Building Blocks for Digital Design

The construction of most digital systems is a large task. Disciplined designers in any field will subdivide the original task into manageable subunits – building blocks – and will use the standard subunits wherever possible. In digital hardware, the building blocks have such names as *adders, registers,* and *multiplexers.*

Logic theory shows that all digital operations may be reduced to elementary logic functions. We could regard a digital system as a huge collection of AND, OR, and NOT circuits, but the result would be unintelligible. We need to move up one level of abstraction from gates and consider some of the common operations that digital designers wish to perform. Some candidates are:

(a) Moving data from one part of the machine to another.
(b) Selecting data from one of several sources.
(c) Routing data from a source to one of several destinations.
(d) Transforming data from one representation to another.
(e) Comparing data arithmetically with other data.
(f) Manipulating data arithmetically or logically, for example, summing two binary numbers.

We can perform all these operations with suitable arrangements of AND, OR, and NOT gates, but always designing at this level would be onerous, lengthy, and error-prone. Such an approach would be comparable to programming every software problem in binary machine language. Instead, we need to develop building blocks to perform standard digital system operations. The building blocks will allow us to suppress much irrelevant detail and design at a higher level. The procedure is analogous to giving the architect components such as doors, walls, and stairs instead of insisting that he design only with boards, nails, and screws.

INTEGRATED CIRCUIT COMPLEXITY

In Chapter 2, you studied low-level building blocks–AND, OR, and NOT. These come in integrated circuit packages containing a few gates that are not interconnected and that can be used in synthesizing elementary circuits from Boolean algebraic equations. The industry's name for devices of this complexity, containing up to 10 gates, is *SSI (small-scale integration).*

Digital designers find that operations such as those we listed above–(a) to (f)–occur in nearly any system design. As a result, manufacturers have provided integrated circuit packages of interconnected gates to perform these operations. Chips that contain from 10 to about 100 gates are called *MSI (mediumscale integration).* Circuits with a nominal complexity of about 100 to 1,000 gates are classified as *LSI (large-scale integration),* and circuits with more than the equivalent of 1,000 gates are called *VLSI (very-large-scale integration).* Microprocessors and memory arrays are examples of VLSI integrated circuits, as are many special-purpose chips. The boundaries between the categories are only suggestive; in fact, the use of the term LSI as a measure of chip complexity is waning.

Each class of integrated circuit–SSI, MSI, LSI, and VLSI–is important at its appropriate level of design. The digital designer should try to design at the highest conceptual level suitable to the problem, just as the software specialist should seek to use prepackaged programs or a high-level language instead of assembly language when possible. In software programming, the accomplished problem solver has not only a knowledge of Fortran, but also of computer organization, system structure, assembly language, and machine processes. Similarly, to achieve excellence in solving problems with digital hardware, we need skill in using all our tools, from the elementary to the complex.

COMBINATIONAL BUILDING BLOCKS

Combinational and Sequential Circuits

In this chapter, we will develop a set of building blocks that have hardware implementations of the MSI or LSI level of complexity, and have no internal storage capacity, or "memory." Such circuits, with outputs that depend only on the present values of the inputs, are called *combinational.* The important class of circuits that depend also on the condition of past outputs is called *sequential.* We will present sequential circuits and sequential building blocks in Chapter 4. Table 12-1, at the end of Chapter 12, is a list of useful SSI and MSI integrated circuits.

The Multiplexer

A multiplexer is a device for selecting one of several possible input signals and presenting that signal to an output terminal. It is analogous to a mechanical switch, such as the selector switch of a stereo amplifier (Fig. 3-1). The amplifier switch is used for selecting the input that will drive the speaker. Except for the moving hand, the electronic analog is easily constructed. We use Boolean variables instead of mechanical motion to select a given input. Consider a two-position switch with inputs **A** and **B** and output **Y**, such as shown in Fig. 3-2. Introduce a variable **S** to describe the position of the switch and let **S** = **0** if the switch is up and **S** = **1** if the switch is down. A Boolean equation for the output **Y** is

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

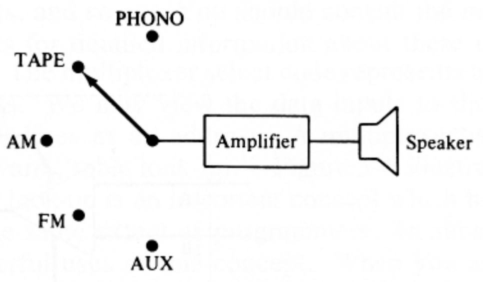


Figure 3–1. A mechanical selector switch.

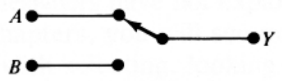


Figure 3–2. A two-position mechanical switch.

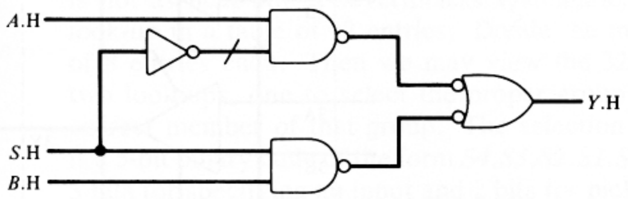


Figure 3–3. Implementation of the electronic switch $Y = A \bullet \overline{S} + B \bullet S$.

Using this equation, we can build an electronic analog of the switch; Fig. 3-3 is one design. There, **S** is the *select* input.

Commercial MSI devices correspond to the more complex switch shown in Fig. 3-4. The right-hand switch, called *an enable* (or *strobe)*, acts as a Boolean AND function. If we call the closed position of the enable switch **G** and the open position **G**, then

$$Y = G \bullet (A \bullet \overline{S} + B \bullet S) \qquad (3\text{–}1)$$

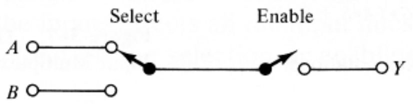


Figure 3–4. An enabled selector switch.

Figure 3-5 is a schematic for a device based on this equation that behaves like common commercial devices. Such a circuit is called a 2-input multiplexer *(mux):* its mixed-logic symbol and the representation of Eq. (3-1) are shown in Fig. 3-6. Figure 3-7 is a diagram of a typical commercial device, the 74LS157 Quad Two-Input Multiplexer. This chip has four 2-input multiplexer units (designated l through 4) packaged in a single integrated circuit chip, with each mux sharing a common select input **S** and a common enable input **En**.

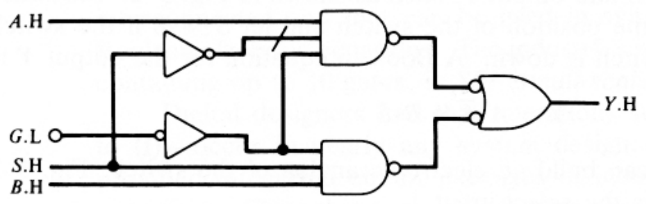
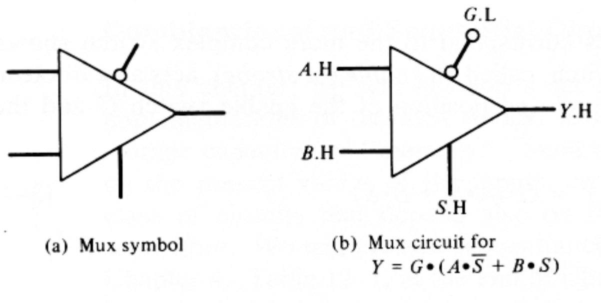


Figure 3–5. An enabled two-position electronic selector switch.



G.L
A.H
B.H
Y.H
S.H
(a) Mux symbol
(b) Mux circuit for $Y = G \bullet (A \bullet \overline{S} + B \bullet S)$

Figure 3–6. Mixed-logic multiplexer notations.

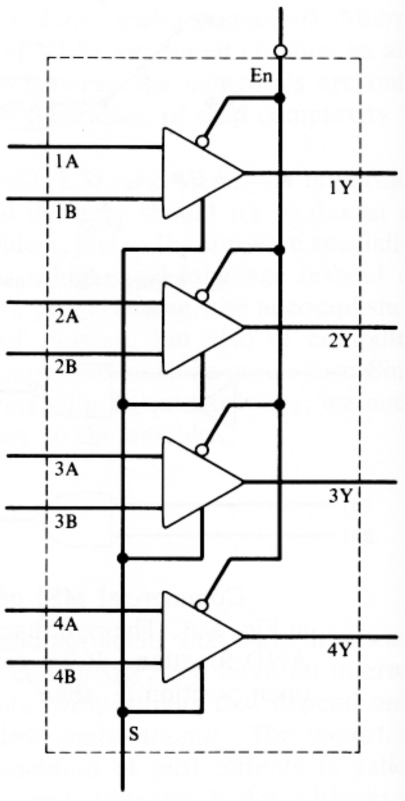


Figure 3–7. The 74LS157 Quad Two-Input Multiplexer.

If we desire to select an output from among more than two inputs, the multiplexer must have more than one select input. The select inputs to a mux form a binary code that identifies the selected data input. One select line has 2' = 2 possible values; two select lines allow the specification of $2^2 = 4$ different values. For instance, if we have select lines **S1** and **S0**, the pair **S1, S0** represents a binary number that may identify one of four possible inputs:

S1	*S0*	Selected input position
0	0	0
0	1	1
1	0	2
1	1	3

Commercial integrated circuits provide 2-, 4-, 8-, and 16-input multiplexers, with a variety of inverted and noninverted outputs, separate and common enable inputs, and so on. You should consult the manufacturers' integrated circuit data books for detailed information about these useful devices.

The multiplexer select code represents an *address*, or *index*, into the ordered inputs. We may view the data inputs to the mux as a vector or table, and the select lines as an address. A multiplexer is thus a hardware analog of a 1-bit software "table look-up." Figure 3-8 illustrates the analogy. In systems design, table look-up is an important concept which hardware designers have not exploited to the same extent as programmers. In subsequent chapters, you will see many powerful uses of this concept. When you are faced with selecting, looking up, or addressing one of a small number of items, think MUX.

Table look-up, or input selection, is convenient for up to 16 inputs, using the appropriate MSI integrated circuit chip. For more than 16 inputs the solution is not as neat but is nevertheless systematic. Suppose that you need to do a look-up in a table of 32 entries. Divide the inputs into groups, say four groups of 8 entries each. Then we may view the 32-element look-up as consisting of two look-ups, one to select the proper group of 8, and the second to pick the correct member of that group. The selection index for the 32-element look-up is a 5-bit binary code of the form **S4,S3,S2,SI,S0.** Each group of 8 inputs requires 3 bits for specifying an input and 2 bits for picking the proper group of 8. Figure 3-9 shows a realization that uses four 8-input multiplexers and one 4-input mux.

The conventional symbol for a multiplexer shows the inputs as having T = H, but the output may be either high- or low-active, depending on the particular chip. As mixed logicians, we realize that we may present all the inputs in T = L form without affecting the circuit; then the output will be of opposite polarity to that in the conventional symbol for the device. In Fig. 3-10 we show the two equivalent mixed-logic forms for an element of a 74LS352 Dual Four-Input Multiplexer, a circuit whose output polarity is inverted. Changing the polarity of the inputs affects all the input lines and the output (all the *data* paths) but has no effect on the selection or enabling systems.

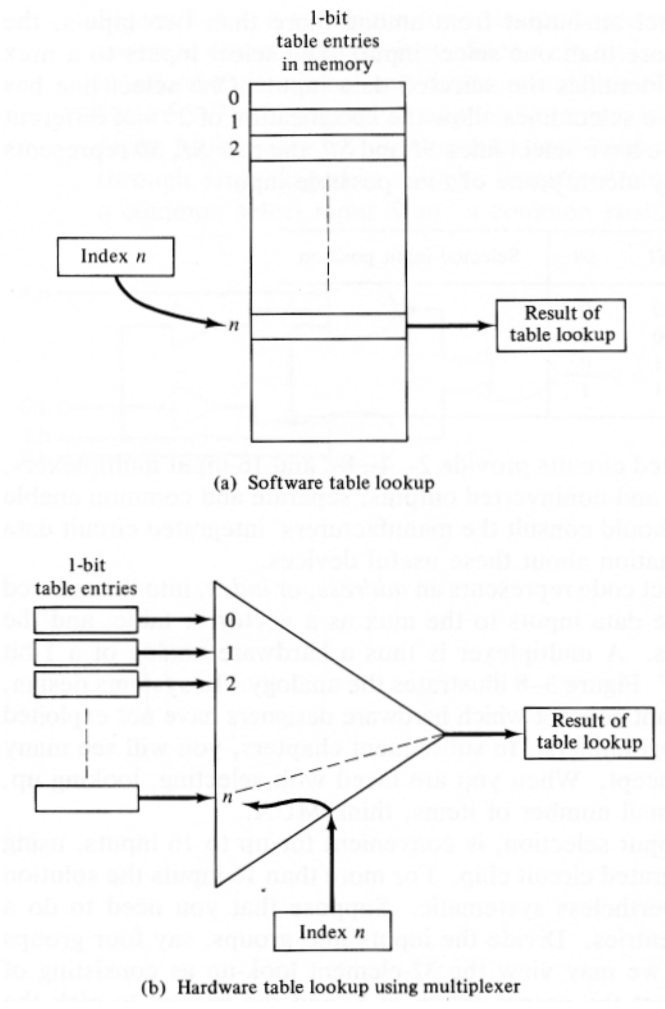


Figure 3–8. A hardware analog of a software table lookup. A single multiplexer provides a 1-bit lookup. Several multiplexers addressed by a common signal form a multibit lookup.

The Demultiplexer

A demultiplexer sends data from a single source to one of several destinations. Whereas the multiplexer is a data selector, the demultiplexer is a data distributor or data router. *A* mechanical analog is the switch used to route the power amplifier output of an automobile radio either to a front or a rear speaker, as illustrated in Fig. 3-11. This switch is the same type of two-position mechanical switch shown in Fig. 3-1. *A* mechanical switch can transmit a signal in either direction, whereas the electronic analog can transmit data in only one direction. Since we cannot use a multiplexer in the reverse direction, we are forced to provide a demultiplexer to handle this operation.

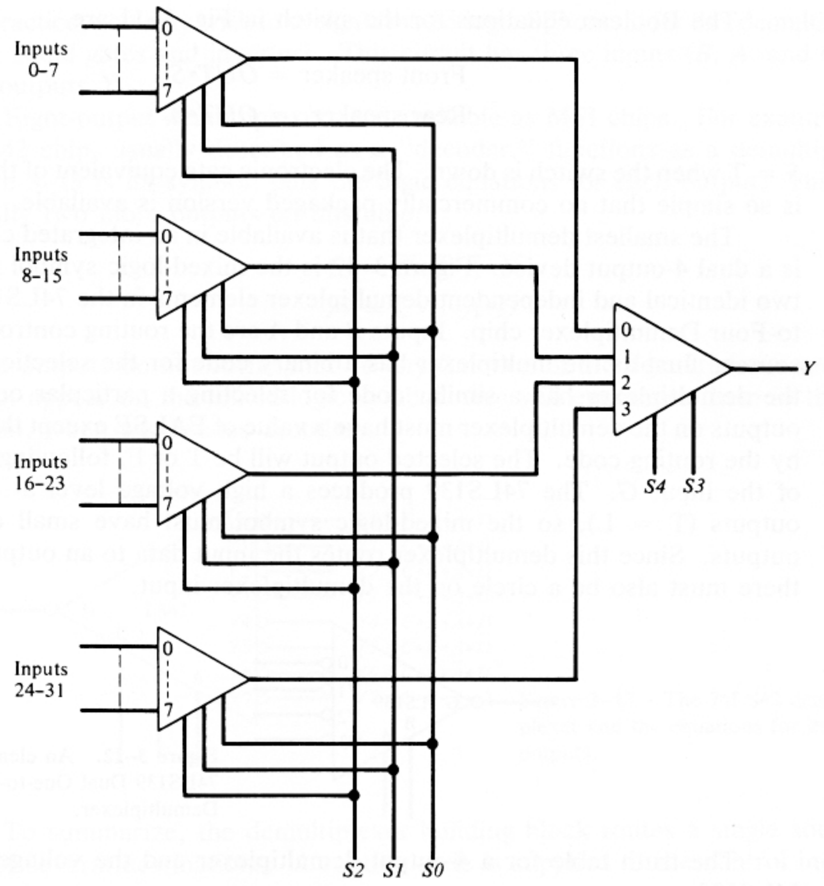


Figure 3–9. A 32-element table lookup.

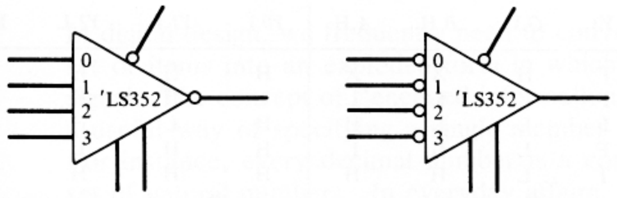


Figure 3–10. Mixed-logic symbols for the 74LS352 4-input multiplexer.

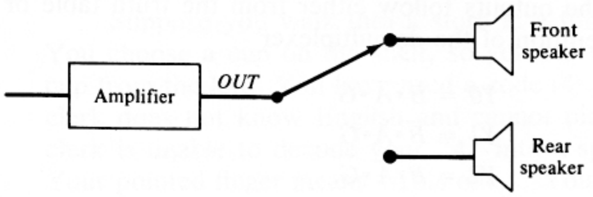


Figure 3–11. A mechanical distributor switch.

The Boolean equations for the switch in Fig. 3-11 are

$$\text{Front speaker} = OUT \bullet \overline{S}$$
$$\text{Rear speaker} = OUT \bullet S$$

S = **T** when the switch is down. The electronic gate equivalent of these equations is so simple that no commercially packaged version is available.

The smallest demultiplexer that is available in an integrated circuit package is a dual 4-output device. Figure 3-12 is the mixed-logic symbol for one of the two identical and independent demultiplexer elements in the 74LS139 Dual One-to-Four Demultiplexer chip. Inputs **B** and **A** are the routing controls for the data source. Just as the multiplexer has a binary code for the selection of an input, the demultiplexer has a similar code for selecting a particular output. All the outputs on the demultiplexer must have a value of **FALSE** except the one selected by the routing code. The selected output will be **T** or **F**, following the condition of the input **G**. The 74LS139 produces a high voltage level at all unselected outputs (**T** = L), so the mixed-logic symbol must have small circles on the outputs. Since this demultiplexer routes the input data to an output unchanged, there must also be a circle on the demultiplexer input.

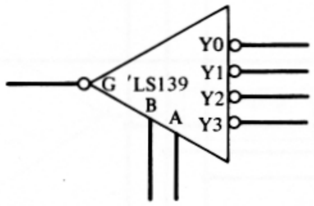


Figure 3–12. An element of the 74LS139 Dual One-to-Four Demultiplexer.

The truth table for a 4-output demultiplexer and the voltage table for the 74LS 139 are:

Demultiplexer logic							74LS139 Voltage						
G	*B*	*A*	*Y0*	*Y1*	*Y2*	*Y3*	*G*.L	*B*.H	*A*.H	*Y0*.L	*Y1*.L	*Y2*.L	*Y3*.L
F	X	X	F	F	F	F	H	X	X	H	H	H	H
T	F	F	T	F	F	F	L	L	L	L	H	H	H
T	F	T	F	T	F	F	L	L	H	H	L	H	H
T	T	F	F	F	T	F	L	H	L	H	H	L	H
T	T	T	F	F	F	T	L	H	H	H	H	H	L

The logic equations for the outputs follow either from the truth table or from the description of the operation of the demultiplexer

$$Y0 = \overline{B} \bullet \overline{A} \bullet G$$
$$Y1 = \overline{B} \bullet A \bullet G$$
$$Y2 = B \bullet \overline{A} \bullet G$$
$$Y3 = B \bullet A \bullet G$$

For practice, you may wish to design a mixed-logic SSI circuit for this demultiplexer, using Nand gates and inverters. This circuit has three inputs (**B**, **A**, and **G**) and four outputs $\mathbf{Y_i}$

Eight-output demultiplexers are available as MSI chips. For example, the 74LS42 chip, usually described as a "decoder," functions as a demultiplexer. Figure 3-13 is its symbol, plus the logic equations for each output. There are actually two more outputs on this chip:

$$Y8 = \overline{C} \bullet \overline{B} \bullet \overline{A} \bullet \overline{D}$$
$$Y9 = \overline{C} \bullet \overline{B} \bullet A \bullet \overline{D}$$

These outputs do not correspond to any function of the demultiplexer and they do not appear on the mixed-logic symbol. We will encounter them in the next section, when we discuss decoders.

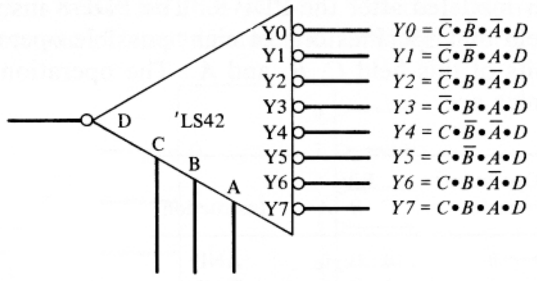

'LS42
D
C
B
A
Y0
Y1
Y2
Y3
Y4
Y5
Y6
Y7
$Y0 = \overline{C} \bullet \overline{B} \bullet \overline{A} \bullet D$
$Y1 = \overline{C} \bullet \overline{B} \bullet A \bullet D$
$Y2 = \overline{C} \bullet B \bullet \overline{A} \bullet D$
$Y3 = \overline{C} \bullet B \bullet A \bullet D$
$Y4 = C \bullet \overline{B} \bullet \overline{A} \bullet D$
$Y5 = C \bullet \overline{B} \bullet A \bullet D$
$Y6 = C \bullet B \bullet \overline{A} \bullet D$
$Y7 = C \bullet B \bullet A \bullet D$

Figure 3–13. The 74LS42 demultiplexer and the equations for its outputs.

To summarize, the demultiplexer building block routes a single source to one of several destinations. A routing code is supplied to the control inputs to select the destination.

The Decoder

In digital design, we frequently need to convert an encoded representation of a set of items into an exploded form in which each item in the set has its own signal. The concept of "encoded information" pervades our lives. Encoding is a useful way of specifying a single member of a large set in a compact form. For instance, every decimal number is a code for a particular member of the set of natural numbers. In everyday affairs, we usually do not need to decode the code explicitly, but sometimes the decoding becomes necessary.

Suppose you walk into a store in a foreign country to buy a coffee cup. You choose a cup on the shelf, so you tell the clerk that you want the fourth cup from the left. You have used a code (4) to identify the desired cup, but the clerk does not know English and cannot pick out the correct cup. Since the clerk is unable to decode your "4" into a specific item, you point to the cup. Your pointed finger means "This one." You were forced to decode your code.

Whenever we use a number to designate a particular object, decoding must occur. Usually, we do this implicitly or intuitively, without thinking about it, but sometimes, as in the china shop, the decoding becomes very explicit.

In hardware, codes are frequently in the form of binary numbers and, in most cases, the decoding required to gain access to an item is buried within a building block. For example, an 8-input multiplexer has a 3-bit select code to specify the particular input. We purposely include within the mux the decoding of the select code – the mux building block contains the circuitry to translate "input 4" on the control lines to *"this* input."

In computer programming we specify a memory location by giving its address. In the hardware (the memory unit of the computer), this numeric address must be decoded to gain access to the particular memory cell.

Another common use of codes is in the operation code of a typical computer instruction. Most computers allow only one operation to be specified in each instruction, and the operation code describes the particular operation. In Part II of this book you will study the art of digital design and will participate in the design of a minicomputer modeled after the PDP-8. The PDP-8 instruction has a 3-bit operation code field that specifies one of eight possible operations. For now we will call the 3 bits of this field **C**, **B**, and **A**. The operation codes and their instruction mnemonics are

Operation code	Bits C	B	A	Instruction
0	0	0	0	AND
1	0	0	1	TAD
2	0	1	0	ISZ
3	0	1	1	DCA
4	1	0	0	JMS
5	1	0	1	JMP
6	1	1	0	IOT
7	1	1	1	OP

From the viewpoint of the computer programmer, the decoding of the operation code is buried inside the computer. But we are studying hardware design, and we must face the decoding problem squarely. To implement this instruction set, we require eight logic variables *(AND ... OP)* to control the specific activities of each instruction. Only one of these eight variables will be true at any time. The translation from the operation code into the individual logic variables is a decoding. We could build the decoding circuits from gates, using the methods of the previous chapters. For instance, the logic equations for two of the variables are

$$TAD = \overline{C}\bullet\overline{B}\bullet A$$
$$JMS = C\bullet\overline{B}\bullet\overline{A}$$

Decoding is so common in digital design that our appropriate posture is to package the decoding circuitry into a logical building block. The *decoder* building block has the characteristic that only one output is true for a given encoded input and the remaining outputs are false. Integrated circuits for decoders are available in several forms, typically with 2, 3, or 4 inputs. The main limitation on the size of the input code is the number of pins required for the outputs, since the number of outputs grows exponentially with the size of the code. For instance, a 3-bit binary decoder has 8 outputs ($2^3 = 8$), whereas a full 4-bit binary decoder has 16 outputs ($2^4 = 16$), requiring a larger integrated circuit package. Decoding a 5-bit binary number would produce 32 outputs – too many for useful packaging as an MSI chip.

The 74LS42 Four-Line-to-Ten-Line Decoder is a typical MSI decoder. This chip will decode a decimal number 0 through 9, expressed as a 4-bit binary code, into one of 10 individual outputs. The binary representation for 9 is 1001; the 4-bit codes from 1010 through 1111 do not arise from the encoding of the decimal numbers. By eliminating 6 of the possible 16 output pins, the 74LS42 circuit can be made to fit conveniently into a 16-pin chip. Figure 3-14a is the mixed-logic symbol for the 74LS42. The outputs are all low-active (T = L) – a characteristic of most commercial MSI decoders.

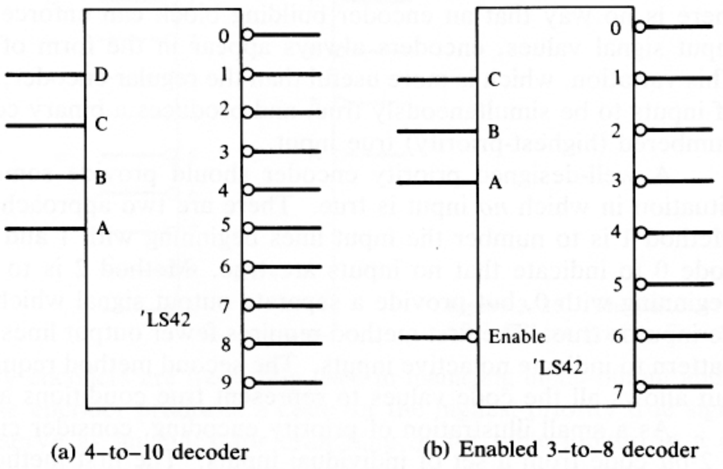


Figure 3–14. Two uses of the 74LS42 decoder.

The 74LS42 serves as a 3-bit decoder when the high-order bit ***(D)*** of the input code is false. The 8 outputs from the resulting 3-bit decoding are ***Y0*** through ***Y7***.

Another important use of the 74LS42 is as an enabled 3-to-8 decoder. The enabling feature is similar to that found on multiplexers in that outputs are always false unless the circuit is enabled. Whenever the **D** input to the 74LS42 is H, outputs **Y0** through **Y7** are false, regardless of the condition of the inputs **C, B**, and **A**; thus the 8 outputs for the 3-bit code are false and the chip is disabled. When input **D** is L, the chip is enabled for 3-bit decoding, and exactly 1 of the 8 outputs **Y0** through **Y7** is true. When the normal mixed-logic representation for the decoder is used, the **D** code bit acts as a disabling or not-enabling signal (with **T** = H). To use the 74LS42 as an enabled decoder, we usually represent the ***D*** input as an enabling signal (with **T** = L) rather than as a part of the input code. The mixed-logic notation for this enabled 3-to-8 decoder building block is shown in Fig. 3-14b.

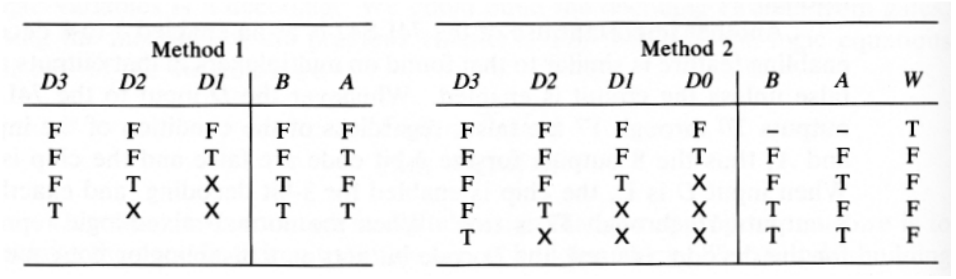
In addition to the 74LS42 chip's uses as a decoder, this same chip serves as a demultiplexer, as we saw in the previous section. In that application, we viewed the **A**, **B**, and **C** inputs as a select code, and the **D** input as a data signal to be routed to a selected destination. This duality of function is characteristic of decoders and demultiplexers. In practice, the decoding applications far outnumber those of demultiplexing.

The Encoder

The converse of the decoding operation is encoding–the process of forming an encoded representation of a set of inputs. This operation does not occur in digital design as frequently as decoding, yet it is of sufficient importance to be a candidate for one of our standard building blocks. In strict analogy with decoding, we should require that exactly one input to an encoder be true. Since there is no way that an encoder building block can enforce this restriction on input signal values, encoders always appear in the form of *priority encoders*. This variation, which is more useful than the regular encoder, allows any number of inputs to be simultaneously true, and produces a binary code for the highest-numbered (highest-priority) true input.

A well-designed priority encoder should provide some way to denote a situation in which *no* input is true. There are two approaches to this problem. Method 1 is to number the input lines beginning with 1 and reserve the output code 0 to indicate that no inputs are true. Method 2 is to number the inputs beginning with 0, but provide a separate output signal which is true only when no input is true. The first method requires fewer output lines but uses up a code pattern to indicate no active inputs. The second method requires an extra output but allows all the code values to represent true conditions at the input.

As a small illustration of priority encoding, consider circuits that produce a 2-bit code from a set of individual inputs. The first method will handle only 3 input lines, whereas the second method accommodates 4 inputs. Here are truth tables for the two styles of priority encoders (remember, **X** in the truth table means "both values" and — means "don't care"):



Method 1				
D3	*D2*	*D1*	*B*	*A*
F	F	F	F	F
F	F	T	F	T
F	T	X	T	F
T	X	X	T	T

Method 2						
D3	*D2*	*D1*	*D0*	*B*	*A*	*W*
F	F	F	F	–	–	T
F	F	F	T	F	F	F
F	F	T	X	F	T	F
F	T	X	X	T	F	F
T	X	X	X	T	T	F

Equations for the output variables can be derived by the methods described in Chapter 1. For instance, the logic equations for the outputs for method 2 are

$$\begin{aligned} B &= \overline{D3}\bullet D2 + D3 &&= D2 + D3 \\ A &= \overline{D3}\bullet\overline{D2}\bullet D1 + D3 &&= \overline{D2}\bullet D1 + D3 \\ W &= \overline{D3}\bullet\overline{D2}\bullet\overline{D1}\bullet\overline{D0} \end{aligned}$$

Commercial priority encoders come in a variety of forms representing both of these methods, sometimes also having an enabling control input, and occasionally with extra inputs and outputs to permit several chips to be cascaded. Customarily, the inputs and code outputs are low-active (**T** = L). Figure 3-15 is the mixed-logic symbol for a typical chip, the 74LS147 Ten-Line-to-Four-Line Priority Encoder, which conforms closely to method I above and produces a 4-bit output code, **D**, **C**, **B**, **A**. The chip has only 9 input lines, not 10 as the name suggests. The tenth line, corresponding to the output code 0000, is inferred from the absence of any true input.

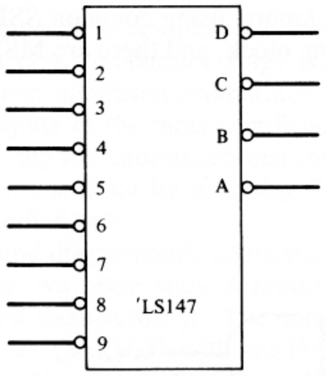


Figure 3–15. The 74LS147 Ten-Line-to-Four Line Priority Encoder.

Priority encoders are frequently used in managing input-output and interrupt signals. The encoder produces a code for the highest-priority true signal. This code may serve as an index for branching or for table lookup in a computer program.

The Comparator

Comparators help us to determine the arithmetic relationship between two binary numbers. Occasionally, we need to compare one set of *n* bits with another reference set of *n* bits to determine if the first set is identical to the reference set. The proper way to determine identity is with a logical COINCIDENCE operation. For instance, to find if a single bit *A* is identical to a reference bit *B*, we use

For a pattern of *n* bits, we need the logical AND of each such term:

$$A.EQ.B = A \odot B \qquad (3\text{–}2)$$

$$A.EQ.B = (A0 \odot B0)\bullet(A1 \odot B1)\bullet\ .\ .\ .\ \bullet(An \odot Bn) \qquad (3\text{–}3)$$

We can make an important distinction based on whether the reference set of bits is an unvarying (constant) or a varying pattern. Expanding the single-bit Eq. (3-2) into its AND, OR, NOT form, we have

$$A.EQ.B = A \bullet B + \overline{A} \bullet \overline{B}$$

If *B* is constant, this equation can be simplified into one of two forms:

$$A.EQ.B = A \text{ if } B = \text{T}$$
$$A.EQ.B = \overline{A} \text{ if } B = \text{F}$$

Consider a comparison of an arbitrary 4-bit A with a fixed 4-bit **B = T, F, F, T**. Equation (3-3) can be reduced to

$$A.EQ.B = A0 \bullet \overline{A1} \bullet \overline{A2} \bullet A3$$

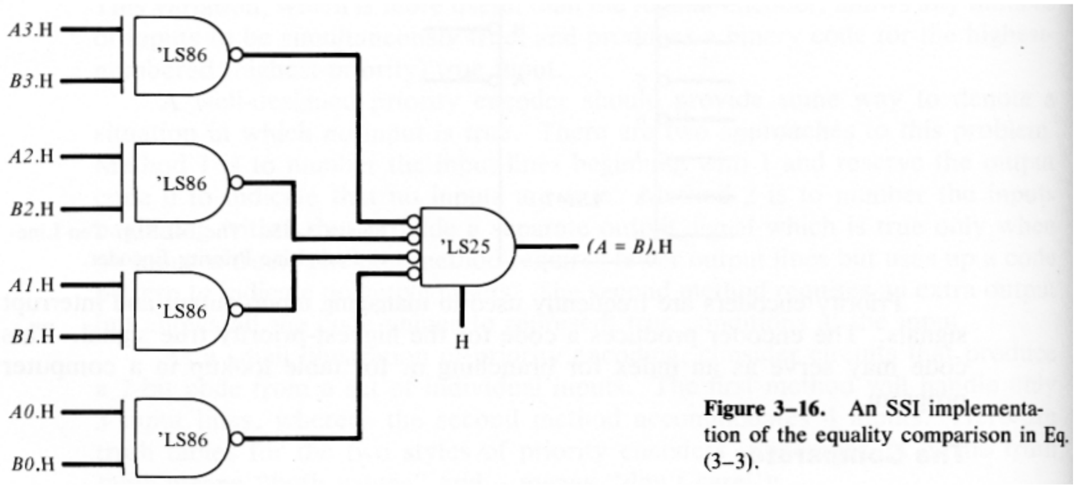


Figure 3–16. An SSI implementation of the equality comparison in Eq. (3–3).

which can be realized with a 4-input AND element.

If the reference pattern is not fixed, we are stuck with Eq. (3-3). Figure 3-16 is a circuit for 4-bit inputs, using common SSI chips. This type of circuit is a candidate for a building block, and there are MSI chips that perform multibit arithmetic comparisons.

We frequently treat arrays of bits as representations of positive binary numbers. Sometimes we need to determine if one such representation **A** is arithmetically greater than, equal to, or less than another reference pattern **B**. In addition to chips that perform only the equality comparison, there are also several arithmetic magnitude comparators, which have outputs capable of simultaneously showing the values of the conditions **A** < **B**, **A** = **B**, and **A** > **B**. A common MSI chip of this type is the 74LS85 Four-Bit Magnitude Comparator. This chip has three status inputs that permit several chips to be cascaded so as to yield comparisons of multiples of 4 bits. Figure 3-17 contains the arrangement

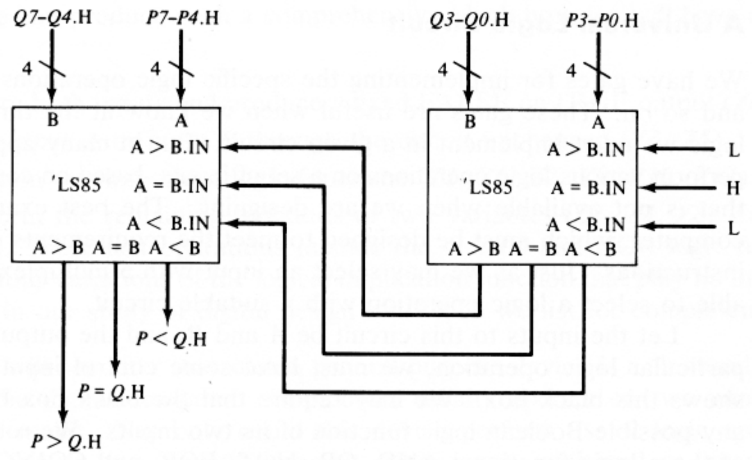


Figure 3–17. An 8-bit magnitude comparison using the 74LS85.

of 74LS85 comparators to accomplish a comparison of 8-bit quantities **P** and **Q**. The status outputs of a comparator stage connect to the corresponding status inputs of the next most significant comparator. The results of the final comparison are available as outputs of the most significant stage. It is necessary to pass to the least significant chip the information that the previous (nonexistent) comparison showed equality; this is done by asserting **T** (a high voltage) on the **A=B.IN** pin, and **F** on the other two.

There are several diagrammatic conventions in Fig. 3-17. To avoid cluttering functional diagrams, we often show a group of similar signal lines as a single line with *a numbered* slash across it. The notation for inputs **Q7**.H, **Q6**.H, **Q5**.H, and **Q4**.H appears as **Q7-Q4**.*H* with a "*" mark on the wire. Figure 3-17 also contains three other similarly collected groups of inputs. This use of the numbered slash is a widely accepted convention; the *numbered* slash has nothing to do with inversion.

Figure 3-17 forced us to make a choice. We usually desire that signals move toward the right in a circuit diagram, with an output on the left feeding inputs to the right. We also usually wish for the least significant part of numerical information to be on the right and the more significant parts on the left. In this figure, variables **Q7-Q0** and **P7-P0** represent numbers. We cannot easily accommodate both these goals in the diagram without creating a nightmare of lines, so we usually choose to show the numbers in their customary order rather than adhere to the convention of rightward-moving signals. In practice you will encounter both conventions.

Since Fig. 3-17 is the first instance of this drafting custom in the book, we have placed arrows on all the signal lines to show the direction of the signals' travel. Although we frequently use arrows in high-level functional diagrams, in most instances we would not use such arrows on an actual circuit diagram; the chip's nomenclature shows which pins are inputs and which are outputs.

A Universal Logic Circuit

We have gates for implementing the specific logic operations AND, OR, EOR, and so on. These gates are useful when we know at the time we design what logic we must implement in a given circuit. But in many applications we must perform various logic operations on a set of inputs, based on command information that is not available when we are designing. The best example is the digital computer, which must be designed to meet the requirements of any of its set of instructions. Just as we may select an input with a multiplexer, so must we be able to select a logic operation with a suitable circuit.

Let the inputs to this circuit be **A** and **B**, and the output Z. To select the particular logic operation, we must have some control inputs S_i. Figure 3-18 shows this black box. We may require that the black box be able to perform any possible Boolean logic function of its two inputs. We routinely use several of these logic functions: AND, OR, NOT, EOR, and COINCIDENCE. As we mentioned in Chapter 2, there are 16 functions of two variables. They are enumerated in Table 3-1. Some are already familiar:

$$Z1 = A \cdot B \qquad Z7 = A + B \qquad Z10 = \overline{B}$$
$$Z6 = A \oplus B \qquad Z9 = A \odot B \qquad Z12 = \overline{A}$$

TABLE 3–1 LOGIC FUNCTIONS OF TWO VARIABLES

A	B	Z 0	Z 1	Z 2	Z 3	Z 4	Z 5	Z 6	Z 7	Z 8	Z 9	Z 10	Z 11	Z 12	Z 13	Z 14	Z 15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

There are some others in the table that at first sight appear to be uninteresting but are in fact useful:

$$Z0 = \mathrm{F} \qquad Z3 = A \qquad Z5 = B \qquad Z15 = \mathrm{T}$$

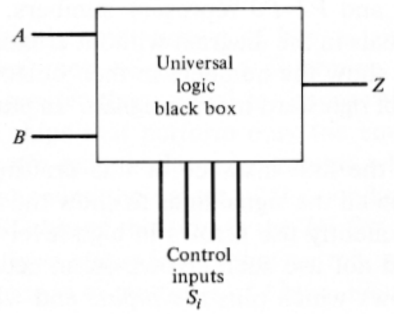


Figure 3–18. Inputs and output for the universal logic circuit.

If we can produce such a comprehensive black box, we will have a circuit that can:

(a) Ignore both inputs and produce a fixed FALSE or TRUE output (**Z0, Z15**).

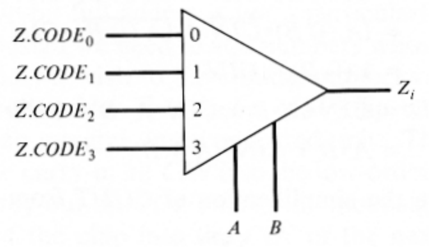
(b) Pass input A or input *B* through the circuit unchanged **(Z3, Z5)**.

(c) Perform our important logic functions **(Z1, Z6, Z7, Z9, Z10, Z12)**.

(d) Perform the remaining functions of two variables **(Z2, Z4, Z8, Z11, Z13, Z14).** These last operations include the NAND and NOR logic functions and four variations of the logical implication function that play no important role in our study of digital design but which we list for completeness.

You will see later that such a general-purpose device is a "natural" at the heart of the digital computer. Computers usually operate on two numbers to produce a result. Not only could this device perform useful logic operations upon two inputs, but it could transmit either input, unaltered or inverted. In addition, it could be a source of **T** and **F** bit values.

Many designs for producing the 16 Boolean functions are known, but from our viewpoint the most elegant is a single 4-input multiplexer. To produce a function, our circuit must receive a 4-bit code specifying the particular function **Zi**. The obvious code values are 0 to 15, corresponding to **Z0** through **Z15**. Call the code **Z.CODE**, with the code bits designated **Z.CODE$_0$** through **Z.CODE$_3$.** Notice, in Table 3-1, how the definition of each function **Zi** is exactly the binary representation of the corresponding **Z.CODE**. In an unusual interpretation of the multiplexer in its table-lookup role, we may use the "data" variables **A** and **B** as the select inputs of the mux, and feed the function-specifying code **Z.CODE** into the data inputs:



Thus we may produce all 16 Boolean functions of two variables with one-half of a 74LS153 Multiplexer chip. This is tight design!

The universal logic circuit is elegant, but it is capable of performing logic operations only. If it is to be used as the heart of a computer, it should also be able to perform arithmetic operations. Let us leave our universal logic circuit for a moment and discuss the structure of circuits that can perform arithmetic on binary numbers. Later we will consider circuits that can perform both logic and arithmetic.

Binary Addition

The full adder. We assume that you are familiar with the process of binary addition and the representation of numbers in the two's-complement notation. For each bit position, the truth tables defining the addition process are given as Table 3-2. **A** and **B** are the bits to be added, **CIN** is the carry bit generated by the previous bit position, **SUM** is the sum bit for the current bit position, and **COUT** is the carry generated in the current bit position. A device for summing three bits in this manner is called *a full adder.* (A similar circuit without the **CIN** input is called *a half adder.)*

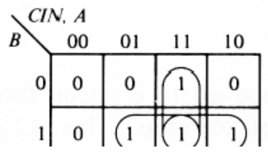
TABLE 3–2 TRUTH TABLE FOR BINARY ADDITION

CIN	*A*	*B*	*SUM*	*COUT*
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The full-adder truth table yields Boolean equations for the sum and carry bits:

$$\begin{aligned}
SUM &= \overline{CIN}\cdot\overline{A}\cdot B + \overline{CIN}\cdot A\cdot\overline{B} + CIN\cdot\overline{A}\cdot\overline{B} + CIN\cdot A\cdot B \\
&= (A \oplus B)\cdot\overline{CIN} + (A \odot B)\cdot CIN \\
&= (A \oplus B)\cdot\overline{CIN} + (\overline{A \oplus B})\cdot CIN \\
&= A \oplus B \oplus CIN \\
COUT &= \overline{CIN}\cdot A\cdot B + CIN\cdot\overline{A}\cdot B + CIN\cdot A\cdot\overline{B} + CIN\cdot A\cdot B \\
&= A\cdot B + CIN\cdot(A + B)
\end{aligned}$$

You may derive the simplification of **COUT** from the K-map:



To perform addition on arrays of bits representing unsigned binary numbers, we may connect full adders together as in Fig. 3-19. As a concrete example, let's add two 3-bit binary numbers **A** and **B**, where **A** = 101 and **B** = 110. The result of the binary addition is 1011. The corresponding values that would be present on the wires of the hardware are shown in Fig. 3-20.

This method of connecting full adders is called the *ripple carry* configuration,

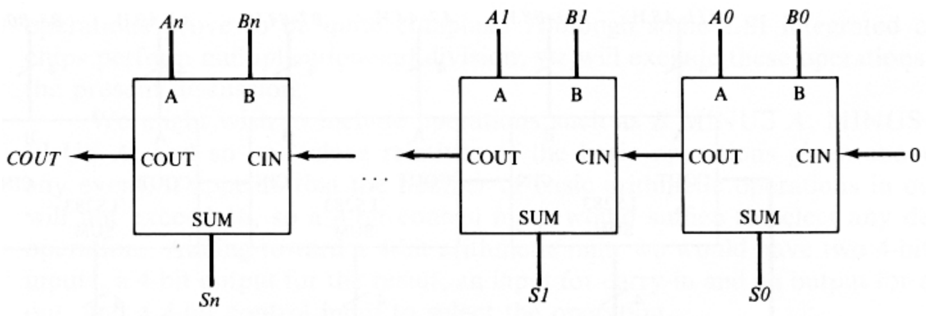


Figure 3–19. Addition with cascaded full adders.

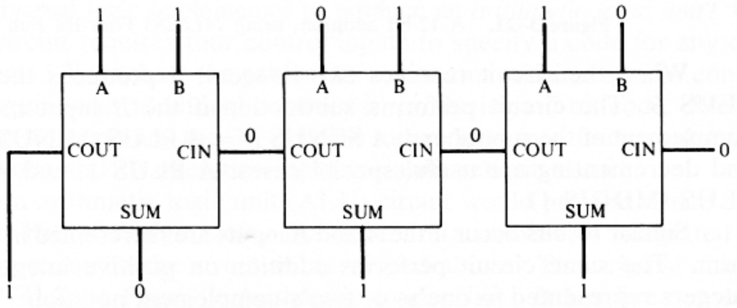


Figure 3–20. 101 + 110 = 1011, using full adders.

since stage zero must produce output before stage 1 can become stable. After stage one becomes stable, stage two will begin to develop its stable outputs. In other words, the carry does indeed ripple down the chain of adders. This is the simplest but slowest way to perform binary addition. Presently we will look at ways of speeding up the process.

Cascading single-bit full adders is not a particularly useful way to perform addition. In digital design we need to add numbers whose binary representations span several bits, and we wish to have building blocks suited to this task. The 74LS283 Four-Bit Full Adder is a useful MSI chip constructed with four full adders packaged as an integral, interconnected unit. The device accepts two 4-bit inputs and a single carry-in bit **CIN** into the low-order bit position; it produces a 4-bit sum and a carry-out bit **COUT** from the most significant bit position. By feeding the **COUT** of the chip into the **CIN** of the next-most-significant stage, it is easy to produce binary adders for any reasonable word length. In the typical application, the **CIN** to bit 0 would be forced to be false, representing numeric 0. In Fig. 3-21, we show the data paths for a 12-bit full adder composed of three 4-bit full adders.

Signed arithmetic. The multibit full adder circuit of Fig. 3-21 does binary addition on 12-bit positive numbers. If the inputs **A** and **B** represent signed integers in the two's-complement notation, the circuit of Fig. 3-21 can perform signed arithmetic. In the two's-complement notation, the leftmost bit represents the sign of the number, and so the circuit shown in Fig. 3-21 can handle 11-bit integers plus a sign.

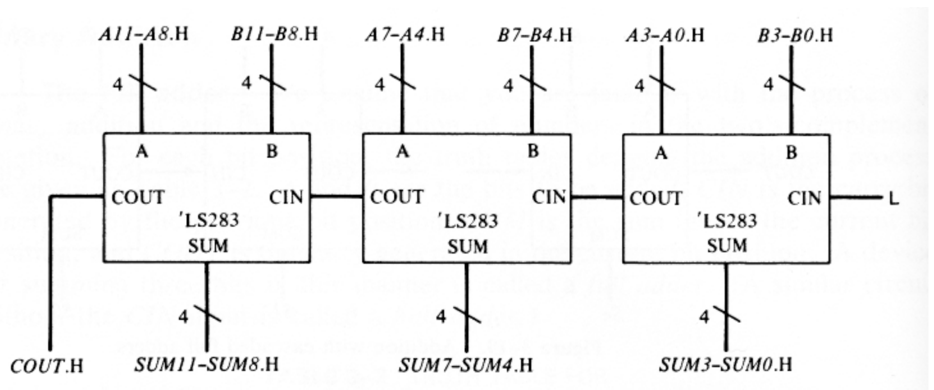


Figure 3–21. A 12-bit addition, using 74LS283 Four-Bit Full Adders.

When the circuit receives two integers, it produces the (signed) sum: **A** PLUS **B**. The circuit performs subtraction if the *B* input receives the two'scomplement of the subtrahend: **A** MINUS **B** = **A** PLUS (MINUS **B***)*. Incrementing and decrementing are useful special cases: **A** PLUS **1**, and **A** MINUS **1** = **A** PLUS (MINUS **1**).

Similar results occur if the **A** and **B** inputs are represented in one's-complement form. The same circuit performs addition on positive integers and on signed integers represented in one's- or two's-complement notation. The hardware did not change-only the interpretation of the data.

The Arithmetic Logic Unit

This building block, as its name implies, combines the logic capability of the universal logic circuit with a general set of binary arithmetic operations built around cascaded full adders. With the multibit full adder, we may produce arithmetic operations such as subtraction and incrementing only by manipulating the input data; the adder circuit itself only adds.

If we are to have a general arithmetic capability as a building block, such special preparation of the input data is inappropriate; the arithmetic unit should allow us to select an operation. This is a similar situation to that which led us to the universal logic circuit for the performance of arbitrary logic operations on its inputs.

For operands **A** and **B**, each consisting of several bits, some useful types of binary arithmetic operations are

Addition:	*A* PLUS *B*
Subtraction:	*A* MINUS *B*
Incrementing:	*A* PLUS 1
Decrementing:	*A* MINUS 1
Negation:	MINUS *A*

To supplement operations of this type, we might wish to have a source of special constants, such as 0, plus 1, minus 1, and so on. It would be nice to include multiplication and division in our list, but these operations prove to be quite complex. Although some LSI integrated circuit chips perform multiplication and division, we will exclude these operations from the present discussion.

We might wish to include operations such as **B** MINUS **A**, MINUS **B**, **A** PLUS **A**, and so on–close relatives of the basic operations given above. In any event, it appears that the number of basic arithmetic operations in our list will not exceed 16, so a 4-bit control input would suffice to select any desired operation. Aiming toward a 4-bit arithmetic unit, we would have two 4-bit data inputs, a 4-bit output for the result, an input for carry-in and an output for carryout, and a 4-bit control input to select the operation.

Can we combine these arithmetic operations with the logic capability of the universal logic implementer to produce an *arithmetic logic unit?* Our universal logic circuit requires four control inputs to specify a code for any of its 16 logic functions. To include the arithmetic operations within a similar control structure will require one additional control bit, producing a 5-bit code. We may use this fifth bit to separate the 32 possible operations into two groups–the 16 logic operations and 16 operations that are arithmetic in nature.

An arithmetic logic unit (ALU) circuit would provide a nice building block for our bag of design tools, and there are a number of integrated circuit chips that approximate the structure developed above. The original chip of this type is the 74LS181 Four-Bit Arithmetic Logic Unit. It has five control inputs and provides the 16 logic functions of two variables, operating simultaneously on each pair of bits. It also provides 16 other operations that have an arithmetic character. The 74LS181 does not have all the arithmetic operations in our wish list, but it is capable of adding, subtracting, incrementing, and decrementing numbers in the two's-complement representation. Several of the "arithmetic" operations of the 74LS181 are somewhat bizarre, and are of little interest to us, but do no harm.

The 74LS181 allows the cascading of 4-bit chips to provide arithmetic capability for long words. The arrangement of the data paths is the same as in Fig. 3-21 for the full adder.

Most of the building blocks described in this chapter are tools for performing a single useful function, such as decoding or multiplexing. The ALU is a step toward more powerful LSI building blocks that perform a variety of functions on a small number of bits, under the control of a set of input signals that we may view as an operation code. Such building blocks are frequently called *bit slices*, alluding to their ability to be ganged together to process larger numbers of bits. Most bit-slice devices have internal registers and are designed to support the basic operations required in modern computer processors. You will encounter additional bit-slice components in Chapter 4 and later in this book.

Speeding Up the Addition

Bit-slice circuits such as the 74LS283 Four-Bit Adder and the 74LS181 Arithmetic Logic Unit are helpful digital building blocks, but if they depended on the simple ripple-carry scheme for binary addition they would be very slow. Binary addition as a combinational process. You know that, at least in theory, any combinational process can be expressed as a truth table and implemented as a two-level sum-of-products function. This approach has only limited practical value in binary arithmetic, since the truth tables for a multibit sum become too large to manage. For instance, the truth table for a 12-bit sum has 24 input variables (25 if we allow for separately specifying the initial carry-in to bit position 0).

Truth tables are a way of specifying in detail the outputs for each combination of input values. In nonarithmetic work we can usually find a sample repetitive pattern of one or two bits that serves as a model for the behavior of the entire circuit, and we can express the repeating function as a small truth table or as an equation. This works well in logic operations, since for each bit the result of an operation depends only on the data entered for that bit, and not on the data an adjacent or more distant bits. Unfortunately, arithmetic does not have this sample property, because of the complex way in which the carry bits affect the result. So two-level binary addition, although desirable because of its speed, is intractable when there are more than a few bits. Within a small bit-slice, however, it is sometimes feasible to produce two-level addition-for instance, four-bit data inputs yield five 9-input truth tables, for the 4 bits of the sum and the single carry-out bit. Each of these truth tables has $2^9 = 512$ rows–painful but not impossible to produce if the rewards are great enough. But you can see that this as hardly a promising general approach.

Ripple-carry is a serial method, slow and simple; two-level circuits are fully parallel, fast but difficult. We need an intermediate technique that provides *some* parallelism with a reasonable effort. A widely used approach as to cast the problem of addition into terms of *carry generate* and *carry propagate* functions. For the moment, consider a one-bit full adder, with inputs $\mathbf{A_i}$, $\mathbf{B_i}$, and $\mathbf{C_i}$, and outputs $\mathbf{S_i}$ and $\mathbf{C_{i+1}}$. We will focus on some properties of the data inputs $\mathbf{A_i}$ and $\mathbf{B_i}$. We introduce a carry-generate function $\mathbf{G_i}$ that is true only when we can guarantee that the data inputs will generate a carry-out. We introduce a carry-propagate function $\mathbf{P_i}$ that as true only when a carry-in will be propagated as an identical carry-out. For a 1-bit sum, the truth tables for $\mathbf{G_i}$ and $\mathbf{P_i}$ are

A_i	B_i	G_i	P_i
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Using these functions, we may express the carry-out and sum:

$$C_{i+1} = G_i + P_i \bullet C_i \qquad (3\text{–}4)$$

$$S_i = P_i \oplus C_i \qquad (3\text{–}5)$$

(To verify the equation for $\mathbf{S_i}$, you may wish to refer to Table 3-2, our original definition of the full adder.) These equations express the sum and carry-out an terms of just the generate and propagate operators and the carry-in-an important property that we will use when we extend these concepts to bit-slice adders. With these equations, we may implement multibit full adders, but the ripple-carry effect is still present, since each bit's carry-in depends on the preceding bit's carry-out. However, we may expand the equation for C, in terms of the equations for less-significant bits, to achieve a degree of *carry look-ahead.* For instance

$$C_1 = G_0 + P_0 \bullet C_0 \quad (3\text{–}6)$$

$$C_2 = G_1 + P_1 \bullet (G_0 + P_0 \bullet C_0) \quad (3\text{–}7)$$

We can derive similar equations (of increasing complexity) for C_3 and C_4. Again, each equation involves only the generate and propagate operators and the original carry-in.

When the generate and propagate operators apply to one-bit slices

$$G_i = A_i \bullet B_i \quad (3\text{–}8)$$

$$P_i = A_i \oplus B_i \quad (3\text{–}9)$$

These equations involve only the arithmetic data inputs for the ith one-bit unit. By substituting and simplifying, we may derive the following equations for the carry bits:

$$C_1 = A_0 \bullet B_0 + A_0 \bullet C_0 + B_0 \bullet C_0$$

$$C_2 = \mathrm{A}_1 \bullet \mathrm{B}_1 + A_1 \bullet A_0 \bullet B_0 + A_1 \bullet A_0 \bullet C_0 + A_1 \bullet B_0 \bullet C_0 + B_1 \bullet A_0 \bullet B_0 + B_1 \bullet A_0 \bullet C_0 + B_1 \bullet B_0 \bullet C_0$$

These equations yield two-level results. The expansion of C_3 and C_4 are more lengthy, and will be left as homework problems. Within a bit-slice chip, for instance the 74LS181, these equations could be used for high-speed implementations of the sum outputs $\mathbf{S_0}$ through $\mathbf{S_3}$ and the carry-out bit $\mathbf{C_4}$.

If we are building a large adder or ALU from smaller bit-slices, we are still faced with the ripple-carry problem across the boundaries of each chip, even though within each chip the carry-out is being computed rapidly. For instance, in Fig. 3-21, the most-significant carry-out cannot be computed until its corresponding carry-in (into bit 8) is stable, which in turn must await the stabilization of the carry-in to bit 4. We have carry look-ahead within the chips, but not among the chips.

The concept of carry generate and propagate, as typified by Eqs. (3-6) and (3-7), may be extended to larger bit slices. The 74LS181, in addition to producing the carry-out $\mathbf{C_4}$, also produces two outputs **G** and **P** that are equivalent to our one-bit $\mathbf{G_3}$ and $\mathbf{P_3}$. The 74LS181's **G** output is true whenever the data inputs assure that its carry-out is true; the **P** output is true only when the data inputs are such that the block carry-out is the same as the block carry-in. These **G** and **P** functions are considerably more complex than our one-bit Eqs. (3-8) and (3-9), but still involve only the arithmetic data inputs for the chip.

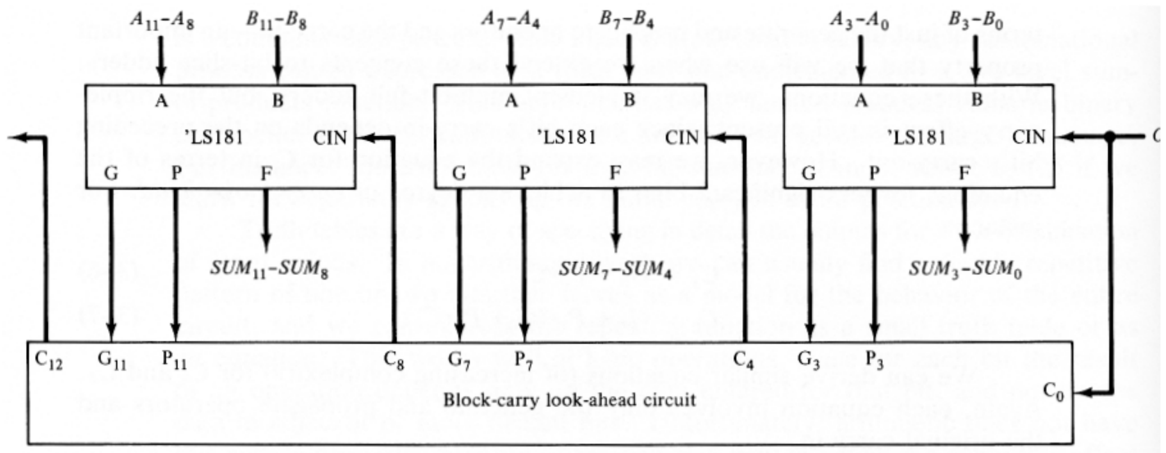


Figure 3–22. A 12-bit ALU containing a block-carry look-ahead circuit.

Figure 3-22 is the circuit for a 12-bit adder constructed with 74LS181 ALUs and a block-carry look-ahead circuit. Instead of relying on each 74LS181 to send its computed carry-out on to the next most significant 74LS181, we send the **G** and **P** outputs to the block-carry look-ahead box. The look-ahead box is a combinational circuit that accepts all the **G's** and **P's** and the initial carry-in, and simultaneously computes all the carry-outs that must be sent to the 74LS181s. The more significant 74LS181 chips do not have to wait for their carry-in signals to ripple in from lower stages.

We may build the look-ahead box from equations such as the following, which can be inferred from the intuitive meaning of the generate and propagate operators in Fig. 3-22.

$$C_4 = G_3 + P_3 \bullet C_0$$
$$C_8 = G_7 + P_7 \bullet G_3 + P_7 \bullet P_3 \bullet C_0$$

Integrated circuits that perform the block-carry look-ahead functions are available. The 74LS182 Look-Ahead Carry Generator supports the look-ahead process in up to four 74LS181 chips. Furthermore, the 74LS182 produces its own version of **G** and **P**, so that look-ahead circuits of more than 16 bits may be constructed by adding levels of 74LS182s. Two levels of 74LS182 will support 64-bit addition. Each new level of look-ahead circuitry increases the time required for the adder outputs to stabilize, but only by about 10 nanoseconds. Since the 74LS181 requires about 20 nanoseconds to perform a binary addition and produce its generate and propagate outputs, the 12-bit adder in Fig. 3-22 requires about 30 nanoseconds and a 64-bit adder would require only about 40 nanoseconds. On the other hand, the 74LS181 requires about 25 nanoseconds to compute its carryout. Were the 74LS181s used in a ripple-carry configuration, as in Fig. 3-21, each stage would require 25 nanoseconds. The 12-bit addition would require 50 nanoseconds, with longer words requiring an additional 25 nanoseconds for each additional 4 bits.

Arithmetic is a vital function in most computer applications, and much effort has gone into producing fast and efficient arithmetic circuits. Multiplication and division present their own sets of difficulties; fast division is a particularly challenging problem. We will not cover these specialized areas; consult either the textbooks listed at the end of the chapter or the technical literature on specific computers.

DATA MOVEMENT

One of the most important operations in digital design is moving data between a source and a destination. This often occurs inside computers, and is a major activity of peripheral devices. Frequently, the data itself involves several bits–an n-bit byte or word–that must move through the system in parallel. Typical data paths in modern digital design are 8 to 64 bits wide.

If there is only a single source of data and a single destination, we have no problem. We simply run n wires from the source's output to the destination's input. With several sources and various destinations, the situation becomes more complex and requires that we allow data to be moved from any source to any destination. One alternative is separate data paths from each of **S** sources to each of **D** destinations. An item that is a source of data at one time may be a destination of data at other times. In Fig. 3-23, we show the data paths in two configurations: three sources **S1-S3** with three different destinations **D1-D3**, and four common sources and destinations. **A. B. C. D.**

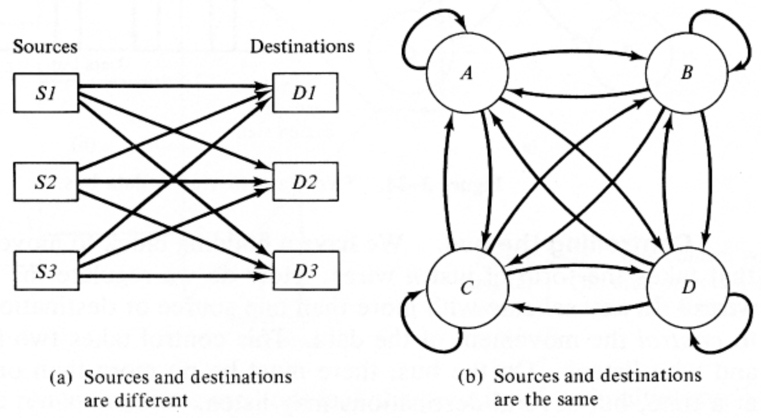

Sources
Destinations
S1
S2
S3
D1
D2
D3
(a) Sources and destinations are different
A
B
C
D
(b) Sources and destinations are the same

Figure 3–23. Moving data by complete connection of sources and destinations. All paths are *n* bits wide.

There are obvious drawbacks to this scheme. The number of wires becomes very large as the number of sources and destinations increases, in general being S x D x N. Adding new nodes to the system involves massive rewiring, affecting each source and destination. A completely connected system allows several simultaneous data transactions to occur over the independent data paths, and this is the main advantage of the scheme. When high-speed parallel movements of data are vital, we would expect to pay the price of inflexibility and complex wiring, and would adopt some form of the completely connected system.

The Bus

In most applications, especially when the number of sources and destinations is not fixed at design time, we need a more flexible solution to the problem of moving data. By giving up parallelism, we may achieve this flexibility. Suppose we run all sources into a single node and take all destination paths from this node. We call this configuration *a data bus,* or just *a bus.* Figure 3-24 consists of two ways to represent a bus, Fig. 3-24b being the more common way. (Remember that all the data paths are actually *n* bits wide, although we usually only draw a single path representing the entire word or byte.) The bus structure permits only one data transfer to occur at a time, since all data paths funnel through the bus node. However, adding or deleting elements on the bus is simple, and this overwhelming advantage accounts for the widespread use of this configuration in digital computers.

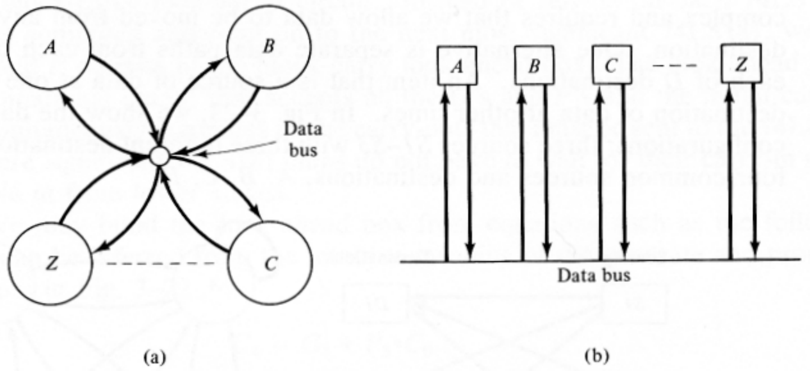


Figure 3–24. Two ways to view a data bus.

Controlling the bus. We have a building block to move data–the bus–that takes the form of just *n* wires. How do we regulate the traffic over these wires? In any scheme with more than one source or destination, there is a need to *control* the movement of the data. This control takes two forms: who talks, and who listens. On the bus, there must be no more than one talker (source) at a time, but several destinations may listen.

The responsibility for listening on the bus (receiving data) is part of each destination device and is not directly a part of the bus operation. All destinations are physically capable of listening; whether they actually accept data is under their control. Maintaining control over the bus sources, to assure only one talker at a time, is very much a concern of the designer of the bus. We shall mention four control mechanisms, two of which you have already encountered.

Bus access with the multiplexer. Our job is to select one source from several candidates. The digital designer, when encountering the concept of selection, has a knee-jerk response–the multiplexer. For each bit of the bus's data path, attach a multiplexer output to the bus, making each source an input to the mux. We control this collection of *n* multiplexers with a common source-select code feeding into the multiplexers' select inputs. We show the idea in Fig. 3-25. In this approach, we collect the control for access to the bus in one spot, and assure that only one source is talking at a time–both important advantages. Further, it is easy to debug, since we maintain explicit centralized control over which source has access to the bus. On the other hand, the data mux method of bussing requires considerable hardware; we use an **S**-wide mux for each of the *n* data bits in the bus path. If *n* is large, we have a boardful of data multiplexers. Adding new sources is convenient as long as we do not exhaust the input capability of our muxes. If we exceed this capacity, we have a difficult hardware-modification job. For instance, with 8-input multiplexers, we may manage up to eight sources, but the ninth source causes great agony. Thus, the data multiplexer method of bussing suffers from a certain inflexibility and is not very conserving of hardware. Nevertheless, it is a good method of bussing a moderate number of sources and we will use it in the design of a minicomputer in Chapters 7 and 8.

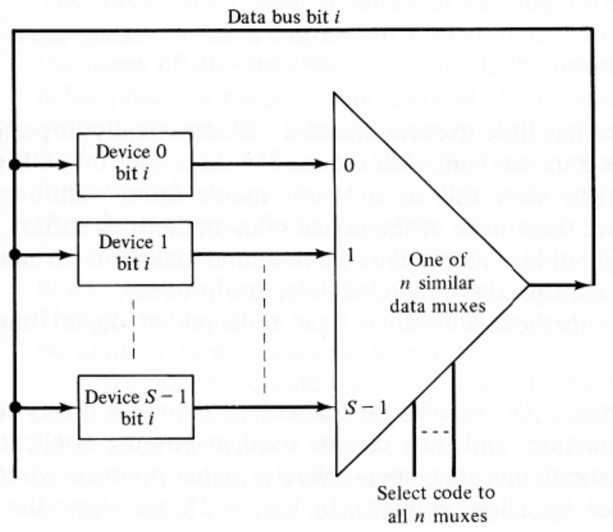


Figure 3–25. Bus control with multiplexers. The mux selects a single source, and the bus routes it to all receivers.

The remaining three methods lack the security of the multiplexer's encoded selection control.

Bus access with OR gates. A primitive form of bus control is to merge all sources into the bus data path, using OR gates. For S n-bit sources, we would have n OR gates, each accepting S inputs. This produces the merging required to give all sources access to the single bus path, but it does not provide the control needed to allow only one source onto the bus at a time. Each bit of each source is either **T** or **F**; we must arrange for all sources but one to have all their bits false, while the one designated source presents its **T** or **F** data through the OR gates onto the bus. This approach places the responsibility for access with each source, rather than directly with the bus as in the multiplexer method. Each source must have its own gating signal to open or close the gate on its data bits. Typically, the sources have some form of AND gate on each data bit: the data forms one input and the control signal forms the other. (It is this usage that gave rise to the "gate" terminology in digital circuits.) The method is shown in Fig. 3-26.

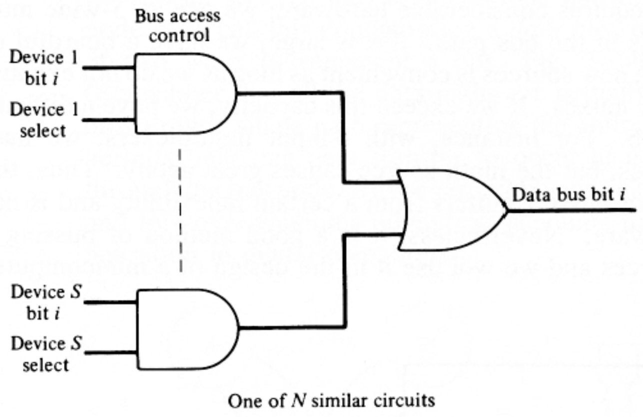


Figure 3–26. Controlling access to the data bus with OR gates.

The OR-gate method has little to recommend it. Electronically, it performs the same functions as the mux method, with the mux circuits split into OR gates and AND gates. We might view this as a "poor man's mux," although its components will cost more than those of the actual mux method. It suffers from the same inflexibility of input-size as the mux method and lacks the certainty of control provided by the multiplexer's encoded selection process.

The remaining two methods introduce new concepts of digital building blocks.

Open-collector gates. Open-collector technology provides a way to implement the OR logic function, and thus can be used in bussing applications. We must adhere to the stipulation that open-collector gates produce wired-OR when truth is represented by a low voltage. In Fig. 3-27, we show the 7407 Open-Collector Buffer used as a bus driver.

Since their primary use was for bussing, where several destinations may listen in on the bus, open-collector chips usually can carry more current than their normal TTL counterparts. This provision of extra power is called *buffering,* and such chips are called *buffers* or *drivers.*

The advantage of open-collector circuits is the elimination of the wide OR gates. As long as only one source at a time is on the bus (at most one input is asserting truth), we may connect a large number of open-collector outputs together. Aside from this, achieving proper control of the bus with open-collector wired-OR logic involves the same concerns as the ordinary OR-gate method: we must still control each of the sources so that at most one is talking at a time.

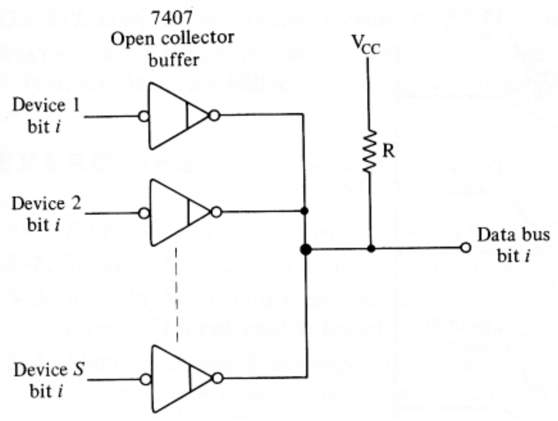


Figure 3–27. Controlling access to the data bus with open-collector buffers.

Three-state outputs. The *three-state output* has, as its name implies, three stable states instead of the customary two. In addition to the usual high and low voltage levels, the third state provides *a high-impedance* mode, usually called Z, in which the output appears as if it were disconnected from its destinations. The three-state output requires an enabling three-state control input. When the output is *enabled,* the circuit transmits the normal H or L signal presented at the input of the three-state circuit. If the output is *disabled,* the circuit output is for practical purposes not there at all. (Logicians should note that three-state outputs are not the same as ternary logic, which is a true base-3 system.)

Many SSI, MSI, and LSI chips incorporate three-state data outputs. The fundamental use is in bussing, so three-state outputs often provide power buffering like their open-collector cousins. We might select the 74LS244 Octal Three-State Buffer as our prototype three-state chip. This is one of several SSI building blocks that perform no logic but simply afford three-state control of their inputs. This particular chip has two three-state sections-a two-buffer unit controlled by another three-state-enable input.

We find three-state outputs in many useful building blocks. The multiplexers discussed earlier in this chapter have an enable input that holds their output false when the chip is disabled. In the three-state varieties, the output is "disconnected" when the chip is disabled. In Chapter 4, you will see more examples of three-state outputs in MSI building blocks.

The uses of three-state output control in data bussing are substantial. We may connect almost any number of three-state devices together and, with proper three-state enabling of only one source at a time, control access to the bus. Often the chips providing the bus's source data will have three-state output control built in; in other cases, we may need to add buffers such as the 74LS244 to achieve three-state control. Figure 3-28 is a typical three-state bussing configuration.

There are two drawbacks to three-state bus control. First, as in the last two bussing methods, the control of access is decentralized, residing with the sources rather than with the bus structure itself. This makes more difficult the task of assuring that only one source at a time is talking on the bus. Second,

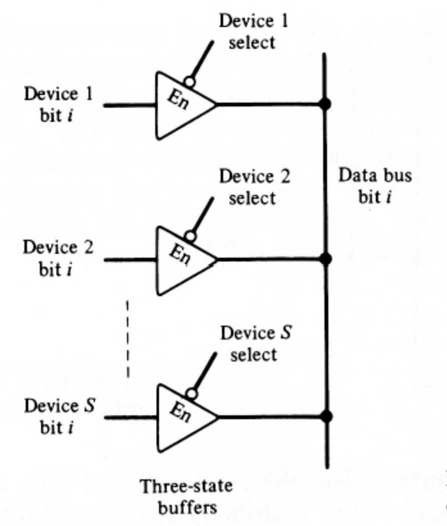


Figure 3–28. Controlling access to the data bus with three-state buffers.

the three-state bus is more difficult to debug than the bus formed from multiplexers. The three-state bus itself is just a collection of *n* wires. A debugger who sees bad data on the bus cannot easily identify which source or sources are contributing. Failure of the three-state circuitry often requires tedious disconnecting of sources from the bus until the guilty party identifies itself by a change in the bus's behavior. In the multiplexer method, we may check inputs, controls, and output directly, and quickly determine which element is misbehaving.

These drawbacks to three-state bus control are insufficient to counteract the tremendous advantages that this technology offers, and three-state control is used in most modern applications of data bussing.

One caveat: Do not try to use three-state control at the output of a control signal. Control signals must be either true or false (asserted or negated) at all times, and we cannot afford to have them simply not there. Only with data whose use is *governed* by control signals do we have the opportunity to have certain data sources disconnected some of the time.

READINGS AND SOURCES

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

FAST: Fairchild Advanced Schottky TTL. Fairchild Camera and Instrument Corp., Digital Products Division, South Portland, Maine.

FLETCHER, WILLIAM *I., An Engineering Approach to Digital Design.* Prentice-Hall, Englewood Cliffs, N.J., 1980. Chapter 4 discusses MSI building blocks.

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

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

The TTL Data Book. Texas Instruments, P.O. Box 225012, Dallas, Texas 75265.

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

EXERCISES

3-1 What are SSI, MSI, LSI, and VLSI?

3-2 What distinguishes a combinational circuit from a sequential one?

3-3 Explain the structure and the function of the multiplexer. What are the two major types of output enable found in MSI multiplexers?

3-4 Figure 3-5, which corresponds to the commercial equivalent of the enabled multiplexer, is not a direct counterpart of Fig. 3-4 or Eq. (3-1). Give a logic equation that best describes the device shown in Fig. 3-5.

3-5 By consulting a TTL data book, make a list of the available MSI multiplexers and their distinguishing characteristics. You must inspect the data sheets to discern the details; chip names alone are not sufficient. (Since the multiplexer is such an important digital building block, this effort is well worthwhile.)

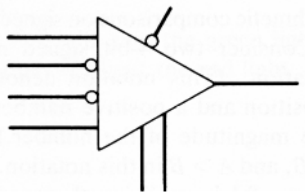
3-6 Why not have one select input for each multiplexer data input rather than encoding the select information?

3-7 The 74LS251 Eight-Input Multiplexer has a three-state output-enable feature. Construct a 16-input multiplexer building block from two 74LS251 chips and an inverter.

3-8 Build the 4-input multiplexer in Fig. 3-10, using SSI gates.

3-9 Show how to construct a 64-input multiplexer building block using eight 74LS251 chips and a 74LS42 decoder.

3-10 The 4-input multiplexer symbol below looks like a mixed-logic notation. Why do we not find this symbol useful?



3-11 The 74LS42 serves as a demultiplexer and a decoder. Characterize the difference in these two views of it.

3-12 Build the 4-output demultiplexer in Fig. 3-12, using SSI gates. Will your design also serve as a decoder? If so, how?

3-13 What is the most important characteristic of the outputs of a decoder?

3-14 Explain how we may view the 74LS42's decoding capability as either a 4-to-10 or an enabled 3-to-8 decoder.

3-15 The 74LS42 is sometimes called a BCD (binary-coded decimal) decoder. Why is this an appropriate name?

3-16 Construct a building block that will decode a 4-bit binary code into one of 16 outputs, using 74LS42 decoders and any necessary gates. 3-17. What is the purpose of an encoder? Why are practical encoders *priority* encoders?

3-18 Explain the difference between the concepts of encoding and decoding.

3-19 Using SSI gates, design a priority encoder that accepts five inputs and produces a 3-bit output code. Use method 1 of the text.

3-20 *Parity* is an important concept, frequently used in error-detection circuits within digital systems. The parity of a group of bits is odd if there are an odd number of 1-bits in the group; even parity implies an even number of 1-bits. Although rapid parity-computing circuits are available, the EXCLUSIVE-OR function provides the basis for parity computation.

(a) Show that the EXCLUSIVE OR of two bits computes odd parity.

(b) Show that, in general, $\mathbf{A}_1 \oplus \mathbf{A}_2 \oplus \mathbf{A}_3 \oplus \ldots \oplus \mathbf{A}_n$ expresses an odd-parity function of n-bits.

3-21 Multiplexers offer another interesting approach to parity computation. In the following, the output should be asserted if the parity is odd.

(a) Show how a 4-input multiplexer (for instance, one-half of a 74LS153) can be used to compute the parity of a 3-bit group.

(b) Write the logic equation (in terms of AND, OR, and NOT) for the actions of the multiplexer in part (a), and show that this equation is equivalent to the EXCLUSIVE-OR use suggested in the preceding exercise.

(c) Show how to use four 4-input multiplexers to compute the parity of a 9-bit group. How many 74LS153 chips would be required for this design?

3-22 The 74LS280 Parity Generator/Checker accepts 9 bits of data and reports the parity. Use this chip and any necessary SSI gates to design a circuit that will assert an output when the parity of a 10-bit group is odd.

3-23 Derive logic equations for determining if one 4-bit positive binary number is greater in magnitude than another. Design a circuit for this logic, using SSI gates.

3-24 The 74LS85 Magnitude Comparator has outputs for designating $\mathbf{A} < \mathbf{B}$, $\mathbf{A} = \mathbf{B}$, and $\mathbf{A} > \mathbf{B}$. How may we determine if $\mathbf{A} \leq, \mathbf{B}$? $\mathbf{A} \geq \mathbf{B}$**?** $\mathbf{A} \neq \mathbf{B}$**?**

3-25 Draw a circuit for 10-bit magnitude comparison, using 74LS85 chips.

3-26 Performing arithmetic comparisons on signed numbers is more complex than comparing magnitudes. Consider two 4-bit signed numbers **A** and **B**, recorded in signed. magnitude notation. (This notation denotes a negative number with a 1 in the leftmost bit position and a positive number with a 0 in that bit position; the other bits record the magnitude of the number.) Develop logic equations to determine if $\mathbf{A} < \mathbf{B}$, $\mathbf{A} = \mathbf{B}$, and $\mathbf{A} > \mathbf{B}$ in this notation. Explore whether the 74LS85 Magnitude Comparator is useful in realizing these equations. Produce a circuit (either with or without the 74LS85) for generating the three comparisons.

3-27 Design a 3-bit full adder equivalent to Fig. 3-19, using 1-bit full adders fabricated from SSI gates.

3-28 Modify Fig. 3-21 to perform the operation **A** (+) **B** (+) 1.

3-29 Using 74LS283 Four-Bit Full Adders and any necessary SSI gates, design a circuit that will accept a 12-bit signed number in the two's-complement representation and produce the negative of that number.

3-30 Devise a circuit that will accept a 12-bit signed number in the two's-complement representation and produce the absolute value of that number.

3-31 Verify Eq. (3-5) for the full-adder sum expressed in terms of the carry-generate and carry-propagate operators.

3-32 Derive equations for **C**$_3$ and **C**$_4$, similar to Eqs. (3-6) and (3-7), using the carry-generate and carry-propagate operators.

3-33 Derive the expressions for the 74LS181's **G** and **P** outputs. (Hint: consider the generate and propagate operators for the bits within the 4-bit slice; ask yourself what conditions must apply to obtain truth on the overall **G** and **P**.) Verify your results by consulting a 74LS181 data sheet.

3-34 Describe a data bus. Give the main advantages and disadvantages of this method of moving data. Why is the bus such a widely used concept? 3-35. Discuss the merits of controlling a bus with:
(a) Multiplexers.
(b) OR gates.
(c) Open-collector buffers.(d) Three-state buffers.

3-36 Three-state control of outputs is common. Why do we not employ three-state control of inputs?

3-37 Design bussing systems similar to Fig. 3-25 for six 4-bit devices, using:
(a) Open-collector bus drivers.
(b) Three-state bus drivers.

3-38 The multiplexer bus control method shown in Fig. 3-25 has the desirable property that only one source can be talking on the bus at any time. Devise a three-state bus control system that also has this "guaranteed single-talker" feature.

3-39 A logic probe is a small laboratory instrument that, when touched to a point in a digital circuit, indicates the digital voltage level at that point. A typical logic probe shows a low voltage level as a green light and a high voltage level as a red light. If the voltage level is outside the acceptable ranges, the probe shows no light or indicates invalidity in some other way. Read the sections in Chapter 12 on integrated circuit data sheets and performance parameters. Using this information, specify the following voltages for a logic probe designed to operate with the 74LS family of integrated circuits:
(a) The highest voltage that will light the green light.
(b) The lowest voltage that will light the red light.