We hope that a program of study using this text will give the student or professional a firm and *real* understanding of the principles of modern digital system design. No book or course of study can substitute for experience, but an understanding of the basic principles is essential for gaining that experience. We believe that this book plainly presents the basic principles upon which modern digital system design is based. It presents these principles in a realistic framework that ensures an understanding of their applicability.

We'd like to acknowledge the assistance of Stella Sears, Linda Keener, Charles Small, George Mineah, Steve Curtis, and Margaret Wiatrowski.

Claude A. Wiatrowski
Charles H. House

LOGIC CIRCUITS AND MICROCOMPUTER SYSTEMS

CHAPTER

ONE

THE ALGORITHMIC STATE MACHINE

The solutions to many engineering problems take the form of a sequence of specific actions. In fact, much of our own life can be described by such a sequence. For example, when we prepare breakfast for ourselves, we follow such a sequence. First, we decide whether we have enough time to fix breakfast so as not to be late for work. If we don't have enough time, we will stop at McDonald's for breakfast. If we do have enough time, we look for all the ingredients needed. If we don't have everything, we'll also have to go out for breakfast. Next, we put a pan on the range, turn up the heat, beat a couple of eggs, and pour them into the pan. Every few seconds we stir the eggs and, at the same time, check to see if they are done to our taste. When they're done, we scoop them onto a plate and eat them. Of course, we would also be performing other sequences of actions simultaneously to ensure that coffee and toast would arrive on the table with our eggs.

You'll notice that the above sequence of actions has two important attributes. First, the sequence depends on time. Actions occur in a particular time order. In fact, some statements of action mention time specifically. For example, we stirred the eggs every few *seconds.* The second important attribute is that alternate actions were performed depending on some decision. For example, we either ate at McDonald's or fixed our own breakfast depending on whether or not we had the ingredients for making breakfast.

The control of traffic at an intersection is an example of an engineering problem that is solved by a sequence of actions. In this case, the sequence of actions is the lighting of traffic signals in a sequence designed to safely control the flow of traffic. Figure 1-1 shows a simple traffic intersection. The letters N, S, E, and W will stand for the traffic lights controlling northbound, southbound, eastbound, and westbound traffic, respectively. In the simplest intersection, the lights controlling north and

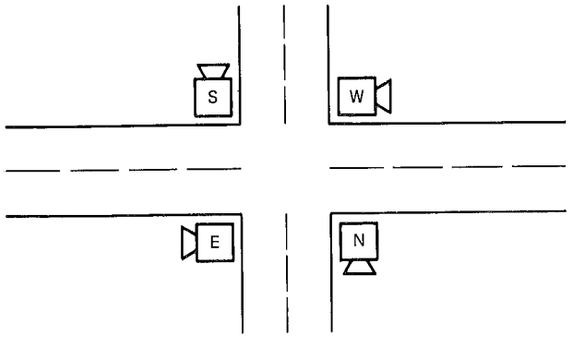


Figure 1-1 Traffic intersection.

southbound traffic will always have identical indications, as will the lights controlling east and westbound traffic.

Since we have to start somewhere, let's pick a safe traffic situation that we know can occur. Let the north and southbound lights be red and the east and westbound lights be green. Next, the east and westbound lights will turn yellow in preparation for changing the flow of traffic through the intersection. The east and westbound lights will turn red as the north and southbound lights turn green. After north and southbound traffic is allowed through the intersection, the north and southbound lights will turn yellow in preparation for another change in traffic flow. Finally, the north and southbound lights will turn red, and the east and westbound lights green. Since this is the condition in which we originally started, we can repeat the described sequence indefinitely.

You should notice that this description of the operation of a traffic light is difficult to follow. It is difficult to visualize the operation of even a simple traffic intersection from such a cumbersome description. For this reason, a sequence of actions is often described by a diagram called a flowchart.

Figure 1-2 is a flowchart for the simple traffic intersection that was previously described. The rectangles are called action blocks, and the actions to be performed are written in these blocks. The action blocks are connected by lines with arrows indicating the sequence in which the actions are to be performed. Often the description of the action is condensed to make the diagram smaller and more easily comprehended. In this case, the notation *NS RED* means that the northbound and southbound lights are set to red. This flowchart explicitly states that the red/green conditions will last for 20 s, while the red/yellow conditions will last for 5 s. Flowcharts are often drawn without time specifications. In this latter case, all action blocks are assumed to take the same amount of time. This time interval must be specified somewhere else and is often called the *CLOCK* interval. You can see that, if we could build a machine that would follow the sequence of Fig. 1-2, that machine could be used to control traffic lights at an intersection. In fact, such a device is called a sequential digital machine, and the design of such a machine with electronic components will be studied in this book. The *controlling* machine and the actual light bulbs of the *controlled* traffic signals are considered separate parts of one system. The actions are also called *OUT-*

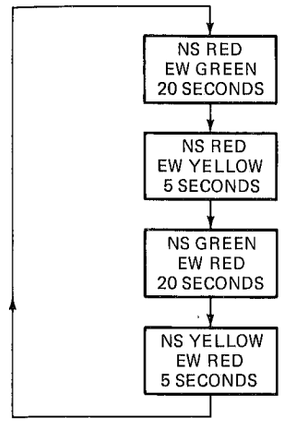


Figure 1-2 Flowchart for traffic intersection.

PUTS since they represent commands to perform an action sent from the sequential machine controller to the controlled traffic lights.

The traffic intersection we've described controls traffic efficiently when the amount of traffic on both streets is approximately equal. If the north-south street had more traffic than the east-west street, the north and southbound lights could have simply been given longer green intervals than the east and westbound lights.

An Algorithm with Inputs

Next, consider an intersection where the north-south street is a major thoroughfare that musn't be stopped unnecessarily. Furthermore, the east-west street is the entrance to a county fairground that has no traffic most of the time. However, when the fairground road is in use, a large number of east and westbound vehicles must be safely moved across the north-south street. One solution to this problem is to install traffic sensors in the pavement of the east-west street, as shown in Fig. 1-3. The traffic signals

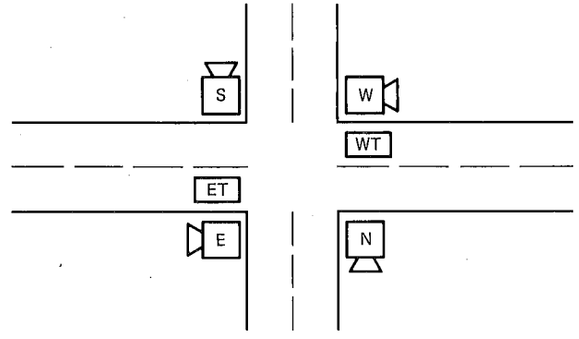


Figure 1-3 Traffic intersection with eastbound and westbound traffic sensors.

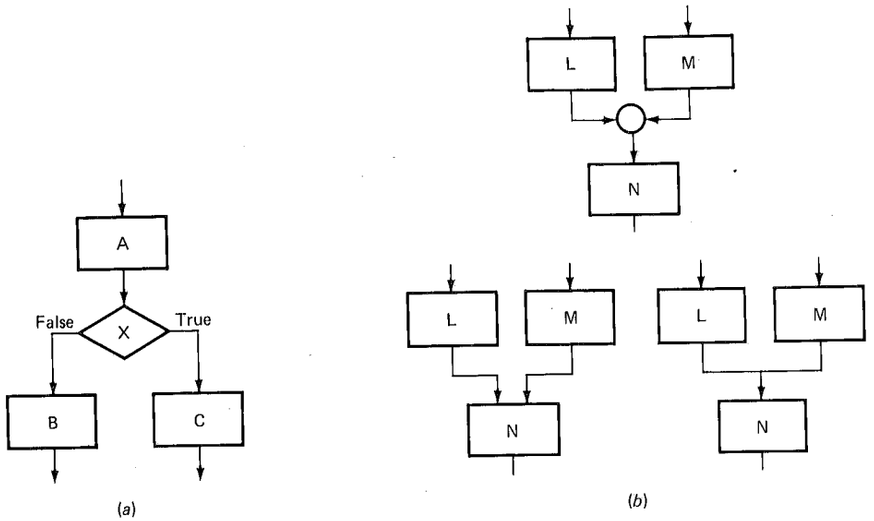


Figure 1-4 (*a*) Conditional branching; (*b*) joining branches.

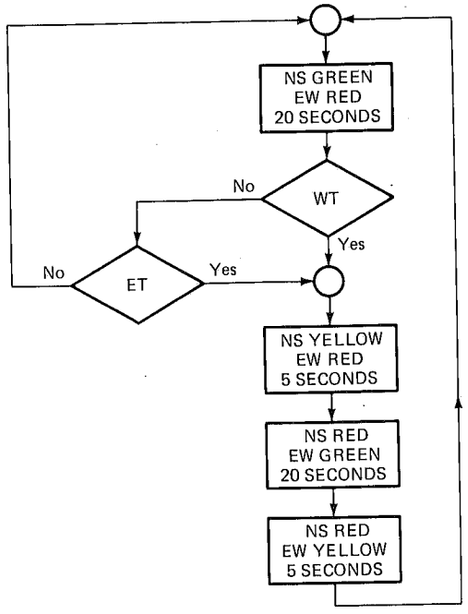


Figure 1-5 Flowchart for traffic intersection with traffic sensors.

normally show green to the major north-south thoroughfare and red to the east-west street. North-south traffic will be stopped only when a vehicle is waiting on the east-west street.

Notice that the solution to this traffic control problem involves a decision to be made based on the condition of the east and westbound traffic sensors. Figure 1-4*a* shows the diamond symbol used on a flowchart to indicate a decision. In Fig. 1-4*a*, action *A* is followed by action *B* only if *X* is false, while action *A* is followed by action *C* only if *X* is true. The decision diamond guides the sequence of actions from *A* to either *B* or *C* depending on the state of *X*. Since the decision merely guides the sequence of actions and is not an action itself, there is no time interval associated with the decision diamond.

If we've split the sequence of actions into two alternative paths, we will probably have to join alternative paths. Figure 1-4*b* shows three notations used in flowcharts to indicate that action *N* will be the next to occur after either action *L* or *M*.

Now we can draw the flowchart for the intersection of Fig. 1-3. Figure 1-5 incorporates two decisions. If neither the east nor westbound traffic sensors indicate a vehicle, the north-south lights will remain green. However, if either sensor indicates a vehicle present, the traffic lights will sequence to allow it to cross. Since the condition of the traffic sensors is sent to the controller that will implement the flowchart of Figure 1-6, these conditions are usually called *INPUTS*. Thus, the electronic circuits we will learn to design have *INPUTS*, on which decisions are based, and *OUTPUTS*, which perform actions. In addition, these circuits have a *CLOCK*, which times their sequencing.

Another Algorithm

Start with the intersection of Fig. 1-1, controlled by the flowchart in Fig. 1-2. Now, we'd like to add an input to allow an emergency vehicle to safely pass through the intersection by stopping traffic in *all* directions. Ignoring the electronic details of the vehicle's radio link, we'll assume that an input called *EMER* exists which, when true, should cause all lights to turn red. In Fig. 1-6 the input *EMER* is tested between every change in the traffic lights. If *EMER* is false, the lights function normally. If *EMER* is true, the lights are all set to red by a new action block that did not appear in the flowchart of Fig. 1-2. After all lights are red, the *EMER* input is tested every 5 s. When *EMER* becomes false (i.e., the emergency vehicle has passed), the normal traffic sequence is resumed.

This solution has one serious drawback. If the emergency vehicle driver pushes his button *just after* one of the 20-s actions is started, almost 20 s will elapse before the lights turn red. Figure 1-7 eliminates this problem by breaking up each 20-s interval into four 5-s intervals. Now, at most, 5 s can elapse between pushing the emergency button and all the traffic lights turning red. Because all action blocks in Fig. 1-7 take the same amount of time (5 s), no time notations are made in the blocks. The notation *CLOCK = 5 SECONDS* at the bottom of the figure indicates the time of each action block in the figure.

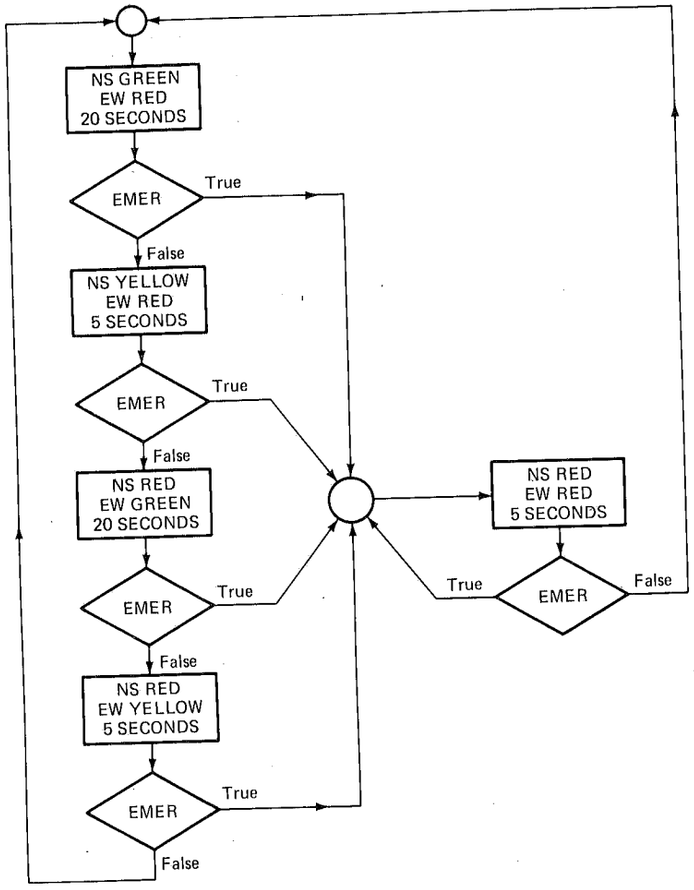


Figure 1-6 Flowchart for intersection with emergency vehicle input.

An Algorithm Definition

The sequence of decisions and actions we have been describing is called an algorithm. Algorithms may be described by plain text, flowcharts, tables, or any other appropriate technique. Any useful algorithm, no matter how it is represented, has certain important characteristics. First, an algorithm must have only a finite number of steps. Note that the traffic light algorithm meets this criteria, even though the entire traffic controlling algorithm repeats forever. The simplest traffic algorithm used only four steps to achieve its solution, even though the solution was to be repeated indefinitely. Second, each step (action or decision) in an algorithm must be precisely and unambiguously defined, as must the order of performing the steps. Third, an algorithm must have one or more outputs. Finally, an algorithm may or may not have inputs,

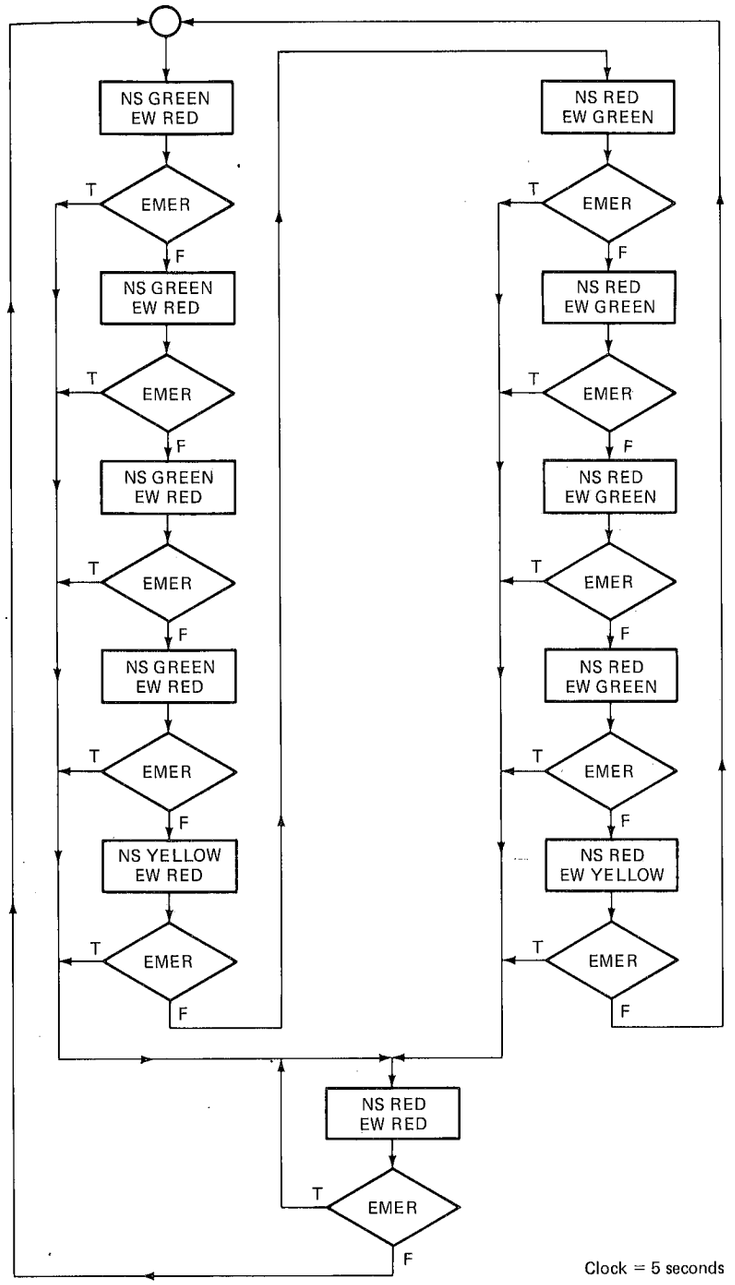


Figure 1-7 Flowchart for intersection with emergency vehicle input with a maximum delay of 5 s.

which it uses in its operation. Although the simplest traffic light controller discussed previously would be said to have no inputs, it does have *time* as an input. Because time is ordinarily an important part of most algorithms we'll be using, it is usually implicitly assumed to be an input.

DIGITAL SOLUTIONS

The task of the engineer is to discover and implement solutions to a problem. We've already discussed a large class of solutions to engineering problems. These solutions are called algorithms, and we have described them with flowcharts. The task of implementing these solutions remains. Not only must we be able to describe the method of controlling traffic at an intersection, but we must also be able to design and build an electronic device to actually perform the function of controlling traffic. One characteristic of all the algorithms we've discussed stands out. Every output or input we've used can be in only two conditions. The green westbound traffic light can be either on or off. The eastbound traffic sensors indicate either the presence or absence of traffic. Algorithms with this characteristic (all inputs and outputs may be in only two conditions) may be implemented by *digital* circuits. Since digital circuits need only recognize and generate two alternative conditions, they are less expensive to fabricate than circuits that must distinguish between hundreds of conditions. In fact, digital circuits that may be used in the solution of any algorithm are mass-produced inexpensively. As we'll see in later chapters, clever techniques exist to design digital implementations of algorithms that are not digital by nature. This is often done to take advantage of inexpensive and reliable digital circuit components.

Digital Signals

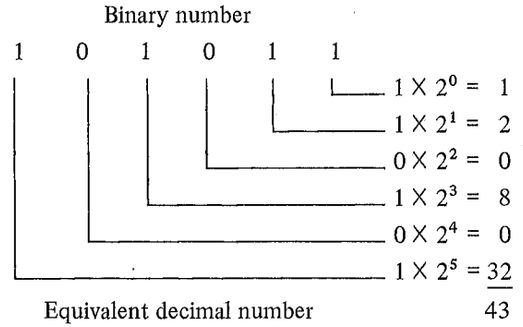
Digital circuits are connected to each other and to inputs and outputs with wires that carry digital signals. These signals are electrical voltages that represent the two conditions that the signal may assume. For example, the output wire to the green westbound traffic light may have two conditions. If the light is to be turned on, the wire will transmit a signal of 3.0 V, and, if the light is to be turned off, the wire will transmit a signal of 0.4 V. In real digital circuits, each condition is represented by a range of voltages. By using a range of voltages, the precision required in the circuitry is reduced. For example, the light turned on may be represented by any voltage greater than 2.4 V, and turned off by any voltage less than 0.8 V.

Describing digital signals by referring to voltages is very cumbersome. In addition, different *families* of digital circuits may use different voltages. The two digital signal conditions are often called 1 and 0 both for convenience and to make their description independent of any specific family of circuits. The condition or state specified by 1 usually corresponds to the more-positive voltage, and the state specified by 0 to the less-positive voltage. One common correspondence of condition, voltage, and state is:

Westbound green light	Voltage	State
On	≥ 2.4	1
Off	≤ 0.8	0

Binary Numbers

Digital circuits must often represent more than two states. Combinations of 1s and 0s can be used to represent integer numbers. Such a representation is called a binary number. Each digit of the binary number is given a weight equal to a power of 2:



This is exactly analogous to decimal numbers, in which each digit is given a weight equal to a power of 10. Just as in the decimal system, the rightmost digit is the least significant digit, with the smallest weight. In a binary system, this is called the least significant bit (LSB). If we can represent integer numbers as combinations of 0s and 1s, we should be able to count in binary. The binary counting sequence and its decimal equivalent are:

Binary				Decimal
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

Negative Numbers

One way to represent negative binary integers is called *two's complement*. Consider the 3-bit binary integer 101. This binary number represents the decimal number 5. We want to find a way to represent -5. The next bit to the left of this binary number would have a weight of 2^3 or 8. Change this weight to -8. This does not change the

representation for +5, which is still 0101. The number 1011 has a decimal equivalent, which is found to be:

1	0	1	1		
				1×2^0	= 1
				1×2^1	= 2
				0×2^2	= 0
				$1 \times (-2^3)$	= -8
Equivalent decimal number					-5

Thus, we have found a way to represent the number -5. In fact, this method will represent other negative numbers. The counting sequence for two's-complement numbers is:

Binary				Decimal
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7

All positive numbers start with a 0 bit, while all negative numbers start with a 1 bit. The binary equivalent of -4 is identical to the binary equivalent of +12. You must know that the number is supposed to be two's complement to interpret it correctly. An easy way to find the negative equivalent of a binary number is to change all the 0s to 1s and 1s to 0s and, then, to increment the resulting number by one. For example, to represent the number -3, we start with the binary equivalent for +3:

0011

Then, we change all the 1s to 0s and 0s to 1s:

1100

Finally, we increment the result by one (count up one):

1101

This is the two's-complement representation of -3. You will see in a later chapter that the two's-complement representation simplifies binary arithmetic.

Representing Time

As previously mentioned, time is an important part of most algorithms. We would like to have a way of representing the manner in which digital signals change with time. Two common representations exist. The first is called a *timing diagram* and is shown in Fig. 1-8. Six digital signals are shown on this diagram, corresponding to the outputs of the simple traffic light controller. Each signal is represented as a horizontal line which may take two positions. The upper position corresponds to the 1 condition, and the lower to the 0 condition. Time is plotted horizontally, increasing to the right. For example, at 25 s, the NS RED turns on, the NS YELLOW turns off, the NS GREEN is off and stays off, the EW RED turns off, the EW YELLOW is off and stays off, and the EW GREEN turns on. By plotting several signals on the same timing diagram, the time relationships among all these signals can be described. A timing diagram for a real operating circuit can be viewed by connecting an instrument called an *oscilloscope* to the circuit. A newer instrument called a *logic-timing analyzer* also displays timing diagrams.

A timing diagram becomes cumbersome for a large number of signals or a long period of time. For that reason, digital signals are often represented in tabular form. Many (but not all!) digital circuits have the characteristic that signals only change at fixed time intervals, called *clock* intervals. The traffic controller of Fig. 1-8 has this characteristic. Signals only change at 5-s intervals. For that reason, we can represent all the information contained in the timing diagram of Fig. 1-8 by listing the states of all the outputs at the beginning of each 5-s interval. Table 1-1 is a table describing

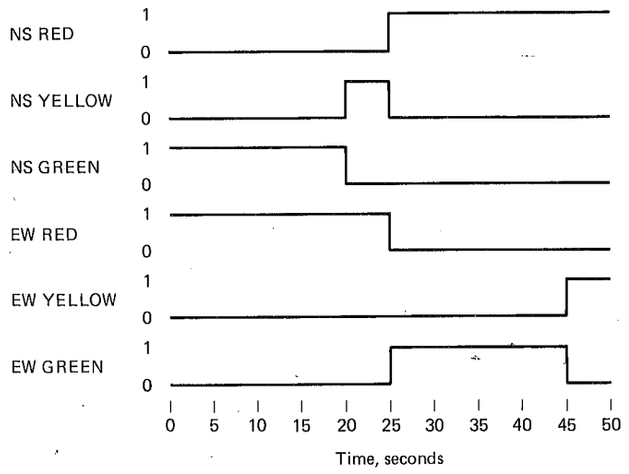
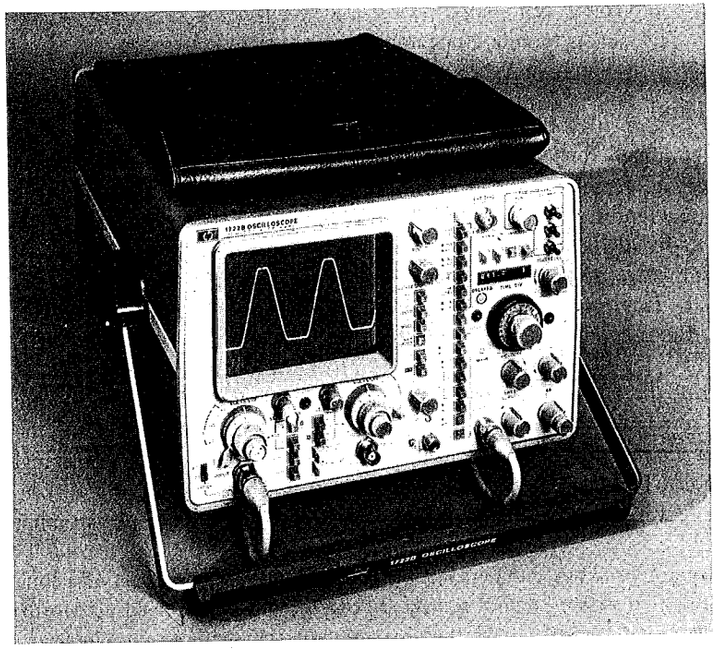


Figure 1-8 Timing diagram for traffic-light output.

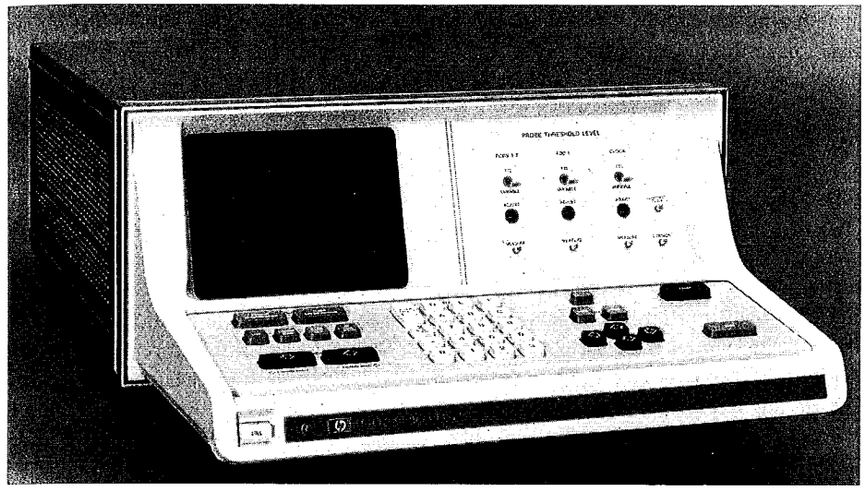


An oscilloscope (*Hewlett-Packard Co.*).

the traffic controller of Fig. 1-8. Since additional signals are merely added columns and additional time intervals are just added rows, very large and complex systems are easily represented in tabular form. However, the tabular description is usually restricted to circuits in which signals change only at regular clock intervals. The tabular representation of a real operating circuit may be viewed by connecting an instrument called a *logic-state analyzer* to the circuit.

Table 1-1 Tabular representation of logic signals

	NS RED	NS YELLOW	NS GREEN	EW RED	EW YELLOW	EW GREEN
0	0	0	1	1	0	0
5	0	0	1	1	0	0
10	0	0	1	1	0	0
15	0	0	1	1	0	0
20	0	1	0	1	0	0
25	1	0	0	0	0	1
30	1	0	0	0	0	1
35	1	0	0	0	0	1
40	1	0	0	0	0	1
45	1	0	0	0	1	0



A logic analyzer (*Hewlett-Packard Co.*).

Logic Circuits

Logic circuit elements are usually fabricated from small pieces of silicon as *monolithic integrated circuits.* These integrated circuits (or ICs) are packaged in plastic, ceramic, or metal containers, with leads that allow the various circuits to be interconnected.

Digital ICs are divided into families and subfamilies, according to the electronic circuitry used to implement the device. Compatibility among families and subfamilies may or may not require special conversion circuits. In any case, sets of interconnection rules are developed to assist the designer in interconnecting circuits within and among families. These interconnection rules are available from the manufacturers of the ICs. Standardization within an IC family allows us to concentrate on the function of the circuit without regard to electronic details.

Although thousands of types of digital integrated circuits are manufactured, they may be classified into much fewer general types. In fact, all the algorithms discussed so far may be implemented using only two types of circuits: read-only memories (ROMs) and D flip-flops. Theoretically, all algorithms could be implemented with these two types of digital circuits. However, the many additional types of circuits available offer additional economies in implementations of more complex algorithms.

Circuit Diagrams

Interconnections among logic circuits are specified by a diagram in which they are represented by lines. These lines "connect" symbols representing each logic circuit. Although two conductors are required to complete an electric circuit, only one line is drawn. The other conductor of every electric circuit is assumed to be a *ground* connection, common to all logic circuits. Each logic circuit also requires a *power* connec-

tion to a power supply. Neither the power nor ground connections will be shown on diagrams in this book.

D Flip-Flops

A type D flip-flop is a memory circuit. It remembers a single digital signal or bit. In its simplest form, a D flip-flop has two inputs and one output, as shown in Fig. 1-9. The rectangular block represents the type D flip-flop with its inputs labeled *D* and *C* and its output labeled *Q*. The straight lines leading to the block represent wires connecting to other parts of the circuit. These wires carry digital signals to and from the flip-flop. The flip-flop remembers the logic state at its *D* or data input at the time that its *C* or clock input makes a transition from a 0 to a 1. The flip-flop remembers this *D* input state on its *Q* output and thus makes it available for other circuits to use. The timing diagram of Fig. 1-9 illustrates the operation of this circuit. The clock input is normally a regular periodic signal, like the one shown. It is generated by another electronic circuit, not important to the discussion. The flip-flop *does* something only when a transition from 0 to 1 occurs on its clock input, at times *W*, *X*, *Y*, and *Z* on the timing diagram. At time *W*, the state of the *D* input, a zero, is remembered on the *Q* output. Even though the *D* input changes to a 1 later, the *Q* output remembers the zero and will not change until time *X*, when a 0-to-1 transition of the clock input occurs. Since the *D* input is a 1 at time *X*, a 1 appears on the *Q* output. At time *Y*, the *Q* output remains a 1 because the *D* input is also a 1. Note that even though the *D* input changes to a zero, the *Q* output remains a 1 until the next clock transition at time *Z*, when *Q* becomes a zero. Again, to summarize, the *Q* output remembers the state of the *D* input that existed when the last 0-to-1 transition occurred on the clock (*C*) input.

The clock input of the type D flip-flop we are using is *edge-sensitive*. That is, it responds only to the 0-to-1 transition or edge of the clock signal. To distinguish edge-sensitive inputs from other inputs, we will use a small triangle placed inside the circuit symbol next to the edge-sensitive input, as shown in Fig. 1-9.

Actually, the *Q* output doesn't change until a small time after the clock transition. This time, called *delay time*, is due to the electronic circuitry in the flip-flop. Typical delay times are several nanoseconds (10^{-9} s).

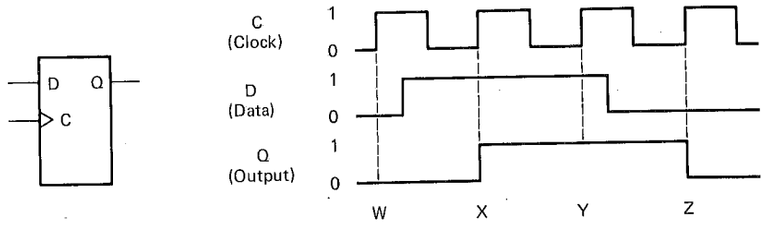
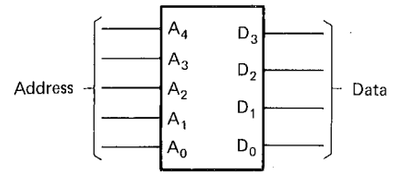


Figure 1-9 Type D flip-flop.

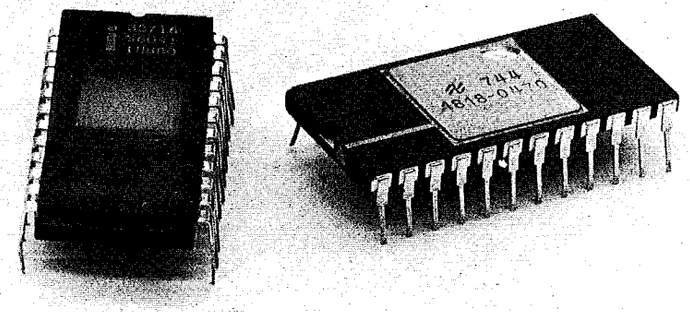


A_4	A_3	A_2	A_1	A_0	D_3	D_2	D_1	D_0
0	0	0	0	0	0	1	1	0
0	0	0	0	1	0	1	0	1
0	0	0	1	0	0	1	1	1
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.

Figure 1-10 Read-only memory.

Read-Only Memories

While type D flip-flops can remember new data at each clock transition, read-only memories (ROMs) remember a preset pattern of data. This pattern is specified when the ROM circuit is manufactured and is unchangeable thereafter. Figure 1-10 shows a ROM represented as a block with connecting wires. The ROM illustrated has five inputs, called address lines, and four outputs, called data lines. There are 32 places in the ROM where 4 bits of data are stored. Each of these 32 places is called a *memory location* or *memory word*. Each memory location is identified by a *memory address*. A memory address is a particular combination of 1s and 0s placed on the address lines of the ROM. Since this ROM has five address lines, 2^5 or 32 combinations of 1s and 0s may be specified as addresses. Placing one such combination of address bits on the address inputs causes data that were stored in that location when the circuit was manufactured to appear on the data lines. For example, in the ROM illustrated, placing 00000 on the address lines causes 0110 to appear on the data lines. You specify the contents of the data word that you want put in each of the memory locations of the ROM. The manufacturer builds those data into the circuit permanently. Notice that



An EPROM is on the left with transparent window and a ROM is on the right (*Hewlett-Packard Co.*).

ROMs have no clock input. The output data simply change as soon after you change the address as the internal electronic circuitry will allow. The time it takes for correct data to appear on the outputs after an address change is called the *access time.* Access time can vary from several nanoseconds (10^{-9} s) to several microseconds (10^{-6} s), depending on the nature of the electronic circuits in the ROM.

A ROM having five address and four data lines is called a 32 X 4 ROM. It has 32 memory locations, each having 4 bits of data. ROMs come in a wide variety of sizes. A ROM having 10 address inputs and 8 data outputs would have 1024 locations (2^{10} memory locations) of 8 bits each. Two common abbreviations are used to specify size. Since 1024 is so close to 1000, the abbreviation K always stands for 1024 when referring to memory circuits. Thus, a 1024 X 8 memory is called a 1K X 8 memory. Since 8 bits is a common unit used by computers, it is given the name *byte.* Thus, a 1024 X 8-bit memory is called a 1K-byte for 1KB memory.

Finally, it should have occurred to you that having a new ROM fabricated with a special pattern is an expensive and time-consuming undertaking. Two kinds of ROMs have been developed to minimize these problems. The first is called a programmable ROM, often called a *PROM.* This kind of ROM is manufactured with either all 1s or all 0s in every bit of every location. A special machine called a PROM programmer allows individual bits to be changed to the opposite state once and only once after the PROM is manufactured. Because all PROMs are identical when they leave the manufacturer, they take advantage of manufacturing economies of scale and are quite inexpensive, even when purchased in small quantities. PROMs allow read-only memories

A PROM Programmer. An ultraviolet light for erasing EPROMs is on the left. Sockets for PROMs are in the center. The keyboard allows manual entry of data. PROMs may also be copied from a master PROM (*Pro-Log Corporation*).

to be incorporated in many circuits when small quantities would not justify the cost of manufacturing a specially patterned ROM.

PROMs that can only be programmed once are inconvenient when new digital circuits are being developed in the laboratory. Mistakes are made, specifications change, and new ideas must be continually tried. For this reason, a part called an *erasable programmable read-only memory* or EPROM has been developed. An EPROM is programmed like a PROM. However, it can be erased and returned to its original manufactured state for reprogramming with a new pattern. The EPROM is erased by shining an ultraviolet light onto the circuit through a transparent window in its package.

Hex Notation

Specifying a large ROM with 1s and 0s is very inconvenient. A 1KB ROM would require a 1024-line table, with each line containing 10 address and 8 data bits. One common method of simplifying the specification of large numbers of bits is to use hex notation. Hex notation uses a single character (0-9 and A-F) to specify 1 of 16 possible combinations of 4 bits:

4-bit group				Hex equivalent
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	A
1	0	1	1	B
1	1	0	0	C
1	1	0	1	D
1	1	1	0	E
1	1	1	1	F

Note that since the *order* of the bits is important, the rightmost bit is always called the least significant, and the leftmost bit the most significant. If the digital signals are given symbolic names (e.g., $A_5, A_4, A_3, A_2, A_1, A_0$), then the least significant bit is almost always the one whose name has the smallest subscript. Groups of more than 4 bits are encoded by dividing the bits into groups of 4, starting from the rightmost or least significant bit, and encoding each group separately. Note that, if the number of bits is not a multiple of 4, zeros are assumed for nonexistent bits. Also, hex numbers are often followed by the letter H to differentiate them from decimal numbers.

0000 : 1111 0FH

11 : 0000 : 1110 30EH

Circuit Parameters

All logic circuits have many electric parameters associated with them. Two important parameters are *fan-out* and *propagation delay*. Fan-out is the specification of the number of inputs to which a given output may be connected. Exceeding the fan-out specification will cause the circuit to malfunction. All logic circuit outputs have a fan-out specification. Propagation delay is the amount of time it takes for a change in an input logic signal to change an output logic signal. Thus, there is a propagation delay between the address inputs and data outputs of a ROM or between the clock input and Q output of a flip-flop. Logic circuits have many other parameters, but these two are the most important. We'll consider other parameters in a later chapter.

IMPLEMENTING A TRAFFIC LIGHT CONTROLLER

We're now ready to design a controller to implement the flowchart of Fig. 1-2. This flowchart is redrawn in Fig. 1-11 with additional blocks added so that all blocks are 5 s long. How should we proceed to use D flip-flops and ROMs to implement an electronic circuit to perform this function? Notice that there are 10 separate action blocks in Fig. 1-11. The controller will sequence through these actions one at a time at each 5-s interval. The controller must obviously remember which of these 10 actions it is currently performing. Four type D flip-flops could remember 1 of 16 different things, by using all possible combinations of 1s and 0s. We could use 10 of those combinations to specify which action block we're currently performing. The flowchart of Fig. 1-11 has a combination of 1s and 0s next to each action block. These 1s and 0s are called *state variables.* These combinations of 1s and 0s are the *states* of the controller and are also listed in the flowchart as hex numbers. Figure 1-12 shows the four type D flip-flops which will remember the current state of the controller. The flip-flop Q outputs are subscripted to differentiate among them. Moreover, the flip-flop labeled Q_0 is the rightmost or least significant state bit in the flowchart of Fig. 1-11. We've connected all the clock inputs together and to a clock generator circuit since we know we will want to change the current state of the machine every 5 s. Let's not worry about from where the next state will come. First, let's determine how to generate the controller's outputs.

Outputs

We must generate six outputs to control the actual light bulbs in the traffic signals. These six outputs are shown in Table 1-2. Each signal is given a name that makes it easy to remember the function of that output. The logic signals to appear on each output are specified by each action block or state of Fig. 1-11. Use the four current state outputs of the flip-flops as address inputs to a ROM, as shown in Fig. 1-13. Store the six outputs corresponding to each of the 10 states in 10 locations of that read-only memory. Table 1-3 specifies the contents of each ROM location and was obtained directly from the flowchart of Fig. 1-11. For example, the first action block in the

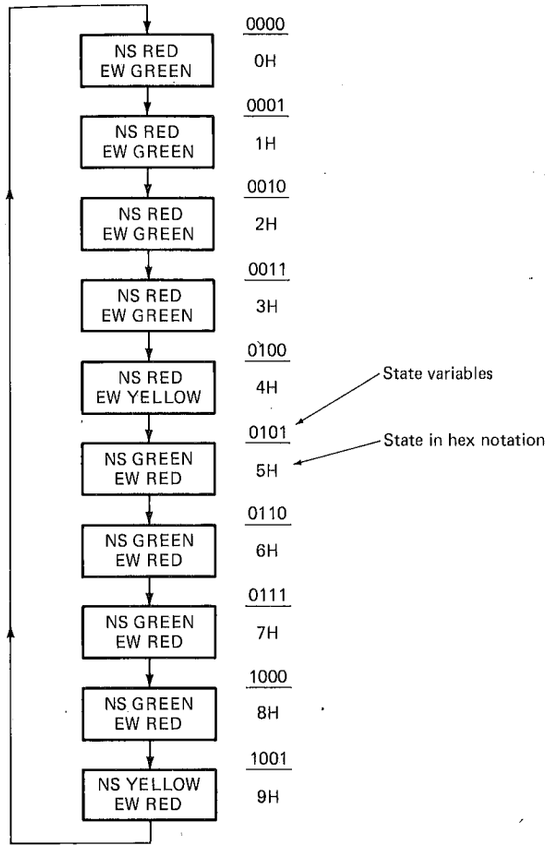


Figure 1-11 Flowchart for traffic-light controller.

flowchart corresponds to state 0H. Since the ROM address inputs are connected to the current state outputs in identical order, state 0H also is address 0H in the ROM. We see from the flowchart that we want the north and south red lights on, the east and west green lights on, and all other lights off. We specify the contents of memory location 0H as:

$$\begin{aligned} D_0 &= 0 \quad \text{or} \quad NSG = 0 \\ D_1 &= 0 \quad \text{or} \quad NSY = 0 \\ D_2 &= 1 \quad \text{or} \quad NSR = 1 \\ D_3 &= 1 \quad \text{or} \quad EWG = 1 \\ D_4 &= 0 \quad \text{or} \quad EWY = 0 \\ D_5 &= 0 \quad \text{or} \quad EWR = 0 \end{aligned}$$

Whenever the current state of 0H appears on the four flip-flop outputs, the ROM outputs corresponding to that address will cause the traffic lights to be set to the proper indication. Continuing through all blocks of the flowchart, we generate 10 lines in

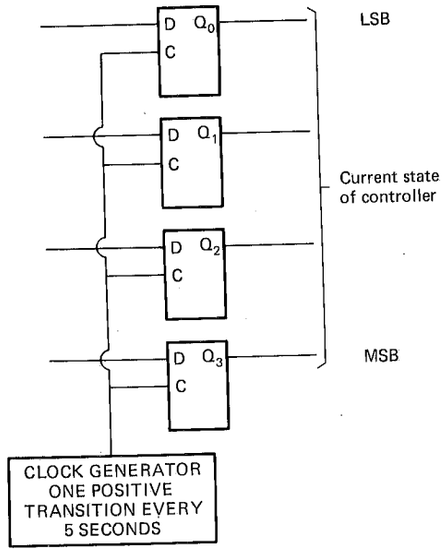


Figure 1-12 Current state register using D flip-flops.

Table 1-3, coresponding to the proper outputs for each of the 10 states. Since four address lines can specify 16 memory locations, six locations of the ROM are not needed for the controller. In Table 1-3, these six unused addresses are specified as all zeros. These locations could be specified as anything because they do not affect the operation of the controller.

Next State

Notice that the *D* inputs in Fig. 1-13 are not connected to anything. Recall that a type D flip-flop remembers the condition of its *D* input at the 0-to-1 transition of its clock input. Thus, whatever signals we apply to the *D* inputs of Fig. 1-13 will become the new state of the controller after the clock transition. We know what this state should be from the flowchart of Fig. 1-11. The transitions from state to state are

Table 1-2 Output signal definitions

Signal abbreviation	If signal is 0	If signal is 1
NSG	NS green is off	NS green is on
NSY	NS yellow is off	NS yellow is on
NSR	NS red is off	NS red is on
EWG	EW green is off	EW green is on
EWY	EW yellow is off	EW yellow is on
EWR	EW red is off	EW red is on

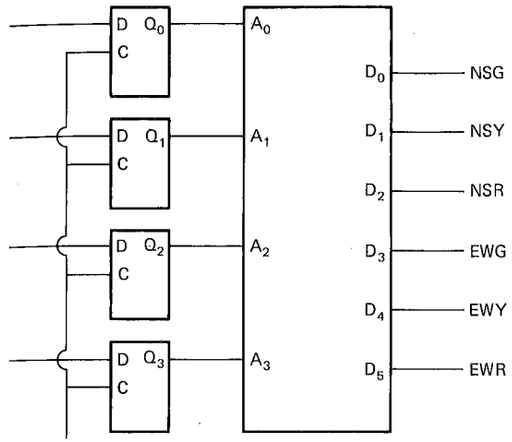


Figure 1-13 Traffic controller output ROM.

indicated by the lines and arrows connecting the states. If we are in state 3H, the line segment directed to state 4H shows us that the next state after 3H should be 4H. Whenever we have 3H as the current state on the *Q* outputs of the flip-flops, we should have 4H on the *D* inputs. The next clock transition will cause the next state to be state 4H. We can generate the next state to be applied to the *D* inputs by storing the next state corresponding to each current state in the ROM, as shown in Fig. 1-14. The four new ROM outputs are the next state to be remembered in the flip-flops after the clock pulse. The 4 additional bits in each ROM location are determined from the

Table 1-3 Traffic controller outputs

State				ROM	Outputs						ROM
Q_3	Q_2	Q_1	Q_0	address	D_5	D_4	D_3	D_2	D_1	D_0	contents
0	0	0	0	0H	0	0	1	1	0	0	0CH
0	0	0	1	1H	0	0	1	1	0	0	0CH
0	0	1	0	2H	0	0	1	1	0	0	0CH
0	0	1	1	3H	0	0	1	1	0	0	0CH
0	1	0	0	4H	0	1	0	1	0	0	14H
0	1	0	1	5H	1	0	0	0	0	1	21H
0	1	1	0	6H	1	0	0	0	0	1	21H
0	1	1	1	7H	1	0	0	0	0	1	21H
1	0	0	0	8H	1	0	0	0	0	1	21H
1	0	0	1	9H	1	0	0	0	1	0	22H
				AH	0	0	0	0	0	0	00H
				BH	0	0	0	0	0	0	00H
				CH	0	0	0	0	0	0	00H
				DH	0	0	0	0	0	0	00H
				EH	0	0	0	0	0	0	00H
				FH	0	0	0	0	0	0	00H

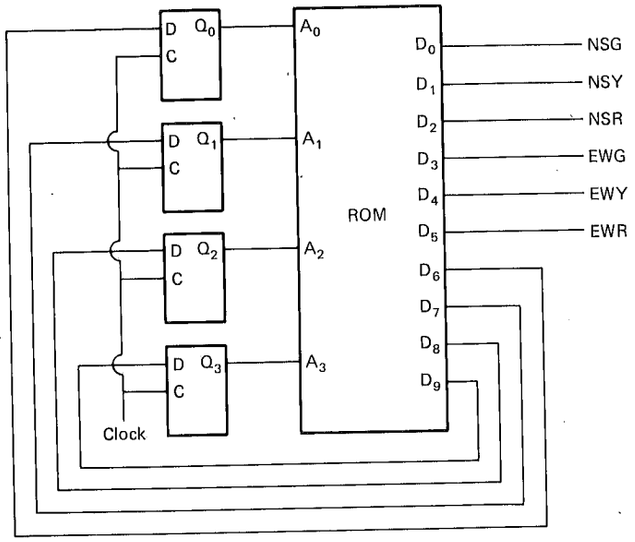


Figure 1-14 Traffic controller.

arrows in the flowchart of Fig. 1-11. If the current state is 6H, then ROM location 6H will contain the next state 0111 in bits D_9, D_8, D_7, and D_6. The next state for each current state is determined to complete the ROM contents of Table 1-4.

Table 1-4 and Fig. 1-14 represent the complete design of a digital controller to control traffic lights at a simple intersection. The controller is a sequential digital

Table 1-4 Traffic controller ROM contents

Current state ROM address	Next state				Outputs						ROM contents
	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	
0H	0	0	0	1	0	0	1	1	0	0	04CH
1H	0	0	1	0	0	0	1	1	0	0	08CH
2H	0	0	1	1	0	0	1	1	0	0	0CCH
3H	0	1	0	0	0	0	1	1	0	0	10CH
4H	0	1	0	1	0	1	0	1	0	0	154H
5H	0	1	1	0	1	0	0	0	0	1	1A1H
6H	0	1	1	1	1	0	0	0	0	1	1E1H
7H	1	0	0	0	1	0	0	0	0	1	221H
8H	1	0	0	1	1	0	0	0	0	1	261H
9H	0	0	0	0	1	0	0	0	1	0	022H
AH	0	0	0	0	0	0	0	0	0	0	000H
BH	0	0	0	0	0	0	0	0	0	0	000H
CH	0	0	0	0	0	0	0	0	0	0	000H
DH	0	0	0	0	0	0	0	0	0	0	000H
EH	0	0	0	0	0	0	0	0	0	0	000H
FH	0	0	0	0	0	0	0	0	0	0	000H

machine and could actually be constructed and used to control traffic. It consists of a few simple parts. A group of flip-flops storing the current state (also called the *current state register*) is sequenced to a new state every 5 s by a *clock generator* circuit. A *read-only memory* generates the appropriate *outputs* for each state as well as the *next state* that is to occur after each clock transition.

IMPLEMENTING A TRAFFIC CONTROLLER WITH INPUTS

If we attempt to implement a controller for the flowchart of Fig. 1-5, we'll first draw a flowchart with action blocks of equal time duration, as shown in Fig. 1-15. We can assign a state to each action block and store the appropriate outputs in a ROM. How-

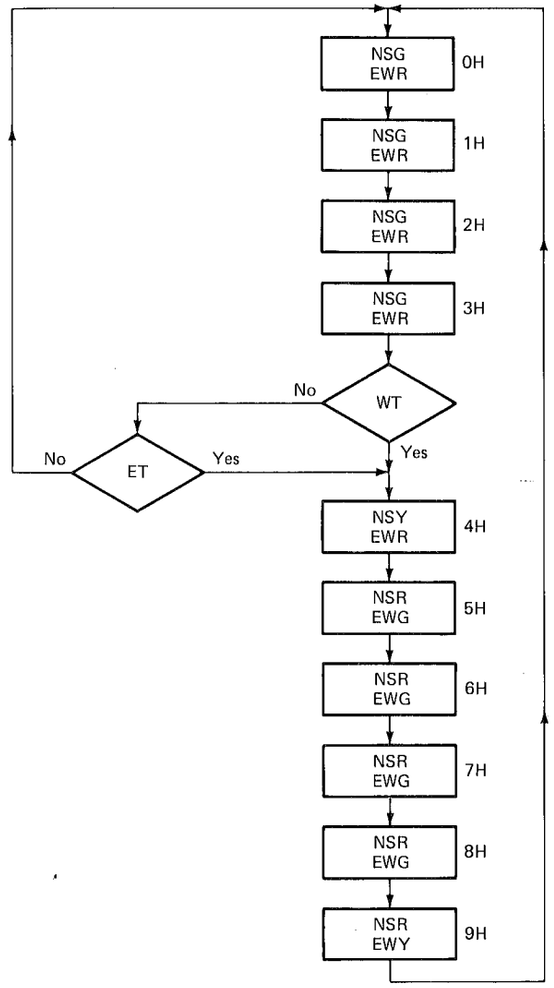


Figure 1-15 Flowchart of traffic-light controller with traffic sensor inputs.

ever, when we come to storing the next state in the ROM, a serious problem arises. The next state after state 3 is *either* state 4 or 0, depending on inputs *ET* and *WT*. The simple traffic controller previously implemented had no inputs. We didn't consider how we'd route the sequence to alternate states, depending on the condition of an input.

The next state now depends on the two inputs *ET* and *WT* as well as the current state Q_3, Q_2, Q_1, and Q_0. The next state will be easily specified for each combination of these six variables, two *inputs* and four *state variables.* By using a ROM with six address inputs, as shown in Fig. 1-16, we can look up the next state for any of the 40 possible combinations of these six digital signals (24 combinations won't occur because the state variables assume only 10 combinations rather than 16). The ROM contents of Table 1-5 are obtained directly from the flowchart of Fig. 1-15. There are four rows in the table for each current state. These four rows correspond to the four possible combinations of *WT* (A_1) and *ET* (A_0). In most cases all four rows are identical, indicating that neither the next states nor the outputs are dependent on the inputs *ET* and *WT*. Only in state 3H are any differences observed because the next state *is* a function of logic signals *ET* and *WT* whenever the controller is in state 3H.

Table 1-5 is long and cumbersome. Since many lines are identical, it seems we should be able to abbreviate this table. Table 1-6 also specifies the ROM's contents. Whenever the condition of an address or data bit doesn't matter, that bit is shown as an X and called a *don't care* bit. Most of the groups of four lines, corresponding to a

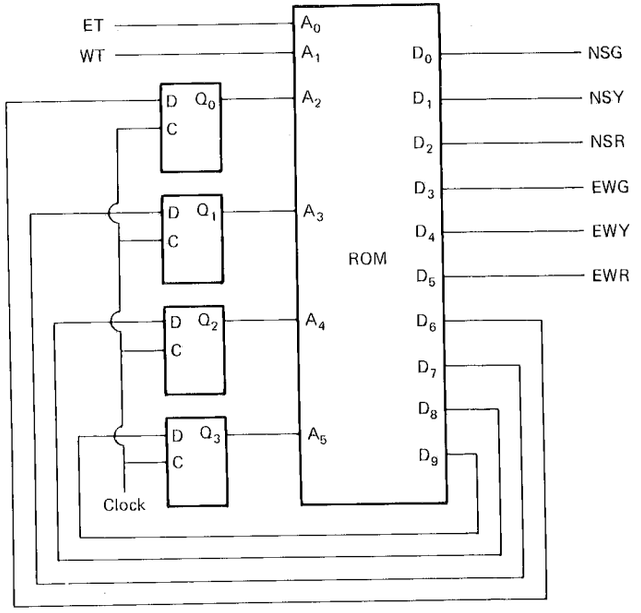


Figure 1-16 Traffic-light controller with traffic sensor inputs.

Table 1-5 ROM contents for traffic controller with traffic sensors

State				WT	ET	Next state				Outputs					
A_5	A_4	A_3	A_2	A_1	A_0	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1
0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	1
0	0	0	0	1	0	0	0	0	1	1	0	0	0	0	1
0	0	0	0	1	1	0	0	0	1	1	0	0	0	0	1
0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	1
0	0	0	1	0	1	0	0	1	0	1	0	0	0	0	1
0	0	0	1	1	0	0	0	1	0	1	0	0	0	0	1
0	0	0	1	1	1	0	0	1	0	1	0	0	0	0	1
0	0	1	0	0	0	0	0	1	1	1	0	0	0	0	1
0	0	1	0	0	1	0	0	1	1	1	0	0	0	0	1
0	0	1	0	1	0	0	0	1	1	1	0	0	0	0	1
0	0	1	0	1	1	0	0	1	1	1	0	0	0	0	1
0	0	1	1	0	0	0	0	0	0	1	0	0	0	0	1
0	0	1	1	0	1	0	1	0	0	1	0	0	0	0	1
0	0	1	1	1	0	0	1	0	0	1	0	0	0	0	1
0	0	1	1	1	1	0	1	0	0	1	0	0	0	0	1
0	1	0	0	0	0	0	1	0	1	1	0	0	0	1	0
0	1	0	0	0	1	0	1	0	1	1	0	0	0	1	0
0	1	0	0	1	0	0	1	0	1	1	0	0	0	1	0
0	1	0	0	1	1	0	1	0	1	1	0	0	0	1	0
0	1	0	1	0	0	0	1	1	0	0	0	1	1	0	0
0	1	0	1	0	1	0	1	1	0	0	0	1	1	0	0
0	1	0	1	1	0	0	1	1	0	0	0	1	1	0	0
0	1	0	1	1	1	0	1	1	0	0	0	1	1	0	0
0	1	1	0	0	0	0	1	1	1	0	0	1	1	0	0
0	1	1	0	0	1	0	1	1	1	0	0	1	1	0	0
0	1	1	0	1	0	0	1	1	1	0	0	1	1	0	0
0	1	1	0	1	1	0	1	1	1	0	0	1	1	0	0
0	1	1	1	0	0	1	0	0	0	0	0	1	1	0	0
0	1	1	1	0	1	1	0	0	0	0	0	1	1	0	0
0	1	1	1	1	0	1	0	0	0	0	0	1	1	0	0
0	1	1	1	1	1	1	0	0	0	0	0	1	1	0	0
1	0	0	0	0	0	1	0	0	1	0	0	1	1	0	0
1	0	0	0	0	1	1	0	0	1	0	0	1	1	0	0
1	0	0	0	1	0	1	0	0	1	0	0	1	1	0	0
1	0	0	0	1	1	1	0	0	1	0	0	1	1	0	0
1	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0
1	0	0	1	0	1	0	0	0	0	0	1	0	1	0	0
1	0	0	1	1	0	0	0	0	0	0	1	0	1	0	0
1	0	0	1	1	1	0	0	0	0	0	1	0	1	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0

Table 1-5 ROM contents for traffic controller with traffic sensors (*Continued*)

State				WT	ET	Next state				Outputs					
A_5	A_4	A_3	A_2	A_1	A_0	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0
1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0

Table 1-6 Abbreviated ROM contents

State				WT	ET	Next state				Outputs					
A_5	A_4	A_3	A_2	A_1	A_0	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	X	X	0	0	0	1	1	0	0	0	0	1
0	0	0	1	X	X	0	0	1	0	1	0	0	0	0	1
0	0	1	0	X	X	0	0	1	1	1	0	0	0	0	1
0	0	1	1	0	0	0	0	0	0	1	0	0	0	0	1
0	0	1	1	0	1	0	1	0	0	1	0	0	0	0	1
0	0	1	1	1	0	0	1	0	0	1	0	0	0	0	1
0	0	1	1	1	1	0	1	0	0	1	0	0	0	0	1
0	1	0	0	X	X	0	1	0	1	1	0	0	0	1	0
0	1	0	1	X	X	0	1	1	0	0	0	1	1	0	0
0	1	1	0	X	X	0	1	1	1	0	0	1	1	0	0
0	1	1	1	X	X	1	0	0	0	0	0	1	1	0	0
1	0	0	0	X	X	1	0	0	1	0	0	1	1	0	0
1	0	0	1	X	X	0	0	0	0	0	1	0	1	0	0
1	0	1	0	X	X	0	0	0	0	0	0	0	0	0	0
1	0	1	1	X	X	0	0	0	0	0	0	0	0	0	0
1	1	0	0	X	X	0	0	0	0	0	0	0	0	0	0
1	1	0	1	X	X	0	0	0	0	0	0	0	0	0	0
1	1	1	0	X	X	0	0	0	0	0	0	0	0	0	0
1	1	1	1	X	X	0	0	0	0	0	0	0	0	0	0

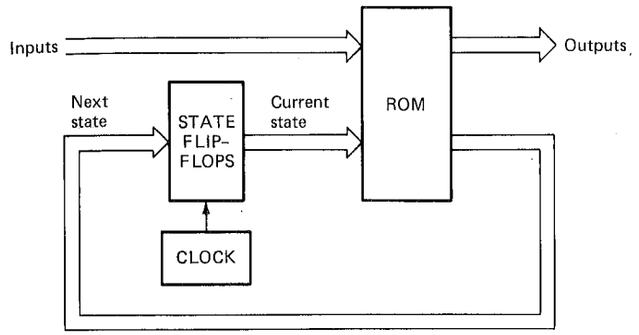


Figure 1-17 Sequential digital machine.

single state in Table 1-5, may be abbreviated as a single line in Table 1-6. The abbreviated table is useful while the design is in progress. A complete table is usually generated to program a PROM after the design has been completed.

Figure 1-16 incorporates all the features required of any digital logic circuit. Although we'll study many different approaches to logic circuits other than Fig. 1-16, they all accomplish the same function as this simple state flip-flop and ROM controller. Figure 1-17 shows a generalized diagram of a digital logic circuit. Many controller circuits are built exactly in this fashion. It is very important that you really understand the operation of this circuit before proceeding.

BLOCK DIAGRAMS

Most real digital circuits are so complex that they must be divided into simpler subcircuits. Numbers of small circuits are easier to design than one large circuit. A *block*

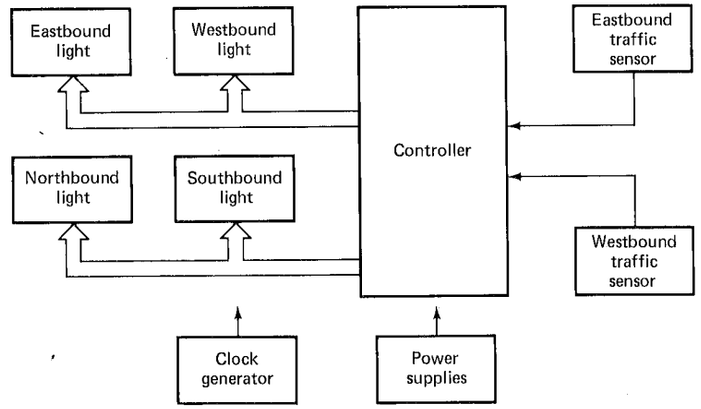


Figure 1-18 Traffic-light block diagram.

diagram shows these subcircuits (blocks) and their interconnections. Block diagrams are an aid to planning and understanding the operation of large digital systems. Even a simple system like the traffic lights has a block diagram shown in Fig. 1-18. Note that multiple signals are represented by broad lines and the number and nature of these signals have not yet been specified. The next step in the design process would be to specify these signals connecting the blocks.

The arrangement of the blocks and their interconnection is a very important part of the design process. Especially in large digital systems, *the arrangement of the block diagram will have more effect on cost and performance than the design of the circuitry in each block.* The block diagram and the algorithm design are the most creative part of designing a digital system.

THE FORMAL ALGORITHMIC STATE MACHINE

Returning to the individual circuit block, we see the digital control circuit we will use is represented by the generalized controller of Fig. 1-17 and the definitions of all its inputs and outputs. Once you decide to use the general model of the digital sequential machine for your solution and define all the inputs and outputs of that machine, you are in a position to describe the algorithm to be used as the solution to the problem. The description of the algorithm may be plain text, a flowchart, or any other convenient technique. After describing the algorithm, the actual circuit may be implemented. The implementation amounts to connecting the ROMs, flip-flops, inputs, and outputs in an appropriate manner and specifying the contents of the ROM to implement the algorithm desired.

In order to minimize design errors, we will formalize some of the techniques we've been using. First, we will formalize the notation for input and output signals. Then the flowchart notation will be more rigorously specified and the flowchart renamed the ASM chart.

Input and Output Signals

Previously, we gave input and output signals names to distinguish them from each other and to make it easy to remember what each signal did. These names are called *mnemonics.* The meaning of each output signal was defined in a table, like Table 1-2. Since this table requires additional effort to generate and must be referenced often while reading circuit drawings, we'd like to find a way to include this information in the mnemonic itself. The signal naming system we will use is the same as that used by Clare.[1] We shall put one additional letter ahead of the mnemonic. If this first letter is an *H*, the action specified by that mnemonic is performed when the associated digital signal is a 1 (high). If the first letter is an *L*, the action specified by that mnemonic occurs when the signal is a 0 (low). For example, the mnemonic *HNSG*

[1]Christopher R. Clare, *Designing Logic Systems Using State Machines*, McGraw-Hill, New York, 1973, pp. 8–9.

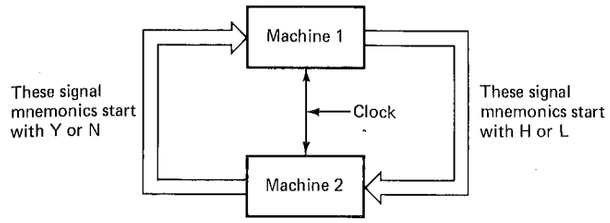


Figure 1-19 Interconnected sequential machines.

means that the north-south green light is on whenever this signal is a 1. The mnemonic *LEWG* means that the east-west green light is on whenever this signal is a 0. By adding this single letter to each output mnemonic, we are able to tell if the action occurs when the signal is a 1 or if the action occurs when the signal is a 0.

To distinguish inputs from outputs, we'll use two different letters for input signals. If an input signal mnemonic is preceded by a *Y*, the condition specified by the mnemonic is true when the signal is a 1. If the mnemonic is preceded by an *N*, the condition specified is true when the signal is a 0. For example, the mnemonic *YWT* means that there is westbound traffic present if this signal is a 1. The mnemonic *NEMER* means that the emergency button is pushed if this signal is a 0.

Now we are able to distinguish between input and output signals and, at the same time, precisely define their operation. Two more complications arise when two digital controllers are connected to each other, as shown in Fig. 1-19. Since the output of one machine is the input of the other, we obviously can't keep our input and output naming convention if each signal is to have only one name. The signals starting with *Y* or *N* will be inputs of machine 1 but outputs of machine 2. Likewise, the signals starting with *H* or *L* will be outputs of machine 1 but inputs of machine 2. This naming difficulty causes no problems because the action of the signals with respect to the two machines is well-defined.

A more important characteristic of two interconnected digital controllers is shown in Fig. 1-20. The input of machine 2, called *HRST*, is also an output of machine 1. Machine 1 makes this signal a 1 (*asserts* it) during machine 1's state 4. Note that *HRST*

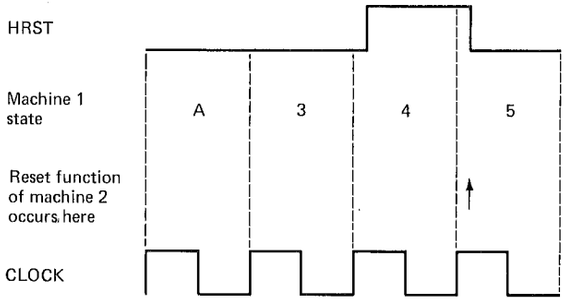


Figure 1-20 Timing of a delayed-action output.

changes a small time after the clock transition, as we've discussed previously. Therefore, *machine 2 cannot respond to this input until the next clock transition.* Thus, the function specified by the signal *HRST* will not be performed by machine 2 until machine 1 is in state 5. Although the signal *HRST* was asserted by machine 1 in state 4, the function associated with *HRST* was not performed until machine 1 reached state 5. This delayed action will occur any time the output of one sequential machine is the input to another and both machines use the same clock signal. This was not the case for the outputs of the traffic light controller. These outputs caused the traffic lights to turn on immediately. The output and the action caused by it were simultaneous. To distinguish immediate from delayed-action outputs, we will prefix immediate outputs with the letter *I*. Thus, a mnemonic for the traffic light controller output would be *IHEWG*. No prefix is added for delayed-action outputs because they are usually more numerous than immediate-action outputs. The prefix *I* only makes sense on the mnemonic of an output signal.

Synchronous and Asynchronous Inputs

Since an input to an ASM (algorithmic state machine) may come from any number of sources, it may change state (0 to 1 or 1 to 0) at any time. These inputs are called asynchronous because they are not synchronized with the clock transitions of the ASM. Asynchronous inputs may cause many problems in an ASM, as will be discussed in a later chapter. To simplify the current discussion, we will assume that all inputs to our ASMs will be synchronous. That is, all inputs can only change state just after the clock transition. Any input, such as a traffic sensor, that is not inherently synchronized with a clock, will be assumed to have additional internal circuitry to synchronize it to the clock transitions.

ASM Charts

An ASM chart is simply a flowchart describing a sequential digital machine. ASM charts were first described by Clare.[1] An ASM chart is drawn following a set of simple but precise rules. A *state box* is drawn for each machine state, as shown in Fig. 1-21. The outputs for that state are listed inside the box. The name of the state is encircled and placed to either side of the box. This allows the state to be referenced by name even before the particular combination of state variables representing that state is

[1] Clare, *Designing Logic Systems Using State Machines*, pp. 16–19.

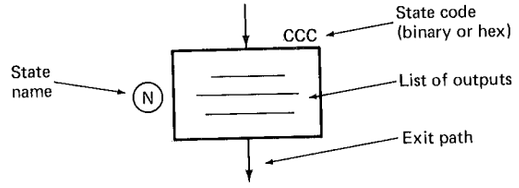


Figure 1-21 ASM state box.

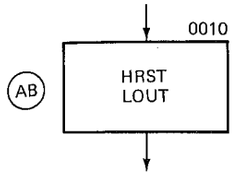


Figure 1-22 Example of state box.

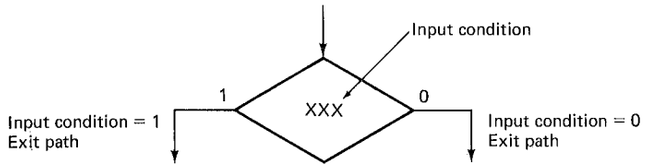


Figure 1-23 Decision diamond.

assigned. This combination of state variables is called the *state code* and is written in hex or binary on top of the state box. Finally, the line drawn from the state box, indicating the path to the next state, is called the *exit path.* Figure 1-22 is an example of a state box. It shows a state named *AB*, which has outputs *HRST* and *LOUT*. The state code for this state is 0010.

ASM charts also have decision diamonds, just like flowcharts. Figure 1-23 shows a decision diamond. The input to be tested is drawn inside the diamond. The diamond exits are always marked 0 and 1. Since 0 and 1 refer to actual digital signal values, there can be no confusion arising from words such as TRUE, FALSE, YES, or NO. Note that a 1 will correspond to the true condition only if the input mnemonic begins with a *Y*. If the input mnemonic begins with an *N*, the true condition will be represented by the 0 branch.

An *ASM block* consists of a single state box and all the decision diamonds connected to its exit path. Figure 1-24 is an example of such an ASM block. Each exit path

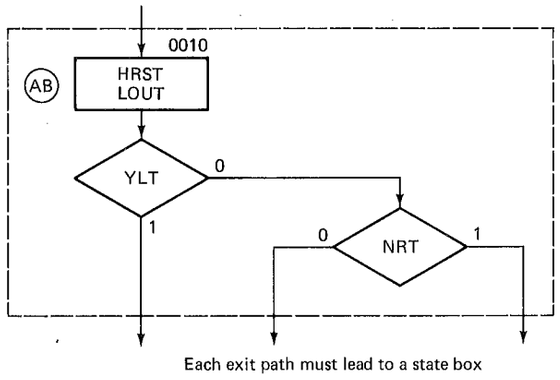


Figure 1-24 ASM block.

must lead to a state box. Since only state boxes have a time period associated with them, *each ASM block represents a single time period or clock interval.* An ASM block shows the current state, its outputs, and the conditions for each next-state path. Thus, each ASM block completely defines all the rows of a ROM table corresponding to a single current state.

Conditional Outputs

In our previous examples, input decisions were made only to determine which of several next-state paths would be followed. In other words, only the next-state entries were different for each input combination in the ROM table. The outputs for a given current state were always the same, regardless of the inputs. This is an unnecessary restriction. The outputs of a single state could easily be made different for each input combination simply by specifying them differently for each row in the ROM table. Outputs that depend on inputs are shown on an ASM chart by a *conditional output box.* Figure 1-25 shows the symbol for a conditional output box. The line entering the box *always* comes from a decision diamond specifying the condition required for these outputs to occur. The conditional outputs are listed in the box and can cause immediate or delayed actions, just like the outputs listed in a state box.

Figure 1-26 shows an example of an ASM block with a conditional output. The output *HRST* always occurs in state *AC*; so *HRST* is listed in the state box. The output *LOUT* only occurs in state *AC if NRT* is 0; so *LOUT* is listed in a conditional output box in the exit path corresponding to *NRT* = 0. *Conditional output boxes have no additional time associated with them. Only state boxes have a time interval associated with them.* In the example of Fig. 1-26, the conditional output *LOUT* occurs in the same time period in which *HRST* occurs. All conditional outputs in an ASM block occur in the state time of the single state box in that block.

Let's look at three possible timing diagrams for the ASM chart of Fig. 1-26. These timing diagrams are shown in Fig. 1-27. In Fig. 1-27*a*, the input *NRT* is 1 during state *AC*. The conditional output *LOUT* is not asserted, and the next state is *AE*. *HRST* is always asserted in state *AC*. In Fig. 1-27*b*, input signal *NRT is* asserted during state *AC*. Conditional output *LOUT* will be low during state *AC*, and the next state will be *AD*.

Now, study Fig. 1-27*c*. In this timing diagram we have purposefully made *NRT* asynchronous. Changing *NRT* in the middle of state *AC* simply changes the address sent to the ROM. As soon as the ROM looks up this new address, the conditional output will change. Thus, the conditional output will also be asynchronous if it de-

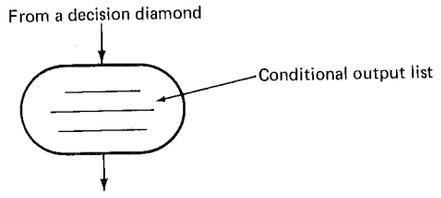


Figure 1-25 Conditional output box.

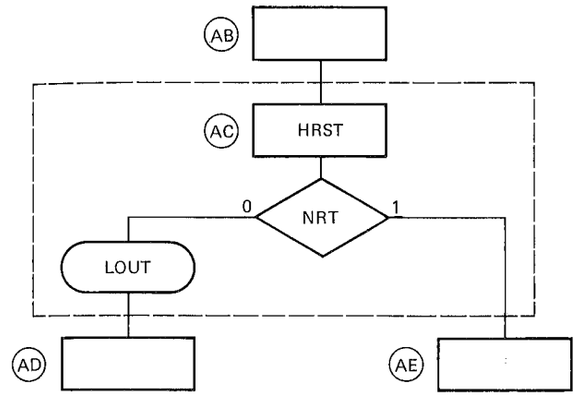


Figure 1-26 ASM block with conditional output.

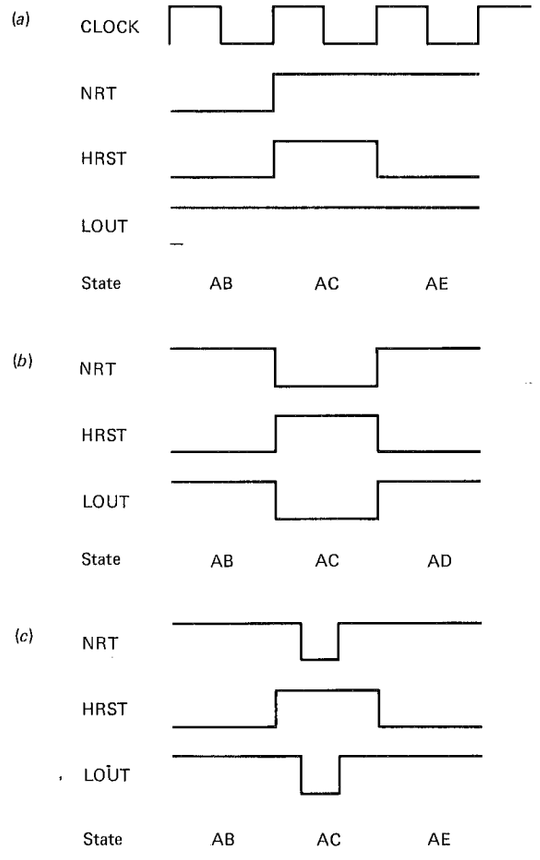


Figure 1-27 Conditional output timing diagrams.

pends on an asynchronous input. However, the transition out of state *AC* is made at the next clock transition. Since the input *NRT* has returned to a 1 by this clock transition, the next state will be *AE*. Because *NRT* was asynchronous, the machine asserted *LOUT* and went to state *AE*, even though this seems impossible from the ASM chart in Fig. 1-26.

Problems and Abbreviations

Each exit path from an ASM block leads to only one state. In addition, the decision structure of the block must not indicate two or more simultaneous exit paths for any set of inputs. Since the actual machine can only be in one state at a time, an ASM chart that specifies two next states would be meaningless. Never divide an exit into two or more paths without using a decision diamond. On occasion, it is convenient to divide an exit path for conditional outputs, as shown in Fig. 1-28. Both of the ASM blocks in Fig. 1-28 perform the same function. In each case only one next state can be specified, regardless of the inputs. The form of Fig. 1-28*a* is not recommended except for the most simple cases, like the one illustrated. Although two next states can't be simultaneously specified, two conditional outputs can be specified simultaneously. Figure 1-28*a* will work because all exit paths converge to the same next state. The form of Fig. 1-28*b* is preferred because it *ensures* that only one next state is specified and will always unambiguously specify conditional outputs in more complex charts.

Two abbreviations are commonly used in ASM charts. First, the dotted line around each ASM block is omitted. This causes no problems since an ASM block always

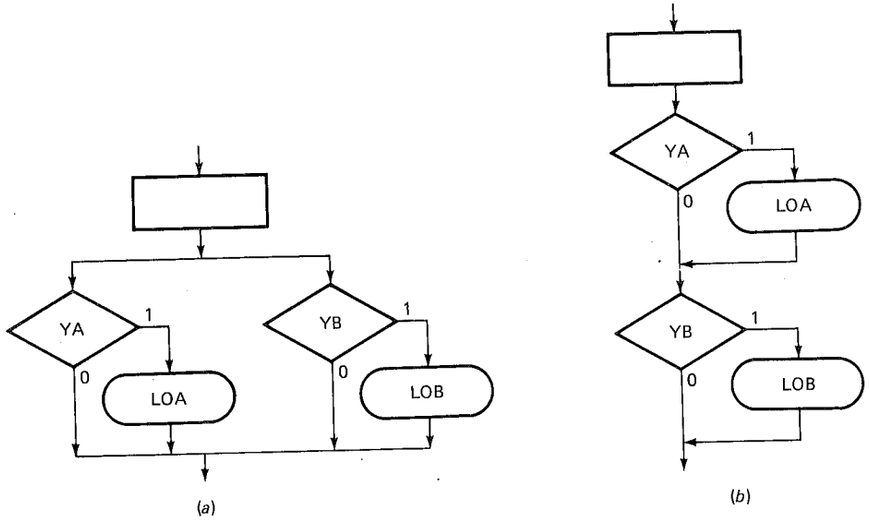


Figure 1-28 (*a*) Parallel paths for conditional outputs. (*b*) Series paths for conditional outputs (preferred).

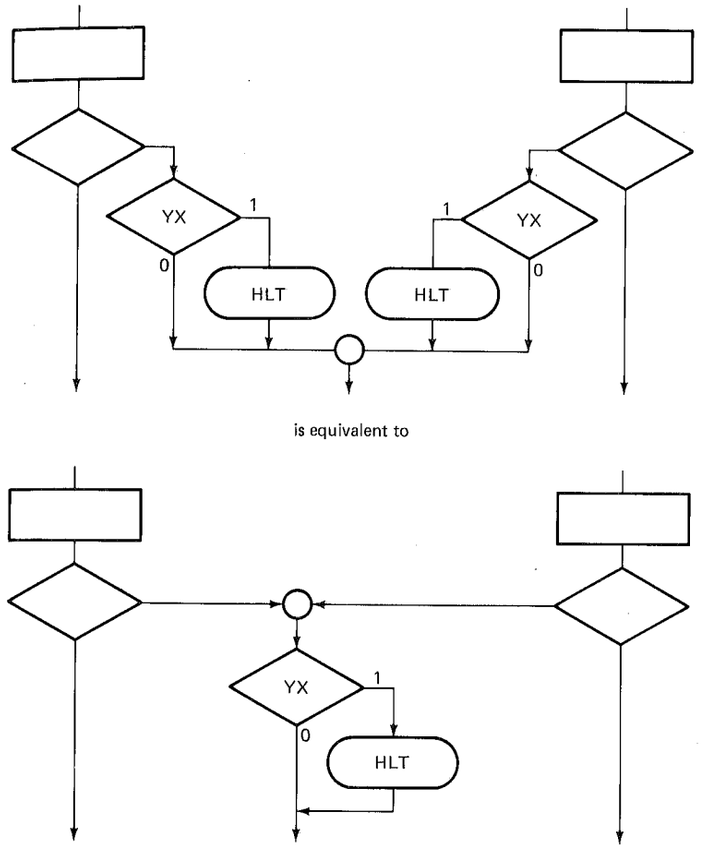


Figure 1-29 Sharing decisions and conditional outputs.

consists of a single state box and all decision diamonds and conditional output boxes in its exit paths. Second, decision diamonds and conditional output boxes may be shared among ASM blocks, as shown in Fig. 1-29. This abbreviation *does not affect the implementation* in any way. It simply reduces the amount of drawing required for the ASM chart.

DESIGN PROCEDURE FOR ASMS

The design procedure for ASMs may be summarized as follows:

1. Write a specification of the problem to be solved.
2. Translate this specification into an overall flowchart if necessary to clarify operation.

3. Develop a block diagram of the system including signal definitions.
4. Implement each block in the block diagram.
   a. Develop the algorithm for each block, either in written or flowchart form.
   b. Translate these algorithms into ASM charts.
   c. Implement the ASM charts with logic circuits.

In order to show you how all the procedures we've discussed so far fit into this framework, let's work an example following the steps outlined above..

A DESIGN EXAMPLE FOR AN AUTOMATED BANK TELLER

The problem to be solved is to design an automated bank teller that will dispense cash if a customer enters a correct account number and a correct amount. This written description corresponds to step one in the general design procedure. A simple, overall flowchart may be drawn, as shown in Fig. 1-30, to complete step 2 of the procedure. This flowchart tells us several things about the elements of the block diagram. First, since the customer must enter the account number and amount, the teller must be able to prompt the customer to enter these numbers. Thus, the block diagram of Fig. 1-31 includes three lighted messages that are used to prompt the customer. Second, the flowchart shows that not only must numbers be entered by the customer, but also that those numbers must be compared to determine their validity. Thus, the block diagram has a comparator circuit associated with the keyboard, used to enter the

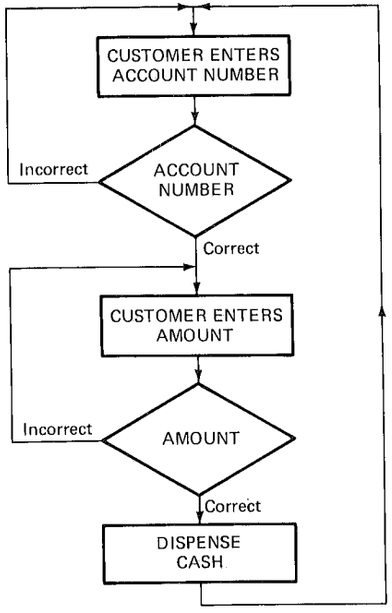


Figure 1-30 Overall flowchart for bank teller.

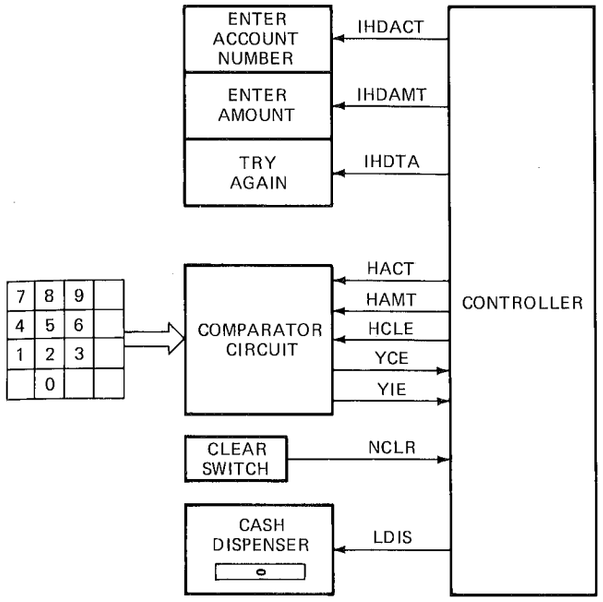


Figure 1-31 Automated bank teller block diagram.

numbers. Third, the flowchart dispenses cash, which implies that a cash dispenser must appear on the block diagram. Fourth, you should notice from the overall flowchart that the ASM could "get stuck" in the middle of a transaction if the customer fails to complete his transaction. A *CLEAR* switch is included in the block diagram so that a new customer can force the teller to start a new transaction, even if the last transaction was never completed. Finally, a controller block connects to all the other blocks to sequence the overall operation of the bank teller. This arrangement of blocks in which a controller block directs the operation of input blocks (keyboard and clear switch), output blocks (displays and cash dispenser), and processing blocks (comparator) is very common. We can complete step 3 of the general design procedure by specifying the various signals that connect the blocks together:

IHDACT. Display "Enter Account Number."
IHDAMT. Display "Enter Amount."
IHDTA. Display "Try Again."
HACT. Set comparator to accept the account.
HAMT. Set comparator to accept the amount.
YCE. A correct entry has been made.
YIE. An incorrect entry has been made.
HCLE. Resets both *YCE* and *YIE* to 0.
LDIS. Dispense the cash.
NCLR. The clear switch has been pressed.

All outputs of the controller must be continuously asserted as long as that action is desired. For example, *HAMT* must be a 1 as long as the customer is supposed to be entering the amount. If neither *YCE* nor *YIE* is asserted, no entry has been made or an entry is in progress. After a correct entry, *YCE* remains a 1. After an incorrect entry, *YIE* remains a 1. Asserting *HCLE* will reset either or both *YIE* and *YCE* to 0. The clock period is very short, say 50 ms. It is short enough that the message may be lighted simultaneously with asserting the corresponding comparator control signal. Even though the comparator control signal is delayed-action, the customer can't see the message and respond to it in such a short time as 50 ms. Now, we can proceed to step 4, which tells us to implement each block. We'll only implement one block in this example, the controller block. Figure 1-32 is a flowchart describing the algorithm to be implemented by the controller block. It is *not* an ASM chart, but merely serves as a guide to develop such a chart. We could have used a written description for the controller algorithm if it had seemed more appropriate.

The flowchart in Fig. 1-32 and the logic signal specifications given previously

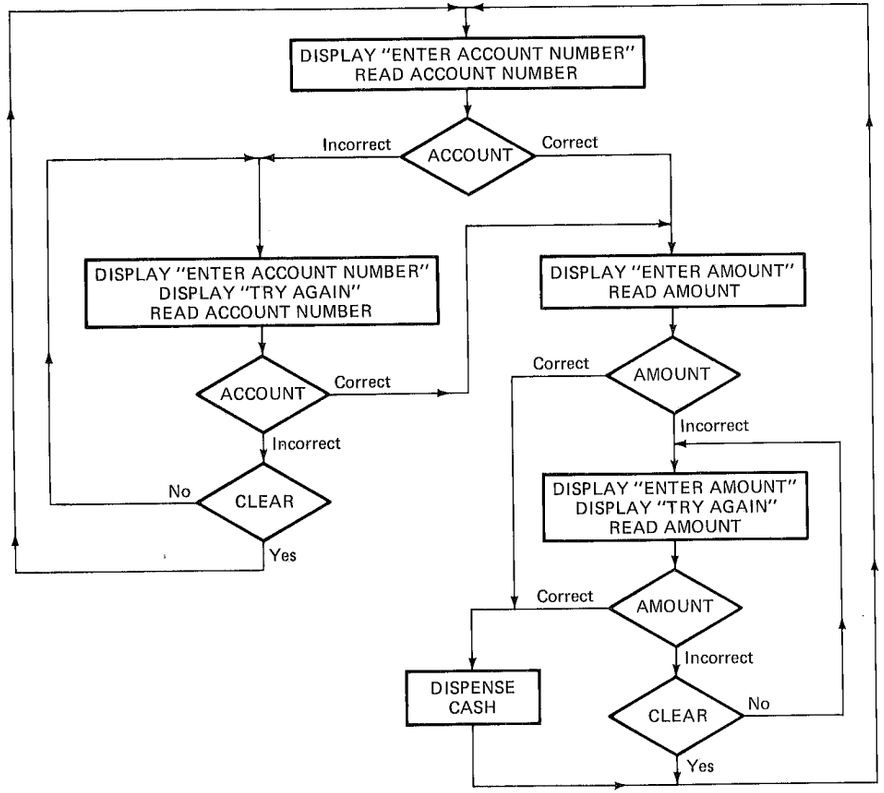


Figure 1-32 Flowchart for bank teller controller.

Table 1-7 Chart to convert algorithm to ASM chart for bank teller without conditional outputs

State		Outputs	YCE	YIE	NCLR	Next state	Comments
A	Read account number	HACT, IHDACT	0	0	×	A	Display and read account number if no entry
			1	×	×	C	Correct entry
			0	1	×	B	Incorrect entry
B	Try again to read account number	HACT, IHDACT	1	×	×	C	Correct entry
		IHDTA	0	×	1	B	No entry or incorrect entry, stay here
			0	×	0	D	Clear switch pressed
D	Clear YIE and YCE before new account number	HCLE	×	×	×	A	Extra state needed to reset YIE and YCE Reset occurs on clock edge that causes transition to state A
C	Clear YIE and YCE before amount entry	HCLE	×	×	×	E	Same as state D except that read amount is next state
E	Read amount	HAMT, IHDAMT	1	×	×	F	Correct entry, go get cash
			0	0	1	E	Display and read amount if no entry
			0	0	0	A	Clear switch pressed. State D not needed because YIE = YCE = 0
			0	1	×	G	Incorrect amount entered
F	Dispense cash and clear YIE and YCE	LDIS, HCLE	×	×	×	A	Dispense cash and go back to get another account
G	Try to read amount again	HAMT, IHDAMT	1	×	×	F	Correct entry, dispense cash
		IHDTA	0	×	1	G	No entry or another incorrect entry
			0	×	0	D	Clear switch pressed

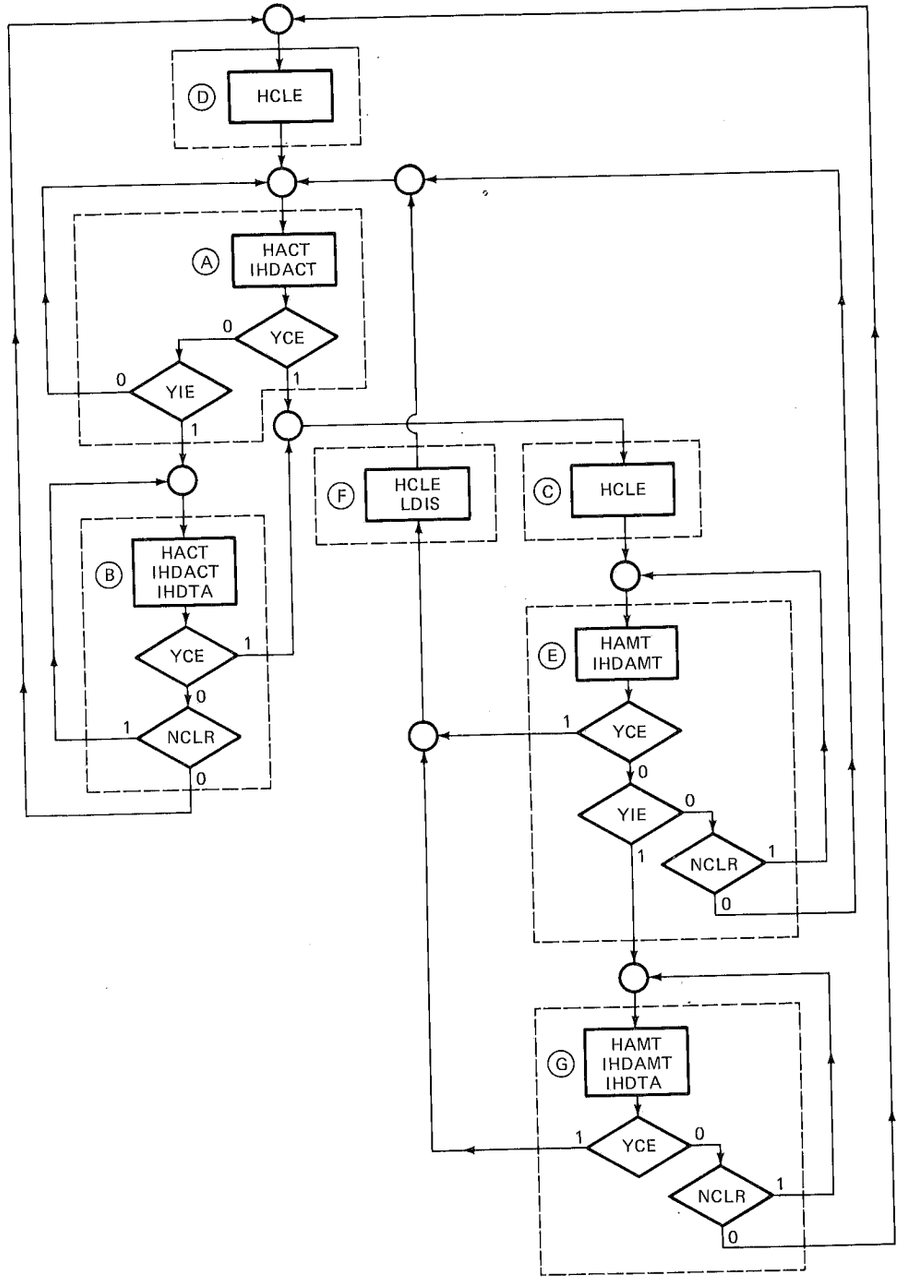


Figure 1-33 ASM chart for bank teller without conditional outputs.

must now be merged to form an ASM chart. Let's assume that we desire to implement this algorithm without conditional outputs. The thought process used in developing the ASM chart is shown in Table 1-7. We start in a state easily determined from the algorithm specification. In this case, there obviously must be an initial state to read the account number entered by the customer and it must have outputs *HACT* and *IHDACT*. Note that we are trying to *implement a sequence of outputs* required; so both state and outputs appear first in our chart. After we've listed state and outputs, we list inputs so that the next state can be determined. Although we have three inputs, all combinations need not be listed as several "don't care" conditions exist. For exam-

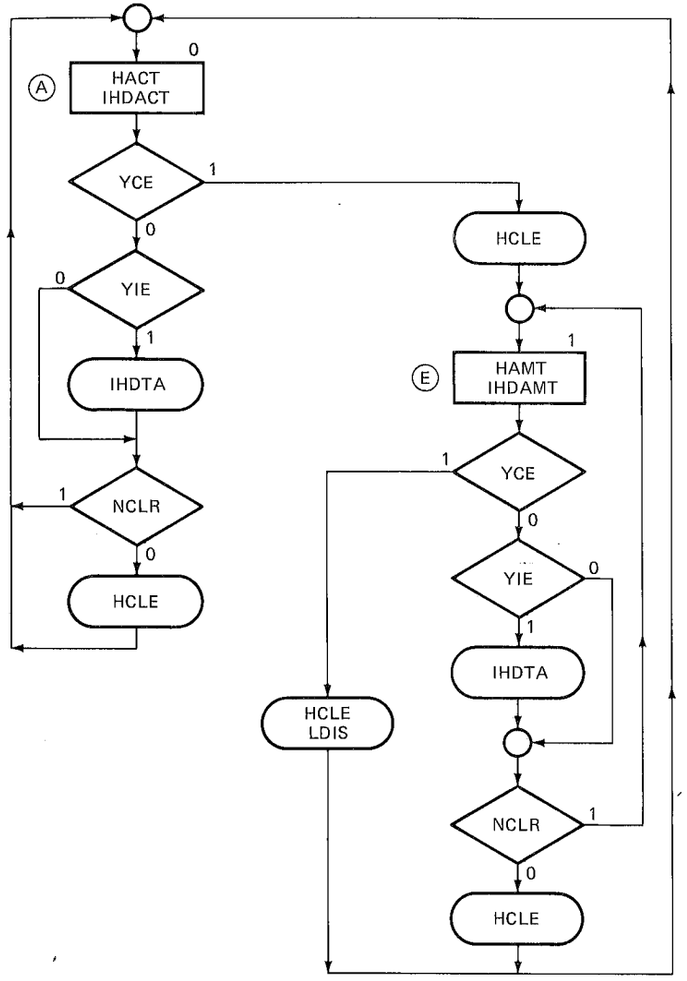


Figure 1-34 ASM chart for bank teller with conditional outputs.

ple, if *YCE* = 1, that is, a correct entry has been made, we need not worry about the clear switch or the incorrect-entry inputs. In the process of filling out the next-state information for state *A*, we determine a need for two additional states. State *C* is required for a correct entry, while state *B* is required for an incorrect entry. The next step is to list these new states and their outputs, which may, in turn, give rise to additional states. Eventually, no new states will be required (since an algorithm has a finite number of steps) and the chart will be finished. Table 1-7 may be immediately drawn as the ASM chart of Fig. 1-33. The intermediate step of generating Table 1-7 may not always be necessary. You may find it easier to draw only the chart of Fig. 1-33, especially if the algorithm is short and simple. Even if you draw the ASM chart directly, you still use the thought process represented by Table 1-7. Before we continue, you should notice that we needed two states *C* and *D* to do nothing but assert *HCLE* one clock transition ahead of the state in which *YIE* and *YCE* must be reset. Since *HCLE* is a delayed-action output, these additional states were needed to assure that the previous comparator outputs were cleared before a new decision was made.

How would the controller block change if we were to allow conditional outputs? Figure 1-34 shows the bank-teller ASM chart using conditional outputs where appropriate. Conditional output boxes were used to assert *IHDTA* when an incorrect entry was made. The "TRY AGAIN" display will be lighted without requiring additional states. A conditional output box is also used to assert *LDIS* when the cash is to be dispensed. Conditional output boxes were used to assert *HCLE*. In this way, the same clock transition that causes the next state to occur will also reset *YCE* and *YIE* when desired. This saves the extra states *C* and *D* in Fig. 1-33 needed to assert *HCLE* before the comparator inputs were changed. Figure 1-35 is the timing diagram for the bank teller without conditional outputs. State *C* must be added to ensure that *HCLE* is asserted one state before state *E* in order to reset *YCE* at the correct time. The timing diagram of a bank teller using conditional outputs is shown in Fig. 1-36. The input *YCE* can cause the conditional output *HCLE* to occur *while still in state A*. An additional state *C* is not needed to ensure that *YCE* will be reset at the beginning of state *E*.

Finally, only specifying the ROM contents completes our design procedure. Figure 1-37 and Table 1-8 show the implementation of the controller with conditional

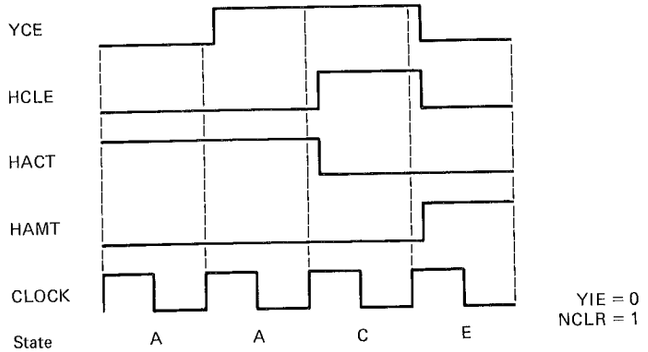


Figure 1-35 Timing diagram for bank teller without conditional outputs.

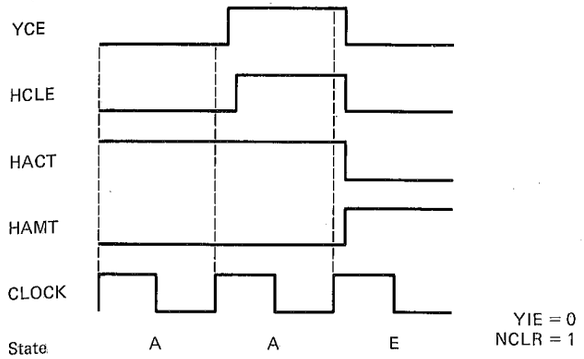


Figure 1-36 Timing diagram for bank teller with conditional outputs.

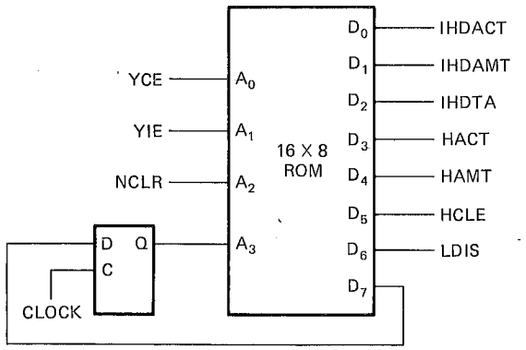


Figure 1-37 Bank teller with conditional outputs.

Table 1-8 ROM contents of bank teller with conditional outputs

Current state A_3	NCLR A_2	YIE A_1	YCE A_0	Next state D_7	LDIS D_6	HCLE D_5	HAMT D_4	HACT D_3	IHDTA D_2	IHDAMT D_1	IHDACT D_0
0	X	X	1	1	1	1	0	1	0	0	1
0	1	0	0	0	1	0	0	1	0	0	1
0	1	1	0	0	1	0	0	1	1	0	1
0	0	0	0	0	1	1	0	1	0	0	1
0	0	1	0	0	1	1	0	1	1	0	1
1	X	X	1	0	0	1	1	0	0	1	0
1	1	0	0	1	1	0	1	0	0	1	0
1	1	1	0	1	1	0	1	0	1	1	0
1	0	0	0	0	1	1	1	0	0	1	0
1	0	1	0	0	1	1	1	0	1	1	0

outputs. This example is a good illustration of the advantages of conditional outputs. The controller without conditional outputs has six states, requiring three flip-flops and a 64 X 10 read-only memory. The controller with conditional outputs has only two states, requiring one flip-flop and a 16 X 8 read-only memory!

DESIGN CRITERIA

In many respects, you now know all there is to know about designing digital circuits. The rest of this book will cover other implementation techniques used to reduce development effort or circuit cost. You now understand the basic principles of digital design, although you might have difficulty recognizing what you've learned if you immediately turned to the section of this book on microcomputers. Since the rest of this book will be devoted to "refinements" of the basic ASM, we should logically ask the question: What makes any additional technique we might learn an "improvement"? What are the criteria by which we may judge digital designs? This is a very difficult question to answer because criteria for good designs have changed over the years and are changing right now.

When all electronic components were expensive, good design techniques focused on reducing the number of components required. The Karnaugh maps of the next chapter were among many techniques developed to systematically reduce the number of components required. With the advent of the integrated circuit, the electronic components themselves were essentially "free." The cost of a system was related to packaging and interconnection costs (as well as the cost of power supplies, cooling fans, testing, and many other peripheral factors). With the advent of microcomputers and other large integrated circuits, the cost of *developing* the digital system is likely to be paramount. Our criteria for "good" digital system design will consist of only two factors (in order of importance):

1. Minimize the cost of development
2. Minimize the cost of the hardware of each system while maintaining the required level of performance

While these criteria seem obvious, their application is not always simple. For example, a large portion of the cost of development usually consists of the cost of finding mistakes in the original design (called debugging). It pays to take extra care (and spend additional time and money) with the initial design because these extra costs are more than made up by the lessened cost of debugging. This book will continually stress *systematic* design procedures, even at the expense of additional circuit cost, just to reduce design costs.

Even the hardware cost criteria are not obvious. Testing and repairing digital systems often account for a large part of hardware cost. It may pay to add additional interconnections and circuitry to reduce the cost of testing and repairing the system.

PROBLEMS

1-1 (*a*) Count in binary and hex from 0 to 31 decimal. List the binary and hex numbers equivalent to each decimal number on the same line.

(*b*) Count in two's-complement binary from −16 to +15 decimal. List the binary number equivalent to each decimal number on the same line.

1-2 Figure P1-1 shows a D flip-flop and a partial timing diagram.

(*a*) Fill in the output in the timing diagram as it would be seen on an oscilloscope.

(*b*) Make a table listing both D input and Q output as would be generated by a logic-state analyzer.

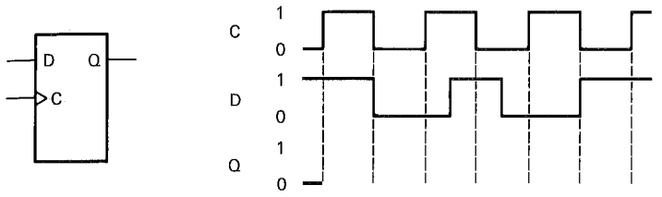


Figure P1-1 D flip-flop timing diagram.

1-3 Figure P1-2 shows a ROM and a partial timing diagram. The contents of this ROM are:

A_2	A_1	A_0	D_1	D_0
0	0	0	0	0
0	0	1	1	1
0	1	0	1	0
0	1	1	1	1
1	0	0	1	1
1	0	1	0	1
1	1	0	0	0
1	1	1	0	0

(*a*) Complete the timing diagram by filling in the outputs.

(*b*) Make a table listing only the outputs D_1 and D_0 as they would be seen on a logic-state analyzer.

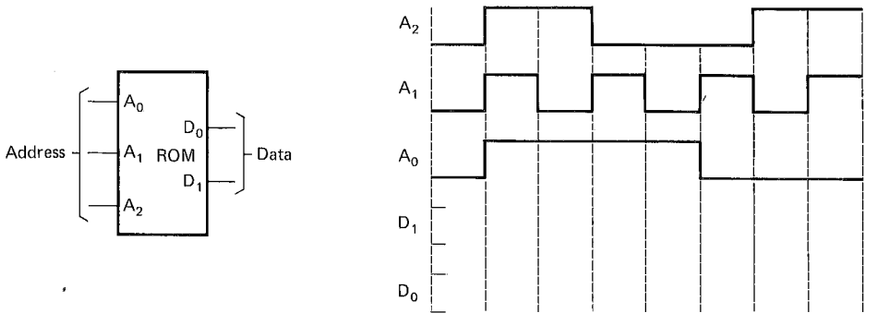


Figure P1-2 ROM timing diagram.

1-4 Consider a traffic intersection combining traffic sensors as described in Fig. 1-5 and an emergency-vehicle input as described in Fig. 1-7.

(*a*) Draw a circuit diagram and ASM chart for this controller.

(*b*) Make a ROM contents table and convert it to hex.

1-5 Figure P1-3 is a diagram of a very simple rapid-transit system with three stations.

(*a*) Draw an ASM chart and circuit diagram that will move a train back and forth over the line, stopping at each station for 10 s. Track sensor inputs *A*, *B*, and *C* correspond to each station. The train must stop within 2 s of receiving a track sensor input to keep from overshooting the platform. The conductor has an override button which causes an input *NOVER* to become true whenever he wants to extend the 10-s loading delay to load more passengers. You'll use three outputs: Move East (*HEAST*), Move West (*HWEST*), and Stop Moving (*STOP*).

(*b*) Make a complete ROM contents table for this controller.

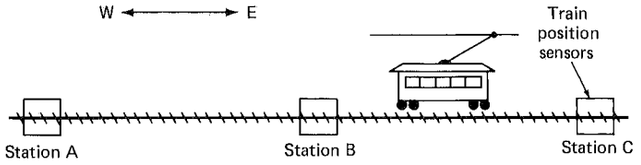


Figure P1-3 Commuter railroad.

1-6 Figure P1-4 shows a more complex rapid-transit system with an additional station (*D*) on a branch line. The train should alternate trips to *A* and *D* along the main line and branch line, respectively. Of course, it stops at *B* on each trip.

(*a*) Draw the ASM chart and circuit diagram for this transit system, following the additional operating specifications of Prob. 1-5. Besides one additional track sensor input *D*, you'll need two more outputs *HMAIN* and *HBRANCH* to throw the switch.

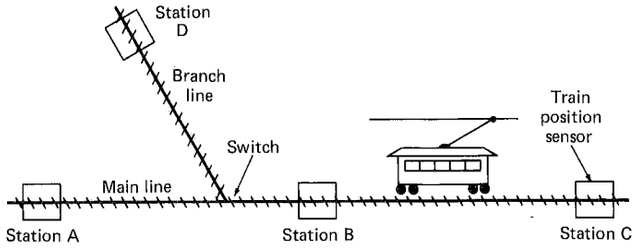


Figure P1-4 Commuter railroad with branch line.

1-7 A washing machine controller is to be constructed. The controller has the following inputs:

YHOT. 1 if hot/cold switch specifies hot water wash
NSTRT. 0 to start washing; 1 to stop, even in midcycle
YFULL. 1 if water filled to top
YEMPTY. 1 if water completely empty
YTIME. 1 if timer indicates done

The controller must operate the following outputs:

HHOT. 1 to select hot water
0 to select cold water

LPUMP. 0 to turn on water pump
HFILL. 1 to direct water into washer
0 to direct water out of washer
LAG. 0 to agitate wash and start timer, set *YTIME* to 0
LSPIN. 0 to spin wash and start timer, set *YTIME* to 0

When the controller receives a start signal, it fills the washer with the correct temperature of water and agitates until the timer indicates it is done. It empties the soapy water and fills the washer with cold rinse water and agitates again until the timer indicates it is done. Finally, it spins the clothes dry after emptying the rinse water. Draw the ASM chart for this controller. Use conditional outputs.

1-8 Draw an ASM chart with correct input and output mnemonics to implement an ordinary traffic light with an emergency input. The emergency input sets both the EW and NS lights to red as before. However, when the emergency button is released, the traffic lights continue sequencing *from the state in which they were when the emergency button was pressed.* The easiest way to accomplish this is to use conditional outputs.

1-9 Design a controller with two inputs and one output. The inputs are *YA* and *YB*. The output is *HT*. This controller responds only to 0-to-1 *transitions* on its inputs. The output *HT* sets to a 1 if a 0-to-1 transition occurs on *YA*. The output resets to a 0 if a 0-to-1 transition occurs on *YB*.