Logic Design with MSI Circuits

There are several specialized MSI components that have extensive use in digital systems.

These are classified as *standard* components.

These include adders, subtractors, comparators, decoders, encoders and multiplexers.

1. Binary Adder – Subtractor

The most basic arithmetic operation is the addition of two binary digits. This simple addition consists of four possible elementary operations:

$$0+0=0$$
$$0+1=1$$
$$1+0=1$$
$$1+1=10$$

The first three operations produce a *sum* (*S*) of one digit, but when both augend and addend bits are equal to 1 a *carry* (*C*) is also generated (this propagates to the next most significant stage of the addition).

1.1 Half Adder

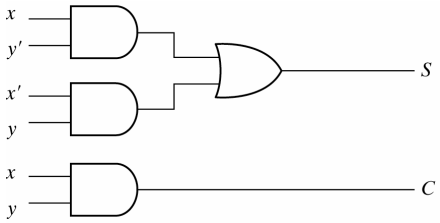
Performs the addition of two bits.

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = x'y + xy'$

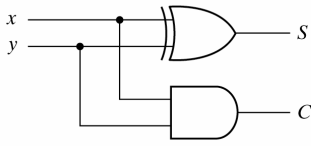
$C = xy$

Implementation:



$S = x'y + xy'$

$C = xy$



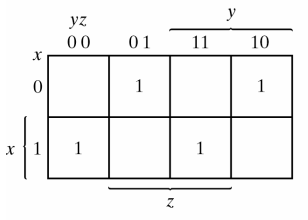
$S = x \oplus y$

$C = xy$

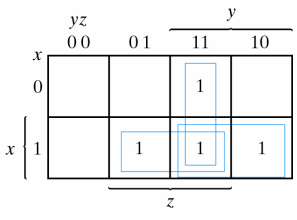
1.2 Full Adder

Performs the arithmetic sum of three bits.

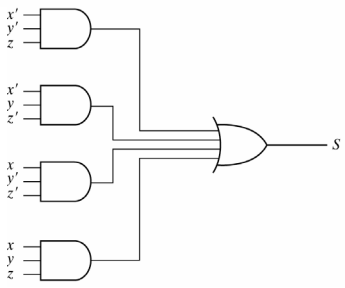
x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

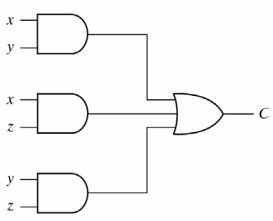


$S = x'y'z + x'yz' + xy'z' + xyz$



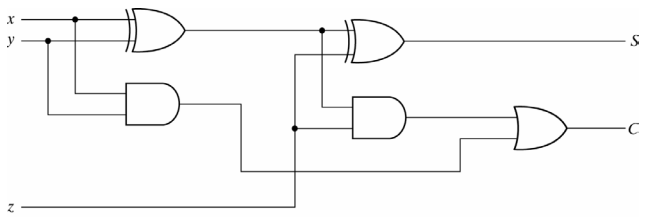
$C = xy + xz + yz$
$= xy + xy'z + x'yz$





Alternative Implementation:

The full adder can be also realized with two half adders and one OR-gate:



The output S from the second half adder is the X-OR of *z* and the output of the first half adder, giving:

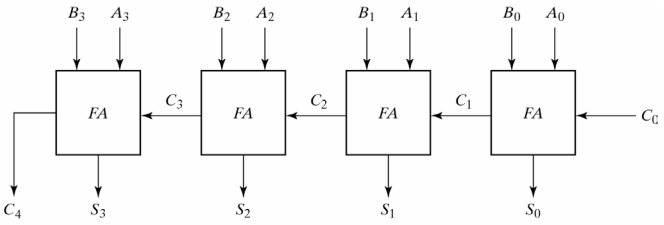
$$S = z \oplus (x \oplus y)$$
$$= z'(xy' + x'y) + z(xy' + x'y')'$$
$$= z'(xy' + x'y) + z(xy + x'y')$$
$$= xy'z' + x'yz' + xyz + x'y'z$$

The carry output is:

$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$

1.3 Binary Adder

Produces the arithmetic sum of two binary numbers. It can be realized with full adders (FAs) connected in cascade. A 4-bit binary *ripple* adder is realized as shown below:



An *n*-bit adder requires *n* full adders with each output carry connected to the input carry of the next higher-order full adder.

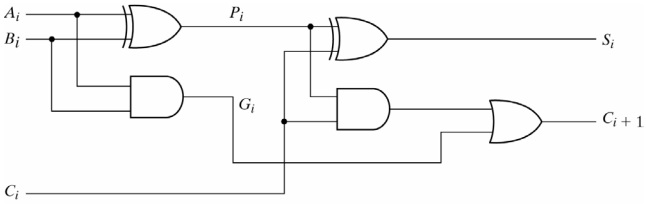
Example: Consider the two binary numbers, A = 1011 and B = 0011. Their sum S = 1110 is formed with the 4-bit adders as follows:

Subscript i:	3	2	1	0	
Input carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}

1.4 Carry Propagation

The longest propagation time in a binary ripple adder is the time it takes the carry to propagate through all full adders.

The number of gate levels for the carry propagation can be found from the circuit of the full adder:



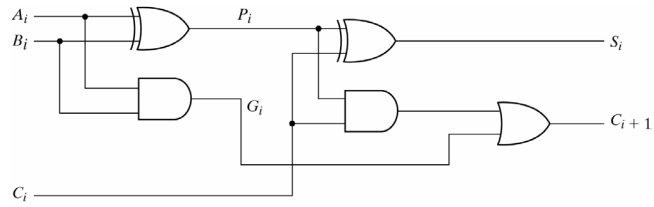
The subscript *i* denotes a given stage in the adder. The signals P_i and G_i settle to their steady state values after they propagate through their gates. P_i and G_i are common to all full adders and depend only on the input augend and addend bits.

The signal from the input carry C_i to the output carry C_{i+1}, propagates through two gates. If there are four full adders, the output carry C_4 would have 2 X 4 = 8 gate levels from C_0 to C_4.

Clearly, carry propagation time the limiting factor on the speed with which two numbers are added.

The most widely used technique for reducing the carry propagation time in a parallel adder employs the principle of *carry lookahead*.

Consider again the circuit of the full adder:



If two new binary variables are defined:

$$P_i = A_i \oplus B_i$$
$$G_i = A_i B_i$$

the output sum and carry can be expressed as:

$$S_i = P_i \oplus C_i$$
$$C_{i+1} = G_i + P_i C_i$$

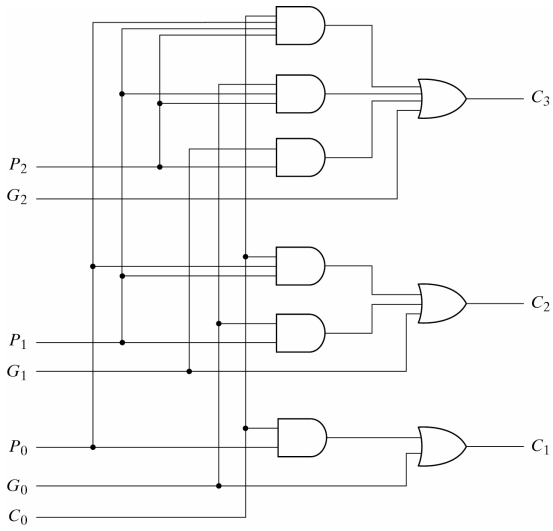
G_i is called a *carry generate* and it produces a carry of 1, regardless of the input carry C_i.

P_i is called a *carry propagate* because it is the term associated with the propagation of the carry from C_i to C_{i+1}.

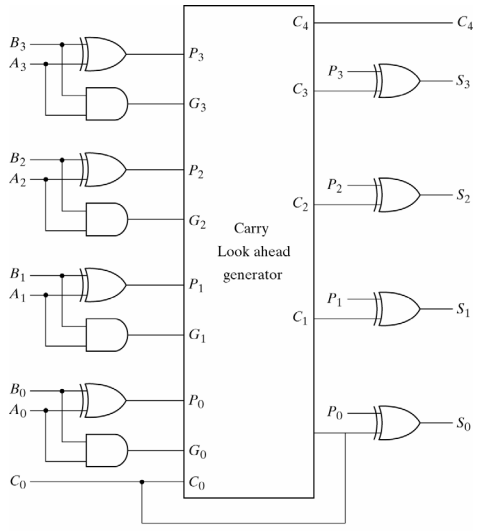
We can now write the Boolean functions for the carry outputs of each stage and substitute for each C_i its value from the previous equations:

$$C_0 = \text{input carry}$$
$$C_1 = G_0 + P_0 C_0$$
$$C_2 = G_1 + P_1 C_1 = G_1 + P_1(G_0 C_0) = G_1 + P_1 G_0 + P_1 P_0 C_0$$
$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

These expressions are implemented in the following carry lookahead generator:



The construction of a 4-bit adder with a carry lookahead scheme is shown below:



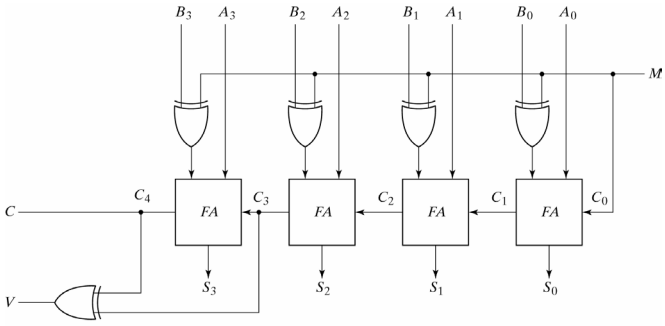
All output carries are generated after a delay through two levels of gates. Thus outputs S_1 through S_3 have equal propagation times.

The two level circuit for the output carry C_4 is not shown. This can be derived by the equation-substitution method.

1.5 Binary Subtractor

The subtraction of unsigned binary numbers can be simplified by means of *complements*. For example, $A - B$ can be done by taking the 2's complement of B and adding it to A. The 2's complement can be obtained by taking the 1's complement and adding 1 to the least significant pair of bits. The 1's complement can be realized with inverters and a 1 can be added to the sum through the input carry.

A 4-bit adder-subtractor circuit is shown below:



M = 0; addition M = 1; subtraction

The V bit detects an overflow when the two binary numbers to be added are signed.

1.6 Overflow

Sometimes, when an adder/subtractor is using signed arithmetic, there is arithmetic *overflow* from the most significant magnitude bit into the sign bit. An overflow may occur if the two numbers added are both positive or negative.

An example of 4 possible situations that may arise is given below for a 4-bit (n = 4) word. For each case the carries C_{n-1} and C_n are recorded:

	C_nC_{n-1}		C_nC_{n-1}	
	0 0		0 1	→ interpreted
+1	0,001	+5	0,101	as −5
+3	0,011	+6	0,110	
+4	0,100	+11	1,011	

	C_nC_{n-1}		C_nC_{n-1}	
	1 1		1 0	→ interpreted
+5	0,101	−5	1,011	as +5
−3	1,101	−6	1,010	
+2	0,010	−11	0,101	

In the case where the sum should +11 or -11, the corresponding binary sum is wrong. It is obvious that an overflow flag should be raised when C_{n-1} = 1 and C_n = 0, or, when C_{n-1} = 0 and C_n = 1. Hence, the equation for overflow is:

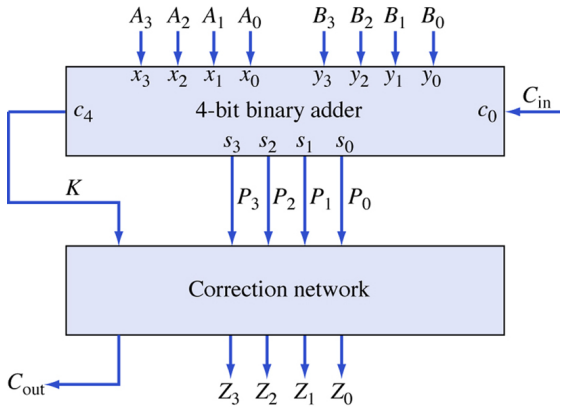
$$V = C_{n-1} \oplus C_n$$

2. Decimal Adder

Computers or calculators that perform arithmetic operations directly in the decimal number system represent decimal numbers in *binary-coded* form.

The 8421 weighted coding scheme is the most commonly occurring in digital systems and is often referred to as simply BCD (binary-coded decimal).

When using BCD, a single-decade decimal adder can be realized by first performing conventional binary addition on two binary-coded operands and then applying a corrective procedure. This is illustrated below:



Since each operand digit does not exceed 9, the output sum cannot be greater than 9 + 9 +1 = 19, the 1 in the sum being an input carry.

The various outputs from the 4-bit binary adder and the required outputs from the single-decade decimal adder are summarized in the table below:

	Binary sum					Required BCD sum				
Decimal sum	K	P_3	P_2	P_1	P_0	C_{out}	Z_3	Z_2	Z_1	Z_0
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	1
2	0	0	0	1	0	0	0	0	1	0
3	0	0	0	1	1	0	0	0	1	1
4	0	0	1	0	0	0	0	1	0	0
5	0	0	1	0	1	0	0	1	0	1
6	0	0	1	1	0	0	0	1	1	0
7	0	0	1	1	1	0	0	1	1	1
8	0	1	0	0	0	0	1	0	0	0
9	0	1	0	0	1	0	1	0	0	1
10	0	1	0	1	0	1	0	0	0	0
11	0	1	0	1	1	1	0	0	0	1
12	0	1	1	0	0	1	0	0	1	0
13	0	1	1	0	1	1	0	0	1	1
14	0	1	1	1	0	1	0	1	0	0
15	0	1	1	1	1	1	0	1	0	1
16	1	0	0	0	0	1	0	1	1	0
17	1	0	0	0	1	1	0	1	1	1
18	1	0	0	1	0	1	1	0	0	0
19	1	0	0	1	1	1	1	0	0	1

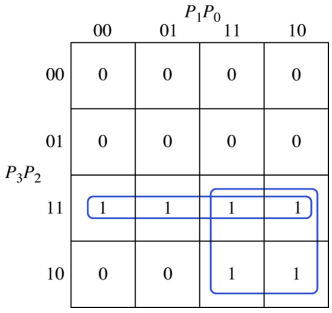
No correction to the binary sum is needed when $KP_3P_2P_1P_0 \leq 01001$.

However, 0110 (decimal 6) must be added to $P_3P_2P_1P_0$ when $KP_3P_2P_1P_0 > 01001$.

The logic circuit that detects the necessary correction can be derived from the table entries.

Clearly, a correction is needed when the binary sum has an output carry $K = 1$.

For the other six combinations from 1010 through 1111 (that also need a correction), a Boolean expression is required to detect them:



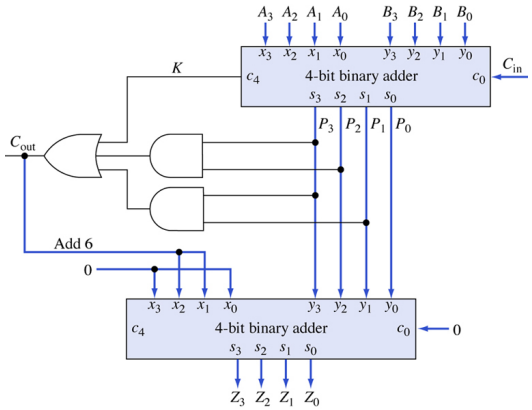
$$f = P_3P_2 + P_3P_1$$

The condition for a correction and an output carry can be expressed by the Boolean function:

$$\text{Add 6} = K + P_3P_2 + P_3P_1$$

The first term corresponds to the decimal sums 10 to 15 where the carry bit K is 1. The remaining two terms correspond to the decimal sums 16 to 19 where $K = 0$.

The logic diagram of a single-decade BCD adder is shown below:



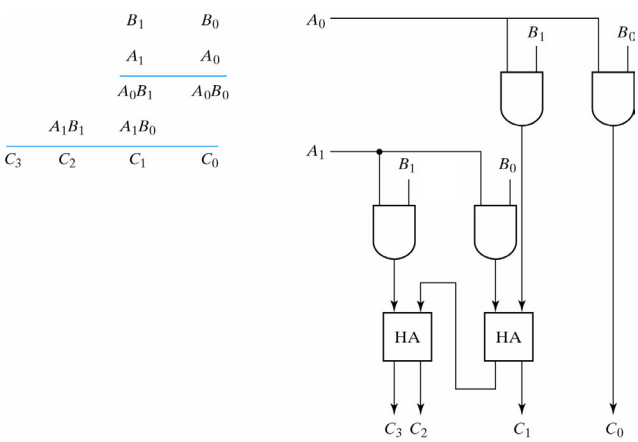
Whenever $C_{out} = 0$, the outputs from the upper 4-bit binary adder are sent to the lower 4-bit adder and decimal 0 is added.

However, whenever $C_{out} = 1$, decimal 6 is added to the outputs of the upper 4-bit binary adder so that the correct sum digit is obtained.

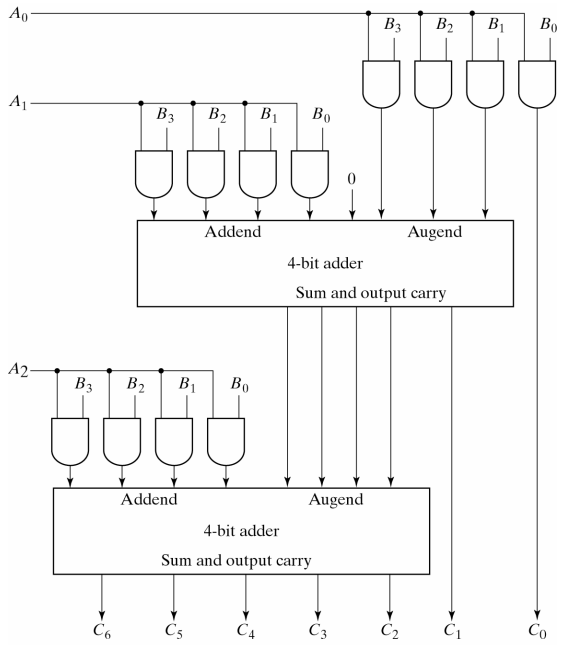
A decimal adder for two n-digit BCD numbers can be constructed by cascading the above system in much the same way as was done for the ripple binary adder.

3. Binary Multiplier

Multiplication of binary numbers is performed in the same way as in decimal numbers. The multiplicand is multiplied by each bit of the multiplier starting from the LSB. Each such multiplication forms a partial product. Successive partial products are shifted one position to the left. The final product is obtained from the sum of the partial products. A 2-bit by 2-bit binary multiplier is shown below. It is realized using AND-gates and two half adder (HA) circuits:



A combinational circuit binary multiplier with more bits can be constructed in a similar fashion. For J multiplier bits and K multiplicand bits (J X K) AND-gates and (J – 1) K-bit adders to produce a product of $J + K$ bits. The logic diagram of a 4-bit by 3-bit binary multiplier is shown below:



4. Magnitude Comparator

A circuit that compares two numbers, A and B, and determines their relative magnitudes.

4.1 1-bit Comparator

For this case, it is simply the X-NOR function:

$$f = AB + A'B'$$

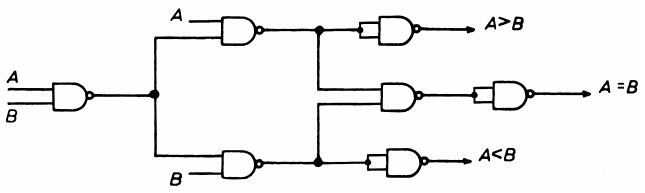
So the output is 1 if $A = B = 0$ <u>or</u> if $A = B = 1$. In addition to the equality relation, the outcome must indicate whether $A > B$, or $A < B$:

A	B	$A < B$	$A = B$	$A > B$
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

and from the table it is easy to show that:

$$A < B = A'B$$
$$A > B = AB'$$
$$A = B = A'B' + AB$$

with the following NAND-gate realization:



4.2 4-bit Comparator

In this case an algorithm is required. Let the words be:

$$A = A_3A_2A_1A_0$$
$$B = B_3B_2B_1B_0$$

The equality relation of each pair of bits can be expressed logically with the X-NOR function as:

$$x_i = A_iB_i + A_i'B_i' \quad \text{for } i = 0, 1, 2, 3$$

where x_i = 1 only if the pair of bits in position i are equal (i.e., if both are 1 or both are 0).

For the equality condition to exist, all x_i variables must be equal to 1. This dictates an AND operation of all variables:

$$(A = B) = x_3x_2x_1x_0$$

To determine whether $A > B$ or $A < B$, examine the relative magnitudes of pairs of significant digits starting from the most significant position:

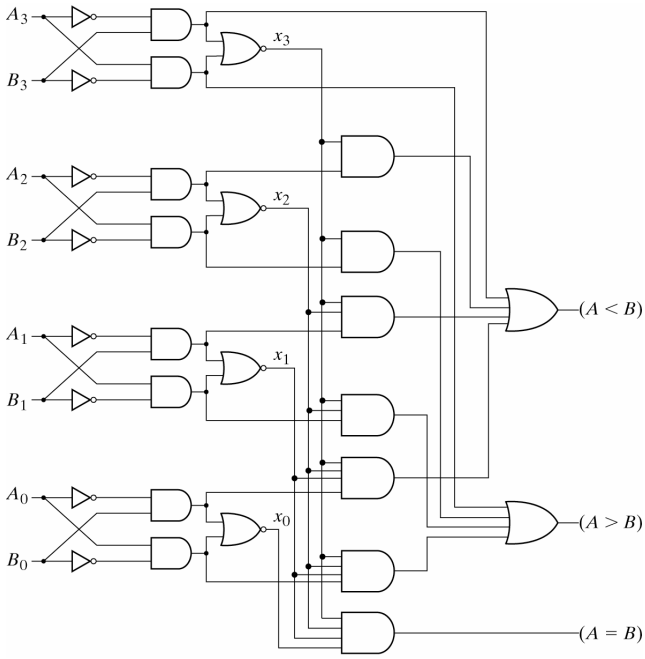
- If the two digits are equal, compare the next lower significant pair of digits. The comparison continues until a pair of unequal digits is found.
- If the corresponding digit of A is 1 and that of B is 0, conclude that $A > B$.
- If the corresponding digit of A is 0 and that of B is 1, conclude that $A < B$.

The sequential comparison can be expressed logically by the two Boolean functions:

$$(A > B) = A_3B_3' + x_3A_2B_2' + x_3x_2A_1B_1' + x_3x_2x_1A_0B_0'$$
$$(A < B) = A_3'B_3 + x_3A_2'B_2 + x_3x_2A_1'B_1 + x_3x_2x_1A_0'B_0$$

The symbols ($A > B$) and ($A < B$) are binary outputs that are equal to 1 when $A > B$ or $A < B$, respectively.

Finally, the logic diagram of the 4-bit magnitude comparator is as follows:

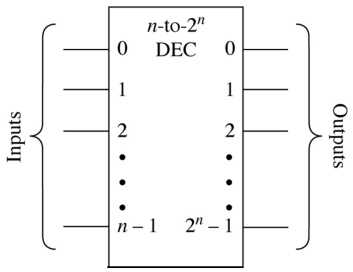


The four *x* outputs are generated with X-NOR circuits and applied to an AND gate to give the output binary variable (*A* = *B*).

The other two outputs use the *x* variables to generate the Boolean functions shown before.

5. Decoders

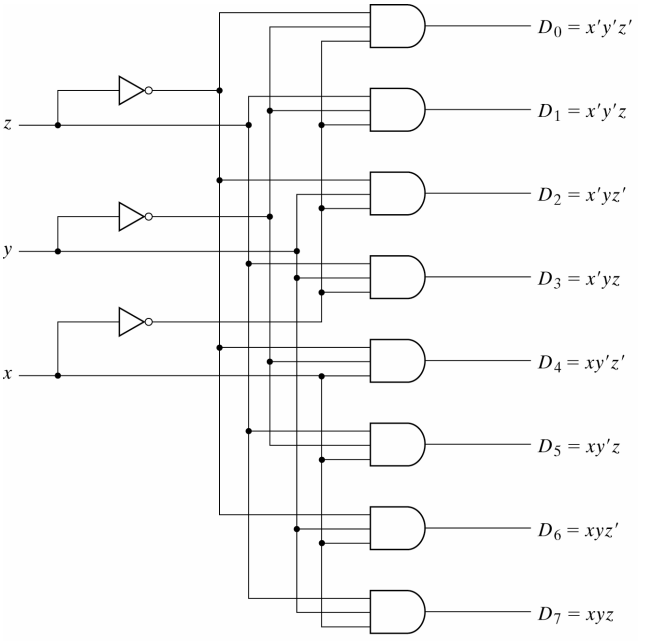
Often, digital information represented in dome binary form must be converted into some alternative digital form. This is achieved by a multiple-input, multiple output network referred to as a *decoder*. The most commonly used decoder is the *n-to-2^n-line decoder*:



The structure of a such decoder is straightforward. Consider the truth table of a 3-to-8-line decoder:

Inputs			Outputs							
x	*y*	*z*	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

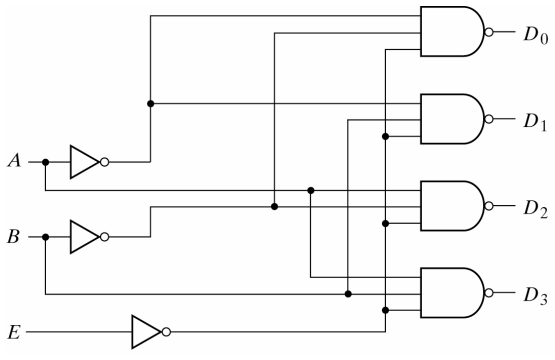
This corresponds to the logic diagram shown below:



A particular application for this decoder is binary-to-octal conversion. The input variables represent a binary number, and the outputs represent the eight digits in the octal number system.

5.1 Decoders with an Enable Input

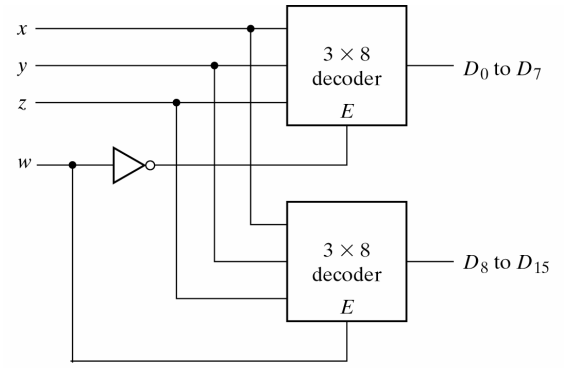
Some decoders include one or more enable inputs to control the circuit operation. The logic diagram and truth table of a 2-to-4-line decoder are shown below:



E	*A*	*B*	D_0	D_1	D_2	D_3
1	*X*	*X*	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

A decoder with enable input can function as a *demultiplexer*. The above decoder can function as a 4-to-1-line demultiplexer when *E* is taken as a data input line and *A* and *B* are taken as the selection inputs.

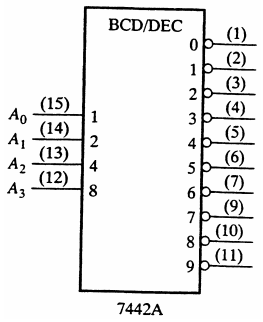
Decoders with enable inputs can be connected together to form a larger decoder circuit. A 4-to-16-line decoder realized using two 3-to-8-line decoders is shown below:

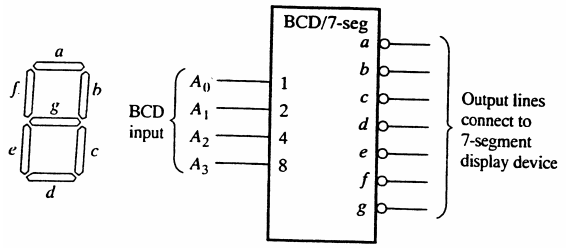


When w = 0, the top decoder is enabled and the other is disabled. The bottom decoder outputs are all 0's, and the top eight outputs generate minterms 0000 to 0111.

When w = 1, the enabled conditions are reversed; the bottom decoder generates minterms 1000 to 1111, while the outputs of the top decoder are all 0's.

The n-to-2^n-line decoder is only one of several types of decoders. *Function-specific* decoders exist having fewer than 2^n outputs. Examples include the BCD-to-decimal decoder (7442A) and the BCD-to-7-segment decoder.





5.2 Combinational Logic Implementation

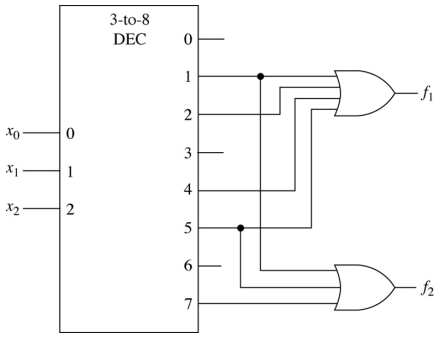
An n-to-2^n-line decoder is a *minterm generator*. Recall that any Boolean function is describable by a sum-of-minterms. Thus, by using OR-gates in conjunction with an n-to-2^n-line decoder realizations of Boolean functions are possible. However, these realizations do not correspond to minimal sum-of-products.

Consider the pair of expressions:

$$f_1(x_2, x_1, x_0) = \sum(1, 2, 4, 5)$$

$$f_2(x_2, x_1, x_0) = \sum(1, 5, 7)$$

Using a single 3-to-8-line decoder and two OR-gates, the following realization is obtained:



When more than ½ the total number of minterms must be OR-ed, it is usually more economical to use NOR-gates rather than OR-gates to do the summing. Consider the pair of expressions:

$$f_1(x_2, x_1, x_0) = \sum(0, 1, 3, 4, 5, 6)$$

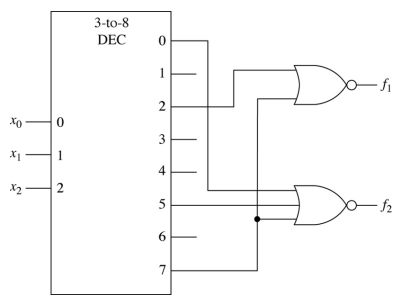
$$f_2(x_2, x_1, x_0) = \sum(1, 2, 3, 4, 6)$$

These may be realized with a 3-to-8-line decoder and two OR-gates having a total of 11 terminals between them. However, a more efficient realization is to re-write the expressions as:

$$f_1''(x_2, x_1, x_0) = f_1'(x_2, x_1, x_0) = \overline{\sum(2, 7)}$$

$$f_2''(x_2, x_1, x_0) = f_2'(x_2, x_1, x_0) = \overline{\sum(0, 5, 7)}$$

This corresponds to the realization shown below:



A total of five gate-input terminals are needed.

6. Encoders

Perform the inverse operation of decoders. An encoder has 2^n (or fewer) input lines and n output lines. The output lines generate the binary code corresponding to the input value. An example of an encoder is the *octal-to-binary* encoder whose truth table is as follows:

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

The equations for the three outputs are:

$$z = D_1 + D_3 + D_5 + D_7$$
$$y = D_2 + D_3 + D_6 + D_7$$
$$x = D_4 + D_5 + D_6 + D_7$$

The encoder can be realized with three OR-gates.

6.1 Priority Encoder

The encoder defined before has the limitation that only one input can be active at any given time. If two inputs are active simultaneously, the output produces an undefined combination. This is resolved by establishing an input priority function.

The truth table of a four-input priority encoder is:

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

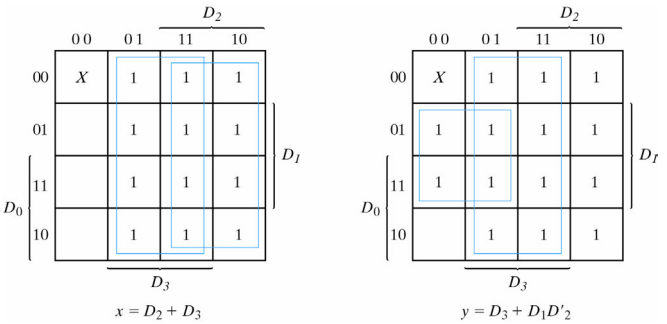
In addition to the two outputs, x and y, the circuit has a third output *V*; this is a *valid* bit indicator and is set to 1 when one or more inputs are equal to 1.

X's in the output represent *don't-care* conditions.

X's in the input columns are for representing the truth table in condensed form. Instead of listing all 16 minterms of four variables, the truth table uses an *X* to represent either 1 or 0.

According to the table, D_3 has the highest priority followed by D_2 and D_1.

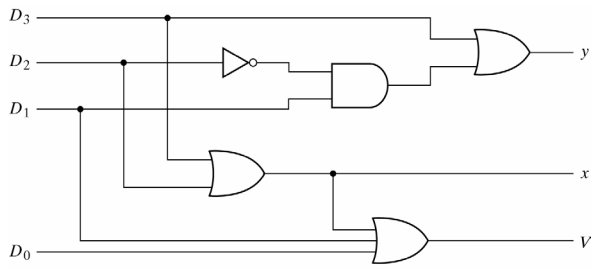
The maps for simplifying outputs *x* and *y* are shown below:



The condition for output *V* is an OR function of all the input variables:

$$V = D_0 + D_1 + D_2 + D_3$$

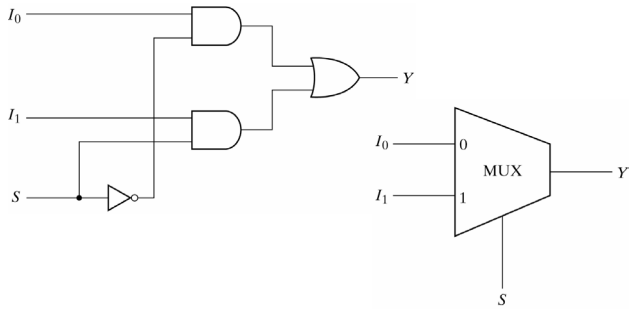
The priority encoder is implemented as follows:



7. Multiplexers

A multiplexer is a circuit that selects binary information from one of many input lines and directs it to a single output. Normally, there are 2^n input lines and *n* selection lines whose bit combination determine which input is selected.

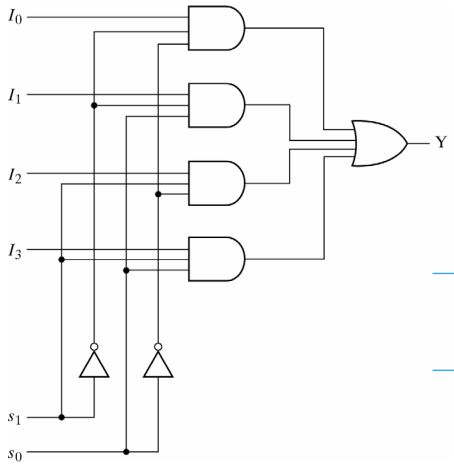
The logic and block diagrams of a 2-to-1-line multiplexer are shown below:



The circuit has two data input lines, I_1 and I_2, one output line *Y*, and one selection line *S*.

When *S* = 1, the lower AND gate is enabled and I_1 has path to the output. This multiplexer acts like a switch that selects one of the two sources.

A 4-to-1-line multiplexer is shown below:



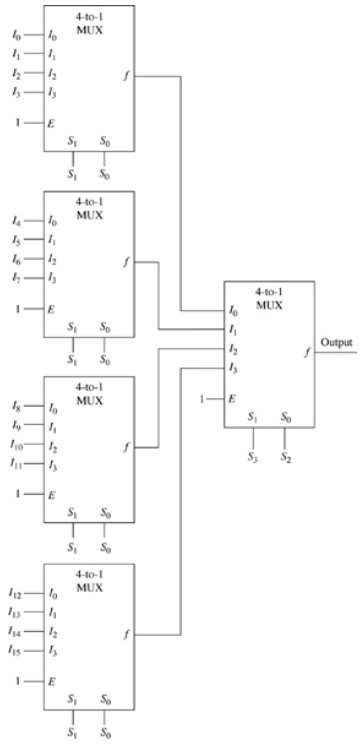
s_1	s_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

A multiplexer is also called a *data selector*, since it selects one of many inputs and steers the binary information to the output line.

In general, a 2^n-to-1-line multiplexer is constructed from an n-to-2^n decoder by adding to it 2^n input lines, one to each AND gate. The outputs of the AND gates are applied to a single OR gate.

As in decoders, multiplexers may have an *enable* input to control the operation of the unit.

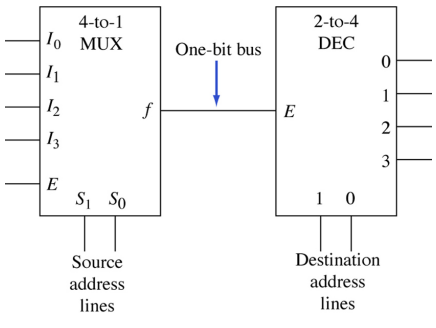
By interconnecting several multiplexers in a treelike structure, it is possible to produce a larger multiplexer. For example, a 16-to-1 line multiplexer may be constructed using five 4-to-1-line multiplexers as follows:



7.1 MUX/DeMUX Transmission System

One of the primary applications of multiplexers is to provide for the transmission of information from several sources over a single path. This process is known as *multiplexing*. E.g., the multiplexing of conversations on the telephone system.

When a multiplexer is used in conjunction with a demultiplexer, an effective means is provided for connecting information from several source locations to several destination locations. This basic application is illustrated below:



By using n of the structures shown above in parallel, an n-bit word from any of four source locations is transferred to the four destination locations.

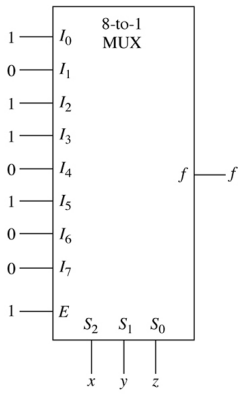
7.2 Logic Design with Multiplexers

Consider the Boolean function of three variables:

$$f(x, y, z) = \sum(0, 2, 3, 5)$$

The function can be implemented with an 8-to-1-line multiplexer:

x	y	z	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0



The realization is obtained by placing x, y, and z on the S_2, S_1, and S_0 lines respectively, logic-1 on data input lines I_0, I_2, I_3, and I_5 and logic-0 on the remaining data input lines. Also the multiplexer must be enabled by setting E = 1.

If at least one input variable of a Boolean function is assumed to be available in both its normal and complemented form, then any *n*-variable function can be realized with a 2^{n-1}-to-1-line multiplexer.

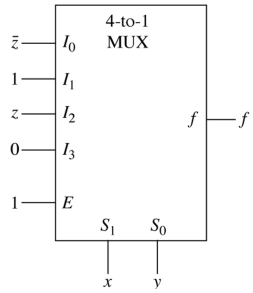
For example, reconsider the previous function:

$$f(x, y, z) = \sum(0, 2, 3, 5)$$
$$= x'y'z' + x'yz' + x'yz + xy'z$$

Doing some simple factoring becomes:

$$f(x, y, z) = x'y'(z') + x'y(z' + z) + xy'(z)$$
$$= x'y'(z') + x'y(1) + xy'(z) + xy(0)$$

which is realized using a 4-to-1-line multiplexer:



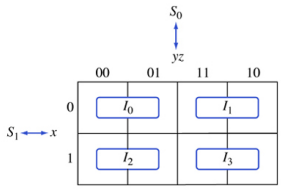
The last term, *xy*(0), was included to indicate what input must appear on the I_3 line to provide for the appropriate output when selected with *x* = *y* =1.

Karnaugh maps provide a convenient tool for obtaining multiplexer realizations. First it is necessary to establish which variables to assign to the select lines. Next the inputs for the I_i data lines are read directly from the map.

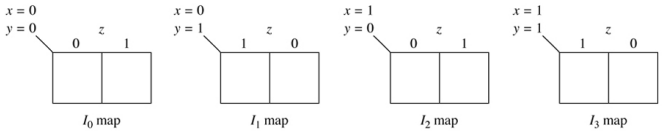
To illustrate this, again consider the three-variable function:

$$f(x, y, z) = \sum(0, 2, 3, 5)$$

Assume that *x* is placed on the S_1 line and *y* is placed on the S_0 line, the resulting map is:



submaps:

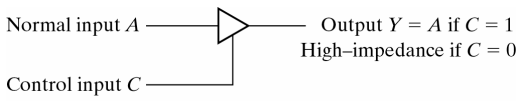


Grouping the 1-cells, the expressions for the sub-functions may be written. That is, $I_0 = z'$, $I_1 = 1$, $I_2 = z$, and $I_3 = 0$. The logic realization is as before.

7.3 Three-State Gates

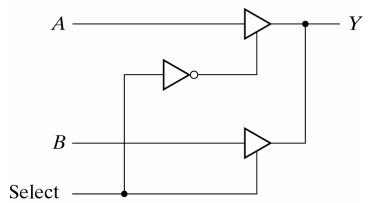
A multiplexer can be constructed with three-state gates. A three-state gate is a digital circuit that exhibits three states. Two of the states are signals equivalent to logic 1 and 0. The third state is a *high-impedance* state which behaves like an open circuit.

The graphic symbol of a three-state buffer is:

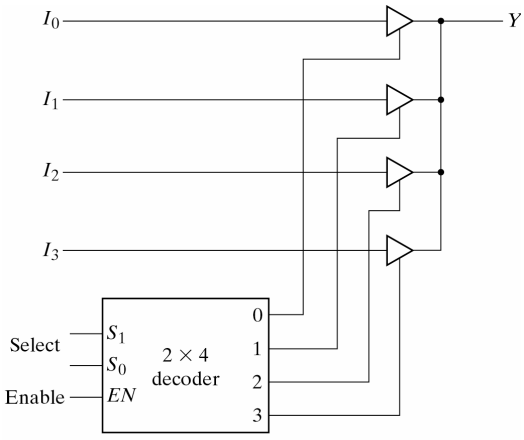


The presence of the high-impedance state allows the connection of a large number of three-state gate outputs to a common line without endangering loading effects.

The realization of a 2-to-1-line multiplexer with three-state buffers is shown below:



and the construction of a 4-to-1-line multiplexer is shown below:



The use of a decoder ensures that no more than one buffer control input is active at any given time.

When the *EN* input of the decoder is 0, all of its four outputs are 0, and the bus line is in a high-impedance state.

When *EN* is active, one of the buffers will be active depending on the binary value in S_1 and S_2 of the decoder.