COMPUTER TRAINING PRIMER

JULY 1970

FAIRCHILD
SYSTEMS TECHNOLOGY

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Continued)

FAIRCHILD COMPUTER TRAINING PRESCHOOL PRIMER

INTRODUCTION

This computer primer affords a prospective trainer the opportunity to prepare himself in digital computer basics before attending the Fairchild Computer Training Courses. The approximate studying time is ten hours.

Course Description

Only those subjects prerequisite for a person entering the training program are presented, and their extent of coverage has been purposely limited in scope to minimize study time. The order of presentation starts with explanations of number systems and a little discussion of their application in digital computers. Examples of arithmetic using both octal and binary numbers and conversion from one number system to another are included.

Next, a brief description of a hypothetical general purpose digital computer and its use for solving a problem are presented. The development of a small computer program, including flowcharts and use of the program are included. Also, various types and levels of computer programming are concisely described.

A rather concise introduction to Boolean algebra precedes the explanation of the basic logic elements comprising digital computer circuits. These two topics, Boolean logic and circuit elements that implement the logic, are interrelated to one another and to the previously described binary arithmetic.

The primer is concluded with a brief description of the FST-1 Computer System. The computer's outstanding operational features and the functional organization of its major units are described.

Objective

The only objective to this primer was stated in the opening paragraph of this introduction: to present the prospective digital computer trainee with basic information prerequisite to his entering the Fairchild digital-computer training course. The most important concepts are stated, and from time-to-time throughout the text these are reiterrated for the reader's advantage. The main ideas and information that should "stick with" the trainee after the presentation of a particular topic are itemized so the trainee may test himself to make sure that he has placed the proper emphasis on learning the subject.

A quiz shall be given to trainees at the beginning of the Fairchild computer course. This is to test each person to make sure that he has sufficient background, as covered in this self-study course, to proceed with the more specific and much more involved training to be offered in the main computer course itself.

* * * * * * * *

NUMBER SYSTEMS

Introduction

A digit is a symbol that represents a quantity—such as 1 or 2—or nothing—such as 0. It has only one unit. A number may be one digit, such as all the numbers 0 through 9, or it may be a group of digits, such as 10, 11, 422, and so on. Digital computers operate with digits. These computer digits are the most basic concept upon which intelligence, information, quantities, and various codes that represent instructions are formulated. Thus, digital computers "talk" and "do things" because of and with digitally represented instructions and quantities. Digits and numbers comprise the most basic elements of digital-computer language.

There are numerous number systems having their own digits, and these digits represent various numbers. Our generally used number system has ten digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) and is called the decimal system. Its radix is 10, which means it has ten digits. Three other number systems widely used in the digital-computer business are the octal system, radix 8; the binary system, radix 2; and the hexadecimal system, radix 16. The same digits, borrowed from our Arabic number symbols, are used in part or in total, as needed, to represent the basic quantities for these various number systems. Therefore, since there are occassions when various number systems are used and represented with the same digits in digital-computer notation, it is necessary in these cases to designate the number system represented by a number. This is done by placing a subscript to the number that designates the radix (or as it is more frequently called the "base") of the number. Thus, 22_8 is read, "twenty-two to the base eight." In this case, the number system in use is the octal number system. Any other radix may likewise be designated.

The binary number system is the simplest. For this reason, it is the most practical system for digital computers, which are very simple-minded machines. The computer circuits are based on the binary number system. Likewise, Boolean logic is based on a binary-logic concept devised by George Boole, a nineteenth century logician. Thus, Boolean logic may also be called binary logic and provides the rules upon which binary arithmetic is performed. Thus, it is essential that the binary number system, binary arithmetic, and a little Boolean (binary) logic be understood to appreciate computer operation and the functioning of computer circuits.

The decimal number system is readily understood and used in our everyday lives. Thus, it is used most widely, and the other number systems are related to it, first, and then, to one another in the ensuing explanations.

Although the hexadecimal number system is used in the computer business, it is not used in the central processor of the FST-1. This leaves the octal and binary number systems that must be mastered.

The octal number system, base eight, is useful because it is easier to relate the binary to the octal number system than to the decimal number system. The reason for this is that 8 is a power of 2 whereas 10 is not. Consequently, the octal number system is widely and frequently used in digital-computer notation.

Decimal Numbers

Because the decimal number system has been used for so long and for most people is the only system that they use, we usually work without needing to know its organization. Also, since we learn its rules very early in life and use them over and over again, we work with decimal numbers and do not even have to think about the rules: the whole decimal number system has become second nature to us.

The least-significant digit in a decimal number represents units, the next left-hand digit represents sets of 10 units each, the next left-hand digit represents sets of 10 sets of 10 units (or 100 units), and so forth. Thus, the digits in a decimal number represent sets of units with the number of units per set contingent on the location of the digit. So each location corresponds to a particular power of the radix 10. An example of the buildup of a multi-digit decimal number is shown in Figure 1.

Units Per Set	100,000	10,000	1,000	100	10	1
Power of Ten	5	4	3	2	1	0
Digit	4	0	1	3	5	4
						4
					5	0
				3	0	0
			1	0	0	0
		0	0	0	0	0
	4	0	0	0	0	0
	4	0	1	3	5	4

Figure 1. Build-Up of Multi-Digit Decimal Number

Octal Numbers

The octal number system is based on eight digits, 0 through 7. The 8 and 9 do not exist in the octal system. Also, the location of digits by columns in the octal number system represent corresponding powers of eight. An example of the build-up of a multi-digit octal number is shown in Figure 2.

Units Per Set	100,000	10,000	1,000	100	10	1
Powers of Eight	5	4	3	2	1	0
Digit	4	0	1	3	5	4
						4
					5	0
				3	0	0
			1	0	0	0
		0	0	0	0	0
	4	0	0	0	0	0
	4	0	1	3	5	4

Figure 2. Build-Up of Multi-Digit Octal Number

Note that the build-up of the decimal and octal numbers shown are identical. It worked out this way because there were no eight or nine digits in the decimal number, and, of course, these two digits do not exist in the octal number system. However, to show the relationship between the octal and decimal number systems, in Figure 3 the same number, 401354, used in both preceding examples is used as an octal number, and its decimal equivalent, 131820, is compared with it. Also, both the octal notation and its decimal-equivalent are shown. Thus, by multiplying the octal-number digit (from the digit row) times the number in either the octal-notation or decimal-equivalent row, the octal number or its decimal equivalent may be built.

Not only does Figure 3 illustrate the build-up of an octal number, it also shows a direct method of converting an octal number to its decimal equivalent. There is another method for doing this, but because it requires doing octal arithmetic, it is more difficult and cumbersome to perform—unless one is adept at octal arithmetic. Figures 4 and 5 are octal addition and multiplication tables. By eliminating the two digits 8 and 9, the quantities represented by these decimal numbers, 8_{10} and 9_{10}, are now represented by octal numbers 10_8 and 11_8. All octal numbers 12_8 through 19_8 represent quantities likewise two less than the corresponding decimal number. The next higher-quantity digit has a value four less than its decimal equivalent: 20_8 versus 16_{10}. The additional decrease in the value represented by octal numbers as compared to that value represented by decimal numbers continues through digit 7. Since there are no 8's and 9's in octal numbers, the number following 77_8 is 100_8. The decimal value of 100_8 is 64_{10}.

Units Per Set	Octal Notation	100,000	10,000	1,000	100	10	1
	Decimal Equivalent	32,758	4,096	512	74	8	1
Powers of Ten or Eight		5	4	3	2	1	0
Digit	Octal Number	4	0	1	3	5	4
	Decimal Equivalent	1	3	1	8	2	0

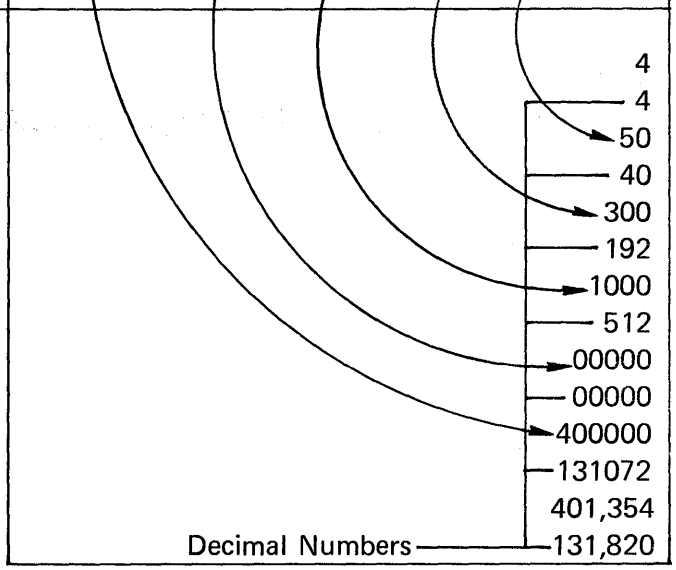


Figure 3. Comparison of Build-Up of Octal Number and its Decimal Equivalent

	01	02	03	04	05	06	07
1	02	03	04	05	06	07	10
2	03	04	05	06	07	10	11
3	04	05	06	07	10	11	12
4	05	06	07	10	11	12	13
5	06	07	10	11	12	13	14
6	07	10	11	12	13	14	15
7	10	11	12	13	14	15	16

Figure 4. Octal Addition Table

1	02	03	04	05	06	07
2	04	06	10	12	14	16
3	06	11	14	17	22	25
4	10	14	20	24	30	34
5	12	17	24	31	36	43
6	14	22	30	36	44	52
7	16	25	34	43	52	61

Figure 5. Octal Multiplication Table

Thus, the larger the octal number, the greater is its difference in value from the corresponding decimal number. Figure 6 relates a few of the powers-of-eight written in octal notation to corresponding powers-of-eight written in decimal notation.

Octal Notation		Decimal Notation
10_8	- 8_{10}	
100_8	- 64_{10}	
$1{,}000_8$	- 512_{10}	
$10{,}000_8$	- $4{,}096_{10}$	
$100{,}000_8$	- $32{,}768_{10}$	
$1{,}000{,}000_8$	- $262{,}144_{10}$	
$10{,}000{,}000_8$	- $2{,}097{,}152_{10}$	
$100{,}000{,}000_8$	- $16{,}777{,}216_{10}$	

Figure 6. Powers-of-Eight, Octal Versus Decimal Notation

Octal and Decimal Number Conversions

Because decimal arithmetic is used, the indirect method, using division by base number, to convert a number represented by one base notation to its equivalent-value number in another base notation is first explained for converting from a decimal number to an octal number. Later, an example is given for the converse, octal-to-decimal conversion.

The following rules are stated for converting from a decimal to an octal number. Nevertheless, the same principles and technique apply for conversions from any number system to another.

1. Take the octal radix, represented in decimal notation. This is the number 8_{10}, and it is the divisor.

2. Divide the decimal number by 8 repetitively until the dividend is exhausted. See Figure 7. Use decimal arithmetic rules.

3. Form the octal number from the remainders as shown in Figure 7.

An octal-to-decimal conversion is illustrated in Figure 8. You will note that the divisor is the radix 10_{10} but it is represented in octal notation as 12_8. Also, all of the arithmetic is carried out by octal rules of addition and multiplication as shown in Figures 4 and 5.

Another method for converting from octal to decimal numbers uses decimal-number arithmetic and generally is easier to use unless one is quite adept in octal-number arithmetic. To use this method, referring to Figure 9, multiply the most significant digit of a number by 8, add the next most significant digit, multiply the resulting sum by 8, add the next most significant digit, continuing until the least-significant digit is processed.

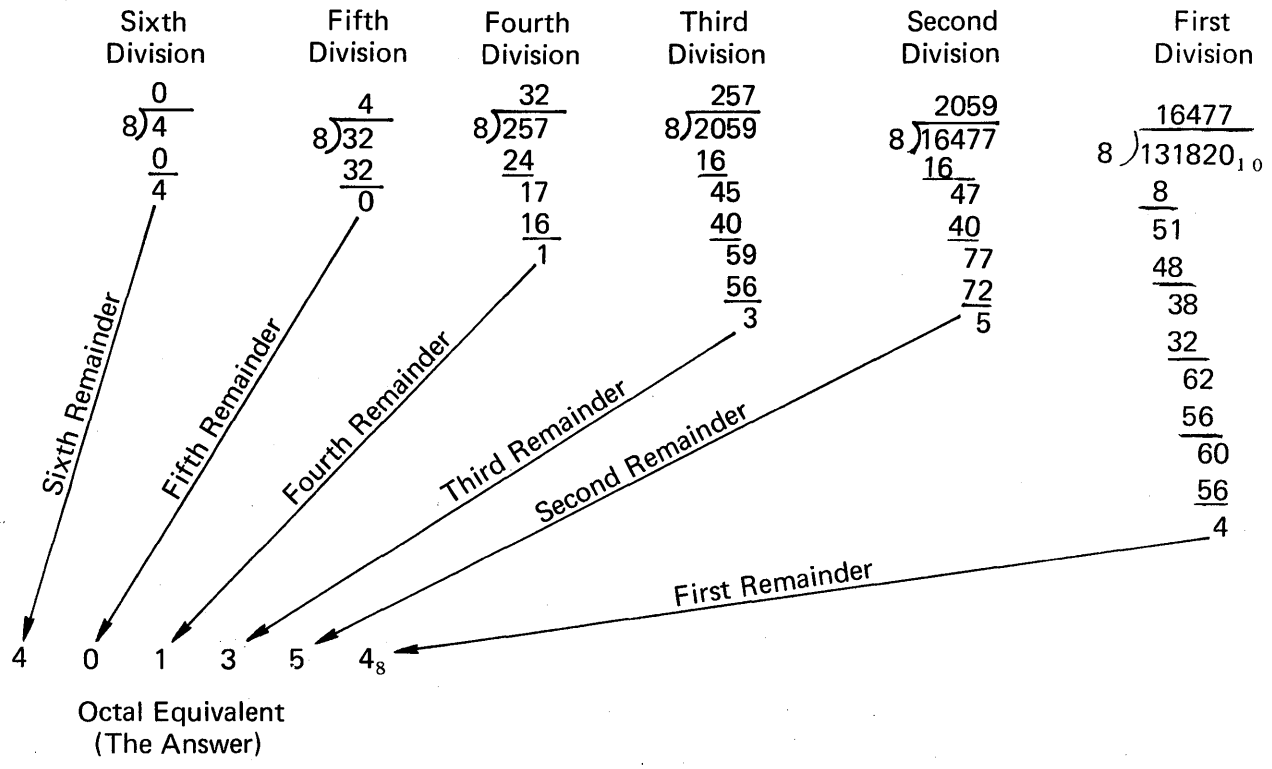


Figure 7. Decimal-to-Octal Number Conversion

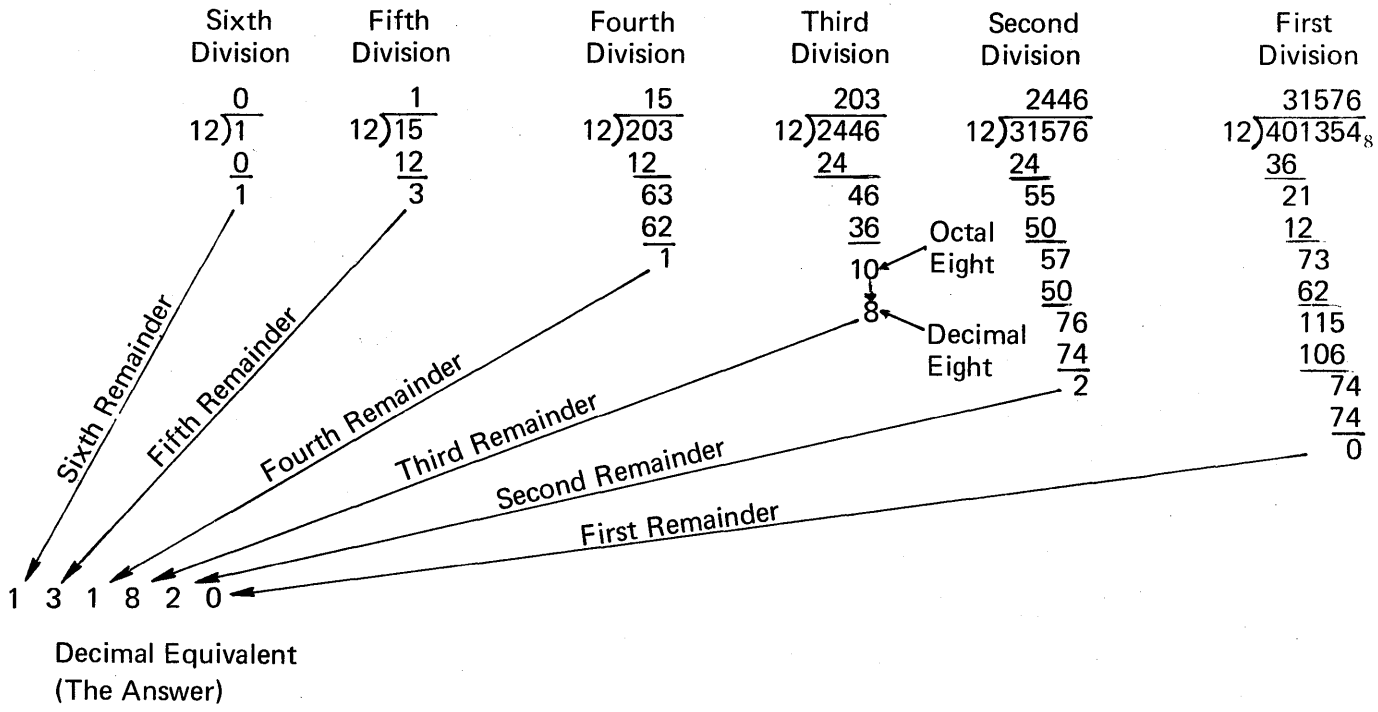


Figure 8. Octal-to-Decimal Number Conversion by Division Using Octal Arithmetic

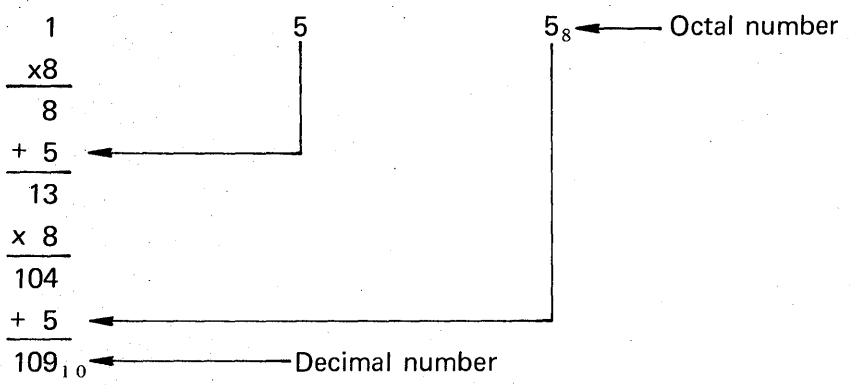


Figure 9. Octal-to-Decimal Number Conversion by Addition/Multiplication

It is suggested that not too much time be spent practicing on the octal-to-decimal conversion since it is seldom used in operational and maintenance work.

Primarily, an understanding of the principles involved and the ability to convert from decimal to octal numbers are all that should be mastered at this time. Digital computer operational codes, addresses, and the like are noted in the octal number system and there is no need to change these to the decimal number system. However, the digital computer works in the binary number system, and, therefore, the binary number system and its relationship to both the octal and decimal number systems are prerequisite to learning digital computer operation and logical functioning.

Binary Numbers

The binary number system is based on the radix 2_{10}. You will note that the number 2 is a decimal number: Thus, the reason for showing the subscript 10. There is no digit 2 in the binary system. Because the decimal system is most widely used and understood, it is customary to use its digits as a reference in designating radices. Likewise, even though there is no 8 in the octal system, octal numbers are customarily designated with 8 as subscripts: 10_8, 21_8, 100_8, 557_8, and so forth. When working with mixed number systems, it is a good idea to designate the bases to avoid ambiguity.

There are only two binary digits: 0 and 1. Both of these digits may be used in various combinations to represent any desired quantity. By virtue of the location of a digit it represents a particular value based on the power of two. However, whereas with a many-digit number system many sets of the radix's power may be designated, only one set may be designated in the binary system. This is accomplished with the digit 1. The digit 0 indicates nothing. Either of these binary digits is called a bit, coined from the term BInary digiT. Thus, each bit position must be either a 0 or a 1. The location of a bit in the binary number determines its value–based on a particular power of two. Figure 10 shows the build-up of a 6-bit number.

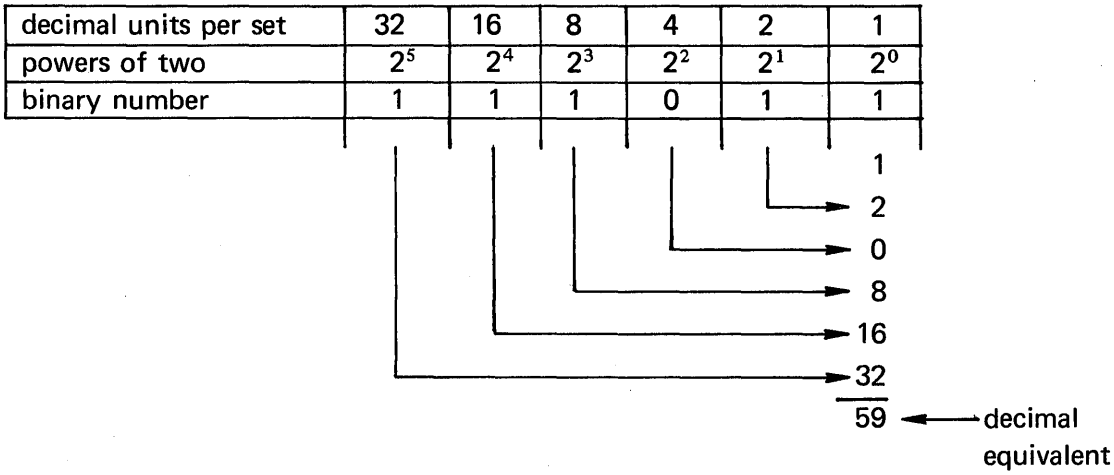


Figure 10. Build-Up of Binary Number

Binary and Octal/Decimal Number Conversions

It is also possible to convert the decimal number 59 to a binary number using the division method. This is demonstrated in Figure 11.

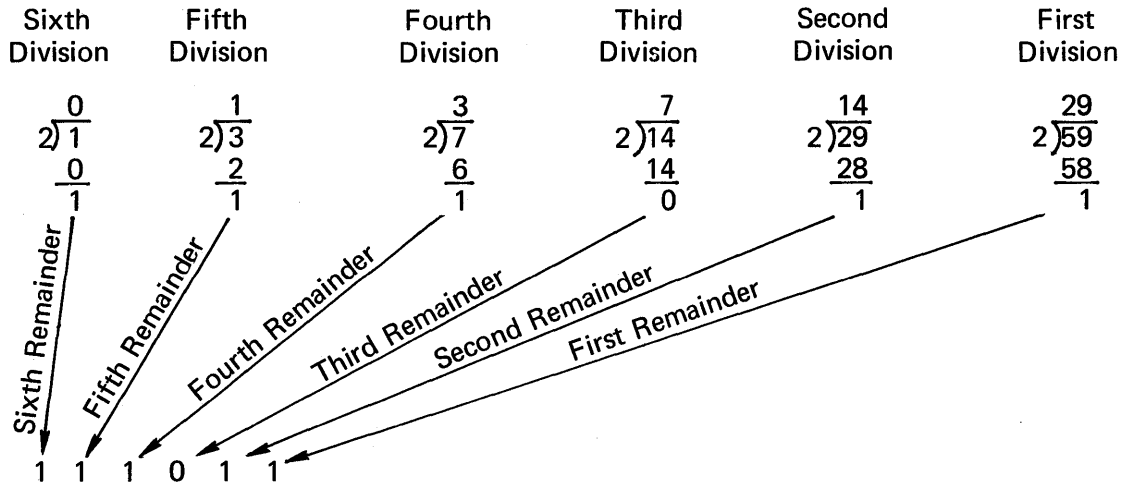


Figure 11. Decimal-to-Binary Number Conversion

Conversely, the binary number may be converted to a decimal number using the division method. However, binary arithmetic rules must be used. These are very simply stated:

1. 1 + 0 = 1
2. 1 + 1 = 10
3. 1 x 0 = 0
4. 1 x 1 = 1
5. 1 - 0 = 1
6. 1 - 1 = 0
7. 1 ÷ 0 = 0
8. 1 ÷ 1 = 1

In addition, carries and borrows are allowed.

With the preceding binary arithmetic rules applying, Figure 12 shows the conversion of the binary number 111011 to the decimal number 59.

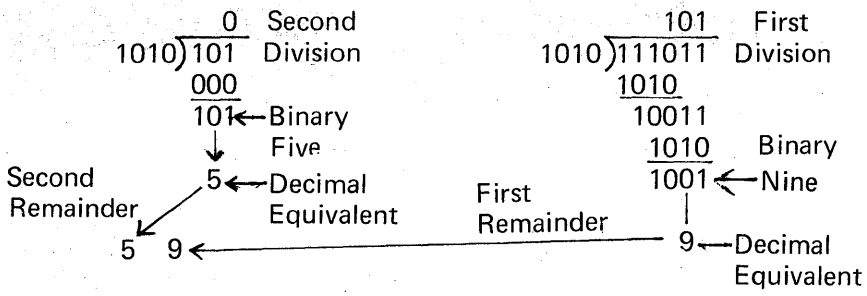


Figure 12. Binary-to-Decimal Number Conversion by Division Using Binary Arithmetic

Figure 13 shows how to convert the binary number 11110 to the decimal number 30 using decimal arithmetic rules.

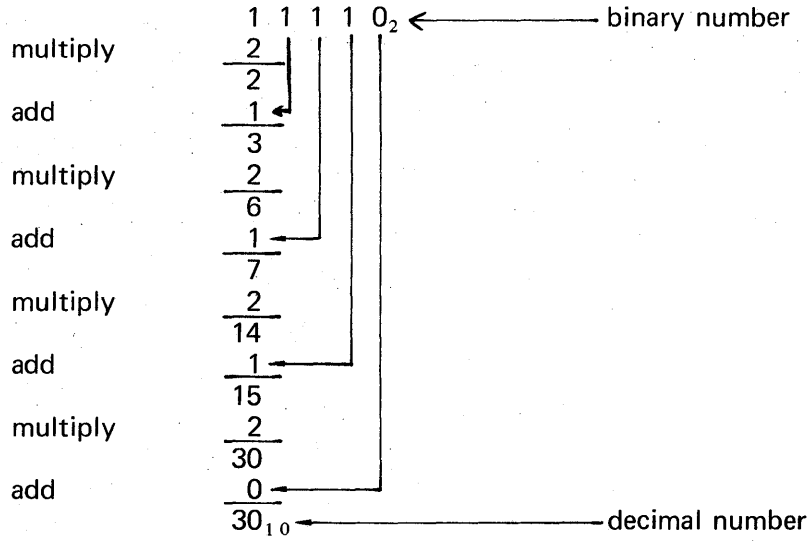


Figure 13. Binary-to-Decimal Number Conversion by Multiplication/Addition Using Decimal Arithmetic

Both decimal and octal numbers are frequently converted to binary number and vice versa in the digital computer business. Thus, it is essential that the techniques for binary-to-decimal, decimal-to-binary, binary-to-octal, and octal-to-binary conversions be mastered to the degree that they are performed quickly and accurately.

Coded Binary Numbers

A frequently used technique used for relating octal and binary quantities to one another is to code a binary number as octal digits or vice versa. This is easily accomplished because the powers of eight may be expressed as powers of two. Thus, by dividing binary numbers into groups of threes, each group may be coded to represent all octal digits from 0 through 7. Conversely, any octal digit may be represented by a group of three bits. A few examples of binary-coded-octal numbers and their octal equivalents are shown in Figure 14. There are no unused quantities in binary-coded-octal numbers, and conversions in either direction are readily performed mentally. In addition, if all the binary digits (bits) are converted to an octal number, the octal number thus obtained is the same number obtained by converting groups of three bits to octal digits and combining these digits to form an octal number. However, because of our familiarity with the decimal number system, in Figure 14 the binary number equivalent to 777_8 is converted to its decimal equivalent 511_{10}, and, then, 511_{10} is converted to 777_8 using the division method. Also, the build-up of the octal number 777_8 is shown side-by-side with the build-up of the decimal equivalent 511_{10}. Figure 14 illustrates nearly all that must be understood about the relationships among binary, octal, and decimal numbers. The only other concept not yet explained is the representation of a decimal number in the binary-coded-decimal (BCD) notation. This is covered next.

A decimal number is not so easily related to a binary number as is an octal number. This is because it requires four binary digits (bits) to represent quantities corresponding to those represented by the decimal digits 8 and 9. But four bits can represent 16 different quantities, 0 through 15. Thus, the representations 10 through 15 in a 4-bit group are wasted. Also, the binary-coded-decimal (BCD) number is not the same as the decimal number converted from the binary number. For these two reasons, binary-coded-octal numbers are preferred from standpoints of ease-of-use and conversion to binary numbers. Nevertheless, because most business is carried out in the decimal number system and because most of us are, therefore, more oriented to the decimal number system, decimal numbers are sometimes represented in 4-bit groups, each of which is readily converted to a decimal number and vice versa. Figure 15 shows the number 698_{10} coded in BCD format. Conversion in either direction is readily apparent.

Binary Number Arithmetic

Binary numbers are used to represent quantities in the simpliest way. By having only two digits in the number system, addition, subtraction, multiplication, and division are easily carried out by the manipulation of combinations of zeros and ones. However, it requires more digits in the binary number system to represent any quantity greater than one, and the greater the magnitude of the binary number, the more bits, and the more difficult it is to recognize the larger numbers. For this reason, we continually talk and think in decimal

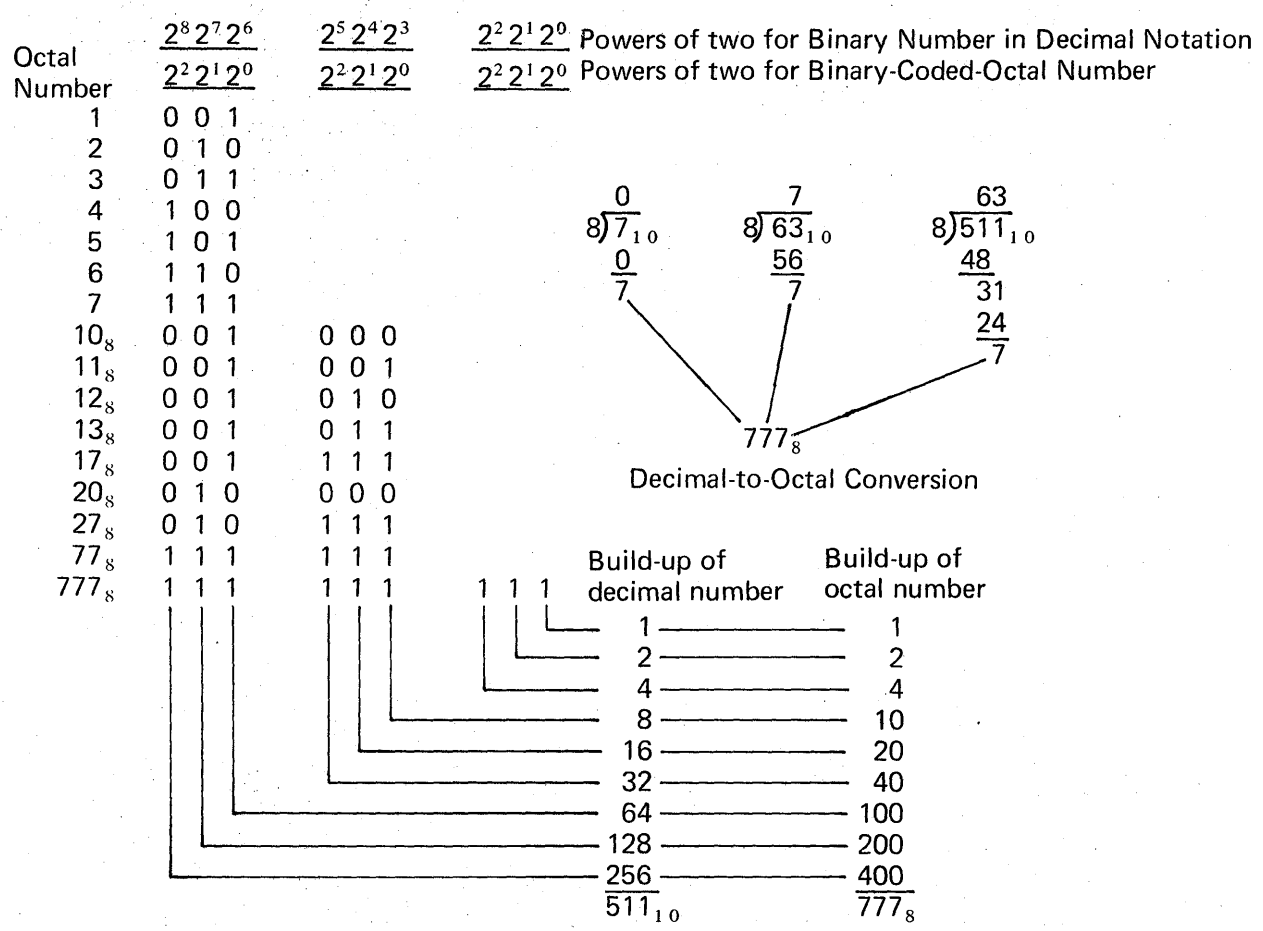


Figure 14. Binary/Binary-Coded-Octal Numbers and Relationships to Octal and Decimal Numbers

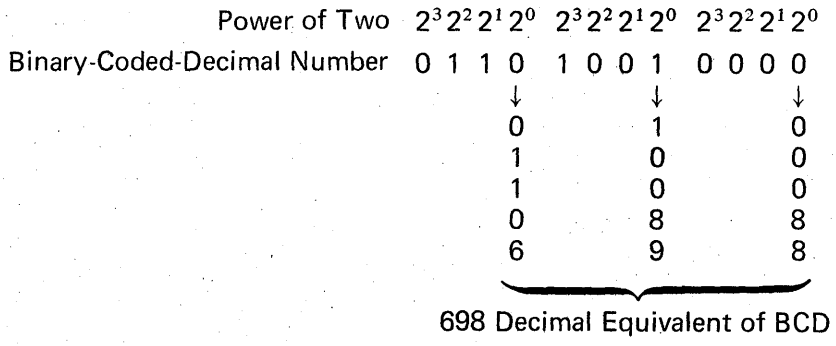


Figure 15. Binary-Coded-Decimal Number and Conversion to Decimal Number Equivalent

or octal numbers but the machines work in binary numbers. For these reasons, the following explanations of binary arithmetic are referenced to the decimal equivalents. Decimal numbers are used because of our familiarity with them. However, as time permits, it is a good practice to get use to octal numbers and arithmetic, thus simplifying conversions of the machine's and the people's working numbers.

Binary numbers are usually classified as so many bits long. The number of bits determines the maximum magnitude that a number may represent. Computers are designed to handle numbers of certain bit sizes. This imposes a limitation on the size of the number.

Nevertheless, some rather large numbers are handled by computers. Typical bit lengths are 12, 16, 24, 32, and 64. The FST-1 uses 24 operational bits, so the maximum magnitude of a 24-bit number is $16{,}777{,}215_{10}$. However, to simplify explanations the binary arithmetic discussions that follow are based on smaller numbers; the principles, of course, are applicable for any size number.

A simple but very important concept that we must get use to in thinking about binary numbers is that there is no symbol for digit or number 2. All quantities must be expressed in 0's and 1's. The addition of 1 plus 1 gives the number 10_2, which is equivalent to 2_{10}. The addition of $10_2 + 10_2$ equals 100_2: That is, $2_{10} + 2_{10}$ equals 4_{10}. Figure 16 is the binary addition table.

Augend Digit	0	1	0	1
Addend Digit	0	0	1	1
Sum Digit	0	1	1	0
Carry	0	0	0	1

Figure 16. Binary Addition Table

Bit Addition

An example of adding two binary numbers appears in Figure 17. Binary addition is performed just like decimal addition. Start at the right and add the first pair of bits. Continue leftward, adding pairs of bits, considering whether a carry has occurred in the previous bit addition.

Decimal	Binary	
13	001101	Augend
21	010101	Addend
34	100010	Sum

Figure 17. Binary Addition and Decimal Equivalent

Digital computers add binary numbers in a manner similar to the preceding example.

In adding two bits, four cases are possible.

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

Bit Subtraction

To subtract two bits

1 - 1 = 0
0 - 0 = 0
1 - 0 = 1
0 - 1 = 1 with a 1 borrowed for the next left-hand bit

Figure 18 is an example of binary subtraction.

Decimal	Binary	
34	100010	Minuend
21	010101	Subtrahend
13	001101	Difference

Figure 18. Binary Subtraction and Decimal Equivalent

Figure 19 is the binary subtraction table.

Minuend	0	1	0	1
Subtrahend Digit	0	0	1	1
Difference Digit	0	1	1	0
Borrow	0	0	1	0

Figure 19. Binary Subtraction Table

Bit Multiplication

In any number system, zero times zero yields zero. Since the only remaining pair is one times one, bit multiplication is simply summarized in Figure 20.

		Multiplicand Digit	
		0	1
Multiplier Digit	0	0	0
	1	0	1

Figure 20. Single-Bit Multiplication Table

Figure 21 illustrates the multiplication of two binary numbers. As in decimal multiplication, a bit of the multiplier multiplies the entire multiplicand. The result is placed below the multiplier so the right-most bit of the result is aligned under the multiplier bit just used. The multiplication example in the center of the figure is self-explanatory. However, it may

be shortened if we consider that for each 0 in the multiplier, the partial product corresponding to that 0 bit is equal to zero. Thus, by showing only the partial product corresponding to each 1 in the multiplier, the multiplication example on the right-hand side of the figure apply.

```
 13          1101          1101          Multiplicand
  9          1001          1001          Multiplier
---         ----          ----
117          1101          1101   }      Partial Products
              0000          1101   }
             0000          ----          Full Product
            1101          1110101
           ----
           1110101
```

Figure 21. Binary Multiplication, Long and Short Method, with Decimal Equivalent

Bit Division

Binary division may be considered the reverse of binary multiplication. The process of binary division is identical to that of decimal division. Figure 22 is the binary digit division table and has the same format as the binary multiplication table.

		Dividend Digit	
		0	1
Divisor Digit	0	0	0
	1	0	1

Figure 22. Single-Bit Binary Division Table

Two division examples are given in Figure 23. These are in short-division form. The same quantities of the preceding multiplication examples are used, the products as the minuends and the multiplicands as the divisors.

```
   13                    divisor    1101  quotient
9)117                   1001)1110101  dividend
  9                          1001
  27                         1011
  27                         1001
                               1001
    9                          1001
13)117                              no remainder
  117                        1001
                        1101)1110101
                             1101
                               1101
                               1101
                                    no remainder
```

Figure 23. Binary Division with Decimal Equivalent

Thus far, we have discussed binary numbers in terms of magnitude only. To distinguish positive from negative numbers, it is customary to use one bit to designate the sign of a binary number and to place this bit at the beginning of the number. For positive numbers this bit is zero, and for negative numbers it is one. This, of course, decreases the magnitude of the maximum number by a factor of two, because each left-hand bit position represents the next power of two. Disregarding the sign, with N bits a number 2^N - 1 bits long may be represented. Using the left-hand bit as a sign this decreases the number size to 2^N -2 bits.

In the interest of simplicity we can disregard the sign of binary numbers for now and assume that we are dealing with positive, whole numbers. However, computers must take into consideration both the sign and whether or not a number is whole or a fraction.

Complements

The true complement of a number is that number which if added to the original number produces a sum equal to the power of the radix in the location of the original number's most-significant digit plus one. For instance, the complement of 6 in the decimal system is 4. Adding the complement 4 to the original number gives a sum of 10. The original number's most-significant digit is in the units or 10^0 position. Thus, the power is 0 and 0 + 1 equals 1. The number 10 is the base 10 raised to the first power, 10^1.

Thus, in the decimal system a complement is obtained by subtracting a number from a number equal to a power of 10. The power of 10 used is such to produce a number with one more digit than the original number. For example:

1. For numbers 1 through 9—subtract from 10
2. For numbers 10 through 99—subtract from 100
3. For numbers 100 through 999—subtract from 1000
4. And so forth

This method applies to numbers having any radix. However, this gives the TRUE complement. There are other special complements that are used, usually to simplify circuits or to enhance computer operation.

Complementing numbers in the binary system may be effected likewise. For example, the complement of the binary number 1 is 1, and it is determined by subtracting 1 from the next power of the base 2, 10. Other examples are given in Figure 24.

Next higher power of radix 2	100	100	100000
Original number	10	11	11011
Complement	10	01	00101

Figure 24. Example of Obtaining TRUE Complement by Subtracting Next Higher Power of Two

Another very simple method of complementing a binary number is to change all ones to zeros, change all zeros to ones, and add one to the binary number thus formed. Figure 25 illustrates this technique.

Original number	10	11	11011
Ones Complement	01	00	00100
Add 1	1	1	1
True (twos) complement	10	01	00101

Figure 25. Example of Obtaining TRUE Complement by Reversing Bits and Adding One to Result

The complements of binary numbers have many uses in digital computer technology—as do other variations of representing binary numbers. The use of complements in arithmetic, however, is our immediate concern. To economize on arithmetic circuits in digital computers, the arithmetic unit usually is capable of either adding or subtracting but not both. Thus, the complement of a number is used as one of the quantities either to effect the desired operation, either addition or subtraction depending on the form of the arithmetic unit.

If a computer's arithmetic unit is an adder, then, the complement is used as the subtrahend in subtraction. If the arithmetic unit is a subtracter, then, the complement is used as the addend in addition. Figure 26 shows both uses of using a true complement of a binary number.

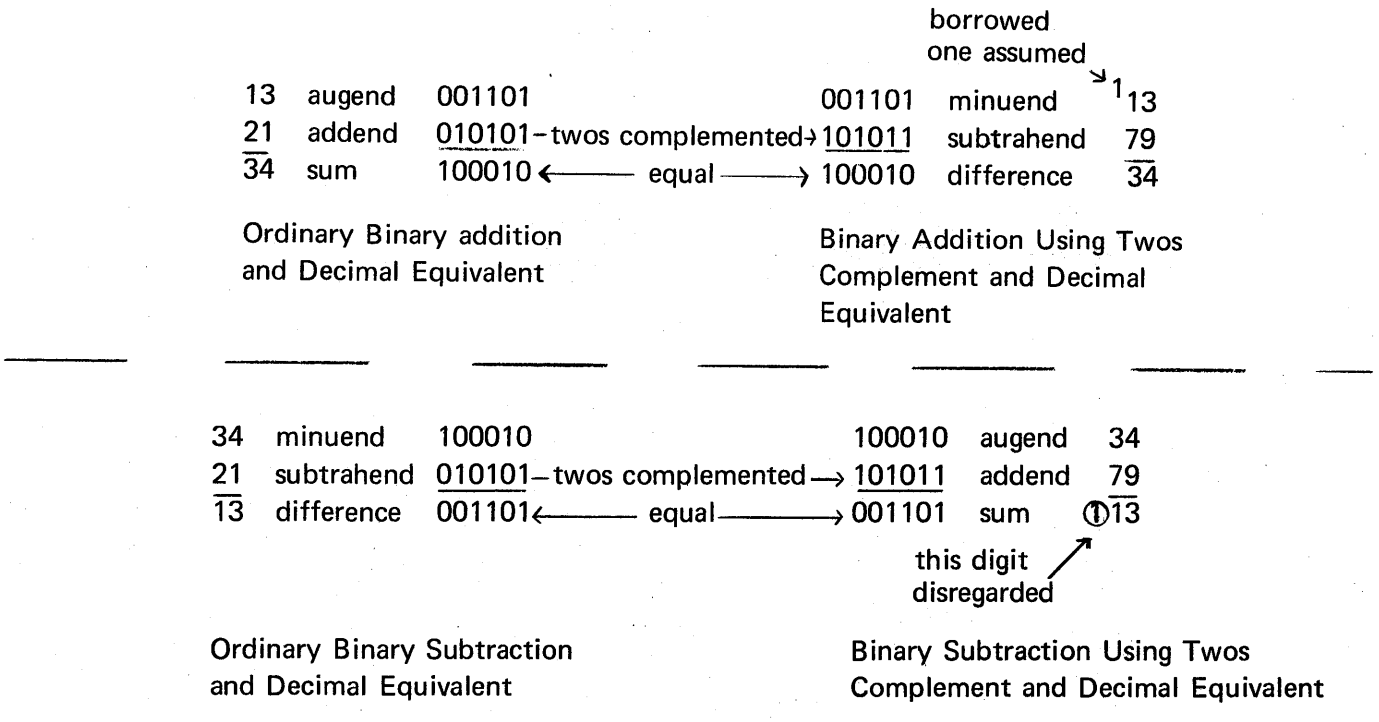


Figure 26. Examples Comparing Ordinary Binary Addition and Subtraction to Twos-Complement Methods

Some liberties are taken in complementary arithmetic so borrows are assumed and carries are disregarded for the most-significant digits, as illustrated in the preceding examples.

Another precaution that must be taken before complementing a number is to add zeros to the left-hand side of the number, as needed, until the number has the same number of digits as the other number to be involved. The complement is obtained by subtracting from the number equal to the radix raised to the power equal to one more than the number of digits in the extended number. For example:

Direct Addition		Complementary Addition	
Augend	34	34	
Addend	08	02	← wrong complement from 10^1 (10)
	42	32	← wrong answer

Borrow Assumed

1	34	
	92	← correct complement, from 10^2 (100)
	42	← correct answer

The preceding examples use decimal numbers but the same principle applies to any number system. In complementing binary numbers the same precaution is necessary. The simpliest way to make sure it is taken care of is to be sure to always have the same number of bits designated in the two numbers involved. For example:

	100010	(34)	Augend
Extension Bits	001000	(8)	Addend
	101010	(42)	Sum

001000	———	Binary 8
110111	———	ones complement of binary 8
1	———	add one
111000	———	twos (true) complement of binary 8
		(Note: does *not* equal decimal 2)

Borrow Assumed

1	100010	(34)
	111000	---
	101010	(42)

Note in the preceding example that the addend extension is necessary to get the correct answer by subtracting the complement from the original augend. It made no difference in doing the straight-forward binary addition, however, but to prevent inadvertently making mistakes in arithmetic, it is suggested that binary numbers be appropriately extended to the full bit size.

A short-cut method for complementing a binary number is illustrated in Figure 27. Starting with right-hand bit (least-significant), copy the original number until the first bit to the left-hand side of the least-significant 1. Then, including the first bit more significant than the first 1 change all 1's to 0's and vice versa.

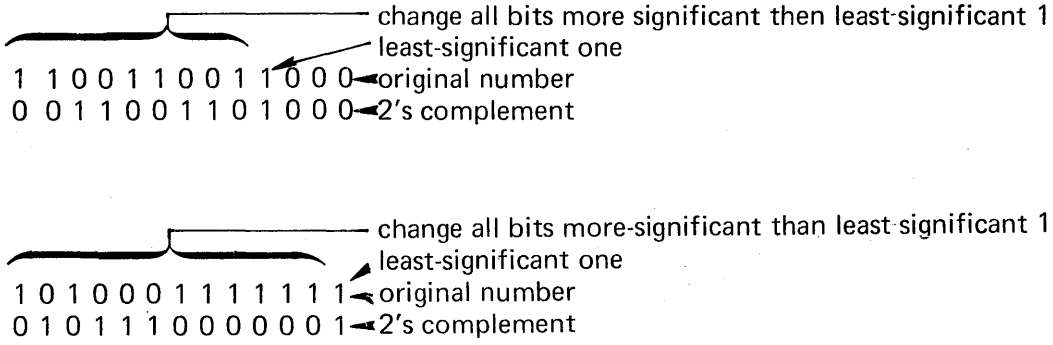


Figure 27. Short-Cut Method of Conversion to Twos Complement

Number System Summary

The foregoing coverage of number systems affords enough background for one to get started in digital-computer arithmetic and to begin to understand the languages that digital computers use. In summary, for operational and maintenance purposes, a person should have mastered the following:

1. Understand the building of numbers in the three mainly used number systems: decimal–base 10; octal–base 8; binary–base 2.

2. Convert numbers from any of these three number systems to the other two number systems. First, master the easiest technique and use it. For example, if working in octal arithmetic is difficult, as it is for most people, use the technique for converting from octal to decimal numbers that can be performed using decimal arithmetic.

3. Perform simple binary arithmetic, such as addition, subtraction, multiplication, and division.

4. Convert binary numbers to either the ones or twos complement.

5. Using the twos complement method, add or subtract to perform binary subtraction and addition, respectively.

COMPUTER WORDS

Information transferred to and from or formed within a digital computer is referred to as "data." In computerese, the word "data" designates a combination of binary ones and zeros. These ones and zeros are called bits and are formatted as "words." The position of

bits are usually referred to by a "bit number." In terms of magnitude only, Figure 28 illustrates a 24-bit word. The bit positions are designated by numbers starting with 0 as the least-significant bit and 23 as the most-significant bit.

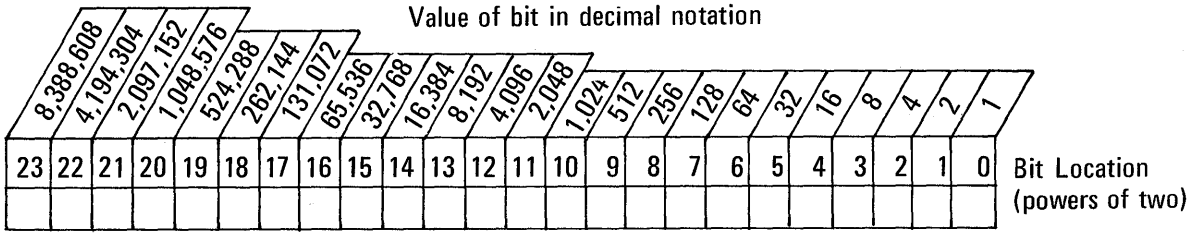


Figure 28. Format of Computer Word of 24 Magnitude Bits

Magnitude

A 1 in any bit position gives the decimal number value shown. The sum of all the values of bits that are 1's is the total magnitude represented by the computer word. However, for the FST-1 computer, bit 23 is set aside to indicate the sign of the number, a 0 for a positive number and a 1 for a negative number.

The advantage of numbering bits starting with zero and using it to designate the least-significant bit makes it easy to determine the value of a 1-bit in each bit position. The bit number is also the power of 2 that the radix, 2, is raised. Thus, bit position 0 as a 1 designates a value equal to 2^0 (or 1) bit $1-2^1$, bit $2-2^2$, and so forth.

Instructions

Some computer words represent instructions. Instructions tell the computer what to do. Computer instructions are also in the form of numbers comprising bits in various combinations having various meanings. The computer programs comprise certain instructions that may be arranged in certain orders, transferred from one operational unit to another, and stored on various media. Again, bits, BInary digiTS, comprise instructions and the computer "speaks" and "talks" and "works" only using numbers comprised of bits.

Formatting instruction words makes it possible for an instruction to comprise various parts that have different information. These parts of an instruction word vary for different types of instructions. However, for the time being let us concern ourselves with one type of instruction word. Becoming familiar with this one type of word can become a jumping-off point for tackling other instruction words later.

Figure 29 represents an instruction word, similar to what may be found in most digital computers.

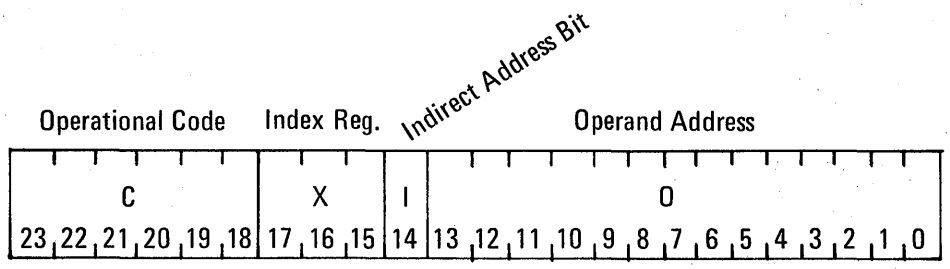


Figure 29. Typical Instruction Word Format

Bits 0 through 13 are called the "operand address." They tell the computer where in memory to fetch or to store an operand. An operand may be any of the quantities entering into or resulting from a computer operation; it may be an argument, a result, a parameter, or even an address of the next instruction.

Let us skip over bits 14 through 17 for now.

Bits 18 through 23 are called the "operational code" or more frequently the "op code." The computer has a repertoire of operational codes that it is able to perform. The op-code bits indicate one of these operations. Thus, the machine-level program commands comprise op codes. By decoding op codes obtained from a storage unit called memory, the computer is able to follow instructions from the computer program stored in memory.

Getting back to bits 14 through 17, these have special uses that afford the programmer with a more-versatile programming capability. The bit designated "I" is called the "indirect-address" or just "indirect" bit. It tells the computer to obtain its operand from the location in another instruction word located at the operand address in the current instruction. For the present, consider the significance of this bit as a special programming tool.

Bits 15 through 17 are called "index" bits. They designate special computer registers that have 14 bit positions corresponding to the operand-address bit positions. Again, this is for programming, and its use is for operand-address modification. When an instruction has indexing, the operand address stored in the designated index register is added to the operand address of the current instruction. The resulting sum represents another operand address that is then used instead of the one in the instruction. This is also a special programming tool, and its programming significance can be appreciated only after acquiring additional programming background.

There are other types of computer instructions that will be covered later.

Data, either as magnitude or instructions, are held or manipulated in units called registers and these are our next topic.

WORD REGISTERS

The various data handled by digital computers correspond to "word registers." A register here is a place where a record is kept. However, the word "register" in computerese implies

a temporary record, whereas permanent or semi-permanent records are usually called memory or storage media. Thus, we shall use the word "register" to mean a hardware unit that temporarily stores information (data) in the form of bits—or a unit that handles these bits during their manipulation.

It is not important at this time to know what the hardware is that comprises registers. Primarily, it is only necessary to understand the purpose of registers and their functional formats.

Typical Register Configurations

Two typical registers correspond to those shown in Figures 28 and 29, representing magnitude and instructions. In actual practice, a machine register comprises just so many bit positions. Whether the bits represent magnitude or instructions depends on how they are used by the machine. Therefore, until we get around to discussing hardware, our only concern with registers pertains to function, significance of the bits (format), and size.

Functional Registers

Certain basic functions are performed by the digital computer registers. These basic functions are explained in the following descriptions. Various computer manufacturers give these registers different names. Of course, special registers may be added when it is desired to afford the user of some additional capability or operational versatility.

Figure 30 is a simplified functional block diagram showing the organization of the basic registers comprising a typical digital computer.

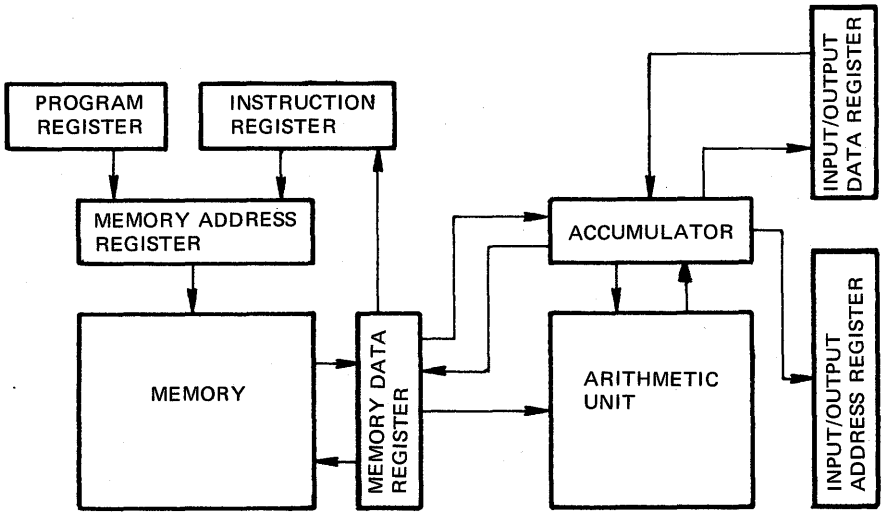


Figure 30. Organization of Registers of a Typical Digital Computer

MEMORY

The memory serves to store the program that is to be executed, store the input data unit it is needed for processing, store intermediate results during the computation, and store final results until they are ready for output. The memory is a principal element of the computer, and the cost and speed of the computer are largely governed by the cost and speed of the memory. The memory stores its contents in the form of words. A computer memory may be capable of storing fron one thousand words to several hundred thousand words or more.

MEMORY ADDRESS REGISTER

A word is stored away in the memory, or retrieved from the memory, by designating the location of a particular memory word-position in the memory, and giving the command to "store" or "fetch." The memory address register designates this address. There will be as many unique addresses (locations) in the memory as the number of words that the memory is capable of storing. The store or fetch command is given by the Control Unit, an all-prevading logical complex that determines what operation is performed at what time. It can be visualized as the "big brother" which causes the computer to perform their operations.

MEMORY DATA REGISTER

When storing a data word into memory, the data word is placed into the memory data register, the address at which it is to be stored is placed into the memory address register, and the store command causes the data word to be stored at the desired address. When fetching a data word, the address of the word is placed into the memory address register, and the fetch command causes the word to be transferred from the specified address to the memory data register.

PROGRAM REGISTER

The program consists of a sequence of instructions that the computer is to perform. The program is stored in the memory in the proper sequence; that is, the first instruction is stored in location 001, the second is stored in 002, and so forth. The program register keeps track of the location from which the next instruction is to be fetched, by counting the instructions as they are performed. For example, if the fifth instruction, which was located in memory position 005, is being performed, the protram register has counted to 006, the address of the next instruction. When the time comes to fetch the next instruction from memory to determine the operation it specifies, the program register will transfer the contents of 006 into the memory address register.

INSTRUCTION REGISTER

It is extremely important to recognize that the memory contains both instructions to be performed (the program) and the data on which the instructions are to be

performed, and that these two kinds of stored words are treated in two entirely separate ways. The address of an instruction is specified by the program register; when fetched into memory data register, the instruction is transferred to the instruction register, where it is examined by the control unit to determine:

What operation (add, multiply, or some other) ?
Where is the addend (or multiplicand, or other) located?
Where is the augend (or multiplier, or other) located?
Where should the result be placed?

If, as is usually the case, one of the operands is to be obtained from the memory, the appropriate address, contained in the instruction, is furnished to the memory address register by the instruction register.

ACCUMULATOR

As the instruction register is taxed with the handling of all instruction words as they are retrieved from memory, the accumulator serves an equally important function for the data words. In general, the accumulator contains one of the operands for any arithmetic operation, the other operand being in memory, and the result of an arithmetic operation is usually placed into the accumulator. Data words can be fetched from memory to the accumulator, or stored from the accumulator into the memory.

ARITHMETIC UNIT

Performs the specified arithmetic operation between a word contained in the accumulator and a word fetched from memory into the memory data register. (This describes a single-address computer; more-complex computers can carry out arithmetic operations between two words in memory.)

INPUT/OUTPUT DATA REGISTER

Data coming from an external device is placed into this register by the device, and subsequently transferred into the accumulator for use inside the computer. Data going to an external device is transferred from the accumulator to the input/output data register, from which it is removed by the device. The input/output data register serves the purpose of temporary storage and speed-matching. Direct communication between the accumulator and the external device is inconvenient because the two are seldom ready at the same instant, and time would be wasted by one waiting for the other. In general, the computer will be much faster than the external device. The input/output data register serves somewhat like an RFD mailbox, with which we can send or receive a letter without standing and waiting for the postman to come by.

INPUT/OUTPUT ADDRESS REGISTER

Several external devices are usually connected to the computer, and the input/output address register designates the one device that is to send or receive data to or from the input/output data register at any time.

COMPUTER PROGRAMS AND INSTRUCTIONS

Before discussing the computer programs and the instructions that comprise them, three additional definitions of terms are in order.

- INSTRUCTION REPERTOIRE

A computer is capable of performing only those operations that have been built into it. These operations are defined by the instruction repertoire, which is a list of the instructions that can be used to make up a program for the computer.

- INSTRUCTION

We may define an instruction as one step in a program, but now let us amplify the makeup and function of an instruction. It consists of three basic parts:

Operation Code ("Op Code"): The portion of an instruction that specifies the operation to be performed (add, transfer to input/output data register, store into memory, and others).

Modifier: An appendage to the code that further amplifies it, modifies the operation. For example, if the result of an "add" operation is normally placed into the accumulator, the modifier may be used to alter the operation so the result is placed into memory.

Address: The memory location that contains the word that will enter in the operation specified by the Op Code.

It is important to realize that instructions (and, for that matter, most data as well) for a computer are absolutely content-position-sensitive: for example, if we have a seven-character instruction, the first three will always comprise the Op Code, the fourth will always be the modifier, and the last three will always be the address. Any other order will make no sense to the computer.

- PERIPHERAL EQUIPMENT

That which is referred to above as an "external devices." Also referred to as I/O Devices.

Hypothetical Computer

To illustrate the workings of a computer and its program, we shall postulate an instruction repertoire. This instruction repertoire, by the definition of the instructions, will define the operation of the control unit and the arithmetic unit. Since the control unit causes the next instruction in the program to be executed immediately upon completion of the current instruction, the operation of the program register and instruction register are implicit in the computer, thus for programming purposes we can ignore them. We shall, therefore, work with the modified computer organization of Figure 31, which shows only the parts of the computer that are of interest to the programmer. Note that there are several other changes from Figure 30 to Figure 31.

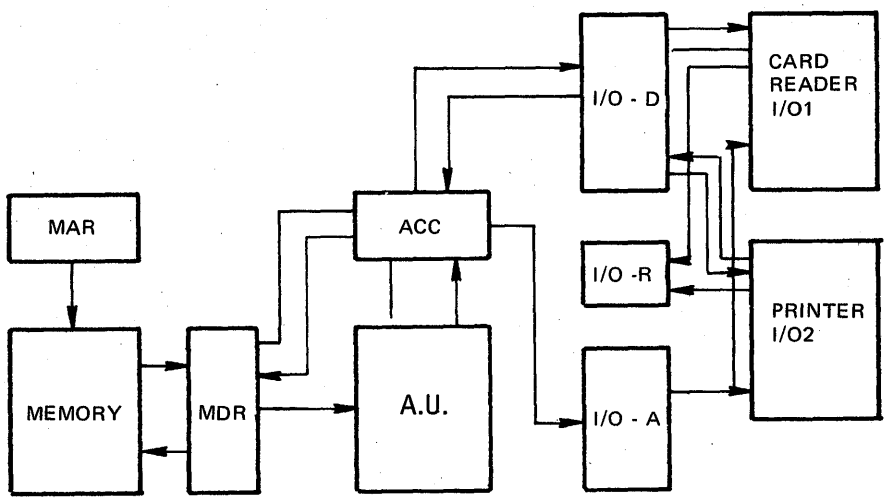


Figure 31. Simplified Functional Block Diagram of Hypothetical Computer System

For brevity, the accumulator, arithmetic unit, memory address register, memory data register, input/output data register, and input/output address register have been designated, in the text that follows, by the abbreviations ACC, A.U., MAR, MDR, I/O-D, and I/O-A, respectively.

The two units of peripheral equipment necessary to perform an electric-company billing problem, described later have been added to Figure 30 and designated "I/O1" and "I/O2." A unit designated I/OR (I/O Ready) has been added. I/OR is the means by which the I/O device can inform the computer that it is ready, or it has placed a word into I/O-D, or it has taken a word from I/O-D or some other message, the meaning of which is well known to the computer at the time it occurs. I/OR is merely a latch, or signal, like the nod of one's head: the interpretation of it is up to the computer.

For our simple instruction repertoire, let us ignore the modifier character, and work with simple six-character instructions. All instructions will have the format in which the first three characters are the Op Code, and the last three characters are the associated address, if needed. The memory contains one thousand locations, with addresses designated 000 through 999. First, the arithmetic instructions are:

ADDXXX: add the contents of memory location XXX to the contents of ACC, and place the result in ACC. Symbolically: ACC + XXX → ACC.

SUBXXX: subtract the contents of memory location XXX from the contents of ACC, and place the result in ACC. Symbolically: ACC - XXX → ACC.

MLTXXX: multiply the contents of XXX by the contents of ACC, and place the result in ACC: (ACC) (XXX) → ACC.

Next, the instructions used for data transfer and manipulation are:

LODXXX: fetch the contents of XXX to ACC: XXX → ACC.

STOXXX: store the contents of ACC in XXX: ACC → XXX.

TCA: transfer the contents of ACC to I/O-A: ACC → I/O-A.

TCD: transfer the contents of ACC to I/O-D: ACC → I/O-D.

TDC: transfer the contents of I/O-D to ACC when I/O-D is set.

(Note that LODXXX and STOXXX involve the use of MAR and MDR, but they are not mentioned in the instruction because they are implicit operators in the function of storing and fetching into or from memory.)

The last three instructions are used in the program to make decisions. The simple statement that a program proceeds through a fixed sequence of instructions is not strictly true, in that the program is able to make the decision to transfer to another part of itself rather than execute the next sequential instruction, on the basis of results obtained while performing computation. The instructions used to perform this kind of operation are called transfer or jump or branch instructions; their use will be made more clear in the example below. In our instruction repertoire, the branch instructions are:

BIOXXX: take the next instruction from location XXX if I/OR is set. If I/OR is not set, take the next sequential instruction.

BCNXXX: take the next instruction from location XXX if ACC is zero or negative. If ACC is positive, take the next sequential instruction.

BBBXXX: ,take the next instruction from XXX.

Since the instructions, as well as the data, are stored in the memory, these instructions address memory in the same sense as the data instructions. Returning for a moment to Figure 31, we can see that if the conditions specified by a branch instruction are satisfied, the address of the next instruction is inserted into MAR from the instruction register (where the whole branch instruction is stored while being executed), instead of from the program register. XXX is also inserted into the program register when a transfer is executed,

because the sequential selection of instructions will proceed from XXX, not the location of the transfer instruction.

We have therefore specified a computer with eleven instructions and 1000 words of memory to perform the task at hand: this is the hardware with which the programmer must work. The programmer must allocate the memory locations to particular tasks, and write a program using these instructions. To complete the creation of a computer and program system to perform the desired function, we shall use the electric-company billing operation to illustrate the process.

Electric-Bill-Calculation Program

One programmer task is to allocate the available memory locations to the various functions for which the memory must be used. In this problem, these are:

Storage of constants needed
Storage of the program
Storage of the input data
Storage of intermediate results (a "scratchpad" area)
Storage of output data

Figure 32 shows a reasonable memory allocation for this problem; such a diagram is called a *memory map*.

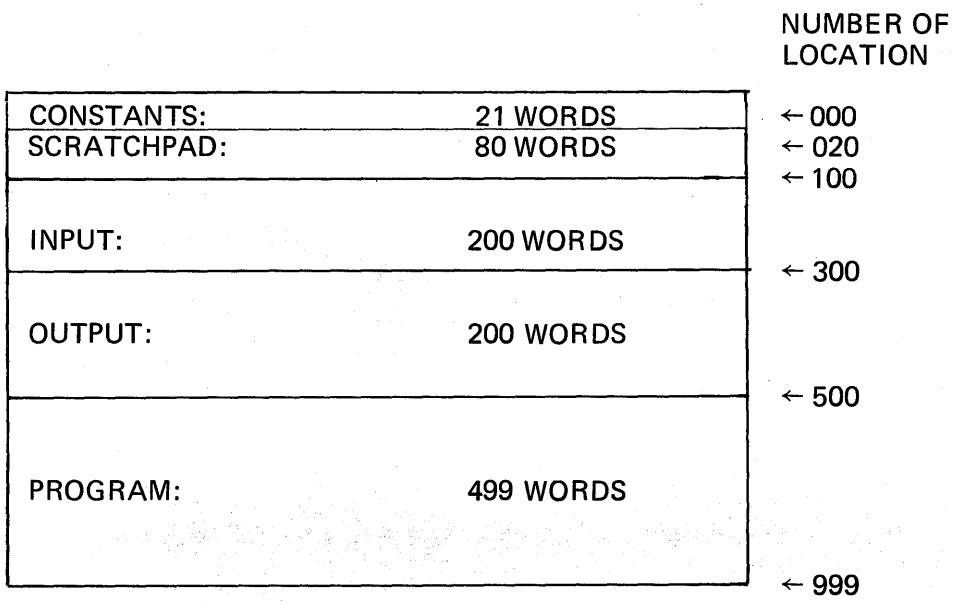


Figure 32. Memory Map for Electric Billing Program

The creating of a program consists fundamentally of three steps:

1. Specification of the operations that will be performed and the sequence in which they will be performed. This specification takes the form of a program flowchart that defines the general logic of the program.

2. Specification of the individual elements (routines) of the program. This is the same process as above, but in more detail and for each area of the program. This specification takes the form of a detailed flowchart for each routine.

3. Writing the instructions. If the flowcharts are properly done, the instruction sequence for each routine may be created easily and directly from the detailed flowchart, and the individual routines may be joined to form the program using the program flowchart.

For our electric company billing problem, the program flowchart is shown in Figure 33. The program basically performs three routines:

1. Read the portion of a card that contains the number of kilowatt-hours used by the customer during the past two months. For the sake of simplicity, we will assume that the remaining information on the card (customer name and address, customer identifying number) is transferred directly from the card reader to the printer that is printing the invoice. Therefore, the computer is required only to take the kilowatt-hours (KWH) used, compute the amount due, and provide it to the printer.

2. Compute the amount due using the following schedule of rates:

   2.00 dollars for the first 25 KWH.
   4.01 cents/KWH for the next 125 KWH.
   2.50 cents/KWH for the remainder.

3. Print the amount due, and go on to the next card.

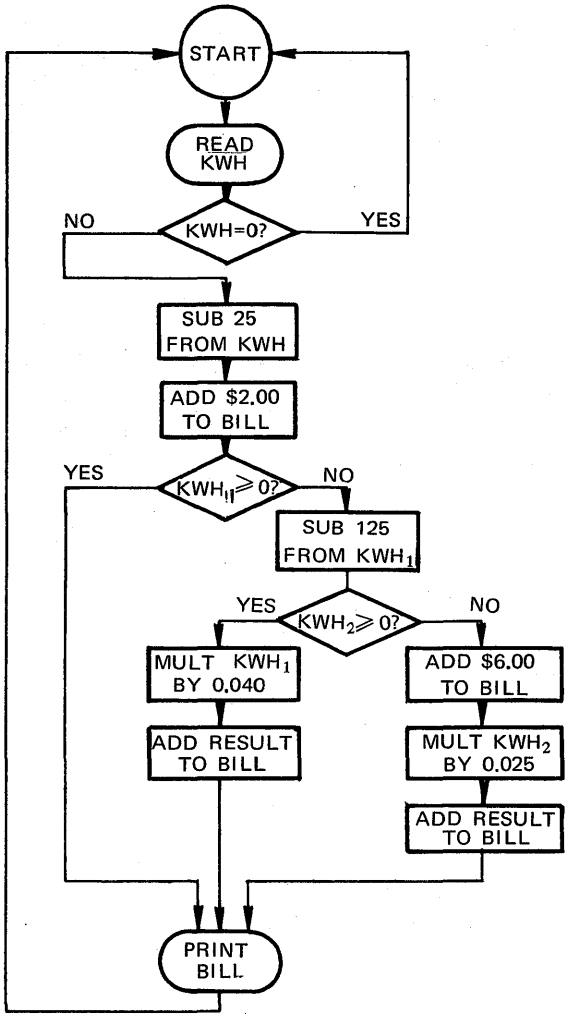


Figure 33 Program Flowchart for Electric Billing Problem

The detailed flowchart for the first routine, "read KWH," is shown in Figure 34. It comprises the following steps:

1. Fetch the command "read-a-card" from memory and load into ACC.

2. Wait for the I/O-D to indicate that the card reader has placed the A word into the I/O-D register.

3. Transfer the contents of the I/O-D to the ACC.

4. Transfer the contents of the ACC ot the first input-area location in memory.

Once the detailed flowcharts exists, the creation of a sequence of instructions to perform the routine is straightforward.

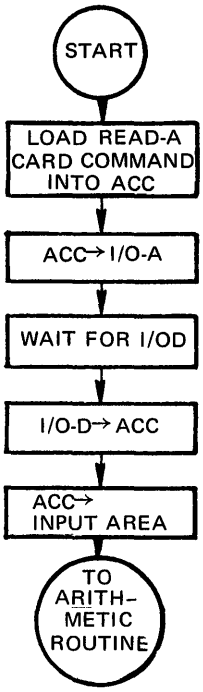


Figure 34. Read KWH Routine Flowchart

The computer contains 1000 words of memory. Each word consists of four characters. This allows us to store one instruction per word in the memory. The Op Code is stored in the first character of memory (suitably coded so, for example, a first-character "A" is interpreted as "add," and "S" as "subtract," and so forth). The address (which will be three digits) is the following three characters. ACC, MDR, and I/O-D will also have a one-word capacity. Thus, full memory words may be operated upon and manipulated.

The list of instructions to implement the detailed flowchart comprises both the instructions and their assigned locations in memory. The program and the data are stored in the memory. In the instruction sequence below, the memory address where each instruction is stored is shown in the left-hand column, the instruction sequence is listed in the center column, and the right-hand column shows the contents of ACC after the instruction on that line has been executed. This makes following the program sequence easier.

Address	Op Code	and Address	ACC Content
500	LOD	000	"read-a-card"
501	TCA	NNNN	
502	TDC	NNN	word from card reader

We have chosen to store the command that will be interpreted by the card reader as "read-a-card" in the first location of the constant area (000). Therefore, the first two instructions retrieve that constant to ACC and transfer it to I/O-A. (The "NNN" in the address portion of an instruction indicates that no address is associated with that instruction—because it does not deal with memory—and filler or dummy characters occupy those character positions in memory.) The third instruction accepts the word from I/O-D when the card reader has placed it there and set I/OR; we end the input (the number of KWH used) in ACC.

Address	Op Code and	Address	ACC Content
503	BCN	500	KWH
504	SUB	002	KWH-25
505	BCN	513	KWH-25
506	STO	020	KWH-25
507	SUB	003	KWH-25-125
508	BCN	515	KWH-25-125
509	MLT	005	(KWH-25-125) (0.025)
510	ADD	001	(KWH-150) (0.025) + 2.00
511	ADD	006	(KWH-150) (0.025) + 8.00
512	BBB	518	(KWH-150) (0.025) + 8.00
513	LOD	001	2.00
514	BBB	518	2.00
515	LOD	020	KWH-25
516	MLT	004	(KWH-25) (0.040)
517	ADD	001	(KWH-25) (0.040) + 2.00

We have, therefore, created the program for the first part of the program flowchart, which is represented by the detailed flowchart of Figure 35. Figure 35 shows the detailed flowchart for the arithmetic portion of the program; the listing of instructions is below. Note in Figure 36 that we have assigned the necessary constants to be stored in specific locations of the constant area.

From the electric rate schedule, note that the filling operation falls into one of four categories:

1. No KWH is used; therefore no bill. The test in step 503 will, if ACC is less than or equal to 0, transfer the program back to location 500, which will cause the next card to be read, thus giving no billing output for this card.

2. Less than 25 KWH used: the minimum bill of 2.00 is created. Step 503 shows ACC less than or equal to 0. Therefore the program will go to step 504, subtract 25 from the input KWH, and test in step 505 to see if ACC is less than or equal to 0. If there were 25 KWH or less, ACC is less than or equal to zero, and BCN 513 transfers the program to location 513 where the required 2.00 is loaded into ACC. Step 514 then transfers the program to location 518, where the "output routine" starts.

3. Between 26 and 150 KWH used: the bill is 2.00 plus 0.40 times the KWH over 25. Step 505 shows the ACC is still positive after 25 has been subtracted from KWH; step 506 stores the number into the scratchpad, location 020, for further use if needed. Step 507 subtracts 125 from KWH-25, and 508 discovers that this resulted in a negative number (or zero). Therefore, the original number of KWH is 150 or

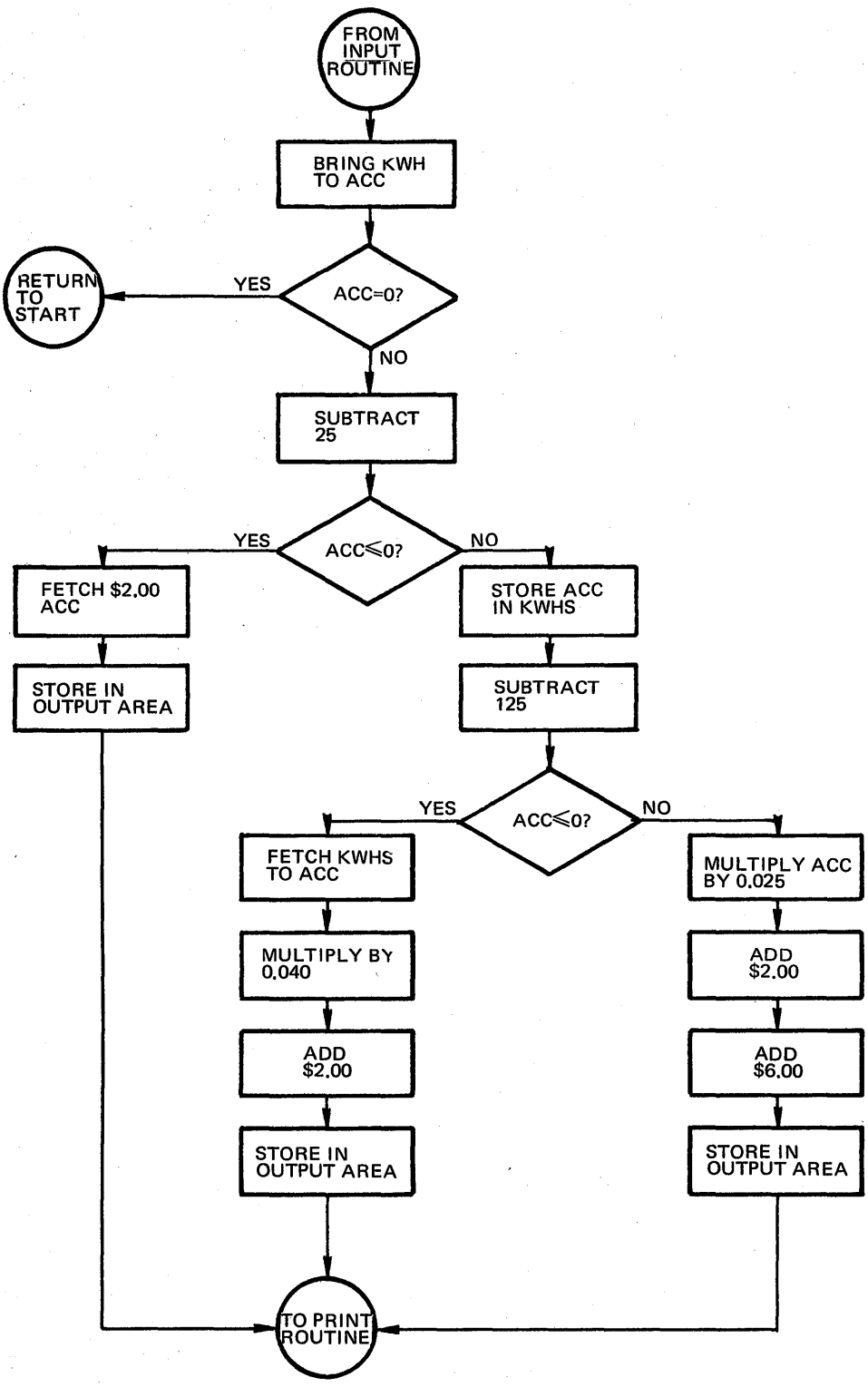


Figure 35. Arithmetic Routine Detailed Flowchart

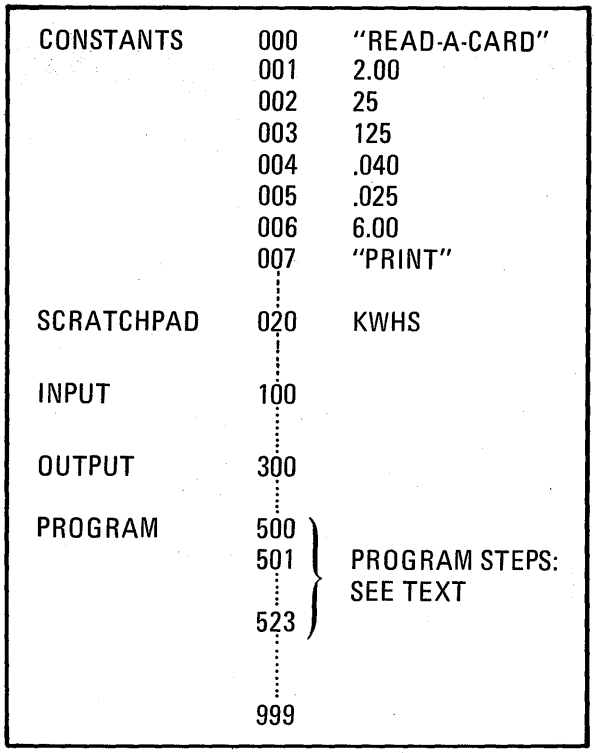


Figure 36. Memory Assignments for Electric Billing Program

less. The program is, therefore, transferred to step 515. Where the number of KWH in excess of 25 is retrieved from the scratchpad, multiplied by 0.040 in step 516, and 2.00 added to the result in step 517. The correct result is now ready for output in step 518.

4. Over 150 KWH used: the bill is 2.00 for the first 25 KWH, plus 6.00 for the next 125 KWH, plus 0.025 for each KWH over 150. The tests in step 503, 505, and 508 all find ACC is less than or equal to 0, and at the end of step 508, ACC contains the number of KWH over 150. This is multiplied by 0.025 in step 509, 2.00 is added in step 510, 6.00 is added in step 511, and the program is transferred to the output routine by step 512.

The output routine is shown in detailed flow diagram form in Figure 37, and the program is listed below.

Address	Op Code and Address		ACC Content
518	TCD	NNN	Bill
519	LOD	007	"print" command
520	TCA	NNN	
522	JIO	500	
523	JJJ	522	

At the end of the arithmetic routine, in all cases, the amount of the bill is in ACC. This is transferred into I/O-D, from which the printer will obtain it for printing, in step 518. The "print" command is obtained from the constant storage in step 518, and is transferred to I/O-A in step 519. The printer recognizes this command, procures the output data from I/O-D, and sets I/OR to signify it has accomplished this mission. Until this is accomplished, the program will shuttle ("loop") between steps 522 and 523. Step 522 transfers the program back to start (read-a-new-card) after I/OR has been set, but until I/OR is set, step 522 leads to step 523, which transfers the program back to step 522.

Additional Programming Definitions

Once the basic operations are understood as above described, the following definitions relate the example given to the real-life operations of the normal computer process.

Bulk Input-Output—although an input and an output area were assigned in the memory in the above example and referred to in the flowchart, no use was made of them. In a normal operation, a large amount of input data (instead of the one word used above) would be brought into the computer, initially (one word-at-a-time), and stored in the input area. One word-at-a-time would then be taken from the input area, processed, and the results stored in the output area. At the end of the process, then, a group of input words would have been processed, creating a group of output words, and this entire group would then be transferred (a word-at-a-time) to the output device.

Arithmetic Operations—we have simplified the A.U. in the above example, to a "black box" that magically takes in two numbers and performs a specified operation on them. The design of an A.U. is an all-pervading consideration, involving the machine code, provision of instructions for decimal-point and sign manipulation, and other factors.

Program Modification—Since the program is stored in the memory in the same way as the data which is being operated upon, the program can be made dynamic; that is, instructions can be altered in the course of executing the program. An example would be the loading of a number of input data words into the input area starting at location 101. The first word would be put away by an STD 101 instruction; the constant 1 would then be added to 101, and the resulting 102 would replace the 101 in the STO instruction. This incrementing would continue until all the data were stored.

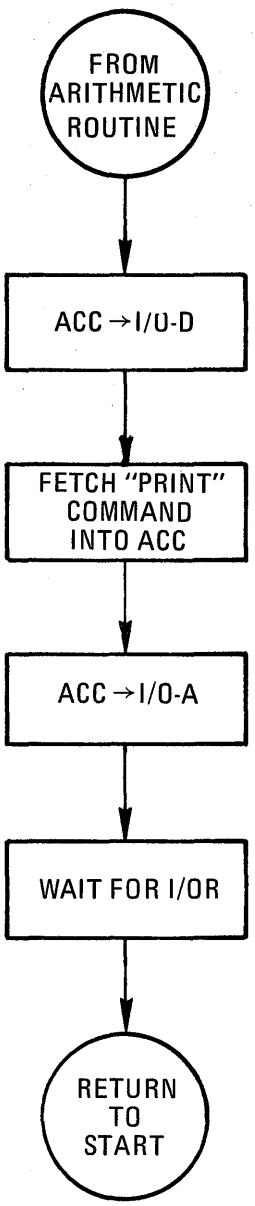


Figure 37. Output Routine Detailed Flowchart

Algorithm—A procedure for solving a particular problem: for example, the detailed flowchart for the arithmetic routine, above. An algorithm provides a basis upon which a routine may be written.

Types of Comptuer Programs

A computer program written directly in the language of the repertoire of instructions for a particular computer is called a machine-level program. It lists each instruction in the order-of-execution.

A machine-level program that uses op-code symbols and octal-number addresses, and the like must be machine-coded. The result is a listing of binary numbers, each number corresponding to a program instruction that the machine uses. As stated before, digital computers only work with binary numbers. Thus, with the actual machine-code listing of binary numbers in the proper sequence, we have a program that can be loaded into the computer memory. The processing circuits of the computer may then be used by the program to perform the appropriate operations.

At one time, the generation of machine-code listings was a laborious job done by people. However, this work may be done by the computer itself, and, to do this, a computer program is needed. There are various programs that do this translation, but the written program to be translated must be written according to an appropriate programming language. These programming languages are classified at different levels according to how much of the work they do. Thus, the translating program that works with a people-type language is a higher-level program than one that works with a machine-type language. Two types of programs are assembly and compiling programs. A compiler's language resembles the language of people more than does the assembler's language. Thus, a compiler is the higher-level type of program.

There are various types of assemblers and compilers, contingent on the job to be performed. These two types of programs are defined below.

ASSEMBLY PROGRAM

An assembler is a program that translates human-understandable notation to machine-understandable codes and keeps track of memory assignments. For example, the assembler would perform two operations in the above simple example.

1. Translate the Op Codes (ADD, SUB, and others) to a code that is recognized by the control unit.

2. Allow the use of symbolic addresses while programming. For example, we might designate the address of the beginning of the output routine by the term "out," and write instructions 512 and 514 as TTT "out," without bothering to determine, at that time, the exact location where "out" will be located when the address of the output routine is finally located. An equivalent between address 518 and "out" would be then specified, and the

assembler programs would go back through the users program and insert address 518 for each occurrence of "out."

COMPILER PROGRAM

A compiler is a program that allows the programmer to write the program in a completely human-oriented language that is not associated with any particular computer. The compiler translates the program written in compiler language to a sequence of instructions that causes that particular computer to perform the program. The compiler handles memory allocation and generates machine instructions and machine codes. Many machine instructions may be generated to execute one compiler instruction. For example, the compiler command "multiply" may result in a long string of "add" and "shift" instructions in the machine. Three examples of compiler languages are:

ALGOL—algorithmic-oriented language
COBOL—common business-oriented language
FORTRAN—formula-translating system.

UTILITY PROGRAMS

Most software systems include a variety of programs that perform miscellaneous day-to-day functions. Some of these programs offer only minor conveniences, while some are essential to daily operations. This assortment of routines is sometimes referred to, collectively, as utility or service routines.

The most common utility routines are generated sort/merge programs. A sort is a program that rearranges records stored on tape or other auxiliary storage into some desired order based on key fields in the record. A merge takes several previously ordered files and combines them into a single-ordered file. Usually, sort/merge programs are supplied to the user in very general form along with routines that assist in adapting the basic routines to satisfy particular requirements of the user. For this reason they are called "generated" programs.

Another form of utility routine is the linkage editor that combines several segments of coding into one routine. The use of such a system allows the repeated use of sub-routines in many different applications.

Debugging and program test routines are forms of utility programs that aid the programmer in finding programming errors and ensuring that programs operate under worst-case conditions. These routines are written in such a way that the programmer can "patch" his program as he discovers errors. This allows him to make several tests before he is forced to reassemble the program. In several such systems the programmer is allowed to make patches using the same mnemonics and symbolic names he used in the original coding. The most frequently used utility programs are the type that facilitate routine data handling. These include inquiries to auxiliary storage

devices, tape copying and editing, and media conversions such as card-to-tape, tape-to-card, and tape-to-print.

DIAGNOSTIC PROGRAMS

A diagnostic program's sole function is to "exercise" all parts of the computer, and such a program is so organized that a malfunction will create clues to its own location and the clues will be printed out by the diagnostic program.

INPUT/OUTPUT ROUTINES

Since almost every program uses the input/output facilities of the computer, many software systems have evolved with standardized subroutines that perform these tasks. This leads to standardized handling of data and error conditions and guarantees compatibility when more than one programmer works on a system. It also saves a considerable amount of programming time since each coder does not have to solve the problem individually. Standardized input/output routines vary from very simple read and write programs to elaborate systems of file control with object time device assignment. Some basic input/output routines may handle only magnetic tapes whereas more elaborate routines may process tapes, unit records, mass storage, disc files, display devices, and communications channels. Generally, such elaborate systems operate under control of an operating system as later discussed. The simpler input/output routines operate as part of the assembly processor and are usually confined to magnetic tape and unit record devices. By means of instructions such as OPEN, CLOSE, GET, and PUT the programmer is able to accomplish all his input/output operations on logical records. All translation from logical to physical terms and all error detection and correction is done by the standardized subroutines.

Operating Systems

To the user of a data-processing system, a significant measure of performance is the internal operating speed of a machine. This feature tells the user how fast a particular job can be processed once a run has started. This does not take into account, however, such variables as setup time and delays while awaiting operator instructions. As machine speeds have increased, such "overhead" operations have taken on an increasingly important segment of time. To counteract the degradation of system performance by manual intervention, operating systems have come into wider use. The general purpose of these systems is to automate, as much as possible, manual procedures and, at the same time, to provide a standard and helpful machine environment for the programmer, installation manager, and machine operator. This environment is intended to relieve programmers and managers of many details not essential to their functions.

Most operating systems have three major tasks: language translation, job control, and data control. Operating systems can accept properly identified programs in several languages and call on the proper translator to reduce the statements to some form of machine language. Generally, a translator working under control of an operating system will produce mahcine

code in standard format regardless of the source language. This module will be in relocatable form. That is, actual machine addresses will not be specified and it can be combined with other modules of code as specified by the programmer. This technique of putting together several pieces to make a program is highly efficient as it maximizes the use that can be made of various routines previously coded. Once programs are reduced to this standard format, the source language is immaterial. Thus, for example, a bond-yield program written in FORTRAN can be combined with a check-writing program written in COBOL to achieve a combination of the best features of each language. In some operating systems, libraries of user routines are maintained and complete facilities for updating are available. In other systems, program modules must be loaded into the system in the appropriate sequence in order to be properly linked. While this affects the efficiency of the linking process, it does not change the nature of the advantages to be gained.

In the early data-processing systems, it was the practice for the operator to load one program into the machine, set up any special equipment that was needed, have the computer execute the program, remove the results, and then get ready for the next run. It soon became apparent that while the operator was mounting tapes or getting a program deck from a card file, the computer was idle. Such idle time was frequently a large portion of available time. In order to minimize this loss of efficiency, monitor programs are now available which automatically load programs into the computer from auxiliary storage devices such as tapes or discs. In this way, a minimum of time is lost in job-to-job transition and, because of the decrease in human intervention, operator errors are minimized. With monitor systems, a large amount of operator responsibility is relieved, and any mistakes the programmer may make can be corrected and recorded in the form of cards so that subsequent use of the program will be successful.

The increase in flexibility afforded the programmer by an effective operating system is also a factor in operational efficiency. With a job-to-job transition automatically provided, the programmer can decide the next processing step based on object time conditions and cause the appropriate program to be called. Previously, this would have necessitated elaborate operating instructions and constant operator action. In a similar way, a program that is too lengthy to fit in the computer's memory can be constructed as a series of overlays that can be transferred in and out of the machine as needed.

An operating system provides a uniform language and procedure for communicating with the computer operator. Since all programs operate under its control, they use its facilities to inform the operator of program status. This means that whenever a program in the installation requires that a tape be mounted, for instance, the operator will get the same message. Without an operating system, each programmer might make up his own message, frequently causing operator confusion and delay.

This standard operating environment also helps the programmer. He does not have to decide when each new program overlay is needed. Also, operating standards are maintained with a minimum of programmer indoctrination and supervision.

Some of the more-advanced operating systems include multi-program features that provide significant time saving advantages. With this capability, two or more tasks can be processed

concurrently. Control may be assigned exclusively to one task or another depending upon priorities. The method of deciding when to shift control varies from one system to another and is often different from job to job within one system. For instance, one task may retain control until it requests an input/output operation. At that point, the other task will take over, while the first would be stalled. This method can be made more elaborate by adding the additional condition that regardless of other conditions, no task may retain control for more than some set period of time. When the period is over, control automatically passes to the next task. In order to maximize the usefulness of such a scheme, the programmer must be relieved of the responsibility of ensuring that tasks proceeding together do not interfere with each other. This responsibility must be assigned to the operating system.

Multi-programming increases the throughput capability of the computer by utilizing, more fully, the resources available. When one task cannot use the central processor because it is waiting for a card-read operation, for instance, the other task takes over. If one program does not use the printer, for example, it can be run with another program that does use the printer. This implies that idle time is minimized. The operating system assumes the responsibility of determining tasks sequences so that effective use can be made of all components of the system. In some small, research-oriented systems, an operating system is used not for the sake of efficiency, but to improve the method of communication between user and machine. Such systems allow a researcher to communicate with the computer in a simplified manner. The operating system then translates these shorthand instructions into program calls and parameter lists and causes the appropriate action to be taken. When the results are observed, the user decides on his next course of action and again instructs the operating system. This interaction continues until all paths have been tried. Thus, an effective operating system, coupled with a library of generalized subprograms, is almost essential to an interactive data-processing environment.

Some operating systems provide complete facilities for handling various types of data sets. Others provide only partial facilities and in still other schemes, data control facilities are built into language translators rather than into the operating system. A potential user should determine which of these schemes to use in his proposed installation and whether this is the best method for his purposes. The goal of complete data management facilities in an operating system is to eliminate the need for programmer attention to the characteristics of the data storage and data organization, and to allow him to concentrate on the logical properties of the data. In order to accomplish this, the designer of the software devices several standard forms of data organization and access methods. The programmer selects the method which is best suited to his application and uses pseudo-instructions which perform logical operations upon the data set. Some of the functions this type of software offers are:

1. Control of the physical movement of data between central storage and auxiliary storage or external data sets.

2. Detection and correction of errors associated with data transfer.

3. Data buffering, blocking and deblocking, overlapping data operations with computing, and data set label processing.

4. Dynamic scheduling of input/output devices.

5. Logical handling of data sets without reference to physical characteristics of the storage devices; thus, data sets stored on discs, cards, tape, or other media can be handled similarly and independently of device type.

Operating systems have become increasingly complex and increasingly versatile in order to help users take full advantage of the advanced facilities of their hardware. A good operating system can be a very effective means of multiplying programmer productivity, standardizing machine operations, increasing system throughput, and reducing the chances of undetected errors.

Programming Summary

The foregoing programming explanations afford some idea of a modern approach to computer programming. However, the details of program makeup and languages can be very complex—involving considerable background, training, and practice to produce an accomplished programmer. Nevertheless, most computer users need only know the types of programs available, their functions, and how to use them. For the time being, the following knowledge should have been obtained from these preceding computer-program descriptions:

1. What constitutes the actual instruction-word in binary format.

2. How program instructions are located in memory to constitute a program.

3. How the computer processor gets the next instruction.

4. What form the program is in (machine-code listing) for actual computer use.

5. That mnemonics and short-hand symbolic languages are used by programmers to actually write most programs.

6. A program's algorithm is shown on flowchart.

7. What the various types of programs are:

   Assemblers
   Compilers
   Utility Programs
   Diagnostic Programs
   Input/Output Routines

8. What an operating system does.

BOOLEAN LOGIC

Introduction

Boolean algebra was introduced by a nineteenth-centry mathematician, George Boole. Boolean algebra is based on a single-valued function that may be in one of two discrete possible status: true or false, high or low, positive or negative, +3 volts or 0 volts, 0 volts or -10 volts, and so forth. In all cases the state must be one or the other and never partly one and partly the other. Thus, a switch may be either open or closed, not half-open and half-closed. The latter might exist but Boolean algebra and the logic upon which it is based ignores any but the two possible states, open and closed.

Boolean logic may be annotated by Boolean algebra, logic diagrams, and various other techniques. A logic designer usually uses the two techniques mentioned, and with practice, a person may develop considerable expertise employing them. However, unless a person first understands the bases of Boolean logic and their relations with the system of notation for Boolean algebra and then practices and actually uses this notation, he will undoubtedly not be able to do much good with it. Therefore, other techniques are used, hereinafter, to present the basic concepts of Boolean logic: these should be well understood to be able to work with logic-level circuits. Also, some simple rules of Boolean algebra are given but for the immediate necessary task of following logic diagrams, they need not be mastered. Nevertheless, the computer technician should know these rules, and if he has the opportunity to regularly use them, then, he shall benefit considerably if he masters the techniques of Boolean algebra and its annotation.

The physical devices that are used to implement logic may be electronic (transistors, diodes, vacuum tubes), mechanical (switches), or pneumatic, hydraulic, or light-sensitive devices. To appreciate the logic implementation, however, a person may be completely ignorant of the working of these devices. True, for designing and detail troubleshooting and subsequent repair of these devices, their functional characteristics and uses must be more thoroughly understood. Thus, in the ensuing explanations the logic is first described in terms of simple switching arrays; next, the Boolean algebra equations and laws are presented; and finally physical devices are covered.

Boolean Logic Versus Binary Arithmetic

Figure 38 illustrates the two possible states of a switch; an open switch represents a 0 and a closed switch a 1.

Figure 38. Switch Analogy

By various arrangements of switches binary arithmetic can be implemented.

Although the conventional arithmetic symbols, plus (+), minus (-), times (x), and divide by (÷) still apply in binary arithmetic, the plus (+) and times (x) have different significance in binary (Boolean) logic. The "+" designates an OR function and the "x" an AND function. It is essential that these symbols be called OR (+) and AND (x) whenever they are used to signify logic.

OR Function

Figure 39 illustrates the OR function.

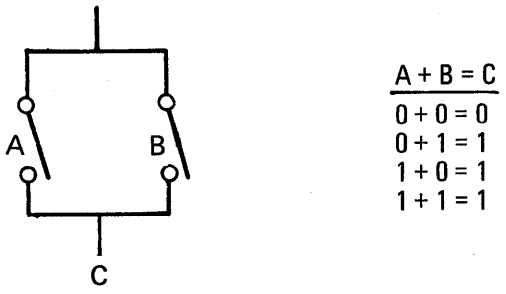


Figure 39. Switch Analogy of OR Function

The two switches represent two functions, A and B. Either function may have only one of two states, a 0 if open or a 1 if closed. Thus, the function C is effected by either A or B or both. The various combinations of this parallel arrangement of switches is shown in the Boolean-algebra equation in the illustration. The table of combinations is commonly called a "truth table."

Figure 40 shows three other, slightly more complicated, OR functions.

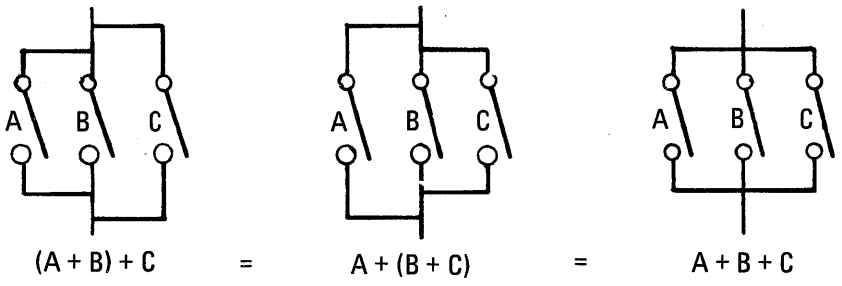


Figure 40. Compound OR Function

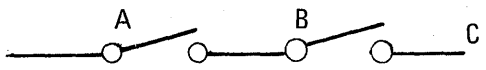
The OR gate may thus be extended to comprise any number of functions. Also, in Boolean algebra, the preceding switching configurations illustrates two laws: the commutative and associative laws.

$A + B = B + A$ – commutative law for Boolean OR function

$(A + B) + C = A + (B + C) = A + B + C$ – associative law for Boolean AND functions

AND Function

Two or more switches in series are analogous to the Boolean AND function. This is illustrated in Figure 41.

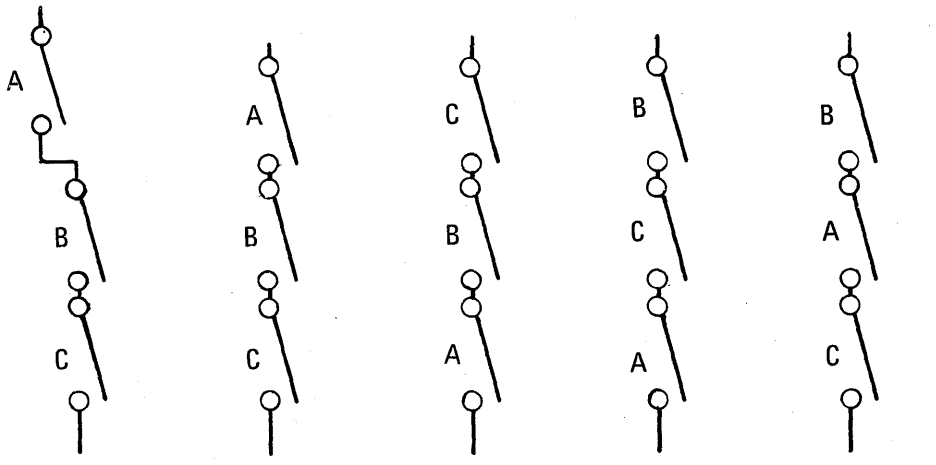


A x B = C
0 x 0 = 0
0 x 1 = 0
1 x 0 = 0
1 x 1 = 1

Figure 41. Switch Analogy of AND Function

The truth table for a two-input AND gate is logically the opposite of the OR function truth table.

Figure 42 illustrates the switch arrangement for the commutative and associative laws for the AND function.



A x B x C = CBA = BCA = BAC commutative law

(A x B)C = (A x C)B = (B x C)A = (B x A)C associative law

Figure 42. Switch Analogy of Commutative/Associative Laws for AND Function

As with the OR function, the AND function may be extended and combined with the OR function. This is simply illustrated in Figure 43 to represent several Boolean identities.

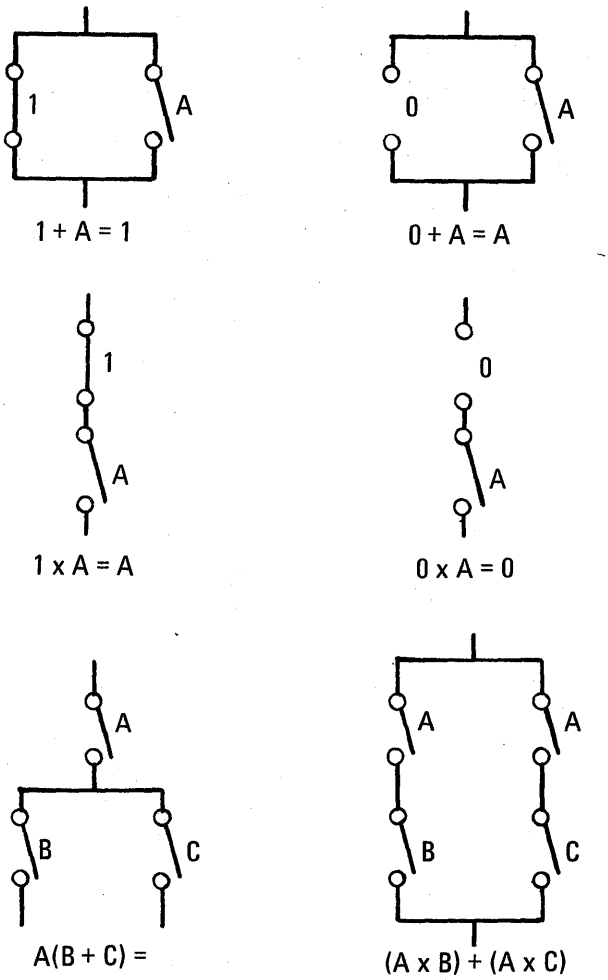


Figure 43. Identities of Simple AND, OR, and Compound Switch Arrangements

Complements

As previously discussed under Binary Arithmetic, a complement of a number in a binary system is the opposite state: 0 complemented gives a 1 and vice versa. Another way of stating this is:

0 = 1 or 1 = 0, "read *not* zero equals 1" or
"*not* 1 equals zero"

Thus for any function or group of functions, a bar used as above illustrated signifies negation, which is the same as the complement in binary arithmetic or Boolean logic.

Two laws that apply to Boolean logic are called De Morgan's laws. In Boolean algebra, these are as follows:

$$\overline{A \times B \times C \ldots\ldots N} = \overline{A} + \overline{B} + \overline{C} \ldots\ldots + \overline{N}$$

$$\overline{A + B + C + \ldots\ldots + N} = \overline{A} \times \overline{B} \times \overline{C} \ldots\ldots \times \overline{N}$$

The NOT function (complementing) is a principle Boolean logic operation that is performed by the logic inverter.

AND/OR Identities

The OR- and AND-function identities are summarized in Figure 44. For simplicity the times sign (x) has been omitted from AND function equations.

OR	AND
A + B + C = (A + B) + C = A + (B + C)	ABC = (AB)C = A(BC)
A + B = B + A	AB = BA
A + A = A	AA = A
A + 1 = 1	A1 = A
A + 0 = A	A0 = 0
	A(B + C) = AB + AC

Figure 44. OR- and AND-Function Identities

Boolean Identities Summary

A summary of Boolean identities are given in Figure 45.

LOGIC HARDWARE

The hardware circuits that implement the precedingly described AND, OR, and NOT functions vary considerably. Fortunately, all these variations need not be known, and even those that we should know about do not require an understanding of a detailed-circuit nature. Nevertheless, a functional level understanding of logic units and a few logic-support units is prerequisite to developing maintenance expertise. Therefore, the various logic and logic-support units, specific types of hardware, and their interrelations ensue.

Logic Systems

In a DC or level-logic system a bit may be only one of two voltage levels. If the more-positive voltage is the 1-level and the other is the 0-level, the system is said to employ DC positive logic. On the other hand, a DC negative-logic system designates the more-negative voltage state of the bit as the 1-level and the more-positive as the 0-level. It should be emphasized that the absolute values of the two voltages are of no significance in these definitions. In particular, the 0 state need not represent a zero voltage level (although in some systems it might). Figure 46 illustrates positive-logic and negative-logic voltage levels, respectively.

Fundamental laws

OR	AND	NOT
$A + 0 = A$	$A0 = 0$	$A + \bar{A} = 1$
$A + 1 = 1$	$A1 = A$	$A\bar{A} = 0$
$A + A = A$	$AA = A$	$\bar{\bar{A}} = A$
$A + \bar{A} = 1$	$A\bar{A} = 0$	

Associative laws

$$(A + B) + C = A + (B + C) \qquad (AB)C = A(BC)$$

Commutative laws

$$A + B = B + A \qquad AB = BA$$

Distributive law

$$A(B + C) = AB + AC$$

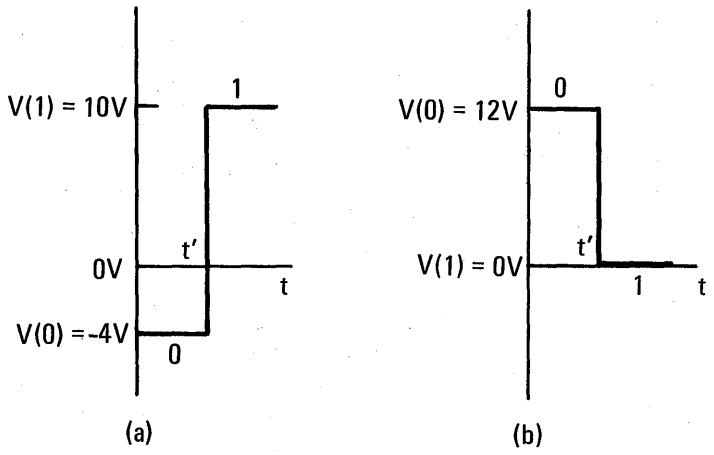
De Morgan's laws

$$\overline{AB\ldots} = \bar{A} + \bar{B} + \ldots \quad \overline{A + B + \ldots} = AB\ldots$$

Auxiliary identities

$$A + AB = A \qquad A + \bar{A}B = A + B$$
$$(A + B)(A + C) = A + BC$$

Figure 45. Summary of Boolean Identities

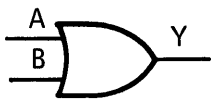


Illustrating the definitions of (a) positive and (b) negative logic. The numerical value of the voltage V(1) of the 1 state and of the voltage V(0) of the 0 state is arbitrary. A transistion from one state to the other occurs at t = t'.

Figure 46. Positive- Versus Negative-Logic Voltage Levels

OR Gate

Figure 47 shows a 2-input OR gate and its truth table. Additional inputs may be available on OR gates, but in order to obtain a 0 output, all inputs must be 0's. In all other cases, the outputs are 1's.



OR Gate

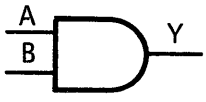
Input		Output
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Truth Table

Figure 47. OR Gate Symbol and Truth Table

AND Gate

Figure 48 shows a 2-input AND gate. Here also there may be additional inputs. Converse to the OR gate truth table, the inputs must be all 1's to have a 1 at the output. All other combinations of input give 0 outputs.



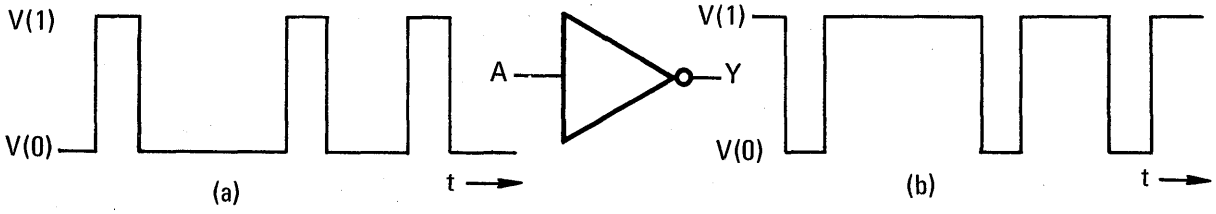
AND Gate

Input		Output
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Figure 48. AND Gate Symbol and Truth Table

NOT Circuit

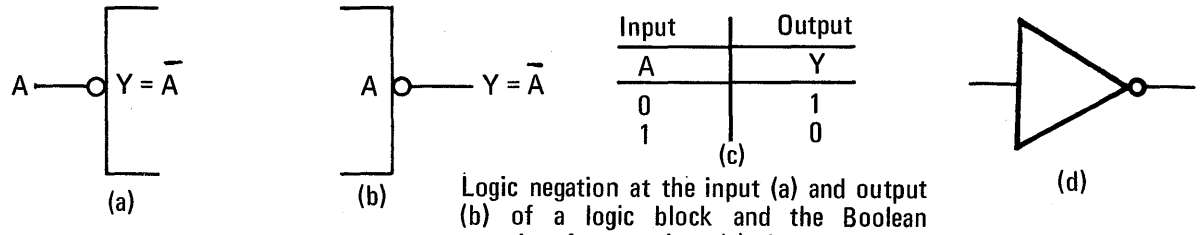
The NOT circuit has a single input and a single output and performs the operation of LOGIC NEGATION in accordance with the following definition: The output of a NOT circuit takes on the 1 state if, and only if, the input does not take on the 1 state. The symbol to indicate a LOGIC NEGATION is a small circle drawn at the point where a signal line joines a logic symbol. Negation at the input of a logic block is indicated in Figure 49 (a) and at the output in Figure 49 (b). The truth table and the Boolean expression for negation are given in Figure 49 (c) and the symbol for an inverter in Figure 49 (d). The equation is to be read "Y equals NOT A" or "Y is the complement of A." (Sometimes a prime (') is used instead of the bar (-) to indicate the NOT operation.



(a) The input A and (b) the output Y of a NOT circuit.

Figure 49. Logic Negation Symbol and Truth Table

A circuit that accomplishes a logic negation is called a NOT circuit, or, since it inverts the sense of the output with respect to the input, it is also known as an inverter. The output of an inverter is relatively more positive if and only if the input is relatively less positive. In a truly binary system only two levels V (0) and V (1) are recognized, and the output, as well as the input, of an inverter must operate between these two voltages. When the input is a V (0), the output must be at V (1), and vice versa. Ideally, then, a NOT circuit inverts a signal while preserving its shape and the binary levels between which the signal operates, as indicated in Figure 50.



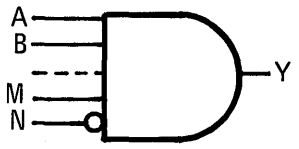
Logic negation at the input (a) and output (b) of a logic block and the Boolean equation for negation; (c) the truth table; (d) an inverter symbol.

Figure 50. Input Versus Output of NOT Circuit

Inhibit Gate

A NOT circuit preceding one terminal (N) of an AND gate acts as an INHIBITOR. This modified AND circuit implements the logical statement: If A = 1, B = 1, ..., M = 1, then Y = 1 provided that N = 0. However, if N = 1, then the coincidence of A, B, ..., M is inhibited, and Y = 0. Such a configuration is also called an anticoincidence circuit. The logical block symbol is drawn in Figure 51 together with its Boolean equation. The equation is to be read "Y equals A and B and . . . . and M and not N." The truth table for a three-input AND gate with one inhibitor terminal (C) is given in Figure 51.

(a) The logic block and Boolean expression for an AND with an inhibitor terminal N. (b) The truth table for $Y = AB\overline{C}$. The column on the left numbers the eight possible input combinations.



$Y = AB \ldots M\overline{N}$

(a)

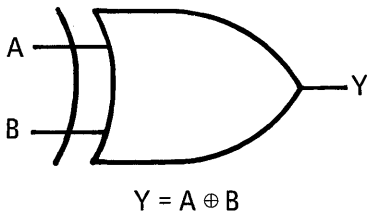
	Input			Output
	A	B	C	Y
1	0	0	0	0
2	0	1	0	0
3	1	0	0	0
4	1	1	0	1
5	0	0	1	0
6	0	1	1	0
7	1	0	1	0
8	1	1	1	0

(b)

Figure 51. Inhibitor Gate Symbol and Truth Table

Exclusive OR Gate

An exclusive OR gate (OE) obeys the definition: The output of a two-input EXCLUSIVE OR assumes the 1 state if one and only one input assumes the 1 state. The symbol for an exclusive OR gate is shown in Figure 52 (a) and its truth table in Figure 52 (b).



$Y = A \oplus B$

(a)

Input		Output
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

(b)

(a) The symbol for an EXCLUSIVE OR symbol and its Boolean expression; (b) the truth table.

Figure 52. Exclusive OR Gate Symbol and Truth Table

The above definition is equivalent to the statement: "If A = 1 or B = 1 but not simultaneously, then Y = 1." In Boolean notation

$$Y = (A + B)\ \overline{(AB)}$$

This function is implemented in logic diagram form in Figure 53 (a).

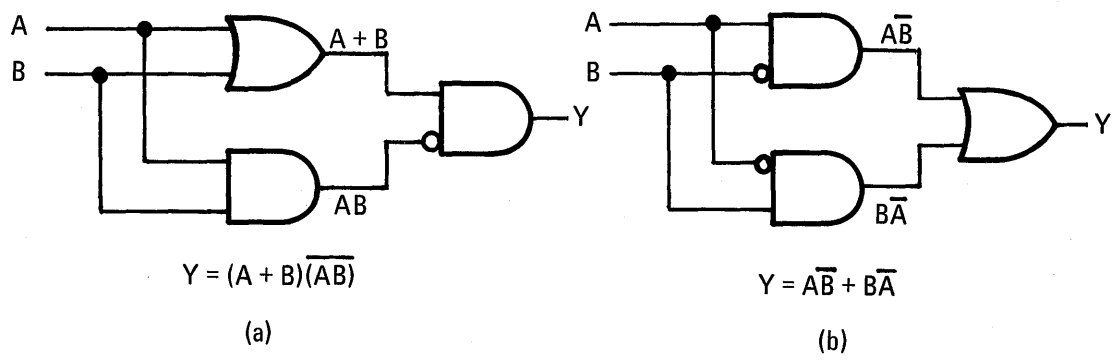


Figure 53. Two Logic Blocks for the EXCLUSIVE OR (OE) Gate

A second logical statement equivalent to the definition of the OE is the following: "If A = 1 and B = 0, or if B = 1 and A = 0, then Y = 1." The Boolean expression is

$$Y = A\bar{B} + \bar{B}A$$

The block diagram that satisfies this logic is indicated in Figure 53 (b).

An EXCLUSIVE OR is employed within the arithmetic section of a computer. Another application is as an inequality comparator, matching circuit, or detector because, as can be seen from the truth table, Y = 1 only if A ≠ B. This property is used to check for the inequality of two bits. If bit A is not identical with bit B, then an output is obtained. Equivalently, "If A and B are both 1 or if A and B are both 0, then no output is obtained, and Y = 0." This latter statement may be put into Boolean form as

$$Y = \overline{AB + \bar{A}\bar{B}}$$

This equation leads to a third implementation for the OE block, which is indicated by the logic diagram of Figure 54 (a). An equality detector gives an output Z = 1 if A and B are both 1 or if A and B are both 0; thus

$$Z = \bar{Y} = AB + \bar{A}\bar{B}$$

If the output Z is desired, the negation in Figure 54 (a) may be omitted or an additional inverter may be cascaded with the output of the OE.

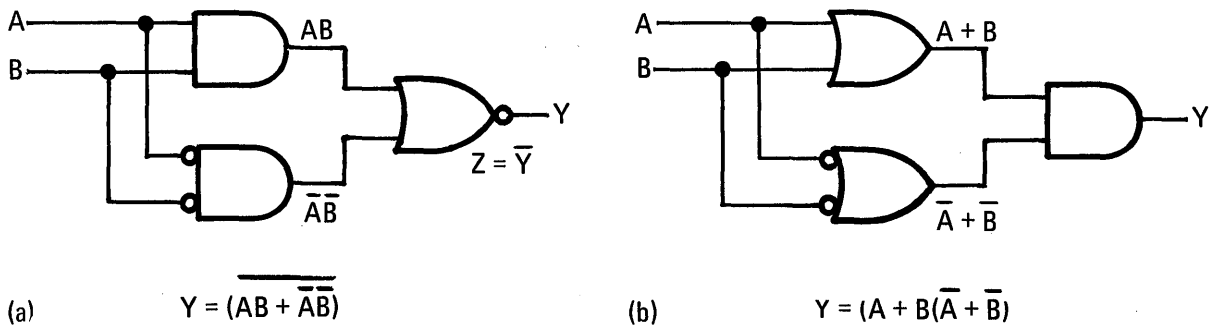


Figure 54. Two Additional Logic Diagrams for the EXECLUSIVE OR (OE) Gate.

A fourth possibility for the OE is

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

which may be verified from the definition or from the truth table. This logic is depicted in Figure 54 (b).

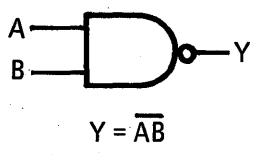
It has been demonstrated that there often are several ways to implement a logical circuit. In practice, one of these may be realized more advantageously than the others. Boolean algebra is sometimes employed for manipulating a logic equation to transform it into a form that is better from the point of view of implementation in hardware.

Positive Versus Negative Logic

If the output and all inputs of a logic gate are complemented, a 1 becomes a 0 and vice versa and positive logic is changed to negative logic. Thus, the same gate is either a negative AND or a positive OR contingent on how the binary levels are defined. It is logically irrelevant how the gate circuit is implemented. This concept may be proven by applying De Morgan's law, if one so desires.

NAND Gate

A negated AND is called a NOT-AND or a NAND gate. The logic symbol, Boolean expression, and truth table for the NAND are given in Figure 55. The NAND may be implemented by placing a transistor NOT circuit after a diode AND gate. Circuits involving diodes and transistors are called diode-transistor logic (DTL) gates. Figure 56 is an example of a DTL circuit.



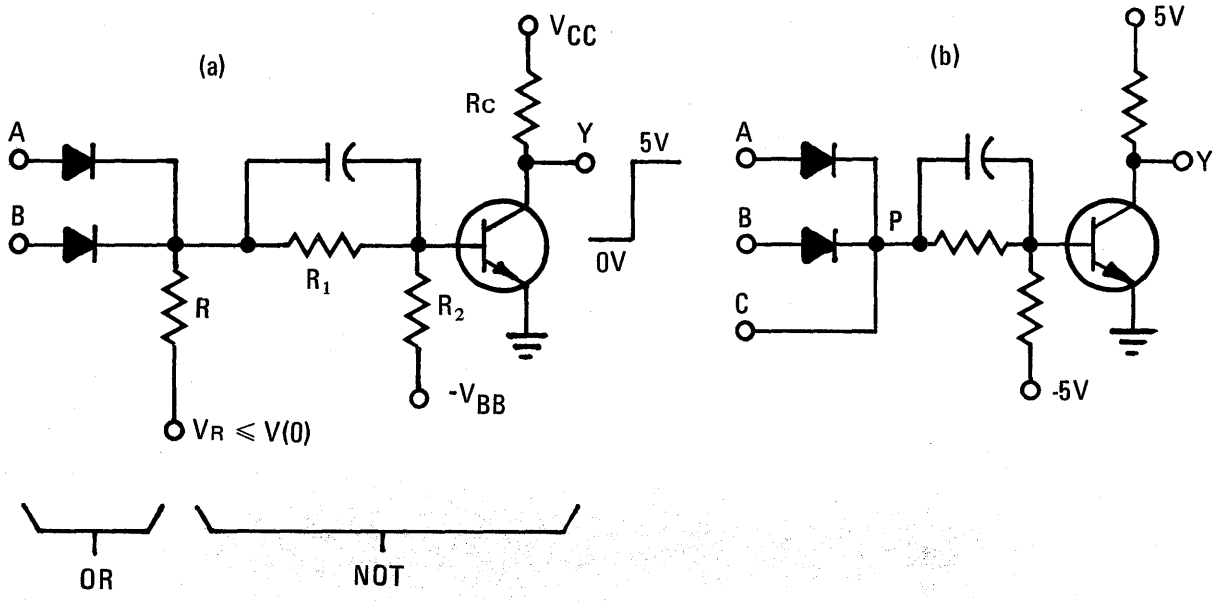
(a)

Input		Output
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

(b)

(a) The logic symbol and Boolean expression for a two-input NAND gate; (b) the truth table.

Figure 55. NAND Gate Symbol and Truth Table

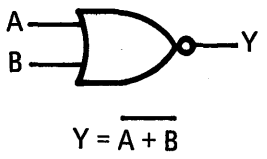


(a) A positive OR in cascade with a NOT to form a NOR gate. (b) A more practical form of positive NOR (or negative NAND) gate.

Figure 56. Example of DTL Circuit

NOR Gate

A negation following an OR is called a NOT-OR or a NOR gate. The logic symbol, Boolean expression, and truth table for the NOR are given in Figure 57. Figure 58 is an example or a DTL positive NAND gate.



(a)

Input		Output
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

(b)

Figure 57. NOR Gate Symbol and Truth Table

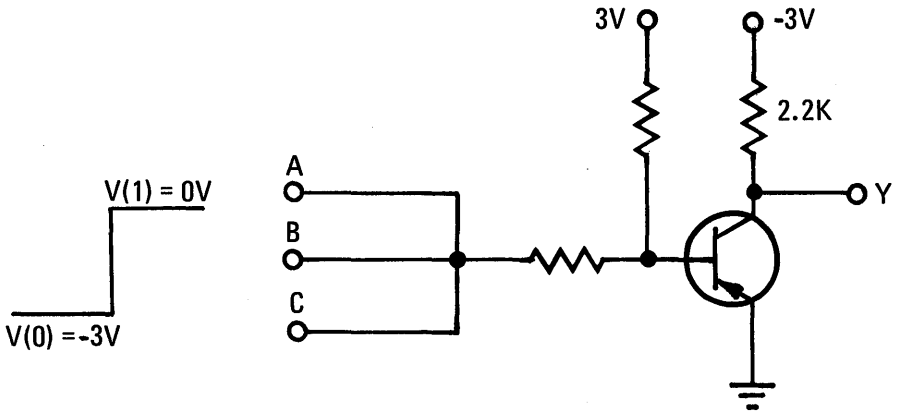


Figure 58. Example of DTL Positive NAND Gate

Logic Gate Uses and Interrelations

With the aid of De Morgan's laws it can be shown that, regardless of the hardware involved, a positive NAND is also a negative NOR, whereas a negative NAND may equally well be considered a positive NOR.

It is clear that a single input NAND is a NOT. Also, a NAND followed by a NOT is an AND. Thus, it is pointed out that all logic can be performed by using only the two connectives AND and NOT. Therefore, we now conclude that by repeated use of the NAND circuit alone, any logical function can be carried out. A similar argument leads equally well to the result that all logic can be performed by using only the NOR circuit.

We have seen that a particular gating circuit may perform one or another function, depending on whether positive or negative logic is used. Thus a positive AND gate is a negative OR gate, and vice versa. Suppose, however, our interest is in preserving the type of logic performed by the gate (that is, an AND gate is to remain an AND gate) but we wish to reverse the logic from positive to negative or the other way around. Then the following generalization provides a method for achieving this modification.

> THEOREM I: A circuit using positive (negative) logic can be converted into a configuration performing the same logic function but with negative (positive) logic, provided that all supply voltages are reversed in polarity, the input voltage levels are reversed in polarity, all diodes are reversed, and all transistors are changed from PNP to NPN and vice versa.
> Proof: If all voltages in a circuit are reversed, then the current in any branch is also reversed. If all diodes and transistors are reversed (PNP $\rightleftharpoons$ NPN), then a diode or transistor that was reverse-biased (conducting) in the original circuit remains reverse-biased (conducting) in the converted network. Hence, the two circuits perform the same logic. However, if the original level V (1) was more positive than V (0), then -V (1) is now more negative than -V (2) and positive logic has been changed to negative logic.

As an application of this theorem, suppose it were desired to build a positive NAND with the binary levels V (0) = -12V and V (1) = 0V. Thus, the circuit of a negative NAND can be converted into the configuration of a positive NAND by use of the theorem above stated.

The following theorem is useful in converting a logic circuit with one set of binary voltages to an equivalent circuit with another set of logical levels (in which, perhaps, neither is 0V).

> THEOREM II: If the same voltage is added to all supply voltages, to any leads that are grounded, and to both binary levels, then the logic function performed by the circuit remains unchanged.

Proof: The voltage difference between any two nodes is invariant under the above procedure, which is equivalent simply to shifting the zero reference of voltage. Hence, the same currents flow in the modified circuit under the identical logical conditions as in the original circuit. Clearly, the logic performed is unchanged.

Packaging of Logic Circuits

A digital computer uses a large number of switching circuits, but, as we have already emphasized, the variety of different types of gates is quite small. Hence, the fundamental circuits, which are used over and over again, are mounted on a number of plug-in units called logic cards. The advantage with respect to manufacturing, replacement, trouble-shooting and convenience, is apparent. Several manufacturers market such logic cards consisting of a glass epoxy printed-circuit board with the individual components mounted to the board through funnel eyelets. Also, a number of vendors have a line of micrologic gates manufactured by integrated-circuit techniques. These replace the conventional lumped-component circuits. Building a digital system consists principally in interconnecting these packages to perform the desired logic. A card or integrated-circiut (IC) package might consist of ten 2-input NAND gates, or eight NOT circuits (inverting amplifiers), or three 5-input OR gates, and so forth.

Often the logical design calls for an AND followed by an OR or vice versa. Such a configuration is known as two-level logic. One of the most useful logic packages for a large-scale computer is the AND-OR-INVERT (AOI) or AND-NOR configuration. The number of inputs, called the fan-in of each AND, is not critical and neither is the number of AND clusters that feed the OR gate. The number of outputs from a logic circuit is called the fan-out.

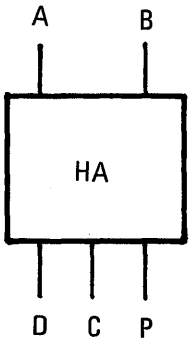
Adder

A digital computer must obviously contain circuits that perform arithmetic operations: addition, subtraction, multiplication, and division. The basic operations are addition and subtraction, since multiplication is essentially repeated addition, and division is essentially repeated subtraction. It is entirely possible to build a computer in which an adder-subtractor is the only arithmetic unit present. Multiplication, for example, may then be performed by programming; that is, the computer may be given instructions telling it how to use the adder repeatedly to find the product of two numbers.

Suppose we wish to sum two numbers in decimal arithmetic and obtain, say, the hundreds digit. We must add together not only the hundreds digit of each number but also a carry from the tens (if one exists). Similarly in binary arithmetic we must add not only the digit of like significance of the two numbers to be summed but also the carry bit (should one be present) of the next lower-significant digit. This operation may be carried out in two steps: first, add the two bits corresponding to the 2K digit, and then add the resultant to the carry from the 2K-1 bit. A two-input adder is called a half-adder because to complete an addition requires two such half-adders.

We shall first show how a half-adder-subtractor is constructed from the basic logic gates and then indicate how the full or complete adder-subtractor is assembled. A half-adder-subtractor has two inputs—A and B—representing the bits to be added, and three outputs—D (for the digit of the same significance as A and B represent), C (for the carry bit), and P (for the borrow bit). In a half-adder D and C are used, while in a half-subtractor D and P are used.

The symbol for a half-adder-subtractor is given in Figure 59 (a) and the truth table in Figure 59 (b). Note that the D column gives the sum of A and B as long as the sum can be represented by a single digit. When, however, the sum is larger than can be represented by a single digit, then D gives the digit in the result which is of the same significance as the individual digits being added. Thus, in the first three rows of the truth table D gives the sum of A and B directly. Since the decimal equation "1 plus 1 equals 2" is written in binary form as "01 plus 01 equals 10," then in the last row D = 0. Because a 1 must now be carried to the place of next higher significance, C = 1. Finally, where subtraction of B from A is contemplated, the P (borrow) column gives the digit which must be borrowed from the place of next higher significance when B is larger than A, as in the second row of Figure 59 (b).



(a)

Input		Output		
A	B	D	C	P
0	0	0	0	0
0	1	1	0	1
1	0	1	0	0
1	1	0	1	0

(b)

(a) The symbol for a half-adder-subtractor; (b) the truth table for the digit (D), carry (c), and borrow (P).

Figure 59. Half-Adder-Subtractor Logic Symbol and Truth Table

From Figure 59 (b) we see that D obeys the EXCLUSIVE-OR (OE) function, C follows the logic of an AND gate, and P obeys the logic "B and not A." Figure 60 shows a configuration which satisfies this half-adder-subtractor logic based upon the OE circuit. Any of the other implementations given for the OE may be used in the half-adder-subtractor. A half-adder thus may be constructed by using only NOR circuits.

A parallel binary adder is indicated in Figure 61. Each digit except the least-significant one (2^0) requires a complete adder consisting of two half-adders in cascade. The sum digit for the 2^0 bit is $S_0 = D_0$ of a half-adder because there is no carry to be added to A_0 plus B_0. The sum S_k ($k \neq 0$) of A_k plus B_k is made in two steps. First, the digit D_k is obtained from one half-adder, and then D_k is summed with the carry C_{k-1}, which may have resulted

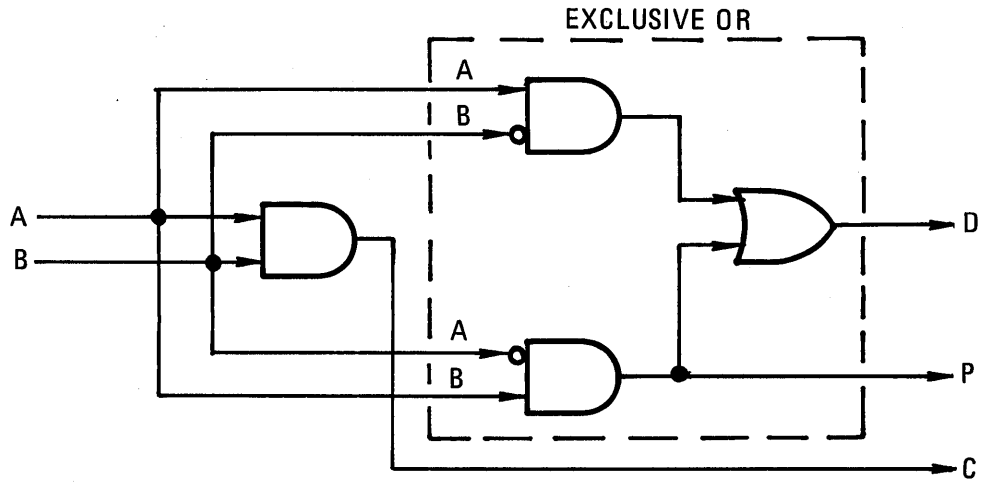


Figure 60. Half-Adder-Subtractor Logic Diagram

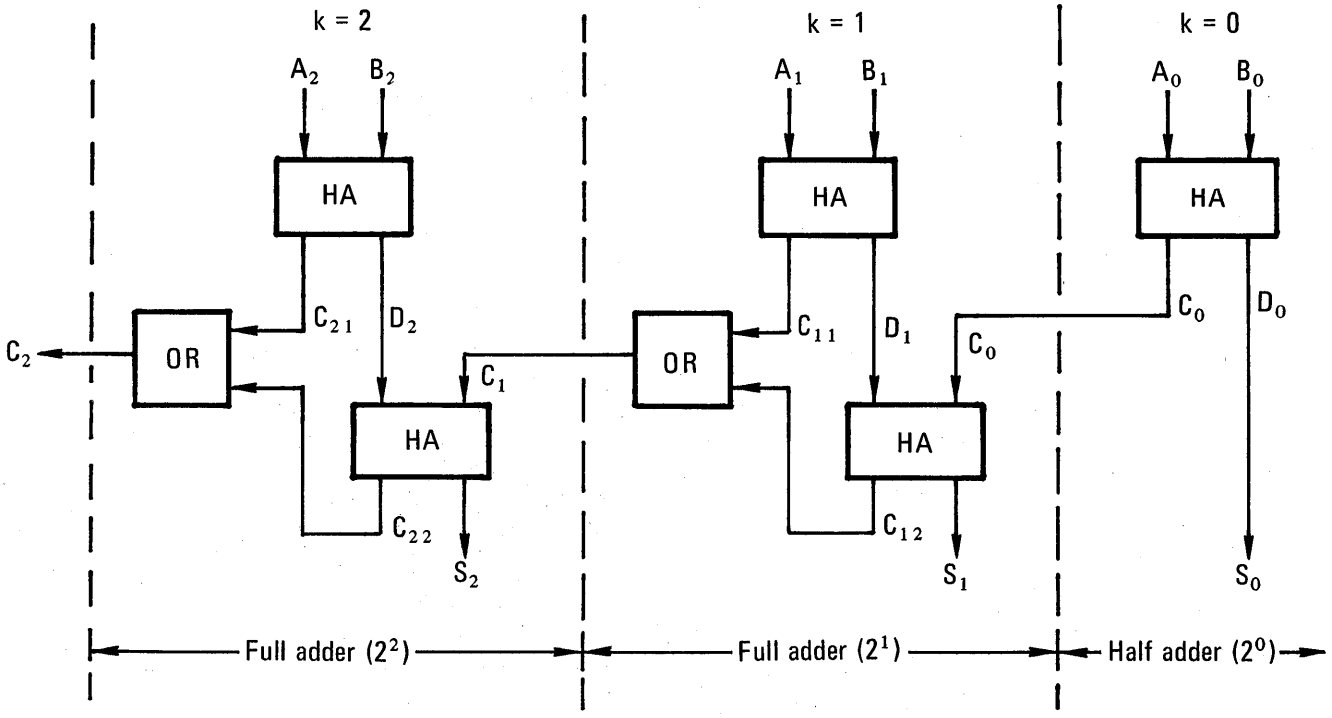


Figure 61. Parallel Binary Adder Consisting of Half-Adders

from the next lower place. As an example, consider K = 2. There, the carry bit C_1 may be the result of the direct sum of A_1 plus B_1 if each of these is 1. This first carry is called C_{11}. A second possibility is that $A_1 = 1$ and $B_1 = 0$ (or vice versa) so $D_1 = 1$, but that there is a carry C_0 from the next lower-significant bit. The sum of $D_1 = 1$ and $C_0 = 1$ gives rise to the carry bit designated C_{12}. It should be clear that C_{11} and C_{12} cannot both be 1, although they will both be 0 if $A_1 = 0$ and $B_1 = 0$. Since either C_{11} or C_{12} must be transmitted to the next stage, an OR gate is equally effective for subtraction, provided that the borrow bit P is used in place of the carry C.

It is possible to construct a complete adder without the use of half-adders. The circuit has three inputs: A, B, and the carry C. A truth table for such an adder is given in Figure 62. The output carry C' in a serial system is delayed one synchronizing interval T and then becomes the input carry C. From the truth table we can verify that the Boolean expressions for the sum S and the carry C' are given by

$$S = \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC$$

$$C' = \bar{A}BC > A\bar{B}C + AB\bar{C} + ABC$$

By algebraic manipulation these expresions may be transformed into a number of different forms. In particular it turns out that

$$S = (A + B + \bar{C}) + ABC$$

$$C' = AB + BC + CA$$

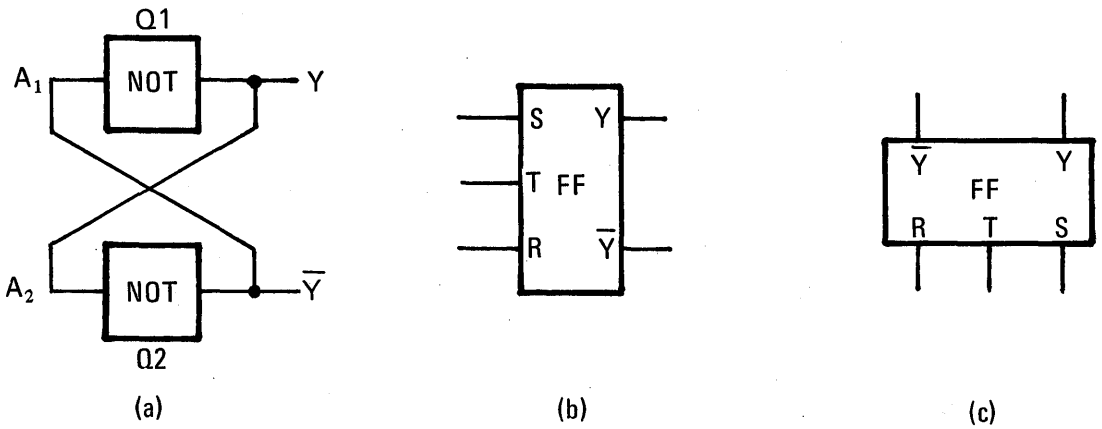
These expressions may also be verified from the truth table.

Input			Output	
A	B	C	S	C'
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 62. Truth Table for Three-Input Adder

FLIP-FLOP

In addition to the AND, OR, and NOT logic gates a fourth important basic circuit, called the FLIP-FLOP, is required in many digital systems. A FLIP-FLOP consists of two NOT circuit interconnected in the manner shown in Figure 63 (a). Each NOT could be, for example, a transistor INVERTER. For the present we are interested only in certain external



(a) A FLIP-FLOP assembled from two NOT circuits; (b or c) the logic symbol. An input to T effectively applies excitation to S and R simultaneously.

Figure 63. FLIP-Flop Configuration and Symbols

characteristics that are relevant in digital systems. The most important property of the FLIP-FLOP is that, on account of the interconnection, the circuit may persist indefinitely in a state in which one device (say Q1) is on while the other (Q2) is off. A second stable state of the FLIP-FLOP is one in which the roles of the two devices are interchanged so Q1 is off and Q2 is on. Since the FLIP-FLOP has two stable states, it may be used to store one bit of information. For these reasons the FLIP-FLOP is also called a BINARY.

An output, designated as Y in Figure 63 (a), may be taken from a collector. This output may take on two voltage levels, corresponding to either Y = 1 or Y = 0. If we designate the output at the other collector as Y, then the FLIP-FLOP has two stable states, one in which Y = 1 and Y = 0. The existence of these stable states is consistent with the interconnection shown in Figure 63 (a). For example, if the output Y of one NOT circuit is 1 then so also is the input A_2 to the second NOT circuit. The second inverter then has the state 0 at its output Y and at the input A_1 to the first gate. This result is consistent with our original assumption that the first NOT gate had a 1 at its output. It is readily verified that the situation in which both outputs are in the same state is not consistent with the interconnection.

A FLIP-FLOP is represented in block form as in Figure 63 (b), where three input terminals are indicated–S (set), R(reset), and T (trigger). An excitation of the *set* input causes the FLIP-FLOP to establish itself in the state Y = 1. If the binary is already in that state, the

excitation has no effect. A signal at the *reset* input causes the FLIP-FLOP to establish itself in the state Y = 0. If the binary is already in that state, the excitation has no effect. The waveform of the input signal (a pulse, a step), by other circuits through which the excitation is applied to the binary.

A triggering signal applied to the T input causes the FLIP-FLOP to change its state regardless of the existing state of the binary. Thus each successive excitation applied to T causes a transfer, and T is referred to as the toggle or complementing input. This type of excitation is called symmetrical triggering and is used in binary counters and in other applications. Unsymmetrical triggering through the S or R input is most useful in logic applications, as we demonstrate below.

FLIP-FLOPS in Registers

Suppose that it is required to carry out the addition of two numbers that are stored in the main computer memory. Now, ordinarily, it will not be possible to extract both numbers from the memory simultaneously. Since in the adders previously described both numbers are applied simultaneously, it will generally be required that at least one of the numbers be stored, temporarily, in a one-word memory device. Similarly, it may not be feasible to return the arithmetic unit output immediately to the main memory. In this case, a one-word memory or storage device, which is called a register, is needed.

A set of N flip-flop circuits may clearly be used to store an N-digit binary number, since we have but to set the states of the binaries at 0 or 1, depending on the value of the digit that the FLIP-FLOP is to represent. The binary number may appear in serial form as a train of pulses and one method for inserting the number into the register is as shown in Figure 64. The input pulse train is applied to a delay line, which is tapped at time-delay intervals TD equal to the basic pulse separation time (a one-bit delay T). Hence, at the moment the last pulse (2^3) of the train appears at the input of the delay line, the earlier pulses will appear at the delay-line taps. If, at this moment, the register line is pulsed, then, the AND circuits will transmit to each binary the pulse (or lack of pulse) at the corresponding delay-time taps. The output of each AND circuit is coupled to the set input of a FLIP-FLOP so the AND circuit pulse (if one is present) will leave the corresponding binary in state 1. Thus, the 2^3 bit is registered in FF3, the 2^2 bit in FF2, and so forth. The register may be cleared by a pulse on the reset line. This pulse will cause each binary to remain in, or return to state 0. The circuit of Figure 64 is a serial-to-parallel converter, because each bit of information in a pulse train is now available in a separate FLIP-FLOP. A temporal code (a time arrangement of bits) has been changed to a spatial code (information stored in a static memory).

Flip-Flop Memory Elements

One of the reasons systems design seems complicated is the fact that so many operations are functions of time, and logical equations involving time are very complicated. Thankfully, most time equations can be avoided if the designer expresses the problem in terms of a flowchart, thus reducing the design to a series of relatively simple operations—and relatively

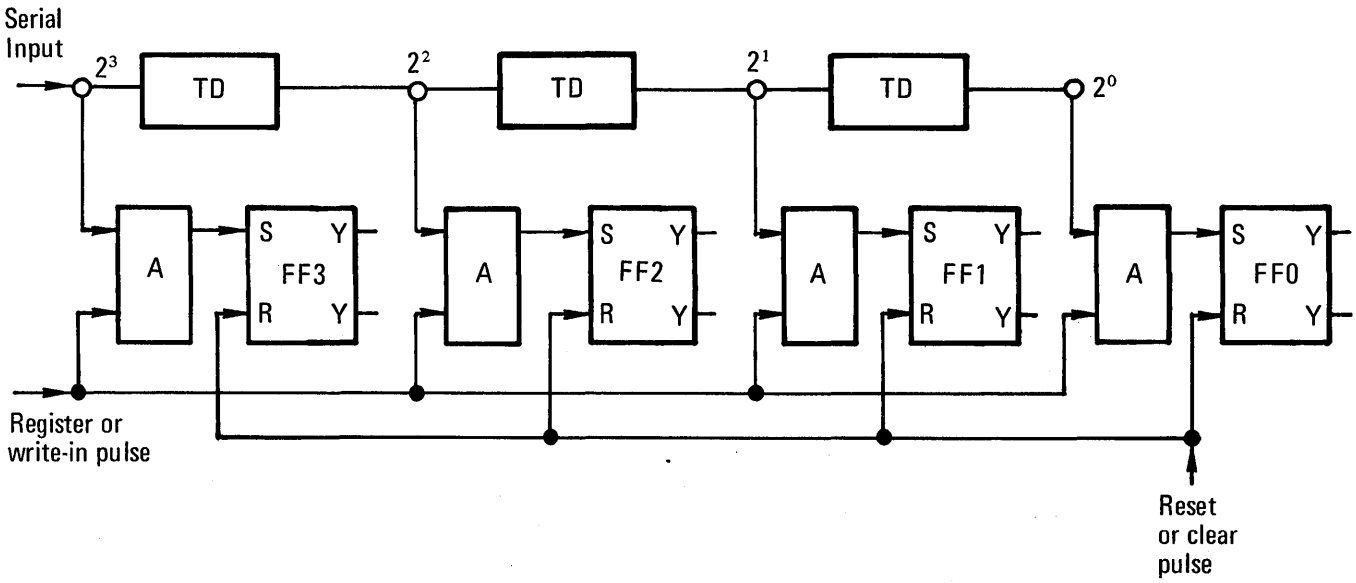


Figure 64. Four-Bit Register Used for Serial-to-Parallel Conversion

simple equations. The device is then guided from step-to-step by flip-flop circuits, which keep track of the present function and which will activate the next function at the proper time.

When used as a memory element, a FF in the set state is said to store a "1." In the clear state it stores a "0." If the stored bit is Q, then the "1" output is "Q," and the "0" output is "$\overline{Q}$."

NOR Logic RS Flip-Flop

A typical NOR gate flip-flop is shown in Figure 65. In this case a 1 will set (S) or reset (R) the flip-flop if applied to the "S" or "R" input, respectively.

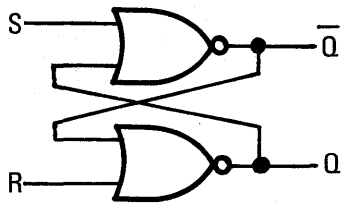


Figure 65. Typical NOR-Logic Flip-Flop

NAND Logic Flip-Flop

Many FF's require a logical zero to energize the inputs. The recommended symbol for this condition is shown in Figure 66.

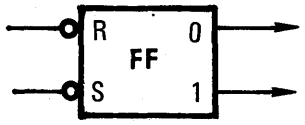


Figure 66. Flip-Flop Set With Logic Zero

A typical example is the NAND FF shown in Figure 67.

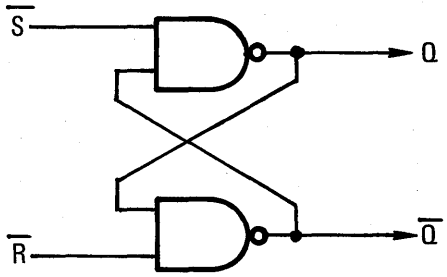
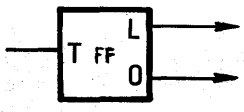


Figure 67. Typical NAND-Logic Flip-Flop

In the above circuit, the normal (or rest) levels for R and S are "1." If the system "0" is ground, then grounding either input will activate the FF. ($R \cdot S = 0$) is constrained.

The Type-T Flip-Flop

This is the triggered or complementing flip-flop. A pulse applied to the T terminal will change the state.



Some type-T FF's contain a differentiator so that an applied step will internally generate a pulse of the proper width. Others, especially integrated circuits, do not (why?), so a pulse must be applied.

The Type-D Flip-Flop

The type-D FF is a delay or shift FF. It has a clock input (also called a trigger or pulse input) plus two steering or gating lines. Three symbols for it are shown in Figure 68.

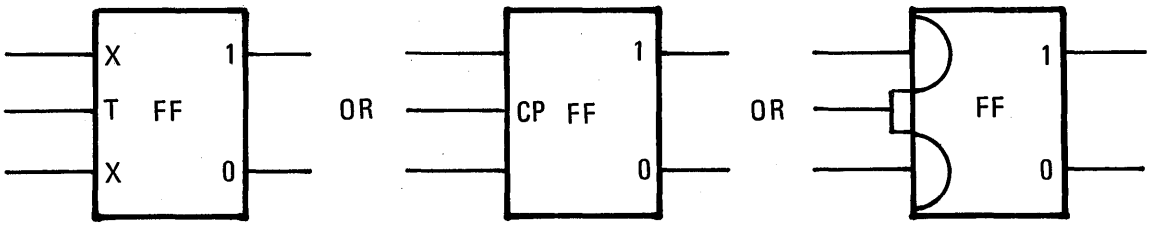


Figure 68. Type-D Flip-Flop Symbology

The shift flip-flop will assume the state of X when and only when the clock pulse is applied. A type-D can be made from four NAND gates as shown in Figure 69.

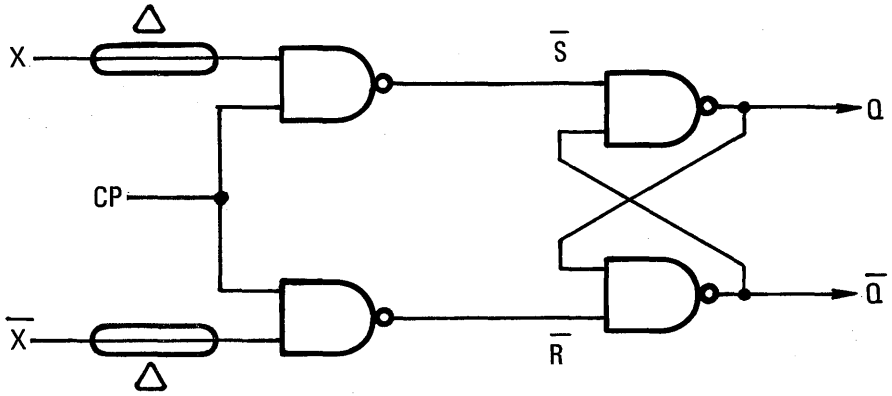


Figure 69. Type-D Flip-Flop

A complementing FF can be made from a type-D flip-flop as shown in Figure 70.

For proper operation, $T < \Delta < P$ (why?). Δ is either inherent or intentional circuit delay.

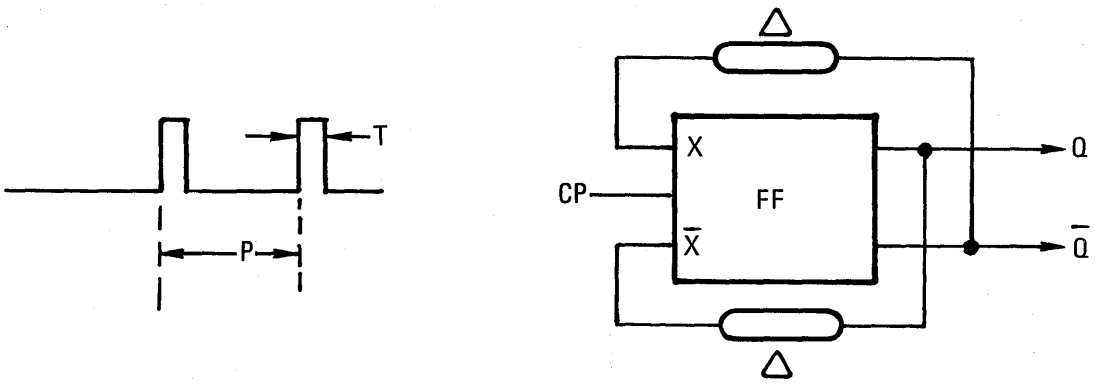
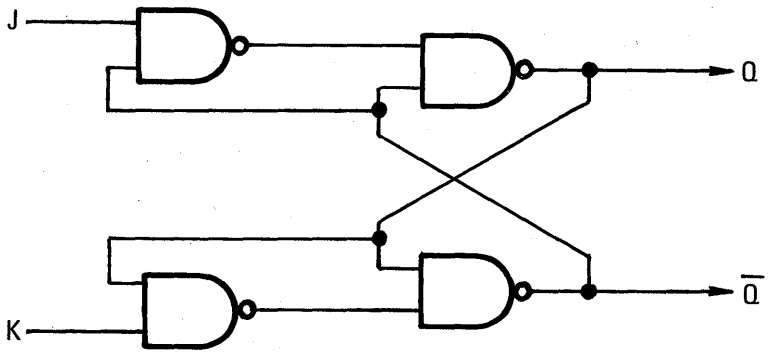


Figure 70. Complementing Flip-Flop

The JK Flip-Flop

In this variation, the J terminal acts as a set terminal, and the K as a reset. Unlike the RS, however, if J and K are both energized, the FF will change state, or complement. A NAND version is shown in Figure 71.

A far more versatile FF can be produced by adding a clock line to the JK FF.



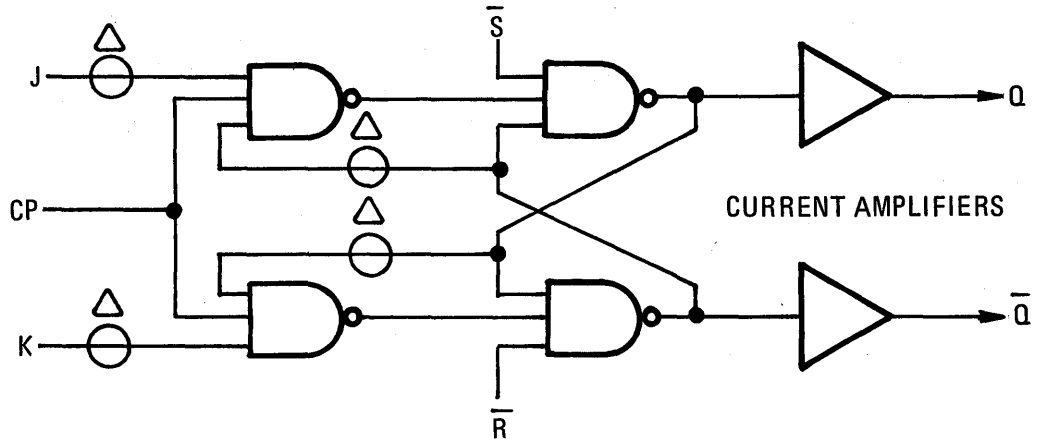
If J = 1, circuit will set
If K = 1, circuit will reset
If J · K = 1, circuit will change state—provided $T < \Delta < P$, as before
If (J + K) = 0, circuit will rest in either state.

Figure 71. Simple JK Flip-Flop

The Clocked JK Flip-Flop

The clocked JK has all of the abilities of the above types of FF's. It is a good general purpose FF. A NAND version is shown in Figure 72.

Very versatile flip-flops are available with multiple J and K terminals.



Let Q = old state, Q′ = new state and let X indicate that either state is possible; then

$\overline{R}$	$\overline{S}$	J	K	CP	Q′
1	1	0	0	0	X = Q
0	1	0	0	0	0
1	0	0	0	0	1
1	1	0	0	1	X = Q
1	1	1	0	1	1
1	1	0	1	1	0
1	1	1	1	1	$\overline{Q}$

Figure 72. Clocked JK Flip-Flop

Master-Slave JK Flip-Flop

Figure 73 is an example of two JK flip-flops connected in a "master-slave" relationship. The combined operation of the master and the slave is coordinated by the clock pulse. During the high-to-low transition of the clock pulse, the transfer from the master to the slave is inhibited, thus maintaining a steady state of the slave. Simultaneously, the data input lines A and B to the master are enabled so new data may be entered to the master.

During the low-to-high transition of the clock pulse, the transfer of data from the master to the slave is enabled via line C and D simultaneously. The inputs to the master are inhibited.

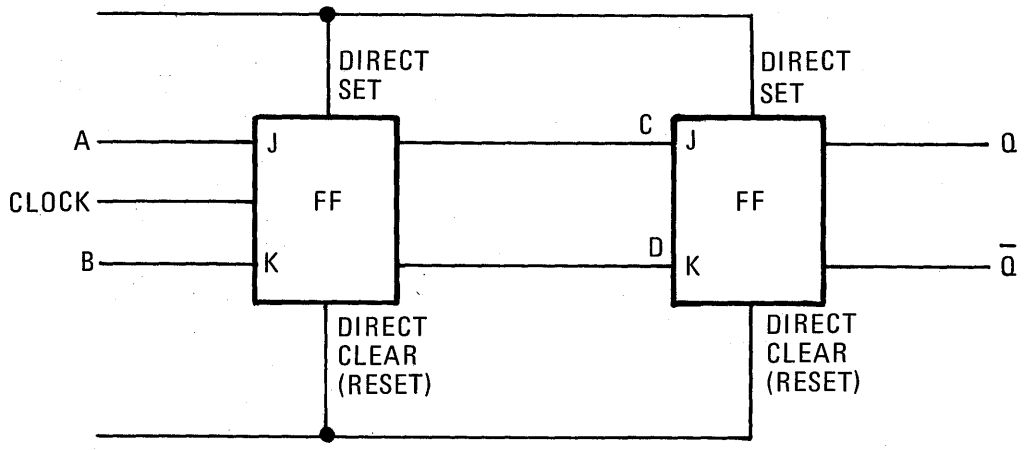


Figure 73. Example of Master-Slave JK Flip-Flop

There are two terminals, direct clear (C) and direct set (S), that may be used to directly clear (reset) or set the JK flip-flop. Thus, when several JK flip-flops are interconnected to form a register, a counter, or other logic device, the various flip-flops may be set up to a predetermined configuration of states.

Binary Register

The binary register symbol represents a group of flip-flops used in parallel to constitute a single register (as to store four bits of a character). It is necessary to indicate the number of "bits" of individual flip-flops in the register. Examples in Figure 74 show four "S" inputs grouped on one multiple input line and four each "1" and "0" grouped output lines. In some applications individual input and output lines are shown as in right-hand part of the figure.

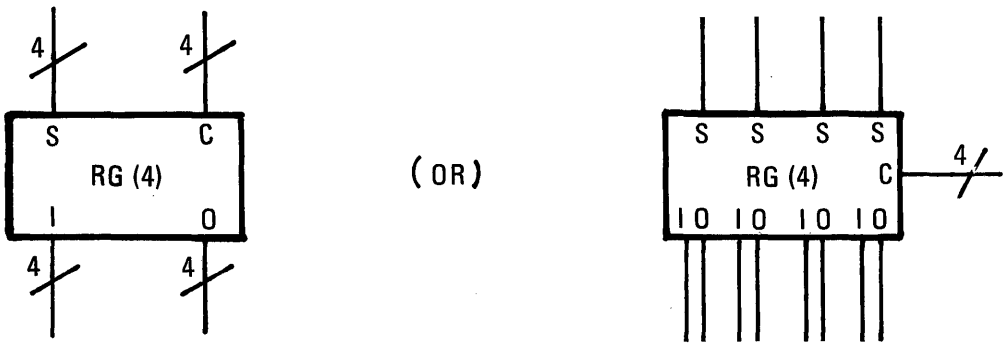


Figure 74. Example of Binary Registers

Shift Register

The shift register symbol represents a binary register with provision for displacing or shifting the content of the register one stage at a time to the right or left by means of the "shift" input. Examples are shown in Figure 75.

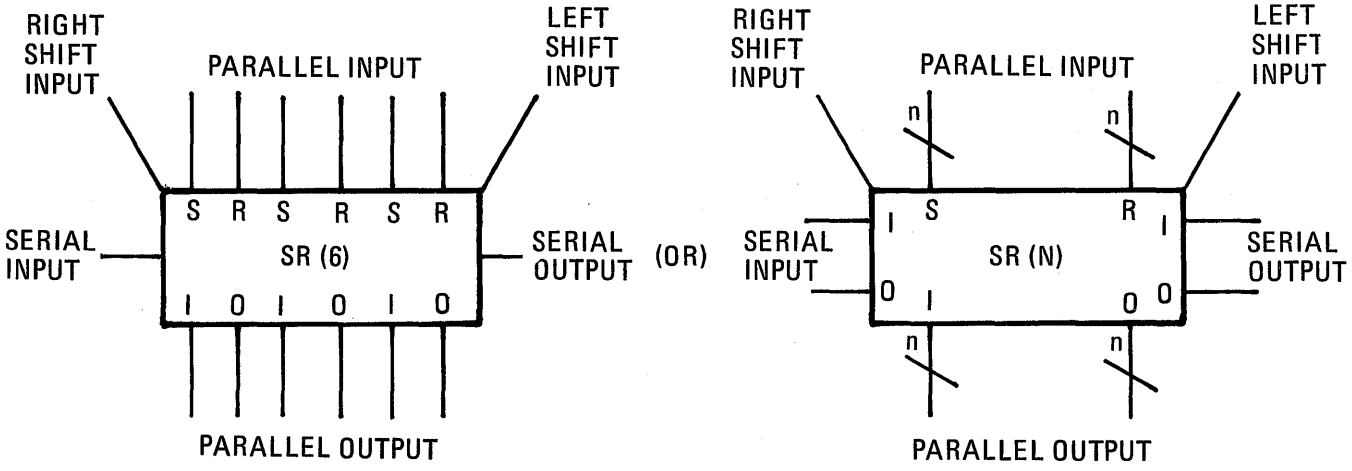


Figure 75. Example of Shift Registers

The words "right shift input" are placed at a left-hand corner of the symbol to indicate a shift from left to right. If the shift is from right to left, the words "left shift input" are placed at a right-hand corner of the symbol.

The choice of one of the register symbols depends on the diagram arrangement of the associated symbology.

Single-Shot

The symbols in Figure 76 are used to represent single-shot (SS) functions. Output-signal shape, amplitude, duration, and polarity are determined by the circuit characteristics of the "SS," (not by the input signal) and may be shown inside or outside the symbol. The unactuated state of the "SS" is either zero or one. When actuated, it changes to the opposite state and remains in the opposite state for the duration of the active time of the device.

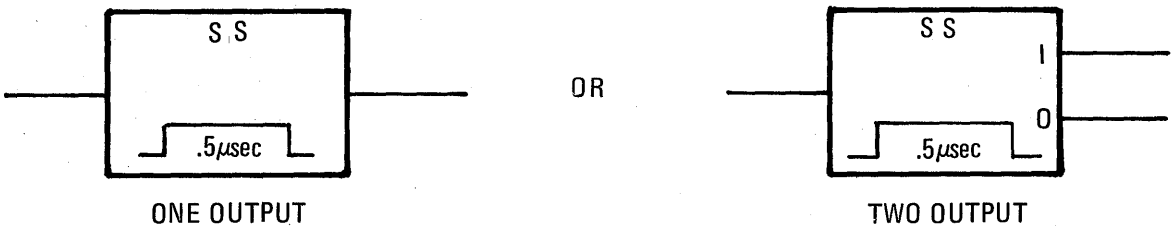


Figure 76. Example of Single-Shot

Schmitt Trigger

The symbols shown in Figure 77 represent the Schmitt Trigger (ST) function. The device is actuated when the input signal crosses a certain "threshold" voltage. Output signal amplitude and poarity are determined by the circuit characteristics of the "ST," (not by the input signal). Stylized waveforms may be shown (inside or outside the symbol), indicating amplitude, polarity, threshold voltage and duration. The unactuated state of "ST" is either zero or one. When actuated, it changes to the opposite state and remains in the opposite state as long as the input exceeds the threshold value.

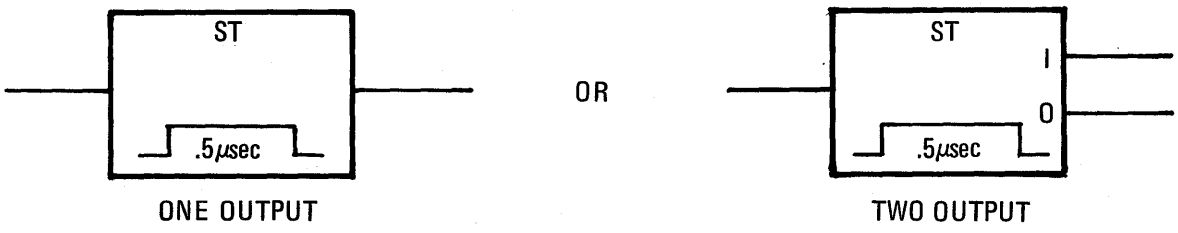


Figure 77. Example of Schmitt Trigger

Dot-AND and Dot-OR Gates

Where functions have the capability of being combined according to the AND (or OR) function, simply by having the outputs connected, that capability may be shown by enveloping the branched connection with a smaller sized AND or OR symbol as shown in Figure 78.

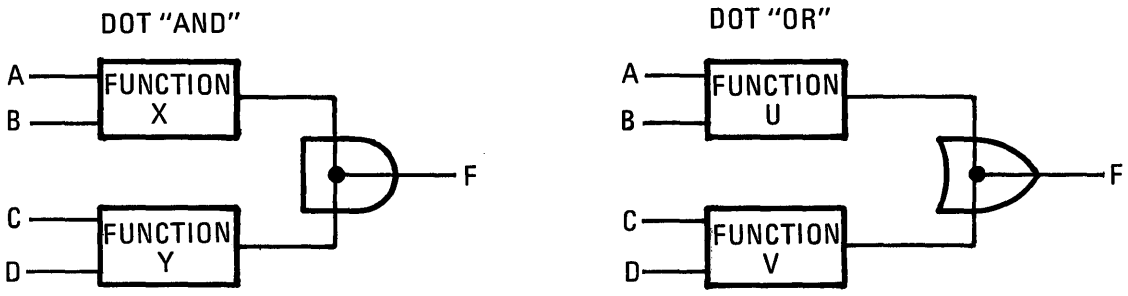


Figure 78. Example of Dot-AND and -OR Gates

Amplifier

This symbol represents a linear or nonlinear current or voltage amplifier. This amplifier may have one or more stages and may or may not produce gain or inversion. Level changers and inverters, pulse amplifiers, emitter followers, cathode followers, relay pullers, lamp drivers, and shift register drivers are examples of devices for which the symbol shown in Figure 79 is applicable.

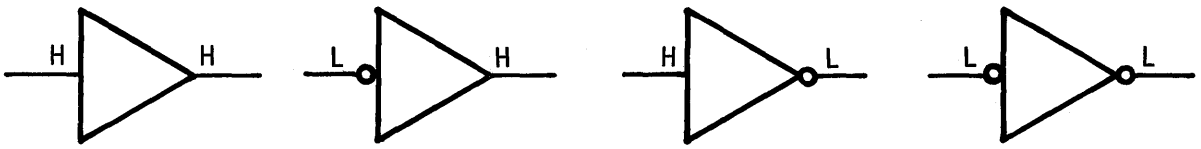


Figure 79. Amplifier Symbol

Time Delay

The duration of the delay is included with the symbol. If the delay device is tapped, the delay time with respect to the input shall be included adjacent to the tap output. Twin vertical lines indicates the input side. Figure 80 shows examples of time-delay symbols.

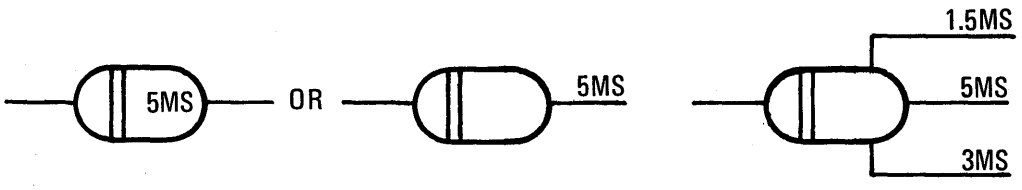


Figure 80. Time Delay Symbol

DIGITAL COMPUTER SYSTEM

The term "digital computer system" indicates or implies that digital computers comprise organized assemblages of equipment, data, instructions (programs, routines, and the like), and various media upon and by which these things are stored, distributed, and interrelated. Digital computers are indeed composed of various and different kinds of items that together perform various and different but interrelated functions. Thus, in a broad sense, the term "digital computer" means a system that computes with digits.

A narrower meaning of the term "digital computer" is the "central processor." The central processor does the computing under direction of the stored instructions (programming) on and with data, both of which are obtained from "memory." The actual computing equipment (the central processor), of course, can do nothing without instructions, and since a memory is needed to store the program, at least the central processing and memory are needed with which to compute. In the larger and the earlier computers, the memory was considered part of the central processor. Now, the memory is frequently considered a separate module or a group of modules so the size of the memory may be varied to fit the user's requirements.

Other functions—input, output, and control—are also within the domain of the central processor. Again, in the interest of modularity and customization, some of the hardware performing these functions are located partly with the central processor, partly with the memory, and partly with the peripheral units. Nevertheless, even though these central-processing subfunctions are somewhat spread out and combined with functions of the memory and the peripheral units, these subfunctions still exist.

The various subfunctions necessary for computing, notwithstanding their physical locations, are shown in Figure 81. Each of these subfunctions comprise a subsystem. Putting them all together, they perform the central-processing function—computing.

Another representation of a computer but with the central-processing subfunctions dispersed is shown in Figure 82. Memory has been completely separated from the central processor, whereas part of the input/output subfunctions and associated control also have been separated. Thus, with this sort of design, the computer system may be made more modular and may be customized, adding or leaving off peripheral input/output and memory modules to fit particular needs. This type of systemization has been designed into the FST-1. The various interface units and the memory are thought of as separate-to-central-processor subfunctions. The central processor thereby is reduced in physical and functional size and comprises primarily arithmetic functions. The main computing (central-processing) control are also within the domain of this reduced concept of the central processor.

Since the computer program that directs (instructs) the central processor is in memory and because the central processor cannot function without computer program direction, the term "computer" must now be thought of as a system, of which various portions, physically and functionally dispersed, nevertheless, interact to compute. Hence, we now see the term "computing system" or the like frequently used rather than "computer."

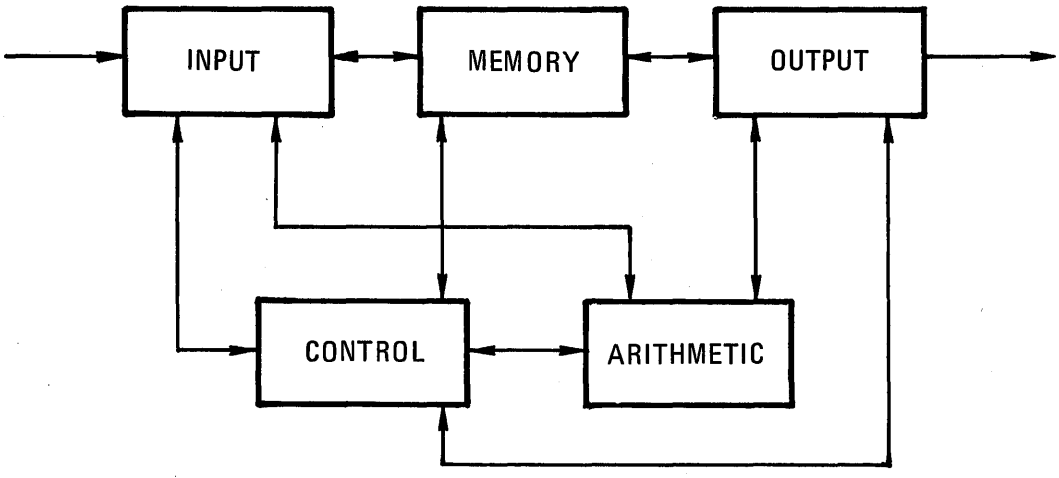


Figure 81. Computing Subfunctions

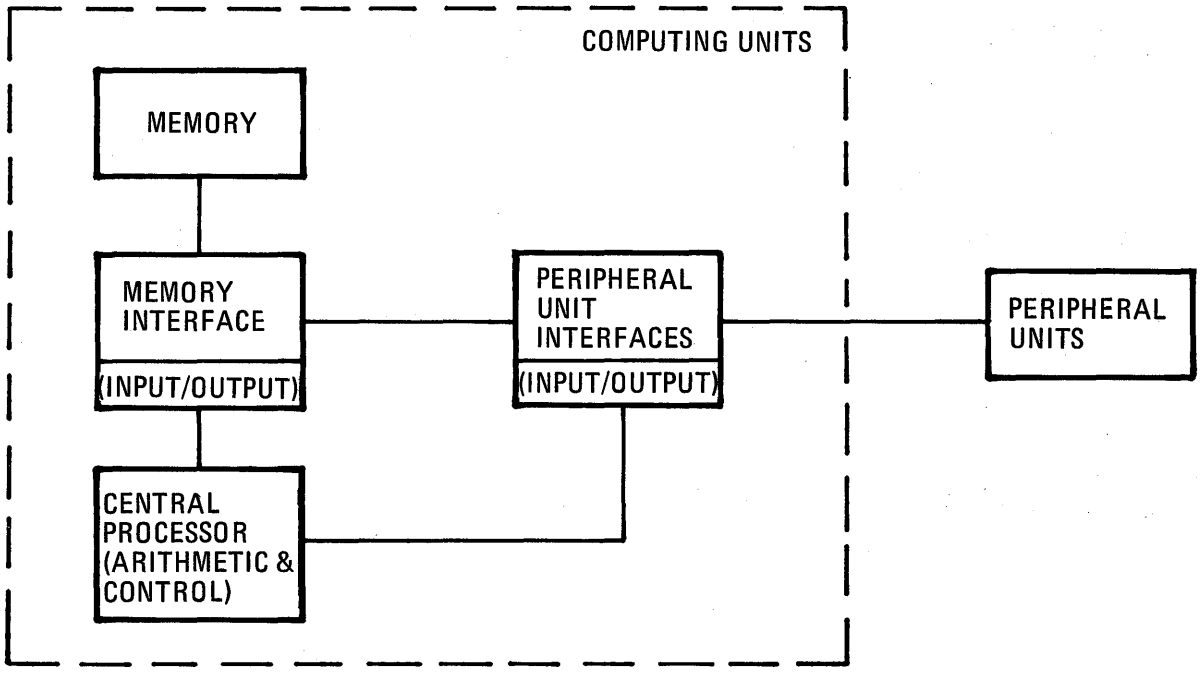


Figure 82. Modular Layout of Computer Units

FST-1 Computer System

The configuration of computer modules comprising the FST-1 Computer System affords a versatile, variable-size computer capability. The central processor, main memory, and peripheral units are separate modules that may be interconnected in sundry configurations. Thus, the desired complement-of-equipment may be custom-built. Figure 83 depicts the functional interconnections of the major units.

The core-memory and input/output control units constitute functional and physical entities separate from the arithmetic-control and data-processing control units.

Each memory module has a capacity of 4096 words, and one to four memory modules may be used with one CPU. Also, memory units are shared by the CPU and peripheral units, with memory access for each peripheral unit regulated by a memory-interface control as part of each memory module. However, a common memory-interface-control module for all memory-peripheral units is located in the CPU main frame. Other than for memory interface, the common-memory-interface control does not affect central processing.

Communications between memory and each peripheral unit and the CPU are time-shared on a priority basis. Once priority is established, the unit obtaining access independently exchanges data with memory under control of that unit's input/output control and the common memory-interface control.

Communications and the transfer-of-data between the CPU and peripheral units are effected by the input/output controls of the peripheral units and the CPU's accumulator interface on a priority basis. Once a peripheral unit gains access to the CPU, communications and data transfers between the peripheral unit and the CPU are carried out independent of the main program.

The complement-of-equipment may vary considerably. A simple computer system may comprise a CPU, a 4096-word core memory, an input/output peripheral unit (such as a Teletypewriter), and associated power supplies. Building blocks of memory, peripheral units, and interface modules are easily added to a system to increase its capability and diversity. A more-complex system may comprise additional units such as a card reader, a card punch, a magnetic tape, a line printer, and a magnetic-disc auxiliary-storage memory. These variations are easily envisioned with reference to Figure 83.

Facilities for one or two memory-access subsystems for each CPU are available. The memory-access subsystems operate relatively independently of one another, transferring data and control signals over their respective buses. In Figure 83 memory buses "A" and "B" are shown with each bus connecting all four memory modules to the CPU and the peripheral units. Both buses may be in use simultaneously as long as the memory modules are communicating with different units. This dual memory-access capability adds to the versatility of an FST-1 computer system that has two or more memory modules.

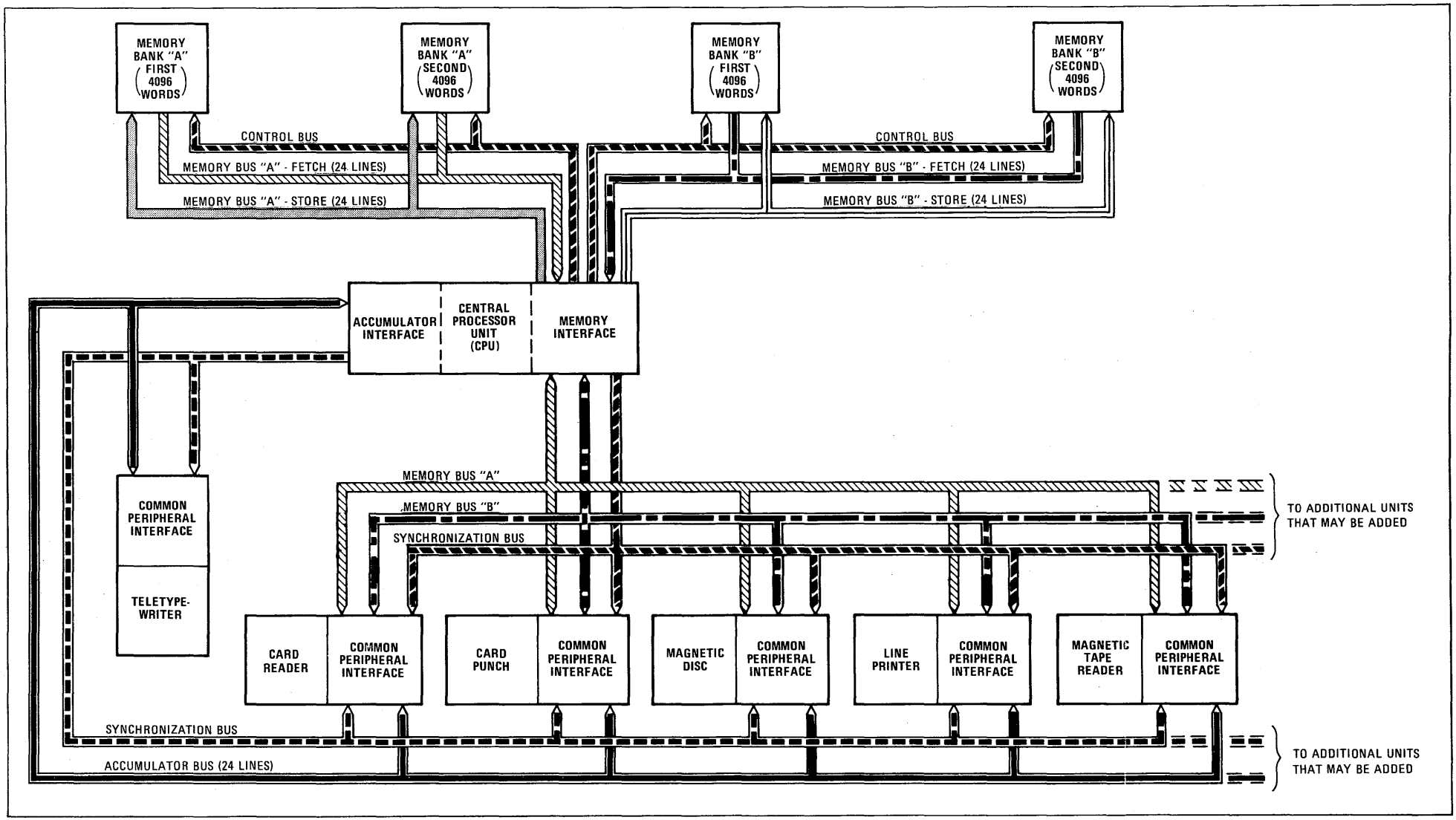


FIGURE 83. FST-1 Computer System, Typical Equipment Configuration

Outstanding Operational Features

- 24-bit data word
- Magnetic ferrite-core memory
- 4096-word memory modules
- Options to comprise a maximum of four memory modules, a total of 16,384 words per CPU
- Dual memory-access subsystems via two memory buses
- 1.75 microseconds memory-cycle time
- Random direct memory access, stored or retrieved at 571,000 words per second per memory bus
- Separate interface control between memory modules and CPU or peripheral units
- Interrupt subsystem for communications and datum transfer between CPU and peripheral units via accumulator bus
- 16 external interrupt channels and a maximum of 63 interrupt locations in each memory
- Eight index registers for address modification
- Indirect addressing for most instructions
- 75 (93_8) operational codes of following types of instructions
    - 8 (10_8) fetch and store
    - 9 (11_8) arithmetic
    - 6 logical
    - 8 register and state
    - 19 (23_8) branch (transfer-of-control)
    - 9 (11_8) shift
    - 16 (20_8) input/output

- Twos-complement, single- and double-precision arithmetic—add, subtract, and hardware-controlled multiply and divide

- Control-panel features

    - Program switches for manual program control

    - Control-flip-flop status indicators

    - Console switches for manually controlling a program execution

    - Program-counter-register status indicators

    - Switch-register switches for manual input of computer word to command register or accumulator

    - Index- and accumulator-register displays

    - Computer-control switches

Computer System Summary

The foregoing explanations of the modular construction of computer systems and particularly that of the FST-1 comptuer system are rather general and brief. To go into more detail such that would provide a thorough understanding of the FST-1 operation would be an encroachment on the training course itself.

Our goal in providing this preschool indoctrination is to provide an FST-1 prospective trainee with comprehensive background that will prepare him for the FST-1 training course. This training course covers the organization of the FST-1 compuer system, its operation, functional circuits, and use of engineering-level logic schematics.

The important points that should stand out in regard to comptuer systems in general are itemized below.

1. The five main functions: input/output/control/arithmetic/memory.

2. The dispersion of some of these five main functions outside of the CPU itself: separated memory modules and input/output interfaces.

3. The general relationship between memory and the CPU and peripheral units.

4. The dependence of the CPU and their hardware units on computer programming for direction (instructions) and vice versa: A stored-program digital computer cannot compute without a program and the program cannot be carried out without the hardware that understands its instructions and can operate accordingly.