

associated with the relevant input combinations as question marks in the map (Figure 10-8).

The ? characters indicate *don't care* states, which can be considered to represent either 0 or 1 values at the designer's discretion. In the example shown in Figure 10-8, we have no interest in the ? character at $a = 0, b = 0, c = 1, d = 0$ or the ? character at $a = 0, b = 1, c = 1, d = 1$, because neither of these can be used to form a larger group. However, if we decide that the other three ? characters are going to represent 1 values, then they can be used to form larger groups, which allows us to minimize the function to a greater degree than would otherwise be possible.

It should be noted that many electronics references use X characters to represent don't care states. Unfortunately, this may lead to confusion as design tools such as logic simulators use X characters to represent *don't know* states. Unless otherwise indicated, this book will use ? and X to represent *don't care* and *don't know* states, respectively.

	00	01	11	10
00	1	?	1	1
01				
11		?	?	?
10	?		1	1

$$y = (\bar{c} \& \bar{d}) \mid (a \& c)$$

Figure 10-8. Karnaugh Map for an incompletely specified function

Populating Maps Using 0s Versus 1s

When we were extracting Boolean equations from truth tables in the previous chapter, we noted that in the case of a function whose output is logic 1 fewer times than it is logic 0, it is generally easier to extract a *sum-of-products* equation. Similarly, if the output is logic 0 fewer times than it is logic 1, it is generally easier to extract a *product-of-sums* equation.

The same thing applies to a Karnaugh Map. If the output is logic 1 fewer times than it is logic 0, then it's probably going to be a lot easier to populate the map using logic 1's. Alternatively, if the output is logic 0 fewer times than it is logic 1, then populating the map using logic 0s may not be a bad idea.

When a Karnaugh Map is populated using the 1s assigned to the truth table's output, the resulting Boolean expression is extracted from the map in sum-of-products form. By comparison, if the Karnaugh Map is populated using the 0s assigned to the truth table's output, then the groupings of 0s are used to generate expressions in product-of-sums form (Figure 10-9).

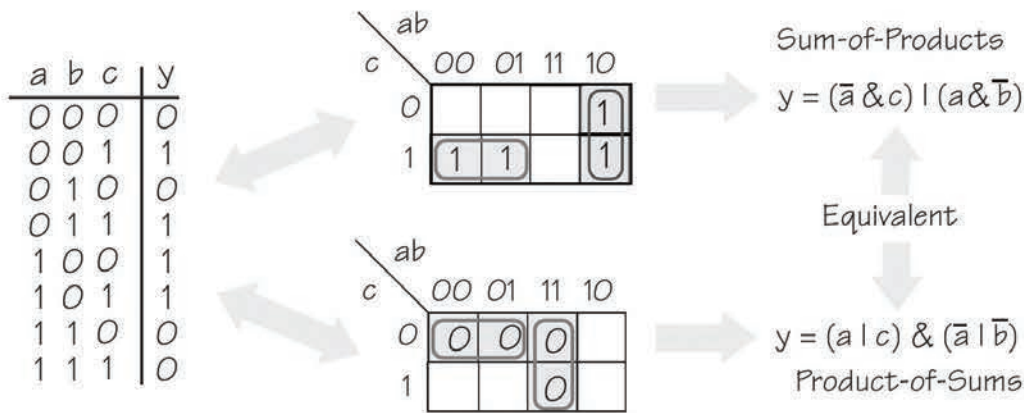


Figure 10-9. Populating Karnaugh Maps with 0s versus 1s

Although the sum-of-products and product-of-sums expressions appear to be somewhat different, they do produce identical results. The expressions can be shown to be equivalent using algebraic means, or by constructing truth tables for each expression and comparing the outputs.

Karnaugh Maps are most often used to represent 3-input and 4-input functions. It is possible to create similar maps for 5-input and 6-input functions, but these maps can quickly become unwieldy and difficult to use. Thus, the Karnaugh technique is generally not considered to have any application for functions with more than six inputs.

Using Primitive Logic Functions to Build More Complex Functions

The primitive functions NOT, AND, OR, NAND, NOR, XOR, and XNOR can be connected together to build more complex functions which may, in turn, be used as building blocks in yet more sophisticated systems. The examples introduced in this chapter were selected because they occur commonly in designs, are relatively simple to understand, and will prove useful in later discussions.

Scalar versus Vector Notation

A single signal carrying one bit of binary data is known as a *scalar* entity. A set of signals carrying similar data can be gathered together into a group known as a *vector* (see also the glossary definition of “vector”).

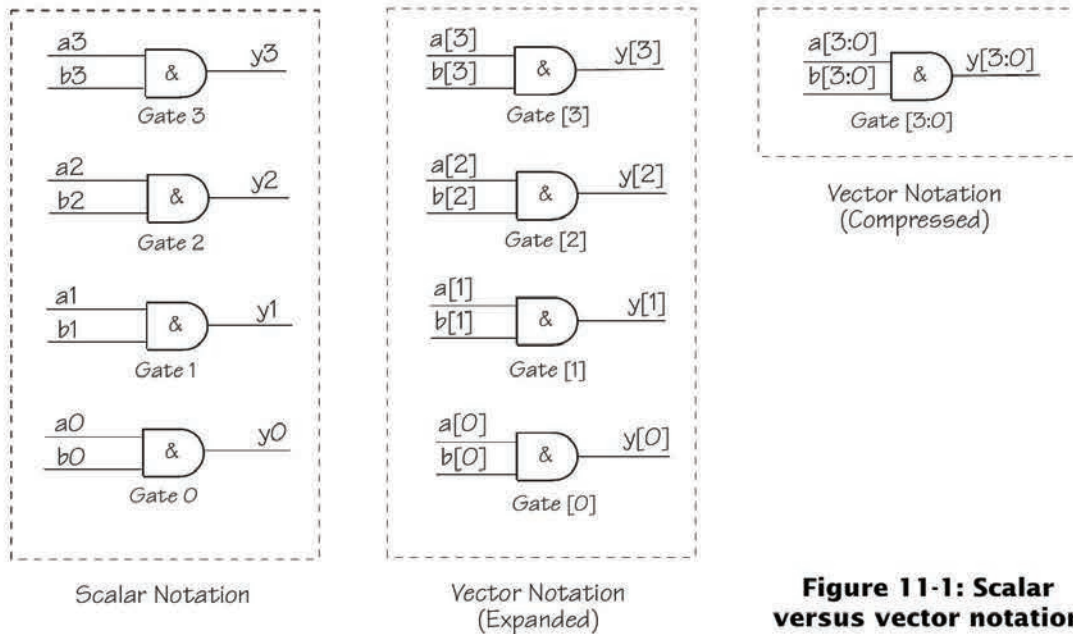


Figure 11-1: Scalar versus vector notation

Consider the circuit fragments shown in Figure 11-1. Each of these fragments represents four 2-input AND gates. In the case of the scalar notation, each signal is assigned a unique name: for example, a_3 , a_2 , a_1 , and a_0 . By comparison, when using vector notation, a single name is applied to a group of signals, and individual signals within the group are referenced by means of an index: for example, $a[3]$, $a[2]$, $a[1]$, and $a[0]$. This means that if we were to see a schematic (circuit) diagram containing two signals called a_3 and $a[3]$, we would understand this to represent two completely different signals (the former being a scalar named “ a_3 ” and the latter being the third element of a vector named “ a ”).

A key advantage of vector notation is that it allows all of the signals comprising the vector to be easily referenced in a single statement: for example, $a[3:0]$, $b[3:0]$, and $y[3:0]$. Thus, vector notation can be used to reduce the size and complexity of a circuit diagram while at the same time increasing its clarity.

Equality Comparators

In some designs it may be necessary to compare two sets of binary values to see if they contain the same data. Consider a function used to compare two 4-bit vectors: $a[3:0]$ and $b[3:0]$. A scalar output called *equal* is to be set to logic 1 if each bit in $a[3:0]$ is equal to its corresponding bit in $b[3:0]$: that is, the vectors are equal if $a[3] = b[3]$, $a[2] = b[2]$, $a[1] = b[1]$, and $a[0] = b[0]$ (Figure 11-2).

The values on $a[3]$ and $b[3]$ are compared using a 2-input XNOR gate. As we know from Chapters 5 and 6, if the values on its inputs are the same (both 0s or both 1s), the output of an XNOR will be 1, but if the values on its inputs are different, the output will be 0. Similar comparisons are performed between the other inputs: $a[2]$ with $b[2]$, $a[1]$ with $b[1]$, and $a[0]$ with $b[0]$. The final AND gate is used to gather the results of the individual comparisons. If all the inputs to the AND gate are 1, the two vectors are the same, and the output of the AND gate will be 1. Correspondingly, if any of the inputs to the AND gate are 0, the two vectors are different, and the output of the AND gate will be 0.

Note that a similar result could have been obtained by replacing the XNORs with XORs and the AND with a NOR, and that either of these implementations could be easily extended to accommodate input vectors of greater width.

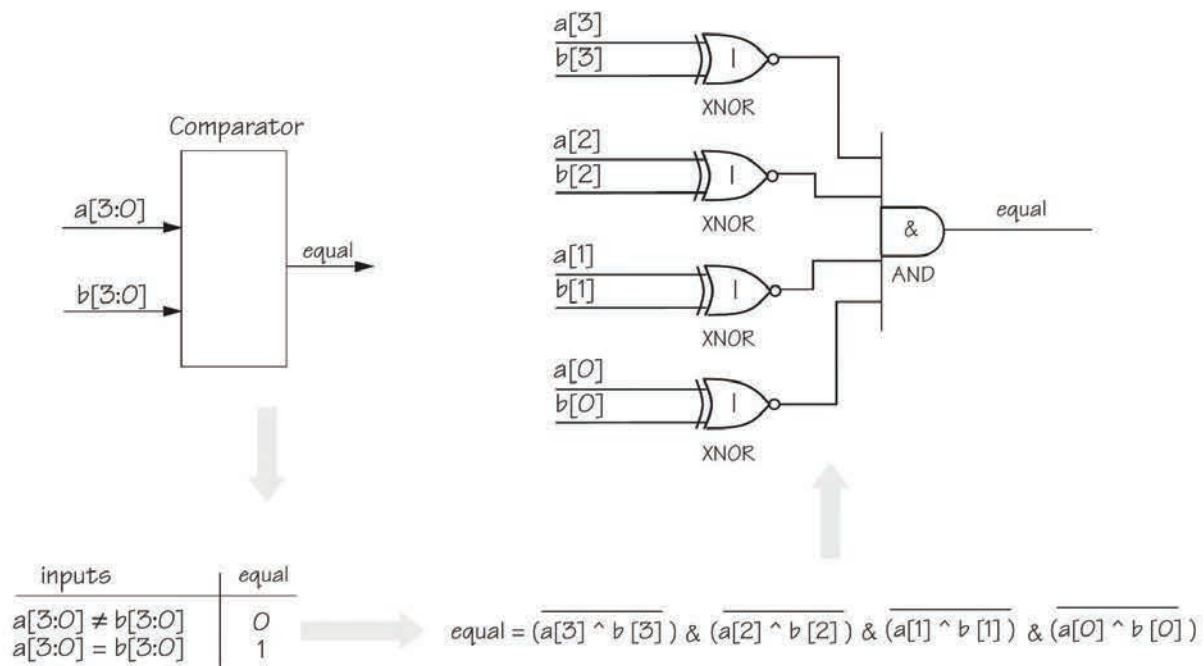


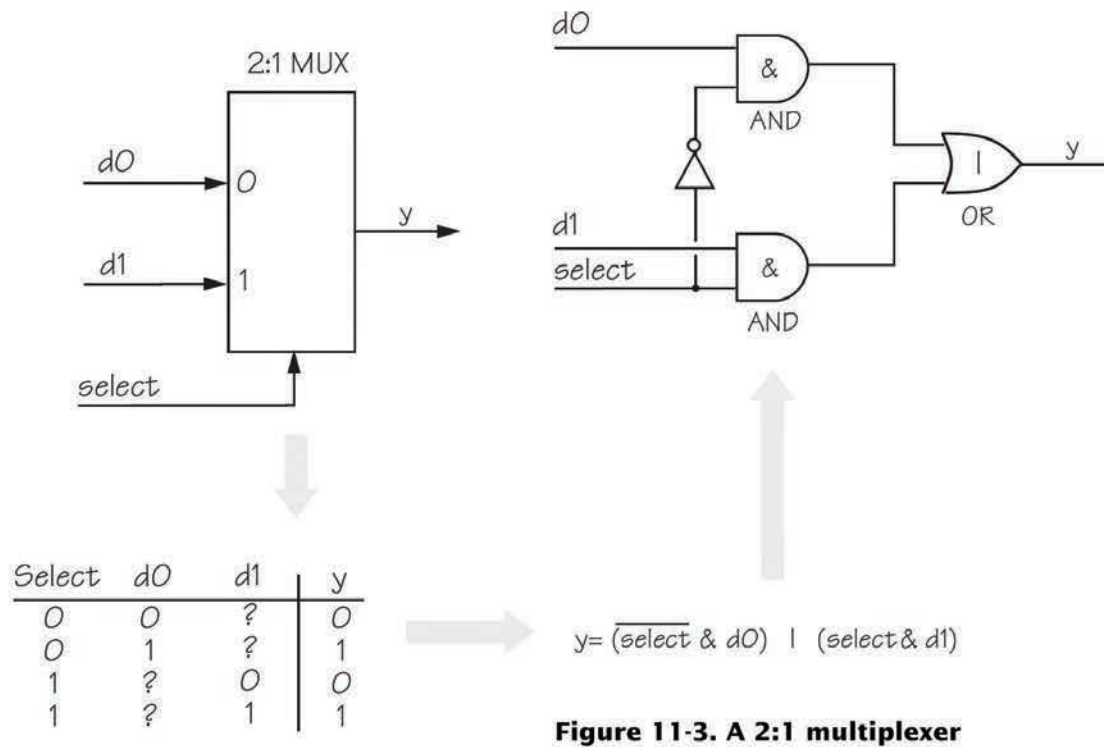
Figure 11-2. Equality comparator

Multiplexers

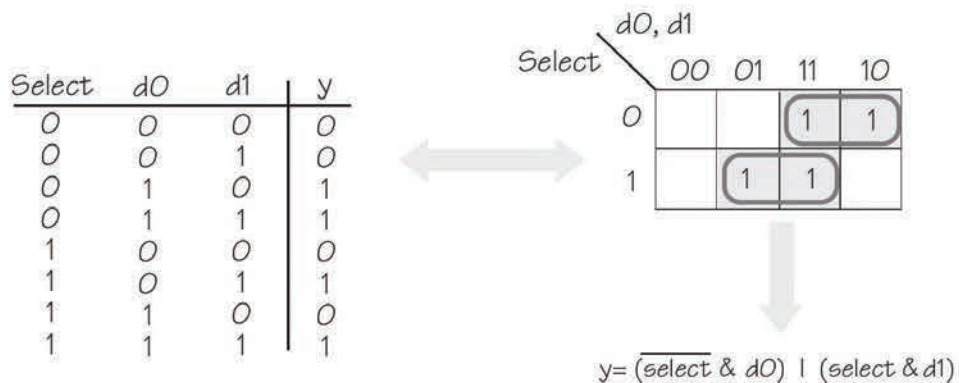
A multiplexer uses a binary value, or *address*, to select between a number of inputs and to convey the data from the selected input to the output. For example, consider a 2:1 (“two-to-one”) multiplexer (Figure 11-3).

The 0 and 1 annotations on the multiplexer symbol represent the possible values of the *select* input and are used to indicate which data input will be selected.

The ? characters in the truth table indicate *don't care* states. When the *select* input is presented with a 0, the output from the function depends only on the value of the $d0$ data input, and we don't care about the value on $d1$. Similarly, when *select* is presented with a 1, the output from the function depends only on the value of the $d1$ data input, and we don't care about the value on $d0$. The use of don't care states reduces the size of the truth table, better represents the operation of this particular function, and simplifies the extraction of the sum-of-products expression because the don't cares are ignored.

**Figure 11-3. A 2:1 multiplexer**

An identical result could have been achieved using a full truth table combined with a Karnaugh Map minimization (Figure 11-4).¹

**Figure 11-4. Deriving the 2:1 multiplexer equation by means of a Karnaugh Map**

¹ Karnaugh Map minimization techniques were introduced in Chapter 10.

Larger multiplexers are also common in designs: for example, 4:1 multiplexers with four data inputs feeding one output and 8:1 multiplexers with eight data inputs feeding one output. In the case of a 4:1 multiplexer, we will require two select inputs to choose between the four data inputs (using binary patterns of 00, 01, 10, and 11). Similarly, in the case of an 8:1 multiplexer, we will require three select inputs to choose between the eight data inputs (using binary patterns of 000, 001, 010, 011, 100, 101, 110, and 111).

Decoders

A decoder uses a binary value, or *address*, to select between a number of outputs and to assert the selected output by placing it in its active state. For example, consider a 2:4 (“two-to-four”) decoder (Figure 11-5).

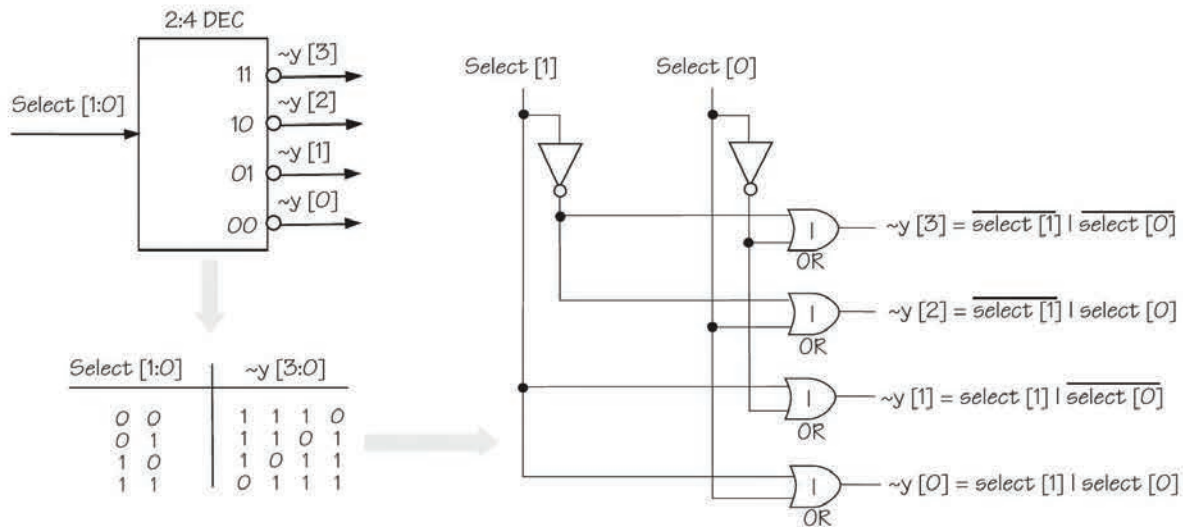


Figure 11-5. A 2:4 decoder with active-low outputs

The 00, 01, 10 and 11 annotations on the decoder symbol represent the possible values that can be applied to the $select[1:0]$ inputs and are used to indicate which output will be asserted.

The truth table shows that when a particular output is selected, it is asserted to a 0, and when that output is not selected, it returns to a 1. Because the outputs are asserted to 0s, this device is said to have *active-low* outputs.

An *active-low* signal is one whose active state is considered to be logic 0.² The active-low nature of this particular function is also indicated by the bobbles (small circles) associated with the symbol's outputs and by the tilde (“~”)³ characters in the names of the output signals. Additionally, from our discussions in Chapters 9 and 10, we know that as each output is 0 for only one input combination, it is simpler to extract the equations in product-of-sums form.

Larger decoders are also commonly used in designs: for example, 3:8 decoders with three select inputs and eight outputs, 4:16 decoders with four select inputs and sixteen outputs, etc.

Tri-State Functions

There is a special category of gates called tri-state functions whose outputs can adopt three states: 0, 1, and Z. Let's first consider a simple tri-state buffer (Figure 11-6).

The tri-state buffer's symbol is based on a standard buffer with an additional control input known as the *enable*. The active-low nature of this particular

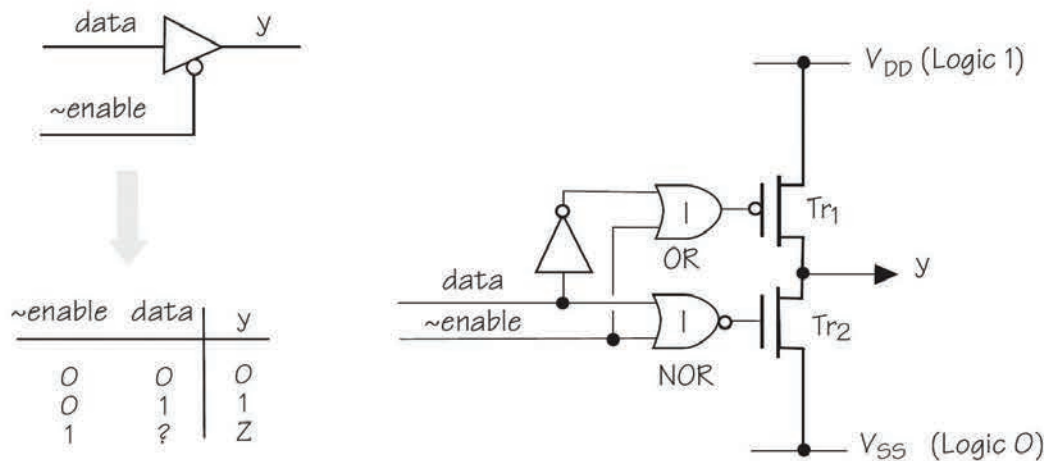


Figure 11-6. Tri-state buffer with active-low enable

² Similar functions can be created with *active-high* outputs, which means that when an output is selected it is asserted to a logic 1.

³ The tilde ‘~’ characters prefixing the output names $\sim y[3]$, $\sim y[2]$, $\sim y[1]$, and $\sim y[0]$ are used to indicate that these signals are active-low. The use of tilde characters is discussed in more detail in Appendix A.

function's enable is indicated by the bobble associated with this input on the symbol and by the tilde character in its name, $\sim enable$. (Similar functions with active-high enables are also commonly used in designs.)

The Z character in the truth table represents a state known as *high-impedance*, in which the gate is not driving either of the standard 0 or 1 values. In fact, in the high-impedance state the gate is effectively disconnected from its output.

Although Boolean algebra is not well equipped to represent the Z state, the implementation of the tri-state buffer is relatively easy to understand. When the $\sim enable$ input is presented with a 1 (its inactive state), the output of the OR gate is forced to 1 and the output of the NOR gate is forced to 0, thereby turning both the Tr_1 and Tr_2 transistors OFF, respectively. With both transistors turned OFF, the output y is disconnected from V_{DD} and V_{SS} , and is therefore in the high-impedance state.

When the $\sim enable$ input is presented with a 0 (its active state), the outputs of the OR and NOR gates are determined by the value on the *data* input. The circuit is arranged so that only one of the Tr_1 and Tr_2 transistors can be ON at any particular time. If the *data* input is presented with a 1, transistor Tr_1 is turned ON, thereby connecting output y to V_{DD} (which equates to logic 1). By comparison, if the *data* input is presented with a 0, transistor Tr_2 is turned ON, thereby connecting output y to V_{SS} (which equates to logic 0).

Tri-state buffers can be used in conjunction with additional control logic to allow the outputs of multiple devices to drive a common signal. For example, consider the simple circuit shown in

Figure 11-7.

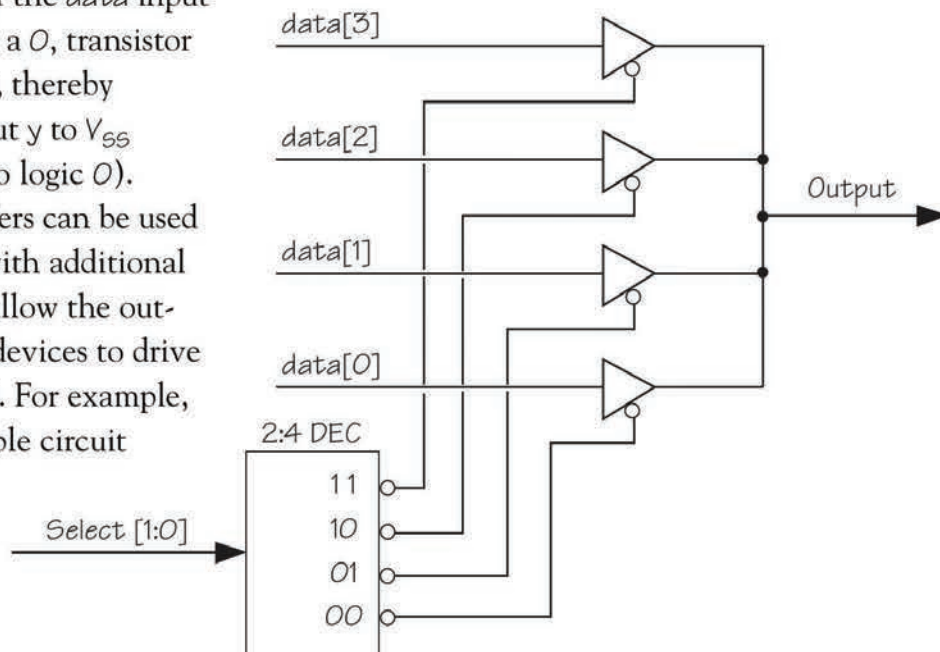


Figure 11-7. Multiple devices driving a common signal

The use of a 2:4 decoder with active-low outputs ensures that only one of the tri-state buffers is enabled at any time. The enabled buffer will propagate the data on its input to the common output, while the remaining buffers will be forced to their tri-state condition.

With hindsight it now becomes obvious that the standard primitive gates (AND, OR, NAND, NOR, etc.) depend on internal Z states to function (when any transistor is turned OFF, its output effectively goes to a Z state). However, the standard primitive gates are constructed in such a way that at least one of the transistors connected to the output is turned ON, which means that the output of a standard gate is always driving either 0 or 1.

Combinational versus Sequential Functions

Logic functions are categorized as being either *combinational* (sometimes referred to as *combinatorial*) or *sequential*. In the case of a *combinational* function, the logic values on that function's outputs are directly related to the current *combination* of values on its inputs. All of the previous example functions have been of this type.

In the case of a sequential function, the logic values on that function's outputs depend not only on its current input values, but also on previous input values. That is, the output values depend on a *sequence* of input values. Because sequential functions remember previous input values, they are also referred to as *memory elements*.

RS Latches

One of the simpler sequential functions is that of an RS latch, which can be implemented using two NOR gates connected in a back-to-back configuration (Figure 11-8). In this NOR implementation, both *reset* and *set* inputs are active-high as indicated by the lack of bobbles associated with these inputs on the symbol. The names of these inputs reflect the effect they have on the q output; when *reset* is active q is reset to 0, and when *set* is active q is set to 1.

The q and $\sim q$ outputs are known as the *true* and *complementary* outputs, respectively.⁴ In the latch's normal mode of operation, the value on $\sim q$ is the

⁴ In this case, the tilde '~' character prefixing the output name $\sim q$ is used to indicate that this signal is a complementary output. Once again, the use of tilde characters is discussed in detail in Appendix A.

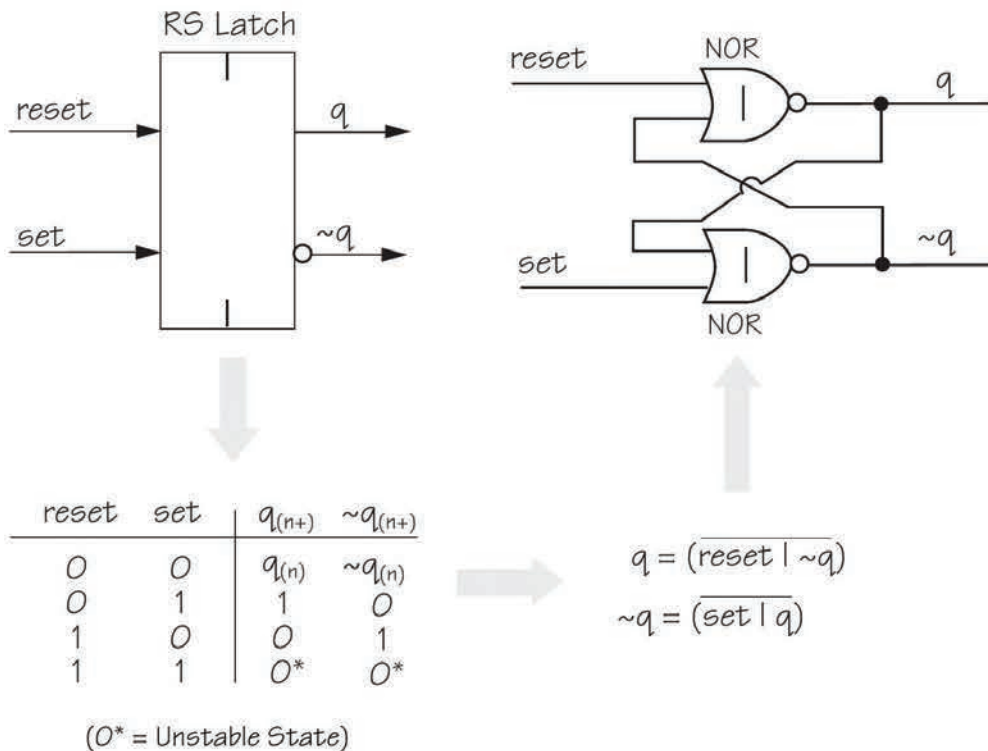


Figure 11-8. NOR implementation of an RS latch

inverse, or complement, of the value on *q*. This is also indicated by the bobble associated with the $\sim q$ output on the symbol. The only time $\sim q$ is not the inverse of *q* occurs when both *reset* and *set* are active at the same time (this unstable state is discussed in more detail below).

The truth table column labels $q_{(n+)}$ and $\sim q_{(n+)}$ indicate that these columns refer to the future values on the outputs. The *n+* subscripts represent some future time, or “now-plus.” By comparison, the labels $q_{(n)}$ and $\sim q_{(n)}$ used in the body of the truth table indicate the current values on the outputs. In this case the *n* subscripts represent the current time, or “now.” Thus, the first row in the truth table indicates that when both *reset* and *set* are in their inactive states (logic 0s), the future values on the outputs will be the same as their current values.

The secret of the RS latch’s ability to remember previous input values is based on a technique known as *feedback*. This refers to the feeding back of the outputs as additional inputs into the function. In order to see how this works, let’s assume that both the *reset* and *set* inputs are initially in their inactive

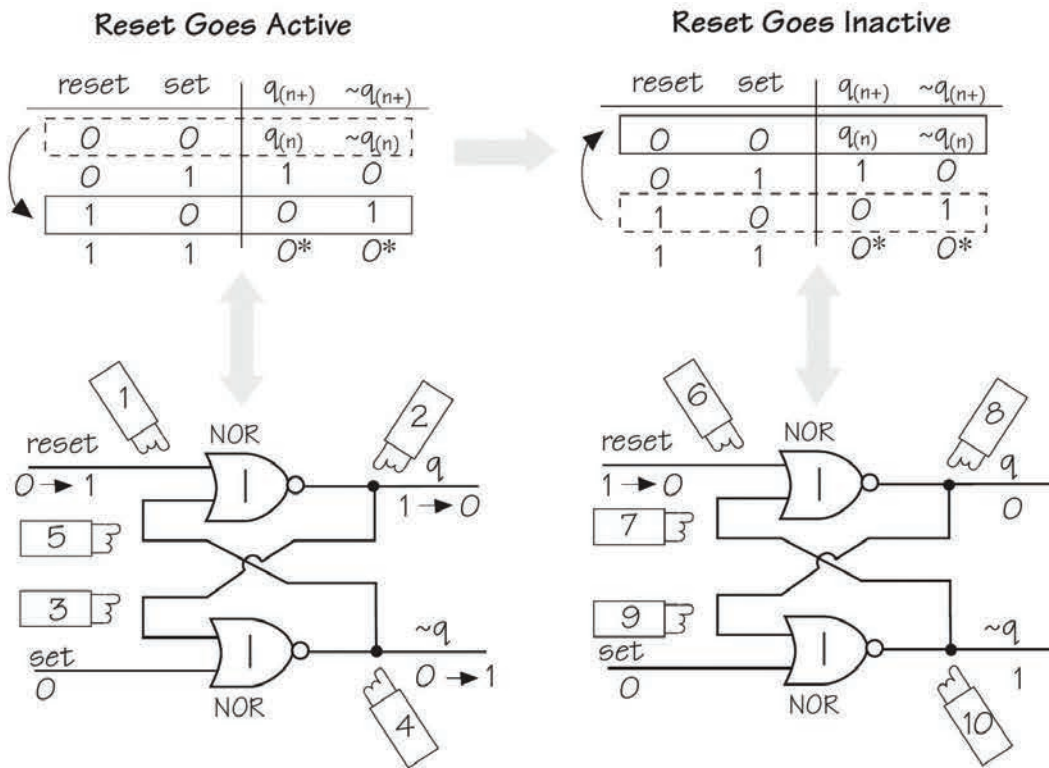


Figure 11-9. RS latch: reset input goes active then inactive

states, but that some previous input sequence placed the latch in its *set condition*; that is, q is 1 and $\sim q$ is 0. Now consider what occurs when the *reset* input is placed in its active state and then returns to its inactive state (Figure 11-9).

As a reminder, if *any* input to a NOR is 1, its output will be forced to 0, and it's only if *both* inputs to a NOR are 0 that the output will be 1. Thus, when *reset* is placed in its active (logic 1) state [1], the q output from the first gate is forced to 0 [2]. This 0 on q is fed back into the second gate [3] and, as *both* inputs to this gate are now 0, the $\sim q$ output is forced to 1 [4]. The key point to note is that the 1 on $\sim q$ is now fed back into the first gate [5].

When the *reset* input returns to its inactive (logic 0) state [6], the 1 from the $\sim q$ output continues feeding back into the first gate [7], which means that the q output continues to be forced to 0 [8]. Similarly, the 0 on q continues feeding back into the second gate [9], and as both of this gate's inputs are now at 0, the $\sim q$ output continues to be forced to 1 [10]. The end

result is that the 1 from [7] causes the 0 at [8] which is fed back to [9], and the 0 on the *set* input combined with the 0 from [9] causes the 1 at [10] which is fed back to [7].

Thus, the latch⁵ has now been placed in its *reset condition*, and a self-sustaining loop has been established. Even though both the *reset* and *set* inputs are now inactive, the *q* output remains at 0, indicating that *reset* was the last input to be in its active state. Once the function has been placed in its *reset condition*, any subsequent activity on the *reset* input will have no effect on the outputs, which means that the only way to affect the function is by means of its *set* input.

Now consider what occurs when the *set* input is placed in its active state and then returns to its inactive state (Figure 11-10).

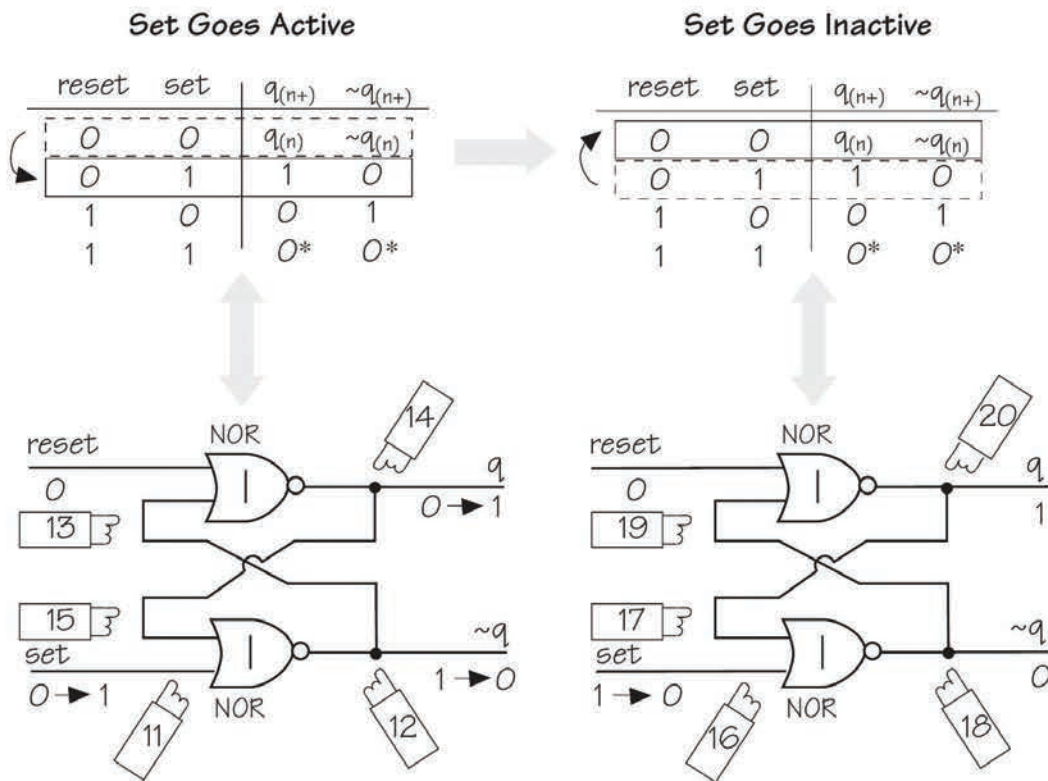


Figure 11-10. RS latch: set input goes active then inactive

⁵ The term *latch*—which is commonly associated with a fastening for a door or gate—comes from the Old English *lacchen*, meaning “to seize.”

When *set* is placed in its active (logic 1) state [11], the $\sim q$ output from the second gate is forced to 0 [12]. This 0 on $\sim q$ is fed back into the first gate [13] and, as both inputs to this gate are now 0, the *q* output is forced to 1 [14]. The key point to note is that the 1 on *q* is now fed back into the second gate [15].

When the *set* input returns to its inactive (logic 0) state [16], the 1 from the *q* output continues feeding back to the second gate [17] and the $\sim q$ output continues to be forced to 0 [18]. Similarly, the 0 on the $\sim q$ output continues feeding back into the first gate [19], and the *q* output continues to be forced to 1 [20]. The end result is that the 1 at [17] causes the 0 at [18] which is fed back to [19], and the 0 on the *reset* input combined with the 0 at [19] causes the 1 at [20] which is fed back to [17].

Thus, the latch has been returned to its *set condition* and, once again, a self-sustaining loop has been established. Even though both the *reset* and *set* inputs are now inactive, the *q* output remains at 1, indicating that *set* was the last input to be in its active state. Once the function has been placed in its *set condition*, any subsequent activity on the *set* input will have no effect on the outputs, which means that the only way to affect the function is by means of its *reset* input.

The unstable condition indicated by the fourth row of the RS latch's truth table occurs when both the *reset* and *set* inputs are active at the same time. Problems occur when both *reset* and *set* return to their inactive states simultaneously or too closely together (Figure 11-11).

When both *reset* and *set* are active at the same time, the 1 on *reset* [21] forces the *q* output to 0 [22] and the 1 on *set* [23] forces the $\sim q$ output to 0 [24]. The 0 on *q* is fed back to the second gate [25], and the 0 on $\sim q$ is fed back to the first gate [26].

Now consider what occurs when *reset* and *set* go inactive simultaneously ([27] and [28], respectively). When the new 0 values on *reset* and *set* are combined with the 0 values fed back from *q* [29] and $\sim q$ [30], each gate initially sees both of its inputs at 0 and therefore both gates attempt to drive their outputs to 1. After any delays associated with the gates have been satisfied, both of the outputs will indeed go to 1.

When the output of the first gate goes to 1, this value is fed back to the input of the second gate. While this is happening, the output of the second gate

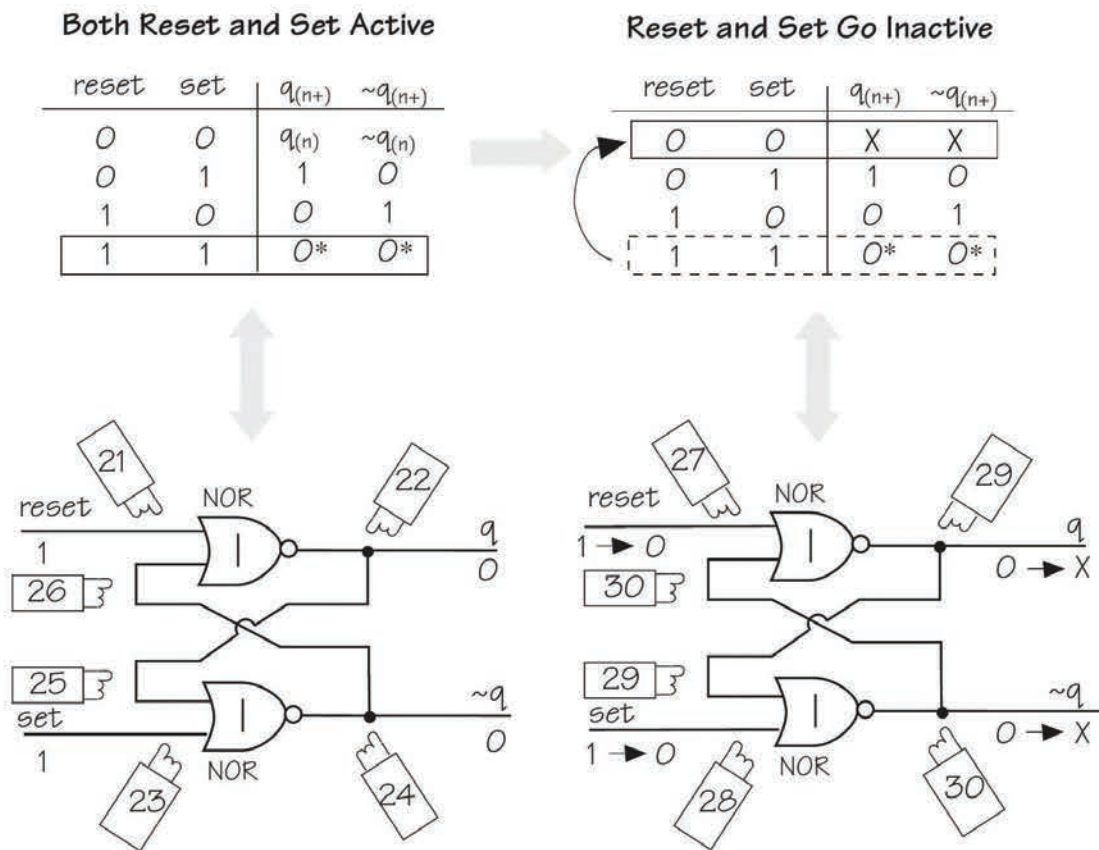


Figure 11-11. RS latch: the reset and set inputs go inactive simultaneously

goes to 1, and this value is fed back to the input of the first gate. Each gate now has its fed-back input at 1, and both gates therefore attempt to drive their outputs to 0. As we see, the circuit has entered a *metastable* condition in which the outputs oscillate between 0 and 1 values.

If both halves of the function were exactly the same, these metastable oscillations would continue indefinitely. But there will always be some differences (no matter how small) between the gates and their delays, and the function will eventually collapse into either its *reset condition* or its *set condition*. As there is no way to predict the final values on the q and $\sim q$ outputs, they are indicated as being in X, or *don't know*, states ($\boxed{29}$ and $\boxed{30}$). These X states will persist until a valid input sequence occurs on either the *reset* or *set* inputs.

An alternative implementation for an RS latch can be realized using two NAND gates connected in a back-to-back configuration (Figure 11-12).

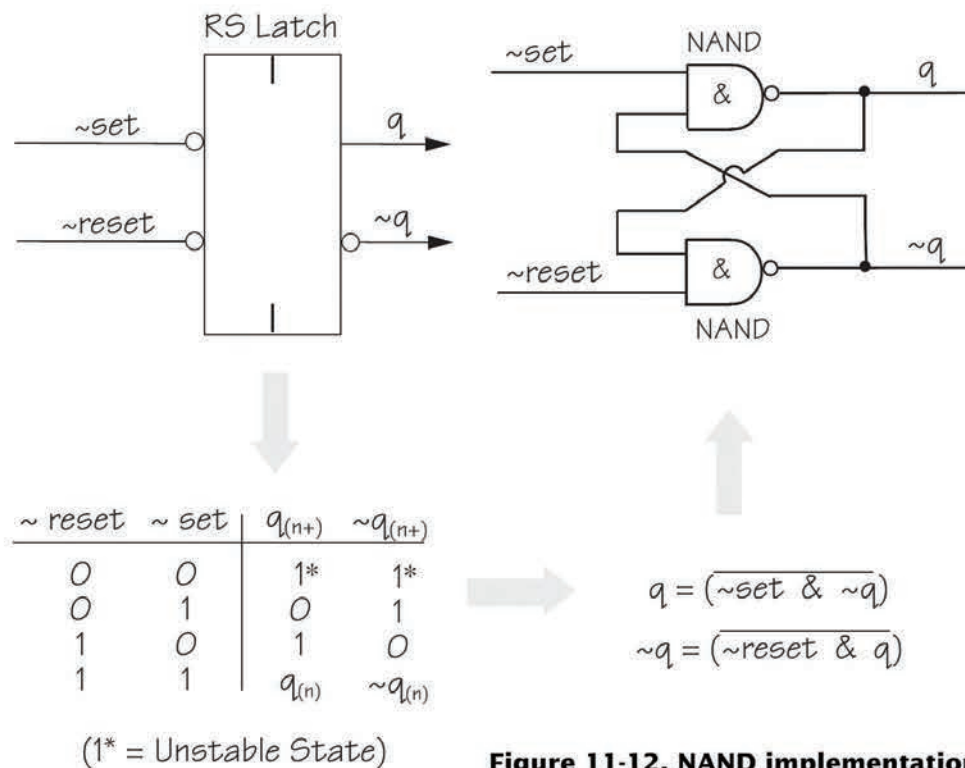


Figure 11-12. NAND implementation of an RS latch

In a NAND implementation, both the $\sim reset$ and $\sim set$ inputs are active low, as is indicated by the bobbles associated with these inputs on the symbol and by the tilde characters in their names. As a reminder, if *any* input to a NAND is 0, the output is 1, and it's only if *both* inputs to a NAND are 1 that the output will be 0. Working out how this version of the latch works is left as an exercise to the reader.⁶

D-Type Latches

A more sophisticated function called a D-type (“data-type”) latch can be constructed by attaching two ANDs and a NOT to the front of an RS latch (Figure 11-13).

⁶ This is where we see if you’ve been paying attention <grin>.

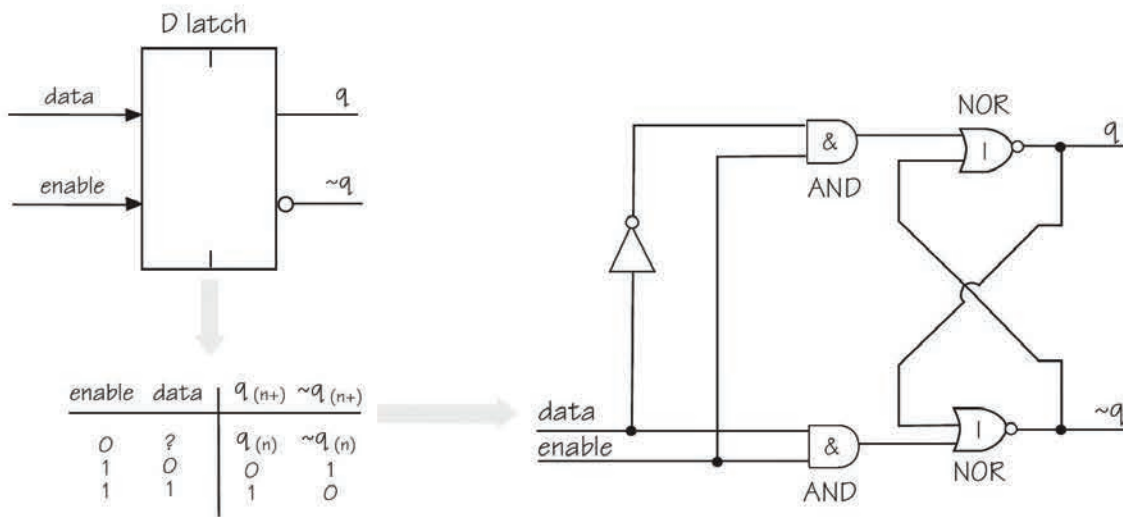


Figure 11-13. D-type latch with active-high enable

The *enable* input is active high for this configuration, as is indicated by the lack of a bobble on the symbol. When *enable* is placed in its active (logic 1) state, the true and inverted versions of the *data* input are allowed to propagate through the AND gates and are presented to the back-to-back NOR gates. If the *data* input changes while *enable* is still active, the outputs will respond to reflect the new value.

When *enable* returns to its inactive (logic 0) state, it forces the outputs of both ANDs to 0, and any further changes on the *data* input have no effect. Thus, the back-to-back NOR gates remember the last value they saw from the *data* input prior to the *enable* input going inactive.

Consider an example waveform (Figure 11-14). While the *enable* input is in its active state, whatever value is presented to the *data* input appears on the *q* output and an inverted version appears on the $\sim q$ output. As usual, there will always be some element of delay between changes on the inputs and corresponding responses on the outputs. When *enable* goes inactive, the outputs remember their previous values and no longer respond to any changes on the *data* input. As the operation of the device depends on the logic value, or level, on *enable*, this input is said to be *level-sensitive*.

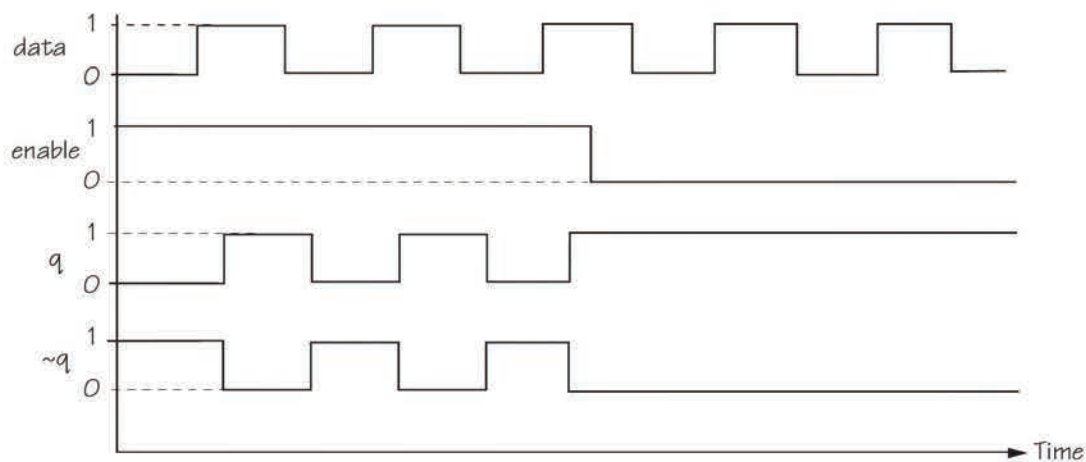


Figure 11-14. Waveform for a D-type latch with active-high enable

D-Type Flip-flops

In the case of a D-type flip-flop (which may also be referred to as a *register*), the data *appears* to be loaded when a transition, or edge, occurs on the clock input, which is therefore said to be *edge-sensitive* (the reason we say “appears to

be loaded when an edge occurs” is discussed in the sidebar on the next page). A transition from 0 to 1 is known as a *rising-edge* or a *positive-edge*, while a transition from 1 to 0 is known as a *falling-edge* or a *negative-edge*. A D-type flip-flop’s clock input may be positive-edge or negative-edge triggered (Figure 11-15).

The chevrons (arrows “>”) associated with the clock inputs on the symbols indicate

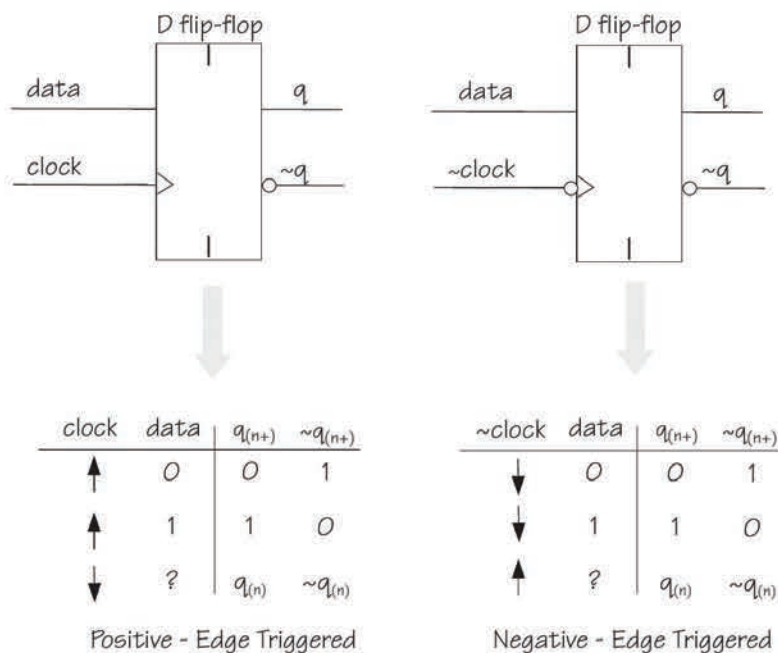


Figure 11-15. Positive-edge and negative-edge D-type flip-flops

that these are edge-sensitive inputs. A chevron without an associated bobble indicates a positive-edge clock, and a chevron with a bobble indicates a negative-edge clock. The last rows in the truth tables show that an inactive edge on the clock leaves the contents of the flip-flops unchanged (these cases are often omitted from the truth tables).

Consider an example waveform for a positive-edge triggered D-type flip-flop (Figure 11-16). As the observer initially has no knowledge as to the contents of the flop-flop, the q and $\sim q$ outputs are initially shown as having X, or *don't know*, values.

There are a number of ways to implement a D-Type flip-flop. The most understandable from our point of view would be to use two D-type latches in series (one after the other). The first latch could have an active-low enable and the second could have an active-high enable. Both of these enables would be connected together, and would be known as the *clock* input to the outside world.

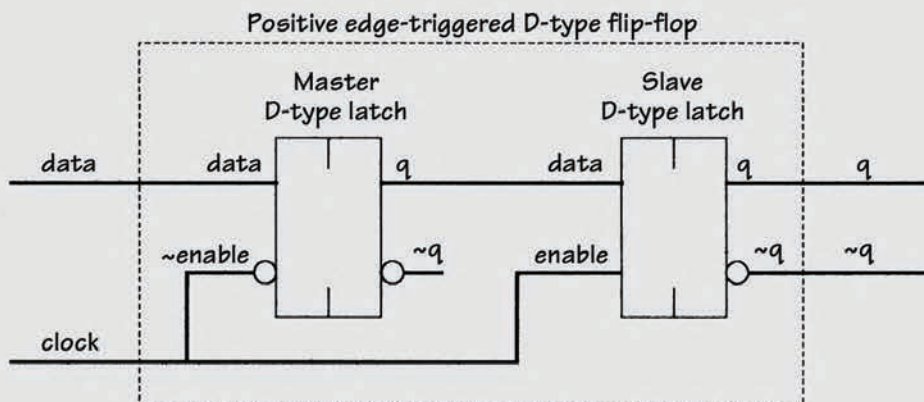
This is known as a *master-slave* relationship, where the first latch is the “master” and the second is the “slave.”

When the *clock* input is 0, the master latch is enabled and passes whatever value is presented to its data input

through to its outputs (only its q output is actually used in this example). Meanwhile, the slave latch is disabled and continues to store (and output) its existing contents.

When the *clock* input is subsequently driven to a 1, the master latch is disabled and continues to store (and output) its existing contents. Meanwhile the slave latch is now enabled and passes whatever value is presented to *its* data input (the value from the output of the master latch) through to *its* outputs.

Thus, everything is really controlled by voltage levels, but from the outside world it *appears* that the flip-flop was loaded by a *rising-edge* on the *clock* input.



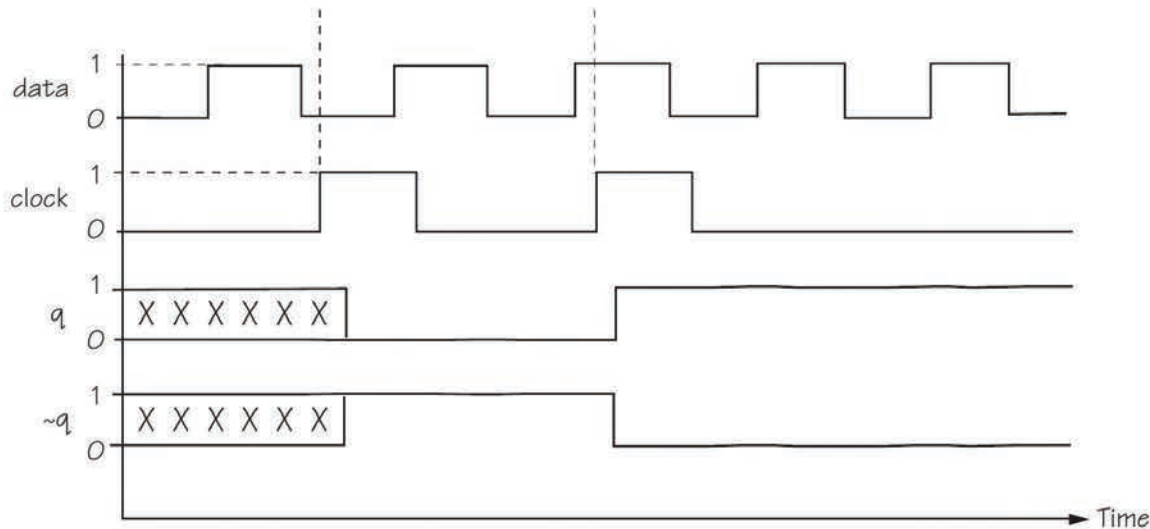


Figure 11-16. Waveform for positive-edge D-type flip-flop

The first rising edge of the clock loads the 0 on the *data* input into the flip-flop, which (after a small delay) causes *q* to change to 0 and $\sim q$ to change to 1. The second rising edge of the clock loads the 1 on the *data* input into the flip-flop; *q* goes to 1 and $\sim q$ goes to 0.

Some flip-flops have an additional input called $\sim clear$ or $\sim reset$ which forces *q* to 0 and $\sim q$ to 1, irrespective of the value on the *data* input (Figure 11-17). Similarly, some flip-flops have a $\sim preset$ or $\sim set$ input, which forces *q* to 1 and $\sim q$ to 0, and some have both $\sim clear$ and $\sim preset$ inputs.

The examples shown in Figure 11-17 reflect active-low $\sim clear$ inputs, but active-high equivalents are also available. Furthermore, as is illustrated in Figure 11-17, these inputs may be either *asynchronous* or *synchronous*. In the more common asynchronous case, the effect of $\sim clear$ going active is immediate and overrides both the *clock* and *data* inputs (the “asynchronous” qualifier reflects the fact that the effect of this input is *not* synchronized to the *clock*). By comparison, in the synchronous case the effect of $\sim clear$ is synchronized to the active edge of the *clock*.⁷

⁷ The component symbols used in this book are relatively traditional and simple. One disadvantage of this is that, as seen in Figure 11-17, there’s no way to tell if a *clear* or *preset* input is synchronous or asynchronous without also looking at its truth table. There are more modern and sophisticated symbol standards that do cover all eventualities, but their complexity is beyond the scope of this book to explain.

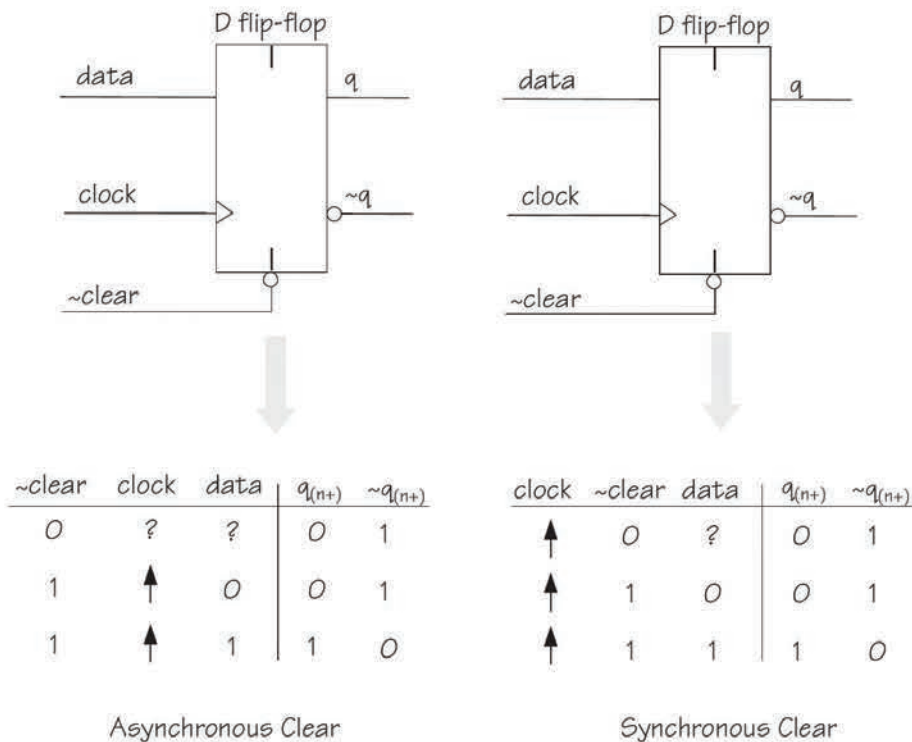


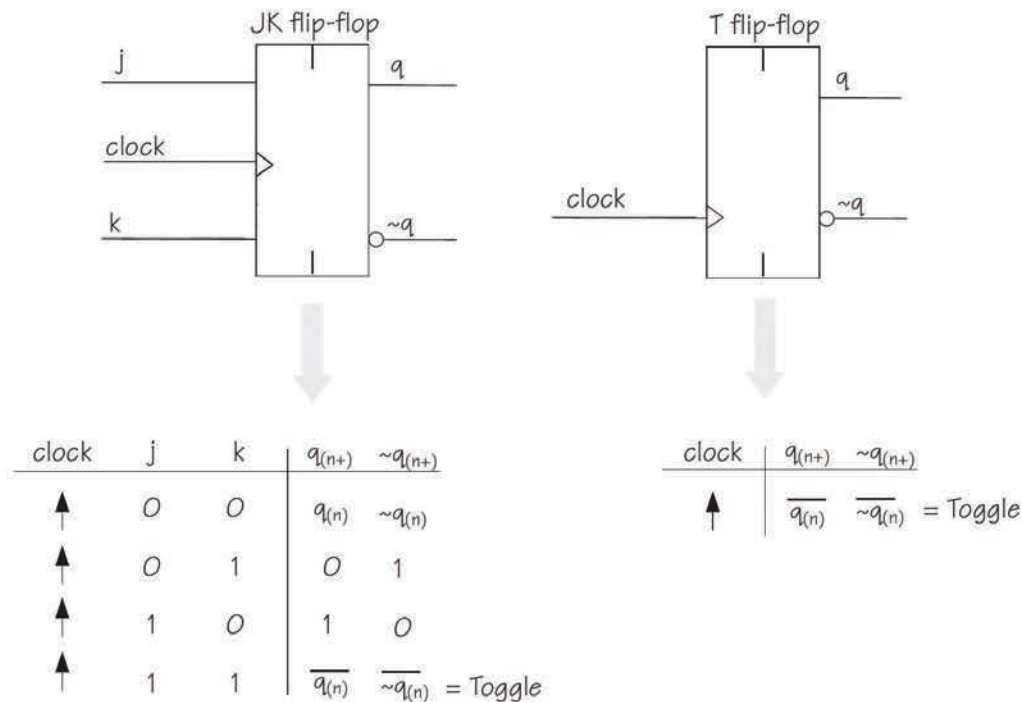
Figure 11-17. D-type flip-flops with asynchronous and synchronous clear inputs

JK and T Flip-flops

The majority of examples in this book are based on D-type flip-flops. However, for the sake of completeness, it should be noted that there are several other flavors of flip-flops available. Two common types are the JK and T (for Toggle) flip-flops (Figure 11-18).

The first row of the JK flip-flop's truth table shows that when both the j and k (data) inputs are 0, an active edge on the *clock* input leaves the contents of the flip-flop unchanged. The two middle rows of the truth table show that if the j and k inputs have opposite values, an active edge on the *clock* input will effectively load the flip-flop (the q output) with the value on j (the $\sim q$ output will take the complementary value). The last line of the truth table shows that when both the j and k inputs are 1, an active edge on the *clock* causes the outputs to toggle to the inverse of their previous values.⁸ By comparison, the

⁸ This may be the origin of the term "flip-flop," because the outputs *flip* and *flop* back and forth.

**Figure 11-18. JK and T flip-flops**

T flip-flop doesn't have any data inputs; the outputs simply toggle to the inverse of their previous values on each active edge of the *clock* input.

Shift Registers

As was previously noted, another term for a flip-flop is “register.” Functions known as *shift registers*—which facilitate the shifting of binary data one bit at a time—are commonly used in digital systems. Consider a simple 4-bit shift register constructed using D-type flip-flops (Figure 11-19).

This particular example is based on positive-edge triggered D-type flip-flops with active-low $\sim clear$ inputs (in this case we're only using each register's q output). Also, this example is classed as a *serial-in-parallel-out (SIPO)* shift register, because data is loaded in serially (one after the other) and read out in parallel (side by side).

When the $\sim clear$ input is set to 1 (its inactive state), a positive-edge on the *clock* input loads the value on the *serial_in* input into the first flip-flop, $dff[0]$. At the same time, the value that used to be in $dff[0]$ is loaded into $dff[1]$, the

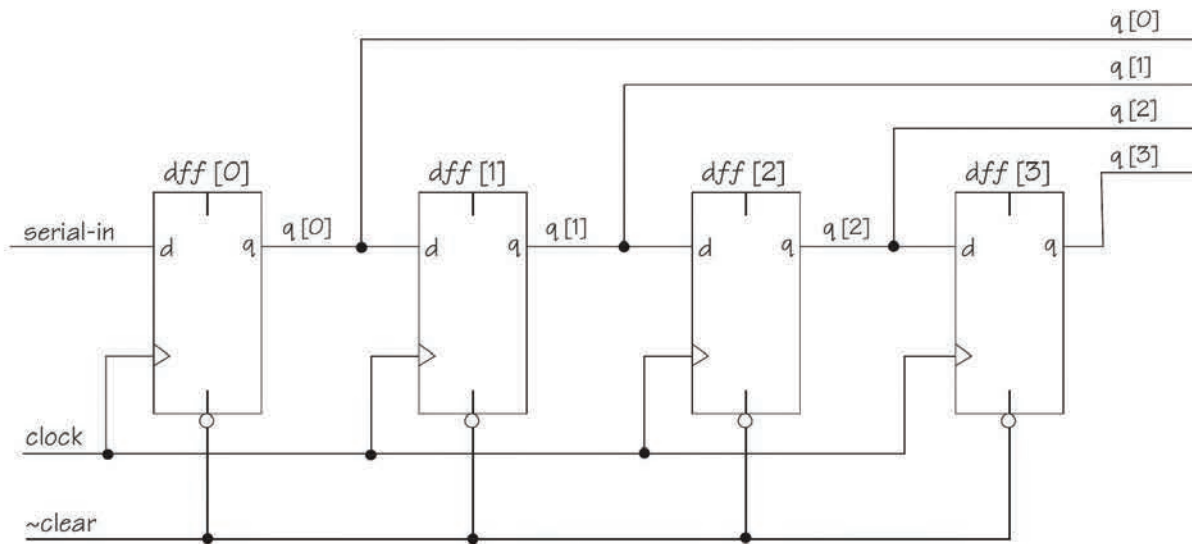


Figure 11-19. SIPO shift register

value that used to be in $dff[1]$ is loaded into $dff[2]$, and the value that used to be in $dff[2]$ is loaded into $dff[3]$.

This may seem a bit weird and wonderful the first time you see it, but the way in which this works is actually quite simple (and of course capriciously cunning). Each flip-flop exhibits a delay between seeing an active edge on its clock input and the ensuing response on its q output. These delays provide sufficient time for the next flip-flop in the chain to load the value from the previous stage before that value changes. Consider an example waveform where a single logic 1 value is migrated through the shift register (Figure 11-20).

Initially all of the flip-flops contain don't know X values. When the $\sim clear$ input goes to its active state (logic 0), all of the flip-flops are cleared to 0. When the first active edge occurs on the clock input, the $serial_in$ input is 1, so this is the value that's loaded into the first flip-flop. At the same time, the original 0 value from the first flip-flop is loaded into the second, the original 0 value from the second flip-flop is loaded into the third, and the original 0 value from the third flip-flop is loaded into the fourth.

When the next active edge occurs on the clock input, the $serial_in$ input is 0, so this is the value that's loaded into the first flip-flop. At the same time, the original 1 value from the first flip-flop is loaded into the second, the 0 value from the second flip-flop is loaded into the third, and the 0 value from the third flip-flop is loaded into the fourth.

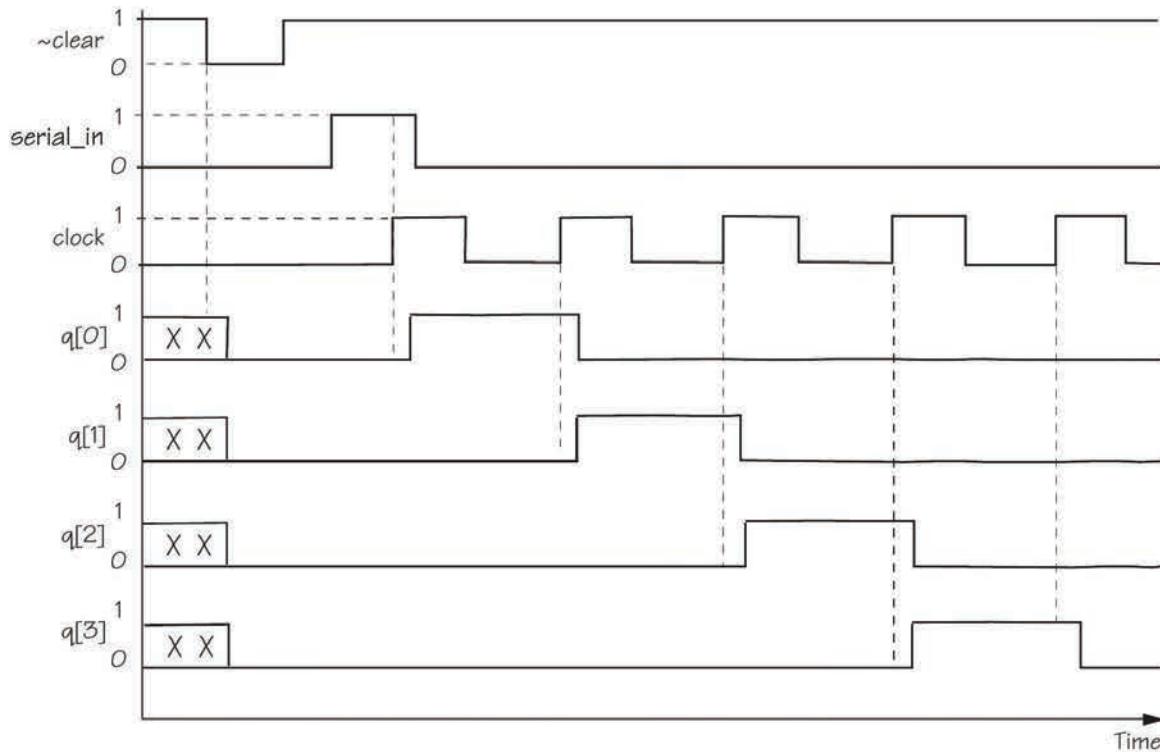


Figure 11-20. Waveform for SIPO shift register

Similarly, when the next active edge occurs on the clock input, the *serial_in* input is still 0, so this is the value that's loaded into the first flip-flop. At the same time, the 0 value from the first flip-flop is loaded into the second, the 1 value from the second flip-flop is loaded into the third, and the 0 value from the third flip-flop is loaded into the fourth. And so it goes . . .

Other common shift register variants are the *parallel-in-serial-out* (PISO), and the *serial-in-serial-out* (SISO); for example, consider a 4-bit SISO shift register (Figure 11-21).

Counters

Counter functions are also commonly used in digital systems. The number of states that the counter will sequence through before returning to its original value is called the *modulus* of the counter. For example, a function that counts from 0000_2 to 1111_2 in binary (or 0 to 15 in decimal) has a modulus of sixteen and would be called a modulo-16, or mod-16, counter. Consider a modulo-16 counter implemented using D-type flip-flops (Figure 11-22).

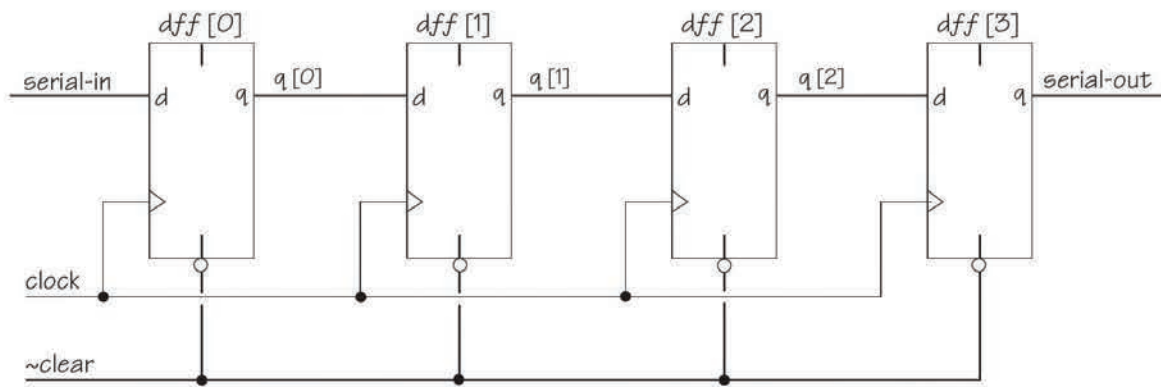


Figure 11-21. SISO shift register

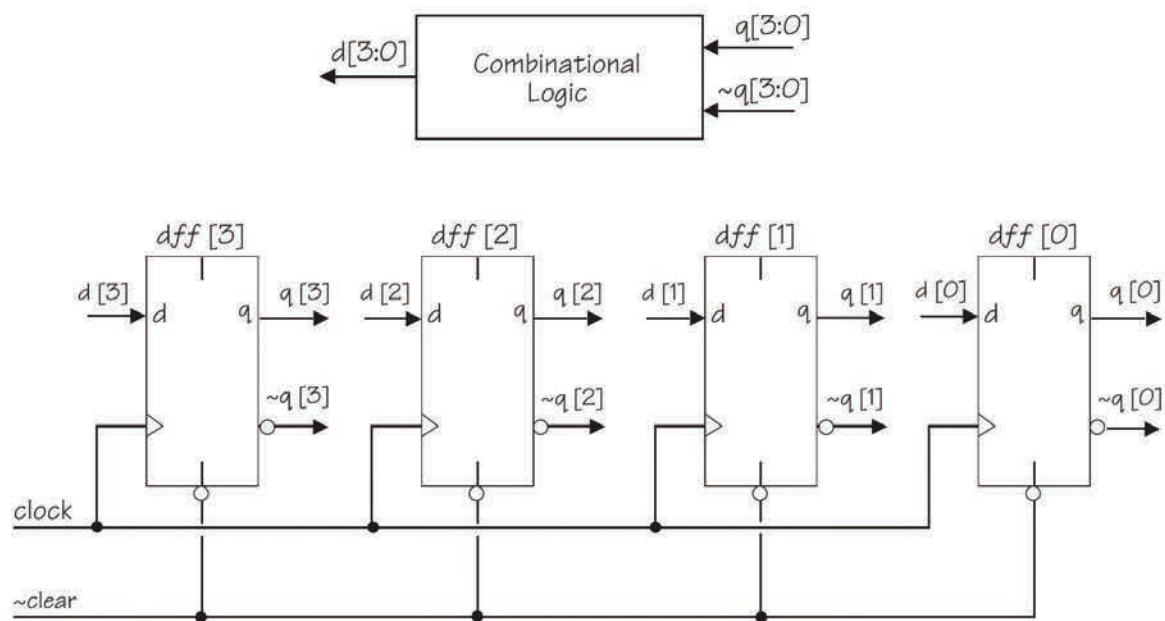


Figure 11-22. Modulo-16 binary counter

This particular example is based on positive-edge triggered D-type flip-flops with active-low $\sim clear$ inputs. The four flip-flops are used to store the current count value which is displayed on the $q[3:0]$ outputs. When the $\sim clear$ input is set to 1 (its inactive state), a positive-edge on the clock input causes the counter to load the next value in the count sequence.

A block of combinational logic is used to generate the next value, $d[3:0]$, which is based on the current value $q[3:0]$ (Figure 11-23). Note that there is no need to create the inverted versions of $q[3:0]$, because these signals are already available from the flip-flops as $\sim q[3:0]$.

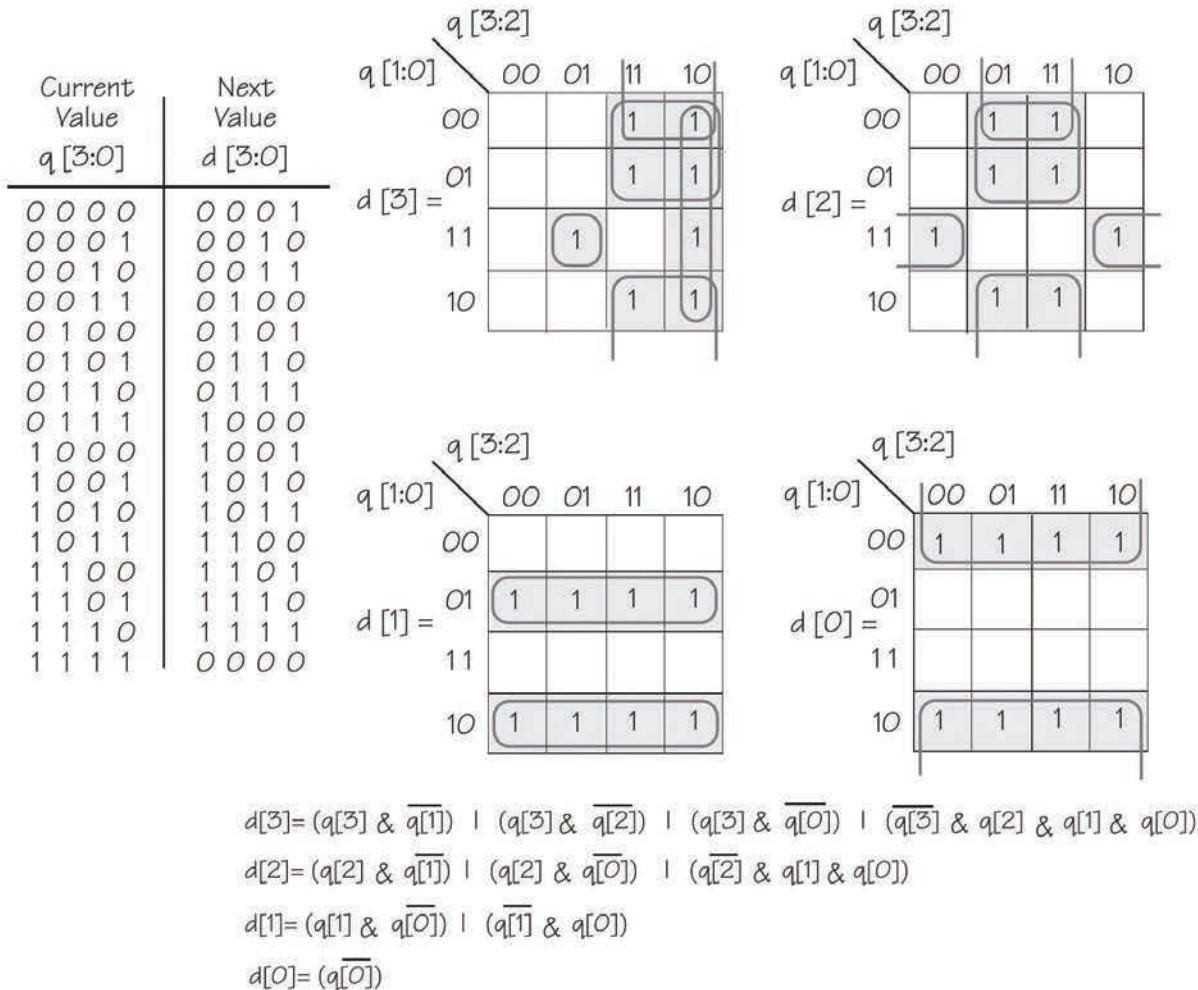


Figure 11-23. Generating the next count value

Setup and Hold Times

One point we've glossed over thus far is the fact that there are certain timing requirements associated with flip-flops. In particular, there are two parameters called the setup and hold times which describe the relationship between the flip-flop's data and clock inputs (Figure 11-24).

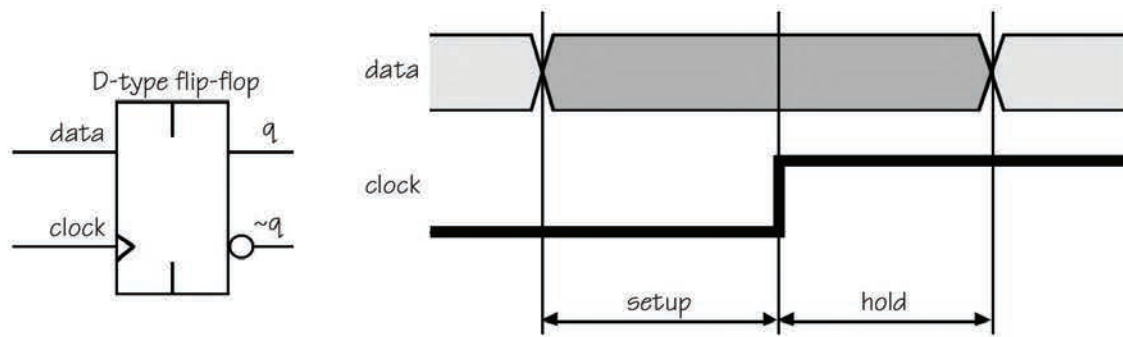


Figure 11-24. Setup and hold times

The waveform shown here is a little different to those we've seen before. What we're trying to indicate is that when we start (on the left-hand side), the value presented to the *data* input may be a 0 or a 1, and it can change back and forth as often as it pleases. However, it must settle one way or the other before the *setup* time; otherwise when the active edge occurs on the *clock* we can't guarantee what will happen. Similarly, the value presented to the *data* input must remain stable for the *hold* time following the *clock*, or once again we can't guarantee what will happen. In our illustration, the period for which the value on the *data* input must remain stable is shown as being the darker gray.

The setup and hold times shown above are reasonably understandable. However things can sometimes become a little confusing, especially in the case of today's deep submicron (DSM) integrated circuit technologies.⁹ The problem is that we may sometimes see so-called *negative* setup and hold times (Figure 11-25).

Once again, the periods for which the value on the *data* input must remain stable are shown as being the darker gray. These effects, which may seem a little strange at first, are caused by internal delay paths inside the flip-flop.

Last but not least, we should note that there will also be setup and hold times between the *clear* (or *reset*) and *preset* (or *set*) inputs and the *clock* input. Also, there will be corresponding setup and hold times between the *data*, *clear* (or *reset*) and *preset* (or *set*) inputs and the *enable* input on D-type latches (phew!).

⁹ Integrated circuits (and DSM technologies) are introduced in Chapter 14.

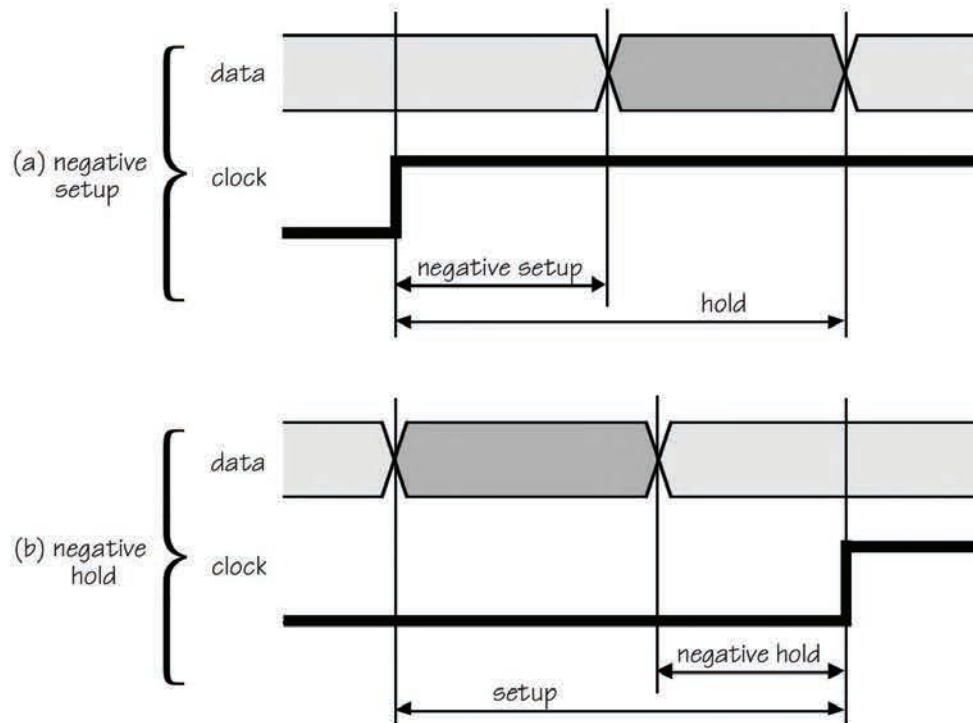


Figure 11-25. Negative setup and hold times

Brick by Brick

Let us pause here for a brief philosophical moment. Consider, if you will, a brick formed from clay. Now, there's not a lot you can do with a single brick, but when you combine thousands and thousands of bricks together you can create the most tremendous structures. At the end of the day, the Great Wall of China is no more than a pile of bricks molded by man's imagination.¹⁰

In the world of the electronics engineer, transistors are the clay, primitive logic gates are the bricks, and the functions described above are simply building blocks.¹¹ Any digital system, even one as complex as a supercomputer, is constructed from building blocks like comparators, multiplexers, shift registers, and counters. Once you understand the building blocks, there are no ends to the things you can achieve!

¹⁰ But at approximately 2,400 km in length, it's a very impressive pile of bricks (the author has walked—and climbed—a small portion of the beast and it fair took his breath away).

¹¹ We might also note that clay and transistors share something else in common—they both consist predominantly of silicon!

State Diagrams, State Tables, and State Machines

Consider a coin-operated machine that accepts nickels and dimes¹ and, for the princely sum of fifteen cents, dispenses some useful article called a “gizmo” that the well-dressed man-about-town could not possibly be without. We may consider such a machine to comprise three main blocks: a *receiver* that accepts money, a *dispenser* that dispenses the “gizmo” and any change, and a *controller* that oversees everything and makes sure things function as planned (Figure 12-1).

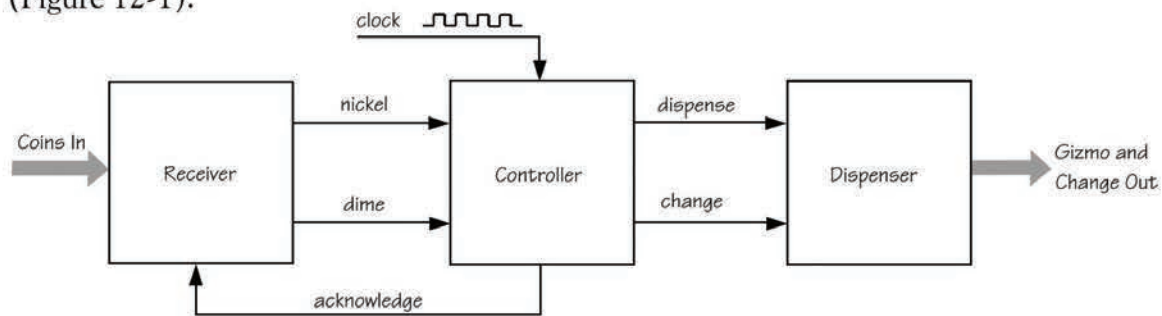


Figure 12-1. Block diagram of a coin-operated machine

The connections marked *nickel*, *dime*, *dispense*, *change*, and *acknowledge* represent digital signals carrying logic 0 and 1 values. The user can deposit nickels and dimes into the receiver in any order, but may only deposit one coin at a time. When a coin is deposited, the receiver determines its type and sets the corresponding signal (*nickel* or *dime*) to a logic 1.

The operation of the controller is synchronized by the *clock* signal. On a rising edge of the *clock*, the controller examines the *nickel* and *dime* inputs to see if any coins have been deposited. The controller keeps track of the amount of money deposited and determines if any actions are to be performed.

¹ For the benefit of those readers who do not reside in the United States, nickels and dimes are American coins worth five and ten cents, respectively.

Every time the controller inspects the *nickel* and *dime* signals, it sends an *acknowledge* signal back to the receiver. The *acknowledge* signal informs the receiver that the coin has been accounted for, and the receiver responds by resetting the *nickel* and *dime* signals to 0 and awaiting the next coin. The *acknowledge* signal can be generated in a variety of ways which are not particularly relevant here.

When the controller decides that sufficient funds have been deposited, it instructs the dispenser to dispense a “gizmo” and any change (if necessary) by setting the *dispense* and *change* signals to 1, respectively.

State Diagrams

A useful level of abstraction for a function such as the controller is to consider it as consisting of a set of *states* through which it sequences. The *current state* depends on the *previous state* combined with the *previous* values on the *nickel* and *dime* inputs. Similarly, the *next state* depends on the *current state* combined with the *current* values on the *nickel* and *dime* inputs. The operation of the controller may be represented by means of a *state diagram*, which offers a way to view the problem and to describe a solution (Figure 12-2).

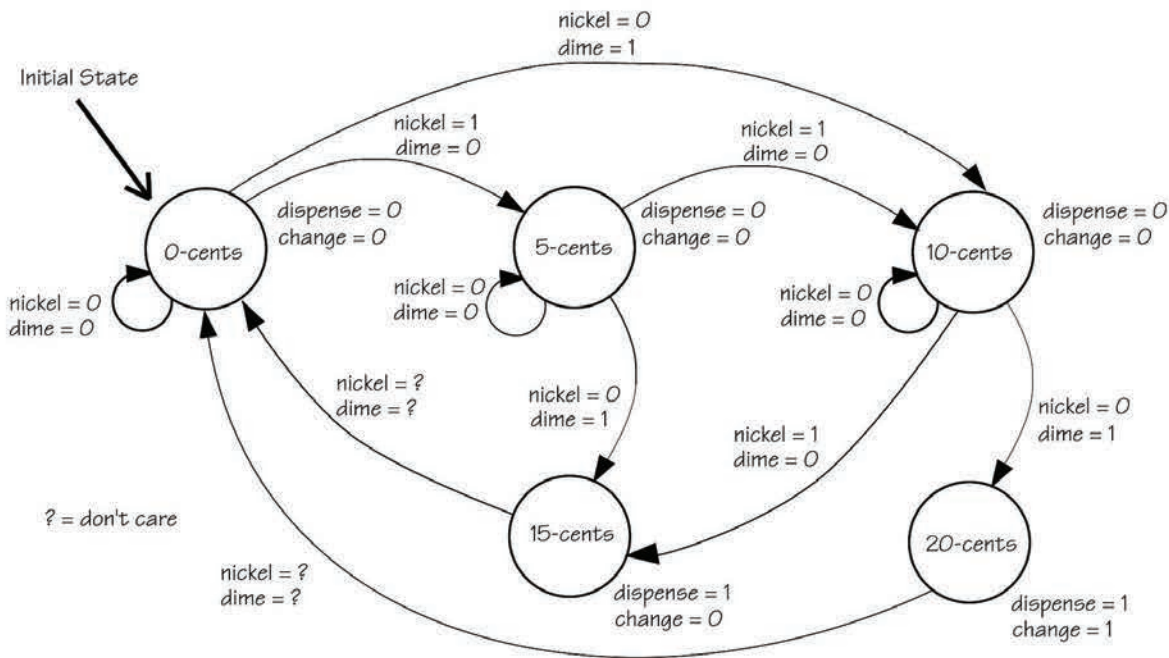


Figure 12-2. State diagram for the controller

The states are represented by the circles labeled *0-cents*, *5-cents*, *10-cents*, *15-cents*, and *20-cents*, and the values on the *dispense* and *change* outputs are associated with these states. The arcs connecting the states are called *state transitions* and the values of the *nickel* and *dime* inputs associated with the state transitions are called *guard conditions*. The controller will only sequence between two states if the values on the *nickel* and *dime* inputs match the guard conditions.

Let's assume that the controller is in its initial state of *0-cents*. The values of the *nickel* and *dime* inputs are tested on every rising edge on the clock.² As long as no coins are deposited, the *nickel* and *dime* inputs remain at 0 and the controller remains in the *0-cents* state. Once a coin is deposited, the next rising edge on the clock will cause the controller to sequence to the *5-cents* or the *10-cents* states depending on the coin's type. It is at this point that the controller sends an *acknowledge* signal back to the receiver instructing it to reset the *nickel* and *dime* signals back to 0 and to await the next coin.

Note that the *0-cents*, *5-cents*, and *10-cents* states have state transitions that loop back into them (the ones with associated *nickel* = 0 and *dime* = 0 guard conditions). These indicate that the controller will stay in whichever state it is currently in until a new coin is deposited.

So at this stage of our discussions, the controller is either in the *5-cents* or the *10-cents* state depending on whether the first coin was a nickel or dime, respectively. What happens when the next coin is deposited? Well this depends on the state we're in and the type of the new coin. If the controller is in the *5-cents* state, then a nickel or dime will move it to the *10-cents* or *15-cents* states, respectively. Alternatively, if the controller is in the *10-cents* state, then a nickel or dime will move it to the *15-cents* or *20-cents* states, respectively.

When the controller reaches either the *15-cents* or *20-cents* states, the next clock will cause it to dispense a "gizmo" and return to its initial *0-cents* state (in the case of the *20-cents* state, the controller will also dispense a nickel in change).

² The controller is known to sequence between states only on the rising edge of the clock, so displaying this signal on every state transition would be redundant.

State Tables

Another form of representation is that of a *state table*. This is similar to a truth table (inputs on the left and corresponding outputs on the right), but it also includes the *current state* as an input and the *next state* as an output (Figure 12-3).

<i>Current state</i>	<i>clock</i>	<i>nickel</i>	<i>dime</i>	<i>dispense</i>	<i>change</i>	<i>Next state</i>
0-cents	↑	0	0	0	0	0-cents
0-cents	↑	1	0	0	0	5-cents
0-cents	↑	0	1	0	0	10-cents
5-cents	↑	0	0	0	0	5-cents
5-cents	↑	1	0	0	0	10-cents
5-cents	↑	0	1	0	0	15-cents
10-cents	↑	0	0	0	0	10-cents
10-cents	↑	1	0	0	0	15-cents
10-cents	↑	0	1	0	0	20-cents
15-cents	↑	?	?	1	0	0-cents
20-cents	↑	?	?	1	1	0-cents

Figure 12-3. State table for the controller

In this instance the *clock* signal has been included for purposes of clarity (it's only when there's a rising edge on the *clock* that the outputs are set to the values shown in that row of the table). However, as for the state diagram, displaying this signal is somewhat redundant and it is often omitted.

State Machines

The actual implementation of a function such as the controller is called a *state machine*. In fact, when the number of states is constrained and finite, this is more usually called a *finite state machine (FSM)*. The heart of a state machine consists of a set of registers³ known as the *state variables*. Each state, *0-cents*, *5-cents*, *10-cents*, . . . is assigned a unique binary pattern of 0s and 1s, and the pattern representing the *current state* is stored in the state variables.

³ For the purposes of these discussions we'll assume that these registers are D-type flip-flops as were introduced in Chapter 11.

The two most common forms of synchronous, or clocked, state machines are known as *Moore* and *Mealy* machines after the men who formalized them. A Moore machine is distinguished by the fact that the outputs are derived only from the values in the state variables (Figure 12-4). The controller function featured in this discussion is a classic example of a Moore machine.

By comparison, the outputs from a Mealy machine may be derived from a combination of the values in the state variables and one or more of the inputs (Figure 12-5).

In both of the Moore and Mealy forms, the input logic consists of primitive gates such as AND, NAND, OR, and NOR. These combine the values on the inputs with the *current state* (which is fed back from the state variables) to generate the pattern of 0s and 1s representing the *next state*. This new pattern of 0s and 1s is presented to the inputs of the state variables and will be loaded into them on the next rising edge of the *clock*.

The output logic also consists of standard primitive logic gates that generate the appropriate values on the outputs from the *current state* stored in the state variables.

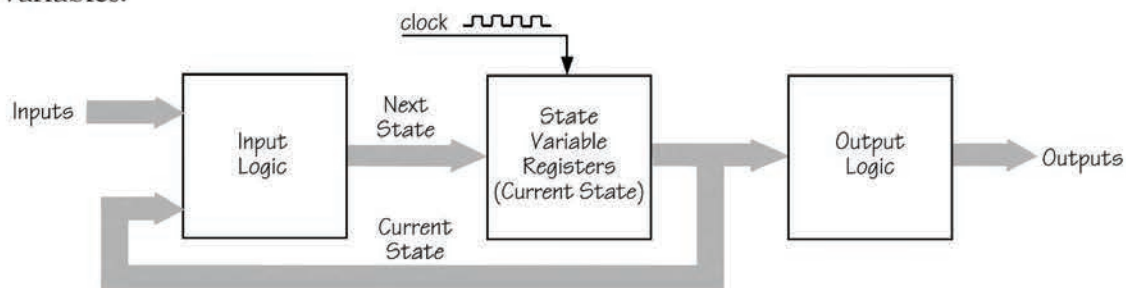


Figure 12-4. Block diagram of a Moore machine

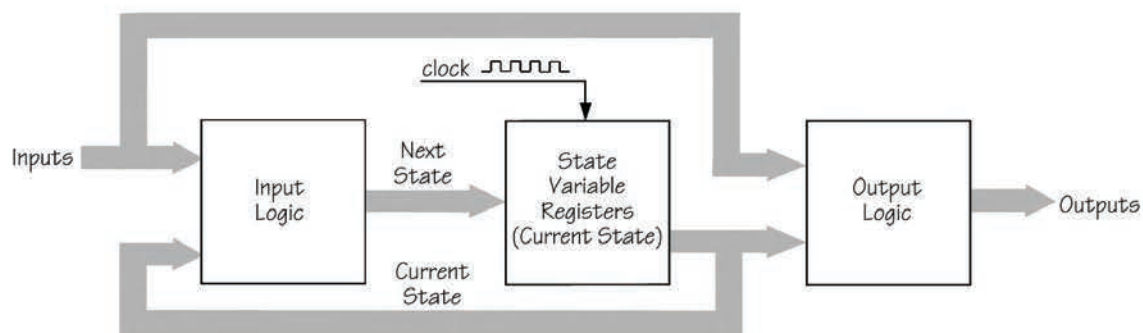


Figure 12-5. Block diagram of a Mealy machine

State Assignment

A key consideration in the design of a state machine is that of *state assignment*, which refers to the process by which the states are assigned to the binary patterns of 0s and 1s that are to be stored in the state variables.

A common form of state assignment requiring the minimum number of registers is known as *binary encoding*. Each register can only contain a single binary digit, so it can only be assigned a value of 0 or 1. Two registers can be assigned four binary values (00, 01, 10, and 11), three registers can be assigned eight binary values (000, 001, 010, 011, 100, 101, 110, and 111), and so forth. The controller used in our coin-operated machine consists of five unique states, and therefore requires a minimum of three state variable registers.

The actual process of binary encoded state assignment is a nontrivial problem. In the case of our controller function, there are 6,720 possible combinations⁴ by which five states can be assigned to the eight binary values provided by three registers. Each of these solutions may require a different arrangement of primitive gates to construct the input and output logic, which in turn affects the maximum frequency that can be used to drive the system clock. Additionally, the type of registers used to implement the state variables also affects the supporting logic; the following discussions are based on the use of D-type flip-flops.

Assuming that full use is made of *don't care* states, an analysis of the various binary encoded solutions for our controller yields the following . . .

138 solutions requiring 7 product terms
 852 solutions requiring 8 product terms
 1,876 solutions requiring 9 product terms
 3,094 solutions requiring 10 product terms
 570 solutions requiring 11 product terms
 190 solutions requiring 12 product terms

. . . where a *product term* is a group of *literals* linked by & (AND) operators—for example, $(a \& b \& c)$ —and a *literal* is any true or inverted variable. Thus, the product term $(a \& b \& c)$ contains three literals (a , b , and c).

⁴ This number would be somewhat reduced if all the mirror-image combinations were taken into account, but that would not significantly lessen the complexity problem of determining the optimal combination.

But wait, there's more! A further analysis of the 138 solutions requiring seven product terms yields the following:

66 solutions requiring 17 literals

24 solutions requiring 18 literals

48 solutions requiring 19 literals

Thus, the chances of a random assignment resulting in an optimal solution is relatively slight. Fortunately, there are computer programs available to aid designers in this task.⁵ One solution resulting in the minimum number of product terms and literals is shown in Figure 12-6.

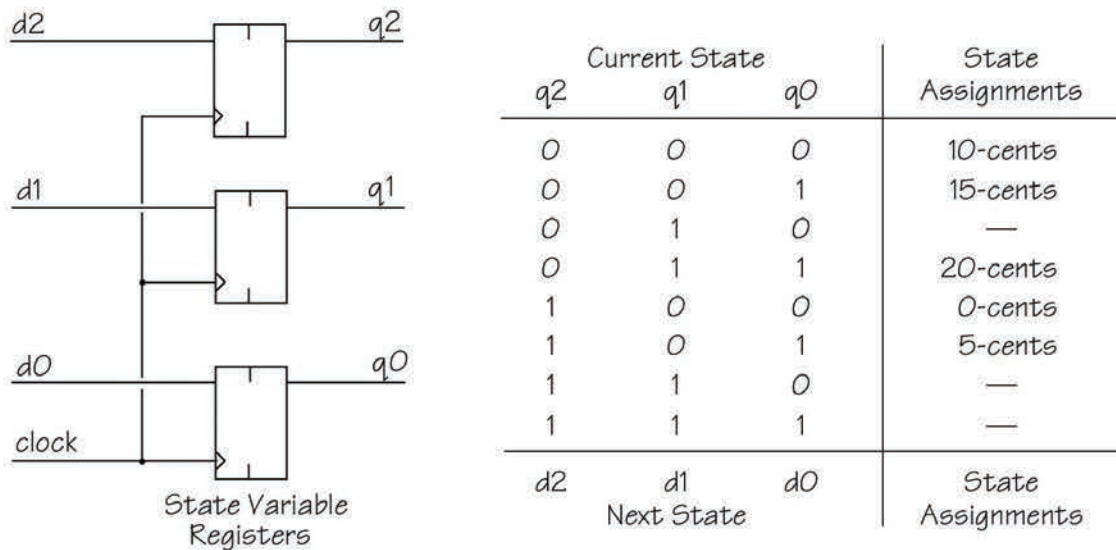


Figure 12-6. Example binary encoded state assignment

A truth table for the controller function can now be derived from the state table shown in Figure 12-3 by replacing the assignments in the *current state* column with the corresponding binary patterns for the state variable outputs ($q2$, $q1$, and $q0$), and replacing the assignments in the *next state* column with the corresponding binary patterns for the state variable inputs ($d2$, $d1$, and $d0$). The resulting equations can then be derived from the truth table by means of standard algebraic or Karnaugh map techniques. As an alternative, a computer

⁵ The author used the program BOOL, which was created by his friend Alon Kfir (a man with a size-16 brain if ever there was one).

program can be used to obtain the same results in less time with far fewer opportunities for error.⁶ Whichever technique is employed, the state assignments above lead to the following minimized Boolean equations:

$$\begin{aligned}
 d0 &= (\overline{q0} \& \overline{q2} \& \text{dime}) \vee (q0 \& q2 \& \overline{\text{nickel}}) \vee (\overline{q0} \& \text{nickel}) \\
 d1 &= (\overline{q0} \& \overline{q2} \& \text{dime}) \\
 d2 &= (q0 \& \overline{q2}) \vee (q2 \& \overline{\text{nickel}} \& \overline{\text{dime}}) \vee (\overline{q0} \& q2 \& \overline{\text{dime}}) \\
 \text{dispense} &= (q0 \& \overline{q2}) \\
 \text{change} &= (q1)
 \end{aligned}$$

The product terms shown in bold appear in multiple equations. However, regardless of the number of times a product term appears, it is only counted once because it only has to be physically implemented once. Similarly, the literals used to form product terms that appear in multiple equations are only counted once.

Another common form of state assignment is known as *one-hot encoding*, in which each state is represented by an individual register. In this case, our controller with its five states would require five register bits. The one-hot technique typically requires a greater number of logic gates than does binary encoding. However, as the logic gates are used to implement simpler equations, the one-hot method results in faster state machines that can operate at higher clock frequencies.

Don't Care States, Unused States, and Latch-Up Conditions

It was previously noted that the analysis of the binary encoded state assignment made full use of *don't care* states.⁷ This allows us to generate a solution that uses the least number of logic gates, but there are additional considerations that must now be discussed in more detail.

The original definition of our coin-operated machine stated that it is only possible for a single coin to be deposited at a time. Assuming this to be true,

⁶ Once again, the author used BOOL (“What’s the point of barking if you have a dog?” as they say in England).

⁷ The concept of *don't care* states was introduced in Chapter 10.

then the *nickel* and *dime* signals will never be assigned 1 values simultaneously. The designer (or a computer program) can use this information to assign *don't care* states to the outputs for any combination of inputs that includes a 1 on both nickel and dime signals.

Additionally, the three binary encoded state variable registers provide eight possible binary patterns, of which only five were used. The analysis above was based on the assumption that *don't care* states can be assigned to the outputs for any combination of inputs that includes one of the unused patterns on the state variables. This assumption also requires further justification.

When the coin-operated machine is first powered-up, each state variable register can potentially initialize with a random logic 0 or 1 value. The controller could therefore power-up with its state variables containing any of the eight possible patterns of 0s and 1s. For some state machines this would not be an important consideration, but this is not true in the case of our coin-operated machine. For example, the controller could power-up in the *20-cents* state, in which case it would immediately dispense a “gizmo” and five cents change. The owner of such a machine may well be of the opinion that this was a less than ideal feature.

Alternatively, the controller could power-up with its state variables in one of the unused combinations. Subsequently, the controller could sequence directly—or via one or more of the other unused combinations—to any of the defined states. In a worst-case scenario, the controller could remain in the unused combination indefinitely or sequence endlessly between unused combinations; these worst-case scenarios are known as *latch-up conditions*.

One method of avoiding latch-up conditions is to assign additional, dummy states to each of the unused combinations and to define state transitions from each of these dummy states to the controller's initialization state of *0-cents*. Unfortunately, in the case of our coin-operated machine, this technique would not affect the fact that the controller could wake up in a valid state other than *0-cents*. An alternative is to provide some additional circuitry to generate a *power-on reset* signal—for example, a single pulse that occurs only when the power is first applied to the machine. The power-on reset can be used to force the state variable registers into the pattern associated with the *0-cents* state. The analysis above assumed the use of such a power-on-reset.

Analog-to-Digital and Digital-to-Analog

As we began our discussions in Chapter 1 by separating the analog and digital views of the world, it seems appropriate to close this section of the book by reuniting them. While some systems operate solely on digital data, others have to interact with the analog world. It may be necessary to convert an analog input into a form that can be manipulated by the digital system, or to transform an output from a digital system into the analog realm. These tasks are performed by analog-to-digital (A/D) and digital-to-analog (D/A) converters, respectively (Figure 13-1).

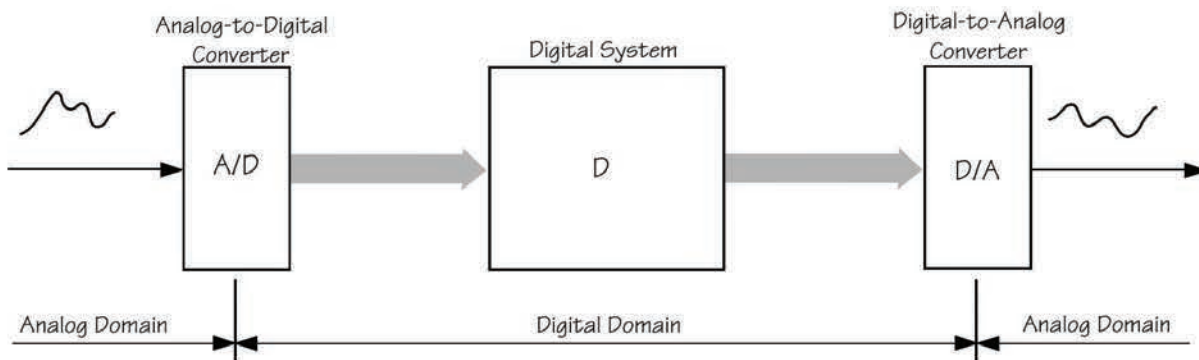


Figure 13-1. Analog-to-digital (A/D) and digital-to-analog (D/A) converters

Analog-to-Digital

A *transducer* is a device that converts input energy of one form into output energy of another. Analog effects can manifest themselves in a variety of different ways such as heat and pressure. In order to be processed by a digital system, the analog quantity must be detected and converted into a suitable form by means of an appropriate transducer called a *sensor*. For example, a microphone is a sensor that detects sound and converts it into a corresponding voltage. The analog-to-digital conversion process can be represented as shown in Figure 13-2.

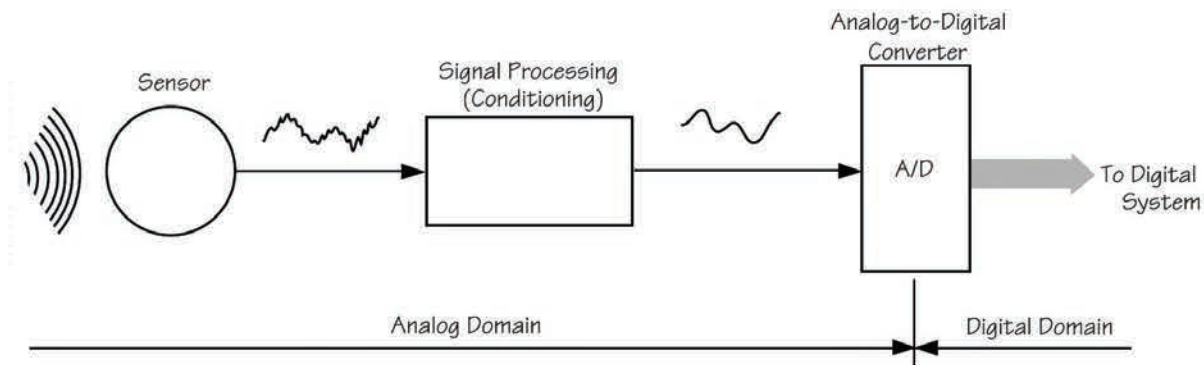


Figure 13-2. Analog-to-digital conversion process

The output from the sensor typically undergoes some form of signal processing such as filtering and amplification before being passed to the A/D converter. This signal processing is generically referred to as *conditioning*. The A/D converter accepts the conditioned analog voltage and converts it into a series of equivalent digital values by *sampling* and *quantization* (Figure 13-3).

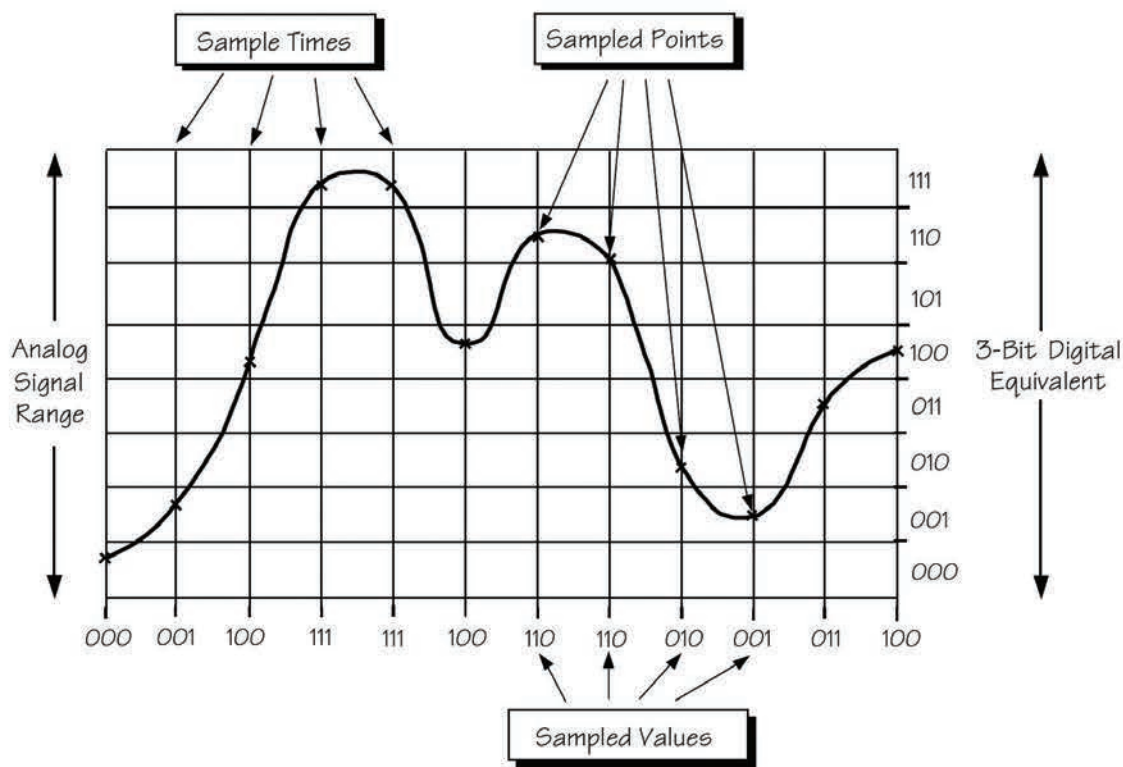


Figure 13-3. The sampling and quantization of a digital signal

The sampling usually occurs at regular time intervals and is triggered by the digital part of the system. The complete range of values that the analog signal can assume is divided into a set of discrete bands or quanta. At each sample time, the A/D converter determines which band the analog signal falls into (this is the “quantization” part of the process) and outputs the equivalent binary code for that band.

The main factor governing the accuracy of the conversion is the number of bands used. For example, a 3-bit code can represent only eight bands, each encompassing 12.5% of the analog signal’s range, while a 12-bit code can represent 4,096 bands, each encompassing 0.025% of the signal’s range.

Digital-to-Analog

A D/A converter accepts a digital code and transforms it into a corresponding analog current or voltage suitable for use by means of an appropriate transducer called an *actuator*. For example, a loudspeaker is an actuator that converts an electrical signal into sound. The digital-to-analog conversion process can be represented as shown in Figure 13-4.

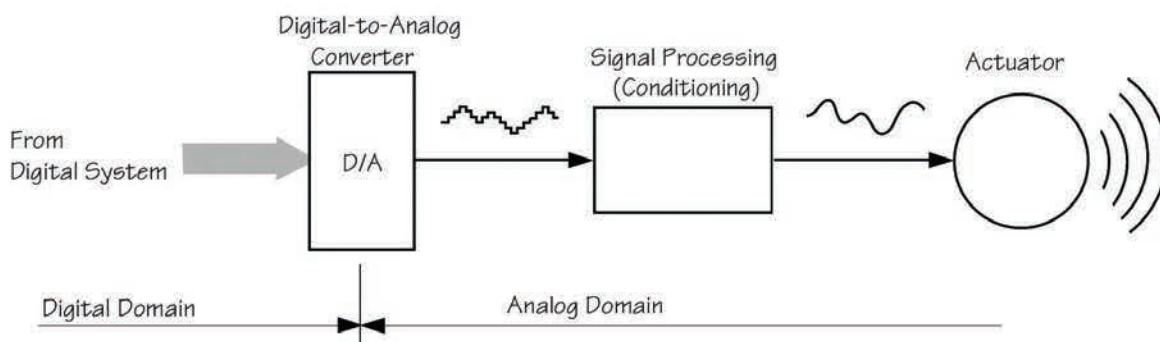


Figure 13-4. Digital-to-analog conversion process

The conversions usually occur at regular time intervals and are triggered by a clock signal from the digital part of the system. The output from the D/A converter typically undergoes some form of conditioning before being passed to the actuator. For example, in the case of an audio system, the “staircase-like” signal coming out of the D/A converter will be “smoothed” before being passed to an amplifier (not shown in Figure 13-4) and, ultimately, to the loudspeaker.

Integrated Circuits (ICs)

In the 1950s, transistors and other electronic components were available only in individual packages. These discrete components were laid out on a circuit board and hand connected using separate wires. At that time, an electronic gate capable of storing a single binary bit of data cost more than \$2. By comparison, in the early 1990s, enough gates to store 5,000 bits of data cost less than a cent. This vast reduction in price was primarily due to the invention of the *integrated circuit* (IC).¹

A functional electronic circuit requires transistors, resistors, diodes, etc. and the connections between them. A monolithic integrated circuit (the “monolithic” qualifier is usually omitted) has all of these components formed on the surface layer of a sliver, or chip, of a single piece of semiconductor; hence the term *monolithic*, meaning “seamless.” Although a variety of semiconductor materials are available, the most commonly used is silicon, and integrated circuits are popularly known as “silicon chips.” Unless otherwise noted, the remainder of these discussions will assume integrated circuits based on silicon as the semiconductor.

An Overview of the Fabrication Process

The construction of integrated circuits requires one of most exacting production processes ever developed. The environment must be at least a thousand times cleaner than that of an operating theater, and impurities in materials have to be so low as to be measured in parts per billion.² The process begins with the growing of a single crystal of pure silicon in the form of a cylinder with a diameter that can be anywhere up to 300 mm.³ The cylinder is cut into paper-thin slices called *wafers*, which are approximately 0.2 mm thick (Figure 14-1).

¹ In conversation, IC is pronounced by spelling it out as “I-C”.

² If you took a bag of flour and added a grain of salt, this would be impure by comparison.

³ This 300 mm value was true as of 2002. However, in 1995 (when the first edition of this tome hit the streets), the maximum diameter was 200 mm, so who knows what it will be in the future?

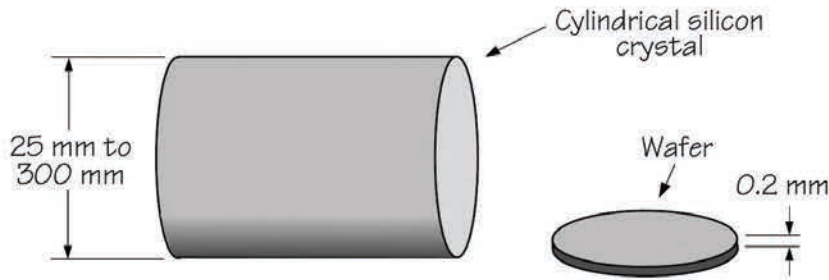


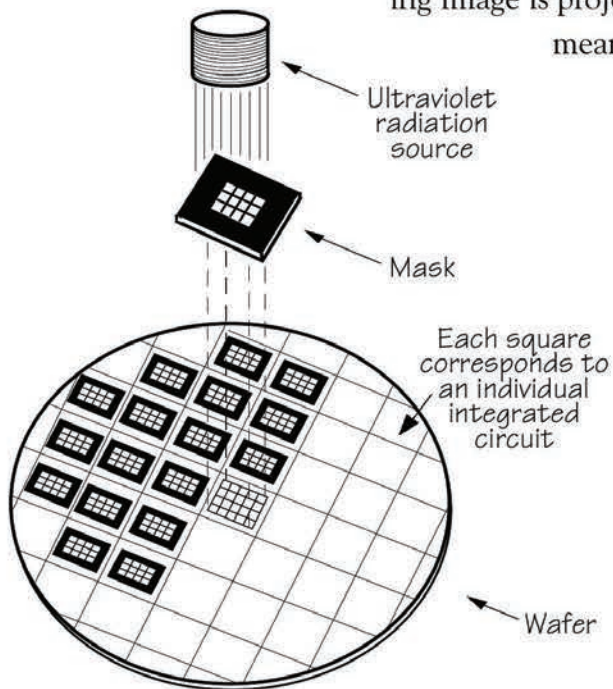
Figure 14-1. Creating silicon wafers

The thickness of the wafer is determined by the requirement for sufficient mechanical strength to allow it to be handled without damage. The actual thickness necessary for the creation of the

electronic components is less than $10\text{ }\mu\text{m}$ (ten-millionths of a meter). After the wafers have been sliced from the cylinder, they are polished to a smoothness rivaling the finest mirrors.

The most commonly used fabrication process is *optical lithography*, in which ultraviolet light (UV) is passed through a stencil-like⁴ object called a *photo-mask*, or just *mask* for short. This square or rectangular mask carries patterns formed by areas that are either transparent or opaque to ultraviolet frequencies (similar in concept to a black and white photographic negative) and the result-

ing image is projected onto the surface of the wafer. By means of some technical wizardry that we'll



consider in the next section, we can use the patterns of ultraviolet light to grow corresponding structures in the silicon. The simple patterns shown in the following diagrams were selected for reasons of clarity; in practice, a mask can contain millions of fine lines and geometric shapes (Figure 14-2).

Each wafer can contain hundreds or thousands of identical integrated circuits. The pattern projected onto the wafer's surface corresponds to a

Figure 14-2. The opto-lithographic step-and-repeat process

⁴ The term "stencil" comes from the Middle English word *stencelled*, meaning "adorned brightly."

single integrated circuit, which is typically in the region of $1\text{mm} \times 1\text{mm}$ to $10\text{mm} \times 10\text{mm}$, but may be even larger. After the area corresponding to one integrated circuit has been exposed, the wafer is moved and the process is repeated until the pattern has been replicated across the whole of the wafer's surface. This technique for duplicating the pattern is called a *step-and-repeat* process.

As we shall see, multiple layers are required to construct the transistors (and other components), where each layer requires its own unique mask. Once all of the transistors have been created, similar techniques are used to lay down the tracking (wiring) layers that connect the transistors together.

A More Detailed Look at the Fabrication Process

To illustrate the manufacturing process in more detail, we will consider the construction of a single NMOS transistor occupying an area far smaller than a speck of dust. For reasons of electronic stability, the majority of processes begin by lightly doping the entire wafer to form either N-type or, more commonly, P-type silicon. However, for the purposes of this discussion, we will assume a process based on a pure silicon wafer (Figure 14-3).

Assume that the small area of silicon shown here is sufficient to accommodate a single transistor in the middle of one of the integrated circuits residing somewhere on the wafer. During the fabrication process the wafer is often referred to as the *substrate*, meaning “base layer.” A common first stage is to either grow or deposit a thin layer of silicon dioxide (glass) across the entire surface of the wafer by exposing it to oxygen in a high-temperature oven (Figure 14-4).

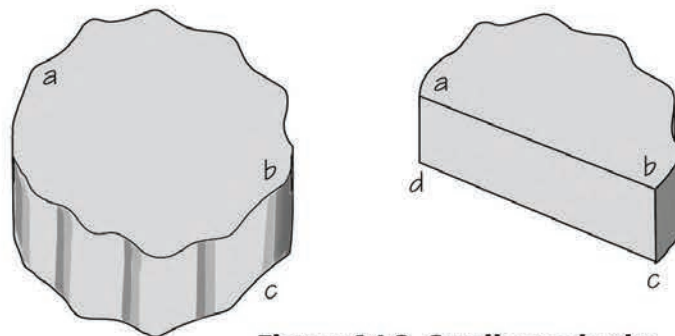


Figure 14-3. Small area in the middle of the silicon wafer

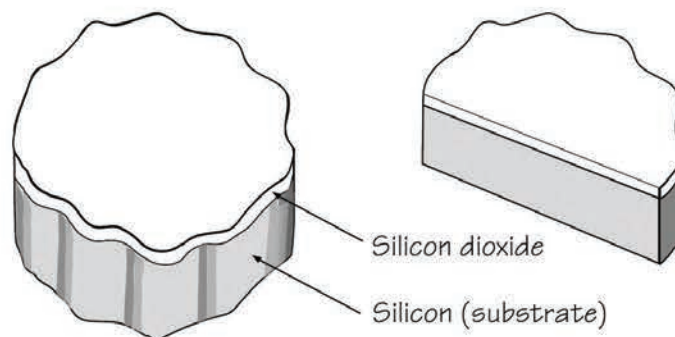


Figure 14-4. Grow or deposit a layer of silicon dioxide

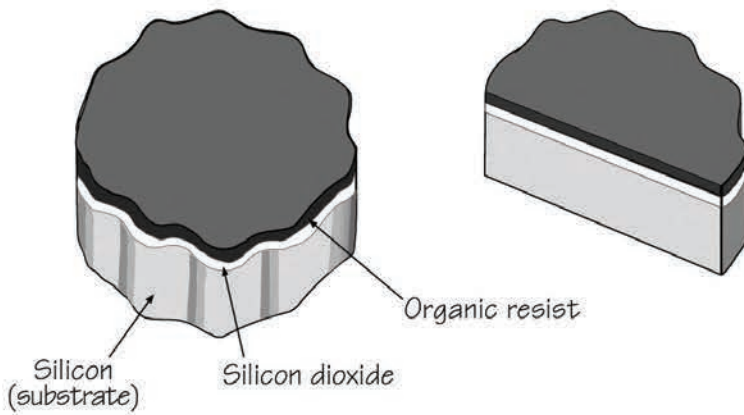


Figure 14-5. Apply a layer of organic resist

After the wafer has cooled, it is coated with a thin layer of organic resist,⁵ which is first dried and then baked to form an impervious layer (Figure 14-5).

A mask is created and ultraviolet light is applied. The ionizing ultraviolet radiation passes through the transparent areas of the mask into the resist, silicon

dioxide, and silicon. The ultraviolet breaks down the molecular structure of the resist, but does not have any effect on the silicon dioxide or the pure silicon (Figure 14-6).

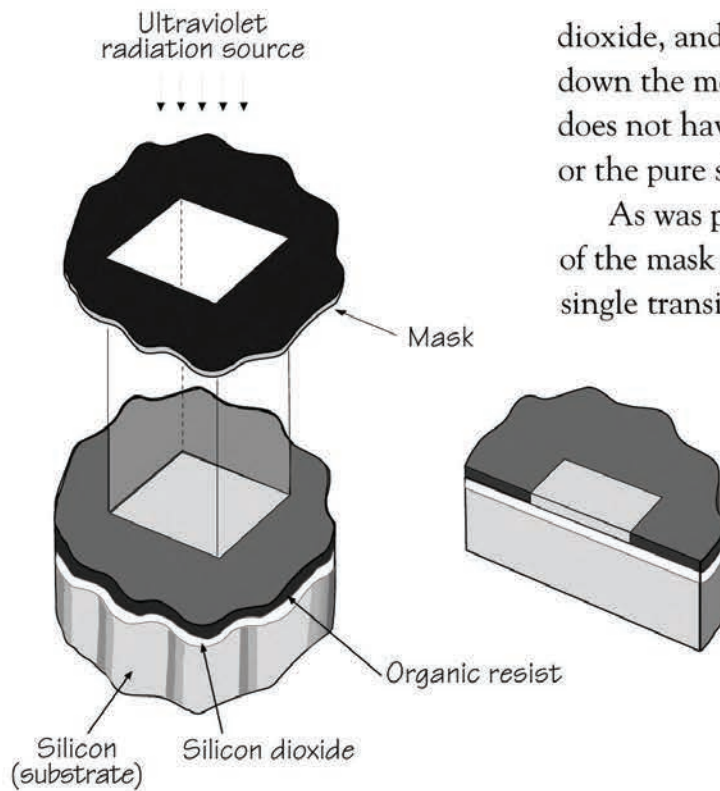


Figure 14-6. The exposed resist is degraded by the ultraviolet light

As was previously noted, the small area of the mask shown here is associated with a single transistor. The full mask for an integrated circuit can consist of millions of similar patterns. After the area under the mask has been exposed, the wafer is moved, and the process is repeated until the pattern has been replicated across the wafer's entire surface, once for each integrated circuit. The wafer is then bathed in an organic solvent to dissolve the degraded resist. Thus, the

⁵ The term "organic" is used because this type of resist is a carbon-based compound, and carbon is the key element for life as we know it.

pattern on the mask has been transferred to a series of corresponding patterns in the resist (Figure 14-7).

A process in which ultraviolet light passing through the transparent areas of the mask causes the resist to be degraded is known as a *positive-resist* process; *negative-resist* processes are also available. In a negative-resist process, the ultraviolet radiation passing through the transparent areas of the mask is used to cure the resist, and the remaining uncured areas are then removed using an appropriate solvent.

After the unwanted resist has been removed, the wafer undergoes a process known as *etching*, in which an appropriate solvent is

used to dissolve any exposed silicon dioxide without having any effect on the organic resist or the pure silicon (Figure 14-8).

The remaining resist is then removed using an appropriate solvent, and the wafer is placed in a high temperature oven where it is exposed to a gas containing the selected dopant (a P-type dopant in this case). The atoms in the gas diffuse into the substrate resulting in a region of doped silicon (Figure 14-9).⁶

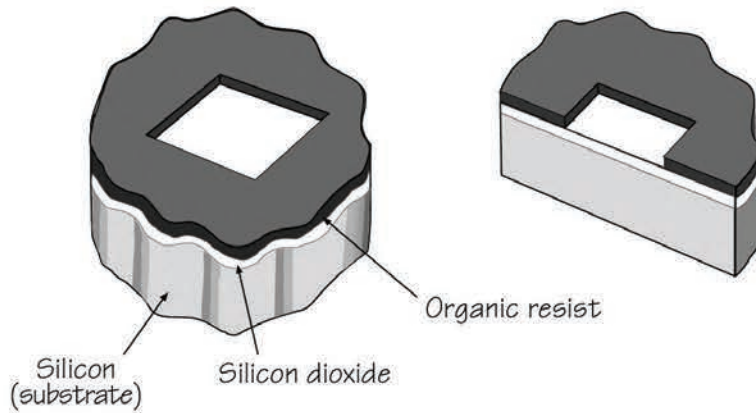


Figure 14-7. The degraded resist is dissolved with an organic solvent

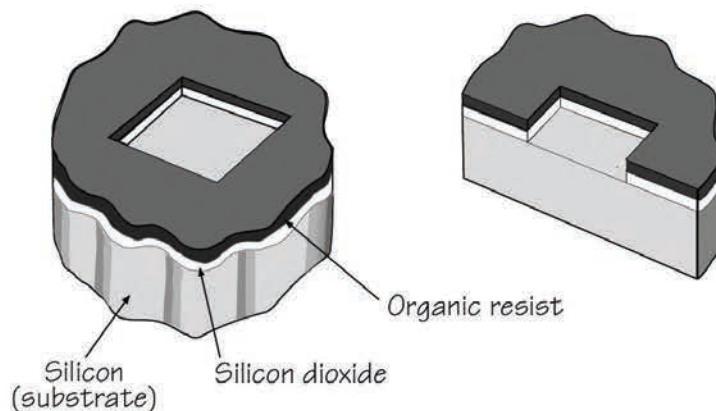


Figure 14-8. Etch the exposed silicon dioxide

⁶ In some processes, diffusion is augmented with *ion implantation* techniques, in which beams of ions are directed at the wafer to alter the type and conductivity of the silicon in selected regions.

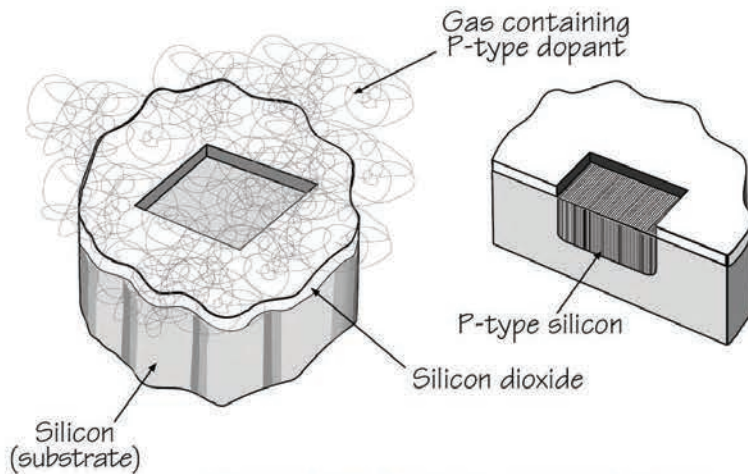


Figure 14-9. Dope the exposed silicon

The remaining silicon dioxide layer is removed by means of an appropriate solvent that doesn't affect the silicon substrate (including the doped regions). Then additional masks and variations on the process are used to create two N-type diffusion regions, a gate electrode, and a

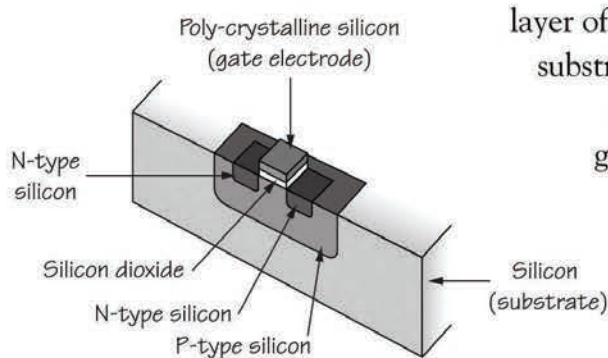


Figure 14-10. Add n-type diffusion regions and the gate electrode

layer of insulating silicon dioxide between the substrate and the gate electrode (Figure 14-10).

In the original MOS technologies, the gate electrode was metallic: hence the “metal-oxide semiconductor” appellation. In modern processes, however, the gate electrode is formed from *poly-crystalline silicon* (often abbreviated to *polysilicon* or even just *poly*), which is also a good conductor.

The N-type diffusions form the transistor's *source* and *drain* regions (you might wish to refer back to Figure 4-9a in Chapter 4 to refresh your memory at this point). The gap between the source and drain is called the *channel*. To provide a sense of scale, the length of the channel may be in the order of $0.1\ \mu\text{m}$ (one-tenth of one-millionth of a meter), and the thickness of the silicon dioxide layer between the gate electrode and the substrate may be in the order of $0.05\ \mu\text{m}$ (see also the discussions on *device geometries* later in this chapter).

Another layer of insulating silicon dioxide is now grown or deposited across the surface of the wafer. Using similar lithographic techniques to those described above, holes are etched through the silicon dioxide in areas in which it is desired to make connections, and a metalization layer of aluminum interconnections (think of them as wires) called *tracks* is deposited (Figure 14-11).

The end result is an NMOS transistor; a logic 1 on the track connected to the *gate* will turn the transistor ON, thereby enabling current to flow between its *source* and *drain* terminals. An equivalent PMOS transistor could have been formed by exchanging the P-type and N-type diffusion regions. By varying the structures

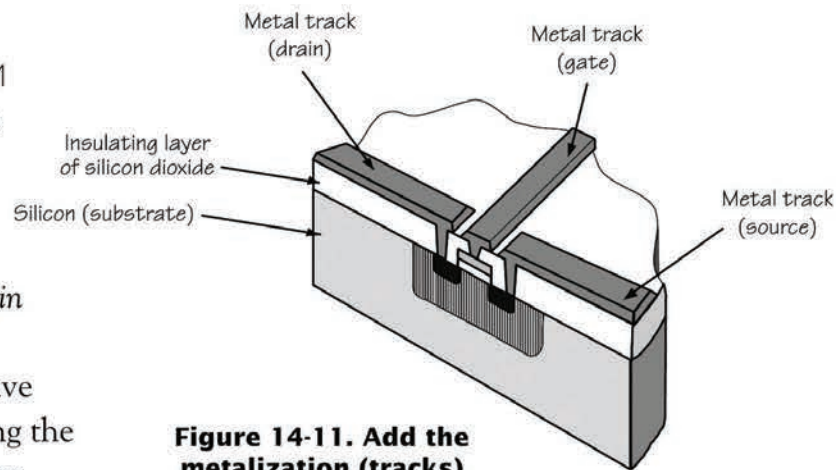


Figure 14-11. Add the metalization (tracks)

created by the masks, components such as resistors and diodes can be fabricated at the same time as the transistors. The tracks are used to connect groups of transistors to form primitive logic gates and to connect groups of these gates to form more complex functions.

An integrated circuit contains three distinct levels of conducting material: the *diffusion* layer at the bottom, the *polysilicon* layers in the middle, and the *metalization* layers at the top. In addition to forming components, the diffusion layer may also be used to create embedded wires. Similarly, in addition to forming gate electrodes, the polysilicon may also be used to interconnect components. There may be several layers of polysilicon and several layers of metalization, with each pair of adjacent layers separated by an insulating layer of silicon dioxide. The layers of silicon dioxide are selectively etched with holes known as *vias*, which allow connections to be made between the various tracking layers.

Early integrated circuits typically supported only two layers of metalization. The tracks on the first layer predominantly ran in a “North-South” direction, while the tracks on the second predominantly ran “East-West.”⁷ As the number

⁷ In 2001, a group of companies announced a new chip interconnect concept called *X Architecture* (www.xinitiative.org) in which logic functions on chips are wired together using diagonal tracks (as opposed to traditional North-South and East-West tracking layers). Initial evaluations apparently show that this diagonal interconnect strategy can increase chip performance by 10% and reduce power consumption by 20%. However, it may take awhile before design tools and processes catch up . . . watch this space!

of transistors increased, engineers required more and more tracking layers. The problem is that when a layer of insulating silicon dioxide is deposited over a tracking layer, you end up with slight “bumps” where the tracks are (like snow falling over a snoozing polar bear—you end up with a bump).

After a few tracking layers, the bumps are pronounced enough that you can’t continue. The answer is to re-planarize the wafer (smooth the bumps out) after each tracking and silicon dioxide layer combo has been created. This is achieved by means of a process called *chemical mechanical polishing* (CMP), which returns the wafer to a smooth, flat surface before the next tracking layer is added. With manufacturers using this process, high-end silicon chips could support up to eight tracking layers by 2002.

Relatively large areas of aluminum called *pads* are constructed at the edges of each integrated circuit for testing and connection purposes. Some of the pads are used to supply power to the device, while the rest are used to provide input and output signals (Figure 14-12).

The pads can be connected to the internal components using the diffusion, polysilicon, or metalization layers. In a step known as *overglassing*, the entire surface of the wafer is coated with a final *barrier* layer (or *passivation* layer) of silicon dioxide or silicon nitride, which provides physical protection for the underlying circuits from moisture and other contaminants. One more lithographic step is required to pattern holes in the barrier layer to allow connections to be made to the pads. In some cases, additional metalization may be deposited on the pads to raise them fractionally above the level of the barrier layer. Augmenting the pads in this way is known as *silicon bumping*.

The entire fabrication process requires numerous lithographic steps, each involving an individual mask and layer of resist to selectively expose different parts of the wafer.

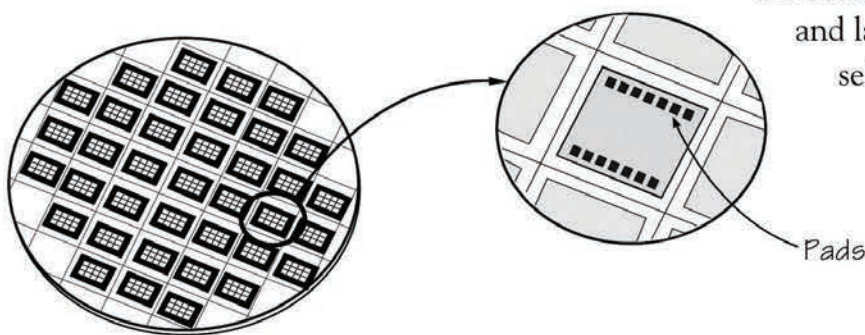


Figure 14-12. Power and signal pads

The Packaging Process

The individual integrated circuits are tested while they are still part of the wafer in a process known as *wafer probing*. An automated tester places probes on the device's pads, applies power to the power pads, injects a series of signals into the input pads, and monitors the corresponding signals returned from the output pads. Each integrated circuit is tested in turn, and any device that fails the tests is automatically tagged with a splash of dye for subsequent rejection. The *yield* is the number of devices that pass the tests as a percentage of the total number fabricated on that wafer.

The completed circuits, known as *die*,⁸ are separated by marking the wafer with a diamond scribe and fracturing it along the scribed lines (much like cutting a sheet of glass or breaking up a Kit Kat[®] bar) (Figure 14-13).

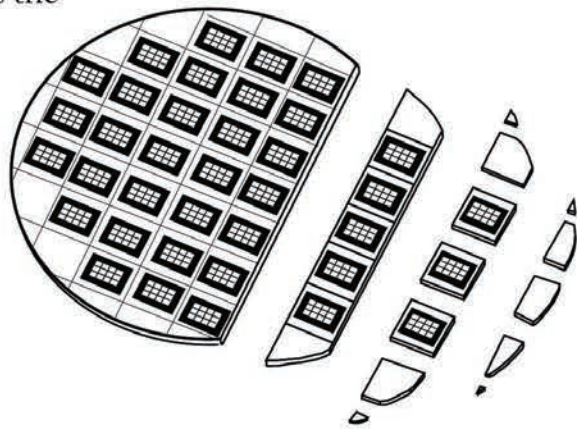


Figure 14-13. Die separation

Following separation, the majority of the die are packaged individually. Since there are almost as many packaging technologies as there are device manufacturers, we will initially restrain ourselves to a relatively traditional process. First, the die is attached to a metallic lead frame using an adhesive (Figure 14-14).

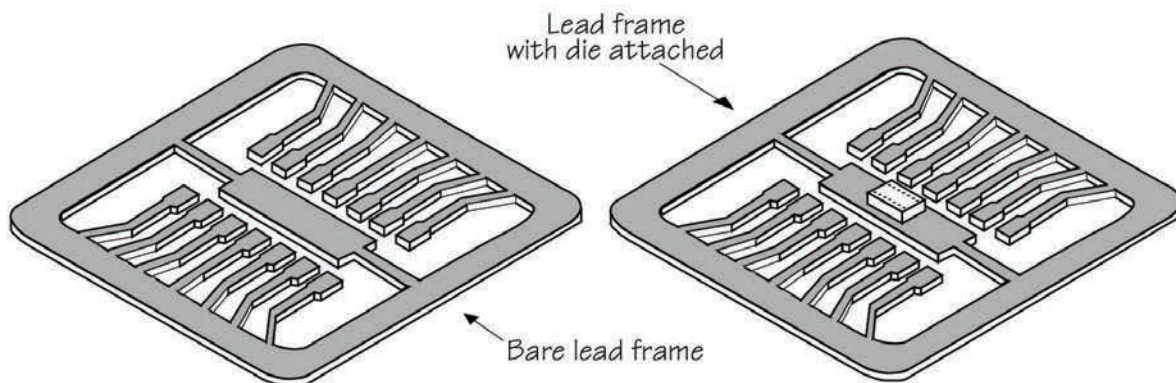


Figure 14-14. Die attached to lead frame

⁸ The plural of die is also die (in much the same way that “a shoal of herring” is the plural of “herring”).

One of the criteria used when selecting the adhesive is its ability to conduct heat away from the die when the device is operating. An automatic wire bonding tool connects the pads on the die to the leads on the lead frame with wire bonds finer than a human hair.⁹ The whole assembly is then encapsulated in a block of plastic or epoxy (Figure 14-15).

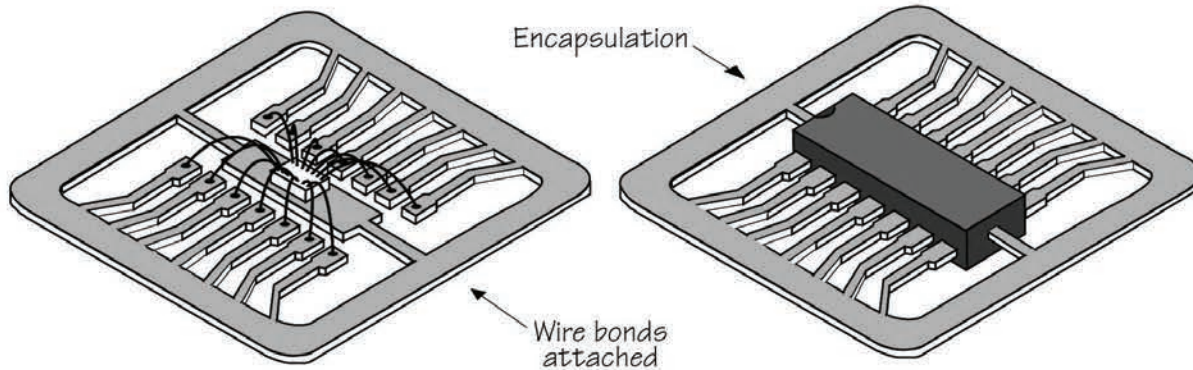


Figure 14-15. Wire bonding and encapsulation

A dimple or notch is formed at one end of the package so that the users will know which end is which. The unused parts of the lead frame are cut away and the device's leads, or pins, are shaped as required; these operations are usually performed at the same time (Figure 14-16).

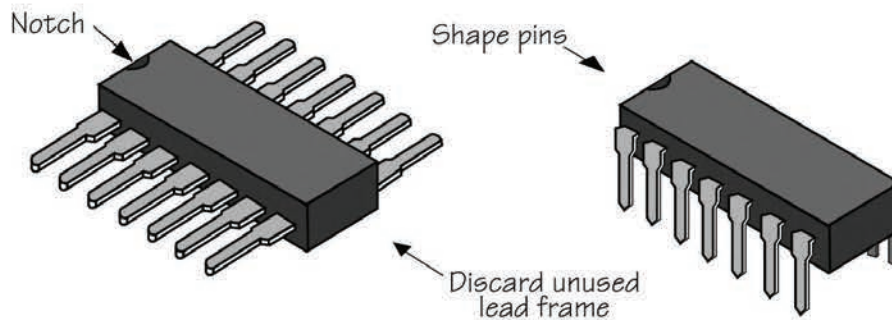


Figure 14-16. Discard unused lead frame and shape pins

⁹ Human hairs range in thickness from around 0.07 mm to 0.1 mm. A hair from a typical blond lady's head is approximately 0.075 mm (three quarters of one tenth of a millimeter) in diameter. By comparison, integrated circuit bonding wires are typically one-third this diameter, and they can be even thinner.

An individually packaged integrated circuit consists of the die and its connections to the external leads, all encapsulated in the protective package. The package protects the silicon from moisture and other impurities and helps to conduct heat away from the die when the device is operating.

There is tremendous variety in the size and shape of packages. A rectangular device with pins on two sides, as illustrated here, is called a *dual in-line (DIL)* package. A standard 14-pin packaged device is approximately 18 mm long by 6.5 mm wide by 2.5 mm deep, and has 2.5 mm spaces between pins. An equivalent *small outline package (SOP)* could be as small as 4 mm long by 2 mm wide by 0.75 mm deep, and have 0.5 mm spaces between pins. Other packages can be square and have pins on all four sides, and some have an array of pins protruding from the base.

The shapes into which the pins are bent depend on the way the device is intended to be mounted on a circuit board. The package described above has pins that are intended to go all the way through the circuit board using a mounting technique called *lead through hole (LTH)*. By comparison, the packages associated with a technique called *surface mount technology (SMT)* have pins that are bent out flat, and which attach to one side (surface) of the circuit board (an example of this is shown in Chapter 18).

It's important to note that the example shown above reflects a very simple packaging strategy for a device with very few pins. By 2002, some integrated circuits had as many as 1,000 pins (with 2,000- and 4,000-pin devices on the horizon). This multiplicity of pins requires a very different approach. In one technique known as *solder bump bonding*, for example, the pads on the die are not restricted to its periphery, but are instead located over the entire face of the die. A minute ball of solder is then attached to each pad, and the die is flipped over and attached to the package substrate (this is referred to as a "flip-chip" technique). Each pad on the die has a corresponding pad on the package substrate, and the package-die combo is heated so as to melt the solder balls and form good electrical connections between the die and the substrate (Figure 14-17).

Eventually, the die will be encapsulated in some manner to protect it from the outside world. The package's substrate itself may be made out of the same material as a printed circuit board, or out of ceramic, or out of some even more esoteric material. Whatever its composition, the substrate will contain multiple

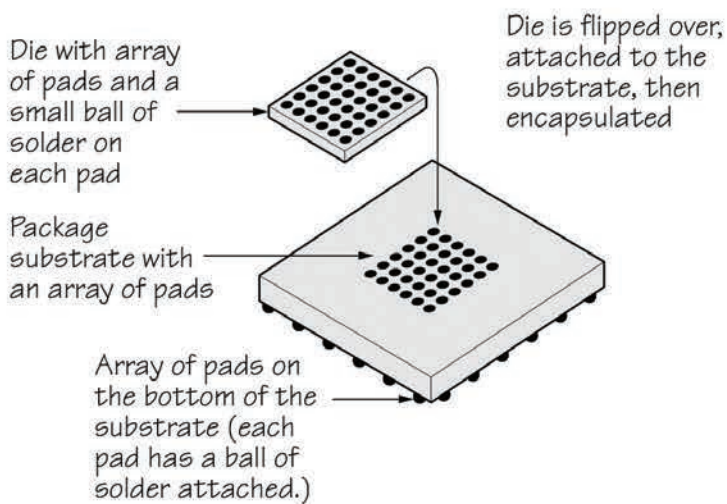


Figure 14-17. A solder bump bonded ball grid array packaging technique

internal wiring layers that connect the pads on the upper surface with pads (or pins) on the lower surface. The pads (or pins) on the lower surface (the side that actually connects to the circuit board) will be spaced much farther apart—relatively speaking—than the pads that connect to the die.

At some stage the package will have to be attached to a circuit board. In one technique known as a *ball*

grid array (BGA), the package has an array of pads on its bottom surface, and a small ball of solder is attached to each of these pads. Each pad on the package will have a corresponding pad on the circuit board, and heat is used to melt the solder balls and form good electrical connections between the package and the board.

Modern packaging technologies are extremely sophisticated. For example, by 2002, some ball grid arrays had pins spaced 0.3 mm (one third of a millimeter) apart! In the case of *chip-scale packages* (CSP), the package is barely larger than the die itself. In the early 1990s, some specialist applications began to employ a technique known as *die stacking*, in which several bare die are stacked on top of each other to form a sandwich. The die are connected together and then packaged as a single entity.

As was previously noted, there are a wide variety of integrated packaging styles. There are also many different ways in which the die can be connected to the package. We will introduce a few more of these techniques in future chapters.¹⁰

¹⁰ Additional packaging styles and alternative mounting strategies are presented in the discussions on circuit boards (Chapter 18), hybrids (Chapter 19), and multichip modules (Chapter 20).

Integrated Circuits versus Discrete Components

The tracks linking components inside an integrated circuit have widths measured in fractions of a millionth of a meter and lengths measured in millimeters. By comparison, the tracks linking components on a circuit board are orders of magnitude wider and have lengths measured in tens of centimeters. Thus, the transistors used to drive tracks inside an integrated circuit can be much smaller than those used to drive their circuit board equivalents, and smaller transistors use less power. Additionally, signals take a finite time to propagate down a track, so the shorter the track, the faster the signal.

A single integrated circuit can contain tens (sometimes hundreds) of millions of transistors. A similar design based on discrete components would be tremendously more expensive in terms of price, size, operating speed, power requirements, and the time and effort required to design and manufacture the system. Additionally, every solder joint on a circuit board is a potential source of failure, which affects the reliability of the design. Integrated circuits reduce the number of solder joints and hence improve the reliability of the system.

In the past, an electronic system was typically composed of a number of integrated circuits, each with its own particular function (say a microprocessor, a communications function, some memory devices, etc.). For many of today's high-end applications, however, electronics engineers are combining all of these functions on a single device, which may be referred to as a *system-on-chip* (SoC).

Different Types of ICs

The first integrated circuit—a simple phase shift oscillator—was constructed in 1958.¹¹ Since that time, a plethora of different device types have appeared on the scene. There are far too many different integrated circuit types for us to cover in this book, but some of the main categories—along with their approximate dates of introduction—are shown in Figure 14-18.¹²

¹¹ The first integrated circuits typically contained around six transistors. By the latter half of the 1960s, devices containing around 100 transistors were reasonably typical.

¹² The white portions of the timeline bars in this figure indicate that although early incarnations of these technologies may have been available, they perhaps hadn't been enthusiastically received during this period. For example, Xilinx introduced the world's first FPGA as early as 1984, but many design engineers didn't really become interested in these little rascallions until the late 1980s.

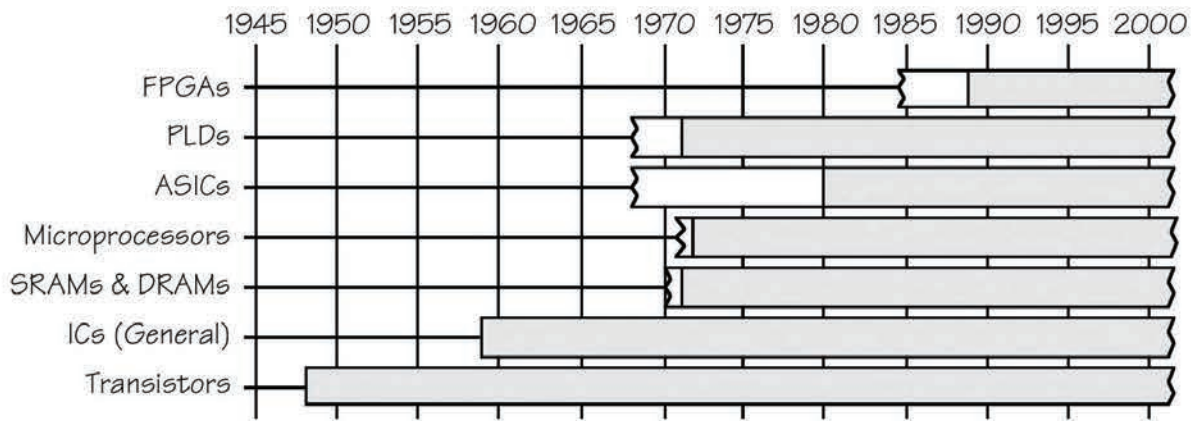


Figure 14-18. Timeline of device introductions (dates are approximate)

Memory devices (in particular SRAMs and DRAMs) are introduced in Chapter 15; programmable integrated circuits (PLDs and FPGAs) are presented in Chapter 16; and application-specific integrated circuits (ASICs) are discussed in Chapter 17.

Technology Considerations

Transistors are available in a variety of flavors called *families* or *technologies*. One of the first to be invented was the *bipolar junction transistor (BJT)*, which was the mainstay of the industry for many years. If bipolar transistors are connected together in a certain way, the resulting logic gates are classed as *transistor-transistor logic (TTL)*. An alternative method of connecting the same transistors results in logic gates classed as *emitter-coupled logic (ECL)*. Another family called *metal-oxide semiconductor field-effect transistors (MOSFETs)* were invented some time after bipolar junction transistors. *Complementary metal-oxide semiconductor (CMOS)* logic gates are based on NMOS and PMOS MOSFETs connected together in a complementary manner.

Logic gates constructed in TTL are fast and have strong drive capability, but consume a relatively large amount of power. Logic gates implemented in CMOS are a little slower than their TTL equivalents and have weaker drive capability, but their static (non-switching) power consumption is extremely low. Technology improvements continue to yield lower-power TTL devices and higher-speed CMOS devices. Logic gates built in ECL are substantially faster than their TTL counterparts, but consume correspondingly more power.

Finally, gates fabricated using the gallium arsenide (GaAs) semiconductor as a substrate are approximately eight times faster than their silicon equivalents, but they are expensive to produce, and so are used for specialist applications only.

If an integrated circuit containing millions of transistors were constructed entirely from high-power transistors, it could literally consume enough power to incinerate itself (or at least melt itself down into an undistinguished puddle of gunk). As a compromise, some integrated circuits use a combination of technologies. For example, the bulk of the logic gates in a device may be implemented in low-power CMOS, but the gates driving the output pins may be constructed from high-drive TTL. A more extreme example is that of *BiCMOS* (*Bipolar CMOS*), in which the function of every primitive logic gate is implemented in low-power CMOS, but the output stage of each gate uses high-drive bipolar transistors.

Supply Voltages

Towards the end of the 1980s and the beginning of the 1990s, the majority of circuits using TTL, CMOS, and BiCMOS devices were based on a 5.0-volt supply. However, increasing usage of portable personal electronics such as notebook computers and cellular telephones began to drive the requirement for devices that consume and dissipate less power. One way to reduce power consumption is to lower the supply voltage, so by the mid-to-late 1990s, the most common supplies were 3.3 volts for portable computers and 3.0 volts for communication systems. By 2002, some specialist applications had plunged to 1.8 volts, with even lower supplies on the horizon. Unfortunately, lowering the supply voltage can drastically affect the speed of traditional technologies and greatly lessens any speed advantages of BiCMOS over CMOS. A relatively new low-voltage contender that appeared in the latter half of the 1990s was BiNMOS, in which complex combinations of bipolar and NMOS transistors are used to form sophisticated output stages providing both high speed and low static power dissipation.

Equivalent Gates

One common metric used to categorize an integrated circuit is the number of logic gates it contains. However, difficulties may arise when comparing devices, as each type of logic function requires a different number of transistors.

This leads to the concept of an *equivalent gate*, whereby each type of logic function is assigned an equivalent gate value, and the relative complexity of an integrated circuit is judged by summing its equivalent gates. Unfortunately, the definition of an equivalent gate can vary, depending on whom one is talking to. A reasonably common convention is for a 2-input NAND to represent one equivalent gate. A more esoteric convention defines an ECL equivalent gate as being “one-eleventh the minimum logic required to implement a single-bit full-adder,” while some vendors define an equivalent gate as being equal to an arbitrary number of transistors based on their own particular technology. The best policy is to establish a common frame of reference before releasing a firm grip on your hard-earned lucre.

The acronyms SSI, MSI, LSI, VLSI, and ULSI represent *Small-*, *Medium-*, *Large-*, *Very-Large-*, and *Ultra-Large-Scale Integration*, respectively. By one convention, the number of gates represented by these terms are: SSI (1-12), MSI (13-99), LSI (100-999), VLSI (1,000-999,999), and ULSI (1,000,000 or more).

Device Geometries

Integrated circuits are also categorized by their *geometries*, meaning the size of the structures created on the substrate. For example, a $1\text{ }\mu\text{m}^{13}$ CMOS device has structures that measure one-millionth of a meter. The structures typically embraced by this description are the width of the tracks and the length of the channel between the source and drain diffusion regions; the dimensions of other features are derived as ratios of these structures.

Geometries are continuously shrinking as fabrication processes improve. In 1990, devices with $1\text{ }\mu\text{m}$ geometries were considered to be state of the art, and many observers feared that the industry was approaching the limits of manufacturing technology, but geometries continued to shrink regardless:

Anything below $0.5\text{ }\mu\text{m}$ is referred to as *deep-submicron (DSM)*, and at some point that isn't particularly well defined (or is defined differently by different people) we move into the realm of *ultra-deep-submicron (UDSM)*.

¹³ The μ symbol stands for “micro” from the Greek *micros*, meaning “small” (hence the use of μP as an abbreviation for microprocessor). In the metric system, μ stands for “one millionth part of,” so $1\text{ }\mu\text{m}$ means one millionth of a meter.

Year	Geometry
1990	1.00 μm
1992	0.80 μm
1994	0.50 μm
1996	0.35 μm
1997	0.25 μm
1999	0.18 μm
2000	0.13 μm
2001	0.10 μm
2002	0.09 μm

With devices whose geometries were 1 μm and higher, it was relatively easy to talk about them in conversation. For example, one might say “I’m working with a one micron technology.” But things started to get a little awkward when we dropped below 1 μm , because it’s a bit of a pain to have to keep on saying things like “zero point one-three microns.” For this reason, it’s become common to talk in terms of “nano,” where one nano (short for “nanometer”) equates to one thousandth of a micron—that is, one thousandth of one millionth

of a meter. Thus, when referring to a 0.13 μm technology, instead of mumbling “zero point one-three microns,” you would now proclaim “one hundred and thirty nano.” Of course both of these mean exactly the same thing, but if you want to talk about this sort of stuff, it’s best to use the vernacular of the day and present yourself as hip and trendy as opposed to an old fuddy-duddy from the last millennium.

While smaller geometries result in lower power consumption and higher operating speeds, these benefits do not come without a price. Submicron logic gates exhibit extremely complex timing effects, which make corresponding demands on designers and design systems. Additionally, all materials are naturally radioactive to some extent, and the materials used to package integrated circuits can spontaneously release alpha particles. Devices with smaller geometries are more susceptible to the effects of noise, and the alpha decay in packages can cause corruption of the data being processed by deep-submicron logic gates. Deep-submicron technologies also suffer from a phenomenon known as *subatomic erosion* or, more correctly, *electromigration*, in which the structures in the silicon are eroded by the flow of electrons in much the same way as land is eroded by a river.

What Comes After Optical Lithography?

Although new techniques are constantly evolving, technologists can foresee the limits of miniaturization that can be practically achieved using optical lithography. These limits are ultimately dictated by the wavelength of ultraviolet radiation. The technology has now passed from using standard *ultraviolet* (UV)

to *extreme ultraviolet (EUV)*, which is just this side of soft X-rays in the electromagnetic spectrum. One potential alternative is true X-ray lithography, but this requires an intense X-ray source and is considerably more expensive than optical lithography. Another possibility is *electron beam lithography*, in which fine electron beams are used to draw extremely high-resolution patterns directly into the resist without a mask. Electron beam lithography is sometimes used for custom and prototype devices, but it is much slower and more expensive than optical lithography. Thus, for the present, it would appear that optical lithography—in the form of *extreme ultraviolet lithography (EUVL)*—will continue to be the mainstay of mass-produced integrated circuits.

How Many Transistors?

The geometry table presented on the previous page reflects commercially available processes (although widespread adoption typically takes some time), but experimental processes in the laboratories are much further advanced. For example, in December 2000, Intel announced that they had constructed an incredibly small, unbelievably fast CMOS transistor only 0.03 μm (30 nano) in size.

In the first half of 2002, Intel announced its McKinley microprocessor—an integrated circuit based on a 0.13 μm (130 nano) process containing more than 200 million transistors! And by the summer of 2002, Intel had announced a test chip based on a 0.09 μm (90 nano) process that contained 330 million transistors. Some observers are predicting that, using Intel's 0.03 μm (30 nano) process, by 2005, we could have 500 million transistors on a single chip running at 10 GHz¹⁴ with a supply voltage of less than one volt. And by 2010, we could be looking at more than 1.8 billion transistors on a chip! At this level of processing power, we will soon have the capabilities required to create Star Trek-style products like a universal real-time translator.^{15,16}

¹⁴ The unit of frequency is the Hertz (Hz). One gigahertz (1 GHz) means “one thousand million cycles per second.”

¹⁵ Speaking of Star Trek, a company called Time Domain (www.timedomain.com) based in Huntsville, Alabama, USA is using ultra wideband wireless technology to create an incredibly low power, Tricorder-like handheld radar. This will allow police, firefighters, and rescue workers to “see through walls” and, for example, detect earthquake victims trapped under piles of rubble.

¹⁶ There are predictions of geometries as low as 0.009 μm (9 nano) becoming available somewhere between 2024 and 2028, at which point the author's mind boggles!

Moore's Law

In 1965, Gordon Moore (who was to co-found Intel Corporation in 1968) was preparing a speech that included a graph illustrating the growth in the performance of memory ICs. While plotting this graph, Moore realized that new generations of memory devices were released approximately every 18 months, and that each new generation of devices contained roughly twice the capacity of its predecessor.

This observation subsequently became known as *Moore's Law*, and it has been applied to a wide variety of electronics trends. These include the number of transistors that can be constructed in a certain area of silicon (the number doubles approximately every 18 months), the price per transistor (which follows an inverse Moore's Law curve and halves approximately every 18 months), and the performance of microprocessors (which again doubles approximately every 18 months).

Liquid crystal displays are often used for personal electronic appliances such as notebook computers. These displays are arranged as a matrix of points called *pixels*, each of which can be light or dark, and each of which is controlled by a transistor. Up until now, these displays have typically been fabricated as a sandwich of liquid crystal between two slices of glass. However, the following interesting little snippet was reported in the November 1994 edition of the IEEE Spectrum magazine:

"Researchers at the French National Center for Scientific Research, in Thiais, say that they have developed an all-plastic, no-metal transistor. Using modern printing techniques, the group built up thin layers of various materials

to construct paper-thin field-effect transistors, then employed plastic-like organic polymers and graphite inks to carry electricity. This development could lead to flexible computer screens that roll up like window shades."

Personally the author has enough problems with his window shades spontaneously rolling up (which can offer one of life's unforgettable moments when parading in full glory in front of the bathroom mirror) without his computer screen doing the same thing. However, this opens the door to yet more science fiction predictions coming true in the foreseeable future: for example, comic books, newspapers, and textbooks with animated pictures and graphics. We truly do live in interesting times.

Memory ICs

Memory devices are a special class of integrated circuits that are used to store binary data for later use. There are two main categories of semi-conductor memories: *read-only memory (ROM)* and *read-write memory (RWM)*. Other components in the system can read (extract) data from ROM devices, but cannot write (insert) new data into them. By comparison, data can be read out of RWM devices and, if required, new data can be written back into them. The act of reading data out of a RWM does not affect the master copy of the data stored in the device. For a number of reasons, mainly historical, RWMs are more commonly known as *random-access memories (RAMs)*.

ROM and RAM devices find a large number of diverse applications in electronic designs, but their predominant usage is as memory in computer systems (Figure 15-1).¹

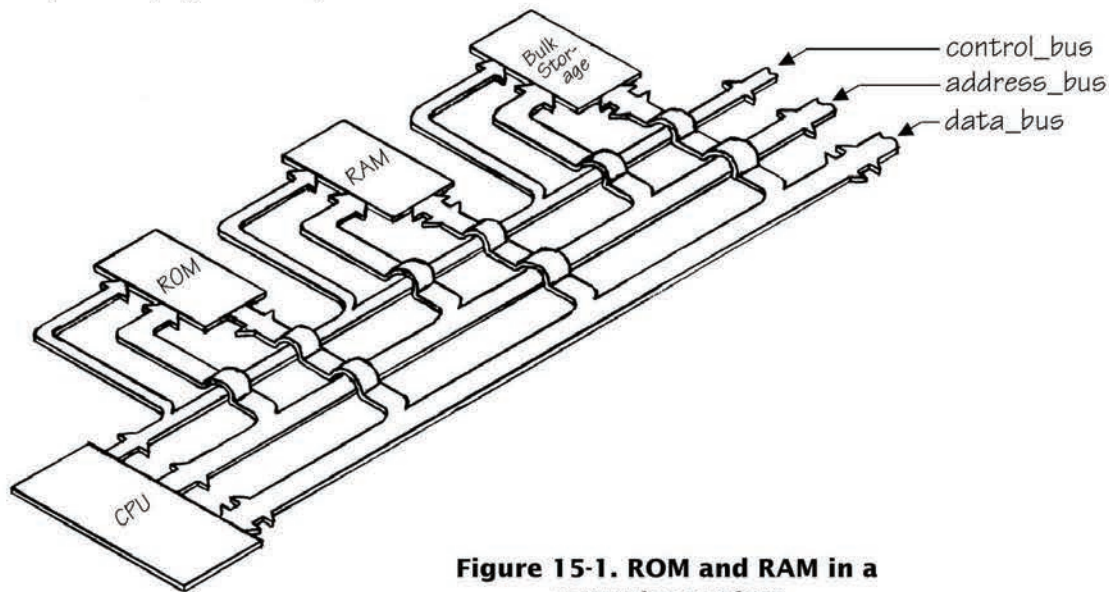


Figure 15-1. ROM and RAM in a computer system

¹ In conversation, ROM is pronounced as a single word to rhyme with “bomb,” while “RAM” is pronounced to rhyme with “ham.”

The brain of the computer is the *central processing unit* (CPU), which is where all of the number crunching and decision-making are performed. The CPU uses a set of signals called the *address bus* to point to the memory location in which it is interested. The address bus is said to be *unidirectional* because it conveys information in only one direction: from the CPU to the memory. By means of control signals, the CPU either reads data from, or writes data to, the selected memory location. The data is transferred on a set of signals called the *data bus*, which is said to be *bidirectional* because it can convey information in two directions: from the CPU to the memory and vice versa.

The bulk storage is often based on magnetic media such as a disk or tape,² which can be used to store a large quantity of information relatively cheaply. Because magnetic media maintains its data when power is removed from the system it is said to be *nonvolatile*.

A general-purpose computer is a system that can accept information from the outside world (from a keyboard, mouse, sensor, or some other input device), process that information, make decisions based on the results of this processing, and return the information to the outside world in its new form (using a computer screen, a robot arm, or some other output device).

The concept of a general-purpose computer on a single chip (or on a group of related chips called a *chipset*) became known as a *microprocessor*. (A *microcontroller* may be considered to be a microprocessor augmented with special-purpose inputs, outputs, and control logic like counter-timers). The written abbreviations for microprocessors and microcontrollers are μP and μC , respectively, where " μ ", pronounced "mu" for "micro" comes from

the Greek *micros*, meaning "small" (in conversation you always use the full names for these devices and never try to pronounce their abbreviations).

Digital signal processing refers to the branch of electronics concerned with the representation and manipulation of signals in digital form. These signals include voice and video, and this form of processing appears in such things as compressing and decompressing audio and video data, telecommunications, radar, and image processing (including medical imaging). A *digital signal processor* (DSP) is a special form of microprocessor that has been designed to perform a specific processing task on a specific type of digital data much faster and more efficiently than can be achieved using a general-purpose device. (In conversation, DSP is always spelled out as "D-S-P").

² The use of tape is becoming increasingly rare for almost any form of mainstream application.

One of the major disadvantages of currently available bulk storage units is their relatively slow speed.³ The CPU can process data at a much higher rate than the bulk storage can supply or store it. Semiconductor memories are significantly more expensive than bulk storage, but they are also a great deal faster.

ROM devices are said to be *mask-programmable* because the data they contain is hard-coded into them during their construction (using photo-masks as was discussed in the previous chapter). ROMs are also classed as being *nonvolatile*, because their data remains when power is removed from the system. By comparison, RAM devices initialize containing random logic 0 or logic 1 values when power is first applied to a system. Thus, any meaningful data stored inside a RAM must be written into it by other components in the system after it has been powered-up. Additionally, RAMs are said to be *volatile*, because any data they contain is lost when power is removed from the system.

When a computer system is first powered up, it doesn't know much about anything. The CPU is hard-wired so that the first thing it does is read an instruction from a specific memory address: for example, address zero. The components forming the system are connected together in such a way that this hard-wired address points to the first location in a block of ROM.⁴ The ROM contains a sequence of instructions that are used by the CPU to initialize both itself and other parts of the system. This initialization is known as *boot-strapping*, which is derived from the phrase "*pulling yourself up by your boot-straps*." At an appropriate point in the initialization sequence, instructions in the ROM cause the CPU to copy a set of master programs, known collectively as the *operating system* (OS), from the bulk storage into the RAM. Finally, the instructions in the ROM direct the CPU to transfer its attention to the operating system instructions in the RAM, at which point the computer is ready for the user to enter the game.

³ Note the use of the qualifier "relatively." Modern bulk storage is actually amazingly fast, but not as fast as the rest of the system.

⁴ Actually, if the truth be told, these days the block of memory pointed to by the CPU on power-up is typically formed from another form of memory like FLASH, which is non-volatile like ROM, but which can be re-programmed (if necessary) like RAM (FLASH is introduced later in this chapter).

Underlying RAM and ROM Architectures

The smallest unit of memory, called a *cell*, can be used to store a single bit of data: that is, a logic 0 or a logic 1. A number of cells physically grouped together are classed as a *word*, and all the cells in a word are typically written to, or read from, at the same time. The core of a memory device is made up of a number of words arranged as an array (Figure 15-2).

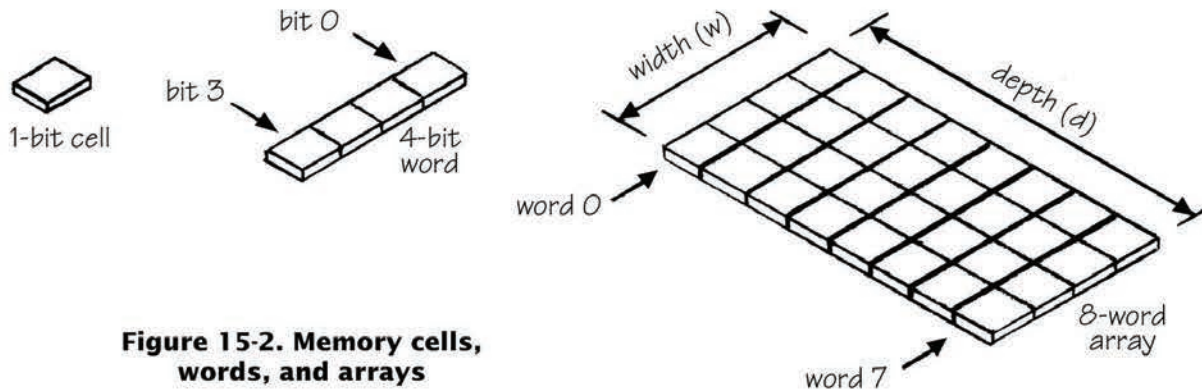


Figure 15-2. Memory cells, words, and arrays

The width (w) of a memory is the number of bits used to form a word, where the bits are usually numbered from 0 to $(w - 1)$.⁵ Similarly, the depth (d) of a memory is the number of words used to form the array, where the words are usually numbered from 0 to $(d - 1)$. The following examples assume a memory array that is four bits wide and eight words deep—real devices can be much wider and deeper.

For the purposes of this discussion, it is convenient to visualize a ROM as containing an array of hard-coded cells. In reality, the physical implementation of a ROM is similar to that of a PROM, which is discussed in more detail in Chapter 16.

In the case of *Dynamic RAMs (DRAMs)*,⁶ each cell is formed from a transistor-capacitor pair. The term “dynamic” is applied because a capacitor loses its charge over time and each cell must be periodically recharged to retain its data.

⁵ Note that there is no official definition as to the width of a word: this is always system-dependent.

⁶ In conversation, DRAM is pronounced as “D-RAM.” That is, spelling out the “D” and following it with “RAM” to rhyme with “ham.”

This operation, known as *refreshing*, requires the contents of each cell to be read out and then rewritten. Some types of DRAM require external circuitry to supervise the refresh process, in which case a special independent controller device is employed to manage a group of DRAMs. In other cases, a DRAM may contain its own internal self-refresh circuitry.

In the case of *Static RAMs* (SRAMs),⁷ each cell is formed from four or six transistors configured as a latch or a flip-flop. The term “static” is applied because, once a value has been loaded into an SRAM cell, it will remain unchanged until it is explicitly altered or until power is removed from the device.

For external components to reference a particular word in the memory, they must specify that word’s address by placing appropriate values onto the address bus. The address is decoded inside the device, and the contents of the selected word are made available as outputs from the array (Figure 15-3).

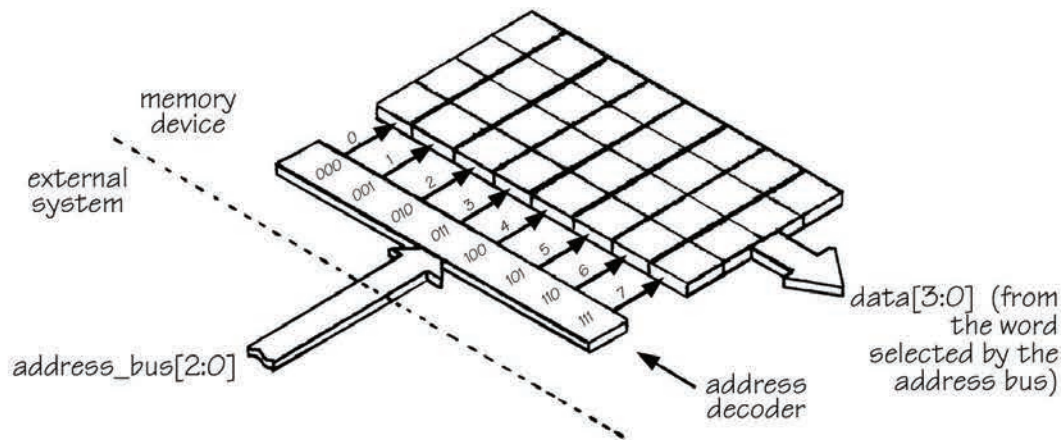


Figure 15-3. Address bus decoding

Standard memory devices are constrained to have a depth of 2^n words, where n is the number of bits used to form the address bus. For example, the 3-bit address bus illustrated in Figure 15-3 can be decoded to select one of eight words ($2^3 = 8$) using a 3:8 decoder.⁸

⁷ In conversation, SRAM is pronounced as “S-RAM.” That is, spelling out the “S” and following it with “RAM” to rhyme with “ham.”

⁸ Decoders were introduced in Chapter 11.

This leads to an interesting quirk when referencing the size of a memory device. In SI units,⁹ the qualifier *k* (kilo)¹⁰ represents one thousand (1,000), but the closest power of two to one thousand is 2^{10} , which equals 1,024. Therefore, a 1 kilobit (1 kb or 1 Kb) memory actually refers to a device containing 1,024 bits.

Similarly, the qualifier *M* (mega)¹¹ is generally taken to represent one million (1,000,000), but the closest power of two to one million is 2^{20} , which equals 1,048,576. Therefore, a 1 megabit (1 Mb) memory actually refers to a device containing 1,048,576 bits. In the case of the qualifier *G* (giga),¹² which is now generally taken to represent one billion (1,000,000,000),¹³ the closest power of two is 2^{30} , which equals 1,073,741,824. Therefore, a 1 gigaabit (1 Gb) memory actually refers to a device containing 1,073,741,824 bits.

If the width of the memory is equal to a byte or a multiple of bytes, then the size of the memory may also be referenced in terms of bytes. For example, a memory containing 1,024 words, each 8 bits wide, may be referred to as being either an 8 kilobit (8 Kb) or a 1 kilobyte (1 KB) device (note the use of “b” and “B” to represent *bit* and *byte*, respectively).

Because multiple memory devices are usually connected to a single data bus, the data coming out of the internal array is typically buffered from the external system by means of tri-state gates.¹⁴ Enabling the tri-state gates allows the device to drive data onto the data bus, while disabling them allows other devices to drive the data bus.

In addition to its address and data buses, a ROM requires a number of control signals, the two most common being `~chip_select` and `~read`.

⁹ The metric system of measurement was developed during the French Revolution and its use was legalized in the U.S. in 1866. The International System of Units (SI) is a modernized version of the metric system.

¹⁰ The term *kilo* comes from the Greek *khiloi*, meaning “thousand” (strangely enough, this is the only prefix with an actual numerical meaning).

¹¹ The term *mega* comes from the Greek *mega*, meaning “great” (hence the fact that Alexander the Great was known as *Alexandros Megos* in those days).

¹² The term *giga* comes from the Latin *gigas*, meaning “giant.”

¹³ See the discussions in Chapter 3 on why we now take “one billion” to represent *one thousand million* rather than *one million million*.

¹⁴ Tri-state gates were introduced in Chapter 11.

(The $\sim read$ control is sometimes called $\sim output_enable$. Alternatively, some devices have both $\sim read$ and $\sim output_enable$ controls.) These control signals are commonly *active-low*; that is, they are considered to be ON when a logic 0 is applied.¹⁵ The $\sim chip_select$ signal indicates to the ROM that its attention is required, and it is combined with $\sim read$ to form an internal signal $\sim rd$ which is used to control the tri-state gates (Figure 15-4).

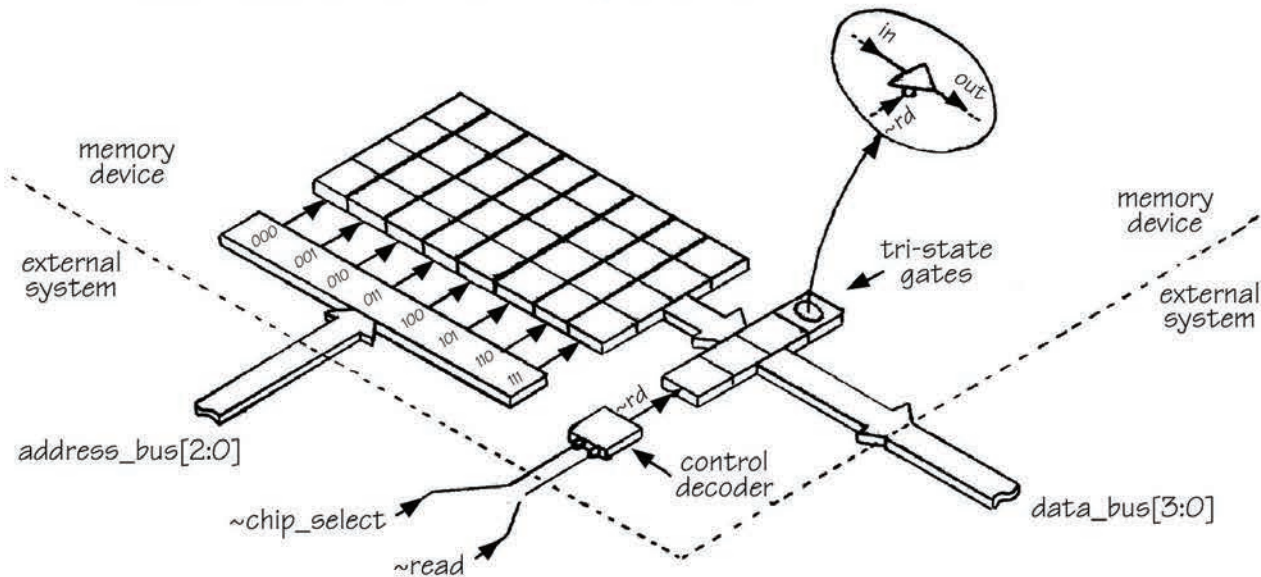


Figure 15-4. ROM control decoding and data bus

When $\sim rd$ is active, the tri-state gates are enabled and the data stored in the word selected by the address bus is driven out onto the data bus. When $\sim rd$ is inactive, the tri-state gates are disabled and the outputs from the device are placed into high-impedance Z states.

In addition to the control signals used by ROMs, RAMs require a mechanism to control the writing of data *into* the device. Some RAMs employ separate buses for reading and writing called *data_out* and *data_in*, respectively. These components usually have an additional control signal called $\sim write$ which is also active-low. Once again, the $\sim chip_select$ signal indicates to the RAM that its attention is required, and it is combined with $\sim read$ to form an internal $\sim rd$

¹⁵ Tilde ' $\sim$ ' characters prefixing signal names are used to indicate that these signals are active-low. The use of tilde characters is discussed in detail in Appendix A.

signal which is used to control the tri-state gates. Additionally, $\sim\text{chip_select}$ is combined with $\sim\text{write}$ to form an internal $\sim\text{wr}$ signal (Figure 15-5).

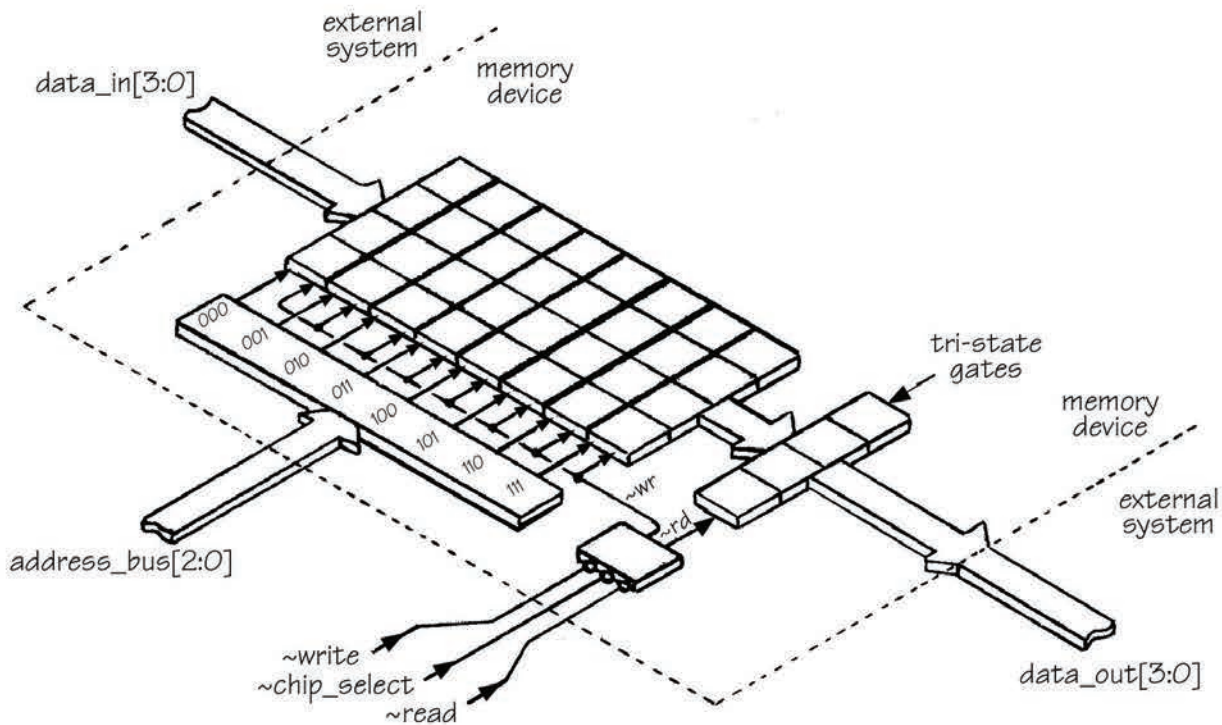


Figure 15-5. RAM with separate data in and data out busses

When $\sim\text{wr}$ is active, the data present on the data_in bus is written into the word selected by the address_bus . The contents of the word pointed to by the address_bus are always available at the output of the array, irrespective of the value on $\sim\text{wr}$. Therefore, devices of this type may be written to and read from simultaneously.

In contrast to those devices with separate data_in and data_out buses, the majority of RAMs use a common bus for both writing and reading. In this case, the $\sim\text{read}$ and $\sim\text{write}$ signals are usually combined into a single control input called something like $\text{read}\sim\text{write}$. The name $\text{read}\sim\text{write}$ indicates that a logic 1 on this signal is associated with a read operation, while a logic 0 is associated with a write. When the $\sim\text{chip_select}$ signal indicates to the RAM that its attention is required, the value on $\text{read}\sim\text{write}$ is used to determine the type of operation to be performed. If $\text{read}\sim\text{write}$ carries a logic 1, the internal $\sim\text{rd}$ signal

is made active and a read operation is initiated; if *read~write* carries a logic 0, the internal $\sim wr$ signal is made active and a write operation is executed (Figure 15-6).

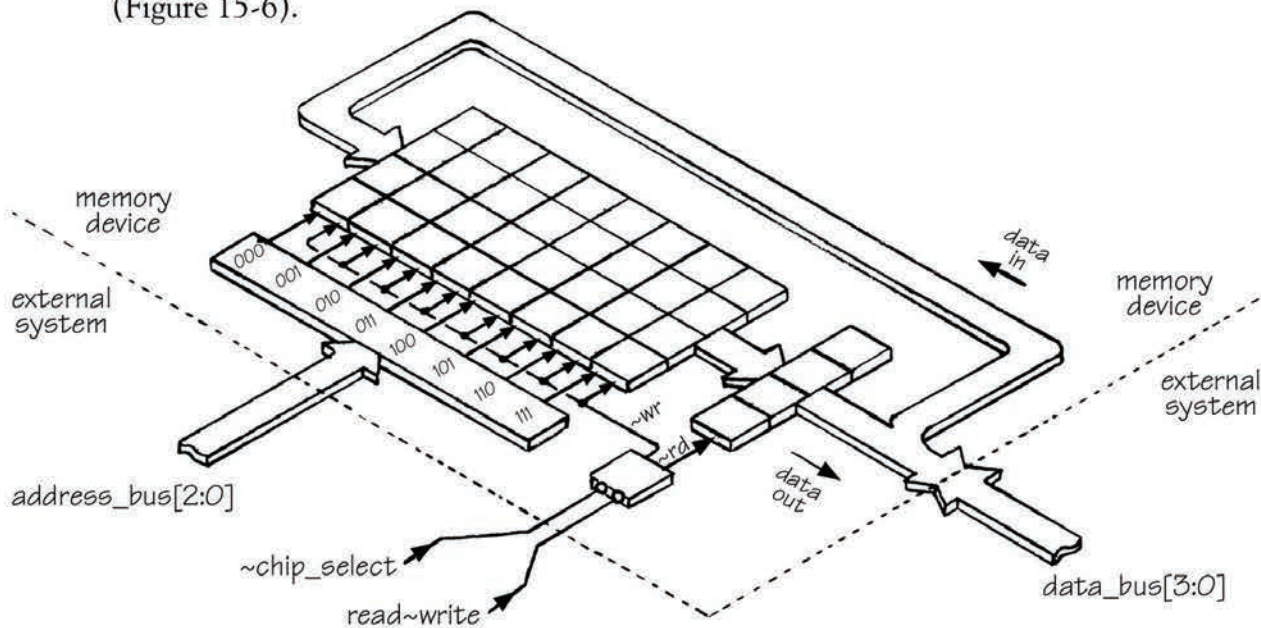


Figure 15-6. RAM with single bi-directional databus

When $\sim rd$ is active, the tri-state gates are enabled and the data from the word selected by the *address_bus* is driven onto the *data_bus*. When $\sim wr$ is active (and therefore $\sim rd$ is inactive), the tri-state gates are disabled. This allows external devices to place data onto the *data_bus* to be written into the word selected by the *address_bus*.

If the value on the *address_bus* changes while $\sim rd$ is active, the data associated with the newly selected word will appear on the *data_bus*. However, it is not permissible for the value on the *address_bus* to change while $\sim wr$ is active because the contents of multiple locations may be corrupted.

Increasing Width and Depth

Individual memory devices can be connected together to increase the width and depth of the total memory as seen by the rest of the system. For example, two 1,024-word devices, each 8 bits wide, can be connected so as to appear to be a single 1,024-word device with a width of 16 bits. An *address_bus* containing 10 bits is required to select between the 1,024 words (Figure 15-7).

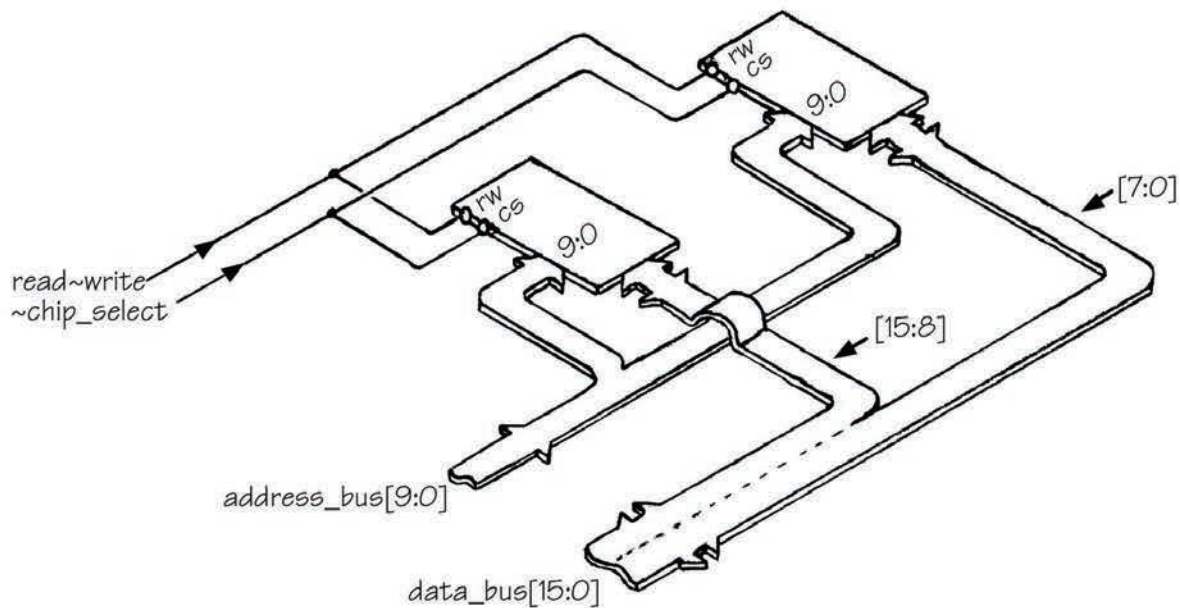


Figure 15-7. Connecting memory devices to increase the width

In this case, the `address_bus` and control signals are common to both memories, but each device handles a subset of the signals forming the `data_bus`. Additional devices can be added to further increase the total width of the `data_bus` as required. Alternatively, two 1,024-word devices, each 8 bits wide, can be connected so as to appear to be a single device with a width of 8 bits and a depth of 2,048 words. An `address_bus` containing 11 bits is required to select between the 2,048 words (Figure 15-8).

In this case, the `data_bus`, the ten least-significant bits of the `address_bus`, and the `read~write` control are common to both memories. However, the most significant bit (MSB) of the `address_bus` is decoded to generate the `~chip_select` signals. A logic 0 on the most-significant bit selects the first device and deselects the second, while a logic 1 deselects the first device and selects the second.

Additional address bits can be used to further increase the total depth as required. If the `address_bus` in the previous example had contained 12 bits, the two most-significant bits could be passed through a 2:4 decoder to generate four chip select signals. These signals could be used to control four 1,024-word devices, making them appear to be a single memory with a width of 8 bits and a depth of 4,096 words.

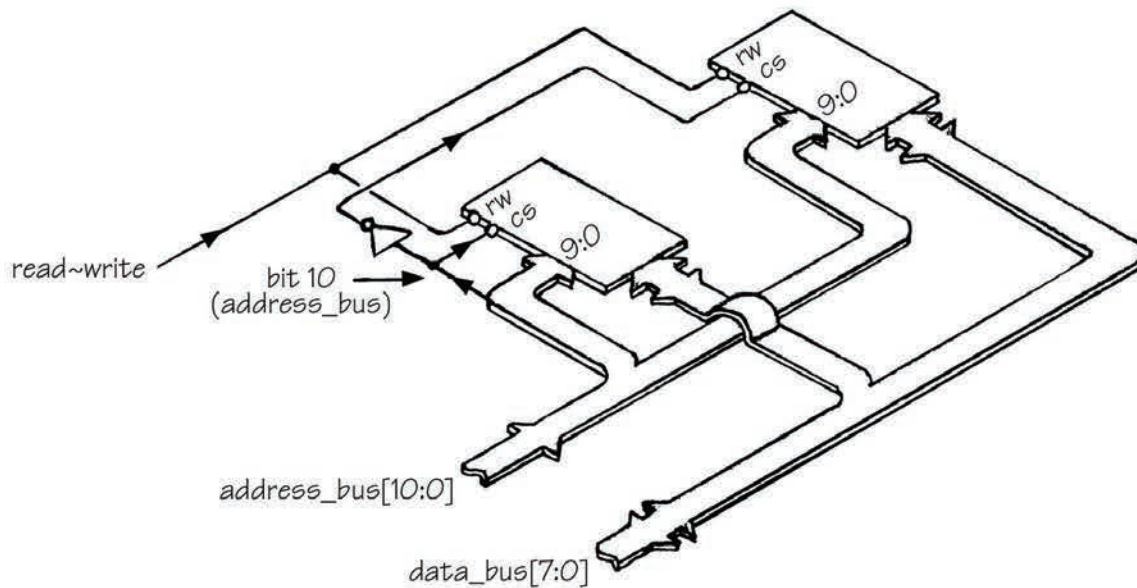


Figure 15-8. Connecting memory devices to increase the depth

Alternative Technologies

SRAMs are faster than DRAMs, but each SRAM cell requires significantly more silicon real estate and consumes much more power than a corresponding DRAM cell. The first SRAM (256-bit) and DRAM (1,024-bit) devices were both created in 1970. For the next few decades, both types of memory quadrupled their capacities approximately every three years, but by the beginning of the twenty-first century this had slowed to a doubling every two to three years.

By mid 2002, the highest density SRAMs contained 16 megabits. These were typically based on 0.18 micron processes with a roadmap in place to shrink to 0.15 and 0.13 microns. By comparison, by the end of 2002, mainstream DRAM devices produced using 0.15 micron processes will contain 256 megabits. DRAM vendors will move to volume production using 0.13 micron processes in 2003, and 512 megabit devices based on these processes will become mainstream in 2004/2005. These will be succeeded by 1 gigabit devices based on 0.09 micron technologies in the second half of the decade.¹⁶

¹⁶ One problem with DRAM memory cells is that there is a limit to the amount by which the size of their capacitors can be reduced before they become too small to retain any data. This problem will hopefully be addressed by MRAM devices, which are introduced later in this chapter.

Due to their regular structures and the numbers of components produced, memory devices used to lead semiconductor technology by approximately two years. For example, DRAMs based on 0.35 micron processes became available as early 1994, but other devices with these geometries didn't hit the mainstream until 1996. By 2002, however, high-end general-purpose logic processes had become just as advanced as memory processes, and both memory and logic foundries routinely offered 0.13 micron technologies.

PROMs

Because ROMs are constructed containing pre-defined data, they are an expensive design option unless used for large production runs. As an alternative, *programmable read-only memories* (PROMs)¹⁷ were developed in the mid-1970s. PROMs are manufactured as standard devices that may be electrically programmed by the designer and are non-volatile. PROMs are *one-time programmable* (OTP) and slightly slower than their ROM equivalents, but are significantly cheaper for small- to medium-sized production runs.

EPROMs

To satisfy the designers' desire to combine the advantages of DRAMs (programmable) and ROMs (nonvolatile), *erasable programmable read-only memories* (EPROMs)¹⁸ were developed in the late 1970s. EPROMs are electrically programmable by the designer and are non-volatile, but can be erased and reprogrammed should the designer so desire. An EPROM device is presented in a ceramic package with a

Until the latter half of the 1990s, DRAM-based computer memories were asynchronous, which means they weren't synchronized to the system clock. Every new generation of computers used a new trick to boost speed, and even the engineers started to become confused by the plethora of names, such as fast page mode (FPM), extended data out (EDO), and burst EDO (BEDO). In reality, these were all based on core DRAM concepts; the differences are largely in how you wire the chips together on the circuit board (OK, sometimes there is a bit of tweaking on the chips also).

Over time, the industry migrated to synchronous DRAM (SDRAM), which is synchronized to the system clock and makes everyone's lives much easier. Once again, however, SDRAM is based on core DRAM concepts—it's all in the way you tweak the chips and connect them together.

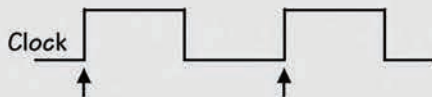
¹⁷ In conversation, PROM is pronounced just like the high school dance of the same name.

small quartz window mounted in the top. The quartz window is usually covered by a piece of opaque tape. To erase the device, the tape is removed and the EPROM is exposed to ultraviolet radiation for approximately 20 minutes. EPROMs are relatively expensive due to their complicated packaging requirements, but are ideal for prototyping and other applications that require regular changes to the stored data. EPROMs are often used during the development of a design and are subsequently replaced by equivalent PROM or ROM devices in cheaper plastic packages after the design has stabilized. (The actual construction of EPROMs and some of the other types of memory devices introduced during the remainder of this chapter are discussed in greater detail in Chapter 16.)

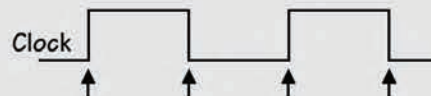
By 2002, there were a variety of different types of SDRAM on the market (way too many to cover here). Perhaps the most well known are the ones used in general-purpose personal computers, which are called PC100 and PC133, which operate at 100 MHz and 133 MHz, respectively.

In the future, memory manufacturers may move to faster speeds (say PC166 or higher), but this may cause problems, because the original SDRAM interface was not designed with these high frequencies in mind.

However, the original SDRAM specification was based on using only one of the clock edges (say the rising edge) to read/write data out-of/into the memory as shown below:



An alternative known as *double data rate (DDR)* is to design the memory in such a way that data can be read/written on both edges of the clock:



This effectively doubles the amount of data that can be pushed through the system without increasing the clock frequency (it sounds simple if you say it fast, but making this work is trickier than it may at first appear).

Furthermore, a *quad data rate (QDR)* memory is one with separate data in and data out busses (as was illustrated in Figure 15-5), both of which can be used on both edges of the clock.

¹⁸ In conversation, EPROM is pronounced as "E-PROM." That is, spelling out the "E" and following it with "prom."

EEPROMs

A further technology called *electrically-erasable read-only memory* (EEPROM or E²PROM)¹⁹ was developed towards the end of the 1970s. E²PROMs are electrically programmable by the designer and are nonvolatile, but can be electrically erased and reprogrammed should the designer so desire. Additionally, by means of additional circuitry, an E²PROM can be erased and reprogrammed while remaining resident on the circuit board, in which case it may be referred to as *in-system programmable* (ISP).

An EPROM cell is based on a single transistor and is therefore very efficient in terms of silicon real estate, while an equivalent E²PROM cell is based on two transistors and requires more area on the silicon. However, an EPROM device must be erased and reprogrammed in its entirety, while an E²PROM device can be erased and reprogrammed on a word-by-word basis.

¹⁹ In conversation, some people pronounce EEPROM as "E-E-PROM" (spelling out "E-E" followed with "prom"), while others say "E-squared-prom."

PC100 and PC133 memory modules are 64-bits wide (although dual configurations have been used to provide 128 bit wide busses in really high-end computers).

Thus, assuming a standard 64-bit (8-byte) wide PC100 (100 MHz) memory module, the peak data throughput will be $8 \times 100 = 800$ megabytes per second. If a DDR version becomes available as discussed earlier, this would rise to 1,600 megabytes per second. Furthermore, a 128-bit wide DDR (using dual 64-bit modules in parallel) would have a peak bandwidth of 3,200 megabytes per second.

Towards the end of the 1990s, an alternative memory concept called *Rambus* began to gain attention (as usual this is based on core DRAM concepts presented in a cunning manner). By around 2000, personal computers equipped with memory modules formed from Rambus DRAM (RDRAM) devices were available. These memory modules were 16-bits (2-bytes) wide, and were accessed on both edges of a 400 MHz clock, thereby providing a peak data throughput of $2 \times 2 \times 400 = 1,600$ megabytes per second. (Just to be confusing, these devices are actually referenced as "800 MHz Rambus DRAM," but this is just marketing spin trying to make using both edges of a 400 MHz clock sound more impressive.)

In the summer of 2002, a new flavor of Rambus was announced with a 32-bit wide bus using both edges of a 533 MHz clock (the official announcements quote "1,066 MHz Rambus DRAM," but this is the same marketing spin as noted above). Whatever, this will provide a peak bandwidth of 4,200 megabytes per second.

FLASH

Yet another technology called FLASH is generally regarded as an evolutionary step that combines the best features from EPROM and E²PROM. The name FLASH is derived from its fast reprogramming time compared to EPROM. FLASH has been under development since the end of the 1970s, and was officially described in 1985, but the technology did not initially receive a great deal of interest. Towards the end of the 1980s, however, the demand for portable computer and communication systems increased dramatically, and FLASH began to attract the attention of designers.

All variants of FLASH are electrically erasable like E²PROMs. Some devices are based on a single transistor cell, which provides a greater capacity than an E²PROM, but which must be erased and reprogrammed on a device-

wide basis similar to an EPROM.

Other devices are based on a dual transistor cell and can be erased and reprogrammed on a word-by-word basis (see Chapter 16 for more details).

FLASH is considered to be of particular value when the designer requires the ability to reprogram a system in the field or via a communications link while the devices remain resident on the circuit board.

Computer systems are very complicated and there's always the chance that an error will occur when reading or writing to the memory (a stray pulse of "noise" may flip a logic 0 to a logic 1 while your back is turned). Thus, serious computers use error-correcting code (ECC) memory, which includes extra bits and special circuitry that tests the accuracy of data as it passes in and out of memory and corrects any (simple) errors.

MRAMs

A technology that is attracting a great deal of interest for the future is *magnetic random access memory* (MRAM),²⁰ which may be able to store more data, read and write data faster, and use less power than any of the current memory technologies. In fact, the seeds of MRAM were laid as far back as 1974, when IBM developed a component called a *magnetic tunnel junction* (MTJ), which comprises a sandwich of two ferromagnetic layers separated by a

²⁰ In conversation, MRAM is pronounced "M-RAM." That is, by spelling out the "M" and following it with "RAM" to rhyme with "ham."

Whatever flavor of DRAM you are using, a single device can only contain a limited amount of data, so a number of DRAMs are gathered together onto a small circuit board called a *memory module*.

Each memory module has a line of gold-plated pads on both sides of one edge of the board. These pads plug into a corresponding connector on the main computer board.

A *single in-line memory module (SIMM)* has the same electrical signal on corresponding pads on the front and back of the board

(that is, the pads on opposite sides of the board are “tied together”).

By comparison, in the case of a *dual in-line memory module (DIMM)*, the pads on opposite sides of the board are electrically isolated from each other and form two separate contacts.

Last but not least, we have the *RIMM*, which really doesn’t stand for anything per se, but which is the trademarked name for a *Rambus memory module (RIMMs)* are similar in concept to DIMMs, but have a different pin count and configuration).

thin insulating layer. A memory cell is created by the intersection of two wires (say a “row” line and a “column” line) with an MJT sandwiched between them. MRAMs have the potential to combine the high speed of SRAM, the storage capacity of DRAM, and the non-volatility of FLASH, while consuming very little power. It’s taken close to 30 years, but MRAMs test chips are predicted to become available around 2003, with volume production in 2004.

nvRAMs

Finally, a class of devices known as *nonvolatile RAMs (nvRAMs)*²¹ may be used in cases where high speed is required and cost is not an overriding issue. These devices are generally formed from an SRAM die mounted in a package with a very small battery, or as a mixture of SRAM and E²PROM cells fabricated on the same die.

²¹ In conversation, nvRAM is pronounced as “n-v-RAM.” That is, spelling out the “n” and the “v” and following them with “RAM” to rhyme with “ham.”

Programmable ICs

Programmable integrated circuits are standard devices constructed in such a way that a designer can configure them to perform a specific function. These devices, which were introduced in the mid-1970s, are generically known as *programmable logic devices (PLDs)*,^{1,2} and can range in complexity from a few hundred to tens of thousands of equivalent gates. Traditional PLDs are based on logic gates formed from diodes (Figure 16-1).

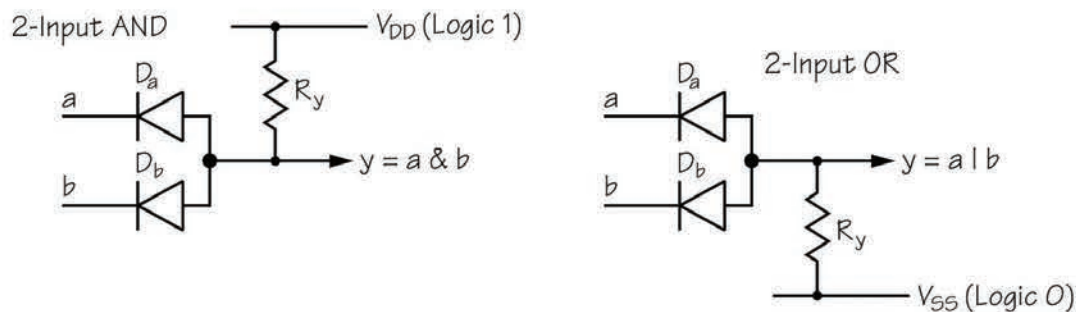


Figure 16-1. Diode implementations of AND and OR gates

In the case of the 2-input AND, the pull-up resistor R_y attempts to hold the output y at V_{DD} (logic 1). When both of the inputs are at logic 1, there is no difference in potential across the diodes and y remains at logic 1. However, if input a is connected to V_{SS} (logic 0), diode D_a will conduct and y will be pulled down to logic 0; similarly for input b and diode D_b . Additional inputs can be formed by adding more diodes.

In the case of the 2-input OR, the pull-down resistor R_y attempts to hold the output y at V_{SS} (logic 0). When both of the inputs are at logic 0, there is no difference in potential across the diodes and y remains at logic 0. However,

¹ In conversation, PLD is pronounced by spelling it out as “P-L-D.”

² Originally all PLDs contained a modest number of equivalent logic gates and were fairly simple. As more *complex* PLDs (CPLDs) arrived on the scene, however, it became common to refer to their simpler cousins as *simple* PLDs (SPLDs). (Note that CPLDs are introduced in more detail later in this chapter.)

if input a is connected to V_{DD} (logic 1), diode D_a will conduct and y will be pulled up to logic 1; similarly for input b and diode D_b . Once again, additional inputs can be formed by adding more diodes.

Fusible-link Technologies

In the most common PLD technology, which is known as *fusible-link*, each diode has an associated link known as a *fuse*. The designer can individually remove unwanted fuses by applying pulses of relatively high voltage and current, in the order of 20 volts and 300 milliamps, respectively, to the device's inputs. These pulses are sufficient to effectively vaporize the fuse (Figure 16-2).

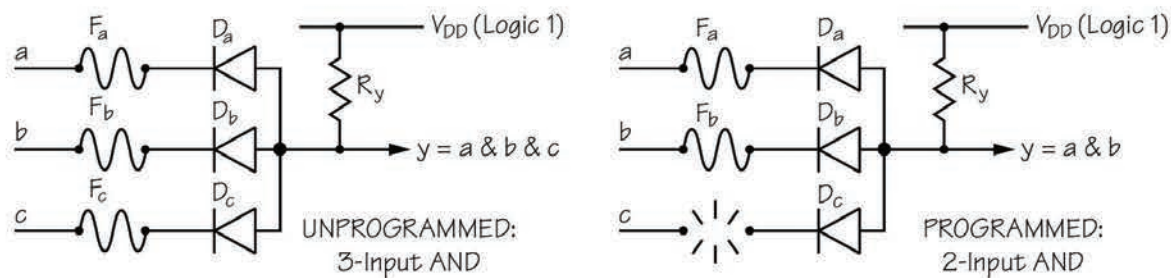


Figure 16-2. Removing unwanted fusible links

This process is typically known as *programming the device*, but may also be referred to as *blowing the fuses* or *burning the device*. Devices based on fusible links are said to be *one-time programmable (OTP)*,³ because once a fuse has been blown it cannot be replaced.

Antifuse Technologies

As an alternative to fusible links, some PLDs employ *antifuse* technologies in which each diode has an associated *antifuse*. Unlike fusible links—which will conduct unless they are forcibly removed—antifuse links will not conduct in the device's unprogrammed state. The user can individually enable desired antifuse links by applying signals of relatively high voltage and current to the device's inputs (Figure 16-3).

³ In conversation, OTP is pronounced by spelling it out as "O-T-P."

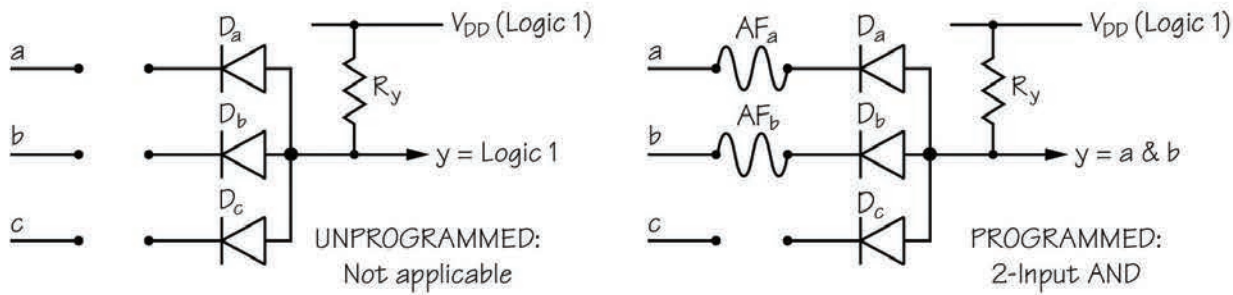


Figure 16-3. Adding desired antifuse links

Antifuse links are formed by creating a connection—called a *via*—of amorphous (noncrystalline) silicon between two layers of metalization. In its unprogrammed state, the amorphous silicon is an insulator with a very high resistance in excess of one billion ohms. The programming signal effectively grows a link by changing the insulating amorphous silicon into conducting polysilicon (Figure 16-4).

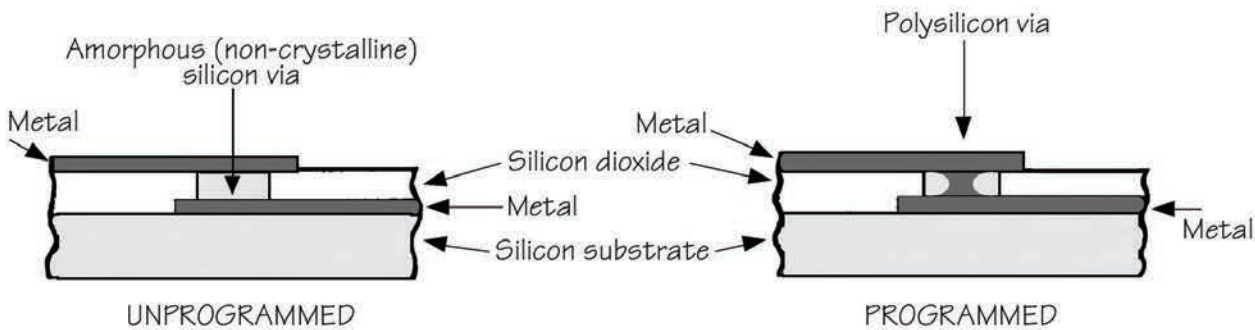


Figure 16-4. Growing an antifuse link

The programmed links in the original antifuse technologies had resistance values in the order of 1,000 ohms, but modern processes have reduced this value to 50 ohms or lower. Devices based on antifuses are also *one-time programmable* (OTP), because once an antifuse link has been grown it cannot be removed.

Special PLD Notation

Due to the fact that PLD structures are very regular, and also because they differ from standard logic gates, a special form of notation has been adopted. This notation is applicable to both fusible-link and antifuse PLDs.

Consider a device delivered with four diode-link pairs forming a 4-input AND function (Figure 16-5).

The AND symbol indicates the function and encompasses the pull-up resistor to V_{DD} . The diode-fuse pairs are represented by the presence or absence of crosses. In a fusible-link device, a cross indicates that the corresponding link has been left intact, while the absence of a cross indicates that the link has been blown away. Thus, the diagram for an *unprogrammed* fusible-link device will show crosses at each intersection of the matrix. By comparison, in antifuse technology, a cross indicates that a link has been grown, while the absence of a cross indicates that no link is present. Thus, the diagram for an *unprogrammed* antifuse device will not show any crosses.

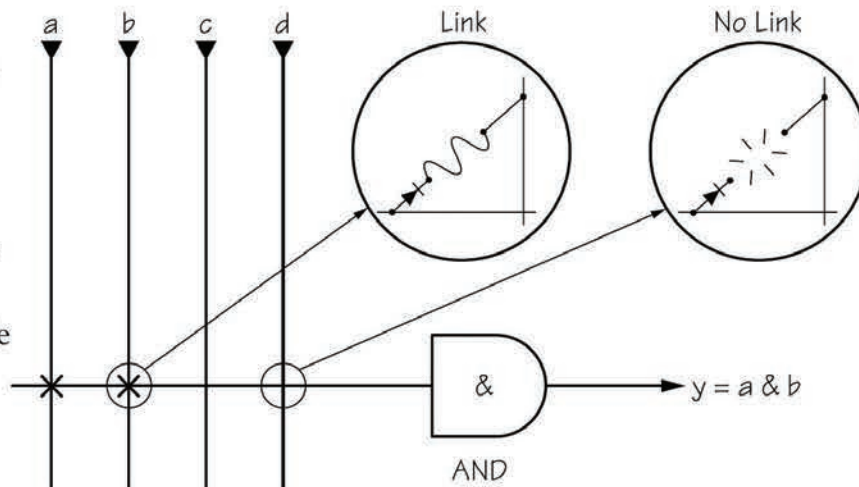


Figure 16-5. Special PLD notation

The majority of this book follows the convention laid down in Appendix A, which is that active-low and complementary signals are indicated by prefixing their names with tilde “~” characters.

However, rules are made for the guidance of wise men and the blind obedience of fools, and the purposes of the discussions in this chapter are better served by representing inverted PLD input signals by means of lines, or bars, over their names.

Actually, this serves to illustrate an important point: although mixing conventions is not usually recommended, engineers are “switch hitters” who will happily use whatever tool is most appropriate and best suited to get the job done.

Generic PLD Structures

To increase the versatility of PLDs, their inputs are inverted inside the device and both true and inverse versions of the inputs are presented to an array (Figure 16-6).

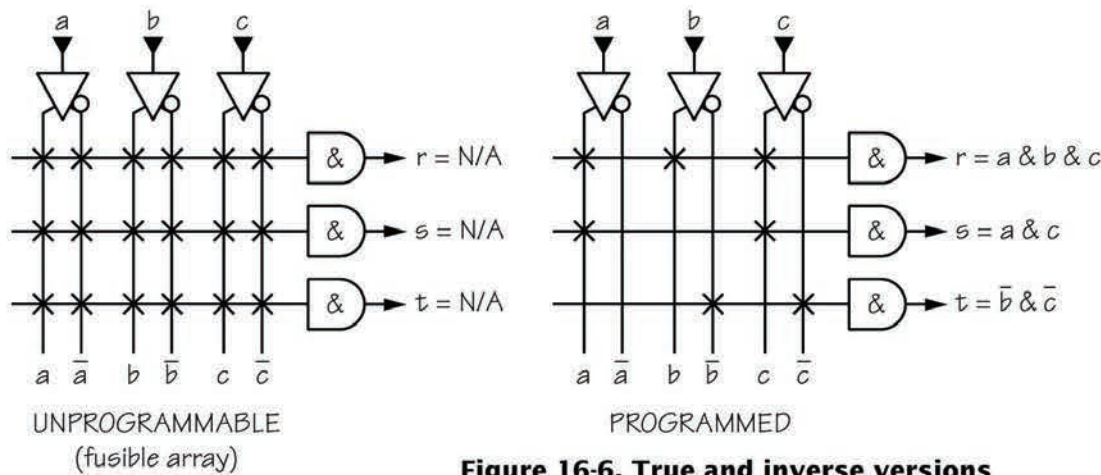


Figure 16-6. True and inverse versions of the inputs are presented to the array

The number of AND functions is independent of the number of inputs; additional ANDs can be formed by introducing more rows into the array. Similar techniques can be used to create an array of OR functions, and PLDs typically contain an AND array feeding into an OR array (Figure 16-7).

The number of OR functions is independent of both the number of inputs and the number of AND functions; additional ORs can be formed by introducing more columns into the OR array.

PLDs are not obliged to have AND input arrays feeding OR output arrays; NOR output arrays are also available. However, while it would be possible to create other structures such as OR-AND, NAND-OR, or NAND-NOR, these alternatives tend to be relatively rare or simply not provided. The reason it isn't necessary for the vendor to supply all possible variations is because AND-OR (and AND-NOR) structures directly map onto the *sum-of-products* representations most often used to specify equations.⁴ Other equation formats can be mapped to these structures using standard Boolean algebraic techniques—for example, DeMorgan Transformations.⁵

⁴ Sum-of-products representations were introduced in Chapter 9.

⁵ DeMorgan Transformations were introduced in Chapter 9.

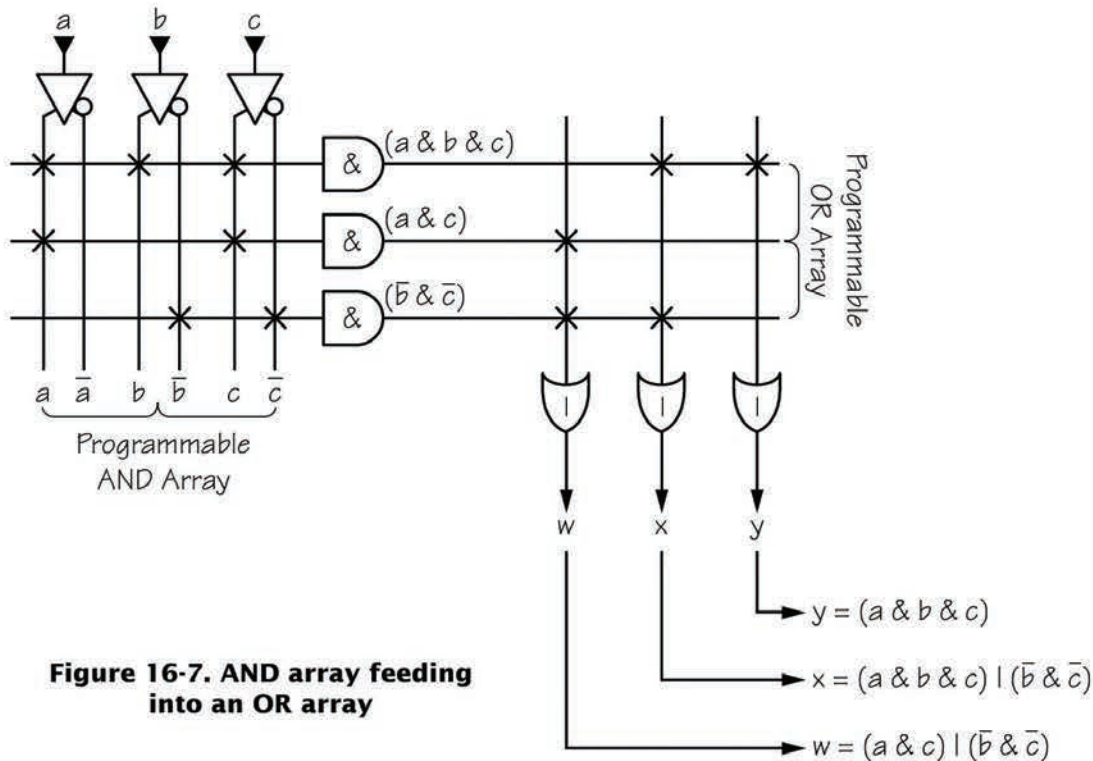


Figure 16-7. AND array feeding into an OR array

Although core PLD concepts are relatively simple, the wider arena may prove to be a morass of confusion to the unwary. There are a multiplicity of PLD alternatives, most of which seem to have acronyms formed from different combinations of the same three or four letters.⁶ This may be, as some suggest, a strategy to separate the *priests* from the *acolytes*, or it may be that the inventors of the devices had no creative energy left to dream up meaningful names for them. Whatever the case, the more common PLD variants are introduced below.

Programmable Logic Arrays (PLAs)

The most user-configurable of the traditional PLDs are *programmable logic arrays* (PLAs),⁷ because both the AND and OR arrays are programmable (Figure 16-8).

⁶ A common engineering joke is to refer to TLAs, which is an acronym that stands for “three letter acronyms” (in conversation, TLA is pronounced by spelling it out as “T-L-A”).

⁷ In conversation, PLA is spelled out as “P-L-A.”

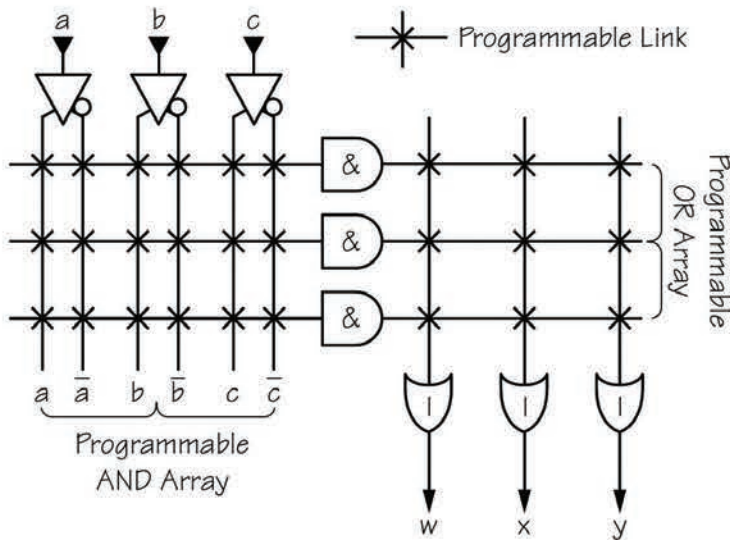


Figure 16-8. Generic structure of a PLA (programmable AND, programmable OR)

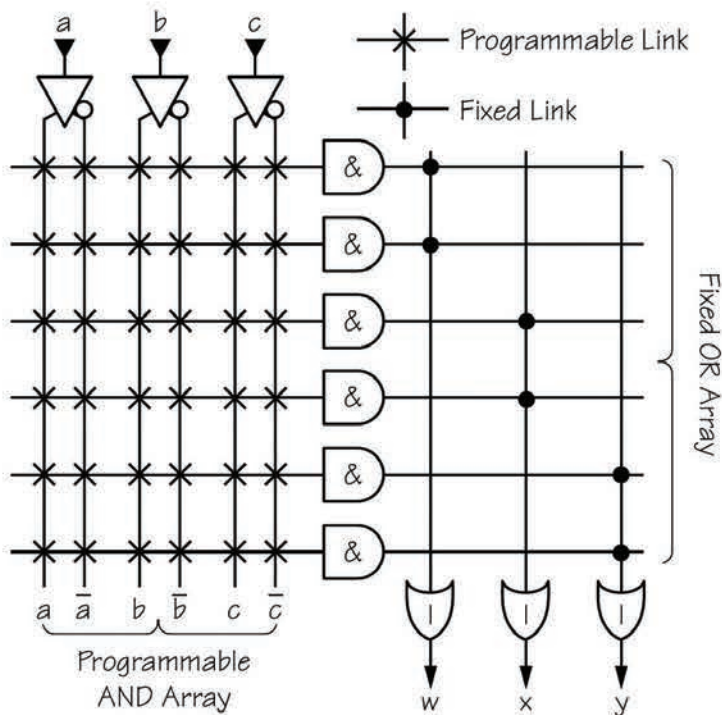


Figure 16-9. Generic structure of a PAL (programmable AND, predefined OR)

This diagram indicates a fusible link technology, because all of the links are present when the device is in its unprogrammed state. Similarly, the following examples are based on fusible link devices unless otherwise stated.

Programmable Array Logic (PAL)

Many applications do not require both the AND and OR arrays to be programmable. In the case of *programmable array logic (PAL)*^{8,9} devices, only the AND array is programmable while the OR array is predefined (Figure 16-9).

Although PALs are less flexible than PLAs, they operate somewhat faster because hard-wired connections take less time to switch than their programmable equivalents.¹⁰

⁸ In conversation, PAL is pronounced as a single word to rhyme with "gal."

⁹ PAL is a registered trademark of Monolithic Memories Inc.

¹⁰ There are also variations on PALs called *generic array logic (GAL)* devices from Lattice Semiconductor Corporation (www.latticesemi.com).

Programmable Read-Only Memories (PROMs)

In the case of *programmable read-only memory* (PROM) devices, the AND array is predefined and only the OR array is programmable (Figure 16-10).¹¹

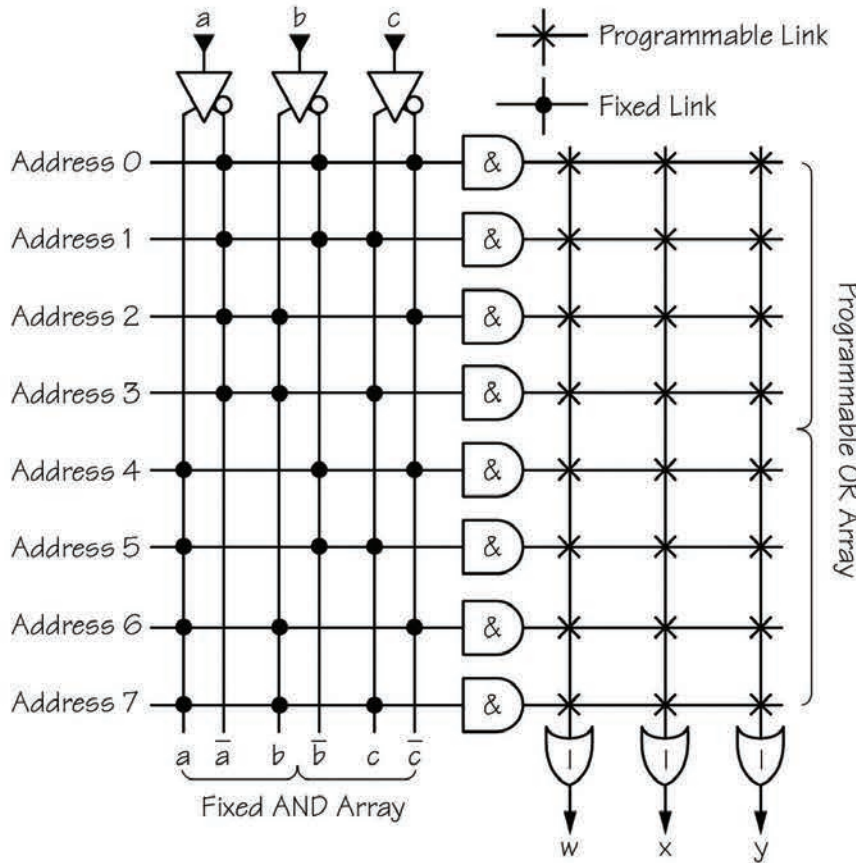


Figure 16-10. Generic structure of a PROM (predefined AND, programmable OR)

PROMs are useful for equations requiring a large number of product terms, but tend to have fewer inputs because every input combination is decoded and used. The internal structure of a PROM is similar to that of a ROM, the main difference being that a ROM's contents are defined during its construction. PROMs are generally considered to be memory devices, in which each address applied to the inputs returns a value programmed into the device.

¹¹ PROMs were introduced in Chapter 15.

Additional Programmable Options

If you thought the above was cunning, just wait until you see some of the really clever tricks that can be used to augment PLDs and make them even more useful.

Tri-stateable Outputs

Many PLDs have tri-state buffers on their outputs.¹² All of the buffers share a common *enable* control, which therefore requires only a single input pin on the device (Figure 16-11).

Some devices may contain additional programmable links associated with the outputs. By means of these additional links (which were omitted from Figure 16-11 for reasons of clarity), each output can be individually programmed to be either permanently enabled or to remain under the control of the *enable* input as required (Figure 16-12).

In this example, the tri-state buffer driving the *x* output has been disconnected from the *enable* control and connected to a constant logic 0 (as these tri-state buffers have active-low control inputs—as is indicated by the bobbles on their symbols—a logic 0 enables them while a logic 1 disables them). As a result, the *x* output is permanently enabled, irrespective of the value on the *enable* input.

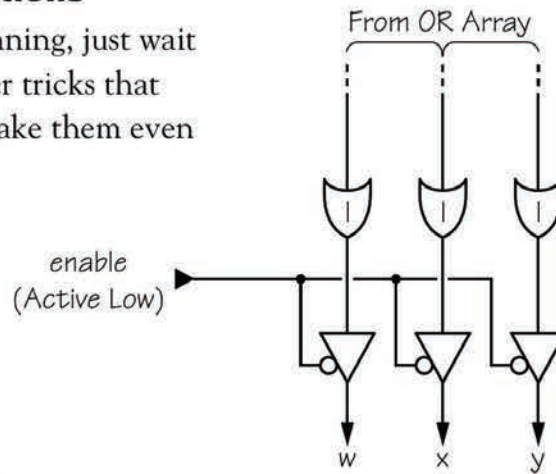


Figure 16-11. PLD with tri-stateable outputs

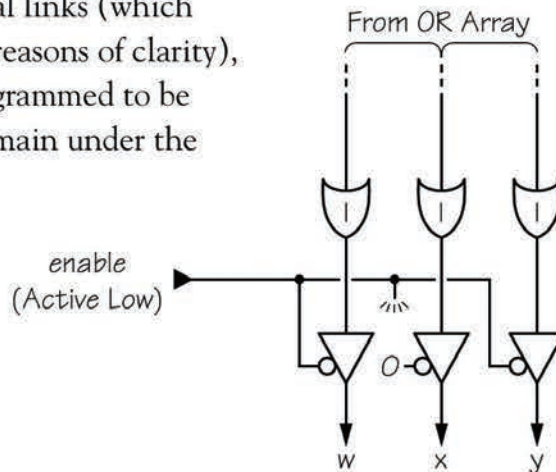


Figure 16-12. PLD with individually tri-stateable outputs

¹² Tri-state buffers were introduced in Chapter 11.

Using Outputs as Inputs

To further increase the fun, some devices are constructed in such a way that their tri-state outputs are fed back into the AND array (Figure 16-13).

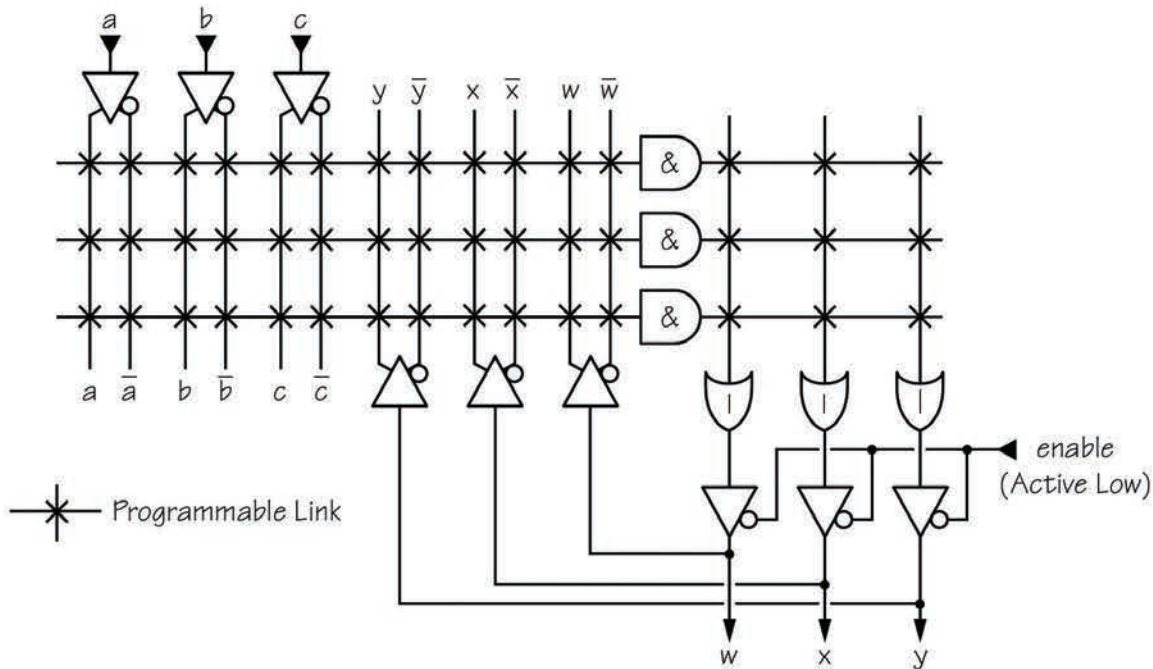


Figure 16-13. PLA with tri-state outputs fed back as inputs

Once again, there are additional programmable links that have been omitted from Figure 16-13 for reasons of clarity. By means of these links, each output's tri-state buffer can be individually configured so as to always drive a high-impedance Z state. Pins that are configured in this way can be used as additional inputs (Figure 16-14).

In this example, the unprogrammed device began with three inputs and three outputs, but the user actually required a device with four inputs and only two outputs. By means of the appropriate links, the tri-state buffer on the y output was disconnected from the *enable* control and connected to a constant logic 1. Because w and x are still required to function as outputs, all of their associated links in the AND array must be blown away to ensure that these pins will not have any effect as inputs. The ability to configure pins as outputs or inputs provides a great amount of flexibility; instead of purchasing myriad devices with every conceivable combination of inputs and outputs, the designer can configure a single device type as required.

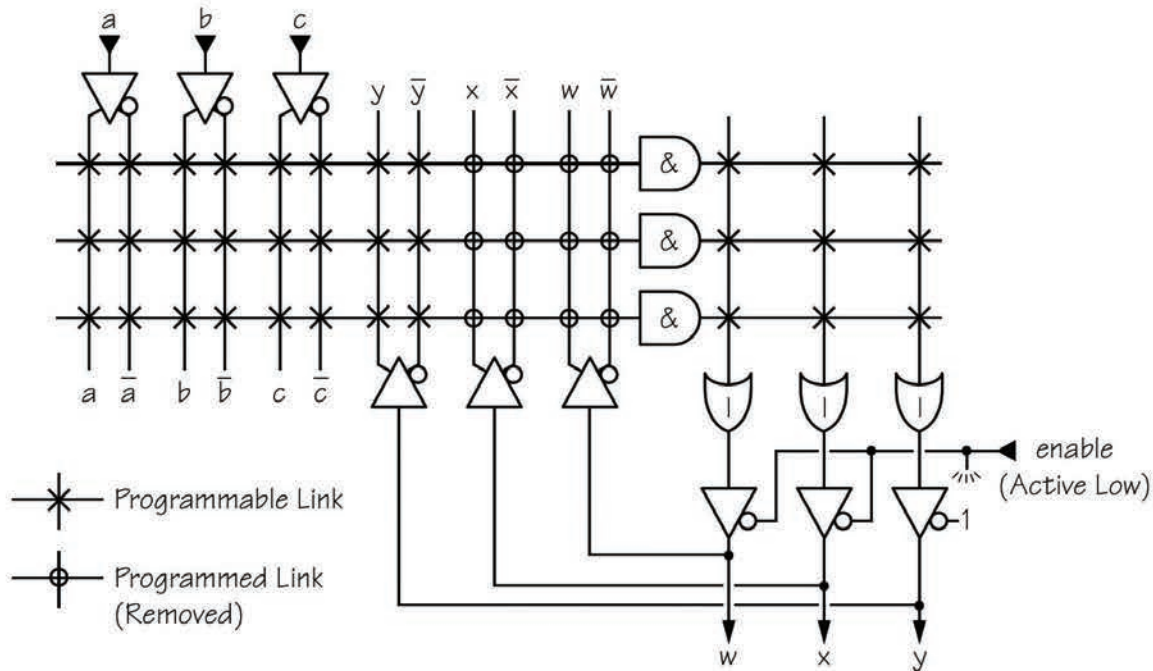


Figure 16-14. PLA with one tri-state output configured as an input

Registered Outputs

Certain PLDs are equipped with registers on the outputs, and others with latches. Depending on the particular device, the registers (or latches) may be provided on all of the outputs or on a subset of the outputs. Registered devices are particularly useful for implementing finite state machines (FSMs).¹³ All of

the registers share a common clock signal (this will be an enable signal in the case of latches) which therefore requires only a single input pin on the device (Figure 16-15).

In this example, the outputs are shown as being registered with D-type flip-

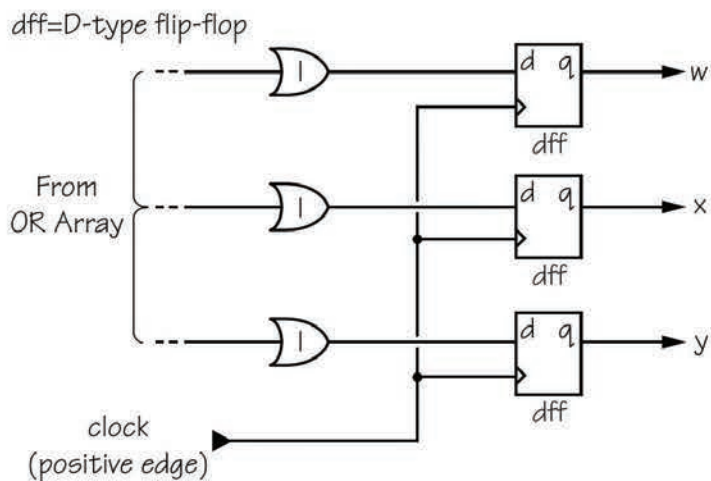


Figure 16-15. PLD with registered outputs

¹³ FSMs were introduced in Chapter 12.

flops, but alternative register types such as JK flip-flops or T-type flip-flops may be more suitable for certain applications. It can be inconvenient to support a dedicated device for each type of register. As a solution, some devices have configurable register elements whose types can be selected by programming appropriate fuses.

Registered (or latched) outputs may also incorporate by-pass multiplexers (Figure 16-16).

By means of appropriate fuses, the control inputs to the multiplexers can be individually configured to select either the *non-registered* data or its *registered* equivalent.

Other common programmable options are the ability to select *true* or *complemented* outputs and TTL- or CMOS-compatible output drivers. An individual PLD typically only provides a subset of the above capabilities, but these may be combined in a variety of ways; for example, registered outputs may be followed by tri-state buffers.

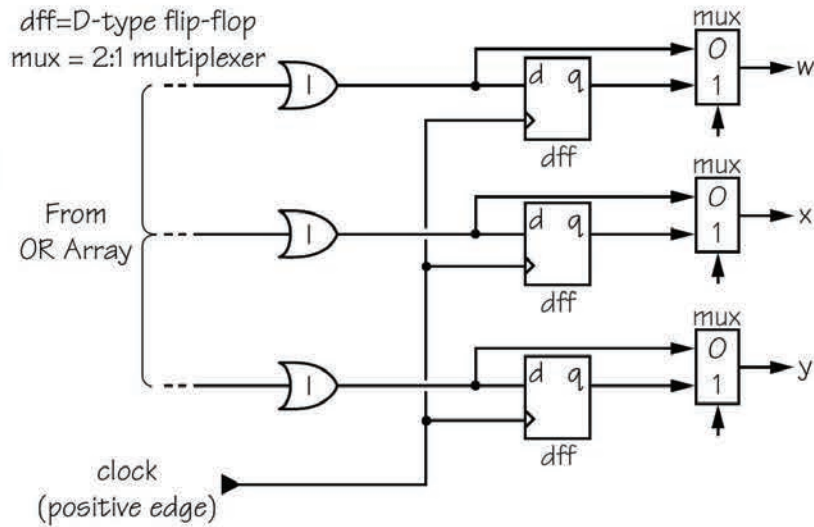


Figure 16-16. PLD with registered outputs and by-pass multiplexers

Programming PLDs

Programming a traditional PLD is relatively painless because there are computer programs and associated tools dedicated to the task. The user first creates a computer file known as a *PLD source file* containing a textual description of the required functionality (Figure 16-17).

In addition to Boolean equations, the PLD source file may also support truth tables, state tables, and other constructs, all in textual format. The exclamation marks (called shrieks) shown in these equations provide a textual way of indicating inversions. Additional statements allow the user to specify which outputs are to be tri-statable, which are to be registered, and any of the other programmable options associated with PLDs.

```

INPUT    =  a , b , c ;
OUTPUT   =  w , x , y ;

EQUATION w    =  (a & b) ;
EQUATION x    =  (a & b) | (a & !b & !c) ;
EQUATION y    =  (a & b) | (b & c) | (a & !c) ;

```

Figure 16-17. Very simple example PLD source file

A special computer program is used to process the PLD source file. This program makes use of a knowledge database that contains details about the internal construction of all of the different types of PLDs. After the user has instructed the program as to which type of device they wish to use, it analyzes the equations in the source file and performs algebraic minimization to ensure optimal utilization of the device's resources. The program accesses the knowledge database for details about the designated device and evaluates which fuses need to be blown to implement the desired functionality. The program then generates a textual output file comprising 0 and 1 characters, which represent the fuses to be blown; this file is in an industry-standard format known as JEDEC (Figure 16-18).

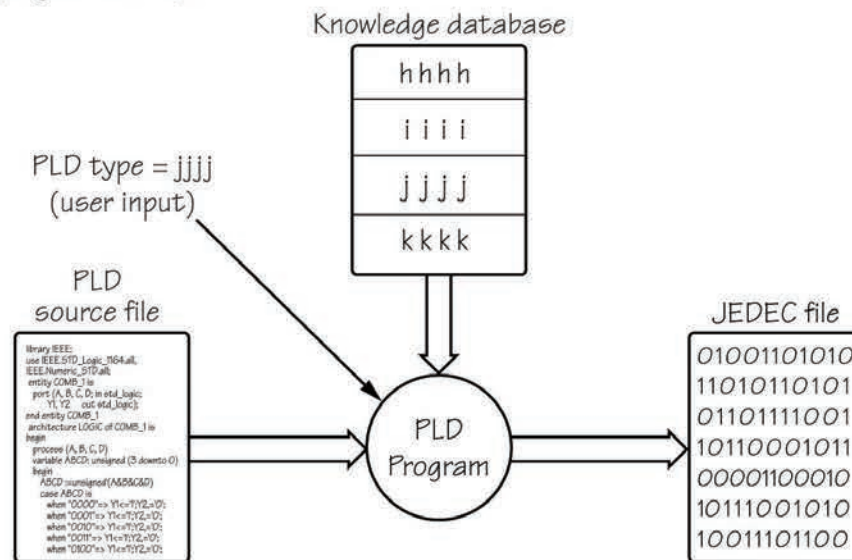


Figure 16-18. Using the textual source file to create a fuse file

As an alternative to the user specifying a particular device, the program can be instructed to automatically select the best device for the task. The program can base its selection on a variety of criteria, such as the speed, cost, and power consumption of the devices. The program may also be used to partition a large design across several devices, in which case it will output a separate JEDEC file for each device.

Finally, the designer takes a virgin device of the appropriate type and places it in a socket on a special tool, which may be referred to as a *programmer*, *blower*, or *burner*. The main computer passes the JEDEC file to the programmer, which uses the contents of the file to determine which fuses to blow. The designer presses the GO button, the burner applies the appropriate signals to the device's inputs, and a new device is born (Figure 16-19).

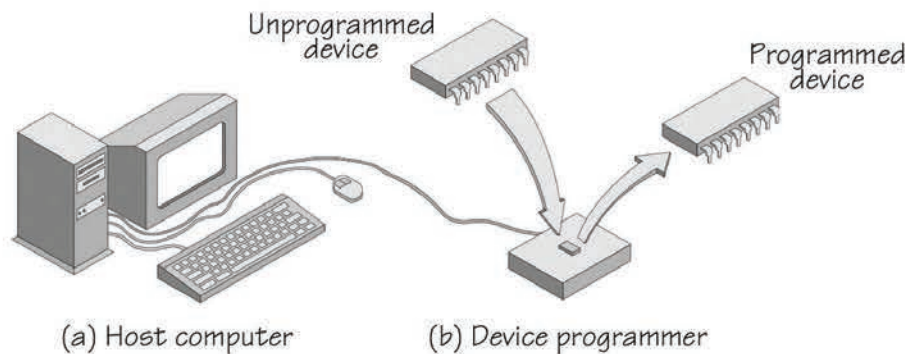


Figure 16-19. Programming a physical PLD

Reprogrammable PLDs

One consideration with fusible link and antifuse technologies is that, once they have been programmed, there is no going back. This may be of particular concern with PROMs as the data they store is prone to change.

EPROMs

One alternative is a technology known as *erasable programmable read-only memory (EPROM)*. An EPROM transistor has the same basic structure as a standard MOS transistor, but with the addition of a second polysilicon *floating gate* isolated by layers of oxide (Figure 16-20).

In its unprogrammed state, the floating gate is uncharged and doesn't affect

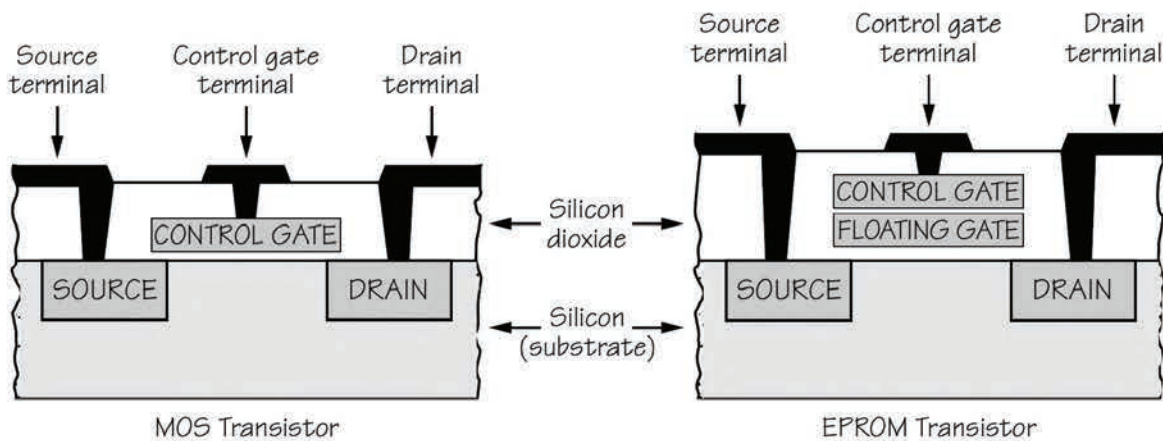


Figure 16-20. Standard MOS transistor versus EPROM transistor

the normal operation of the control gate. To program the transistor, a relatively high voltage in the order of 12V is applied between the control gate and drain terminals. This causes the transistor to be turned hard on, and energetic electrons force their way through the oxide into the floating gate in a process known as *hot (high energy) electron injection*. When the programming signal is removed, a negative charge remains on the floating gate. This charge is very stable and will not dissipate for more than a decade under normal operating conditions. The stored charge on the floating gate inhibits the normal operation of the control gate, and thus distinguishes those cells that have been programmed from those which have not.

EPROM cells are efficient in terms of silicon real estate, being half the size of DRAM cells and an order of magnitude smaller than fusible links. However, their main claim to fame is that they can be erased and reprogrammed. An EPROM cell is erased by discharging the electrons on the floating gate. The energy required to discharge the electrons is provided by a source of ultraviolet (UV) radiation. An EPROM device is delivered in a ceramic package with a small quartz window in the top; this window is usually covered with a piece of opaque sticky tape. For the device to be erased, the tape is first removed from the circuit board, the quartz window is uncovered, and the package is placed in an enclosed container with an intense ultraviolet source.

The main problems with EPROM devices are their expensive ceramic packages with quartz windows and the time it takes to erase them, which is in the order of 20 minutes. A foreseeable problem with future generations of these

devices is paradoxically related to improvements in process technologies that allow transistors to be made increasingly smaller. The problem is one of scaling, because not all of the structures on a silicon chip are shrinking at the same rate. Most notably, the transistors are shrinking faster than the metal interconnections and, as the features become smaller, a larger percentage of the surface of the die is covered by metal. This makes it difficult for the EPROM cells to absorb the ultraviolet and increases the required exposure time.

EEPROMs

A somewhat related technology is that of *electrically-erasable programmable read-only memory* (EEPROM or E²PROM). An E²PROM cell is approximately 2.5 times larger than an EPROM cell because it contains two transistors. One of the transistors is similar to that of an EPROM transistor in that it contains a floating gate, but the insulating oxide layers surrounding the floating gate are very much thinner. The second transistor can be used to erase the cell electrically, and E²PROM devices can typically be erased and reprogrammed on a word-by-word basis.

FLASH

Finally, a development known as FLASH can trace its ancestry to both EPROM and E²PROM technologies. The name FLASH was originally coined to reflect the technology's rapid erasure times compared to EPROM. Components based on FLASH can employ a variety of architectures. Some have a single floating gate transistor cell with the same area as an EPROM cell, but with the thinner oxide layers characteristic of an E²PROM component. These devices can be electrically erased, but only by erasing the whole device or a large portion of it. Other architectures have a two-transistor cell—which is very similar to that of an E²PROM cell—allowing them to be erased and reprogrammed on a word-by-word basis.

Initial versions of FLASH could only store a single bit of data per cell. By 2002, however, technologists were experimenting with a number of different ways of increasing this capacity. One technique involves storing distinct levels of charge in the FLASH transistor's floating gate to represent two bits per cell. An alternative approach involves creating two distinct storage nodes in a layer below the gate, thereby supporting two bits per cell.

In-System Programmable

One advantage of E²PROM and FLASH devices is that they can be reprogrammed while still on the circuit board, in which case they may be referred to as *in-system programmable (ISP)*.¹⁴ A large proportion of FLASH and E²PROM devices use hot (high energy) electron injection similar to EPROM and require 12V signals for programming. Some designers see this as an advantage, because these devices cannot be accidentally erased by means of the 5.0V (or lower, such as 3.3V, 3.0V, etc.) signals used during the circuit's normal operation. However, an increasing number of devices use *cold* (low energy) *Fowler-Nordheim electron tunneling* and can be programmed using the standard signal voltage (5.0V, 3.3V, 3.0V, etc.).

In addition to their use as memories, EPROM, E²PROM, and FLASH technologies are also employed in other PLD types, which are generically known as EPLDs, E²PLDs, and FLASH PLDs. For example, each fusible link in a standard PLA is replaced by an EPROM cell in an EPLA, an E²PROM cell in an E²PLA, or a FLASH cell in a FLASH PLA. Regardless of the technology used—from fusible link to FLASH—all of the components are created on the surface of a single piece of silicon substrate. However, it may be useful to visualize the device as comprising two distinct strata, with the AND and OR arrays on the lower level and the links “floating” above them (Figure 16-21).

In yet another alternative, some PLDs are constructed with each link being replaced with an SRAM cell. Unlike the other reprogrammable options, PLDs

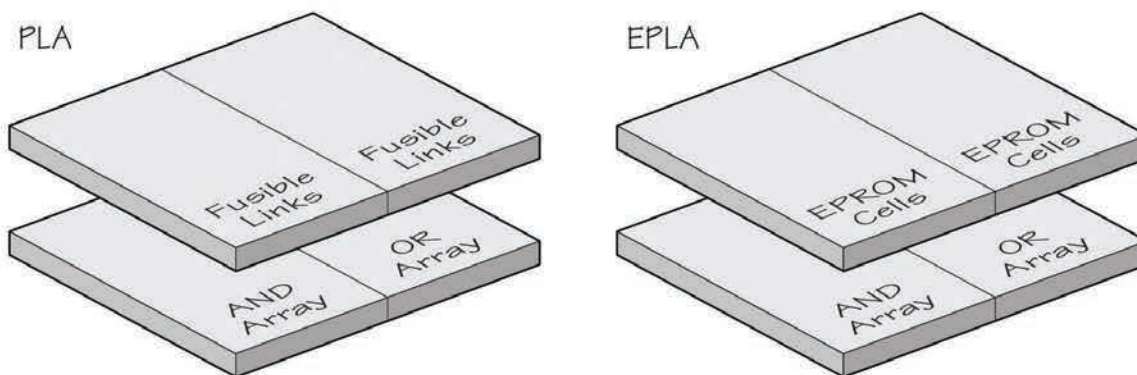


Figure 16-21. A fusible-link PLA versus an EPLA

¹⁴ In conversation, ISP is spelled out as “I-S-P.”

based on SRAM cells are volatile and lose their data when power is removed from the system. However, these devices can be dynamically reprogrammed while the circuit is in use. Some SRAM-based devices employ a double-buffering scheme using two SRAM cells per link. In this case, the device can be reprogrammed with a new pattern while it is still operating with an old pattern. At the appropriate time, the device can be instructed to switch from one bank of SRAM cells to the other.

Reprogrammable devices also convey advantages over fusible link and antifuse devices in that they can be more rigorously tested at the factory. For example, reprogrammable devices typically undergo one or more program and erase cycles before being shipped to the end user.

Complex PLDs (CPLDs)

The term *complex PLD (CPLD)* is generally taken to refer to a class of devices that contain a number of simple PLA or PAL functions—generically referred to as *simple PLDs (SPLDs)*—sharing a common programmable interconnection matrix (Figure 16-22).

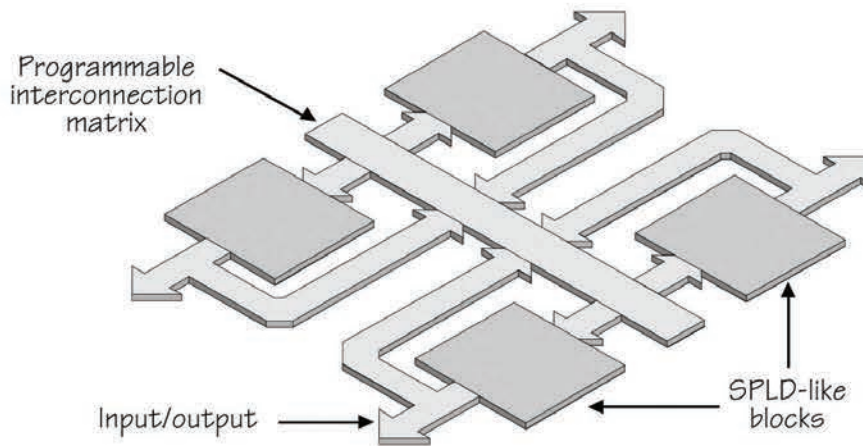


Figure 16-22. Generic CPLD structure

In addition to programming the individual PLD blocks, the connections between the blocks can be configured by means of the *programmable interconnection matrix*. Additional flexibility may be provided with a CPLD whose links are based on SRAM cells. In this case, it may be possible to configure each PLD block to either act in its traditional role or to function as a block of SRAM.

Field-Programmable Gate Arrays (FPGAs)

SPLDs and CPLDs are tremendously useful for a wide variety of tasks, but they are somewhat limited by the structures of their programmable AND and OR planes. At the other end of spectrum are full-blown *application-specific integrated circuits (ASICs)*.¹⁵

Around the middle of the 1980s, a new breed of devices called *Field-Programmable Gate Arrays (FPGAs)* began to appear on the scene. These devices combined many aspects traditionally associated with ASICs (such as high-density) with characteristics traditionally associated with SPLDs and CPLDs (such as the ability to program them in the field).

FPGAs are somewhat difficult to categorize, because each FPGA vendor fields a proprietary architecture; however, a generic architecture that illustrates the sophistication of FPGAs could be represented as shown in Figure 16-23.

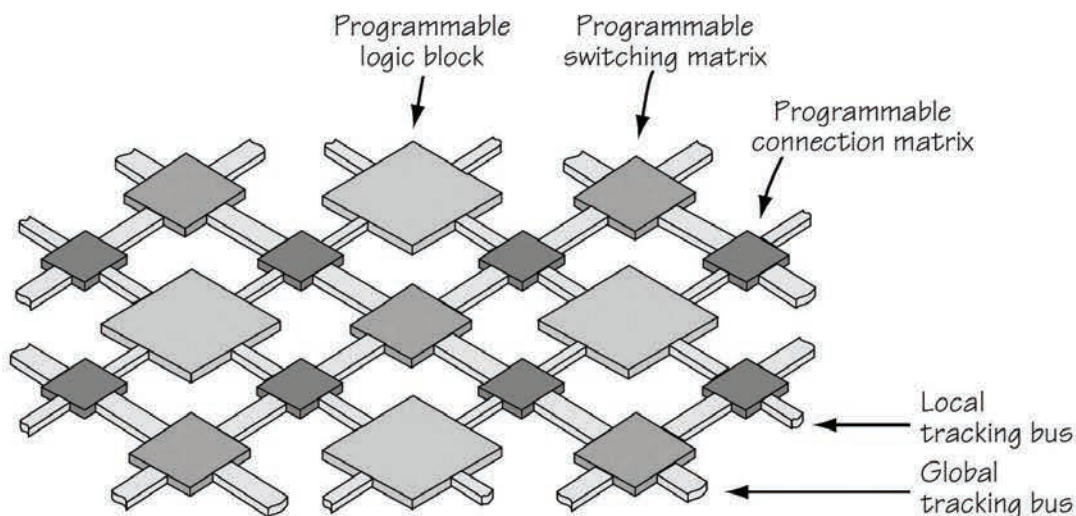


Figure 16-23. Generic field-programmable gate array (FPGA) structure

The device consists of a number of programmable logic blocks, each connected to a number of programmable connection matrices, which are in turn connected to a number of programmable switching matrices. Each programmable logic block may contain a selection of primitive logic gates and register elements. By programming the appropriate links, each logic block can

¹⁵ ASICs are introduced in detail in Chapter 17.

be individually configured to provide a variety of combinational and/or sequential logic functions. The programmable connection matrices are used to establish links to the inputs and outputs of the relevant logic blocks, while the programmable switch matrices are used to route signals between the various connection matrices. In short, by programming the appropriate links in the connection and switch matrices, the inputs and outputs of any logic block can be connected to the inputs and outputs of any other logic block.

The majority of FPGAs are based on SRAM programmable switches (some are based on antifuse technology, but these are in the minority). The two predominant architectural variants are to use *look-up tables (LUTs)* or multiplexers. First consider the LUT approach, of which there are two main sub-variants (Figure 16-24)

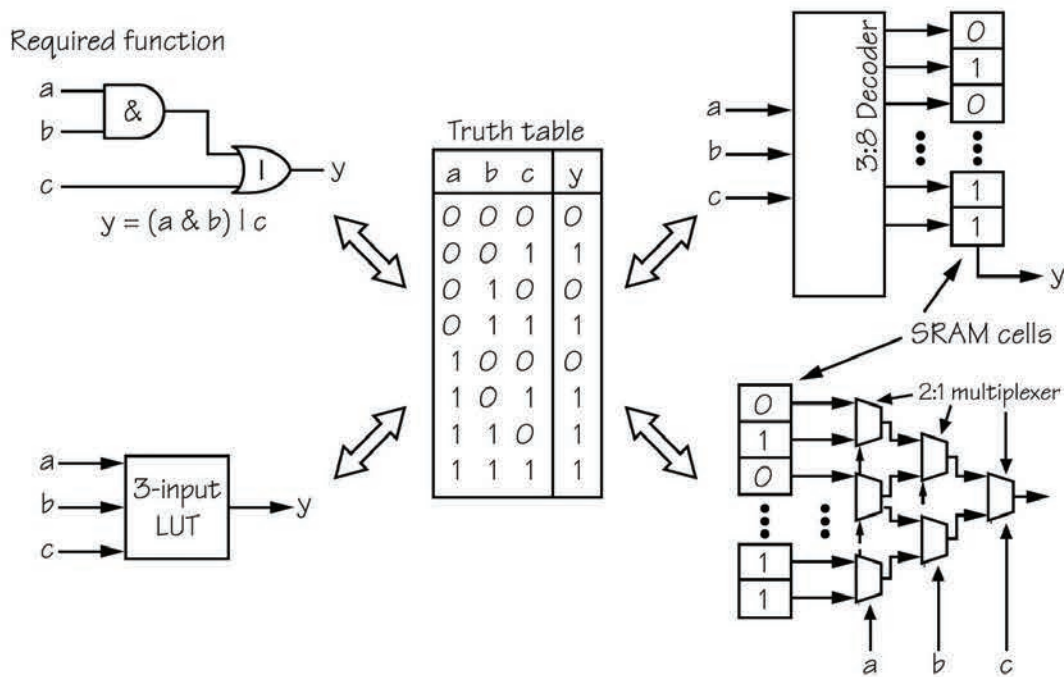


Figure 16-24: Look-up table (LUT)-based FPGAs

In Figure 16-24—which assumes 3-input LUTs¹⁶—the required function (upper left) is converted into an LUT (lower left) whose contents can be

¹⁶ FPGA vendors can spend countless hours debating the advantages and disadvantages of 3-input versus 4-input versus 5-input LUTs (you can only imagine how much fun it is to be cornered by one of their salespersons at a party).

represented using a truth table (center). One technique involves using a 3:8 decoder (upper right) to select one of eight SRAM cells that contain the required truth table output values. Alternatively, the SRAM cells can be used to drive a “pyramid” structure of multiplexers (lower right), where the multiplexers are themselves controlled by the values on the input variables and “funnel down” to generate the output.

As opposed to LUTs, some FPGA architectures are based almost exclusively on multiplexers (Figure 16-25).

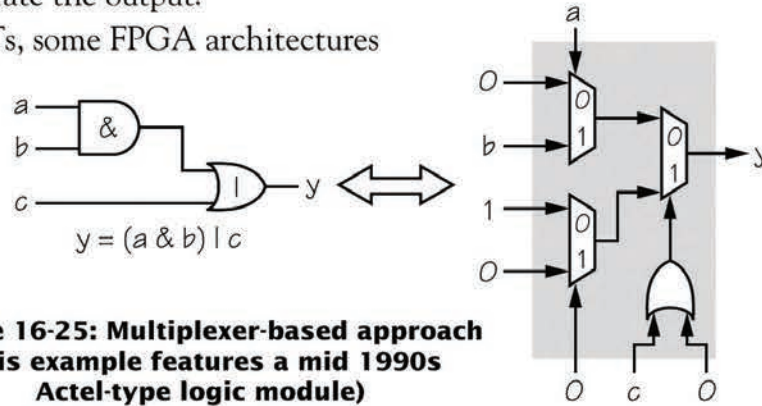


Figure 16-25: Multiplexer-based approach (this example features a mid 1990s Actel-type logic module)

There are several considerations to bear in mind when using FPGAs. First, interconnect delays are not as predictable as they are with SPLDs and CPLDs. Second, each vendor employs special fitting software to map designs into their devices, which makes it pretty darn tricky to migrate a design from one vendor to another while maintaining anything like the same propagation delays. Third, the majority of synthesis tools are geared towards fine-grained ASIC architectures, so they output gate-level netlists, but FPGA fitting tools usually do a less than superb job of placement or packing from these netlists. Having said this, there have been tremendous strides in FPGA design tools over the years, and modern versions of these tools work very hard to address these issues.

The original FPGAs contained a significant number of logic gates, but nothing like the little rascals you see these days. For example, by 2002, platform FPGAs from Xilinx (www.xilinx.com) were available with upwards of 10 million equivalent logic gates, which is pretty impressive whichever way you look at it. Furthermore, FPGAs of this caliber don't simply sport simple programmable logic, but may include such things as embedded microprocessor cores, incredibly fast communications functions, large blocks of RAM, large numbers of special-purpose macro-cells such as multipliers and adders, and . . . the list goes on.

Why Use Programmable ICs?

A single programmable device can be used to replace a large number of simpler devices, thereby reducing board space requirements, optimizing system performance by increasing speed and decreasing power consumption, and reducing the cost of the final system. Additionally, replacing multiple devices with one or more programmable devices increases the reliability of the design. Every integrated circuit in a design has to be connected to the circuit board by means of soldered joints, and every joint is a potential failure mechanism.¹⁷ Thus, using programmable devices reduces the number of components, which reduces the number of joints, thereby reducing the chances of a bad or intermittent connection.

Electronic designs typically undergo a number of changes during the course of their development. A design modification may require the addition or removal of a number of logic functions and corresponding changes to the tracks connecting them. However, when a programmable device is used to replace a number of simpler devices, the effects of any design modifications are minimized.

Reprogrammable EPLDs and E²PLDs are often employed during the initial stages of a design. Once the design has stabilized, these devices may be replaced by cheaper, one-time programmable PLD equivalents. In certain cases involving large production runs, both reprogrammable and one-time programmable devices may eventually be replaced by mask-programmed equivalents, in which the patterns are hard-coded during the construction of the component.

Traditionally, the majority of designers have not availed themselves of the capabilities for in-circuit reprogrammability offered by devices based on the E²PROM technology. However, the current interest in *design-for-test (DFT)* and the increasing requirement for portable computer and telecommunication equipment are driving an interest in in-circuit reprogrammability. This is especially true in the case of components based on FLASH and SRAM technologies, which allow these devices to be easily and quickly programmed, erased, and reprogrammed while remaining embedded in the middle of a system.

To facilitate the testing of a system, these highly reprogrammable devices can first be configured so as to make internal signals on the circuit board

¹⁷ Circuit boards are introduced in more detail in Chapter 18.

available to the test system. After the circuit has been tested, the devices can be erased and reprogrammed with the patterns required to make the board perform its normal function. In some cases, it may make sense to create a single design that can be used to replace a number of different variants. The designers may create one circuit board type that can be reprogrammed with different features for diverse applications.

Portable electronic equipment such as computers can be reprogrammed in the field or remotely by means of a telephone link. Of most interest is the concept of *adaptive hardware*, which can be reprogrammed on-the-fly. For example, state machines may be reprogrammed to follow different sequences, pattern detectors may be reprogrammed to search for different patterns, and computational devices may be reprogrammed with different algorithms. For all of the reasons discussed above, programmable integrated circuits continue to form an extremely significant market segment.

Application-Specific Integrated Circuits (ASICs)

As opposed to a standard “off-the-shelf” integrated circuit whose function is specified by the manufacturer, an *application-specific integrated circuit* (ASIC)¹ is a device whose function is determined by design engineers to satisfy the requirements of a particular application. (The programmable logic devices introduced in the previous chapter fall into both camps; in one respect they are standard devices in that they are mass produced, but they may also be classed as subspecies of ASIC because of the design engineers’ ability to customize their functionality.)

In 1967, Fairchild Semiconductor introduced a device called the *Micromosaic*, which contained a few hundred transistors. The key feature of the *Micromosaic* was that the transistors were not initially connected together. Design engineers used a computer program to specify the function they wished the device to perform. This program determined the necessary interconnections required to link the transistors and generated the masks required to complete the device. Although relatively simple, the *Micromosaic* is credited as being the forerunner of the modern ASIC and the first real application of *computer-aided design* (CAD).

A formal classification of ASICs tends to become a bit fluffy around the edges. However, in addition to the plethora of programmable logic devices introduced in Chapter 16, there are generally taken to be three other major categories: *gate array* (including sea-of-gates), *standard cell* (including compiled cells), and *full custom* (which also includes compiled cells) (Figure 17-1).

¹ In conversation, ASIC is pronounced “A-SIC,” that is, by spelling out the “A” to rhyme with “hay” followed by “SIC” to rhyme with “tick.”

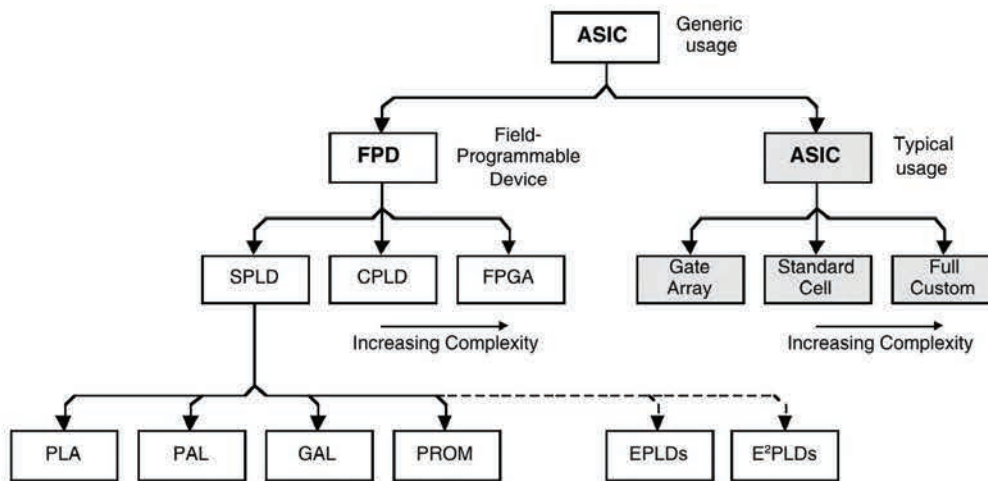


Figure 17-1. Generic and typical usage of the term ASIC

Gate Array Devices

A silicon chip may be considered to consist of two major facets: components such as transistors and resistors, and the tracks connecting them together. In the case of *gate arrays*, the ASIC vendor prefabricates wafers of silicon chips containing unconnected transistors and resistors.² Gate arrays are based on the concept of a *basic cell*, which consists of a selection of components; each ASIC vendor determines the numbers and types of components provided in their particular basic cell (Figure 17-2).

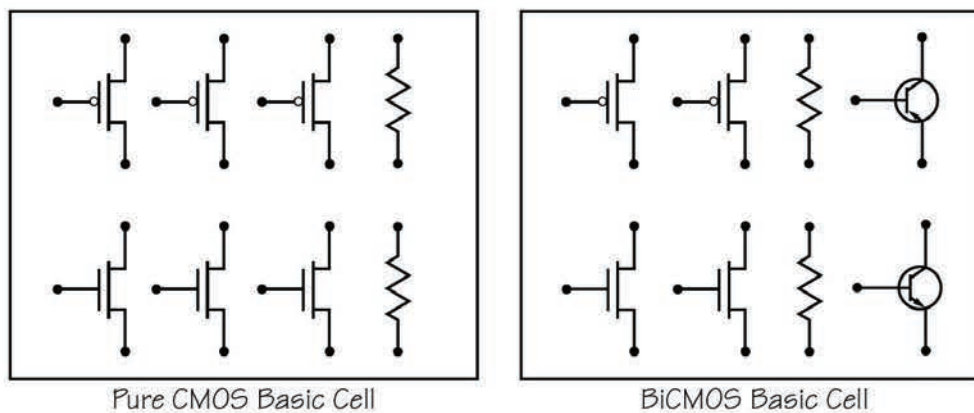


Figure 17-2. Examples of gate array basic cells

² In the early days, gate arrays were also known as *uncommitted logic arrays* (ULAs), but this term has largely fallen into disuse.

Channeled Gate Arrays

In the case of *channeled gate arrays*, the basic cells are typically presented as either single-column or dual-column arrays; the free areas between the arrays are known as the *channels* (Figure 17-3). (Note that, although these diagrams feature individual silicon chips, at this stage in the process these chips are still embedded in the middle of a wafer.)

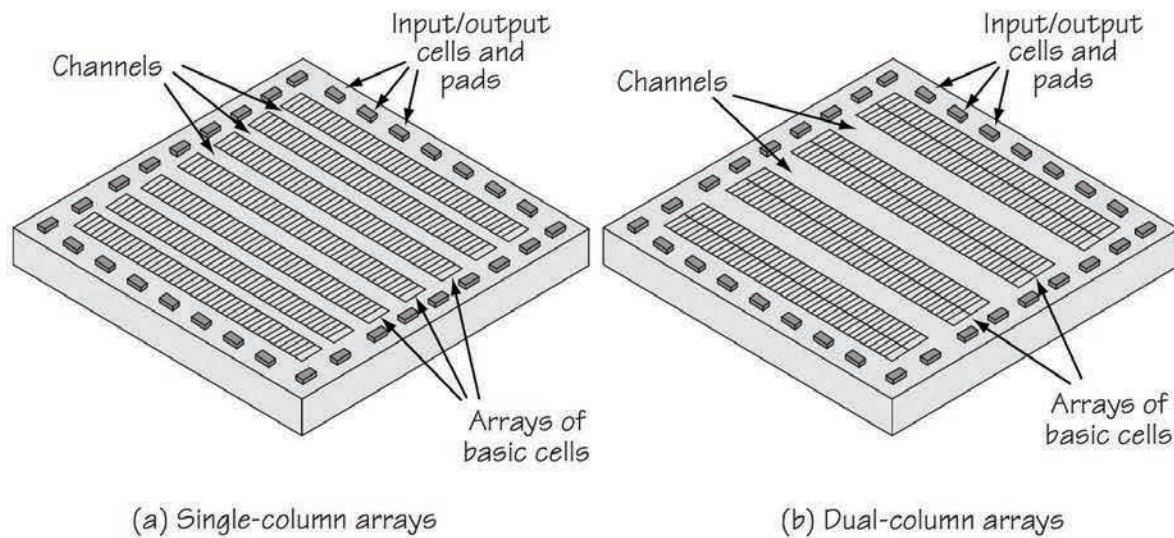


Figure 17-3. Channeled gate array architectures

Channel-less Gate Arrays

In the case of *channel-less* or *channel-free* devices, the basic cells are presented as a single large array. The surface of the device is covered in a “sea” of basic cells, and there are no dedicated channels for the interconnections. Thus, these devices are popularly referred to as *sea-of-gates* or *sea-of-cells* (Figure 17-4).

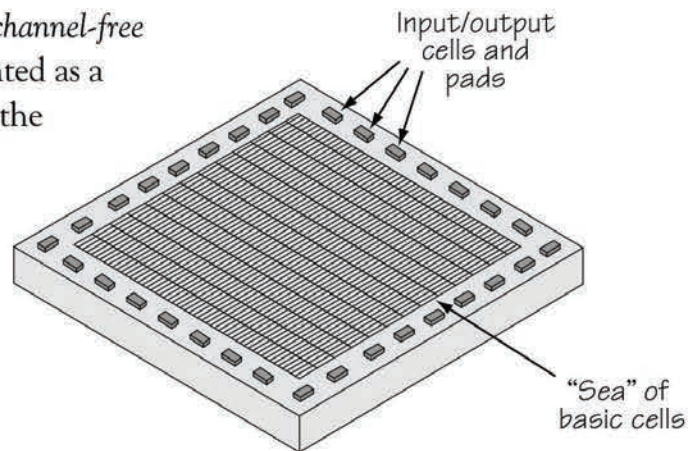


Figure 17-4. Channel-less gate array architecture