# 4

# Building Blocks with Memory

In the preceding chapters, we tacitly assumed that electronic devices are infinitely fast and that they generate outputs that depend only on the present input values. In this chapter, we explore the interesting consequences of violating these assumptions.

First, we examine what can happen when there are finite propagation delays within gates. Output signals from assemblies of gates sometimes have spurious short pulses that are not predicted by standard Boolean algebra. These spurious pulses are seldom useful, but we must contend with them, usually by waiting until they have gone away.

Next, we explore gate circuits that include feedback. Some of these circuits exhibit *memory,* which is an essential tool for the system designer. We consider useful sequential (memory) building blocks: flip-flops, registers, counters, and so on. These are basic tools for developing the digital architectures in the laboratory portion of the course.

We then discuss large memory arrays—RAMs, ROMs, and programmable logic devices.

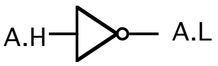
## THE TIME ELEMENT

### Hazards

The outputs of real gates cannot change instantaneously when an input is changed. Integrated circuits operate by movement of holes and electrons within some physical material, usually silicon. Not even very light particles such as electrons can move at infinite speeds, and their movement will always involve delays. The time between a change in an input signal and a corresponding change in an output is called the *propagation delay of* the circuit. When inputs change, an output may undergo a change from L to H or from H to L. The corresponding propagation delays are denoted $t_{pLH}$ and $t_{pHL}$. Propagation delays depend on the input waveforms, temperature, output loadings, operating power, logic family, and a host of other parameters. We will avoid all these factors by abstracting propagation delay by setting both $t_{pLH}$ and $t_{pHL}$ to $t_{p.}$ This is woefully inadequate to describe real world devices but serves nicely to illustrate the principals involved.

Another source of delay is the wire carrying signals between gates. Electricity in a wire can travel only about 8 inches in a nanosecond, so when wires become long, the *interconnection delays* may become serious.

Our purpose here is to show how these delays can create spurious outputs called *hazards.* Consider the following simple circuit that changes the voltage polarity of a signal:



Assume that the voltage at the input A has been stable for a long time. The output will also be stable and of the opposite voltage level. If the voltage at the input changes, the output will change a short time later. When an input changes from L to H, the output will change from H to L after a propagation delay $t_{pHL}$; similarly, an input H to L transition will produce an L to H output transition after a time $t_{pLH}$. Figure 4–1 is *a timing diagram,* a graph *of* input and output values (either voltage or logic) as a function of time. Each variable's graph is called *a waveform.*

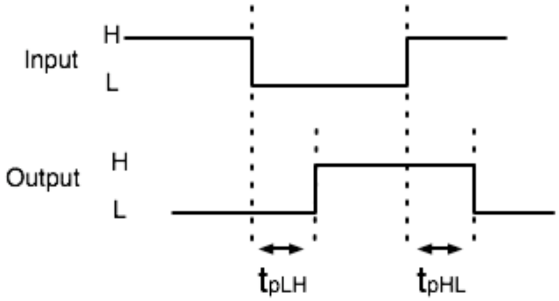
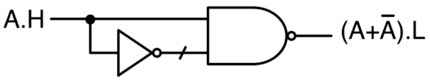


**Figure 4–1.** Timing diagram showing propagation delays in a logic circuit

To see what can happen when we introduce time into Boolean algebra, consider the following circuit, whose output is $A+\overline{A}$



Of course, we know that $A+\overline{A}=T$ regardless of the logic value of A, and we predict, from Boolean algebra, that the output of the circuit will always be L. But assume that each circuit element has a propagation delay $t_P$ for any transition. If A changes from T to F, the voltage pattern in Figure 4–2 will prevail; there is a spurious high-voltage (F) output that lasts for one gate delay.

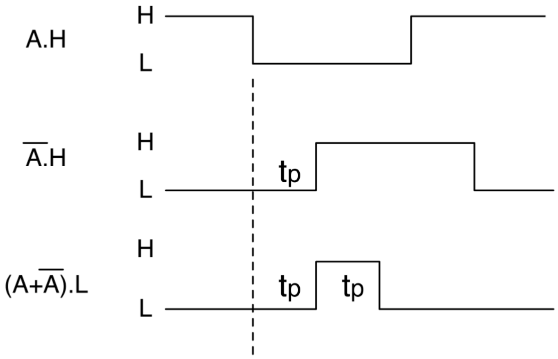
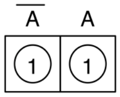


**Figure 4–2.** A hazard caused by propagation delay in an inverter

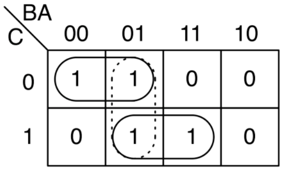
These spurious outputs of combinational circuits, called *hazards* or *glitches,* are common in digital systems. Fortunately, given sufficient time they will die out and the outputs of gates will assume the values predicted by classical Boolean algebra.

Occasionally, it is necessary to generate gate outputs that are *clean* - that have no hazards. It can be shown that a function *may* have a hazard if the function's Karnaugh map has adjacent l's not enclosed in the same circle. The preceding example, when plotted on a one-variable K-map, becomes



The two adjacent l's do not share a common circle, and indeed the circuit has a hazard. If we circle both l's in the K-map, we have the TRUE function, which is hazard free.

The following function is a more complex example

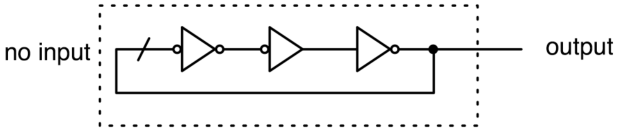


The theory is that a circuit based on the two solid loops may or may not contain a hazard; however, if we build a circuit that includes the dashed loop, we can be sure that the circuit will have no hazards. Using the dashed loop requires extra hardware (additional AND and OR gates), a necessary penalty when we cannot tolerate hazards. This technique of eliminating hazards works in simple sum-of-products circuits derived from K-maps. In more general circuits, the elimination of hazards is quite complex, and therefore we must use finesse instead of brute force. Rather than use design techniques that require hazard-free signals, we will make our designs *insensitive* to the hazards that occur when combinational inputs are changing. A standard technique is to wait a fixed time after gate inputs change, during which time the hazards will die out. We may then proceed to use the stable signals. This idea is the basis of synchronous (clocked) design, which we introduce in Chapter 5.

### Circuits with Feedback

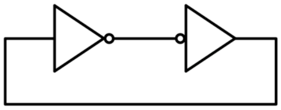
In the preceding section, we discussed purely combinational circuits. Except for momentary hazards, the behavior of the circuits is adequately described by Boolean algebraic or truth table methods used in the previous chapters. After a sufficient time to "settle," the circuit's outputs become a function only of the inputs. We now consider another class of circuits, in which the value of the outputs after the settling time depends not only on the external inputs but also on the original value of the outputs. Such circuits exhibit *feedback:* the output feeds back to contribute to the inputs of earlier elements in the circuit.

Feedback yields curious results in some circuits. The following circuit, which has no external inputs, consists of three inverters and feedback:



The voltage at the output is fed back into the input where, after a short time, it appears inverted on the output. The new voltage causes a similar inversion; the output voltage *oscillates* rapidly.

Remove one inverter from this circuit, produces the following circuit:



If you construct this circuit with real inverters and apply operating power, the output voltages of each inverter will go through a period of instability, during which one output will settle at a high level and the other at a low level. Although there is no way to predict which output will be high and which low, the circuit will remain stable after the settling time. You can verify the stability by tracing voltages around the circuit. Redrawing the circuit, as in Figure 4–3, helps to illustrate the stability. Since neither of the inverter feedback circuits shown above has external inputs, Boolean algebra is powerless to describe the circuit's behavior.

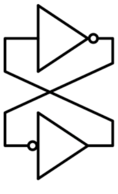


**Figure 4–3.** Memory displayed by a circuit with feedback

## SEQUENTIAL CIRCUITS

The circuit in Fig. 4–3 exhibits a primitive form of memory: the circuit "remembers" the resolution of the initial voltage conflict. Without external inputs, this memory is useless. In contrast, certain feedback circuits with external inputs not only exhibit memory, but also allow the designer to control the value stored in the memory. Controllable memory is the digital designer's most powerful tool. Digital systems with memory are called *sequential circuits.*

Sequential devices may be synthesized from gates, but this procedure is not within the scope of this book, except in that it shows the typical structure of some simple memory elements. Manufacturers have packaged proven gate designs of various sequential circuits, and we can use these as building blocks once we know their behavior. Sequential building blocks have names such as *latch, flip-flop,* and *register.*

### Unclocked Sequential Circuits

**The latch.** The latch is the simplest data storage element. Its logic diagram is in Figure 4–4. To describe the action of the latch, we must introduce time as a parameter. This was not necessary in combinational logic, but it is always necessary in sequential logic. The timing diagram is frequently used to portray sequential circuit behavior. To analyze the latch circuit, consider the several cases shown in the timing diagram, Fig. 4–5.

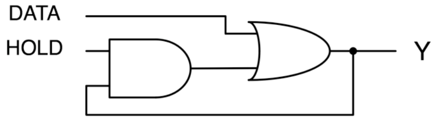


**Figure 4–4.** A latch circuit

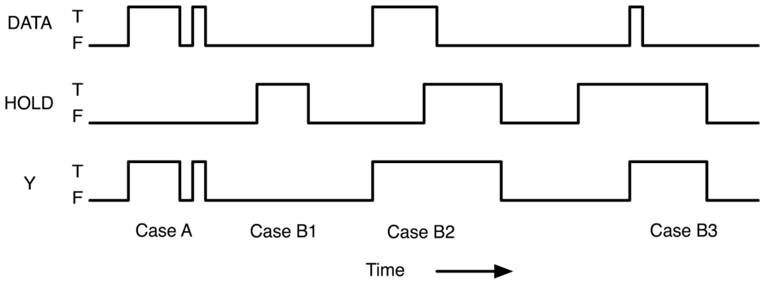


**Figure 4–5.** A Timing diagram for a latch. Note the 1's catching behavior

CaseA. HOLD = F. In this case, Y = DATA

CaseB. HOLD = T. Any occurrence of DATA = T will be captured, and the output will thereafter remain true until HOLD becomes false. We consider three sub cases:

(a) DATA is false throughout the period when HOLD is true. Then Y is false.

(b) DATA is true when HOLD is true. When HOLD becomes true, the latch captures the (true) value of DATA and stores it as long as HOLD remains true. (After HOLD becomes false, case A applies.)

(c) DATA is false when HOLD becomes true. At the beginning, Y is false. The first occurrence of a true signal on the DATA line will cause Y to become true; the output will remain true until HOLD becomes false.

The latch has the property of passing true input data to its output immediately. This behavior is sometimes useful in digital design, but it can be quite dangerous. Suppose that while HOLD is true, a glitch or noise pulse on the DATA line causes DATA to become true momentarily. This momentary true, or 1, will cause output Y to become true and remain true as long as HOLD is true. This behavior is sometimes called 1's catching; it is only rarely useful.

The latch circuit in Figure 4–4 is not frequently used, and it is not generally available as a library circuit. A true latch is a memory element that exhibits combinational behavior at some values of its inputs. There are other varieties of latch; unfortunately, designers use the term loosely to describe various signal-capturing events. We will soon develop more satisfactory memory devices.

Timing diagrams may be used to show gross voltage or logic behavior, or to show fine detail. The timing diagrams in Figures 4–1 and 4–2 show the fine detail of gate delays. On the other hand, the timing diagram in Fig. 4–5 shows only the gross behavior of the latch circuit and is accurate only when the time scale is sufficiently large. On a fine time scale, the output Y in Fig. 4–5 would be shifted slightly to the right to account for the delays incurred while changes in DATA or HOLD are absorbed by the gates in the circuit.

**The asynchronous RS flip-flop.** The feedback circuit in Fig. 4–3 exhibits a peculiar form of memory: it remembers which inverter had a low output after "power-up." The circuit has two stable states, and is indeed a memory, albeit a useless one, since there is no way to change it from one state to the other. By changing *the* inverters to two-input NOR gates, we obtain a useful device known as the *asynchronous RS flip flop* (see Fig. 4–6). We will study voltage behavior in this circuit before we introduce the concept of logic truth.

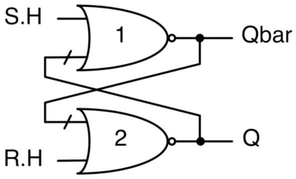


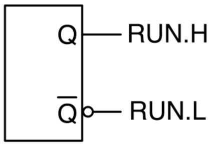
**Figure 4–6.** An asynchronous RS flip-flop constructed with NOR gates

The RS flip-flop is *a bistable* device, which means that in the absence of any inputs it can assume either of two stable states. To see this, assume that R = S = L, and assume that the output, Qbar, of gate-1 is L. Gate-2 will then present a high voltage level to Q. When this H feeds back to the input of gate 1, it will produce an L at Qbar, which is consistent with our original assumption about its polarity. We can describe this behavior by saying that the circuit is in *a stable state* when gate-1 outputs L and gate-2 outputs H. Once the circuit assumes this state, it will remain there as long as there are no changes in the R and S inputs.

There is another stable state during which gate-1 outputs H and gate-2 outputs L. We could predict this from the symmetry of the circuit, but you should verify it by tracing signals as we just did.

We have shown that the circuit of two cross-coupled NOR gates can exist in two stable states. We call one of the stable states the *set state* and the other the *reset state.* By convention, the set state corresponds to Q = H, and the reset state to Q = L.

The conventional representation of a flip-flop is a rectangle from which Q.H emerges at the upper right side. Most flip-flops produce two voltages of opposite polarity and the second output appears below the Q.H output. In data books, the second output is usually called $\overline{Q}$. Since this output behaves like Q with a voltage inversion, mixed logicians prefer to designate the signal as Q.L, the alternative voltage form of Q.H. Nevertheless, the nomenclature within the flip-flop symbol, like our other building blocks, must conform to normal usage so there will be no confusion about the interpretation of the pins of the module. The interior of the symbol serves to identify pin functions; the external notations for inputs and outputs represent specific signals in a logic design. Thus, if we have a flip-flop whose output is a logic variable RUN*,* our standard notation for the output is



Now we will consider the S and R inputs to the RS flip-flop. We know that as long as S and R are FALSE (low), the flip-flop remains in its present state. We may use the S and R lines to force the flip-flop into either state. S is a control input that places the RS flip-flop into the set state, Q = TRUE, (high), whenever S = TRUE, (low). Analogously, R = H resets the flip-flop by making Q = L. The obvious association of truth and voltage is T = H at S, R, and Q, so that we set the flip-flop by making S = T, and we reset by making R = T. This leads us to our usual mixed-logic notation for an RS flip-flop constructed of NOR gates:

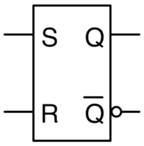
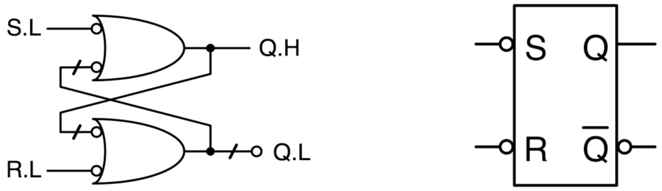


Figure 4–7 is a similar asynchronous RS flip-flop designed with NAND gates. This figure, a mixed-logic diagram of the cross-coupled gates, emphasizes that T = L at the inputs of this flip-flop.



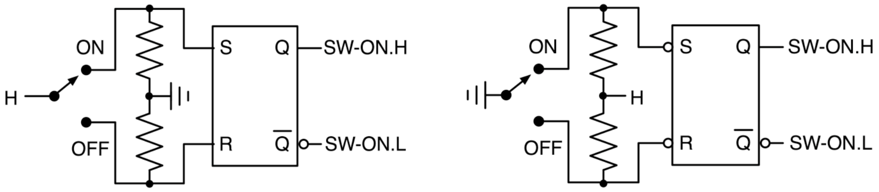
(a) Mixed logic circuit (b) circuit symbol

**Figure 4–7.** An asynchronous RS flip-flop constructed with NAND gates

The term *asynchronous* associated with the RS flip-flop implies that there is no master clocking signal that governs the activity of the flip-flop; suitable changes of S or R cause the outputs to react immediately. Asynchronous means *unclocked.* Its counterpart is *a clocked,* or *synchronous,* circuit. (Some workers refer to all unclocked storage elements as latches; we will not adopt this practice.) The asynchronous RS flip-flop is sensitive to noise, or glitches, at the S input when in the reset state, and at the R input when in the set state. This sensitivity is occasionally useful, but in general you should avoid using asynchronous devices, since glitches are undesirable byproducts of gate delays and noise is usually unpredictable in digital systems. Part of our goal is to develop design techniques that bypass these inevitable problems. Therefore, one of our dictums will be: don't use asynchronous RS flip-flops as a general design tool.

**Switch debouncing.** However, there is one standard use of the asynchronous RS flip-flop, as *a switch debouncer.* It is an unfortunate fact that mechanical switches do not make or break contact cleanly. At closure there will be several separate contacts over a period of many microseconds. The same is true during switch opening. The switch bounces. Since we do not wish to use a bouncy or spiky signal in our digital designs, we need a way to clean up the switch output.

Whenever a mechanical switch changes its position, we wish the associated digital signal to undergo one smooth change of voltage level. The asynchronous RS flip-flop is well suited for this. Figure 4–8 contains two switch-debouncing circuits. The resistors keep the control inputs inactive unless the voltage from the switch forces one input to become active. When the switch is off, it is constantly resetting the flip-flop, producing a constant F output. As the switch moves toward the on position, there will be a period of oscillation or bounce on the R input, caused by the mechanical switch breaking and making its contact with its off terminal. The S input is false throughout all of this, and the repeated resetting does not affect the false output of the flip-flop. There follows a "long" period when the switch moves between its off and on positions, during which time both S and R are false. Then the switch begins its bouncy contact with the on terminal. The first contact causes S to become true, setting the flip-flop to its true state, where it remains throughout the on-position bounce and until the switch is returned to off.



(a) High-active inputs (b) Low-active inputs

**Figure 4–8.** Mechanical switch debouncing circuits using asynchronous RS flip-flops

**RS flip-flop Ambiguous behavior.** Of the four voltage combinations of the S and R inputs, we have used three: to hold, set, and reset. What happens when S and R are simultaneously true? In the NOR-gate version, the voltages at both outputs of the flip-flop will be low—a disturbing situation. In the NAND gate version, both will be high. Although this deviation from voltage complementarity is unwelcome, it nevertheless represents a well-defined and stable configuration of the flip-flop. But watch what happens when we try to retreat from this configuration of inputs. If we change only one of the inputs, the flip-flop enters either the set or reset state without difficulty. But if we try to change both inputs simultaneously (in an attempt to move to the hold state), the flip-flop is in deep trouble. Consider the NOR-gate version of the RS flip-flop, Figure 4–6. If the voltages at S and R are both high then they are low at both Qbar and Q. If the voltages at S and R both become low simultaneously, then after one gate delay both gates in the flip-flop will produce high outputs. These high outputs, feeding back to the inputs of the NOR gates, will result in low gate outputs after one more gate delay. And so on. The circuit oscillates rapidly, at least at the beginning, with both outputs producing either high or low voltage levels "in phase." The resulting changes occur so rapidly that the flip-flop is forced out of the digital mode of operation for which it was designed, and the output voltages quickly cease to conform to reliable digital voltage levels—an example of *metastable behavior* discussed in appendix *. Eventually, the slight differences in the physical properties of the two gates will allow the flip-flop to drop into the set state or the reset state. The time required for the voltages to settle and the final result are uncertain, so this behavior is of no use to designers. Therefore, it is considered improper design practice to allow R and S to be asserted at the same time.

**Excitation tables.** Timing diagrams are useful for displaying the time dependent characteristics of sequential circuits, but for most purposes a tabular form is better. The *excitation table* is the sequential counterpart of the truth table or voltage table for combinational circuits. The excitation table looks much like a truth table, but it contains the element of time. In a sequential circuit, the new outputs depend on the present inputs and also on the present values of the outputs. We can display the behavior of the RS flip-flop of Figure 4–6 in the following excitation table:

| S | R | $Q_{(t)}$ | $Q_{(t+\Delta)}$ | |
|---|---|---|---|---|
| L | L | q | q | Hold |
| L | H | q | L | Reset |
| H | L | q | H | Set |
| H | H | q | | Disallowed |

$Q_{(t)}$ is the value of output Q at time t; $Q_{(t+\Delta)}$ is the value of Q at a small time Δ after t, where Δ is sufficiently long for the effects of the gate delays to settle down.

The excitation table is also useful for displaying the logical behavior of sequential circuits. For instance, the following excitation table describes the logical behavior of RS flip-flops, using a modification of the previous notation:

| S | R | Q | Q’ | |
|---|---|---|---|---|
| F | F | q | q | Hold |
| F | T | q | F | Reset |
| T | F | q | T | Set |
| T | T | q | | Disallowed |

In the literature, notations for excitation tables vary greatly and in this chapter we will use a variety of forms. You should be able to recognize these notational differences.

**Clocked Sequential Circuits**

Asynchronous flip-flops are l's catchers. A more useful class of flip-flop is available for general digital design. In these flip-flops, outputs will not change unless another signal, called the *clock,* is asserted. Since the activity is synchronized with the clock signal, these flip-flops are called *synchronous.* Digital systems usually have a repetitive clock with a square waveform. The clock signal alternates between its H and L signal levels. Depending on the application, we may view either H or L as representing truth on the clock line, although in almost all our applications we shall use the T = H assignment for clock signals; you will encounter clocked circuits throughout the remainder of this book.

**Clocked RS flip-flop.** We can derive a clocked flip-flop from an asynchronous RS flip-flop by gating the R and S input signals to restrict the time during which they are active, as in Figure 4–9. The flip-flop outputs may change whenever the clock is true—a potentially risky situation similar to the 1's catching of the latch circuit. In digital systems, flip-flop outputs often contribute to combinational circuits that produce inputs to other flip-flops. Shortly after the rise of the clock, the system is in "shock" owing to the changing of flip-flops. During this period of shock, hazards may be present that can feed erroneous signals into flip-flop inputs while the clock is *still true,* resulting in false setting or resetting of the flip-flops.

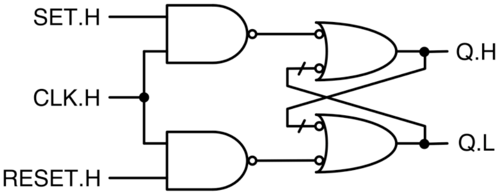


**Figure 4–9.** A clocked RS flip-flop circuit (bad design)

It is natural to try to avoid this problem by making the true portion of the clock signal as narrow as possible. Unfortunately, this is not a good solution, since the system's behavior is crucially dependent on the quality of the clock and narrow clock signals are difficult to generate and distribute.

The aim is to reduce the time during which the flip-flop outputs respond to the inputs. Since altering the clock waveform leads to difficulties, can we achieve the goal by further modification of the flip-flop circuit itself? Can we devise a flip-flop that will recognize R and S only at a *single instant* and ignore the inputs at other times? Such behavior would be desirable because all flip-flops would change at precisely the same time if they were clocked from the same source. This would mean that we could arrange for all the R and S inputs on all flip-flops to be stable at the time of clocking, and the flip-flops would not be influenced by the shock of the changes induced just after clocking.

Flip-flops that allow output changes to occur only at a single clocked instant are called *edge-driven* or *edge-triggered.* An edge is a voltage transition on the clock signal, and may be either a positive edge (L→H) or a negative edge (H→L). The clocked circuit in Figure 4–9 is *level-driven,* since its outputs may change at any time during the true part of the clock cycle. In your designs of clocked sequential circuits, use only edge-driven devices.

**Master-slave flip-flop.** The master-slave flip-flop is a relic from the early days of integrated circuit technology, but is still widely used because of its pseudo-edge-driven characteristics. It is a relatively simple device that we can easily discuss at the gate level, so we will show how one is derived by extending the clocked RS flip-flop. Figure 4–10 is a master-slave flip-flop schematic. The master flip-flop will respond to inputs S and R as long as the clock signal is high. This period must be long enough to ensure that S and R are stable when the clock goes from high to low. This H→L transition, the negative clock edge, isolates the master flip-flop from the inputs S and R. The master flip-flop will now remain unchanged until the next positive clock edge.

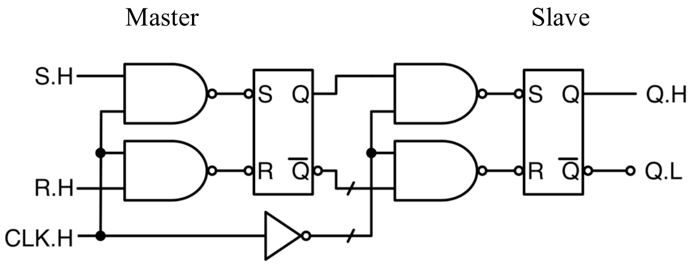


**Figure 4–10.** A master-slave clocked RS flip-flop

Because of the voltage inverter, the slave flip-flop does not become sensitive to its input until one gate delay after the negative clock edge. At that time, it receives its S and R inputs from a stable master flip-flop. The net effect is that the outputs of the master-slave combination change only on the negative clock edge rather than during a clock level.

**Pure edge-driven flip-flop.** The master-slave flip-flop appears to be an attractive edge-driven device. Why are we not content with this design? Because the master flip-flop is still a l's catcher during the positive half of the clock cycle. This means that R and S must stabilize during the negative half of the clock, since the master flip-flop will react to any T glitches during the positive clock phase. We could greatly simplify our digital circuit designs if we could eliminate the 1's-catching behavior. We need a flip-flop that samples its inputs only on a clock edge and changes its outputs only as a result of the clock edge. Such a device is called *a pure edge-driven flip-flop*. The F→T clock transition is called the *active edge.* It may be either the H→L or L→H transition, although in the most useful integrated circuits the L→H transition is the active edge.

The property of changing state and sensing inputs only at a given instant gives the designer a powerful tool for combating glitches and noise. We can now choose the time to look at signals and can fix that time to allow adequate stabilization of the system. We will make constant use of pure edge-driven sequential circuits in our designs. The internal structure of these devices is rather complex, but for purposes of digital system design it is not necessary for us to examine their construction in detail. Hereafter, in all our discussions of clocked sequential circuits, we will assume the use of pure edge-driven devices.

**Excitation tables for edge-driven flip-flops.** Assume that the edge-driven flip-flop is subjected to a steady stream of active clock edges. Each clock edge will cause the flip-flop to enter either its set or its reset state, in accordance with the values of its inputs and the current value stored in the flip-flop. Let us call the value stored in the flip-flop $Q_n$ after the flip-flop has received n clock triggers. If the flip-flop is in the set state after the $n^{th}$ clock edge, then $Q_n$ = T; if in the reset state, $Q_n$ = F. After the appearance of the next clock edge, the value of Q will be $Q_{n+1}$. The excitation table for edge-driven devices is a tabulation of $Q_{n+1}$ for all combinations of the exciting variables.

In the remainder of this chapter, we will use excitation tables to classify flip-flops. For the excitation table to be valid, we must ensure that the control inputs are stable for a short time before the active clock edge (the *setup time*), and perhaps for a short time after the active clock edge (the *hold time*). The input voltages may go through wild excursions prior to the onset of the setup time and after the hold time, as long as they remain stable during the setup and hold times. (Setup and hold times are device dependent and will be shown in data books.)

## CLOCKED BUILDING BLOCKS

In this section, we present the common building blocks for clocked digital design

### The JK Flip-Flop

Whereas the RS flip-flop displays ambiguous behavior if both R and S are true simultaneously, the JK flip-flop produces unambiguous results in all combinations of its inputs. A logical excitation table for the basic JK flip-flop is:

| Clock | J | K | Qn | Qn+1 | |
|---|---|---|---|---|---|
| F | X | X | q | q | |
| T | X | X | q | q | |
| ↑ | F | F | q | q | Hold |
| ↑ | F | T | q | F | Reset |
| ↑ | T | F | q | T | Set |
| ↑ | T | T | q | $\overline{q}$ | Toggle |

J is the counterpart of the S input of an RS flip-flop, and K is the counterpart of R. The first two lines of the excitation table demonstrate the edge-triggered behavior of the flip-flop: when the clock signal is a stable, false or true, the output of the flip-flop is insensitive to the other inputs. Often these lines do not appear in the excitation table, since such behavior is expected of an edge-triggered device. The remaining four lines in the table describe the flip-flop behavior when the clock undergoes its active (F→T) transition. The first three of these lines are analogous to the RS flip-flop. The last line shows that if both control inputs are true when the clock fires, the flip-flop will complement its output. This behavior is called *toggling.*

Now is the time to suppress some of the excitation table's detailed behavior and introduce the standard notation for such tables.

(a) Omit the first two rows. All edge driven devices imply this behavior

(b) In the above table omit the Qn column

(c) In the Qn+1 column replace q by its equivalent, Qn

The abbreviated (and standard) excitation table for the JK then becomes:

| Clock | J | K | Qn+1 | |
|---|---|---|---|---|
| ↑ | F | F | Qn | Hold |
| ↑ | F | T | F | Reset |
| ↑ | T | F | T | Set |
| ↑ | T | T | $\overline{Qn}$ | Toggle |

Library JK flip-flops come in various forms. The most interesting variations are:

(a) Active clock edge: positive or negative. On all clocked devices, we show the clock input as a small wedge inside the device symbol. A negative edge-triggered flip-flop has a small circle on the clock input, a positive edge triggered flip-flop would not have a circle:

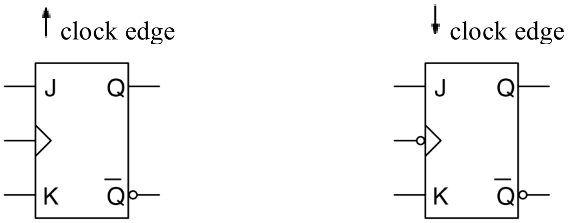


**Figure 4–11.** JK flip flops with positive and negative clock edges

(b) Availability of asynchronous R and S inputs. These are often called *direct clear* or *preclear* and *direct set* or *preset.* One, both, or neither may be present. Direct set usually appears at the top of the flip-flop symbol, and direct clear at the bottom. Truth is usually a low voltage level, in which case these inputs will bear small circles. As long as an asynchronous input is asserted, it will override the normal synchronous behavior of the flip-flop. Often the asynchronous set and clear pins are not named—their function is implied from their placement on the flip-flop symbol

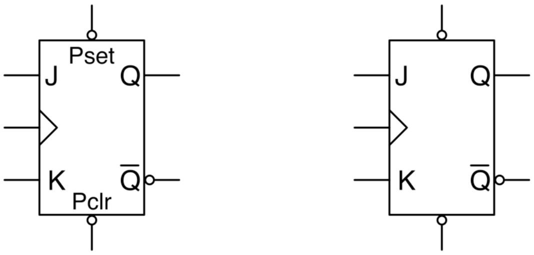


**Figure 4–12.** JK flip-flops with asynchronous set and clear

| Preset | Preclear | Clock | J | K | Qn+1 | Action |
|---|---|---|---|---|---|---|
| L | L | X | X | X | ---- | Disallowed |
| L | H | X | X | X | H | Direct set |
| H | L | X | X | X | L | Direct clear |
| H | H | ↑ | L | L | Qn | Hold |
| H | H | ↑ | L | H | L | Clear |
| H | H | ↑ | H | L | H | Set |
| H | H | ↑ | H | H | $\overline{Q}_n$ | Toggle |

Excitation table for the flip flops of figure 4–12

The JK flip-flop is our most powerful storage element, and you must master its use. There are several ways of using a single flip-flop, and later you will see many larger constructions based on this flexible element. (Be careful, some library JK's work on negative clock edges without telling you. Verify before using)

**JK flip-flop as controlled storage.** The most general use of the JK flip-flop, and the one that gives it such power and flexibility, is as a storage element under explicit control. In digital design, whenever we must set, clear, or toggle a signal to form a specific value for later use, we usually think of a JK flip-flop. Another standard use is setting a flag at one time but clearing it at a later time—for this situation automatically think JK. The penalty for this generality is the need to control two separate inputs.

**JK flip-flop for storing data.** The JK flip-flop is basically a controlled storage element. On occasion, we wish to adopt a different posture and view the JK flip-flop as a medium for entering and storing data. From the excitation table, we see that Qn+1=Qn whenever J = K = F at the clock edge. This is simply a data-storage mode. All that is necessary to continue holding data in the flip-flop is to ensure that J = K = F during the setup time before each clock edge.

**JK flip-flop for entering data.** The J and K inputs are not data lines; they are control lines for the flip-flop storage. Nevertheless, we can view the JK flip-flop as a data-entry device. We can enter data in three ways:

(a) Clearing, followed by later setting if necessary.
(b) Setting, followed by later clearing if necessary.
(c) Forcing the data into the flip-flop in one clock cycle.

The rule for case (a) is:

> If you are sure that the flip-flop is cleared, you may enter data D into the flip-flop on a clock edge by having J = D, independent of the value of K.

Case (b) is analogous to case (a). The rule is:

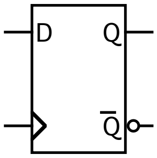
> If you are sure that the flip-flop's output is true, you may enter data D into the flip-flop on a clock edge by having K = $\overline{D}$, regardless of the value of J.

You should verify the rules for cases (a) and (b).

As for case (c), the designer usually cannot guarantee that a flip-flop will be in a given state. Proceeding as we did in cases (a) and (b) would waste one clock cycle for the initial clearing or setting operation. It would be nice to have a mode that would force data to enter the flip-flop at a clock edge, regardless of the present condition at the output. Such a data-entry mode is called *a jam transfer,* since the data is "jammed" into the flip-flop independent of prior conditions. Examination of the excitation table for the JK flip-flop shows that such a mode is indeed available. We enter data D as follows: If D = F, J must equal F and K must equal T. If D = T, J must equal T and K must equal F. Combining these conditions, we see that Qn+1 will equal D whenever J = D and K = $\overline{D}$.

**The D Flip-Flop**

The D (Delay) flip-flop has a simpler excitation table than the JK, and is used in applications that do not require the full power of the JK flip-flop. The symbol and excitation table for the D flip-flop are:



| D | Qn+1 |
|---|---|
| 0 | 0 |
| 1 | 1 |

Most libraries will have these common varieties of D flip-flops:

(a) The active clock edge can be either positive, ↑, or negative, ↓, which is shown by the absence or presence of a small circle on the clock terminal.

(b) Direct (asynchronous) set and clear inputs appear in these combinations: both, neither, or clear only. Almost always, these inputs, when present, are low active, and appear in the diagram with the small circle. These asynchronous inputs are l's catchers, and you should only use them with great caution.

(c) Some D flip-flops have only the Q output; others provide both polarities. Although it appears to be ideal for data storage, there are, in fact, just a few common uses of the D flip-flop in good design.

**D flip-flop as a delay.** As its name implies, the D flip-flop serves to delay the value of the signal at its input by one clock time. You will see such a use in Chapter 6 when we discuss the single-pulser circuit for manual switch processing.

**D flip-flop as a synchronizer.** One natural application of the D flip-flop is as a synchronizer of an input signal. Clocked logic must sometimes deal with input signals that have no fixed temporal relation to the master clock. An example is a manual pushbutton such as a stop switch on a computer console. The operator may close this switch at any time, perhaps so near the next edge of the system clock that the effect of the changing signal cannot be fully propagated through the circuit before the clock edge arrives. If the inputs to clocked elements are not stable during their setup times, their behavior is not predictable after the clock edge: some outputs may change, others may not. We need some way to process this manual switch signal so that it changes only when the active clock edges appear. This is called *synchronization.* Since the output of a clocked element changes only in step with the system clock, we may use the D flip-flop as a synchronizer by feeding the unsynchronized signal to the flip-flop input. We deal with this matter more fully in later chapters.

**D flip-flop for data storage.** The D flip-flop appears to be well suited to data entry and storage. Unfortunately, designers use it far too often for this purpose. The problem is that every clock pulse will "load" new data and this is seldom wanted. We usually need a device that allows us to control when the flip-flop accepts new data, just as we could with the JK flip-flop. With the D flip-flop, it seems natural to *gate the clock* by AND'ing it with a control signal in order to produce a clock edge at the flip-flop only when we wish to load data. This is a dangerous practice, as you will see in later chapters. Clocked circuit design relies on a clean clock signal that arrives at all clock inputs simultaneously. We have the best chance of meeting these conditions if we use unmodified clock signals. This means that the devices will be clocked every cycle, so we must seek other ways of affecting the necessary control over the flip-flop activities.

**The enabled D flip-flop.** To alleviate the problems caused by gating the clock input to a D flip-flop, we will construct a new type of device called the *enabled D flip-flop.* Figure 4–13 shows the principle. The circuit consists of a D flip-flop with a multiplexer on its input. A new control signal LOAD appears, in addition to the customary data input.

The system clock goes directly to the clock input, thereby avoiding the problems of a gated clock. As long as LOAD is false, the data selector selects the current value of the flip-flop output as input to the flip-flop. The net effect is that Q recirculates unchanged: the flip-flop stores data. When LOAD=T, the multiplexer routes the external signal DATA into the D input, where it will be loaded into the flip-flop on the next clock edge. The loading process is a jam transfer. Further, and most important, the enabled D is *insensitive to glitches* on the LOAD signal as long as it has stabilized before the clock edge

The enabled D flip-flop is the element of choice for simple data storage applications. Although we can accomplish the same effect with the JK flip-flop, the enabled D device provides a more natural way of handling data. Curiously, some libraries don't contain enabled D flip-flops, but they are easily synthesized in any event.

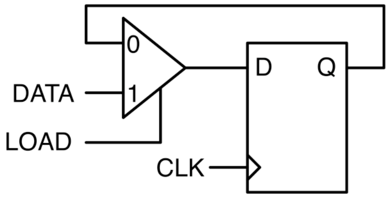


**Figure 4–13.** An enabled D flip-flop

## REGISTER BUILDING BLOCKS

A *register* is an ordered set of flip-flops. It is normally used for temporary storage of a related set of bits for some operation. This is a common activity in digital design, especially when the system must process byte, or word-organized data. You are familiar with the use of the word *register* in the context of digital computers, but the notion is more general than just accumulators and instruction registers. Multiple-bit storage is such a desirable architectural element that it is a natural candidate for building blocks. Your library will likely contain a wide variety of register elements, usually configurable as to width and presence or absence of direct set and direct clear.

### Data Storage

**Enabled D register.** The most elegant data storage element for registers contains the enabled D flip-flop. As you have seen, we favor the enabled D configuration because we may hook the system clock directly to the device's clock input. The apparently small point of not gating the clock line is really of great importance to the reliability of the system, and you should adopt the practice routinely. Some libraries will just call this an “enabled D” register. You will need to experiment with a simulator to see if it conforms to the logic of figure 4–13.

**Pure D register.** There are a few occasions when a register of pure D flip-flops is the element of choice. Pure D registers are also available, usually with a common asynchronous (direct) clear input. The only reason to choose such an element is if you want the direct clear feature; you know to be wary of its 1's catching properties.

### Counters

**Modulus counting.** Counting is a necessary operation in digital design. Since all binary counters are modulus counters, we will explore the concept of modulus counting before we examine the hardware for it.

Counting the positive integers is an infinite process. We have a mathematical rule for writing down the integer n + 1 if we are given the integer n. This may cause the creation of a new column of digits; for example, if *n* is the three-digit decimal number 999, then n + 1 is the four-digit number 1000. In an abstract mathematical sense, the creation of the fourth digit is trivial. Not so in hardware.

Hardware counters are limited to a given number of columns of digits, and thus there is a maximum number that a counter can represent. A three-digit decimal counter can represent exactly $10^3$ different numbers, from 000 through 999. We define such a counter as *a modulus* (mod) 1000 counter. (A *number M, modulo some modulus N*, written M modulo N, is defined as the remainder after dividing M by N.) Another way of viewing this is that the counter will count normally from 000 through 999, and one more count will cause it to cycle back to 000. An automobile's odometer behaves much the same way.

**Counting with the JK flip-flop.** The JK flip-flop, operating in its toggle mode, goes through the following sequence

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Clock pulse number: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | ... |
| Flip-flop output Q: | 0 | 1 | 0 | 1 | 0 | 1 | 0 | ... |

We see that the flip-flop behaves as a modulo-2 binary counter. Counters of higher moduli can be formed by concatenating other binary counters. For instance, a modulo-4 counter made from two modulo-2 counters must behave as follows

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Clock pulse number | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
| Counter outputs | 00 | 01 | 10 | 11 | 00 | 01 | 10 | 11 | 00 | ... |

Can we devise a logic configuration that will cause two JK flip-flops to count in this fashion? One answer is in Fig. 4–14. Here, for drafting convenience, we draw the least significant bit $Q_0$ on the left, whereas $Q_0$ appears on the right in the usual mathematical representation of the number $Q_1$ , $Q_0$. $Q_0$ alternates in value (toggles) at each clock. At alternate clock edges, $Q_1$ is clocked when $Q_0$ = T; at these times the value $Q_1$ toggles.

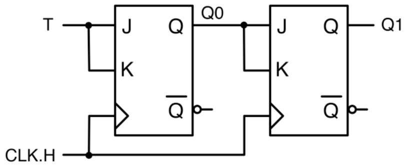


**Figure 4–14.** A two-bit binary counter. The least significant bit is on the left

Figure 4–15 contains another solution that appears to give equivalent results. Again, $Q_0$ will toggle at each clock pulse, since J = K = T on that flip-flop. This is necessary for a binary counting sequence. Every time $Q_0$ generates a ↓ transition, $\overline{Q0}$ generates a ↑ transition, which serves as the clock to the second stage. Figure 4–16 is a timing diagram for this circuit.

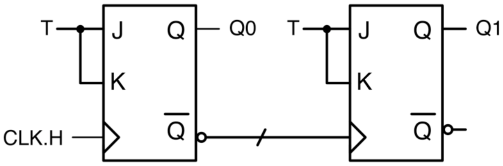


**Figure 4–15.** A binary ripple counter. The least significant bit is on the left.

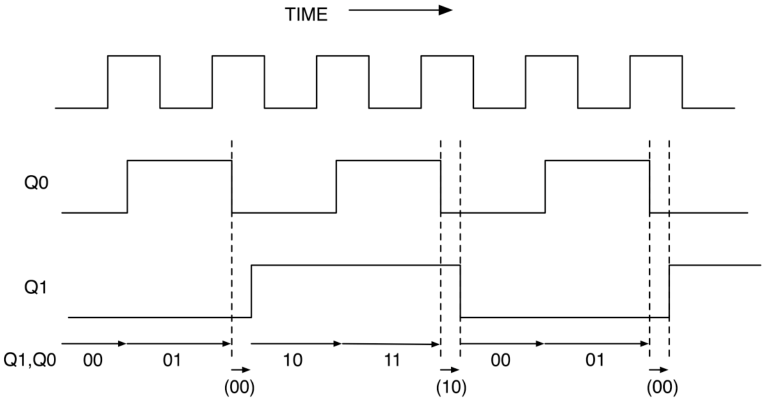


**Figure 4–16.** A timing diagram for a 2-bit ripple counter. Each stage suffers a cumulative propagation delay. Note the (00) and (10) transients. In synchronous counters there is only one delay.

The timing diagram for Fig. 4–14 is almost identical to Fig. 4–16; the difference is due to propagation delays. In Fig. 4–14, if we assume that $t_p$ is the flip-flop propagation delay, both $Q_1$ and $Q_0$ will change, $t_p$ nano-seconds after the clock edge, since J and K were stable during the setup time of both flip-flops. We define such counters as *synchronous.*

By contrast, $Q_1$ in Fig. 4–15 cannot change until tp nano-seconds after $Q_0$ has changed. Counters that change their outputs in this staggered fashion are called *asynchronous,* or *ripple,* counters, since a change in output must ripple through all the lower-order bits before it can serve as a clock for a high-order bit. $Q_1$ is behaving well in isolation, but if you are looking at the time relation of $Q_1$ and $Q_0$ you see the presence of transient bit patterns, which violate the binary count sequence

Ripple counters are easily extensible to any number of bits. Thus a modulo16 ripple counter would be as in Fig. 4–17. This simple configuration is useful if you are not interested in the temporal relation of $Q_3$ to any lower-order bits. A common example, the digital watch, has a 32,768—($2^{15}$) Hz quartz crystal oscillator as the primary timing source. The watch display is driven at a rate of 1 Hz, using the output of a 15-stage ripple counter.

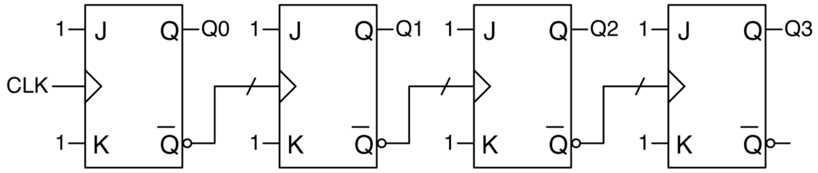
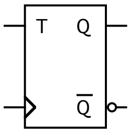


**Figure 4–17.** A 4-bit (modulo 16) ripple counter

Figure 4–17 uses the dreadful “logic 1” notation; and even though we do not like it, you will encounter it in your travels so it is best to face it now. “logic 1” implies a H voltage, *nothing more—it has nothing to do with logic*. Only if you restrict yourself to the positive-logic straightjacket does it also mean T. Similarly, “logic 0” means a L voltage, nothing more. This can trip you up in unsuspecting ways. Suppose you label inputs 0,1,2,3 to some logic block. 2 and 3 are perfectly good variable names in this context but 0 and 1 are not. Instead you will be feeding your logic block with *voltage* L, *voltage* H, *signal* 2, and *signal* 3. Of course, if you follow our convention of always starting variable names with a letter you will avoid this ambiguity.

**The toggle flip-flop** If you are in the counter domain, using the JK flip-flop is overkill. It is convenient to introduce a toggle flip-flop with this symbol and excitation table:



| T | Qn+1 |
|---|---|
| 0 | Qn |
| 1 | $\overline{Qn}$ |

Using the toggle flip-flop we can redraw figure 4–17:

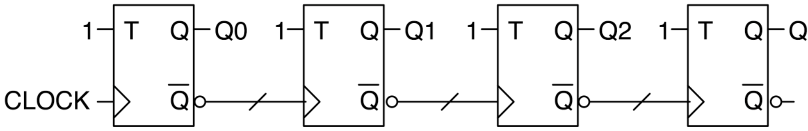


**Figure 4–18.** A 4-bit (modulo 16) ripple counter

To discover the problems that can arise with ripple counters, let us consider when transient patterns are generated in figure 4–16. Reverting to normal mathematical ordering, (Q1,Q0), we see the sequence as: 00→01→(00)→10 →11→(10)→00, where (00) and (10) are transient patterns lasting for $t_p$ seconds. Restating the binary patterns as equivalent decimal numbers the sequence is: 0, 1, (0), 2, 3, (2), 0. Count values are often used as select inputs to multiplexers; consider what happens when Q1, Q0, are so used in circuit 4–18, where we naively expect Y to select variables V0, V1, V2, V3 for further processing by downstream logic. Instead we momentarily inject (V0) and (V2) into the normal binary sequence.

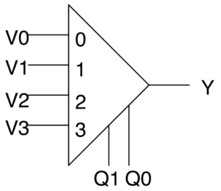


**Figure 4–19.**

Figure 4–14 represents a 2-bit special case of synchronous counters. The rule for changing the nth bit of a binary counter is that all lower bits must be 1. Using this rule, we can construct a modulo-16 synchronous counter from Toggle flip-flops, as in Fig. 4–20. At the cost of extra AND gates, we have manipulated the inputs to each flip-flop to cause the flip-flops to toggle at the proper time. Since a common clock signal runs to each flip-flop, the output changes will occur simultaneously, without ripple.

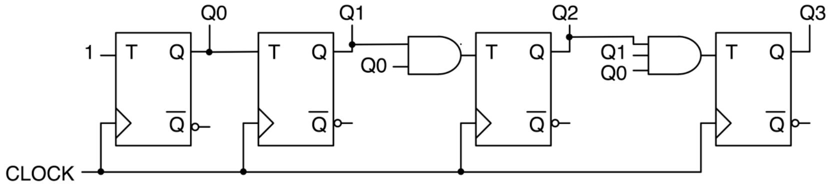


**Figure 4–20.** A 4-bit synchronous counter

Your simulation library will likely contain a wide variety of multi-bit counters classified by:

number of bits
up/down control
non-enabled or enabled outputs, (enables are most likely tri-state)
synchronous load (similar to enabled D flip-flops of 4–13)
asynchronous clear
may be configurable as to number of bits
modulus (may be binary or decimal)
cascadeability

It will pay big dividends to carefully test each flavor using a simulator before blindly using one in a synthesis.

## Shift Registers

*A shift register* performs an orderly lateral movement of data from one bit position to an adjacent position. We may construct a simple shift register from D flip-flops, as shown in Fig. 4–21. This circuit accepts a single bit of DATA and shifts it down the chain of flip-flops, one shift per clock pulse, (a right shift). Data enters the circuit serially, one bit at a time, but the entire 4-bit shifted result is available in parallel. Bits shifted off the right-hand end are lost. Such a circuit is a primitive serial-in, parallel-out right-shift register. (Be careful here, normal digital drafting conventions have inputs on the left and outputs on the right. Contrast this with binary data representations where the high order bit is on the left and low order bit on the right. Stop and think about which convention is being used whenever encountering registers or counters.)

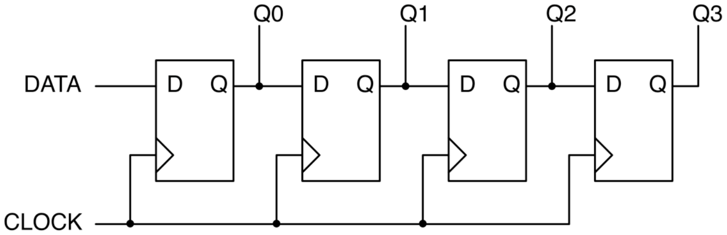


**Figure 4–21.** A simple serial-in, parallel-out, right-shift, register

In practice, we have need for four shift register configurations: serial-in, parallel-out; parallel-in, serial-out; parallel-in, parallel-out; and serial-in, serial-out. The parallel-in, parallel-out variety is the most general, subsuming the other forms. Let's design one. Assume that we are building a 4-bit general shift register. What features do we require?

(a) We must be able to load initial data into the register, in the form of a 4-bit parallel load operation.

(b) We must be able to shift the assembly of bits right or left one bit position, accepting a new bit at one end and discarding a bit from the other end.

(c) When we are not shifting or loading, we must retain the present data unchanged.

(d) We must be able to examine all 4 bits of the output.

Suppose we start with an assembly of four identical and independent D flip-flops, clocked by a common clock signal. Let the flip-flop inputs be $D_3 \ldots D_0$ and the outputs be $Q_3 \ldots Q_0$, from left to right. Let the external data inputs be $DATA_3 \ldots DATA_0$, We have four shift register operations: load, shift-left, shift-right, and hold. These will require at least 2 bits of control input to the circuit; let S1 and S0 be the names of two such control bits. Our task is to derive the proper input to each D flip-flop, based on the value of the control inputs S1 and S0. In our design of an enabled D flip-flop, we encountered a related problem, actually a subset of the present problem. There we had two operations, hold and load, that we implemented with one control input, using a multiplexer. We may employ the same technique here, using a four-input multiplexer to provide input to each flip-flop. We may then define codes S1, S0 for our four operations. Using S1 and S0 as mux selector signals, we may infer the proper inputs to the multiplexers. Here are the inputs for a typical bit i of the shift register:

| Clock | S1 | S0 | Result desired | Selected mux position | Required mux input |
|---|---|---|---|---|---|
| ↑ | 0 | 0 | Hold | 0 | Qi |
| ↑ | 0 | 1 | Shift right | 1 | Qi+1 |
| ↑ | 1 | 0 | Shift left | 2 | Qi-1 |
| ↑ | 1 | 1 | Load | 3 | DATAi |

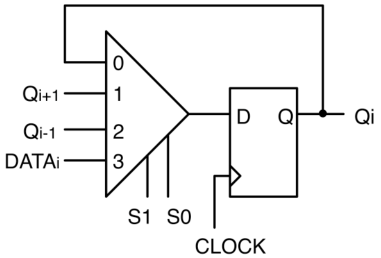


**Figure 4–22.** A typical bit Qi of a general shift register

When designing module symbols, most digital drafting tools allow complete freedom; you should use this freedom to create symbols that clearly reveal module functionality with minimum clutter. For example, we can define a "dot" to mean a module terminal that connects to a common wire. Using this convention, figure 4-23 pictorially represents the operational behavior of a 4-bit shift register. Further, the symbol lends itself to shift registers of arbitrary length.

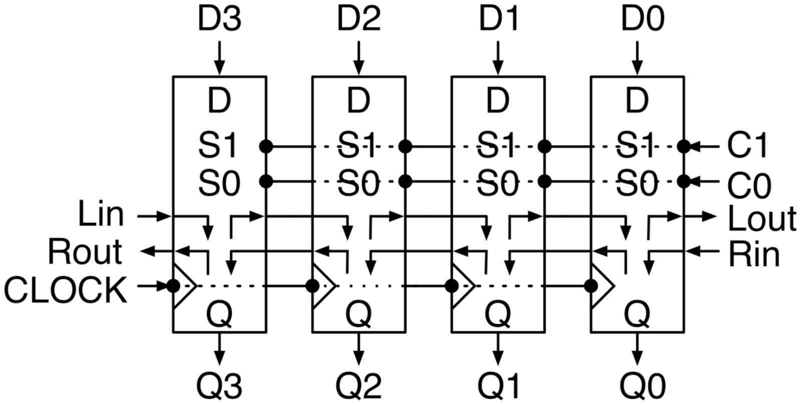


**Figure 4–23.** A 4-bit universal shift register

| C1 | C0 | Operation | Bit-wise operations |
|---|---|---|---|
| 0 | 0 | Hold | |
| 0 | 1 | Right shift | $\text{Lin} \rightarrow Q_3$, $(Q_i \rightarrow Q_{i-1})\ i = 3 \cdots 1$, $\text{Lout} = Q_0$ |
| 1 | 0 | Left shift | $\text{Rout} = Q_3$, $(Qi+1 \leftarrow Qi) i = 0 \cdots 2$, $Q_0 \leftarrow \text{Rin}$ |
| 1 | 1 | Load | $(Di \rightarrow Qi)\ i = 0 \cdots 3$ |

Lin is an external "left-input"; it will enter the *left*-most bit position on a shift *right* command, Rin is an analogous external "right-input"

## MEMORY

Modern integrated circuit memory technology is one of the crowning achievements of our Silicon age and it is hard to communicate what astonishing developments have happened, and continue to happen, with such simple starting materials: sand for Silicon, charcoal for converting sand to Silicon, Aluminum and Copper for wires, and minute amounts of dopants in columns III and IV of the periodic chart. To those who have been in the field a long time each new announcement of larger and faster memory modules is a source of delight and amazement.

You will need to change your mind-set away from gates when considering memories. Memories are fundamentally *area* devices and optimization depends on distributing transistors on a Silicon surface in such a way as to minimize area but still implement desired macro logic behavior.

The field is broad and we will have to break it into subcategories to keep from losing our way during our guided tour of memory technology. We will strive to give you the basic operating principles of the various memory technologies, abstracting away technical details that do not contribute to understanding device fundamentals. Our aim is to equip you with basic understanding; if you wish to delve deeper by reading the technical literature then, indeed, we encourage you to do so.

At the highest level we divide memory into non-volatile and volatile categories. In both technologies the transistors must be placed in rectangular arrays for spatial density reasons, and the technology for reading and writing them will be similar.

## Non-volatile memory

Non-volatile memory is commonly called ROM (Read Only Memory). Once written, it will retain data indefinitely. We are surrounded by non-volatile memory. For example, your cell phone and digital camera retain data when the battery is discharged or even removed—the essence of non-volatile memory. We encourage you to think of all the systems in your computer, household, and car that depend on non-volatile memory. You will be surprised at the tally.

ROM is a bit of a misnomer, somehow data must be entered at least once, perhaps more often, so a better acronym is PROM (Programmable Read Only Memory). Programming depends on underlying transistor structures; we will only consider today’s dominant technology, EEPROM (Electrically Erasable PROM).

A transistor that is turned on or off stores bits in non-volatile memory. How do we turn these transistors on, and keep them on? Fundamentally because the highly purified $SiO_2$ used in integrated circuits is a fantastic insulator. If you completely surround a conductor with ultra pure $SiO_2$ any trapped charge will stay there for years, and you can use this trapped charge to turn a transistor on, and it will remain that way until charge bleeds away.

Review the operation of an NMOS transistor (appendix *). The source and drain are N-type silicon separated by P-type silicon under the gate (a P-channel). A positive voltage on the gate attracts negative charge in the P-channel forming a conductive path between source and drain. An FGFet, (Floating Gate Field Effect Transistor), places a floating gate underneath the control gate, and modulates its behavior by interposing charge that can nullify the normal action of the control gate. Making the control gate positive will then no longer turn on the transistor.

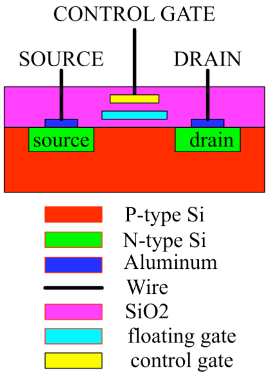
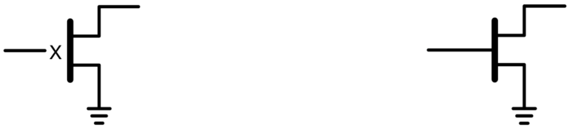


**Figure 4–24.** Cross-section of an FGFet transistor.

A negative charge on the floating gate will attract more positive charge in the channel making it more difficult for the control gate to attract enough negative charge to open the channel, thus blocking the control gate

How you get charge onto or off the floating gate is technology dependent and will be deferred to the literature references; suffice it to say that you can do this, but it takes hundreds of times longer to change the status of a transistor than it does to detect its status. The bits are "slow write—fast read", where "fast read" means nano-seconds. Lets symbolically represent an N-type FGFet as follows:



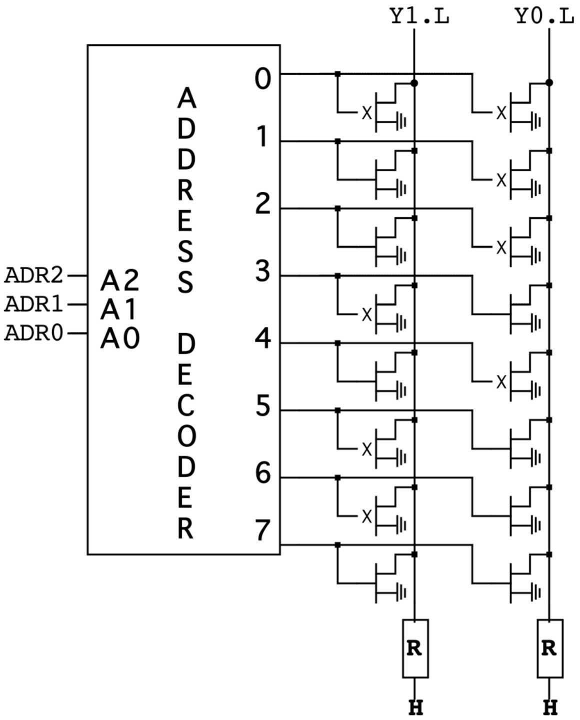
(a)
**Figure 4-25.** Symbol for a blocked gate open-drain FGFet transistor

(b)
Symbol for an unblocked Open-drain FGFet transistor

To show how these blocked transistors can be used to construct a memory you should go back to chapter 2 and review how open-drain transistors implement a "wired-OR". There the rational for using open-drain logic was the assumption that transistors were in separate peripherals sharing a common wire and the only way to simultaneously avoid fights and distribute transistors was to use open-drain devices.

Memory devices need to arrange transistors in rectangular or square arrays to achieve high packing densities and this automatically means distributed logic; open-drain devices are well suited for this. Accessing these transistors requires logic to select a transistor's row and column and some means of detecting its state. To see how this is done, consider a trivially small, 8x2 EEPROM that stores this table: (commercial EEPROM's are much larger, 64k x 8 is typical).

| Row address | ADR2 | ADR1 | ADR0 | Y1,0 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 00 |
| 1 | 0 | 0 | 1 | 10 |
| 2 | 0 | 1 | 0 | 10 |
| 3 | 0 | 1 | 1 | 01 |
| 4 | 1 | 0 | 0 | 10 |
| 5 | 1 | 0 | 1 | 01 |
| 6 | 1 | 1 | 0 | 01 |
| 7 | 1 | 1 | 1 | 11 |



(a) FGfet transistor structure for an 8x2 EEPROM

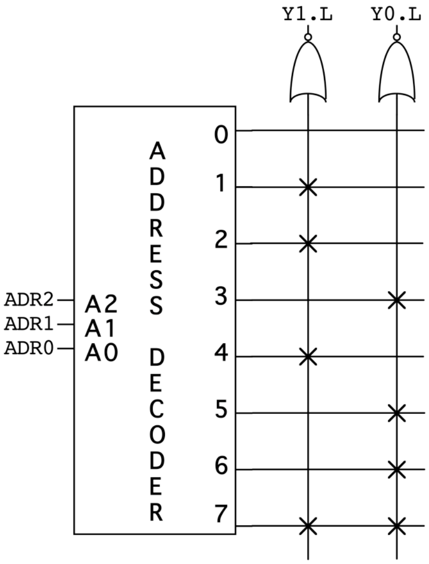
**Figure 4–26**. EEPROM memory structures

Before moving on, consider the distributed nature of Figure 4–26. The resistor, R, is trying to pull a line high, and will do so unless one of the unblocked transistors overcomes the resistive drive and pulls the line low. The unblocked transistors correspond to the 1's in the truth table and thus corresponding outputs, Y1 and Y0, are low true signals.

This is just our old friend, the "wired OR" where we distribute transistors in a rectangular array to achieve high packing density—something every memory designer struggles to achieve. The transistor array is often called the "OR plane" recalling it's logical function and its physical structure. Further, the address decoder geometry will be distributed vertically to match the row spacing of the OR plane.

(We have simplified the EEPROM's wired OR to make it intelligible with the background you have. The FGfet's and array structure are accurate, but sometimes the wired OR resistor is simulated by a technique called precharging—see the references on VLSI design if you wish to delve deeper; otherwise accept the simpler, and logically equivalent, resistor explanation)

Figure 4–26, while accurate, is too cluttered and we need some way to subsume OR plane detail into something that portrays its logic function, reveals the distributed nature of the OR function, while hiding internal details like precharging; in other words we need to find the proper abstraction level. Figure 4–27 shows a common representation.



Equivalent shorthand for Fig. 4–26, the crosses represent unblocked transistors

**Figure 4–27.** EEPROM memory structures

The crosses in Figure 4–27 represent unblocked transistors—the ones that are capable of overcoming the resistor trying to pull the line high—and therefore the ones that will generate the 1's in the target truth table; (do not confuse the crosses in 4–7 with the x's in 4–26).

Now for an important point: viewed as an abstraction, Figure 4–27 is the description of the data we want the OR plane to generate *independent* of the implementation technology. As such, it can be viewed as an *input description* to programming hardware which will charge floating gates, burn fuses or antifuses, or whatever the technology of the moment requires, to implement the requisite data. For this course, you should work at a still higher abstraction level—the data to be burned into each memory location—and not worry too much about the underlying hardware. In practice your data will be in a file that you submit to programming hardware and charge will be injected into the proper transistors to correctly program any PROM regardless of technology.

EEPROM and FLASH are the dominant technologies in most of today's nonvolatile memory applications, basically because charge can be injected or removed at the relatively low voltages supplied by batteries. Flash memory is just a large EEPROM organized as blocks with the ability to erase a block in parallel. EEPROM is reserved for those applications that need individual byte eraseability. They have the additional advantage that they can be reprogrammed inside a system; you don't want to take your cell phone apart every time you update the address book.

## Volatile memory

Any memory whose bits fade away when power is removed is volatile; common varieties go by names such as SRAM, SDRAM, DRAM, DDR2 and too many more to go into each variety in detail. Instead, we will explore the two main types of storage cells: flip-flops for *static* RAM and capacitors for *dynamic* RAM. RAM is an acronym for Random Access Memory, meaning any location in the memories address space can be accessed with the same latency. Uniform latency also applies to ROM's, which are therefore also random access devices, but unfortunately the RAM appellation is, by convention, usually reserved for volatile memory.

Static RAM, (SRAM), maintains data as long as power is applied. SRAM cells use several transistors and tend to consume more power than other memory technologies. This disadvantage is offset by fast access times so for applications like cache memory where speed is more important than power consumption, SRAM's will be the technology of choice.

Dynamic RAMs (DRAM) store bits by the presence or absence of about 2x$10^5$ electrons, stored on tiny capacitors. DRAM memory cells are physically small with just one transistor and its associated capacitor. Unfortunately small size comes with a disadvantage—capacitors are not perfect and charge will leak away over time unless periodically refreshed every few milli-seconds. The cost and complexity of external refresh logic is more than compensated by DRAM's large capacity, which makes it the technology of choice for computer main memory. Small embedded systems are an exception and can usually get by on a mix of ROM's and SRAM's, avoiding the complexity that comes with DRAM technology.

Enough words, lets build things!

**Designing a high speed 1M x 32 cache memory using SRAMs**

Let's choose an industry standard 512kx8, 10ns asynchronous SRAM. Asynchronous means it doesn't need a clock—just supply an address and read data will be available $t_{AA}$ ns later, according to the timing diagram in figure 4–28.

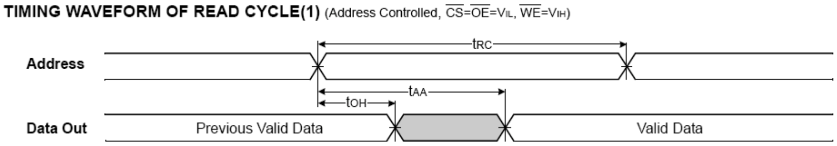

TIMING WAVEFORM OF READ CYCLE(1) (Address Controlled, $\overline{CS}=\overline{OE}=V_{IL}$, $\overline{WE}=V_{IH}$)
Address
tRC
tAA
tOH
Data Out
Previous Valid Data
Valid Data

**Figure 4–28.** Timing diagram for a read cycle of an asynchronous SRAM

After the address changes, data at the new address will appear on the data out lines after $t_{AA}$ ns. You must hold that address for $t_{RC}$ ns.

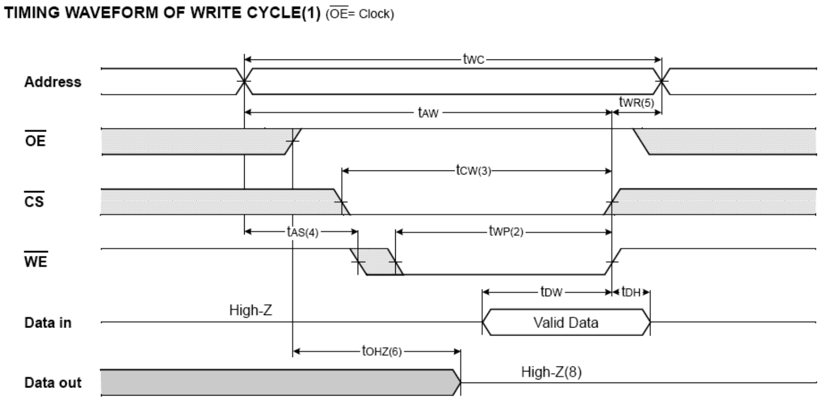
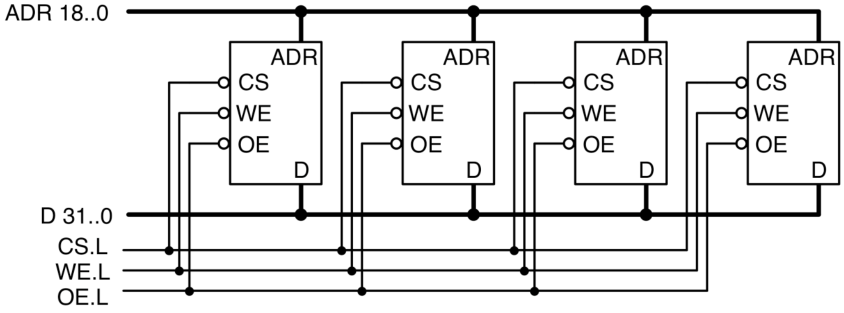


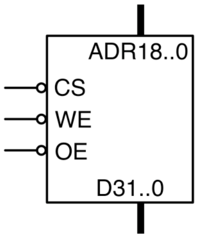
**Figure 4–29.** Timing diagram for a write cycle of an asynchronous SRAM

To write data, present a new address, disable chip outputs by bringing Output Enable (OE) high, select the chip by bringing Chip Select (CS) low, present write data, and then cycle Write Enable (WE). The write takes place on the rising edge of WE.

Since the memory is 32 bits wide we need 4 chips to form one bank of 512k x 32, and two banks to get 1M x 32.



(a) 512k x 32 bank made from four 512k x 8 chips



(b) Equivalent hierarchical symbol for a 512k x 32 static RAM

**Figure 4–30.** Making a 512k x 32 memory bank from four 512k x 8 memory chips

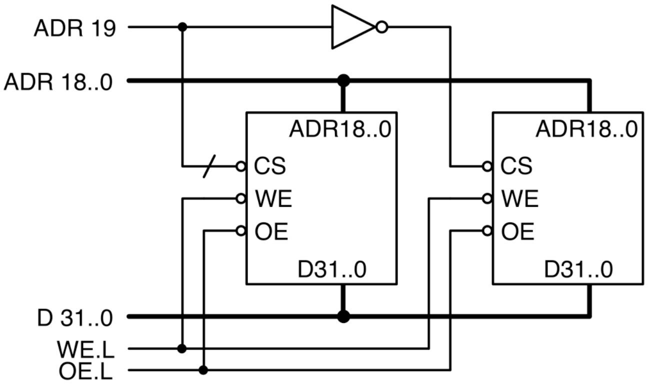


**Figure 4–31.** A 1M x 32 memory made from two 512k x 32 banks of SRAM The heavy lines are a bus - just a bundle of wires

(a) The address bus contains 19 wires, the data bus 32 wires

(b) The high order address bit, A19, selects banks, putting either bank 0 or bank 1 on the tri-state data bus

This is our first encounter with bidirectional signals. Pins and wires are always in short supply, especially in memory chips, and data-in/data-out signals usually share the same wire. How to avoid fights and collisions? Tri-state to the rescue!

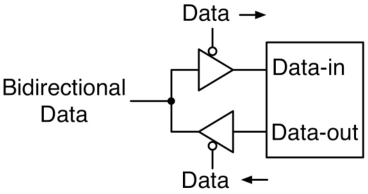


**Figure 4–32.** Using tri-state buffers so one wire can share bidirectional data

**Static vs. Dynamic RAM's** Static RAM cells are essentially small flip-flops, usually using 4 transistors per cell. As flip-flops they have the nice read and write protocols used above. Dynamic RAM cells on the other hand employ just one transistor to access the storage element, a tiny capacitor. As you might expect from the transistor count, dynamic RAMs pack about 4 times as many bits per unit area as SRAMs. Controlling them however is a matter of some delicacy and is deferred to appendix *. Unless you intend to become a professional designer working with large memories, you should stay with static RAMs and enjoy the pleasant interface they present to the designer.

## Programmable Logic

What's logic doing in a chapter on memory? The stunning advances in memory technology are paralleled by similar advances in programmable logic. The field is broad and full treatment will be deferred to the laboratory portion of the course. For now, we will only explore how memory can morph into logic

Look at Figure 4–33 that computes SUM and Cout and assume that you are not allowed to peer inside the black box. From that vantage point what could you infer about the box's internals? Only that wires SUM and Cout have values corresponding to the truth tables for the full adder—not how these values were generated. Gates could calculate the values, or the input values could be used as an address to look up SUM and Cout.

AND, OR, and NOT can be used to calculate any logic function and are therefore called a complete set of primitives, but now we see that Memory could also be a universal logic primitive. From a manufacturing standpoint this universality could be very attractive; build one thing and it will solve all possible logic problems. Well, yes and no.

When viewed as a memory, the horizontal lines emanating from the decoder are viewed as memory addresses. While it's a little odd, you could also think of them as canonical minterms of the address bits. If you replace the input address with logic variables then the oddity immediately disappears, the decoder now produces canonical minterms of those variables. Replace A2 with Cin, A1, with A, and A0 with B and you have a complete set of minterms for a 3-variable truth table. Each horizontal line represents a minterm, for example line 3 = m3 = $\overline{\text{Cin}} \bullet \text{A} \bullet \text{B}$

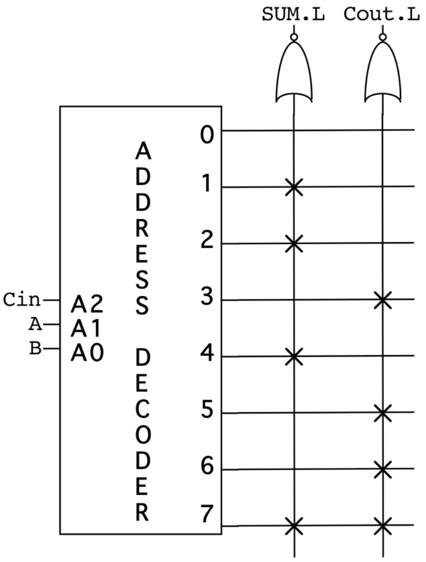


**Figure 4–33.** An 8x2 EEPROM used to generate full adder logic

Viewing decoder outputs as minterms makes perfect sense for generating canonical sum-of-product logic functions for a few variables, but no sense for a 64k x 8 EEPROM; what would you do with 64k minterms? Large PROMS are perforce viewed as data storage devices. Why? Fundamentally, because the address decoder does too much work, it decodes each and every address. Think of using an ROM to generate a 10-bit AND. The OR plane will have 1024 rows, only one of which will generate a "1"—a horrible waste of Silicon.

**Array Logic** An early form of programmable logic struck at the heart of the problem by building a decoder that only generated minterms for the "1's" of the truth table. Decoding is an AND process and we will need to find a way of building distributed AND functionality across the chips area. By the principle of duality, if we can build distributed OR's you would expect to do so for distributed AND's, and that is indeed the case. (We leave this as grist for your mental mill; hint, review open drain logic in Chapter 2).

Minterms use exactly one occurrence of each input variable in either negated or non-negated form. All this is hidden inside the address decoder of a PROM or RAM but now we need to get it out in the open and explicitly look at the inputs to the minterm generator. To be universally applicable we should provide both terms for every variable.

Figure 4–34 shows the standard array logic graphic symbol for generating an input variable, X, in true and negated form, before sending them into the distributed AND minterm generator. The device will buffer the inputs to provide solid drive to the distributed AND structure.

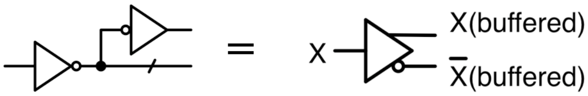


**Figure 4–34.** An array logic input buffer

**Programmable AND Plane Logic**. Using the compact symbol for input buffers, we can build an abstract picture of AND plane logic that portrays the distributed nature of the active logic elements, the minterms generated, and an input description suitable for driving programming hardware. The crosses in Figure 4–35 correspond to underlying transistor or fuse structures that will feed the corresponding minterm to the distributed AND devices.

Programming hardware will place crosses wherever you desire, generating as many, or as few minterms, as you desire, with one caveat; to save Silicon, AND planes will be restricted to some size deemed optimum by the chip designer. Most of your target equations will have just a few minterms, with an occasional one requiring many. If the chip designer makes large AND planes he can handle complex equations but that wastes Silicon for the more common small equations. Techniques exist to bypass this problem but will not be considered here.

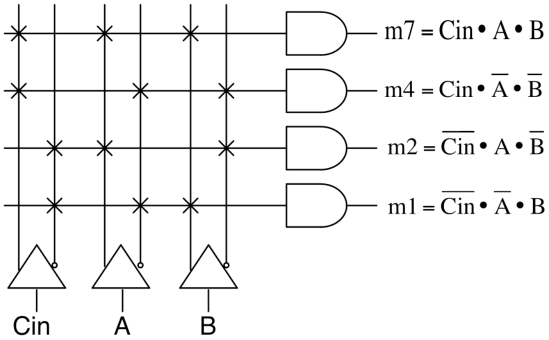

$m7 = Cin \bullet A \bullet B$
$m4 = Cin \bullet \overline{A} \bullet \overline{B}$
$m2 = \overline{Cin} \bullet A \bullet \overline{B}$
$m1 = \overline{Cin} \bullet \overline{A} \bullet B$
Cin
A
B

**Figure 4–35.** An abstract description of a programmable AND plane

To create a sum-of-products expression all we need is to add an OR to sum the minterms as in Figure 4–36.

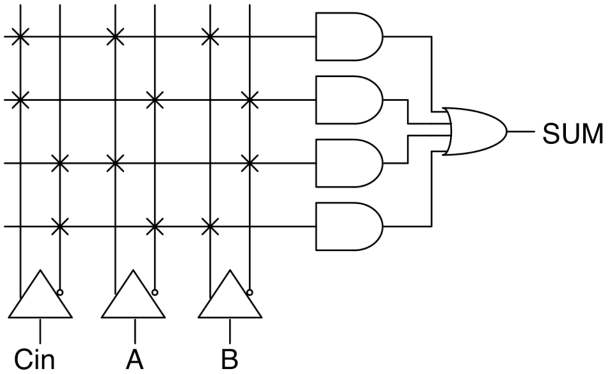


**Figure 4–36.** SUM(Cin,A,B) implemented in programmable array logic

We see that placement of abstract crosses leads immediately to a sum-of-products expression; it is a small step to reverse the process and use minterms as input to the programming hardware.

Commercially available arrays are indeed powerful and are organized as macrocells. One commercial device has 512 macrocells, each macrocell has 36 inputs, one 5-wide OR, and the ability to take the output of one macrocell and feed it into an adjacent macrocell, effectively expanding the OR to any width. In addition, a macrocell will usually have a flip-flop fed from the programmable array making it a truly general-purpose element. For a cost in the dollar range you get an effective gate count of many 10's of thousands, and 512 flip-flops, (although in real world devices only a tiny fraction will wind up being used). Amazing!

## THE METASTABILITY PROBLEM

We began this chapter with a discussion of hazards, a nuisance created by the characteristics of physical devices used to implement logical concepts. In Chapter 5 you will encounter other design pitfalls rooted in physical behavior—pitfalls that arise through the interactions of several components of a design. There remains to discuss the most alarming physical problem of all—metastability. We will alert you to the problem and give some advice, but you should look to appendix * for a more extensive treatment of this topic.

Digital devices are fundamentally analog devices that behave digitally only when stringent rules of operation are obeyed. Sequential devices contain amplifiers (gates) and feedback loops to achieve their storage properties. In addition to establishing proper voltage levels at the inputs, to assure proper operation of a sequential device you must adhere to the setup times, hold times, and other timing specified in the data sheets. When the operational requirements are met, the device's outputs will be proper digital voltage levels, and changes in the level of the output will occur quickly and cleanly. Except during the rapid period of transition, the circuit remains in one of its stable states. You have seen that there are difficulties associated with the RS flip-flop when one tries to move from the R = S = T input configuration to the hold configuration, in which R = S = F. The difficulties arose from the attempt to change both inputs simultaneously. As long as no more than one input is changing at a time, the sequential circuit performs well, but if the voltage level of more than one input is allowed to change at nearly the same time, the circuit is being required to perform outside the framework of design for digital operation and the result may be unpleasant. For the proper operation of clocked circuits, the setup and hold times require that certain inputs must not change too near the time that the clock signal is changing.

Violation of the timing requirements of a sequential circuit may throw the circuit into *a metastable state,* during which the outputs may hold improper or nondigital values for an unspecified duration. In one form of metastability, the output voltage lingers for an indefinite period in the transition region between digital voltage levels, before it eventually resolves into a stable value. In another form of metastability, the output appears to be a proper digital value, but after an unpredictable interval switches to another value. Metastability can be disastrous. In synchronous design, we sidestep the problem by never changing the inputs in the vicinity of the clock. As you will see, this allows vast simplification of the design of complex circuits. But every circuit is at some point exposed to external reality—other circuits with different clocks, unclocked or nondigital devices, and human operators, for instance. Signals from such sources are not tied to our clock and may change at any time during our clock cycle. Therefore, although we can simplify our design by using good practices, no amount of digital or analog wizardry will eliminate the problem of metastability. However, by proper design or choice of components, we may lower the probability of finding the circuit in a metastable state to a satisfactory level. In appendix *, we discuss metastability in more detail and offer guidelines for dealing with the problem.

## CONCLUSION

You have completed Part I of this book, in which we have explored the fundamental tools underlying digital design. From basic combinational circuits we have developed a set of building blocks that range from simple logic gates to complex ALUs, from flip-flops to large memories. Now you are ready to begin the exciting activity of digital design. Part II introduces you to this process.

## READINGS AND SOURCES

BLAKESLEE, THOMAS *R., Digital Design with Standard MSI and LSI,* 2nd ed. John Wiley & Sons, New York, 1979. Sound design practices.

DIETMEYER, DONALD L., *Logic Design of Digital Systems,* 2nd ed. Allyn & Bacon, Boston, 1978. Chapter 12: hazards. Chapter 13: traditional asynchronous design.

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

FLETCHER, WILLIAM I., *An Engineering Approach to Digital Design.* Prentice-Hall, Englewood Cliffs, N.J., 1980. Chapter 5 contains a good discussion of flip-flops.

HILL, FREDERICK J., and GERALD R. PETERSON, *Digital Logic and Microprocessors.* John Wiley & Sons, New York, 1984.

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

HwANG, KAI, *Computer Arithmetic—Principles, Architecture, and Design.* John Wiley & Sons, New York, 1979.

KLINGMAN, EDWIN *E., Microprocessor System Design.* Vol. *2, Microcoding, Array Logic, and Architectural Design.* Prentice-Hall, Englewood Cliffs, N.J., 1982. Bit slices and programmable logic.

MANO, M. MORRIS, *Digital Design.* Prentice-Hall, Englewood Cliffs, N.J., 1984.

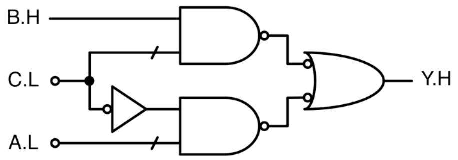
MICK, JOHN, and JAMES BRICK, *Bit-Slice Microprocessor Design.* McGraw-Hill Book Co., New York, 1980. A collection of design notes for the Advanced Micro Devices 2900 bit-slice family. This book is useful far beyond the Am2900 chips.

MYERS, GLENFORD J., *Digital System Design with LSI Bit-Slice Logic.* John Wiley & Sons, New York, 1980.

WIATROWSKI, CLAUDE A., and CHARLES H. House, *Logic Circuits and Microcomputer Systems,* McGraw-Hill Book Co., New York, 1980.

## EXERCISES

**4-1.** Show that the following combinational circuit contains a hazard.

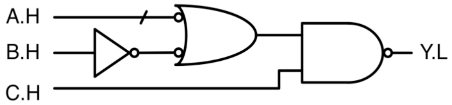


(a) Write the logic equation corresponding to the circuit, and draw a K-map with circles corresponding to the circuit.

(b) Most of the time our design techniques will nullify the bad effects of hazards; nevertheless, suppose that you must eliminate the above hazard from the circuit. Starting with the K-map you drew for part (a), produce a hazard-free map by making certain that adjacent 1's share at least one circle. Write the logic equation and draw the hazard-free circuit

(c) Prove, by using a timing diagram, that your new circuit is free of hazards.

**4-2.** Assume that each combinational circuit element has a propagation delay of $t_p$. What is the total (worst-case) propagation delay in the following circuit?



**4-3.** In Fig. 3-5, the circuit for the enabled multiplexer imposes the enabling operation on each of the initial AND gates, forcing them to have three inputs. Suggest why, in Fig. 3-5, the enabling operation was not designed as a single final AND gate with only two inputs.

**4-4.** A circuit consisting of a closed loop of an odd number of inverters (greater than one) can function as an oscillator. Assume that the propagation delay through an inverter is 10 nanoseconds.

(a) With a timing diagram, show the oscillatory behavior of a loop of three inverters.

(b) The oscillator consisting of a loop with just a single inverter is not stable. Speculate about why this circuit is unsatisfactory.

**4-5.** What is feedback in digital design? Draw a gate circuit that exhibits feedback with memory.

**4-6.** Why are combinational methods inadequate to deal with sequential circuits?

**4-7.** Explain "1's catching." Why is this behavior usually a disadvantage in digital design?

**4-8.** Explain the terms *asynchronous* and *synchronous.*

**4-9.** Show that the asynchronous RS flip-flop has two stable states.

**4-10.** Why do we usually avoid asynchronous flip-flops in digital design?

**4-11.** What is switch debouncing? Why can we usually not use a mechanical switch signal directly in a digital design? Draw a switch-debouncing circuit.

**4-12.** Using a timing diagram, analyze the behavior of the switch debouncer shown in Fig. 4–8a or 4–8b.

**4-13.** Assume that two (noisy) mechanical switches generate the DATA and HOLD signals for the latch in Fig. 4–4. Is there any sequence of switch closings and openings that would yield a clean output signal at Y?

**4-14.** The RS flip-flop exhibits anomalous output behavior if both *R* and S are true.

(a) What is the anomaly?

(b) Does the anomaly occur in outputs X and Q of Fig. 4–6?

(c) In Fig. 4–6, assume that R = S = T. What is the value of Q if both signals become false, but *R* becomes false slightly before S?

(d) Under similar conditions, what value does Q assume after precisely simultaneous T→F transitions of R and S?

**4-15.** What is an edge-driven flip-flop? Why is it desirable? What is the defect in the master-slave flip-flop? What is a pure edge-driven flip-flop? What kind of flip-flops do we use in digital design?

**4-16.** Consider an edge-driven JK flip-flop with the direct set input and the K input asserted (true), and the direct clear input and the J input negated (false). What will be the flip-flop's output shortly after the next active clock edge arrives?

**4-17.** The text describes three cases in which the JK flip-flop may be used to store a bit. Two of these cases are (a) clearing, followed by later setting if the data bit is true; (b) setting, followed by later clearing if the data bit is false. Verify the text's rules for implementing these two cases.

**4-18.** Do you want to observe metastability in action? Use a simulator to create an asynchronous flip-flop. Start with both R and S True and simultaneously make them False. What behavior do you observe on the Q and $\overline{Q}$ outputs? How would a real RS behave?

**4-19.** What is the difference between the names used for inputs and outputs inside a mixed-logic circuit symbol and the names appearing outside the symbol?

**4-20.** There are four possible transitions, Qn to Qn+1, for a clocked flip-flop output: 0→0, 0→1, 1→0, and 1→1. These transitions are given the names t0, $t\alpha$, $t\beta$, and t1, respectively. Consider the ways in which we can make a D flip-flop and a JK flip-flop execute each of these transitions. Fill in the missing elements in the following table:

|  | D flip flop |  | JK flip flop |  |  |
|---|---|---|---|---|---|
| Transition | Qn | Dn | Qn | Jn | Kn |
| t0 | 0 | 0 | 0 | 0 | X |
| $t\alpha$ |  |  |  |  |  |
| $t\beta$ |  |  |  |  |  |
| t1 |  |  |  |  |  |

[In each case there will be two ways that the JK flip-flop can execute the transition. For instance, the 0→0 (t0) transition occurs by *clearing* the flip-flop to 0 (having J = 0, K = 1), or by *holding* the previous 0 (having J = 0, K = 0). These cases give rise to the X (don't-care) entry in the table.]

**4-21.** Compare the asynchronous RS flip-flop and the synchronous JK, D, and enabled D flip-flops as to their best uses in digital design.

**4-22.** Two types of clocked flip-flop behavior that are occasionally useful are the T (toggle) and the SOC (set overrides clear) flip-flop modes. A toggle flip-flop changes its output Q only when its input TOG is true at the time of the clock edge. A SOC flip-flop behaves like a clocked RS flip-flop except that it ignores the value of input R whenever input *S* is true. Write excitation tables defining each type.

**4-23.** By means of external gates, convert a JK flip-flop into a type T (toggle) and a type SOC (set overrides clear) flip-flop.

**4-24.** By analogy with Fig. 4–12, construct a type T (toggle) flip-flop from a D flip-flop

**4-25.** What is a register? How does it differ from a flip-flop?

**4-26.** Construct synchronous modulo-2, modulo-4, and modulo-8 counters using:

(a) D flip-flops.
(b) JK flip-flops.
(c) T (toggle) flip-flops.

**4-27.** Repeat Exercise 4–26 with ripple counters instead of synchronous counters.

**4-28.** For a 4-bit ripple counter, demonstrate how the output ripple can produce hazards in circuits that receive the outputs.

**4-29.** Use counters in your simulator library to build a divide-by-24 circuit. The output of your circuit should be true during 1 of every 24 clock periods. This and similar circuits are *frequency dividers.*

**4-30.** There are many special counting sequences that are of some interest in digital design. The *binary counter* produces the sequence of binary integers. The *gray code counter* produces a sequence in which exactly one bit changes in moving from one element of the sequence to the next. For a 2-bit counter, the gray code is 00, 01, 11, 10. (Where have you seen this sequence in this book?) Build a series of 2-bit gray code counters using the following approaches:

(a) Use logic gates to compute the inputs to D flip-flops.
(b) Use multiplexers to look up the inputs to D flip-flops.
(c) Use logic gates to compute the inputs to JK flip-flops.
(d) Use multiplexers to look up the inputs to JK flip-flops.

**4-31.** The *moebius counter* produces another special sequence. The algorithm for *N* bits numbered $C_N \ldots C_1$ is

$$Ck \leftarrow Ck{+}1 \quad \text{when } k = N-1 \ldots 1$$

$$C_N \leftarrow \overline{C1}$$

(a) Design a 4-bit moebius counter, using JK flip-flops as the storage elements.
(b) Design a 4-bit moebius counter using a shift register as the basic storage element.
(c) How many elements are in an N-bit moebius sequence that begins with 0? Determine the answer empirically.

**4-32.** Use your simulator to make compact symbol for the universal 1-bit shift register and store it in your library. Use eight of these shift registers to implement an 8-bit shift registers and verify correct behavior for: Load, Right-shift, Left-shift, and Hold.

**4-33.** Modify the shift register of 4–32c to include a fifth mode of operation. This new mode will preserve the most significant (leftmost) bit during a right shift; in other words, after the shift, the two leftmost bits will be the same. This is called an *arithmetic right* shift—useful in computing with signed two's-complement numbers.

**4-34.** Describe the principal characteristics of the RAM, ROM, PROM, and EPROM.

**4-35.** How do static and dynamic RAMs differ? What advantages do dynamic RAMs offer? What disadvantages?

**4-36.** Assume you want to calculate the following functions, X,Y,Z by array logic as in Figure 4–27c. Show the Design PROMs that realize the following sets of logic functions:

$$X = A \bullet B \bullet C + A \bullet \overline{B} \bullet C + \overline{A} \bullet B \bullet C + \overline{A} \bullet \overline{B} \bullet \overline{C}$$

(a) $$Y = A \bullet B \bullet \overline{C} + A \bullet \overline{B} \bullet C + \overline{A} \bullet \overline{B} \bullet C$$

$$Z = A \bullet B \bullet \overline{C} + \overline{A} \bullet B \bullet \overline{C} + \overline{A} \bullet \overline{B} \bullet \overline{C}$$

(b) $$X = A \bullet B + A \bullet \overline{C} + \overline{A} \bullet C$$

$$Y = \overline{A} \bullet B + B \bullet C + \overline{A} \bullet B \bullet \overline{C}$$

$$Z = B \bullet C + \overline{B} \bullet \overline{C} + \overline{A}$$

**4-37.** Design PLAs that realize the sets of logic functions in Exercise 4–48.

**4-38.** The following prescription will convert an n-bit binary number into an n-bit gray code (n is the most significant bit):

$$\text{Gray}_n = \text{binary}_n$$

$$\text{gray}_k = \text{binary}_k \oplus \text{binary}_{k+1} \quad (k = n - 1, \dots, 2, 1)$$

(a) Tabulate the 5-bit binary and 5-bit gray codes.
(b) Design a PROM that converts 5-bit binary numbers into 5-bit gray codes.

**4-39.** When k = 1, 2, n, bit k of an n-bit binary number is equal to the XOR of the corresponding gray code bits from k through n (n is the most significant bit). That is

$$\text{binary}_k = \text{gray}_k \oplus \text{gray}_{k+1} \oplus \cdots \oplus \text{gray}_n$$

(a) Tabulate the 4-bit gray code and the 4-bit binary code.
(b) Design a PLA that converts a 4-bit gray code into a binary number.