Computer Architecture

Text Book:

Computer System Architecture - M. Morris Mano

UNIT - I

1. DATA REPRESENTATION

1.1 Data Representation

In digital computers, binary information is stored in the memory or in processor registers. Registers contain either data or control information. Control information is a bit or a group of bits used to specify the sequence of command signals needed for manipulation of the data. Data are numbers and other binary coded information that are used to achieve required computational results.

Data Types

The data types in the registers of digital computers are classified as:

- ✓ Numbers used in arithmetic computations
- ✓ Letters of the alphabet used in data processing
- ✓ Other discrete symbols used for specific purposes.

Registers are made up of flip-flops. Flip-Flops are two state devices that can store only 1's and 0's. So, all types of data, except binary numbers, are represented in the registers in binary coded form.

Number Systems

A number system of base or radix 'r' is a system that uses distinct symbols for 'r' digits. Numbers are represented by a string of digit symbols. To find the quantity that the number represents, you multiply each digit by an integer power of 'r' and then form the sum of all weighted digits.

Decimal Number system

The number system that is commonly used is the decimal number system. This decimal number system employs the radix 10 system. The 10 symbols are 0,1,2,3,4,5,6,7,8 and 9, For example, the string of digits 987.4 is interpreted as give below to represent the quantity.

$9 \times 10^2 + 8 \times 10^1 + 7 \times 10^0 + 4 \times 10^1$

The quantity is 9 hundreds, plus 8 tens, plus 7 units, plus 4 tenths.

Binary Number system

The binary number system uses the radix 2. The two digit symbols of this number system are 0 and 1. The string of digits 1010001 is interpreted to represent the quaintly 81.

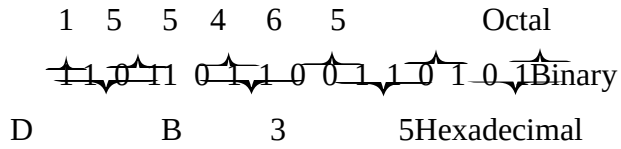
$1x2^6+0x2^5+1X2^4+0X2^3+0x2^2+0x2^1+1x2^0=81$

To show the difference between the radix numbers, the digits are enclosed in parentheses and the radix of the number is given as the subscript. The above string is equated as $(1010001)_2 = (81)_{10}$.

Octal and Hexadecimal Representation

Each octal digit corresponds to three binary digits and each hexadecimal digit corresponds to four binary digits, since $2^3 = 8$ and $2^4 = 16$. The conversion from binary to octal is easily accomplished by partitioning the binary number into groups of three bits each. The Corresponding octal digits obtained gives the octal equivalent of the binary number. We know that a 16-bit register is composed of 16 binary storage cells, where each cell can hold

either a 0 or 1. Now consider the bit configuration stored in the register as shown in the following page.



The 16-bit register can store any binary number from 0 to $2^{16} - 1$. The decimal equivalent of the binary number in the above example is 34217. Starting from the low-order bit, we split the register into groups of three bits each. The sixteenth bit remains in a group by itself. Each group of three bits is assigned its octal equivalent and placed on top of the register. The string of octal digits 155465 thus obtained represents the octal equivalent of the binary number.

Similarly, conversion from binary to hexadecimal is done. The only difference is that the bits are split into groups of four. The string of hexadecimal digits DB35 thus obtained represents the hexadecimal equivalent of the binary number.

1-1.Binary –code Octal Numbers

Octal Number	Binary – Coded Octal	Decimal Number	
0	000	0	
1	001	1	
2	010	2	Code for single octal digit
3	011	3	
4	100	4	
5	101	5	
6	110	6	
7	111	7	
10	001 000	8	
32	011 010	26	
45	100 101	37	
56	101 110	46	
374	011 111 100	252	

1-2Binary - coded Hexadecimal Numbers

Hexadecimal Number	Binary – Coded Hexadecimal	Decimal Number	
0	0000	0	
1	0001	1	
2	0010	2	Code for single Hexadecimal digit
3	0011	3	
4	0100	4	
5	0101	5	
6	0110	6	
7	0111	7	

8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15
16	0001 0110	22
32	0011 0010	50
64	0110 0100	100
F9	1111 1001	249

When comparing the binary- coded octal and hexadecimal numbers with their binary number univalent, we find that the bit combination in all three representations is exactly the same. For example, the decimal number 99, when converted to binary becomes 1100011. The binary-coded octal equivalent to the decimal number 99 is 001 100 011 and the binary – coded hexadecimal of the decimal number 99 is 0110 0011. If we eliminated the leading zeros in these three binary representations, then their bit combination is identical. This is because of the straight forward conversion between binary numbers and octal or hexadecimal numbers.

Hence, a string of 1's and 0's stored in a register represents a binary number. But the same string of bits may be interpreted as an octal number in binary-coded form, it we divide the bits in groups of three or as a hexadecimal number in binary-coded form, if we divide the bits in groups of four.

In a digital computer, the register contains many bits. To specify the content of registers by their binary values, it requires a long string of binary digits. It is easy to specify the content of the register by their octal or hexadecimal equivalent because the number of digits is reduced to One-third in the octal designation and by one-fourth in the hexadecimal designation.

Alphanumeric Representation

In digital computers, data are in the form of not only numbers but also the letters of the alphabet and some special characters. An alphanumeric character set is a set of elements that includes the 10 decimal digits, the 26 letters of the alphabet and a number of special characters, such as #, $, &, +, +, !etc. The standard alphanumeric binary code is the American Standard Code for Information Interchange – ASCII. This ASCII code uses seven bits to code 128 characters. The binary code for few uppercase letters, decimal digits and special characters are given in the page no 6.

In digital computer operations, binary codes play an important role. The codes must be in binary because registers can hold only binary information. It is important that binary codes merely change the symbols and not the meaning of the discrete elements they represent. The Operations specified for the computer must consider the meaning of the bits stored in registers so that operations are performed on operands of the same type. While inspecting the bits of a computer register at random, we find that it represents some type of coded information rather than a binary number.

Table 1-3

Character	Binary Code
A	100 0001
B	100 0010
---	--- ---
---	--- ---
Z	101 1010
0	011 0000
1	011 0001
2	011 0010
3	011 0011
4	011 0100
5	011 0101
6	011 0110
7	011 0111
8	011 1000
9	011 1001
(	010 1000
-	010 1101
/	010 1111
+	010 1011
SPACE	010 0000

Binary codes can be interpreted for any set of discrete elements such as chess pieces, their positions on the chessboard, the musical notes etc., They can also be used to interpret instructions that specify control information for the computer.

1.2 Complements

Complements are used to simplify the subtraction operation and for logical manipulation. There are two types of complements for each base 'r' system. They are r's complement and the (r-1)'s complement. The two types are referred to as the 2's and 1's complement for binary numbers and the 10's and 9's complement for decimal numbers.

(r-1)'s Complement

9's Complement

10^n represents a number that consists of a single 1 followed by n 0's. $10^n - 1$ is a number represented by n 9's. For example, with n=5 we have $10^5 = 100000$ and $10^5 - 1 = 99999$. That is, the 9's complement of a decimal number is obtained by subtracting each digit from 9. Example: The 9's complement of 46530 is 99999 – 46530 = 53469.

1's complement

2^n is represented by a binary number that consists of a 1 followed by n 0's. $2^n - 1$ is a binary number represented by n 1's. For example, with n = 5, we have $2^5 = (100000)_2$ and $2^5 - 1 = (11111)_2$. That is, the 1's complement of a binary number is obtained by subtracting each digit from 1. We know that the subtraction of a binary digit from 1. We know that the subtraction of a binary digit from 1 causes the bit to change from 0 to 1 (or) from 1 to 0. Therefore, the 1's complement of a binary number is formed by changing 1's into 0's and 0's into 1's. Example: the 1's complement of 01100101 is 0011010.

(r's) complement

The r's complement of an n-digit number N in base r is defined as $r^n - N$ when N is not equal to 0. It is 0 when N is equal to). r's complement is obtained by adding 1 to the (r-1)'s complement because $r^n - N = ((r^n - 1) - N) + 1$.

10's complement and 2's complement

The 10's complement of the decimal number 3256 is given by (9999-3256) +1 = 6744 ie.; first change the given decimal to 9's complement and add 1 to it. The 2's complement of binary 100110 is 011001 + 1= 011010 which is obtained by adding 1 to the 1's complement of the given binary value.

Since, 10^n is a number represented by a 1 followed by n)'s, then $10^n - N$, which is the 10's complement of N, can be formed by leaving all least significant 0's unchanged, subtracting the first nonzero least significant digit from 10, and then subtracting all higher of 345200 is 654800 which is obtained by leaving the two zeros unchanged, subtracting 2 from 10 and subtracting the other three digits from 9.

Similarly, the 2's complement can be formed by leaving all least significant 0's and the first 1 unchanged, and then replacing 1's by 0's and 0's by 1's in all other higher significant bits. For example, the 2's complement of 11001100 is 00110100 which is obtained by leaving the two low order 0's and the first 1 unchanged and then replacing 1's by 0's and 0's by 1's in the other four most significant bits.

Hence, the complement of the complement restores the number to its original value. The r's complement of N is $r^n - N$. the complement of the complement is $r^n - (r^n - N) + N$ which is the original number.

1.3. Fixed point representation

Integers that are positive, including zero, can be represented as unsigned numbers. But a sign is needed to represent negative integers. Commonly in arithmetic, negative number is indicated by a minus sign and a positive number by a plus sign. In computers, everything must be represented with 1's and 0's only, including the sign of the number. So, the bit in the leftmost position of the number is used to represent the sign. This sign bit is conventionally equal to 0 for a positive number and 1 for a negative number. Along with the sign, a number may have a decimal point also. This can also be called as binary point. It is important to know the position of the binary point to represent fractions, integers of mixed integer-fraction numbers. There are two ways to specify the position of the binary point in a register.

- By giving it a fixed position
- By giving a floating point representation.

The fixed point method assumes that the binary point is always fixed in one position. The two positions widely used are:

- ✓ A binary point in the extreme left of the register to make the stored number a fraction.
- ✓ A binary point in the extreme right of the register to make the stored number an integer.

In both cases, the binary point is not actually present, but its presence is assumed from the fact that the number stored in the register is treated as a fraction or as an integer.

Integer representation

When a positive integer is represented in binary number, the sign is represented by 0 and the magnitude by a positive binary number. When the number is negative, the sign is represented by 1 but the rest of the number may be represented in one of the following three possible ways.

- Signed – magnitude representation
- Signed 1's complement representation
- Signed 2's complement representation

A negative number represented as signed-magnitude consists of the magnitude and a negative sign. The same number is represented in the other two ways as either the 1's or 2's complement of its positive value.

Consider the signed number 12 stored in an 8 bit register. The binary equivalent of 12 is 1100. It is stored in the 8 bit register as 00001100. This is because each of the eight bits of the register must have a value and therefore 0's must be inserted in the most significant positions following the sign bit. There is only one way to represent +12. But there are three different ways to represent – 12 with eight bits.

In signed magnitude representation 1 0001100
(Change only the sign bit i.e., positive 0 to negative 1)

In signed 1's complement representation 1 1110011
(Change3 all 1's into 0's into 1's including the sign bit and then add 1 to it)

In signed 2's complement representation 1 1110100
(Change all 1's in to 0's and 0's into 1's including the sign bit and then add 1 to it).

Normally the signed magnitude method is used only in ordinary arithmetic. The signed 1's complement method is useful in logical operation since the change of 1 to 0 and 0 to 1 is equivalent to a logical complement operation. The 2's complement method is useful in the representation of negative numbers addition, subtraction etc.

Arithmetic addition:

The addition of two numbers in the magnitude system follows the same rules of ordinary arithmetic. If the signs are the same, we add the two numbers and give the sum the common sign. If the signs are different, we subtract the smaller number from the larger and give the result the sign of the larger number.

Examples:

- 5	=	11111011
+12	=	00001100
+ 7	=	00000111

+ 5	=	00000101
+12	=	00001100
+17	=	00010001

The rule for adding numbers in the signed 2's complement system does not require a comparison or subtraction, but only addition and complementation. First add the two numbers including their sign bits and discard any carry out of the sign bit position i.e., the leftmost bit is any.

Arithmetic subtraction

The subtraction of two signed binary numbers is very simple. Whennegative numbers are in 2's complement form. First take the 2's complement of the subtrahend, including the sign bit and add it to the minuend, including the sign bit. Finally discard any carry out of the sign bit. This procedure reveals the fact that a subtraction operation can be changed to an addition operation if the sign of the negative number is easily done by taking its 2's complement. The reverse is also true because the complement of a negative number in complement form produces the equivalent positive number.

Consider the subtraction of (-5) and (-12)i.e.

((-5) – (-12)) = + 7

In binary with eight bits,

-5 = 11111011 and -12 = 11110100

This is obtained by changing all 1's to 0 and 0's to 1 of the binary numbers +5 and +12 for all the eight bits and then adding one to the least bit.

Now once again complement -12 binary value ie.; 11110100 is changed to 00001011. Add one to the least bit. You will get 00001100. Now add this to binary value of -5 to get the result +7. i.e., 11111011 + 00001100 = 1 00000111. Discard the end carry 1 and obtain the correct answer 00000111, this is + 7.

It is important to note that binary numbers in the signed 2's complement system are added and subtracted by the same basic addition and subtraction rules as unsigned numbers.

Overflow

When any two numbers consisting of n digits are added and the sum occupies n + 1 digits, then we say that an overflow has occurred. This is a problem in digital computers because the registers have a finite width. The n+1 result cannot be accommodated in the register of definite width of n bits. So many computers detect acorresponding flip-flop is set which can then be checked by the user.

After the addition of two binary numbers, the detection of an overflow depends on whether the numbers are considered to be signed or unsigned. In the addition of two unsigned numbers, an overflow is detected from the end carry out of the most significant position. In the case of signed numbers, the leftmost bit always represents the sign and the negative numbers are in $2's_{s}$ complement form. So in the addition of two signed numbers, the sign bit is treated as part of the number and the end carry does not indicate an overflow.

Thus, an overflow cannot occur when adding a positive and a negative numbers. An overflow may occur when the numbers added are both positive (or) both negative.

An Overflow can be detected by observing the carry into the sign bit position and the carry out of the sign bit position. If these two carries are not equal, an overflow condition is produced.

Decimal Fixed point representation

The representation of signed decimal numbers in BCD is similar to the representation of signed numbers in binary. We can either use the signed magnitude system or the signed complement system. The sign of a decimal number is usually represented with four bits to confirm with the 4-bit code of the decimal digits. A 4-bit decimal code requires 16 flip-flops, four flip-flops for each digit.

Considerable amount of storage space is wasted by representing numbers in decimal because the number of bits needed to store a decimal number in a binary code is greater than the number of bits needed for its equivalent binary representation. For this reason, some computers and electronic calculators perform arithmetic operations directly with the decimal data (in a binary code). This helps in eliminating the need for conversion to binary and back to decimal.

Addition of decimal numbers are done by adding all digits, including the sign digit, and discarding the end carry. The subtraction of decimal numbers either unsigned or in the signed 10's complement system is the same as in the binary case. Take the 10's complement of the subtrahend and add it to the minuend.

1.4 Floating point representation

The floating point representation of a number has two parts.

- ✓ A signed, fixed point number called the mantissa.
- ✓ The position of the decimal or binary point called the exponent.

The fixed point mantissa may be a fraction or an integer. For example, the decimal number +3245.698 is represented in floating point with a fraction as +0.3245698 and an exponent as +04. The value of the exponent indicates that the actual position of the decimal point is four positions to the right of the indicated decimal point in the fraction. This representation is equivalent to the scientific notation $+0.3245698 \times 10^{-4}$. Hence, a floating point is always represented as $m \times r^e$. In the register, only the mantissa 'm' and the exponent 'e' are physically represented, including the signs. The radix 'r' and the radix-point position of the mantissa are always assumed. The circuits that manipulate the floating point numbers in registers confirm with these two assumptions in order to provide the correct computational results.

A floating point binary number is also represented in the same manner except that it uses base 2 for the exponent. For example, the binary number +1001.11 is represented with an 8-bit fraction as 01001110 and 6-bit exponent as 000100. The fraction has a 0 in the left most position to denote positive. The binary point of the fraction follows the sign bit is not shown in the register. The exponent has the equivalent binary number +4. The floating point number is equivalent to

$$Mx2^e = + (.1001110)_2 \times 2^{+4}$$

A floating point number is said to be normalized if the most significant digit of the mantissa is non-zero. For example, the decimal number 430 is normalized but 00043 is not. In the same way, the 8-bit binary 00010101 is not normalized by shifting it three positions to the left and discarding the leading 0's to obtain 10101000. The three shifts multiply the numbers by $2^3 = 8$. The exponent must be subtracted by 3, to keep the same value for the floating point number is provided by the normalized numbers. A zero cannot be normalized because it does not have a nonzero digit. It is represented in floating point as all 0's in the mantissa and exponent.

Arithmetic operations with floating point numbers are more complicated. Their execution takes longer time and requires more complex hardware. However, floating point representation is a must for scientific computations because of the scaling problems involved with fixed-point computations. Many computers and electronic calculators have the built-in capability of performing floating point computations have a set of subroutines to help the user to program scientific problems with floating point numbers.

1.5 Error detection codes

Binary information transmitted through some form of communication medium is subject to external noise that could change bits from 1 to 0, and vice versa. An error detection code is a binary code that detects digital errors during transmission. The detected errors cannot be corrected but their presence is indicated. The usual procedure is to observe the frequency of errors. If errors occur infrequently at random, the particular erroneous information is transmitted again. If the error occurs too often, the system is checked for malfunction.

The most common error detection code used is the parity bit. A parity bitis an extra bit included with a binary message to make the total number of 1's either odd or even. A message of three bits and two possible parity bits is shown in Table 1-4. The P(odd) bit is chosen in such a way as to make the sum of 1's (in all four bits) odd. The P(even) bit is chosen to make the sum of all1's even. In either case, the sum is taken over the message and the P bit. In any particular application, one or the other type of parity will be adopted. The even-parity scheme has the disadvantage of having a bit combination of all 0's,while in the odd parity there is always one bit (of the four bits that constitute the message and P) that is 1. The P(odd) is the complement of the P(even).

During transfer of information from one location to another, the parity bit is handled as follows. At the sending end, the message (in this case three bits) is applied to a parity generator , where the required parity bit is generated. The message including the parity bit, is transmitted to its destination. At the receiving end, all the incoming bits (in this case, four) are applied to a parity checker that checks the proper parity adopted (odd or even). An error is detected if the checks parity does not confirm to the adopted parity. The parity method detects the presence of one, three, or any odd number of errors. An even number of errors is not detected.

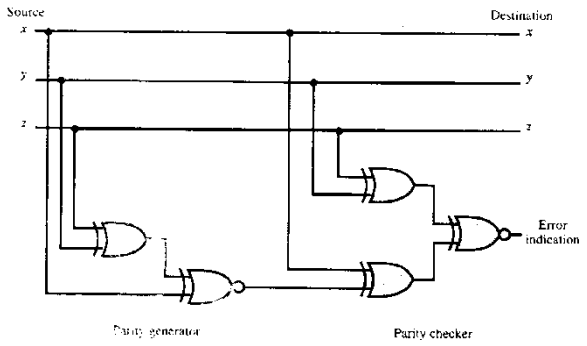
Parity generator and checker networks are logic circuits constructed with exclusive-OR functions. An odd function is a logic function whose value is binary 1 if, and only if, an odd number of variables are equal to 1. According to this definition, the P(even) function is the exclusive-OR of x, y, and z because it is equal to 1 when either one or all three of the variables are equal to 1 (Table 1-4). The p(odd) function is the complement of the P(even) function.

As an example, consider a 3-bit message to be transmitted with an odd parity bit. At the sending end, the odd-parity bit is generated by a parity generator circuit. As shown in Fig.1.1, this circuit consists of one exclusive-OR and one exclusive-NOR gate. Since P(even) is the exclusive-OR of x, y, z, and P(odd) is the complement of P(even), it is necessary to employ an exclusive-NOR gate for the needed complementation. The message and-the odd-parity bit are transmitted to their destination where they are applied to a parity checker. An error has occurred during transmission if the parity of the four bits received is even, since the binary information transmitted was originally odd.

TABLE 1-4 Parity Bit Generation

Message xyz	P(odd)	P(even)
000	1	0
001	0	1
010	0	1
011	1	1
100	0	1
101	1	0
110	1	0
111	0	1

Figure 1.1 Error detection with odd parity bit



The output of the parity checker would be 1 when an error occurs, that is, when the number of 1's in the four inputs is even. Since the exclusive-OR function of the four inputs is an odd function, we again need to complement the output by using an exclusive-NOR gate. It is worth noting that the parity generator can use the same circuit as the parity checker if the fourth input is permanently held at a logic-0 value. The advantage of this is that the same circuit can be used for both parity generation and parity checking.

It is evident from the example above that even-parity generators and checkers can be implemented with exclusive-OR functions. odd-parity networks need an exclusive-NOR at the output to complement the function.

2. REGISTER TRANSFER AND MICROOPERATIONS

2.1 Register Transfer Language

The operations executed on data stored in registers are called microoperations. A microoperation is an elementary operation performed on the information stored in one or more registers. The result of the operation may replace the previous binary information of a register or may be transferred to another register.

It is possible to specify the sequence of microoperations in a computer by explaining every operation in words, but this procedure usually involves a lengthy descriptive explanation. It is more convenient to adopt a suitable symbology to describe the sequence of transfers between registers and the various arithmetic and logic microoperations associated with the transfers.

The symbolic notation used to describe the microoperation transfers among register is called a register transfer language. The term "register transfer" implies the availability of hardware logic circuits that can perform a stated microoperation and transfer the result of the operation to the same or another register. The word language is borrowed from programmers, who apply, a natural language

A register transfer language is a system for expressing in symbolic form the microoperation sequences among the registers of a digital module. It is a convenient tool for describing the internal organization of digital computers in concise and precise manner. It can also be used to facilitate the design process of digital systems.

2.2 Register Transfer

Computer registers are designated by capital letters (sometimes followed by numerals) to denote the function of the register.

The register that holds an address for the memory unit is usually called a memory address register and is designated by the name MAR. Other designations for registers are PC (for program counter) IR (for instruction register) and R1 (for processor register)

The individual flip-flops in an n-bit register are numbered in sequence from 0 through n – 1, starting from 0 in the rightmost position and increasing the numbers toward the left.

The most common way to represent a register is by a rectangular box with the name of the register inside, as in Fig 2-1 (a). The individual bits can be distinguished as in (b). The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c). A 16-bit register is partitioned into two parts in (d). Bits 0 through 7 are assigned the symbol L (for low byte) and bits 8 through 15 are assigned the symbol H (for high byte). The name of the 16-bit register is PC. The symbol PC (0-7) or PC (L) refers to the low-order byte and PC (8-15) or PC (H) to the high-order byte.

Information transfer from one register to another is designated in symbolic form by means of a replacement operator. The statement

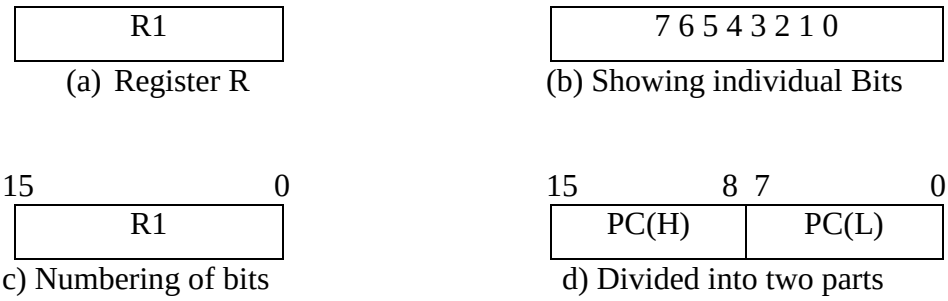
$$R2 \leftarrow R1$$

denotes a transfer of the content of register R1 into register R2. It designates a replacement of the content of R2 by the content of R1. By definition, the content of the source register R1 does not change after the transfer

A statement that specifies a register transfer implies that circuits are available from the outputs of the source register to the inputs of the destination register and that the destination register has a parallel load capability. Normally we want the transfer to occur only

under a predetermined control condition. This can be shown by means of an if-then statement.

Figure 2.1 Block diagram of register



If (P = 1) then (R2←R1)

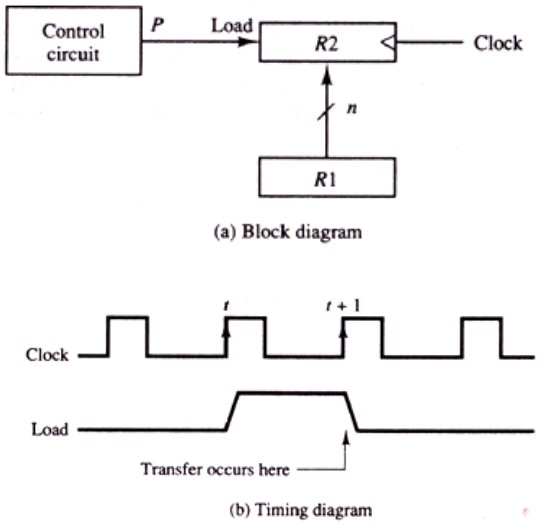
Where P is a control signal generated in the control section. It is sometimes convenient to separate the control variables from the register transfer operation by specifying a control function. A control function is a Boolean variable that is equal to 1 or 0. The control function is included in the statement as follows.

P: R2 ← R1

The control condition is terminated with a colon. It symbolizes the requirement that the transfer operation be executed by the hardware only if P = 1.

Every statement written in a register transfer notation implies a hardware construction for implementing the transfer. Figure 2-2 shows the block diagram that depicts the transfer from R1 to R2. The letter n will be used to indicate any number of bits for the register. It will be replaced by an actual number when the length of the register is known. Register R2 has a load input that is activated by the control variable. It is assumed that the control variable is synchronized with the same clock as the one applied to the register.

Figure 2.2 Transfer from R1 to R2 when P = 1



As shown in the timing diagram P is activated in the control section by the edge of a clock pulse at time t. The next positive transition of the clock at time t + 1 finds the load input active and the data inputs of R2 are then loaded into the register in parallel. P may go back to 0 at time t + 1, otherwise, the transfer will occur with every clock pulse transition while P remains active

Note that the clock is not included as a variable in the register transfer statements. It is assumed that all transfers occur during a clock edge transition. Even though the control condition such as P becomes active just after time t the actual transfer does not occur until the register is triggered by the next positive transition of the clock at t + 1.

The basic symbols of the register transfer notation are listed in Table 2-1. Registers are denoted by capital letters, and numerals may follow the letters. Parentheses are used to denote a part of a register by specifying the range of bits or by giving a symbol name to a portion of a register. The arrow denotes a transfer of information and the direction of transfer. A comma is used to separate two or more operations that are executed at the same time. The statement

T: R2←R1, R1 ←R2

denotes and operation that exchange the contents of two registers during one common clock pulse provided that T = 1. This simultaneous operation is possible with registers that have edge-triggered flip-flops.

Table 2-1 Basic symbols for Register Transfers

Symbol	Description	Examples
Letter (and numerals)	Denotes a register	MAR, R2
Pare theses ( )	Denotes a part of a register	R2 (0-7), R2 (L)
Arrow ←	Denotes transfer of information	R2 ← R1
Comma,	Separates two microoperations	R2 ← R1, R1 ← R2

2.3 Bus and Memory Transfers

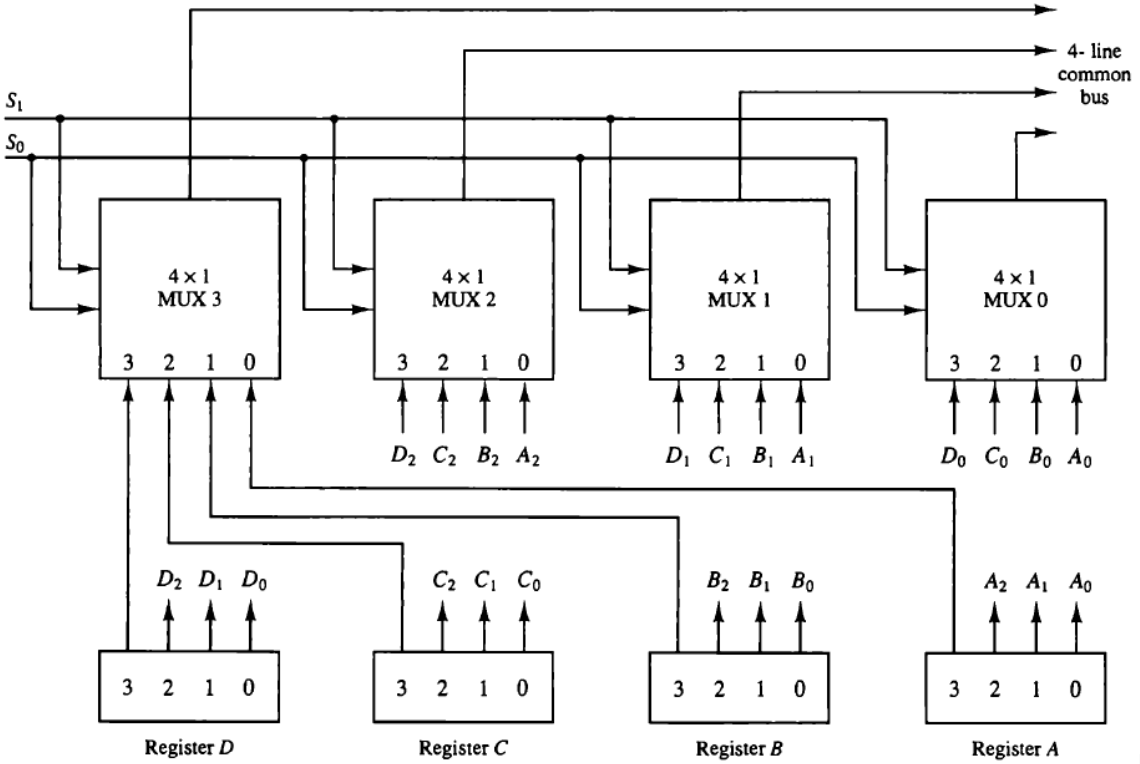
A typical digital computer has many registers, and paths must be provided to transfer information from one register to another. The number of wires will be excessive if separate lines are used between each register and all other registers in the system. A more efficient scheme for transferring information between registers in a multiple-register configuration is a common bus system. A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals determine which register is selected by the bus during each particular register transfer.

One way of constructing a common bus system is with multiplexers. The multiplexers select the source register whose binary information is then placed on the bus. The construction of a bus system for four register is shown in Fig. 2.3. Each register has four bits, numbered 0 through 3. The bus consists of four 4 x 1 multiplexers each having four data inputs, 0 through 3, and two selection inputs, S1 and S0.

For example, output 1 of register A in connected to input 0 of MUX 1 because this input is labeled A_1. The diagram shows that the bits in the same significant position in each

register are connected to the data inputs of one multiplexer to form one line of the bus. Thus MUX 0 multiplexes the four 0 bits of the register, MUX 1 multiplexes the four 1 bits registers, and similarly for the other two bits.

Figure 2.3 Bus system for four registers



The two selection lines S_1 and S_0 are connected to the selection inputs of all four multiplexers. The selection lines choose the four bits of one register and transfer them into the four-line common bus. When $S_1S_0 = 00$, the 0 data inputs of all four multiplexers are selected and applied to the outputs that form the bus. This causes the bus lines to receive the content of register A since the outputs of this register are connected to the 0 data inputs of the multiplexers. Similarly, register B is selected if $S_1S_0 = 01$, and so on. Table 2-2 shows the register that is selected by the bus for each of the four possible binary value of the selection lines.

Table 2-2 Function table for bus of fig 2.3

S_1	S_0	Register selected
0	0	A
0	1	B
1	0	C
1	1	D

In general, a bus system will multiplex k registers of n bits each to produce an n-line common bus. The number of multiplexers needed to construct the bus is equal to n, the number of bits in each register. The size of each multiplexer must be k x 1 since it multiplexes k data lines, For example, a common bus for eight registers of 16 bits each requires 16 multiplexers, one for each line in the bus. Each multiplexer must have eight data input lines and three selection lines to multiplex one significant bit in the eight registers.

The transfer of information from a bus into one of many destination registers can be accomplished by connecting the bus lines to the inputs of all destination registers and activating the load control of the particular destination register selected. The symbolic statement for a bus transfer may mention the bus or its presence may be implied in the statement. When the bus is included in the statement the register transfer is symbolized as follows:

BUS ← C, R1 ← BUS

The content of register C is placed on the bus, and the content of the bus is loaded into register R1 by activating its load control input. If the bus is known to exist in the system, it may be convenient just to show the direct transfer.

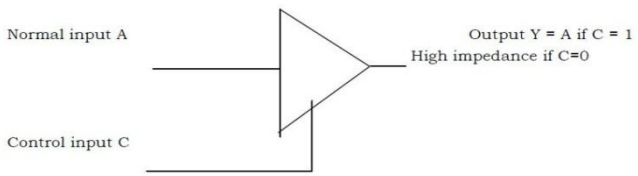
R1 ← C

From this statement the designer knows which control signals must be activated to produce the transfer through the bus.

Three State Bus Buffers

A bus system can be constructed with three-state gates instead of multiplexers. A three-state gate is a digital circuit that exhibits three states. Two of the states are signals equivalent to logic 1 and 0 as in a conventional gate. The third state is a high impedance state. The high impedance state behaves like an open circuit which means that the output is disconnected and does not have a logic significance. Three state gates may perform and conventional logic such As AND or NAND.

Figure 2.4 Graphic symbols for three-state buffer



The graphic symbol of a three state buffer gate is shown in fig.2.4. it is distinguished from a normal buffer by having both a normal input and a control input. The control input determines the output state. When the control input is equal to1, the output is enabled and the gate behaves like conventional buffer, with the output equal to the normal input. When the control input is 0, the output is disabled and the gate goes to a high impedance state, depending on the value of the normal input.

Memory Transfer

The transfer of information from a memory word to the outside environment is called a read operation. The transfer of new information to be stored into the memory is called a write operation. A memory word will be symbolized by the letter M. The particular memory word among the many available is selected by the memory address during the transfer. It is necessary to specify the address of M when writing memory transfer operations. This will be done by enclosing the address in square brackets following the letter M.

Consider a memory unit that receives the address from a register, called the address register, symbolized by AR. The data are transferred to another register, called the data register, symbolized by DR. The read operation can be stated as follows:

Read: DR ← M (AR)

This causes a transfer of information into DR from the memory word M selected by the address in AR.

The write operation transfers the content of a data register to a memory word M selected by the address. Assume that the input data are in register R1 and the address is in AR. The write operation can be stated symbolically as follows:

Write: M (AR) ← R1

This causes a transfer of information from R1 into the memory word M selected by the address in AR.

2.4 Arithmetic Microoperations

The microoperations most often encountered in digital computer are classified into four categories:

1. Register transfer microoperations transfer binary information from one register to another.
2. Arithmetic microoperations perform arithmetic operations on numeric data stored in registers.
3. Logic microoperations perform bit manipulation operations on nonnumeric data stored in registers.
4. Shift microoperations perform shift operations on data stored in registers.

The register transfer microoperation was introduced in Sec. 2.2. This type of microoperation does not change the information content when the binary information moves from the source register to the destination register. The other three types of microoperations change the information content during the transfer.

The basic arithmetic microoperations are addition subtraction increment decrement and shift. Arithmetic shifts are explained later in conjunction with the shift microoperations. The arithmetic microoperation defined by the statement

R3 ← R1 + R2

specifies an add microoperation. It states that the contents of register R1 are added to the contents of register R2 and the sum transferred to register R3. To implement this statement with hardware we need three registers and the digital component that performs the addition operation. The other basic arithmetic microoperations are listed in Table 2-3. Subtraction is

most often implemented through complementation and addition. Instead of using the minus operator, we can specify the subtraction by the following statement.

-

$R3 \leftarrow R1 + R2 + 1$

R2 is the symbol for the 1's complement of R2. Adding 1 to the 1's complement produces the 2's complement. Adding the contents of R1 to the 2's complement of R2 is equivalent to R1 – R2.

Table 2-3 Arithmetic Microoperation

Symbolic designation	Description
$R3 \leftarrow R1 + R2$	Contents of R1 plus R2 transferred to R3
$R3 \leftarrow R1 - R2$	Contents of R1 minus R2 transferred to R3
$R2 \leftarrow \overline{R2}$	Complement the contents of R2 (1`s Complement)
$R2 \leftarrow \overline{R2} + 1$	2`s Complement the contents of R2 (negate)
$R3 \leftarrow R1 - \overline{R2} + 1$	R1 plus the 2`s complement of R2 (Subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of R1 by one
$R1 \leftarrow R1 - 1$	Decrement the contents of R1 by one

The increment and decrement microoperations are symbolized by plus one and minus one operations, respectively. These microoperations are implemented with a combinational circuit or with a binary up-down counter.

Binary Adder

The digital circuit that forms the arithmetic sum of two bits and a previous carry is called a full-adder (see Fig. 1-17). The digital circuit that generates the arithmetic sum of two binary numbers of any length is called a binary adder. The binary adder is constructed with full-adder circuits connected in cascade, with the output carry from one full-adder connected to the input carry of the next full-adder.

Figure 2.5 4 bit binary adder

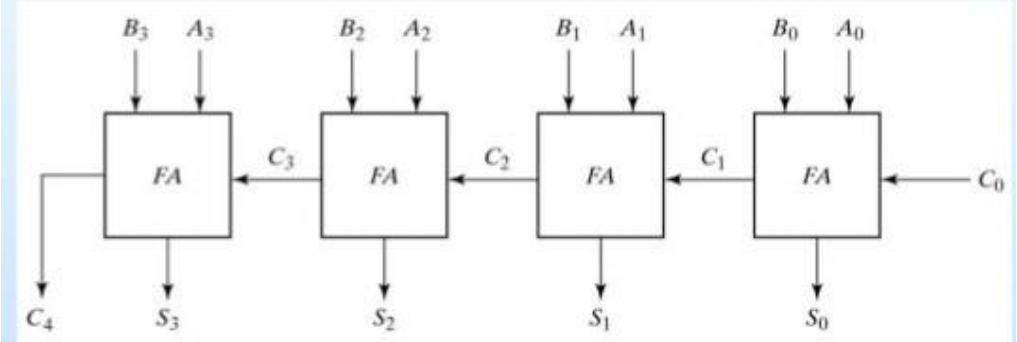


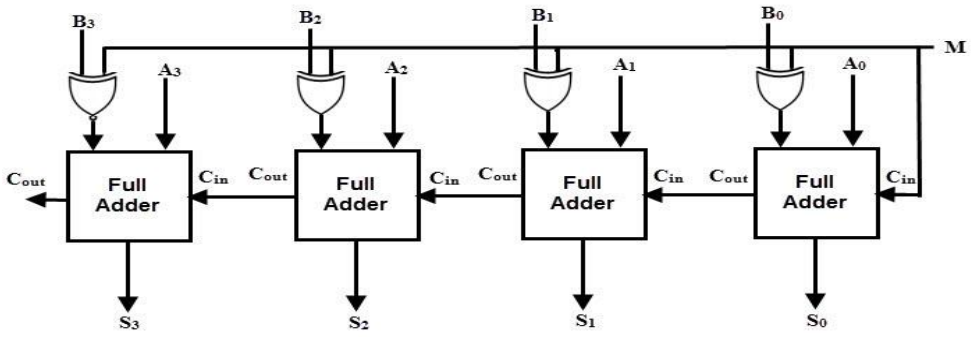
Figure 2.5 shows the interconnections of four full-adders (FA) to provide a 4-bit binary adder. The augend bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the low-order bit. The carries are connected in a chain through the full-adders. The input carry to the binary adder is c0 and the output carry is C4. The S outputs of the full-adders generate the required sum bits.

An n-bit binary adder requires n full-adders. The output carry from each full-adder is connected to the input carry of the next-high-order full-adder. The n data bits for the A inputs come from one register (such as R1), and the n data bits for the B inputs come from another register (such as R2). The sum can be transferred to a third register or to one of the source registers (R1 or R2) replacing its previous contents.

Binary Adder-Subtractor

The addition and subtraction operations can be combined into one common circuit by including an exclusive-OR gate with each full-adder. A 4-bit adder-subtractor circuit is shown in fig.2.6. The mode input M controls the operation. When M = 0 the circuit is adder and when M=1 the circuit becomes a subtractor, Each exclusive-OR gate receives input M and one of the inputs of B. When M = 0, We have B⊕0 =B. The full-address receive the value of B, the input carry is 0 and the circuit performs A plus B. When M=1 we have B ⊕ 1 = B' and Co = 1. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the

Figure 2.6 4-bit adder – subtractor



2's complement of B. for unsigned numbers, this gives A- B if A≥ B or the 2' s complement of (B-A) if A< B. For signed numbers, the result is A- B provided that there is no overflow.

2.5 Logic Microoperations

Logic microoperations specify binary operations for strings of bits stored in registers. These operations consider each bit of the register separately and treat them as binary variables. For example, the exclusive- OR microoperation with the contents of two registers R1 and R2 is symbolized by the statement.

P: R1 ← R1⊕ R2

It specifies a logic microoperation to be executed on the individual bits of the registers provided that the control variable P = 1. As a numerical example, assume that each register has four bits. Let the content of R1 be 1010 and the content of R2 be 1100. The exclusive-OR microoperation stated above symbolizes the following logic computation.

1010 Content of R1
1100 Content of R2
0110 Content of R1 after P + 1

The content of R1, after the execution of the microoperation, is equal to the bit-by – bit exclusive-OR operation on pairs of bits in R2 and previous values of R1. The logic microoperations are seldom used in scientific computations, but they are very useful for bit manipulation of binary data and for making logical decisions.

Special symbols will be adopted for the logic microoperations OR, AND and complement to distinguish them from the corresponding symbols used to express Boolean functions. The symbol v will be used to denote and OR microoperation and the symbol ∧ to denote and ANDmicrooperation. Another reason for adopting two sets of symbols is to able to distinguish the symbol + when used to symbolize an arithmetic plus from a logic OR operation. Although the + symbol has two meanings it will be possible to distinguish between them by noting where the symbol occurs. When the symbol + occurs in a microoperation, it will denote and arithmetic plus. When it occurs in a control (Or Boolean) function it will denote an OR operation. We will never use it to symbolize an OR microoperation. For example in the statement.

P + Q; R1 ← R2 + R3, R4 ← R5 v R6

The + Between P and Q is an OR operation between two binary variables of a control function. The + between R2 and R3 specifies an add microoperation. The OR microoperation is designated by the symbol v between registers R5 and R6.

List of Logic Microoperations

There are 16 different logic operations that can be performed with two binary variables. They can be determined from all possible tables obtained with two binary variables as shown in Table 2-4. In this table, each of the 16 columns F0 through F15 represents a truth table of one possible Boolean function for the two variables x and y. Note that the functions are determined from the 16 binary combinations that can be assigned to F.

Table 2-4 Truth Table for 16 functions of two Variables

x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

The 16 Boolean functions of two variables x and y are expressed in algebraic from in the first column of Table 2-4. The 16 logic microoperations are derived from these functions by replacing variable x by the binary content of register A and variable y by the binary content of register B.

The logic microoperations listed in the second column represent a relationship between the binary content of two registers A and B

Hardware Implementation

The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers to perform the required logic function.

Although there are 16 logic microoperations most computers use only four – AND, OR, XOR (exclusive-OR) and complement from which all others can be derived.

In a typical application register A is a processor register and the bits of register B constitute a logic operand extracted from memory and placed in register.

The selective-set operation sets to 1 the bits in register A where there are corresponding 1's in register B. It does not affect bit positions that have 0's in B. the following numerical example clarifies this operation:

```
1010   A before
1100   B (logic operand)
----
1110   A after
```

The two leftmost bits of B are 1's, so the corresponding bits of A are set to 1. One of these two bits was already set and the other has been changed from 0 to 1. The two bits of A with corresponding 0's in B remain unchanged.

The selective complement operation complements bits in A where there are corresponding 1'a in B. It does not affect bit positions that have 0's in B. for example:

```
1010   A before
1100   B (logic operand)
----
0110   A after
```

Again the two leftmost bits of B are 1s, so the corresponding bits of A are complemented.

One can deduce that the selective-complement operation is just an exclusive-OR microoperation. Therefore the exclusive-OR microoperation can be used to selectively complement bits of a register.

The selective-clear operation clears to 0 the bits in A only where there are corresponding 1's in B. for example:

```
1010   A before
1100   B (logic operand)
----
0010   A after
```

Again the two leftmost bits of B are 1's, so the corresponding bits of A are cleared to 0. One can deduce that the Boolean operation performed on the individual bits is AB'.

Again the two leftmost bits of B are 1's, so the corresponding bits of A are cleared to 0. One can deduce that the Boolean operation performed on the individual bits is AB. The corresponding logic microoperation is

$$A \leftarrow A \wedge \bar{B}$$

The mask operation is similar to the selective-clear operation except that the bits of A are cleared only where there are corresponding 0's. in B. The mask operation is an AND micro operation as seen from the following numerical example:

```
1010   A before
1100   B (logic operand)
----
1000   A after Masking
```

The two rightmost bits of A are cleared because the corresponding bits of B are 0s. The two leftmost bits are left unchanged because the corresponding bits of B are 1 s.

The insert operation that executes a new value in to a group of bits. This is done by first masking the bits and then ORing them with the required value. For example, suppose that an A register contains eight bits, 0110 1010. To replace the four leftmost bits by the value 1001 we first mask the four unwanted bits;

0110 1010	A before
0000 1111	B (logic operand)
0000 1010	A after Masking

And then insert the new value:

0000 1010	A before
1001 0000	B (insert)
1001 1010	A after insertion

The mask operation is an AND microoperation and the insert operation is and OR microoperation.

The clear operation compares the words in A and B and produces an all 0's result if the two numbers are equal. This operation is achieved by an exclusive-OR microoperation as shown by the following example:

1010	A
1010	B
0000	$A \leftarrow A \oplus B$

When A and B are equal, the two corresponding bits are either both 0 or both 1. In either case the exclusive-OR operation produces a 0. The all – 0`s result is then checked to determine if the two numbers were equal.

2.6. Shift Microoperations

Shift microoperations are used for serial transfer of data. They are also used in conjunction with arithmetic, logic and other data-processing operations.

A logical shift is one that transfers 0 through the serial input. We will adopt the symbols and shr for logical shift-left and shift-right microoperations. For example:

R1←shI R1

R2 ←shr R2

are two microoperations that specify a 1-bit shift to the left of the content of register R1 and a 1-bit shift to the right of the content of register R2. The register symbol must be the same on both sides of the arrow. The bit transferred to the end position through the serial input is assumed to be 0 during a logical shift.

The circular shift (also known a rotate operation) circulates the bits of the register around the two ends without loss of information. This is accomplished by connecting the serial output of the shift register to its serial input. We will use the symbols cir for the circular shift left and right respectively.

An arithmetic shift is a microoperation that shifts a signed binary number to the left or right. An arithmetic shift-left multiplies a signed binary number by 2. An arithmetic shift

right divides the number by 2. Arithmetic shifts must have the sign bit unchanged because the sign of the number remains the same when it is multiplied or divided by 2. The leftmost bit in a register holds the sign bit and the remaining bits hold the number. The sign bit is 0 for positive and 1 for negative. Negative numbers are in 2' s complement form

2.7. Arithmetic Logic shift Unit

Instead of having individual registers performing the microoperations directly computer systems employ a number of storage registers connected to a common operational unit called an arithmetic logic unit abbreviated ALU. To perform a microoperation, the contents of specified registers are placed in the inputs of the common ALU. The ALU performs an operation and the result of the operation is then transferred to a destination register. The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period. The shift microoperations are often performed in a separate unit but sometimes the shift unit is made part of the overall ALU.

One stage of an arithmetic logic shift unit is shown in Fig. 2.7. The subscript I designates a typical stage. Inputs Ai and Bi are applied to both the arithmetic and logic units. A particular microoperation is selected with inputs S1 and S0 . A 4 x 1 multiplexer at the output chooses between an arithmetic output in Ei and a logic output in Hi. The data in the multiplexer are selected with inputs S3 and S2. The other two data inputs to the multiplexer receive inputs A_{i-1} for the shift-right operation and $A_{i+1}$for the shift-left operation. The diagram shows just one typical stage. The circuit of Fig.2.7 must be repeated n times for and n-bit ALU. The output carry C_{i+1} of a given arithmetic stage must be connected to the input carry Ci of the next stage in sequence.

Figure 2.7 One stage of arithmetic logic shift unit

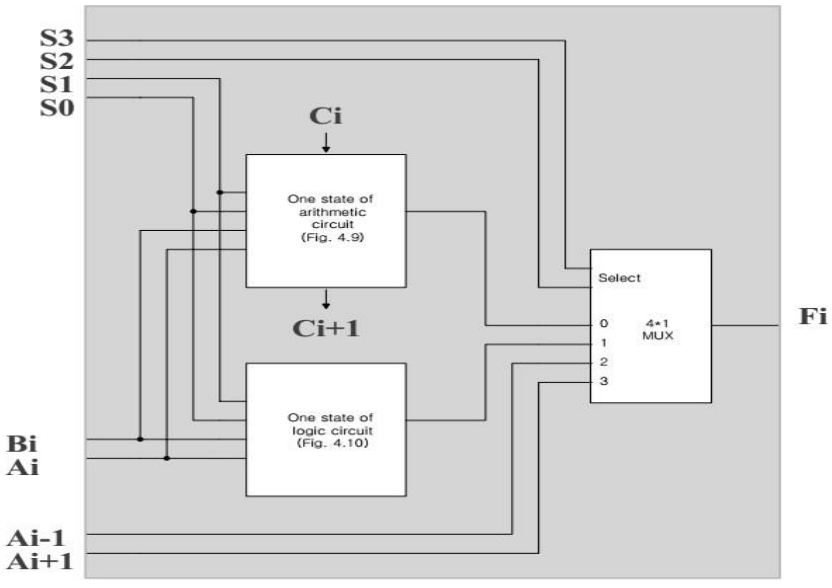


fig: one stage of arithmetic logic shift unit

The input carry to the first stage is the input carry C_{in} which provides a selection variable for the arithmetic operations.

The circuit whose one stage is specified in Fig.2.7 provides eight arithmetic operation four logic operations and two shift operations. Each operation is selected with the five variables S3, S2, S1, S0 and C_{in}. The input carry C_{in} is used for selecting an arithmetic operation only.

UNIT II

3. BASIC COMPUTER ORGANIZATION AND DESIGN

3.1 Instruction Codes

The organization of the computer is defined by its internal registers, the timing and control structure, and the set of instructions that is uses.

The internal organization of a digital system is defined by the sequence of microoperations it performs on data stored in its registers. The general- purpose digital computer is capable of executing various microoperations and in addition, can be instructed as to what specific sequence of operations it must perform. The user of a computer can control the process by means of a program. A program is a set of instructions that specify the operations, operations, and the sequence by which processing has to occur. The data processing task may be altered by specifying a new program with different instructions or specifying the same instructions with different data.

A computer instruction is a binary code that specifies a sequence of microoperations for the computer. Instruction codes together with data are stored in memory. The computer reads each instruction from memory and places it in a control register. The control then interprets the binary code of the instruction and proceeds to execute it by issuing a sequence of microoperations. Every computer has its own unique instruction set.

An instruction code is a group of bits that instruct the computer to perform a specific operation. It is usually divided into parts each having its own particular interpretation. The most basic part of an instruction code is its operation part. The operation code of an instruction is a group of bits that define such operations as add, subtract, multiply, shift, and complement. The number of bits required for the operation code of an instruction depends on the total number of operations available in the compute. The operation code must consist of at least n bits for a given 2^n (or less) distinct operations.

An operation code is a part of an instruction stored in computer memory. It is a binary code that tells the computer to perform a specific operation. The control unit receives the instruction from memory and interprets the operation code bits. It then issues a sequence of control signals to initiate microoperations in internal computer registers. For every operation code, the control issues a sequence of microoperations needed for the hard-ware implementation of the specified operation.

The Operation part of an instruction code specifies the operation to be performed. This operation must be performed on some data stored in processor register or in memory. An instruction code must therefore specify not only the operation but also the registers or memory word where the result is to be stored.

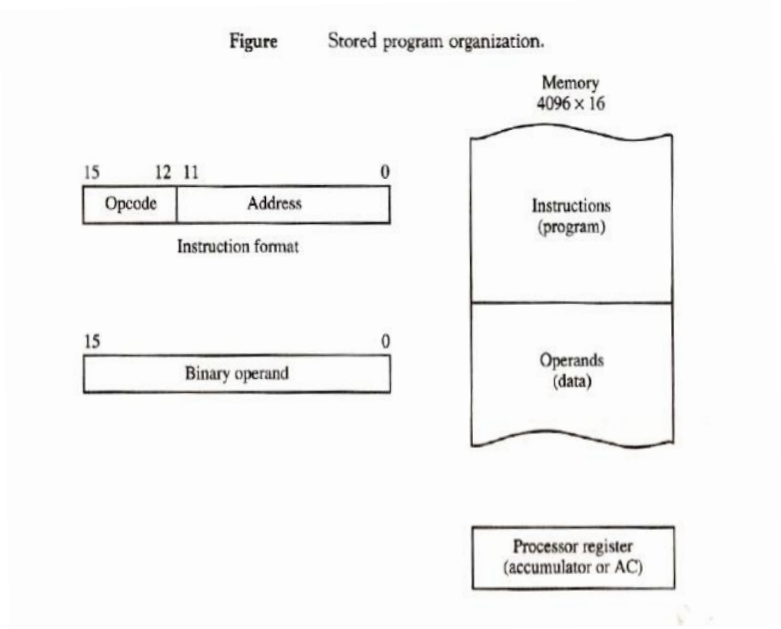
Stored program Organization

The simplest way to organize a computer is to have one processor register and instruction code format with two parts. The first part specifies the operation to be performed and the second specifies and address. The memory address tells the control where to find an operand in memory. This operand is read from memory and used as the data to be operated on together with the data stored in the processor register.

Figure 3.1 depicts this type of organization. Instructions are stored in one section of memory and a data in another. For a memory unit with 4096 words we need 12 bits to specify an address since $2^{12} = 4096$. If we store each instruction code in one 16-bit memory

word, we have available four bits for the operation code (abbreviated as opcode) to specify one out of 16 possible operation and 12 bits to specify the address of an operand. The control reads a 16-bit instruction from the program portion of memory. It uses the 12 bit address part of the instruction to read and 16 bit operand from the data portion of memory. It then executes the operation specified by the operation code.

Figure 3.1 Stored program organization



Computer that have a single processor register usually assign to it the name accumulator and label it AC. The operation is performed with the memory operand and the content of AC.

If an operation in an instruction code does not need and operand from memory, the rest of the bits in the instruction can be used for other purposes. For example, operations such as clear AC, complement AC, and increment AC operate on data stored in the AC register.

Indirect Address

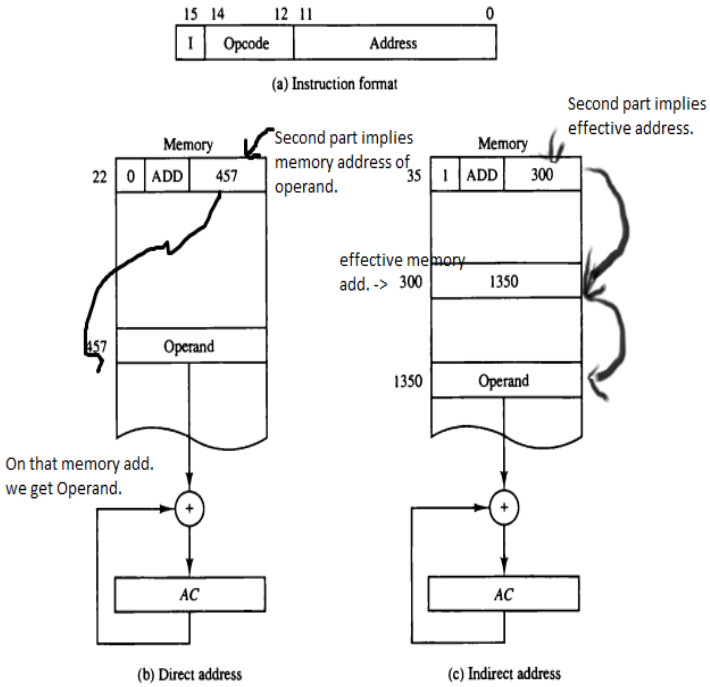
It is sometimes convenient to use the address bits of an instruction code not as an address but as the actual operand. When the second part of an operand, the instruction is said to have immediate operand. When the second part specifies the address of an operand, the instruction is said to have a direct address. When the bits in the second part of the instruction designate an address of a memory word in which the address of the operand is found. One bit of the instruction code can be used to distinguish between a direct and an indirect address.

Consider the instruction code format shown in fig.3.2.a..It consists of a 3-bit operation code, a 12-bit address, and an indirect address mode bit designated by I. the mode bit is 0 for a direct and 1 for an indirect address.

A direct address instruction is shown in fig. 3.2.b. It is placed in address 22 in memory. The I bit is 0, so the instruction is recognized as a direct address instruction. The opcode specifies an ADD instruction, and the address part is the binary equivalent of 457. The control finds the operand in memory at address 457 and adds it to the content of AC. The instruction in address 35 shown in fig.3.2.c.has a mode bit I = 1. Therefore, it is

recognized as an indirect address instruction. The address part is the binary equivalent of 300. The control goes to address 300 to find the address instruction needs two references to memory to fetch and operand. The address of the operand in this case is 1350. The operand found in address 1350 is then added to the content of AC.

Figure 3.2 Demonstration of direct and indirect address



3.2 Computer Registers

The memory unit has a capacity of 4096 words and each word contains 16 bits. Twelve bits of an instruction word are needed to specify the address of an operand. This leaves three bits for the operation part of the instruction and a bit to specify a direct or indirect address. The data register (DR) holds the operand read from memory. The accumulator (AC) register is a general-purpose processing register. The instruction read from memory is placed in the instruction register (IR). The temporary register (TR) is used for holding temporary data during the processing

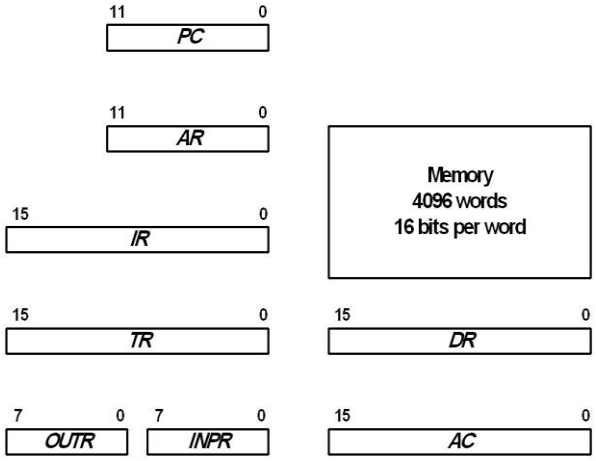
Table 3-1 List of registers for the Basic Computer

Register symbol	Number of bits	Register Name	Function
DR	16	Data register	Holds memory operand
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
PC	12	Program counter	Holds address of instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds input character
OUTR	8	Output register	Holds out put character

The memory address register (AR) has 12 bits since this is the width of a memory address, The program counter (PC) also has 12 bits and it holds the address of the next instruction to be read from memory after the current instruction is executed. The PC goes through a counting sequence and causes the computer to read sequential instructions previously stored in memory. Instruction words are read and executed in sequence unless a branch instruction is transferred to PC to become the address of the next instruction. To read an instruction, the content of PC is taken as the address for memory and a memory read cycle is initiated. PC is then incremented by one, so it holds the address of the next instruction in sequence.

Two registers are used for input and output. The input register (INPR) receives an 8-bit character from an input device. The output register (OUTR) holds an 8-bit character for an output device.

Figure 3. 3 Basic computer registers and memory



Common Bus System

The basic computer has eight registers, a memory unit, and a control unit. Paths must be provided to transfer information from one register to another and between memory and register. The number of wires will be excessive if connections are made between the outputs of each register and the inputs of the other register. A more efficient scheme for transferring information in a system with many registers is to use a common bus.

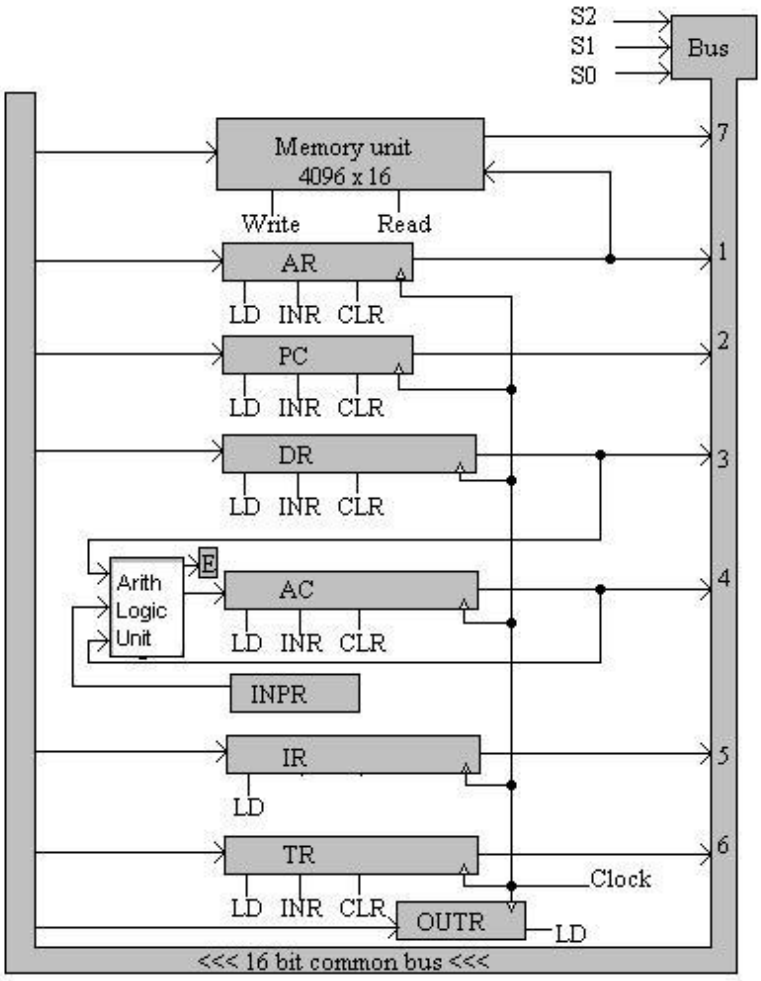
The connection of the registers and memory of the basic computer to a common bus system is shown in fig.3.4.

The outputs of seven registers and memory are connected to the common bus. The specific output that is selected for the bus lines at any given time is determined from the binary value of the selection variables S2, S1 and S0. The number along each output shows the decimal equivalent of the required binary selection. For example, the number along the output of DR is 3. The 16-bit outputs of DR are placed on the bus lines when S2S1S0 = 011 since this is the binary of each register and the data inputs of the memory. The particular register whose LD (Load) input is enabled receives the data from the bus during the next clock pulse transition. The memory receives the contents of the bus when its write input is activated, The memory places its 16-bit output onto the bus when the read input is activated and S2S1S0 = 111.

Four registers, DR, AC, IR, and TR, have 16 bits each. Two registers, AR and PC, have 12 bits each since they hold a memory address. When the contents of AR or PC are

applied to the 16-bit common bus, the four most significant bits are set to 0's. When AR or PC receive information from the bus, only the 12 least significant bits are transferred into the register.

Figure 3.4 Common Bus system



The input register INPR and the output register OUTR have 8 bits each and communicate with the eight least significant bits in the bus. INPR is connected to provide information to the bus but OUTR can only receive information from the bus. This is because INPR receives a character form and input device which is then transferred to AC. OUTR receives a character form AC and delivers it to an output device. There is on transfer from OUTR to any of the other register.

The 16 lines of the common bus receive information from six registers and the memory unit. The bus lines are connected to the inputs of six registers and the memory. Five registers have three control inputs: LD (load), INR (increment), and CLR (clear).

The input data and output of the memory are connected to the common bus, but the memory address is connected to AR. Therefore, AR must always be used to specify a memory address. By using a single register for the address, we eliminate the need for an address bus that would have been needed otherwise. The content of any register can be specified for the memory data input during a write operation. Similarly, any register can receive the data from memory after a read operation except AC.

The 16 inputs of AC come from an adder and logic circuit. This circuit has three sets of inputs. One set of 16-bit inputs come from the outputs of AC. They are used to implement register microoperations such as complement AC and shift AC. Another set of 16-bit inputs come from the data register DR. The inputs from DR and AC are used for arithmetic and logic microoperations, such as add DR to AC or AND DR to AC. The result of an addition is transferred to AC and the end carry-out of the addition is transferred to flip-flop. A third set of 8-bit inputs come from the input register INPR.

Note that the content of any register can be applied onto the bus and an operation can be performed in the adder and logic circuit during the same clock cycle. The clock transition at the end of the cycle transfers the content of the bus into the designated destination register and the output of the adder and logic circuit into AC. For example, the two microoperations

$$DR \leftarrow AC \text{ and } AC \leftarrow DR$$

can be executed at the same time. This can be done by placing the content of AC on the bus (with S2S1S0 = 100) enabling the LD (load) input of DR, transferring the content of DR through the address and logic circuit into AC, and enabling the LD (load) input of AC, all during the same clock cycle

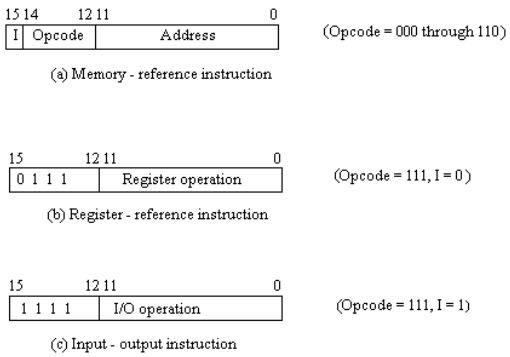
3.3 Computer Instructions

The basic computer has three instruction code formats. Each format has 16 bits. The operation code (opcode) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered. A memory-reference instruction uses 12 bits to specify and address and one bit to specify the addressing mode I. I is equal to 0 for direct address and to 1 for indirect address

The register reference instructions are recognized by the operation code 111 with a 0 in the leftmost bit (bit 15) of the instruction. A register-reference instruction specifies an operation on or a test of the AC register. An operand from memory is not needed; therefore, the other 12 bits are used to specify the operation or test to be executed. Similarly, an input-output instruction does not need a reference to memory and is recognized by the operation code 111 with a 1 in the leftmost bit of the instruction. The remaining 12 bits are used to specify the type of input-output operation or test performed.

The type of instruction is recognized by the computer control from the four bits in positions 12 through 15 of the instruction. If the three opcode bits in positions 12 through 14 are not equal to 111, the instruction is a memory-reference type and the bit in position 15 is taken as the addressing mode I. If the 3-bit opcode is equal to 111, control then inspects the bit in position 15. If this bit is 0, the instruction is a register o reference type.

Figure 3.5 Basic computer Instruction formats



If the bit is 1, the instruction is an input-output type. Note that the bit in position 15 of the instruction code is designated by the symbol I but is not used as a mode bit when the operation code is equal to 111,

The instructions for the computer are listed in Table 3-2.

Table3-2 Basic Computer Instruction

Symbol	Hex Code I = 0	Hex Code I = 1	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load AC from memory
STA	3xxx	Bxxx	Store content of AC into memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instr. if AC is positive
SNA	7008		Skip next instr. if AC is negative
SZA	7004		Skip next instr. if AC is zero
SZE	7002		Skip next instr. if E is zero
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

Instruction Set Completeness

A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function that is known to be computable. The set of instructions are said to be complete if the computer includes a sufficient number of instructions in each of the following categories.

1. Arithmetic, logical and shift instructions
2. Instructions for moving information to and from memory and processor register
3. Program control instructions together with instructions that check status conditions\
4. Input and output instructions

Arithmetic, logical, and shift instructions provide computational capabilities for processing the type of data that the user may wish to employ. The bulk of the binary information in a digital computer is stored in memory, but all computations are done in processor registers. Therefore, the user must have the capability of moving information between these two units. Decision making capabilities are an important aspect of digital computers. For example, two numbers can be compared, and if the first is greater than the second, it may be necessary to proceed differently than if the second is greater than the first. Program control instructions such as branch instructions are used to change the sequence in which the program is executed. Input and output instructions are needed for communication between the computer and the user. Programs and data must be transferred into memory and results of computations must be transferred back to the user.

There is one arithmetic instruction, ADD, and two related instructions, complement AC (CMA) and increment AC (INC). With these three instructions we can add and subtract binary numbers when negative numbers are in signed-2s complement representation.

The circulate instructions, CIR and CIL, can be used for arithmetic shifts as well as any other type of shifts desired

Multiplication and division can be performed using addition, subtraction, and clear AC (CLA). The AND and complement provide a NAND operation. Moving information from memory to AC is accomplished with the load AC (LDA) instruction. Storing information from AC into memory is done with the store.

3.4 Timing and control

The timing for all registers in the basic computer is controlled by a master clock generator. The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit. The clock pulses do not change the state of a register unless the register is enabled by a control signal. The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and microoperations for the accumulator.

There are two major types of control organization, hardwired control and microprogrammed control. In the hardwired organization, the control logic is implemented with gates, flip-flops, decoders, and other digital circuits. It has the advantage that it can be optimized to produce a fast mode of operation. In the microprogrammed organization, the control information is stored in a control memory. The control memory is programmed to initiate the required sequence of microoperations. A hardwired control requires changes in the wiring among the various components if the design has to be modified or changed. In the microprogrammed control, any required changes or modifications can be done by updating the microprogram in control memory.

The block Diagram of the control unit is shown in fig 3.6. It consists of two decoders, a sequence counter, and a number of control logic gates. An instruction read from memory is placed in the instruction register (IR)

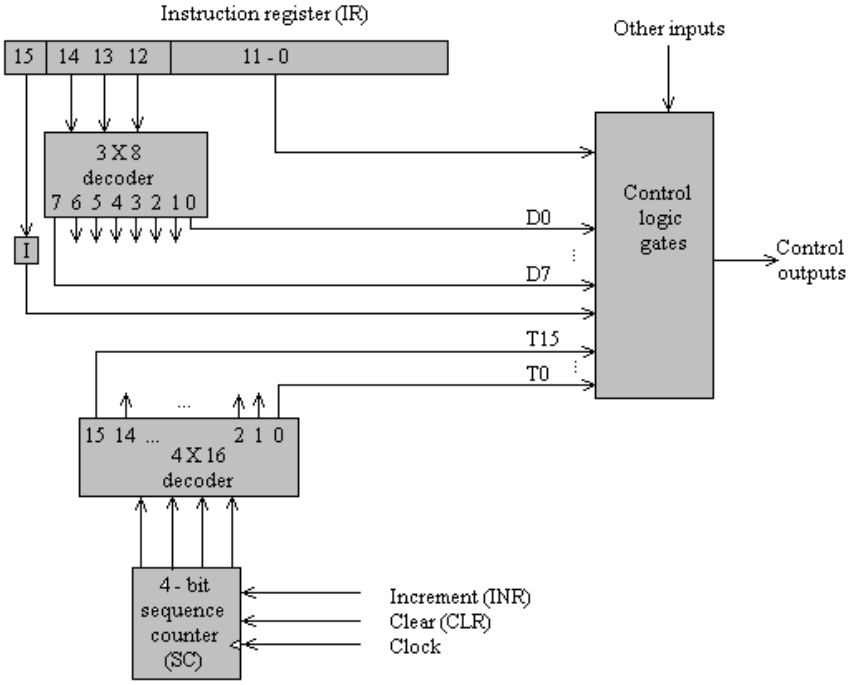
The instruction register is shown again in Fig, 3.6 where it is divided into three parts the I bit, the operation code and bits 0 through 11. The operation code in bits12 through 14 are decoded with a 3 x 8 decoder. The eight outputs of the decoder are designated by the binary value of the corresponding operation code. Bit 0 through 11 are applied to the control logic gates. The 4bit sequence counter can count in binary from 0 through 15. The outputs of the counter are decoded into 16 timing signals T_0 through T_{15}.The sequence counter SC can be incremented or cleared synchronously.

Most of the time the counter is incremented to provide the sequence of timing signals out of the 4 x 16 decoder. Once in awhile, the counter is cleared to 0, causing the next active timing signal to be T_0. As an example, consider the case where SC is incremented to provide timing signals T_0, T_1, T_2, T_3, and T_4 in sequence. At time T_4 SC is cleared to 0 if decoder output D_3 is active. This is expressed symbolically by the statement

D_3T_4: SC ← 0

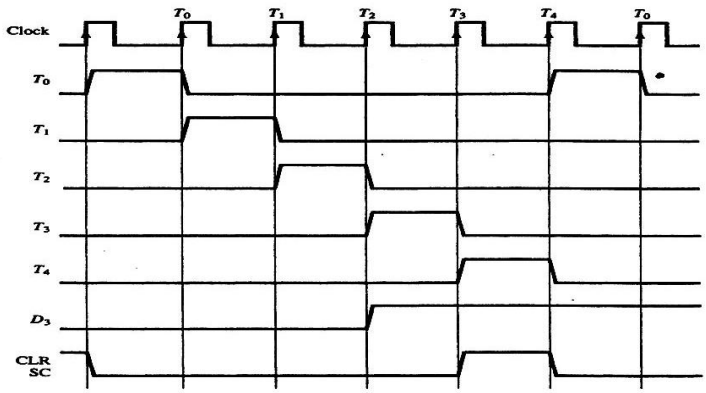
The timing diagram of Fig. 3.7 shows the time relationship of the control signals. The sequence counter SC responds to the positive transition of the clock. Initially, the CLR input of SC is active. The first positive transition of the Clock clears SC to 0, which in turn activates the timing signal T_0 out of the decoder.

Figure 3.6 Control unit of basic computer



T_0 is active during one clock cycle. The positive clock transition labeled T_0 in the diagram will trigger only those registers whose control inputs are connected to timing signal T_0. SC is incremented with every positive clock transition, unless its CLR input is active. This produces the sequence to timing signals T_0, T_1, T_2,T_3, T_4, and so on as shown in the diagram. If SC is not cleared the timing signals will continue with T_5, T_6 up to T_{15} and back to T_0.

Figure 3.7 Example of control timing signals



The last three waveforms in fig.3.7 show how SC is cleared when D_3T_4= 1. Output D_3 from the operation decoder becomes active at the end of timing signal T_2. When timing signal T_4 becomes active the output of the AND gate that implements the control function D_3T_4 becomes active. This signal is applied to the CLR input of SC. On the next positive clock transition the counter is cleared to 0. This causes the timing signal T_0 to become active instead of T_5 that would have been active if SC were incremented instead of cleared.

A memory read or write cycle will be initiated with the rising edge of a timing signal. It will be assumed that a memory cycle time is less than the clock cycle time. According to this assumption, a memory read or write cycle initiated by a timing signal will be completed by the time the next clock goes through its positive transition. The clock transition will then be used to load the memory word into a register. This timing relationship is not valid in many computers because the memory cycle time is usually longer than the processor clock cycle. In such a case it is necessary to provide wait cycles in the processor until the memory word is available.

3.5 Instruction Cycle

The program is executed in the computer by going through a cycle for each instruction. Each instruction cycle in turn is subdivided into a sequence of subcycles or phases. In the basic computer each instruction cycle consists of the following phases:

1. Fetch an instruction from memory
2. Decode the instruction
3. Read the effective address from memory if the instruction has a indirect address
4. Execute the instruction

Upon the completion of step 4, the control goes back to step 1 to fetch, decode, and execute the next instruction, This process continues indefinitely unless a HALT instruction is encountered.

Fetch and decode

Initially, the program counter PC is loaded with the address of the first instruction in the program. The sequence counter SC is cleared to 0, providing a decoded timing signal T_0,T_1,T_2, and so on. The microoperations for the fetch and decode phases can be specified by the following register transfer statements.

T_0: AR←PC

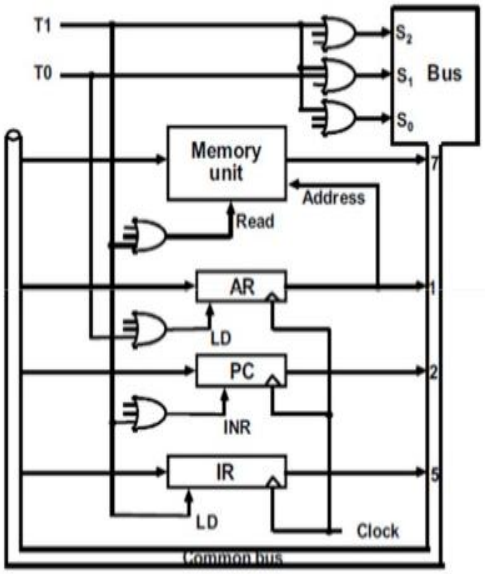
T_1: IR←M [AR] PC←PC +1

T_2: D_0, ….D_7←decode IR (12-14) AR←IR (0-11) 1←IR(15)

Since only AR is connected to the address inputs of memory, it is necessary to transfer the address from PC to AR during the clock transition associated with timing signal T_0. The instruction read from memory is then placed in the instruction register IR with the clock transition associated with timing signal T_1. At the same time, PC is incremented by one to prepare it for the address of the next instruction in the program. At time T2, the operation code in IR is decoded, the indirect bit is transferred to flip-flop I, and the address part of the instruction is transferred to AR. The sequence counter SC incremented after each clock pulse to produce the sequence T_0, T_1 and T_2.

Figure 3.8 shows how the first two register transfer statements are implemented in the bus system. To provide the data path for the transfer of PC to AR we must apply timing signal T_0 to achieve the following connection:

Figure 3.8 Register transfer for the fetch phase



1. Place the content of PC onto the bus by making the bus selection inputs $S_2S_1S_0$ equal to 010.
2. Transfer the content of the bus to AR by enabling the LD input of AR.

The next clock transition initiates the transfer from PC to AR since T_0=1. In order to implement the second statement

T_1: IR←M (AR) PC←PC + 1

It is necessary to use timing signal T_1 to provide the following connections in the bus system.

1. Enable the read input of memory.
2. Place the content of memory onto the bus by making $S_2S_1S_0$ = 111.
3. Transfer the content of the bus to IR by enabling the LD input of IR.
4. Increment PC by enabling the INR input PC.

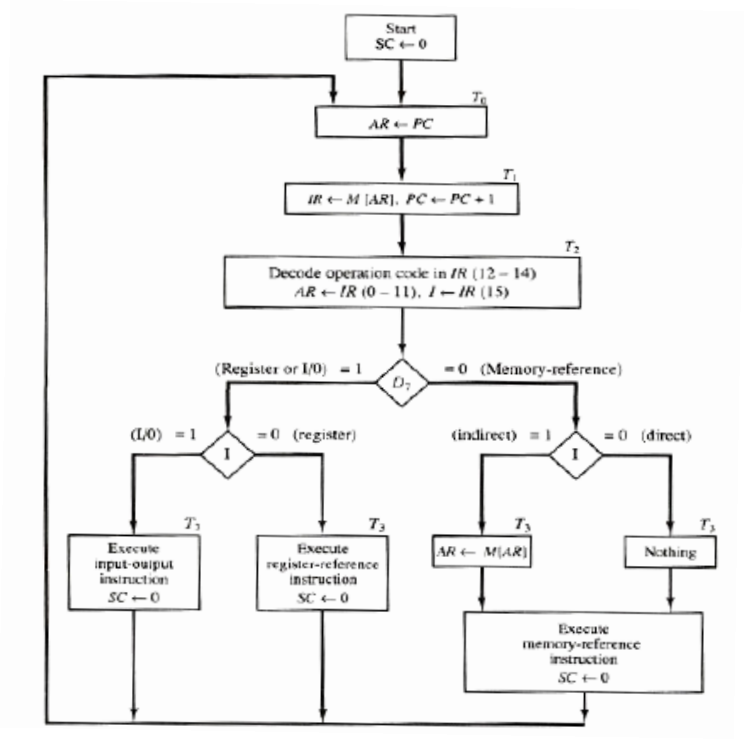
The next clock transition initiates the read and increment operations since T_1 = 1. Figure 3.8 duplicates a portion of the bus system and shows how T_0 and T_1 are connected to the control inputs of the registers, the memory, and the bus selection inputs. Multiple input OR gates are included in the diagram because there are other control functions that will initiate similar operations.

Determine the Type of Instruction

The timing signal that is active after the decoding is T3. During time T3 the control unit determines the type of instruction that was just read from memory. The flowchart of fig. 3.9 presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding.

We determine that if D7 =1, the instruction must be a register-reference or input output type. If D_7 = 0, the operation code must be one of the other seven values 000 through 110, specifying a memory reference instruction.

Figure 3.9 Flowchart for instruction cycle



Control then inspects the value of the first bit of the instruction which is now available in flip-flop I. If $D_7 = 0$ and I = 1, we have a memory reference instruction with an indirect address. It is then necessary to read the effective address from memory. The microoperation for the indirect address condition can be symbolized by the register transfer statement

$$AR \leftarrow M(AR)$$

Initially, AR holds the address part of the instruction. This address is used during the memory read operation. The word at the address given by AR is read from memory and placed on the common bus. The LD input of AR is then enabled to receive the indirect address that resided in the 12 least significant bits of the memory word.

The three instruction types are subdivided into four separate paths. The selected operation is activated with the clock transition associated with timing signal T_3. This can be symbolized as follows:

D'_7IT_3: AR←M (AR)
$D'_7I'T_3$: Nothing
$D_7I'T_3$: Execute a register reference instruction
D_7IT_3: Execute an input output instruction

Register-Reference Instructions

Register-reference instructions are recognized by the control when $D_7 = 1$ and I = 0. These instructions use bits 0 through 11 of the instruction code to specify one of 12 instructions. These 12 bits are available in IR(0-11). They were also transferred to AR during time T_2

The control functions and microoperations for the register-reference instructions are listed in Table 3-3. These instructions are executed with the clock transition associated with

timing variable T_3. Each control function needs the Boolean relation $D_7I'T_3$ which we designate for convenience by the symbol. The control function is distinguished by one of the bits in IR(0-11). By assigning the symbol B, to bit i of IR, all control functions can be simply denoted by rBi .For example the instruction CLA has the hexadecimal code 7800 (see Table 2-2) Which gives the binary equivalent 0111 1000 0000 0000. The first bit is a zero and is equivalent to I′. The next three bits constitute the operation code and are recognized from decoder output D_7. The control function that initiates the microoperation for this instruction is $D^7I'T_3B_{11} – rB_{11}$. The execution of a register-reference instruction is completed at time T3. The sequence counter SC is cleared to 0 and the control goes back to fetch the next instruction with timing signal T_0.

The first seven register-reference instructions perform clear complement circular shift, and increment microoperation on the AC or E registers. The next four instructions cause a skip of the next instruction in sequence when a stated condition is satisfied. The skipping of the instruction is achieved by incrementing PC once again.

The condition control statements must be recognized as part of the control conditions. The AC is positive when the sign bit in AC (15) = 0, it is negative when AC(15) = 1. The content of AC is zero (AC=0) if all the flip-flops of the register are zero. The HLT instruction clears a start stop flip-flop S and stops the sequence counter from counting.

Table 3-3 Execution of register reference instructions

$D_7I'T_3 = r$ (common to all register-reference instructions)
$IR(i) = B_i$ [bit in IR(0–11) that specifies the operation]

	r:	$SC \leftarrow 0$	Clear SC
CLA	rB_{11}:	$AC \leftarrow 0$	Clear AC
CLE	rB_{10}:	$E \leftarrow 0$	Clear E
CMA	rB_9:	$AC \leftarrow \overline{AC}$	Complement AC
CME	rB_8:	$E \leftarrow \overline{E}$	Complement E
CIR	rB_7:	$AC \leftarrow$ shr AC, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow$ shl AC, $AC(0) \leftarrow E$, $E \leftarrow AC(15)$	Circulate left
INC	rB_5:	$AC \leftarrow AC + 1$	Increment AC
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$	Skip if positive
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$	Skip if negative
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1)$	Skip if AC zero
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$	Skip if E zero
HLT	rB_0:	$S \leftarrow 0$ (S is a start–stop flip-flop)	Halt computer

3.6 Input-Output and Interrupt

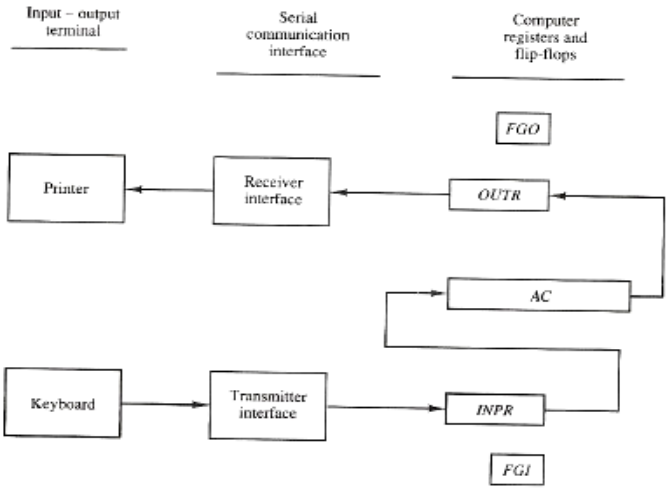
Instructions and data stored in memory must come from some input device. Computational results must be transmitted to the user through some output device. Commercial computers include many types of input and output devices

Input-Output Configuration

The terminal sends and receives serial information. Each quantity of information has eight bits of an alphanumeric code. The serial information from the keyboard is shifted into the input register INPR. The serial information for the printer is stored in the output register OUTR. These two registers communicate with a communication interface serially and with the AC in parallel. The input-output configuration is shown in Fig.3.10. The transmitter interface receives serial information from the keyboard and transmits it to INPR. The receiver interface receives information from OUTR and sends it to the printer serially.

The input register INPR consists of eight bits and holds an alphanumeric input information. The 1-bit input flag FGI is a flip-flop. The flag bit is set to 1 when new information is available in the input device and is cleared to 0 when the information is accepted by the computer.

Figure 3.10 Input output configuration



The flag is needed to synchronize the timing rate difference between the input device and the computer. The process of information transfer is as follows. Initially, the input flag FGI is cleared to 0. When a key is struck in the keyboard, an 8-bit alphanumeric code is shifted into INPR and the input flag FGI is set to 1. As long as the flag is set, the information in INPR cannot be changed by striking another key. The computer checks the flag bit, if it is 1 the information from INPR is transferred in parallel into AC and FGI is cleared to 0. Once the flag is cleared new information can be shifted into INPR by striking another key.

The output register OUTR works similarly but the direction of information flow is reversed. Initially, the output flag FGO is set to 1. The computer checks the flag bit, if it is 1, the information from AC is transferred in parallel to OUTR and FGO is cleared to 0. The output device accepts the coded information, prints the corresponding character and when the operation is completed, it sets FGO to 1. The computer does not load a new character into OUTR when FGO is 0 because this condition indicates that the output device is in the process of printing the character,

Input – Output Instructions

Input and output instructions are needed for transferring information to and from AC register, for checking the flag bits, and for controlling the interrupt facility. Input-output instructions have an operation code 1111 and are recognized by the control when D7 = 1 and I = 1. The remaining bits of the instruction specify the particular operation. The control functions and microoperations for the input-output instructions are listed in Table 3-4.

Table 3-4 Input-output instructions

$D_7IT_3 = p$
$IR(i) = B_i$, i = 6, ..., 11

	p:	SC ← 0	Clear SC
INP	pB_{11}:	AC(0-7) ← INPR, FGI ← 0	Input char. to AC
OUT	pB_{10}:	OUTR ← AC(0-7), FGO ← 0	Output char. from AC
SKI	pB_9:	if(FGI = 1) then (PC ← PC + 1)	Skip on input flag
SKO	pB_8:	if(FGO = 1) then (PC ← PC + 1)	Skip on output flag
ION	pB_7:	IEN ← 1	Interrupt enable on
IOF	pB_6:	IEN ← 0	Interrupt enable off

Program Interrupt

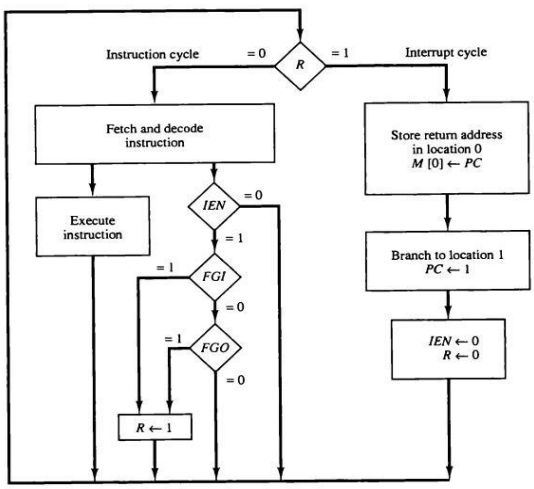
The process of communication just described is referred to as programmed control transfer. The computer keeps checking the flag bit, and when it finds it set, it initiates an information transfer. The difference of information flow rate between the computer and that of the input-output device makes this type of transfer inefficient. To see why this is inefficient, consider a computer that can transfer information at a maximum rate of 10 characters per second. This is equivalent to one character every 100,000μs. Two instructions are executed when the computer checks the flag bit and decides not to transfer the information. This means that at the maximum rate, the computer will check the flag 50000 times between each transfer. The computer is wasting time while checking the flag instead of doing some other useful processing task.

An alternative to the programmed controlled procedure is to let the external device inform the computer when it is ready for the transfer. In the meantime the computer can be busy with other tasks. This type of transfer uses the interrupt facility. While the computer is running a program, it does not check the flags. However, when a flag is set the computer is interrupted from proceeding with the current program and is informed of the fact that a flag has been set. The computer deviates from what it is doing to take care of the input or output transfer. It then returns to the current program to continue what it was doing before the interrupt.

The interrupt enable flip-flop IEN can be set and cleared with two instructions. When IEN is cleared to 0 (with the IOF instruction) the flags cannot interrupt the computer. When IEN is set to 1 (with the ION instruction) the computer can be interrupted. These two instructions provide the programmer with the capability of making a decision as to whether or not to use the interrupt facility.

In the Fig 3.11 an interrupt flip-flop R is included in the computer. When R = 0, the computer goes through an instruction cycle. During the execute phase of the instruction cycle IEN is checked by the control. If it is 0, it indicates that the programmer does not want to use the interrupt so control continues with the next instruction cycle. If IEN is 1, control checks the flag bits. If both flags are 0, it indicates that neither the input nor the output registers are ready for transfer of information. In this case, control continues with the next instruction cycle. If either flag is set to 1 while IEN = 1, flip-flop R is set to 1. At the end of the execute phase, control checks the value of R, and if it is equal to 1 it goes to an interrupt cycle instead of an instruction cycle.

Figure 3.11 Flowchart for interrupt cycle



The interrupt cycle is a hardware implementation of a branch and save return address operation. The return address available in PC is stored in a specific location where it can be found later when the program returns to the instruction at which it was interrupted. This location may be a processor register, a memory stack, or a specific memory location,

Interrupt Cycle

The interrupt cycle is initiate after the last execute phase if the interrupt flip-flop R is equal. to 1. This can happen with any clock transition except when timing signals T0. T1. Or T2 are active. The condition for setting flip-flop R to1 can be expressed with the following register transfer statement.

We now modify the fetch and decode phases of the instruction cycle. Instead of using only timing signals T0, T1 and T2.

We will AND the three timing signals with R so that the fetch and decode phases will be recognized from the three control functions RT_0, R T_1, and R T_2. The reason for this is that after the instruction is executed and SC is cleared to 0, the control will go through a fetch phase only if R = 0. Otherwise if R = 1 the control will go through and interrupt cycle. The interrupt cycle stores the return address (available in PC) into memory location), branches to memory location 1 and clears IEN, R and SC to). This can be done with the following sequence of microoperations.

RT_0: AR←0. TR ← PC

RT_1: M(AR) ← TR,PC ← 0

RT_2: PC ← PC + 1, IEN ← 0, R←0, SC←0

During the first timing signal AR is cleared to 0 and the content of PC is transferred to the temporary register TR. With the second timing signal the return address is stored in memory at location 0 and PC is cleared to 0. The third timing signal increment PC to 1, clears IEN and R and the control goes back to T_0 by clearing SC=0.

3.7Design of Basic computer

The basic computer consists of the following hardware components:

1. A memory unit with 4096 words of 16 bits each
2. Nine registers : AR, PC, DR, AC, IR, TR. OUTR, INPR, and SC
3. Seven flip-flop: I, S, E,R, IEN. FGI, and FGO
4. Two decoders: a 3 x 8 operation decoder and a 4 x 16 timing decoder
5. A 16 bit common bus.
6. Control logic gates.
7. Adder and logic circuit connected to the input of AC

Control Logic Gates

The inputs from the two decoders, the I flip-flop and bits 0 through 11 of IR. The other inputs to the control logic are; AC bits 0 through 15 to check if AC = 0 and to detect the sign bit in AC(15) DR bits 0 through 15 to check if DR – 0 and the values of the seven flip-flops.

The outputs of the control logic circuit are:

1. Signals to control the inputs of the nine registers
2. Signals to control the read and write inputs of memory
3. Signals to set, clear, or complement the flip-flops
4. Signals for S_2, S_1, and S_0 to select a register for the bus
5. Signals to control the AC adder and logic circuit

Control of Registers and Memory

The control inputs of the registers are LD (Load) INR (increment) and CLR (clear). Suppose that we want to derive the gate structure associated with the control inputs of AR. The statements that change the content of AR

$R'T_0$: AR←PC

$R'T_2$: AR←IR(0-11)

D'_7T_3: AR←M (AR)

RT_0: AR←0

D_5T_4: AR←AR + 1

Figure 3.12 Flowchart for computer operation

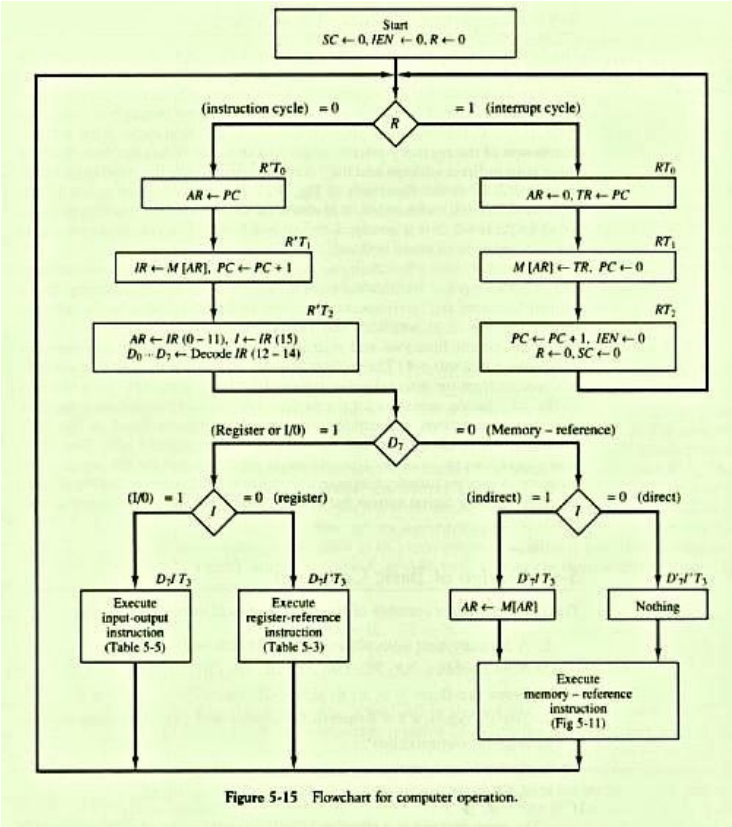


Figure 5-15 Flowchart for computer operation.

The first three statements specify transfer of information from a register or memory is placed on the bus and the content of the bus is transferred into AR by enabling its LD control input. The fourth statement clears AR to 0. The last statement clears AR to 0. The last statement increments AR by 1. The control functions can be combined into three Boolean expressions as follows:

$LD(AR) = R'T_0 + R'T_2 + D'_7T_3$

$CLR(AR) = RT_0$

$INR(AR) = D\%T_4$

Where LD(AR) is the load input of QR, CLR(AR) is the clear input of AR, and INR(AR) is the increment input of AR.

The read operation is recognized from the symbol ←M(AR)

$Read = RT_1 + D'_7T_3 + (D_0 + D_1 + D_2 + D_6)T_4$

The output of the logic gates that implement the Boolean expression above must be connected to the read input of memory.

Control of single Flip-flops

The control gates for the seven flip-flops a can be determined in a similar manner.

pB7: IEN ← 1

pB6: IEN←0

Where $p = D'_7 IT_3$ and B_7 and B_6 are bits 7 and 6 of IR respectively. Moreover, at the end of the interrupt cycle IEN is cleared to 0.

RT_2: IEN ←0

Control of Common Bus

The 16-bit common bus is controlled by the selection inputs S2,S1, and S0,. The decimal number shown with each bus input specifies the equivalent binary number that must be applied to the selection inputs in order to select the corresponding register.. Each binary number is associated with a Boolean variable X1 through X7 corresponding to the gate structure that must be active in order to select the register or memory for the bus.

UNIT III

4. CENTRAL PROCESSING UNIT

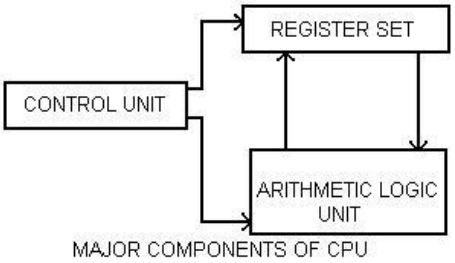
4.1 Introduction

The part of the computer that performs the bulk of data processing operations is called the central processing unit and is referred to as the CPU. The CPU is made up of three major parts, as shown in Fig. 4.1

The register set stores intermediate data used during the execution of the instructions. The arithmetic logic unit (ALU) performs the required microoperations for executing the instructions. The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.

The CPU performs a variety of functions dictated by the type of instructions that are incorporated in the computer.

Figure 4.1 Major components of CPU



4.2 General Register Organization

When a large number of registers are included in the CPU, it is most efficient to connect them through a common bus system. The registers communicate with each other not only for direct data transfers, but also while performing various microoperations. Hence it is necessary to provide a common unit that can perform all the arithmetic. Logic, and shift microoperations in the processor.

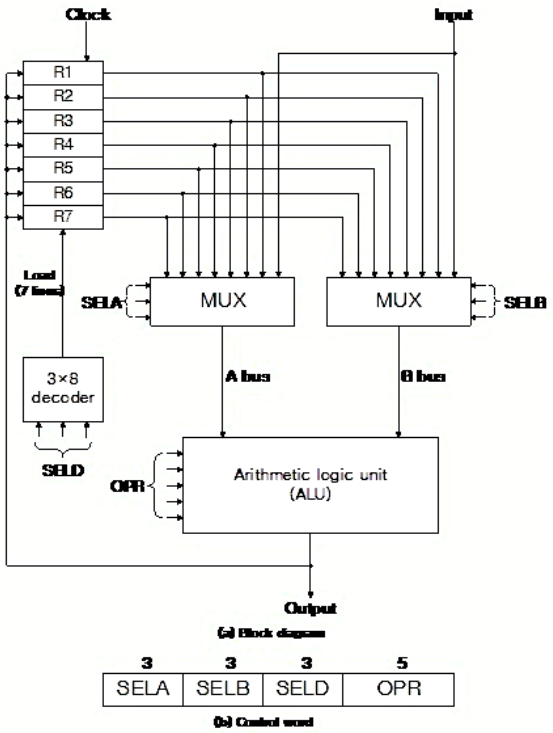
A bus organization for seven CPU registers is shown in Fig. 4.2. The output of each register is connected to two multiplexers (MUX) to form the two buses A and B. The selection lines in each multiplexer select one register or the input data for the particular bus. The A and B buses form the inputs to a common arithmetic logic unit (ALU) . The operation selected in the ALU determines the arithmetic or logic microoperation that is to be performed. The result of the microoperation is available for output data and also goes into the inputs of all the registers. The register that receives the information from the output bus is selected by a decoder. The decoder activates one of the register load inputs, thus providing a transfer path between the data in the output bus and the inputs of the selected destination register.

The control unit that operates the CPU bus system directs the information flow through the registers and ALU by selecting the various components in the system.

To perform the operation

$$R1 \leftarrow R2 + R3$$

Figure 4.2 Register set with common ALU



The control must provide binary selection variables to the following selector inputs:

1. MUX A selector (SELA): to place the content of R2 into bus A.
2. MUX B selector (SELB): to place the content of R3 into bus B.
3. ALU operation selector (OPR): to provide the arithmetic addition A + B.

Control word

There are 14 binary selection inputs in the unit and their combined value specifies a control word. The 14-bit control word is defined in "Fig. 4.2 (b). It consists of four fields. Three fields contain three bits each, and one field has five bits. The three bits of SELA select a source register for the A input of the ALU. The three bits of SELB select a register for the B input of the ALU. The three bits SELD select a destination register using the decoder and its seven load outputs. The five bits of OPR select one of the operations in the ALU. The 14-bit control word when applied to the selection inputs specify a particular microoperation

The encoding of the register selections is specified in Table 4-1. The 3-bit binary code listed in the first column of the table specifies the binary code for each of the three fields. The register selected by fields SELA, SELB, and SELD is the one whose decimal number is equivalent to the binary number in the code. When SELA or SELB is 000, the corresponding multiplexer selects the external input data. When SELD = 000, no destination register is selected but the contents of the output bus are available in the external output.

The ALU provides arithmetic and logic operations. In addition, the CPU must provide shift operations. The shifter may be placed in the input of the ALU to provide a preshift capability, or at the output of the ALU to provide post shifting capability. In some cases, the shift operations are included with the ALU.

Table 4-1 Encoding of Register Selection Fields

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

Table 4-2 Encoding of ALU Operations

OPR Select	Operation	Symbol
00000	Transfer A	TSFA
00001	Increment A	INCA
00010	ADD A + B	ADD
00101	Subtract A - B	SUB
00110	Decrement A	DECA
01000	AND A and B	AND
01010	OR A and B	OR
01100	XOR A and B	XOR
01110	Complement A	COMA
10000	Shift right A	SHRA
11000	Shift left A	SHLA

Examples of Microoperations

A control word of 14 bits is needed to specify a microoperation in the CPU. The control word for a given microoperation can be derived from the selection variables. For example, the subtract microoperation given by the statement

$$R1 \leftarrow R2 - R3$$

Specifies R2 for the A input of the ALU, R3 for the B input of the ALU, R1 for the destination register, and an ALU operation to subtract A – B. The binary control word for the subtract microoperationis 010 011 001 00101 and is obtained as follows:

Field:	SELA	SELB	SELD	OPR
Symbol:	R2	R3	R1	SUB
Control word:	010	011	001	00101

4.3 Stack Organization

A useful feature that is included in the CPU of most computers is a stack or last-in, first-out (LIFO) list. A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved. The operation of a stack can be compared to a stack of trays. The last tray placed on top of the stack is the first to be taken off.

The stack in digital computers is essentially a memory unit with an address register that can count only (after initial value is loaded into it). The register that holds the address for the stack is called a stack pointer (SP) because its value always points at the top item in the stack.

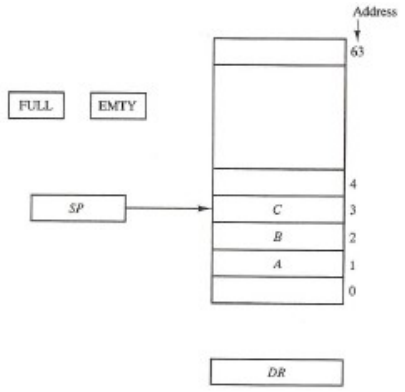
The two operations of a stack are the insertion and deletion of items. The operation of insertion is called push (or push-down) because it can be thought of as the result of pushing a

new item on top. The operation of deletion is called pop (or pop-up) because it can be thought of as the result of removing one item so that the stack pops up.

Register Stack

The stack pointer register SP contains a binary number whose value is equal to the address of the word that is currently on top of the stack. Three items are placed in the stack: A,B, and C, in that order. Item C is on top of the stack so that the content of SP is now 3. To remove the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of SP. Item B is now on top of the stack since SP holds address 2. To insert a new item, the stack pushed by incrementing SP and writing a word in the next-higher location in the stack. Note that item C has been read out but not physically removed. This does not matter because when the stack is pushed, a new item is written in its place.

Figure 4.3 Block diagram of 64-word stack



In a 64-word stack the stack pointer contains 6 bits because 2^6= 64. Since SP has only six bits, it cannot exceed a number greater than 63 (111111 in binary). When 63 is incremented by 1, the result 0 since 111111 + 1 = 1000000 in binary, but SP can accommodate only the six least significant bits. Similarly, when 000000 is decremented by 1, the result is 111111. The one-bit register FULL is set to 1 when the stack is full, and the one-bit register EMTY is set to 1 when the stack is empty of items. DR is the data register that holds the binary data to be written into or read out of the stack.

Initially, SP is cleared to 0, EMTY is set to 1, and FULL is cleared to 0, so that SP points to the word at address 0 and the stack is marked empty and not full. If the stack is not full (if FULL = 0), a new item is inserted with a push operation. The push operation is implemented with the following sequence of microoperations:

SP← SP + 1	Increment stack pointer
M[SP] ← DR	Write item on top of the stack
If (SP = 0) then (FULL ← 1)	Check if stack is full
EMTY ← 0	Mark the stack not empty

The stack pointer is incremented so that it points to the address of the next-higher word. A memory write operation inserts the word from DR into the top of the stack.

SP holds the address of the top of the stack and the M[SP] donates the memory word specified by the address 1. The last item is stored at address 0. If SP reaches 0, the stack is full of items, so FULL is set to 1. If an item is written in the stack. obviously the stack cannot be empty, so EMTY is cleared to 0.

A new item is deleted from the stack if the stack is not empty (if EMPY = 0). The pop operation consists of the following sequence of micro operations:

DR←M[SP]	Read item from the top of stack
SP←SP – 1	Decrement stack pointer
If (SP=0) then (EMTY←1)	Check is stack is empty
FULL ← 0	Mark the stack not full

The top item is read from the stack into DR. The stack pointer is then decremented. If its value reaches zero, the stack is empty, so EMTY is set to 1. This condition is reached if the item read was in location 1. Once this item is read out, SP is decremented and reaches the value 0, which is the initial value of SP. An erroneous operation will result if the stack is pushed when FULL = 1 or popped when EMPTY = 1.

Memory Stack

The implementation of a stack is the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer. Figure 8-4 shows a portion of computer memory partitioned into three segments program, data, and stack. The programe counter PC ponts at the address of the nest instruction in the program. The address register AR points at an array of data. The stack pointer SP points at the top of the stack. The three registers are connected to a common address bus, and either one can provide an address for memory. PC is used during the fetch phase to read an instruction. AR is used during the execute phase to read an operand. SP is used to push or pop items into or from the stack.

As shown in Fig. 4.4, the initial value of SP is 4001 and the stack grows with decreasing addresses. Thus the first item stored in the stack is at address 4000, the second item is stored at address 3999, and the last address that can be used for the stack is 3000. No provisions are available for stack limit checks.

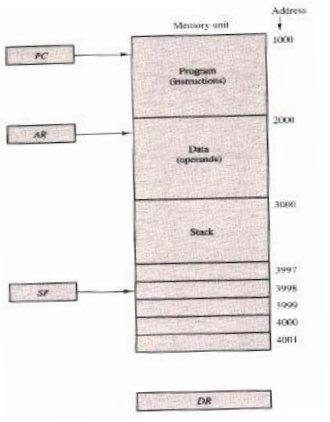
We assume that the items in the stack communicate with a data register DR. A new item is inserted with the push operation as follows.

SP←SP – 1

M[SP] ← DR

The stack pointer is decremented so that it points at the address of the next word. A memory write operation inserts the word from DR into the top of the stack. A new item is deleted with a pop operation as follows.

Figure 4.4 Computer memory with program, data and stack segments.



DR ←M[SP]

SP←SP + 1

The top item is read from the stack into DR. The stack pointer is then incremented to point at the next item in the stack.

Most computers do not provide hardware to check for stack overflow (full stack) or underflow (empty stack). The stack limits can be checked by using two processor registers: one to hold the upper limit (3000in this case), and the other to hold the lower limit (4001 in this case). After a push operations, SP is compared with the upper-limit register and after a pop operation, SP is compared with the lower–limit register.

Reverse Polish Notation

A stack organization is very effective for evaluating arithmetic expressions. The common arithmetic expressions are written in infix notation with each operator written between the operands. Consider the simple arithmetic expression.

A*B + C*D

The star (denoting multiplication) is placed between two operands A and B or C and D. The plus is between the two products. To evaluate this arithmetic expression it is necessary to compute the product A*B, store this product while computing C*D, and the sum the two products.

The Polish mathematician Lukasiewicz showed that arithmetic expressions can be represented in prefix notation. This representation, often referred to as Polish notation. places the operator before the operands. The postfix notatio referred to as reverse Polish notation (RPN), places the operator after the operands. The following examples demonstrate the three representations:

A +B	Infix notation
+AB	Prefix or Polish notation
AB+	Postfix or reverse Polish notation.

The reverse Polish notation is in a form suitable for stack manipulation. The expression

A*B + C* D

is written in reverse Polish notation as

AB * CD * +

and is evaluated as follows. Scan the expression from left to right. When an operator is reached, perform the operation with the two operands found on the left side of the operator. Remove the two operands and replace them by the number obtained from the result of the operation. Continue to scan the expression and repeat the procedure for every operator encountered until there are no more operators.

For the expression above we find the operator * after A and B. We perform the operation A*B and replace A,B, and * by the product to obtain

(A*B) CD* +

The next operator is a * and its previous two operands are C and D, so we perform C*D and obtain an expression with two operands and one operator.

(A*B) (C*D) +

The next operator is + and the two operands to be added are the two products, so we add the two quantities to obtain the result.

The conversation from infix notation to reverse Polish notation must take into consideration the operational hierarchy adopted for infix notation.

Consider the expression (A+B)*[C*(D+E)+F]

The expression can be converted to reverse Polish notation, without the use of parentheses, by taking into consideration the operation hierarchy. The converted expression is

AB + DE + C* F+*

Proceeding from left to right, we first add A and B, then add D and E. Reverse Polish notation, combined with a stack arrangement of registers, is the most efficient way known for evaluating arithmetic expressions.

The following numerical example may clarify this procedure. Consider the arithmetic expression

(3*4) + (5*6)

In reverse Polish notation, it is expressed as 34 * 56 * +

First the number 3 is pushed into the stack, then the number 4. The next symbol is the multiplication operator*. This causes a multiplication of the two topmost items in the stack. The stack is then popped and the product is placed on top of the stack, replacing the two original operands. Next we encounter the two operands 5 and 6, so they are pushed into the stack. The stack operation that results from the next* replaces these two numbers by their product. The last operation causes an arithmetic addition of the two topmost numbers in the stack to produce the final result of 42.

4.4 Instruction Formats

The physical and logical structure of computers is normally described in reference manuals provided with the system. Such manuals explain the internal construction of the CPU, including the processor registers available and their logical capabilities.

The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register. The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are;

1. An operation code field that specifics the operation to be performed.
2. An address field that designates a memory address or a processor register.
3. A mode field that specifies the way the operand or the effective address is determined.

The operation code field of an instruction is a group of bits that define various processor operations such as add, subtract, complement, and shift. The bits that define the mode field of an instruction code specify a variety of alternatives for choosing the operands from the given address.

A register address is a binary number of k bits that defines one of 2^k registers in the CPU. Thus a CPU with 16 processor registers R0 through R15 will have a register address field of four bits. The binary number 0101, for example, will designate register R5.

Computers may have instructions of several different lengths containing varying number of addresses. The number of address fields in the instruction format of a computer

depends on the internal organization of its registers. Most computers fall into one of three types of CPU organizations:

1. Single accumulator organization.
2. General register organization.
3. Stack organization.

All operations are performed with an implied accumulator register. The instruction format in this type of computer uses one address field. For example, the instruction that specifies an arithmetic addition is defined by an assembly language instruction as

ADD X

where X is the address of the operand. The ADD instruction in this case results in the operation AC ← AC + M [X]. AC is the accumulator register and M[X] symbolizes the memory word located at address X.

The instruction format in this type of computer needs three register address fields. Thus the instruction for an arithmetic addition may be written in an assembly language as

ADD R1, R2, R3

to denote the operation R1 ← R2 + R3. The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers. Thus the instruction

ADD R1, R2

would denote the operation R1 ← R1 + R2. Only register addresses for R1 and R2 need be specified in this instruction

Computers with multiple processor registers use the move instruction with a mnemonic MOV to symbolize a transfer instruction. Thus the instruction

MOV R1, R2

denotes the transfer R1 ← R2 (or R2 ← R1, depending on the particular computer). Thus transfer – type instructions need two address fields to specify the source and the destination.

General register – type computers employ two or three address fields in their instruction format. Each address field may specify a processor register or a memory word. An instruction symbolized by

ADD R1, X

would specify the operation R1 ← R1 + M [X]. It has two address fields, one for register R1 and the other for the memory address X.

Computers with stack organized would have PUSH and POP instructions which require an address field. Thus the instruction

PUSH X

will push the word at address X to the top of the stack.

The instruction

ADD

in a stack computer consists of an operation code only with no address field. This operation has the effects of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack.

Some computers combine features from more than one organizational structure. For example, the Intel 8080 microprocessor has seven CPU registers, one of which is an accumulator type. All arithmetic and logic instructions, as well as the load and store instructions, use the accumulator register, so these instructions have only one address field. On the other hand, instructions that transfer data among the seven processor registers have a format that contains two register address fields. Moreover, the Intel 8080 processor has a stackpointer and instructions to push and pop from a memory stack. The processor, however, does not have the zero-address-type instructions which are characteristic of a stack – organized CPU.

The arithmetic statement

X = (A + B) * (C + D)

using zero, one, two, or three address instructions. We will use the symbols ADD, SUB, MUL, and DIV for the four arithmetic operations; MOV for the transfer–type operation and LOAD and STORE for transfers to and from memory and AC register. We will assume that the operand are in memory address A, B, C and D, and the result must be stored in memory at address X.

Three – Address Instructions

Computers with three–address instruction formats can use each address field to specify whether a processor register or a memory operand. The program in assembly either a processor register or a memory operand. The program in assembly language that evaluates X = (A + B) * (C + D) is shown below, together with comments that explain the register operation of each instruction.

ADD	R1, A , B	R1 ← M [A] + M [B]
ADD	R2, C, D	R2 ← M [C] + M [D]
MUL	X, R1, R2	M [X] ← R1 * R2

It is assumed that the computer has two processor registers, R1 and R2. The symbol M [A] denotes the operand at memory address symbolized by A.

The advantage of the three – address format is that it result in short programs when evaluating arithmetic expressions. The disadvantage is that the binary–coded instructions require too many bits to specify three addresses.

Two–Address Instructions

Two–address instructions are the most common in commercial computers. Here again each address field can specify either a procedure register or a memory word. The program to evaluate X = (A+B) * (C+D) is as follows.

MOV	R1, A	R1 ← M [A]
ADD	R1, B	R1 ← R1 + M [B]
MOV	R2, C	R2 ← M [C]
ADD	R2, D	R2 ←R2 + M [D]
MUL	R1, R2	R1 ←R1 * R2

MOV	X , R1`R1←M [X] ← R1	

The MOV instruction moves or transfers the operands to and from memory and processor registers.

One–Address Instructions

One-address instructions use an implied accumulator (AC) register for all data manipulation. For multiplication and division there is a need for a second register. However, here we will neglect the second register and assume that the AC contains the result of all operations. The program to evaluate X = (A + B) * (C + D) is

LOAD A	AAC ← M [A]	
ADD	B	AC ←AC + M [B]
STORE	T	M [T] ← AC
LOAD C	C	AC ← M [C]
ADD	D	AC ← M [C]
MUL	T	AC ← AC + M [T]
STORE	X	M [X] ← AC

All operations are done between the AC register and a memory operand.

Zero – Address Instructions

A stack–organized computer does not use an address field for the instructions ADD and MUL. The PUSH and POP instructions need an address field to specify the operand that communicates with the stack. The following program shows how X = (A+ B) * (C + D) will be written for a stack organized computer. (TOS stands for top of stack.)

PUSH	A	TOS←A
PUSH	B	TOS←B
ADD		TOS← (A + B)
PUSH	C	TOS←C
PUSH	D	TOS←D
ADD		TOS ←(C + D)
MUL		TOS← (C + D) * (A + B)
POP	X	M [X]←TOS

RISC Instructions

The instruction set of a typical RISC processor is restricted to the use of load and store instructions when communicating between memory and CPU. All other instructions are executed within the registers of the CPU without referring to memory. A program for a RISC –type CPU consists of LOAD and STORE instructions that have one memory and one memory and one register address, and computational–type instructions that have three addresses with all three specifying processor registers. The following is a program to evaluate X = (A+B) * (C + D)

LOAD	R1, A	R1← M [A]
LOAD	R2, B	R2 ←M [B]

LOAD	R3, C	R3← M [C]
LOAD	R4, D	R4 ←M [D]
ADD	R1, R1, R2	R1 ← R1 + R2
ADD	R3, R3, R4	R3 ← R3 + R4
MUL	R1, R1, R3	R1 ←R1 + R3
STORE	X, R1	M [X] ← R1

The load instructions transfer the operands from memory to CPU registers. The add and multiply operations are executed with data in the registers without accessing memory. The result of the computations is then stored in memory with a store instruction.

4.5. Addressing Modes

The addressing mode specifies a rule for interpreting or modifying the address field of the instructions before the operand is actually referenced. Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions

1. To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
2. To reduce the number of bits in the addressing field of the instruction.

The control unit of a computer is designed to go through an instruction cycle that is divided into three major phases.

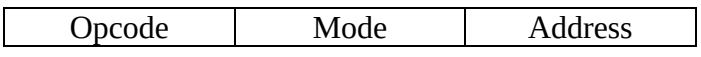
1. Fetch the instruction from memory
2. Decode the instruction
3. Execute the instruction.

There is one register in the computer called the program counter or PCthat keeps track of the instructions in the program stored in memory. PC holds the address of the instruction to be executed next and is incremented each time an instruction is fetched from memory.

The operation code specifies the operation to be performed. The mode field is used to locate the operands needed for the operation. There may or may not be an address field in the instruction. If there is an address filed, it may designate a memory address or a processor register. Most addressing modes modify the address field of the instruction there are two modes that need no address field at all. There are the implied and immediate modes.

Implied Mode : In this mode the operands are specified implicitly in the definition of the instruction. For example the instruction complement accumulator is an implied mode instruction, all register reference instructions that use an accumulator are implied mode instructions. Zero address instructions in a stack organized computer are implied mode instructions since the operands are implied to be on top of the stack.

Figure 4.5 Instruction format with mode field



Immediate Mode: In this mode the operand is specified in the instruction itself, In order words, an immediate mode instruction has an operand field rather than an address field. The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction.

Register Mode: In this mode the operands are in registers that reside within the CPU. The particular register is selected from a register field in the instruction.

Register Indirect Mode: In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory. In other words, the selected register contains the address of the operand rather than the operand itself. The advantage of a register indirect mode instruction is that the address field of the instruction uses fewer bits to select a register.

Auto increment or Auto decrement Mode:

This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory. When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table. This can be achieved by using the increment or decrement instruction.

The effective address is defined to be the memory address obtained from the computation by the given addressing mode. The effective address is the address of the operand in a computational type instruction.

Direct Address Mode:

In this mode the effective address is equal to the address part of the instruction.

Indirect Address Mode:

In the mode the address field of the instruction gives the address where the effective address is stored in memory.

Relative Address Mode:

In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address.

Indexed Addressing Mode:

In this mode the content of an index register is added to the address part of the instruction to obtain the effective address.

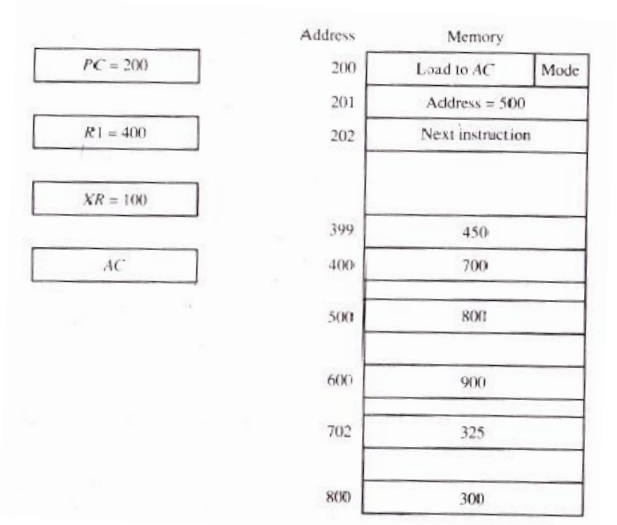
Based Register Addressing Mode:

In this mode the content of a base register is added to the address part of the instruction to obtain the effective address.

Numerical Example

In Fig.7. The two – word instruction at address 200 and 201 is a “load to AC” instruction with an address field equal to 500. The first word of the instruction specifics the operation code and mode, and the second word specifies the address part. PC has the value 200 for fetching this instruction. The content of processor register R1 is 400 and the content of an index register XR is 100. AC receives the operand after the instruction is executed.

Figure 4.6 Numerical example for addressing modes



The mode field of the instruction can specify any one of a number of modes. For each possible mode we calculate the effective address and the operand that must be loaded into AC. In the direct address mode the effective address is the address part of the instruction 500 and the operand to be loaded into AC is 800. In the immediate mode the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into AC. (The effective address in this case is 201.) In the indirect mode the effective address is stored in memory at address 500. Therefore, the effective address is 800 and the operand is 300. In the relative mode that effective address is500 + 202= 702 and the operand is 325. In the index mode the effective address is XR + 500 = 100+500 = 600 and the operand is 900. In the register mode the operand is in R1 and 400is loaded into AC. In the register indirect mode the effective address is 400, equal to the content of R1 and the operand loaded into AC is 700. The auto increment mode is the same as the register indirect mode except that R1 is incremented to 401 after the execution of the instruction. The auto decrement mode decrements R1 to 399 prior to the execution of the instruction. The operand loaded into AC is now 450.

4.6 Reduced Instruction Set

A computer with a large number of instructions is classified as a complex instruction set computer, abbreviated CISC.

In the early 1980s, a number of computer designers recommended that computers use fewer instructions with simple constructs so they can be executed much faster within the CPU without having to use memory as often. This type of computer is classified as a reduced instruction set computer or RISC.

CISC Characteristics

The essential goal of a CISC architecture is to attempt to provide a single machine instruction for each statement that is written in a high-level language.

Another characteristic of CISC architecture is the incorporation of variable-length instruction formats. Instructions that require register operands may be only two bytes in length, but instructions that need two memory addresses may need five bytes to include the entire instruction code.

The instructions in a typical CISC processor provide direct manipulation of operands residing in memory. For example, an ADD instruction may specify one operand in memory through index addressing and a second operand in memory through a direct addressing. Another memory location may be included in the instruction to store the sum. This requires three memory references during execution of the instruction. As more instructions and addressing modes are incorporated into a computer, the more hardware logic is needed to implement and support them, and this may cause the computations to slow down. Major characteristics of CISC architecture are:

1. A large number of instructions—typically from 100 to 250 instructions
2. Some instructions that perform specialized tasks and are used infrequently
3. A large variety of addressing modes – typically from 5 to 20 different modes.
4. Variable – length instruction formats
5. Instructions that manipulate operands in memory.

RISC Characteristics

The concept of RISC architecture involves an attempt to reduce execution time by simplifying the instruction set of the computer. The major characteristics of a RISC processor are,

1. Relatively few instructions
2. Relatively few addressing modes
3. Memory access limited to load and store instruction
4. All operations done within the registers of the CPU
5. Fixed–length, easily decoded instruction format.
6. Single–cycle instruction execution
7. Hardwired rather than micro programmed control.

The small set of instructions of a typical RISC processor consists mostly or register–to-register operations, with only simple load and store operations for memory access.

Results are transferred to memory by means of store instructions. The use of only a few addressing modes results from the fact that almost all instructions have simple register addressing. The instruction length can be fixed and aligned on word boundaries. By simplifying the instructions and their format, it is possible to simplify the control logic. For faster operations, a hardwired control is presented in micro programmed control.

Other characteristics attributed to RISC architecture are:

1. A relatively large number of registers in the processor unit
2. Use of overlapped register windows to speed-up procedure call and return
3. Efficient instruction pipeline
4. Compiler support for efficient translation of high-level language programs into machine language programs

A large number of registers is useful for storing intermediate results and for optimizing operand references. The advantage of register storage as op-posed to memory storage is that registers can transfer information to other registers much faster than the

transfer of information to and from memory. Thus register-to-memory operations can be minimized by keeping the most frequent accessed operands in registers. For this reason a large number of registers in the processing unit are sometimes associated with RISC processors.

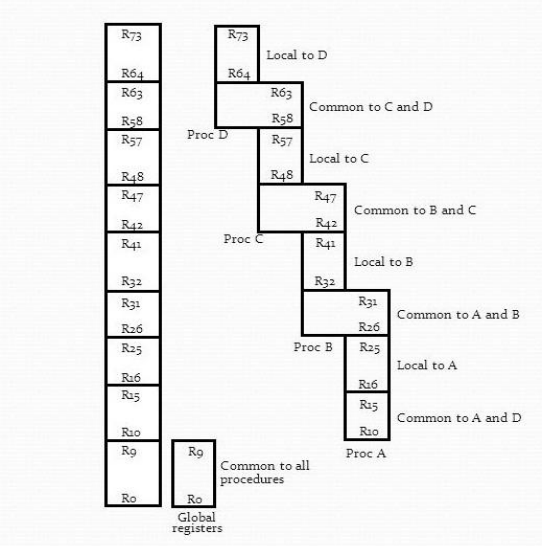
Overlapped Register Windows

Procedure call and return occurs quite often in high-level programming languages. When translated into machine language, a procedure call produces a sequence of instructions that save register values, pass parameters needed for the procedure, and then calls a subroutine to execute the body of the procedure. After a procedure return, the program restores the old register values, passes results to the calling program, and returns from the subroutine. Saving and restoring registers and passing of parameters and results involve time-consuming operation.

A characteristic of some RISC processors is their use of overlapped register windows to provide the passing of parameters and avoid the need for saving and restoring register values. Each procedure call results in the allocation of a new window consisting of a set of registers from the register file for use by the new procedure. Each procedure call activates a new register window by incrementing a pointer, while the return statement decrements the pointer and causes the activation of the previous window. Windows for adjacent procedures have overlapping registers that are shared to provide the passing of parameters and results.

The concept of overlapped register windows is illustrated in Fig. system has a total of 74 registers. Registers RO through R9 are global that hold parameters shared by all procedures. The other 64 registers are divided into four windows to accommodate procedures A, B, C, and D. Each register window consists of 10 local registers and two sets of six registers common to adjacent windows. Local registers are used for local variables. Common registers are used for exchange of parameters and results adjacent procedures. The common overlapped registers permit parameters to be passed without the actual movement of data. Only one register window is activated at any given time with a pointer indicating the active window. Each procedure call activates a new register window by incrementing the pointer. The high registers of the calling procedure overlap the low register of the called procedure, and therefore the parameters automatically transfer from calling to called procedure .

Figure 4.7 Overlapped Register Windows



Suppose that procedure A calls procedure B. Registers R26 through R31 are common to both procedures, and therefore procedure A stores the parameters for procedure B in these registers. Procedure B uses local registers R32 through R41 for local variable storage. If procedure B calls procedure C, it will pass the parameters through registers R42 through R47. When procedure B is ready to return at the end of its computation, the program stores results of the computation in registers R26 through R31 and transfers back to the register window of procedure A. Note that registers R10 through common to procedures A and D because the four windows have a circular organization with A being adjacent to D.

The 10 global registers R0 through R9 are available to all procedures. This includes 10 global registers, 10 local registers, six low overlapping register, and six high overlapping registers.

number of global registers = G

number of local registers in each window = L

number of registers common to two windows = C

number of windows = W

The number registers available for each window is calculated as folk

window size = L + 2C + G

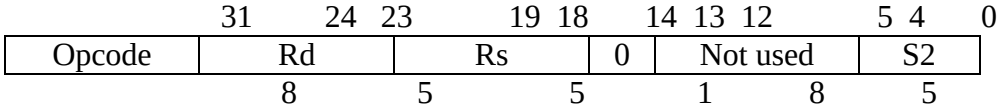
The total number of registers needed in the processor is

register file = (L + C)W + G

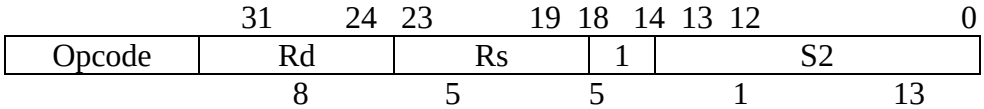
Berkeley RISC I

The Berkeley RISC is a 32-bit integrated circuit CPU. It supports 32-bit addresses and either 8-, -16 -, or 32-bit data. It has a 32-bit instruction format and a total of 31 instructions. There are three basic addressing modes: register addressing, immediate operand, and relative to PC addressing for branch instructions. It has a register file of 138 registers arranged into 10 global registers and 8 windows registers in each. Since only one set of 32 registers in a window is

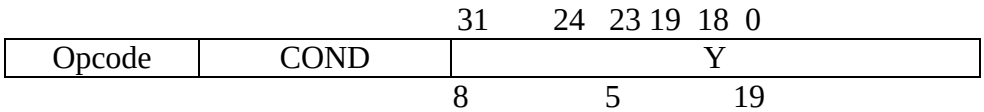
Figure 4.8 Berkeley RISC I instruction formats.



(a) Register mode: (S2 specifies a register)



(b) Register—immediate mode: (S2 specifies an operand)



(C) P C relative mode

accessed at any given time, the instruction format can specify a processor register with a register field of five bits.

Figure 4.8 shows the 32-bit instruction formats used for register-to-register instructions and memory access instructions. Seven of the bits in the operation code specify an operation, and the eighth bit indicates whether to update the status bits after an ALU operation. For register-to-register instructions, the 5-bit Rd field selects one of the 32 registers as a destination for the result of the operation. The operation is performed with the data specified in fields Rs and S2. Rs is one of the source registers. If bit 13 of the instruction is 0, the low-order 5 bits of S2 specify register. If bit 13 of the instruction is 1, S2 specifies a sign-extended 13-bit constant. Thus the instruction has a three-address format, but the second source may be either a register or an immediate operand. Memory access instructions use Rs to specify a 32-bit address in a register and S2 to specify an offset. Register R0 contains all 0's, so it can be used in any field to specify a zero quantity. The third instruction format combines the last three fields to form a 19-bit relative address Y and is used primarily with the jump and call instructions. The COND field replaces the Rd field for jump instructions and is used to specify one of 16 possible branch conditions.

UNIT IV

5. COMPUTER ARITHMETIC

5.1 Addition and Subtraction

There are three ways of representing negative fixed–point binary numbers : signed – magnitude, signed -1's complement, or signed-2's complement. Most computers use the signed – 2's complement representation when performing arithmetic operations with integers. For floating–point operations, most computers use the signed–magnitude representation for the mantissa.

Addition and Subtraction with Signed–Magnitude Numbers

The representation of numbers in signed – magnitude is familiar because it is used in everyday arithmetic calculations. We designate the magnitude of the two numbers by A and B. When the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed. These conditions are listed in the first column of Table 5-1. The other columns in the table show the actual operation to be performed with the magnitude of the numbers. The last column is needed to prevent a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0

The algorithms for addition and subtraction are derived as follows. Addition (Subtraction) algorithm : when the signs of A and B are identical (different), add the two magnitudes and attach the sign of A to the result when the signs of A and B are different (identical), compare the magnitudes and

TABLE 5-1 Addition and Subtraction of Signed – Magnitude Numbers

Eight Conditions for Signed-Magnitude Addition/Subtraction

	Operation	ADD Magnitudes	SUBTRACT Magnitudes		
			A > B	A < B	A = B
1	(+A) + (+B)	+ (A + B)			
2	(+A) + (-B)		+ (A – B)	- (B – A)	+ (A – B)
3	(-A) + (+B)		- (A – B)	+ (B – A)	+ (A – B)
4	(-A) + (-B)	- (A + B)			
5	(+A) - (+B)		+ (A – B)	- (B – A)	+ (A – B)
6	(+A) - (-B)	+ (A + B)			
7	(-A) - (+B)	- (A + B)			
8	(-A) - (-B)		- (A – B)	+ (B – A)	+ (A – B)

Subtract the smaller number from the larger. Choose the sign of the result to be the same as A if A > B or the complement of the sign of A if A < B. If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

Hardware Implementation

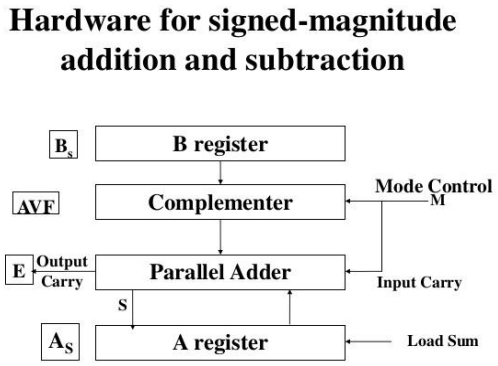
Let A and B be two registers that hold the magnitudes of the numbers, and $A_s$and B_s be two flip–flops that hold the corresponding signs. The result of the operation may be transferred to a third register,however, a saving is achieved if the result is transferred into A and As. Thus A and A_s together form an accumulator register.

Consider now the hardware implementation of the algorithms above. First, a parallel adder is need to perform the micro operation A + B. Second, a comparator circuit is needed to establish if A > B, A =B, or A<B. Third, two parallel–subtract or circuits are needed to perform the micro operations A –B and B –A. The sign relationship can be determined from an exclusive–OR gate with A_s and B_s as inputs.

Figure 5.1 shows a block diagram of the hardware for implementing the addition and subtraction operations. It consists of registers A and B and sign flip–flops A_s and B_s. Subtraction is done by adding A to the 2's complement of B. The output carry is transferred to flip–flop E, where it can be checked to determine the relative magnitudes of the two numbers. The add–overflow flip–flop AVF holds the overflow bit when A and B are added.

The addition of A plus B is done through the parallel adder. The S (sum) output of the adder is applied to the input of the A register. The complementer provides an output of B or the complement of B based on the state of the mode control M. The complementer consists of exclusive –OR gates and the parallel adder consists of full–adder circuits. The M signal is also applied to the input carry of the adder. When M =0, the output of B is transferred to the adder, the input carry is 0, and the output of the adder is equal to the sum A + B.

Figure 5.1 Hardware for signed magnitude addition and subtraction

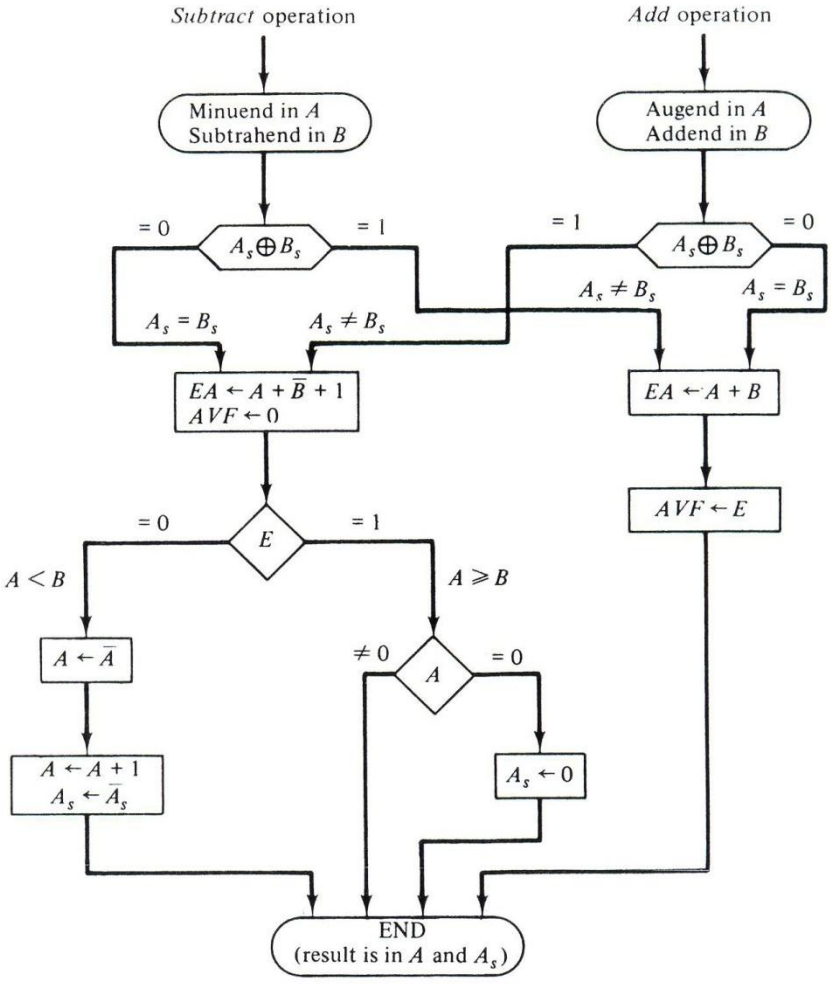


When M =1, the 1's complement of B is applied to the adder, the input carry is 1, and output $S = A + \overline{B} + 1$. This is equal to A plus the 2's complement of B, which is equivalent to the subtraction A –B.

The flowchart for the hardware algorithm is presented in Fig. 5.2. The two signs A_s and B_s are compared by an exclusive–OR gate. If the output of the gate is 0, the signs are identical; if it is 1, the signs are different. For an add operation, identical signs dictate that the magnitudes be added. For a subtract operation, different signs dictate that the magnitudes be added. The magnitudes are added with a microoperation$EA \leftarrow A + B$, where EA is a register that combines E and A. The carry in E after the addition constitutes an overflow if it is equal to 1. The value of E is transferred into the add-overflow flip – flop AVF.

The two magnitudes are subtracted if the signs are different for an add operation or identical for a subtract operation. The magnitudes are subtracted by adding A to the 2's complement of B. No overflow can occur if the numbers are subtracted so AVF is cleared to 0. A 1 in E indicates that $A \geq B$ and the number in A is the correct result. If this number is zero, the sign A_s must be made positive to avoid a negative zero. A 0 in E indicates that $A < B$. For this case it is necessary to take the 2's complement of the value in A. This operation can be done with one microoperation$A \leftarrow \bar{A}+ 1$. However, we assume that the A register has circuits for microoperations complement and increment, so the 2's complement is obtained from these two microoperations. In other paths of the flowchart, the sign of the result is the same as the sign of A, so no change in A_s is required. When $A < B$,the sign of the result is the complement of the original sign of A. It is then necessary to complement A_s to obtain the correct sign.

Figure 5.2 Flow chart for add and subtract operations



The final result is found in register A and its sign in A_s. The value in AVF provides an overflow indication.

The leftmost bit of a binary number represents the sign bit : 0 for positive and 1 for negative. If the sign bit is 1, the entire number is represented in 2's complement form. The addition of two numbers in signed -2's complement form consists of adding the numbers with the sign bits treated the same as the other bits of the number. A carry – out of the sign – bit position is discarded. The subtraction consists of first taking the 2's complement of the subtrahend and then adding it to the minuend.

When two numbers of n digits each are added and the sum occupies n+1 digits, we say that an overflow occurred. An overflow can be detected by inspecting the last two carried out of the addition. When the two carries are applied to an exclusive–OR gate, the overflow is detected when the output of the gate is equal to 1.

The register configuration for the hardware implementation is shown in Fig. 10-3. We name the A register AC (accumulator) and the B register BR. The leftmost bit in AC and BR represent the sign bits of the numbers. The two sign bits are added or subtracted together with the other bits in the complementer and parallel adder. The overflow flip – flop V is set to 1 if there is an overflow.

The algorithm for adding and subtracting two binary numbers in signed 2's complement representation is shown in the flowchart of Fig. 5.4. The sum is obtained by adding the contents of AC and BR (including their sign bits). The overflow bit V is set to 1 if the exclusive–OR of the last two carries is 1, and it is clearedto 0 otherwise. The subtraction operation is accomplished by adding the content of AC to the 2's complement of BR. Taking the 2's complement of BR has the effect of changing a positive number to negative, and vice versa. An overflow must be checked during this operation because the two numbers added could have the same sign. The programmer must realize that if an overflow occurs, there will be an erroneous result in the AC register.

Figure 5.3 Hardware for signed – 2's complement addition and subtraction

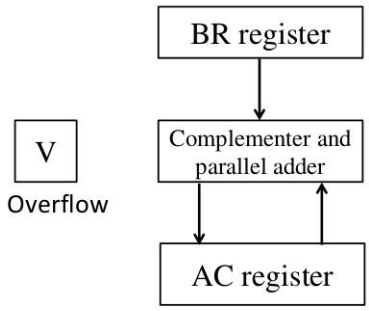
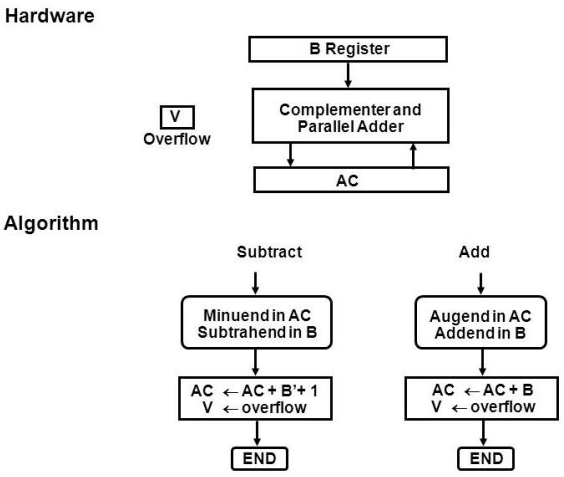


Figure 5.4 Algorithm for adding and subtracting numbers in signed -2's complement representation.



Comparing this algorithm with its signed – magnitude counterpart, we note that it is much simpler to add and subtract numbers if negative numbers are maintained in signed -2's

complement representation. For this reason most computers adopt this representation over the more familiar signed–magnitude.

5.2 Multiplication Algorithms

Multiplication of two fixed–point binary numbers in signed – magnitude representation is done with paper and pencil by a process of successive shift and add operations. This process is best illustrated with a numerical example.

```
23          10111        Multiplicand
19    x     10011        Multiplier
          ---------
          10111
         10111
        00000          +
       00000
      10111
      ---------
437   110110101    Product
```

If the multiplier bit is a 1, the multiplicand is copied down; otherwise, zeros are copied down. The numbers copied down in successive lines are shifted one position to the left from the previous number. Finally, the numbers are added and their sum forms the product.

When multiplication is implemented in a digital computer, it is convenient to change the process slightly. First, instead of providing registers to store and add simultaneously as many binary numbers as there are bits in the multiplier, it is convenient to provide an adder for the summation of only two binary numbers and successively accumulate the partial products in a register. Second, instead of shifting the multiplicand to the left, the partial product is shifted to right, which results in leaving the partial product and the multiplicand in the required relative positions. Third, when the corresponding bit of the multiplier is 0, there is no need to add all zeros to the partial product since it will not alter its value.

The hardware for multiplication consists of the equipment shown in Fig. 5.1 plus two more registers. These registers together with registers A and B are shown in Fig. 5.5. The multiplier is stored in the Q register and its sign in Q_s. The sequence counter SC is initially set to a number equal to the number of bits in the multiplier. The counter is decremented by 1 after forming each partial product. When the content of the counter reaches zero, the product is formed and the process stops.

The multiplicand is in register B and the multiplier in Q. The sum of A and B forms a partial product which is transferred to the EA register. Both partial product and multiplier are shifted to the right. this shift will be denoted by the statement shr EAQ to designate the right shift depicted in Fig. 5.5. The least significant bit of A is shifted into the most significant position of Q, the bit from E is shifted into the most significant position of A, and 0 is shifted into E. After the shift, one bit of the partial product is shifted into Q, pushing the multiplier bits one position to the right. In this manner, the rightmost flip –flop in register Q, designated by Q_n, will hold the bit of the multiplier, which must be inspected next.

Figure 5.5 Hardware for multiply operation.

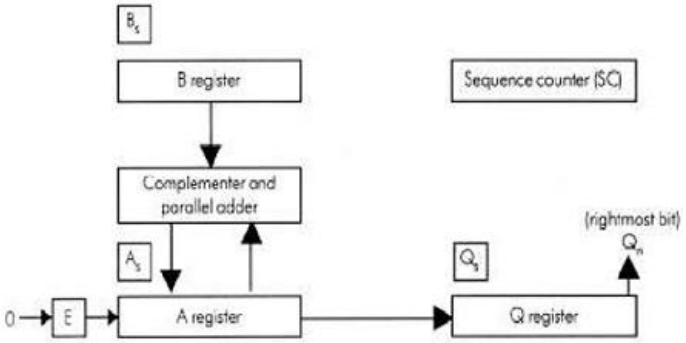


Figure 5.6 is a flowchart of the hardware multiply algorithm. Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in Bs and Q_s, respectively. The signs are compared , and both A and Q are set to correspond to the sign of the product since a double length product will be stored in registers A and Q. Registers A and E and cleared and the sequence counter SC is set to a number equal to the number of bits of the multiplier. We are assuming here that operands are transferred to registers from a memory unit that has words of n bits. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist on n -1 bits.

The low – order bit of the multiplier in Q_n is tested. If it is a 1, the multiplicand in B is added to the present partial product in A. If it is a 0, nothing is done. Register EAQ is then shifted once to the right to form the new partial product. The sequence counter is decremented by 1 and its new value checked. If it is not equal to zero, the process is repeated and a new partial product is formed. The process stops when SC =0. Note that the partial product formed in A is shifted into Q one bit at a time and eventually replaces the multiplier. The final product is available in both A and Q, with A holding the most significant bits and Q holding the least significant bits.

Figure 5.6 Flowchart for multiply operation

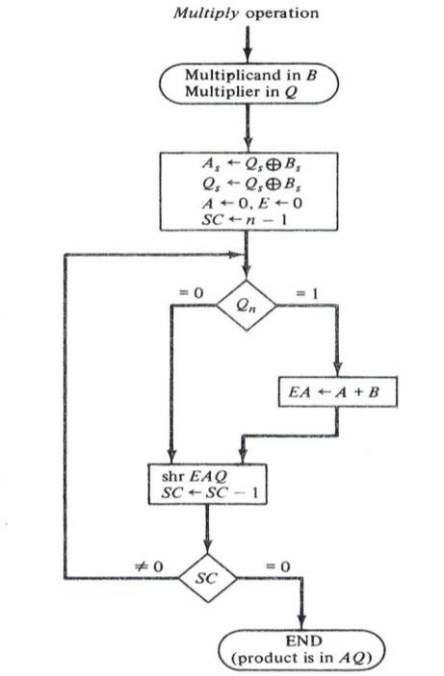


Table 5-2 Numerical example for binary multiplier

Multiplicand B = 10111

	C	A	Q	P
Multiplier in Q	0	00000	10011	101
Q_0 = 1; add B		10111		
First partial product	0	10111		100
Shift right CAQ	0	01011	11001	
Q_0 = 1; add B		10111		
Second partial product	1	00010		011
Shift right CAQ	0	10001	01100	
Q_0 = 0; shift right CAQ	0	01000	10110	010
Q_0 = 0; shift right CAQ	0	00100	01011	001
Q_0 = 1; add B		10111		
Fifth partial product	0	11011		
Shift right CAQ	0	01101	10101	000
Final product in AQ = 0110110101				

Booth Multiplication Algorithm

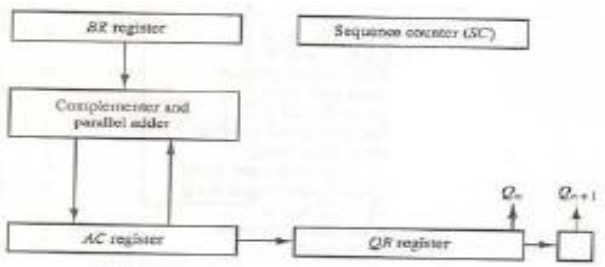
Booth algorithm gives a procedure for multiplying binary integers in signed -2's complement representation. It operates on the fact that strings of 0's in the multiplier require no addition but just shifting, and a string of 1's in the multiplier from bit weight 2^k to 2^m can be treated2^{k+1}-2^m. for example for binary number 001110(+14) when (k=3, m=1), the number can be represented as $2^{k+1} - 2^m = 2^4$-2^1=16- 2=14. Therefore, the multiplication M x 14, where M is the multiplicand and 14 the multiplier, can be done as M x 2^4 –M x 2^1. Thus the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.

Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial product, or left unchanged according to the following rules:

1. The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier.
2. The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous 1) in a string of 0's in the multiplier.
3. The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

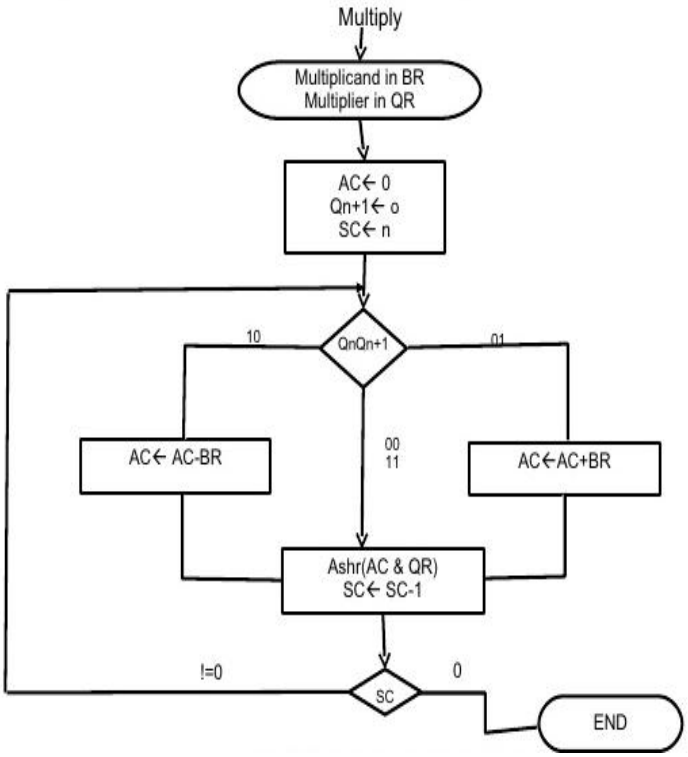
The hardware implementation of Booth algorithm requires the register configuration shown in Fig. 5.7. We rename registers A, B and Q, as AC, BR, and QR, respectively. Q_n designates the least significant bit of the multiplier in register QR. An extra flip – flop $Q_{n+1}$is appended to QR to facilitate a double bit inspection of the multiplier. The flowchart for Booth algorithm is shown in Fig. 5.8. AC and the appendedbitQ_n+1 are initially cleared to 0 and the sequence counter SC is set to a number n equal to the number of bits in the multiplier.

Figure 5.7 Hardware for Booth algorithm



The two bits of the multiplier in Q_n and Q_{n+1} are inspected. If the two bits are equal to 10, it means that the first 1 in a string of 1's has been encountered. This requires a subtraction of the multiplicand from the partial product in AC. If the two bits are equal to 01, it means that the first 0 in a string of 0's has been encountered. this requires the addition of the multiplicand to the partial product in AC.

Figure 5. 8 Booth algorithm for multiplication of signed -2's complement numbers.



The next step is to shift right the partial product and the multiplier (including bit Q_{n+1}). This is an arithmetic shift right (ashr) operation which shifts AC and QR to the right and leaves the sign bit in AC unchanged . The sequence counter is decremented and the computational loop is repeated n times.

TABLE 5-3 Example of Multiplication with Booth Algorithm

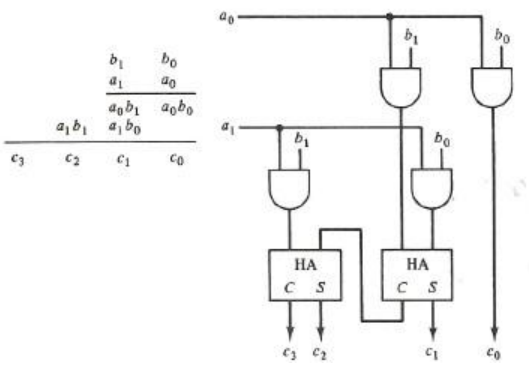
Q_nQ_{n+1}	BR=10111 BR+ 1=01001	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract BR	01001			
		01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	10111			
		11001			010
	ashr	11100	10110	0	001
0 0	ashr	11110	01011	0	
1 0	Subtract BR	01001			
		00111			
	ashr	00011	10101	1	000

Array Multiplier

Checking the bits of the multiplier one at a time and forming partial products is a sequential operation that requires a sequence of add and shift micro operations. The multiplication of two binary numbers can be done with one micro operation by means of a combinational circuit that forms the product bits all at once. This is a fast way of multiplying two numbers since all it takes is the time for the signals to propagate through the gates that form the multiplication array.

To see how an array multiplier can be implemented with a combinational circuit, consider the multiplication of two 2 – bit numbers as shown in Fig. 5.9. The multiplicand bits are b_1 and b_0, the multiplier bits are a_1 and a_0, and the product is $c_3c_2c_1c_0$. The first partial product is formed by multiplying a0 by b1b$_0$. The multiplication of two bits such as a_0 and b_0 produces a 1 if both bits are 1; otherwise, it produces a 0. This is identical to an AND operation and can be implemented with an AND gate. As shown in the diagram, the first partial product is formed by means of two AND gates. The second partial product is formed by multiplying a_1 by b_1b_0 and is shifted one position to the left. The two partial products are added with two half – adder (HA) circuits. Usually, there are more bits in the partial products and it will be necessary to use full–adders to produce the sum. Note that the least significant bit of the product does not have to go through an adder since it is formed by the output of the product does not have to go through an adder since it is formed by the output of the first AND gate.

Figure 5. 9 2 – bit by 2- bit array multiplier



5.3 Division Algorithms

Division of two fixed – point binary numbers in signed – magnitude representation is done with paper and pencil by a process of successive compare, shift, and subtract operations. Binary division is simpler than decimal division because the quotient digits are either 0 or 1 and there is no need to estimate how many times the dividend or partial remainder fits into the divisor. The division process is illustrated by a numerical example in Fig. 5.10. The divisor B consists of five bits and the dividend A, of ten bits. The five most significant bits of the dividend are compared with the divisor. Since the 5-bits of A and compare this number with B. The 6 bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend. The difference is called a partial remainder because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder. The process is continued by comparing a partial remainder with the divisor. If the partial remainder is greater than or equal to the divisor, the quotient bit is equal to 1. The divisor is then shifted right and subtracted from the partial remainder. If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case. Note that the result gives both a quotient and a remainder.

In a digital computer, it is convenient to change the process slightly. Instead of shifting the divisor to the right, the dividend, or partial remainder, is shifted to the left, thus leaving the two numbers in the required relative position. Subtraction may be achieved by adding A to the 2's complement of B. The information about the relative magnitudes is then available from the end – carry.

The hardware for implementing the division operation is identical to the multiplication operation.. Register EAQ is now shifted to the left with 0 inserted into $Q_n$and the previous value of E lost. The numerical example is repeated in fig 5.11

Figure 5.10 Example of binary division

Divisor:

B=10001

```
                10001)0111000000(11010
                      01110
                      011100
                      -10001
                      _______
                      -010110
                      --10001
                      _______
                      --001010
                      ---010100
                      ----10001
                      ________
                      ----000110
                      -----00110
```

Figure 5.11 Example of binary division with digital hardware

	E	A	Q	SC
Dividend :		01110	00000	5
ShlEAQ	0	11100	00000	
addB+1		01111		
E=1	1	01011		
Set Q_n =1	1	01011	00001	4
ShlEAQ	0	01010	00010	
Add B +1		01111		
E=1	1	00101		
Set Q_n =1	1	00101	00011	3
ShlEAQ	0	01010	00110	
Add B+1		01111		
E=0; leave Q_n=0	0	11001	00110	
Add B		10001		2
Restore remainder	1	01010		
ShlEAQ	0	10100	01100	
Add B +1		01111		
E=1	1	00011		
Set Q_n=1	1	00011	01101	1
ShlEAQ	0	00110	11010	
Add B+1		01111		
E=0; leave Q_n=0	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A		00110		
Quotient in Q			11010	

The divisor is stored in register B and the double length dividend is stored in registers A and Q. The dividend is shifted to the left and the divisor is subtracted by adding the 2's complement value. The information about the relative magnitude is available in E. If E=1 it signifies that A≥B. a quotient bit 1 is inserted into Q_n and the partial reminder is shifted to the left to repeat the process. If E=0 it signifies that A<B so the quotient in Q_n remains 0. The value of B is then added to restore the partial remainder in A to its previous value. The partial remainder is shifted to the left and the process is repeated again until all five quotient bits are formed. While the partial remainder is shifted left, the quotient bits are shifted also and after five shifts, the quotient is in Q and the final remainder is in A.

The sign of the quotient is determined from the signs of the dividend and the divisor. If the two signs are alike, the sign of the quotient is plus. If they are unalike, the sign is minus. The sign of the remainder is the same as the sign of the dividend.

Divide Overflow

The division operation may result in a quotient with an overflow. This is not a problem when working with paper and pencil but is critical when the operation is implemented with hardware. This is because the length of registers is finite and will not hold a number that exceeds the standard length.

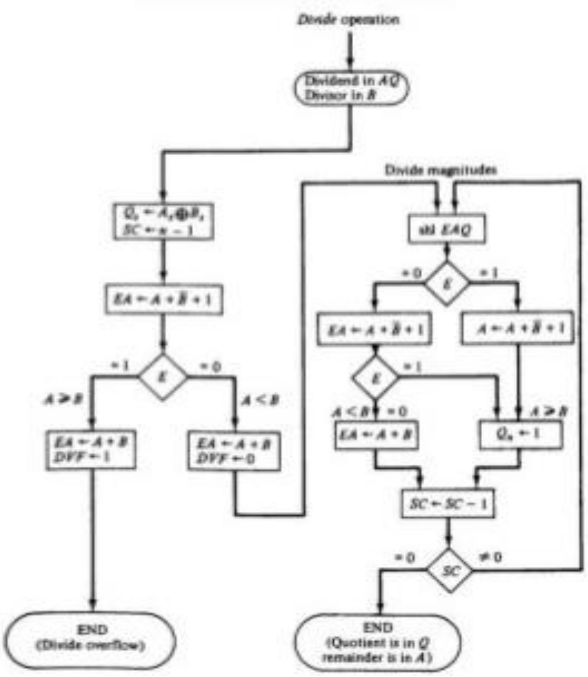
When the dividend is twice as long as the divisor, the condition for overflow can be stated as follows: A divide–overflow condition occurs if the high order half bits of the dividend constitute a number greater than or equal to the divisor. Another problem associated with division is the fact that a division by zero must be avoided. The divide overflow condition takes care of this condition as well. This occurs because any dividend will be greater than or equal to a divisor which is equal to zero. Overflow condition is usually detected when a special flip–flop is set. We will call it a divide–overflow flip–flop and label it DVF.

The occurrence of a divide overflow can be handled in a variety of ways. In some computers it is the responsibility of the programmers to check if DVF is set after each divide instruction. They then can branch to a subroutine that takes a corrective measure such as rescaling the data to avoid overflow. In some older computers, the occurrence of a divide overflow stopped the computer and this condition was referred to as a divide stop. Stopping the operation of the computer is not recommended because it is time consuming. The procedure in most computers is to provide an interrupt request when DVF is set. The interrupt causes the computers is to provide an interrupt request when DVF is set. The interrupt causes the computer to suspend the current program and branch to a service routine to take a corrective measure. The most common corrective measure is to remove the program and type an error message explaining the reason why the program could not be completed. The best way to avoid a divide overflow is to use floating point data.

Hardware Algorithm

The hardware divide algorithm is shown in the flowchart of Fig. 5.12. The dividend is in A and Q and the divisor in B. The sign of the result is transferred into Qs to be part of the quotient. A constant is set into the sequence counter SC to specify the number of bits in the quotient. As in multiplication, we assume that operands are transferred to registers from a memory unit that has words in n bits. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of n-1 bits.

Figure 5.12 Flowchart for divide operation



A divide – overflow condition is tested by subtracting the divisor in B from half of the bits of the dividend stored in A. If A≥ B, the divide-overflow flip–flop DVF is set and the operation is terminated prematurely. If A < B, no divide overflow occurs so the value of the dividend is restored by adding B to A.

The division of the magnitudes starts by shifting the dividend in AQ to the left with the high–order bit shifted into E. If the bit shifted into E is 1, we know that EA> B because EA consists of a 1 followed by n-1 bits while B consists of only n-1 bits. In this case, B must be subtracted from EA and 1 inserted into Q_n for the quotient bit. Since register A is missing the high – order bit of the dividend (which is in E), its value is EA -2^{n-1}. Adding to this value the 2's complement of B results in

$$(EA - 2^{n-1}) + (2^{n-1} - B) = EA - B$$

The carry from this addition is not transferred to E if we want E to remain a1.

If the shift – left operation inserts a 0 into E, the divisor is subtracted by adding its 2's complement value and the carry is transferred into E. If E = 1, it signifies that A ≥ B; therefore, Q_n is set to 1. If E =0, it signifies that A < B and the original number is restored by adding B to A. In the latter case we leave a 0 in Q_n (0 was inserted during the shift).

This process is repeated again with register A holding the partial remainder. After n- 1 times, the quotient magnitude is formed in register Q and the remainder is found in register A. The quotient sign is in Q_s and the sign of the remainder in A_s is the same as the original sign of the dividend.

Other Algorithms

The hardware method just described is called the restoring method. The reason for this name is that the partial remainder is restored by adding the divisor to the negative difference. Two other methods are available for dividing numbers, the comparison method and the non restoring method. In the comparison method A and B are compared prior to the subtraction operation. Then If A ≥ B, B is subtracted from A. If A < B nothing is done. The partial remainder is shifted left and the numbers are compared again. The comparison can be

determined prior to the subtraction by inspecting the end–carry out of the parallel–adder prior to its transfer to register E.

In the nonrestoring method, B is not added if the difference is negative but instead, the negative difference is shifted left and then B is added. In the nonrestoring method, B is subtracted if the previous value of Qn was a 1, but B is added if the previous value of Qn was a 0 and no restoring of the partial remainder is required. This process saves the step of adding the divisor if A is less than B, but it requires special control logic to remember the previous result. The first time the dividend is shifted, B must be subtracted. Also, if the last bit of the quotient is 0, the partial remainder must be restored to obtain the correct final remainder.

5.4 Float Point Arithmetic Operations

The most common way is to specify them by a real declaration statement as opposed to fixed–point numbers, which are specified by an integer declaration statement. Any computer that has a compiler for such high–level programming language must have a provision for handling floating–point arithmetic operations. The compiler must be designed with a package of floating–point software subroutines. The hardware method is more expensive, it is so much more efficient than the software method.

Basic Considerations

A floating point number in computer registers consists of two parts : a mantissa m and an exponent e. The two parts represent a number obtained from multiplying m time a radix r raised to the value of e; thus

$$m \times r^e$$

The mantissa may be a fraction or an integer. The location of the radix point and the value of the radix r are assumed and are not included in the registers. The decimal number 537.25 is represented in a register with m = 53725 and e = 3 and is interpreted to represent the floating –point number

$$.53725 \times 10^3$$

A floating – point number is normalized if the most significant digit of the mantissa is nonzero. In this way the mantissa contains the maximum possible number of significant digits. A zero cannot be normalized because it does not have a nonzero digits. A zero cannot be normalized because it does not have a nonzero digit. It is represented in floating–point by all 0's in the mantissa and exponent.

Floating – point representation increases the range of numbers that can be accommodated in a given register.

Arithmetic operations with floating–point numbers are more complicated than with fixed–point numbers and their execution takes longer and requires more complex hardware. Adding or subtracting two numbers requires first an alignment of the radix point since the exponent parts must be made equal before adding or subtracting the mantissas. The alignment is done by shifting one mantissa while its exponent is adjusted until it is equal to the other exponent. Consider the sum of the following floating – point numbers:

$$.5372400 \times 10^2$$

$$+ .1580000 \times 10^{-1}$$

It is necessary that the two exponents be equal before the mantissas can be added. We can either shift the first number three positions to the left, or shift the second number three positions to the right. When the mantissas are stored in registers, shifting to the left causes a

loss of most significant digits. Shifting to the right causes a loss of least significant digits. The second method is preferable because it only reduces the accuracy, while the first method may cause an error. The usual alignment procedure is to shift the mantissa that has the smaller exponent to the right by a number of places equal to the difference between the exponents. After this is done, the mantissas can be added :

$$.5372400 \times 10^2$$
$$+.0001580 \times 10^2$$
$$. 5373980 \times 10^2$$

When two normalized mantissas are added, the sum may contain an overflow digit. An overflow can be corrected easily by shifting the sum once to the right and incrementing the exponent. When two numbers are subtracted, the result may contain most significant zeros as shown in the following example:

$$. 56780 \times 10^5$$
$$-.56430 \times 10^5$$
$$. 00350 \times 10^5$$

A floating – point number that has a 0 in the most significant position of the mantissa is said to have an underflow. To normalize a number that contains an underflow, it is necessary to shift the mantissa to the left and decrement the exponent until a nonzero digit appears in the first position.

Floating – point multiplication and division do not require an alignment of the mantissas. The product can be formed by multiplying the two mantissas and adding the exponents. Division is accomplished by dividing the mantissas and subtracting the exponents.

The operations performed with the exponents are compare and increment (for aligning the mantissas), add and subtract (for multiplication and division), and decrement (to normalize the result). The exponent may be represent in any one of the three representation : signed – magnitude, signed – 2's complement, or signed -1's complement.

A fourth representation employed in many computers is known as a biased exponent. In this representation, the sign bit is removed from being a separate entity. The bias is a positive number that is added to each exponent as the floating – point number is formed, so that internally all exponents are positive.

The advantage of biased exponents is that they contain only positive numbers. It is then simpler to compare their relative magnitude without being concerned with their signs. As a consequence, a magnitude comparator can be used to compare their relative magnitude during the alignment of the mantissa and the smallest possible exponent.

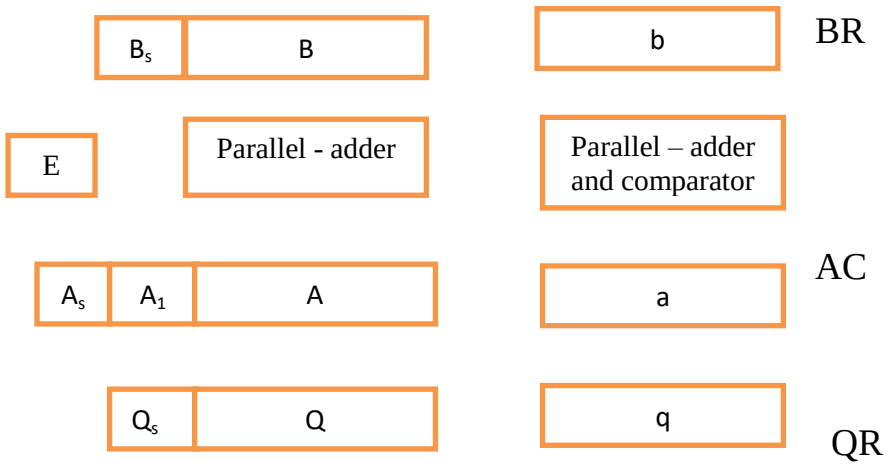
Register Configuration

The register configuration for floating – point operations is quite similar to the layout for fixed – point operations.

There are three registers, BR, AC, and QR. Each register is subdivided into two parts. the mantissa part has the same uppercase letter symbols as in fixed–point representation.

Each floating – point number has a mantissa in signed magnitude representation and a biased exponent. Thus the AC has a mantissa

Figure 5.13 Registers for floating–point arithmetic operation



whose sign is in A_s and a magnitude that is in A. The exponent is in the part of the register denoted by the lowercase letter symbol a. The diagram shows explicitly the most significant bit of A, labeled by A_1. The bit in this position must be a 1 for the number to be normalized. Note that the symbol AC represents the entire register, that is, the concatenation of A_s, A and a. Register BR is subdivided into Bs, B and b, and QR into Qs, Q, and q. A parallel–adder adds the two mantissas and transfers the sum into A and the carry into E. A separate parallel adder is used for the exponents. Since the exponents are biased, they do not have a distinct sign bit but are represented as a biased positive quantity. The floating–point numbers are so large that the chance of an exponent overflow is very remote, the exponent overflow will be neglected. The exponents are also connected to a magnitude comparator that provides three binary outputs to indicate their relative magnitude.

The number in the mantissa will be taken as a fraction, so the binary point is assumed to reside to the left of the magnitude part. The numbers in the registers are assumed to be initially normalized. After each arithmetic operation, the result will be normalized. Thus all floating – point operands coming from and going to the memory unit are always normalized.

Addition and Subtraction

During addition or subtraction, the two floating–point operands are in AC and BR. The sum or difference is formed in the AC. The algorithm can be divided into four consecutive parts:

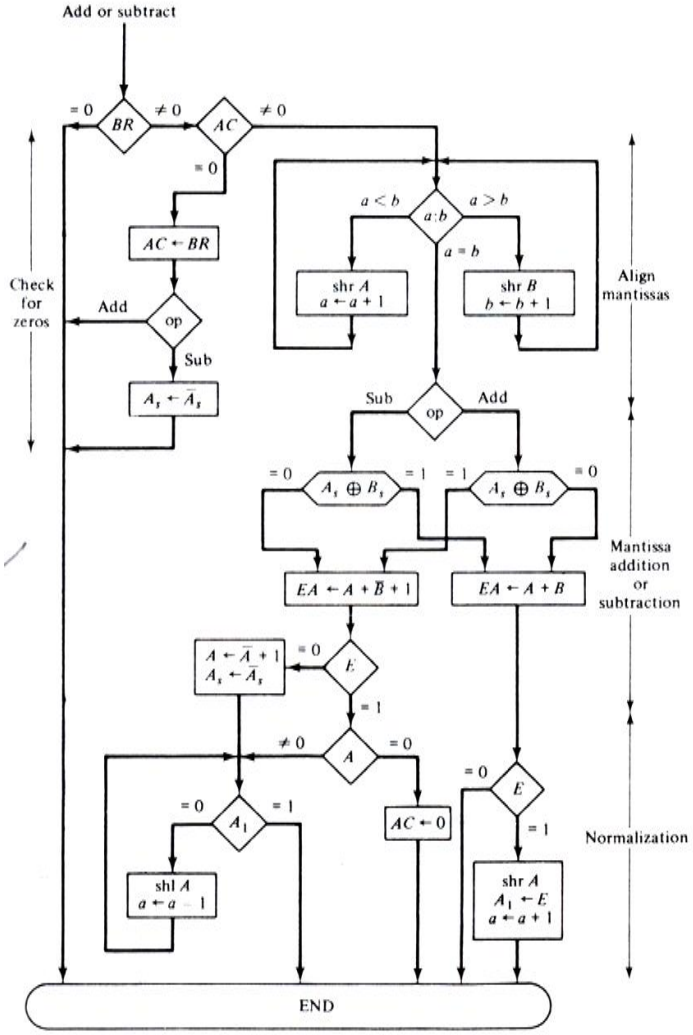
1. Check for zeros
2. Align the mantissas
3. Add or subtract the mantissas
4. Normalize the result

A floating–point number that is zero cannot be normalized. If this number is used during the computation, the result may also be zero. Instead of checking for zeros during the normalization process we check for zeros at the beginning and terminate the process if necessary. The alignment of the mantissas must be carried out prior to their operation. After

the mantissas are added or subtracted, the result may be normalized. The normalization procedure ensures that the result is normalized prior to its transfer to memory.

The flowchart for adding or subtracting two floating–point binary numbers is shown in Fig.5.14. If BR is equal to zero, the operation is terminated, with the value in the AC being the result. If AC is equal to zero, we transfer the content of BR into AC and also complement its sign if the numbers are to be subtracted. If neither number is equal to zero, we proceed to align the mantissas.

Figure 5.14 Addition and subtraction of floating point numbers



The magnitude comparator attached to exponents a and b provides three outputs that indicate their relative magnitude. If the two exponents are equal, we go to perform the arithmetic operation. If the exponents are not equal, the mantissa having the smaller exponent is shifted to the right and its exponent incremented. This process is repeated until the two exponents are equal.

The addition and subtraction of the two mantissas is identical to the fixed - point addition and subtraction of the two mantissas is identical to the fixed - point addition and subtraction algorithm. The magnitude part is added or subtracted depending on the operation and the

signs of the two mantissas. If an overflow occurs when the magnitudes are added, it is transferred into flip–flop E. If E is equal to 1, the bit is transferred into A_1 and all other bits of A are shifted right. The exponent must be incremented to maintain the correct number. No underflow may occur in this case because the original mantissa that was not shifted during the alignment was already in a normalized position.

If the magnitudes were subtracted, the result may be zero or may have an underflow. If the mantissa is zero, the entire floating–point number in the AC is made zero. Otherwise, the mantissa must have at least one bit that is equal to 1. The mantissa has an underflow if the most significant bit in position A_1 is 0. In the case, the mantissa is shifted left and the exponent decremented. The bit in A_1 is checked again and the process is repeated until it is equal to 1. When A_1=1, the mantissa is normalized and the operation is completed.

Multiplication

The multiplication of the mantissas is performed in the same way as in fixed–point to provide a double – precision product. The double–precision answer is used in fixed–point numbers to increase the accuracy of the product. In floating–point, the range of a single precision mantissa combined with the exponent is usually accurate enough so that only single precision numbers are maintained. Thus the half most significant bits of the mantissa product and the exponent will be taken together to form a single precision floating–point product.

The multiplication algorithm can be subdivided into four parts.

1. Check for zeros
2. Add the exponents
3. Multiply the mantissas
4. Normalize the product

Steps 2 and 3 can be done simultaneously if separate adders are available for the mantissas and exponents.

The two operands are checked to determine if they contain a zero. If either operand is equal to zero, the product in the AC is set to zero and the operation is terminated. If neither of the operands is equal to zero, the process continues with the exponent addition.

The exponent of the multiplier is in q and the adder is between exponents a and b. It is necessary to transfer the exponents from q to a, add the two exponents, and transfer the sum into a. Since both exponents are biased by the addition of a constant, the exponent sum will have double this bias. The correct biased exponent for the product is obtained by subtracting the bias number from the sum.

The multiplication of the mantissas is done as in the fixed – point case with the product residing in A and Q. Overflow cannot occur during multiplication, so there is not need to check for it.

The product may have an underflow, so the most significant bit in A is checked. If it is a 1, the product is already normalized. If it is a 0, the mantissa in AQ is shifted left and the exponent decremented. Note that only one normalization shift is necessary. The multiplier and multiplicand were originally normalized and contained fractions. The smallest normalized operand is 0.1, so the smallest possible product is 0.01. Therefore, only one leading zero may occur.

Although the low – order half of the mantissa is in Q, we do not use it for the floating – point product. Only the value in the AC is taken as the product.

Floating – point division requires that the exponents be subtracted and the mantissas divided. the mantissa division is done as in fixed – point except that the dividend has a single – precision mantissa that is placed in the AC. For integer representation, a single – precision dividend must be placed in register Q and register A must be cleared. The zeros in A are to the left of the binary point and have no significance. In fraction representation, a single – precision dividend is placed in register A and register Q is cleared. The zeros in Q are to the right of the binary point and have no significance.

If the dividend is greater than or equal to the divisor, the dividend fraction is shifted to the right and its exponent incremented by 1.

The division algorithm can be subdivided into five parts

1. Check for zeros
2. Initialize registers and evaluate the sign
3. Align the dividend
4. Subtract the exponents
5. Divide the mantissas

The two operands are checked for zero. If the divisor is zero, it indicates an attempt to divide by zero, which is an illegal operation. The operation is terminated with an error message. An alternative procedure would be to set the quotient in QR to the most positive number possible (if the dividend is positive) or to the most negative possible (if the dividend is negative). If the dividend in AC is zero, the quotient in QR is made zero and the operation terminates.

If the operands are not zero, we proceed to determine the sign of the quotient and store it in Q_s. The sign of the dividend in A_s is left unchanged to be the sign of the remainder. The Q register is cleared and the sequence counter SC is set to a number equal to the number of bits in the quotient.

The dividend alignment is similar to the divide – overflow check in the fixed – point operation. The proper alignment requires that the fraction dividend be smaller than the divisor. The two fractions are compared by a subtraction test. The carry in E determines their relative magnitude. The dividend fraction is restored to its original value by adding the divisor. If $A \geq B$, it is necessary to shift A once to the right and increment the dividend exponent. Since both operands are normalized, this alignment ensures that $A < B$.

The divisor exponent is subtracted from the dividend exponent. Since both exponents were originally biased, the subtraction operation gives the difference without the bias. The bias is then added and the result transferred into q because the quotient is formed in QR.

The magnitudes of the mantissas are divided as in the fixed–point case. After the operation, the mantissa quotient resides in Q and the remainder in A. The floating–point quotient is already normalized and resides in QR. The exponent of the remainder should be the same as the exponent of the dividend. The binary point for the remainder mantissa lies (n –1) positions to the left of A1. the remainder can be converted to a normalized fraction by subtracting n -1 from the dividend exponent and by shift and decrement until the bit in A1 is equal to 1.

UNIT V

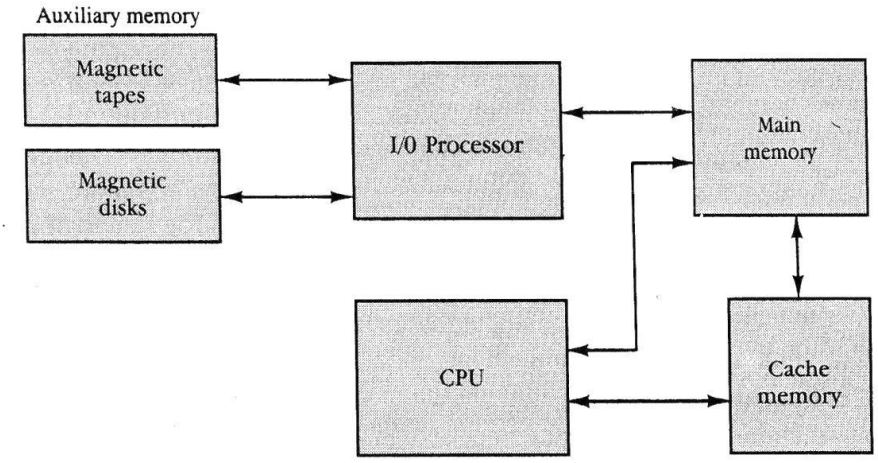
6. MEMORY ORGANIZATION

6.1 Memory Hierarchy

The memory unit is an essential component in any digital computer since it is needed for storing programs and data. Most general–purpose computers would run more efficiently if they were equipped with additional storage beyond the capacity of the main memory. There is just not enough space in one memory unit to accommodate all the programs used in typical computer. The memory unit that communicates directly with the CPU is called the main memory. Devices that provide backup storage are called auxiliary memory. The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. They are used for storing system programs, large data files, and other backup information.

Figure 6.1 illustrates the components in a typical memory hierarchy. At the bottom of the hierarchy are the relatively slow magnetic tapes used to store removable files. Next are the magnetic disks used as backup storage. The main memory occupies a central position by being able to communicate directly with the CPU and with auxiliary memory devices through an I/O processor. When programs not residing in main memory are needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space for currently used programs and data.

Figure 6.1 Memory Hierarchy in a computer system



A special very–high–speed memory called a cache is sometimes used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate. The cache memory is employed in computer systems to compensate for the speed differential between main memory access time and processor logic. CPU logic is usually

faster than main memory access time, with the result that processing speed is limited primarily by the speed of main memory. The cache is used for storing segments of programs currently being executed in the CPU and temporary data frequently need in the present calculations.

While the I/O processor manages data transfers between auxiliary memory and main memory, the cache organization is concerned with the transfer of information between main memory and CPU. As the storage capacity of the memory increases, the cost per bit for storing binary information decreases and the access time of the memory becomes longer. The auxiliary memory has a large storage capacity, is relatively inexpensive, but has low access speed compared to main memory. The cache memory is very small, relatively expensive, and has very high access speed. Thus as the memory access speed increases, so does its relative cost.

Auxiliary and cache memories are used for different purposes. The cache holds those parts of the program and data that are most heavily used, while the auxiliary memory holds those parts that are not presently used by the CPU. Moreover, the CPU has direct access to both cache and main memory but not to auxiliary memory. The transfer from auxiliary to main memory is usually „pp done by means of direct memory access of large blocks of data.

CPU process a multiprogramming number of independent programs concurrently. This concept, called multiprogramming, refers to the existence of two or more programs in different parts of the memory hierarchy at the same time. In this way it is possible to keep all parts of the computer busy by working with several programs in sequence. In a multiprogramming system, when one program is waiting for input or output transfer, there is another program ready to utilize the CPU.

Computer programs are sometimes too long to be accommodated in the total space available in main memory. Moreover, a computer system uses many programs and all the programs cannot reside in main memory at all times. A program with its data normally resides in auxiliary memory. When the program or a segment of the program is to be executed, it is transferred to main memory to be executed by the CPU. It is the task of the operating system to maintain in main memory a portion of this information that is currently active. The part of the computer system that supervises flow of information between auxiliary memory and main memory is called the memory management system.

6.2 Main Memory

The main memory is the central storage unit in a computer system. It is a relatively large and fast memory used to store programs and data. The main memory are memory available in two possible operating modes, static and dynamic. The static R consists essentially of internal flip-flops that store the binary information. The stored information remains valid as long as power is applied to the unit. The dynamic RAM stores the binary information in the form of electric charges that are applied to capacitors. The capacitors are provided inside the chip by MOS transistors. The stored charge on the capacitors tend to discharge with time and the capacitors must be periodically recharged by refreshing the dynamic memory. Refreshing is done by cycling through the words every few milliseconds to restore the decaying charge. The dynamic RAM offers reduced power consumption and larger storage capacity in a single memory chip. The static RAM is easier to use and has shorter read and write Cycles.

RAM was used to refer to a random-access memory, but now it is used to designate a read/write memory to distinguish it from a read-only memory, although ROM is also random access. RAM is used for storing the bulk of the programs and data that are subject to change.

ROM is used for storing programs that are permanently resident in the computer and for tables of constants that do not change in value once the production of the computer is completed.

The ROM portion of main memory is needed for bootstrap loader storing an initial program called a bootstrap loader. The bootstrap loader is a program whose function is to start the computer software operating when power is turned on. Since RAM is volatile, its contents are destroyed when power is turned off. The contents of ROM remain unchanged after power is c turned off and on again. The startup of a computer consists of turning the power on and starting the execution of an initial program. Thus when power is turned on, the hardware of the computer sets the program counter to the first address of the bootstrap loader. The bootstrap program loads a portion of the operating system from disk to main memory and control is then transferred to the operating system, which prepares the computer for general use.

RAM and ROM Chips

A RAM chip is better suited for communication with the CPU if it has one or more control inputs that select the chip only when needed. Another common feature is a bidirectional data bus that allows the transfer of data either from memory to CPU during a read operation, or from CPU to memory during a write operation. A bidirectional bus can be constructed with three-state buffers. A three-state buffer output can be placed in one of three possible states: a signal equivalent to logic 1, a signal equivalent to logic 0, or a high-impedance state. The logic 1 and 0 are normal digital signals. The high-impedance state behaves like an open circuit, which means that the output does not carry a signal and has no logic significance.

The capacity of the memory is 128 words of eight bits (one byte) per word. This requires a 7-bit address and an 8-bit bidirectional data bus. The read and write inputs specify the memory operation and the two chips select (CS) control inputs are for enabling the chip only when it is selected by the microprocessor. The availability of more than one control input to select the chip facilitates the decoding of the address lines when multiple chips are used in the microcomputer. The read and write inputs are sometimes combined into one line labeled R/W.

The function table listed in Fig. 6.2(b) specifies the operation of the RAM chip. The unit

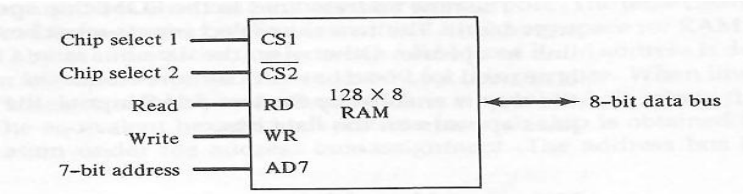
$\overline{\text{CS2}}$

is in operation only when CS1=1 and CS2 = 0. The bar on top of the second select variable indicates that this input is enabled when it is equal to 0. If the chip select inputs are not enabled, or if they are enabled but the read or write inputs are not enabled, the memory is inhibited and its data bus is in a high-impedance state.

$\overline{\text{WR}}$

When CS1 = 1 and CS2 = 0, the memory can be placed in a write or read mode. When the WR input is enabled, the memory stores a byte from the data bus into a location specified by theaddress input lines. When the RD input is enabled, the content of the selected byte is placed into the data bus.

Figure 6.2 Typical RAM chip



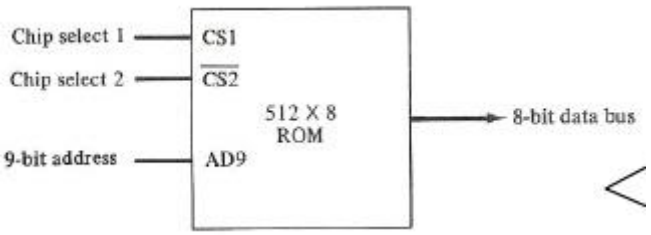
(a) Block diagram

CS1	$\overline{CS2}$	RD	WR	Memory function	State of data bus
0	0	×	×	Inhibit	High-impedance
0	1	×	×	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	×	Read	Output data from RAM
1	1	×	×	Inhibit	High-impedance

(b) Function Table

A ROM chip is organized externally in a similar manner. However, since a ROM can only read, the data bus can only be in an output mode. The block diagram of a ROM chip is shown in Fig. 12.3. For the same-size chip, it is possible to have more bits of ROM than of RAM, because the internal binary cells in ROM occupy less space than in RAM.

Figure 6.3 Typical ROM chip



The nine address lines in the ROM chip specify any one of the 512 bytes stored in it. The

two chip select inputs must be CS1 = 1 and $\overline{CS2}$ = 0 for the unit to operate. Otherwise, the data bus is in a high-impedance state. There is no need for a read or write control because the unit can only read.

Memory Address Map

The designer of a computer system must calculate the amount of memory required for the particular application and assign it to either RAM or ROM. The interconnection between memory and processor is then established from knowledge of the size of memory needed and the type of RAM and ROM chips available. The addressing of memory can be established by means of a table that specifies the memory address assigned to each chip. The table, called a memory address map, is a pictorial representation of assigned address space for each chip in the system.

The memory address map for this configuration is shown in Table 6-1. The component column specifies whether a RAM or a ROM chip is used. The hexadecimal address column assigns a range of hexadecimal equivalent addresses for each chip. The address bus lines are listed in the third column. Although there are 16 lines in the address bus, the table shows only 10 lines because the other 6 are not used in this example and are assumed to be zero. Thesmall x's under the address bus lines designate those lines that must be connected to the address inputs in each chip. The RAM chips have 128 bytes and need seven address lines. The ROM chip has 512 bytes and needs 9 address lines. The x's are always assigned to the low-order bus lines: lines 1 through 7 for the RAM and lines 1 through 9 for the ROM. It is now necessary to distinguish between four RAM chips by assigning to each a different address. For this particular example we choose bus lines 8 and 9 to represent four distinct binary combinations. Note that any other pair of unused bus lines can be chosen for this purpose. The table shows that the nine low-order bus lines constitute a memory space for RAM equal to 2^9= 512 bytes. The distinction between a RAM and ROM address is done with another bus line. Here we choose line 10 for this purpose. When line 10 is 0, the CPU selects a RAM, and when this line is equal to 1, it selects the ROM.

Table 6-1 Memory Address Map for Microprocomputer

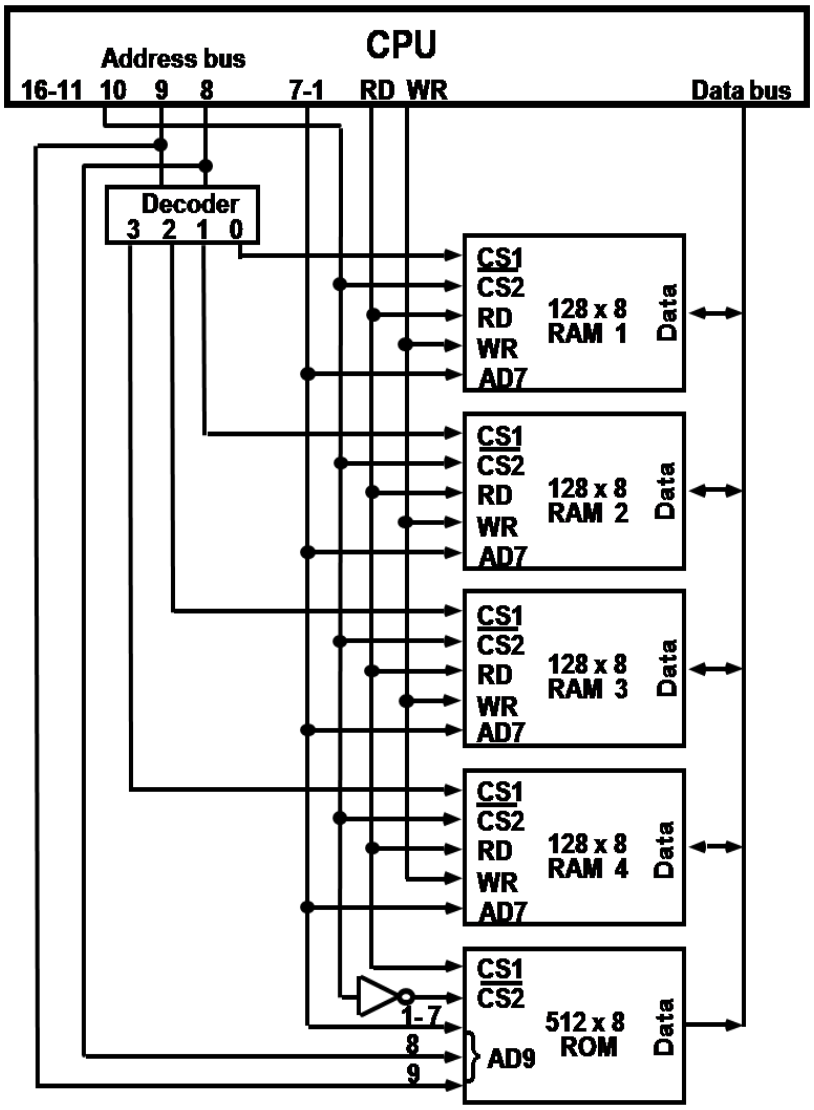
Component	Hexa address	Address bus									
		10	9	8	7	6	5	4	3	2	1
RAM 1	0000 - 007F	0	0	0	x	x	x	x	x	x	x
RAM 2	0080 - 00FF	0	0	1	x	x	x	x	x	x	x
RAM 3	0100 - 017F	0	1	0	x	x	x	x	x	x	x
RAM 4	0180 - 01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200 - 03FF	1	x	x	x	x	x	x	x	x	x

Memory Connection to CPU

RAM and ROM chips are connected to a CPU through the data and address buses. The low-order lines in the address bus select the byte within the chips and other lines in the address bus select a particular chip through its chip select inputs. This configuration gives a memory capacity of 512 bytes of RAM and 512 bytes of ROM.

The selection between RAM and ROM is achieved through bus line 10. The RAMs are selected when the bit in this line is 0, and the ROM when the bit is 1.

Figure 6.4 Memory connection to the CPU



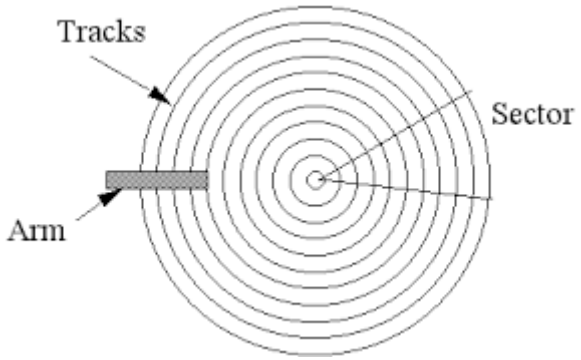
6.3 Auxiliary Memory

The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. Other components used, but not as frequently, are magnetic drums, magnetic bubble memory, and optical disks. The important characteristics of any device are its access mode, access time, transfer rate, capacity, and cost. The average time required to reach a storage location in memory and obtain its contents is called the access time. In electromechanical devices with moving parts such as disks and tapes, the access time consists of a seek time required to position the read-write head to a location and a transfer time required to transfer data to or from the device. Because the seek time is usually much longer than the transfer time, auxiliary storage is organized in records or blocks. A record is a specified number of characters or words. Reading or writing is always done on entire records. The transfer rate is the number of characters or words that the device can transfer per second, after it has been positioned at the beginning of the record.

Magnetic Disks

A magnetic disk is a circular plate constructed of metal or plastic coated with magnetized material. Often both sides of the disk are used and several disks may be stacked on one spindle with read/write heads available on each surface. All disks rotate together at high speed and are not stopped or started for access purposes. Bits are stored in the magnetized surface in spots along concentric circles called tracks. The tracks are commonly divided into sections called sectors. In most systems, the minimum quantity of information which can be transferred is a sector.

Figure 6.5 Magnetic disk



Some units use a single read/write head for each disk surface. In this type of unit, the track address bits are used by a mechanical assembly to move the head into the specified track position before reading or writing. In other disk systems, separate read/write heads are provided for each track in each surface. The address bits can then select a particular track electronically through a decoder circuit. This type of unit is more expensive and is found only in very large computer systems.

Permanent timing tracks are used in disks to synchronize the bits and recognize the sectors. A disk system is addressed by address bits that specify the disk number, the disk surface, the sector number and the track within the sector. After the read/write heads are positioned in the specified track, the system has to wait until the rotating disk reaches the specified sector under the read/write head. Information transfer is very fast once the beginning of a sector has been reached. Disks may have multiple heads and simultaneous transfer of bits from several tracks at the same time.

A track in a given sector near the circumference is longer than a track near the center of the disk. If bits are recorded with equal density, some tracks will contain more recorded bits than others. To make all the records in a sector of equal length, some disks use a variable recording density with higher density on tracks near the center than on tracks near the circumference. This equalizes the number of bits on all tracks of a given sector.

Disks that are permanently attached to the unit assembly and cannot be removed by the occasional user are called hard disks. A disk drive with removable disks is called a floppy disk. The disks used with a floppy disk drive are small removable disks made of plastic coated with magnetic recording material. There are two sizes commonly used, with diameters of 5.25 and 3.5 inches. The 3.5-inch disks are smaller and can store more data than can the 5.25-inch disks. Floppy disks are extensively used in personal computers as a medium for distributing software to computer users.

Magnetic Tape

A magnetic tape transport consists of the electrical, mechanical, and electronic components to provide the parts and control mechanism for a magnetic-tape unit. The tape itself is a strip of plastic coated with a magnetic recording medium. Bits are recorded as magnetic spots on the tape along several tracks. Usually, seven or nine bits are recorded simultaneously to form a character together with a parity bit. Read/write heads are mounted one in each track so that data can be recorded and-read as a sequence of characters.

Magnetic tape units can he stopped, started to move forward or in reverse, or can be rewound. However, they cannot be started or stopped fast enough between individual characters. For this reason, information is recorded in. blocks referred to as records. Gaps of unrecorded tape are inserted between records where the tape can be stopped. The tape starts moving while in a gap and attains its constant speed by the time it reaches the next record. Each record on tape has an identification bit pattern at the beginning and end. By reading the bit pattern at the beginning, the tape control identifies the record number. By reading the bit pattern at the end of the record, the control recognizes the beginning of a gap. A tape unit is addressed by specifying the record number and the number of characters in the record. Records may be of fixed or variable length.

6.4 Associative Memory

Many data-processing applications require the search of items in a table stored in memory. The established way to search a table is to store all items where they can be addressed in sequence. Many search algorithms have been developed to minimize the number of accesses while searching for an item in a random or sequential access memory.

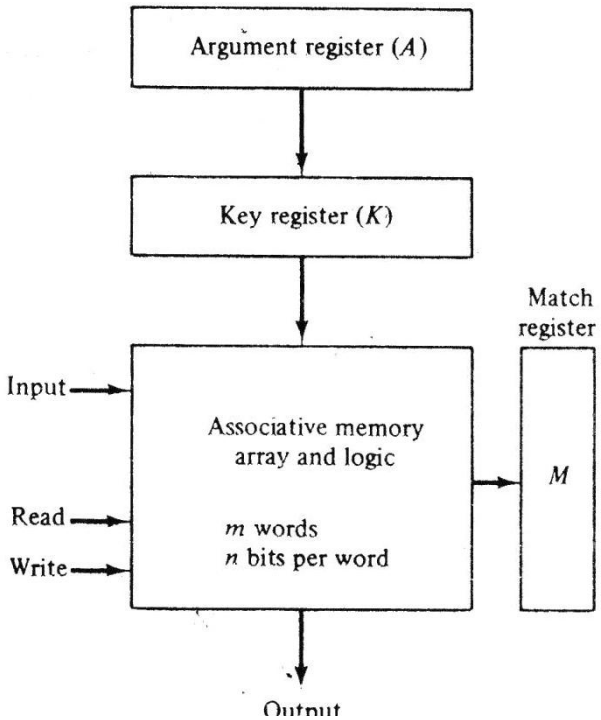
A memory unit accessed by content is called an associative memory or content addressable memory (CAM). This type of memory is accessed simultaneously and in parallel on the basis of data content rather than by specific address or location. When a word is written in an associative memory, no address is given. The memory is capable of finding an empty unused location to store the word.

Hardware Organization

The block diagram of an associative memory is shown in Fig. 6.6. It consists of a memory array and logic for m words with n bits per word. The argument register A and key register K each have n bits, one for each bit of a word. The match register M has m bits, one for each memory word. Each word in memory is compared in parallel with the content of the argument register. The words that match the bits of the argument register set a corresponding bit in the match register. After the matching process, those bits in the match register that have been set indicate the fact that their corresponding words have been matched. Reading is accomplished by a sequential access to memory for those words whose corresponding bits in the match register have been set.

The key register provides a mask for choosing a particular field or key in theargument word. The entire argument is compared with each memory word if the key register contains all l's. Thus the key provides a mask or identifying piece of information which specifies how the reference to memory is made.

Figure 6.6 Block diagram of associative memory



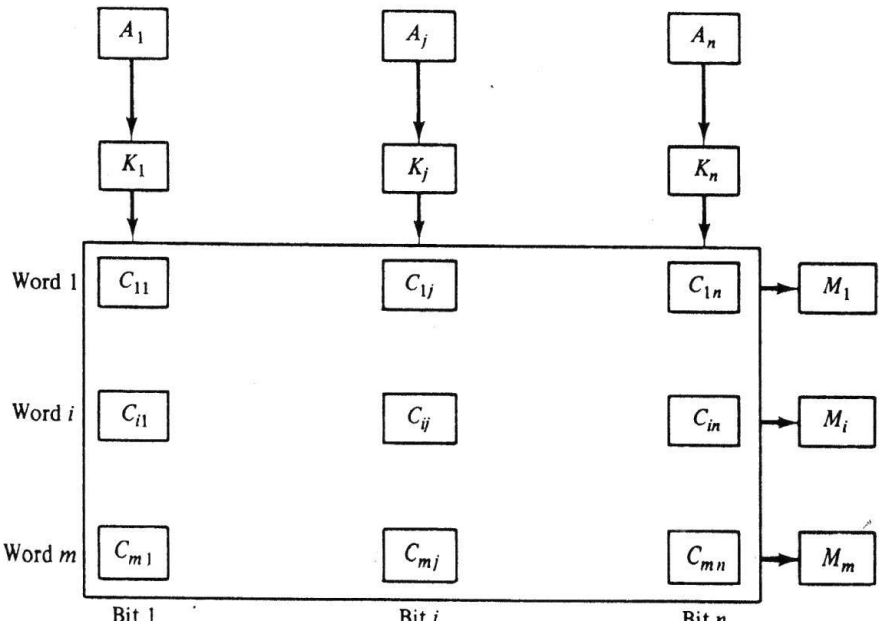
To illustrate with a numerical example, suppose that the argument register A and the key register K have the bit configuration shown below. Only the three leftmost bits of A are compared with memory words because K has l's in these positions.

A	101 111100	
K	111 000000	
Word 1	100 111100	no match
Word 2	101 000001	match

Word 2 matches the unmasked argument field because the three leftmost bits of the argument and the word are equal.

The relation between the memory array and external registers in an associative memory is shown in Fig. 6.7. The cells in the array are marked by the letter C with two subscripts. The first subscript gives the word number and the second specifies the bit position in the word. Thus cell C_{ij} is the cell for bit j in word i. A bit A_j in the argument register is compared with all the bits in column j of the array provided that $K_j = 1$. This is done for all columns j = 1, 2, . . . , n. If a match occurs between all the unmasked bits of the argument and the bits in word i , the corresponding bit in the match register is set to 1. If one or more unmasked bits of the argument and the word do not match, M_i is cleared to 0.

Figure 6.7 Associative memory of m word, n c ells per word



Match Logic

The match logic for each word can be derived from the comparison algorithm s or two binary numbers. First, we *neglect t*he key bits and compare the argument in A with the bits stored in the cells of the words. Word i is equal to the argument in A if $A_j=F_{ij}$for j = 1, 2, . . . , n . Two bits are equal if they are both 1 or both 0. The equality of two bits can be expressed logically by the Boolean function

$$x_j = A_j F_{ij} + A_j F_{ij}'$$

Wherex_j = 1 if the pair of bits in position j are equal; otherwise, x_j = 0.

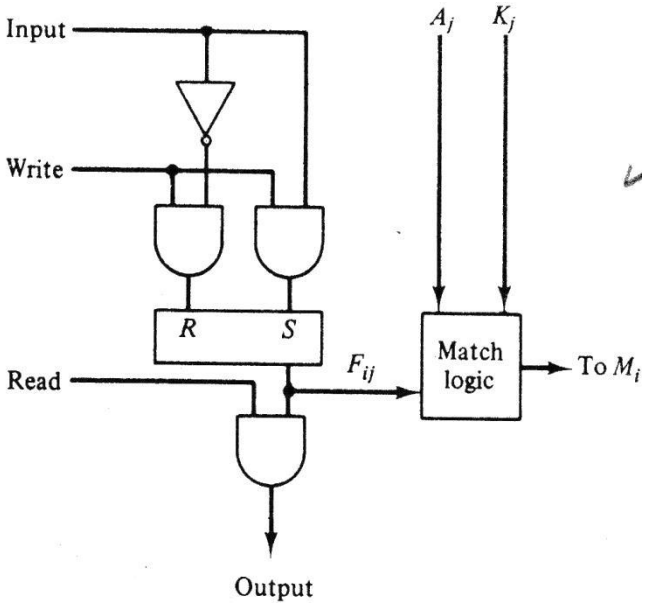
For a word i to be equal to the argument in A we must have all x_j variables equal to 1. This is the condition for setting the corresponding match bit M_i to 1. The Boolean function for this condition is

Mi = $x_1 x_2 x_3 \ldots\ldots\ldots x_n$

and constitutes the AND operation of all pairs of matched bits in a word. Figure 6.8 One cell of associative memory. The requirement is that if K_j = 0, the corresponding bits of A_j and F_{ij}need no compatriot. This requirement is achieved b ORing each term with K_j‘ thus:

$$x_j + K_j' = \{x_j \;\; if\; K_j = 1$$
$$\{1 \;\; if\; K_j = 0$$

Figure 6.8 One cell of Associative memory



If we substitute the original definition of xj, the Boolean function above can be expressed as follows:

$$M_i = \prod_{j=1}^{n} (A_j F_{ij} + A_j' F_{ij}' + K_j')$$

where II is a product symbol designating the AND operation of all n terms. We need *m* such functions, one for each word $i = 1, 2, 3, \ldots, m$.

if the key register contains all 0's, output M_i will be a 1 irrespective of the value of A or the word. This occurrence must be avoided during normal operation.

Read Operation

If more than one word in memory matches the unmasked argument field, all the matched words will have l's in the corresponding bit position of the match register. The matched words are read in sequence by applying a read signal to each word line whose corresponding M_i, bit is a 1.In most applications, the associative memory stores a table with no two identical items under a given key. By connecting output M, directly to the read line in the same word position (instead of the M register), the content of the matched word will be presented automatically at the output lines all-zero output will indicate that no match occurred and that the searched item is not available in memory.

Write Operation

An associative memory must have a write capability for storing the information to be searched. If the entire memory is loaded with new information at once prior to a search operation then the writing can be done by addressing each location in sequence instead of having m address lines, one for each word in memory, the number address lines can be reduced by the decoder to d lines, where m = 2^d.If unwanted words have to be deleted and new words inserted one at a time, there is a need for a special register to distinguish between active and inactive words. This register, sometimes called a tag register, would have as many bits as there are words in the memory. A word is deleted from memory by clearing its tag bit

to 0. Words are stored in memory by scanning the tag register until the first 0 bit is encountered. The words that have a tag bit of 0 must be masked (together with the K, bits) with the argument word so that only active words are compared.

6.5 Cache Memory:

Analysis of a large number of typical programs has shown that the references memory at any given interval of time tend to be confined within a few localized areas in memory. This phenomenon is known as the property of locality of reference. When a program loop is executed, the CPU repeatedly refers to the set of instructions in memory that constitute the loop. Every time a given subroutine is called, its set of instructions are fetched from memory. Thus loops and subroutines tend to localize the references to memory for fetching instructions. To a lesser degree, memory references to data also tend to be localized. Table-lookup procedures repeatedly refer to that portion in memory where the table is stored. Iterative procedures refer to common memory locations and array of numbers are confined within a local portion of memory. The result of all these observations is the locality of reference property, which states that over a short interval of time, the addresses generated by a typical program refer to a few localized areas of memory repeatedly, while the remainder of memory is accessed relatively infrequently.

If the active portions of the program and data are placed in a fast small memory, the average memory access time can be reduced. Such a fast small memory is referred to as a cache memory. The cache is the fastest component in the memory hierarchy and approaches the speed of CPU components.

The fundamental idea of cache organization is that by keeping the most frequently accessed instructions and data in the fast cache memory, the average memory access time will approach the access time of the cache. Although the cache is only a small fraction of the size of main memory, a large fraction of memory requests will be found in the fast cache memory because of the locality of reference property of programs.

The basic operation of the cache is as follows. When the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast memory. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word. A block of words containing the one just accessed is then transferred from main memory to cache memory. The block size may vary from one word to about 16 words adjacent to the one just accessed.

The performance of cache memory is frequently measured in terms of a quantity called hit ratio. When the CPU refers to memory and finds the word in cache, it is said to produce a hit. If the word is not found in cache, it is in main memory and it counts as a, miss, ratio of the number of hits divided by the total CPU references to memory (hits plus misses) is the hit ratio. Hit ratios of 0.9 and higher have been reported. This high ratio verifies the validity of the locality of reference property.

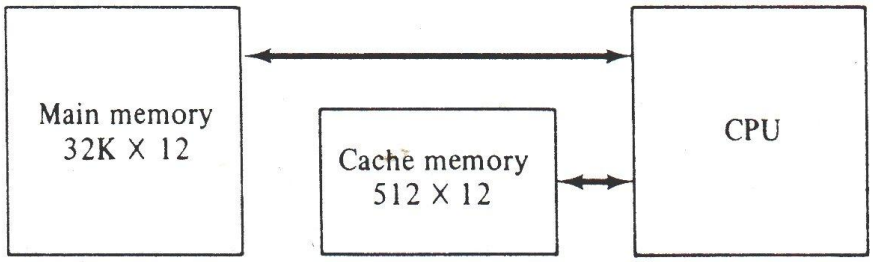
The average memory access time of a computer system can be improved considerably by use of a cached If the hit ratio is high enough so that most of the time the CPU accesses the cache instead of main memory, the average access time is closer to the access time of the fast cache memory.

The transformation of data from main memory to cache memory is referred to as a mapping process. Three types of mapping procedures are of practical interest when considering the organization of cache memory:

1. Associative mapping
2. Direct mapping
3. Set-associative mapping

The main memory can store 32K words of 12 bits each. The cache is capable of storing 512 of these words at any given time. For every word stored in cache, there is a duplicate copy in main memory. The CPU communicates with both memories. It first sends a 15-bit address to cache. If there is a hit, the CPU accepts the 12-bit data from cache. If there is a miss, the CPU reads the word from main memory and the word is then transferred to cache.

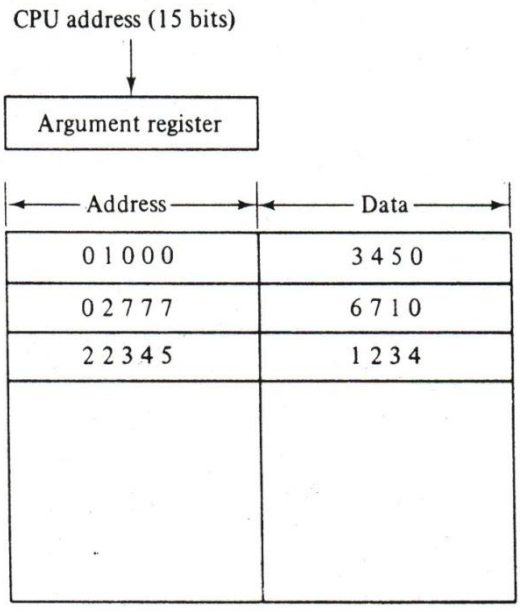
Figure 6.9 Example of cache memory



Associative Mapping

The associative memory stores both the address and content (data) of the memory word. This permits any location in cache to store any word from main memory. The diagram shows three words presently stored in the cache. The address value of 15 bits is shown as a five-digit octal number and its corresponding 12-bit word is shown as a four-digit octal number. A CPU address of 15 bits is placed in the argument register and the associative memory is searched for a matching address. If the address is found, the corresponding 12-bit data is read and sent to the CPU. If no match occurs, the main memory is accessed for the word. The address data pair is then transferred to the associative cache memory. If the cache is full an address data pair must be displaced to make room for a pair that is needed and not presently in the cache.

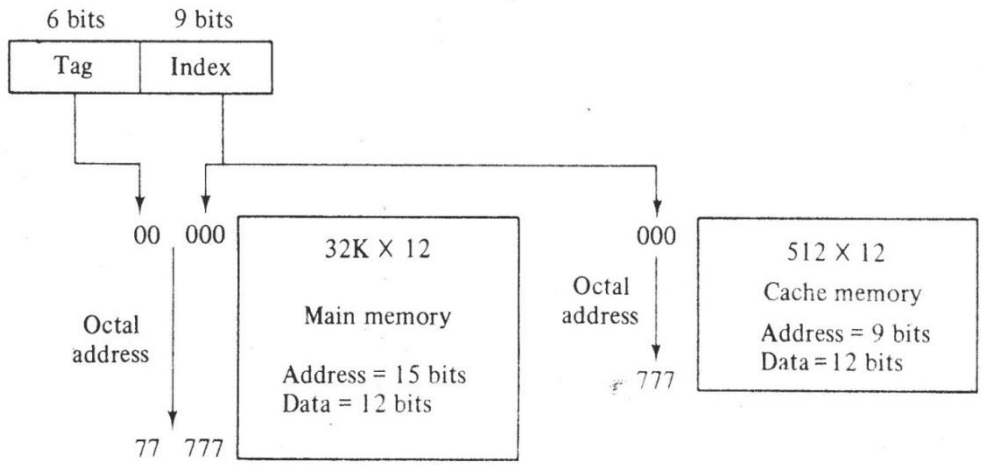
Figure 6.10 associative mapping cache(all numbers in octal)



Direct Mapping

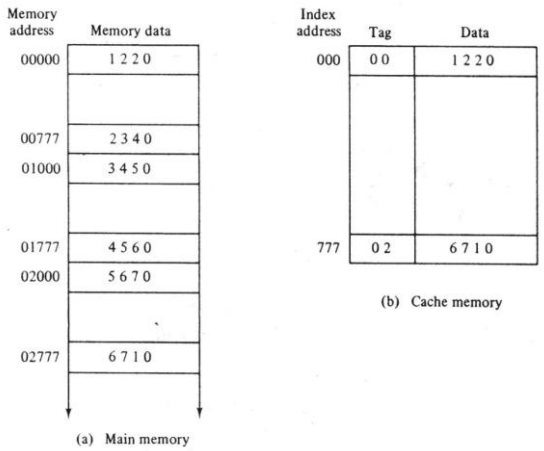
Associative memories are expensive compared to random-access memories because of the added logic associated with each cell. The CPU address of 15 bits is divided into two fields. The nine least significant bits constitute the index field and the remaining six bits form the tag field.

Figure 6.11 Addressing relationships between main and cache memories



In the general case, there are 2^k words in cache memory and 2^n words in main memory. The n-bit memory address is divided into two fields: k bits for the index field and n-k bits for the tag field. The direct mapping cache organization uses the n-bit address to access the main memory and the k-bit index to access the cache. When the CPU generates a memory request, the index field is used for the address to access the cache. The tag field of the CPU address is compared with the tag in the word read from the cache.

Figure 6.12 Direct mapping cache organization

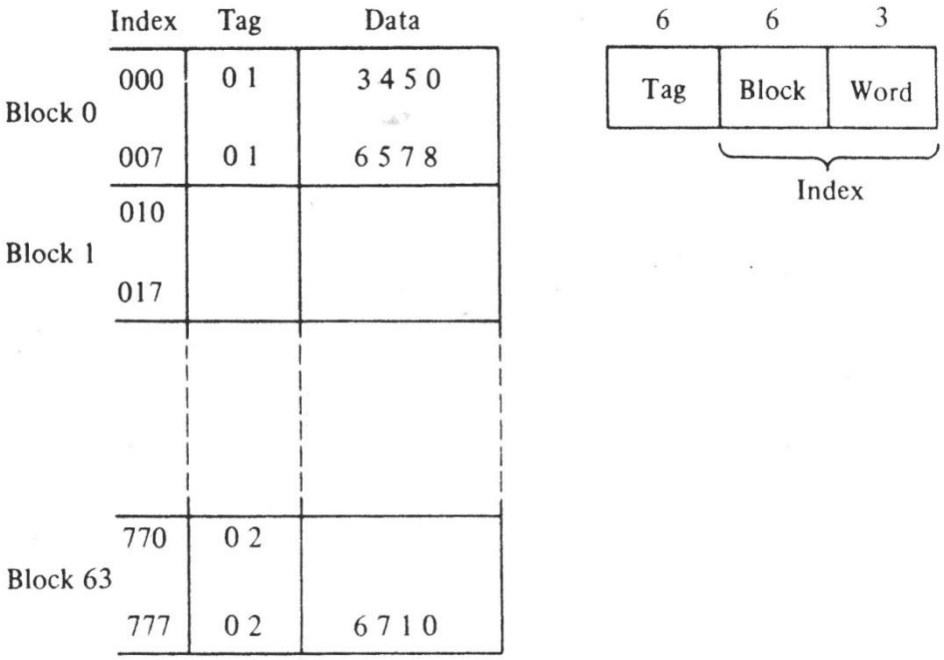


If the two tags match, there is a hit and the desired data word is in cache. If there is no match, there is a miss and the required word is read from main memory. It is then stored in the cache together with the new tag, replacing the previous value.

The word at address zero is presently stored in the cache (index = 000, tag = 00, data = 1220). Suppose that the CPU now wants to access the word at address 02000. The index address is 000, so it is used to access the cache. The two tags are then compared. The cache tag is 00 but the address tag is 02, which does not produce a match. Therefore, the main memory is accessed and the data word 5670 is transferred to the CPU. The cache he word at index address 000 is then replaced with a tag of 02 and data of 5670.

The index field is now divided into two parts: the block field and the word field. In a 512-word cache there are 64 blocks of 8 words each, since 64 x 8 = 512. The block number is specified with a 6-bit field and the word within the block is specified with a 3-bit field. The tag field stored within the cache is common to all eight words of the same block. Every time a miss occurs, an entire block of eight words must be transferred from main memory to cache memory.

Figure 6.13 Direct mapping cache with block size of 8 words



Set-Associative Mapping

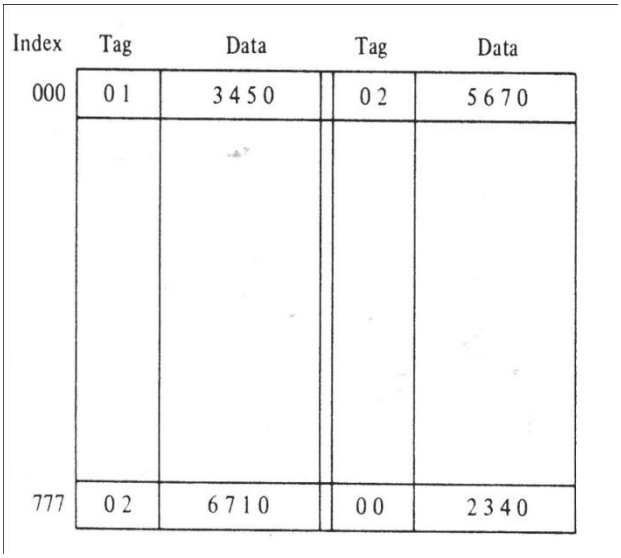
A third type of cache organization, called set-associative mapping, is an improvement over the direct-mapping organization in that each word of cache can store two or more words of memory under the same index address. Each data word is stored together with its tag and the number of tag data items in one word of cache is said to form a set. Each tag requires six bits and each data word has 12 bits, so the word length is (6 + 12) = 36 bits. An index address of nine bits can accommodate 512 words. Thus the size of cache memory is 512 × 36. It can accommodate 1024 words of main memory since each word of cache contains two data

words. In general, a set-associative cache of set size k will accommodate k words of main memory in each word of cache.

The octal numbers listed in Fig.6.14 are with reference to the main memory contents illustrated in Fig. 6.12(a). The words stored at addresses 01000 and 02000 of main memory are stored in cache memory at index address 000. Similarly, the words at addresses 02777 and 00777 are stored in cache at index address 777. When the CPU generates a memory request, the index value of the address is used to access the cache. The tag field of the CPU address is then compared with both tags in the cache to determine if a match occurs. The comparison logic is done by an associative search of the tags in the set similar to an associative memory search: thus the name "set-associative." The hit ratio will improve as the set size increases because more words with the same index but different tags can reside in cache.

The most algorithms common replacement algorithms used are: random replacement, firs -in, first-out (FIFO), and least recently used (LRU)

Figure 6.14 Two way Set associative mapping cache



Writing into Cache

When the CPU finds a word in cache during a read operation, the main memory is not involved in the transfer. However, if the operation is a write, there are two ways that the system can proceed.

The simplest and most commonly used procedure is to update main memory with every memory write operation, with cache memory being up-dated in parallel if it contains the word at the specified address. This is called the write-through method.

The second procedure is called the write-back method. In this method only the cache location is updated during a write operation. The location is then marked by a flag so that later when the word is removed from the cache it is copied into main memory.

The Cache Initialization

The cache is initialized when power is applied to the computer or when the main memory is loaded with a complete set of programs from auxiliary memory.

The cache is initialized by clearing all the valid bits to 0. The valid bit of a particular cache word is set to 1 the first time this word is loaded from main memory and stays set unless the cache has to be initialized again. The initialization condition has the effect of forcing misses from the cache until fills with valid data.

6.6 Virtual Memory

Virtual memory is a concept used in some large computer systems that permit the user to construct programs as though a large memory space were available, equal to the totality of auxiliary memory. Each address that is referenced by the CPU goes through an address mapping from the so-called virtual address to a physical address in main memory. Virtual memory is used to give programmers the illusion that they have a very large memory at their disposal, even though the computer actually has a relatively small main memory. A virtual memory system provides a mechanism for translating program-generated addresses into correct main memory locations. This is done dynamically, while programs are being executed in the CPU. The translation or mapping is handled automatically by the hardware by means of a mapping table.

Address Space and Memory Space

An address used by a programmer will be called a virtual address, and the set of such addresses the address space. An address in main memory is called a location or physical address. The set of such locations is called the memory space. Thus the address space is the set of addresses generated by programs as they reference instructions and data.

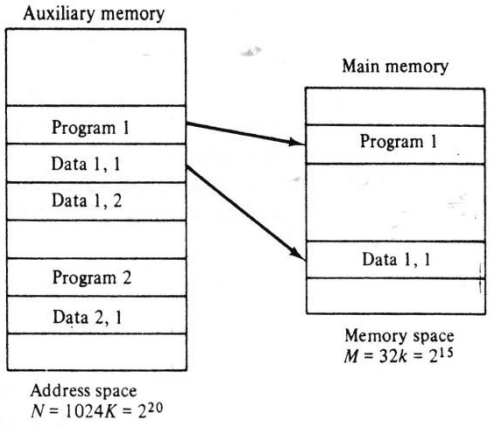
The memory space consists of the actual main memory locations directly addressable for processing. In most computers the address and memory spaces are identical. The address space is allowed to be larger than the memory space is allowed to be larger than the memory space computers with virtual memory.

Consider a computer with a main memory capacity of 32K words (K = 1024). Fifteen bits are needed to specify a physical address in memory since $32K = 2^{15}$. Suppose that the computer has available auxiliary memory for storing 2^{20} = 1024K words. Thus auxiliary memory has a capacity for storing information equivalent to the capacity of 32 main memories. Denoting the address space by N and the memory space by M, we then have for this example N = 1024K and M = 32K.

In a multiprogram computer system, programs and data are transferred to and from auxiliary memory and main memory based on demands imposed by the CPU. Suppose that program 1 is currently being executed in the CPU. Program 1 and a portion of its associated data are moved from auxiliary memory into main memory as shown in Fig. 6.15. Portions of programs and data need not be in contiguous locations in memory since information is being moved in and out, and empty spaces may be available in scattered locations in memory.

In a Virtual memory system, programmers are told that they have the total address space at their disposal. Moreover, the address field of the instruction code has a sufficient number of bits to specify all virtual addresses. In our example, the address field of an instruction code will consist of 20 bits but physical memory addresses must be specified with only 15 bits. Thus CPU, will reference instructions and data with a 20-bit address, but the information at this address must be taken from physical memory because access to auxiliary storage for individual words will be prohibitively long.

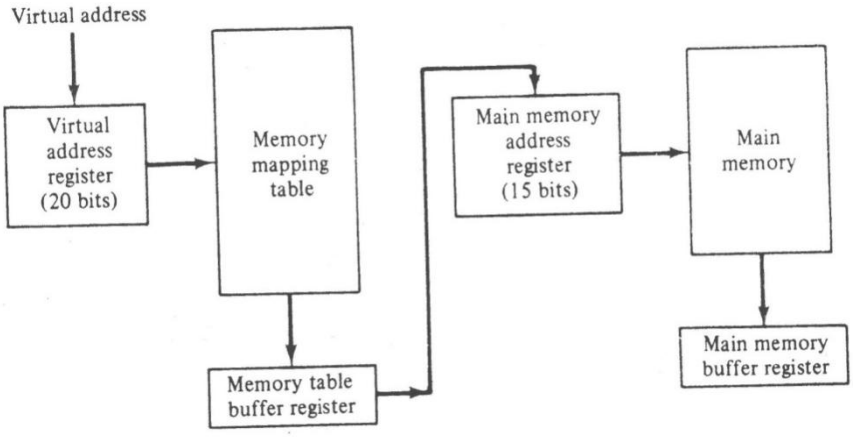
Figure 6.15 Relation between address and memory space in a virtual memory system



A table is then needed, as shown in Fig. 6.16, to map a virtual address of 20 bits to a physical address of 15 bits. The mapping is a dynamic operation, which means that every address is translated immediately as a word is referenced, by CPU.

The mapping table may be stored in a separate memory as shown in Fig.6.16 or in main memory. In the first case, an additional memory unit is required as well as one extra memory access time. In the second case, the table takes space from main memory and two accesses to memory are required with the pro ram running at half speed. A third alternative is to use an associative memory.

Figure 6.16 Memory table for mapping a virtual address



Address Mapping Using Pages.

The information in the address space and the memory space are each divided into groups of fixed size. The physical memory is broken down into groups of equal size called blocks, which may range from 64 to 4096 words each. The term page refers to groups of address space of the same size, a page refers to the organization of address space, while a block refers to t e organization of memory space. The programs are also considered to be split into pages. Portions of programs are moved from auxiliary memory to main memory in

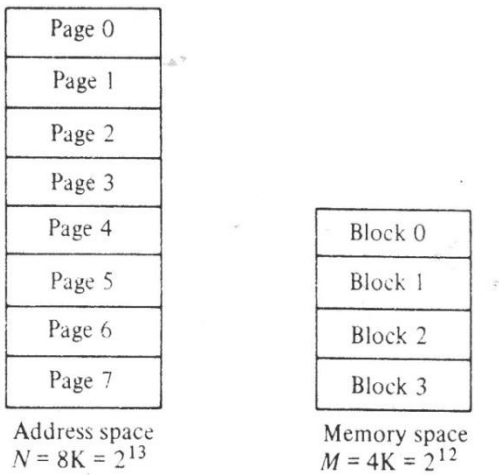
records equal to the size of a page. The term "page frame" is sometimes used to denote a block.

Consider computer with an address space of 8K and a memory space of 4K. If we split each into groups of 1K words we obtain eight pages and four blocks as shown in Fig.6.17. At any given time, up to four pages of address space may reside in main memory in any one of the four blocks.

The mapping from address space to memory space is facilitated if each virtual address is considered to be represented by two numbers: a page number address and a line within the page. In a computer with 2P words per page, p bits are used to specify a line address and the remaining high-order bits of the virtual address specify the page number. In the example of Fig. 6.17, a virtual address has 13 bits. Since each page consists of 2^{10} = 1024 words, the high-order three bits of a virtual address will specify one of the eight pages and the low-order 10 bits give the line address within the page. Note that the line address in address space and memory space is the same; the only mapping required is from a page number to a block number

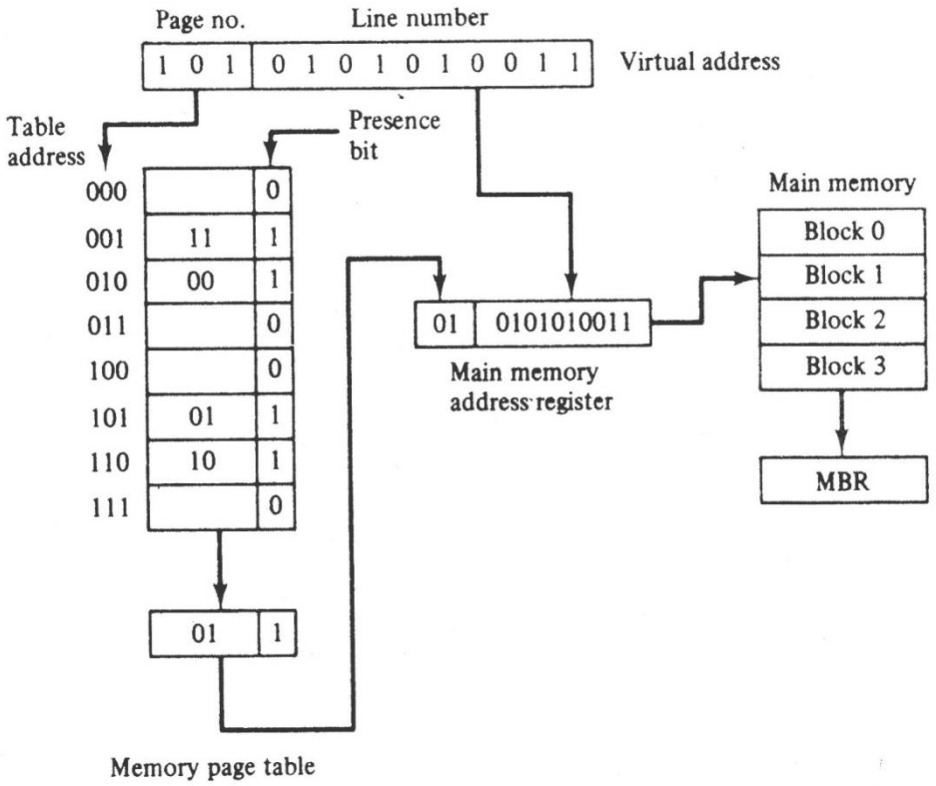
The organization of the memory mapping table in a paged system is shown in Fig. 6.18. The memory-page table consists of eight words, one for each page. The address in the page table denotes the page number and the content of the word gives the block number where that page is stored in main memory. The table shows that pages 1, 2, 5, and 6 are now available in main memory in blocks 3, 0, 1, and 2, respectively. A presence bit in each location indicates whether the page has been transferred from auxiliary memory into main memory. The CPU references a word in memory with a virtual address of 13 bits. The tree high-order bits of the virtual address specify a page number and also an address for the memory-page table. The content of the word in the memory page table at the page number address is read out into the memory table buffer register.

Figure 6.17 Address space and memory space split into groups of 1Kwords



If the presence bit is a 1, the block number thus read is transferred to the two high-order bits of the main memory address register. The line number from the virtual address is transferred into the 10 low-order bits of the memory address register. A read signal to main memory transfers the content of the word to the main memory buffer register ready to be used by the CPU.

Figure 6.18 Memory table in a page system



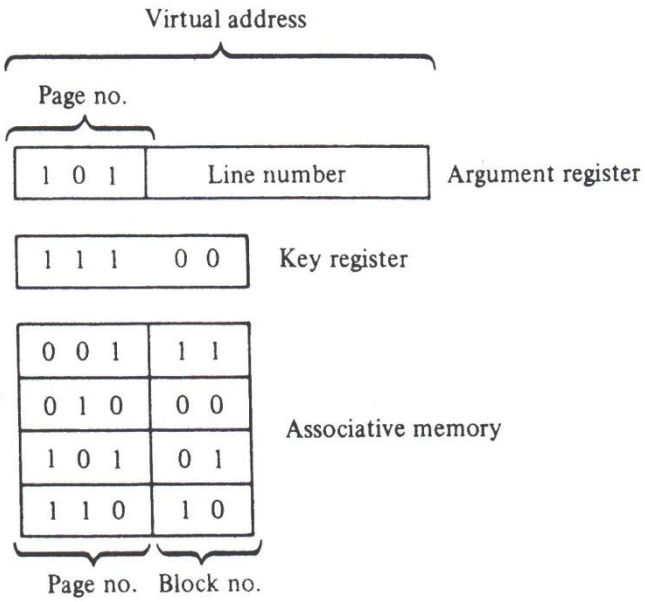
If the presence bit in the word read from the page table is 0, it signifies that the content of the word referenced by the virtual address does not reside in main memory. A call to the operating system is then generated to fetch the required page from auxiliary memory and place it into main memory before resuming computation.

Associative Memory Page Table

In general, a system with n pages and m blocks would require a memory-page table of n locations of which up to m blocks will be marked with block numbers and all others will be empty. As a second numerical example, consider an address space of 1024K words and memory space of 32K words. If each page or block contains 1K words, the number of pages is 1024 and the number of blocks 32. The capacity of the memory-page table must be 1024 words and only 32 locations may have a presence bit equal 1 to 1. At any given time, at least 992 locations will be empty and not in use.

A more efficient way to organize the page table would be to construct it with number of words equal to the number of blocks in main memory. In this way the size of the memory is reduced and each location is fully utilized. This method can be implemented by means of an associative memory with each word in memory containing a page number together with its corresponding block number.

Figure 6.19 An Associative memory page table



The page field in each word is compared with the page number in the virtual address. If a match occurs, the word is read from memory and its corresponding block number is extracted.

Consider again the case of eight pages and four blocks as in the example of fig. 6.18. Each entry in the associative memory array consists of two fields. The first three bits specify a field for storing the page number. The last two bits constitute a field for storing the block number. The virtual address is placed in the argument register. The page number bits in the argument register are compared with all page numbers in the page field of the associative memory. If the page number is found, the 5-bit word is read out from memory. The corresponding block number, being in the same word, is transferred to the main memory address register. If no match occurs, a call to the operating system is generated to bring the required page from auxiliary memory.

Page Replacement

The memory management software system handles all the software operations for the efficient utilization of memory space. It must decide (1) which page in main memory ought to be removed to make room for a new page, (2) when a new page is to be transferred from auxiliary memory to main memory, and (3) where the page is to be placed in main memory.

When a program starts execution, one or more pages are transferred into main memory and the page table is set to indicate their position. The program is executed from main memory until it attempts to reference a page that is still in auxiliary memory. This condition is called page fault. When page fault occurs, the execution of the present program is suspended until the required page is brought into main memory. Since loading a page from auxiliary memory to main memory is basically an I/O operation, the operating system assigns this task to the I/O processor. In the meantime, control is transferred to the next program in memory that is waiting to be processed in the CPU. Later, when the memory block has been assigned and the transfer completed, the original program can resume its operation.

When a page fault occurs in a virtual memory system, it signifies that the page referenced by the CPU is not in main memory. A new page is then transferred from auxiliary

memory to main memory. If main memory is full, it would be necessary to remove a page from a memory block to make room for the new page. The policy for choosing pages to remove is determined from the replacement algorithm that is used. The goal of a replacement policy is to try to r move the page least likely to be referenced in the immediate future.

Two of the most common replacement algorithms used are the first in, first-out (FIFO) and the least recently used (LRU). The FIFO algorithm selects for replacement the page that has been in memory the longest time. Each time a page is loaded into memory, its identification number is pushed into a FIFO stack. FIFO will be full whenever memory has no more empty blocks. When a new page must be loaded, the rage least recently brought in is removed. The page to be removed is easily determined because its identification number is at the top of the FIFO stack. The FIFO replacement policy has the advantage of being easy to implement. It has the disadvantage that under certain circumstances pages are removed and loaded fro-n memory too frequently).

The LRU policy is more difficult to implement but has been more attractive the assumption that the least recently used page is a better candidate for removal than the least recently loaded page as in FIFO. The LRU algorithm can be implemented by associating a counter with every page that is in main memory. When a page is referenced, its associated counter is set to zero. At fixed intervals of time, the counters associated with all pages presently in memory are incremented by 1. The least recently used page is the page with the highest count. The counters are often called aging registers, as their count indicates their age, that is, how long ago their associated pages have been referenced.

.

MODEL QUESTION PAPER
COMPUTER ARCHITECTURE

Max. Marks: 75
Time : 3 Hrs

PART A - (10X2=20 marks)
(Answer ALL the questions)

1. Convert the following decimal number 1936 to hexadecimal.
2. What is circular shift operation?
3. Define instruction code.
4. What is hardwired control?
5. What is a CPU? Write its major components.
6. Write the three types of CPU organizations.
7. Draw the block diagram of the hardware implementation for addition and subtraction.
8. Write parts of the division algorithm for floating point arithmetic operation.
9. Define multiprogramming.
10. Write short note on write-through.

PART B – (5x5=25 marks)
(Answer ALL the questions)

11.a. Write short notes on complements.

or

b. Describe Binary adder in detail.

12.a. Explain Stored program Organization.

or

b. Write the phases of the instruction cycle..

13.a. Explain any FIVE addressing modes in detail.

or

b. Write the major characteristics of CISC and RISC.

or

14.a. Describe Booth multiplication algorithm in detail.

or

b. Explain Addition and Subtraction for floating point arithmetic operation.

15.a. Write short notes on Main memory.

Or

b. Describe Virtual memory.

PART C – (3x10=30 marks)
(Answer any **THREE** of the following)

16. Explain Logic Microoperations in detail.
17. Describe Timing and Control unit with block diagram.
18. What is meant by Stack? Explain Stack Organization.
19. Describe Division Algorithms.
20. Explain in detail about Cache memory