2

Realizing Logic in Hardware

REPRESENTING TRUE AND FALSE WITH PHYSICAL DEVICES

Boolean algebra and truth tables are our tools for expressing logical relationships. To use these tools in the real world, we must have some physical way to represent TRUE and FALSE, the fundamental constants of logic. Digital systems record T and F in several ways:

(a) *Punched cards.* An old technology going back to Babbabge, and before then to Jacquard's loom (1801). A hole punched at a given spot in the card might represent TRUE; no hole at that spot would then represent FALSE.

(b) *Magnetic tapes or disks.* Magnetic tapes or disks represent logic data with magnetized areas on the recording surface. The designer might choose a south pole sticking out of the surface to be a 1, in which case a north pole would be 0.

(c) *Switches.* A switch has two states, closed and open. The digital designer may choose either state (but not both!) to represent logic truth; for example, an open switch may represent 1.

(d) *Voltages.* In digital electronic circuits, T and F are represented by voltage. For instance, the popular CMOS family of digital circuits produces two voltage levels: ~0 V and ~1.8 V.

Each of these four examples has only two states—in a punched card either there is a hole or there is not a hole. This two-valued, or binary, characteristic of the digital world makes Boolean algebra the natural way to formalize the behavior of these physical devices. Conversely, the need to implement logical constructs in physical devices makes these binary devices useful. If more than two values existed, more complex algebras would be needed to handle the multiplicity of values. Perhaps fortunately, engineers have had only limited success in designing reliable nonbinary devices. It is questionable whether multi-valued devices would be desirable, since the resulting systems would be harder to troubleshoot. In practice, it is important to know that any signal can have only two values, TRUE or FALSE.

The designer may select the physical representation for T and F. In a punched card, representing 1 with a hole seems natural, but such an assignment is not a logical necessity. With equal validity, we could let the absence of a hole represent 1. On magnetic discs either a north or a south pole can be a 1; both conventions exist. The same is true of switches.

In this book, we will use electronic logic circuits extensively. We let H stand for the high-output voltage level of a digital device and let L stand for the low-output voltage level. Each family of devices has its own H and L output voltage ranges. In CMOS, 0 and +5 volts are often used to interface between devices while internal

H,L voltages are usually lower depending on device geometry. Actual values are of little importance for synthesis since the abstraction to H,L is perfectly serviceble. In this chapter, we will use simple devices to build the basic logic functions described in Chapter 1; historically, each such device is called a *gate*. Here we concentrate on gates—the basic tools for realizing logic equations. In Chapters 3 and 4, you will meet more complex elements constructed from gates.

Like other forms of digital devices, electronic logic circuits offer a choice for representing T and F. Be flexible in your choice. Sometimes it is advantageous to let H represent T and to let L represent F; at other times the converse is more convenient. Either will do, if you let the rest of the world know your choice.

MIXED LOGIC: REPRESENTING AND, OR, and NOT

We may represent logic truth by either of the two voltage levels in a digital electronic device. If we apply this notion faithfully, a powerful and beautiful design tool emerges as we represent logic equations with physical hardware. We now undertake the development of clear and systematic ways of building and describing hardware circuits for logic expressions. Efforts at solving digital problems yield logic equations and logical structures; the hardware must faithfully embody these equations and structures. Furthermore, we certainly wish the documentation of our hardware (our *circuit diagrams)* to convey the spirit of the solution to the original problem. Documenting the hardware for a logic circuit is called *digital drafting.*

The foregoing thoughts suggest some criteria for drafting methods:

(a) We wish to synthesize (create) a physical realization of any logic expression directly from the logic, in a straightforward, natural, and rigorous manner.

(b) We wish to be able to analyze (pick apart) a physical realization and directly recover the original logic expressions.

These are strong conditions; many digital drafting and construction techniques in use today do not meet them. The conditions require that the circuit diagram clearly and fully display both the logic and hardware. We can identify several implications of these requirements:

(1) The drafting notation should represent the Boolean expressions in AND, OR, and NOT form—the natural way we develop our logic.

(2) The correspondence between a logical value (T or F) and its voltage counterpart (H or L) should be evident everywhere in the circuit diagram.

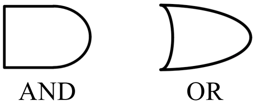
(3) The notation should clearly identify each physical device in the circuit. The key to satisfying these requirements is a representation called *mixed logic*. This notation was first published in a coherent form in 1968(*)

(*) P. M. Kintner, "Electronic Digital Techniques," McGraw Hill 1968; and F. Prosser and D. Winkel, "Mixed logic leads to maximum clarity with minimum hardware," Computer Design, May 1977, pp. 111-117.

But mixed logic was used in the Philco TRANSAC computers in 1957 and the technique is probably even older than that. We will develop mixed-logic methodology carefully, since the principle is vital to clear, top-down design.

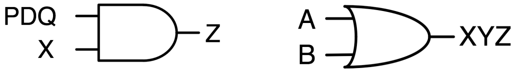
Mixed Logic

Showing the logic. We choose a unique symbol for each of the natural logical operators. The following shapes represent the *logical* AND and OR operators:



Whenever we see these shapes, we know that we are representing a logical AND or a logical OR function. Furthermore, every logical AND and OR in our original expression will appear in the circuit diagram as the corresponding shape.

Now we make a giant leap from pure mathematics to transistors. Transistors accept voltage inputs and produce voltage outputs. How can we map logic to voltage? In figure 2–1(a) the AND symbol is an abstraction representing an assemblage of transistors which accepts voltage on two wires, PDQ and X, and outputs a voltage on wire labeled Z. In 2–1(b) the OR symbol is an abstraction representing a different transistor structure which accepts voltages on wires labeled A,B and produces a voltage on wire labeled XYZ.



(a) (b)

Figure 2–1

But what voltage represents Truth? Our circuit diagam must unambiguously convey what voltage represents truth on every wire, and also, imply a corresponding transistor structure that maps input voltages to an output voltage while preserving the logic implied by the symbol. (For the moment we will assume that such transistor structures exist).

In circuit diagrams, graphic symbols imply a physical device that performs the logic operation. This can be a primitive gate on the surface of an integrated circuit, cells of a programmable gate array configured to implement the logic implied by the graphic symbol, or individual gates of an individually packaged integrated circuit, an obsolete technology largely of historical interest. Let's see how to record the voltage information on the diagram without altering the logic.

Logic conventions. How does a device represent T and F? There are two logic levels (T and F) and two voltage levels (H and L). Two useful possibilities exist:

(a) T is represented by H (and F is represented by L).

(b) T is represented by L (and F is represented by H).

The first form, with T = H, is called *positive logic;* the form with T = L is *negative logic.* When one of these relationships of truth and voltage is used consistently throughout a design, we refer to *a positive-logic convention* or a *negative-logic* *convention* for the design. The *mixed-logic convention*—our convention—allows us to use positive or negative logic at any point in our design, as we desire. The clarity gained by this innocent-sounding step is enormous.

Showing the device. In our exposition of mixed logic, we represent T = L by a small circle on the corresponding terminal of the logic symbol. The absence of a small circle means that T = H at that point. The circles do not change the logic operation. For example, each of the symbols in Figure 2–2 is a mixed-logic implementation of a logical AND function of two variables. We emphasize this point again: each of these symbols (and there are four more) represents a different *physical realization* of the same truth table:

Logic		
Inputs		Output
F	F	F
F	T	F
T	F	F
T	T	T

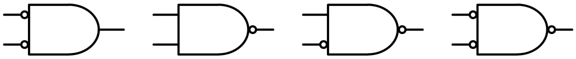
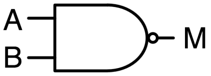


Figure 2–2. Symbols for a logical AND

To reiterate, we have left the pure logical operators of Chapter 1 and now deal with real physical devices to implement the logical AND. If you wish, consider the shape to represent the *logic*, and the presence or absence of small circles to be voltage polarities that will make each one of the four *different* transistor structures act like an AND.

Each symbol defines a particular type of physical device. Since we know the logical truth table (because of the symbol's shape) and the voltage representation of truth on each input and output (by the presence or absence of circles), we can immediately write down the voltage table for any symbol.

For example:



=

Logic		
A	B	M
F	F	F
F	T	F
T	F	F
T	T	T

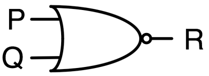
=

Voltage		
A	B	M
L	L	H
L	H	H
H	L	H
H	H	L

Here, the symbol completely defines the behavior of both the logic and the voltage. We hope that the voltage table is one readily implemented in CMOS (some of the AND gates in Figure 2–2 are not). This one is, and is usually referred to as a 2-input NAND. The term NAND arose historically from the view that this gate implemented the logical NAND (NOT AND) function in positive logic. The user of positive logic is forced into this interpretation, but since NAND is not a familiar and intuitive logic function, mixed logicians are not much interested in expressing logic in terms of

NANDs. We discuss the positive-logic convention later in this chapter.

Another common CMOS device is the 2-Input NOR Gate. This device can perform the logical OR function when T = H at the inputs and T = L at the output:



⇔

Logic

P	Q	R
F	F	F
F	T	T
T	F	T
T	T	T

⇔

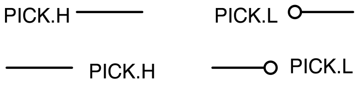
Voltage

P	Q	R
L	L	H
L	H	L
H	L	L
H	H	L

The name NOR derives historically from conventional positive logic, in which this gate performs the logical NOR (NOT OR) function.

Signal names in mixed logic. In a physical circuit, voltages represent values of the logic variables. We will refer to the actual voltage representation of a logic variable as *a signal.* When we label a circuit's inputs and outputs with the names of logic variables, we need a rule for also describing the voltage polarity of that variable at that point. Naming the logic variable is not enough; we must also name the signal. Here is the convention used in this book for creating names of signals that correspond to names of logic variables:

If a signal has T = L, append a terminal L to the logic variable's name. If a signal has T = H, append a terminal H to the logic variable's name. For example, the logic variable PICK might appear in a circuit as the signal PICK.L or as PICK.H, depending on its particular voltage representation at that point in the circuit. The terminal L is always associated with a small circle on a line in the circuit diagram; the H form is always associated with a line having no circle:



Understand two points thoroughly:

(a) The logic variable is the same in both representations. PICK.H and PICK.L are two different ways of physically forming the logic variable PICK.

(b) L does *not* mean that the voltage is low. Rather, it means that *if* the voltage is low, *then* the value of the logic variable is *true.* H is interpreted analogously.

Frequently, a circuit diagram contains both signals for a logic variable (on different wires!). We display the voltage convention on the wires of the circuit diagram and also in the signals' names. You may think this is redundant, but it is an important aid to clarity. If you show line and signal notations rigorously, your diagrams will show *all* the logic and *all* the voltage information in the circuit, clearly and conveniently.

Whenever it is necessary to write the name of a logic variable or signal more than once in a design, the need for the signal notation arises. This happens when signals appear on more than one page of the circuit diagram, when the designer prepares a master list of signal names for the circuit, and in other circumstances. Figure 2–3 illustrates a common digital drafting situation. In the figure, a signal generated on one page serves as an input elsewhere. With proper notation, there is never any doubt as to which signal is meant.

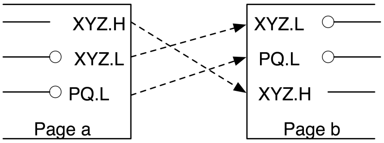
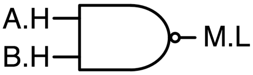


Figure 2–3. Illstrating the need for a convention for naming signals

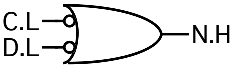
Naming signals is important. The particular notation for distinguishing the two signals for a variable is not crucial as long as you are consistent. Some people prefer to append a + and – , or ↑ and ↓, to a variable's name; others use a terminal / to indicate that T = L. There is no widely accepted standard convention. We use H and L in this book because of their strength in displaying voltage assignments.

Using this signal convention, we reach the final form of the earlier AND implementation:



	Logic				Voltage		
	A	B	M		A.H	B.H	M.L
	F	F	F		L	L	H
=	F	T	F	=	L	H	H
	T	F	F		H	L	H
	F	T	T		H	H	L

AND/OR duals. Here is another example, this time for the OR operation:



	Logic				Voltage		
	C	D	N		C.L	D.L	N.H
=	F	F	F	=	H	H	L
	F	T	T		H	L	H
	T	F	T		L	H	H
	T	T	T		L	L	H

Inspect the voltage table. You can see that it is equivalent to the one in the previous example, and therefore again corresponds to the 2-input NAND gate. (Remember that truth tables display the same logic if rows are scrambled). But this time the logic operation is OR. In this particular use of OR, T = L at both inputs and T = H at the output. We have identified two uses for the 2-input NAND gate—to implement AND and to implement OR. The AND-OR duality of gate usage is always present, and is related to the principle of duality in Boolean algebra. On a given gate, the dual symbols for AND and OR have reversed circles.

Similarly, you may derive mixed-logic notations for other devices. In Figure 2–4, we have drawn some common gates and have shown their mixed-logic symbols and their standard names. In practice, it is simple to find the mixed-logic uses of any gate directly, and the other one follows immediately by swapping AND-OR shapes and reversing the circles. Also, the conventional gate names are usually sufficient for writing down the mixed-logic symbols. Many integrated circuit technologies embody AND and OR, as well as NAND and NOR, giving us powerful and flexible tools for implementing the logic.

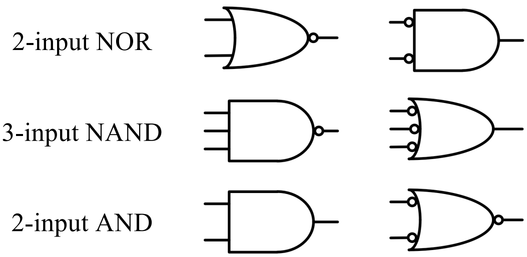


Figure 2–4. Mixed logic symbols for various gates

Some detailed circuit examples. To bring home the exact implications of the mixed-logic notations, let's study the circuit in Figure 2–5. Figure 2–5 means:

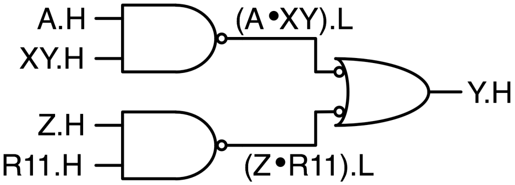


Figure 2–5. Schematic of Y=A•XY+Z•R11

(a) There are two physical devices (both 2-input NAND gates) that function as logical ANDs when T = H at the inputs and T = L at the outputs.

(b) There is another physical device (again a 2-input NAND gate) that functions as the logical OR when T = L at its inputs and T = H at its output.

(c) There are seven wires carrying voltages for the logic variables A, XY, Z, R11, (A•XY), (Z•R11), and Y.

(d) Truth (T) is represented by a high voltage (H) on signal wires A.H, XY.H, Z.H, R11.H, and Y.H.

(e) Truth (T) is represented by a low voltage (L) on signal wires (A•XY).L and (Z•R11).L.

(f) The circuit implements the logic equation Y = A•XY + Z•R11

Consider another example, Figure 2–6. This drawing tells us:

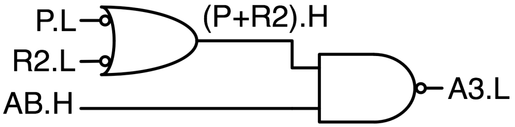


Figure 2–6. Schematic of A3=(P+R2)•AB

(a) There is one physical device (a 2-input NAND gate) that functions as the logical OR when T = L at its inputs and T = H at its output.

(b) There is another physical device (again a 2-input NAND gate) that functions as the logical AND when T = H at its inputs and T = L at its output.

(c) There are five wires carrying voltages for the logic variables P, R2, (P + R2), AB, and A3.

(d) Truth (T) is represented by a high voltage (H) on signal wires (P + R2).H and AB.H.

(e) Truth (T) is represented by a low voltage (L) on signal wires P.L, R2.L, and A3.L.

(f) This circuit implements the logic equation A3 = AB•(P + R2).

Figures 2–5 and 2–6 are straightforward illustrations of the mixed-logic drafting conventions.

Mixed-Logic Theory

With this introduction to basic notations, let's examine the theoretical basis of mixed logic. We will first consider the logical identity operation—one that we usually take for granted, but which provides important insight into the properties of mixed logic. Look at Figure 2–7.

A logic variable A is the input to a box that performs the logical identity operation and produces Y as output: Y = A. The truth table for the behavior of the box is just that of the identity function. If we consider the possible voltage implementations of this box, we have four choices: A.H and Y.H; A.H and Y.L; A.L and Y.H; and A.L and Y.L. Each of these choices of voltage representation yields a voltage table, and each choice results in a different mixed-logic realization. As usual, we use mixed-logic circles to indicate that T = L.

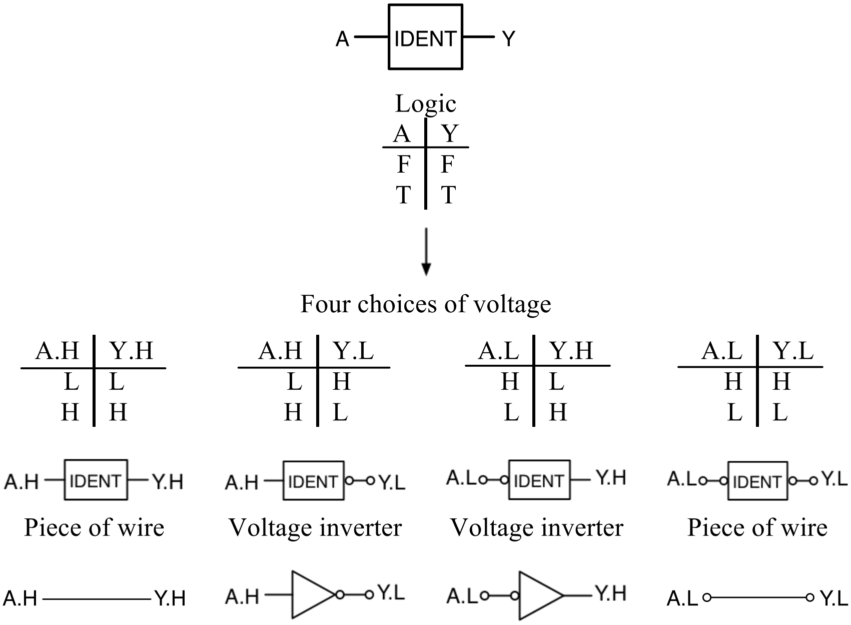
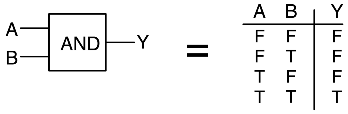


Figure 2–7. Realizations of the logical identity operation

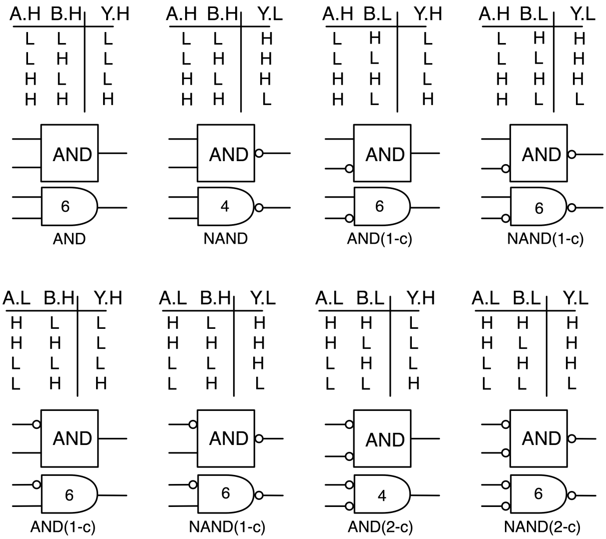
Contemplate the devices required to realize the four voltage tables that correspond to our four diagrams. When A.H yields Y.H, we need only a piece of wire, since the voltage does not change in passing through the identity box. Similarly, when A.L yields Y.L, we need only a piece of wire. On the other hand, the voltage table for A.H and Y.L specifies a voltage inverter. The voltage table for A.L and Y.H also specifies a voltage inverter. The triangle with a single circle on the input or output is the customary symbol for the voltage inverter.

We have four ways of performing the identity operation, two of which require pieces of wire and two of which require a voltage-inverting device. The mixed logician may use any of the four, as required in the circuit. The obvious choice is to use minimal hardware, and in most instances the wire will suffice. However, if the design calls for a change in the voltage representation with no change in logic, then the mixed logician has convenient ways to change the voltage representation. The use of a real device—the physical voltage inverter—to accomplish the logical identity operation is an immediate consequence of mixed-logic theory.

Now consider the logical AND operation, as developed in Figure 2–8. Two logic variables A and B are inputs to a box that must provide the logical AND of A and B as its output Y: $Y = A \bullet B$. When there are two inputs and one output, there are $2^3 = 8$ ways to select the voltage conventions. Eight voltage tables result from the eight assignments. Each voltage assignment produces its own mixed-logic diagram: the square box labeled "AND" shows the logic, and the circles show signals in which T = L. Since logical AND is of great importance to designers, we use the special AND shape instead of the box labeled "AND."



A	B	Y
F	F	F
F	T	F
T	F	F
T	T	T



A.H	B.H	Y.H
L	L	L
L	H	L
H	L	L
H	H	H

A.H	B.H	Y.L
L	L	H
L	H	H
H	L	H
H	H	L

A.H	B.L	Y.H
L	H	L
L	L	L
H	H	L
H	L	H

A.H	B.L	Y.L
L	H	H
L	L	H
H	H	H
H	L	L

A.L	B.H	Y.H
H	L	L
H	H	L
L	L	L
L	H	H

A.L	B.H	Y.L
H	L	H
H	H	H
L	L	H
L	H	L

A.L	B.L	Y.H
H	H	L
H	L	L
L	H	L
L	L	H

A.L	B.L	Y.L
H	H	H
H	L	H
L	H	H
L	L	L

Figure 2–8. Eight implementations of 2-input logical AND’s, and their common names

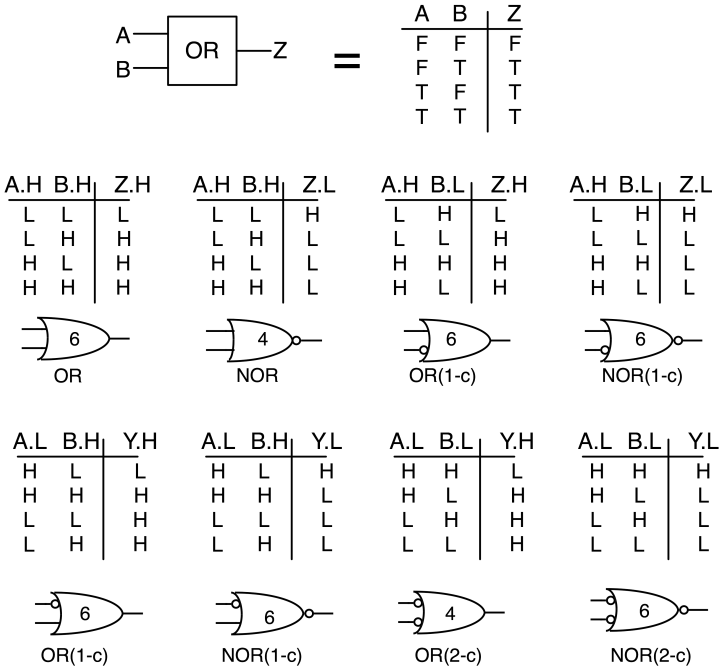
We may ask which of these eight realizations of logical AND correspond to readily synthesizable transistor circuits. While all of them can be implemented in Silicon, the NAND, NOR forms are by far the most common followed by the 2-input AND gate with T=H on both inputs and output and the 2-input OR gate with T=H everywhere. Mixed logic thus gives us four useful building blocks for realizing logical AND. The other four voltage tables in Figure 2–8 offer perfectly valid ways of performing logical AND, and some types of programmable gate arrays can implement them as primitives. None the less, for purely pedagogical reasons, we shall avoid using them in circuit diagrams meant for actual construction. (The two AND(1-c) devices are physically the same and just correspond to switching inputs; the same applies to the two NAND(1-c) devices).

The positive-logic convention allows only one choice for logical AND. Similarly, the negative-logic convention allows only one choice. Mixed logic offers four.

Now we have a slight problem: a mixed logician regards *all of these devices as AND’s*; but in our library of devices we need names that imply their *physical structure.* The solution is simple: all AND.L devices will be named NANDs, with the number of input circles in parenthesis, simarily, all AND.H devices will be named ANDs, with the number of input circles in parenthesis.

The number of transistors required to synthesize these ANDs in static CMOS technology is shown as the numerals 4 or 6 inside the logic symbol and is for reference purposes only—it should not appear on your circuit diagrams. 4-transistor structures will have one gate delay, 6-transistor structures 2 gate delays. All are readily synthesized but 4-transistor devices would be chosen wherever possible to conserve Silicon area and minimize device delay.

The logical OR operation gives analogous results. Figure 2–9 shows the treatment.



A	B	Z
F	F	F
F	T	T
T	F	T
T	T	T

A.H	B.H	Z.H
L	L	L
L	H	H
H	L	H
H	H	H

A.H	B.H	Z.L
L	L	H
L	H	L
H	L	L
H	H	L

A.H	B.L	Z.H
L	H	L
L	L	H
H	H	H
H	L	H

A.H	B.L	Z.L
L	H	H
L	L	L
H	H	L
H	L	L

A.L	B.H	Y.H
H	L	L
H	H	H
L	L	H
L	H	H

A.L	B.H	Y.L
H	L	H
H	H	L
L	L	L
L	H	L

A.L	B.L	Y.H
H	H	L
H	L	H
L	H	H
L	L	H

A.L	B.L	Y.L
H	H	H
H	L	L
L	H	L
L	L	L

Figure 2–9. Eight implementations of 2-input logical OR's, and their common names

Again, we have eight ways of representing voltage at the inputs and output, and each choice transforms the truth table for logical OR into a voltage table. The same four integrated circuits that perform logical AND will also perform logical OR, but the voltages representing truth are different.

The two OR(1-c) devices are physically the same and just correspond to switching inputs; the same applies to the two NOR(1-c) devices.

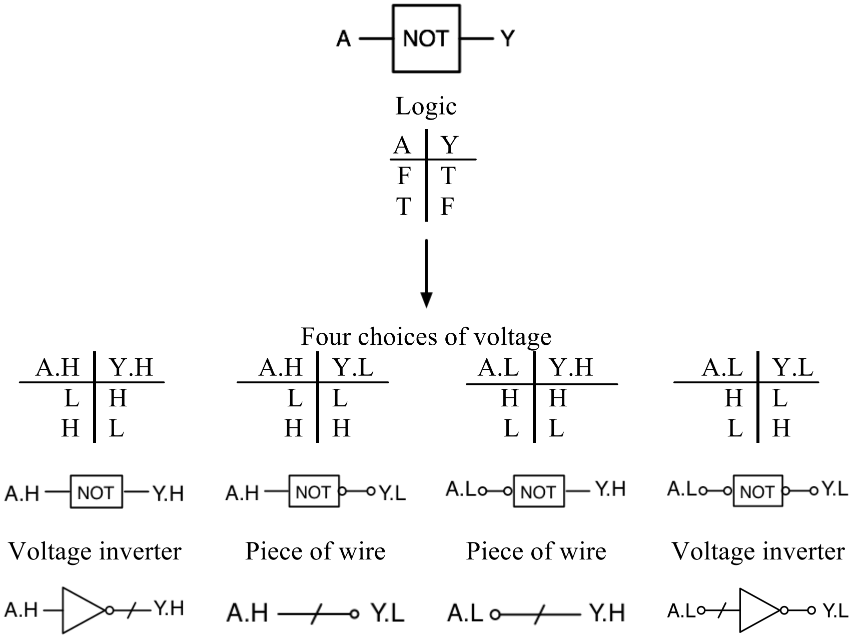
We have the same naming problem as above: a mixed logician regards *all of these devices as ORs*; but in our library of devices we need names that imply their *physical structure.* The solution is simple: all OR.H devices will be named ORs, with the number of input circles in parenthesis, simarily, all OR.L devices will be named NOR's, with the number of input circles in parenthesis.

The number of transistors required to synthesize these ORs in static CMOS technology is shown as the numerals 4 or 6 inside the logic symbol and is for reference purposes only—it should not appear on your circuit diagrams.

4-transistor structures will have one gate delay, 6-transistor structures 2 gate delays. All are readily synthesized but 4-transistor devices would be chosen wherever possible to conserve Silicon area and minimize device delay.

Your target technology, or simulator, will determine which of these devices are available. If the 6-transistor structures are not available they can be readily synthesized by adding just one inverter to the standard NAND-NOR 4 transistor structures—which is the tack we will take in this book. Some programmable gate arrays can implement all flavors without additional hardware and should be used if available.

Last, let's look at logic inversion. In Figure 2–10 the development is similar to that of the logical identity operator, but Y must equal NOT A. The truth table shows that for logical NOT the logical value of the variable must be inverted. There are four voltage realizations of this truth table: A.H and Y.H, A.H and Y.L, A.L and Y.H, and A.L and Y.L. Each gives rise to its own mixed-logic diagram, with the square box labeled "NOT" surrounded by appropriate circles to display the voltage representation of truth at the input and output.



A	Y
F	T
T	F

A.H	Y.H
L	H
H	L

A.H	Y.L
L	L
H	H

A.L	Y.H
H	H
L	L

A.L	Y.L
H	L
L	H

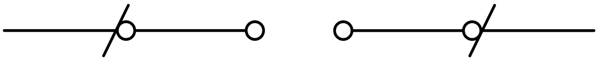
Figure 2–10. Realizations of the logical identity operation

The four voltage tables correspond, respectively, to a voltage-inverting device, a piece of wire, a piece of wire, and a voltage inverting device. This is interesting: the mixed logician may implement logical NOT with a piece of wire! You can see what is required to achieve this design: the voltage representing truth is different at the two ends of the wire.

The positive logician and the negative logician each have only one way to implement logical NOT—with a voltage inverter. The mixed logician potentially has four ways, two using a piece of hardware and two using just the wire connecting the input and output.

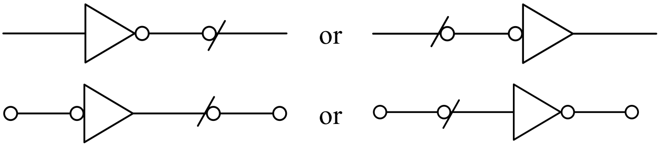
We get the logical identity "for free" along a wire, as long as the voltage representations are the same at each end of the wire. We get logical inversion "for free" along a wire if the voltage representations for truth are different at the two ends. Occasions arise when we wish to change the voltage representation of a signal without performing any logic or to perform logical NOT while maintaining the same voltage representation at the input and output. In digital design, we do not consciously perform logical identity operations; we assume that we have identity logic at any point in our circuit unless otherwise specified. On the other hand, we do indeed frequently perform logical NOT; it is one of our three important logical operators. Now we make an important choice: the mixed logician chooses to use the *voltage inverter* solely to generate the *logical identity*. This means that we will never insert a voltage inverter—a real device—into our circuit unless absolutely necessary. This choice also assures that the voltage inverter is never used for logical NOT. A corollary is that logical NOT is performed along a wire if and only if the voltages representing truth are different at the two ends.

We already have useful symbols for AND and OR; these correspond to real devices in a circuit. Since we generate logical NOT without a real device, we need a notation to display logical inversion. In a diagram, logical NOT appears as a slash along the line. In a logic diagram, the slash is necessary to display the logic. In a circuit diagram that shows both voltage and logic, the slash is not strictly necessary, since we may infer logic inversion from the difference in voltage representations at the two ends of the wire. However, we always include the slash in mixed-logic circuit diagrams. Formally, the slash shows the point at which the voltage representation changes, and so one side of the slash should have a circle representing T = L. On either side of the slash we have our usual identity operation: logical inversion occurs at the slash:



The exact point along the wire at which we put the slash is arbitrary; the designer chooses a convenient place.

When we need the same voltage representation for the input signal and its inverted output, we may insert a voltage inverter to the left or to the right of the slash:



In practice, we usually omit the T = L circle on the logic-inversion slash, since it conveys no information that cannot be gained by following the wires from the slash to the ends. In this book, we will use the slash without the circle except, in a few cases, for emphasis.

Once again, note carefully that the voltage-inverting device performs no change in logic; the same logic variable is implemented on both sides of the voltage inverter, but with different voltage polarities. The device inverts voltage, not logic. To avoid using the term “inverter,” which smacks of logic inversion, we sometimes refer to the voltage inverter as the “oops” function, implying that we are satisfied with our logic variable, but, “oops,” we need to switch voltage levels.

Let’s implement Z = X•Y, making input X available as signal X.L (i.e., T = L), and Y available with T = H.

Two implementation are shown in Figure 2–11, the first one assumes the availability of the AND(1-inv) function, which we would prefer if available. However, this gate is unlikely to be available as a primitive except in some programmable gate arrays. The other uses standard NAND-NOR gates along with a voltage inverter.

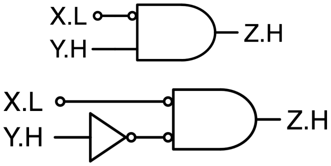


Figure 2–11.

Two implementations of Z = X•Y

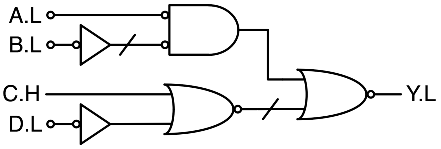
We inserted a voltage inverter to change signal Y.H into Y.L, the input required for our AND gate. You will find that adding voltage inverters where necessary requires no active thought; the little mixed-logic voltage-polarity circles direct you to do the right thing.

This is the entire mixed-logic theory. We have laid a firm foundation for achieving our original circuit-design goals. We have tools to realize logical AND, OR, and NOT. We have notations that show the voltage representing truth at every point in the circuit and that show each physical device. We are able to keep a strict separation of logic and voltage in our diagram—all the logic is there and all the voltage behavior is there. These mixed-logic notations and theory can be applied immediately to more complex digital building blocks. As we now turn to the analysis and synthesis of mixed-logic circuits, you will see that the mixed-logic methods allow the creation of a circuit from the original logic equation (synthesis) as well as the recovery of the original logic equation from the circuit (analysis), an advantage shared by no other method.

BUILDING AND READING MIXED-LOGIC CIRCUITS

Analyzing Mixed Logic Circuits

In setting the goals of our circuit design methods, we said that a good circuit diagram should present the logic in a way that allows the reader to retrieve the designer's original expression. Mixed logic fulfills this condition; analyzing a mixed-logic circuit is simple. Here is the prescription: ignore circles and inverters (since they perform no logic in themselves), interpret the slash as logical NOT and the AND and OR symbols as logical AND and OR, and read the original logic equation from the diagram. For instance, read the circuit in Figure 2–12 to recover the logic equation



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

Figure 2–12. Implementation of $Y = A \bullet \overline{B} + \overline{(C + D)}$

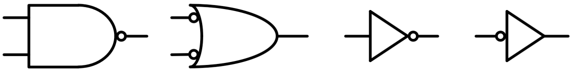
In words, the thought process is:

Y equals (A AND (B-NOT)) OR ((C OR D)-NOT)

If you feel the need for additional information on the diagram, insert the intermediate signal names and the missing circles on the slashes. You will quickly develop the skill to read a mixed-logic circuit without these extra legends.

Synthesizing Mixed-Logic Circuits

To implement a given logic equation, begin by sketching a picture of the equation, using the AND and OR logic symbols, and using the slash for logical NOT. Label the inputs and outputs with appropriate names of logic variables. At this point, the emphasis is on the logic, and it is this initial logic framework that allows us to analyze completed circuits with relative ease. Next, if the input or output variables have fixed voltage representations, add the appropriate .H, or .L and its companion circle. This step converts some of the logic variables into signals and fixes the voltage representation on some of the lines. You may assume that logic variables without a stated representation are available in either signal form, and you may use this flexibility in synthesizing the circuit. Usually, the devices available for implementing the AND and OR operations will be restricted. The tighter the restrictions, the easier the synthesis. For instance, if you are given only NAND and INVERTER gates, the synthesis is straightforward however complex the equations. There is no flexibility, since the only symbols available are:



With a wide choice of building blocks, we have more flexibility, and therefore more decisions. We wish to optimize the circuit. An obvious criterion for optimization is to minimize the number of voltage inverters, but other factors are often relevant. When a variety of gates is available, the designer may display virtuosity in developing a satisfying implementation of the original equation.

Producing a valid circuit design is easy; producing an aesthetic design is an art. Nevertheless, the general process is straightforward. Select a likely gate, insert the corresponding mixed-logic symbol by adding required circles to the logic symbol, add voltage inverters where needed to make the circles in the design conform to the requirements of the logic, and then move to a neighboring element. As the synthesis proceeds, you may notice that a different choice of gate in a previous step results in fewer inverters. Within reason, you would probably wish to backtrack and introduce the change. In a short time, the circuit will converge to an acceptable solution.

AND and OR functions of more than two variables can be handled either with multiinput gates or with gates with fewer inputs. For instance, either of the forms in Figure 2–13 describes a three-variable AND function. Design considerations dictate the choice and are technology dependent; in static CMOS, device physics limits inputs to 3 or 4—most progammable gate arays allow wider inputs. The original logic is not affected but the number of serial stages impacts circuit delay.

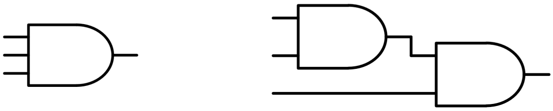
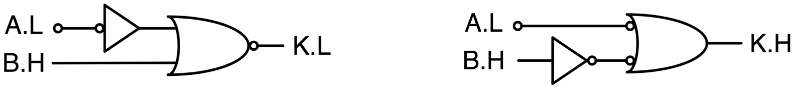


Figure 2–13. Realizing a 3-input AND with 2-input gates

Mixed-logic style. Experience in synthesizing mixed-logic circuits leads to concern about several points of style. We present a few points here and you will discover others as your mixed-logic skills grow.

Our habit of dropping the circle adjacent to the / operator is a matter of convention; you are free to accept or reject this convention. The slash is itself a convention, since we may infer the existence of logical NOT from a diagram of a voltage circuit without the slash. Nevertheless, we strongly recommend that you use the slash to indicate logical NOT; it adds greatly to the clarity of the diagram.

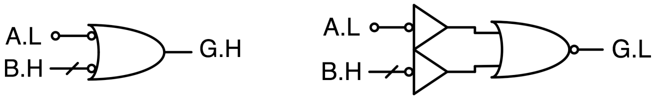
In dealing with voltage inverters and voltage polarities in mixed-logic syntheses, opportunities for decisions occur. Consider a design for K = A + B, with input signals A.L and B. H. Any implementation with readily available gates will have an inverter on one input. Figure 2–14 shows two designs. Which is better? The designer who will need signal K.H later would probably choose Figure 2–14b. In other situations, Figure 2–14a would be appropriate.



(a) (b)

Figure 2–14. Two circuits for K=A+B with signals A.L and B.H

The next illustration is more clear-cut. To realize $K = A + \overline{B}$, again with inputs A.L and B.H, a good circuit is (a)

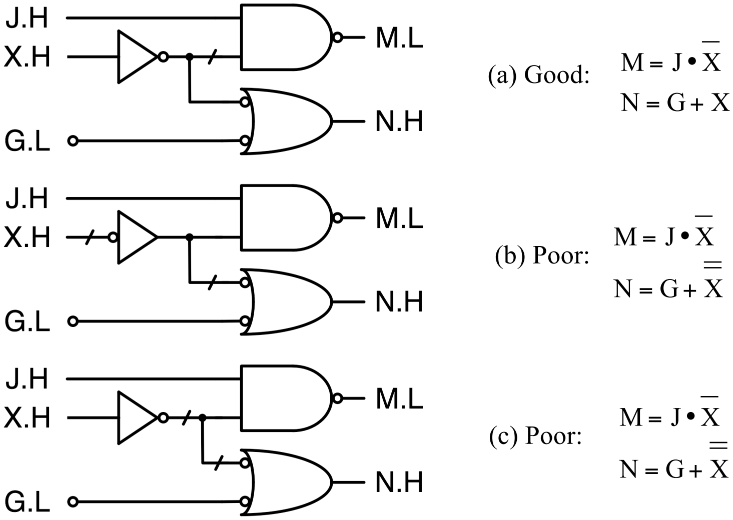


(a) (b)

Figure 2-15

The circuit has no inverters. Designs using more gates are inferior, even if they perform the same logic; for example, the (b) circuit has two inverters. Even if we desire G.L as the output in preference to G.H, the second circuit is still bad, since for any voltage signal we are always only *one* inverter away from its other voltage counterpart: we could produce G.L from the first circuit by adding an inverter to the output.

Be wary of subtly changing the logic with your drafting notations. Figure 2–16 shows three partial circuits, each an implementation of the equations $M = J\bullet \overline{X}$ and $N = G + X$. These circuits have identical wires and gates, and therefore must perform equivalent logic. But remember the guideline for digital drafting: the circuit must display the *original* logic. Only Figure 2–16a is a proper implementation of the original logic equations. It is unlikely that the double logic inversion of X in Figs. 2–16b and 2–16c would have survived earlier simplifications of the logic. We would not expect to see such a form presented for construction.



(a) Good: $M = J\bullet\overline{X}$, $N = G + X$

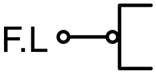
(b) Poor: $M = J\bullet\overline{X}$, $N = G + \overline{\overline{X}}$

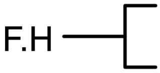
(c) Poor: $M = J\bullet\overline{X}$, $N = G + \overline{\overline{X}}$

Figure 2–16.
A good circuit and two poor ones for $M = J\bullet \overline{X}$ and $N = G + X$

We have found the following rule to be helpful in drafting logical NOT in complex logic circuits: Place the slash as far to the right as practical on its signal line.

Frequently, an input to a device will always be fixed at F or at T. This often occurs with the complex circuits described in Chapters 3 and 4, where you may wish never to enable a certain device feature, or always to enable it. Suppose that we wish to say "never do it"—we wish to make a signal permanently FALSE. Here are two mixed-logic representations for the F operation:

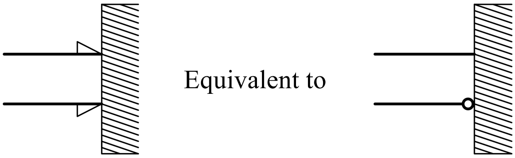




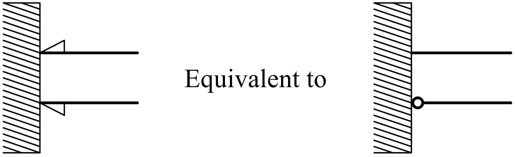
In this unchanging operation, the voltage on the wire never varies. For convenience, we sometimes show only the voltage level (H or L) on the diagram:

Other Mixed-Logic Notations

The presence or absence of the little circle in mixed logic shows the *voltage polarity* at each node in the diagram. Another popular mixed-logic notation for circuit diagrams is a small triangle to show the polarity of the voltage. A triangle lying above the line means T = H at that point; a triangle below the line means T = L. Figure 2–17 illustrates this notation.



(a) Inputs

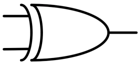
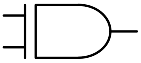


(b) outputs

Figure 2–17. An alternative mixed logic notation

MIXED LOGIC FOR OTHER LOGIC FUNCTIONS: EXCLUSIVE OR and COINCIDENCE

We have stressed AND, OR, and NOT as natural logic elements for designers. Are there other Boolean functions of two variables that are used intuitively in designs? Two more functions are of sufficient value to be included in our set of simple logic building blocks. These correspond to our concepts of "different" and "same." The EXCLUSIVE OR (XOR) logic function is true only if its two inputs have different logical values; one input is true while the other is false. The COINCIDENCE (XNOR) function is true only when both of its inputs are the same—both true or both false. Logic operator notations and drafting symbols for these logic operations are:

	Logic Symbol	Drafting Symbol
EXCLUSIVE OR (XOR)	$\oplus$	
COINCIDENCE (XNOR)	$\overline{\oplus}$	

These operators have the following truth tables:

		Logic	
A	B	$A\oplus B$	$A\overline{\oplus}B$
F	F	F	T
F	T	T	F
T	F	T	F
T	T	F	T

Observe that $\overline{\oplus}$ is the inverse of $\oplus$. The methods used in Chapter 1 yield Boolean equations for these functions in terms of the familiar AND, OR, and NOT operators:

$$A \oplus B = A \bullet \overline{B} + \overline{A} \bullet B$$

$$\overline{A \oplus B} = A \bullet B + \overline{A} \bullet \overline{B}$$

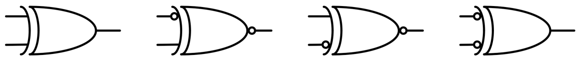
These are well-known and useful expansions of XOR and XNOR. Memorize them. In the evaluation hierarchy for logical operators, they fall below NOT and above AND.

The transistor circuits for XOR and XNOR are similar. If one of them is in your library of primitives, the other will likely be there also. If you are using a simulator to draft circuit diagrams these logic elements will almost certainly be in the library of primitives.

The XOR's voltage table is:

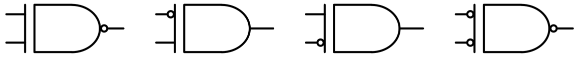
Inputs		Output
L	L	L
L	H	H
H	L	H
H	H	L

Our usual method of comparing a logic truth table with a device's voltage table produces an amazing result: the XOR voltage table yields four mixed-logic realizations for the EXCLUSIVE OR logic function! The drafting symbols are:



You should verify that substituting the indicated voltage values for the T and F in the truth table will in each case give a voltage table that is equivalent to that of the XOR.

This is not all—the XOR voltage table also gives four representations of the logical COINCIDENCE operator. Here are the symbols:



The reader should go through the same exercise with the XNOR voltage table and derive the equivalent logic drafting symbols

Notice the pattern: the XOR symbols have an even number of circles, whereas the XNOR symbols have an odd number of circles. This marvelous XOR gate gives eight building blocks for our drafting kit. We may use any of the four XOR symbols, depending on the requirements for our particular circuit, and still perform XOR logic. For instance, Figure 2–18 is a drawing of two syntheses of a logic equation involving XOR. With mixed logic, these designs arise naturally, with no mental effort wasted on logic-voltage interrelations.

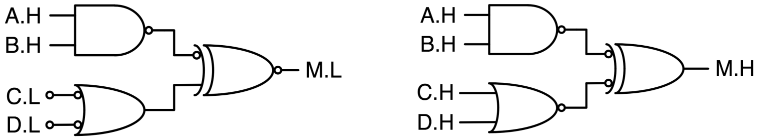


Figure 2–18.

Such elegant and useful results send a thrill of joy through the mixed logician. These eight easily remembered symbols allow us to produce XOR and XNOR in a simple, efficient manner. If your library also includes a device with the XNOR voltage table your tool kit will be further enhanced by another 8 drafting symbols; together these 16 symbols will serve as a basis for construcing almost any conceivale circuit involving the XOR, XNOR logical operators.

Output "Fights" – and Their Avoidance

Sometimes we wish to share a wire between several components, memory modules sharing a set of output data wires (the memory bus) for example. If one memory module is trying to drive a shared wire H, and another module trying to drive it L, we have an output "fight". Unpleasant things happen, things get hot, output transistors may burn up, and at the very least the wire will have some indeterminant voltage between H and L which will confuse downstream logic. Clearly this must be avoided at all costs.

Consider the abstraction in Figure 2–19 where we have 4 different memory modules (often called memory banks) sharing common wires: We need some way to take 3 modules "off line" leaving only one to drive the common wires at a time. Here we model the "off line" mechanism as a simple mechanical switch for illustrative purposes—electronic analogs are readily synthesized and are called Tri-State buffers or Tri-State switches. Tri, in this context means either H, L, or "not there"

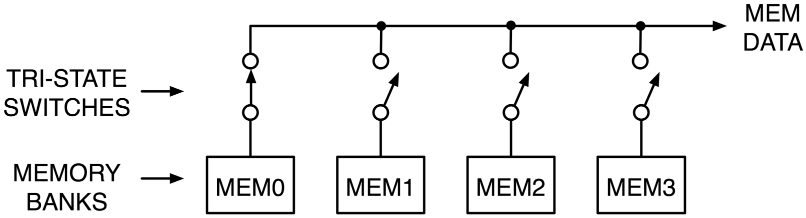


Figure 2–19. Memory modules sharing a common data bus

(The small solid dots on the data bus in Figure 2–28 is the standard notation for wires connected together).

Static CMOS uses P-MOSFET transistors to drive outputs H, and N-MOSFETs to drive outputs L. These transistors are called complements of each other, and CMOS logic (Complimentary MOS) always has one or the other type "on" to provide solid output drive to either H or L. If we could devise some way to have neither type "on" then the output would be undriven, or "floating" just what we need in a tri-state driver.

Careful here—who knows which switch should be turned on? For a CPU communicating with memory, it is obvious the CPU knows which memory bank it needs to access, and so the CPU must communicate to the memory subsystem which bank to turn on. This generalizes; whenever a receiver communicates with a tri-state generator of data, it is incumbent on the receiver to provide correct information to the generator to enable the proper tri-state switch. Further, it had better not turn on more than one, or a fight will ensue.

Tri-state switches or buffers are widely used in digital systems, and with the caveat that no more than one source can drive a wire at the same time, is a robust way of sharing wires, and wires are often in short supply in real hardware. The term buffer connotes a normal device, but with more robust drive capability to overcome the inevitable slowdown that occurs when driving peripheral wires.

The drafting symbols for individual tri-state buffers look like the standard inverter symbol, with the addition of a circle on the symbol's side showing the active voltage to turn on the tri-state switch. These are called "tri-state enables" and are almost always low active signals. Tri-state buffers come in both inverting and non-inverting styles. Figure 2–20 is a concrete example of 4 different 1-bit variables sharing a common wire which feeds down stream logic by means of non-inverting tri-state buffers.

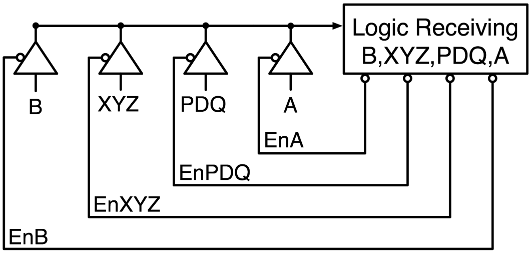


Figure 2–20.

What happens when none of the tri-state switches is turned on? You must avoid this at all costs since the wire is then "floating" and has an indeterminant voltage which will likely cause faulty behavior in the receiver. In chapter 3 we will discuss decoding circuits which will always avoid this problem.

Figure 2–20 shows what you must do to transform ordinary signals into the tri-state signals that can share a wire. Many complex logic elements, like memories, incorporate tri-state buffers as part of their internal structure. Such devices would be labeled "tri-state enabled", and would incorporate the tri-state enable circle as part of the device symbol.

At first glance you might try to map the circuit of Figure 2–20 into this boolean equation:

$$\text{WIRE} = \text{B} \bullet \text{EnB} + \text{XYZ} \bullet \text{EnXYZ} + \text{PDQ} \bullet \text{EnPDQ} + \text{A} \bullet \text{EnA}$$

but this interpretation would be wrong since the logical OR admits any, or all, of its terms to be simultaneously true whereas tri-state sharing demands only one be active at a time. The protocol is thus more akin to a high level CASE or SELECT construct.

When the Receiver Doesn't Know who is Sending Signals on a Shared Wire, (The Open-Drain Protocol)

Non-sense you say? It is really quite common when peripheral circuits share a wire feeding a signal to a CPU. If the peripherals are independent, it is possible for both of them to try simultaneous transmission. Imagine something like a mouse and track-ball driving a signal to a CPU. One would expect the user to be using one or the other, but not both; but what's to prevent someone from trying? The CPU has no control over the user and must protect against inadvertent, stupid, or malicious users. How to avoid a fight?

For two devices, A, B, whose outputs are wired together there are 4 possible results:

A	B	Wire =
L	L	L
L	H	**fight**
H	L	**fight**
H	H	H

If we could modify the output transistors to only drive outputs L, but never drive outputs H, we could always avoid a fight! We could then wire outputs together with impunity! All we need to do is figure out some independent way to provide a *default* H on the wire, independent of the devices trying to drive the line, but, a H that can be overpowered by any device driving the line L. A simple "pull-up" resistor does this nicely. Here, the resistor will "pull-up" the output to H for the default case when neither swich is overpowering it to L.

Here is a discrete switch model of the protocol; in this case the voltage truth table would be:

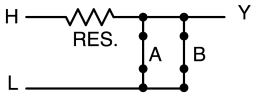
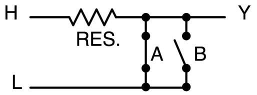
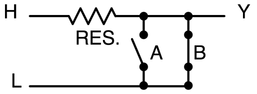
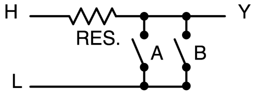
	A	B	Y
	L	L	L, (both A&B are overpowering the resistor's H)
	L	no drive	L, (A is overpowering the resistor's H)
	No drive	L	L, (B is overpowering the resistor's H)
	No drive	No drive	H, (default provided by resistor tied to H)

Figure 2–21. Modeling open-drain voltage table

Transistor analogs of the mechanical switches in figure 2-21 are readily constructed and are commonly called "open collector" devices. "open collector" is a holdover from the days of bipolar transistors; with the advent of CMOS a more appropriate name would be "open drain" and we will henceforth use that terminology.

As opposed to modules with built in tri-state buffers, like memories, open drain buffers are usually separate entities inserted between standard gates driving a shared output wire.

In your first exposure to hardware implementations, you need not delve into the details of open drain logic beyond knowing that a protocol exists for separate devices sharing a wire without output fights. If you blossom into a designer of real hardware devices you will want to re-visit this topic since it is the only way to solve a very common problem of disparate devices sharing a common resource.

Open-Drain Logic (Optional)

This section requires an elemenary understanding of CMOS logic and OHM's law

N-MOS transistors provide solid output drive to L and complimentary P-MOS transistors provide solid drive to H. In traditional static CMOS logic both types of transistors are present and provide solid drive to H or L.

Open-drain buffers have output drive circuits that are the exact electrical analog of the switches in fig. 2-21. The P transistors are simply not there—N transistors provide a solid drive to L, but if they are off the output is undriven, or "floating". the only difference between an open-drain buffer and a normal CMOS inverter is a larger N-FET to provide robust L drive and the absence of the P-transistor.

The resulting devices will be identified as open-drain, (open-collector), buffers in your device library and may be identified by the addition of a line on the output terminal. (Unfortunately the notation has not been standardized—we will use the line notation as in fig. 2-22).

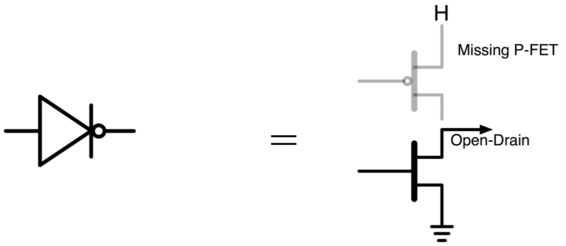


Figure 2–22. Drafting symbol for an open-drain buffer

Lets contrast Figure 2–20 with a superficially similar circuit using open-drain logic:

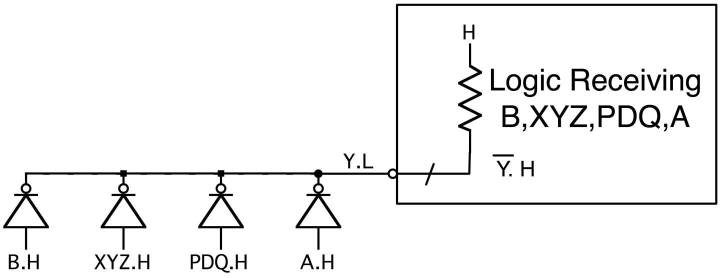


Figure 2–23. A circuit using open-drain logic

Several things:

(1) The receiver no longer has control of who's online; A, B, XYZ, PDQ are *always* online (if their inputs are true).

(2) Y is the OR of signals A, B, XYZ, PDQ. $Y = B + XYZ + PDQ + A$

(3) The OR happens just by wiring the outputs together, we call this a "Wired OR"

For the wired OR to work, all open drain drivers must drive the common wire to a level opposite to the default H generted by the resistor. Open drain buffers use NFET transistors, like the one shown in bold in figure 2-22, to perform this function "by pulling the common line L"

The resistor should be considered part of the receiver since it must be able to handle the case when external circuits are unplugged, in which case Y must revert to its default F value.

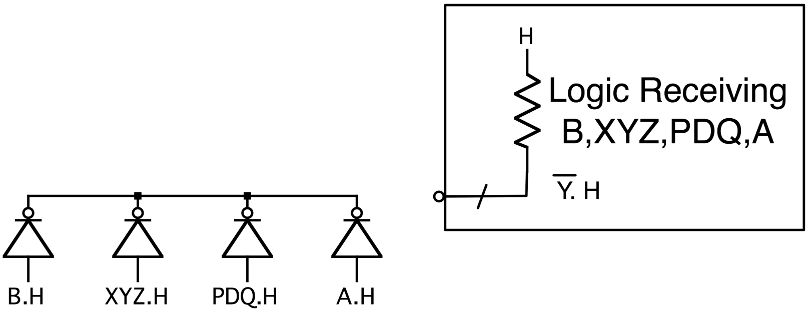


Figure 2–24. Disconnected open-drain logic

We would expect duality to rule when we change all inputs to T.L and in this case we have a "Wired AND". We leave this as grist for your mental mill.

The concept of a wired OR & AND has a non obvious virtue, it spatially separates the individual elements and distributes them along a wire. Spatial distribution is a requirement for efficient implementations of memory structures and we will revisit these concepts when we study memory circuits in the next chapter.

THE POSITIVE-LOGIC CONVENTION (OPTIONAL, and largely of historical interest)

If you are designing a circuit using schematic entry—the technique espoused in the chapter, mixed logic will be the method of choice. If you are using a modern high level design language, mixed logic optimizations will be implemented as a matter of course, "behind the scene". All this means that you are unlikely to encounter a positive logic circuit, but occasionally you may have to deal with one. Only in that case should you burden yourself with the material in this section.

The positive-logic convention has been widely used in the past in engineering practice. The convention is easy to learn, but this ease of learning is deceptive, because designing real circuits with positive logic requires clumsy, constricting rules and transformations.

When the positive-logic convention is used, logical TRUE is represented everywhere by a high voltage, and FALSE is represented by a low voltage. Under these conditions, the voltage table for what we have called a NOR gate can be transformed into the truth table for the logical NOR function:

Voltage				Positive Logic		
A	B	Y		A	B	Y
L	L	H		F	F	T
L	H	L	⇔	F	T	F
H	L	L		T	F	F
H	H	L		T	T	F

Surprise? I should think not, that’s why we called it a NOR gate! Similarly, the *voltage* tables for NAND, AND, and OR gate’s are obtained by assigning T=H for all entries in the *logical* truth tables NAND, AND, and OR, respectively. That's why we gave them these names.

Simarily, in positive logic, the voltage inverter always provides a logical inversion,

An application of De Morgan's law shows that the NAND function may be transformed into the inverted-input OR function:

$$\text{A }\textbf{NAND}\text{ B} = \overline{\text{A} \bullet \text{B}} = \overline{\text{A}} + \overline{\text{B}}$$

Similar transformations of NOR, AND, and OR produce inverted-input AND, inverted-input NOR, and inverted-input NAND logic functions, respectively.

In positive logic, since logic and voltage are tightly bound together, picking a physical gate is equivalent to picking a particular logic function. NAND and NOR gates arise naturally in most transistor-based technologies; fabricating AND and OR gates requires the insertion of voltage inverters. The positive logician, in dealing with NAND and NOR gates, implements NAND and NOR logic or their De Morgan counterparts.

In the customary notation for circuits based on positive logic, a small circle represents the *logical inversion* operation. Figure 2–25 shows positive-logic interpretations of some common gates.

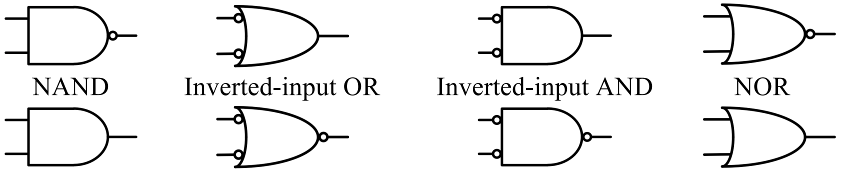


Figure 2–25. Interpretation of gates in the positive-logic convention

(The names lead directly to *both logic and voltage* truth tables in positive logic; in mixed logic we use these same names to signify voltage truth tables *but treat the symbols as pure AND’s and OR’s,* the circles defining what truth means for each symbol).

To realize a logic equation, the positive-logic designer must transform the logic, either algebraically or graphically, into a form that corresponds to the chosen gates; in the process, the flavor of the original logic equation vanishes. An algebraic approach involves repeated applications of De Morgan's law to recast the original logic expression into one using the positive logic supported by the chosen gates. Consider the following equation, to be rewritten so as to accommodate two-input NAND, NOR gates: $Y = A + B \bullet C \bullet \overline{D}$

We must transform this equation into one involving only NAND, NOR, or NOT operators by a sequence of tedious steps. Here is one such sequence:

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

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

$$Y = A + B \bullet \overline{(\overline{C} + D)} \qquad Y = A + B \bullet X \qquad \text{where } X = \overline{C} \text{ NOR } D$$

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

$$Y = A + \overline{(\overline{B} + \overline{X})} \qquad Y = A + Z \qquad \text{where } Z = \overline{B} \text{ NOR } \overline{X}$$

$$Y = \overline{\overline{A + Z}}$$

$$Y = \overline{\overline{A} \bullet \overline{Z}} \qquad Y = \overline{A} \text{ NAND } \overline{Z}$$

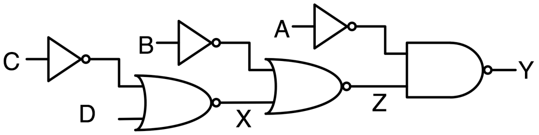


Figure 2–26. Positive logic implementation (??) of $Y = A + B \bullet C \bullet \overline{D}$, using DeMorgans law and 2-input NAND, NOR and NOT gates

Elegant? Hardly! Correct? Actually no; but we leave it as an exercise to uncover the flaw, which could either be in the sequence of logic transformations or in the mapping from equations to hardware.

To be fair, the plethora of inverters is partly due to restricting our library to 2-input NAND, NOR gates. So lets try a different set of transformations:

$$Y = A + B \bullet C \bullet \overline{D} \qquad \text{Eq. 2–1}$$

$$Y = \overline{\overline{A + B \bullet C \bullet \overline{D}}} \qquad \text{Eq. 2–2}$$

$$Y = \overline{\overline{A} \bullet (\overline{B \bullet C \bullet \overline{D}})} \qquad \text{Eq. 2–3}$$

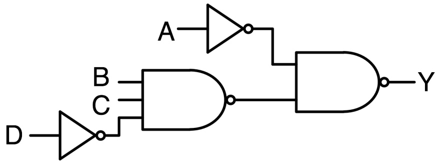


Figure 2–27. Positive logic implementation of $Y = A + B \bullet C \bullet \overline{D}$, using DeMorgans law and multiple-input NAND, NOR and NOT gates

Certainly a much better design, but still removed from the original equation. In fact it is not clear, from the logic diagram alone, which equation it does implement; probably 2–1 and not 2–2 or 2–3, but we have no way of knowing this.

Contrast this with a mixed-logic implementation of eq. 2–1. We start with a pure *logic* implementation of equation 2–1. Here we have *no* freedom; the placement of the logical symbols for AND, OR, NOT simply mirrors the heirarchy rules for those mathematical constructs.

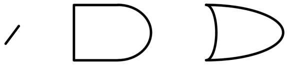


Figure 2–28.

To move from the logic realm to hardware we must specify what hardware library we wish to use. If restricted to NAND, NOT devices, Figure 2–20 immediately, and irrevocally, transforms to Figure 2–29 and from the diagram we can immediately infer the parent logic equation, and the equation immediately leads to the hardware.

This one-to-one correspondence is one of the beauties of mixed logic.

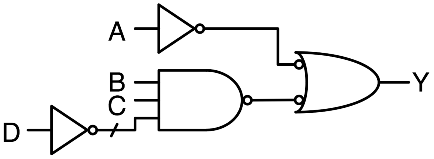


Figure 2–29. A mixed logic implementation of $Y = A + B \bullet C \bullet \overline{D}$

Digital-design textbooks in which positive-logic techniques are used contain prescriptions for drafting circuit diagrams using NAND and NOR gates without performing the algebraic transformations. There are separate sets of rules for using NAND gates alone, NOR gates alone, and certain combinations of NAND, NOR, AND, and OR gates. The prescriptions require starting with a sum-of-products or product-of-sums logic expression, thereby forcing upon the designer a particular form of logic expression *solely to achieve a hardware circuit.*

We will illustrate circuits based on positive logic using the simplest set of rules-those for pure NAND-gate synthesis of two-level circuits. (The term *level* refers to the maximum number of gates encountered from an input to the output.) It is assumed that multiple-input NAND gates are available. The rules are:

(1) Simplify the function as a sum of products.

(2) Draw a NAND gate for each product that has at least two variables. The variables form the inputs to the NAND gate. These NAND gates are the first level of gates.

(3) Draw a single NAND gate, using either the NAND or inverted-input OR graphic symbol at the second level. The inputs come from the firstlevel outputs.

(4) For a product consisting of a single variable, insert an inverter at the first level; alternatively, use the complement of the variable as an input to the second-level NAND gate.

Applying these rules to the preceding example produces the two-level result shown in Figure 2–27. Rules for other two-level syntheses are more complex in application. Positive-logic textbooks do not contain rules for building a circuit for a general multi-level logic expression using arbitrary choices of gates—the rules would be too cumbersome.

To the mixed logician, these techniques are obnoxious and unnecessary. With mixed logic, we can readily develop a circuit for any logic equation, of any complexity, using any desired gates, while preserving the original structure of the logic equation. In mixed logic, there are no special cases for different types of gates.

In subsequent chapters, where we encounter complex integrated circuits with various true-high and true-low inputs and outputs, the benefits of the mixed-logic notation are even more pronounced.

Reading Positive-Logic Circuit Diagrams

The mixed-logic concept of performing a logic operation (NOT) without a physical device, and the related concept of a physical device (the inverter) that performs no logic, evolve directly from our insistence that the original logic be visible in the circuit diagram. In the positive-logic and negative-logic conventions, where logic and voltage are rigidly tied together, the inverter performs *logic* inversion. We sometimes wish to extract a logic equation from a fixed-convention circuit. As we have shown, it is not possible to recover the original logic; the circuit shows only a transformed version of the original. Can a mixed logician read a positive logic circuit? Certainly; there are several approaches. One method is to read the positive-logic diagram directly, interpreting NAND gates as logical NAND (NOT AND), NOR gates as logical NOR (NOT OR), inverters as logical NOT, and so on. Then write a logic equation from the circuit. Using this approach, we write A NAND B as $\overline{A \bullet B}$, and A NOR B becomes $\overline{A + B}$. If you are unhappy with the number of logic inversions in the result, then apply De Morgan's law to try to eliminate some of them. Similar approaches allow us to read negative-logic diagrams.

As an example, consider Figure 2–30, a positive-logic-convention circuit. Following the prescription, we have

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

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

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

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

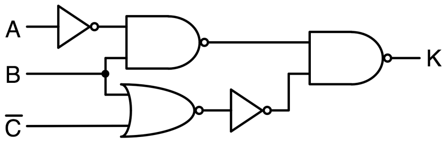


Figure 2–30. A circuit in the positive-logic convention. The circles represent logic inversions.

When do we stop manipulating the expressions for K? From the circuit in Figure 2–30, we don't know. All the above Boolean algebraic forms of K are legitimate, but we have no way to tell what the designer originally had in mind. We can be fairly sure it was *not* the first.

This method of reading positive-logic-convention diagrams produces a Boolean equation from the unmodified circuit diagram and transforms the equation into a more tractable form. Another method is to transform the positive-logic-convention circuit diagram into mixed logic, from which we read off an equation. This is a graphical method, whereas the first method is algebraic. For the graphical method, take the following steps. First, append .H to the positive logic inputs and outputs. If you wish, replace a negated input or output by the non-negated mixed-logic form. For example, an input $\overline{G}$ becomes a mixed logic $\overline{G}.H$, and you may express this as G.L with a circle on the input line. Next, wherever the circles do not match at the ends of a line, insert a slash to emphasize the implied logical NOT. Where a gate is surrounded by slashes, you may simplify the solution by altering the AND or OR gate symbol to its mixed-logic OR or AND counterpart. This is an application of De Morgan's law, and on the diagram the result is an inversion of circles and a change of the logic symbol to its dual. The circle inversions require rectification of the slashes on the gate input and output lines, leading to a simpler circuit. The process is shown in Figure 2–31. After this conversion to a mixed-logic circuit, reading the logic is simple.

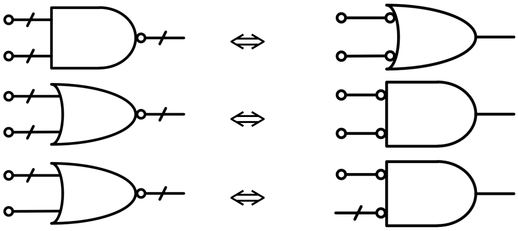


Figure 2–31. Some logic transformations on gates

In Figure 2–32 we depict a conversion of the previous example to mixed logic. The result is the same as the last equation in Figure 2–30. Again, we cannot be sure if this is the designer's original equation, since Figure 2–32 does not preserve the original equation.

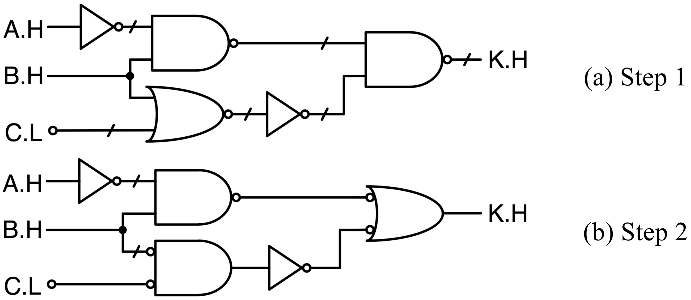


Figure 2–32. Converting Fig 2–30 to mixed logic

We might investigate how a mixed logician would handle the original synthesis problem: $K = \overline{A} \bullet B + \overline{B} \bullet C$. In a mixed-logic synthesis for this equation, one choice for inputs is A.H, B.H, and C.L, in accordance with Figure 2–32, and we might assume that the form of output K is unspecified. Figure 2–33 shows the resulting mixed-logic circuit. Because we permit K to appear in either signal form, Figure 2–33 contains one less inverter than Figure 2–30. Whether this ends up saving a gate will be determined by the needs of the larger design. We may require only signal K.H later, and thus the saved inverter would reappear to convert K.L to K.H. However, the mixed logician is not preoccupied with voltages; the drafting conventions handle voltages automatically. The mixed-logic method creates the opportunity for saving gates.

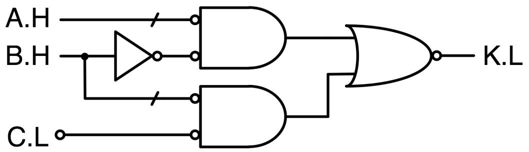


Figure 2–33. A mixed-logic circuit for $K = \overline{A} \bullet B + \overline{B} \bullet C$

In our experience, for a given logic expression, mixed logic always produces a result that is at least as economical of hardware as other systems. Mixed-logic circuits often save hardware while preserving the original logic in the diagram.

We close the discussion of other drafting conventions with a warning. In some parts of the electrical engineering community, "logic 1" and "logic 0" don't refer to *logic* at all, instead they refer to a *high-voltage* level and a *low-voltage* level, respectively. This is an old use of the word *logic,* originating in the early days of digital circuits to distinguish a voltage level (a range of voltage) from an exact value of voltage. The terminology illustrates the confusion that arises when logic values and voltage levels are not kept as separate concepts. This jargon is giving way to more modern concepts of separate logic and voltage, but you will sometimes encounter the old usage, so be alert when you deal with existing documentation and when you talk with other hardware designers.

READINGS AND SOURCES

FLETCHER, WILLIAM *I., An Engineering Approach to Digital Design.* Prentice-Hall, Englewood Cliffs, N.J., 1980. Uses mixed logic.

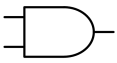
MALEY, G. A., and J. EARLE, *The Logic Design of Transistor Digital Computers.* PrenticeHall, Englewood Cliffs, N.J., 1963. An influential work; look here (for example, at page 114) to see how difficult design was before the advent of mixed logic.

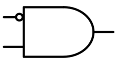
PROSSER, FRANKLIN, and DAVID WINKEL, "Mixed logic leads to maximum clarity with minimum hardware," *Computer Design,* May 1977, page 111.

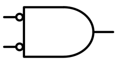
EXERCISES

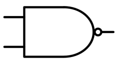
(A simulator, such as Logic Works, with circuit drafting software will be a helpful adjunct in working these problems)

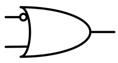
2-1. Write the voltage tables for the following devices:

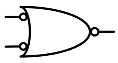

(a)


(b)


(c)


(d)


(e)


(f)

2-2. What is positive logic? Negative logic? What is the positive-logic convention? The negative-logic convention? The mixed-logic convention?

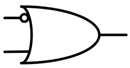
2-3. Explain carefully the meaning of these mixed-logic notations:

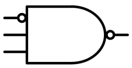
LOADH
LOADH. H
LOADH. L

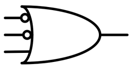
2-4. What is the difference in logic performed by the signals XYZ.H and XYZ.L?

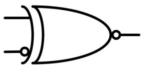
2-5. True or false: The H notation means that the voltage used to represent a given logic variable is high.

2-6. Give the dual of each of these mixed-logic symbols:


(a)


(b)

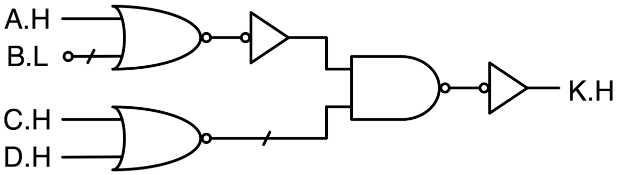

(c)


(d)

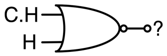
2-7. Why do we wish to maintain a strict separation of the concepts of logic and voltage?

2-8. How does a mixed logician denote logical NOT? What device is used?

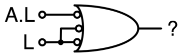
2-9. Fill in all intermediate signal names on this mixed-logic circuit diagram:



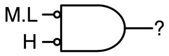
2-10. What logic does each of these circuit elements perform?



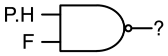
(a)



(b)



(c)

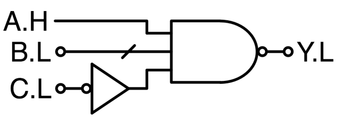


(d)

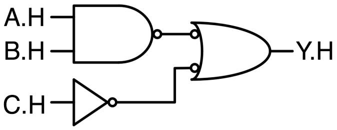
2-11. Explain the difference between synthesis and analysis.

2-12. State the prescription for analyzing mixed-logic circuits.

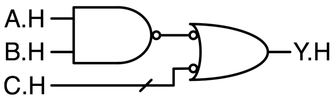
2-13. Analyze these circuits to produce the equations for the outputs:



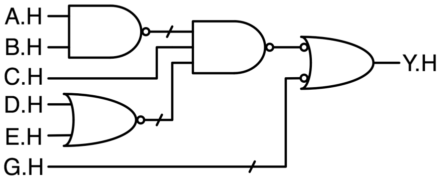
(a)



(b)



(c)



(d)

2-14. Synthesize the following equations using only NAND, NOR or INVERTER gates. The inputs are generated elsewhere as A.L, B.H, C.L, D.L, and E.H. The desired voltage representations for truth on the output is given for each equation.

(a) $Y = A \bullet B + C \bullet D$ Y.H

(b) $Y = (A + D) \bullet \overline{(B + E)}$ Y.L

(c) $Y = A + C + B + E$ Y.L

(d) $Y = \overline{A} \bullet B \bullet C + B \bullet E$ Y.H

(e) $Y = A \bullet D$ Y.L

2-15. Synthesize equations (a…e) using any gate in the Logic Works simulation gate library (or their DeMorgan equivalents). If the DeMorgan symbol is not in the Logic Works simulation gate library, make the equivalent symbol and use it in your schematic. The input polatities are as given in 2–14. Let the output signal convention for each circuit be the natural one arising from your synthesis.

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

(b) $Y = \overline{B} \bullet C \bullet D + \overline{E \bullet B}$

(c) $Y = \overline{(A + B + C)} \bullet (\overline{D} + E)$

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

(e) $Y = A + B + C + D + E$

For problems f and g, use any gate in the Logic Works simulation gate library (or their DeMorgan equivalents) *except* the Logic Works XNOR. You may have to use one of the 8 mixed logic equivalent forms of the XOR logic given in the text. If so, make corresponding symbols and use them.

(f) $Y = (A \bullet B + \overline{C}) \bullet (D \oplus E)$

(g) $Y = (A \bullet B + \overline{C}) \bullet (D \mathbin{\overline{\oplus}} E)$

2-18. Consider the function X of Exercise 1-33. X is true only when two 2-bit binary numbers A1,A0 and B1,B0 are different

(a) Working directly from the definition of X, write a logic equation for X making use of the XOR operator.

(b) Working directly from the definition of X, write a logic equation for $\overline{X}$ making use of the COINCIDENCE operator.

(c) Show how you may use DeMorgan's law to transform the expression for X in part (a) into the expression for $\overline{X}$ in part (b)

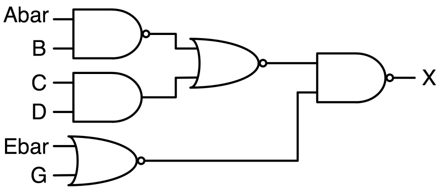
(d) Show the equivalence of your results for parts a) and (b) to the results of exercise 1-33

2-19. Derive a logic expression that is true if one positive 5-bit number A is less than another similar number B.

2-20. Many cars have an alarm buzzer that warns of unfastened seat belts, lights left on, and the key left in the ignition. The system might operate as follows: The alarm should sound if the driver's seat belt is not fastened when the motor is running, or if the passenger seat is occupied and the passenger's seat belt is not fastened when the motor is running, or if the lights are on when the key is not in the ignition switch, or if the key is in the ignition switch when the motor is not running and the driver's door is open. Assign a meaningful name to each variable in the statement, and write a logic equation for the alarm buzzer's control signal. You may find it useful to break the statement of the buzzer's behavior into several statements and combine these statements to produce your final result.

2-21. You are to design a 5-input circuit, 4 inputs of which form the binary representation of a decimal digit (in other words, the BCD code A,B,C,D for a digit 0 through 9). The fifth line is a control signal CL. If the control signal is false, the single output of the circuit should be true only if the decimal input number is 4 or greater. If CL is true, the output should be the inverse of input bit B. You should be able to write an equation for the output without writing the large (32-row) truth table. Express the equation in a circuit, using gates of your choice.

2-22. Here is a circuit with the *positive-logic* convention



(a) Write a logic equation directly from the diagram

(b) Convert the diagram to mixed-logic notation.

(c) Analyze the mixed-logic diagram, and show the equivalence of the result with that obtained in part (a). Design a circuit embodying OUT = A•B•C + D + E, using only the open-drain inverter. You may choose the voltage representations of truth at the inputs and outputs.

2-23. At one time, designers developed a complete logic family of inverters, AND and OR gates, and so on, based on fluid flow. These devices were proposed to immunize military devices from nuclear radiation. It was also suggested that fluid-flow devices might drive mechanical devices such as printers and card readers directly with fluid logic, bypassing the expensive conversion of electronic digital signals to mechanical control signals. The scheme is now only of historical interest, but it is interesting to contemplate how this class of devices might fit in with modern design methods. Suppose you are given a family of fluid flow devices and are asked to build a digital system using them.

(a) What definition of fluid flow (either at rest or moving) would you choose to represent logic truth?

(b) Could you still use our standard drafting symbols to represent designs with fluid logic elements?

(c) Would there still be little circles? If so, what would they mean?

(d) On our standard logic diagrams, a line represents a wire or an electrical path. What would a line represent in a fluid-logic circuit?