1. Combinational Circuits vs. Sequential Circuits

Logic circuits can be divided into two groups: combinational and sequential. A combinational circuit is one in which the output of the circuit depends on the present input values. All circuits that we designed in previous chapters are combinational circuits. On the contrary, a sequential circuit is one in which the output of the circuit depends not only on the present input values; it also depends on the past input values. So how does a circuit remember past input values? It remembers the past values using some storage elements called registers. We will study sequential circuit in later chapters. For now, we will concentrate only on the combinational circuits.

Input and output variables. A combinational circuit has some inputs and gives some outputs. The circuit is usually designed using logic gates. **In general, we may assume that a circuit has n input variables and m output variables.** For each output variable, a truth table is usually given that describes the function of the output variable. Alternatively, for each out variable, a Boolean expression may be given that describes the function. So whatever form is given, we already know that we can easily convert from one form to another.

2. Design of a combinational circuit: BCD to Excess-3 Converter

A combinational circuit can be designed easily from its truth table description or Boolean expression.

For example, consider the task of converting BCD codes to Excess-3 codes. A BCD code of a number

What is a BCD code? A BCD code is a coding system for decimal digits. It assigns 4 binary bits to every decimal digit. So for a 2 digit decimal number, we will require eight binary bits to represent in BCD. For example, the digit 1 is 0001 and 9 is 1001 in BCD. The number 96 will be 1001 0110 in BCD.

What is a Excess-3 code? An Excess-3 code is a coding system where each binary number is represented by adding 3 with it. For example, the number 9 is 1001 in BCD, but in Excess-3 it will 1100. Similarly, the digit 5 is 0101 in BCD, but 1000 in Excess-3.

In many applications, it is required that a BCD coded number is converted to Excess-3 coded number. In such applications, we need to design a circuit that converts each digit of the BCD to a digit in Excess-3 code. So in this circuit, we will have four inputs (4 bits of the BCD digit), and we will need to generate four outputs (4 bits of Excess-3 digit). The circuit has the truth table described in Table 4.1.

BCD digit				**Excess-3 digit**			
a	b	c	d	w	x	y	z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

The truth table above shows the four output variables. As each of the output variables is a separate function, we will need to design each of the output variables separately, possibly by optimizing it using minimizing techniques.

Design of w

First we will design the function for w. The canonical form is $w = \sum(5,6,7,8,9)$ where set of don't care rows is $d = \{10,11,12,13,14,15\}$. The K map is drawn below.

		cd			
		00	**01**	**11**	**10**
ab	**00**	0	0	0	0
	01	0	1	1	1
	11	X	X	X	X
	10	1	1	X	X

The simplification is shown below.

		cd			
		00	**01**	**11**	**10**
ab	**00**	0	0	0	0
	01	0	1	1	1
	11	X	X	X	X
	10	1	1	X	X

a

		cd			
		00	**01**	**11**	**10**
ab	**00**	0	0	0	0
	01	0	1	1	1
	11	X	X	X	X
	10	1	1	X	X

bc

		cd			
		00	**01**	**11**	**10**
ab	**00**	0	0	0	0
	01	0	1	1	1
	11	X	X	X	X
	10	1	1	X	X

bd

So $w = a + bc + bd$.

<u>Design of x</u>

The canonical form of x is $x = \sum(1,2,3,4,9)$ with $d = \{10,11,12,13,14,15)$. The K map is simplification stages are shown below.

		cd			
		00	**01**	**11**	**10**
ab	**00**	0	1	1	1
	01	1	0	0	0
	11	X	X	X	X
	10	0	1	X	X

		cd			
		00	**01**	**11**	**10**
ab	**00**	0	1	1	1
	01	1	0	0	0
	11	X	X	X	X
	10	0	1	X	X

$b\bar{c}\bar{d}$

		cd			
		00	**01**	**11**	**10**
ab	**00**	0	1	1	1
	01	1	0	0	0
	11	X	X	X	X
	10	0	1	X	X

$\bar{b}c$

		cd			
		00	**01**	**11**	**10**
ab	**00**	0	1	1	1
	01	1	0	0	0
	11	X	X	X	X
	10	0	1	X	X

$\bar{b}d$

So $x = b\bar{c}\bar{d} + \bar{b}c + \bar{b}d$.

Similarly we derive that $y = cd + \bar{c}\bar{d}$ and $z = \bar{d}$.

Finally we have the following simplified Boolean expression for the 4 output variables of Excess-3 digits:

$w = a + bc + bd.$

$x = b\bar{c}\bar{d} + \bar{b}c + \bar{b}d.$

$y = cd + \bar{c}\bar{d}.$

$z = \bar{d}.$

Now we design the circuit using AND, OR, and NOT gates. From examining the equations, we can find that we will require 7 AND gates, 3 OR gates, and 3 NOT gates, a total of 13 gates. These will give us a straight forward three level design, provided that the number of inputs of a gate is not a problem. Otherwise we will have to cascade gates for higher number of inputs, and that will increase the level of the circuit.

[Draw the diagram here]

However, if we slightly modify the four equations above as follows, we will require fewer gates.

$w = a + bc + bd = a + b(c + d).$

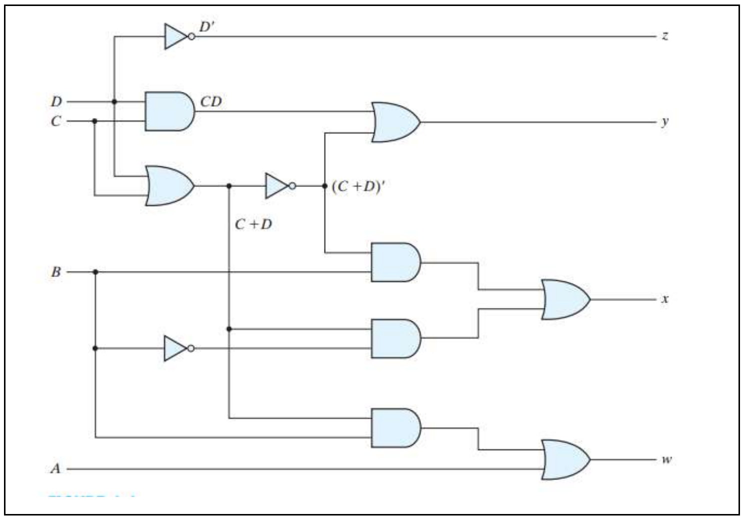
$x = b\bar{c}\bar{d} + \bar{b}c + \bar{b}d = b\bar{c}\bar{d} + \bar{b}(c + d) = b\overline{(c + d)} + \bar{b}(c + d).$

$y = cd + \bar{c}\bar{d} = cd + \overline{(c + d)}.$

$z = \bar{d}.$

If we use the converted equations, we will require less number of gates: 4 AND, 4 OR, and 3 NOT gates, a total of 11 gates. So the converted equation gives us to design the same circuit in 2 gates less.

[Draw the diagram here]



3. Design of Adder Circuits

Adder. Digital computers perform various mathematical functions. The basic of all these functions is addition. So in this section, we will first know how addition circuits are designed. We will first work on basic addition of 1 digit of two numbers. Later we will describe more general addition of n digits.

Half adder. An addition operation of two digits can give four different results based on the two digits. The four results are shown in Table 3.1. In the table x and y are the two digits being added, and c and s are the two bits of the results. It is not difficult to understand that addition of two digits can produce a result that will need two digits, one is called the carry (c), and the other is called the sum (s). The circuit that performs addition of two digits is called a **half adder** circuit.

Table 3.1: Half adder circuit input/output (truth table)

x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Full adder. The bits of the half adder result are named as carry (c) and sum (s). In n digit addition, the sum is kept in the result, and the carry is added with the next significant digits. So in n digit addition, in each stage we will require a circuit that can add three digits instead of two digits, because a carry bit will come from the immediate lowest significant position. We call this circuit a **full adder** circuit. The truth table of the full adder circuit is shown in Table 3.2. Note that the addition three digits can produce at most a two digit result. So the output will contain two bits, one is carry (c), and the other is the sum (s).

Table 3.2: Full adder circuit input/output (truth table)

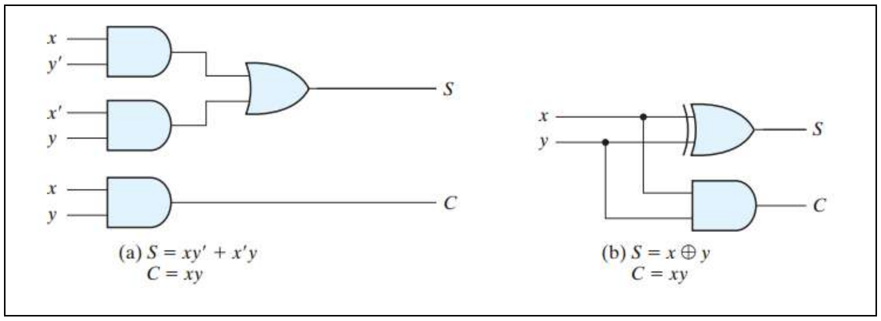
x	y	z	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Design of half adder. We will now design the circuit diagram for a half adder first. We may obtain the simplified circuit without using K map as the input is only 2 variables.

$c = xy.$

$s = \bar{x}y + x\bar{y} = x \oplus y.$

The designs are shown below.



Design of full adder. For the full adder, we first draw the K maps for both output bits.

K map for the carry bit c:

		00	**01**	**11**	**10**
		$\bar{y}\bar{z}$	$\bar{y}z$	yz	$y\bar{z}$
0	$\bar{x}$	0	0	1	0
1	x	0	1	1	1

$c = xy + yz + xz$

K map for the sum bit s:

		00	**01**	**11**	**10**
		$\bar{y}\bar{z}$	$\bar{y}z$	yz	$y\bar{z}$
0	$\bar{x}$	0	1	0	1
1	x	1	0	1	0

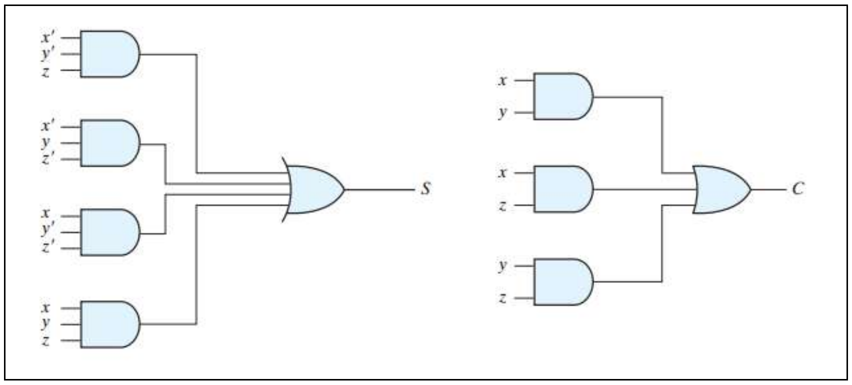
$s = x \oplus y \oplus z$

So the Boolean expressions for the carry and sum are as follows.

$c = xy + yz + xz.$

$s = x \oplus y \oplus z.$

The circuit diagram for the full adder using AND, and OR is shown below. Assume that input variable is available in normal and complemented form. Otherwise we will need NOT gates for complementing some variables.

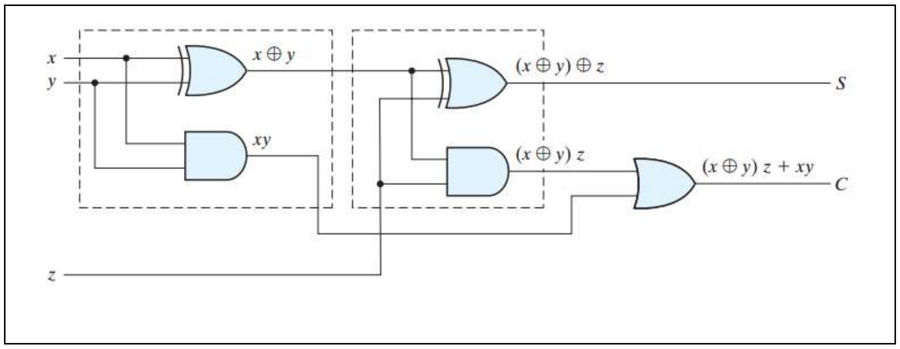


Full adder using two half adder circuits. Note that we can also design a full adder using two half adder circuits. A half adder circuit performs outputs $s = x \oplus y$ and $c = xy$. So if we connect the s output of a half adder to the x input of another half adder and connect the third input z to the second input of the second half adder, the second half adder will give $s = (x \oplus y) \oplus z$ and $c = (x \oplus y)z$. So the s output of the second half adder is obtained, but the problem will remain with c output.

Now observe the truth table of $(x \oplus y)z$ which is given below. The required carry functions output is also given beside to check the difference between the two.

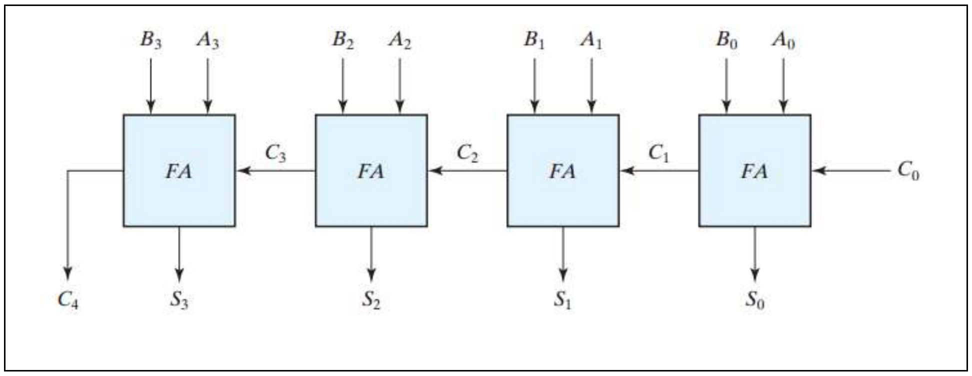
x	y	z	c	$(x \oplus y)z$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	1	1
1	1	0	1	0
1	1	1	1	0

Note the difference in the last two rows of truth table. So it is obvious that to make the two outputs equivalent we only need to add the term xy with $(x \oplus y)z$. So the final output carry of the full adder can be found by ORing the outputs of the second half adder carry and first half adder carry. So this gives us the following circuit design of a full adder using two half adders and an OR gate.



Design of a binary n digit adder. Now we come to point where we will design the more general n bit adder circuit that can add two n digit binary numbers. It will be constructed using n full adders where the carry bit of each full adder circuit is connected to the carry input of the next significant full adder. This will produce a straight forward n bit binary adder circuit.

The design of a 4-bit adder is shown below.


B_3 A_3 B_2 A_2 B_1 A_1 B_0 A_0
FA
C_3 C_2 C_1 C_0
C_4 S_3 S_2 S_1 S_0

Problem with carry propagation time. Consider the 4-bit binary adder circuit designed using full adders above. You will observe that there is a significant signal propagation delay for the last stage (stage 4) of the circuit. This is because the output of the stage 4 full adder will depend on the output of the stage 3 full adder as the carry is an input to this stage. Similarly the output of the stage 3 full adder will depend on the output of the stage 2 full adder and so on.

Consider the circuit of full adder again. Note that each adder requires 3 basic gate delays to provide the output of the carry signal. So the first stage will require 2 basic gate delays to produce the sum S_0 and 3 basic gate delays to produce the carry C_1. The second stage will then need another 2 gate delays to product the carry C_2 and sum S_1. The third stage will require another 2 gate delays to produce S_2 and C_3. In this way, the 4th stage will require in total 3 + 2 + 2 + 2 = 9 gate delays to produce S_4 and C_5. In general for n digit binary addition, the most significant bit will require $2n + 1$ gate delays to product the outputs.

The carry propagation delay will be very problematic when n is large. In such a case, the addition will take a large time. Since the performance of many other computer operations depend on addition, we require that the adder circuit performs it efficiently. In fact there is an alternate representation that will make it faster and complete the output of all stages within only 4 gate delays. Below we describe the idea.

Adder without carry propagation: Carry-look-ahead adder circuit. It is possible to compute the carry bits and sum bits of each full adder stage without creating any dependence on the output of earlier stage.

To understand the idea, let us see what actually carry and sum bits generate in each stage again.

To do this first let us consider only the outputs of the first half adder circuit of each full adder stage. If you already forgot it, you can re-read previous sections to know how a full adder is implemented using two half adders. The first half adder in each full adder circuit stage performs an XOR operation and an AND operation of the two input digits of that stage. So, we define -

$G_i = A_iB_i$ and $P_i = A_i \oplus B_i$. Now we will describe the outputs of the second half adder of each full adder circuit as shown below.

<u>Stage 0:</u>

$C_1 = P_0C_0 + G_0$.
$S_0 = P_0 \oplus C_0$.

<u>Stage 1:</u>

$C_2 = P_1C_1 + G_1 = P_1(P_0C_0 + G_0) + A_1B_1 = P_1P_0C_0 + P_1G_0 + G_1$.
$S_1 = P_1 \oplus C_1$.

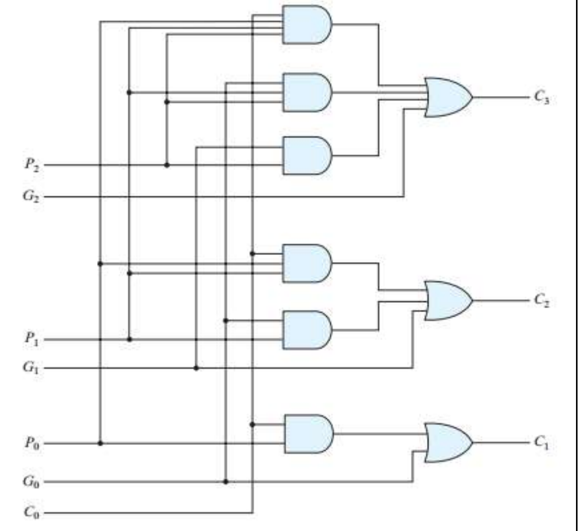
<u>Stage 2:</u>

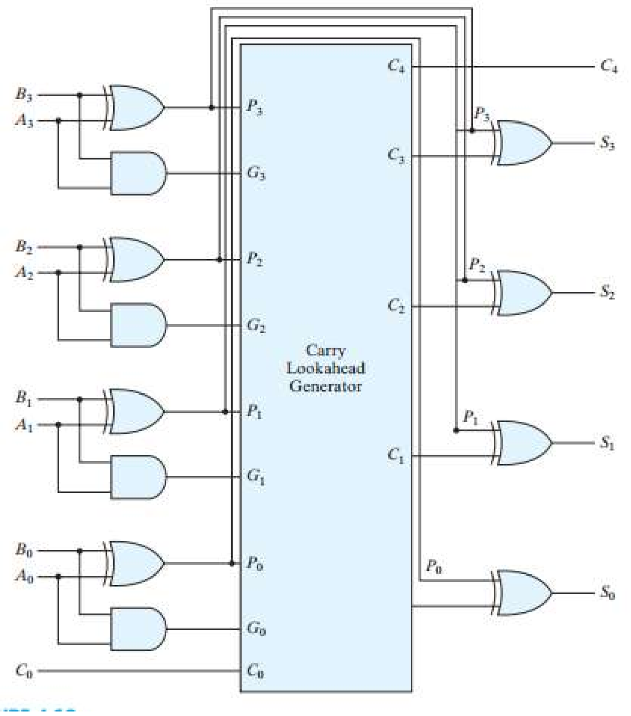
$C_3 = P_2C_2 + G_2 = P_2(P_1P_0C_0 + P_1G_0 + G_1) + G_2 = P_2P_1P_0C_0 + P_2P_1G_0 + P_2G_1 + G_2$.
$S_2 = P_2 \oplus C_2$.

So we see that carry bits of each stage can be generated using P_is, G_is, and C_0. As P_is and G_is are all available from the first half adder of each stage after only 1 basic gate delay, we will need a total of 3 basic gate delays to produce the carry bits, the additional 2 gate delays will be required for the 2 level designs of the carry equations. This design of generating carry bits is called **carry-look-ahead generator** circuit.

After carry bits are available, we will require another 1 gate delay to produce the sum bit. So we need here in total 4 basic gate delays for producing all sum bits and carry bits of all stages. The most important advantage of this design is that it does not depend on the number of bits we are adding. So the design is scalable when n is large.

The final design of the carry-look-ahead circuit and the complete 4-bit full adder circuit is shown below.





3. Design of Sub-tractor Circuits

Similar to the adder circuit, we will design first a half-subtractor, and then we will design a full subtractor circuit, both for 1-bit.

Desing of a half-subtractor. A half-subtractor will have two input bits x and y, and it will output two bits s and b where s is the subtractor bit and b is the borrow bit. Below we show the truth table for a half-subtractor.

x	y	b	s
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

From the truth table, the Boolean expressions for the output bits are as follows.

$s = x \oplus y.$

$b = \bar{x}y.$

Design of a 1-bit full subtractor. In a full subtractor, there is a borrow bit in addition to the two input bits x and y. Let the borrow bit be z. The truth table of the full subtractor is shown below.

x	y	z	b	s
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

We can use K maps to simply the equations for s and b of a full subtractor.

K map for the carry bit c:

		00	**01**	**11**	**10**
		$\bar{y}\bar{z}$	$\bar{y}z$	yz	$y\bar{z}$
0	$\bar{x}$	0	1	1	1
1	x	0	0	1	0

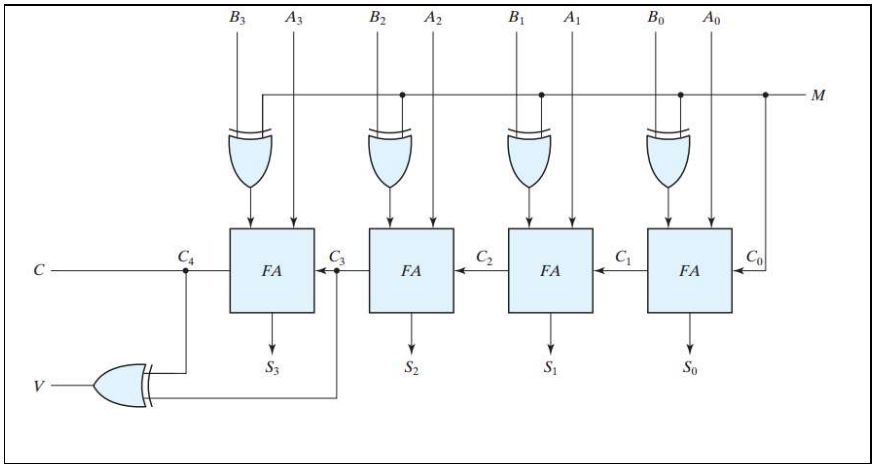
$c = \bar{x}z + \bar{x}y + yz$

K map for the sum bit s:

		00	**01**	**11**	**10**
		$\bar{y}\bar{z}$	$\bar{y}z$	yz	$y\bar{z}$
0	$\bar{x}$	0	1	0	1
1	x	1	0	1	0

$s = x \oplus y \oplus z$

Design of binary n bit subtractor. An n bit subtractor circuit can be designed from an adder circuit and using some basic gates. Remember from the binary arithmetic that $A - B = A + \bar{B} + 1$ where $\bar{B} + 1$ represent the 2's complement of B. So to design a circuit that performs $A - B$, we will use NOT gates to complement each digit of B before connecting them to adder circuit input. Moreover, we give 1 as the carry bit C_0. The design of the circuit is shown below. The circuit is an adder-subtractor. When the M bit is given high (1), it works as a sub-tractor; otherwise it works as an adder.



4. Other Adder Circuits

BCD Adder.

5. Comparator circuits

One bit comparator. Two digits can be compared very easily. Suppose x and y are two binary digits. We need to design three output functions $G(x > y)$, $L(x < y)$, and $E(x = y)$. The functions output 1 when $x > y$, $x < y$, and $x = y$ respectively. The truth table for the three functions is given below.

x	y	G	L	E
0	0	0	0	1
0	1	0	1	0
1	0	1	0	0

1	1	0	0	1

From the truth table, it is very easy to derive Boolean expressions for the three functions G, L, and E as follows.

$G = x\bar{y}$, $L = \bar{x}y$, and $E = \bar{x}\bar{y} + xy = \overline{(x \oplus y)}$.

***n* bit comparator.** For n bit comparator circuit, a truth table will be very big, because there will be 2^{2n} rows. So drawing the truth table will be prohibitive. So let us try to design the n bit comparator using other methods.

<u>*E* function</u>

Let us first design the E function. We have already designed the E function for one digit. So for n digits, we will just need to AND the output of all 1-digit functions. ANDing outputs ensure that E will be 1 when all 1 bit functions give a 1 output. So

$E = x_0 x_1 \dots x_n$ where x_i is the E function for the ith digit and $x_i = \overline{A_i}\,\overline{B_i} + A_i B_i$.

<u>*G* function</u>

A is greater than B if one of the following conditions hold

- $A_{n-1} > B_{n-1}$
- $A_{n-1} = B_{n-1}$ and $A_{n-2} > B_{n-2}$
- $A_{n-1} = B_{n-1}$ and $A_{n-2} = B_{n-2}$ and $A_{n-3} > B_{n-3}$
- And so on.

So we can design a Boolean expression as follows.

$G = A_{n-1}\overline{B_{n-1}} + x_{n-1}A_{n-2}\overline{B_{n-2}} + \dots + x_{n-1}x_{n-2}\dots x_2 x_1 A_0 \overline{B_0}$.

<u>*L* function</u>

A is less than B if one of the following conditions hold

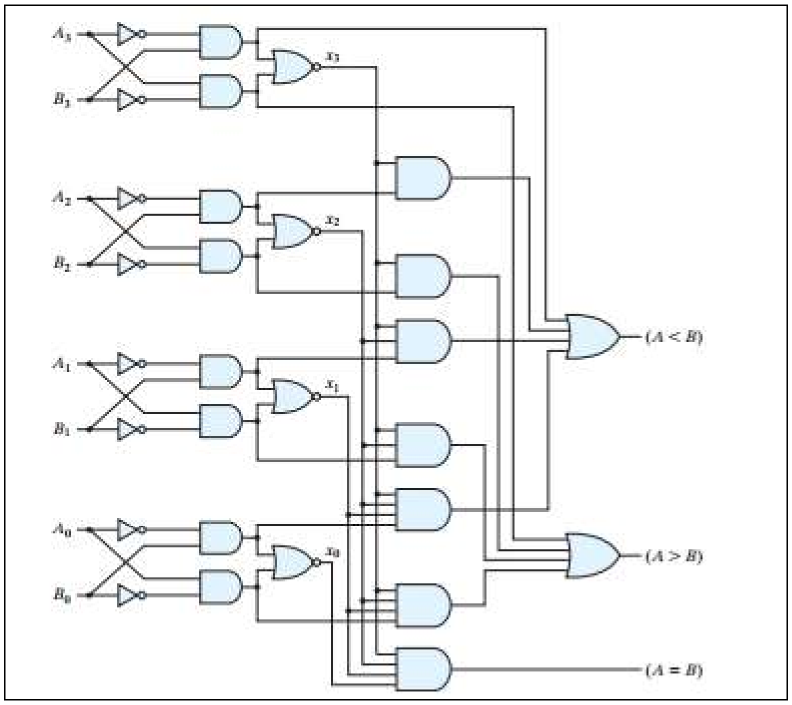
- $A_{n-1} < B_{n-1}$
- $A_{n-1} = B_{n-1}$ and $A_{n-2} < B_{n-2}$
- $A_{n-1} = B_{n-1}$ and $A_{n-2} = B_{n-2}$ and $A_{n-3} < B_{n-3}$

- And so on.

So we can design a Boolean expression as follows.

$L = \overline{A_{n-1}}B_{n-1} + x_{n-1}\overline{A_{n-2}}B_{n-2} + \ \ldots + x_{n-1}x_{n-2}\ldots x_2x_1\overline{A_0}B_0.$

The 4-bit comparator circuit is drawn below. The x_i outputs are computed using the equation $\overline{(A_i\overline{B_i} + \overline{A_i}B_i)}$. Because this will give us the two required terms $A_i\overline{B_i}$ and $\overline{A_i}B_i$ directly that will be used for computing the G and L functions.



6. Decoders

A decoder is a combinational circuit that has

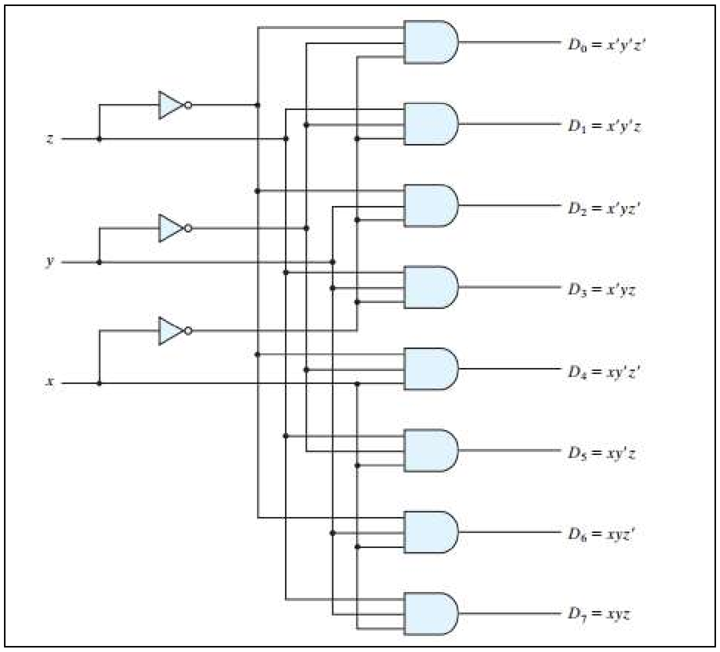
- n input bits
- m output bits where in most cases $m = 2^n$.

The output bits are number as $O_0, O_1, \ldots, O_{m-1}$.

For every input combination, only one output bit will be 1 and all other bits will be 0. The table below shows which output bit will be true for a input combination.

x	y	z	**Output bit**
0	0	0	O_0
0	0	1	O_1
0	1	0	O_2
0	1	1	O_3
1	0	0	O_4
1	0	1	O_5
1	1	0	O_6
1	1	1	O_7

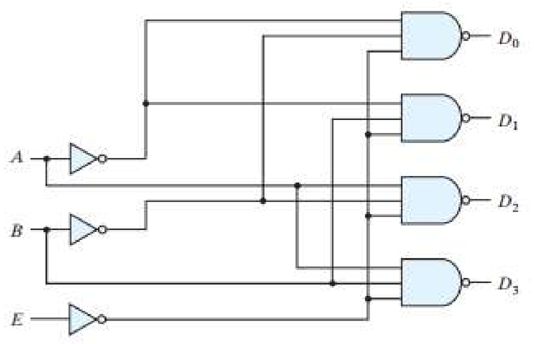
Decoder outputs are actually minterm functions. So, each output bit represents a minterm function, because it becomes true for a specific binary combination, and becomes 0 for every other binary combination. Below we draw the circuit diagram of 3-to-8 line decoder.



Enable bit in the decoder. In practical cases, a decoder has a separate input bit which is called an Enable bit. The Enable bit enables the circuit operation. So we must give a high (1) input to this enable bit. Otherwise, all the output bits will be low (0) irrespective of the inputs. **Enable bit is very important when you will interconnect two or more decoders to extend its functionality.**

Decoder as the de-multiplexer. A de-multiplexer is a circuit that forwards an input line to one of a set of output lines. A decoder with an enable bit can function as a de-multiplexer as follows. You connect your input line to be transmitted with the enable input of the decoder. Then you give a binary patter in the decoder main input bits. That binary pattern will create a high (1) output at the corresponding decoder output line. However if input signal is low, the output will also be low. So it works like a de-multiplexer.

Decoder outputs maxterm functions. In many practical scenarios, you will find that the decoder circuit generates a maxterm function. That means every output bit becomes low (0) when input variables have a specific combination, and becomes high (1) for all other combinations. So in this case, the decoder circuit is actually generating a maxterm function. This type of decoder is usually designed using NAND gates (can also be designed using OR gates at the output) in the output instead of AND gates (for the minterm Decoder described earlier). Below we show a 2-to-4 line maxterm generating decoder circuit.



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	1

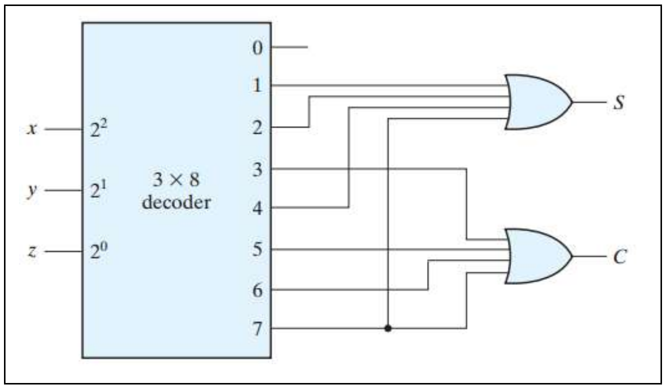
Note that in the maxterm generating decoder, the enable bit must be set to low (0) for normal operation of the decoder. If the enable bit is set to high (1), all decoder bits will be high (1).

Active low outputs. Decoders generating maxterms are sometimes called decoders with active low output.

Designing functions using decoders. It is very easy to design a circuit from a truth table of a function. The process is as follows.

- Use a decoder generating minterms (maxterms).
- Connect input variables to decoder input bits.
- From truth table, find the row numbers where the function is 1 (0). Connect the output lines corresponding to this rows to an OR (AND) gate. The output of the OR (AND) gate will be the desired function.

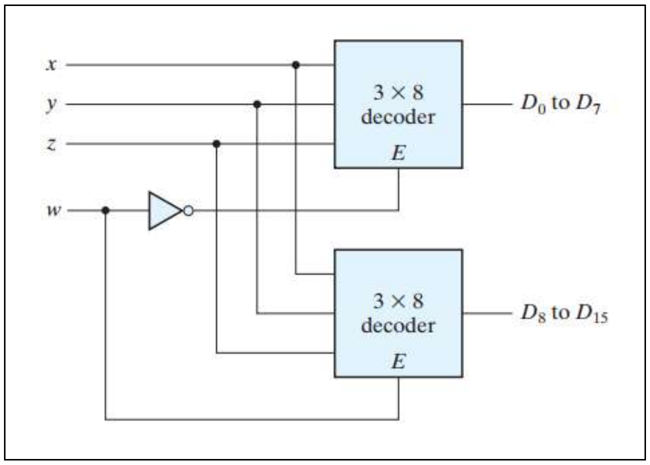
As decoders generate minterm (maxterm), we can combine decoder outputs with OR (AND) gate to design a Boolean function for its minterm expression (maxterm expression). In the figure shown below, we design the 1-bit full adder using a decoder and two OR gates. The decoder in this case generates minterms.



Higher order decoders from lower order decoders. Suppose you need a 4-to-16 line decoder, but you have only 3-to-8 line decoder available in the lab. You can make a 4-to-16 line decoder from 2 3-to-8 line decoders as follows. Assume that the input variables are w, x, y, and z.

- Connect the three lowest significant variables (x, y, and z.) to the three input bits of both the decoder.
- Connect the most significant input variable (w) in complemented form to the first decoder, and in normal form to the second decoder. So when the w bit is high (0), first decoder will be in operation. So its outputs can be denoted as 0 – 7. When w is high (1), the second decoder will be in operation. Its outputs can be denoted as 8 – 15. Observe that two decoders combined generate the accurate output line corresponding to the inputs.

The design of 4-to-16 line decoder using 3-to-8 line decoders is shown below.



The above idea can be extended to create more higher order decoders.

Practise: Design a 4-to-16 line decoder from 2-to-4 line decoders. (Easy!)

7. Enocoder

What is an encoder?An encoder circuit is the opposite of a decoder. The input is m lines and the output is the binary pattern of n bits. The truth table of a 4-to-2 encoder is given below. Note that the truth table is not complete, there are many missing rows, which are actually considered as don't cares.

Input bits				**Output**	
I_3	I_2	I_1	I_0	x	y
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

So the Boolean expressions for the two output bits can easily be derived as:

$x = I_3 + I_2$ and $y = I_2 + I_3$.

For an 8-to-3 line encoder, the input/output table is shown below.

Input bits								**Output**		
I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	x	y	z
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

The Boolean expressions for the three output bits are as follows.

$x = I_4 + I_5 + I_6 + I_7$, $y = I_2 + I_3 + I_6 + I_7$, and $z = I_1 + I_3 + I_5 + I_7$.

Limitation of a normal encoder. A limitation of a normal encoder is that only 1 input line can be high (1) at any given time. If two inputs are set high, the output binary pattern is meaningless. To solve this problem, we design a new type of encoder called priority encoder that allows multiple input lines to be active simultaneously.

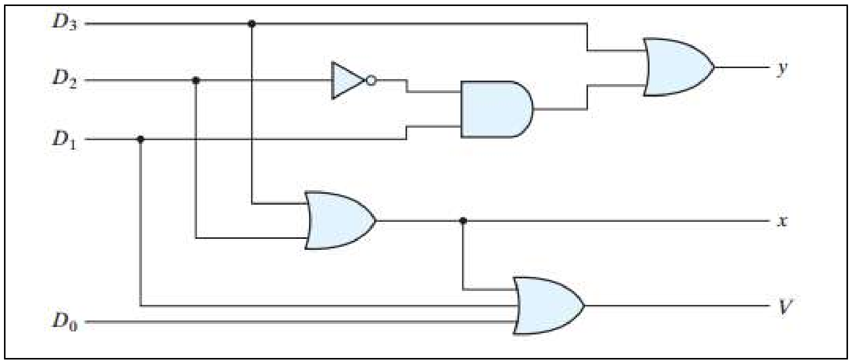
Priority encoder. A priority encoder allows multiple input lines to be high simultaneously. It assigns a priority to each input line. So whenever multiple input lines are set high, input line having the highest priority will be encoded into the output, rest of the lines will be ignored. This is a priority encoder. The input/output table of the priority encoder is shown below. Note that I_7 has the highest priority and I_0 is assigned the lowest priority.

I_3	I_2	I_1	I_0	x	y	z
0	0	0	0	0	0	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

Output equations for the priority encoder are derived below.

$x = \overline{I_3}I_2 + I_3 = I_3 + I_2$, $y = \overline{I_3}\,\overline{I_2}I_1 + I_3 = I_3 + \overline{I_2}I_1$, and $z = I_3 + I_2 + I_1 + I_0$.

The circuit diagram of the priority encoder is shown below.



8. Multiplexer

What is a multiplexer? A multiplexer is the opposite of a de-multiplexer. It has m number of input lines, n number of selection bits, and 1 output line. Usually, we will find that $m = 2^n$. A multiplexer connects one of its inputs to the output. The selection bits define which of the input line will be connected to the output line. The block diagram of a 4x1 multiplexer is given below.

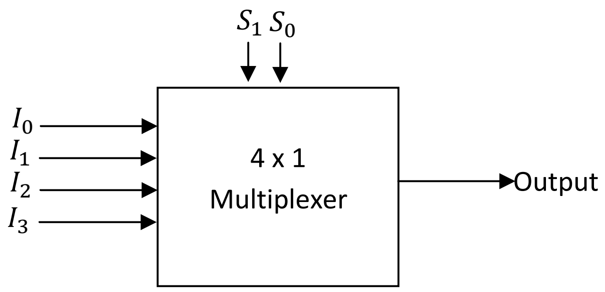


Figure: Bock diagram of a multiplexer

The operation of a multiplexer can defined using the following table.

S_1	S_0	Output
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Design a multiplexer circuit. You can design a multiplexer circuit using basic gates as follows. It is very simple. We need to use AND gates at the first level. For each input line, there will be an AND gate that will pass the input to the output when the section variable has the appropriate combination. The outputs of the AND gates will then simple be ORed using an OR gate. The design for a 4x1 multiplexer is shown in the following figure.

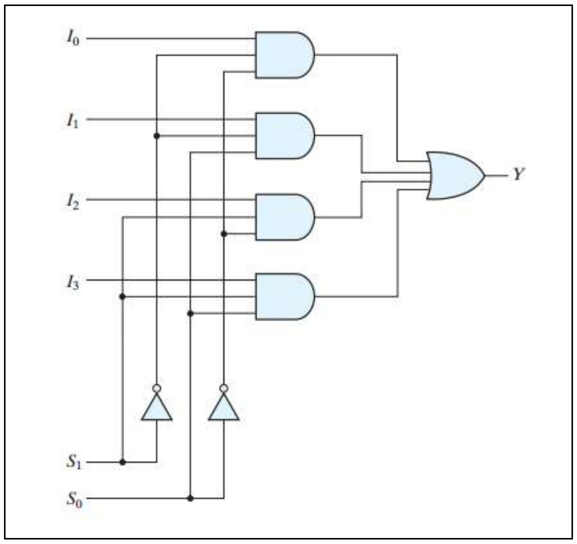


Figure: Design of a 4x1 multiplexer circuit.

Implementing Boolean functions using multiplexer. We can implement Boolean functions using multiplexers. For example, consider the following function:$f(x, y, z) = \sum(1,5,6)$. We will use a 8x1 multiplexer to implement this function. The inputs x, y, z will be connected to the selection bits S_2, S_1, S_0 of the multiplexer circuit. Then, we will give 1 as input to those input lines which correspond to the minterms of the function. Here, the function has the following minterms: 1, 5, and 6. So will will need to give 1 to I_1, I_5, I_6. We will set other inputs to 0. This will give us required outputs. For example, suppose you give input $(x, y, z) = (0,1,1)$. The multiplexer will connect I_3 to output because its selection bits are 0,1,1. Since, we set I_3 to 0 previously, the output will be 0 (correct one). You can verify the same for a 1 output.

A Boolean function can even be implemented using a lower order multiplexer with respect to input variables of the function: the number of selection variables is less than the number of input variables. For example, suppose we have to implement the above function $f(x, y, z) = \sum(1,5,6)$ using a 4x1 multiplexer instead of an 8x1 multiplexer. We can do this using the following procedure:

- First we will connect the higher order variables (in this case x, y) to the selection variables of the multiplexer.
- For the remaining variables (in this case z), we will need to give a Boolean expression of z in the four inputs of the multiplexer (remember that in case of 8x1 multiplexer, we give either 1 or 0 in the inputs).

- To determine which expression of z will need to be given in the inputs of multiplexer, we will require to check the truth table of the function as below:

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

We can see from the truth table that when selection variable $(x, y) = (0,0)$, the function's output is same as the third input variable z. So we can connect z to the input I_0 of the multiplexer so that whenever selections are (0,0), the state of z goes to output. Then see the case for selection variables having the combination (0,1). In this case, from the truth table we can see that output of the function is constant 0. So we will give 0 in the input I_1 of the multiplexer so that when selections are (0,1), 0 goes to the output line. In this way, we can easily find that I_2 and I_3 should be given z and $\bar{z}$ as their inputs.

Constructing higher order multiplexer using lower order multiplexers. It is possible to construct higher order multiplexers using lower order multiplexers that have enable bits. For example, we can construct an 8x1 multiplexer using two 4x1 multiplexers.

Method 1: Multiplexer have enable bits

Suppose the selection variables are x, y, z. The multiplexer can constructed as follows:

- The two 4x1 multiplexers, say M_1 and M_2 will be used in the first level. The two lowest order selection variables y and z will be directly connected to the selection variables of both the multiplexer. The higher order selection variable x will be used to enable only one multiplexer at a time. So x will directly go the enable of M_2, and $\bar{x}$ will be connected to the enable of the second multiplexer M_1. So $x = 0$, M_1 will be enabled, so its inputs will go to its output while M_2 will remain disabled and its inputs will not go to its output. So when $x = 0$, the four inputs of M_2 will go to the output of M_2 depending on the other two selection variables y and z.

- Now, we will need to connect the two outputs of M_1 and M_2 using an OR gate. The output of the OR gate will be output the 8x1 multiplexer.

Method 2: Multiplexers don't have enable bits

In this case, we will need to do the following:

- We will use 2x1 multiplexer at the second level. The outputs of the multiplexers $M_1$and M_2 will be connected to the inputs I_0, I_1 of the 2x1 multiplexer. The most significant selection variable x will now be connected to the selection variable S_0 of 2x1 multiplexer. So, whenever $x = 0$, its input I_0 will go to output which actually sends the first four lines of the first multiplexer M_1. When $x = 1$, the input I_1 will go the output which actually sends the four lines of the second multiplexer M_2.