8

DIGITAL DESIGN BUILDING BLOCKS AND MORE ADVANCED COMBINATIONAL CIRCUITS

8.1 COMBINATIONAL CIRCUITS WITH MORE THAN ONE OUTPUT

Not all combinational circuits have a single output, like it was presented throughout most of Chapter 7. As a matter of fact, many applications have multiple outputs. Let us examine this with an interesting example.

Example 8.1 Design a combinational circuit to decode a four-bit *BCD* number that drives segments of a seven-segment *LED* display. The display must light up showing the corresponding BCD number presented at the input of the decoder. Figure 8.1a depicts a seven-segment LED display. Each segment has been labeled with the letters *a* through *g*. The display has seven inputs, one per segment. Assume that a *high-level* voltage presented at the input of a segment turns such segment *ON*; else when a *low-level* voltage is presented, the segment is *OFF*. When we want the display to show the number *0*, we must ensure to apply high-levels or one's to segments: *a, b, c, d, e,* and *f,* while we need to present a zero to segment *g.* Figure 8.1b shows the wiring of a single segment, and Figure 8.1c shows the schematic representation where the LED is shown with its corresponding symbol. Both (b) and (c) show the current limiting resistor that is placed in-series with the LED so that the appropriate current makes the LED shine when turned ON as the manufacturer specifies.

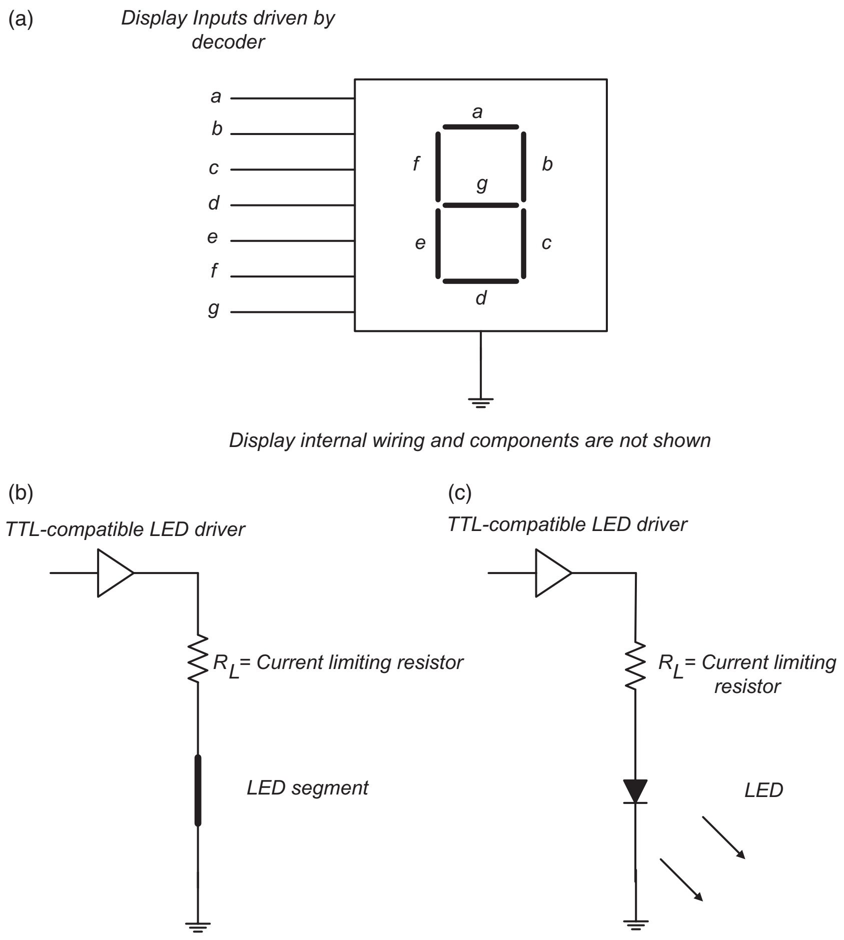

(a)
Display Inputs driven by decoder
a
b
c
d
e
f
g
a
f
b
g
e
c
d
Display internal wiring and components are not shown
(b)
TTL-compatible LED driver
R_L= Current limiting resistor
LED segment
(c)
TTL-compatible LED driver
R_L= Current limiting resistor
LED

Figure 8.1 (a) Seven-segment display assembly; (b) detailed wiring and connectivity of one segment; (c) detailed wiring as shown in part (b) with the segment replaced with a LED schematic symbol.

The LED driver, LED the current limiting resistor, and the LED segment are all assumed to be part of the display assembly. In addition to the seven LED segments, the assembly contains seven drivers and seven resistors.

Brief Calculation of the Current Limiting Resistor

Assume that the current through the LED for the intended typical luminous intensity required by the manufacturer is 10 mA. The manufacturer also specifies a maximum forward voltage drop. This is $V_{DROPMax} = 2.0$ V. Moreover, assume that our LED driver drives *TTL*-compatible voltage levels. Since the minimum voltage at the output of the driver is $V_{OHMin} = 2.4$ V sourcing a current of 10 mA, the current limiting resistor value is calculated as follows:

$$R_L = \frac{V_{OHMin} - V_{DROPMax}}{0.010} = \frac{2.4 - 2.0}{0.010} = 40\ \Omega, \tag{8.1}$$

where $V_{OHMin} = 2.4$ V is dictated by the driver TTL compatibility. The driver must be selected so that it can source at least 10 mA. A driver of somewhat higher current source capability may also be selected to do the job. Ultimately, the series resistor will limit the current needed by each segment.

Let us quickly check the amount of power that the resistor will dissipate.

$$P_R = I_{LED}^2 \times R_L. \tag{8.2}$$

Since $I_{LED} = 0.01$ A and R = 40 Ω, thus:

$$P_R = (0.01)^2 \times 40 = 4\ \text{mW}. \tag{8.3}$$

Since some resistors available can handle 1/16 W (62.5 mW) we can use a 1/16 W-rated resistor. The above analysis does not take into consideration variations of LED current and LED voltage forward drop, resistor variability, power supply changes, and temperature changes. The intent of the above calculation is to provide the reader with the basics to calculate the current limiting resistor value.

Figure 8.2 depicts the LED assembly driven by the BCD-to-seven segment decoder that we need to design. The inputs to the decoder are assumed to be BCD *0000, 0001* through *1001*; the other six binary combinations *(1010–1111)* will be assumed not to be present as decoder inputs.

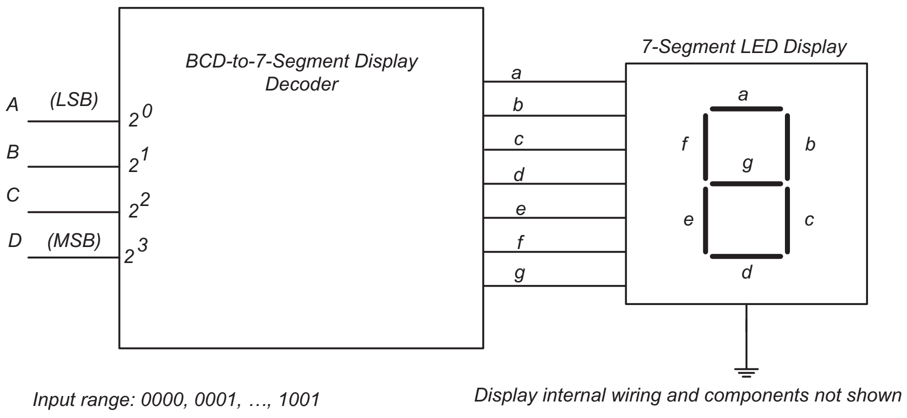

BCD-to-7-Segment Display Decoder
A (LSB)
B
C
D (MSB)
2^0
2^1
2^2
2^3
a
b
c
d
e
f
g
7-Segment LED Display
a
b
c
d
e
f
g
Input range: 0000, 0001, ..., 1001
Display internal wiring and components not shown

Figure 8.2 Decoder driving a seven-segment LED display.

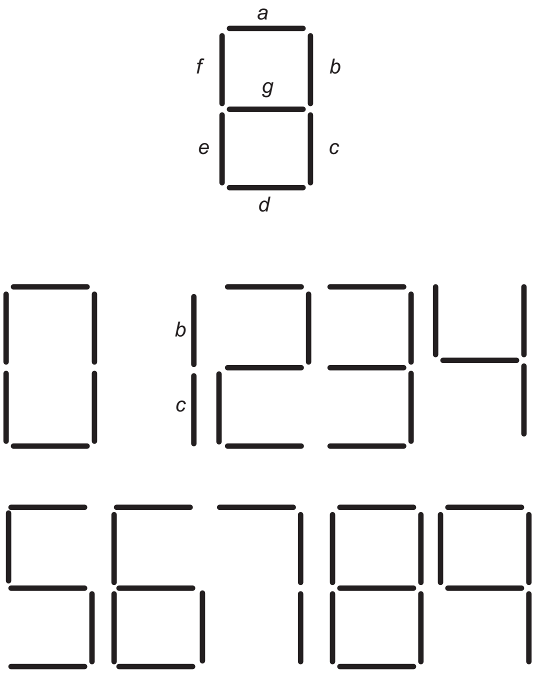


Figure 8.3 Seven-segment LED display segments to numerical mappings.

We will assume that we want the numbers *0* through *9* displayed as depicted by Figure 8.3. Additionally, let us remember that a *high-level voltage* turns a segment ON, while a *low-level* voltage turns it OFF.

At this point we are ready to start working on this example's truth table, which is presented in Table 8.1.

Table 8.1 contains the BCD number bits (*D, C, B, A*) on the four left-hand side columns. Clearly ***A*** is the least significant bit (*LSB*). The columns for each segment are labeled as ***a***, ***b***, and so forth. It is very convenient and important to observe that since the last six binary combinations *1010* through *1111* are not present, because the input number is by definition a BCD number which only spans *0000* through *1001*, it works out to our advantage to place *don't care* conditions (*X's*). So what needs to be done to find a simplified *SOP* forms for each the seven segments? Proceeding we obtain the following seven *K.* maps, depicted by Figure 8.4a through g.

The maximally simplified SOP forms for every segment are given below:

$$\text{Segment a}: a(D, C, B, A) = D + B + C.A + \overline{C}.\overline{A} \tag{8.4}$$

$$\text{Segment b}: b(D, C, B, A) = \overline{C} + D + \overline{B}.\overline{A} + B.A \tag{8.5}$$

$$\text{Segment c}: c(D, C, B, A) = D + C + \overline{B} + A \tag{8.6}$$

Table 8.1 Truth table for Example 8.1, BCD-to-seven-segment decoder

BCD Input bits D: MSB, A: LSB				Outputs to Segments							
D	C	B	A	a	b	c	d	e	f	g	Displays Number
0	0	0	0	1	1	1	1	1	1	0	0
0	0	0	1	0	1	1	0	0	0	0	1
0	0	1	0	1	1	0	1	1	0	1	2
0	0	1	1	1	1	1	1	0	0	1	3
0	1	0	0	0	1	1	0	0	1	1	4
0	1	0	1	1	0	1	1	0	1	1	5
0	1	1	0	1	0	1	1	1	1	1	6
0	1	1	1	1	1	1	0	0	0	0	7
1	0	0	0	1	1	1	1	1	1	1	8
1	0	0	1	1	1	1	0	0	1	1	9
1	0	1	0	X	X	X	X	X	X	X	–
1	0	1	1	X	X	X	X	X	X	X	–
1	1	0	0	X	X	X	X	X	X	X	–
1	1	0	1	X	X	X	X	X	X	X	–
1	1	1	0	X	X	X	X	X	X	X	–
1	1	1	1	X	X	X	X	X	X	X	–

$$\text{Segment d}: d(D, C, B, A) = C.\overline{B}.A + \overline{C}.\overline{A} + \overline{C}.B + B.\overline{A} \tag{8.7}$$

$$\text{Segment e}: e(D, C, B, A) = \overline{C}.\overline{A} + B.\overline{A} \tag{8.8}$$

$$\text{Segment f}: f(D, C, B, A) = C.\overline{A} + C.\overline{B} + \overline{B}.\overline{A} + D \tag{8.9}$$

$$\text{Segment g}: g(D, C, B, A) = \overline{C}.B + B\overline{A} + D + C\overline{B}. \tag{8.10}$$

Each of the seven output functions (*a* through *g*) depends on the same four independent binary variables *A, B, C,* and *D*. Some of the functions have repeated terms, for example, taking a close look at Equations (8.4), (8.7), and (8.8) we see that they have a common term $\overline{C}.\overline{A}$ among them. When we do the logic implementation of functions (a) through (g) we only need to generate the term $\overline{C}.\overline{A}$ once, then feed it into Equations (8.4), (8.7), and (8.8). Before getting into the logic implementation of our seven functions let us identify all other repeated terms. These are: $\overline{B}.\overline{A}$ present in Equations (8.5) and (8.9) and term $\overline{C}.B$ present in Equations (8.7) and (8.10), and term $\boldsymbol{C}.\overline{B}$ present in Equations (8.9) and (8.10).

Following Equations (8.4) through (8.10) these are implemented with logic gates in Figure 8.5a through g. But we are not done yet. Each of the segment functions *a, b,* through *g,* is simplified *SOP* forms in a stand-alone sense. However, since we are implementing all seven functions, which are all functions of input variables *A, B, C,* and *D* there are few other things that we can do in order to reduce the number of logic gates that we use. First by inspection

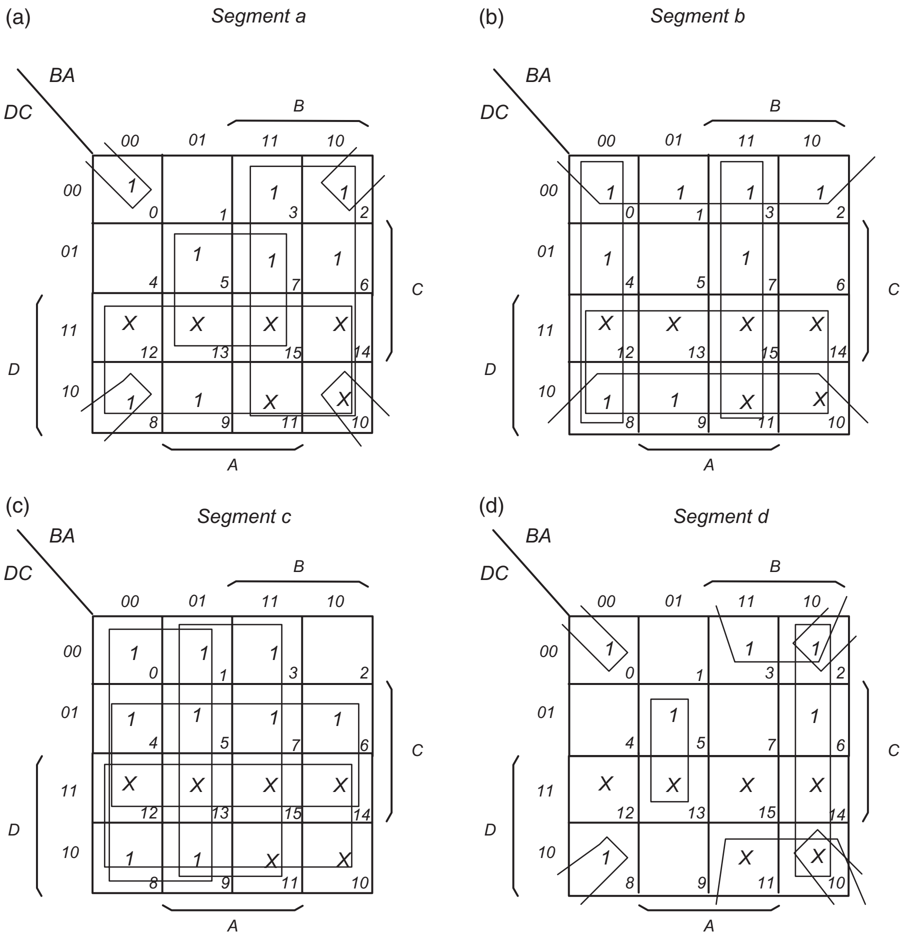


Figure 8.4 Illustration of segments a through g.

of Figures 8.5a through g we can see that each function uses some subset of the variables *A, B, C,* and *D* and their complements ($\overline{A}$, $\overline{B}$, $\overline{C}$, $\overline{D}$); this means that once we have *A, B, C,* and *D* and generate their complements once, using four inverters, the variables, and their complements can be connected to each of the functions that require them. For example, referring to Figure 8.5a, we note that $\overline{C}$ is used in the lower *AND* gate. Additionally, $\overline{C}$ is used in Figure 8.5b as an input to the four-input *OR* gate; thus we do not need to use a second inverter to generate $\overline{C}$ again. The same applies to other uses of $\overline{C}$ throughout the rest of the segment functions. Finally. the above is true for all input variables and their complements.

We can still reduce the number of logic gates a little more. Looking further at Figure 8.5a, note that the term $\overline{C}.\overline{A}$ is the fourth input of the *OR* gate for

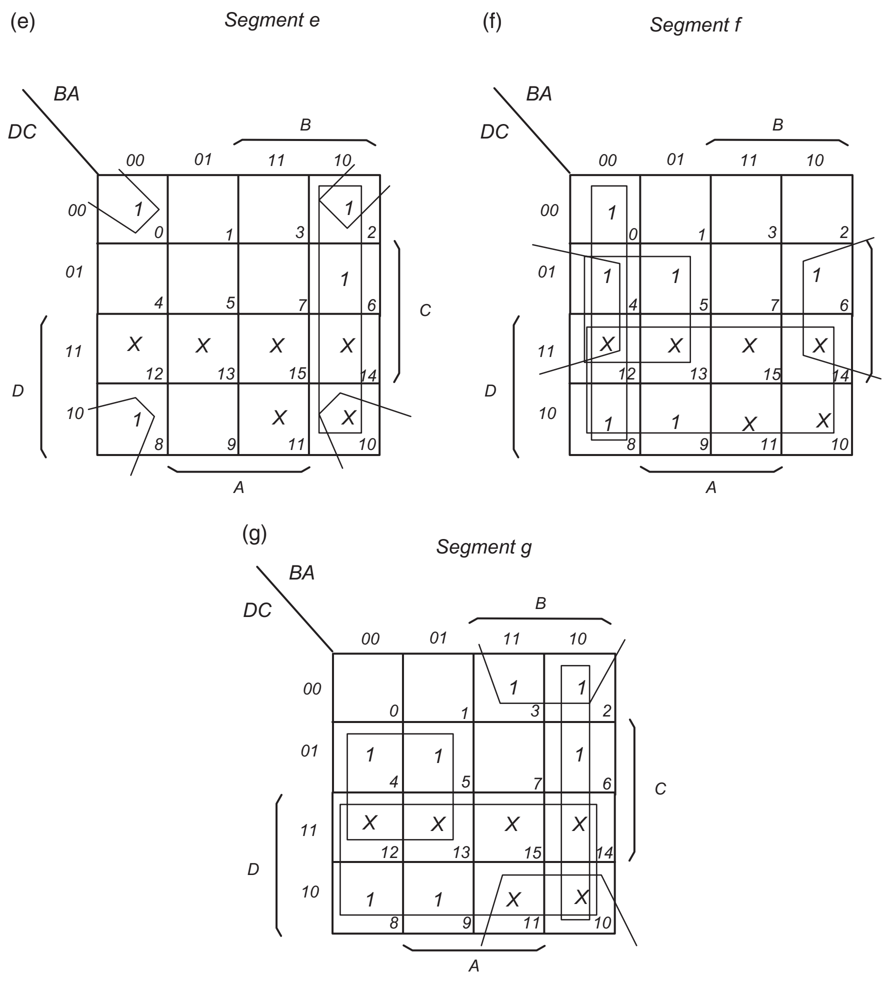


Figure 8.4 (*Continued*)

segment *a*. Term $\overline{C}.\overline{A}$ can also be found as the third input to the *OR* gate of Figure 8.5d and the first *OR* input of the *OR* gate of Figure 8.5e. What does this mean? It means that we do not to repeat the *AND*-ing logic that creates three different $\overline{C}.\overline{A}$ terms in (a), (d), and (e). We actually need just one *AND* gate that produces $\overline{C}.\overline{A}$, and this term is fed to all other users of the $\overline{C}.\overline{A}$ term. This saves us two *AND* gates. Something very similar occurs with terms $\overline{B}.\overline{A}$ and $\overline{C}.B$.

Exercise: Redraw the circuits of Figure 8.5a through g reducing the logic gates by: deleting repetitive logic terms produced by the *AND* gates.

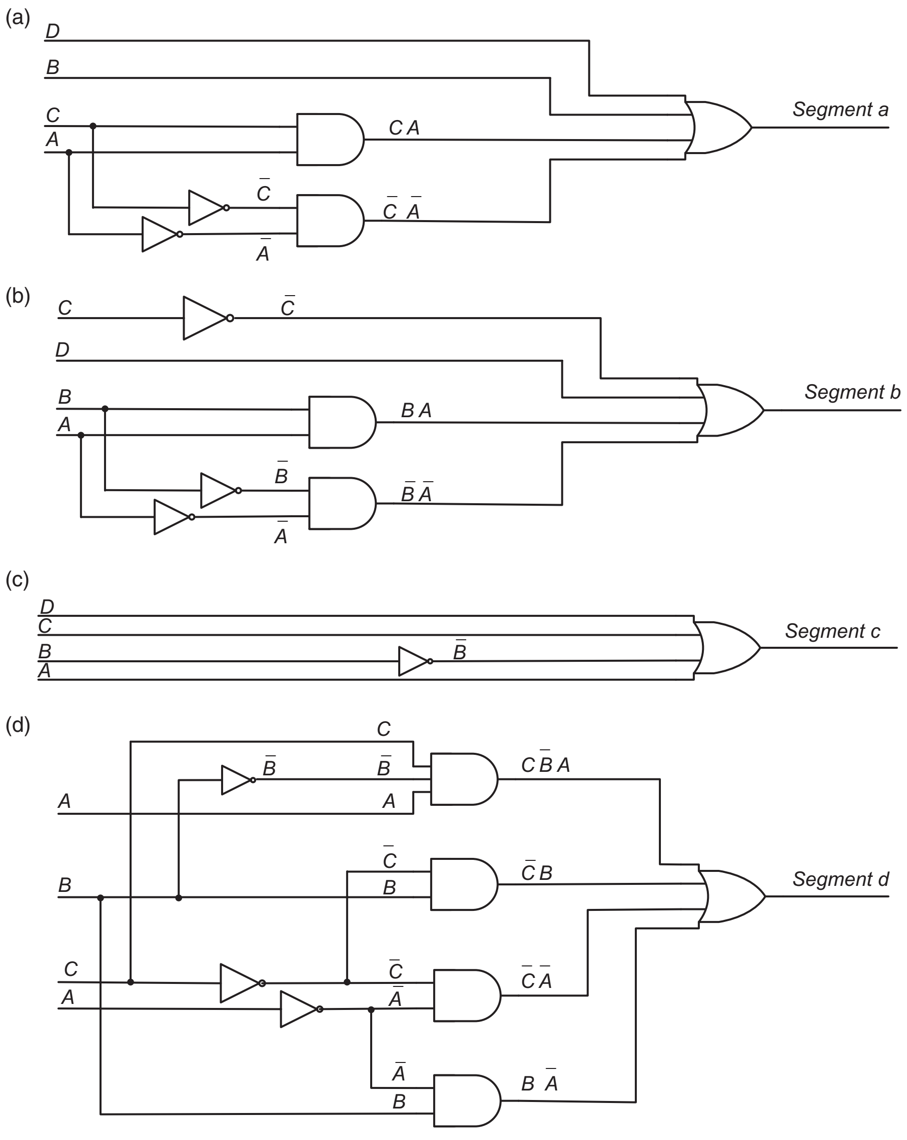


Figure 8.5 Seven-segment decoder logic implementation for segments a through g.

8.2 DECODERS AND ENCODERS

Decoders and encoders are combinational logic circuits. A binary decoder is a digital circuit that has *n* binary inputs and 2^n outputs. For example, a decoder with three inputs produces eight outputs; this decoder is referred to as a *3-to-8* decoder. Let us assume that the outputs are *active high* or *high-true* signals;

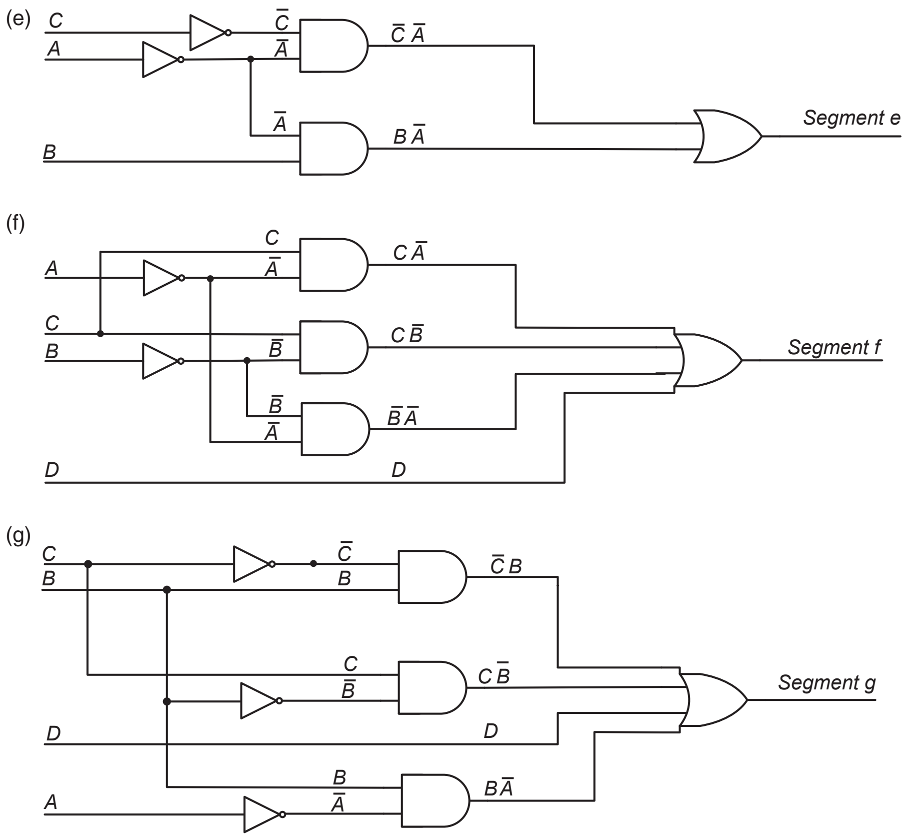


Figure 8.5 (*Continued*)

this means that an asserted signal is interpreted as a high level and this high level is a *one.* Conversely, an inactive or negated output is a low output and such low is a *zero*. Table 8.2 depicts the truth table for such a decoder. The *LSB* input is named *A*, while the *MSB* is named *C*. As expected, note that the three inputs span a total of $2^3 = 8$ binary combinations, starting at *000* through *111*. Each of its 8 outputs is associated with each one of the eight binary combinations. In such way that input *000* is associated with Y_0, input *001* is associated with Y_1 and so on.

Each output $Y_0, \ldots, Y_7$ is respectively associated to its output *000, . . . , 111.* Outputs are asserted in a mutually exclusive fashion, that is, one at a time.

By inspection of Table 8.2 we see that if the input code to the decoder is *100*, output Y_4 is *1* while all other outputs are *zero*. The truth table of our decoder has a fourth input that provides a master enable to the component. When the enable is high, the decoder works as we already described. When

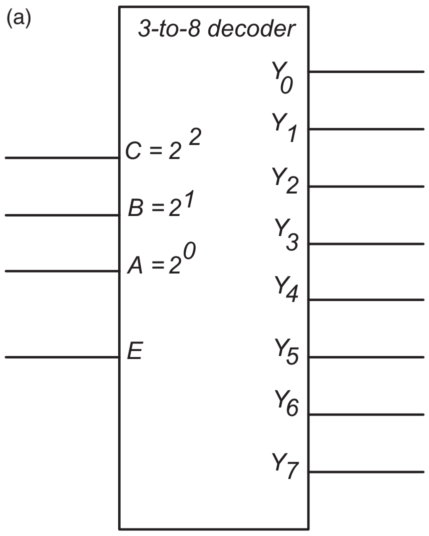
Table 8.2 Truth table of a 3-bit decoder with an active high enable

Inputs				Outputs							
C (2^2)	B (2^1)	A (2^0)	E	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
0	0	0	1	1	0	0	0	0	0	0	0
0	0	1	1	0	1	0	0	0	0	0	0
0	1	0	1	0	0	1	0	0	0	0	0
0	1	1	1	0	0	0	1	0	0	0	0
1	0	0	1	0	0	0	0	1	0	0	0
1	0	1	1	0	0	0	0	0	1	0	0
1	1	0	1	0	0	0	0	0	0	1	0
1	1	1	1	0	0	0	0	0	0	0	1
X	X	X	0	0	0	0	0	0	0	0	0

the enable is zero or negated, all the decoder output are zero, thus the decoder is disabled. That is, no matter what the values of its three binary input bits are, the outputs remain low as long as the enable is low.

From another point of view, the decoder can be seen as a *minterm* generator. Note that our 3-bit input decoder produces $Y_0 = 1$ upon input combination *000*.

Y_0 is *minterm* m_0 since $\overline{C}, \overline{B},$ and A are negated. Let us recall from the previous chapter that $m_0 = \overline{C}.\overline{B}.\overline{A}$ when we have a three-bit or three-variable function. The reader should convince herself that that is the case for every one of the eight *minterms*. Based on the decoder truth table, one cannot have more than one output asserted at any given time. Refer to Table 8.2 once more. The enable provides the decoder with a feature to negate all outputs regardless of the input present at inputs *C, B,* and *A*. This enable is useful when we want to make larger decoders with smaller ones. We will see that the enable allows us to interconnect the decoders in the appropriate manner. An example of this will be discussed soon. The decoder logic implementation is straightforward. Initially ignoring the decoder *enable*, we can think of our *3-to-8* decoder having eight three-input *AND* gates into which we present our eight binary combinations *000* through *111*. Let us name each *AND* gate as *AND gate 0, 1, 2,* and so forth. Upon presenting *000* to the inputs of *AND* gate *0* we want *AND* gate *0* output to assert while all other *7 AND* gates we want to see negated. Similarly upon presenting *001* to the inputs of AND gate *1*, we want *AND* gate *1* output to be asserted while all other *AND* gate outputs need to be negated. This procedure is carried for all *8 AND* gates to obtain our *3-to-8* decoder. Now it is time to go back to the decoder's *enable.* Since we want the enable not to interfere with the decoder functionality when *enable* is *1,* we just use four-input *AND* gates instead of the three-input ones used before. So upon *enable* being a *1* or asserted allows decoder operation as usual. When *enable* is negated all outputs are negated because a zero at the input of every one of the 8 4-input *AND* gates negates all outputs. Figure 8.6 depicts a possible logic implementation of a *3-to-8* decoder with an active high master *enable*.



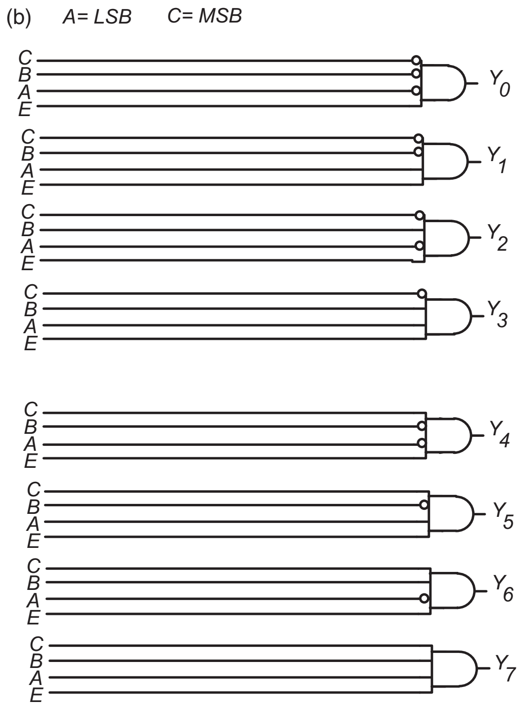


Figure 8.6 (a) Three-to-eight decoder with active high enable symbol; (b) a logic gate implementation of the decoder.

Note that the logic implementation of Figure 8.6b adopted the following notation to offer faster and easier readability of the circuit. The inputs *A, B,* and *C* may or may not have to be inverted depending on which *AND* gate output they need to assert. Instead of drawing explicitly an inverter at the input of every *AND* gate that requires its input to be inverted we draw a *bubble*. A *bubble* represents an inversion in the signal path in which it is drawn. For example, a *NAND* gate has a *bubble* at its output that means that the *NAND* is an *AND* followed by an inverter. Back to our Figure 8.6b explanation, *AND* gate Y_0 has to produce a *1* output upon *enable = 1* and *C.B.A = 000,* thus *AND* gate Y_0 has three bubbles to complement all three inputs *A, B,* and *C.* Similarly note that *AND* gate Y_1 has only two *bubbles* to negate inputs *C* and *B*, while input *A* is presented to the *AND* without inversion. A similar reasoning follows for the rest of the *AND* gates. Just remember that decoder input *A* is the least significant bit (2^0), while decoder input *C* is the most significant bit (2^2).

Example 8.2 Using a 3-to-8 decoder implement the following logic function:

$$f(\mathrm{C,B,A}) = \sum(2,5,7). \tag{8.11}$$

Since function *f* is a three-variable function, and a 3-to-8 decoder is a 3-bit function minterm generator, the implementation of Equation (8.11) consists simply of *OR-ing* the three minterms m_2, m_5, and m_7. Figure 8.7 depicts this implementation.

This is a good time to talk about the decoder unused outputs. Is there anything wrong with that? From an electrical point of view, there is nothing wrong about leaving combinational circuit outputs *floating* or just simply *not-connected* as Figure 8.7 shows. It is *not* correct though to leave any combina-

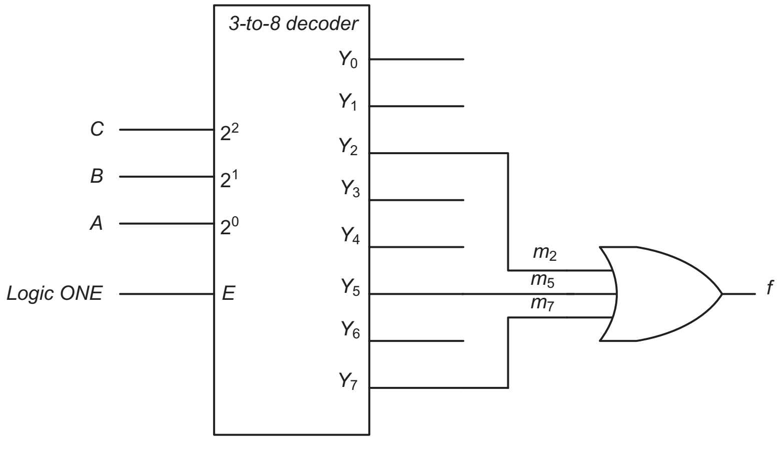


Figure 8.7 Decoder and gates implementation for Example 8.2.

tional circuit inputs *non-connected* or *floating*. Why? Because a floating input has no solid logic or voltage level driving such input. Since logic gates like all electronic circuits are susceptible to electrical and electronic noise, leaving a floating input is an opportunity for random noise to couple into the input and drive the input to the incorrect level. In summary, an unused input must be either tied down to zero ground or tied to a high voltage level or a one, typically the power supply voltage of the logic gate. Of course it is the job of the logic designer making the right choice to what input level to tie the unused input. As a quick example, let us look into a 3-input *AND* gate. Assume the gate is left over logic that we want to use for some other purpose on a board or part of a logic design. However, we only need a 2-input *AND* for this particular application. Can we still use the 3-input *AND* as a 2-input *AND*? The answer is yes, but we should not use the three inputs. Since we have an *AND* gate, tying the unused input to a high voltage level (logic one) in effect removes the third input out of the logic equation. The other two inputs of the *AND* gate behave as a 2-input gate.

Exercise: Prove the above statement with the use of a truth table. Are there any other ways to use a *3*-input *AND* gate, so that it behaves as a *2*-input *AND* gate?

8.2.1 Making Larger Decoders with Smaller Ones

Decoders of larger sizes, such as *5-to-32, 6-to-64* or larger will likely have to be constructed with smaller available decoders. One of the limitations of discrete *IC* decoders is that the larger they are, the larger is their number of pins. It is generally not practical for manufacturers to make huge decoders. Thus, it is usually left to the logic designer to assemble very large decoders using smaller ones or using programmable devices.

Example 8.3 Let us assume that we are given two *2-to-4* decoders with an active high enable input, and somehow we want to build with both of them plus some minimal amount of additional logic a *3-to-8* decoder. Of course for the sake of this example we will assume that we do not have or are not allowed to use a *3-to-8* decoder. Table 8.3 depicts the truth table of a *2-to-4* decoder with active high enable and active high outputs.

What we want to do is somehow connect two *2-to-4* decoders such that both jointly reproduce the truth table of a *3-to-8* decoder such as the one described by Table 8.2 at the beginning of the Decoders and Encoders Section. We will assume that the composite *3-to-8* decoder we are about to build will not necessarily have an enable input. This is not a big imposition; it is just a requirement that we make not to add a few more gates to the logic.

Figure 8.8 depicts the interconnection of two *2-to-4* decoders. Let us understand what such arrangement logically does.

Table 8.3 Truth table of a 2-to-4 decoder with active high enable

Inputs			Outputs			
B (2^1)	A (2^0)	E	Y_0	Y_1	Y_2	Y_3
0	0	1	1	0	0	0
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	1
X	X	0	0	0	0	0

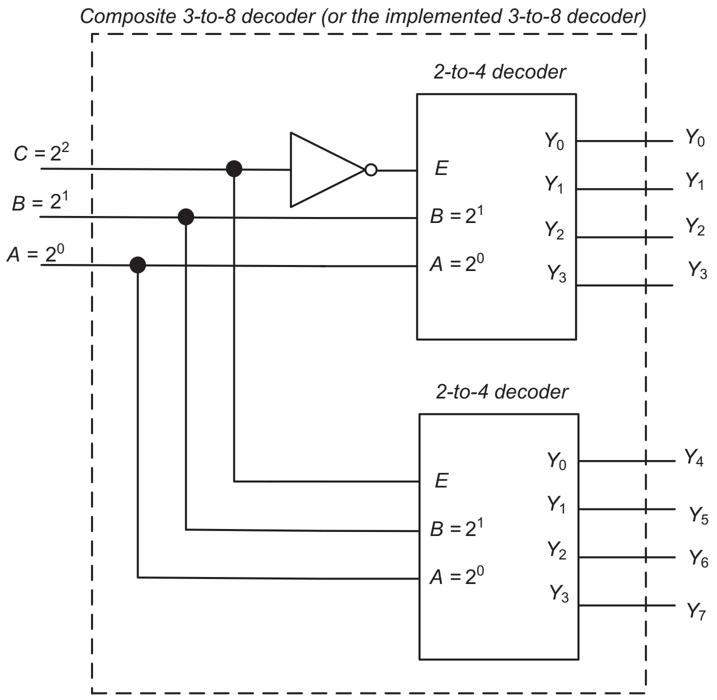


Figure 8.8 Three-to-eight decoder implementation with *2-to-4* decoders and one inverter.

The upper *2-to-4* decoder is wired such that its four outputs Y_0 through Y_3, will behave as the Y_0 through Y_3 outputs of the composite *3-to-8* decoder that we are trying to build. The lower *2-to-4* decoder is wired such that its four outputs Y_0 through Y_3, will behave as the Y_4 through Y_7 outputs of the composite *3-to-8* decoder. Furthermore, notice that both *A* and *B* inputs of each *2-to-4* are tied together and in turn they will also become the composite *3-to-8* decoder *A* and *B* inputs, where *A* is the *LSB*. Finally, the most interesting part of the design of Figure 8.8, is the way in which both enables are handled. The upper *2-to-4* decoder *E* enable input ties through an inverter to input *C,* the *MSB* of the composite decoder. Why? Note that upon *C, B,* and *A* binary combinations *000* through *011* being presented to the composite decoder, since *C* the *MSB* is inverted by the external inverter, the upper decoder behaves just like the composite *3-to-8* but just for the first four binary combinations of inputs (0 through 3). On the other hand, since the *E* enable of the lower *2-to-4* decoder is directly connected to input *C* of the composite, the lower decoder operates as the *3-to-8* composite one for *CBA* binary combinations, four through seven.

Exercise: Carefully trace the behavior of the composite decoder of Figure 8.8 and convince yourself that indeed it operates as a *3-to-8* decoder.

8.2.2 Encoders

An encoder is a combinational logic block that performs the inverse operation of a decoder. For example, for a *2-to-4* decoder, the associated encoder is a logic block with 4 inputs and 2 binary encoded outputs. An important combinational block used in embedded systems is the *priority encoder*. This encoder is important because it expands the number of interrupts that a micro controller is capable of handling using a single micro controller interrupt input line.

Example 8.4 Assume that a single interrupt line, an input to a micro controller, needs to have some logic in front of it to allow four interrupts to be *funneled* into the micro controller single interrupt line. Additionally, we want our priority encoding logic to supply the interrupt priority level of the interrupt with the highest priority on the P_1 and P_0 binary encoded outputs. Refer to the priority encoder schematic symbol in Figure 8.9.

Assume that interrupt priority *3* (I_3) is the highest while priority *0* is the lowest. If two interrupts assert at the same time, say *3* and *1*; since *3* has higher priority than *1,* we want the priority encoder to produce an encoded binary *3* at its output P_1 and P_0. In addition we want our priority encoder to assert a

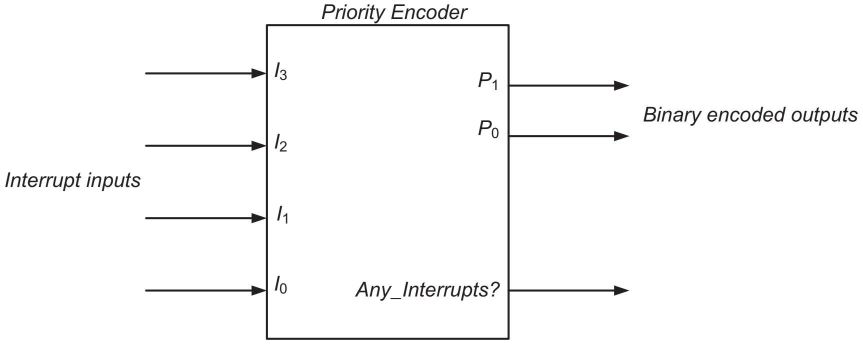


Figure 8.9 Priority encoder schematic symbol.

Table 8.4 Priority encoder truth table

Interrupt Inputs				Outputs		
I_3	I_2	I_1	I_0	P_1	P_0	Any_Interrupts?
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

Table 8.5 Depiction of the expansion of Table 8.4

Interrupt Inputs				Outputs		
I_3	I_2	I_1	I_0	P_1	P_0	Any_Interrupts?
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	1
0	0	1	1	0	1	1
0	1	0	0	1	0	1
0	1	0	1	1	0	1
0	1	1	0	1	0	1
0	1	1	1	1	0	1
1	0	0	0	1	1	1
1	0	0	1	1	1	1
1	0	1	0	1	1	1
1	0	1	1	1	1	1
1	1	0	0	1	1	1
1	1	0	1	1	1	1
1	1	1	0	1	1	1
1	1	1	1	1	1	1

third output to indicate that no interrupts are asserted. Table 8.4 presents a complete description of how we want our priority encoder logic to work. Note that Table 8.4 has several don't-care conditions. For the first line, when no interrupts are asserted the *Any Interrupts?* output is negated meaning, there are no interrupts, thus the priority code bits P_1 and P_0 are don't cares. For the next line of the truth table when interrupt I_0 asserts, while I_1, I_2, and I_3 are zero, priority code bits P_1 P_0 must become *00*. For the last line of Table 8.4, if I_3 asserts regardless the state of interrupt bits I_0, I_1, and I_2, priority code bits P_1 P_0 must become *11* and the *Any-Interrupts?* output must assert.

Let us now consider the same priority encoder, explicitly assigning its $2^4 = 16$ values, to the four interrupt input lines, priority encoded outputs P_1 and P_0 and *Any_Interrupts?*. As usual output *Any_Interrupts?* indicates the presence of an asserted interrupt at the input of the encoder. Thus, we obtain Table 8.5 for the same logic presented by Table 8.4, without using *don't cares* in an explicit form.

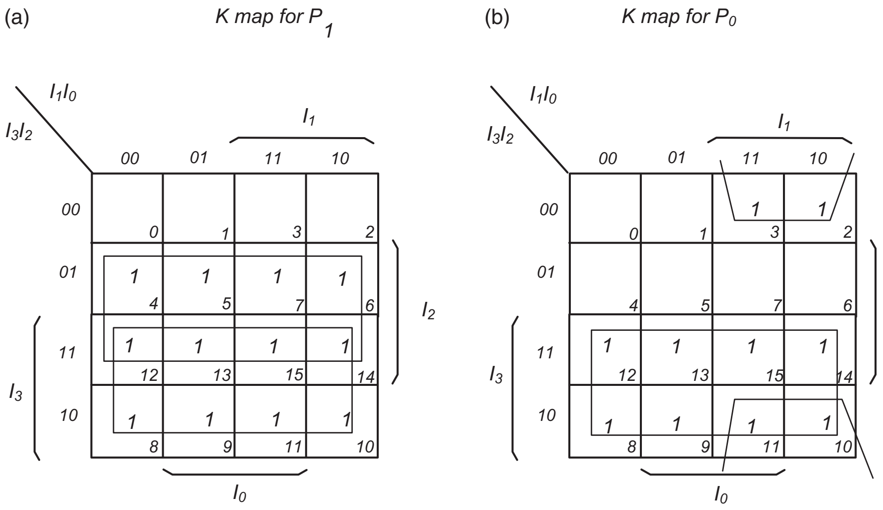


Figure 8.10 (a) Karnaugh map for P_1; (b) Karnaugh map for P_0.

From Table 8.5 we can do the three Karnaugh map to find out the combinational logic of outputs: P_1, P_0, and *Any_Interrupts?* But let us look at the logic of output *Any_Interrupts?*

By carefully inspecting the truth table, it is easy to see that the logic for *Any_Interrupts?* is:

$$\text{Any\_Interrupts?} = I_3 + I_2 + I_1 + I_0. \tag{8.12}$$

This is clear because output *Any_Interrupts?* is zero only when all inputs are zero (Tables 8.4 and 8.5).

For outputs P_1 and P_0 we produce the *K*. maps of Figures 8.10a,b.

By inspection of Figure 8.10a,b we obtain the following:

$$P_1 = I_3 + I_2 \tag{8.13}$$

$$P_0 = I_3 + \overline{I_2}.I_1. \tag{8.14}$$

Drawing the logic gates of Equations (8.12) through (8.13) we obtain Figure 8.11.

8.3 MULTIPLEXERS AND DEMULTIPLEXERS (MUXES AND DEMUXES)

Many years ahead of digital multiplexers and demultiplexers, mechanical versions of them were available. These devices were initially called distributors

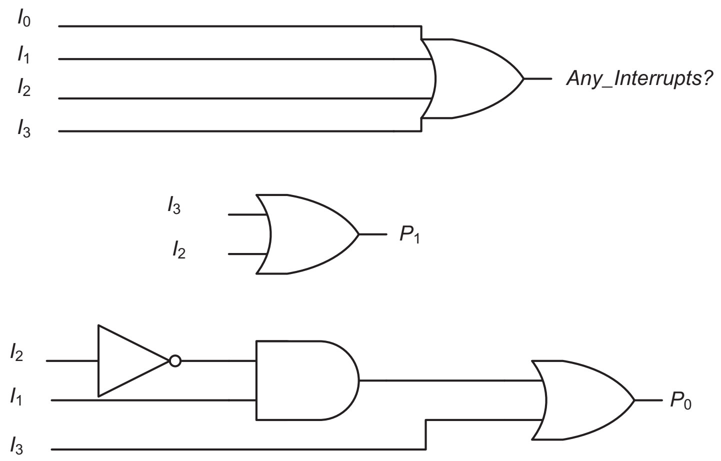


Figure 8.11 Logic implementation of the priority encoder for Example 8.4.

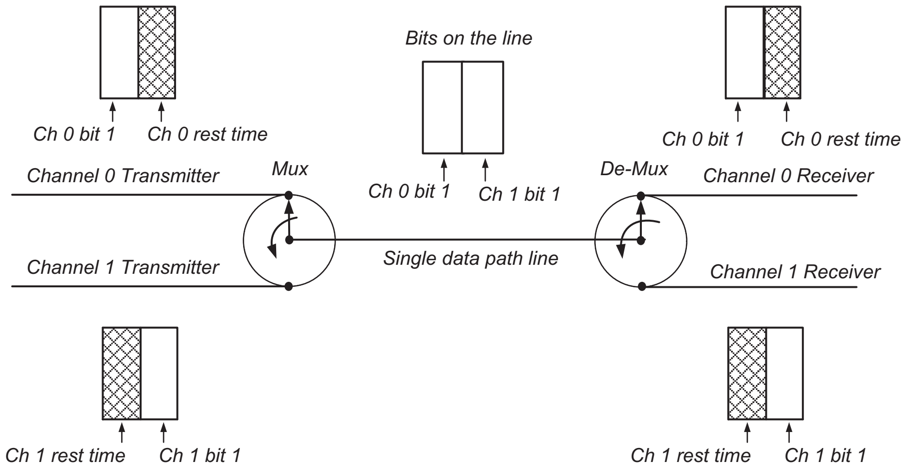


Figure 8.12 Multiplexer (mux) and demultiplexer (de-mux) transmission/reception scheme.

and used mostly as part of telegraph equipment by the end of the nineteenth century. The purpose of these devices is to allow more than one transmitting data source to use a single serial line, connected between the *n* sources and *n* destinations. The serial line between source and destination is time-shared. Let us look at how this works looking at the scheme depicted in Figure 8.12. For simplicity, assume that there are just two transmitting sources, *channel 0* and *channel 1*. Let us assume that each source transmits a bit (either a 0 or a 1) for 1 ms and does not transmit anything for another millisecond. It is conceivable to synchronize the two transmitting sources such that when channel *0* transmits it data bit, channel *1* rests; the next bit time channel *0* rests while channel *1* transmits its data bit. For the sake of simplicity, let us not be concerned with exact timing details of a real implementation.

When *channel 0* transmitter has a bit to send over the line, *channel 0* and its mux rotor must be engaged and *channel 0* receiver and its de-mux rotor must be engaged. This connection has to persist for 1 ms, during which time a bit is transmitted from *channel 0* transmitter (on the left) to *channel 0* receiver (on the right). We are ignoring finite propagation delays over the serial line, rotor rotation times, and several other factors that should not matter at this point. After 1 ms, *channel 1* transmitter has a bit to send over the line, *channel 1* and its mux rotor must be engaged, and *channel 1* receiver and its de-mux rotor must be engaged. During this time bit *1* of *channel 1* gets transmitted. This process repeats indefinitely or until no more transmissions are desired.

Today muxes and de-muxes can be designed to transport analog or digital signals. This chapter focuses on digital devices only.

8.3.1 Multiplexers

Digital multiplexers are devices that allow a number of data sources to route one out of the total data sources to its output. Let us assume that we have a four-input mux, at any given time one input is allowed to pass straight through the mux onto the output. At such time none of the other inputs can go through the mux. This scheme clearly works fine when the data path at the output of the mux can be time-shared by the various inputs to the mux. Multiplexers are referred to as being *1-of-*2^n, where n is the number of input channels. Conceptually we can have *2*, *4*, *8*, *16*, . . . , 2^n input multiplexers. Table 8.6 depicts the truth table of a 1-of-4 mux in a compact fashion using *don't care* conditions. The same truth table is somewhat expanded in Table 8.7 by explicitly stating the values of each input channel data input. Note that in order to *fully expand* the truth table of Table 8.7, since there are seven inputs, the fully expanded truth table would have $2^7 = 128$ entries! Clearly this is not practical, and it is not too clear to understand either.

Truth Tables 8.6 and 8.7 are easy to understand. They should be read in the same manner as the mux operates. For example when input *0* is selected (select

Table 8.6 Compressed 1-of-4 multiplexer truth table

Enable	Input Channel	Data Select Line		Output
E	I_x	S_1	S_0	Y
1	I_0	0	0	I_0
1	I_1	0	1	I_1
1	I_2	1	0	I_2
1	I_3	1	1	I_3
0	X	X	X	0

Table 8.7 Somewhat expanded or more explicit 1-of-4 multiplexer truth table

Enable	Data Inputs				Data Control Line		Output
E	I_0	I_1	I_2	I_3	S_1	S_0	Y
1	0	X	X	X	0	0	0
1	1	X	X	X	0	0	1
1	X	0	X	X	0	1	0
1	X	1	X	X	0	1	1
1	X	X	0	X	1	0	0
1	X	X	1	X	1	0	1
1	X	X	X	0	1	1	0
1	X	X	X	1	1	1	1
0	X	X	X	X	X	X	0

lines set to select such input), regardless of what all other mux inputs input levels are (*don't cares*), the selected input *0* passes straight through the mux to its output *Y*. The same is true for when input *1* or *2* or *3* is selected. The operation of the enable *E* is such that the mux does its thing (route input data to output), upon *E* being high. However, when *E* is low, the mux output is zero. The *E* input is useful when we want to build larger multiplexers using smaller ones.

Figure 8.13a depicts the schematic symbol of a *1-of-4* mux, and b of the same figure depicts a possible logic implementation of such mux.

It is not too hard to figure out the truth table of virtually any size mux just by similarity with the *1-of-4* mux just covered. For example, a *1-of-8* mux will have eight data inputs, three select lines to choose one out of eight inputs to go through the *Mux*, a master enable *E*, that allows us to concatenate the mux with others to build even larger multiplexers, and one output.

Exercise: Derive the truth table and a logic implementation of a *1-of-8* mux. Assume that one has logic gates of the required number of inputs, to facilitate the task. This last assumption does not preclude generality to the exercise. If gates of the required number of inputs are not available or we are not allowed to use them, we can always build gates with larger number of inputs using multiple gates with a smaller number of inputs. For example an *8*-input *OR* gate can be built in a number of ways according. Figure 8.14 shows two possible implementations of an *8*-input *OR* gate (a) using *4*-input and *2*-input gates (b) using all *2*-input gates.

8.3.2 Building Larger Multiplexers

How do we construct a *1-of-8* mux using just *1-of-2* muxes? Assume that all of our *1-of-2* muxes have a master enable input pin. We can think of a mux having a funneling effect on its input data signals from left to the output on

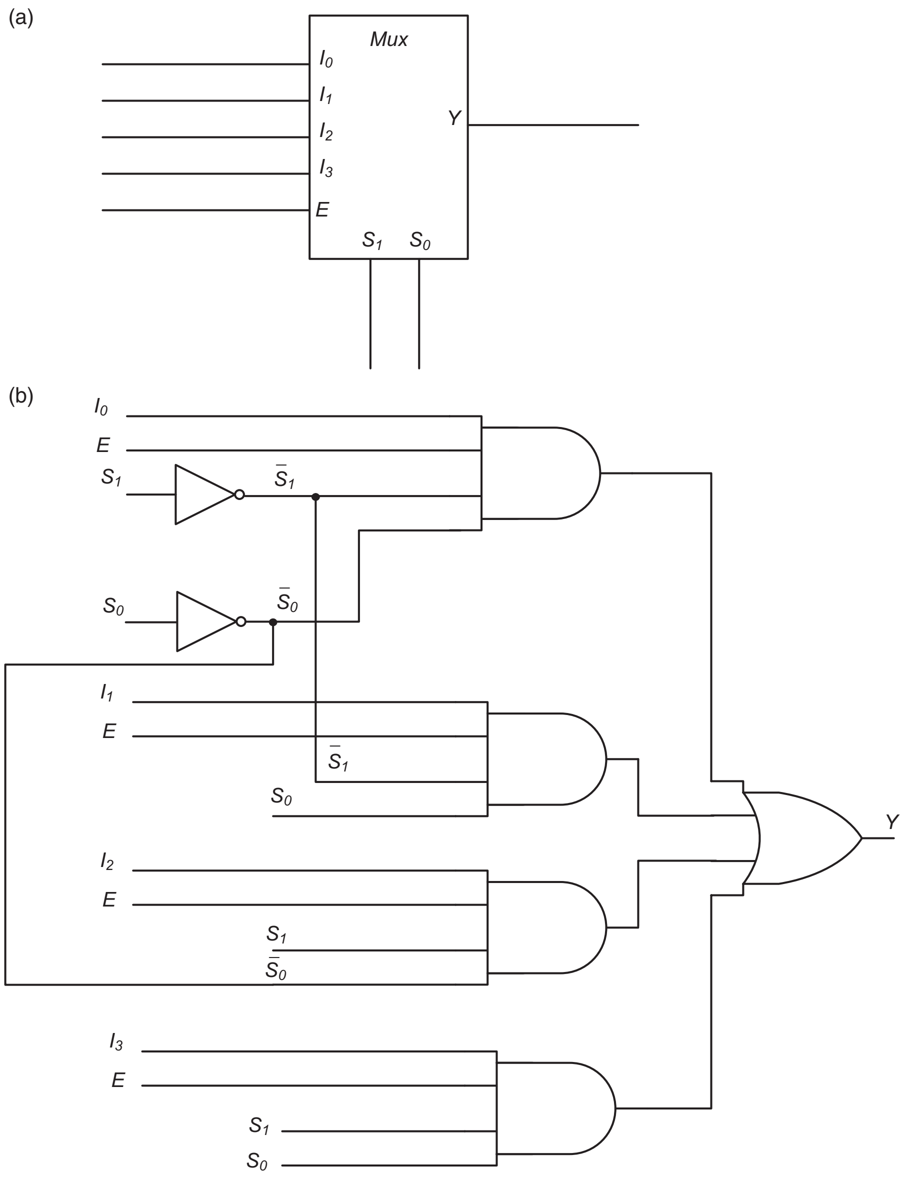


Figure 8.13 (a) *1-of-4* mux schematic symbol; (b) *1-of-4* mux logic implementation.

the right. Let us refer to the graph depicted in Figure 8.15, if we start with a *1-of-2* mux (*Mux 1*) we can feed with two other *1-of-2* muxes (*Muxes 2 and 3*) a total of *4* signals into *Mux 1*. We repeat this process one more time and we can feed *8* signals into *Mux1*, using in addition to *Muxes 2* and *3*, *Muxes 4*, *5, 6,* and *7*.

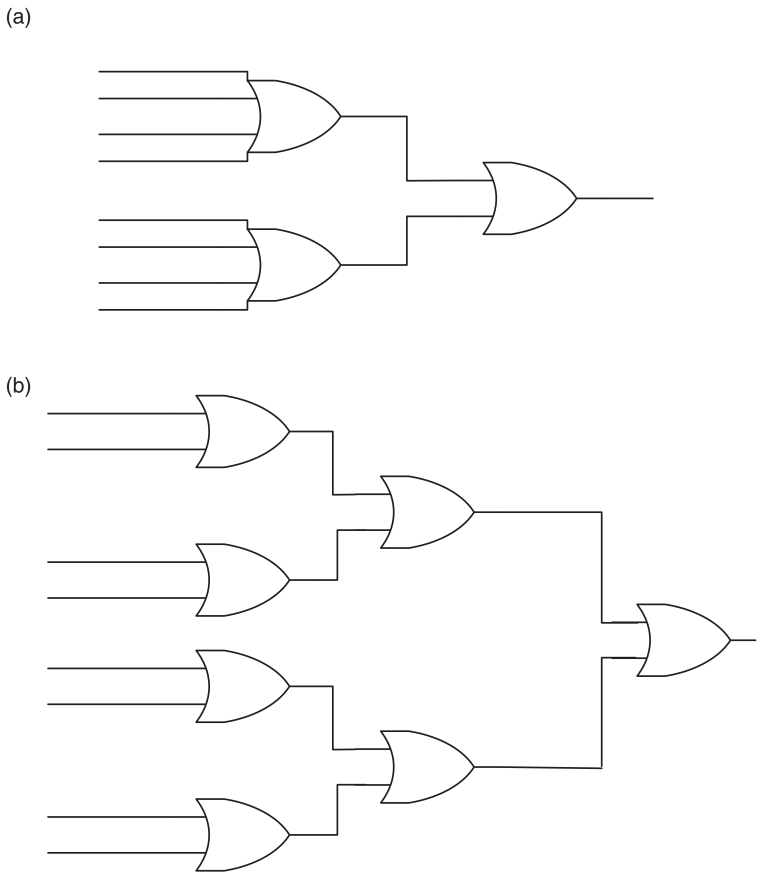


Figure 8.14 Eight-input OR gate implementation (a) using *4*-input and *2*-input gates; (b) using all *2*-input gates.

But we are not done yet; we still need to identify data inputs I_0 through I_7 of the overall composite *1-of-8* mux. So one more time referring to our picture of Figure 8.15, assume that the data select line of *Mux 1* is assigned to be the *MSB* of the select lines of our composite *1-of-8* mux. The select lines of muxes 2 and 3 are tied together and assigned to be the middle bit of the *3*-bit select line group of our *1-of-8* mux. Finally, we assign the select line of muxes *4, 5, 6,* and *7* tied together to the *LSB* of the 3-bit select line group of our composite *1-of-8* mux. Following what was just described can be seen depicted in Figure 8.15. In Figure 8.15 the data paths are highlighted with heavy lines. The select lines are shown with a medium weight line. Finally, the master enable lines are all drawn with a lightweight line. Lines that cross and do not have a heavy dot at their intersection are *not connected.*

The techniques depicted in Figure 8.15 can be generalized to build virtually any mux of any desired number of inputs with other combination of smaller

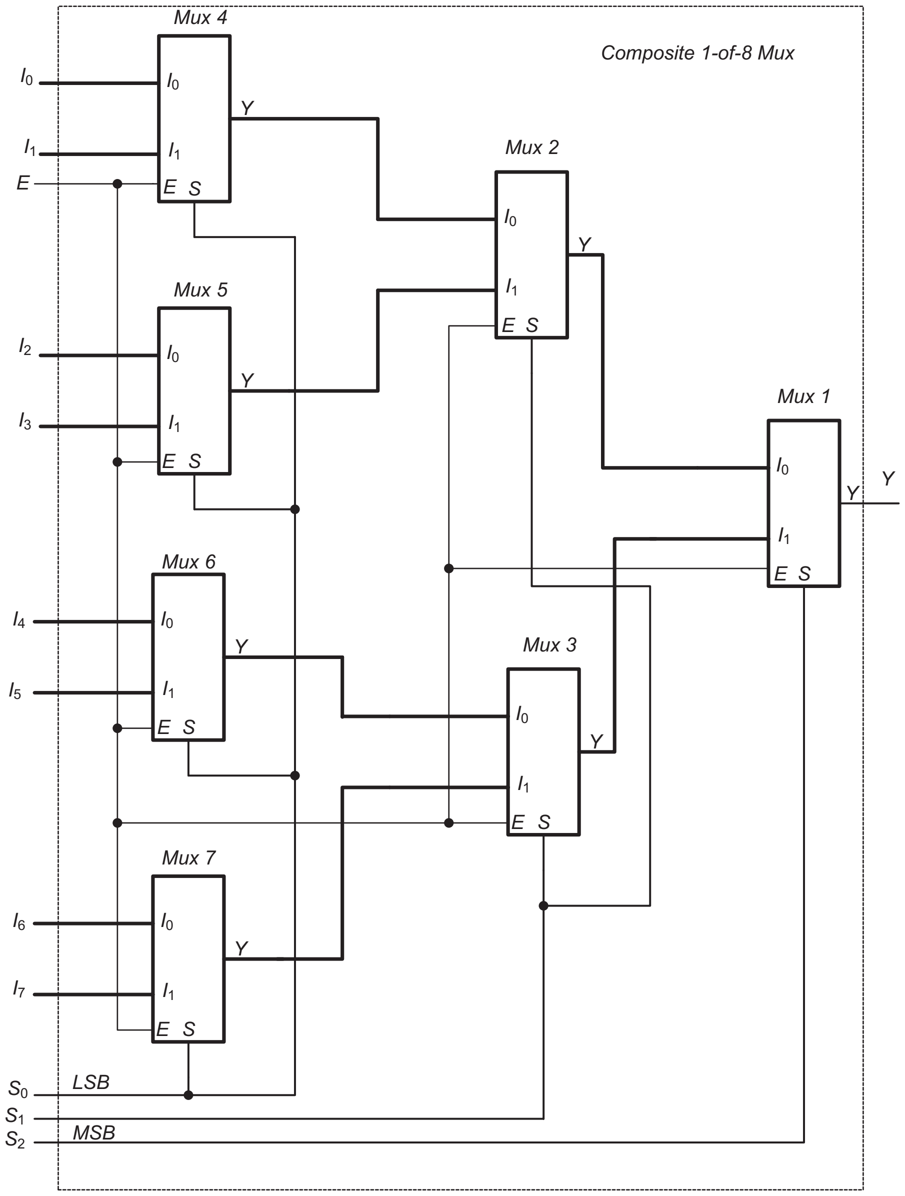


Figure 8.15 A *1-of-8* mux implementation using *1-of-2* muxes.

muxes. Naturally the number of mux inputs is always 2^S, where exponent S is the number of the mux select lines.

Exercise: (1) Using the techniques used for the *1-of-8* mux, build a composite *1-of-32* mux. Hint: Use four *1-of-8* muxes and one *1-of-4* mux. (2) Try a different implementation with another mix of muxes.

8.3.3 De-Multiplexers

From our previous example depicted in Figure 8.12 and knowing the logic of a decoder, we can appreciate that in a way a decoder can be used as a de-multiplexer, in the sense that it reverses what multiplexers do to data. Figure 8.16a shows wedged-shaped symbols for *mux* and *de-mux*. Such wedged

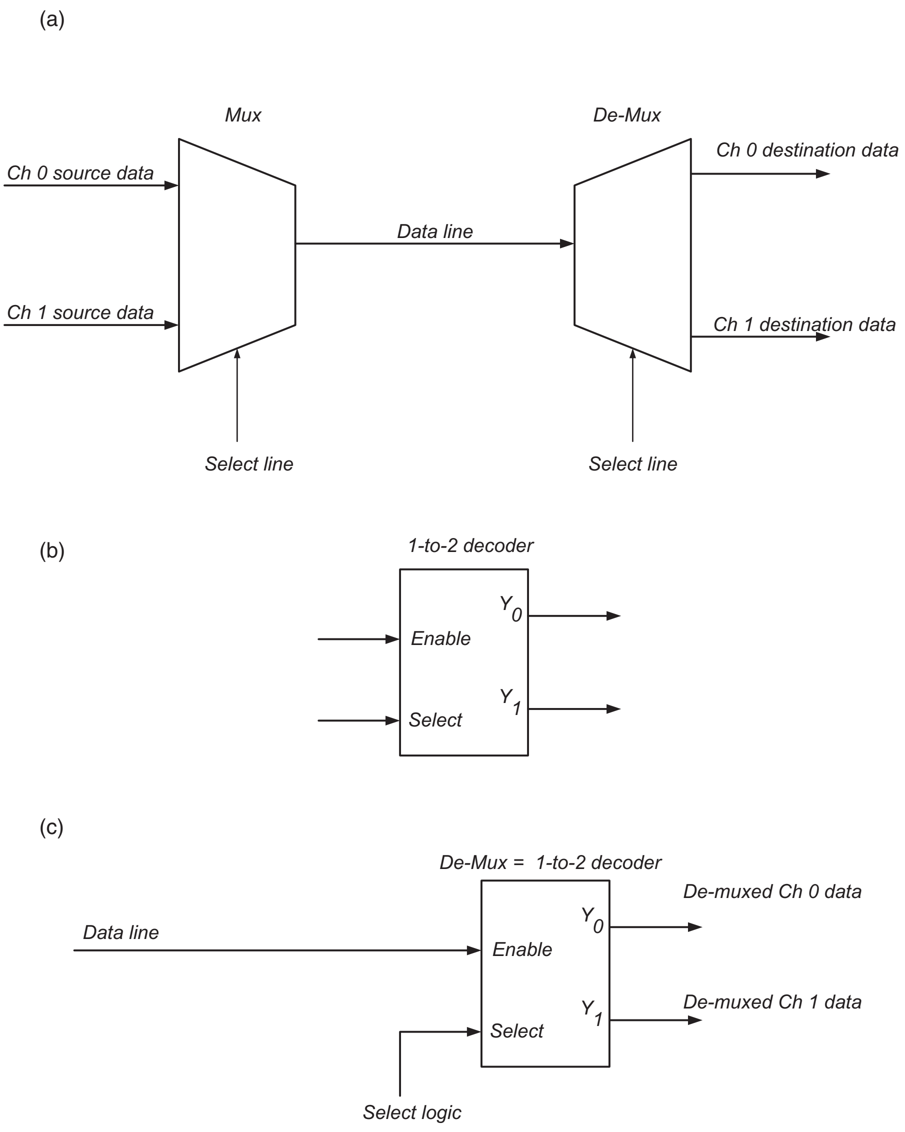


Figure 8.16 (a) System level view of a mux/de-mux application; (b) *1-to-2* decoder with enable; (c) *1-to-2* decoder wired as a de-multiplexing device.

symbols are preferred in computer architecture and systems illustrations. We will use a wedge for a *mux* in later chapters of the book that deal with computer architecture. Figure 8.16b shows the schematic symbol of the simplest decoder one can have a 1-to-2 decoder, with a single select line, two outputs and its master enable line. Finally, Figure 8.16c depicts the use of a decoder as a de-multiplexer in our application of Figure 8.12.

8.4 SIGNED AND UNSIGNED BINARY NUMBERS

The binary numbers that we described on the previous chapter did not have any sign; they were just positive or unsigned binary numbers. If we have n bits to represent a positive number there are 2^n binary combinations of such numbers. Now if we intend to represent positive as well as negative numbers, but continue to use binary-valued terms or *bits,* we must give up some of the positive number binary combinations and allocate them to the negative range. Why? Because we cannot use a negative sign to depict a negative number; this implies the need of three different symbols to represent numbers, the *1*, the *0*, and the "-" sign. We are supposed to represent positive and negative numbers with just *ones* and *zeros*. This will become clearer when we go over some examples.

8.4.1 One's Complement Representation of Binary Numbers: Addition

Let us assume that we are working with *3*-bit binary numbers. The *1's complement* of a binary number is defined as the *bit-to-bit* complementation of every one of its bits. For example, given the *3*-bit binary number *010*, its *1's c*omplement is *101*. Similarly, given the *3*-bit number *101,* its *1's c*omplement is *010.* It is easy to see that *1's c*omplementing a number twice in a row leads to the original number we started with. This is similar to the involution rule covered in the previous chapter (refer to Table 7.11 in Chapter 7).

Now what follows is the most important consideration about *1's c*omplement numbers, given n bits to represent a *1's c*omplement number, the most significant bit *(MSB)* is allocated to represent the number sign. A leading *0* means the number is positive, a leading *1* means that the number is negative. The rest or the $(n - 1)$ remaining bits are assigned to represent the number's magnitude. Figure 8.17a depicts the bit assignments of an *n-bit 1's c*omplement number, Figure 8.17b depicts the bit assignments for a three-bit $(n = 3)$ *1's* complement number.

Now we know how to obtain the *1's c*omplement of a number and we know how the bits are assigned. Let us look into how we obtain the negative number of *3*-bit positive *001.*

Simply take the *1's c*omplement of *1*, which leads to *110.*

$$\text{1's C}(001) = 110. \tag{8.15}$$

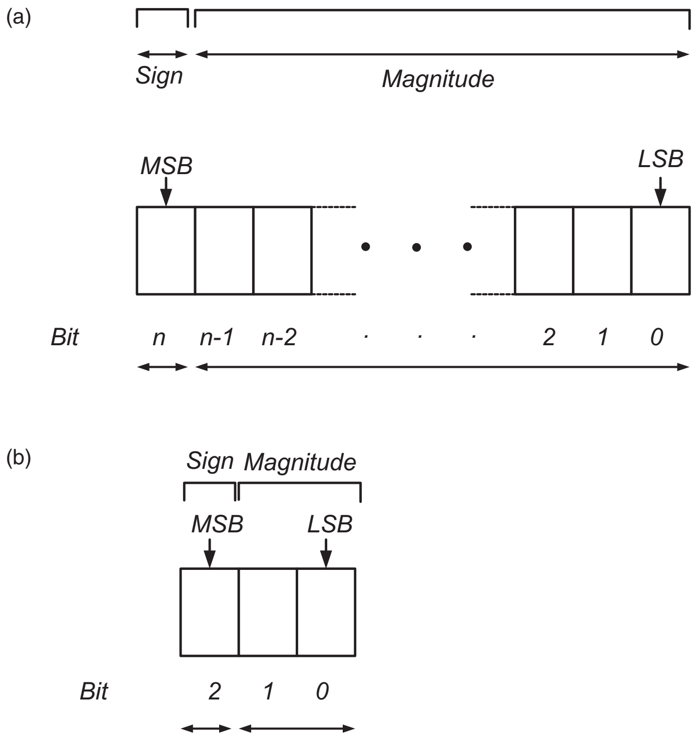


Figure 8.17 One's complement bit assignments (a) for an *n*-bit number; (b) for a *3*-bit number.

Equation (8.15) thus is the representation of decimal number *–1* in three-bit 1's complement form. Conversely, given a *3*-bit *1's c*omplement number such as *110,* by inspection of the number's *MSB* we know that we are dealing with a *3*-bit negative number, check the MSB. One more time in order to find the magnitude of such negative number; again we take the *1's c*omplement of 110.

$$1\text{'s C}(110) = 001. \tag{8.16}$$

Hence the given negative number (i.e., *110*) magnitude is *1.* Table 8.8 depicts the *1's c*omplement of all 3-bit positive numbers.

By inspection of Table 8.8 we can tell that if we want to use the 1's complement representation for positive as well as negative numbers, the first four numbers under the *1's Complement* column have to be negative numbers, because they have a *one MSB*; whereas the last four binary combinations of the same column have to represent four positive binary numbers, because they have a leading *zero*.

Note from Table 8.8 that the number zero has two 1's complement representations, that is *000* and *111*; that is *positive zero* and *negative zero*. We will

Table 8.8 3-bit binary numbers and their associated 1's complement representation

Positive 3-Bit Binary Number	Positive 3-Bit Binary Decimal Equivalent	1's Complement	1's Complement Decimal Equivalent
000	0	111	–0
001	1	110	–1
010	2	101	–2
011	3	100	–3
100	4	011	3
101	5	010	2
110	6	001	1
111	7	000	0

Table 8.9 Basic rules for unsigned or positive binary addition

Augend	Addend	Sum	Carry Out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

see that *2's c*omplement is a better negative number representation system, which will be the topic of our subsequent section.

Table 8.9 depicts the basic addition of two single bit unsigned numbers, *augend* and *addend*, the results is the sum and the right most column is the *carry out*.

So after all of the above why is *1's c*omplement good or what is it for? We can add numbers in *1's c*omplement representation using the fundamental rules of unsigned binary addition given by Table 8.9.

8.4.1.1 Four-Bit 1's Complement Representation

4-bit *1's c*omplement numbers range from *0000* (decimal +*0*) up to *0111* (decimal +7). Negative *4*-bit *1's c*omplement numbers range from *1000 (decimal –7)* up to *1111 (decimal –0).*

The algorithm to obtain the 1's complement of an *n*-bit binary number is simply flipping its *zeros* to *ones* and its *ones* to *zeros*. Given that *X* is our *n*-bit binary number:

$$1'sComplement(X) = 1'sComplement\,(x_{n-1}, x_{n-2}, \ldots, x_1, x_0) = \overline{x_{n-1}}, \overline{x_{n-2}}, \ldots, \overline{x_1}, \overline{x_0}$$

Table 8.10 lists all *4*-bit numbers in *1's c*omplement representation.

From Table 8.10 again we see that *4*-bit *1's c*omplement numbers exhibit plus and minus zero or double representation for the number zero. As a matter of fact, all *n*-bit *1's c*omplement numbers will always produce double representation of the number *zero*.

Table 8.10 Four-bit 1's *c*omplement numbers

Representation in 4-bit 1's Complement	Assigned Decimal Number
0000	+0
0001	+1
0010	+2
0011	+3
0100	+4
0101	+5
0110	+6
0111	+7
1000	–7
1001	–6
1010	–5
1011	–4
1100	–3
1101	–2
1110	–1
1111	–0

Example 8.5 Given the following *4*-bit *1's c*omplement numbers, perform the additions indicated below and double check your results using their decimal equivalent.

(a) *0100 + 0011*,
(b) *0101 – 0001*,
(c) *1011 + 0010*,
(d) *1011 + 0110*.

Solutions

With the aid of Tables 8.9 and 8.10 we perform the operation as follows:

(a) *0100 + 0011 = 0111* and in decimal: 4 + 3 = 7
(b) *0101 – 0001 = 0101 + (–1 in 1's C) = 0101 + 1110 = 0011 and a carry of 1* in order to achieve the correct decimal result of *+4* (since we are subtracting *1* from *5*) the carry must be wrapped around and added back to the previous sum. Thus:

$$\begin{array}{rr} & 0101 \\ & +1110 \\ \hline \text{Carry} = 1 & 0011 \end{array}$$

Finally add the carry and let us refer to it as the *End-Around-Carry (EAC)* of *1* so that:

$$\begin{array}{r} \mathit{0011} \\ \underline{EAC + \mathit{0001}} \\ \mathit{0100} \end{array}$$

in decimal we have that *5 − 1 = +4,* which is the final answer. Note that the first addition and the addition of the *EAC,* the second addition, effectively take two *addition times*, to perform the complete sum.

(c) *1011 + 0010 = 1101* and in decimal, note that *1011* is *−4* in 1's Complement, and *1101* is *−2. Thus: −4 + 2 = −2*

(d) 1011 + 0110 = 0001 and a Carry = 1, treating the Carry as an EAC we obtain:
0001 + 0001 = 0010. In decimal we have that *1011 + 0110 = 0010,* which is *−4* + 6 = +2. Note that when dealing with the *1's* complement addition not wrapping around the carry and adding to the previously obtained addition will not lead to the correct answer.

Note that what we need to do when we want to subtract *B* from *A*, is to add *A* and minus *B,* plus any end-around carry *(EAC)* that comes out of the operation. Because of the subsequent addition of the *EAC* the *1's c*omplement subtraction method is twice as slow as the *2's* complement subtraction. Because of this fundamental reason *1's* complement subtraction is hardly used.

8.4.2 Two's Complement Representation of Binary Numbers: Addition

Having learned *1's c*omplement representation well it is reasonably straightforward to understand *2's c*omplementation. The basic formula to obtain the *2's c*omplement representation of *X* an *n*-bit number is:

$$2\text{'s}\,C(X) = 1\text{'s}\,C(X) + 1 \text{ ignoring the Carry out bit.} \tag{8.17}$$

The sign and magnitude format for *2's c*omplement numbers, i.e. *MSB* is the sign bit, rest of the bits are its magnitude, is identical to the bit assignment for *1's c*omplement numbers (see Fig. 8.17).

Why *2's c*omplement numbers, we might ask ourselves. There are two reasons for them; first we will see after applying Equation (8.17) that there is a single representation for the number zero. Secondly, *2's c*omplementation addition never has to add a carry as an *EAC* (like *1's c*omplement does), the

Table 8.11 Four-bit 2's complement numbers

Representation in 4-bit 2's Complement	Assigned Decimal Number
0000	+0
0001	+1
0010	+2
0011	+3
0100	+4
0101	+5
0110	+6
0111	+7
1000	–8
1001	–7
1010	–6
1011	–5
1100	–4
1101	–3
1110	–2
1111	–1

carry in a *2's c*omplement addition must be set to zero if there is no C_{in} from a less significant bit position. When carry out is ignored, the *2's c*omplement representation of the addition is obtained.

Table 8.11 depicts all four-bit 2's complement numbers.

Example 8.6 Shows how to perform binary additions in *2's c*omplement representation. The decimal equivalents of the same operations are shown. Perform the following four-bit *2's c*omplement additions:

(a) *0100 + 0011*
(b) *0111 + 1110*
(c) *0101 + 1100*
(d) *1011 + 1110*

Solutions to Example 8.6

Use the algorithm presented by Equation (8.17) to perform *2's* complementation.

(a) *0100 + 0011 = 0111*, in decimal: *+4 + 3 = +7*
(b) *0111 + 1110 = 0101 and a carry of 1*, since carry out has to be ignored the result is: *0101*, which in decimal is: *+7 – 2 = +5*
(c) *0101 + 1100 = 0001* and a carry out of 1, since the carry has to be ignored the result is *0001*, which in decimal is: *+5 – 4 = +1*
(d) *1011 + 1110 = 1001* and a carry out of *1*, since the carry has to be ignored the result is *1001*, which in decimal is: *–5 – 2 = –7*

Table 8.12 Some decimal, octal, and hexadecimal numbers

Decimal	Octal	Hex	Decimal	Octal	Hex
0	0	0	9	11	9
1	1	1	10	12	A
2	2	2	11	13	B
3	3	3	12	14	C
4	4	4	13	15	D
5	5	5	14	16	E
6	6	6	15	17	F
7	7	7	16	20	10
8	10	8	17	21	11

8.4.3 Other Numbering Systems

Interestingly, infinitely many numbering systems exist. In computer software the most interesting and usual numbering systems, which we have not discussed yet, are the octal and the hexadecimal systems. We will very briefly touch on this subject since in the author's experience most scientists and engineers already know those numbers. The octal numbering system is simply based on eight uniquely defined digits, which are: 0, 1, . . . , 7. This numbering system is referred to as base *8*. The hexadecimal numbering system or base 16, has 16 uniquely defined digits, which are: *0* through *9* and *A* through *F*. Table 8.12 depicts the first *18* decimal, octal, and hexadecimal numbers.

The arithmetic rules for adding octal-to-octal and hex-to-hex numbers are pretty similar to those of decimal arithmetic. Care must be exercised knowing the uniquely defined digits for each numbering representation.

8.5 ARITHMETIC CIRCUITS: HALF-ADDERS (HA) AND FULL-ADDERS (FA)

Arithmetic circuits can be designed using the same concepts that we use when designing any other logic circuits. Basically truth tables and simplification methods are used to design them. Let us assume that we want to design the logic implementation of an adding cell. That is, a circuit that reads an augend bit (A), an addend bit (B), and produces the sum bit (S) and its carry out (C_{out}). Such circuit is referred to as a half-adder *(HA)* because it does not handle the carry in bit as full-adders do. The full-adder *(FA)* receives three input bits: augend (A), addend (B), and carry in (C_{in}), and it produces the sum bit of all three input bits and a carry out (C_{out}) bit. Table 8.13 depicts the truth table for a half-adder.

We obtain a maximally *SOP* form for output bits C_{out} and S of our half-adder.

Table 8.13 Half-adder truth table

Augend A	Addend B	(Carry out) C_{out}	Sum S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

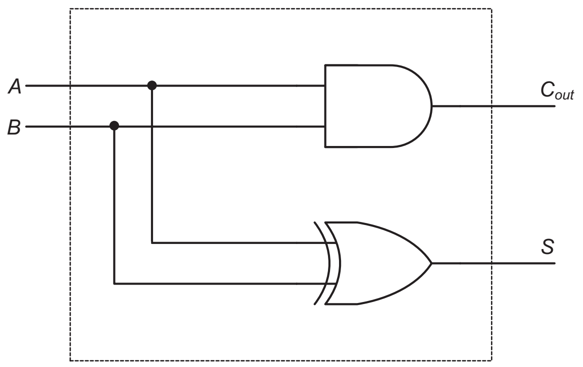


Figure 8.18 Half-adder logic implementation.

Table 8.14 Full-adder truth table

(Carry in) C_{in}	Augend A	Addend B	(Carry out) C_{out}	Sum S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

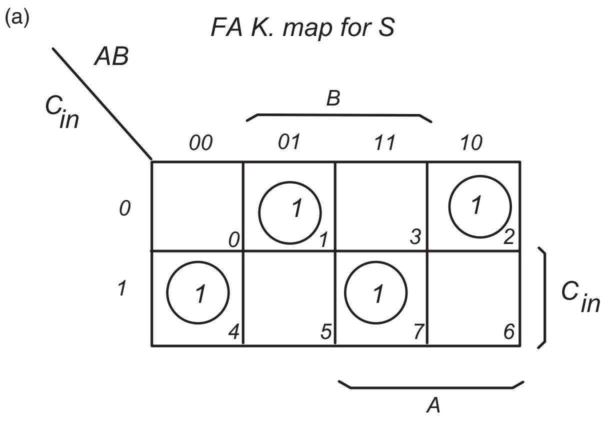
Without doing an explicit *2*-variable *K. map* it can be seen that:

$$C_{out} = AB \tag{8.18}$$

and

$$S = A \oplus B \tag{8.19}$$

The logic implementation for the *HA* is given by Figure 8.18.
Table 8.14 depicts the truth table of a full-adder.



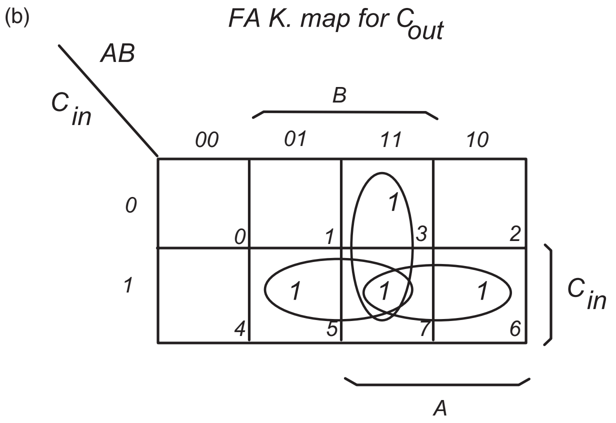


Figure 8.19 (a) Full-adder: *K*. map for C_{out}; (b) full-adder: *K*. map for *S*.

Using a *3*-variable *K*. map we find simplified logic equations to express the sum bit *S* and C_{out} of the *FA*. Figure 8.19 depicts the *K*. maps to obtain the maximally simplified *SOP* form for output bits C_{out} and *S*. From the truth table (Table 8.14) we fill in the *K*. maps for both output bits, C_{out} and *S*, these are depicted in Figure 8.19.

Referring to Figure 8.19a it is evident that none of the minterms m_1, m_2, m_4, and m_7 has any adjacent minterms. So the simplified *SOP* and the canonical *SOP* forms are identical. Moreover, from the canonical equation:

$$S = \sum(1,2,4,7) = \overline{C}_{in}.\overline{A}.B + \overline{C}_{in}.A.\overline{B} + C_{in}.\overline{A}.\overline{B} + C_{in}.A.B \tag{8.20}$$

and since a two-variable *XOR* is:

$$A \oplus B = A.\overline{B} + \overline{A}.B, \tag{8.21}$$

Equation (8.21) is found to be logically equivalent to Equation (8.23) after some Boolean algebra manipulations; that is:

$$S=\sum(1,2,4,7)=\overline{C}_{in}.\overline{A}.B+\overline{C}_{in}.A.\overline{B}+C_{in}.\overline{A}.\overline{B}+C_{in}.A.B=A\oplus B\oplus C_{in}. \tag{8.22}$$

The simplified *SOP* form for output bit C_{out} is:

$$C_{out}=AB+(A+B)C_{in}. \tag{8.23}$$

Writing the canonical form of Equation (8.23) by inspection of Figure 8.19b we obtain:

$$C_{out}=\overline{C}_{in}.A.B+C_{in}.\overline{A}.B+C_{in}.A.\overline{B}+C_{in}.A.B. \tag{8.24}$$

Grouping terms:

$$C_{out}=C_{in}(\overline{A}.B+A.\overline{B})+\overline{C}_{in}.A.B+C_{in}.A.B. \tag{8.25}$$

Applying Equation (8.25) to Boolean algebra rules yields:

$$C_{out}=AB+(A\oplus B)C_{in}. \tag{8.26}$$

The C_{out} of the *FA* has two alternate logic Equations (8.23) and (8.26).

The logic implementations of our full-adder S and C_{out} output bits are depicted in Figure 8.20.

Figure 8.21 depicts the schematic symbol diagram of a full-adder.

Note that this is the first time in this text that the schematic symbol of a combinational circuit is drawn in a somewhat nonconventional form. Conventionally circuits are drawn with inputs on the left-hand side and outputs on the right-hand side. Full-adders violate those conventions for exceptionally good reasons. Note that the C_{in} input to the *FA* is drawn on the right hand side, while its C_{out} output is drawn on the left hand side of the symbol. Inputs A and B are drawn on top and output S at the bottom. Inputs at the top and outputs at the bottom of schematic symbols are within the conventional drawing criteria. The reason why C_{in} is on the right and C_{out} is on the left is primarily due to the arithmetic done by an *FA*; similar to hand addition operations carries move from right-hand side digits to more significant or left-hand side digits. In the next section we will see that an interconnection of full-adders allows us to build multi-bit adders.

8.5.1 Building Larger Adders with Full-Adders

When we perform the addition of two numbers, an augend and an addend, regardless of whether these numbers are decimal or binary, the addition algorithm is always the same.

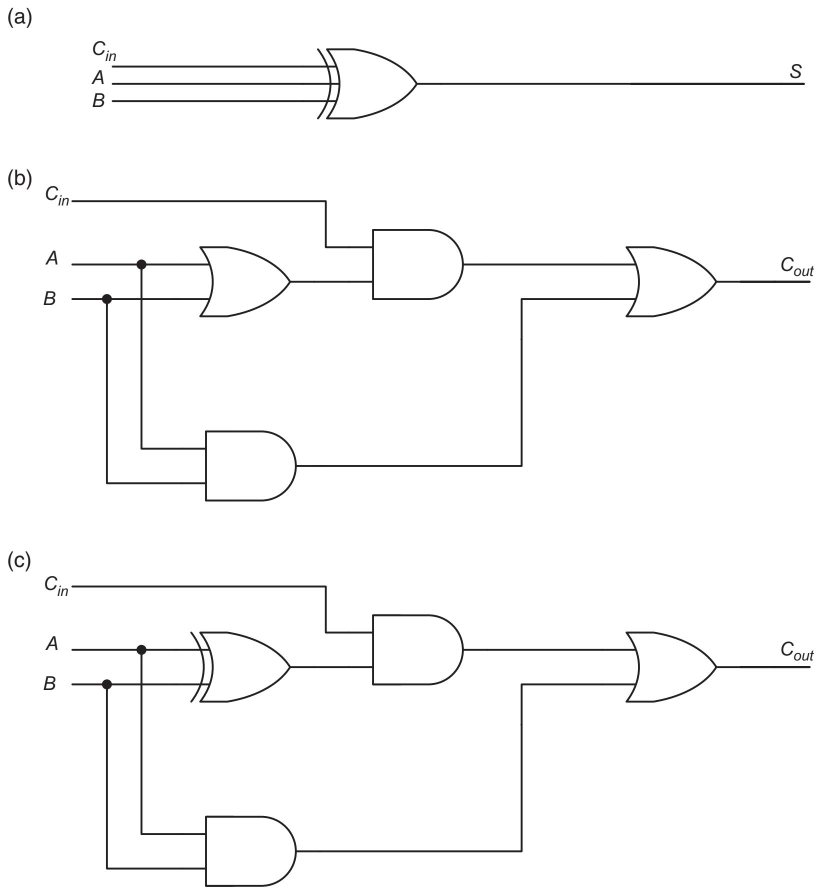


Figure 8.20 (a) *FA* logic implementation of its *S* output; (b) *FA* logic implementation of its C_{out} output; (c) *FA* alternate logic implementation of its C_{out} output.

Example 8.7 Given two 4-bit binary numbers, describe the algorithm that one utilizes in performing the complete addition. Assume our augend has bits $A_3A_2A_1A_0$ which can be annotated in a more compact fashion as $A[3{:}0]$. The addend of bits $B_3B_2B_1B_0$ can also be annotated as $B[3{:}0]$.

To perform the addition of $A[3{:}0]$ and $B[3{:}0]$ we write both numbers as follows:

$$\begin{array}{r} A_3A_2A_1A_0 \\ +\ B_3B_2B_1B_0\,. \\ \hline S_3S_2\,S_1\,S_0 \end{array} \tag{8.27}$$

We will refer to the above layout of numbers, Equation (8.27) being formed by four slices, slice *0* the least significant slice, contains A_0, B_0, *and* S_0, then slice

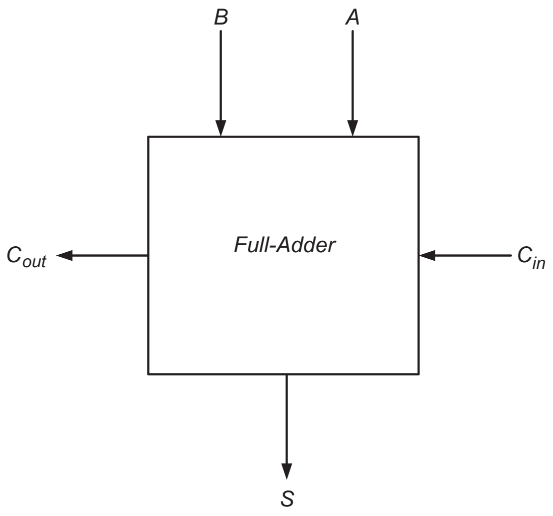


Figure 8.21 *FA* schematic symbol.

1 contains A_1, B_1, and S_1, and similarly for slices *2* and *3*. Slice *3* is the most significant slice of our numbers.

The number arrangement depicted by Equation (8.27) is exactly what we do when we perform an addition with paper and pencil. For now let us assume that there is no C_{in} into slice *0*. We begin the addition from the least significant slice by adding A_0 and B_0 to obtain S_0. This sum may produce a C_{out}* from slice *0* that has to propagate to slice *1*. To obtain the sum for slice *1,* or S_1 we must add C_{out} from slice *0*, which we will name as C_1, to A_1 and B_1. The process continues in the same fashion for all the slices. The last slice, slice *3* of our example, produces C_4, which is the overall C_{out} of the *4*-bit addition.

Example 8.7 actually is the justification for drawing the *FA* inputs and output the way that they are shown in Figure 8.21. Having gone over the algorithm of Example 8.7 we can easily interconnect four FA's to build a 4-bit binary adder. This time however since all *FA's* have the same logic, there will an overall C_{in} to slice *0* and we will refer to it a C_0. Figure 8.22a depicts an interconnection of full-adders that constitute a *4*-bit binary adder. Figure 8.22b shows a more compact manner of showing a 4-bit binary adder. Figure 8.22b does not imply in any way how the adder is internally designed. It can be built with *FA's* or other type of logic. Finally Figure 8.22c depicts the most compact form of all three of representing a *4*-bit binary adder. These types of symbols are very convenient to use when we deal computer architecture issues and micro controllers in general.

* C_{out} is always produced by a preceding slice regarding of its value (0 or 1); unless fast carry logic is used.

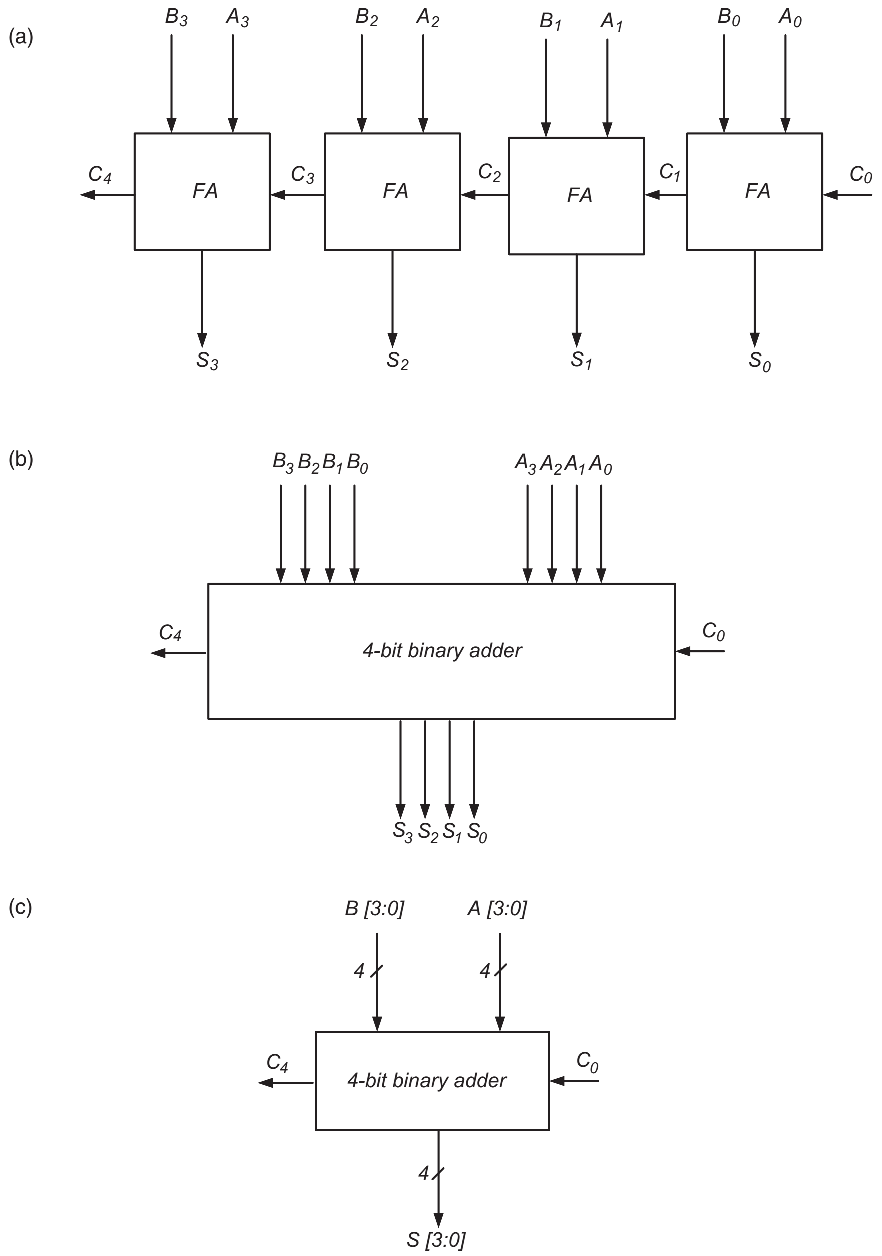


Figure 8.22 (a) *4*-bit binary adders built with *FA*; (b) compact form of a *4*-bit binary adder; (c) an even more compact form of a *4*-bit binary adder.

8.5.2 Notes about Full-Adder Timing

Let us assume a *FA* just like the one depicted in Figure 8.21. When all inputs are applied to the *FA's* simultaneously, there are two delays; one of them is the delay that it takes for output S before to settle down to a valid value. The other delay is the time that it takes the C_{out} output to settle down and become valid. Since it takes some longer time for C_{out} to settle to a valid value, C_{out} practically becomes the gating factor or the slow timing path, for the *complete* sum to be ready. The *complete* sum refers to the availability of valid values for S and for C_{out}. Since S is valid a little earlier that C_{out}, we then say that C_{out} is the long path in the sum. Now let us call this longer delay the full-adder delay, which at the moment we do not care about its absolute value in nano-seconds. When we build a *4*-bit adder like the one shown in Figure 8.22a note that we now call the complete sum the availability of all outputs of the 4-bit adder, i.e. valid S_3, S_2, S_1, S_0, and C_4, which is also the overall carry out of the adder. From the *4*-bit adder point of view we do not care (to a point) about the availability of valid internal carries that propagate through the adder. We do care about them from the perspective that the longer it takes for those carries to propagate through the internal logic the longer it will take to obtain the complete sum. For the *4*-bit adder of Figure 8.22a, implemented with full-adders, the overall adding time is four full-adder delays. So from the time the last input becomes valid at the adder input, we need to wait for the result sum and carry out for four full-adder delays. Should we take the sum reading before such time there is no guarantee about its correctness.

8.5.3 Subtracting with a *4*-bit Adder Using *1's* Complement Representation

Let us continue our example with *4*-bit wide *1's c*omplement numbers. Let us also recall from an earlier section of this chapter that if we need to subtract B from A, where A is the minuend and B is the subtrahend, this can be accomplish using 1's complement arithmetic by adding the *1's* complement of B, the subtrahend to A, the minuend, await for the overall output carry of the 4-bit adder to become valid and add it back into the input carry of the adder. This last step is referred to as adding the End-Around-Carry *(EAC)*. Figure 8.23 shows how that implementation is done.

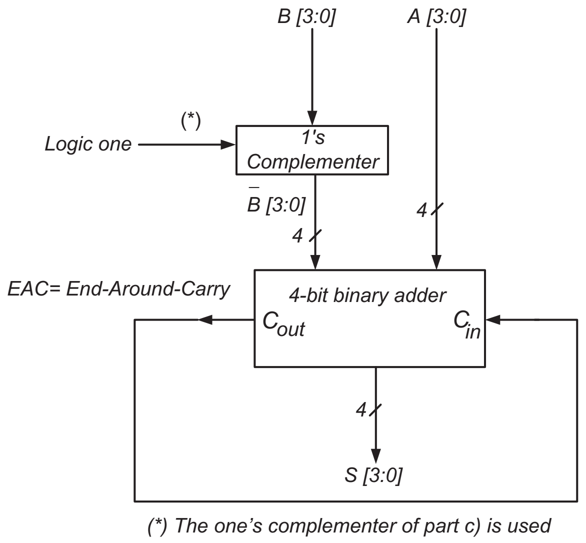
Referring to Figure 8.23a observe that the carry out of the *4*-bit adder is tied back into the carry in. The subtraction is performed as the sum of A with the *1's* complement of B plus any end-around-carry (*EAC*):

$$A - B = A + (-B) = A + 1\text{'s } Complement\ \{B\} + EAC. \qquad (8.28)$$

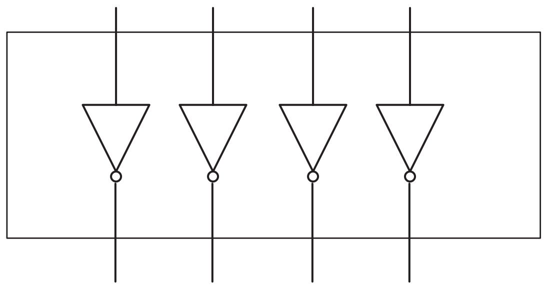
Note that if we deal with 4-bit wide numbers then $A = A[3{:}0]$ and $B = B[3{:}0]$.

Because of the need to add the *EAC*, we have to wait to obtain the complete subtraction, which is two complete *4*-bit adder delays. Figure 8.23b implement

(a)



(b)



(c)

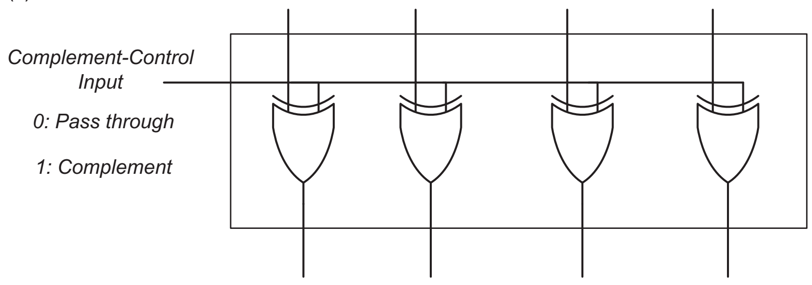


Figure 8.23 (a) 4-bit binary adder configured as a subtractor using *1's* complement arithmetic, (b) hardwired logic for a *1's* complementer, (c) programmable logic for a *1's* complementer. (*) When the complement-control input signal (c) is high, the 4-bit input number at the *1's* Complementer logic will become *1's* Complemented at the *1's* Complementer 4-bit output. When the complement-control input signal (c) is low, the 4-bit input number at the *1's* Complementer logic will pass-through the *1's* Complementer logic to its 4-bit output unchanged.

the simplest possible *1's* complementer, which is just a bit-to-bit inverter. Such one's complementer is said to be *hardwired*. Figure 8.23c shows an implementation using *XOR* gates, why? This implementation allows one to use the same logic as pass-through logic when its control input is zero and it converts the logic into a *1's c*omplementer when the control input is a one. That means that the logic block depicted by Figure 8.23a could also be used as an adder and not just as a subtractor. Of course for this scheme to be complete we should multiplex or gate the end around carry, opening the *EAC* path when we configure the logic as an adder and provide a path for the output carry to feed into the input carry when we configure it as a subtractor. For the sake of practicality, we simply now move on to the *2's* complement adder/ subtractor which is the most effective way of implementing and adder and a subtractor using a *4*-bit adder.

8.5.4 Subtracting with a *4*-bit Adder Using *2's* Complement Representation

The *1's* complement adder/subtractor is interesting but it is not fast enough. We can do better if we do not have to wait for the carry out to travel its way to the carry in (*EAC*). So the *2's c*omplement version of the adder/ subtractor is presented in Figure 8.24. Note the *EAC* path, seen for the *1's* complement implementation is now not connected for the 2's complement implementation.

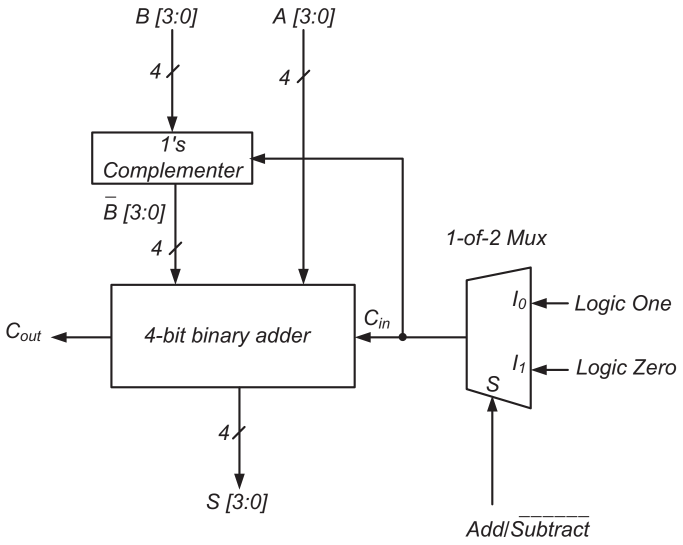


Figure 8.24 2's Complement *4*-bit binary adder/subtractor.

It is important to remember that when using *2's c*omplement arithmetic the carry out has to be ignored.

Example 8.8 Draw a block diagram of a *4*-bit *2*'s complement adder/subtractor that upon its control input being *0* it adds, but if the control input is *1* it subtracts using 2's complement arithmetic. Equation (8.17) repeated below for the reader's convenience shows the algorithm used to obtain the 2's complement of an *n*-bit binary number

$$2\text{'}s\ c(X) = 1\text{'}s\ complement\ (X) + 1 \text{ ignoring the Carry out bit.} \tag{8.29}$$

Figure 8.24 below depicts the solution to Example 8.8.

Referring to Figure 8.24 let us see how the adder part works. Upon setting to zero the control input to the select line of the *1-of-2* mux, the overall adder C_{in} is zero and the *1's* complementer logic box is in pass-through mode, i.e. does not invert its inputs. The logic of Figure 8.24 simply adds with a *high-level* control input. When the control input is zero two things happen, the C_{in} is set to *one* and the one's complementer logic box control input is also set to *one*. The *1's* complementer is set to complement mode and since the C_{in} to the adder/subtractor is set to *1*, the logic is basically executing:

$$A - B = A + (-B) = A + 1\text{'}s\ C\{B\} + 1 = A + 2\text{'}s\ C\{B\}. \tag{8.30}$$

Equation (8.30) in effect performs the subtraction of *B* from *A* in *2's* complement form.

8.6 CARRY LOOK AHEAD (CLA) OR FAST CARRY GENERATION

Let us now go back to our full-adder basic building block with its two logic equations, repeated below for the reader's convenience:

$$S = A \oplus B \oplus C_{in} \tag{8.31}$$

$$C_{out} = AB + (A \oplus B)C_{in}. \tag{8.32}$$

Since a basic multi-bit adder can be thought as a concatenation of full-adders, let us generalize Equations (8.31) and (8.32) for a *FA slice*. So let us re-write Equations (8.31) and (8.32) as if they were the equations of the *i*th *slice.*

$$S_{i+1} = A_i \oplus B_i \oplus C_i \tag{8.33}$$

$$C_{i+1} = A_i B_i + (A_i \oplus B_i)C_i \tag{8.34}$$

where in Equation (8.33) we can appreciate that C_i is the C_{in} to the *i*th slice in question produced by its immediately less significant and adjacent slice, or slice $(i-1)_{th}$. Similarly with C_i on Equation (8.34); and C_{i+1} is the C_{out} of the i_{th} slice.

We will describe a 4-bit adder, which has four slices, slice *0* is the least significant slice and *3* is the most significant slice. Personalizing Equations (8.33) and (8.34) for each of our four slices we obtain the following logic expressions, where we are assuming that *slice 0* is the least significant slice and *slice 3* is the most significant slice of our *4*-bit adder.

For all sum bits:

$$\textit{Slice 0}: S_0 = A_0 \oplus B_0 \oplus C_0 \tag{8.35}$$

$$\textit{Slice 1}: S_1 = A_1 \oplus B_1 \oplus C_1 \tag{8.36}$$

$$\textit{Slice 2}: S_2 = A_2 \oplus B_2 \oplus C_2 \tag{8.37}$$

$$\textit{Slice 3}: S_3 = A_3 \oplus B_3 \oplus C_3 \tag{8.38}$$

and for all the carryouts we obtain:

$$\textit{Slice 0}: C_1 = A_0B_0 + (A_0 \oplus B_0)C_0 \tag{8.39}$$

$$\textit{Slice 1}: C_2 = A_1B_1 + (A_1 \oplus B_1)C_1 \tag{8.40}$$

$$\textit{Slice 2}: C_3 = A_2B_2 + (A_2 \oplus B_2)C_2 \tag{8.41}$$

$$\textit{Slice 3}: C_4 = A_3B_3 + (A_3 \oplus B_3)C_3. \tag{8.42}$$

By close inspection of Equations (8.39) through (8.42) we see that since real logic gates have non-zero gate delays, C_1 has to be generated by the C_{out} logic of *slice 0* before the addition can proceed to *slice 1*. Similarly C_2 has to be generated by *slice 1* C_{out} logic before the addition can proceed to *slice 2*. Exactly the same is true for C_3 and for C_4.

Let us now make a couple of definitions, let the A_i B_i be *Generate*$_i$ or G_i terms for *i* ranging from *0* to *3*. We will also define the term $(A_i \oplus B_i)$ as the *Propagate*$_i$ term or P_i for *i* ranging from *0* to *3*. With those new definitions we rewrite Equations (8.39) through (8.42) and we obtain:

$$\textit{Slice 0}: C_1 = G_0 + P_0C_0 \tag{8.43}$$

$$\textit{Slice 1}: C_2 = G_1 + P_1C_1 \tag{8.44}$$

$$\textit{Slice 2}: C_3 = G_2 + P_2C_2 \tag{8.45}$$

$$\textit{Slice 3}: C_4 = G_3 + P_3C_3. \tag{8.46}$$

Plugging C_1 from Equation (8.43) into Equation (8.44) yields:

$$\textit{Slice 1}: C_2 = G_1 + P_1G_0 + P_1P_0C_0. \tag{8.47}$$

Plugging Equation (8.47) into Equation (8.45) yields:

$$\textit{Slice 2}: C_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0. \tag{8.48}$$

Plugging Equation (8.48) into Equation (8.46) yields:

$$\textit{Slice 3}: C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0. \tag{8.49}$$

Now let us carefully look at Equations (8.43), (8.47), (8.48), and (8.49). Note that each one of those equations, regardless of the number of terms and the number of inputs per *AND* gate, are all in *SOP* form, which means that they can all be implemented in just two levels of logic. Refer to Figure 8.25 and look at the logic implementations that produce carryouts: C_1, C_2, C_3, *and* C_4.

Using the defined generate G_i and propagate P_i terms in Equations (8.43), (8.47), (8.48), and (8.49) these can be rewritten as:

$$\textit{Slice 0}: S_0 = G_0 \oplus C_0 \tag{8.50}$$

$$\textit{Slice 1}: S_1 = G_1 \oplus C_1 \tag{8.51}$$

$$\textit{Slice 2}: S_2 = G_2 \oplus C_2 \tag{8.52}$$

$$\textit{Slice 3}: S_3 = G_3 \oplus C_3. \tag{8.53}$$

So based on Equations (8.43), (8.47) through (8.49), and Equations (8.50) through (8.53), the picture of a *4*-bit adder with *carry look ahead* or *fast carry generation* is depicted in Figure 8.25.

Note that the adder has four clearly marked areas with horizontal dotted lines. Each area is a slice of the adder. Now for the sake of simplicity of the timing analysis that we will get into, assume that all logic gates on the diagram have the same propagation delay. This is not true, but the assumption simplifies the analysis without us getting lost in the details. Note that this adder (Fig. 8.25) unlike the adder implemented with full-adders, (Fig. 8.22a), does not have a carry that propagates from the least to the most significant slice.

The adder with fast carry look ahead produces each slice's carryout with three levels of logic gates. Carefully observe that the first level is the one that generates the P_i and G_i terms, the *AND* gates are the second level and the collecting *OR* gate is the third level. Finally is easy to observe that the sum bit requires one additional logic gate delay *(exclusive OR)* to produce the S_i bit. The above statement is true for the entire adder, slices 0 through 3. It is also important to mention that the same adder with carry look ahead can be used to implement a subtractor by adding the *1's* complementing logic to the subtrahend. We could continue to discuss fast adders but because of space reasons, we refer the reader to the references at the end of this chapter.

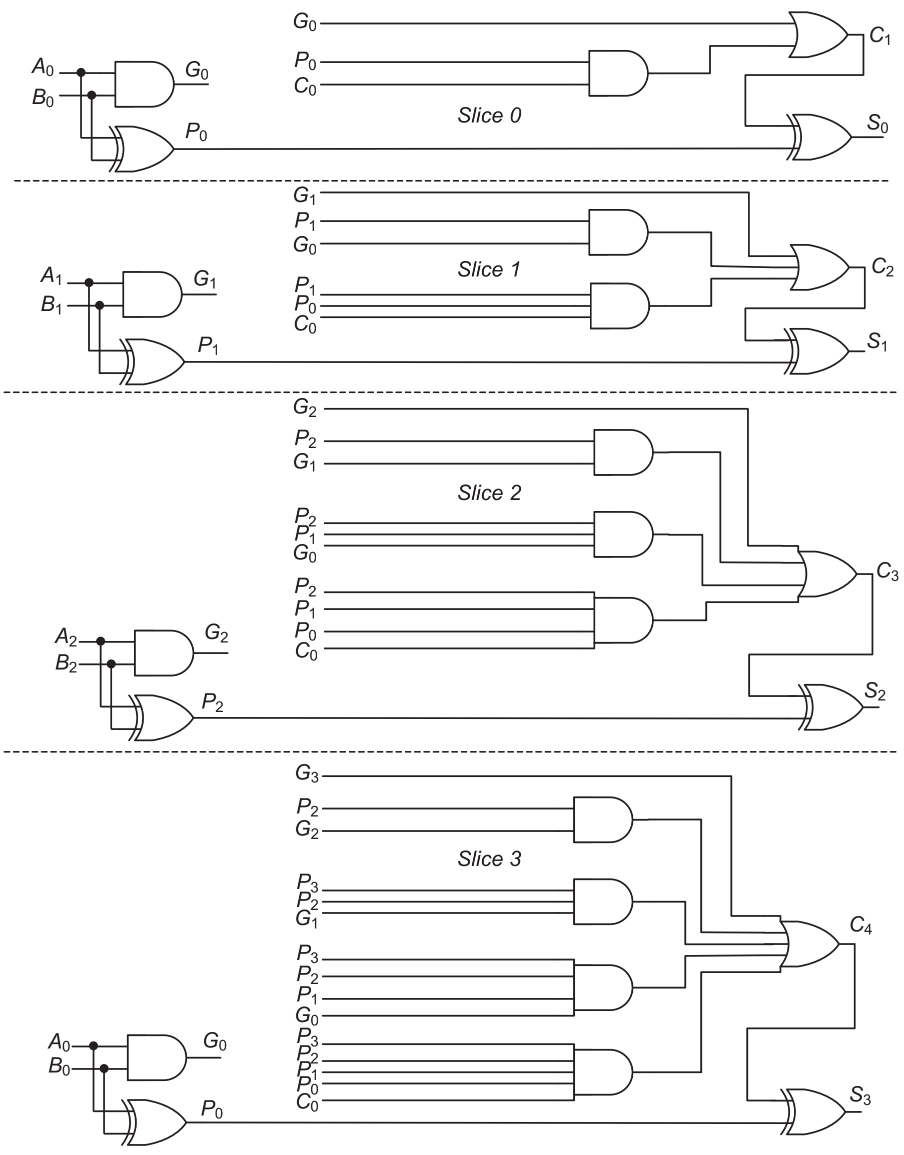


Figure 8.25 Logic implementation of a 4-bit adder with fast carry generation.

8.7 SOME SHORT-HAND NOTATION FOR LARGE LOGIC BLOCKS

We covered several logic blocks like decoders, multiplexers and adders; in most cases we drew every bit of the logic block in an explicit fashion. For example look back at Figures 8.13a and 8.15. In computer architecture literature is common to see a very compact way of drawing multi-bit devices.

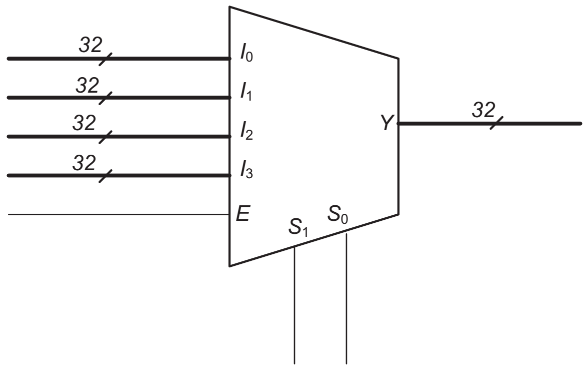


Figure 8.26 Compact graphic representation of a *32*-bit wide *1-of-4* multiplexer.

Assume we need a *1-of-4* multiplexer where each of its input is *32* bits wide. If we drew such device bit by bit it would look cumbersome and hard to read. Instead we draw a single line for each one of the *1-of-4* mux inputs and indicate with a short crossed line the number of bits the mux *leg* has. Note that output *Y* is a 32-bit wide output.

Figure 8.26 is a representation of the *32*-bit wide *1-of-4* multiplexers in compact notation.

Note that such 32-bit wide mux can be constructed with thirty-two regular *1-of-4* multiplexers.

Exercise: Get a large piece of paper and draw a detailed explicit drawing for the 32-bit 1-of-4- mux implemented with 32 individual 1-of-4 muxes.

We draw in a similar fashion very wide adders and subtractors. In the subsequent chapter we will use this compact notation more liberally which we will see that it also applies to registers, counters, and state machines. These are all sequential circuits (circuit with memory capabilities) not covered yet.

8.8 SUMMARY

This chapter covers a good number of combinational circuits: decoders, multiplexers, and arithmetic combinational circuits such as adders and subtractors.

The purpose of this chapter is to make the reader feel at ease designing virtually any simple or complicated combinational circuit by learning how to establish their truth table, how to do the logic simplification, and in some cases, how to partition big circuits into smaller ones to simplify the methodology or even clarify their behavior.

For in-depth coverage of arithmetic circuits the reader is referred to Reference [1].

FURTHER READING

1. Shlomo Waser and Michael J. Flynn, *Introduction to Arithmetic for Digital System Designers*, Holt, Rinehart and Winston, New York, 1982.
2. Robert L. Morris and John R. Miller, *Designing with TTL Integrated Circuits, Texas Instruments Electronic Series*, McGraw-Hill, New York, 1971.
3. Morris Mano, *Digital Design*, Prentice Hall, Upper Saddle River, NJ, 1984.

PROBLEMS

8.1 (a) Define the truth table for a logic circuit that given a three-bit positive binary number input, produces a four-bit output that equals the initial 3-bit number plus 7. This circuit has three inputs and four outputs; (b) Obtain the simplified SOP form of the four outputs; and (c) draw the circuit.

8.2 (a) Define the truth table for a logic circuit that given a four-bit binary number, produces a four-bit output that equals the initial 4-bit number 1's Complemented. This circuit has four inputs and four outputs; (b) Obtain the simplified SOP form of the four outputs; and (c) draw the circuit.

8.3 Implement the logic found for Problem 8.1 with the smallest possible multiplexer and any inverter gates as needed.

8.4 Implement each piece of logic found for Problem 8.2 with the smallest possible multiplexer and any inverter gates as needed.

8.5 Implement a 3-variable XOR function with the smallest possible multiplexer and inverter gates as needed.

8.6 Implement with the smallest possible decoder and minimal number of OR gates a 4-variable XOR function.

8.7 Without simplifying the logic, implement the following function entirely with NAND gates:

$$f(A,B,C,D) = A.B.C + \overline{A.B.D} + A.\overline{B.D}$$

8.8 Without simplifying the logic, implement the following function entirely with NOR gates:

$$f(A,B,C,D) = A.B.C + \overline{A.B.D} + A.\overline{B.D}$$

8.9 Write the truth table for a 4-bit positive binary adder with no carry in and no carry out. Implement and draw the circuit of the 4-bit input and

4-bit output positive binary adder with the smallest number of multiplexers of the smallest size possible and inverter gates as needed.

8.10 Implement and draw the circuit of the adder of Problem 8.9 with the smallest size and smallest possible number of decoders and OR gates as needed.

8.11 Draw the circuit of a two 4-bit input binary adder with carry in and carry out with fast carry-look-ahead logic. This circuit has 2 4-bit inputs, one overall carry-in input, one 4-bit output and one overall carry-out output. Use 4 single bit full-adders in addition to the carry-look-ahead logic.

8.12 Design and draw the circuit of a 1-of-8 multiplexer using only 1-of-2 multiplexers. Write the truth table of the circuit to-be-designed.

8.13 Design and draw the circuit of a 1-of-16 multiplexer using some 1-of-2 multiplexers and some 1-of-4 multiplexers. Write the truth table of the circuit to-be-designed.

8.14 Given a 4-bit 2's Complement binary number design the logic that produces the absolute value of the 2's Complement input. This circuit shall have a 4-bit input and 3-bit output. Write the truth table of the circuit to-be-designed.

8.15 Design a combinational circuit that given a 4-bit input produces the bit-to-bit OR of each one of the 16 binary input combinations. This circuit shall have 4 inputs and one output and must be implemented with the smallest possible decoder and OR gates if needed. Write the truth table of the circuit to-be-designed.

8.16 Implement a 3-bit XOR logic block entirely with NAND gates.

8.17 Implement a 3-bit XOR logic block entirely with NOR gates.