3

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 they design only with boards, nails, and screws.

INTEGRATED CIRCUIT COMPLEXITY

In Chapter 2, you studied low-level building blocks—AND, OR, and NOT. Upon your first encounter with digital logic you will be largely concerned with this level of complexity and you must master it before proceeding. It is helpful if you use a logic simulator for digital drafting to produce and debug your circuit schematics; we highly recommend integrating simulators into your learning environment and list some in the references.

But, design at the Boolean equation level, while a necessary skill, is not the same as designing digital systems; now you will need higher level constructs, like (a)–(f). Unfortunately, there are no well accepted names for devices at this abstraction level; you will often see them referred to as MSI *(medium-scale integration)* devices, but this is a hold over from older technology where these were packaged as small integrated circuit chips. It is best to think of them in terms of their names, which, fortunately, do reveal their function. (if your background is primarily software, think of them as macro's) Most simulators will have libraries of commonly used mid-level abstractions and you must now become adept at designing at this level.

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 C, C++, and functional languages, 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 mid-level complexity, which 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.

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 switch contact, the electronic analog is easily constructed. We use Boolean variables instead of mechanical motion to select a given input.

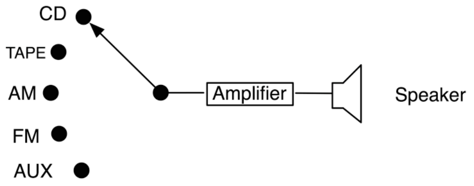
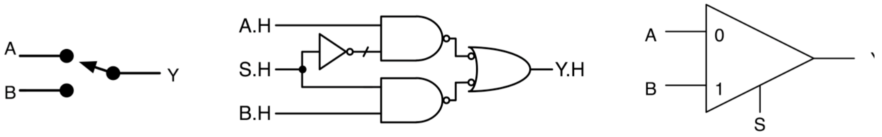


Figure 3–1. A mechanical selector switch

Consider a two-position switch with inputs A and B and output Y, such as shown in Figure 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

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

Using this equation, we can build an electronic analog of the switch; Fig. 3–2 is one design. There, S is the *select* input. The common name for this device is a 2-input MUX.



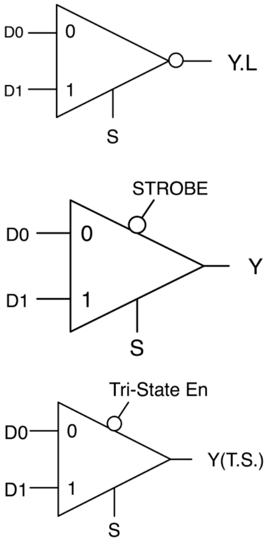
A 2-position mechanical switch

Implementation of the 2-input MUX

Symbol for 2-input MUX

Figure 3–2. different representations of a 2-input MUX

Most libraries will have a wide selection of more capable MUX's, usually including: 2,4,8,16-inputs with inverting, non-inverting, strobe enabled, and tri-state enabled, outputs—see Figure 3–3. Unfortunately, strobed and tri-state enabled MUX's often have identical symbols and you will have to experiment with a simulator to find out which is which—very inconsiderate! Further, the MUX symbol itself is not standardized; we will use the diamond shape since it explicitly displays the direction of data movement.



Inverting MUX

$$Y = D0 \bullet \overline{S} + D1 \bullet S$$

Strobed MUX

$$Y = (D0 \bullet \overline{S} + D1 \bullet S) \bullet STROBE$$

Tri-State MUX

$$Y(Tri - State) = (D0 \bullet \overline{S} + D1 \bullet S) \bullet En$$

Figure 3–3. Various MUX flavors

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 $2^1 = 2$ 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. Common symbols for a 4-input mux are shown in Figure 3–4.

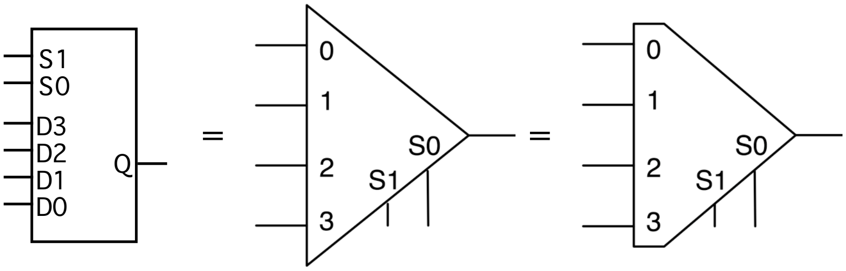
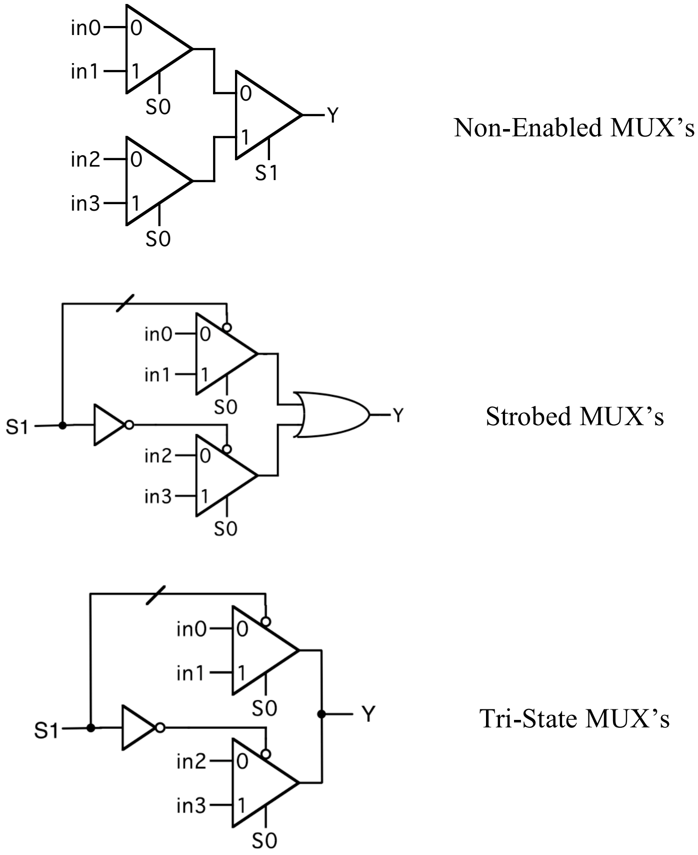


Figure 3–4. Drafting symbols for 4-input MUX's

Making big ones out of little ones

Static CMOS gates are limited to 3 or 4 inputs, which, in turn, limits the number of mux inputs. We need ways of combining small mux's to create wider devices. Figure 3–5 shows combining strategies for mux's with standard, strobed, and tri-state outputs



Non-Enabled MUX's

Strobed MUX's

Tri-State MUX's

Figure 3–5. Making 4-input MUXs from 2-input MUX's

We leave it as an exercise to justify the different merging elements, (mux, OR gate, or tri-state), used in these circuits. Of course you can apply similar techniques starting with wider mux's, as shown in Figure 3–6, to synthesize a 32-input mux from 8-wide tri-state mux's.

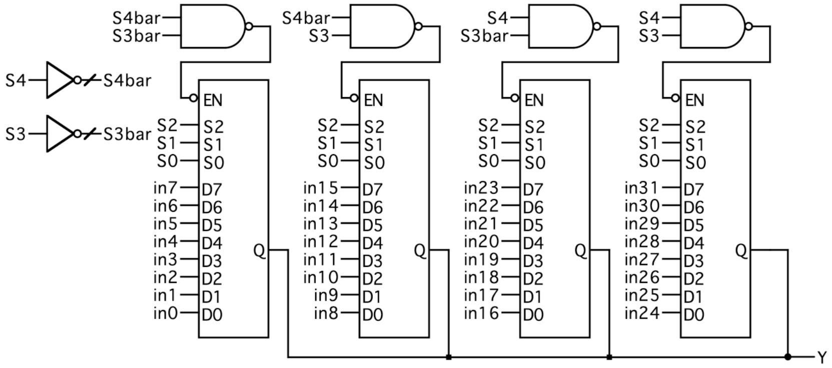


Figure 3–6. Synthesizing a 32 input mux from smaller tri-state mux's

The gates driving the tri-state enables have been carefully configured so only one enable is true at a time—a critical consideration in tri-state logic. Another common convention is to use terminals connected by symbolic wires. Thus, an implied wire connects all terminals labeled S0, this preserves connectivity while eliminating lots of clutter.

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 gate implementation. 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. 3–7 we show the equivalent mixed-logic forms for 4-input Multiplexers. 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.

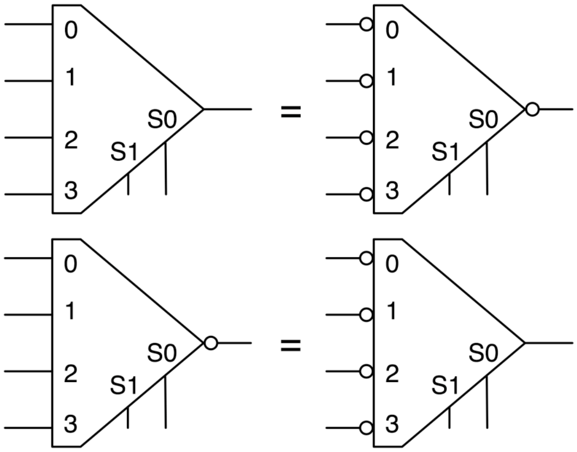
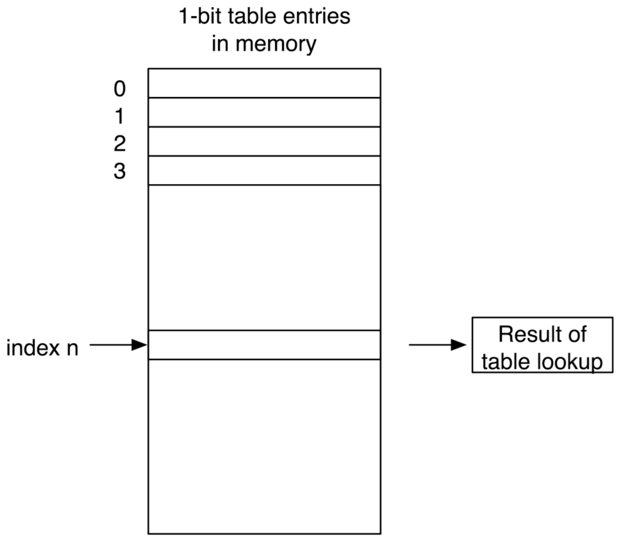
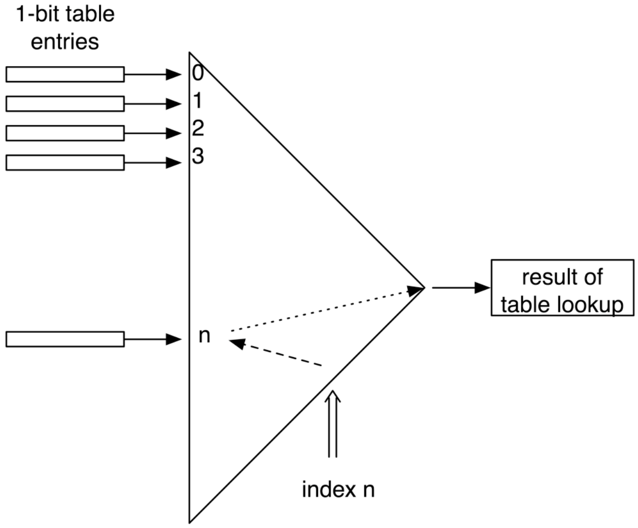


Figure 3–7. Multiplexer mixed-logic equivalent forms

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 that 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.



(a) Software table lookup



(b) Hardware table lookup using multiplexer

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 multi-bit lookup.

The Demultiplexer

A *de-multiplexer,* (dmux), sends data from a single source to one of several destinations and is the logical inverse of a multiplexer. Whereas the multiplexer is a data selector, the de-multiplexer 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. 3–9. 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 de-multiplexer to handle this operation.

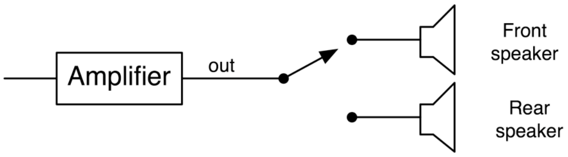


Figure 3–9. A mechanical distributor switch

The Boolean equations for the switch in Fig. 3–9 are:

$$\begin{aligned}\mathrm{Front\_speaker} &= \mathrm{OUT} \bullet \overline{S} \\ \mathrm{Rear\_speaker} &= \mathrm{OUT} \bullet S\end{aligned} \quad (S = T \text{ when the switch is down}).$$

The electronic gate equivalent of these equations is so simple that you are unlikely to find it in your library.

The drafting symbol for the dmux is simply the reversed image of the multiplexer which nicely displays the dmux's inverse mux behavior. The equations and symbol for a 4-wide dmux are:

$$\begin{aligned}Y0 &= \overline{S1} \bullet \overline{S0} \bullet G \\ Y1 &= \overline{S1} \bullet S0 \bullet G \\ Y2 &= S1 \bullet \overline{S0} \bullet G \\ Y3 &= S1 \bullet S0 \bullet G\end{aligned}$$

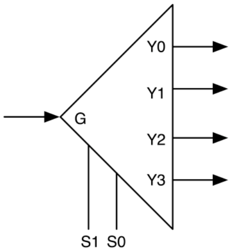


Figure 3–10. Equations and symbol for a dmux

To summarize, the de-multiplexer 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 the laboratory accompanying this book you will study the art of digital design and will participate in the design of two minicomputers. The first is small enough, and simple enough, for new initiates to digital design to finish and verify in a reasonable time. The second is a widely used commercial microcomputer for the more advanced student. In both examples, and indeed for every computer, an operation code determines which instruction to execute, and is clearly of a, “do this one”, type of operation—in other words the operation code must be decoded

The first project is modeled after the PDP-8, which has a 3-bit operation code field specifying 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:

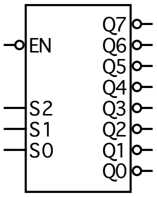
Operation code	Bits C	B	A	Instruction
0	0	0	0	AND
1	0	0	1	TAD
2	0	1	0	ISZ
3	0	1	1	DCA
4	1	0	0	JMS
5	1	0	1	JMP
6	1	1	0	IOT
7	1	1	1	OP

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

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

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

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. Most libraries will include decoders with a selection of output widths; some circuit design programs also allow you to tailor decoders to your own specifications, in the spirit of macro's in a software macro assembler. You need to exercise restraint with decoders, for every additional input bit the number of outputs doubles, (3 inputs = 8 outputs, 4 inputs = 16 outputs, 5 inputs = 32 outputs, …..). A typical library symbol for an 8-wide decoder would be:



Most decoders use NAND gates to decode input bit patterns and thus generate low active outputs when En=T; when En=F, all outputs go F.

The Encoder

The inverse 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 that 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". Priority truth tables are one of the few times the “X” notation is useful, leading to drastically compacted truth tables.

Method 1				
D3	D2	D1	B	A
0	0	0	0	0
0	0	1	0	1
0	1	X	1	0
1	X	X	1	1

Method 2						
D3	D2	D1	D0	B	A	W
0	0	0	0	-	-	1
0	0	0	1	0	0	0
0	0	1	X	0	1	0
0	1	X	X	1	0	0
1	X	X	X	1	1	0

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:

$$
\begin{aligned}
B &= 1XXX + 01XX \\
A &= 1XXX + 001X \\
W &= 0000
\end{aligned}
$$

Many libraries will not include priority encoders and you may have to fashion you own, but doing so in this framework will organize your efforts and clarify your designs.

As is often the case with hardware, we can devise serial or tree based priority encodes. Linear designs are simple, readily extensible to many stages, but have delays that are proportional to the number of stages. Tree based solutions usually take more hardware but have delays that grow as the logarithm of stage length and may thus be preferable for larger priority chains.

A linear solution for a type 2 encoder is shown below. Consider a priority chain where each module accepts a priority-in signal from its neighbor upstream and passes priority-out to its neighbor downstream, unless there is a priority request. Figure 3–11 is the schematic for the first 4 elements of such a chain.

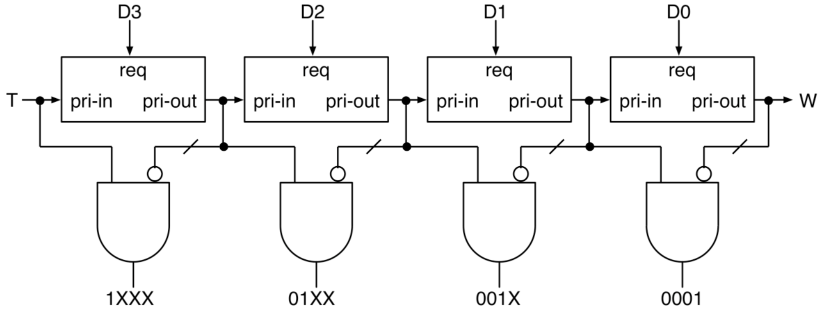


Figure 3–11. Serial priority chain

Module logic is given by this equation:

$$\text{pri.out} = \text{pri.in} \bullet \overline{\text{req}}$$

It is clear that the leftmost active request will inhibit all requests to its right, i.e. the leftmost active request has priority. When used in a software setting, operating systems will sometimes wish to disable the priority chain; something easily done by priming the leftmost module with a "F" instead of "T".

A tree based solution is shown in figure 3-12. In deriving this solution it is helpful to view the topmost module as one that divides the priority chain in half, enabling the proper half by presenting enabling "G" signals to lower tier modules. Request signals percolate upward, pruning lower priority requests on the way until only the highest priority request reaches the top module. The surviving request leaves a trail which guides the grant signal moving downward to the lowest tier, thus identifying the highest priority request. (Each module must be identical so they can be used in recursive descent to construct trees of arbitrary depth.)

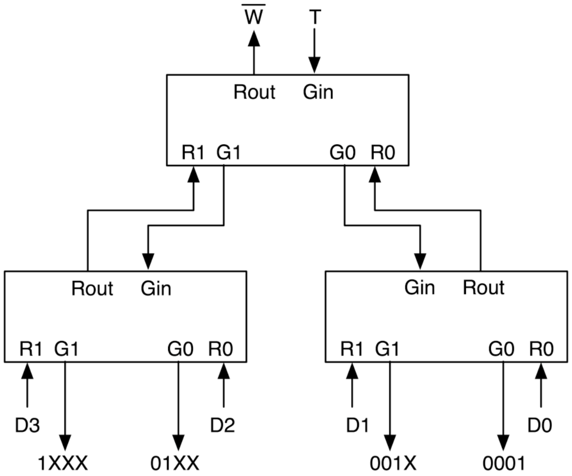


Figure 3-12. A tree based priority chain

Module logic is given by equations:

$$\begin{aligned}&\mathrm{Rout} = \mathrm{R1} + \mathrm{R0}\\&\mathrm{Gl} = \mathrm{R1} \bullet \mathrm{Gin}\\&\mathrm{G0} = \overline{\mathrm{R1}} \bullet \mathrm{R0} \bullet \mathrm{Gin}\end{aligned}$$

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 Equality 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

$$\mathrm{A\,.EQ.\,B} = \mathrm{A} \overline{\oplus} \mathrm{B} \qquad \text{Eq. (3–1)}$$

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

$$\mathrm{A\,.EQ.\,B} = (\mathrm{A}_{n-1} \overline{\oplus} \mathrm{B}_{n-1}) \bullet (\mathrm{A}_{n-2} \overline{\oplus} \mathrm{B}_{n-2}) \bullet \cdots \bullet (\mathrm{A}_0 \overline{\oplus} \mathrm{B}_0) \qquad \text{Eq. (3–2)}$$

We can make an important distinction based on whether the reference set of bits is unvarying, (a constant), or a varying pattern.

> Expanding the single-bit Eq. (3–1) into its AND, OR, NOT form, we have
>
> $$\mathrm{A\,.EQ.\,B} = \mathrm{A} \bullet \mathrm{B} + \overline{\mathrm{A}} \bullet \overline{\mathrm{B}}$$
>
> If *B* is constant, this equation can be simplified into one of two forms:
>
> $$\mathrm{A\,.EQ.\,B} = \mathrm{A} \quad \text{if B = T}$$
>
> $$\mathrm{A\,.EQ.\,B} = \overline{\mathrm{A}} \quad \text{if B = F}$$

Consider a comparison of an arbitrary 4-bit A with a fixed 4-bit B = 1,0,0,1.

> Equation (3–2) can be reduced to
>
> $$\mathrm{A\,.EQ.\,B} = \mathrm{A}_3 \bullet \overline{\mathrm{A}_2} \bullet \overline{\mathrm{A}_1} \bullet \mathrm{A}_0$$
>
> which can be realized with a 4-input AND element.

If the reference pattern is not fixed, we are stuck with Eq. (3–2).

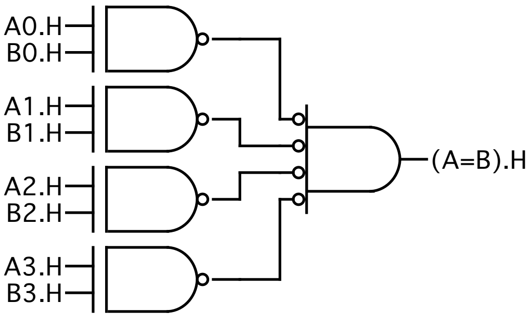


Figure 3–13. 4-bit equality circuit

Wider equality circuits can be handled either serially or by tree structures. A serial compare is just a slight modification of the serial priority logic; a 4-bit equality chain is shown in Figure 3–14.

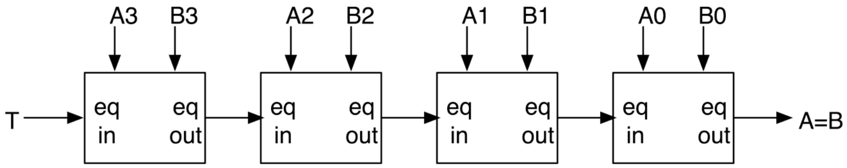


Figure 3–14. A serial compare circuit for two 4-bit numbers

Module logic for a block is: $eq.out = eq.in \bullet (A_i \overline{\oplus} B_i)$ and the circuit is readily extended to an arbitrary number of stages. The only reason for including it in our discussion is to point out that it distributes the AND of figure 3–13 into a number of 1-bit AND's. Unfortunately, while the circuit is simple, delay will be proportional to the number of stages; if speed is important tree structures will be faster.

Assuming we have a 4-bit equality module, as in figure 3–13, we can use them to form a 16-bit tree structured equality comparison as shown in figure 3–15

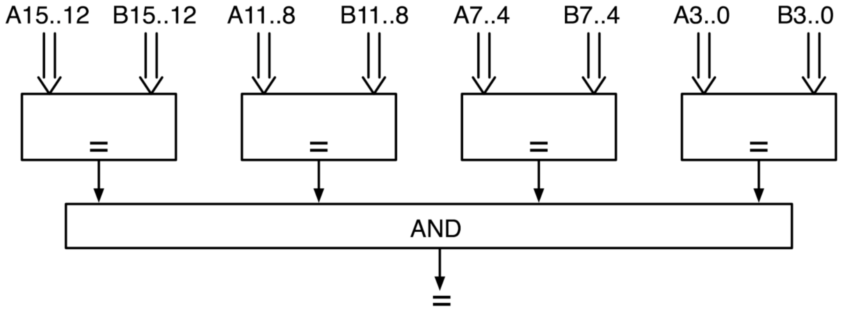


Figure 3–15. A tree structured 16-bit equality circuit

Now that we have a 16-bit compare modules we could tree them to form 32 or 64-bit compares. Circuit complexity is not that much greater than for the serial compare and the speed is much faster.

The Magnitude Comparator

If two numbers are not equal then one must be bigger than the other. There are various ways of attacking the magnitude comparator problem; serial ripple solutions, and faster parallel ones.

Lets consider two n-bit binary numbers, A and B, and a serial circuit that will return 3 signals A>B, A=B, A<B. Start at the most significant bit (why?). There are 4 possibilities for that bit, and that can lead us to an algorithm.

A_{n-1}	B_{n-1}	
0	0	test next bit position on right
0	1	A<B
1	0	A>B
1	1	test next bit position on right

If ($A_{n-1}=B_{n-1}$) then we must look at bit position n-2 and so on, considering each bit position in turn as we progress to lower order bits until we find a pair that differ. We then immediately know that A>B or A<B and that must be passed down through all downstream modules. If we get all the way to the low order bit position without finding a pair that differ then we know that A=B.

By analogy with Figure 3–14 a 4-bit magnitude compare circuit would be:

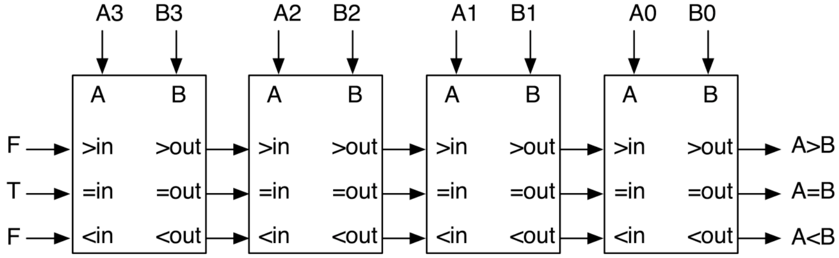


Figure 3–16. A serial 4-bit magnitude compare

We leave the derivation of the algorithm, as well as module and parallelizing logic for the problems.

Using an Arithmetic Unit for Magnitude Comparisons

If you are building a computer you may assume an arithmetic unit is available and this opens up alternate, and faster, means of doing magnitude comparisons. To compare the n-bit number, B, with another n-bit number, A, simply subtract B from A and look at the result, which will either be positive, zero, or negative. Positive and negative tests involve nothing more than looking at the result's sign bit—a trivial operation. The zero test is just a wide AND gate testing for all zero's, slower, but still relatively fast.

As discussed later in this chapter, most computers will have efficient multi-bit subtraction hardware that will be much faster than the ripple circuits discussed above and thus comparisons will inherit that speed.

A Universal Logic Circuit

We have gates for implementing the specific logic operations AND, OR, XOR, and so on. These gates are useful when we know at design time 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, (a truth table), with a suitable circuit.

Let the inputs to this circuit be A and B, and output be f(A,B), using the mathematician's functional notation. To select the particular logic operation, we must have 4 control inputs. Figure 3–17 shows this black box. The "slash 4" is a standard notation for 4 wires and the slash, which is reversed, must not be interpreted as a logical NOT.

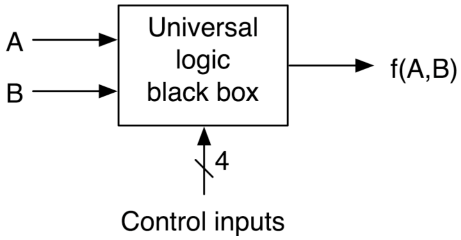


Figure 3–17. Inputs and output for the universal logic circuit

We require the black box to be able to perform any possible Boolean logic function of its two inputs. As we mentioned in Chapter 2, there are 16 functions of two variables. We routinely use several of these logic functions; the useful ones for digital logic are listed in Table 3–1.

AB	CTL	0	AND		A		B	XOR	+	NOR	XNOR	$\overline{B}$		$\overline{A}$		NAND	1
00	T0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
01	T1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
10	T2	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
11	T3	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Table 3–1

"A", "B", "0", "1", seem uninteresting at first sight but are useful functions in the logic design of CPU's.

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.
(b) Pass input A or input B through the circuit unchanged or inverted.
(c) Perform our important logic functions AND, XOR, OR, XNOR,

A NOT, B NOT.

(d) Perform the remaining functions of two variables; the four unlabeled columns play no important role in our study of digital design but we list them 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 should this device perform useful logic operations upon two inputs, but it should transmit either input, unaltered or inverted. In addition, it should 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, (the truth table), specifying the particular function. The obvious code values are {T0..T3}, the 4-tuple representation of the truth table's desired output. By a 4-tuple we simply mean an ordered set of 4 symbols, {T0,T1,T2,T3}, this should *not* be interpreted as a binary number and whenever we use a 4-tuple we will enclose the values in curly brackets to emphasize that fact. 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 truth table into the data inputs:

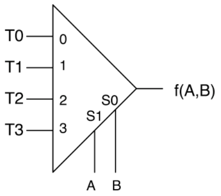


Figure 3–18. A Universal logic generator

Thus we may produce all 16 Boolean functions of two variables with a 4-input mux This is tight design!

Let us build an n-bit logic-unit for a CPU that will generate any one of the 16 possible logic functions, f(A,B), of two n-bit variables, A and B. We observe that the f values are simply truth table values for the logic function, f, we are performing on A,B, lets call them {T0..T3} for brevity, and use these labels to denote implied wires passing the truth table values to each mux. Presumably other parts of the computer will compute {T0..T3} and present them at the appropriate time to the logic unit for processing. For example, an assembly language command to perform the AND of two variables would present {T0..T3} = {0001} to the logic unit to form $A \bullet B$.

In drafting logic units, it is conventional to have the A,B inputs enter at the top, and the result, f(A,B), exit at the bottom of diagram. Changing the standard mux symbol to accommodate this results in Figure 3–19 for the low order 4 bits of a general-purpose logic.

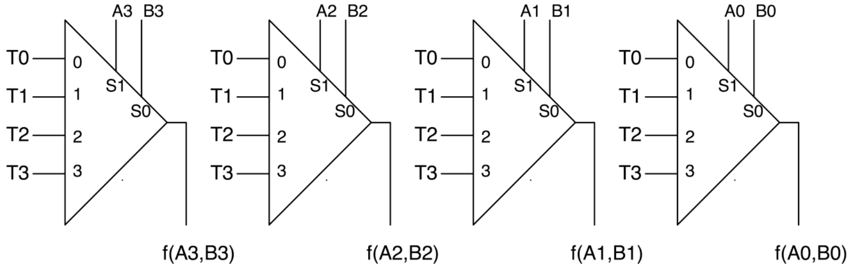
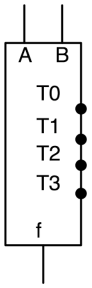


Figure 3–19. Low order 4 bits a general purpose logic unit

Schematic entry packages allow, indeed encourage, the packaging of complex circuits into compact symbols of your choice. A compact symbol re-packaging of Figure 3–18 might be:



Now we can make wide logic units with reduced clutter; figure 3-20 is an 8-bit example, here programmed to generate the AND of two 8-bit numbers. By judicious choice, your symbols can emphasize certain aspects of your design. Here we use dots on the right side of our symbol to emphasize that the truth table for our logical function, f, is fed in parallel to all bits of the logic unit.

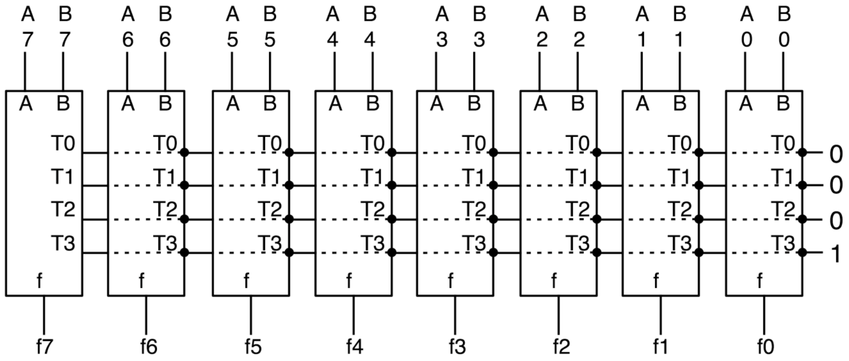


Figure 3–20. An 8-bit logic unit where each bit is the same circuit as figure 3–18

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 T.T. 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:

$$\mathrm{SUM} = \overline{\mathrm{Cin}} \bullet \overline{\mathrm{A}} \bullet \mathrm{B} + \overline{\mathrm{Cin}} \bullet \mathrm{A} \bullet \overline{\mathrm{B}} + \mathrm{Cin} \bullet \overline{\mathrm{A}} \bullet \overline{\mathrm{B}} + \mathrm{Cin} \bullet \mathrm{A} \bullet \mathrm{B}$$

$$\mathrm{SUM} = (\mathrm{A} \oplus \mathrm{B}) \bullet \overline{\mathrm{Cin}} + (\mathrm{A} \overline{\oplus} \mathrm{B}) \bullet \mathrm{Cin}$$

$$\mathrm{SUM} = \mathrm{A} \oplus \mathrm{B} \oplus \mathrm{Cin}$$

$\mathrm{Cout} = \mathrm{A} \bullet \mathrm{B} + \mathrm{Cin} \bullet (\mathrm{A} + \mathrm{B})$ $\quad$ $\mathrm{Cout} = \mathrm{A} \bullet \mathrm{B} + \mathrm{Cin} \bullet (\mathrm{A} \oplus \mathrm{B})$ is also correct

To perform addition on arrays of bits representing unsigned binary numbers, we may connect full adders together as in Fig. 3–21. 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 hardware wires are shown in Fig. 3–22.

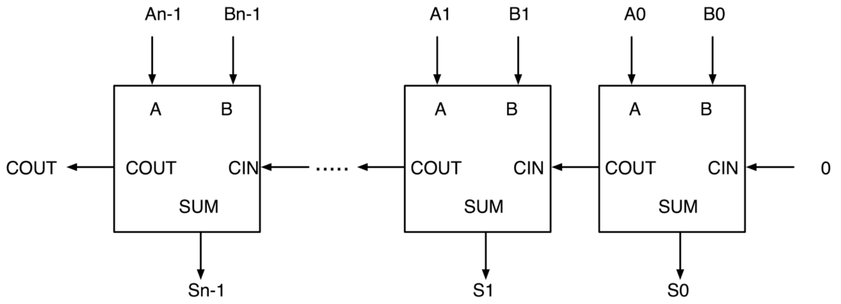


Figure 3–21. Addition with cascaded full adders

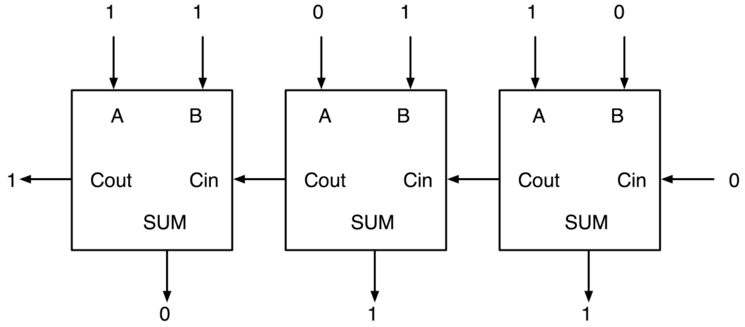


Figure 3–22. 101 + 110 = 1011 using full adders

This method of connecting full adders is called the *ripple carry* configuration, 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. A way to proceed is to abstract adders into 4-bit modules with a standard interface, which is unchanged if internal gate structure is changed to give greater speed, simpler structure, or some other desired metric. Another term for this abstraction is "information hiding" we don't care about the module's internals as long as it has the correct interface to the outside world; it could be a 4-bit ripple carry adder, a more sophisticated carry look ahead adder, or any other 4-bit adder that behaves properly.

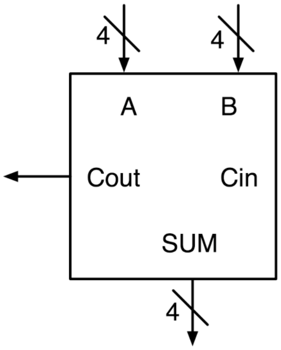


Figure 3–23. an abstract 4-bit add module

An example of a modularized 12-bit add would then look like this:

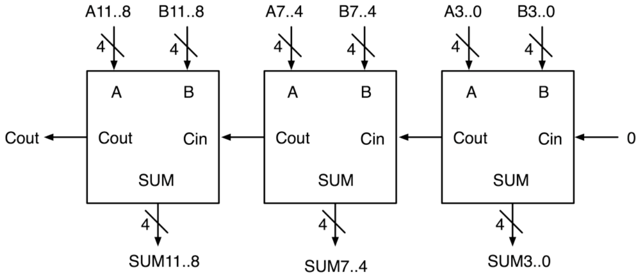


Figure 3–24. A modularized 12-bit adder

The "slash" notation can be extended to any number of bits, thus Figure 3–24 could be represented as in Figure 3–25 where we have gained compactness but lost detail.

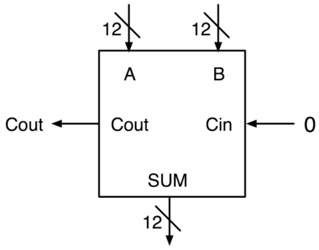


Figure 3–25. Figure 3–24 redrawn using the "slash" notation

Signed arithmetic. The multi-bit full adder circuit of Fig. 3–24 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. 3–24 can handle 11-bit integers plus a sign.

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's complement of the subtrahend: A MINUS B = A PLUS (MINUS B). Incrementing is a useful special case accomplished by setting the low-order Cin to 1.

A Simple Arithmetic Logic Unit

Adding an adder to a universal logic generator is elementary if we gather the relevant equations, the equations for a full adder are:

$$\mathrm{SUM} = \mathrm{A} \oplus \mathrm{B} \oplus \mathrm{Cin}$$
$$\mathrm{Cout} = \mathrm{A} \bullet \mathrm{B} + \mathrm{Cin} \bullet (\mathrm{A} \oplus \mathrm{B})$$

Simply adding an XOR to the universal logic block implements SUM, provided we program the universal logic block to generate A XOR B, {code 0110}:

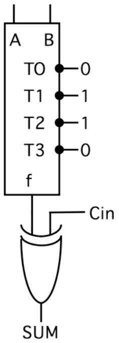


Figure 3–26. A simple adder

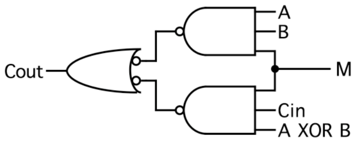
To make a general purpose ALU we need a way to switch between logic and arithmetic, and the XOR is in just the right location to act as that switch. Remember the XOR's truth table, here divided into two halves, one with M=0, the other M=1; M (the "mode" bit) can be used to pass a variable unchanged or pass it inverted depending on its value

M	In	OUT = $M \oplus In$
0	0	0
0	1	1
1	0	1
1	1	0

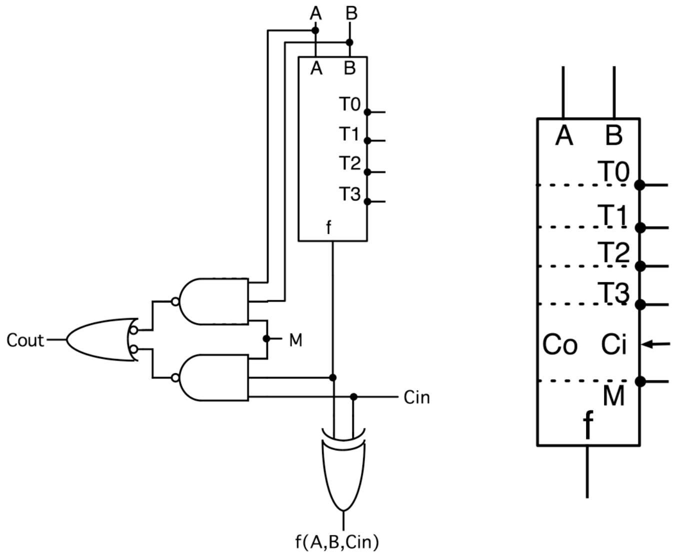
Redraw the above circuit, replacing Cin by X. If we could somehow force X to be zero, the XOR gate would pass the value generated by the logic block unchanged; we then have a *logic* unit. Conversely, if X=Cin we then have an *arithmetic* unit. Let's use a "mode bit", M, to switch back and forth between a logic and arithmetic unit; when M=0 we have a logic unit, when M=1, an arithmetic unit. The relevant equation for X is then:

$$X = M \bullet Cin$$

Remember that in a ripple carry adder Cin = Cout from the preceding stage so the relevant carry chain hardware simply becomes:



The Mode bit, M, turns Cout on or off, when M=0, Cout=0, and the carry chain is shut off with all carries =0; when M=1, the arithmetic carry chain is turned on. We have used the optimization that the logic unit will be generating A XOR B when acting as an adder and can use that in the carry generation circuit. The final logic for the 1-bit ALU then becomes:



(a) hardware (b) equivalent symbol

Figure 3–27. A 1-bit ALU

Using drafting software, a convenient packaging of this hardware might look like 3–27b.

Incorporating the carry logic inside our logic module yields the symbol for a 1-bit Arithmetic Logic Unit (ALU); a so-called “1-bit slice” of an ALU. We can now cascade these bit slices to give a schematic for an arbitrary sized ALU, figure 3–28 shows a 4-bit ALU.

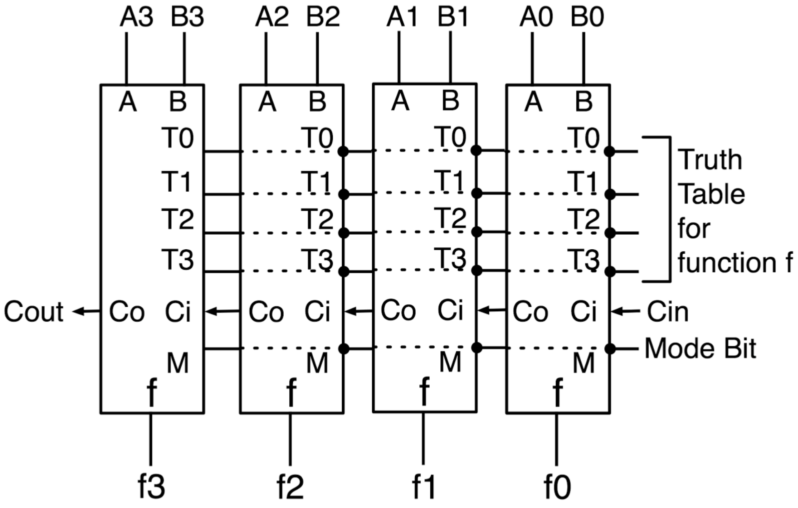


Figure 3–28. 4-bit general purpose ALU

To program this ALU, first decide if f(A,B) is to be a logic function. If so, set

M=0 and feed in the desired truth table on inputs {T0..T3}.

To switch to the arithmetic mode, set M=1 to turn on the carry chain; the function f(A,B) then depends on the truth table inputs as well as Cin to stage 0:

M	Cin	{T0..3}	f(A,B)
0	0	{0000}	0
0	0	{1111}	1
0	0	{0011}	A
0	0	{1100}	$\overline{A}$
0	0	{0101}	B
0	0	{1010}	$\overline{B}$
0	0	{0001}	$A \bullet B$
0	0	{0111}	$A + B$
0	0	{0110}	$A \oplus B$
0	0	{1001}	$A \overline{\oplus} B$

M	Cin	{T0..T3}	f(A,B)
1	0	{0110}	A plus B
1	1	{0110}	A plus B plus 1
1	1	{0011}	A plus 1
1	1	{1100}	MINUS A {2's Complement of A}
1	1	{0101}	B plus 1
1	1	{1010}	MINUS B {2's Complement of B}

Table 3–1. Commands for a general purpose ALU

This ALU is about as simple as one can design, more sophisticated ones exist and are described in the references.

Subtraction, (a detour into complement arithmetic)

Some early computers had separate subtraction hardware but modern systems universally use an *adder* to accomplish *subtraction*. How is this possible? It turns out we can't avoid doing a subtraction somewhere; the trick is to find a trivial way to do it before sending the subtrahend to the adder. The technique is called complement arithmetic, a standard tool known for centuries before the advent of digital computers. People using complements did so in the decimal system and by exploring its use in that number base you will be forced to comprehend the underlying theory; in binary, the operations are so simple that understanding can be compromised so we will take a detour through decimal complements before moving on to binary.

Consider the traditional subtraction formula:

(minuend – subtrahend = difference).

Are there special minuends that simplify subtraction? Upon reflection, the thing that messes up subtraction is borrowing; if we could find a minuend that never caused borrows, things would be as simple as possible. In decimal, a minuend of all 9's has this property; you can subtract digit by digit from either end without borrowing (the 9's complement *operation*). We call the resulting difference the 9's complement *notation* of the subtrahend. To make things concrete, lets consider a simple decimal calculator with a keypad, a plus operation key (only), and a 3-digit display.

Suppose we want to subtract 2 from 5 using only the plus operation. The 9's complement operation on 2 yields 997 in 9's complement notation. We can use the association law of arithmetic to advantage here:

$$5 + (999-2) = 999 + (5-2).$$

The 9's complement of 2 is easily performed *in your head* as you key in the value 997, then key in 5, and then hit the plus key. The result, 1002, doesn't fit in a 3-digit display and the leftmost 1 in the 1002 is discarded. But, the answer is off by 1, do you see why? So we simply add 1 to get the final correct answer. By adding a one we in effect are forming (1000−2) and we know from the start the leftmost 1 will be discarded. When we subtract a number, X, from 1000_{10} we get the 10's complement of X; the general rule is adding a 1 to a 9's complement turns it into a 10's complement. (To reiterate, the term complement is used in two different contexts here; an *operation* performing 1000 − X, and a *notation*, 998, for the result.)

The phrase, *in your head,* was emphasized because it had to be done *external* to the adder, just another way of saying the subtraction was done elsewhere, albeit in a trivial fashion, because of the special minuend value used to form the complement. We're not quite done, suppose you see 993 in the display. Is this the notation for a positive integer value of 993 or the notation for the 10's complement of 7? The adder, or display, neither knows or cares, it is up to us to *choose* a convention. If we want to use the complement number system for negative numbers, represented by their complement notation, we reserve the range 000 to 499 for positive integers, and the range from 500 to 999 to represent negative integers, $(-500,\cdots,-1) \Leftrightarrow (500,\cdots,999)$, in 10's complement notation.

But another problem lurks behind the scene. Add 400 + 300 and the calculator blindly adds them to produce 700, the representation of −300. What's wrong? By working in the complement domain we split the range, into those for positive integers, and those for negative integers, and the sum of 400 + 300 now winds up in the range reserved for negative integers. We leave it as an exercise to show that adding −400 to −300 also produces an erroneous result. Both conditions are defined as overflow and a complement adder must detect this condition and report that fact. (We defer the hardware detection of overflow to the binary domain where it is easily calculated).

To perform the 5−2 operation in binary we proceed exactly as above, the same principles apply, but hardware operations are simpler in the binary domain. Consider a 4-bit adder and ask what is the special minuend that generates no borrows during subtrahend subtraction. Clearly all 1's fit the bill and subtraction results in the 1's complement of the subtrahend.

$$\begin{array}{r} 1111 \\ -\underline{0010} \\ 1101 \end{array}$$

The 1's complement of 2 is thus 1101, simply the bit-by-bit complement of the subtrahend, something easily accomplished in hardware:

Now lets do the associative addition as we did in the decimal domain but use binary rules.

$$0101 + (1111 - 0010) \;=\; 0101 + 1101 \;=\; 10010$$

The leftmost 1 of the 10010 is discarded by our 4-bit hardware, leaving 2 as the final result, which again is off by 1 since we are working with a 1's complement subtrahend. But, looking at the circuit in figure 3-20 we see that a "1" can be added back in by simply turning on the Cin bit to the least significant adder stage, a pleasant result. Adding a "1" in this fashion, in effect, converts the 1's complement to the 2's complement, just as adding a 1 to a 9's complement turns it into a 10's complement.

Again, we split the range into positive integers, $(0, \cdots, 7)_{10} \Leftrightarrow (0000, \cdots, 0111)_2$, and the rest to the 2's complement representation of negative integers, $(-8, \cdots, -1)_{10} \Leftrightarrow (1000, \cdots, 1111)_2$. The left most bit is commonly called the sign bit, but strictly speaking it only indicates which part of the range you are in*. Overflow can occur just as in the decimal case whenever the sum of two numbers of the same sign results in a result with the wrong sign. We can formalize this statement as a logic equation:

overflow = [(sign of A) XNOR (sign of B)] XOR (sign of sum)

In appendix *, where overflow is developed in a more rigorous mathematical fashion, we derive an equivalent formula that requires less hardware:

overflow = (carry into the sign bit) XOR (carry out of the sign bit)

Subtraction—finally.

Now that we know how to handle complement addition, we can do a subtraction by first forming the subtrahend's 2's complement, in a separate machine instruction, before sending it on to the adder. Some machine instruction sets include a "Complement and Increment" instruction for generating 2's complements for just this purpose.

Another technique would be to modify our generic ALU to generate the subtrahend's 1's complement on its way into the adder stage, while simultaneously issuing an ADD instruction, with Cin to the least significant bit turned on. This does a subtraction in one machine instruction at the cost of extra hardware to generate the 1's complement. We leave this modification of figure figure 3-27 as grist for your mental mill, (hint use the XOR as a controlled inverter).

Speeding Up Addition

Ripple-carry schemes for binary addition are simple but very slow, lets see what we can do to speed up the process—which reduces to speeding up carry generation. Binary addition is 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 multi-bit 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 non-arithmetic work we can usually find a simple 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 in adjacent or more distant bits. Unfortunately, arithmetic does not have this simple 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 is 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 is 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 A_i , B_i and C_i , and outputs S_i and C_{i+1}. We will focus on some properties of the data inputs A_i and B_i . We introduce a carry-generate function G_i that is true only when we can guarantee that the data inputs to stage i will generate a carry-out. We introduce a carry-propagate function P_i that is true only when a carry-in will be propagated across stage i to produce a carry out. For a 1-bit sum, the truth tables for G_i and 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:

$$C_{i+1} = G_i + P_i \bullet C_i \tag{3-4}$$

$$S_i = P_i \oplus C_i \tag{3-5}$$

(To verify the equation for 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 in 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 multi-bit 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:

$$C_1 = G_0 + P_0 \bullet C_0 \tag{3-6}$$

$$C_2 = G_1 + P_1 \bullet G_0 + P_1 \bullet P_0 \bullet C_0 \tag{3-7}$$

$$C_3 = G_2 + P_2 \bullet G_1 + P_2 \bullet P_1 \bullet G_0 + P_2 \bullet P_1 \bullet P_0 \bullet C_0$$

$$C_4 = G_3 + P_3 \bullet G_2 + P_3 \bullet P_2 \bullet G_1 + P_3 \bullet P_2 \bullet P_1 \bullet G_0 + P_3 \bullet P_2 \bullet P_1 \bullet P_0 \bullet C_0$$

While messy, note that each equation involves only generate and propagate operators and the original carry-in, and thus all carries can be calculated in *parallel.* In practice, it becomes too complex to do this for more than 4 bits. If incorporated into a 4-bit slice a module symbol might look like

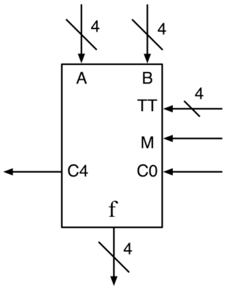
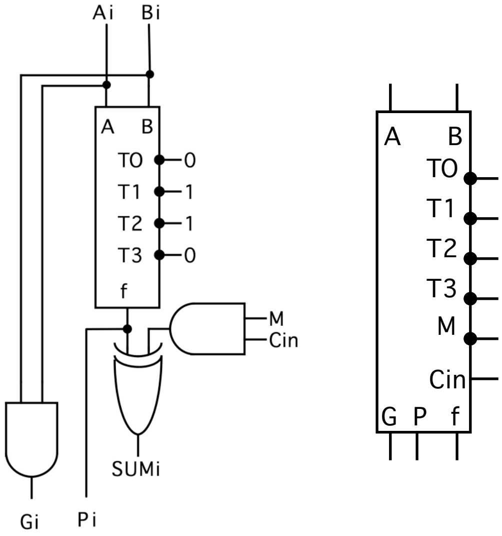


Figure 3–29. A 4-bit ALU slice with carry look-ahead

Figure 3–29 looks suspiciously like Figure 3–23 but that's exactly what information hiding is supposed to accomplish. Let's dive inside the 4-bit block to see how the carry generation circuits work. Each stage is modified to calculate Gi and Pi as shown in Figure 3–30a. Figure 3–30b shows a symbol embodying its behavior as a general ALU. f(A,B) will be a logical function represented by the truth table inputs when M=0; when M=1 and T0-T3={0,1,1,0} then f(A,B,Cin)=SUM(A,B,Cin)



(a) (b)

Figure 3–30. 1-bit ALU modified for carry-lookahead

Using 3–30b, the internal structure of a 4-bit carry look ahead module becomes:

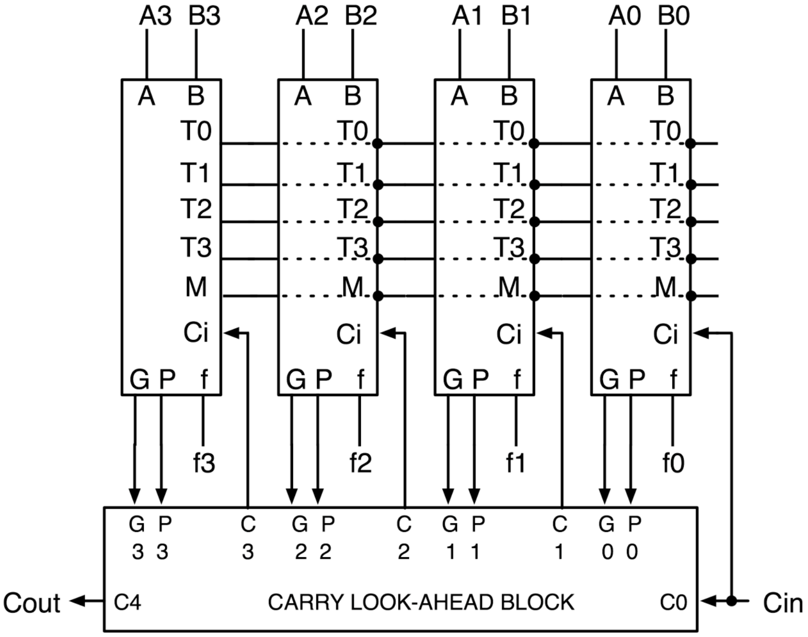


Figure 3–31. Internal structure of a 4-bit carry look ahead ALU

If we are building a large ALU from 4-bit-look-ahead-slices, we are still faced with the ripple-carry problem across the boundaries of each 4-bit slice, even though within each slice the carry-out is being computed rapidly. For instance, in Fig. 3–32, 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 4-bit slice, but not across the slices.

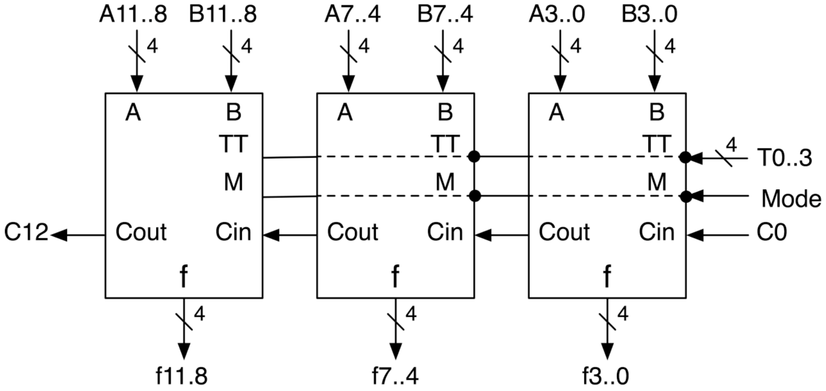


Figure 3–32. A 12-bit ALU with carry look-ahead

But generate and propagate can be applied to 4-bit modules as well as individual adder stages. Group G now refers to a generate inside a 4-bit adder module, independent of Cin to that module; group propagate will send Cin to Cout across the group. So a 16-bit high-speed adder with auxiliary carry look-ahead would look like figure 3–33 (we have emphasized the group look ahead logic and suppressed detail inside each 4-bit slice). Also, figure 3–32 is an adder only but conversion to a general purpose ALU is straightforward.

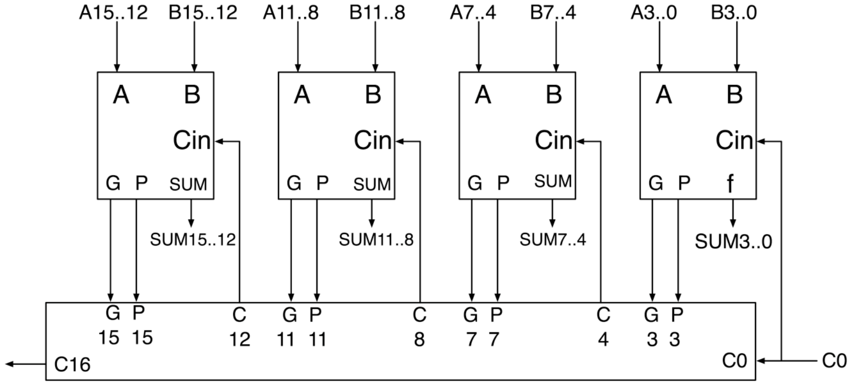


Figure 3–33. A 16-bit adder with group carry look-ahead

Instead of relying on each 4-bit slice to send its computed carry-out rippling on to the next most significant 4-bit slice, 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 4-bit slices. The more significant slices do not have to wait for their carry-in signals to ripple in from lower stages.

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

$$C_4 = G_3 + P_3 \bullet C_0$$

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

etc

The look-ahead block can also produce its own version of G and P, so that look-ahead circuits of more than 16 bits may be constructed by adding levels of block look-aheads. Two levels 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 the block delay.

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.

Modern ALUs and Status Bits (C,S,Z,V)

Most modern ALUs report, and store, carry, sign, zero, and overflow status information any time they perform an arithmetic operation. These 4 status bits, usually called C,S,Z,V, are useful for creating powerful conditional branch instructions. The C bit is just the Cout bit from the most significant bit; the S bit is just the sign bit —the high order bit of the result; the Z bit =1 if the result is zero. An ALU can store these 3 bits in a status register with essentially no overhead. The overflow bit is set when the result of an arithmetic operation is invalid.

DATA MOVEMENT

One of the most important operations in digital design is moving data between a source and a destination. When multi-bit data is involved the transfer medium is a *bus* and buses are used everywhere in digital logic, both internally in CPU's, and externally for communication with peripherals. Originally a bus meant a bundle of wires to transport multi-bit data in parallel; later the concept was expanded to include a general data path, with multiple senders and receivers sharing the bus

Parallel Buses vs. Serial Buses

Data movement inside a CPU uses a parallel set of wires, a bus, with each wire dedicated to one bit of a word or byte; this parallel protocol is very fast but uses precious real estate on the chip's Silicon surface. Circuit designers worry a great deal about "skew", getting all bits to arrive at a destination within a very small time window. If there's lots of skew, the time window will be wide and that will slow down communication – something that clearly must be avoided. Fortunately, skew can be tightly controlled as long as the bus is in Silicon and all high performance computers use parallel buses for inter-register communication as well as communication between a CPU and cache memory. When you move "off chip", control of skew is more difficult and motherboard designers usually try to keep bus physical length as short as possible. If you look at the bus between RAM and CPU on your PC you will see this principle at work.

When you move "off board", for example, communication to a disk drive, skew problems become unmanageable and the modern fix for this is counterintuitive—we go to serial busses. Consider the usual situation where you are sending 8-bit bytes down a bus. The serial bus protocol loads a byte into a high-speed shift register and serializes it, one bit at a time, for communication down a path. The receiver then re-assembles the bits into a byte for the receiver's consumption. Obviously, since only one bit is sent at a time there is no skew. The absence of skew more than makes up for the serialization/de-serialization step and modern serial buses work at 6 GHz, amazing! For a fuller discussion of serial buses see appendix *

Now Back to Generic Bus Protocols. Controlling the Bus. We have a building block to move data—the bus—that takes the form of just one twisted pair for serial busses or *n* wires for parallel busses. 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 bus designer. 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. Unfortunately, knee-jerk responses are sometimes not optimum but the multiplexer does indeed solve the "one talker" problem and is therefore pedagogically useful. 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. 3–34. 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 lots 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. As a result, it is seldom used in real hardware.

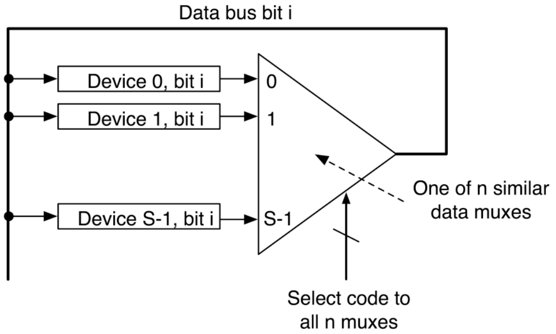
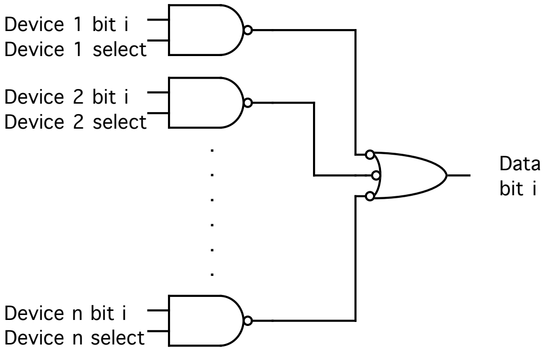


Figure 3–34. Bus control with multiplexers.

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. 3–35.



One of N similar circuits

Figure 3–35. 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 revisit concepts covered in chapter two and are widely used in real hardware.

Open-drain (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-drain gates produce wired-OR when truth is represented by a low voltage. The advantage of open-collector circuits is the elimination of the wide OR gate used to merge signals in figure 3–35.

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. If the receiver knows who it wants to listen to, it can communicate with the sending device and enable its AND gate thereby enabling that source to put it's data on the bus.

What if things are more democratic and each device can individually decide when it wants to talk with the receiver, and what if the receiver doesn't want to listen? Fundamentally, what happens is the listener activates a priority chain and talkers make requests to the chain. The highest priority talker then enables its data on to the bus. This is covered in more detail in appendix (*)

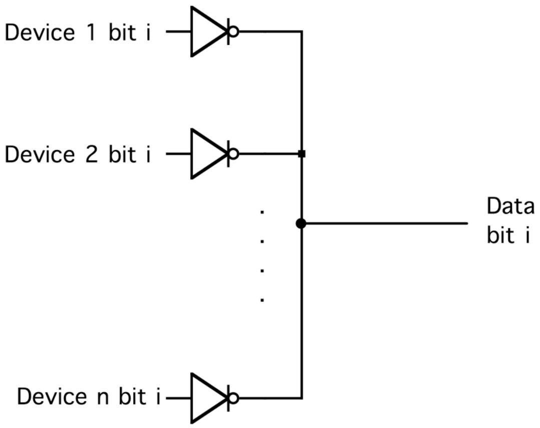


Figure 3–36. Controlling access to the data bus with open-drain buffers

Three-state outputs—the most common protocol. 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 library modules incorporate three-state data outputs. The fundamental use is in bussing, so three-state outputs often provide power buffering like their open-collector cousins. Tri-state busses are fast and that is the reason for their dominance.

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 library 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 modules providing the bus's source data will have three-state output control built in; in other cases, we may need to add tri-state buffers to achieve three-state control. Figure 3–37 is a typical three-state bussing configuration.

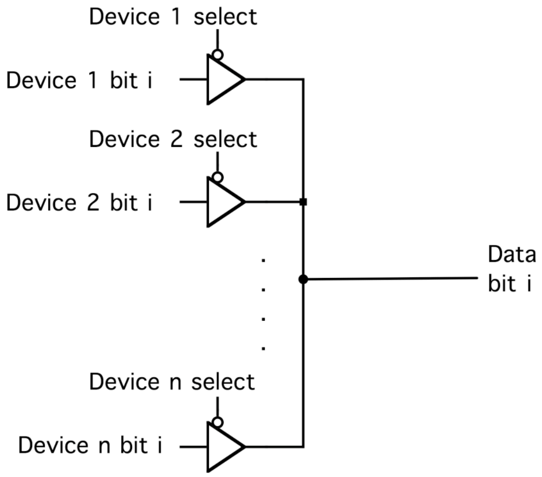


Figure 3–37. Controlling access to the bus with 3-state buffers

Tri-state busses have the same “who’s talking” problem as open collector busses but in most tri-state applications the receiver knows who it wants to listen to and can communicate that to the tri-state drivers. Memory busses are a prime example.

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 as the source 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, En

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 distinguishes a combinational circuit from a sequential one?

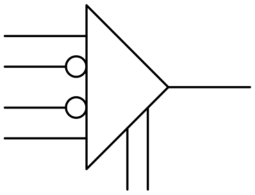
3-2. Explain the structure and the function of the multiplexer. What are the two major types of output enable found in multiplexers.

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

3-4. If your simulator library contains an 8-input 3-state mux, use them to construct a 16-input mux, and verify.

3-5. Consider figure 3–12. How would a mixed logician handle 3–13 if both A,B are low active, if one is? (assume you only have XOR gates).

3-6. The 4-input multiplexer symbol below looks like a mixed-logic notation. Why do we not find this symbol useful?



3-7. Build the 4-output demultiplexer in Fig. 3–10, using NAND, NOR gates and verify using a simulator. Will your design also serve as a decoder? If so, how?

3-8. What is the most important characteristic of the outputs of a decoder?

3-9. Construct a building block that will decode a 4-bit binary code into one of 16 outputs, using the 4-input demux of problem 3–6.

3-10. What is the purpose of an encoder? Why are practical encoders *priority* encoders?

3-11. Explain the difference between the concepts of encoding and decoding.

3-12. Using NAND, NOR gates, design a priority encoder that accepts five inputs and produces a 3-bit output code. Use method 1 of the text. Repeat using a tree encoder. Verify both your circuits with a simulator.

3-13. *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, $A_1 \oplus A_2 \oplus A_3 \oplus .... \oplus A_n$, expresses an odd-parity function of n bits.

3-14. Consider Figure 3–15:

a) It seems a little odd that the equality chain starts on the left with a T input but the other two chains start with an F input. Justify.

b) Could the bit order be reversed, i.e. low order bits be on the left for each of the 3 chains?

c) Build a 3-bit comparator and verify with a simulator, or by hand.

3-15. Construct a 4-bit parallel compare module according to problem 3–12 and verify using a simulator.

3-16. 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 A < B, A = B, and A > B in this notation. Explore whether the module of problem 3–15 is useful in realizing these equations. Produce a circuit (either with or without 3–15's module) for generating the three comparisons.

3-17. Design a 3-bit full adder equivalent to Fig. 3–20, using 1-bit full adders fabricated from NAND, NOR,NOT, and XOR gates.

3-18. Modify Fig. 3–20 to perform the operation A (+) B (+) 1.

3-19. Using the 4-bit ALU module of figure 3–22, and any necessary additional 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.

3-20. Verify the circuit of problem 3–20 using a simulator

3-21. 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.

3-22. Verify Eq. (3–4 and 3–5) for the full-adder sum expressed in terms of the carry-generate and carry-propagate operators.

3-23. Derive the equations for overflow in a 2's complement adder.

3-24. Discuss the merits of controlling a bus with:

c. Multiplexers.
d. OR gates.
e. Open-collector buffers.
f. Three-state buffers.

3-25. Three-state control of outputs is common. Why do we not employ three-state control of inputs?

3-26. Design bussing systems similar to Fig. 3–33 for six 4-bit devices, using:

(a) Open drain bus drivers
(b) Tri-state bus drivers

3-27. The multiplexer bus control method shown in Fig. 3–33 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.