2-5 Draw circuit diagrams that implement the following logical expressions with AND gates, OR gates, and inverters.

(*a*) $AB + CD$
(*b*) $\bar{A}B + A\bar{B}$
(*c*) $(\bar{A}\bar{B} + AB)\,D + CE$
(*d*) $(\bar{A}\bar{B} + C)\,(D + E)$

2-6 Repeat Prob. 2-5 using:

(*a*) Only NAND gates and inverters
(*b*) Only NOR gates and inverters

2-7 Design two circuits to implement the flowchart of Fig. P2-2.

(*a*) Draw the circuit for a ROM controller, and specify the ROM contents.
(*b*) Draw the circuit for a gate controller.

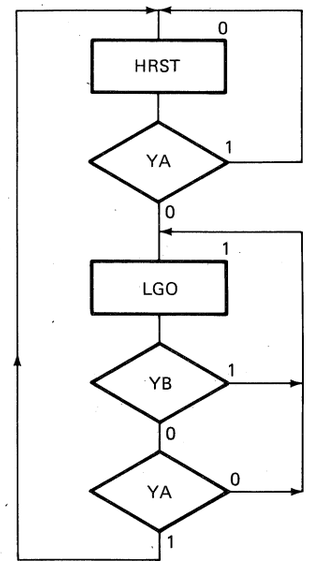


Figure P2-2 ASM chart for Prob. 2-7.

2-8 Design a gate controller for Prob. 1-9.

2-9 The K maps in Fig. 2-16 could have don't cares in positions 10, 11, 12, 13, 14, and 15 because these states were used in the ASM. Determine the minimum SOP representation for these 10 functions if the don't cares are included.

2-10 (*a*) Assume you are using a logic family that has wired ANDs. Repeat Prob. 2-6*a*, using the wired AND function.

(*b*) Assume you are using a logic family that has wired ORs. Repeat Prob. 2-6*b*, using the wired OR function.

CHAPTER

THREE

MSI AND LSI CIRCUITS

Examining many digital circuits, we should notice that similar circuitry and similar algorithms are used repeatedly. These functions are so common that they are manufactured as standard integrated circuits that may be incorporated into any digital system. The economies of scale reduce the cost of these mass-produced parts, regardless of the type of system in which they are used. Thus, the many computers manufactured help reduce the cost of traffic light controllers, and vice-versa. More importantly, these commonly used functions are preengineered. The cost of designing these circuits may be amortized over millions of individual parts. This significantly reduces digital system development costs.

We have already examined three types of mass-produced parts: gates, flip-flops, and ROMs. Gates and flip-flops are called small-scale integrated (SSI) circuits because they contain relatively few electronic components. In this chapter, we'll be primarily concerned with medium- and large-scale integrated (MSI and LSI) circuits. These larger circuits are advantageous because they allow complex functions to be preengineered and mass-produced. Before we examine common MSI circuits and their uses, we shall introduce digital signal busses.

BUS NOTATION

Often digital signals are grouped together for a common purpose. On a circuit diagram they appear as parallel signal lines, running together to various circuits. These signals are collectively called a bus. Because busses are difficult to draw and require a great amount of space on the diagram and their drawings can be confusing and difficult to follow, a shorthand notation has been developed. In Fig. 3-1, both the state variables (*SV0* to *SV3*) and the next-state function (*NS0* to *NS3*) are drawn as busses. Each

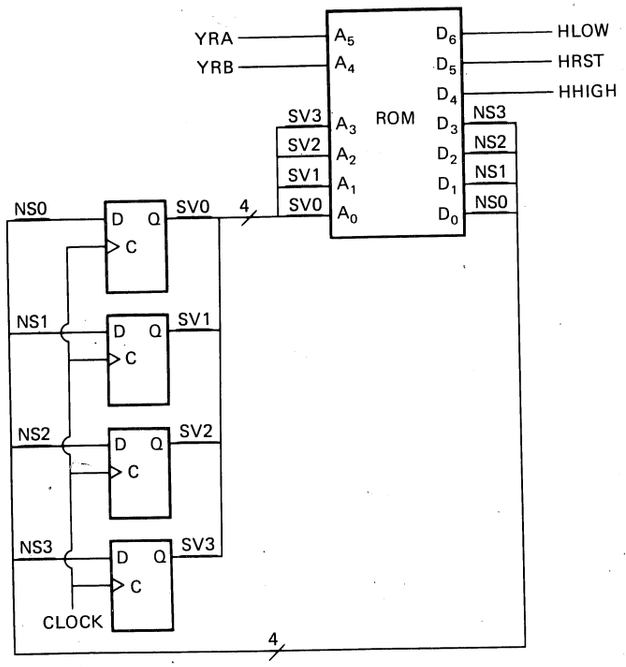


Figure 3-1 Notation for bus.

group of signals or bus is drawn as a single line. The number 4 adjacent to the slash mark indicates that this single line actually represents four digital signals. These signals must be separated whenever they connect to a logic circuit and plainly marked at these connections with their mnemonics.

DATA BUSSES

One common digital bus is the data bus. In Fig. 3-2, an 8-bit data bus allows data (say, 8-bit binary numbers) to be passed among three different ASMs. Control signals determine the origin and destination of data. Data must be transmitted bidirectionally over the same eight wires among three different ASMs. Two circuit techniques may be used to accomplish this data transfer.

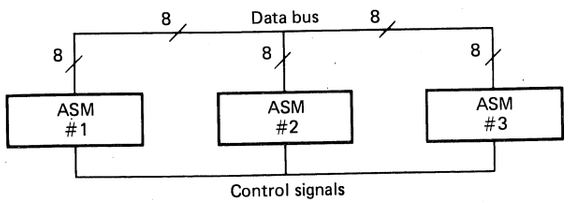


Figure 3-2 Bidirectional data bus.

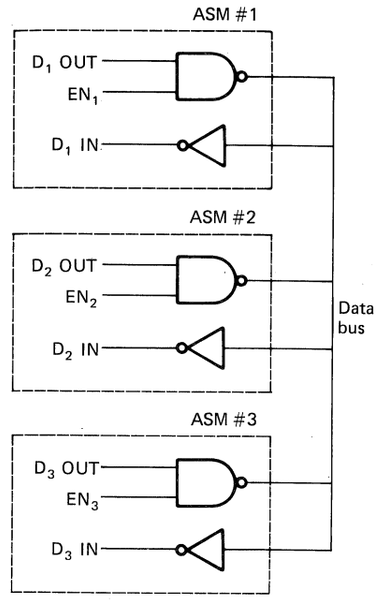


Figure 3-3 Wired AND data bus.

In Fig. 3-3, a wired AND function is used to allow more than one output to connect to a single wire of the 8-bit data bus. Each ASM may receive data from the bus through the inverter or may transmit data by enabling its NAND gate. This circuit is redrawn in Fig. 3-4*a* to explicitly show the wired AND function. Applying De Morgan's laws as shown in Fig. 3-4*b*, we obtain the circuit of Fig. 3-4*c*. From this last circuit, it is easy to see that each ASM may gate its data onto the bus, which will be received by all other ASMs, if it enables its AND gate. Only one ASM at a time may enable a transmitting AND gate. The bus acts as an OR function (because of De Morgan's laws), even though a wired AND is actually implemented. In fact, this kind of data bus is often called a wired OR bus, even if the wired AND function is actually being used, as in this case. We mentioned in Chap. 2 that wired functions slow down the digital signal's propagation through the gates that have their outputs connected together. To speed up the operation of data busses, a new kind of logic circuit was developed to replace the wired function. This new kind of logic circuit is called a *three-state* circuit.

A three-state circuit is identical to its ordinary circuit counterpart except for its output. A three-state inverter is shown in Fig. 3-5. It is an ordinary inverter with a logic-controlled switch in its output lead. When the three-state enable signal is a 0, the switch is open and the inverter output is disconnected. This condition is not a logic 0 or a logic 1, it is a third state called a *high-impedance* or *high-Z* state. Since we may now disconnect logic circuit outputs internally, we may fabricate a data bus as shown in Fig. 3-6. As long as only one three-state enable signal is asserted, only one inverter is connected to the bus allowing the ASM to send data to the others. This bus is not a wired AND or OR because only one output is really connected to the bus at

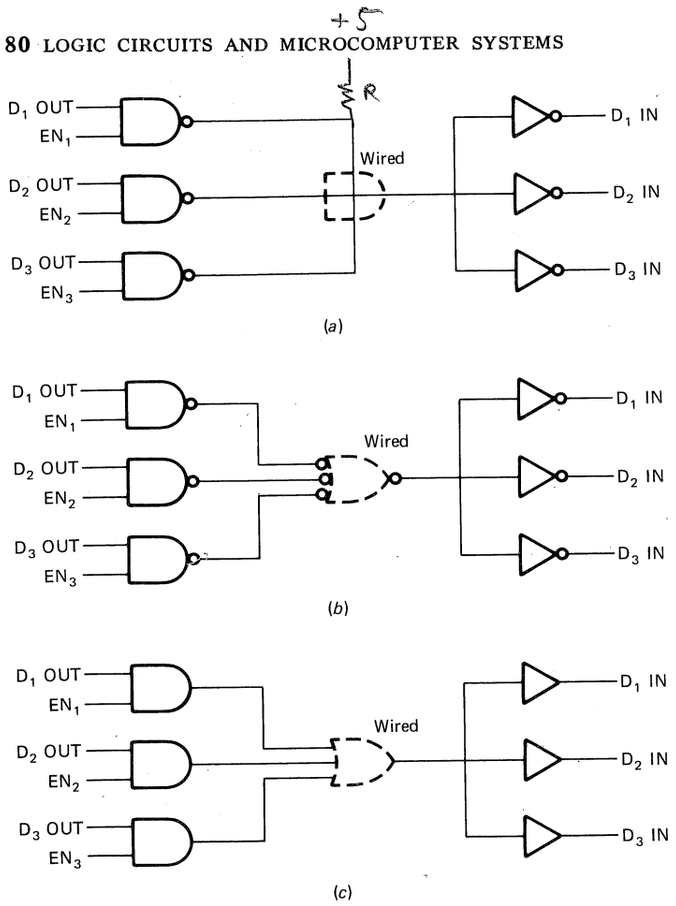


Figure 3-4 (*a*) Wired AND. (*b*) De Morgan's law. (*c*) Equivalent circuit.

any time. The other outputs are disconnected by switches located in the digital circuits.

MULTIPLEXERS

A *multiplexer* (also called a data selector) is a group of gates that can be used to selectively route data from several inputs to one output. Figure 3-7 is a four-line to one-

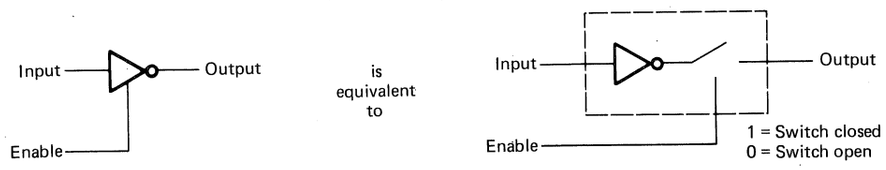


Figure 3-5 Inverter with three-state output.

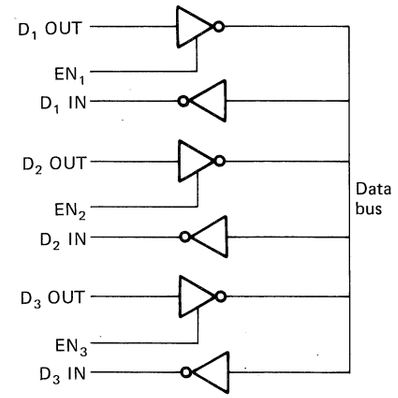


Figure 3-6 Three-state bus.

line multiplexer. The logic signal on any one of the four data inputs D_0 through D_3 can be made to appear on output Y. The signal to appear at Y is selected by inputs A and B. For example, if $B = 1$ and $A = 0$, signal 2 would appear on output Y. The complete logic table for this circuit appears in Table 3-1. Multiplexers are commonly available as MSI circuits. Figure 3-8 is the MSI circuit symbol for the multiplexer of Fig. 3-7. This MSI multiplexer has an extra output W, which is the complement of Y and is simply included for convenience. In addition, a *STROBE* (also called an

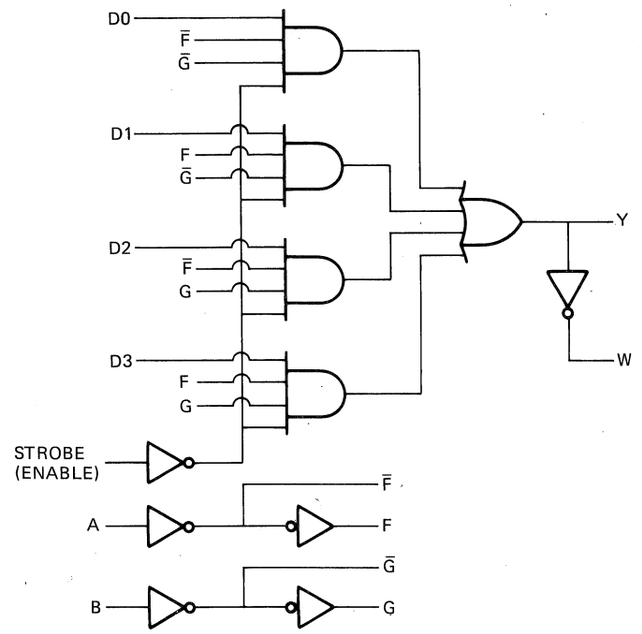


Figure 3-7 4-line to 1-line multiplexer.

Table 3-1 Logic table for multiplexer

STROBE	*B*	*A*	*Y*	*W*
1	X	X	0	1
0	0	0	D_0	$\overline{D_0}$
0	0	1	D_1	$\overline{D_1}$
0	1	0	D_2	$\overline{D_2}$
0	1	1	D_3	$\overline{D_3}$

ENABLE) signal input will set the output *Y* to 0, regardless of any data input. This *STROBE* input may also control a three-state output in some multiplexers. One or more independent multiplexers may be included in a single MSI circuit. Commonly available combinations are:

Single.	16 line to 1 line
Single.	8 line to 1 line
Dual.	4 line to 1 line
Quadruple.	2 line to 1 line

One of the more important uses of a multiplexer is to select input conditions to be tested by an algorithmic state machine. Four input conditions must be tested in the ASM of Fig. 3-9. A multiplexer is used to connect these four inputs to a single ROM address input. The select inputs of the multiplexer are connected to the state variables so that a different input may be tested in each state.

In state number	A_0 is connected to
0	*YINP*
1	*NFOR*
2	*NFOR*
3	*YINP*
4	0
5	*NRST*
6	*NFOR*
7	*YMAX*

The selected input is the condition to be tested in that state, as specified by an ASM chart. Since no input must be tested in state 4, multiplexer input D_4 is permanently

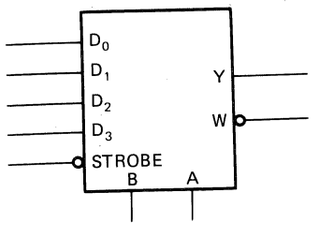


Figure 3-8 MSI multiplexer symbol.

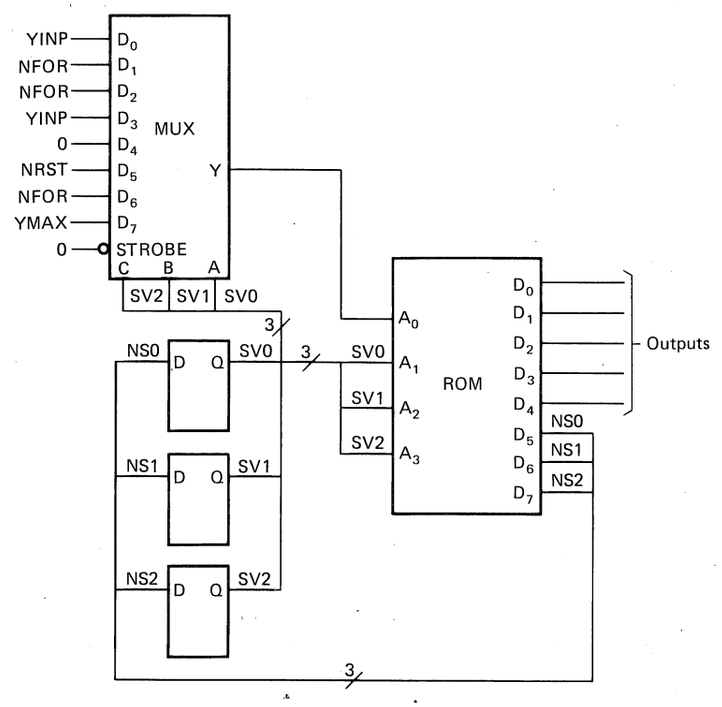


Figure 3-9 Multiplexer for input conditions.

connected to a 0. Without a multiplexer, a 128 X 8 ROM would've been required. With a multiplexer, a 16 X 8 ROM is required. This saving in ROM size is not without disadvantages, however. There can be only one decision diamond after each state box in the ASM chart because only one input is connected to the ROM in each state. Additional states may have to be added to the original ASM chart to test multiple inputs. Additional flip-flops may be required to implement these states. The ASM will respond to input changes more slowly because it will have to sequence through more states.

Multiplexers can still be used to advantage even if more than one input must be tested simultaneously. The machine in Fig. 3-10 is similar to the previous example, except that two multiplexers are used so that two inputs may be tested in each state. This increases the ROM size to 32 X 8, which is still far short of the original 128 X 8 ROM. A very complex ASM may have dozens of inputs to be tested. Seldom must more than two or three be tested simultaneously. In these cases, multiplexing is a necessity for it reduces the ROM size by orders of magnitude.

One more problem exists with our multiplexing scheme. For large numbers of states, the size of the multiplexer becomes excessive. For example, if our ASM had 7 state flip-flops, our multiplexing scheme would require 128-input multiplexers. One hundred and twenty-eight inputs would be needed, even if only a few input signals must be multiplexed. An alternative multiplexer connection is shown in Fig. 3-11.

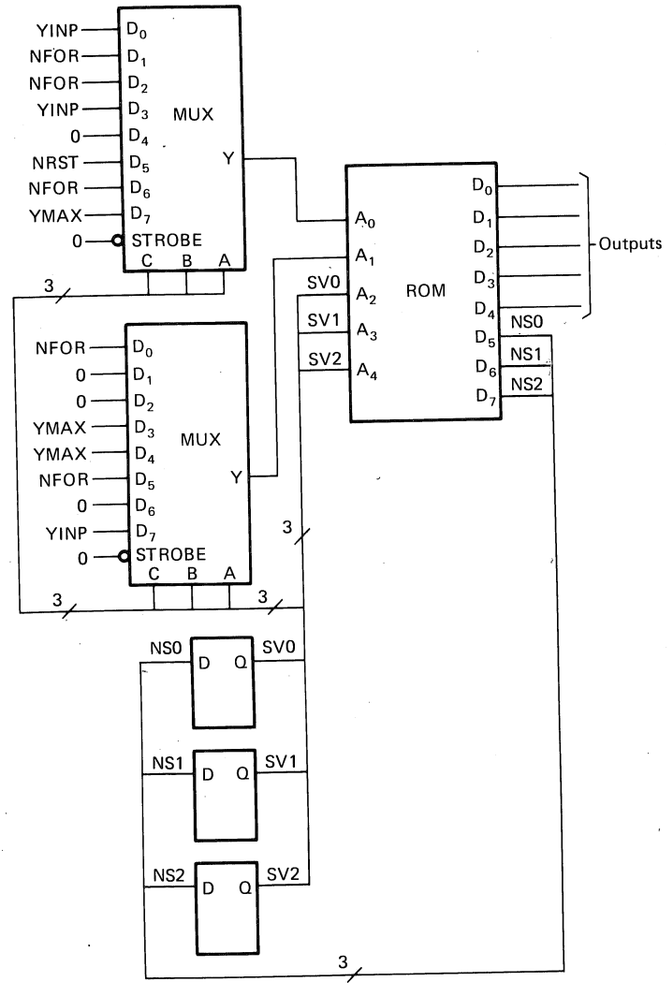


Figure 3-10 Using two multiplexers to test up to two conditions simultaneously.

The input selected by the multiplexer is controlled by ROM outputs, rather than the current state variables. In this way, the multiplexers need only be large enough to connect once to each required input. The ROM can be programmed to select any input in any state. In fact, multiple multiplexers may or may not be the same size. Some inputs that are tested very often may be routed to the ROM directly, without being multiplexed.

We've added three flip-flops to save the state of the multiplexer select lines. If we had tried to connect *SMA*, *SMB*, and *SMC* directly back to the multiplexer, we should have discovered that the circuit would probably malfunction. Remember that

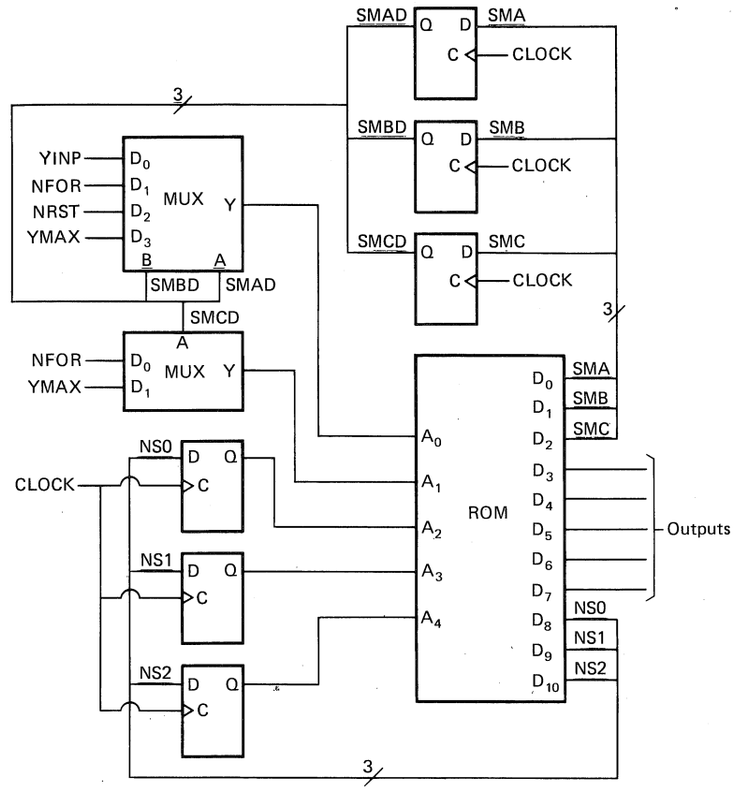


Figure 3-11 Multiplexers controlled by ROM outputs.

we assumed that all ROM address changes caused by state variable or input changes would be synchronous with the clock. If we had not added the three flip-flops in the select lines, a change in the select signals could change the ROM address asynchronously. This, in turn, could change the select signals, which could asynchronously change the address signals. This could change the select signals back again, etc. Thus, even without a clock transition, the circuit would repetitively change its outputs at some rate determined by the propagation delays of the ROM and multiplexer.

Although the three flip-flops synchronize the multiplexer select signals with the clock, they also delay the action of *SMA*, *B*, and *C* by one clock pulse. We shall have to assert the select signals one state early in the ASM chart.

Multiplexing ASM Inputs

A small tram shown in Fig. 3-12 must be automated. The tram car is to travel back and forth over the line, stopping for 10 s at each station. Three sensors *A*, *B*, and *C* sense the tram at the three stations. The tram must stop within 1 s of actuating a

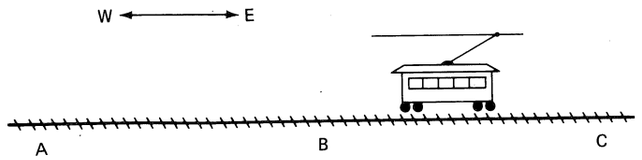


Figure 3-12 Tram.

sensor. An override button will keep the tram stopped whenever the button is pushed while the train is stopped at a station. The ASM chart for this example is shown in Fig. 3-13. Note that, at most, one input is tested in each state. For this reason, we should be able to multiplex the four inputs into one ROM address line. Since six flip-flops are required, we'll choose to control the multiplexer with ROM outputs. If we controlled the multiplexer with the 6 state variables, we'd need a 64-input multiplexer. The implementation of this controller is shown in Fig. 3-14. Two ROM outputs are used to select one of four inputs to be tested.

SMB	*SMA*	Input to be tested
0	0	*YA*
0	1	*YB*
1	0	*YC*
1	1	*YOVER*

The multiplexer select signals must be asserted in the state before they are needed because they actually reach the multiplexer after the next clock pulse loads them into the synchronizing flip-flops. The controller with multiplexer requires a 128 X 11 ROM (a total of 1408 bits), while a controller without the multiplexer would require a 1024 X 9 ROM (a total of 9216 bits).

Expanding Multiplexers

Multiplexers may be combined to form larger multiplexers in two ways. In Fig. 3-15, two 16-input, three-state output multiplexers are combined to form one 32-input multiplexer. The most significant selection bit, bit *E*, is used to enable the three-state output of one or the other multiplexer. Multiplexer outputs may be multiplexed, as shown in Fig. 3-16. Here, four 16-input multiplexers are cascaded to form a 64-input multiplexer. One of the outputs of the 16-input multiplexer is selected by a 4-input multiplexer connected to the most significant bits *E* and *F*.

Generating Logic Functions with Multiplexers

A multiplexer may be used to generate an arbitrary logic function. This use of multiplexers is advantageous when the function would involve a complex gating structure that would be difficult to design and implement. Multiplexer generation of functions

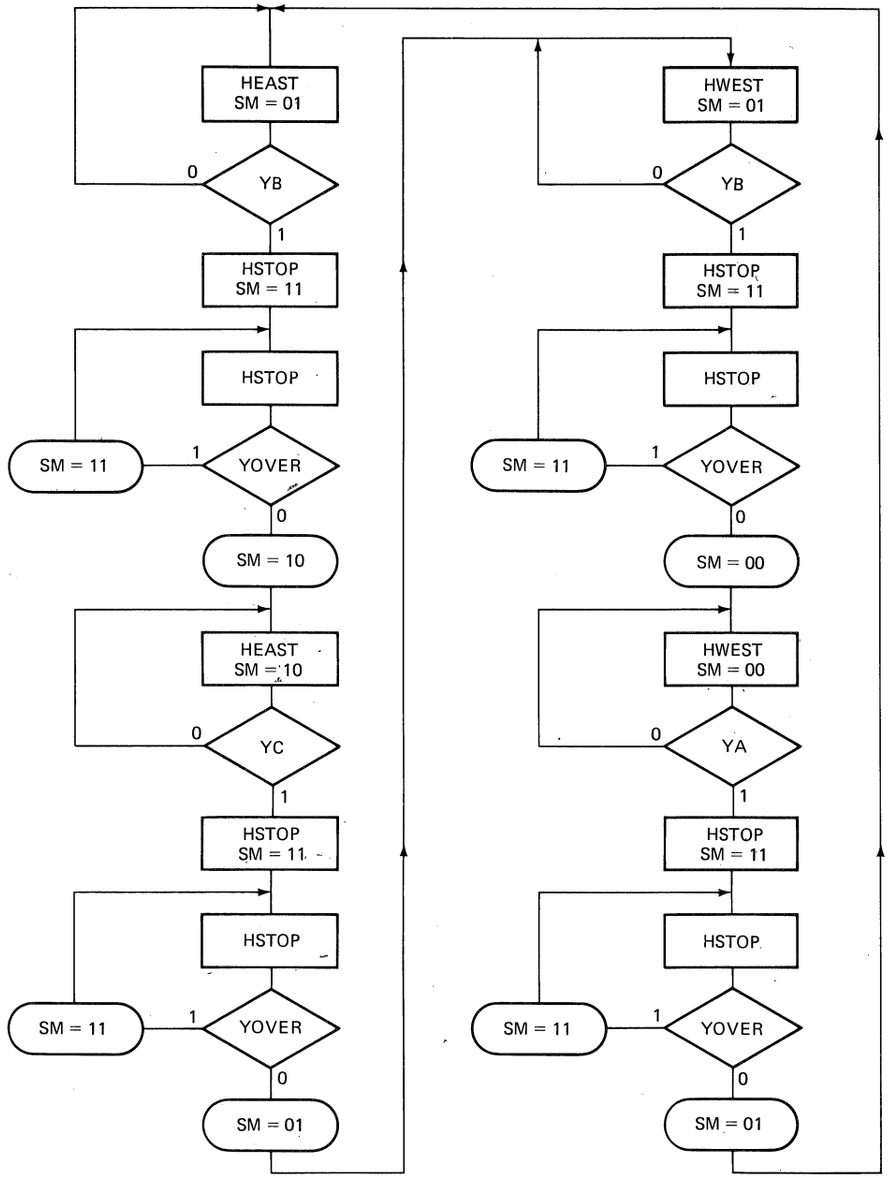
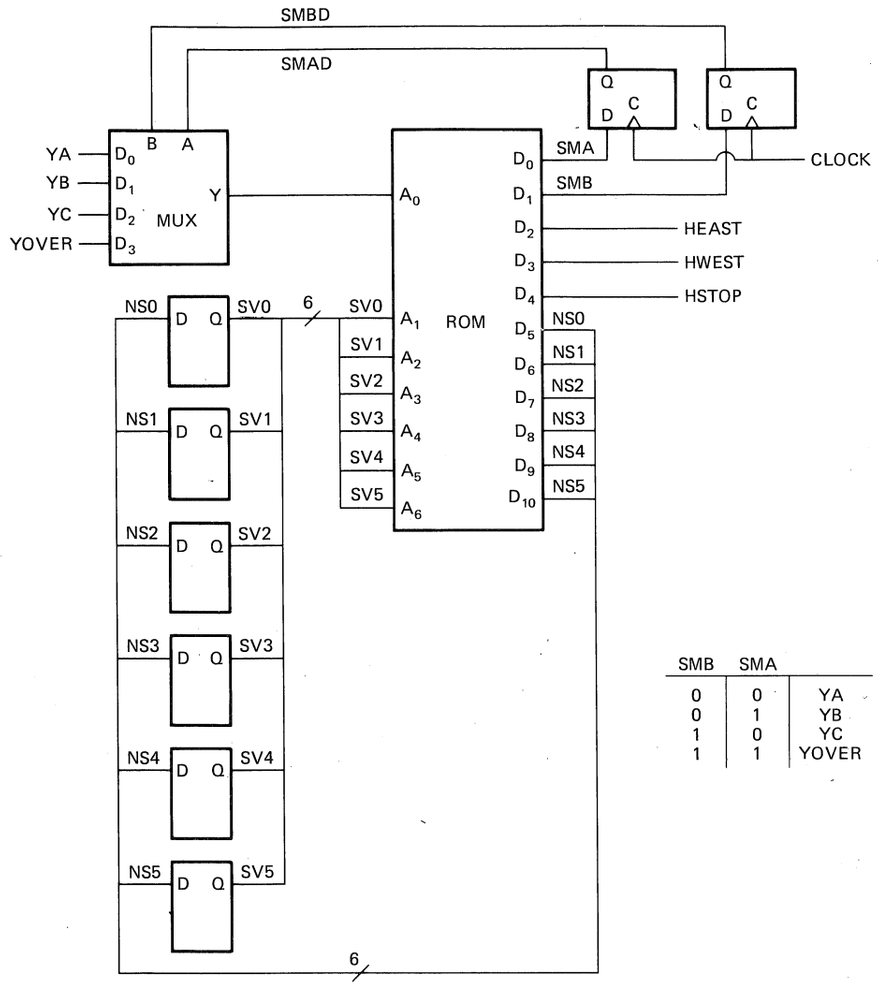


Figure 3-13 ASM chart for tram controller.



SMB	SMA	
0	0	YA
0	1	YB
1	0	YC
1	1	YOVER

Figure 3-14 Tram controller circuit.

is often useful when a few auxiliary functions must be generated in a ROM ASM. The multiplexer implementation is easily designed and the same single integrated-circuit multiplexer will generate *any* logic function.

A multiplexer can generate any logic function of $n + 1$ input variables if the multiplexer has n select inputs. Thus, an eight-input multiplexer can generate any logic function of up to four variables. A logic function of four variables is shown in Table 3-2. The most significant 3 bits of this function are connected to the select inputs of the multiplexer in Fig. 3-17. Thus, each multiplexer data input D_0 through D_7 corresponds to two lines of the function in Table 3-2. Looking at the first two lines of

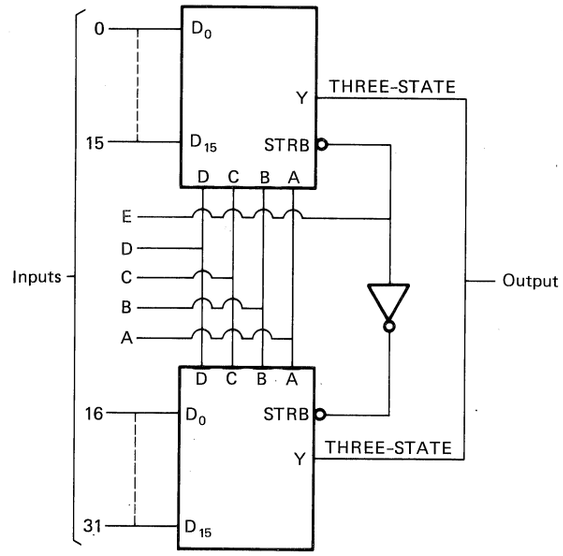


Figure 3-15 Expanding multiplexers with three-state outputs.

this table, we notice that, when D_0 is selected by inputs Z, Y, and X, the output F is identical to input W. We connect D_0 to input W on the circuit diagram. When D_1 is selected, the function F is always 1, independent of input W. Input D_1 is connected to 1 on the circuit diagram. For each pair of lines on the function table, we connected the corresponding multiplexer input to W, $\overline{W}$, 0, or 1 as required to generate the proper function.

Table 3-2 Function of four variables

Multiplexer input	Z	Y	X	W	F
D_0	0	0	0	0	0
	0	0	0	1	1
D_1	0	0	1	0	1
	0	0	1	1	1
D_2	0	1	0	0	0
	0	1	0	1	0
D_3	0	1	1	0	0
	0	1	1	1	1
D_4	1	0	0	0	1
	1	0	0	1	0
D_5	1	0	1	0	1
	1	0	1	1	1
D_6	1	1	0	0	0
	1	1	0	1	1
D_7	1	1	1	0	1
	1	1	1	1	0

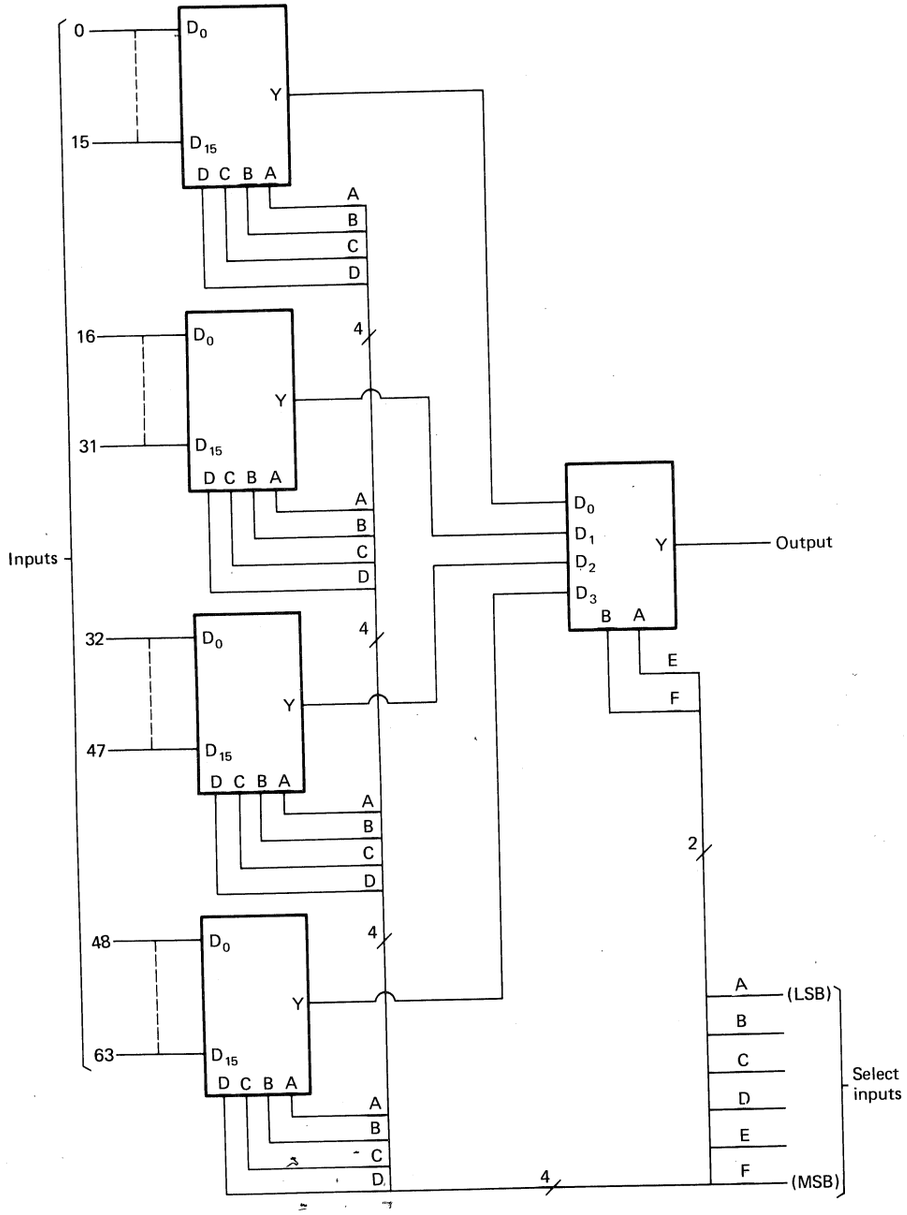


Figure 3-16 Cascading multiplexers.

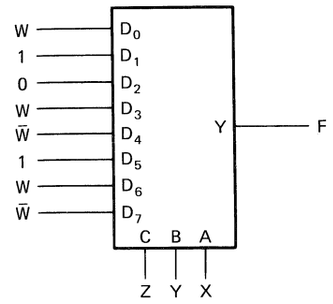


Figure 3-17 Boolean function implemented with multiplexer.

DECODERS

Another common MSI circuit is the *decoder* (also called a *demultiplexer*). A decoder is an arrangement of gates that allows a single enable or data input to be sent to several destinations. A decoder circuit along with its MSI symbol is shown in Fig. 3-18. This is a two-line to four-line decoder (also called a one out of four). Two select inputs A and B determine which of four outputs Y_0 through Y_3 will be connected to the enable or data input.

Enable	B	A	Y_3	Y_2	Y_1	Y_0
1	X	X	1	1	1	1
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	1

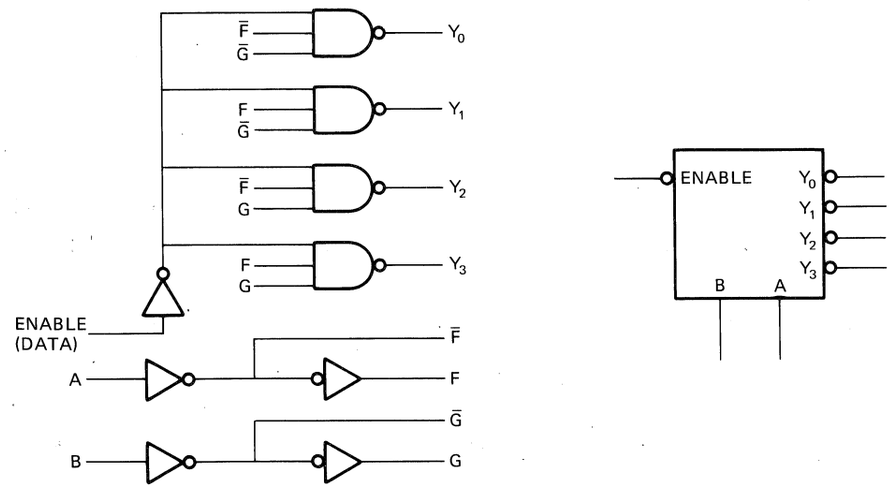


Figure 3-18 Decoder.

Decoders are available in several sizes, packaged one or more in each integrated circuit. Common sizes are:

Single. 4 line to 16 line
Single. 4 line to 10 line
Single. 3 line to 8 line
Dual. 2 line to 4 line

One common use of the decoder does not use the data input. With the enable or data input of Fig. 3-18 permanently connected to a 0, all but one of the decoder's outputs will be 1. A single output will be selected by inputs A and B. Many times, outputs from algorithmic state machines are mutually exclusive. That is, only one of several outputs can occur at one time. For example, only one color traffic light in each direction may be lighted at one time. Since two digital signals can specify four conditions, we can use a decoder, as shown in Fig. 3-19, to generate three traffic light control signals from only two ROM outputs.

HNSB	*HNSA*	*ILNSG*	*ILNSY*	*ILNSR*
0	0	1	1	1
0	1	0	1	1
1	0	1	0	1
1	1	1	1	0

The ASM can turn on only one light at a time. This restriction is entirely acceptable for the control of most intersections. A simple traffic light controller using this technique is shown in Fig. 3-20. Only four ROM outputs are required, rather than the six that would have been required without the decoders. The ROM was a 16 X 10 and has been reduced to a 16 X 8 through the use of decoders. This is not as dramatic an improvement as was obtained with multiplexers since saving one output line doesn't save as many bits in the ROM as saving one address line. ROMs with any number of address lines desired are easily obtainable. However, ROMs are usually manufactured with outputs in multiples of 4 or even 8. For the original traffic controller, a 16 X 12 ROM would have been used, even though only a 16 X 10 was required. Thus, requiring two-fewer outputs would probably save 4 ROM output bits in the implementation. A decoder is a necessity when *many* mutually exclusive outputs must be controlled. In fact, when many outputs are required, it may be useful to try to set up the ASM chart

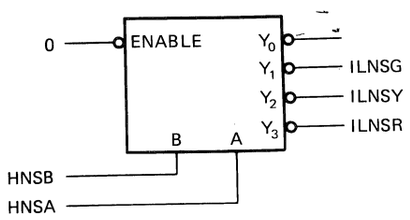


Figure 3-19 MSI decoder used to generate traffic-light control signals.

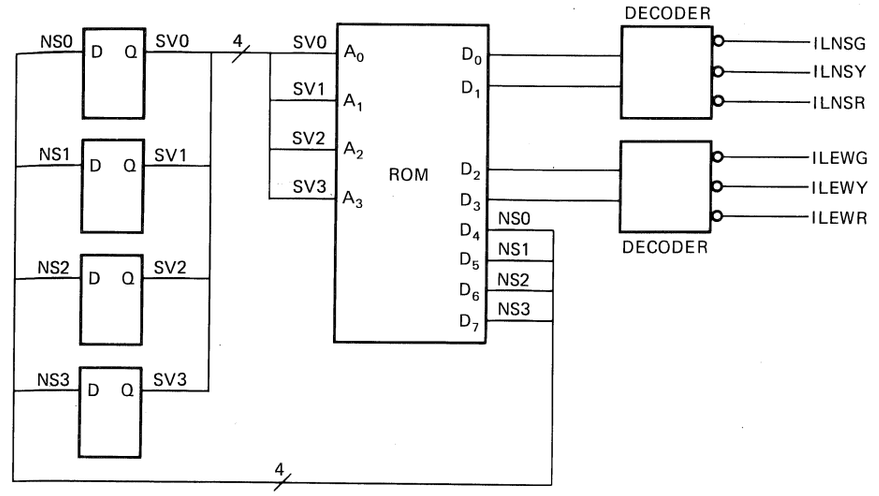


Figure 3-20 Traffic-light controller with decoders.

so that outputs can be made mutually exclusive even if they are not intrinsically mutually exclusive. If this can be done, a decoder could be used to save a large number of ROM output bits. Just as in the case of the input multiplexer, additional states required to sequentially activate outputs could negate the advantages of decoding by requiring more state variables (and a much bigger ROM) or by slowing down the operation of the ASM.

REGISTERS

A register is a group of flip-flops with a common clock input, just like the flip-flops we've been using to store state variables. Groups of such flip-flops in multiples of 4 or 8 are manufactured as MSI circuits, as shown in Fig. 3-21. One important variation is shown in Fig. 3-22. The register shown here has an enable input. When the enable input is a 0, the contents of the register do not change, even though clock pulses continue. The register does not change because the AND gates marked B are enabled so that the signals at the outputs of the flip-flops are routed back to their inputs. Only if the enable input is a 1 will AND gates A be enabled and new data be stored in the register.

A register with an enable input is a sequential circuit that remains in its current state when its enable input is a 0 and changes to the state specified by its data inputs when its enable input is 1. The enable input of such a register may be connected to the output of an ASM controller. This ASM will control the flow of data into that register.

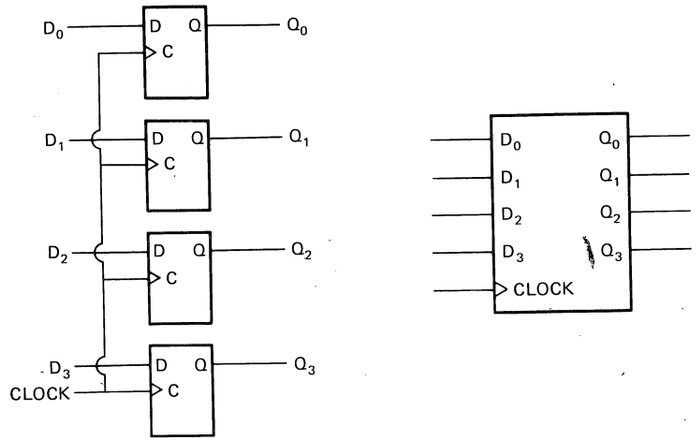


Figure 3-21 4-bit register and MSI circuit symbol.

Simultaneous Sampling with a Register

Multiplexing inputs to an ASM may present another problem not yet mentioned. Often inputs *must* be tested simultaneously. If tested in sequence, the first inputs tested might change their state before the last inputs are tested. An ASM with multiplexed inputs, as shown in Fig. 2-23*a*, performs a test on *YA* in state *X* and a test on

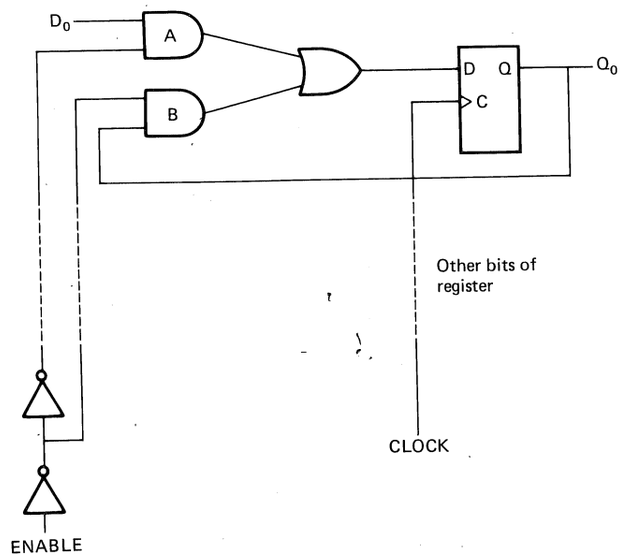
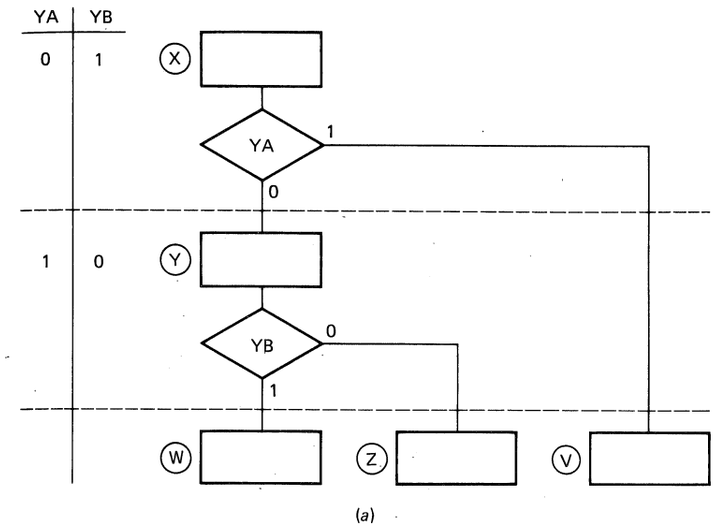
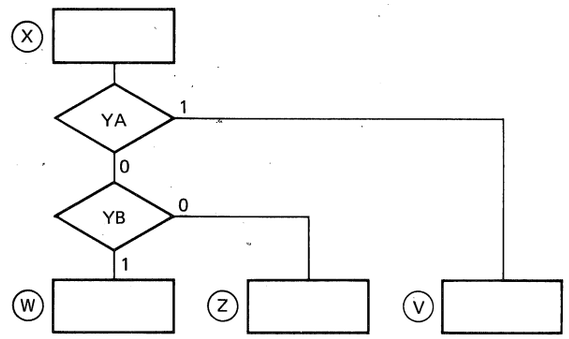


Figure 3-22 Register with enable inputs.



(*a*)



(*b*)

Figure 3-23 (*a*) Inputs changing during sequential testing. (*b*) Simultaneous testing.

YB in state *Y*. If the inputs change between these two states as shown, the next state will be *Z*. Input *YA* is 0 in state *X*, while input *YB* is 0 in state *Y*. State *Z* will be the next state, even though both inputs were never zero simultaneously! A simultaneous test, as might be performed without multiplexed inputs, is shown in Fig. 3-23*b*. In this case, the next state will be *W* if the input during state *X* is 01. The next state will be *V* if the input is 10. In *neither* case will the next state be *Z*, as it was when the inputs were sequentially tested.

If it is necessary to test more than one input whose possible changes between sequential tests would cause an ASM to behave incorrectly, those inputs could all be stored in a register at *one* time to be tested sequentially later. In Fig. 3-24, a 4-bit register is used to store a "snapshot" of four inputs. Whenever the ROM output en-

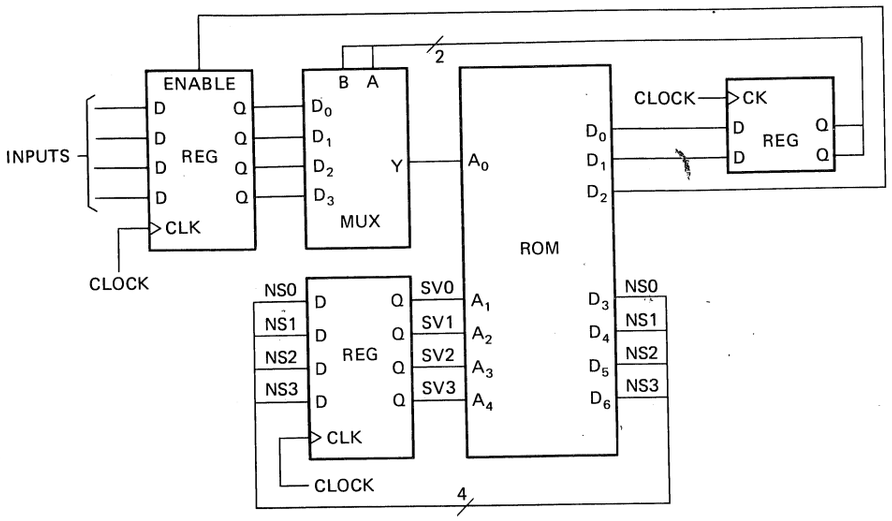


Figure 3-24 ASM using one MSI register to simultaneously save inputs and another MSI register for the state variables.

ables this register, the four input conditions are stored in the register at the next clock pulse *simultaneously*! Since these four inputs were stored at the same time, the ASM can now test them sequentially and be certain that these conditions did actually occur simultaneously at the time the register was enabled.

Expanding Outputs with a Register

ASM outputs may be stored in registers, as shown in Fig. 3-25. Using registers to store these outputs may be desirable for two reasons. First, many outputs may be obtained with a small ROM. These outputs do *not* have to be mutually exclusive, as was necessary when a decoder was used. Second, the ASM can store an output based on some decision and then completely forget about that output until it must be changed again. Without output storage registers, the ASM might have to use many extra states to remember which outputs to keep asserted long after the decision to assert that output had been reached.

In Fig. 3-25, data for register *W* are output from the ROM to the *D* inputs of all four registers. Simultaneously, the select code for register *W* is output to the decoder, along with the enable signal. The decoded enable signal is sent to register *W* so that, on the next clock pulse, the ROM data are loaded into register *W*. The other three registers *X*, *Y*, and *Z* are loaded sequentially with their data in the same manner. All four registers will retain the data stored in them as long as the ASM does not output a decoder enable signal. In this case, only 7 output bits were needed for 16 outputs. Unlike a simple decoder, these outputs need not be mutually exclusive, but can be any combination of 0s and 1s. However, all outputs cannot be made to *change* simul-

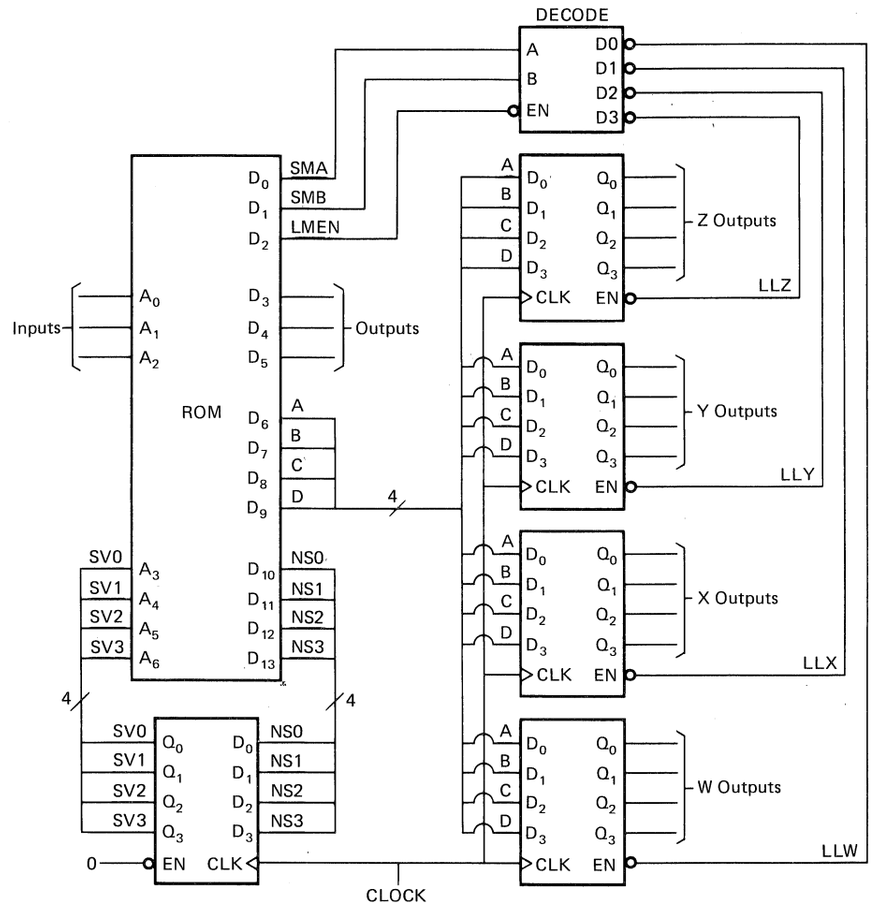


Figure 3-25 Using registers to expand ROM outputs.

taneously. The four outputs of register *W* will change first, followed by the outputs of *X*, then *Y*, and then *Z*.

COUNTERS

An ASM that repeatedly outputs the same output sequence is called a counter. The most common type of counter outputs the binary sequence. MSI counters are usually implemented as 4-bit counters. As we'll see later, several 4-bit MSI counters may be connected together to build larger counters. The simple 4-bit counter shown in Fig. 3-26 contains four flip-flops and associated gates, which together implement an ASM

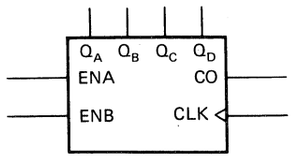


Figure 3-26 MSI counter.

that counts in the sequence shown. The counter remains in the same state as long as either enable input is 0. If both enables are 1, the counter counts at each clock pulse. The current state is output to four wires, labeled Q_A, Q_B, Q_C, and Q_D. A fifth output labeled *CO* is a 1 only when the current state is 1111 *and* both enables are a 1.

		Current state				Next state				
ENA	*ENB*	Q_D	Q_C	Q_B	Q_A	Q_D	Q_C	Q_B	Q_A	*CO*
0	0	q_D	q_C	q_B	q_A	q_D	q_C	q_B	q_A	0
0	1	q_D	q_C	q_B	q_A	q_D	q_C	q_B	q_A	0
1	0	q_D	q_C	q_B	q_A	q_D	q_C	q_B	q_A	0
1	1	0	0	0	0	0	0	0	1	0
1	1	0	0	0	1	0	0	1	0	0
1	1	0	0	1	0	0	0	1	1	0
1	1	0	0	1	1	0	1	0	0	0
1	1	0	1	0	0	0	1	0	1	0
1	1	0	1	0	1	0	1	1	0	0
1	1	0	1	1	0	0	1	1	1	0
1	1	0	1	1	1	1	0	0	0	0
1	1	1	0	0	1	1	0	1	0	0
1	1	1	0	1	0	1	0	1	1	0
1	1	1	0	1	1	1	1	0	0	0
1	1	1	1	0	0	1	1	0	1	0
1	1	1	1	0	1	1	1	1	0	0
1	1	1	1	1	0	1	1	1	1	0
1	1	1	1	1	1	0	0	0	0	1

The *CO* output can be used to connect two 4-bit counters together to make an 8-bit counter. In Fig. 3-27, MSI counter *X* is the least significant (rightmost) 4 bits of an

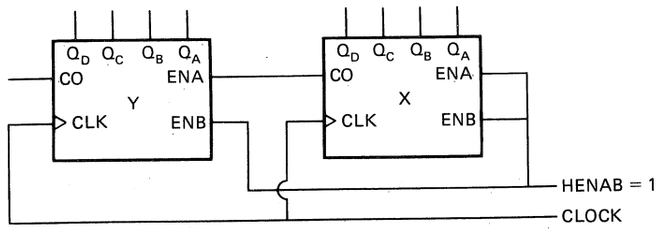


Figure 3-27 8-bit counter.

Table 3-3 Counting sequence for 8-bit counter

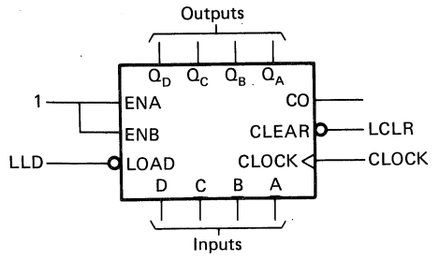
Counter *Y*					Counter *X*				
CO	Q_D	Q_C	Q_B	Q_A	*CO*	Q_D	Q_C	Q_B	Q_A
0	1	1	0	1	0	0	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0	1	1	0	1	1	1	1	1	1
0	1	1	1	0	0	0	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0	1	1	1	0	1	1	1	1	1
0	1	1	1	1	0	0	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

8-bit counter, while MSI counter *Y* is the most significant 4 bits. The *CO* output of *X* is connected to one of the enable inputs of *Y*. The letters *CO* stand for *carry output* and indicate that counter *X* has reached the largest value it can represent (i.e., 1111). The next clock pulse will set counter *X* back to 0000, but will also increment counter *Y*. The counting sequence is shown in Table 3-3. Counter *Y* will count because its *ENB* is connected to the *CO* output of counter *X*. Every time counter *X* reaches its maximum value, its *CO* output will signal *Y* that *Y* should add 1 to its 4 most significant bits on the next clock pulse. MSI counters have two enable inputs so that one may be used to connect multiple MSI counters together, while the other is used to enable and disable the entire counter (*HENA* in Fig. 3-27). You'll also notice that *CO* from counter *Y* occurs in exactly the right place to signal a third counter to count so that a 12-bit counter could be built. Counters have additional inputs to make them more useful. The counter in Fig. 3-28 has a clear and a load input. Both of these inputs are high when the counter is counting. If the clear input is low, the counter's next state will be 0000. The next clock pulse to occur after the clear input becomes a 0 will clear the counter's outputs to 0000. The counter will resume counting only on the first clock pulse after the clear input becomes a 1.

If the load input becomes a 0, the next clock pulse will load the four counter flip-flops with the data on inputs *D*, *C*, *B*, and *A*. In other words, the counter will act as a register as long as the load input is a 0. At each clock pulse, the datum present at input *D* will be stored at output Q_D, *C* at Q_C, *B* at Q_B, and *A* at Q_A. The counter resumes counting from the state last specified by *D*, *C*, *B*, and *A* when the load input returns to a 1.

Time Delays with a Counter

One use of a counter is to implement delays longer than one clock cycle in an ASM. You'll remember that the simple traffic light controller required 10 states. Six of those



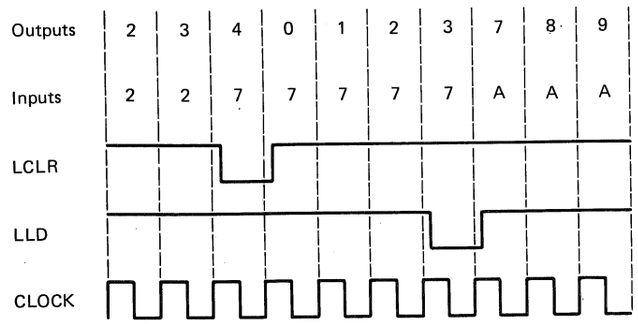


Figure 3-28 MSI counter with load and clear inputs.

states were simply used to delay outputs a longer time than one clock period. The traffic light controller is redrawn in Fig. 3-29, using a counter to implement this time delay. The delay counter is loaded by output *LCLD* (LOW COUNTER LOAD) on the same clock pulse that causes the ASM to enter the red/green states. These are the states marked 1 and 3 in the ASM chart of Fig. 3-30. The carry output of the delay counter is an input to the ASM. While the ASM is in the red/green states, the counter is counting. When the counter finally reaches 1111, the *CO* output signals the ASM to proceed to the next state. You'll notice that the ASM now requires only 4 states instead of 10. We have designed a circuit that does not require the ASM to count with its state flip-flops. Instead, the ASM uses an auxiliary MSI counter. Because counting is such a common requirement, MSI counters are mass-produced. It is economically advantageous to use an MSI counter rather than build the counting function into the ASM by adding state variables and ROM bits. In addition, less design effort will be required using the MSI counter.

Figure 3-31 shows both the timing diagram and state analyzer representation of the traffic controller. The counter's state is shown as a decimal number. Asserting output *LCLD* during state 0 causes the counter to be set to 12 during state 1. The counter counts to 15, at which time the *CO* output becomes 1. On the next clock pulse the ASM tests *CO* and proceeds to state 2. By setting the counter to 12 we were able to have the ASM remain in state 1 for four clock periods. The binary number 1100, which represents 12, is also the two's-complement representation of −4. Thus, the counter can also be thought of as counting down from −4, as shown in Table 3-4.

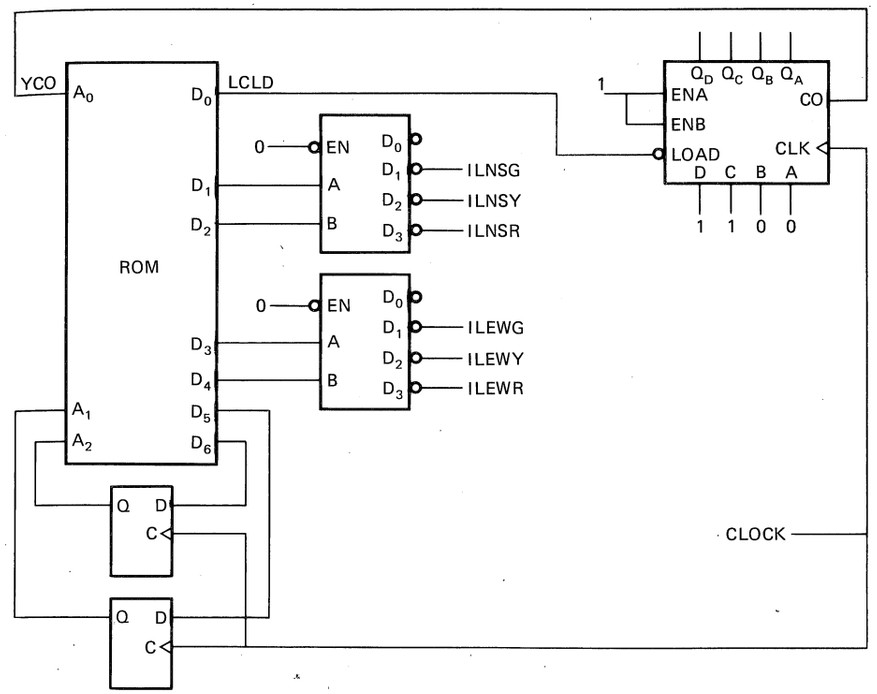


Figure 3-29 Traffic-light controller using counter for delay.

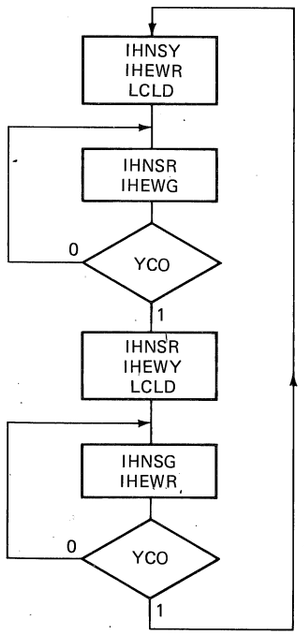
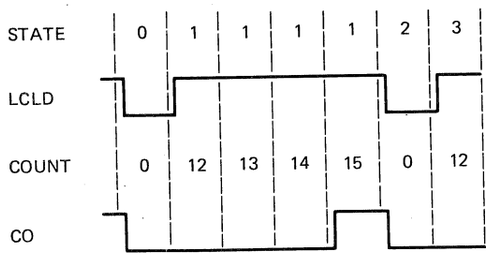


Figure 3-30 ASM chart for traffic light using delay counter.



STATE	LCLD	COUNT	CO
0	0	0	0
1	1	12	0
1	1	13	0
1	1	14	0
1	1	15	1
2	0	0	0
3	1	12	0
3	1	13	0
3	1	14	0
3	1	15	1
0	0	0	0

Figure 3-31 Traffic-light controller sequencing.

The delay is easily changed by changing the binary number connected to the counter data inputs. The binary number 0000 will cause a delay of 16 clock periods, while 1111 will cause the ASM to remain in states 1 and 3 for only one clock period. These inputs might be connected to switches to allow a traffic engineer to set the time. Rather than using multiple counters if we need more than one delay period, the counter's data inputs may be connected to the ASM's ROM outputs, as shown in Fig. 3-32. The delay counter may be loaded with different initial values from the ROM, for different delays.

State Sequencing with a Counter

Many of the ASM charts we've drawn had several states following each other sequentially. Few branches occurred to other states. The simple traffic light controller had no decisions at all. By assigning states in numerical order, we should be able to use a counter for the state register since the next state will always be one greater than the current state. Figure 3-33 shows a circuit and ASM chart for a traffic light controller using this technique. The *CO* output of the state counter is connected to the load input of the counter through an inverter. The next state after state 15 will be loaded

Table 3-4 Interpreting the count as a two's-complement (negative) number

State	LCLD	Count	*CO*
0	0	0	0
1	1	−4	0
1	1	−3	0
1	1	−2	0
1	1	−1	1
2	0	0	0
3	1	−4	0
3	1	−3	0
3	1	−2	0
3	1	−1	1
0	0	0	0

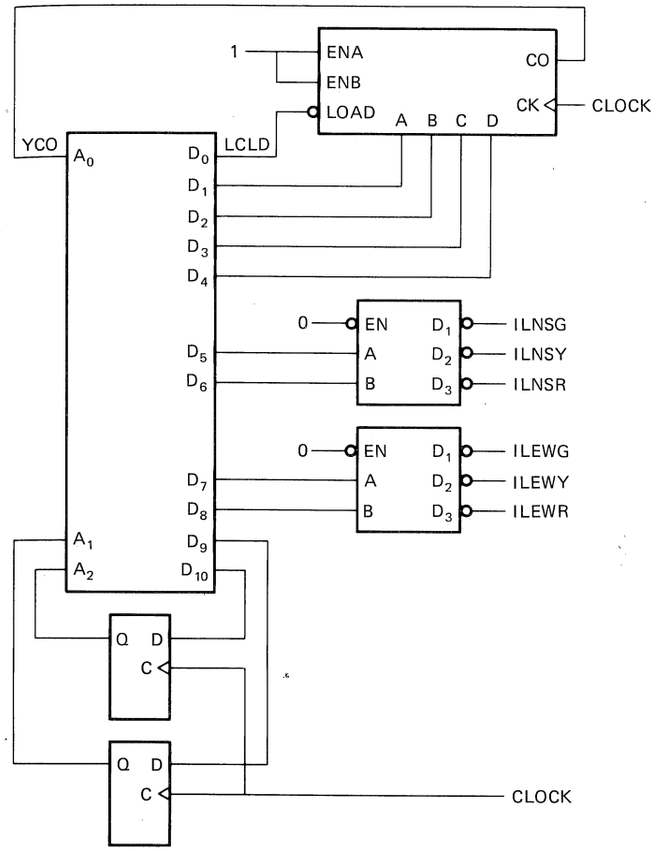
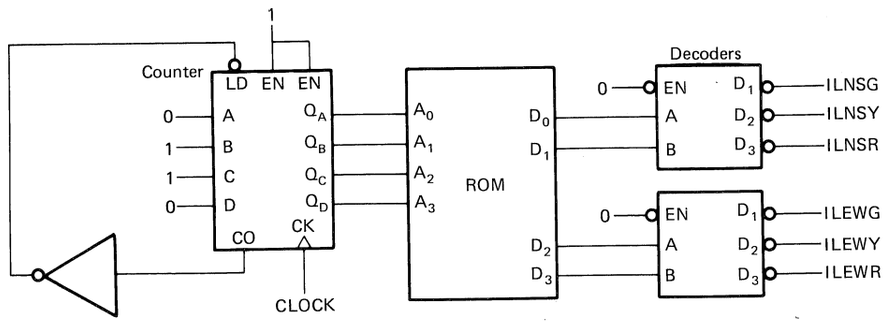


Figure 3-32 Using ROM outputs to set delay.

from the counter's data inputs. In this example, we specified the next state as 6. The counter will cycle through the 10 states needed to implement the traffic light controller. Limited branching could be implemented by connecting one or more counter data inputs to external input conditions. The flexibility of this branching arrangement is very limited.

In Fig. 3-34, the counter's load and data inputs are connected to ROM outputs. Any next state may now be specified for branching from any current state. The ASM chart in Fig. 3-35 shows how this may be done. States 0 through 9 follow sequentially, and the state variable counter simply counts through these states. In state 9, a branch must be made on the condition of input *YI*. If *YI* is 0, the counter will be allowed to increment to state 10. However, if *YI* is a 1, the counter must be set to state 14. This is done by asserting outputs $SV = 1110$ to the counter's data input and asserting *LSVLD* (LOW STATE VARIABLE LOAD). These conditional outputs will cause the counter to be loaded with 14 on the next clock pulse. Thus, we've caused the branch



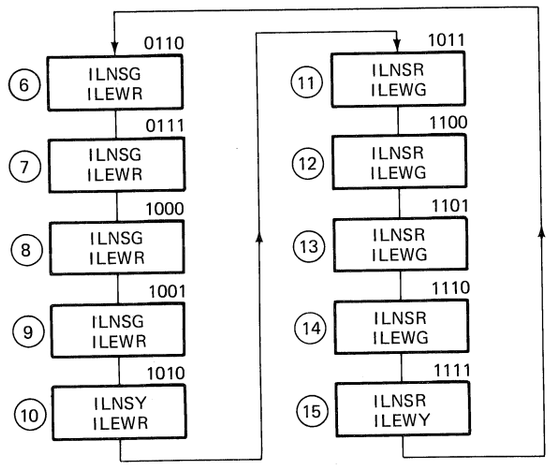


Figure 3-33 Simple traffic light using counter for state sequencing.

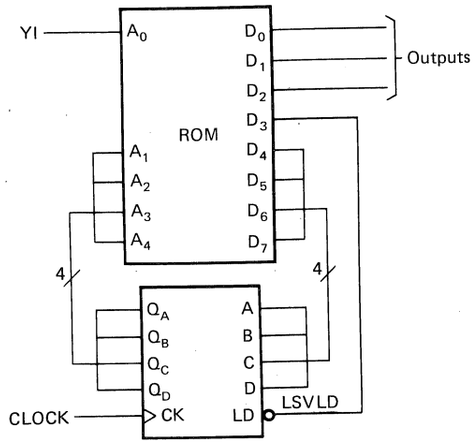


Figure 3-34 Loading an arbitrary next state when using a counter for the state register.

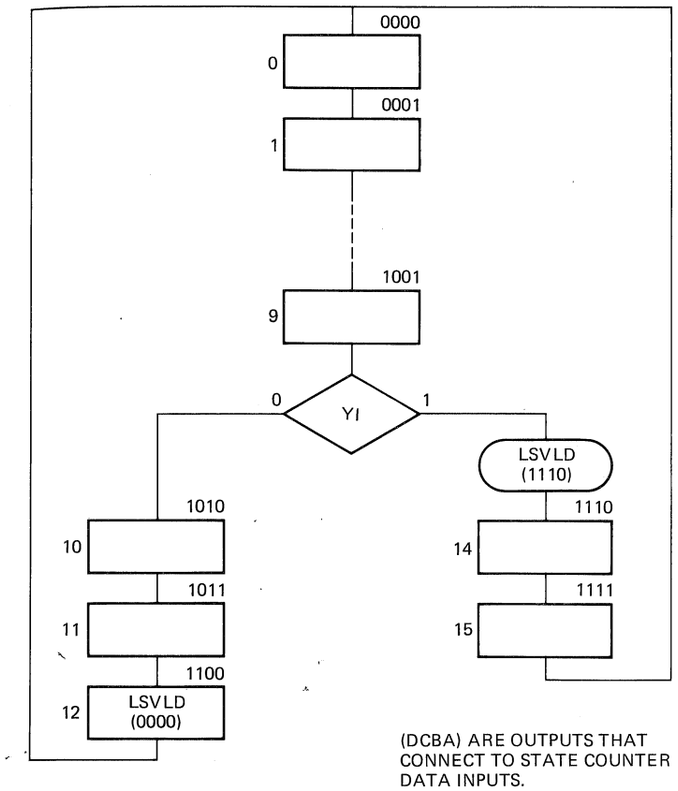


Figure 3-35 ASM chart that loads arbitrary next state into state variable counter.

to occur from state 9 to state 14. The counter's sequence may also be changed unconditionally. For example, in state 12, *LSVLD* and *SV* = 0000 cause the next state to always be state 0.

You should have noticed by now that implementing this controller without a next-state counter would've required four ROM outputs to specify the next state. With a next-state counter it requires five outputs. Hardly a bargain! Loading a state counter from ROM outputs is only advantageous when those ROM outputs can be used for other purposes as well. In Fig. 3-36, four data lines are also used to load three output registers, as well as state branches. Thus, this ASM has 12 outputs and 16 states. This would ordinarily require 16 ROM outputs: 12 outputs and 4 next-state variables. By using a state counter and output latches, all sharing the same ROM data outputs, only 8 outputs are required from the ROM. Finally, just as in past examples of multiplexing and demultiplexing, branching with next-state counters is liable to require extra states. If too many extra states are required, more state variables may be needed, which would partially negate the advantages of this scheme.

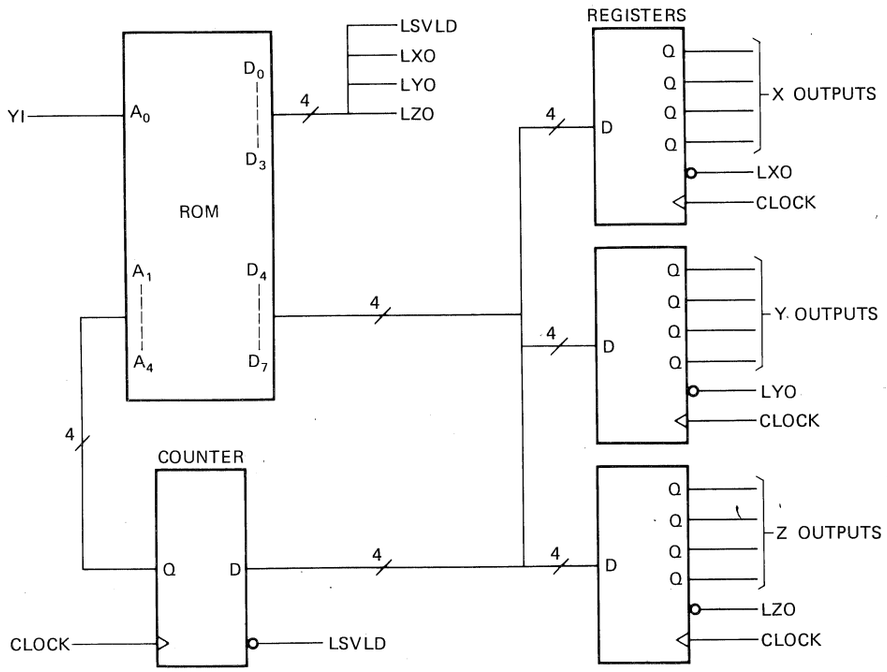


Figure 3-36 Sharing output and next state ROM outputs.

SHIFT REGISTERS

Another common MSI circuit is a shift register. A simplified shift register is shown in Fig. 3-37. It is composed of four flip-flops. At each clock pulse, the datum on Q_C is transferred to Q_D, the datum on Q_B to Q_C, the datum on Q_A to Q_B, and the datum on input *SI* to Q_A. In this way, a new datum is stored at Q_A, while all previous data are shifted toward the most significant bit Q_D. The datum that was stored at Q_D is lost (unless, of course, something else is connected to Q_D).

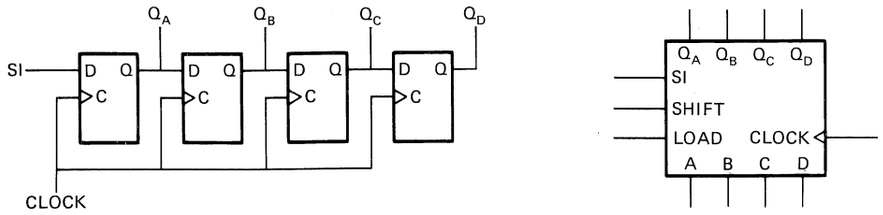


Figure 3-37 Shift register.

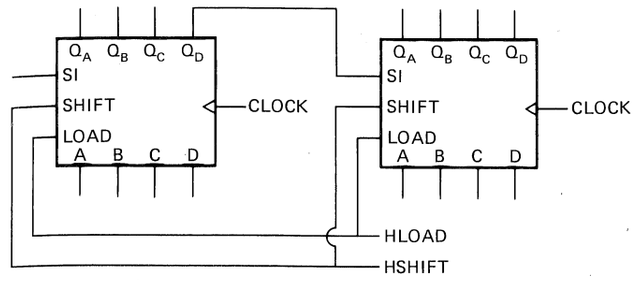


Figure 3-38 Cascaded MSI shift registers.

An actual MSI shift register has several additional useful inputs, as shown in Fig. 3-37. The *SHIFT* input must be high for the shift register to move data; otherwise data will remain stored in their original flip-flops, even though clock pulses continue. The *LOAD* input allows data present on the A, B, C, and D inputs to be stored at Q_A, Q_B, Q_C, and Q_D, respectively, at the next clock transition.

MSI shift registers may be cascaded when longer shift registers are needed. In Fig. 3-38, two 4-bit MSI shift registers are connected together to form an 8-bit shift register. The datum that would've been lost from Q_D of the least significant register is shifted to Q_A of the most significant register.

Serial Data Communication

Very often, two ASMs must communicate with each other. In Fig. 3-39, ASM1 has eight outputs that must be sent to ASM2. In addition, ASM1 sends a signal, *HREADY*, to ASM2 to indicate that the eight outputs have been changed and ASM2 may now test them. Besides the clock, nine wires are required to interconnect the two ASMs. Interconnections are often expensive, especially if they pass through mechanical connectors or travel over long distances.

These same two ASMs may communicate as described above using only *two* interconnections, as shown in Fig. 3-40. Two shift registers are used to send and receive 8 bits of data over a single wire. ASM1 outputs 8 bits of data to *SR1* and asserts *HLOAD* to load these data into the shift register's eight flip-flops. Next, ASM1 outputs *HSHFT*, which causes a bit of data from *SR1* to be shifted into *SR2*. After eight clock pulses, the data from *SR1* have been transferred to *SR2* and ASM1 stops outputting *HSHFT*.

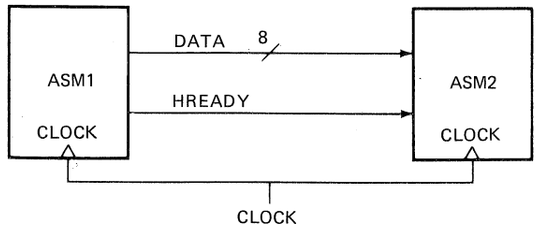


Figure 3-39 Parallel data transmission.

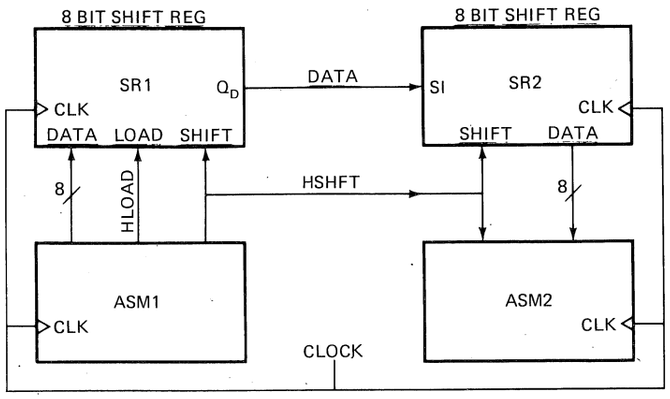


Figure 3-40 Serial data transmission.

Since ASM2 can test *HSHFT*, it now knows that the data shifting operation has been completed and it can test the data available to it from *SR2*.

Although we've saved interconnections, we have paid a penalty in the speed of the data transfer. In Fig. 3-39, data were transferred in 1 clock period, while, in Fig. 3-40, 10 clock periods were required (1 to load *SR1*, 8 to shift, and 1 to transfer from *SR2* to ASM2). This trade-off of speed and number of interconnections is common to all *timeshared* or *serial* systems, like the serial data transmission system just described. Serial data transmission systems are common and very sophisticated. Often only one wire is required for data transmission; the "clock" and "shift" signals are cleverly added to the data.

A TRAFFIC LIGHT CONTROLLER USING MSI CIRCUITS

Let's design a traffic light controller using MSI circuits. This intersection normally allows only north-south traffic through. Only when east and westbound traffic sensors are activated will the traffic lights change to halt north-south traffic. In addition, an emergency input sets all lights to red. When the emergency input is released, the lights should return to the north-south green condition after a delay. The delay is to allow a following emergency vehicle to activate the emergency input, so that the lights will remain red continuously. The times associated with each traffic light condition should be easily changed.

A circuit diagram of such a controller is shown in Fig. 3-41. The circuit is a ROM/flip-flop ASM. The next-state register is a counter. Branching is accomplished by loading the counter from four ROM data outputs. These data outputs are shared with both an output register and another counter, used as a delay timer. Decoders are used to reduce the number of output bits required. A two-line to one-line multiplexer connects either the timer output or east-west sensor as a branch input. The emergency

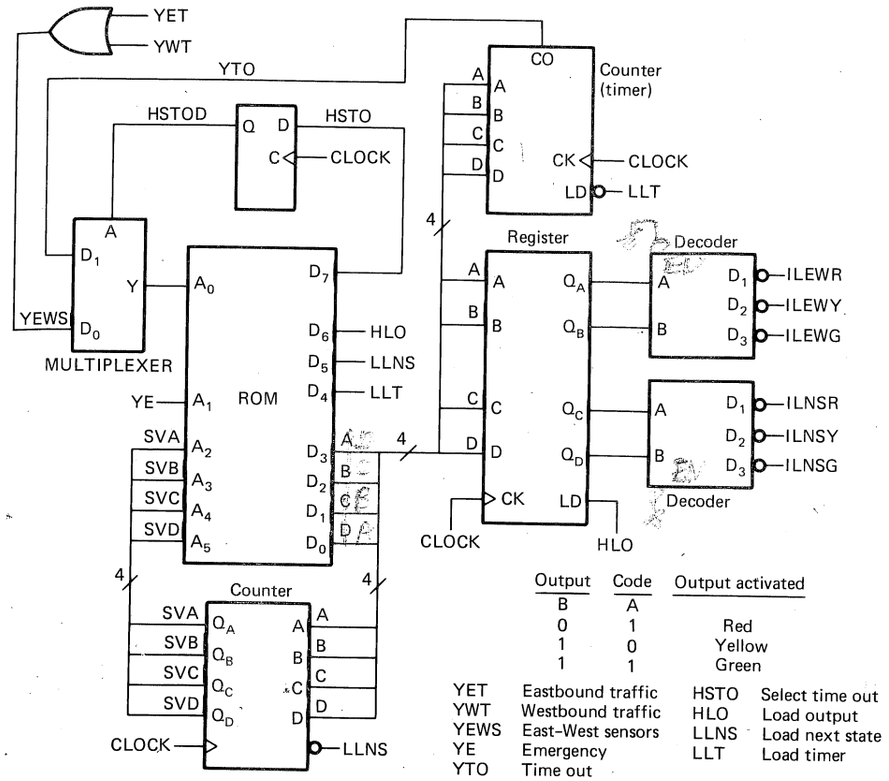
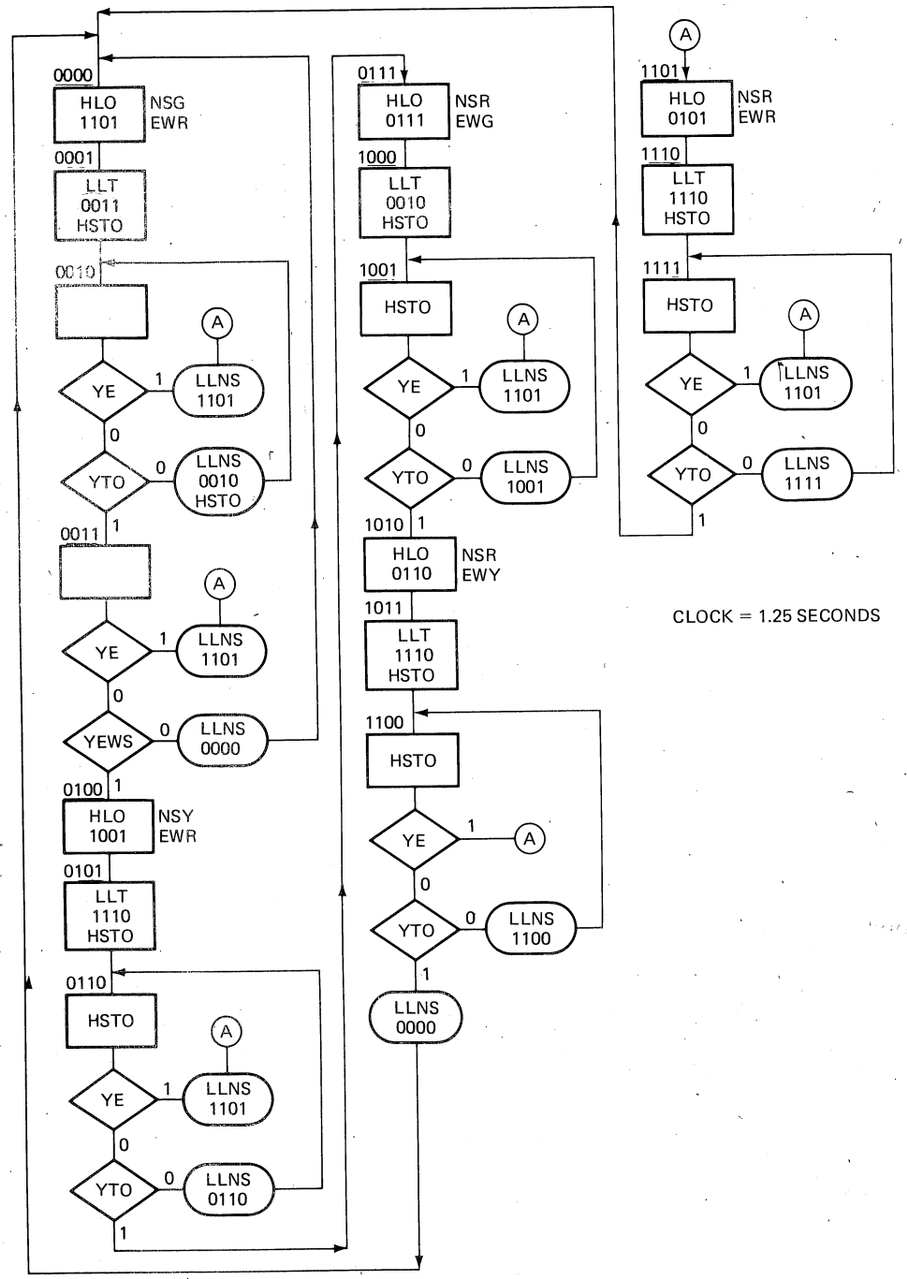


Figure 3-41 Traffic-light controller circuit.

input is not multiplexed. Finally, a simple OR gate combines the east and westbound sensors.

The ASM chart for this controller is shown in Fig. 3-42. Since four data outputs from the ROM are shared by the next-state counter, the timer counter, and the output register, extra states must be added to perform some functions sequentially. For example, state 0000 changes the output register, state 0001 sets the timer, and state 0010 tests for branches. These three functions must be performed sequentially because each requires use of the same ROM data outputs they all share. Since *YTO* and *YEWS* are multiplexed, an additional state, 0011, must be added to test *YEWS*. This additional state delays changing the traffic lights by one clock cycle. Thus, the timer constant in state 0001 provides one less delay cycle than the constant in state 1000. The timer constants specified will cause 20-s green lights and 5-s yellow lights, using the 1.25-s clock specified. Don't forget that *all* states between output register changes must be included when calculating delays. The ROM contents for this circuit are specified in Table 3-5. Data outputs are listed as don't cares when these outputs aren't used. However, some value must be specified when the ROMs are programmed.



CLOCK = 1.25 SECONDS

Figure 3-42 ASM chart for traffic-light controller.

Table 3-5 Traffic light control ROM contents

Current state				*YE*	*YTO*	*HSTO*	*HLO*	*LLNS*	*LLT*	*D*	*C*	*B*	*A*
				YEWS									
A_5	A_4	A_3	A_2	A_1	A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	*X*	*X*	0	1	1	1	1	1	0	1
0	0	0	1	*X*	*X*	1	0	1	0	0	0	1	1
0	0	1	0	1	*X*	0	0	0	1	1	1	0	1
0	0	1	0	0	0	1	0	0	1	0	0	1	0
0	0	1	0	0	1	0	0	1	1	*X*	*X*	*X*	*X*
0	0	1	1	1	*X*	0	0	0	1	1	1	0	1
0	0	1	1	0	0	0	0	0	1	0	0	0	0
0	0	1	1	0	1	0	0	1	1	*X*	*X*	*X*	*X*
0	1	0	0	*X*	*X*	0	1	1	1	1	0	0	1
0	1	0	1	*X*	*X*	1	0	1	0	1	1	1	0
0	1	1	0	1	*X*	1	0	0	1	1	1	0	1
0	1	1	0	0	0	1	0	0	1	0	1	1	0
0	1	1	0	0	1	1	0	1	1	*X*	*X*	*X*	*X*
0	1	1	1	*X*	*X*	0	1	1	1	0	1	1	1
1	0	0	0	*X*	*X*	1	0	1	0	0	0	1	0
1	0	0	1	1	*X*	1	0	0	1	1	1	0	1
1	0	0	1	0	0	1	0	0	1	1	0	0	1
1	0	0	1	0	1	1	0	1	1	*X*	*X*	*X*	*X*
1	0	1	0	*X*	*X*	0	1	1	1	0	1	1	0
1	0	1	1	*X*	*X*	1	0	1	0	1	1	1	0
1	1	0	0	1	*X*	1	0	1	1	*X*	*X*	*X*	*X*
1	1	0	0	0	0	1	0	0	1	1	1	0	0
1	1	0	0	0	1	1	0	0	1	0	0	0	0
1	1	0	1	*X*	*X*	0	1	1	1	0	1	0	1
1	1	1	0	*X*	*X*	1	0	1	0	1	1	1	0
1	1	1	1	1	*X*	1	0	0	1	1	1	0	1
1	1	1	1	0	0	1	0	0	1	1	1	1	1
1	1	1	1	0	1	1	0	1	1	*X*	*X*	*X*	*X*

EXCLUSIVE OR GATES

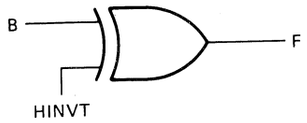
The exclusive OR function (XOR) is a simple logical connective, like the AND and OR connectives. It is represented by the OR sign encircled:

$$F = A \oplus B$$

The function F is a 1 when either A or B is a 1 but is not a 1 when both A and B are 1:

$$F = A \oplus B$$

A	*B*	*F*
0	0	0
0	1	1
1	0	1
1	1	0



HINVT	F
0	B
1	$\bar{B}$

Figure 3-43 XOR gate.

Although generally useful, the XOR function has two especially common applications. The symbol for the XOR gate is shown in Fig. 3-43. The XOR *selectively* inverts the bit B in this diagram. One input, *HINVT*, is thought of as a control signal which can invert bit B if the control signal is a 1. This property of selective inversion is often used to invert an entire binary word, as shown in Fig. 3-44.

The output of an XOR gate is 0 whenever both inputs are identical. Thus, XOR gates can be used to compare single bits or groups of bits to determine equality. A *digital comparator* implemented with XOR gates is shown in Fig. 3-45. The output *HEQL* is 1 if all the bits of word F are equal to the corresponding bits of word G. Digital comparators are also available as single integrated circuits and usually include $F > G$ and $F < G$ outputs, besides an $F = G$ output. In the case of the inequalities, F and G are usually interpreted as positive binary numbers.

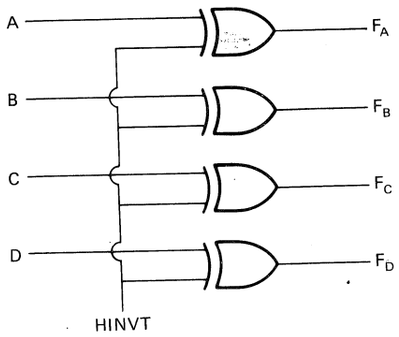


Figure 3-44 Selective inversion.

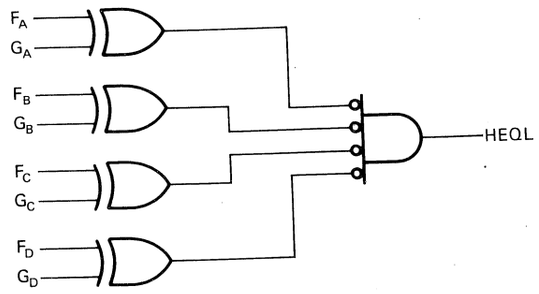


Figure 3-45 Digital comparator.

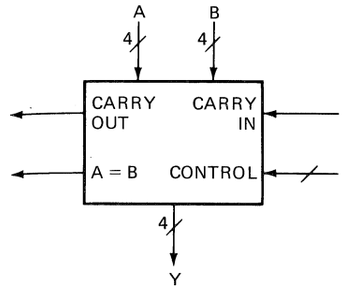


Figure 3-46 Arithmetic logic unit.

ARITHMETIC LOGIC UNITS

Another common MSI circuit is the arithmetic logic unit (ALU). An ALU generates logic functions of many inputs. Some sophisticated ALUs may also contain registers since data registers are almost always used with ALUs. We'll consider only those ALUs without registers. The diagram of Fig. 3-46 is representative of most ALUs. An output word Y is generated as some function of input words A and B. The function to be performed is specified by the control inputs. This function may be addition, subtraction, negation, shifting, bit-by-bit ANDing or ORing, and many others. The connections labeled *CARRY IN* and *CARRY OUT* allow multiple ALUs to be interconnected so that any word length for A, B, and Y may be accommodated. An $A = B$ output allows testing for equality. The ALU is controlled by an ASM and usually must have one or more external data storage registers, as shown in Fig. 3-47. Most digital systems, including computers, use an ALU and registers controlled by an ASM to perform arithmetic computations. Binary arithmetic will be discussed in detail in a later section of this book.

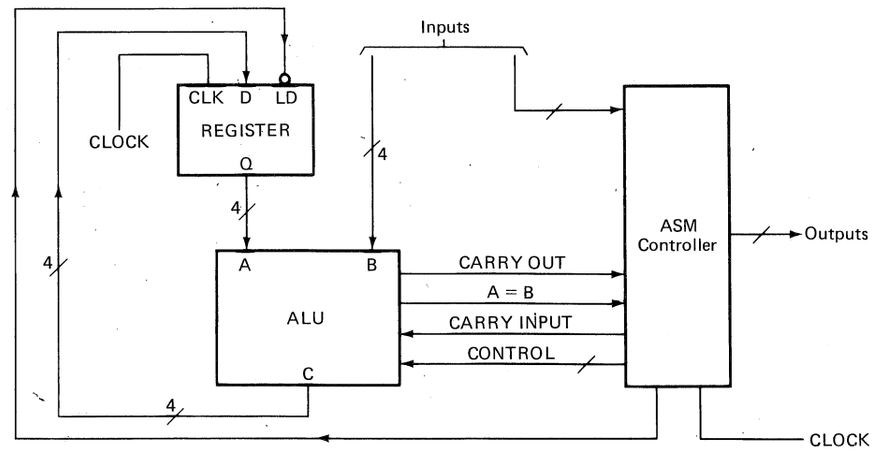


Figure 3-47 Processing with an ALU.

The ALU is a very general processing element. For simpler problems, a simpler processing element could be substituted for the ALU of Fig. 3-47. For example, if only addition were required, an MSI *adder* circuit could be substituted for the ALU of Fig. 3-47. The arrangement of circuit elements or *architecture* shown in Fig. 3-47 is very common. Although subject to many variations, it is characterized by data, stored in registers, being changed by a processing element under the control of an ASM.

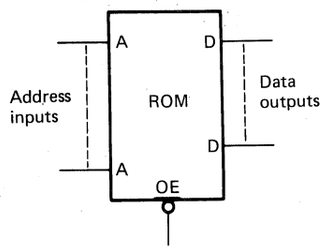
LSI CIRCUITS

Large-scale integrated circuits contain very large numbers of circuits, thousands of flip-flops and gates. We have already used one LSI circuit, the read-only memory. In the last half of this book, we will use LSI circuits, almost exclusively, to design microcomputer systems. Let us examine some common LSI circuits.

Read-Only Memory

Although we have been using ROMs in our designs, one additional input often is included in a ROM that we haven't discussed. This input is called output enable, shown in Fig. 3-48, and is used to enable three-state outputs incorporated into the ROM. ROM outputs may need to be disabled when they are connected to a common data bus with other memories.

If one ROM doesn't have a sufficient number of outputs, two can be easily combined, as shown in Fig. 3-49. In this circuit, two 1KB ROMs have their address inputs paralleled, forming a 1K X 16 read-only memory. If we need more *address* inputs, the problem of combining ROMs is not as simple. The output enable input may be used to expand the number of words, as shown in Fig. 3-50. The most significant address bit is used to enable the three-state outputs of either one ROM or the other. The three-state outputs of one ROM are connected or bussed to the corresponding output bits of the other ROM. The MSB of the address selects one of the two ROMs to be connected to the output signals. Each ROM is again 1KB. Combined this way, they form a 2KB read-only memory. One ROM responds to addresses 000H to 3FFH, while the other ROM responds to addresses 400H to 7FFH.



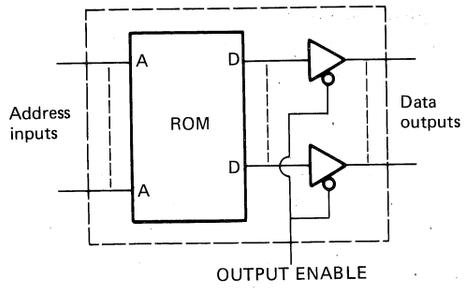


Figure 3-48 ROM with three-state outputs.

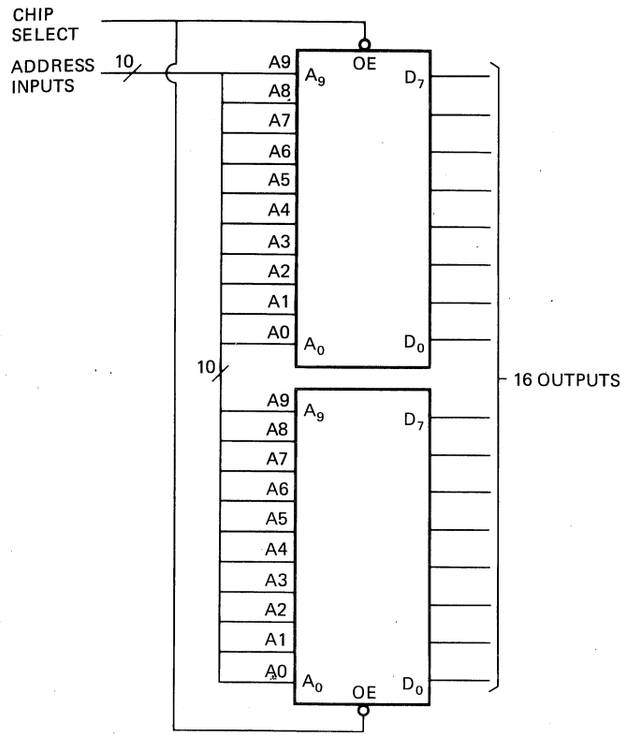


Figure 3-49 1K X 16 ROM.

Read-Write Memory

Often large amounts of data must be remembered in a digital system. Although individual registers could be used to store these data, read-write memory (RWM) is usually a better alternative. An RWM circuit is simply a large number of flip-flops fabricated on a single integrated circuit. Each flip-flop or group of flip-flops is assigned an address and may be changed or sensed whenever it is addressed. Changing a word in RWM is called *writing*. Sensing a word is called *reading*. In Fig. 3-51, notice that this RWM has four data inputs and four outputs. Thus, the states of four flip-flops may be sensed or changed at one time. The RWM circuit contains 1024 flip-flops; so that eight address inputs must determine which one of 256 groups of four flip-flops may be selected. Each group is called a word; so this is a 256-word RWM with each word containing 4 bits. This is represented by the notation 256 X 4. Three control signals are also required for operation of this RWM. An output enable (*OE*) turns on the three-state outputs of the RWM, to allow the data selected at the address inputs to appear on the four outputs. A write (*WR*) signal causes the addressed flip-flops in the circuit to be changed to the state of the four data inputs. Finally, a chip-select (*CS*) signal enables or disables all functions (both read and write) for this integrated circuit, which allows

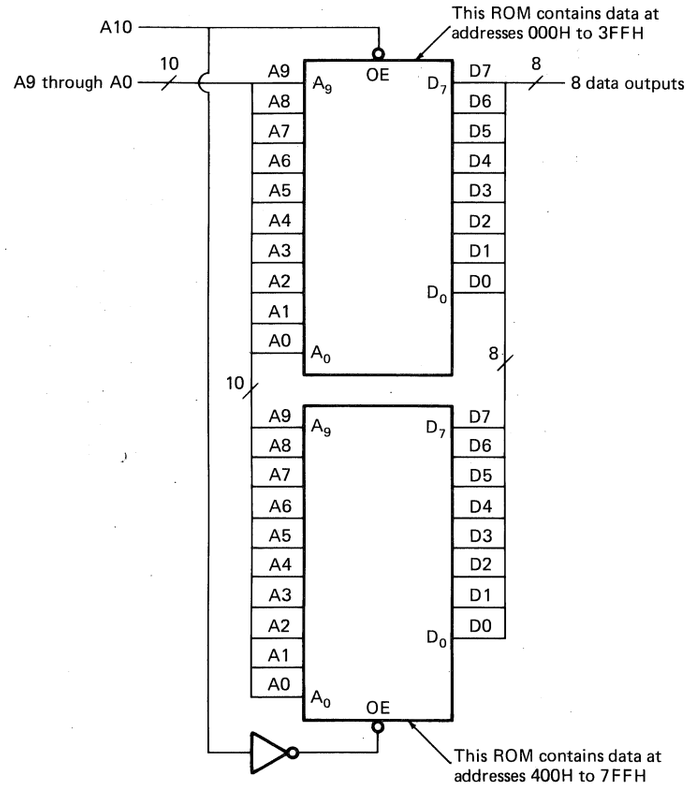


Figure 3-50 2K × 8 ROM.

for expansion by adding additional RWM circuits. Figure 3-52 shows an RWM using four 256 X 4 RWM circuits interconnected to form a 512-byte or 512 X 8 read-write memory. RWM integrated circuits are available in a wide variety of sizes including:

4 X 4
16 X 1
16 X 4
256 X 1
256 X 4
1024 X 1
1024 X 4
4096 X 1
16384 X 1
65536 X 1

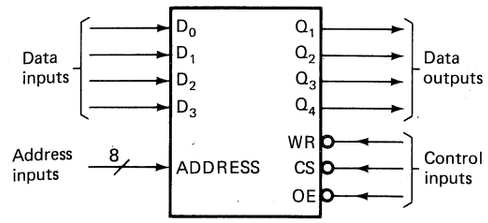
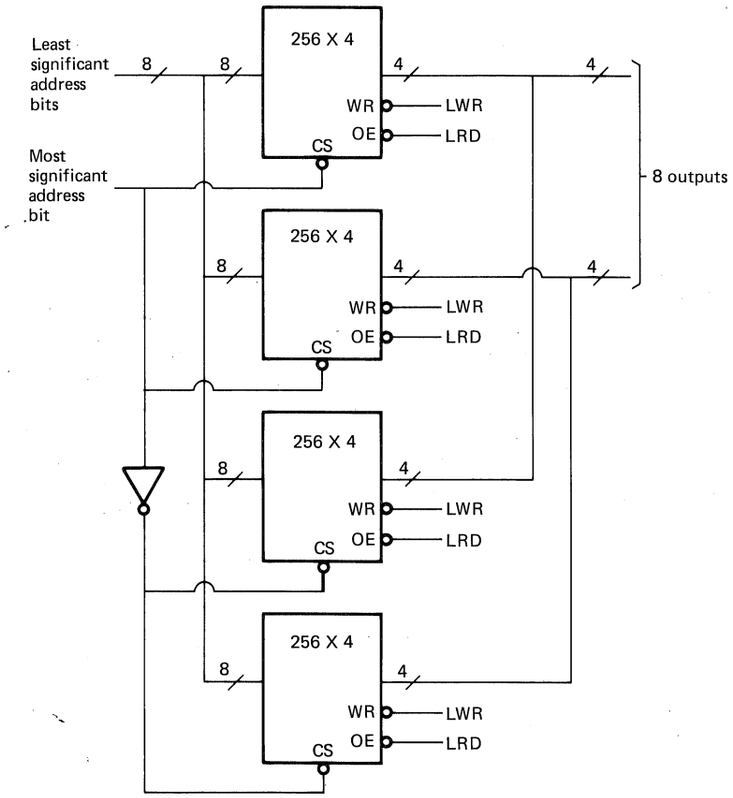


Figure 3-51 Read-write memory.

In addition, performance, which is usually measured by propagation delay and power supply requirements, varies greatly. Considerable effort is often necessary to specify an optimum RWM circuit.

There are many timing parameters associated with RWMs. The data sheet for the RWM must be consulted to ensure that no timing restriction is violated. It is important to know that the write function of an RWM is *not* edge-triggered. As long as the write and chip-select inputs are asserted, the data at the RWMs inputs will affect the con-

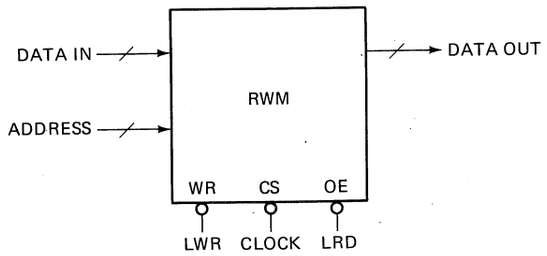


Data inputs not shown for clarity. Data inputs are connected similarly to outputs.

Figure 3-52 512 × 8 RWM.

tents of the memory flip-flops. The RWM must not be enabled to write data inputs until both address and data inputs are stable for a period sufficiently long to ensure they are properly recognized by the RWM. Figure 3-53 shows a read-write memory and its timing diagram. The chip-select input of the RWM is connected to the clock so that the RWM is enabled only in the *last* half of each ASM cycle. Address, data, and control inputs of the RWM change during the *first* half of the ASM cycle because they are either ROM outputs or other signals controlled by ROM outputs. Almost one half of a clock cycle occurs between changing the RWM's inputs and enabling the RWM. This half clock period is calculated to allow sufficient time for all address, data, and control signals to be properly recognized by the RWM.

The RWMs we have been discussing are all *static* RWMs. Another type of RWM is the *dynamic* RWM. Dynamic RWMs allow more bits to be fabricated onto a single integrated circuit. However, dynamic memories forget the data that have been stored in them unless they are periodically *refreshed.* Refreshing is accomplished by making sure that each memory cell is written or read periodically. This refresh period is specified by the manufacturer of the RWM and is usually around 2 ms. Also, dynamic RWMs are constructed so that only a small fraction of memory words must be



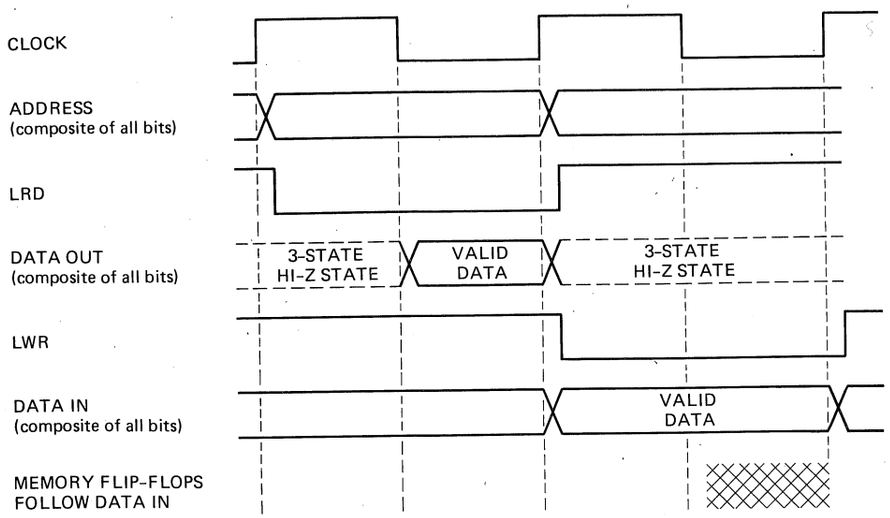


Figure 3-53 RWM timing.

periodically addressed to refresh all the memory words. For example, only 64 memory bits must be addressed to completely refresh a 4096 X 1 RWM.

Finally, RWMs are often called RAMs (random-access memories). We didn't use the term RAM since read-only memories are also random access. The term RAM is very commonly used for a read-write memory for historical reasons.

Programmable Logic Arrays

A ROM is a standardized part that is fabricated in quantity at low cost. However, when used in an ASM, it is often inefficient. Many entries in the ROM will be identical owing to don't cares in the input-variable specification. A gate implementation of next-state and output functions may be made very efficient by appropriate use of K maps or other reduction techniques. However, a gate implementation may not be standardized and fabricated as an integrated circuit unless a very large number of circuits are required. A compromise exists between these two extremes in the form of a programmable logic array (PLA). A PLA is designed to regularize the sum-of-products form of logic functions so that a "standard" SOP may be fabricated as an integrated circuit. In fact, PLAs are commonly available as FPLAs (field programmable logic arrays) that can be programmed after manufacture, much like a PROM. A typical PLA is shown in Fig. 3-54. It consists of 48 AND gates, each with 14 inputs. Each input

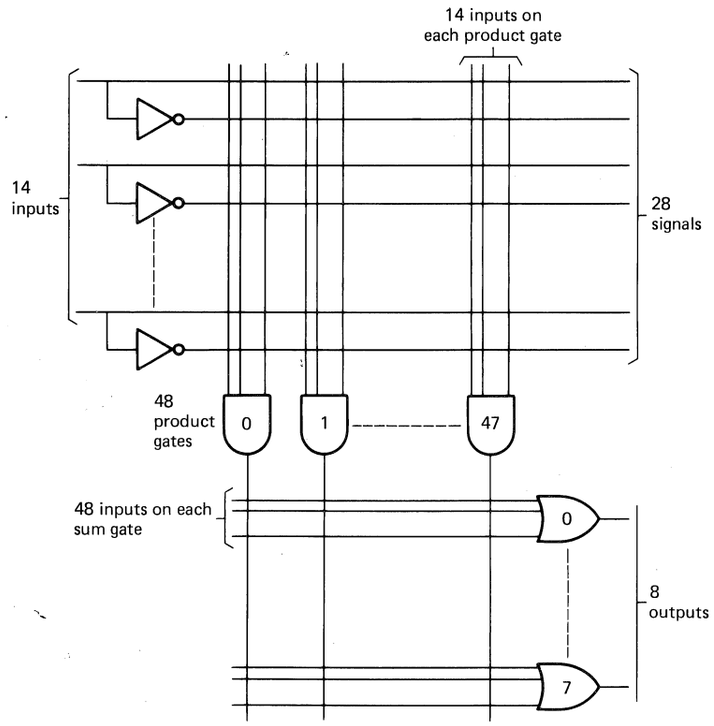


Figure 3-54 Programmable logic array.

Table 3-6 PLA specification for product gate 41

Product Gate	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Q_7	Q_6	Q_5	Q_4	Q_3	Q_2	Q_1	Q_0
41	0	0	X	X	X	X	1	0	1	X	X	X	X	1	1	–	–	1	–	–	1	1

may be connected to one of 14 input variables or to the complement of that variable or not be connected at all. In this way, up to 48 product terms may be generated by the PLA. Eight 48-input OR gates generate eight output functions. Each of these 48 inputs may be connected or not connected to 1 of the 48 product terms. This circuit will generate any eight functions of up to 14 variables with the restriction that a maximum of 48 different product terms are allowed. A ROM that generated any eight functions of 14 variables would require $2^{14} \times 8$ or 131,072 bits. A PLA is specified by listing the input combination corresponding to each of the 48 product terms and the outputs asserted by that product. Thus, the specification shown in Table 3-6 means that, whenever $D_{13} = D_{12} = D_6 = 0$ and $D_7 = D_5 = D_0 = 1$, then outputs Q_7, Q_4, Q_1, and Q_0 will be 1. Physically, this table entry means that AND gate number 41 has six inputs connected to $\overline{D_{13}}$, $\overline{D_{12}}$, D_7, $\overline{D_6}$, D_5, and D_0. The output of AND gate 41 is connected to inputs on OR gates corresponding to outputs Q_7, Q_4, Q_1, and Q_0.

There are two ways to design with PLAs. Generating a next-state and output table for a ROM-implemented ASM usually creates many don't care inputs. It is these don't cares that make a ROM implementation inefficient. However, don't care conditions need not be implemented in a PLA. If the number of lines of our next-state and output table is less than the number of PLA product terms, each line may be implemented as a product term in the PLA. In other words, each line (don't cares and all) of the next-state and output table corresponds to one line of the PLA specification.

If a number of entries of the next-state and output table exceeds the number of product terms available, the functions can be placed on K maps in the hope that the number of distinct product terms needed will be reduced when the K maps are reduced. The remaining product terms can then be programmed into the PLA directly.

Don't forget that all the MSI circuits previously discussed in conjunction with ROMs may be applied to a PLA implementation. Thus, a multiplexer can reduce the number of inputs required or a decoder may be used to increase the number of outputs.

PROBLEMS

3-1 Design a 24-bit multiplexer using three 8-input multiplexers. Design the multiplexer twice, once using each technique shown in the text.

3-2 Use multiplexers to implement the functions in Prob. 2-5. Use the smallest multiplexer possible.

3-3 Decoders can be expanded, as well as multiplexers:
(*a*) Design a 32-output decoder, using two 16-output decoders.
(*b*) Design a 16-output decoder, using *only* 4-output decoders.

3-4 (*a*) Assume you have 4-bit MSI counters, each with two enable inputs and a carry output. Design a 12-bit binary counter, using three of these MSI circuits.
(*b*) Assume that the above counters each have *LOAD* and *CLEAR* inputs. Show how these inputs should be interconnected so that the entire 12-bit counter will load or clear simultaneously.

3-5 Assume you have 8 bits of data on eight separate wires. You must design a circuit to output these 8 bits in serial form, repeatedly. One bit should follow another at each clock edge with no pause.
(*a*) Design a circuit using a shift register.
(*b*) Design a circuit using a multiplexer.

3-6 Repeat Prob. 1-5. Use a decoder for the outputs. Use a counter for the state register.

3-7 Repeat Prob. 3-6. Now, add a multiplexer for inputs.

3-8 Repeat Prob. 3-7. Now add another counter for timing the tram stopped state.

3-9 Repeat Prob. 1-7. Use MSI parts as needed. Be sure to use a multiplexer for inputs. Add a separate counter to perform the cycle timing function (which was just assumed to exist in Prob. 1-7). You do not have to draw the ASM chart, but you should be sure that it is *possible* to draw an ASM chart based on your solution to Prob. 1-7.