5-4 Consider the ASM in Fig. P5-4. Its state register is a 3-bit asynchronous counter. The counter stops counting when it reaches 111. It can be started only by external input *LSTRT*. What is the maximum clock frequency at which this circuit will correctly operate? Use the component specifications in Table P5-1.

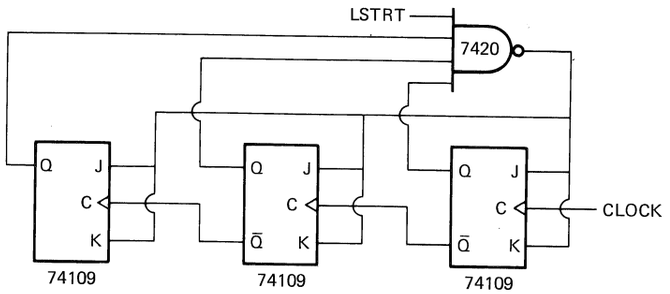


Figure P5-4 Asynchronous counter ASM.

5-5 Consider the grated-clock ASM of Fig. 4-14. Will clock skew between *HCLK* and *GCLK* cause faulty operation? What is the allowable clock skew? What is the maximum skew attributable to the clock drivers? What is the maximum clock rate of this circuit? Assume the NAND gate is a 7410, the AND gate is a 7408, the inverter is a 7404, and the flip-flops are 74109s. The specifications for these components are shown in Table P5-1.

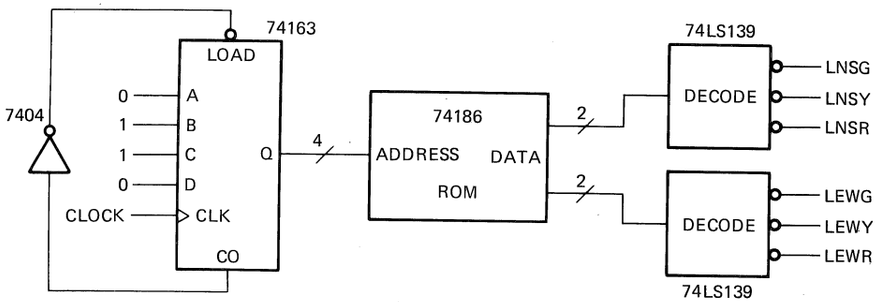


Figure P5-5 Traffic-light controller.

5-6 The circuit of Fig. P5-5 is the simplest traffic light controller. It uses a counter for state register and has no branches in its ASM chart. What is the maximum clock rate of this circuit? Use component parameters in Table P5-1.

5-7 The RWM interface example in this chapter did not use the memory to its full speed capability. Even using an asymmetric clock, the memory write cycle was longer than it might be. Worse yet, the entire ASM was constrained to a 310 ns clock. Design an interface from ASM to RWM that will shorten the write cycle.

Hint: Try dividing the memory cycle into more than one ASM cycle.

CHAPTER

SIX

ASYNCHRONOUS INPUTS AND ASMS

In previous chapters, we have always assumed that external inputs to the ASM will change synchronously with the clock. Inputs were assumed to change just after the clock transition (along with the state variables), which allows sufficient time for the effect of input changes to arrive at state flip-flop inputs before the next clock transition. Inputs will be synchronous if they are outputs of another ASM using the same clock signal. Many times, synchronism will not exist. For example, an input could be generated by another ASM which is timed by a different clock signal. Often, inputs are generated by pushbuttons or other physical events which have no time relationship to any clock signal. An input not synchronized with the ASM clock is called an *asynchronous input.* What problems arise when an asynchronous input is connected to a synchronous ASM? One obvious problem is that any output of the ASM that is a function of *both* the current state *and* an asynchronous input will, itself, be asynchronous. An asynchronous input change will simply propagate through the output ROM or gating. If asynchronous outputs (that change at times other than clock transitions) must be avoided, the ASM chart can have no conditional outputs that depend on asynchronous inputs.

Figure 6-1 shows a conditional output *IHLD* which depends on an asynchronous input *YX* while in state 12. You can see that *IHLD* follows the asynchronous changes of *YX*. *IHLD* is an immediate output; so any changes between clock transitions will cause the *IHLD* function to occur. In the timing diagram of Fig. 6-1, the *IHLD* function would occur twice. The next state would be state 13. The ASM chart of Fig. 6-1 indicates that *IHLD* should occur once and always should be followed by state 14!

Transition Races

Asynchronous inputs can cause a much more serious problem: erroneous next states. A simple ASM is shown in Fig. 6-2. The single input *YI* is an asynchronous input.

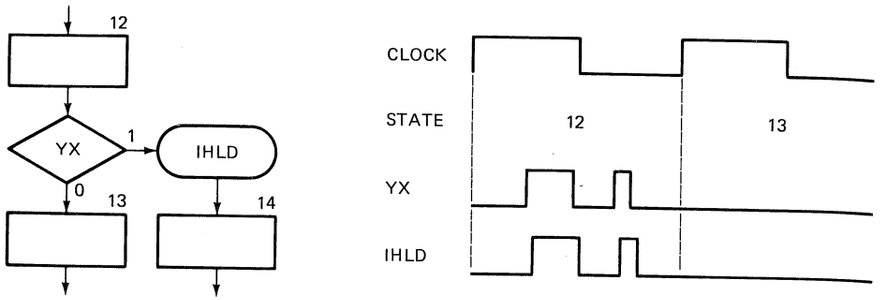


Figure 6-1 Asynchronous output caused by asynchronous input.

Assume that the ASM is in state 00 and asynchronous input *YI* changes state just before the next clock transition. In fact, the input changes so close to the clock transition that the new next state from ROM outputs D_2 and D_1 arrives after the worst-case setup time of the flip-flops. Setup time specified is for the worst flip-flop expected. Flip-flops *A* and *B* may have quite different setup times, as long as neither is larger than the worst-case specification. Thus, the *A* flip-flop, the *B* flip-flop, both flip-flops, or neither flip-flop might recognize this late next-state change.

Let's use a specific example to demonstrate the way in which the circuit could malfunction. If *YI* = 0 and the current state is 00, the next state should be 10. Suppose that YI changes from a 0 to a 1 late in the clock cycle, so late that both flop inputs change after the worst-case setup time. If both flops do not recognize this change, the next state will still be 10. The operation of the ASM is correct. The input change occurred too late to be recognized. However, it is very likely that one or both flip-flops will have an actual setup time less than the worst-case setup time specification. If both flip-flops recognize the change, the next state will be 01 and the ASM is still functioning correctly. Because both flip-flops operate better than specification, the late input

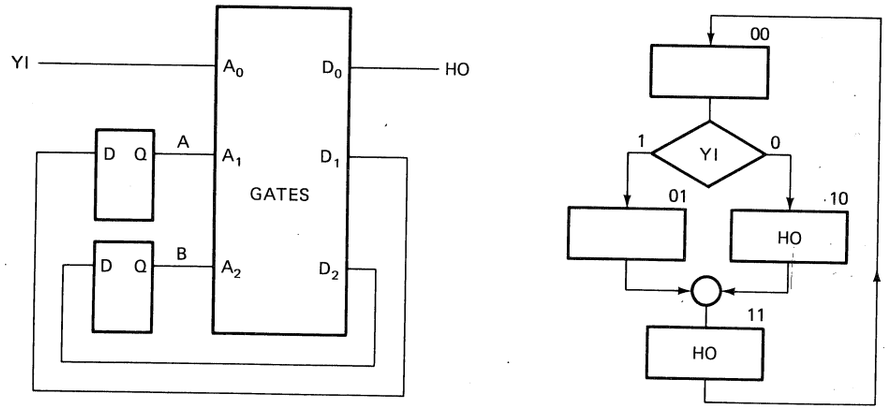


Figure 6-2 ASM with transition race.

change is correctly recognized. However, if the *B* flip-flop is better than the *A* flip-flop, the *B* flip-flop might recognize the state change, while *A* will not, and the next state will be 00! If the *A* flop recognizes the change and the *B* flop does not, the next state will be 11. In both of these last cases, the actual next state was incorrect for either *YI* = 0 or 1. The ASM chart does not even *have* an exit path from state 00 to states 00 or 11!

Not only can setup time differences cause transition races, but differences in propagation times of next-state variables can also cause transition races. Propagation-delay differences can be quite large in gate implementations. A transition race can be avoided by modifying either the ASM chart or the circuit. In Fig. 6-3, asynchronous input *YI* affects only one state variable *B*. If input *YI* changes from 0 to 1 late in the clock cycle, this change may be recognized and the next state will be 01. If the change is not recognized, the next state will be 11. Since the *A* flip-flop is always a 1 in each of these next states, it doesn't matter if *A* does or doesn't recognize the change as it will be 1 in either case! Since only the *B* flip-flop changes, the next state can be only one of the two correct next states. Transition races may be avoided by assigning next states so that all decisions on asynchronous inputs affect only one state flip-flop. In a complex ASM with many asynchronous inputs, additional state flip-flops may be required to follow this rule.

An alternative solution is changing the circuit to ensure that state changes occur only at the beginning of a clock cycle. In Fig. 6-4, a synchronizing register has been added to the circuit to assure that changes in *YI* will only be sent to the ROM just after a clock transition. Any state assignment will work properly in this ASM because *YI* is synchronized with the system clock before being used to select the next state. In a complex ASM this latter solution may eliminate extra state flip-flops, required to meet the state assignment criteria of the former solution. Even though synchronizing flip-flops are required, the total number of flip-flops could be less in a complex ASM. In addition, conditional outputs depending on asynchronous inputs will now be synchronized with the clock. No ROM address input is asynchronous; so no output will be asynchronous. Most importantly, the time and cost of designing this circuit with a synchronizing register is less because states can be arbitrarily assigned. In a complex

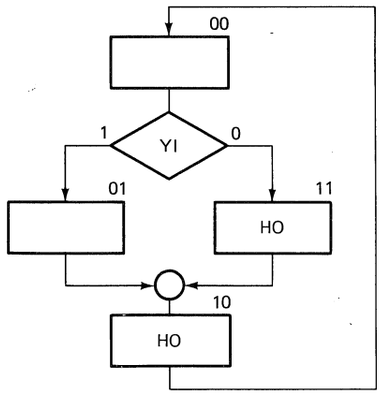


Figure 6-3 ASM without transition races by changing state assignment.

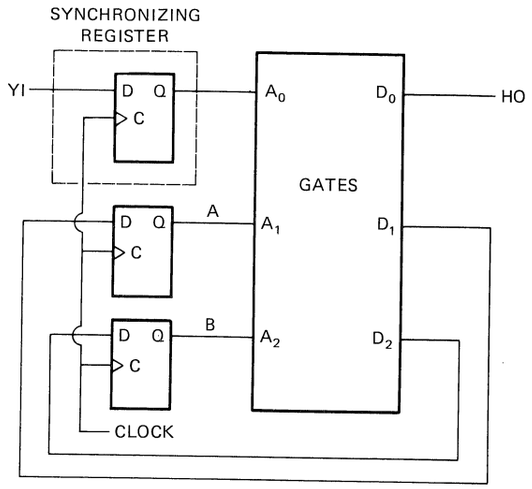


Figure 6-4 ASM without transition races by adding synchronizing register.

ASM, the regularized design of the synchronizing register is much quicker than a specialized state assignment.

Output Races

In an earlier chapter, we saw that nonsimultaneous changes of flip-flop outputs caused undesirable ASM output signals, called output races. The solution proposed for this problem was to restrict the ASM to only one state variable change at each transition. This solution usually requires additional state variable flip-flops and next-state functions. Design is time-consuming and much more difficult if both output *and* transition races must be eliminated through state assignment. When using gates, we can simplify our state assignment criteria for eliminating output races. In order to eliminate races an output change should be *caused* by only one state variable change (even if more than one state variable is changing). State-assignment solutions to races are still cumbersome and time-consuming. The best output-race solution is not to have critical immediate outputs that will respond to the short pulses produced by output races. After all, if a circuit will respond to a short digital pulse produced by an output race, it will also respond to a short pulse produced by electric noise!

Sometimes an immediate output without output races *must* be generated. These outputs can be generated easily in two ways, without resorting to state assignment changes. In Fig. 6-5, a synchronizing register assures that output races will not appear on immediate outputs. These flip-flops only change at clock transitions after output races have disappeared. These "immediate" outputs are delayed one clock cycle from their occurrence on the ASM chart. The ASM chart may have to be modified to account for this delay.

A solution that does not delay outputs is shown in Fig. 6-6. Immediate-acting outputs are enabled only during the second half of the clock cycle, after the output

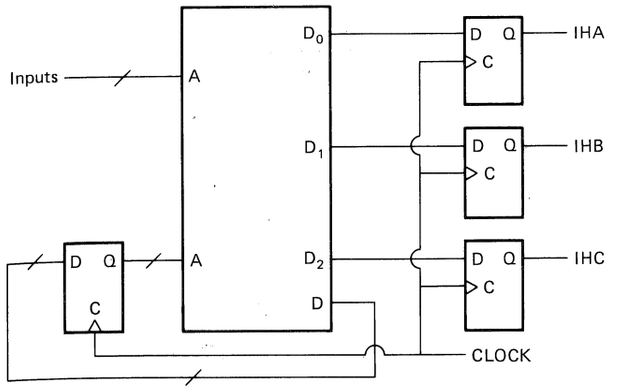


Figure 6-5 Resynchronizing outputs to eliminate races.

races have disappeared. The immediate outputs occur where they appear in the ASM chart, but only during the last half of each clock cycle.

In general, immediate outputs are best used only for slowly responding physical devices (like light bulbs) that do not act quickly enough to be affected by output races.

Hazards

Even in a race-free ASM, outputs may still have unwanted signals. In Fig. 6-7, a simple circuit generates an immediate output *IHO* from three state variables C, B, and A. The output *IHO* should be a 1 for state $C = B = A = 1$ and should also be a 1 for state $C = A = 1$ and $B = 0$. Since the transition between these two states involves the change of only one state variable, no output races can occur. Yet, the output on the timing diagram shown in Fig. 6-7 momentarily becomes 0, even though it should remain 1.

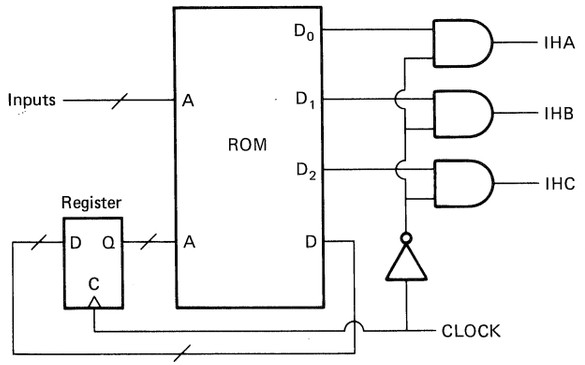


Figure 6-6 Gating outputs to remove races.

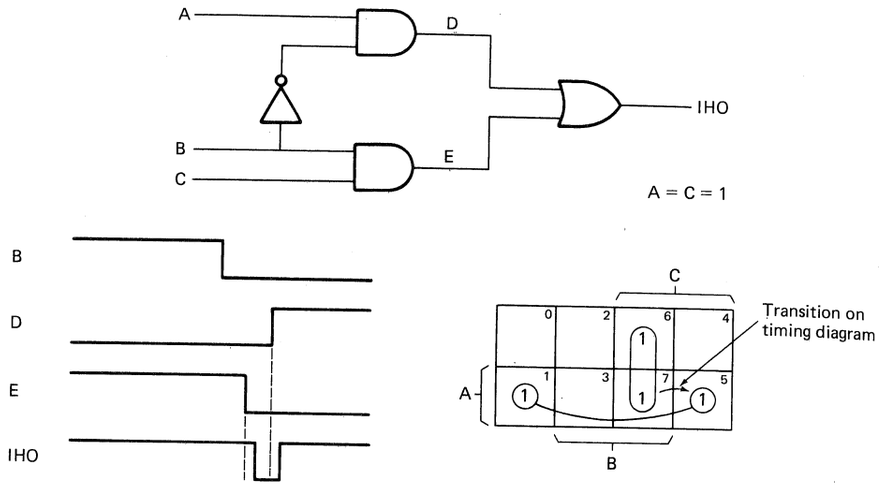


Figure 6-7 Logic circuit with hazard.

You'll notice that this extraneous pulse or *glitch* is caused by the extra delay introduced by the inverter in the circuit. Of course, all the gates and inverters will have different delays, and circuit operation will be unpredictable. Operation will depend on the individual logic circuits used. This erroneous signal does not depend on the state variable assignment. It is only a function of the particular gate implementation of the logic function. Worse yet, you can expect four kinds of hazards, one for each possible output transition, as shown in Fig. 6-8. Fortunately, it has been proved that a two-level logic (sum-of-products or product-of-sums) circuit is free of all hazards if it is free of hazards for the 1-to-1 transition.

It is necessary for us to know how to detect and correct hazards that occur for the 1-to-1 transition, shown in Fig. 6-7. In this circuit, state variable B acts to *select* one of two gates. Each gate implements a product function that is 1 for a particular set of inputs. The hazard occurs because function E returns to a 0 before function D becomes a 1. Thus, the output OR gate momentarily has all 0 inputs. This hazard-

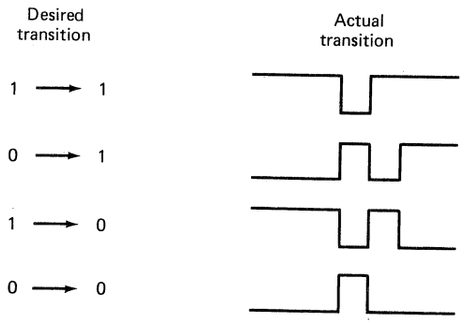


Figure 6-8 Four kinds of hazards.

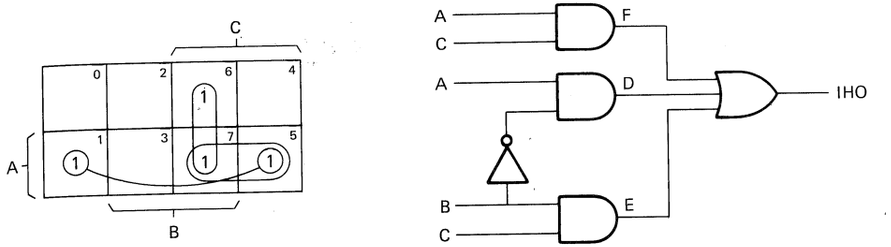


Figure 6-9 Circuit without hazard.

causing transition can be seen on the K map of Fig. 6-7. It is a transition between adjacent squares that are in different groups. In other words, it is a transition involving the change of only one variable between one product gate and another. To avoid this hazard, we must make sure that no allowable state transition changing only one variable "switches" between two different product functions. We must encircle such transitions in a single group on the K map, as shown in Fig. 6-9. No hazard occurs because a new product function generates intermediate variable F, which remains a 1 during the former hazard-causing transition. The output F of this new gate is a 1 for both states 5 and 7 and prevents *IHO* from returning to 0 even momentarily. Since this is a two-level (sum-of-products) circuit free of hazards for all 1-to-1 transitions, we can be confident it is free of all hazards for other transitions as well.

Another example is shown in Fig. 6-10. Looking at the K map for the output function shown, we discover two possible locations where a single variable change (adjacent squares) would "switch" from one product group to another. These are

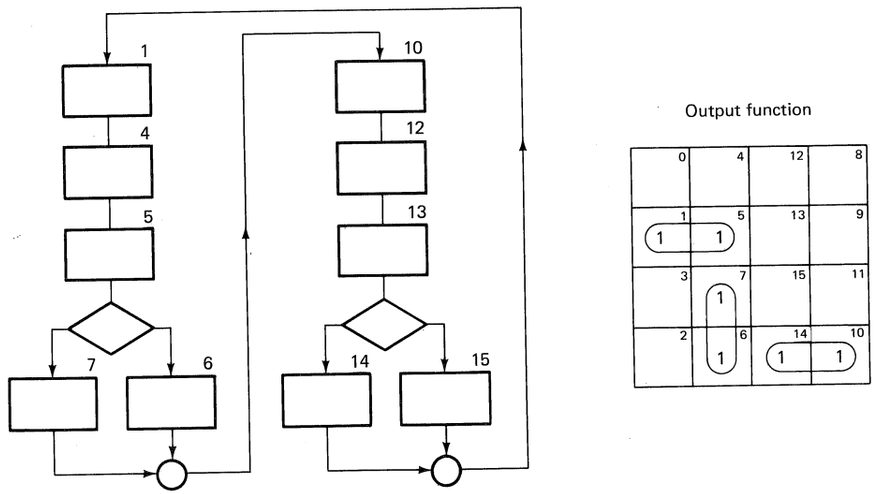


Figure 6-10 Hazard example.

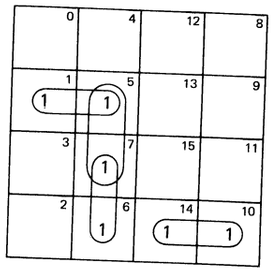


Figure 6-11 Hazard example solution.

between squares 5 and 7, and between squares 6 and 14. However, looking at the ASM chart, we discover that no transition can occur between states 6 and 14. The hazard-free implementation is shown in the K map of Fig. 6-11, in which both squares 5 and 7 have been concluded in a product function.

Hazard elimination is an easier design task than race elimination, although hazard elimination requires extra gates. As with output races, hazards will cause circuit malfunctions only if they appear on immediate outputs. The synchronizing technique of Fig. 6-5 or gating technique of Fig. 6-6 will eliminate hazards as well as races from critical immediate outputs.

ROMs and Glitches

The circuit designer has no control over the internal configuration of a ROM, other than specifying its contents. Hazards potentially exist in all ROM outputs. The output of a ROM *must* be assumed to be unpredictable while its address inputs are changing. Immediate outputs that must not have extraneous short pulses *must* be gated or synchronized. Since all state variables and inputs are used as ROM address bits, any change of inputs or state variables must be assumed to potentially affect all the outputs during such a change. This is true even if the ASM chart does not show explicit dependence of the output on those inputs or state variables that are changing. Thus, asynchronous inputs should *always* be synchronized before becoming ROM address inputs. State-assignment race solutions should not be attempted with ROM implementations.

It may seem that gate design is more flexible in some respects than ROM design. However, the ROM circuit design is, and must be, more structured. This "structure" actually reduces design time and cost, while making circuit operation more reliable. Also, the structured ROM design is often less expensive to build and repair. The "flexibility" of a gate design only increases design time and cost and greatly increases the chance for error. Many of the problems caused by races and hazards are especially vexing. These timing problems depend on the timing parameters of specific individual components and on unspecified asynchronous inputs. Although modern instrumentation, especially the logic-state analyzer, makes it possible to locate intermittent timing problems, such troubleshooting is still a time-consuming and expensive operation. The best design is a conservative design that does not allow the possibility of errors.

ASYNCHRONOUS ASMS

In the simple gate/flip-flop ASM, the clocked flip-flops act as delays to prevent the next state from reaching the gate inputs until after all current state outputs have stabilized. A new current state at the ROM address inputs causes a new next state to appear at the ROM data outputs. This new next state appears on the *D* inputs of the next-state register. This new next state does not appear on the flip-flop's outputs until they are clocked. The delay through the state register, caused by the clock period, ensures that the new next state is correct and stable. Since the next-state gates have an inherent propagation delay, could we use only these gate delays instead of a periodic clock to build a working ASM, as shown in Fig. 6-12? Figure 6-12 is a completely asynchronous ASM with no clock to control timing. A new current state at the gate inputs causes a new next state to appear at the gate outputs after a short delay. This new next state becomes the current state immediately. The next-state outputs are simply connected directly to the current state inputs. Since no clock exists, all inputs and outputs must be asynchronous.

Since gate delays are highly variable, we should expect extremely difficult design problems with transition and output races and hazards. In fact, design is so difficult that it is often divided into classes. One such division includes three classes: pulse, fundamental, and level mode.[1] In a *pulse-mode* circuit, each input must be a pulse (either from 0 to 1 to 0 or vice-versa). Pulses must be long enough to allow the circuit to operate and must not occur simultaneously or so close together that the circuit doesn't stabilize between them. In other words, one input changes from 0 to 1 to 0 (a pulse to 1), and a new current state becomes stable after some gate delay. Another input change may not occur until this current state is stable. A *fundamental-mode* circuit allows inputs to be levels, but transitions between levels must not occur simultaneously (nor too close together) on two or more inputs. Again, one input changes, either from 0 to 1 or from 1 to 0, and a new current state becomes stable after some gate delay. Another input change may not occur until this state is stable. A *level-mode* circuit allows any input transition at any time. A level-mode circuit is the most general of all and, of course, the most difficult to design. Pulse, fundamental, and level modes are methods of operation of a circuit. Obviously, any circuit can be excited by inputs

[1] Fredrick J. Hill and Gerald R. Peterson, *Switching Theory and Logical Design*, 2d ed., John Wiley & Sons, New York, 1974, chaps. 11, 13.

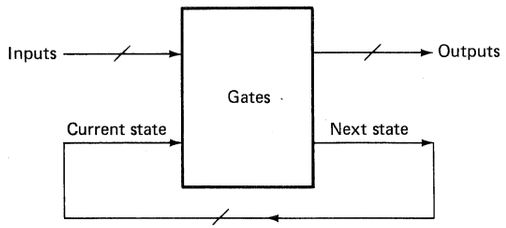


Figure 6-12 Asynchronous ASM.

that conform to these three operation modes. By restricting inputs in pulse- and fundamental-mode operation, we simplify the design process for an asynchronous circuit.

Because of the difficulties in designing asynchronous ASMs, they are seldom used when they can be avoided. Extremely simple and cost-sensitive logic circuits may have to be asynchronous (e.g., children's toys) in order to eliminate the cost of a clock. Even in very cost-sensitive circuits, the trend is away from asynchronous circuitry and toward a custom integrated circuit implementing a synchronous ASM (e.g., a TV game). Changes in the economics of production and design of circuits as well as the increased complexity of most digital circuits have reduced the importance of asynchronous circuitry.

Flip-flops themselves are asynchronous circuits for they do not contain an internal clock. For example, an edge-triggered D flip-flop is an asynchronous ASM with two inputs D and C and two outputs Q and $\overline{Q}$. Thus, integrated-circuit designers must be able to design classical asynchronous ASMs to implement flip-flops. Usually, a limited number of standard flip-flops are designed and their operation verified thoroughly. These standard flip-flop designs are used as building blocks for designing more complex ICs as synchronous circuits.

Although we will not discuss classical asynchronous ASM design, we will describe the way in which small asynchronous circuits are often designed to complement synchronous circuits. Small asynchronous circuits are commonly used to simplify designs and to implement functions difficult for synchronous circuitry.

SR Flip-Flops

Most simple asynchronous circuits are not designed by starting with the model of Fig. 6-12. Asynchronous circuits are usually designed using flip-flops as building blocks. Besides using edge-triggered JK and D flip-flops, another flip-flop, called the *SR* or *set-reset flip-flop*, is commonly used. This flip-flop is an unclocked flip-flop, which also happens to be the simplest example of the classical asynchronous ASM of Fig. 6-12. An SR flip-flop operates like the direct clear and direct preset inputs we used in an earlier chapter for system initialization.

The symbol for one type of SR flip-flop is shown in Fig. 6-13*a*. Both the S and R inputs normally rest at 0. With these inputs both 0, the flip-flop is stable, remembering the output resulting from its last change. If input S becomes a 1 while R remains a 0, the flip-flop will set so that $Q = 1$ and $\overline{Q} = 0$. Even if S now returns to 0, the flop will remain in the *set state*. If input R becomes 1 while S remains 0, the flop will reset so that $Q = 0$ and $\overline{Q} = 1$. The flip-flop will remember this *reset state*, even if R returns to 0. The condition $S = R = 1$ is not allowed.

An SR flip-flop is easily implemented with two gates, as is shown in Fig. 6-13*b*. The circuit is drawn so that it can be easily compared to Fig. 6-11. It is an asynchronous ASM because its next-state output is connected directly back to its current state inputs. Two NOR gates generate the next state from two inputs and the single current state bit. Assume that the circuit starts with $Q = 0$ and $\overline{Q} = 1$, and $YS = YR = 0$. Because Q and YS are both 0 inputs to gate A, $\overline{Q}$ will be 1. If $\overline{Q} = 1$ (regardless of YR),

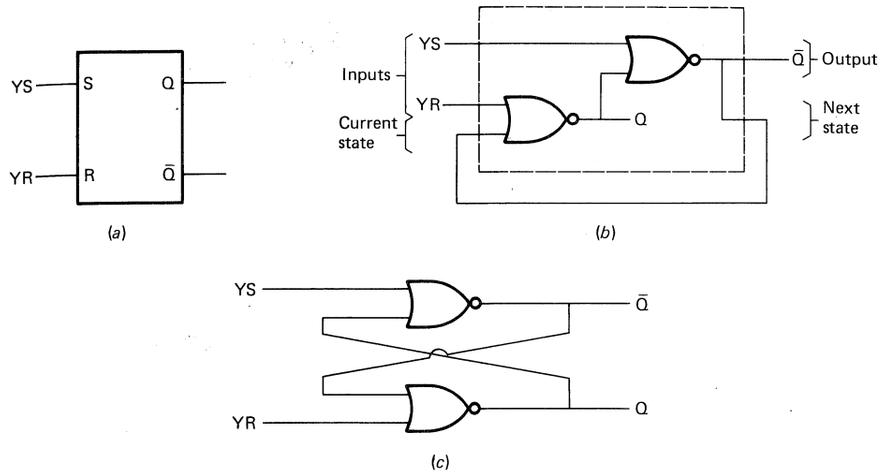


Figure 6-13(*a*) Symbol for SR flip-flop. (*b*) Implementation of SR flip-flop. (*c*) Implementation of SR flip-flop as commonly drawn.

then Q must be 0. Now, let $YS = 1$. This input forces $\overline{Q} = 0$ through gate A. $\overline{Q}$ is an input to gate B and forces $Q = 1$. Even after YS has returned to 0, $Q = 1$ will hold $\overline{Q} = 0$, which will in turn hold $Q = 1$ so that the flip-flop remembers its new state! An exactly analogous sequence describes the operation of the reset input. The SR flip-flop implementation is most often drawn as shown in Fig. 6-13*c*. A similar (but not identical) flip-flop can be made using NAND gates instead of NOR gates.

Although SR flip-flops can remember the last occurrence of two inputs, some kind of synchronization is required between the input sources to assure that both inputs do not simultaneously occur. This synchronization need not be electric. In Fig. 6-14, an SR flip-flop is connected to two mechanical switches. The flip-flop may be connected to an indicator in another room or to a synchronous ASM that controls door motion. It seems as though no synchronization exists between S and R inputs.

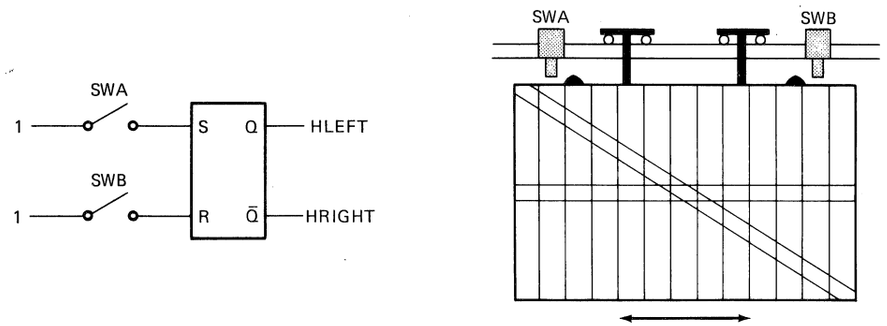


Figure 6-14 SR flip-flops controlled by door switches.

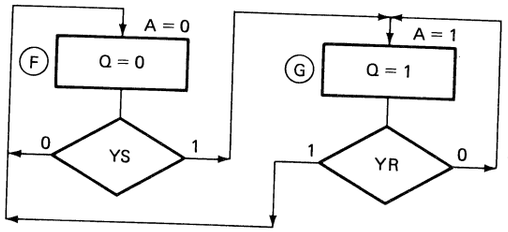


Figure 6-15 ASM chart for SR flip-flop.

However, the switches are mounted next to the sliding door so that only one switch can be activated at a time. This circuit changes state when the door is moved left or right and remembers the direction in which the door was moved.

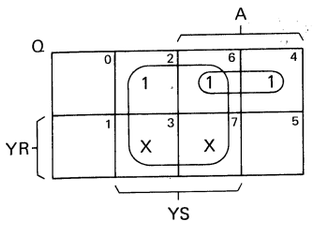
Asynchronous State Machine Design

Is there a regularized way to design asynchronous ASMs? An ASM chart for the SR flip-flop is shown in Fig. 6-15. The flip-flop is reset in state *F*. A reset pulse would not affect state or output; so no decision diamond is shown for *YR*. If *YS* became a 1, the flip-flop would set and go to state *G*. A *YR* input would send the flip-flop back to state *F*. Let's call the current state *A* and the next state *Q*. We can complete the Next-State Table 6-1 directly from the ASM chart of Fig. 6-15. The next-state function *Q* is placed on a K map in Fig. 6-16. Since inputs *YS* and *YR* may not be 1 simultaneously, we put don't cares in squares 3 and 7 of the K map. A standard sum-of-products reduction is shown in Fig. 6-16, along with a change to NOR gates using De Morgan's laws. Finally, the ASM is completed by connecting the next-state output *Q* back to the current state input *A*.

The ASM chart of Fig. 6-15 adequately represented the asynchronous ASM to be implemented. This will not always be the case for more complex asynchronous ASMs. The inputs of a synchronous ASM are sensitive to logic levels, rather than logic-level transitions. A decision made on input *YX* depends only on the state of *YX* at the time of decision. It doesn't matter when *YX* changed to its current level nor how many times *YX* may have changed before. In an asynchronous ASM, input transitions are very important. Input transitions cause the ASM to change state. At least some inputs of an asynchronous ASM must cause state changes when they change. No clock signal

Table 6-1 Next state for SR flip-flop

Current state	Inputs		Next state
A	YS	YR	Q
0	0	*X*	0
0	1	0	1
1	*X*	0	1
1	0	1	0



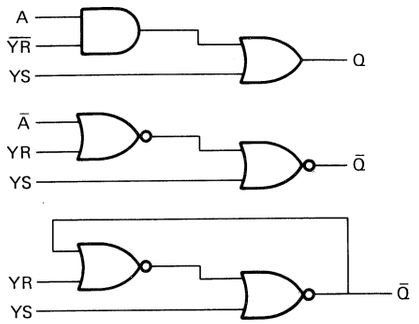
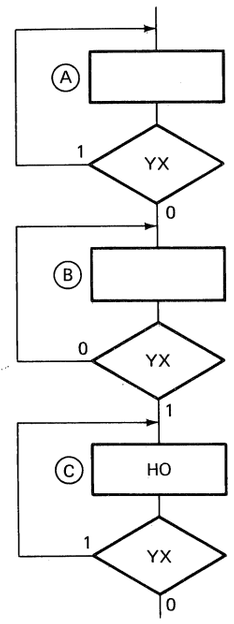


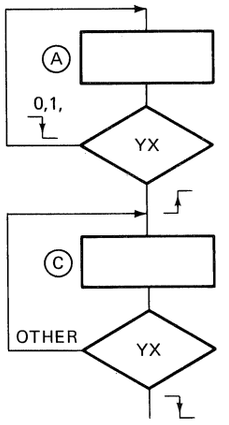
Figure 6-16 Implementation of SR flip-flop from ASM chart.

exists and *only* an input change can cause a state change. Some decision diamonds on an asynchronous ASM chart are more than decisions. At least one of the diamonds after each state box represents the occurrence of the input that actually causes the state change. This input takes the place of the clock.

How can we show a state change dependent on an input transition? Two alternatives are available to us. In Fig. 6-17*a*, output *HO* should be a 1 only after a 0-to-1 transition on input *YX*. Input *YX* is a 1 in state *A*. If input *YX* changes to a 0, the ASM proceeds to state *B*. States *A* and *B* are identical except for input *YX*. States *A*



(*a*)



(*b*)

Figure 6-17(*a*) Transition-sensitive ASM chart by adding states. (*b*) Transition-sensitive ASM chart by notation.

and B have identical values of state variables. What was formerly one state has been split into two state boxes on the ASM chart. Thus, this ASM chart includes the inputs as well as the state variables in its "states." Why? As long as we are in state *B*, we can be sure that $YX = 0$. The transition from $YX = 0$ to $YX = 1$ can be represented by the decision diamond leading to state *C*. Figure 6-17*a* is only a notational convenience. The number of state variables has not changed. Only the number of state boxes has increased! If this ASM had many inputs, you can see that the number of state boxes would become very large. An alternative notation avoids this problem. The ASM chart in Fig. 6-17*b* performs the same actions as the ASM chart in Fig. 6-17*a*. Transitions are now represented by arrows. Up arrows represent transitions from 0 to 1. Down arrows represent transitions from 1 to 0. Thus, the ASM changes from state *A* to state *C* if input *YX* makes a transition from 0 to 1. Four symbols may now be used with decision diamonds. These symbols are 0, 1, up arrow, and down arrow.

The notation of Fig. 6-17*a* is most convenient for classical asynchronous ASM design and produces a circuit like that in Fig. 6-12. Many small asynchronous ASMs are designed using edge-triggered flip-flops as building blocks. The notation of Fig. 6-17*b* is more convenient when designing asynchronous ASMs incorporating edge-triggered flip-flops.

Time-Oriented Design

The design of simple asynchronous ASMs may be considered to be time, rather than state, oriented. Often the input specification for such an ASM is a timing diagram or word description of the time order in which inputs may be expected. Edge-triggered flip-flops are often useful in implementing such designs. The simplest such use of an edge-triggered flip-flop is shown in Fig. 6-18. A single type D edge-triggered flip-flop

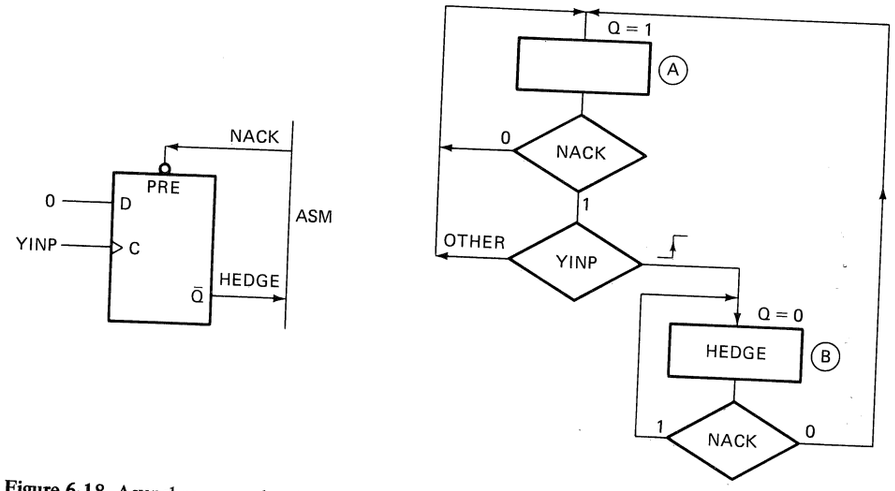


Figure 6-18 Asynchronous edge detector.

is used to detect a 0-to-1 transition on input *YINP*. This transition resets the flip-flop, which signals the ASM connected to it that a positive edge has occurred. The ASM acknowledges that the signal *HEDGE* has been received by momentarily asserting *NACK* in order to set the flip-flop to prepare to receive another edge. The *YINP* decision has been implemented with the transition-sensitive clock input of the flip-flop. The *NACK* decision has been implemented with the level-sensitive preset input.

Small asynchronous ASMs are commonly found in computer interfaces. Often, data and the destination address of that data are multiplexed on a common bus. In Fig. 6-19, a destination address from a computer is decoded from the common bus. The destination address allows the computer to select one peripheral among a group. If the computer peripheral device shown is to be the recipient of the data to be sent, *YADDR* will be 1 and *YASTB* (address strobe) will cause the ASM to change to the

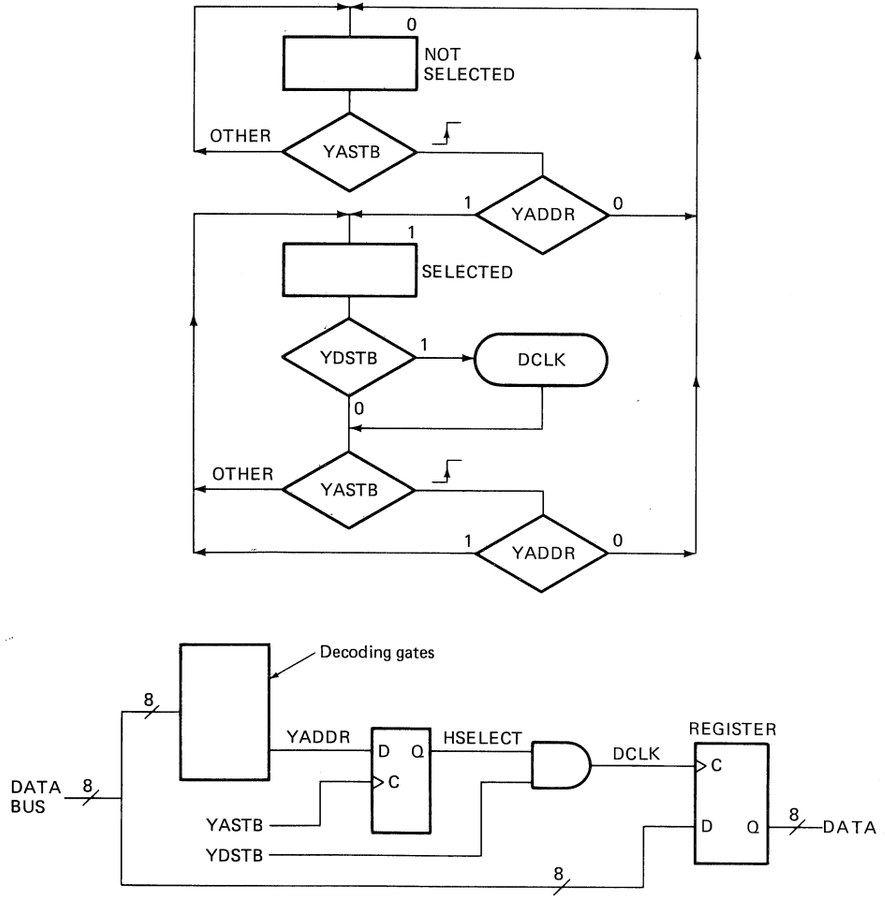


Figure 6-19 Asynchronous ASM for demultiplexing address selection and data from common bus.

selected state. Successive signals on *YDSTB* (data strobe) will cause data on the bus to be stored into this device's data register by *DCLK*. The signal *DCLK* is a conditional output of this ASM. Even though *DCLK* may be output several times, the ASM will not change state until *YASTB* makes a transition. This is indicated by the up arrow next to the *YASTB* decision. Only when the computer desires to send data to a different peripheral will it send another address and *YASTB*. Now *HSELECT* will be 0 if the new address selects a different peripheral, and *YDSTB* will not affect this peripheral's data register.

Asynchronous Design with Edge-Triggered Flip-Flops

Unfortunately, no regularized method exists for designing circuits like those in Figs. 6-18 and 6-19. Generalizations about the use of flip-flop inputs will help the partially intuitive task of designing these circuits.

Flip-flop inputs may be divided into three categories: edge-sensitive, delayed level-sensitive and immediate level-sensitive. Most flip-flops have one *edge-sensitive* input, the clock input. The clock input must be used to implement input-transition-caused state changes. For example, state changes in Fig. 6-19 are caused by positive transitions on input *YASTB*. Thus, input *YASTB* is connected to the flip-flop's clock input. *Delayed level-sensitive* inputs are those level-sensitive inputs that are sensed only at a clock transition. The *D*, *J*, and *K* inputs of flip-flops are delayed level-sensitive. These inputs can be used to select among branches of state changes caused by the clock input. For example, the input *YADDR* in Fig. 6-19 selects between alternate next states. The actual state transition is caused by *YASTB*, but *YADDR* selects the state to which the transition will occur. This is easily accomplished by connecting *YADDR* to the *D* input of the state flip-flop.

Most flip-flops have only one clock input. Often, the most straightforward circuit will require that a state flip-flop be changed by transitions on more than one input. The most straightforward edge-detector ASM chart is shown in Fig. 6-20. A positive edge on input *YINP* is detected by the circuit, and *HEDGE* is asserted. The circuit is reset to look for another edge by a positive transition on input *YACK*. Since only two

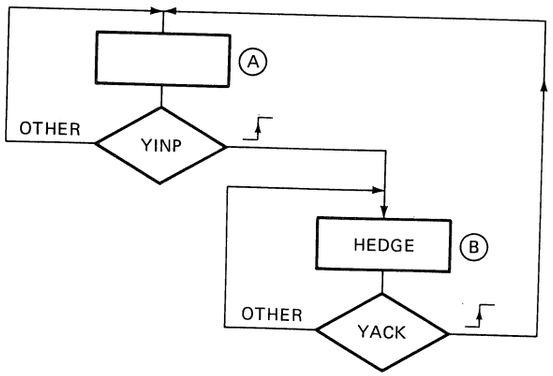


Figure 6-20 Straightforward edge detector.

states are needed, one flip-flop would be required, but that flip-flop would have to respond to transitions on two edge-sensitive inputs. Since such a flip-flop is not readily available, the ASM chart in Fig. 6-18 was devised as an alternative to Fig. 6-20. The *immediate level-sensitive* preset input of the flip-flop was used to set the circuit to look for another input edge. Although this eliminated the need for a flip-flop with two edge-sensitive inputs, using this immediate acting input had an undesirable effect as well. An additional decision diamond appeared after state *A*. As long as input *NACK* is low, the circuit is prevented from changing to state *B*, even though *YINP* may make a 0-to-1 transition. Immediate-acting inputs, like clear and preset, override the action of the clock input. An alternative to using direct-acting inputs is switching the clock source with gates among as many inputs as required. This can be a dangerous practice, and such circuits should be carefully designed. Hazards and races could erroneously clock the flip-flop.

The design of asynchronous ASMs using time-oriented design and edge-triggered flip-flops may be summarized as follows.

1. Draw a timing diagram showing the order in which inputs will occur and outputs should occur. Draw multiple timing diagrams if alternate sequences are possible.
2. Draw an ASM chart from the timing diagram, including edge-sensitive transitions. Use as *few* edge-sensitive transitions as possible to minimize the possibility of needing flip-flops with more than one edge-sensitive input.
3. Try to implement the ASM chart with edge-sensitive D or JK flip-flops. Use clock inputs to change states caused by input transitions. Select among next states with *D*, *J*, and *K* inputs and appropriate gating. Use clear and preset inputs to change states when clock inputs are exhausted. Check your use of immediate-acting inputs against the timing diagram to ensure that these inputs do not override state changes to be caused by the clock input. Go back and change the ASM chart to reflect the override action of the immediate-acting inputs.
4. If no circuit can be found because of insufficient edge-sensitive inputs and timing incompatibilities of immediate-acting inputs, return to step 2 and try to redraw the ASM chart to eliminate these problems.
5. If you still can't find a solution, try switching clock inputs, classical asynchronous design,[1] or a synchronous circuit.

Monostables

A logic circuit often seen in asynchronous designs is a *monostable multivibrator* or simply a *monostable*. Shown in Fig. 6-21, a monostable has one or more edge-sensitive inputs which cause this circuit's output to set ($Q = 1$, $\overline{Q} = 0$). The monostable resets itself after a fixed time delay. This delay is proportional to the product of the values of the resistor and capacitor connected to the monostable. Unfortunately, even if the resistor and capacitor were ideal, this "fixed" time delay would change with the monostable's power supply voltage and temperature. Also, actual capacitors and resistors

[1] Hill and Peterson, *Switching Theory and Logical Design*, chaps. 11, 13.

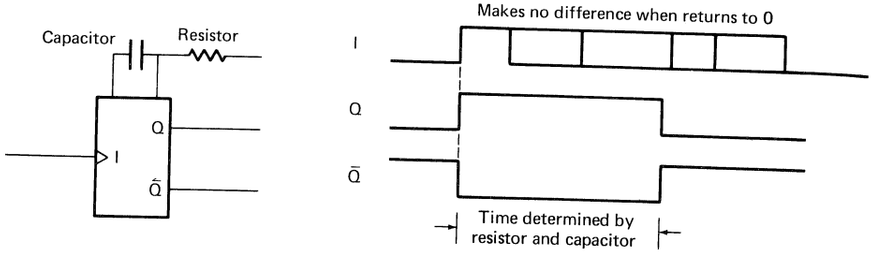


Figure 6-21 Monostable.

may have large initial tolerances and may change value considerably with time, temperature, and humidity. For these reasons, monostable timing can seldom be regarded as accurate. Monostable circuits must be designed and adjusted to account for wide variations in delay times. Monostables also "magnify" noise. If a monostable is accidentally started by a short electric noise pulse, the monostable's output will provide a normal-length pulse, which will easily operate other circuitry connected to it.

Nevertheless, monostables sometimes prove useful, especially in asynchronous circuits. In Fig. 6-19, address and data information may have traveled a long distance

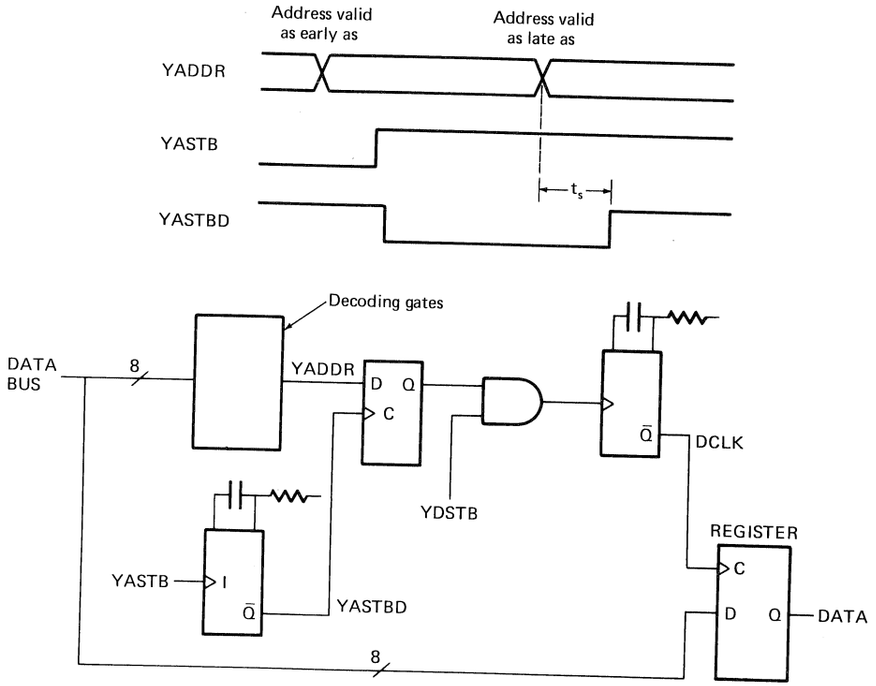


Figure 6-22 Deskewing with monostables.

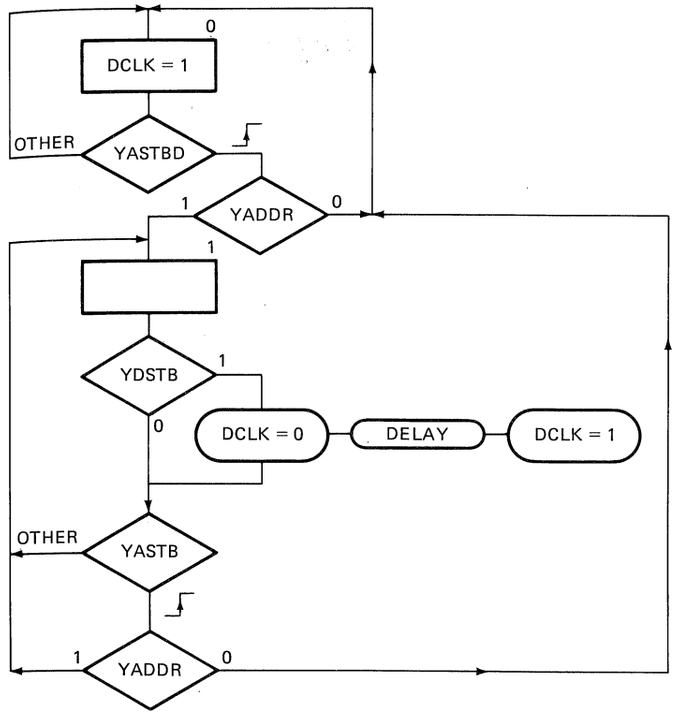


Figure 6-23 ASM with monostable delay.

over various wires and through various gates. In fact, not only will the bits of the address arrive at different times, but the address strobe may arrive before or after different address bits. This variation in arrival times is called data skew and is analogous to clock skew. The address and data strobe must take place only after data have arrived and the setup time of the flip-flops has been satisfied. Monostables are used in Fig. 6-22 to delay the strobes by a time at least equal to the worst time skew expected for signals traveling down the bus from the computer. The timing diagram in Fig. 6-22 shows the address monostable is adjusted so that it will clock the state flip-flop after its setup time, even if the address arrives as late as could be caused by other circuitry. Monostable operation may be indicated in an ASM chart by a DELAY box, as shown in Fig. 6-23. This ASM chart is carefully drawn to make it clear that the *DCLK* monostable affects only the output *DCLK* and will not delay a state transition.

Switch Debouncing

When a mechanical switch closes, its metallic contacts bounce against each other hundreds of times before coming to rest. Several tens of milliseconds may elapse before the contacts come to rest, as shown in Fig. 6-24. The resistor connected to logic "0"

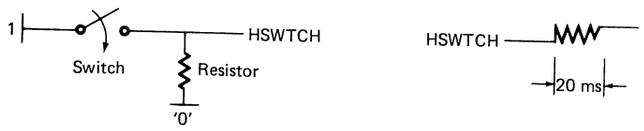


Figure 6-24 Switch bounce.

assures that the signal *HSWTCH* will be a 0 when the switch is open, while the mechanical switch connected to logic 1 causes *HSWTCH* to become 1 when the switch is closed. Obviously, the signal *HSWTCH* is not a suitable input for an ASM. A single operation of the mechanical switch may look like hundreds of transitions to the ASM. Logic circuits used to rectify this problem are called switch debouncers. The oldest and simplest debouncing circuit is shown in Fig. 6-25. The first closure of the switch after it is operated causes the SR flip-flop to set. Additional bounces of the switch are of no consequence because the SR flip-flop simply remembers the first bounce. When the switch is operated back to its original position, the reset input performs the same function. Although this circuit is the simplest logic circuit for debouncing, it does require a switch with two contacts, rather than one, and one additional interconnection wire to the switch. Since the switch and the interconnection wiring may be much more expensive than the logic circuitry, we'd like to find a circuit that will debounce a single contact switch.

The circuit of Fig. 6-26 uses two monostables to debounce a single contact switch. One or the other monostable will set first, depending on the direction the switch is thrown. (Both should set eventually as the bouncing logic signal will be changing in both directions.) The monostables reset after a time delay designed to be greater than the time required for the switch contacts to stabilize. At that time the correct stable state of the switch is clocked into the D flip-flop. Some monostables have both negative and positive edge-triggered inputs, so that only one monostable would be needed. Besides the disadvantage of additional circuitry, the monostable circuit introduces a delay between switch operation and recognition. Unlike the SR flip-flop circuit which operates on the first switch closure, the monostable circuit must wait until bouncing has stopped. This delay is usually not significant but may be if, for example, the switch and logic circuit must precisely control the position of a moving object.

The monostables of Fig. 6-26 provide a time delay. A clock and counter also provide a time delay. You should expect that a synchronous ASM could be built to

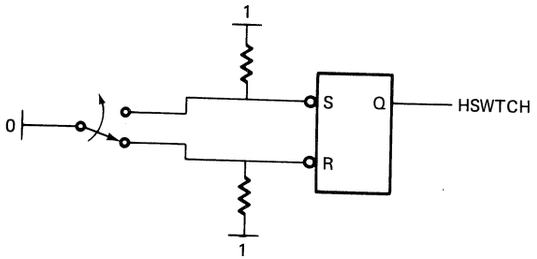


Figure 6-25 Switch debouncing with SR flip-flop.

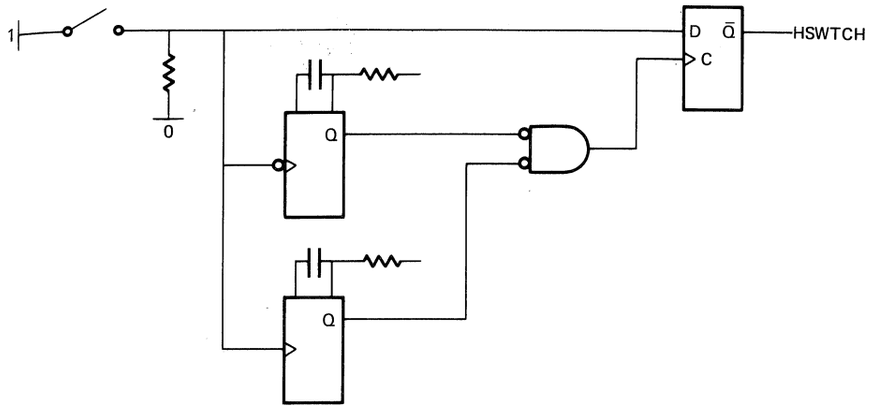


Figure 6-26 Switch debouncing with monostables.

perform the same debouncing function. In fact, four such ASMs are available as a preengineered single integrated circuit just for the purpose of switch debouncing.

PROBLEMS

6-1 (*a*) Assign states to the ASM of Fig. P6-1 so that transition races cannot occur. *All* inputs are asynchronous.

(*b*) Compare the complexity of part (*a*) with the complexity of a circuit using a synchronizing register instead of state assignment.

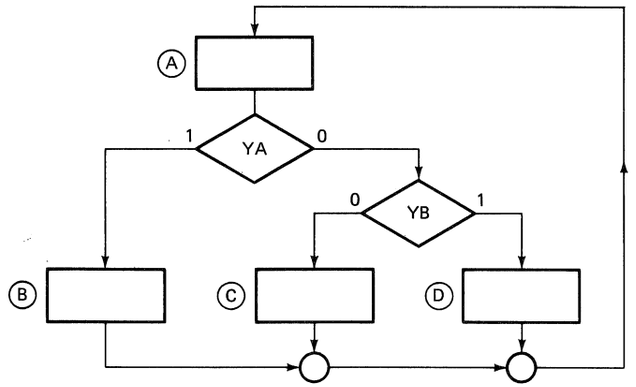


Figure P6-1 ASM chart for Prob. 6-1.

6-2 (*a*) Find the minimal gating for the output function in Fig. P6-2.

(*b*) Find the minimal hazard-free output gating for the output function in Fig. P6-2.

(*c*) Compare the complexity of part (*b*) with a circuit, using the result of part (*a*) and an

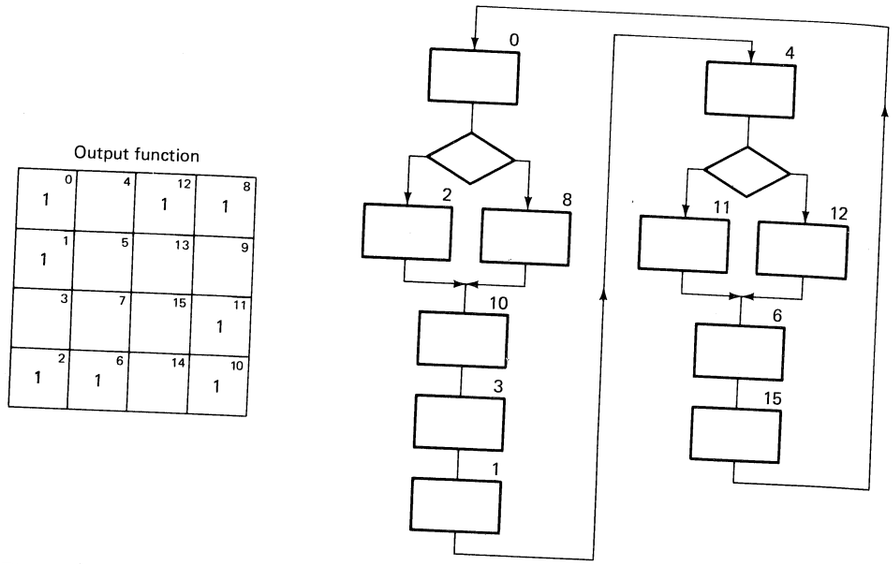


Figure P6-2 Output K map and ASM chart for Prob. 6-2.

output synchronizing register. Assume that delaying this output one clock period is of no consequence.

6-3 (*a*) The circuit in Fig. P6-3 uses 7474 flip-flops. All gates have propagation delays identical to the 7400. The specifications of these parts were given in the problems for Chap. 5. Draw a timing diagram showing the longest time during which output glitches could appear. Include the clock and the state variables on your timing diagram.

(*b*) Add a synchronizing register of 7474s to the outputs of Fig. P6-3. Use a 7404 to provide an inverted clock for this register to halve the output delay. The delay in the resynchronized outputs must be less than one half clock period. Draw the asymmetric clock signal needed to reduce the delay through the output register as much as possible. Use the results of part (*a*) and a 200-ns clock period.

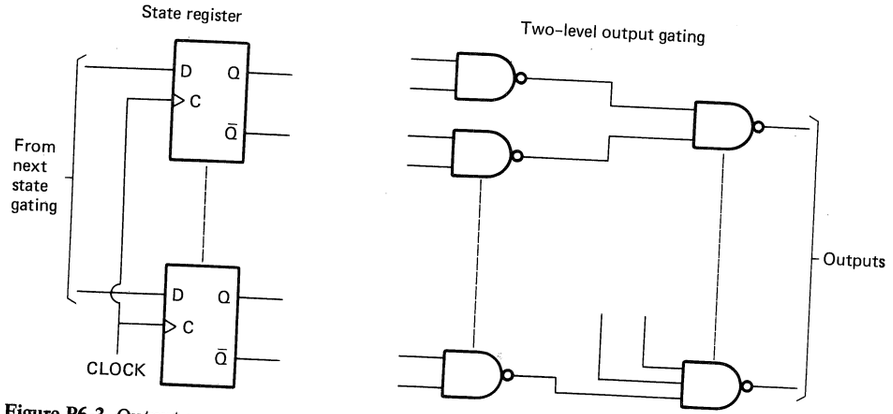


Figure P6-3 Output races.

6-4 The ASM chart of Fig. P6-4 has one asynchronous input *YX*. Complete the timing diagram in Fig. P6-4. Fill in the states and outputs.

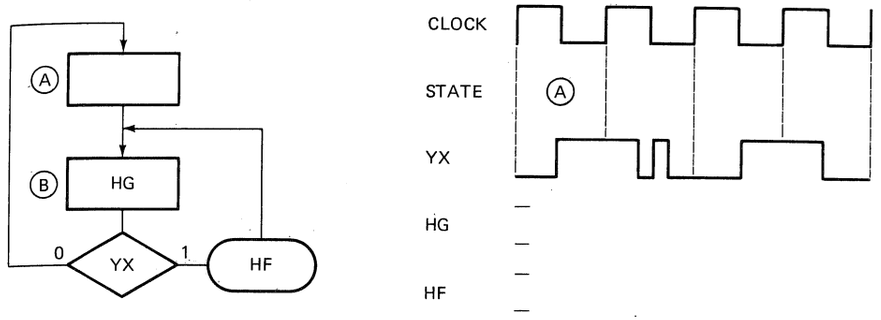


Figure P6-4 Synchronous ASM.

6-5 Implement an SR flip-flop using NAND gates. Describe its operation with a timing diagram.

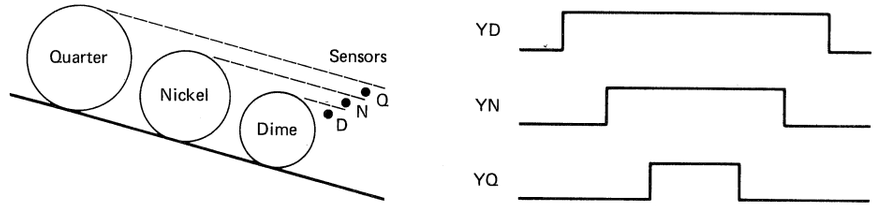


Figure P6-5 Coin sensor and timing diagram for quarter.

6-6 (*a*) The coin sensor in Fig. P6-5 is part of a vending machine. As each coin rolls down a chute, it covers one or more optical sensors, breaking a light beam and causing that sensor's outputs to become a 1. The timing diagram for a quarter is shown. Draw the timing diagrams for a nickel and a dime. Design an *asynchronous* ASM that has three outputs corresponding to the three coins and three inputs corresponding to the three sensors. The ASM's outputs should change to 1 to indicate the coin that rolled past the sensor. The output corresponding to the coin detected should remain asserted until another coin rolls past the sensor. The output may be momentarily incorrect while the coin is rolling past the sensors.

(*b*) Modify the circuit of part (*a*) so that incorrect outputs do not appear even while the coin is rolling past the sensor.

6-7 Design an *asynchronous* controller for the tram of Prob. 1-5. Use a monostable for the 10-s stop delay.