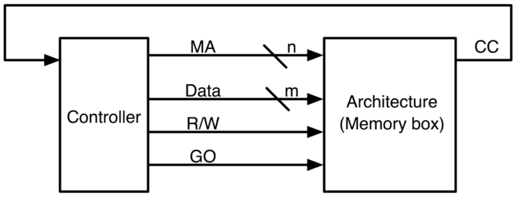


Figure 5-2. A diagram of a memory write machine

We now have a good idea of what signals the controller and architecture must generate and accept, even before we know what hardware we will use to build the controller or what the memory box is like inside. We choose the memory command lines by realizing that we must have data to write into memory and a location where it must be written. The structure in Fig. 5-2 is not affected by the actual type of memory.

We can say something about the nature of the control algorithm that initiates a memory write operation, without knowing exactly how we will translate that algorithm into hardware. The algorithm must look something like Fig. 5-3. The purpose of the first step **STW** is to issue a GO signal to the memory, along with the necessary data and commands to initiate a writing operation. The next step **WAIT**s until the memory has finished the writing.

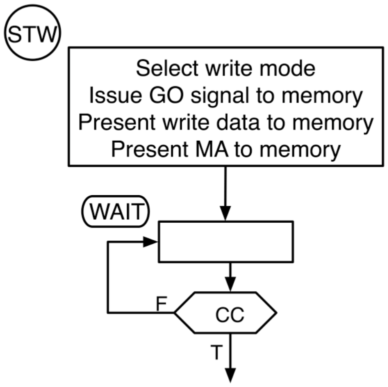


Figure 5-3. Algorithm for writing to a memory

We can accomplish an amazing amount of the design by a general consideration of the problem. Carry the top-down analysis as far as you can, because decisions at this level are more easily altered than they will be once the hardware has intruded.

Refining the Architecture and Control Algorithm

We are now ready to move down one level. We have sketched out the algorithm by ignoring the hardware as much as possible; instead we were trying to reduce our problem to high-level, abstract statements of control and architecture. The only consideration about hardware introduced thus far were general ones that we could state from a knowledge of the variables required for the operation of abstract building blocks such as memories, registers, and arithmetic units. We now begin to refine the control algorithm by introducing more detail.

As we refine the algorithm, we need a more detailed knowledge of the architecture of the system. We therefore begin to elaborate the architecture as a set of building blocks, moving carefully through a set of high-level building blocks toward a selection of the major hardware. We choose the architectural elements by asking what specific building blocks the developing control algorithm requires; we do not select elements by looking in a library and saying, "Hey, this is a neat module—I must fit it into my next design”.

A good architecture will be simple, clear, and easy to control. If it is not, there is no way to rescue the resulting mess with exotic circuits or Boolean algebra. If the architecture is clear and simple, the rest of the design will be relatively straightforward. This step—the first introduction of hardware—is an important point in the design.

After we choose the major building blocks, we know what control signals they will require. At this point, we tabulate these signals and then quit worrying about the hardware. In fact, we suppress consideration of the hardware lest it capture our thought processes and cause us to lose sight of the algorithm we are trying to develop. Let the algorithm drive the design process as much as possible. Now we can continue with an elaboration of the algorithm, whose function is to provide a properly sequenced set of commands to the architecture. Spend a large fraction of the total time for the project on the detailed algorithm-architecture phase.

After we have completely specified the algorithm, we can reconsider the architecture and our choice of building blocks. The detailed construction of the algorithm will usually reveal areas of the design that we can simplify or speed up by using slightly different architectural components. We incorporate these changes into the architecture and make the corresponding changes in the algorithm. At this point the process should have converged to a final solution. We should have:

(a) *The architecture.* This should include a detailed set of components and data paths for the controlled device—usually a specification of the actual library modules for the major components such as registers, ALUs, and memories, and a statement of the command signals that these components require and the status signals that they produce. Our specification of the architecture does not include any logic required to generate these commands, since the generation of commands is assigned to the control algorithm.

(b) *The algorithm.* This will produce a properly sequenced set of command signals to make the architecture perform the original problem. It does not include the hardware to implement the algorithm. We can derive the hardware from the algorithm in a straightforward and mechanical way, as you will see later in this chapter.

If by this time the process has not converged to a stable solution, you probably had trouble at an earlier stage of the design process. In such circumstances, proceeding further is fruitless; you should go back to the beginning and start over. Do not "kludge" your solution. You have not yet burned any fuses or drawn any hardware circuit diagrams, so beginning anew is relatively painless.

And now, a secret: whether in hardware or software, no one designs a system strictly from the top down. A knowledge of low-level components and techniques always influences the design, even at the highest levels. The best top-down hardware designers have an intimate knowledge of hardware, and this knowledge tempers and guides the high-level design decisions. As their understanding of the design expands, good designers use their knowledge of lower level technology to avoid unproductive approaches. The designer dips repeatedly into lower and more detailed levels for short excursions, but invariably returns to the present top level. The top-down approach has the great virtue of providing the discipline that keeps one thinking at the highest useful level. We like to imagine that a complex design proceeds linearly from the top to the bottom, but that is rarely so. But whenever you dip down, your top-down training will pull you back up as soon as possible.

ALGORITHMIC STATE MACHINES

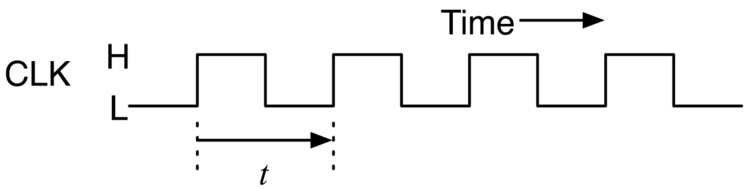
The control algorithm plays a major role in a digital design, so we need a good notation for expressing hardware algorithms. The notation should assist the designer in expressing the abstract algorithm and should support the conversion of the algorithm into hardware. There are several ways of describing the control. For synchronous circuits, the ASM chart technique is the superior notation. ASM stands for *algorithmic state machine.** The name is appropriate, since all controllers are state machines, and we are trying to translate algorithms into controllers. The ASM chart is a flowchart whose notations superficially bear a strong resemblance to the conventional software flowchart. The ASM chart expresses the concept of a sequence of *time intervals* in a precise way, whereas the software flowchart describes only the sequence of events and not their duration.

* T. E. Osborne developed the ASM chart notation and C. R. Clare described the method in his book, *Designing Logic Systems Using State Machines* (New York: McGraw-Hill Book Co., 1973

ASM Chart Notations

States and Clocks

An algorithmic state machine moves through a sequence of *states,* based on the position in the control algorithm (the state) and the values of relevant status variables. The concept of a state implies sufficient knowledge of present and past conditions to determine future behavior. It is the task of the *present state* of the system to produce any required output signals and to use appropriate input information to move at the proper time to the *next state.* In most of this book we are dealing with synchronous systems whose state times are determined solely by a master clock. The most convenient form of clock is a periodic square-wave voltage:



The clock event that triggers the transition from one state to another and other actions of the system is called the *active edge,* and in synchronous systems is usually the rising (L→H) edge of the clock. In a synchronous system, the clock will thus have T = H. Commonly, the clock period t ranges from sub nanoseconds to several microseconds. The frequency *f* of the clock is its number of oscillations per second. Frequency and period are related by the expression

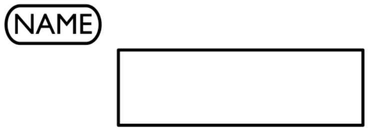
$$f = \frac{1}{t}$$

The unit of frequency is the Hertz (Hz), which is defined as one oscillation per second. Convenient units are:

kilohertz (KHz = 10^3 Hz)
megahertz (MHz = 10^6 Hz)
gigahertz (GHz = 10^9 Hz).

The clock's frequency may vary, and the clock may even stop—desirable during the debugging phases of a design. Within reasonable limits, the duration of the high portion of the clock waveform in relation to the total clock period (the *duty cycle)* is unimportant. The crispness and reliability of the active clock edge is of extreme importance, and designers of systems pay close attention to the production and distribution of an excellent clock signal.

States. Each active transition of the clock causes a change of state from the present state to the next state. The ASM chart describes the control algorithm in such a way that, given the present state, the next state is determined unambiguously for any values of the input variables. The symbol for a state is a rectangle with its symbolic name enclosed in a small circle or oval at the upper corner:



We would represent a purely sequential algorithm as an ASM chart of a sequence of states, as in Fig. 5-4, which also shows the corresponding division of the time axis:

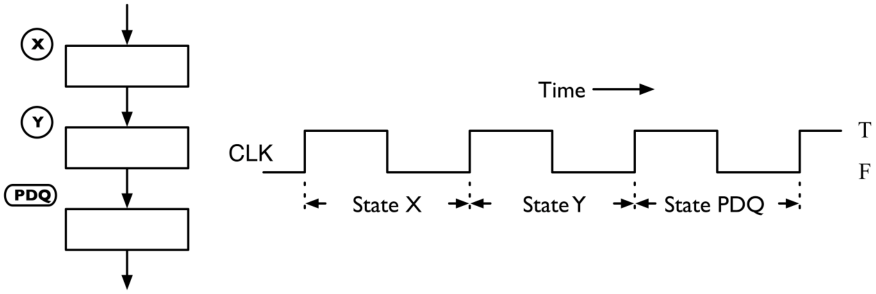


Figure 5-4. A purely sequential ASM and the corresponding time axis

A sequence is an inherent property of an ASM chart; state **Y** follows state **X**, and so on. It is cumbersome to show time relations with a timing diagram such as the above; therefore, you should learn to think of time as rigorously implied in the ASM chart notation.

Outputs. The function of a controller is to send properly sequenced outputs (voltage command signals) to the controlled device according to some algorithm. To indicate an output, we place the command description within the appropriate state rectangle. In this book we use several notations for outputs. Depending on our depth of understanding of our design and on the level of detail we wish to convey, we may use informal expressions of actions or detailed statements of particular output operations. Figure 5-5 contains some examples. In state **PRINT.LINE**, the expression "Start print cycle" represents a set of actions, as yet not fully elaborated, that initiates a printer cycle. The arrows in the next two lines imply actions that are to be consummated at the *end* of this state; at that time LINE is to be loaded into PRINTBUF, and the AC register is to be cleared. The fourth line, MOVING, calls for the assertion of the signal MOVING (making MOVING true) *during* this state. The last line means that the output variable STATUS is to have the value of the variable ERRFLAG (T or F) during this state. Other output notations may be useful; improvising notations is fine as long as you define your terminology.

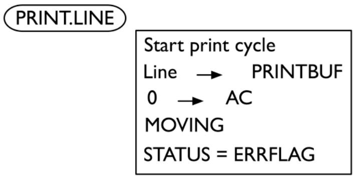
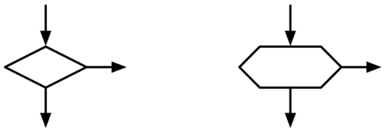


Figure 5-5. ASM output notations

Branches. Purely sequential ASMs are of little interest because they are usually not powerful enough to describe useful algorithms. We need some way to express *conditional branches* so that the next state is determined not only by the present state but also by the present value of one or more test (status) inputs. Our symbol is the same as in conventional flowcharts for software: the diamond or diamond-sided rectangle.



We incorporate this symbol into the ASM chart by appending it to a state rectangle, placing the description of the test input inside the diamond, as in Fig. 5-6.

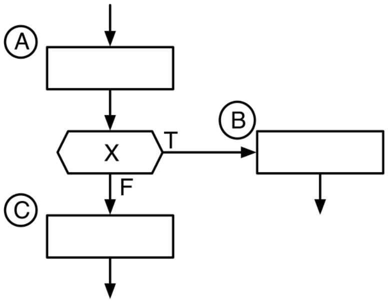
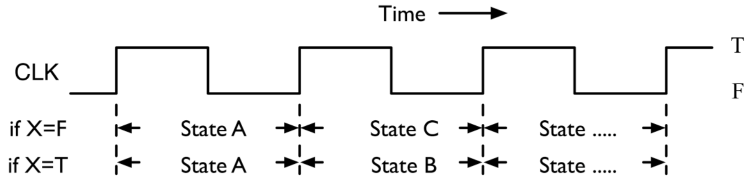


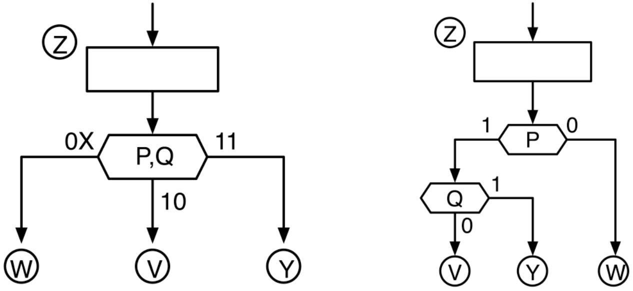
Figure 5-6. An ASM with a conditional branch

In this case a portrayal of the time line would be



The decision to jump to one of the two states **B** or **C** is made *during* state **A**, and the jump occurs at the *end* of state **A**. In hardware implementations the voltage representing input X must be stable for some time before the decision. Ideally, X should be stable for the entire clock period of state A, for then the hardware that decides to jump to **B** or **C** has the maximum time to settle. It is important to realize that a test does not require a separate clock period—it is done "in parallel" with the actions of the parent state rectangle and thus is part of the parent state.

We are not limited to two-way branches from a state. We may draw sequences of test diamonds or we may have more than two paths coming from the same diamond. Two ways of representing a three-way branch are shown in Fig. 5-7. The test structure in Fig. 5-7a is a diagrammatic representation of a truth table in which neither P nor Q appears to dominate. Figure 5-7b conveys the feeling that the test of variable P is of higher priority than the test of Q. Which form is preferable depends on the designer's thought process. Use the ASM notations that best describe your design.



(a) (b)

Figure 5-7. Alternative representations of a three-way branch

Conditional outputs. A command written within a state rectangle indicates that the controller is to produce the output whenever the algorithm is in that state. Sometimes we want a command to occur only when some other condition also exists. We call such a command a conditional output and specify it within an oval, as in Fig. 5-8. Command CMD1 will appear for one state time whenever the ASM is in state **P.** The command CMD2 will occur during one state time whenever the ASM is in state **Q,** but when the ASM is in state **P,** CMD2 will occur only if test input Z is false. In this example, CMD2 is an unconditional output in state **Q** and a conditional output in state **P.** *

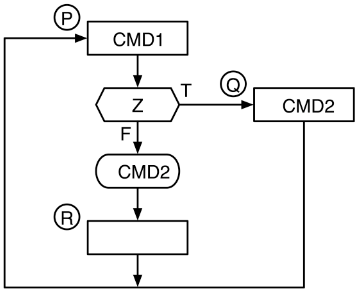


Figure 5-8. An ASM with a conditional output

Test inputs may serve two functions in ASM charts: they may help specify the next state and they may control the issuing of conditional outputs. Ovals for conditional outputs and diamonds for tests of inputs belong to the parent state, since the activities will occur during the same state time. A state thus consists of its rectangle, which is always present, and any test diamonds and conditional output ovals associated with that state. Unconditional outputs are a function only of the parent state; conditional outputs depend on both the state and the path within that state. It would be appropriate to draw a dashed line around the entire structure for each state, but we usually do not do this because the chart defines each state without this aid.

* An ASM with only unconditional outputs is equivalent to the Moore machine of traditional sequential circuit theory; the traditional Mealy machine has conditional outputs. The ASM formulation subsumes both traditional cases.

This is the entire ASM chart notation. Our goal is to use it to help us build digital circuits. In the following chapters we emphasize the design phase, the difficult part of our work. Once we have an architecture and a control algorithm, we must then implement them. In this chapter we next present some standard and systematic methods for realizing any ASM chart control algorithm.

REALIZING ALGORITHMIC STATE MACHINES

Once we have expressed the control algorithm as an ASM chart, it is a simple job to express the flow of control as hardware. We describe two methods—a traditional technique and a style-driven method, deferring to an appendix two elaborations of traditional methods.

Our task is to construct *a state generator* for a given ASM. In any state machine, the concepts of present state and next state are vital. The state generator's task is to record the present state and generate the next state. State machines are sequential circuits, and to keep track of the present state we need a memory. In this part of the book, we use flip-flops as the state memory. There are two ways to express the present state in a flip-flop memory. We may assign a binary number to each state and express the present state as an encoding, using its binary number. In this scheme, *n* flip-flops will encode up to 2^n states. We may describe a state by its name or by its number in binary or in decimal, as we find convenient. Alternatively, we may avoid the encoding by assigning one flip-flop to each state. We will use each of these approaches.

Traditional Synthesis from an ASM Chart

The traditional technique for state generation is to use an encoded representation of the present state and *compute* the code for the next state. The bits of the code are the *state variables: n* state variables describe up to 2^n states. The term state variable used in this way is unfortunate, since there is a more important use for this term—to specify the name of a logic variable for each state. Nevertheless, in this section, we use the term in the traditional way. We make an arbitrary *state assignment* of binary state variable values to states. On the ASM chart, we show the binary assignment for each state above its state rectangle on the right-hand side. We might choose the state assignment shown in Fig. 5-9. We need not label the test diamond or the conditional output oval since they are part of state 00. The state assignment is arbitrary. There may be more hardware associated with one state assignment than another, but this is not an important factor. The two state variables B and A in Fig. 5-9 specify an address that points to the present state. If we could compute the next address and put it into the state flip-flops, we would then be pointing at a new state. As you might guess, we can use gates to build a combinational circuit to compute the next address. Figure 5-10 is the model of this process. This figure displays only the control portion of the digital system, not the architecture.

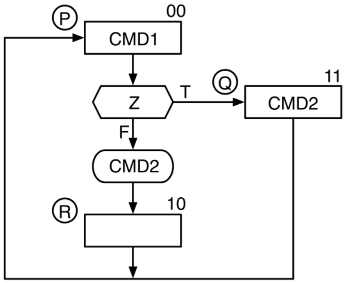


Figure 5-9. A simple ASM with a state assignment

We can use either JK or D flip-flops for the state variables. JK flip-flops usually result in less combinational logic than D flip-flops, but they also require twice as many input lines. The JK form is more compact but yields more obscure results. In this example, we use D flip-flops to provide a more direct comparison with the methods to follow.

For the ASM chart in Fig. 5-9, the state generator model of Fig. 5-10 has one status input Z, two command outputs CMD1 and CMD2, and two state flip-flops B and A. The combinational logic must compute the value of the next-state address:

Present			Next	
B	A	Z	B(D)	A(D)
0	0	0	1	0
0	0	1	1	1
0	1	X	0	0
1	0	X	0	0
1	1	X	0	0

The condensation of rows on variable Z arises because the move from states 10 or 11 does not depend on *Z*.

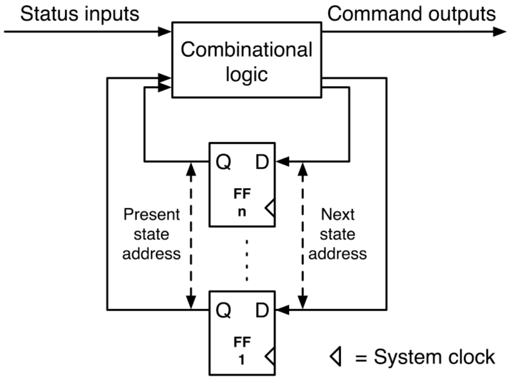


Figure 5-10. A model of an encoded ASM state generator

The state assignment 01 is a possible pattern of the flip-flop's outputs but does not label any state in the algorithm of Fig. 5-9. Hardware is perverse and may get into this state, for example during power-up, when flip-flops may settle into random values. In our example, pathological behavior would result if the next-state logic computes 01 whenever the present state is 01. We would be locked into state 01 and could not get out unless we did something drastic, such as shut off system power. Clearly, if we ever get into state 01, we must get back to the main algorithm loop. Thus, we have arbitrarily chosen to go to state 00. We must always take into account all unused state assignments in encoded designs of state generators.

We may write the equations for state flip-flop inputs B(D) and A(D) by inspecting the logic table above. They are

$$\mathrm{B(D)} = \overline{\mathrm{B}} \bullet \overline{\mathrm{A}} \bullet \overline{\mathrm{Z}} + \overline{\mathrm{B}} \bullet \overline{\mathrm{A}} \bullet \mathrm{Z} = \overline{\mathrm{B}} \bullet \overline{\mathrm{A}}$$

$$\mathrm{A(D)} = \overline{\mathrm{B}} \bullet \overline{\mathrm{A}} \bullet \mathrm{Z}$$

When we are designing more complex state machines, K-maps may help to simplify the expressions for the state flip-flop inputs.

This completes our discussion of the design of the state generator. But the purpose of the algorithm was to produce properly sequenced outputs. An examination of Fig. 5-9 yields these equations for the outputs:

$$\mathrm{CMD1} = \mathrm{STATE\_P} = \overline{\mathrm{B}} \bullet \overline{\mathrm{A}} \tag{5-1}$$

$$\mathrm{CMD2} = \mathrm{STATE\_P} \bullet \overline{\mathrm{Z}} + \mathrm{STATE\_Q} = \overline{\mathrm{B}} \bullet \overline{\mathrm{A}} \bullet \overline{\mathrm{Z}} + \mathrm{B} \bullet \mathrm{A} \tag{5-2}$$

The hardware for the ASM is shown in Fig. 5-11.

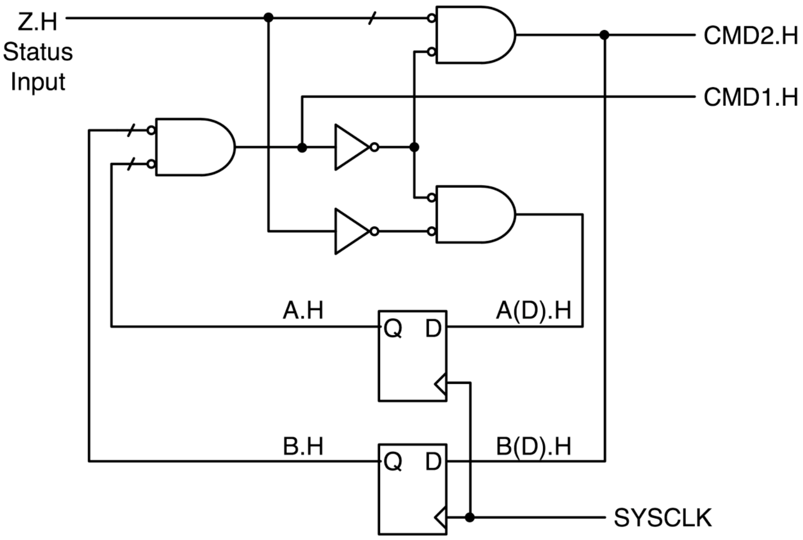


Figure 5-11. Traditional synthesis of the ASM in Fig. 5-9

Initializing the State Machine.

We have bypassed an important aspect of state machines in our treatment of Figures 5-9 and 5-11. Now it's time to face up to this omission. We have properly handled the situation where the state generator winds up in state 01 for whatever reason—it should transition back into the ASM, preferably at the starting state. What we have not solved is what happens when power is turned on? The machine could wake up in any of the 4 states 00, 01, 10, 11. In real world designs we insist on means to reset the state generator, forcing it into a known initial state. Modifying the ASM in figure 5-9 leads to figure 5-12.

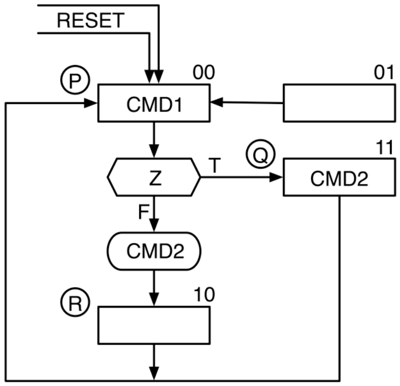


Figure 5-12. A proper ASM

Every real world ASM ***must*** incorporate means to initialize it to a known state—here shown by the RESET arrow; and also ***must*** carefully handle extra states like state 01. Failure to do so will immediately mark you as an inexperienced designer. The hardware of Figure 5-11 properly handles the extra state, 01, but ignores the reset problem; direct clear to the rescue! Remember that direct clear and direct set manipulate flip-flops independent of clocks. We can use them to force state flip-flops into desired values at any time and therein lies their power and danger.

Generating proper reset signals. Obviously, any reset switch must be debounced because bounces last on the order of milli-, or micro-, seconds while system clocks are churning along a GHz frequencies. Second, the debounced reset signal must be synchronized otherwise you could get truncated commands (often called "runt" pulses). Consider what would happen if the state machine had just entered state Q, thus generating CMD2, and then you hit the reset switch shortly thereafter. CMD2 would not last for a full clock cycle, it would be a runt pulse, and may or may not last long enough to complete its business. These considerations lead us to a proper hardware implementation shown in Figure 5-13. From now on we will assume all reset signals follow this protocol.

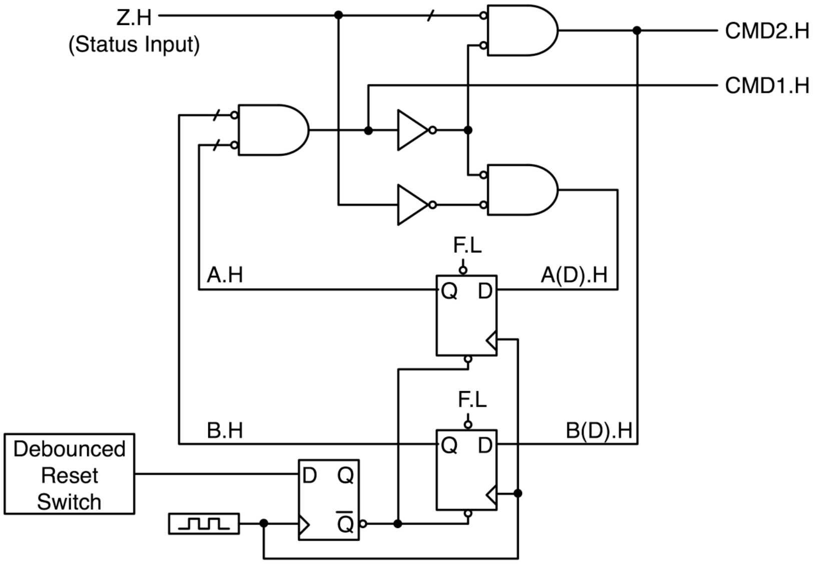


Figure 5-13. A proper implementation of the ASM of Figure 5-12

Comments. The approach we used in this section—to *compute* the code for the next state—is a traditional method. JK flip-flops used to store the state variables may lead to somewhat more compact next-state logic than D flip-flops, since the JK flip-flop is the more flexible device.

Unfortunately, the traditional method results in no obvious correspondence between the hardware for the state generator logic and the algorithm it represents. This is true of D flip-flops, and even truer of JKs. Every change in the algorithm, no matter how minor, requires a fresh design of the next-state combinational logic. The traditional approach violates our goal of clarity in design. Next-state generation is the standard implementation process associated with ASM charts, and we would like both the synthesis and the analysis of our state generators to be as straightforward and mechanical as possible. The next technique, using one flip-flop per state in an un-encoded representation differs dramatically in the method by which the next-state combinational logic is generated.

The One-Hot Method of ASM Synthesis

In this method of generating states, we use one D flip-flop for each state. There is no encoding of the states, so there is no need to specify a state assignment as we did in the previous methods. Since we must always be in only one state at a time, we must arrange for only one of the state flip-flops to be true during each state time. Therefore, we must compute with combinational logic the value T or F of each flip-flop's input to provide the one true input required to produce the next state of the system. This property of exactly one flip-flop being true at a time gives the method its name, "one-hot."

We may make exactly one flip-flop true with the aid of a tabular presentation slightly different from the one used in the previous method. Consider the one-hot implementation of the ASM in Fig. 5-12. For this 3-state system, we need three D flip-flops labeled with the states' names. Table 5-3 contains the information needed to produce the inputs to the one-hot flip-flops.

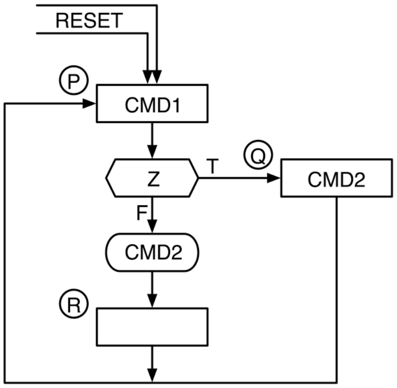


Figure 5-14. A one-hot ASM

Table 5-3 State transition data for a one-hot implementation of the ASM in Fig.5-14

Present state	Next state	Condition for transition
P	Q	Z
	R	$\overline{Z}$
Q	P	T
R	P	T

The equations for the flip-flop inputs follow from the table:

$$NEXT\_STATE\_P = P(D) = R + Q$$

$$NEXT\_STATE\_Q = Q(D) = P \bullet Z$$

$$\mathrm{NEXT\_STATE\_R} = \mathrm{R(D)} = \mathrm{P} \bullet \overline{\mathrm{Z}}$$

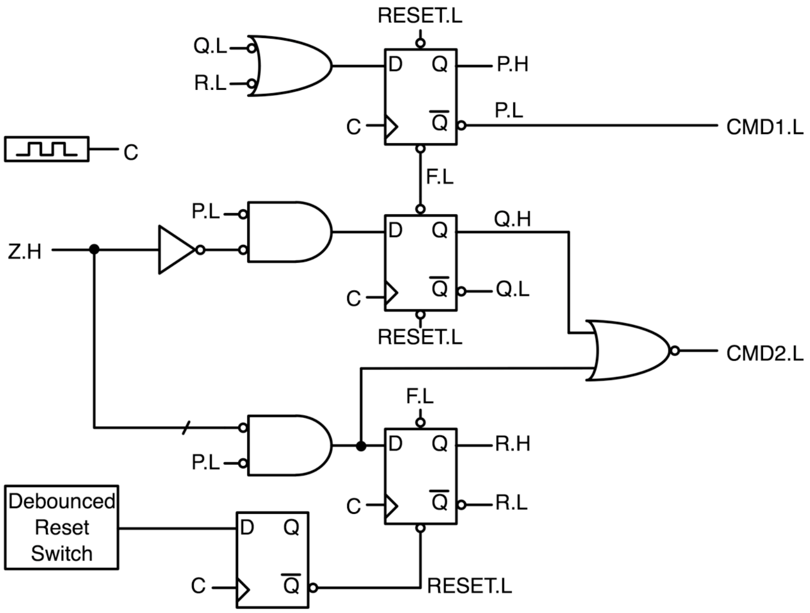


Figure 5-15. A one-hot state machine

Several things:

(a) Next state equations and hardware can be trivially read from the ASM

(b) The hardware leads directly back to the ASM

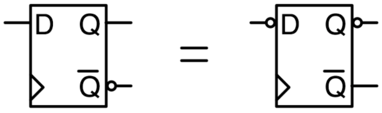
(c) RESET.L signal *clears* flip-flops Q and R but *sets* flip-flop P

(d) The one-hot method usually uses marginally more hardware than a traditional encoded state machine but with modern gate arrays you will have a surfeit of flip-fops and gates so this is of little concern.

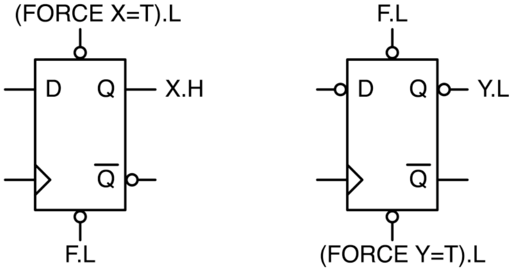
(e) ASM size extends gracefully, adding one more state is easy, it may be harder with encoded control if you cross a 2^n—2^{n+1} boundary

These are powerful incentives for preferring the one-hot method and we recommend you use it in your work unless special circumstances dictate encoded control.

Let us go through a somewhat more complex example, but first let's look at an optimization that's trivial for a mixed logician, but would likely give apoplexy to a positive logician.



This transformation is transparent, but direct set and direct clear are trickier. If direct set = T then the Q pin will go H in either symbol (after all, they are physically identical devices and pins). Remember that Q inside the rectangle represents a pin, not a signal. If you want to force a signal, X, attached to pin Q in the left symbol, to go true you must drive direct set T, and then the polarity of X will be X.H. If you want to force a signal, Y, attached to pin Q in the right symbol, to go true you must drive direct clear T, and then the polarity of Y will be Y.L.



This was a fairly simple illustration. With some familiarity with the one hot controller method, you could have read off the flip-flop inputs directly from the ASM chart without using a transition table. Let's do a more complex example—the ASM in Fig. 5-16.

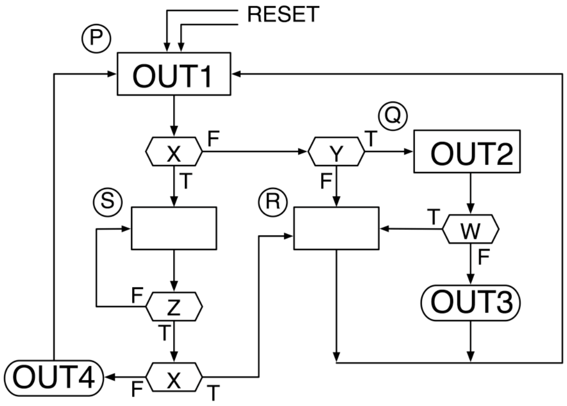


Figure 5-16. A complex 4-state ASM

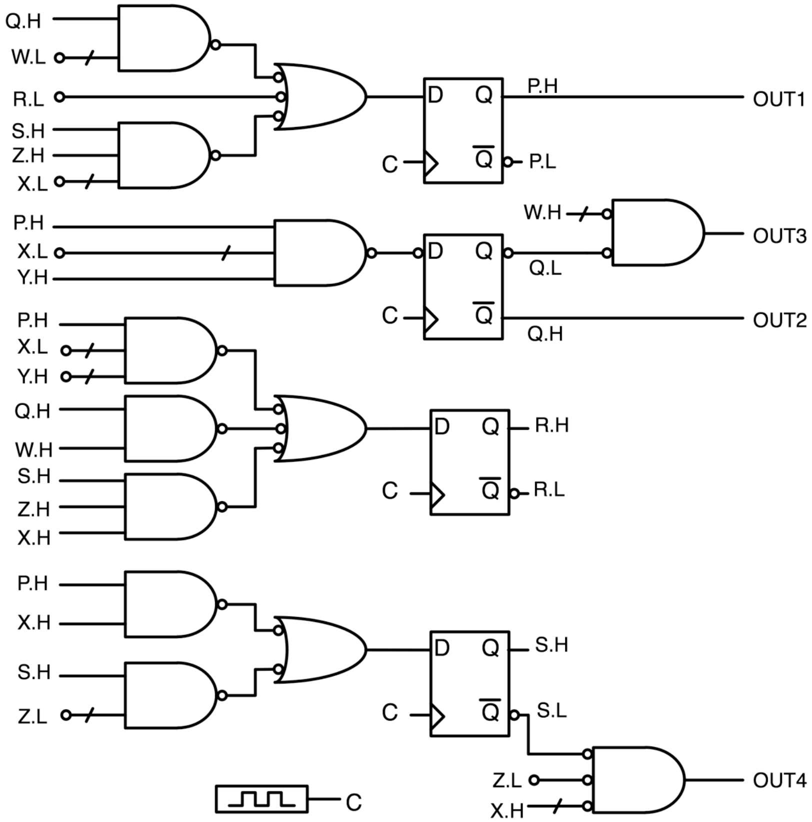


Figure 5-17. State generator for 5-16 (ignoring RESET)

Figure 5-17 is an implementation of this controller, assuming that the status inputs W, X, Y, and Z are available in both voltage forms. Again, there is a one to one correspondence between the ASM and hardware; to convince yourself, you should work backwards from the hardware in Figure 5-17 to recover the parent ASM.

We ignored initializing to concentrate on next state generation—as a working designer you will *never* have that luxury. We leave initializing as grist for your mental mill; you will need to modify Figure 5-17, using flip-flops with direct set and direct clear, and you may assume the RESET switch signal has been properly debounced and synchronized. RESET logic is standard, all state *signals* must be set to F, except the initial state *signal*, P, which should be set to T. (setting the *signal* Q to F will not be the same as *resetting* the corresponding *flip-flop,* which has been used in "inverted form" according to the previous identity)

One Hot vs. Encoded State Machines. The one-hot method has the advantages of ease of design and clarity of circuit. The inputs to the state flip-flops directly describe the conditions under which each state is the next state. As you know, the mixed-logic notation provides for ease of analysis of circuits, so we may read off the next-state conditions from the circuit diagram. The size of the state generator circuit does not grow rapidly with the number of states. At first you may be horrified at the idea of using a flip-flop for each state instead of the more compact encoded scheme used in the other methods. However with modern gate-arrays you will have more flip-flops than you can use for any conceivable state machine.

The beautiful properties of one hot state machines makes them our preferred implementation technique, “but”; real hardware is perverse and conceivably, through noise, an act of God, (or the design, heaven forbid!), two state flip-flops could be set. This is the “two hot” problem. Encoded state machines never have that problem, no matter what happens to the state flip flops they always encode exactly one state. If you are making a device that will be mass marketed, and has a simple ASM, say 8 states or less, perhaps encoded control would be appropriate. If you fall into this niche situation, investigate the MUX method for implementing ASM’s, covered in appendix *.

DESIGN PITFALLS

In our study of design, we have made several important assumptions about our systems.

(a) We have assumed that our ASMs are synchronous, with changes in state and other actions governed by a master clock.

(b) We have assumed that at the time of a state change, all inputs to the ASM are stable; in other words, inputs change synchronously with the system clock.

(c) We have assumed that the system clock edge reaches each element in the circuit simultaneously.

Let's investigate the effect of violating these conditions. Conditions (a) and (b) are design decisions of great importance, violation of which will lead to serious problems. Condition (c) is a subtle matter that we enforce by good construction practices. Let's consider (c) first.

Clock Skew

In a synchronous system, it is important that every clocked element in the system receive its clock edge at precisely the same time. To see why this is so, consider the general model of a controller with just two state flip-flops, shown in Fig. 5-18.

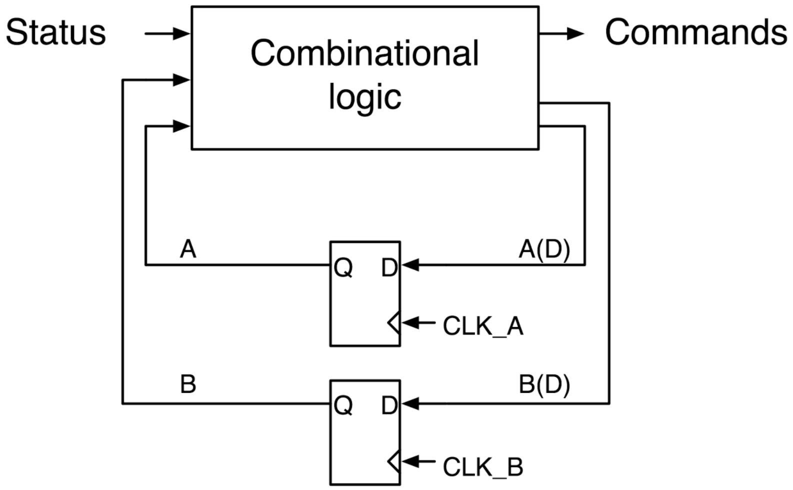


Figure 5-18. A model of a two flip-flop controller

Proper synchronous operation results when the CLK_A and CLK_B active edges occur at the same instant. In that case, the combinational logic network will compute new values for A(D) and B(D) based on the current values of A and B and the status signals and, shortly after the clock edge arrives, A and B will assume new values equal to the old values of A(D) and B(D). The changing flip-flop outputs will throw the combinational logic network into shock as it adjusts to the new inputs and computes new values of A(D) and B(D). The combinational logic outputs will experience hazards and delays caused by the finite propagation time of the gates. A(D) and B(D) may therefore have momentary wrong values but will eventually settle to levels predicted by Boolean algebra and will then wait for the next clock edge to come along.

Now suppose that CLK_B is delayed with respect to CLK_A. Signal A changes when CLK_A fires; this will throw the gates into shock as before, and both A(D) and B(D) may have momentary wrong values for a short time. Suppose that the "late" CLK_B edge comes during this time of instability; then B can record a false value. Even more galling: suppose that before CLK_B fires the combinational logic stabilizes to "new" values of A(D) and B(D), based on the new value of A and the old (unchanged) value of B. At this time, both A(D) and B(D) can be incorrect. Then, when CLK_B fires, an incorrect B is stored, and this change ripples through the logic. This phenomenon, clock skew, occurs when the clock edges do not appear simultaneously at all clock inputs. Clock skew can arise from gates in the clock path, capacitive loading, or from different wire lengths between the clock source and the clock inputs.

If you are working with gate arrays your system will likely be contained in one chip and you won’t have to worry about supplying board-wide clocks; outside the gate array is another matter. Putting gates in the clock can introduce skew.

Don't gate your clock. Gating a clock is bad practice because it introduces skew (and may introduce hazards on the clock line). Suppose that flip-flop A responds to a positive clock edge but that we use a different type of flip-flop for B that acts on a negative clock edge. We may be tempted to create the CLK_B signal by running an inverter from CLK_A, but this is just a case of gate-created clock skew. To avoid this type of skew, it is best to drive all flip-flops with the same active clock edge, thereby eliminating the need for inverters in part of the clock system. In all our clocked building blocks for synchronous design, we use positive-edge clocks.

Beware different length of clock paths. In the rare case where you have to provide a board-wide clock it behooves you to pay as much attention to skew as integrated circuit designers do for intra-chip clocks. It is desirable to have the clock distribution lines spread radially from the clock source to the separate elements of the system rather than linking them together in one long chain. When designing large systems you may have to buffer the clock lines to build up sufficient power, as shown in the radial clock distribution system in Fig. 5-19. This "gating" of the clock lines is acceptable if the three buffers are all of the same kind and, preferably, in the same integrated circuit package, so that they all have precisely the same propagation delays. The distribution wires should also be of the same length, within a few inches, since 8 inches of wire represents about 1 nanosecond of signal propagation time.

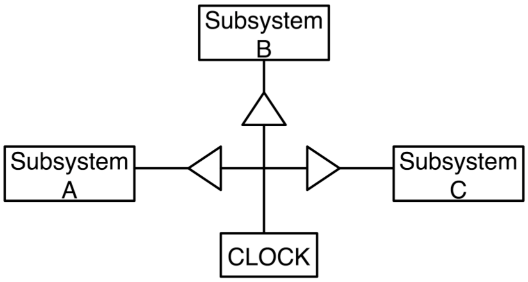


Figure 5-19. A buffered board wide distribution system for the master clock

Asynchronous Inputs and Races

Design assumption (b) is that all the inputs to an ASM change synchronously with the master clock. In practice, inputs often arise from sources *outside our* digital circuit, and the timing of changes in these inputs is beyond our direct control. These inputs are *asynchronous,* and we usually append an asterisk * to their variable name to indicate their asynchronous nature. To see why asynchronous behavior is troublesome, consider the three-state ASM fragment in Fig. 5-20. We assume that we have made the (encoded) state assignment shown in the figure and that the sole test input, IN*, is asynchronous. For the moment, ignore the conditional output.

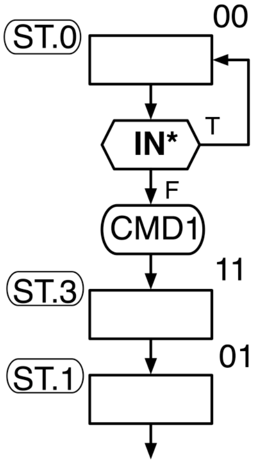


Figure 5-20. An ASM fragment that illustrates races

Transition races. Each state flip-flop requires that its input be stable during the setup time prior to the clock edge. This allows the input values to circulate through the internal circuitry of the flip-flop and stabilize to await the clock edge. If the flip-flop inputs change during this "setup time," the value of the flip-flop's output after the clock edge will be unpredictable. After settling down, the output will assume either a T or F value, but which value is uncertain.

Now assume that the ASM in Fig. 5-20 is in state 00, and IN* = T. Then the inputs to both state flip-flops are 0, and the system is preparing to move next to state 00, the same state as before. If IN* changes to F, the inputs to flip-flops A and B will change to 1, in preparation for the move to state 11. If the change in IN* occurs during the flip-flop setup time, we cannot predict the changes in the flip-flops. Thus the next-state code may be 00, 01, 10, or 11, depending on the outcome of the race at the flip-flop inputs. Although we might argue that either state 00 or state 11 is an acceptable next state, clearly to reach states 01 or 10 is a calamity. This situation, in which the next state depends on the exact timing of the flip-flop input changes, is called a transition race. The situation is obviously intolerable and you must be certain that such races do not appear in your designs. Before considering the solution to the problem, let's investigate another type of race.

Output races. Now see what happens to the conditional output CMD1 in the ASM of Fig. 5-20 when IN* changes at an awkward time. For the moment, ignore the possibility of transition races. In state 00, when IN* is true, CMD1 is false, whereas when IN* is false, CMD1 is true. A change in IN* will cause a corresponding change in CMD1. If IN* changes from T to F late in the state 00 time, CMD1 will be true for only a short part of a clock time, before the system moves into state 11. The possibility of a "runt pulse" for CMD1 is in itself a serious matter, since the output CMD1 may be used in situations that cannot tolerate such a short pulse. This problem is called an output race; it is a direct result of an ASM output being conditional on an asynchronous input. For example, suppose that the purpose of CMD1 is to set a flip-flop that lights a light announcing that we have left state 00. If IN* changes so late in the clock cycle that the flip-flop is not set to true, we will be in state 11 and the light will still be off.

The combination of the transition race and the output race in this ASM may lead to numerous ludicrous results, depending on the exact reactions of the flip-flops to the changing IN* input. We could end up in state 00 with the light on, or in state 01 with the light off, and so on.

Avoiding races. Asynchronous inputs are at the root of the problem of races. Asynchronous inputs are fatal, dangerous, or at best difficult to use safely. There is no way to avoid output races except to avoid conditional outputs that depend on asynchronous inputs. The engineering literature is full of elaborate methods for skirting around the transition race problem by tinkering with the state assignments. The proper approach for good design style is to eliminate the cause of the problem—the asynchronous input. We may do this by synchronizing the input using a D flip-flop clocked by the system clock. The ASM will test the output of this flip-flop, and since this output only changes synchronously with the clock, there will never be a race in the ASM caused by that input. So we have a golden rule for synchronous design:

Don't allow dangerous asynchronous inputs into your ASM chart.

You must be alert to identify asynchronous inputs—they have a habit of sneaking into the design. In our example, the input was easy to detect, since we had added a * to the signal name, but in practice, adding the asterisk is *your* responsibility, whether or not the original name was so equipped. Many useful synchronous integrated circuit chips have asynchronous control inputs for clearing, setting, or loading. The only routine use that we make of such inputs is as *a master clear* signal to be asserted when power is first applied or when the system "hangs up." In the one-hot controller method you saw an illustration of this usage; most controllers will require one such master clear signal. In other circumstances, avoid using asynchronous control signals.

Asynchronous ASMs

Our ASMs have all been synchronous, with a master clock to define the times for state transitions. It is possible to build asynchronous state machines, which depend not on a clock but on changes in the inputs themselves to create transitions between states. Troubles abound in this form of design, primarily because of these factors:

(a) All inputs must be clean, with no glitches, since any instability or noise on the input signals may induce spurious state transitions.
(b) The theory of asynchronous circuits is complex and diverse, involving numerous special cases and usually invoking unacceptably restrictive design conditions.
(c) The debugging of asynchronous systems is difficult.

We will not instruct you in the theory of asynchronous circuits, beyond the microscopic view we took in Chapter 4. Rather, we wish you to avoid this mode of design wherever possible. In Chapter * you will encounter a form of asynchronous ASM used to connect a peripheral device to a minicomputer, but this special case is as far as we will take the subject. If at some later stage in your design career you wish to investigate asynchronous circuits, you will find a rich literature. However, we wager that, after your investigation, you will still refrain from designing circuits in this mode.

Sidestepping the Pitfalls

Getting into trouble in digital design is easy. Asynchronous methods constantly expose the control algorithm to every signal change, intentional or accidental. Designing each circuit to be secure against such an unceasing attack requires enormous effort. Asynchronous control, despite its tempting generality, is too tedious to use as a major tool in design.

Synchronous methods ease the pressure on the designer by isolating the sensitive periods into small, regular intervals preceding clock edges. Expressing problems synchronously effectively moves the asynchronous difficulties away from the algorithm into the clock. This is a great simplification. We must work hard to make the synchronous clock system reliable, but the procedure is the same for all designs. In return, we gain breathing room in the algorithm. Synchronous design causes specialization of the hardware, where extreme versatility is rarely needed, yet it decreases the number of special cases in the process of algorithm design, giving designers valuable systematic methods.

As you saw in Chapter 4, the problem of metastability in the outputs of sequential circuits is inherent in every design. Wherever two or more inputs may change at the same time, metastability is possible. In asynchronous design, virtually every input may present this problem. In synchronous design, the asynchronous external inputs, which we routinely run through synchronizing circuits, are trouble spots. By synchronizing these inputs we have greatly simplified the internal structure of our control algorithm, but we have not eliminated the possibility of metastability. In Appendix * we offer recommendations for dealing with this irritating issue.

Debugging Synchronous Systems

Not only do synchronous methods ease the designer's worries, but they also support a powerful debugging technique. The system clock controls the speed of a synchronous design. Consider the benefits of a design that will behave properly not only at high clock speeds but also at slow speeds, even at zero speed. We are particularly interested in the zero-speed case, because with this feature we may freeze the system in any state by stopping the system clock. We may then debug the logic at leisure. Compare this technique with a system that requires a closely controlled clock frequency. Error conditions may be observable for only one clock cycle. If maintenance engineers cannot slow or stop the machine at will they must troubleshoot it in real time. Debugging systems at high speed is much more complicated than freezing the machine in the erroneous state.

Synchronous designs that work at a variety of clock speeds, including zero, are called *static.* The benefits of static systems are so great that we should strive to use this technique whenever possible. After you have debugged a few dynamic (non-static) systems, you will better appreciate the beauty of static designs. In

Chapter 6, we develop a system clock module for use in static designs.

CONCLUSION

This ends our exposition of basic design methods. We have covered the basic building blocks and there were remarkably few types. Next, for expressing algorithms, we described a language that contained only three constructs: the state box, the conditional output oval, and the conditional branch diamond. With these simple tools we can create digital systems limited only by our imagination.

The basic elements of style emerge from a desire to achieve understandable designs. We have discovered through bitter experience that opaque designs are enormously expensive in the long run. A good designer will use a design approach that always promotes clarity. We mention three important aspects of such an approach:

(a) *Good documentation.* It is tempting to avoid the drudgery of documentation. After all, the real fun is in the design and debugging. It is hard enough to document a simple design; complex designs are seldom documented well enough for anyone but the designer to understand. Often, after a few months, even the designer cannot fathom the design. A good designer will adopt techniques that encourage or require good documentation during the design process. Mixed logic, functional building blocks, and ASM charts are powerful aids to documentation, built into the design discipline.

(b) *Modular designs.* Nearly every design will require small changes during its useful life. Monolithic designs are hard to understand and modify. Our goal is to build more accessible designs, so that we may change part of the complete system without the change rippling through the rest of the design. Too often, digital designers overlook the cost of servicing digital equipment. Since servicing usually falls to other people, poor designers are not forced to live with their abominations. Hardware *will* need repair. Digital devices should be simple and modular so that other people can perform the maintenance. The use of functional building blocks and the separation of architecture from control both encourage modular design.

(c) *Absence of tricks.* Digital design affords unbounded opportunity for clever tricks. Such trickery should not be, but often is, confused with good design. We can benefit from the experience of computer programmers who, after years of maneuvering bits in clever ways, have come to realize that systematic, clear methods yield far more dividends than cute but obscure tricks.

Perhaps we can sum up good design philosophy in a single phrase: common courtesy. Consider the users and maintainers of your system, and ask yourself what they will need in order to deal efficiently with your creation. Let courtesy be your guide.

Summary of Design Guidelines

Here we bring together the three forms of design guidance presented in this chapter.

Basic Approach to Solving a Digital Problem

(a) Design from the *top down.*
(b) Separate the *architecture* from the *control.*
(c) *Refine the design,* letting the control algorithm and the architecture influence each other as you converge on the solution.

Technical Design Considerations

(a) Use *synchronous* (clocked) design techniques.
(b) *Avoid asynchronous inputs* in the algorithm.
(c) Make your designs static—independent of clock speed.

A Courteous Philosophy

(a) Develop good *documentation* during the design.
(b) Keep designs *modular* and *simple.*
(c) *Avoid* obscure *tricks.*

In Chapter 6, we will work out several design examples. In the process, you will see the design tools in action and study in their proper context a number of common design situations and their handling.

So now begins the actual design!

READINGS AND SOURCES

CLARE, CHRISTOPHER *R., Designing Logic Systems Using State Machines.* McGraw-Hill Book Co., New York, 1973. The original exposition of the ASM approach.

DIETMEYER, DONALD L., *Logic Design of Digital Systems,* 2nd ed. Allyn and Bacon, Boston, 1978. Chapter 13: traditional asynchronous design.

ERCEGOVIC, MILOS D., and Tomas LANG, *Digital Systems and Hardware/Firmware Algorithms.* John Wiley & Sons, New York, 1985. Good treatment of sequential systems.

Fletcher, WILLIAM I., *An Engineering Approach to Digital Design.* Prentice-Hall, En-
Glenwood Cliffs, N.J., 1980. Uses ASMs.

HILL, FREDERICK J., and GERALD R. PETERSON, *Introduction to Switching Theory and Logical Design,* 3rd ed. John Wiley & Sons, New York, 1981. Good traditional treatment of sequential circuits.

MALEY, G. A., and J. EARLE, *The Logic Design of Transistor Digital Computers.* Prentice-Hall, Englewood Cliffs, N.J., 1963. An influential early work; you can see how difficult design can be without systematic methods.

MANO, M. MORRIS, *Digital Design.* Prentice-Hall, Englewood Cliffs, N.J., 1984. Presents ASM charts as an alternative to traditional techniques.

MEALY, G. H., "A method for synthesizing sequential circuits," *Bell System Technical Journal,* Vol. 34, September 1955, page 1045. The Mealy state machine.

MILLER, RAYMOND E., *Switching Theory.* Vol. *2, Sequential Circuits.* John Wiley & Sons, New York, 1966. An important early work.

MOORE, E. F., "Gedanken experiments on sequential machines," in *Automata Studies,* edited by C. E. Shannon and and J. McCarthy. Princeton University Press, Princeton, N.J., 1956. The Moore state machine.

WIATROWSKI, CLAUDE A., and CHARLES H. HOUSE, *Logic Circuits and Microcomputer Systems,* McGraw-Hill Book Co., New York, 1980. Uses ASMs. Sensible treatment of asynchronous ASMs.

EXERCISES

5-1. Sketch a general method for the top-down solution of a digital problem.

5-2. How does the ASM chart differ from a software flowchart? Using Fig. 5-8 as an illustration, explain the fundamental differences in viewpoint.

5-3. What is meant by "active clock edge"?

5-4. What is a state time? In a synchronous system, what determines the duration of the state time?

5-5. Draw diagrams to illustrate the following

(a) A four-state cyclic (sequential) ASM.

(b) A three-state ASM with a fixed sequence of states containing a conditional output.

(c) A two-state ASM with a two-way branch in one state and no conditional outputs.

(d) A four-state ASM that can produce this sequence of states: S1, S3, S2, S1, S4, S2, S1, S1, S1, ...

5-6. Explain the difference between an ASM input and an ASM output.

5-7. What is the difference between an ASM conditional branch and an ASM conditional output? Does one imply the other?

5-8. In a synchronous ASM, an unconditional output is stable for virtually the entire duration of the state. For what period is a conditional output stable?

5-9. Produce ASM charts that perform each of the following software operations:

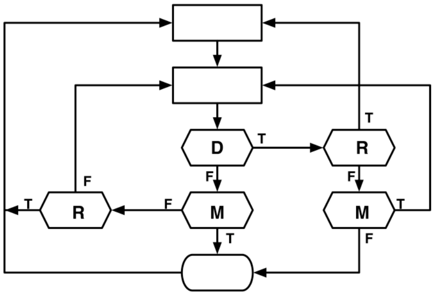
(a) If X = N, then ... .

(b) If X = N, then . . . ; else . . .

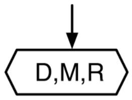
(c) For X from A to B step C, do ... .

(d) While X = Y, do ... .

5-10. Here is a two-state ASM:

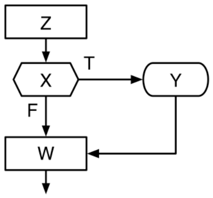


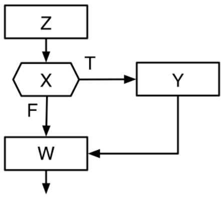
(a) Convert the ASM into a form that has a single decision box of the form below, with eight branches:



(b) Implement the original ASM and your modification. From the viewpoint of the implementer, which form is best?

5-11. Using timing diagrams, show the difference between these two ASMs:



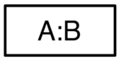


5-12. In many instances, we may remove a conditional output from an ASM by creating a new state dedicated to generating the old conditional output (see the diagrams in Exercise 5-11). Under what circumstances will this translation produce difficulties?

5-13. A conditional output is a function of both state and path. In a logic equation for a conditional output, what logic operator connects the state term with the path term? In other words, what logic operator corresponds to the box in the equation

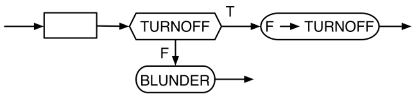
$$\text{Conditional.output} = \text{State} \boxed{?} \text{Path}$$

5-14. The ASM notation

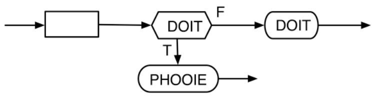


can be used to indicate that "A assumes the value of B at this time." Show that this notation may be viewed as a shorthand for an ASM state that contains a test and a conditional output.

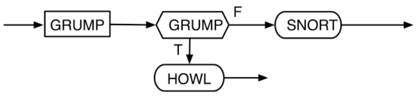
5-15. Consider the following fragments of an ASM chart. Carefully state what, if anything, is wrong with or objectionable about these notations.



(a)



(b)



(c)

5-16. What is a state generator? With an encoded state assignment, how many states can four state variables specify?

5-17. Some older design methods lump the controller and architecture of Fig. 5-1 into the combinational logic of Fig. 5-10. Why is this poor practice?

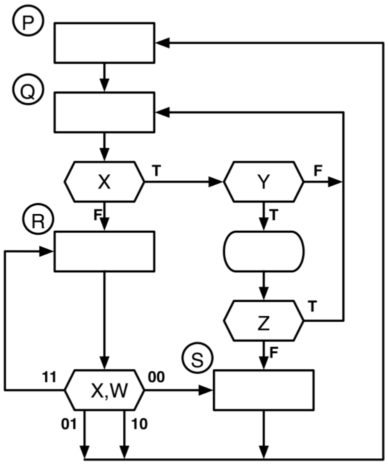
5-18. Design a one-hot controller for Fig. 5-8.

5-19. Produce a realization of the ASM in Fig. 5-9 that is equivalent to Fig. 5-11 but with the state assignment P = 00, Q = 01, R = 10. Is there any difference in hardware complexity?

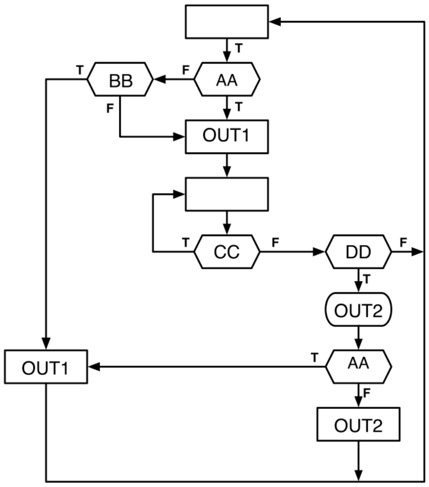
5-20. Can you make a state assignment in Fig. 5-9 that will simplify the hardware for generating CMD1 ?

5-21. Design a traditional state generator for the ASM in Fig. 5-20, using D flip-flops.

5-22. Design a one-hot controller for this ASM. What special precaution must you take when using the one-hot method?



5-23. Devise implementations, including circuit diagrams, for this synchronous ASM, using each of the state-generation techniques given below.

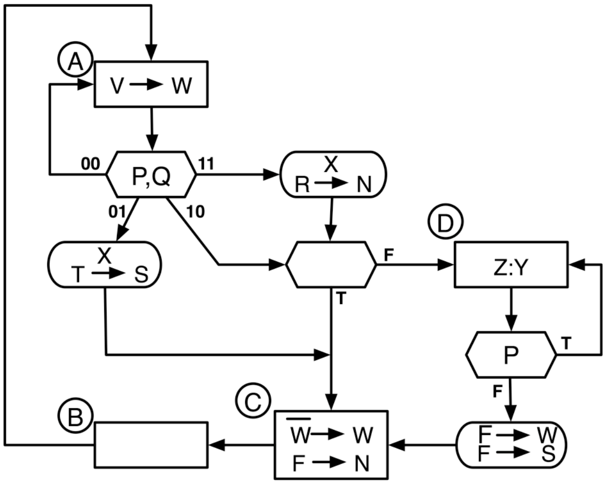


(a) traditional controller, using gates or modules in your simulator library.

(b) One hot controller, using gates or modules in your simulator library.

(c) Verify both constructions using a simulator

5-24. Perform Exercise 5-23 for this ASM. Generate W with a JK flip-flop and N with an enabled D flip-flop. The notation "Z**:**Y" implies "Z assumes the value of Y at this time" (see Exercise 5-14).



5-25. The logic equation for an ASM output has terms involving distinct logic variables for each ASM state in which the output appears.

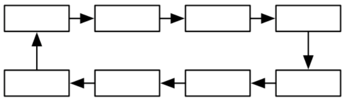
(a) For a state generator with encoded state assignments, show a standard and systematic method of transforming the state code into logic variables for each state, using a decoder.

(b) Instead of using a decoder, we can generate ASM outputs with AND gates to decode the required logic variables for states from the state code. Demonstrate this method. When would you use this method in preference to the method of part (a)?

5-26. In the one-hot state generator method, show how logic equations are generated for conditional and unconditional outputs. Is output signal generation simpler with the one-hot method than with the traditional method?

5-27. Use two 8-input priority encoders to detect when more than 1 bit is true in an 8-bit quantity. *(Hint:* Connect the input signals to each of the encoders, but in opposite order.) Can this design be extended to more than 8 bits? Show a use of this circuit in the design of one-hot controllers to handle the two-hot problem.

5-28. Consider the following eight-state cyclic ASM:



Design state generators for this ASM as a one-hot controller, and a *binary counter.* Which method is simplest in this special case?

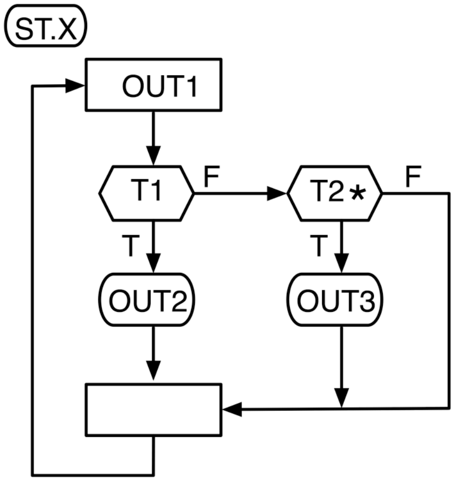
5-29. Why is the clock such an important element in a synchronous design?

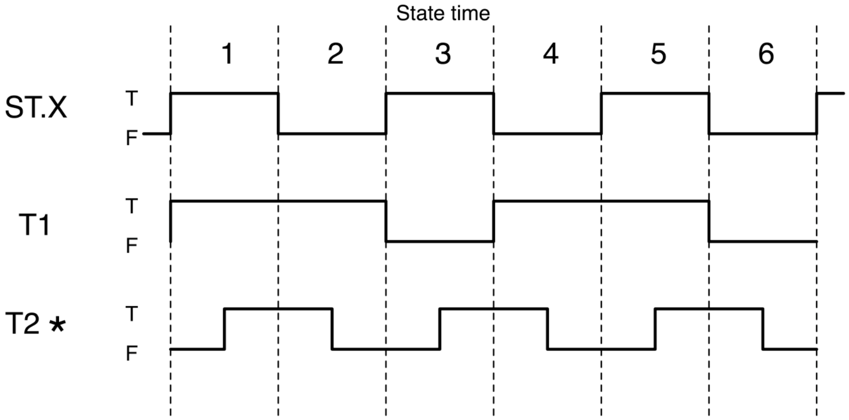
5-30. What is meant by "gating the clock"? Why is this practice dangerous?

5-31. For the ASM in Fig. 5-19, show with a timing diagram how the asynchronous input IN* can cause a transition race.

5-32. Using Fig. 5-19, demonstrate an output race.

5-33. How may you avoid races in your designs? Suppose that, contrary to our basic design approach, you must deal with a synchronous ASM that tests an asynchronous input T2*, as shown below. The logic-level timing diagram shows the condition of three signals over a period of six state times. Complete the logic-level timing diagram for the signals OUT1, OUT2, and OUT3.





5-35. In Fig. 5-11, assume that the propagation delay of gates and flip-flops is t_p

(a) How much clock skew is tolerable between flip-flops A and B during the transition from state 11 to state 00?

(b) Repeat part (a) for the transition 10 → 00.

(c) Repeat part (a) for the transition 01 → 00.

(d)