09-20

Number Systems

All data used by computers is stored as bits. Each bit represents a 0 or a 1. When we say that computers operate in binary (base 2), we mean that internally, data is manipulated as collections of individual bits. Bits are usually combined into bytes, which can represent numbers from 0 to 255. Bytes can also be combined to represent larger numbers. Thus all data in a computer is stored as numbers.

Although all numbers are stored internally in binary, it is convenient for people to use the base 10 numbering system when communicating with the computer. When numerical values are written in programs in high-level languages such as Pascal, Turing, or C, they are usually expressed as base 10 numbers. Before the computer can actually use these numbers, however, they must be converted back into binary form.

When writing in low-level languages it is advantageous to write numerical values in the same system that the computer uses. Thus in assembly language programs, we use base 2 (binary) or more commonly base 16 (hexadecimal) values. Base 16 is useful because one base 16 digit corresponds to four binary digits (see Section 4.1.1).

Number Systems

Numbers and characters can be written in many different bases. Base 2, or the binary system, is based on 1s and 0s. Base 10, the common decimal system, is based on a series of digits from 0 to 9. The hexadecimal system, or base 16, is composed of the numerals 0 to 9 and the letters A to F, a total of sixteen symbols. All numbers, symbols, and commands in hexadecimal are represented using combinations of 0 to 9 and A to F.

The following chart shows a series of base 10, binary, and hexadecimal equivalents.

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Depending upon the level of communication required between the computer and hardware, any base system can be converted into another base system. In computing, the most important conversions are:

- base 10 to base 2 (decimal to binary),
- base 2 to base 10 (binary to decimal),
- base 2 to base 16 (binary to hexadecimal), and
- base 16 to base 2 (hexadecimal to binary).

Fully understanding integrated circuits, low-level programming, interfacing, inputs, and outputs requires an understanding of the three major base systems (decimal, binary, hexadecimal) and the ability to convert among the three systems.

Base 10 to Base 2

The conversion from base 10 to base 2 relies on the arithmetic operation of division. To convert between base 10 and base 2, you must continually divide the base 10 number by 2 until the quotient is zero. The binary answer is found in the remainders collected in reverse.

Here are two examples of base 10 to base 2 conversion. Note that the number appearing in subscript indicates the base.

Example 1:

To convert

$$14_{10} = X_2$$

Collect the binary remainders in reverse (see Figure 4.1).

The answer is 1110. Therefore,

$$14_{10} = 1110_2$$

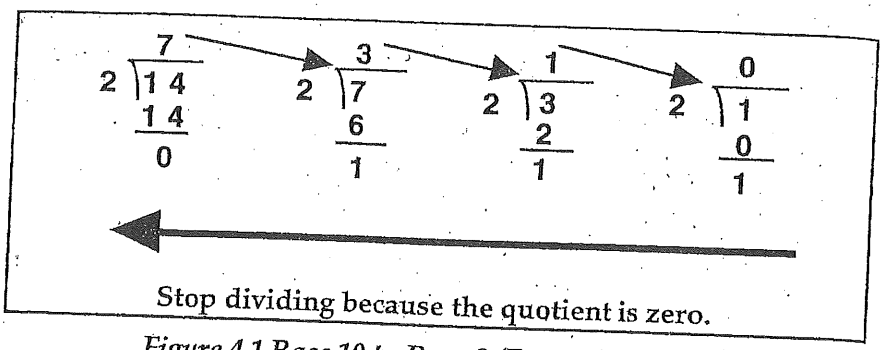


Figure 4.1 Base 10 to Base 2 (Example 1)

Example 2:

Convert

$$75_{10} = X_2$$

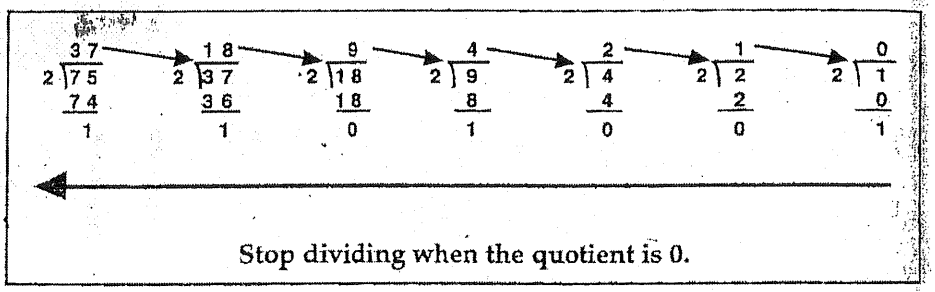


Figure 4.2 Base 10 to Base 2 (Example 2)

Collect the binary remainder in reverse.

The answer is 1001011. Therefore,

$$75_{10} = 1001011_2$$

Base 2 to Base 10

Converting from base 10 to base 2 requires many divisions. Since the conversion from base 2 (binary) to base 10 (decimal) is in the opposite direction, the arithmetic operation used to convert from binary to decimal is multiplication. To convert from base 2 to base 10, you must multiply each binary digit by a power of 2 and add the results to form the base 10 number.

Here is an example of a base 2 to base 10 conversion. Figure 4.3 shows the process for each digit.

Example 1:

Convert

$$10111_2 = X_{10}$$

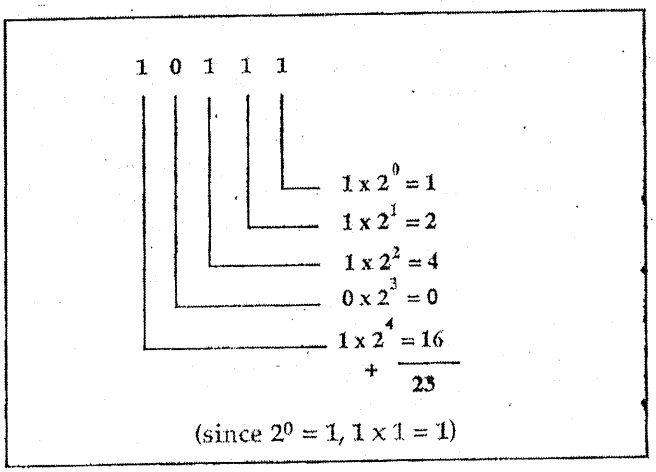


Figure 4.3 Base 2 to Base 10

Therefore

$$10111_2 = 23_{10}$$

Base 2 to Base 16

Base 16 is an important base system since binary values are often displayed in hexadecimal because hexadecimal numbers are shorter and easier to read. Each set of four bits converts into a single hexadecimal digit (see the chart in Section 4.1.1 for more information). Base 2 (binary) to base 16 (hexadecimal) is therefore called a **four-bit conversion.**

Here are two examples of base 2 to base 16 conversion.

Example 1:

Convert

$$10111000_2 = X_{16}$$

Divide the binary number in groups of four starting from the right-most digit.

$1000_2 = 8_{16}$ Therefore,

$1011_2 = B_{16}$ $10111000_2 = B8_{16}$

Example 2:

Convert

$$1011\,1101_2 = X_{16}$$

Separate the binary number into 0001_2 | 0111_2 | 1101_2 so there are groups of four digits. Note that the leading digit is a 1. To make the four binary digit group, three leading 0s were added.

$1101_2 = D_{16}$

$0111_2 = 7_{16}$ Therefore,

$0001_2 = 1_{16}$ $1011\,1101_2 = 17D_{16}$

Base 16 to Base 2

The four-bit conversion process shown in the previous examples is also used to convert from base 16 (hexadecimal) to base 2 (binary). In these conversions, each hexadecimal digit is converted into four bits (see the chart in Section 4.1.1 for more information).

Here are two examples of base 16 to base 2.

Example 1:

Convert

$$A65_{16} = X_2$$

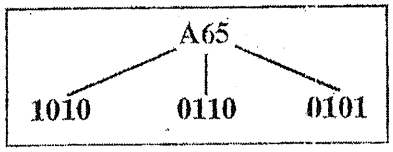


Figure 4.4 Base 16 to Base 2 (Example 1)

Therefore,

$$A65_{16} = 101001100101_2$$

Example 2:

Convert

$$13F_{16} = X_2$$

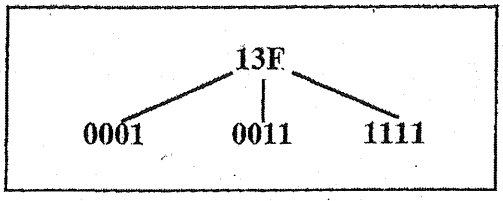


Figure 4.5 Base 16 to Base 2 (Example 2)

Therefore,

$$13F_{16} = 100111111_2$$

as the leading 0s can always be dropped.

4.1.5 Summary of Base Conversions

Conversions among base 2 (binary), base 16 (hexadecimal), and base 10 (decimal) can be accomplished by the methods described in the previous four sections. Base 10 numbers can be converted to any base using similar methods but these conversions are not as useful from a Computer Engineering point of view. The bases that are very useful are:

- base 10 - counting system used in most regular everyday transactions,
- base 2 - counting system used to represent the bits inside a computer as well as machine language programming, and
- base 16 – counting system used to display binary numbers as well as assembly language programming.

Figure 4.6 illustrates which method (multiply, divide, or a four-bit conversion) should be used to convert between the three bases.

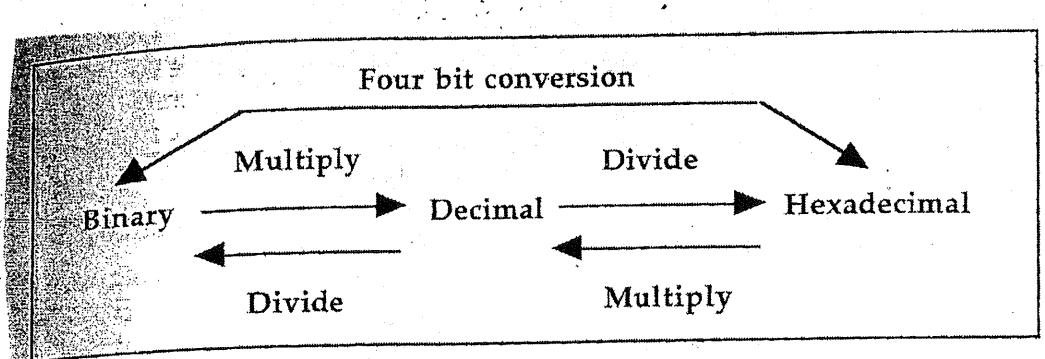


Figure 4.6 Base Conversion Summary

1 Complete the following chart.

Base 10	Base 2	Base 16
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		

2 Complete the following chart.

	Decimal	Binary	Hexadecimal
(a)	14	1110	E
(b)	75	100	
(c)		10111000	
(d)			16D
(e)	256		
(f)		1001011	
(g)			F31C

3 Perform the following conversions.

(a) $123_{10} = X_2$

(b) $101011_2 = X_{10}$

(c) $3A25_{16} = X_{10}$

(d) $374_{10} = X_{16}$

(e) $110110111_2 = X_{16}$

6

Number systems. Practice.

1. Complete the table

Decimal system	Binary system	Hexadecimal system
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		
17		
18		
19		
20		
21		
22		
23		

2. Find the previous and the next numbers. Don't switch to another number system.

10111_2	EF_{16}
101100_2	
	$A00_{16}$
100111_2	

3. Insert inequality/equality signs between the following pairs of numbers(6 marks)

8_{10}	8_{16}
10_{16}	10_{10}
10_{10}	10_2

4. What is a decimal representation of

10111_2

5. Make conversion

13_{10} >> $?_2$ 391_{10} >> $?_2$

7

7

2 Base Arithmetic

Base 10 is the most common system used to carry out routine arithmetic operations such as adding, multiplying, subtracting, and dividing. Simple calculators, for example, are built to automatically display numbers in base 10 despite the fact that calculators store the numbers in memory in binary. More sophisticated calculators contain special keys that will perform these arithmetic operations in other bases as well. It is also possible to build electronic circuits that can add and subtract in binary.

In order to truly understand low-level programming, however, it is very useful to be able to perform the following arithmetic operations by hand:

- adding in binary,
- multiplying in binary,
- subtracting in binary, and
- dividing in binary.

These operations behave in a similar fashion to their base 10 counterparts because they all rely on place value. In the decimal number system, for example, numbers have place value. The first column (from the right) is the units column, the next the tens column, the next the hundreds, and so on. Each successive column is multiplied by ten, since the counting system being used is base 10.

Binary numbers also have place value. In binary, the first column is the units column, the next the 2s column, next the 4s column (2 squared), the next the 8s column (2 cubed) and so on. Each successive column is multiplied by 2 since the counting system being used is base 2.

Roman numerals on the other hand, do not have place value and therefore do not behave well under the four fundamental arithmetic operations. The decimal number 345 is bigger than 99 simply because it has more digits (even though the leading digit of 3 is smaller than the leading digit 9). With Roman numerals, however, X (10 in base 10) is bigger than VIII (8 in base 10) even though X has fewer digits than VIII.

2.1 Adding in Binary

Adding in binary (base 2) is very similar to adding in decimal (base 10), where each column being added has a carry out to the next column. Here is a simple addition in base 10.

Example 1:

Add the base 10 numbers 346 and 578.

$$\begin{array}{r} 346 \\ +578 \\ \hline 924 \end{array}$$

In the right-most column, 8 plus 6 is 14. The 4 is written underneath and a 1 is carried to the next column and the addition is continued.

This same idea of carrying to the next column is used in binary. Here is an example.

Example 2:

Add the binary numbers 1101 and 1001.

```
 1101
+1001
10110
```

Therefore 1101 + 1001 equals 10110.

In the right-most column, 1 + 1 = 10. Remember that the counting is now in binary and not base 10. The 0 is written underneath and the 1 is carried to the next column to the left. In this column, the digits to be added are 0 + 0 + the carry of 1. The answer is 1. Therefore 1 is written underneath and a 0 is carried to the next column.

In the third column, the additions to be made are 1 + 0 + the carry of 0. The answer again is 1.

In the forth column the addition is 1 + 1 + a carry of 0. The answer is 10 (remember the addition is in binary) and therefore 0 is written underneath and a carry of 1 is written beside it since there are no more digits to be added.

This answer of 10110 can be checked by converting all the numbers to base 10 and then adding in base 10. (The method used to do this conversion is described in Section 4.1.3.) The binary number 1101 equals 13 in base 10. The binary number 1001 equals 9 in base 10. Finally, 10110 in binary equals 22 in base 10. Adding the base 10 numbers 13 and 9 equals 22 which is the same as our binary answer of 10110.

2.2 Multiplying in Binary

Multiplying in binary (base 2) is very similar to multiplying in decimal (base 10). Aligning the columns helps to avoid confusion when dealing with so many 1s and 0s.

Here are two examples of multiplying in binary.

Example 1:

In binary, multiply 10001 by 11110.

```
    10001
   x11110
    00000
   10001
  10001
 10001
10001
---------
111111110
```

Therefore $10001_2 \times 11110_2 = 111111110_2$. When the binary numbers are converted to base 10, it can be verified that 17 x 30 = 510.

Example 2:

In binary, multiply 1110 by 1011.

```
  1110
 x1011
  1110
 1110
0000
1110
10011010
```

Therefore $1110_2 \times 1011_2 = 10011010_2$

2.3 How People Subtract in Binary

Subtracting in binary (base 2) can be carried out using two very different methods. The first method, which is how people usually subtract in binary, is similar to subtracting in decimal (base 10) where each column being subtracted has a borrow from the next column.

Here are two examples of the first method of subtracting in binary.

Example 1:

In binary, subtract 1101 from 11101.

```
 11101
-01101
 10000
```

Therefore $11101_2 - 1101_2 = 10000_2$

Example 2:

In binary, subtract 111101 from 1110001.

```
 1110001
-0111101
  110100
```

Therefore $1110001_2 - 111101_2 = 110100_2$

2.4 How Computers Subtract in Binary

Unlike people, who subtract in binary (base 2) by subtracting with borrowing, computers subtract in binary by **complementing**. Complementing uses adding rather than subtracting. This method is used extensively by computers to subtract and also to store negative numbers. The method of subtracting using complementing also works in base 10 (in fact any base system). In base 10, the method works by, in effect, borrowing 99 in one step and paying it back in another step.

Here is an example of subtracting using complementing in base 10. Note that it uses addition rather than subtraction.

Example 1:

Subtract 43 from 52 using complements. To subtract using complements follow these steps.

1. Align the two numbers underneath each other making sure to include leading digits.

```
 52
-43
```

2. Leave the top number alone and take the complement of the bottom number. Since we are avoiding traditional subtracting, take the 43 and find what needs to be added to 43 to get 99. The answer is 56 since 43 + 56 = 99. We are in a 2 digit base 10 number so 1 is subtracted resulting in 99. Therefore 56 is the complement. By writing the top number down and complementing the bottom number, the question becomes

```
 52
+56
```

3. Add the two numbers to get 108.
4. Drop the leading digit so that 108 become 08.
5. Add 1 to the difference so that 8 becomes 9, as before.

Dropping the leading 1 does not seem logical at first but eliminating the leading 1 is equivalent to subtracting 100 from the sum. Subtracting 100 compensates for adding the 99 at the first. The 1 is added at the end to get back to 100.

Complementing seems unnecessarily complicated at first but this five-step method eliminates an entire arithmetic operation (subtracting) and also eliminates the need for the processor to have separate circuitry to process subtraction.

The next example also uses complementing to subtract in binary. This example, however, uses a method called **one's complement**. This method requires no subtracting. It uses only the operation of addition.

Example 2:

In binary, subtract 1101 from 11101. To subtract using one's complement follow these steps.

1. Align the two numbers underneath each other making sure to include leading digits.

```
 11101
-01101
```

2. Leave the top number alone and take the complement of the bottom number. Taking the complement in binary means interchanging any 0 with a 1 and any 1 with a 0. By writing the top number down and complementing the bottom number, the question becomes

```
 11101
+10010
```

3. Add the two numbers to get 101111.
4. Drop the leading digit so that 101111 becomes 01111.
5. Add 1 to the difference so that 01111 becomes 10000. Therefore $11101_2 - 1101_2 = 10000_2$.

These calculations can be checked by converting the binary numbers to base 10 and subtracting. Converting 11101 to base 10 results in the number 29. Similarly, 1101 results in 13 in base 10. Subtracting 13 from 29 gives the result 16. The base 10 number 16 is equal to the answer from Step 5, namely 10000.

Here is another example which uses one's complement to subtract in binary. This example is somewhat different because the first digit in the number being subtracted is a 1, which makes the addition more complicated.

Example 3:

In binary, subtract 1011101 from 1110101. To subtract using one's complement follow these steps.

1. Align the two numbers.

```
 1110101
-1011101
```

2. Leave the top number alone and take the complement of the bottom number. Taking the complement in binary means interchanging any 0 with a 1 and any 1 with a zero. By writing the top number down and complementing the bottom number, the question becomes

```
 1110101
+0100010
```

3. Add the two numbers to get 10010111.
4. Drop the leading digit so that 10010111 becomes 0010111.
5. Add 1 to the difference so that 0010111 becomes 11000. Therefore $1110101_2 - 1011101_2 = 11000_2$.

Exercises

1. Complete the following chart so that the given number is translated to the two other indicated bases.

Question	Decimal	Binary	Hexadecimal
a	255		
b		1110101	
c			F3C2
d	784		

2. Calculate the sum of the following binary numbers.

(a) 11010
+1010

(b) 11010
+1111

(c) 11010
+10010

3. Calculate the product of the following binary numbers.

(a) 11011
x1011

(b) 11110
x11111

(c) 11011
x10001

4. Using the one's complement method, subtract the following binary numbers.

(a) 11011
−10011

(b) 11110
−10011

(c) 11011
−10111

5. **Add the following numbers. Answer should be binary number.**

	Number	Number	Answer
a	100110_2	0011100_2	
b	11101101_2	6_{10}	
c	000010_2	1000010_2	
d	15_{10}	11111101_2	
e	10101010_2	10101001_2	

6. **Multiply the following numbers. Answer should be binary number.**

	Number	Number	Answer
a	$\mathbf{1010_2}$	$\mathbf{9_{10}}$	
b	$\mathbf{1101_2}$	$\mathbf{101_2}$	
c	$\mathbf{10111_2}$	$\mathbf{111_2}$	
d	$\mathbf{15_{10}}$	$\mathbf{3_{10}}$	
e	$\mathbf{11101_2}$	$\mathbf{11_2}$	

7. **Perform the operation**

a) 1111101+10101

b) 10101 x 1101

c) 101011-111

d) 1101101 – 1011

e) 111010101 – 1010111

f) 110110 – 1011

i) 10101 x 10101

h) 10101+1010101

4 Signed Numbers in Binary

Signed binary numbers can be either positive or negative and so they must have a plus or minus sign. In base 10, a signed number will have a + or a – sign in front of the number. In binary signed numbers, however, the + or – is incorporated into the number. Positive numbers are represented in the same way as unsigned numbers.

Negative numbers are represented using **two's complement**. The two's complement representation of a negative number is found by taking the absolute value of the number, inverting each bit of its binary representation, and then adding one to the result.

Bits are numbered from right to left starting at 0. The left-most bit (bit 7 in a one-byte number) is referred to as the **most significant bit (MSb)** because it indicates whether the signed binary number is positive or negative. If the number is positive, the left-most is a 0. If the number is negative, the left-most is a 1. In a two-byte number the left-most bit is bit 15. In a four-byte number the left-most bit is bit 31.

Here are some examples that illustrate the process for calculating the decimal (base 10) value of signed binary (base 2) numbers.

Example 1:

This example calculates the decimal value of the signed binary number 10011000. Immediately we know that the signed number is negative since bit 7 (most significant bit) is a 1.

The two's complement method is used to determine the magnitude of this negative number.

1. Take the complement of the given number. The complement of 10011000 is 01100111.
2. Adding 1 to 01100111 equals 01101000.

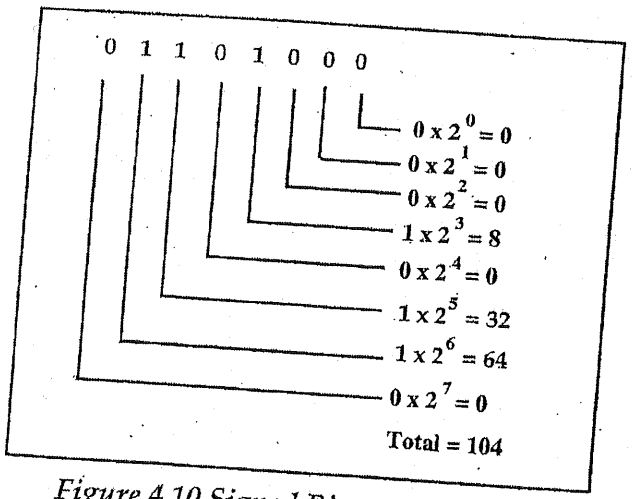


Figure 4.10 Signed Binary Number

Therefore the signed byte $10011000_2 = -104_{10}$.

Example 2:

This example calculates the decimal value of the signed binary number 00011000.

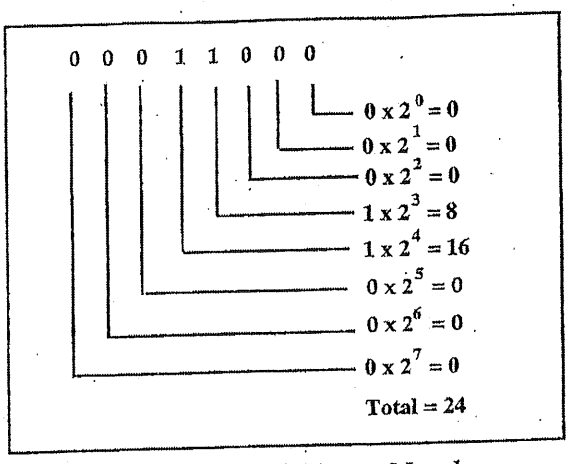


Figure 4.11 Signed Binary Number

Immediately we know that the signed number is positive since the bit 7 (most significant bit) is a 0. The two's complement method can only be used to calculate negative numbers so it cannot be used here to determine the magnitude of this positive number. The standard base 2 to base 10 conversion is used instead. The decimal value of the signed binary number 00011000 is 24_{10}.

Example 3:

This example calculates the decimal value of the largest negative signed one-byte number. The largest negative binary signed one-byte number is 10000000. This number is negative since bit 7 (most significant bit) is a 1. The complement of 10000000 is 01111111. Adding 1 to 01111111 equals 10000000_2. The following calculation converts 10000000_2 to base 10.

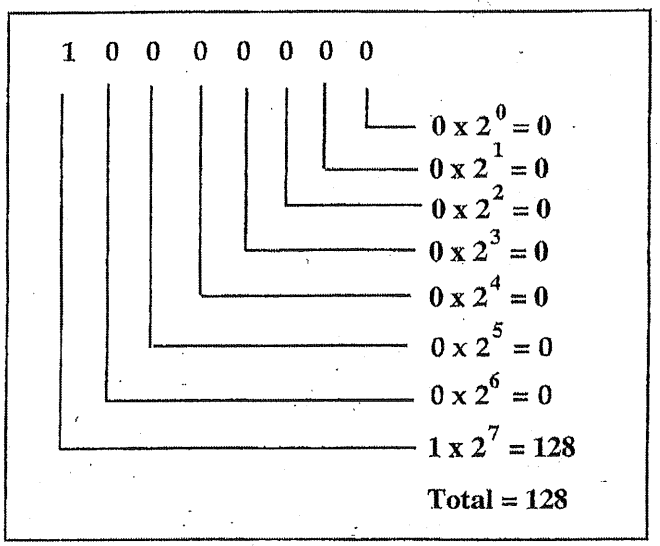


Figure 4.12 Largest Negative Signed Binary Number

Therefore 10000000_2 equals -128_{10}. The value of the largest negative signed one-byte number is -128_{10}.

Example 4:

This example calculates the value of the largest positive signed one-byte number. The largest positive number is 01111111. The bit 7 (most significant bit) is a 0 so the number must be positive and the standard base 2 to base 10 conversion is used. The following calculation converts 01111111 to base 10.

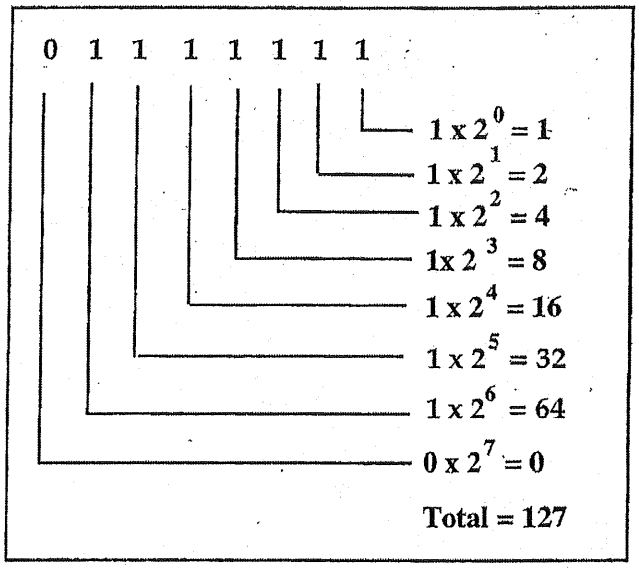


Figure 4.13 Largest Positive Signed Binary Number

Therefore the value of the largest positive signed one-byte number is $+127_{10}$. Thus, a signed byte can hold a value from -128_{10} to $+127_{10}$.

Example 6:

This example calculates the binary (base 2) value for -4_{10}. It uses the two's complement method since the decimal number is negative.

Converting 4_{10} to binary equals 00000100_2. The complement of 00000100_2 is 11111011. Adding 1 to 11111011 results in 11111100. Unlike positive numbers, the binary and hexadecimal representation of a negative number depends upon the number of bits required to store it. For example, -4_{10} is 11111100 when stored in one byte and 1111111111111100 when stored in two bytes.

Example 7:

This example calculates the binary (base 2) value for -17_{10}. It uses the two's complement method since the decimal number is negative.

Converting 17_{10} to binary equals 00010001_2. The complement of 00010001_2 is 11101110. Adding 1 to 11101110 results in 11101111. Therefore the value of -17_{10} is 11101111_2.

Signed integers
Two's complement

Examples

8-bit two's complement integers

sign bit									
0	1	1	1	1	1	1	1	=	127
0	0	0	0	0	0	1	0	=	2
0	0	0	0	0	0	0	1	=	1
0	0	0	0	0	0	0	0	=	0
1	1	1	1	1	1	1	1	=	−1
1	1	1	1	1	1	1	0	=	−2
1	0	0	0	0	0	0	1	=	−127
1	0	0	0	0	0	0	0	=	−128

Two's complement using a 4-bit integer

Two's complement	Decimal
0001	1
0000	0
1111	-1
1110	-2
1101	-3
1100	-4

Calculating two's complement

For example, the signed 8-bit binary representation of 5 is:

0000 0101 (5)

The first bit is 0, so the value represented is indeed a positive 5. To convert to −5 in two's complement notation, the bits are inverted; 0 becomes 1, and 1 becomes 0:

1111 1010

At this point, the numeral is the ones' complement of 5. To obtain the two's complement, 1 is added to the result, giving:

1111 1011 (-5)

The result is a signed binary numeral representing the decimal value −5 in two's complement form. The most significant bit is 1, so the value is negative.

Addition

Adding two's complement numbers requires no special processing if the operands have opposite signs: the sign of the result is determined automatically. For example, adding 15 and -5:

```
 11111 111    (carry)

  0000 1111   (15)
+ 1111 1011   (-5)
===============
  0000 1010   (10)
```

This process depends upon restricting to 8 bits of precision; a carry to the (nonexistent) 9th most significant bit is ignored, resulting in the arithmetically correct result of 10.

Exersices

Convert to binary the following numbers. Use 8-bit two's complement:

1. −7

2. −9

3. −3

Calculate the decimal value of the signed binary numbers

4. 1000 1110

5. 0110 0100

6. 1010 1000

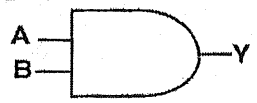
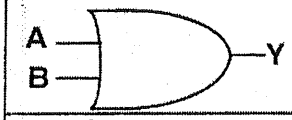
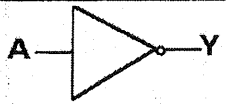
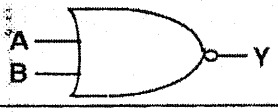
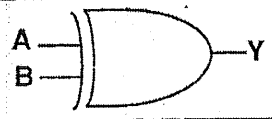
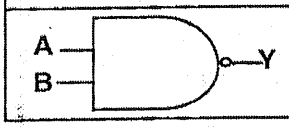
7. Find the smallest (the largest magnitude) 4-bit negative signed number.

8. Find the largest (the largest magnitude) 4-bit positive signed number.

Boolean Algebra and Gates

Table of Gate Representations

Here is a chart which shows the schematic, gate name, algebraic symbol, and Boolean equation for each of the fundamental gates.

Schematic	Gate	Symbol	Boolean Equation	Num.
	AND	•	$Y = A \bullet B$	7408
	OR	+	$Y = A + B$	7432
	NOT	—	$Y = \overline{A}$	7404
	NOR	$\bar{+}$	$Y = \overline{A + B}$	7402
	EOR	⊕	$Y = A \oplus B$	7486
	NAND	$\bar{\bullet}$	$Y = \overline{A \bullet B}$	7400

Logic Circuits

In this section we do three things:
1. **Given a logic circuit, fill in the truth table for the circuit.**
2. **Given a logic circuit, find the Boolean expression for the circuit.**
3. **Given a Boolean expression, draw the logic circuit.**

When Boole developed his system of logic, he indicated that any complex statement can be written in terms of the three basic Boolean operators AND, OR, and NOT.

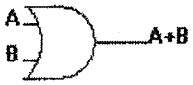
Truth Tables and Boolean Expressions for Logic Circuits

The truth tables and Boolean expressions for outputs of the three basic gates are indicated below.

AND gate

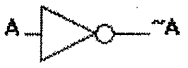
Inputs		Output
0	0	0
0	1	0
1	0	0
1	1	1

OR gate



Inputs		Output
0	0	0
0	1	1
1	0	1
1	1	1

NOT gate



Input	Output
0	1
1	0

They also can be written in the following form:

A AND B = A*B

A OR B = A+B

NOT A = $\overline{A}$

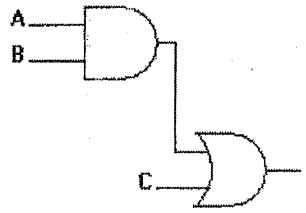
In the following example, we begin to build circuits that include more than one gate.

Example 1

An AND gate and an OR have been combined as indicated below.

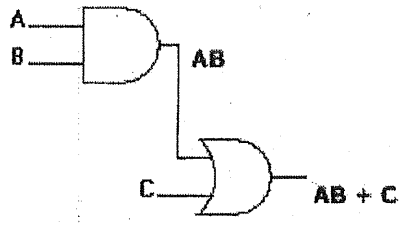
a) Fill in the truth table for this circuit.

b) Indicate the Boolean expression for the circuit.



Solution:

This circuit has three inputs A, B and C and just one output. Since the values of the inputs are either 0 or 1, there are eight possible combinations of inputs. The easiest way to list them is to count in binary from 0 0 0 to 1 1 1.
You can split the question into several steps, and use this steps fill the truth table.



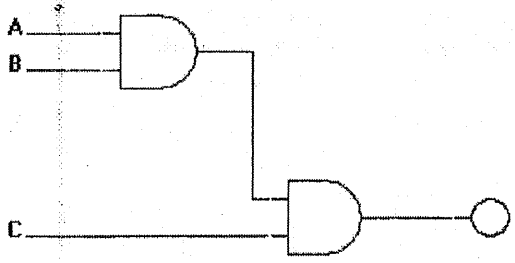
The output of the AND gate is the Boolean expression AB [we no longer include the . sign] and the Boolean expression for the output of the OR gate is AB + C.
We can use this information in the truth table:

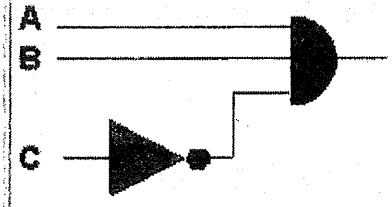
Inputs				Output
A	B	C	AB	AB+C
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	0	1
1	1	0	1	1
1	1	1	1	1

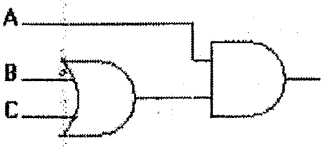
Boolean Expressions for Outputs of Logic Circuits

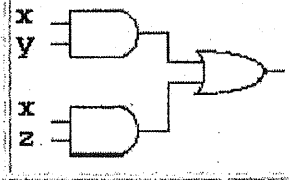
Problems 2

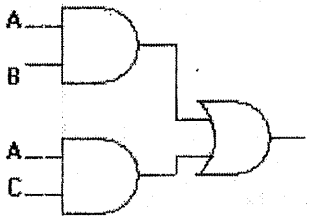
Give the Boolean expression for the output of each circuit:

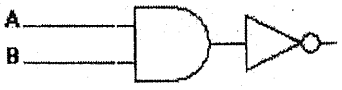
a)


b)


c)


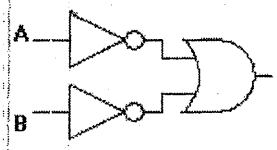
d)


e)


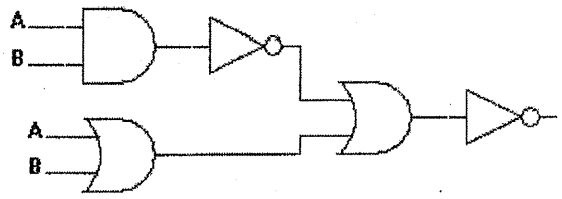
f)


23.

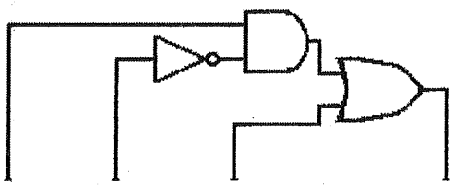
g)



h)

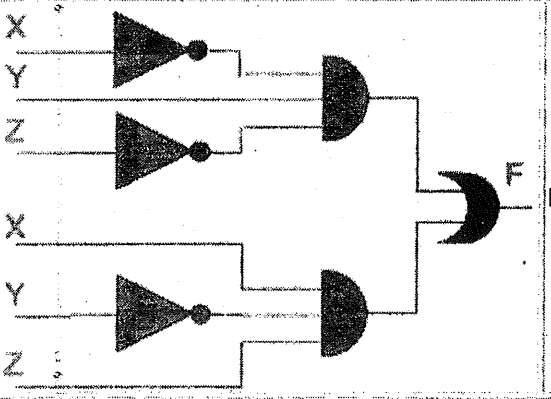


i)

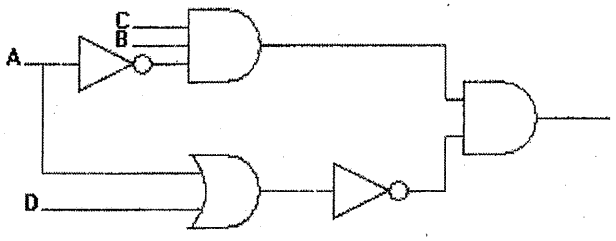


A B C Output

j)



k)



Drawing a Circuit Diagram Given A Boolean Expression for a Circuit

As illustrated above, given a circuit, it is possible to construct a truth table and a Boolean expression for the circuit.

It is also possible, given a Boolean expression such as AB + C, to construct the corresponding circuit. This concept is illustrated in Problems 3.

It is pointed out that there are a number of notations for NOT A.
NOT A, ~A, $\overline{A}$, _ and A' are four of the notations found in the literature. We will use $\overline{A}$.

Problems 3

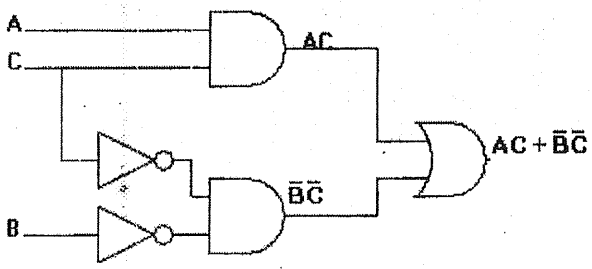
Construct the circuit diagram for these Boolean expressions. Fill in the truth tables.

Examine the truth tables for b) and c). What rule is illustrated?

Examine the truth tables for d) and e). What must be true about these Boolean expressions?

a) $\overline{A} + B$ **b) A(B + C) c) AB + AC d)** $\overline{AB}$ **e)** $\overline{A} + \overline{B}$ **f) AC +** $\overline{BC}$

Answer to part f)



Inputs							Output
A	B	C	AC	$\overline{B}$	$\overline{C}$	$\overline{BC}$	$AC + \overline{BC}$
0	0	0	0	1	1	1	1
0	0	1	0	1	0	0	0
0	1	0	0	0	1	0	0
0	1	1	0	0	0	0	0
1	0	0	0	1	1	1	1
1	0	1	1	1	0	0	1
1	1	0	0	0	1	0	0
1	1	1	1	0	0	0	1

Problems 2

Draw the circuit diagrams and truth tables for these Boolean expressions:

1. B + C

2. AB

3. AB + C

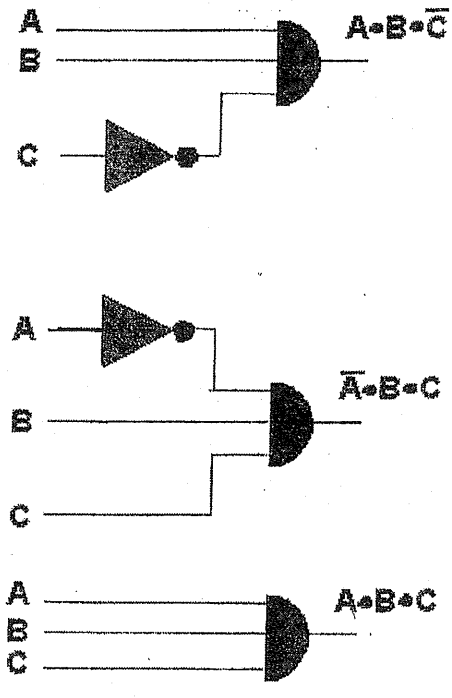
4. A(B + c)

5. $\overline{B + \overline{C} + D}$

6. (X + Z)(X + Y)

7. AB + AC

Examples of a 3-input AND gate.:



Elwood the Electronic Watchdog

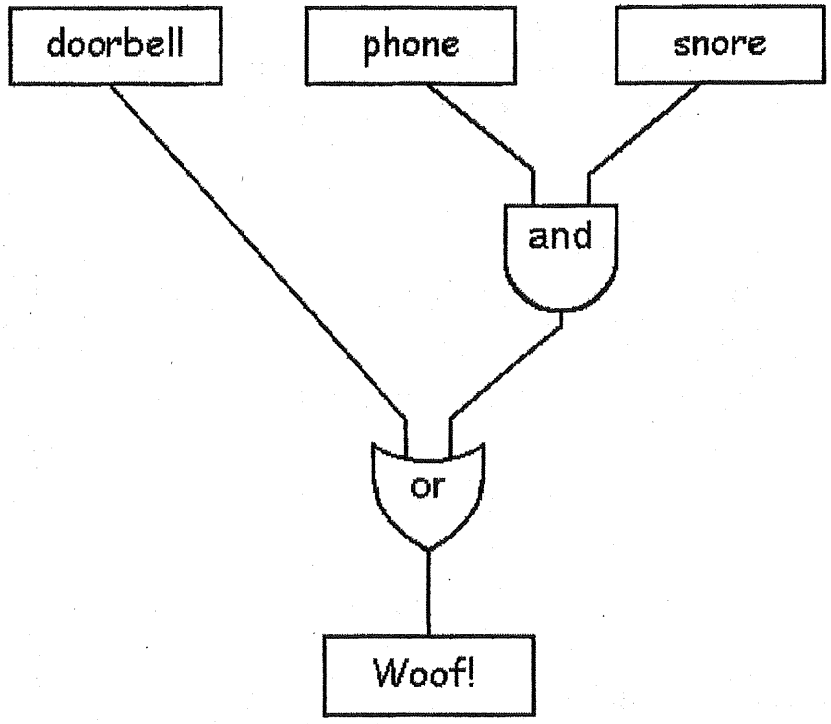
Elwood is a dog who has a very simple behavior. He barks whenever the doorbell rings. Also, he will bark if the phone rings provided that his human, Elmer, is snoring.

Let's design a circuit for Elwood. His input sensors (bits) are:

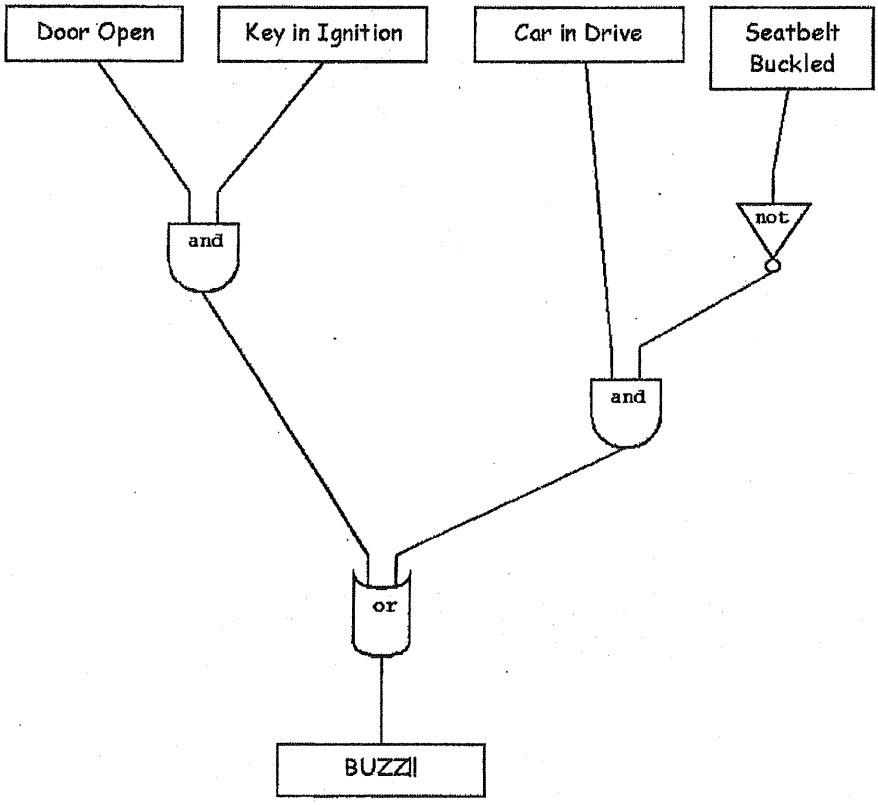
doorbell	A value of 1 indicates ringing. A value of 0 indicates no ringing.
phone	A value of 1 indicates ringing. A value of 0 indicates no ringing.
snore	A value of 1 indicates snoring. A value of 0 indicates no snoring.

Draw the circuit which demonstrates Elwood's barking behavior:

Solution:

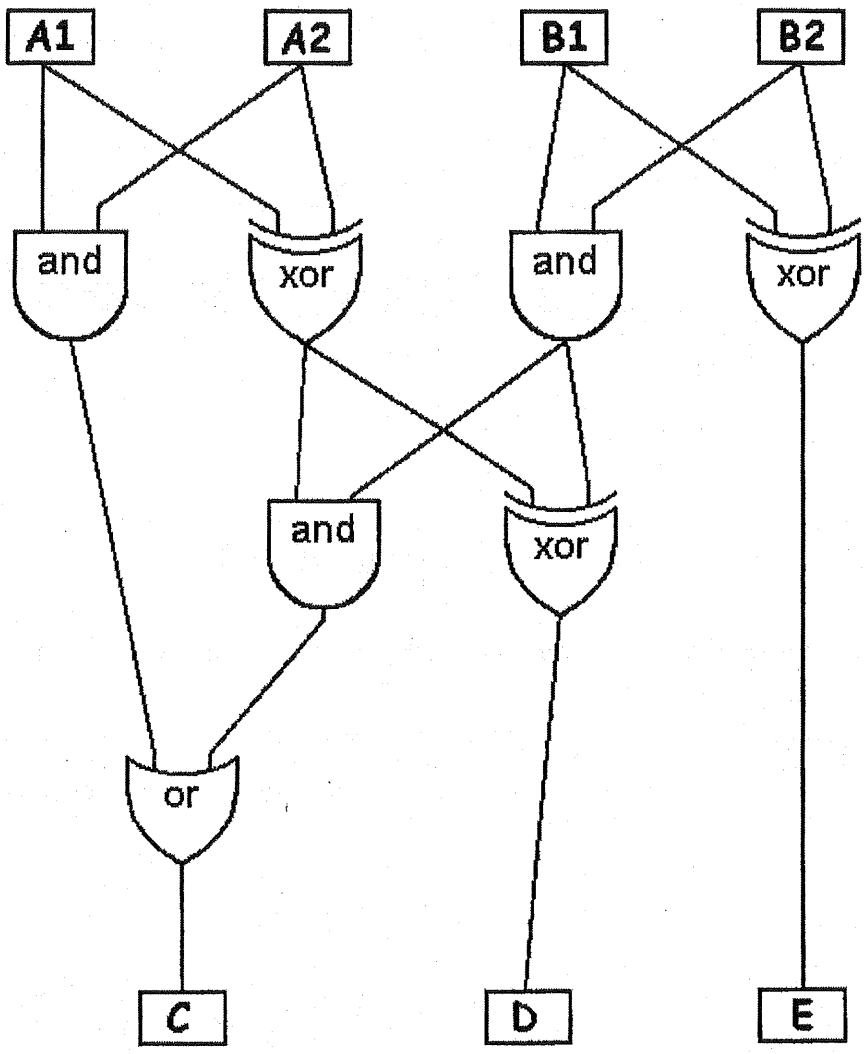


A Simple Sample Circuit



Mystery Circuit

What does this circuit do? Hint: Think of the inputs as a pair of 2 bit binary numbers.



George Boole (1815 - 1864)

The original Working Class Boy Made Good, Boole was born in the wrong time, in the wrong place, and definitely in the wrong class - he didn't have a hope of growing up to be a mathematical genius, but he did it anyway.

Born in the English industrial town of Lincoln, Boole was lucky enough to have a father who passed along his own love of math. Young George took to learning like a politician to a pay-rise and, by the age of eight, had outgrown his father's self-taught limits.

A family friend stepped in to teach the boy basic Latin, and was exhausted within a few years. Boole was translating Latin poetry by the age of twelve. By the time he hit puberty, the adolescent George was fluent in German, Italian and French. At 16 he became an assistant teacher, at 20 he opened his own school.

Over the next few years, depending mainly on mathematical journals borrowed from the local Mechanic's Institute, Boole struggled with Isaac Newton's "Principia" and the works of 18th and 19th century French mathematicians Pierre-Simon Laplace and Joseph-Louis Lagrange. He had soon mastered the most intricate mathematical principles of his day.

It was time to move on.

At the ripe old age of 24, George Boole published his first paper ("Researches on the Theory of Analytical Transformations") in the Cambridge Mathematical Journal. Over the next ten years, his star rose as a steady stream of original articles began to push the limits of 'modern' mathematics.

By 1844 he was concentrating on the uses of combined algebra and calculus to process infinitely small and large figures, and, in that same year, received a Royal Society medal for his contributions to analysis.

Boole soon began to see the possibilities for applying his algebra to the solution of logical problems - his 1847 work, "The Mathematical Analysis of Logic", not only expanded on Gottfried Leibniz' earlier speculations on the correlation between logic and math, but argued that logic was principally a discipline of mathematics, rather than philosophy.

It was this paper that won him, not only the admiration of the distinguished logician Augustus de Morgan (a mentor of Ada Byron's), but a place on the faculty of Ireland's Queen's College.

Not bad for a dead-end kid with no formal education.

. . . . . .

Without a school to run, Boole began to delve deeper into his own work, concentrating on refining his "Mathematical Analysis", and determined to find a way to encode logical arguments into an indicative language that could be manipulated and solved mathematically.

He came up with a type of linguistic algebra, the three most basic operations of which were (and still are) **AND**, **OR** and **NOT**. It was these three functions that formed the basis of his premise, and were the only operations necessary to perform comparisons or basic mathematical functions.

Boole's system (Detailed in his "An Investigation of the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probabilities", 1954) was based on a binary approach, processing only two objects - the now famous yes-no, true-false, on-off, zero-one approach.

Laws in Boolean Algebra

Commutative Law

Mathematics	Boolean
The commutative law in mathematics under the operation of addition is:	The commutative law in Boolean algebra using an OR gate is:
$x + y = y + x$	$A + B = B + A$
and multiplication is:	and using an AND gate is:
$x \cdot y = y \cdot x$	$A \bullet B = B \bullet A$

Associative Law

Mathematics	Boolean
The associative law in mathematics under the operation of addition is:	The associative law in Boolean algebra using an OR gate is:
$(x + y) + z = x + (y + z)$	$(A + B) + C = A + (B + C)$
and multiplication is:	and using an AND gate is:
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	$(A \bullet B) \bullet C = A \bullet (B \bullet C)$

Distributive Law

Mathematics	Boolean
The distributive law in mathematics states that multiplication can be distributed over addition:	Similarly, the distributive law in Boolean algebra states that AND can be distributed over OR:
$x\,(y + z) = xy + xz$	$A \bullet (B + C) = (A \bullet B) + (A \bullet C)$
Addition **cannot** be distributed over multiplication.	In Boolean algebra, OR **can** be distributed over AND gates.
	$A + (B \bullet C) = (A + B) \bullet (A + C)$

The only significant difference between the mathematical and Boolean algebra laws mentioned above is the distributive law. In mathematics, there is ONE distributive law; in Boolean algebra there are TWO.

The Boolean algebra laws (commutative, associative, distributive, and DeMorgan) can be used to simplify or rearrange Boolean equations just as arithmetic laws can be used to rearrange mathematical equations.

The NOT (inverse) is not distributive or AND or OR. The inverse of A and the inverse of B is not the same as the inverse of A and B.

DeMorgan's Theorems

AND and OR gates are related by **DeMorgan's two theorems.** The first theorem is:

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

and the second theorem is:

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

The first theorem states that for any two inputs, the result of the inverse of A or B is the same as the inverse of A and the inverse of B. The two theorems demonstrate the relationship between AND and OR gates. In the first theorem, the NOR gate output is the same as an inverted AND. In the second theorem, a NAND gate output is the same as an inverted OR.

Inverting both sides of the equation in DeMorgan's second theorem produces an interesting result.

$$\overline{\overline{A}+\overline{B}} = \overline{\overline{A \bullet B}} \text{ since } \overline{A}+\overline{B} = \overline{A \bullet B}$$

$$\overline{\overline{A \bullet B}} = A \bullet B \text{ since two NOTs nullify each other}$$

Therefore

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

The gate schematic for the left side of the equation is:

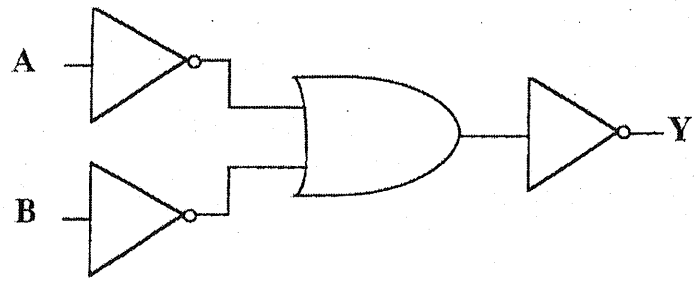


Figure 4.15 Gate Schematic for $\overline{\overline{A}+\overline{B}}$

and the right side of the equation is:

32

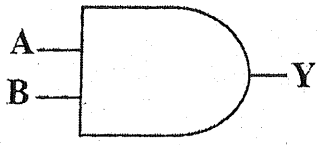


Figure 4.16 Gate Schematic for $A \bullet B$

In Figure 4.15 the OR gate followed by the NOT gate could be replaced with a single NOR gate.

The two circuit diagrams are equivalent using DeMorgan's Laws. From basic principles (Truth Tables) the two circuits again can be shown to be equivalent.

In a similar fashion:

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

The circuit diagram for the left side of the equation

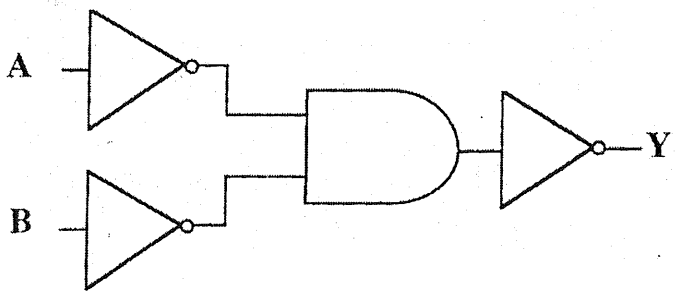


Figure 4.17

is the same as

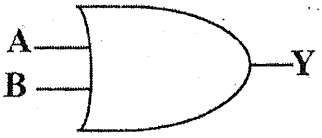


Figure 4.18

Therefore, OR and AND gates can be related using several NOT gates. In Figure 4.17 the AND gate followed by the NOT gate could be replaced with a single NAND gate.

DeMorgan's theorems can be extended to three inputs. They are A, B, and C.

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

The jump from the second to the third step is possible because you can consider $(\overline{A \bullet B})$ to be a single inverted item and then use DeMorgan's first theorem.

The circuit diagram for the left side of the equation is:

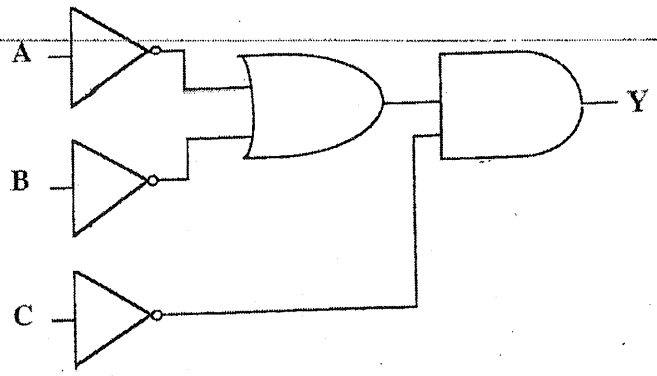


Figure 4.19

The circuit diagram for the right side is:

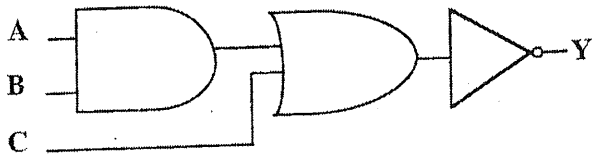


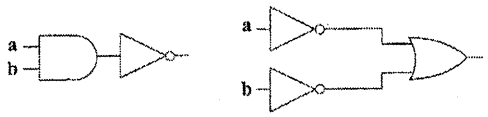
Figure 4.20

The above two circuits are equivalent using DeMorgan's laws.

Exercises

I

Are these two combinatorial circuits equivalent?



The truth tables for both circuits

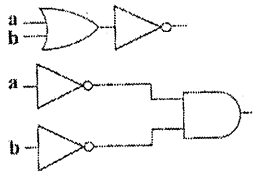
a	b
0	0
0	1
1	0
1	1

a	b
0	0
0	1
1	0
1	1

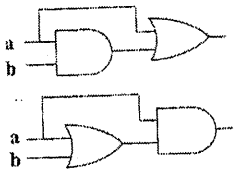
II

Show if these combinatorial circuits are equivalent by working out the Boolean expression and the truth table for each circuit.

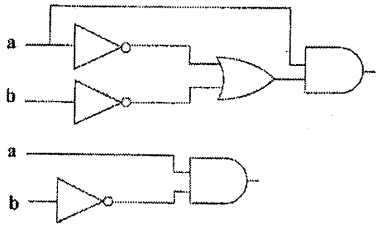
1.



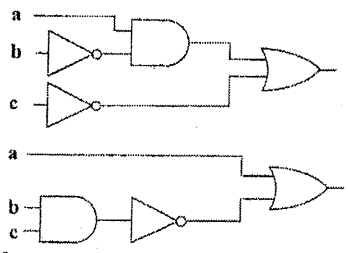
2.



3.



4.



34

Boolean Algebra Theorems

AND, OR, and NOT operations are subject to the following identities:

$ABC = (AB)C = A(BC)$, $A+B+C = (A+B)+C = A+(B+C)$: AND, OR are associative.

$AB = BA$, $A+B = B+A$: AND and OR operations are commutative.

$A + BC = (A+B)(A+C)$, $A(B+C) = AB + AC$: forms of the distributive property.

$\overline{A+B} = \bar{A}\bar{B}$: a form of DeMorgan's Theorem.

$\overline{AB} = \bar{A} + \bar{B}$: a form of DeMorgan's Theorem.

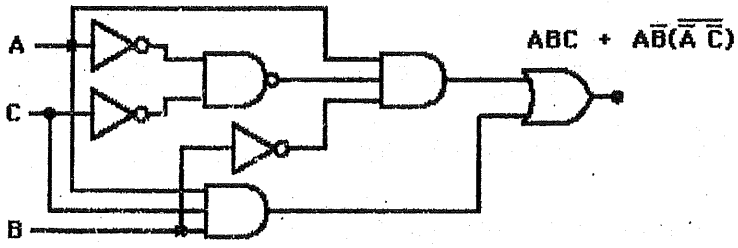
$AA = A$, $A+A = A$, $A + \bar{A} = 1$, $A\bar{A} = 0$, $A = \bar{\bar{A}}$: Single Variable Theorems.

$A + AB = A$, $A + \bar{A}B = A+B$: More two-variable theorems.

$A1 = A$, $A + 1 = 1$, $A + 0 = A$, $A0 = 0$, $\bar{1} = 0$, $\bar{0} = 1$: Identity and Null operations.

Logic Simplification Example

A	B	C	Out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1



It can be simplified by:

$ABC + A\bar{B}(\bar{\bar{A}} + \bar{\bar{C}})$ DeMorgan's theorem

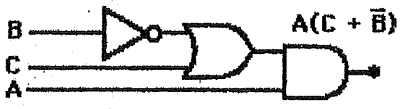
$ABC + A\bar{B}A + A\bar{B}C$ sum of products form

$ABC + A\bar{B} + A\bar{B}C$ $B\bar{A}=AB$ and $AA = A$

$AC(B + \bar{B}) + A\bar{B}$

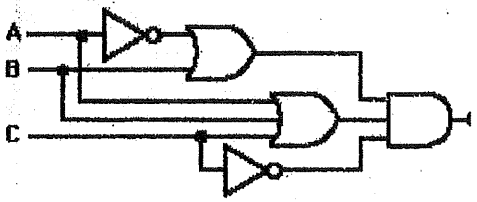
$AC + A\bar{B}$ $B + \bar{B} = 1$

$A(C + \bar{B})$



Exercises

1) Simplify



Order of Operations

1. Brackets
2. Not
3. And
4. Or

2)Prove or disprove the following:

a) $\overline{\overline{A} \bullet \overline{B}} = A + B$

b) $((AB)C + AC = AC$

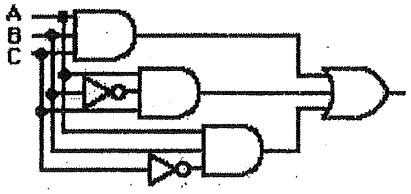
c) A(B+C)+AB=A(B+C)

d) (A+B)(A+C)=A+C

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

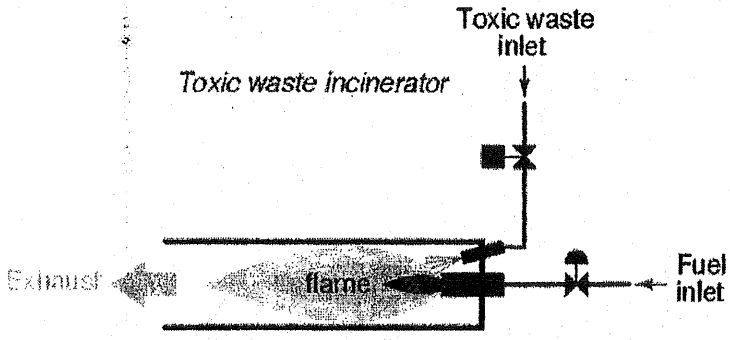
f) (A+B)C+(AC)=C(A+B)

3. Create the equation, simplify, and draw the simplified diagram.



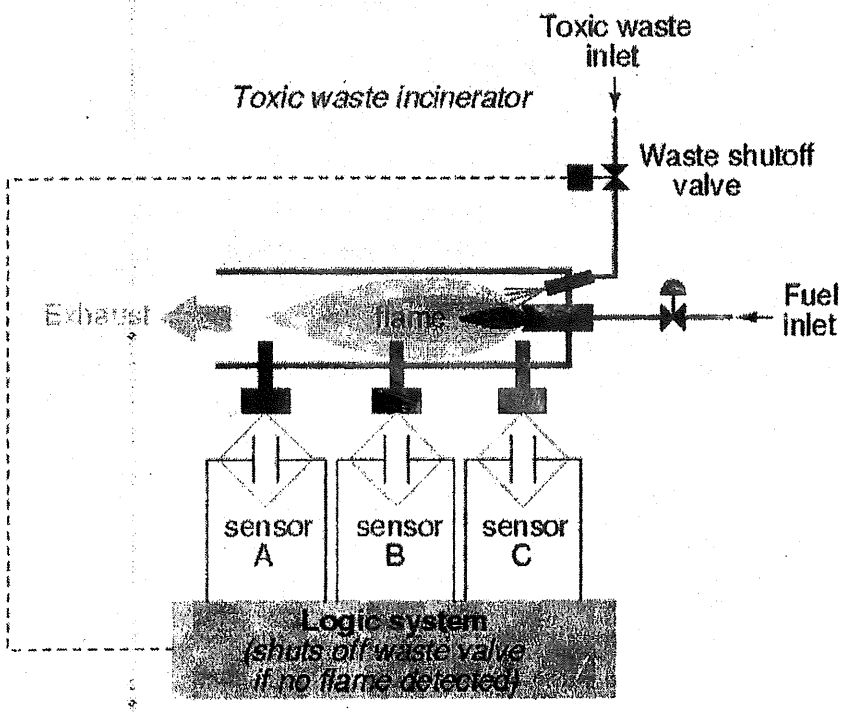
Flame detection circuit

Suppose we were given the task of designing a flame detection circuit for a toxic waste incinerator. The intense heat of the fire is intended to neutralize the toxicity of the waste introduced into the incinerator. Such combustion-based techniques are commonly used to neutralize medical waste, which may be infected with deadly viruses or bacteria:

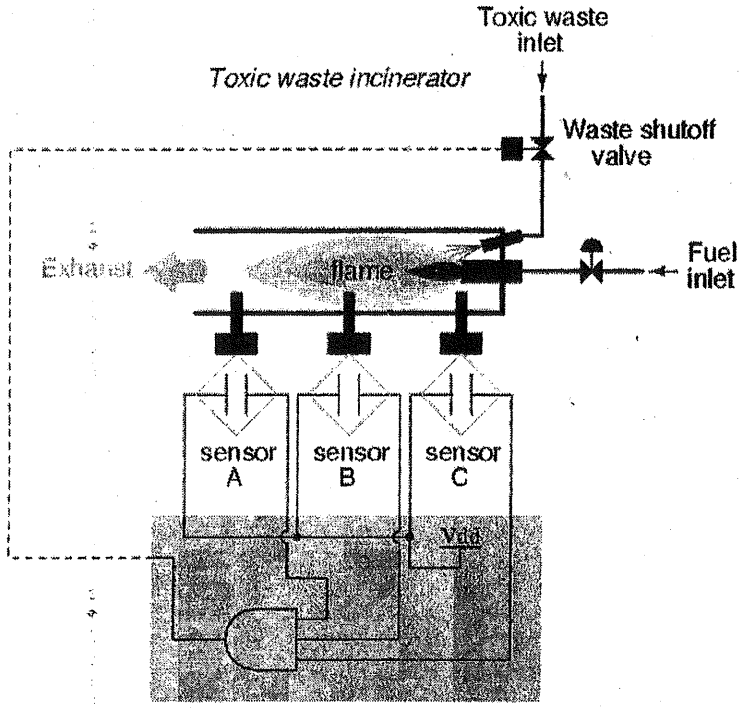


So long as a flame is maintained in the incinerator, it is safe to inject waste into it to be neutralized. If the flame were to be extinguished, however, it would be unsafe to continue to inject waste into the combustion chamber, as it would exit the exhaust un-neutralized, and pose a health threat to anyone in close proximity to the exhaust. What we need in this system is a sure way of detecting the presence of a flame, and permitting waste to be injected only if a flame is "proven" by the flame detection system.

Several different flame-detection technologies exist: optical (detection of light), thermal (detection of high temperature), and electrical conduction (detection of ionized particles in the flame path), each one with its unique advantages and disadvantages. Suppose that due to the high degree of hazard involved with potentially passing un-neutralized waste out the exhaust of this incinerator, it is decided that the flame detection system be made redundant (multiple sensors), so that failure of a single sensor does not lead to an emission of toxins out the exhaust. Each sensor comes equipped with a normally-open contact (open if no flame, closed if flame detected) which we will use to activate the inputs of a logic system:



The following system signal the valve to open if *all three sensors* detect a good flame



Truth Table

sensor inputs

A	B	C	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Output = 0 (close valve)

Output = 1 (open valve)

While this design strategy maximizes safety, it makes the system very susceptible to sensor failures of the opposite kind. Suppose that one of the three sensors were to fail in such a way that it indicated no flame when there really was a good flame in the incinerator's combustion chamber. That single failure would shut off the waste valve unnecessarily, resulting in lost production time and wasted fuel (feeding a fire that wasn't being used to incinerate waste).

It would be nice to have a logic system that allowed for this kind of failure without shutting the system down unnecessarily, yet still provide sensor redundancy so as to maintain safety in the event that any single sensor failed "high" (showing flame at all times, whether or not there was one to detect). A strategy that would meet both needs would be a "two out of three" sensor logic, whereby the waste valve is opened if at least two out of the three sensors show good flame. The Boolean expression for such a system would look like this:

$$\text{Output} = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$

We can easily design a logic gate or relay logic circuit based on that expression: