Chapter 3 – Data Representation

Section 3.1 – Data Types

- Registers contain either data or control information
- Control information is a bit or group of bits used to specify the sequence of command signals needed for data manipulation
- Data are numbers and other binary-coded information that are operated on
- Possible data types in registers:
  - Numbers used in computations
  - Letters of the alphabet used in data processing
  - Other discrete symbols used for specific purposes
- All types of data, except binary numbers, are represented in binary-coded form

- 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
- The string of digits 724.5 represents the quantity

$$7 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 + 5 \times 10^{-1}$$

- The string of digits 101101 in the binary number system represents the quantity

$$1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 45$$

- $(101101)_2 = (45)_{10}$
- We will also use the octal (radix 8) and hexidecimal (radix 16) number systems

$$(736.4)_8 = 7 \times 8^2 + 3 \times 8^1 + 6 \times 8^0 + 4 \times 8^{-1} = (478.5)_{10}$$

$$(F3)_{16} = F \times 16^1 + 3 \times 16^0 = (243)_{10}$$

- Conversion from decimal to radix *r* system is carried out by separating the number into its integer and fraction parts and converting each part separately
- Divide the integer successively by *r* and accumulate the remainders
- Multiply the fraction successively by *r* until the fraction becomes zero

Figure 3-1 Conversion of decimal 41.6875 into binary.

Integer = 41

41	
20	1
10	0
5	0
2	1
1	0
0	1

$(41)_{10} = (101001)_2$

Fraction = 0.6875

0.6875
x 2
1.3750
x 2
0.7500
x 2
1.5000
x 2
1.0000

$(0.6875)_{10} = (0.1011)_2$

$(41.6875)_{10} = (101001.1011)_2$

- Each octal digit corresponds to three binary digits
- Each hexadecimal digit corresponds to four binary digits
- Rather than specifying numbers in binary form, refer to them in octal or hexadecimal and reduce the number of digits by 1/3 or ¼, respectively

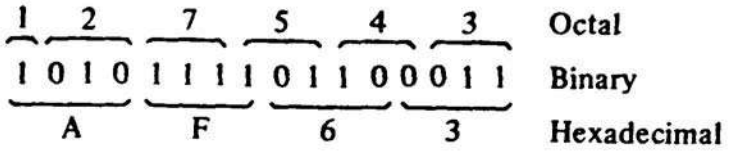


Figure 3-2 Binary, octal, and hexadecimal conversion.

TABLE 3-1 Binary-Coded Octal Numbers

Octal number	Binary-coded octal	Decimal equivalent	
0	000	0	↑
1	001	1	|
2	010	2	Code
3	011	3	for one
4	100	4	octal
5	101	5	digit
6	110	6	|
7	111	7	↓
10	001 000	8	
11	001 001	9	
12	001 010	10	
24	010 100	20	
62	110 010	50	
143	001 100 011	99	
370	011 111 000	248	

TABLE 3-2 Binary-Coded Hexadecimal Numbers

Hexadecimal number	Binary-coded hexadecimal	Decimal equivalent	
0	0000	0	
1	0001	1	
2	0010	2	
3	0011	3	
4	0100	4	
5	0101	5	
6	0110	6	Code
7	0111	7	for one
8	1000	8	hexadecimal
9	1001	9	digit
A	1010	10	
B	1011	11	
C	1100	12	
D	1101	13	
E	1110	14	
F	1111	15	
14	0001 0100	20	
32	0011 0010	50	
63	0110 0011	99	
F8	1111 1000	248	

- A binary code is a group of *n* bits that assume up to 2^n distinct combinations
- A four bit code is necessary to represent the ten decimal digits – 6 are unused
- The most popular decimal code is called *binary-coded decimal* (BCD)
- BCD is different from converting a decimal number to binary
- For example 99, when converted to binary, is 1100011
- 99 when represented in BCD is 1001 1001

TABLE 3-3 Binary-Coded Decimal (BCD) Numbers

Decimal number	Binary-coded decimal (BCD) number	
0	0000	↑
1	0001	
2	0010	
3	0011	Code
4	0100	for one
5	0101	decimal
6	0110	digit
7	0111	
8	1000	
9	1001	↓
10	0001 0000	
20	0010 0000	
50	0101 0000	
99	1001 1001	
248	0010 0100 1000	

- The standard alphanumeric binary code is ASCII
- This uses seven bits to code 128 characters
- Binary codes are required since registers can hold binary information only

TABLE 3-4 American Standard Code for Information Interchange (ASCII)

Character	Binary code	Character	Binary code
A	100 0001	0	011 0000
B	100 0010	1	011 0001
C	100 0011	2	011 0010
D	100 0100	3	011 0011
E	100 0101	4	011 0100
F	100 0110	5	011 0101
G	100 0111	6	011 0110
H	100 1000	7	011 0111
I	100 1001	8	011 1000
J	100 1010	9	011 1001
K	100 1011		
L	100 1100		
M	100 1101	space	010 0000
N	100 1110	.	010 1110
O	100 1111	(	010 1000
P	101 0000	+	010 1011
Q	101 0001	$	010 0100
R	101 0010	*	010 1010
S	101 0011	)	010 1001
T	101 0100	−	010 1101
U	101 0101	/	010 1111
V	101 0110	,	010 1100
W	101 0111	=	011 1101
X	101 1000		
Y	101 1001		
Z	101 1010		

Section 3.2 – Complements

- Complements are used in digital computers for simplifying subtraction and logical manipulation
- Two types of complements for each base r system: r's complement and $(r-1)$'s complement
- Given a number N in base r having n digits, the $(r-1)$'s complement of N is defined as $(r^n - 1) - N$

- For decimal, the 9's complement of N is $(10^n - 1) - N$
- The 9's complement of 546700 is 999999 – 546700 = 453299

- The 9's complement of 453299 is 999999 – 453299 = 546700
- For binary, the 1's complement of N is $(2^n - 1) - N$
- The 1's complement of 1011001 is 1111111 – 1011001 = 0100110
- The 1's complement is the true complement of the number – just toggle all bits

- The r's complement of an n-digit number N in base r is defined as $r^n - N$
- This is the same as adding 1 to the $(r - 1)$'s complement
- The 10's complement of 2389 is 7610 + 1 = 7611
- The 2's complement of 101100 is 010011 + 1 = 010100

- Subtraction of unsigned n-digit numbers: $M - N$
  - Add M to the r's complement of N – this results in
    $M + (r^n - N) = M - N + r^n$
  - If $M \geq N$, the sum will produce an end carry r^n which is discarded
  - If $M < N$, the sum does not produce an end carry and is equal to $r^n - (N - M)$, which is the r's complement of $(N - M)$. To obtain the answer in a familiar form, take the r's complement of the sum and place a negative sign in front.

Example: 72532 – 13250 = 59282. The 10's complement of 13250 is 86750.

M	= 72352
10's comp. of N	= +86750
Sum	= 159282
Discard end carry	= -100000
Answer	= 59282

Example for M < N: 13250 – 72532 = -59282

M	= 13250
10's comp. of N	= +27468
Sum	= 40718
No end carry	
Answer	= -59282 (10's comp. of 40718)

Example for X = 1010100 and Y = 1000011

X	= 1010100
2's comp. of Y	= +0111101
Sum	= 10010001
Discard end carry	= -10000000
Answer X – Y	= 0010001
Y	= 1000011
2's comp. of X	= +0101100
Sum	= 1101111

No end carry
Answer = -0010001 (2's comp. of 1101111)

Section 3.3 – Fixed-Point Representation

- Positive integers and zero can be represented by unsigned numbers
- Negative numbers must be represented by signed numbers since + and – signs are not available, only 1's and 0's are
- Signed numbers have msb as 0 for positive and 1 for negative – msb is the sign bit
- Two ways to designate binary point position in a register
  - Fixed point position
  - Floating-point representation
- Fixed point position usually uses one of the two following positions
  - A binary point in the extreme left of the register to make it a fraction
  - A binary point in the extreme right of the register to make it an integer
  - In both cases, a binary point is not actually present
- The floating-point representations uses a second register to designate the position of the binary point in the first register

- When an integer is positive, the msb, or sign bit, is 0 and the remaining bits represent the magnitude
- When an integer is negative, the msb, or sign bit, is 1, but the rest of the number can be represented in one of three ways
  - Signed-magnitude representation
  - Signed-1's complement representation
  - Signed-2's complement representation

- Consider an 8-bit register and the number +14
  - The only way to represent it is 00001110
- Consider an 8-bit register and the number –14
  - Signed magnitude: 1 0001110
  - Signed 1's complement: 1 1110001
  - Signed 2's complement: 1 1110010
- Typically use signed 2's complement

- Addition of two signed-magnitude numbers follow the normal rules
  - If same signs, add the two magnitudes and use the common sign
  - Differing signs, subtract the smaller from the larger and use the sign of the larger magnitude
  - Must compare the signs and magnitudes and then either add or subtract
- Addition of two signed 2's complement numbers does not require a comparison or subtraction – only addition and complementation
  - Add the two numbers, including their sign bits
  - Discard any carry out of the sign bit position
  - All negative numbers must be in the 2's complement form
  - If the sum obtained is negative, then it is in 2's complement form

+6	00000110	-6	11111010
+13	00001101	+13	00001101
+19	00010011	+7	00000111

+6	00000110	-6	11111010
-13	11110011	-13	11110011
-7	11111001	-19	11101101

- Subtraction of two signed 2's complement numbers is as follows
  - Take the 2's complement form of the subtrahend (including sign bit)
  - Add it to the minuend (including the sign bit)
  - A carry out of the sign bit position is discarded

- An *overflow* occurs when two numbers of *n* digits each are added and the sum occupies *n* + 1 digits
- Overflows are problems since the width of a register is finite
- Therefore, a flag is set if this occurs and can be checked by the user
- Detection of an overflow depends on if the numbers are signed or unsigned
- For unsigned numbers, an overflow is detected from the end carry out of the msb
- For addition of signed numbers, an overflow cannot occur if one is positive and one is negative – both have to have the same sign
- An overflow can be detected if the carry into the sign bit position and the carry out of the sign bit position are not equal

+70	0 1000110	-70	1 0111010
+80	0 1010000	-80	1 0110000
+150	1 0010110	-150	0 1101010

- The representation of decimal numbers in registers is a function of the binary code used to represent a decimal digit
- A 4-bit decimal code requires four flip-flops for each decimal digit
- This takes much more space than the equivalent binary representation and the circuits required to perform decimal arithmetic are more complex
- Representation of signed decimal numbers in BCD is similar to the representation of signed numbers in binary
- Either signed magnitude or signed complement systems
- The sign of a number is represented with four bits
  - 0000 for +
  - 1001 for –
- To obtain the 10's complement of a BCD number, first take the 9's complement and then add one to the least significant digit

- Example: (+375) + (-240) = +135

0	375	$(0000\ 0011\ 0111\ 1010)_{BCD}$
+9	760	$(1001\ 0111\ 0110\ 0000)_{BCD}$
0	135	$(0000\ 0001\ 0011\ 0101)_{BCD}$

Section 3.4 – Floating-Point Representation

- The floating-point representation of a number has two parts
- The first part represents a signed, fixed-point number – the *mantissa*
- The second part designates the position of the binary point – the *exponent*
- The mantissa may be a fraction or an integer
- Example: the decimal number +6132.789 is
    - Fraction: +0.6123789
    - Exponent: +04
    - Equivalent to $+0.6132789 \times 10^{+4}$
- A floating-point number is always interpreted to represent $m \times r^e$
- Example: the binary number +1001.11 (with 8-bit fraction and 6-bit exponent)
    - Fraction: 01001110
    - Exponent: 000100
    - Equivalent to $+(.1001110)_2 \times 2^{+4}$
- A floating-point number is said to be *normalized* if the most significant digit of the mantissa is nonzero
- The decimal number 350 is normalized, 00350 is not
- The 8-bit number 00011010 is not normalized
- Normalize it by fraction = 11010000 and exponent = -3
- Normalized numbers provide the maximum possible precision for the floating-point number

Section 3.5 – Other Binary Codes

- Digital systems can process data in discrete form only
- Continuous, or analog, information is converted into digital form by means of an analog-to-digital converter
- The reflected binary or *Gray code*, is sometimes used for the converted digital data
- The Gray code changes by only one bit as it sequences from one number to the next
- Gray code counters are sometimes used to provide the timing sequences that control the operations in a digital system

TABLE 3-5 4-Bit Gray Code

Binary code	Decimal equivalent	Binary code	Decimal equivalent
0000	0	1100	8
0001	1	1101	9
0011	2	1111	10
0010	3	1110	11
0110	4	1010	12
0111	5	1011	13
0101	6	1001	14
0100	7	1000	15

- Binary codes for decimal digits require a minimum of four bits
- Other codes besides BCD exist to represent decimal digits

TABLE 3-6 Four Different Binary Codes for the Decimal Digit

Decimal digit	BCD 8421	2421	Excess-3	Excess-3 gray
0	0000	0000	0011	0010
1	0001	0001	0100	0110
2	0010	0010	0101	0111
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1100
6	0110	1100	1001	1101
7	0111	1101	1010	1111
8	1000	1110	1011	1110
9	1001	1111	1100	1010
	1010	0101	0000	0000
Unused	1011	0110	0001	0001
bit	1100	0111	0010	0011
combi-	1101	1000	1101	1000
nations	1110	1001	1110	1001
	1111	1010	1111	1011

- The 2421 code and the excess-3 code are both *self-complementing*
- The 9's complement of each digit is obtained by complementing each bit in the code
- The 2421 code is a *weighted code*
- The bits are multiplied by indicated weights and the sum gives the decimal digit
- The excess-3 code is obtained from the corresponding BCD code added to 3

Section 3.6 – Error Detection Codes

- Transmitted binary information is subject to noise that could change bits 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 can prompt the data to be retransmitted
- The most common error detection code used is the *parity bit*

- A parity bit is an extra bit included with a binary message to make the total number of 1's either odd or even

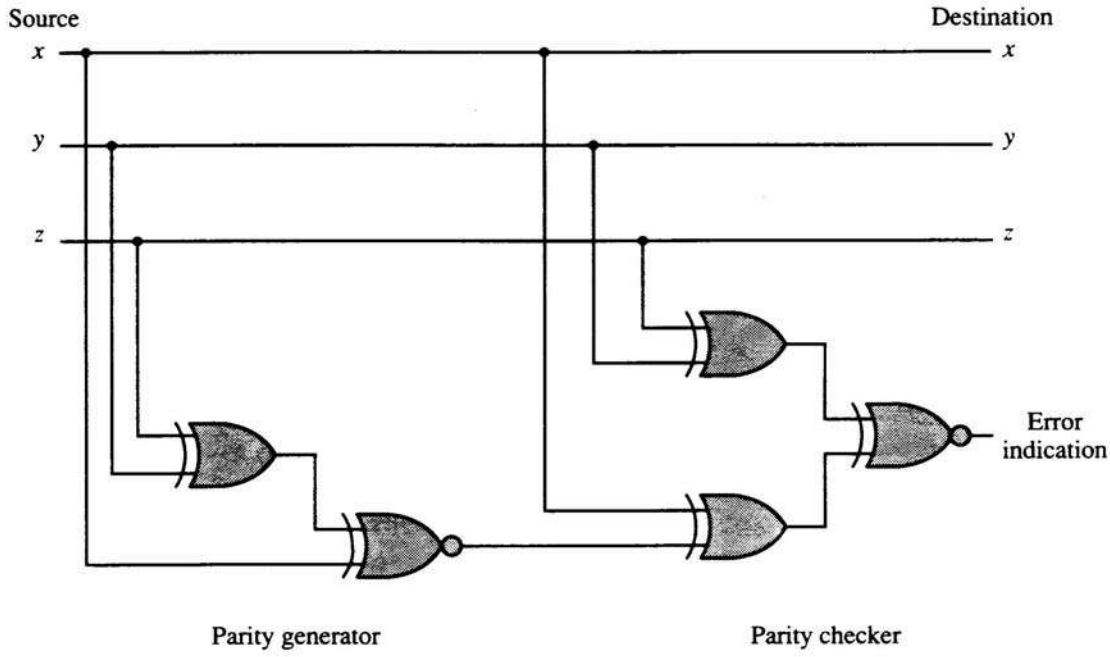
TABLE 3-7 Parity Bit Generation

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

- The P(odd) bit is chosen to make the sum of 1's in all four bits odd
- The even-parity scheme has the disadvantage of having a bit combination of all 0's
- Procedure during transmission:
    - At the sending end, the message is applied to a *parity generator*
    - The message, including the parity bit, is transmitted
    - At the receiving end, all the incoming bits are applied to a *parity checker*
    - Any odd number of errors are detected

- Parity generators and checkers are constructed with XOR gates (odd function)
- An odd function generates 1 iff an odd number if input variables are 1

Figure 3-3 Error detection with odd parity bit.



UNIT-IV

COMPUTER ARITHMETIC

Introduction:

Data is manipulated by using the arithmetic instructions in digital computers. Data is manipulated to produce results necessary to give solution for the computation problems. The Addition, subtraction, multiplication and division are the four basic arithmetic operations. If we want then we can derive other operations by using these four operations.

To execute arithmetic operations there is a separate section called arithmetic processing unit in central processing unit. The arithmetic instructions are performed generally on binary or decimal data. Fixed-point numbers are used to represent integers or fractions. We can have signed or unsigned negative numbers. Fixed-point addition is the simplest arithmetic operation.

If we want to solve a problem then we use a sequence of well-defined steps. These steps are collectively called algorithm. To solve various problems we give algorithms.

In order to solve the computational problems, arithmetic instructions are used in digital computers that manipulate data. These instructions perform arithmetic calculations.

And these instructions perform a great activity in processing data in a digital computer. As we already stated that with the four basic arithmetic operations addition, subtraction, multiplication and division, it is possible to derive other arithmetic operations and solve scientific problems by means of numerical analysis methods.

A processor has an arithmetic processor(as a sub part of it) that executes arithmetic operations. The data type, assumed to reside in processor, registers during the execution of an arithmetic instruction. Negative numbers may be in a signed magnitude or signed complement representation. 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 magnitude representation for the mantissa.

Addition and Subtraction :

Addition and Subtraction with Signed –Magnitude Data

We designate the magnitude of the two numbers by A and B. Where 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 4.1. 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 present a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0.

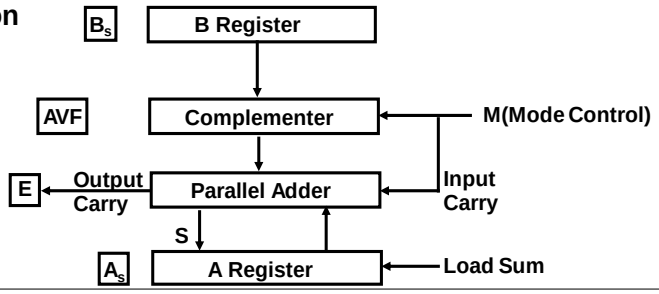
The algorithms for addition and subtraction are derived from the table and can be stated as follows (the words parentheses should be used for the subtraction algorithm)

SIGNED MAGNITUDEADDITION AND SUBTRACTION

Addition: A + B ; A: Augend; B: Addend
Subtraction: A - B: A: Minuend; B: Subtrahend

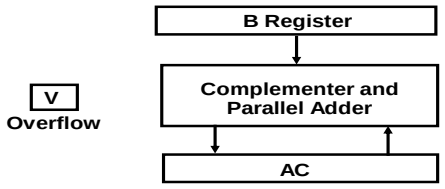
Operation	Add Magnitude	Subtract Magnitude		
		When A>B	When A<B	When A=B
(+A) + (+B)	+(A + B)			
(+A) + (- B)		+(A - B)	- (B - A)	+(A - B)
(- A) + (+B)		- (A - B)	+(B - A)	+(A - B)
(- A) + (- B)	- (A + B)			
(+A) - (+B)		+(A - B)	- (B - A)	+(A - B)
(+A) - (- B)	+(A + B)			
(- A) - (+B)	- (A + B)			
(- A) - (- B)		- (A - B)	+(B - A)	+(A - B)

Hardware Implementation

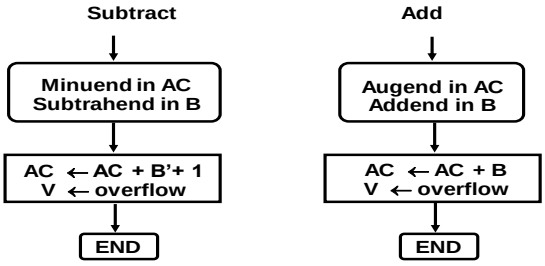


SIGNED 2'S COMPLEMENT ADDITION AND SUBTRACTION

Hardware



Algorithm



Algorithm:

- The flowchart is shown in Figure 7.1. The two signs A, and B, 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 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 complemented B. No overflow can occur if the numbers are subtracted so AVF is cleared to 0.
- 1 in E indicates that A >= B and the number in A is the correct result. If this numbs is zero, the sign A must be made positive to avoid a negative zero.
- 0 in E indicates that A < B. For this case it is necessary to take the 2's complement of the value in A. The operation can be done with one microoperation A 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 is required. However, when A < B, the sign of the result is the complement of the original sign of A. It is then necessary to complement A, to obtain the correct sign.
- The final result is found in register A and its sign in As. The value in AVF provides an overflow indication. The final value of E is immaterial.
- Figure 7.2 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 As and Bs.
- 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 two numbers.
- The add-overflow flip-flop *AVF* holds the overflow bit when A and B are added.
- The A register provides other microoperations that may be needed when we specify the sequence of steps in the algorithm.

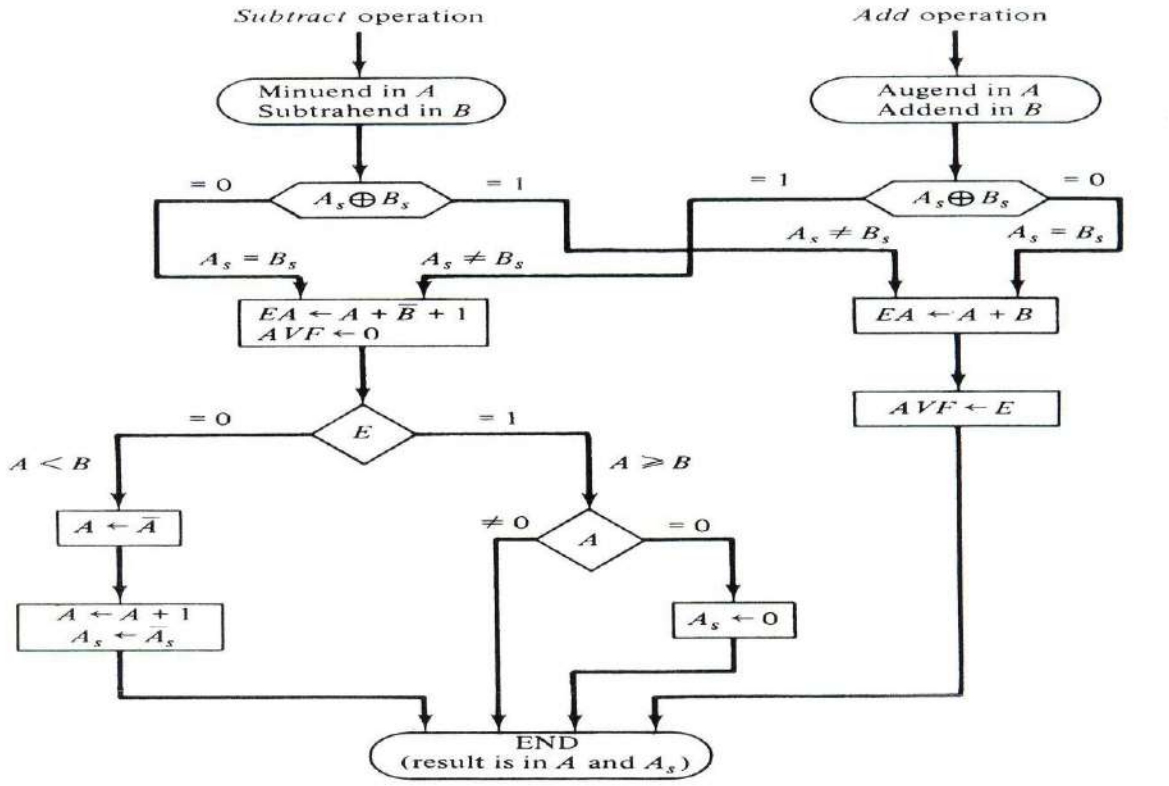
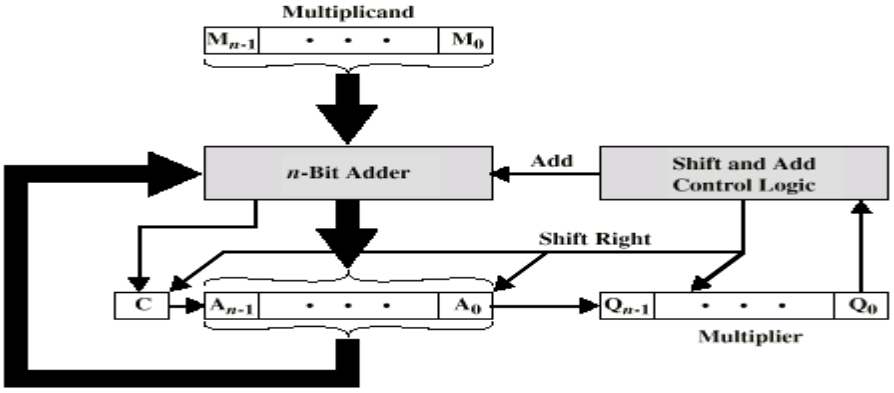


Figure 10-2 Flowchart for add and subtract operations.

Multiplication Algorithm:

In the beginning, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in Bs and Qs respectively. We compare the signs of both A and Q and set to corresponding sign of the product since a double-length product will be stored in registers A and Q. Registers A and E are cleared and the sequence counter SC is set to the number of bits of the multiplier. 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.

Now, the low order bit of the multiplier in Qn is tested. If it is 1, the multiplicand (B) is added to present partial product (A), 0 otherwise. 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. When SC = 0 we stops the process.



C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

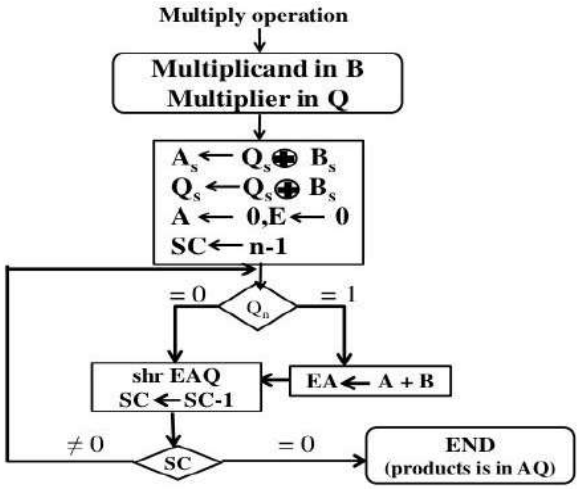


Figure: Flowchart for multiply operation.

Booth's algorithm :

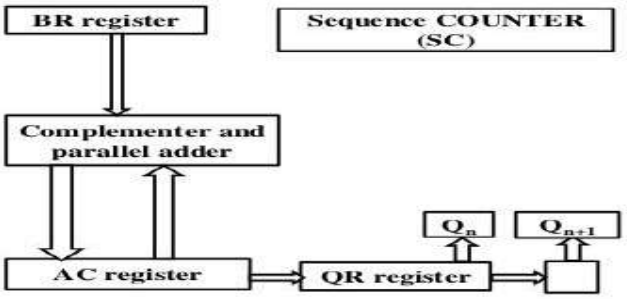
- Booth algorithm gives a procedure for multiplying binary integers in signed- 2's complement representation.
- It operates on the fact that strings of 0's in the multiplier require no addition but just

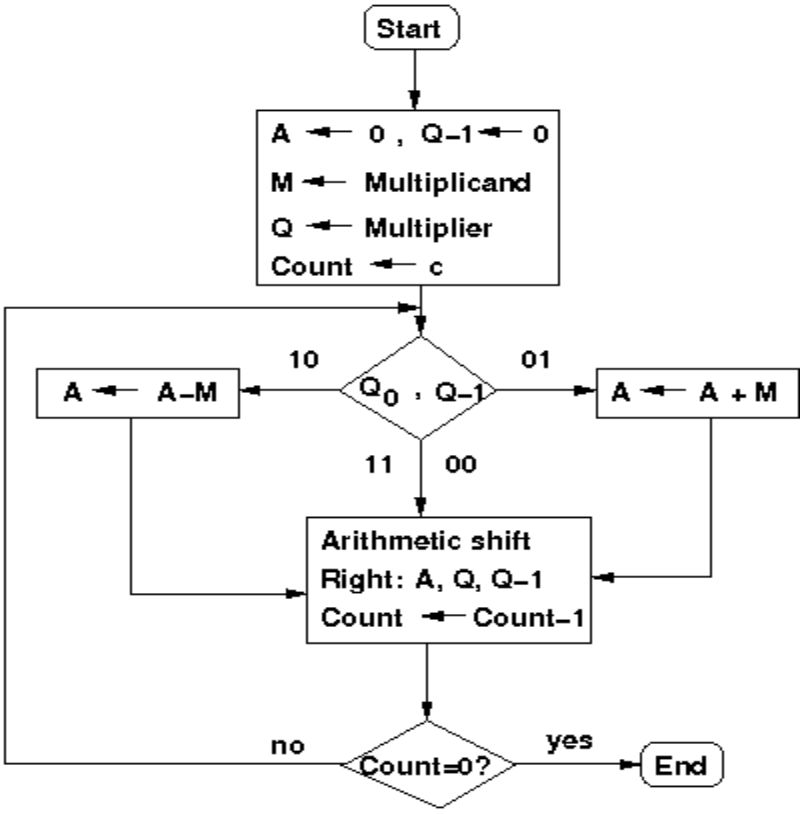
shifting, and a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} - 2^m$.

- For example, the binary number 001110 (+14) has a string 1's from 2^3 to 2^1 (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.

Hardware for Booth Algorithm

- Sign bits are not separated from the rest of the registers
- 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
- Flip-flop Qn+1 is appended to QR to facilitate a double bit inspection of the multiplier





- As in all multiplication schemes, booth algorithm requires examination of the multiplier bits and shifting of partial product.
- Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial, 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.

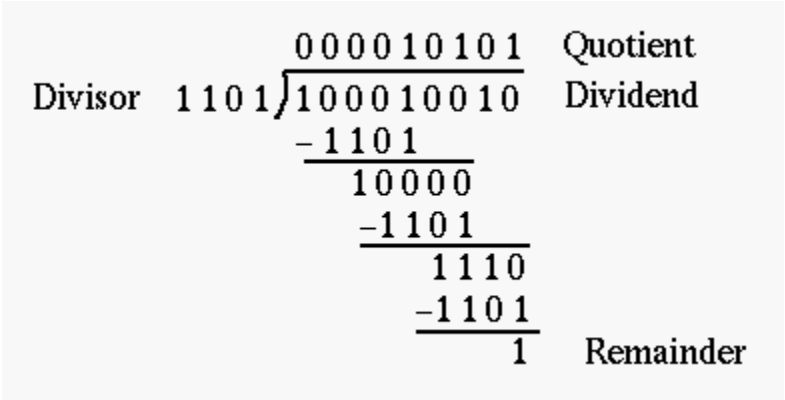
2. The multiplicand is added to the partial product upon encountering the first 0 in a string of 0's in the multiplier.

3. The partial product does not change when multiplier bit is identical to the previous multiplier bit.

- The algorithm works for positive or negative multipliers in 2's complement representation.
- This is because a negative multiplier ends with a string of 1's and the last operation will be a subtraction of the appropriate weight.
- The two bits of the multiplier in Qn and Qn+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.
- When the two bits are equal, the partial product does not change.

Division Algorithms

Division of two fixed-point binary numbers in signed magnitude representation is performed with paper and pencil by a process of successive compare, shift and subtract operations. Binary division is much simpler than decimal division because here 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 described in Figure



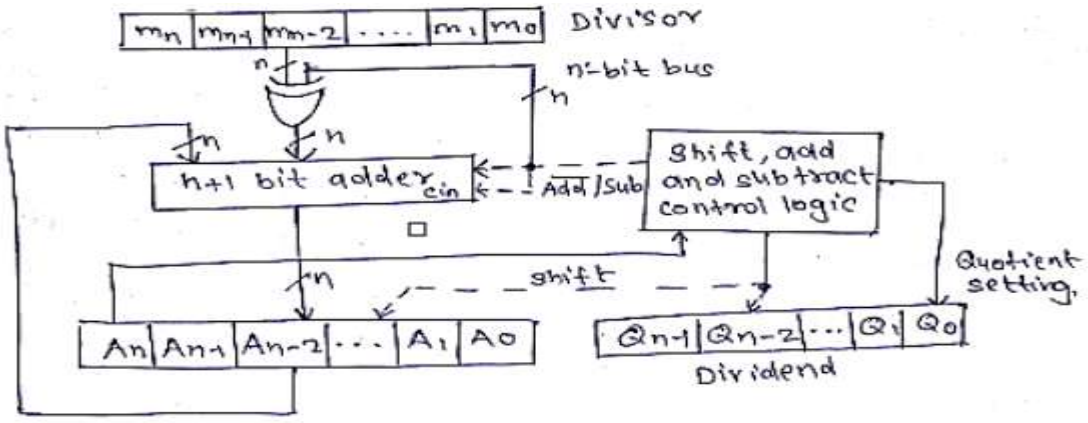
The devisor is compared with the five most significant bits of the dividend. Since the 5-bit number is smaller than B, we again repeat the same process. Now the 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend. Now we shift the divisor once to the right and subtract it from the dividend. The difference is known as a partial remainder because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial

remainder. Comparing a partial remainder with the divisor continues the process. 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. Obviously the result gives both a quotient and a remainder.

Hardware Implementation for Signed-Magnitude Data

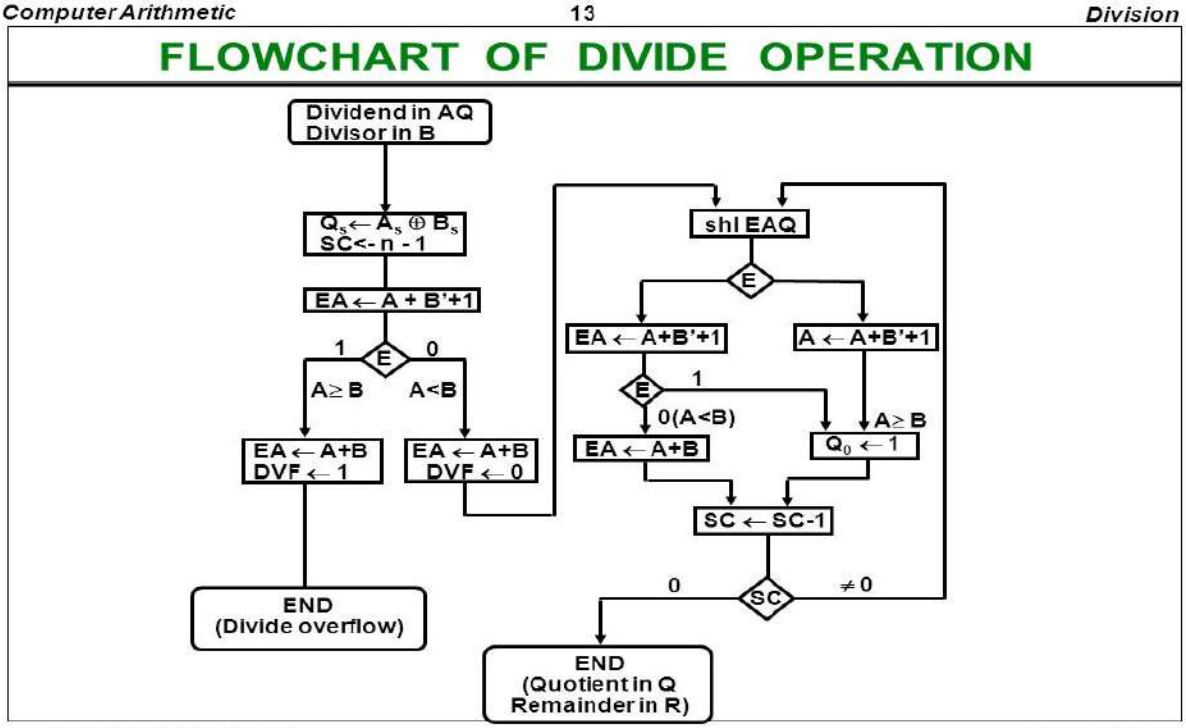
In hardware implementation for signed-magnitude data in a digital computer, it is convenient to change the process slightly. Instead of shifting the divisor to the right, two dividends, or partial remainders, are shifted to the left, thus leaving the two numbers in the required relative position. Subtraction is achieved by adding A to the 2's complement of B. End carry gives the information about the relative magnitudes.

The hardware required is identical to that of multiplication. Register EAQ is now shifted to the left with 0 inserted into Qn and the previous value of E is lost. The example is given in Figure 4.10 to clear the proposed division process. The divisor is stored in the B register 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 its 2's complement value. E



Hardware Implementation for Signed-Magnitude Data

Algorithm:



Example of Binary Division with Digital Hardware

Divisor B = 10001 $\overline{B} + 1 = 01111$

	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\overline{B} + 1$		01111		
E = 1	1	01011		
Set $Q_0 = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\overline{B} + 1$		01111		
E = 1	1	00101		
Set $Q_0 = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\overline{B} + 1$		01111		
E = 0; leave $Q_0 = 0$	0	11001	00110	
Add B		10001		2
Restore remainder	1	01010		
shl EAQ	0	10100	01100	
Add $\overline{B} + 1$		01111		
E = 1	1	00011		
Set $Q_0 = 1$	1	00011	01101	1
Shl EAQ	0	00110	11010	
Add $\overline{B} + 1$		01111		
E = 0; leave $Q_0 = 0$	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

Floating-point Arithmetic operations :

In many high-level programming languages we have a facility for specifying floating-point numbers. The most common way is by a real declaration statement. High level programming languages must have a provision for handling floating-point arithmetic operations. The operations are generally built in the internal hardware. If no hardware is available, the compiler must be designed with a package of floating-point software subroutine. Although the hardware method is more expensive, it is much more efficient than the software method. Therefore, floating- point hardware is included in most computers and is omitted only in very small ones.

Basic Considerations :

There are two part of a floating-point number in a computer - a mantissa m and an exponent e. The two parts represent a number generated from multiplying m times a radix r raised to the value of e. Thus

$$m \times r^e$$

The mantissa may be a fraction or an integer. The position of the radix point and the value of the radix r are not included in the registers. For example, assume a fraction representation and a radix 10. 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 said to be normalized if the most significant digit of the mantissa in nonzero. So the mantissa contains the maximum possible number of significant digits. We cannot normalize a zero 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 for a given register. Consider a computer with 48-bit words. Since one bit must be reserved for the sign, the range of fixed-point integer numbers will be + (24^7 – 1), which is approximately + 101^4. The 48 bits can be used to represent a floating-point number with 36 bits for the mantissa and 12 bits for the exponent. Assuming fraction representation for the mantissa and taking the two sign bits into consideration, the range of numbers that can be represented is

$$+ (1 - 2^{-35}) \times 2^{2047}$$

This number is derived from a fraction that contains 35 1's, an exponent of 11 bits (excluding its sign), and because 2^{11}–1 = 2047. The largest number that can be accommodated is approximately 10^{615}. The mantissa that can accommodated is 35 bits (excluding the sign) and if considered as an integer it can store a number as large as (2^{35} –1). This is approximately equal to 10^{10}, which is equivalent to a decimal number of 10 digits.

Computers with shorter word lengths use two or more words to represent a floating-point number. An 8-bit microcomputer uses four words to represent one floating-point number. One word of 8 bits are reserved for the exponent and the 24 bits of the other three words are used in the mantissa.

Arithmetic operations with floating-point numbers are more complicated than with fixed-point numbers. Their execution also takes longer time 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. We do this alignment by shifting one mantissa while its exponent is adjusted until it becomes equal to the other exponent. Consider the sum of the following floating-point numbers:

$$.5372400 \times 10^{2}$$

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

Floating-point multiplication and division need not do an alignment of the mantissas. Multiplying the two mantissas and adding the exponents can form the product. Dividing the mantissas and subtracting the exponents perform division.

The operations done with the mantissas are the same as in fixed-point numbers, so the two can share the same registers and circuits. The operations performed with the exponents are compared and incremented (for aligning the mantissas), added and subtracted (for multiplication) and division), and decremented (to normalize the result). We can represent the exponent in any one of the three representations - signed-magnitude, signed 2's complement or signed 1's complement.

Biased exponents have the advantage that they contain only positive numbers. Now it becomes simpler to compare their relative magnitude without bothering about their signs. Another advantage is that the smallest possible biased exponent contains all zeros. The floating-point representation of zero is then a zero mantissa and the smallest possible exponent.

Register Configuration

The register configuration for floating-point operations is shown in figure 4.13. As a rule, the same registers and adder used for fixed-point arithmetic are used for processing the mantissas. The difference lies in the way the exponents are handled.

The register organization for floating-point operations is shown in Fig. 4.13. Three registers are there, 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. The exponent part may use corresponding lower-case letter symbol.

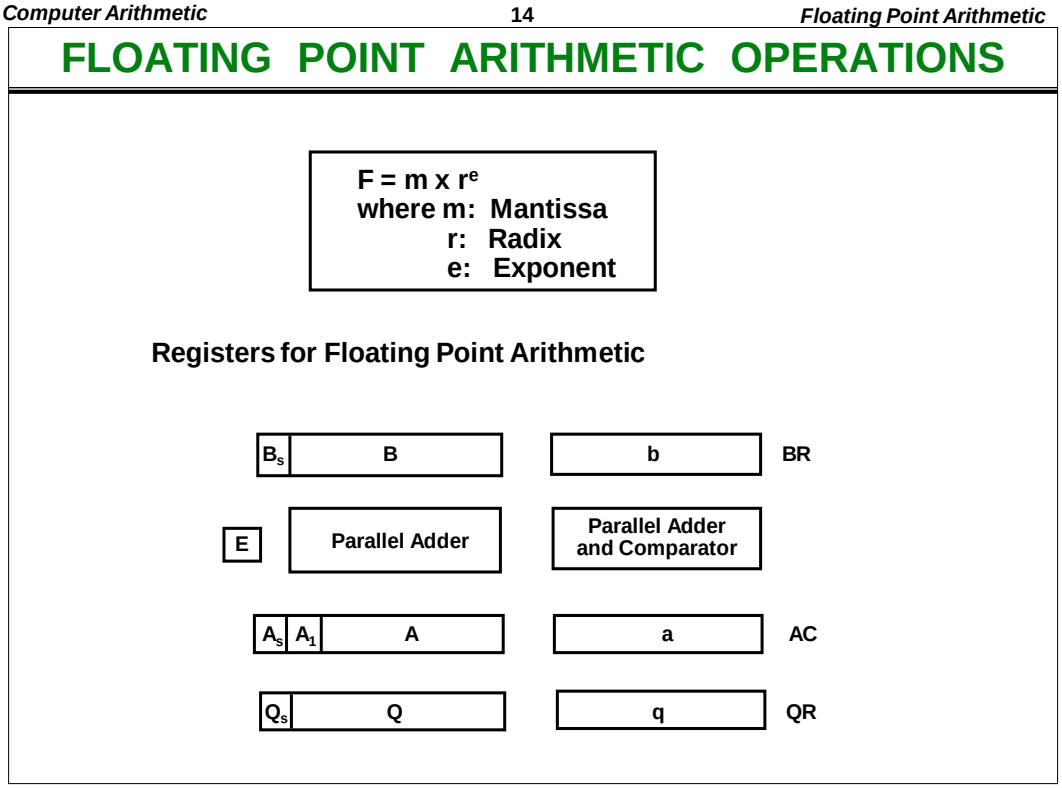


Figure 4.13: Registers for Floating Point arithmetic operations

Assuming that each floating-point number has a mantissa in signed-magnitude representation and a biased exponent. Thus the AC has a mantissa whose sign is in As, and a magnitude that is in A. The diagram shows the most significant bit of A, labeled by A1. The bit in his position must be a 1 to normalize the number. Note that the symbol AC represents the entire register, that is, the concatenation of As, A and a.

In the similar way, register BR is subdivided into Bs, B, and b and QR into Qs, Q and q. A parallel-adder adds the two mantissas and loads the sum into A and the carry into E. A separate parallel adder can be used for the exponents. The exponents do not have a district sign bit because they are biased but are represented as a biased positive quantity. It is assumed that the floating-point number are so large that the chance of an exponent overflow is very remote and so 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 they binary point is assumed to reside to the left of the magnitude part. Integer representation for floating point causes certain scaling problems during multiplication and division. To avoid these problems, we adopt a fraction representation.

The numbers in the registers should initially be normalized. After each arithmetic operation, the result will be normalized. Thus all floating-point operands are always normalized.

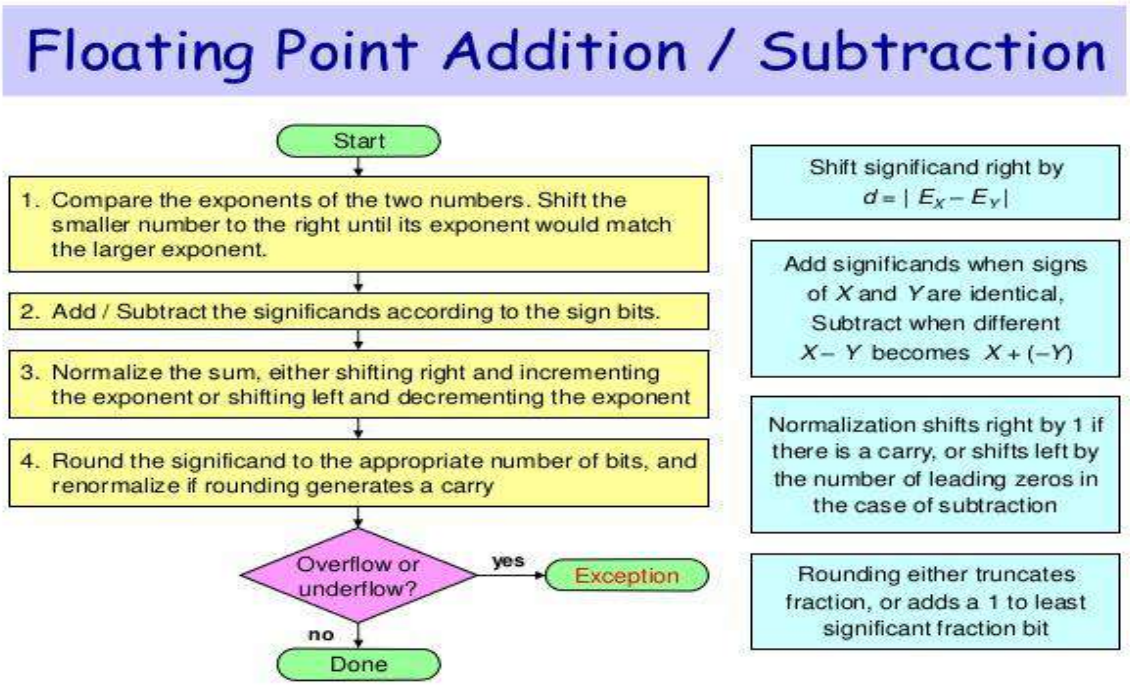
Addition and Subtraction of Floating Point Numbers

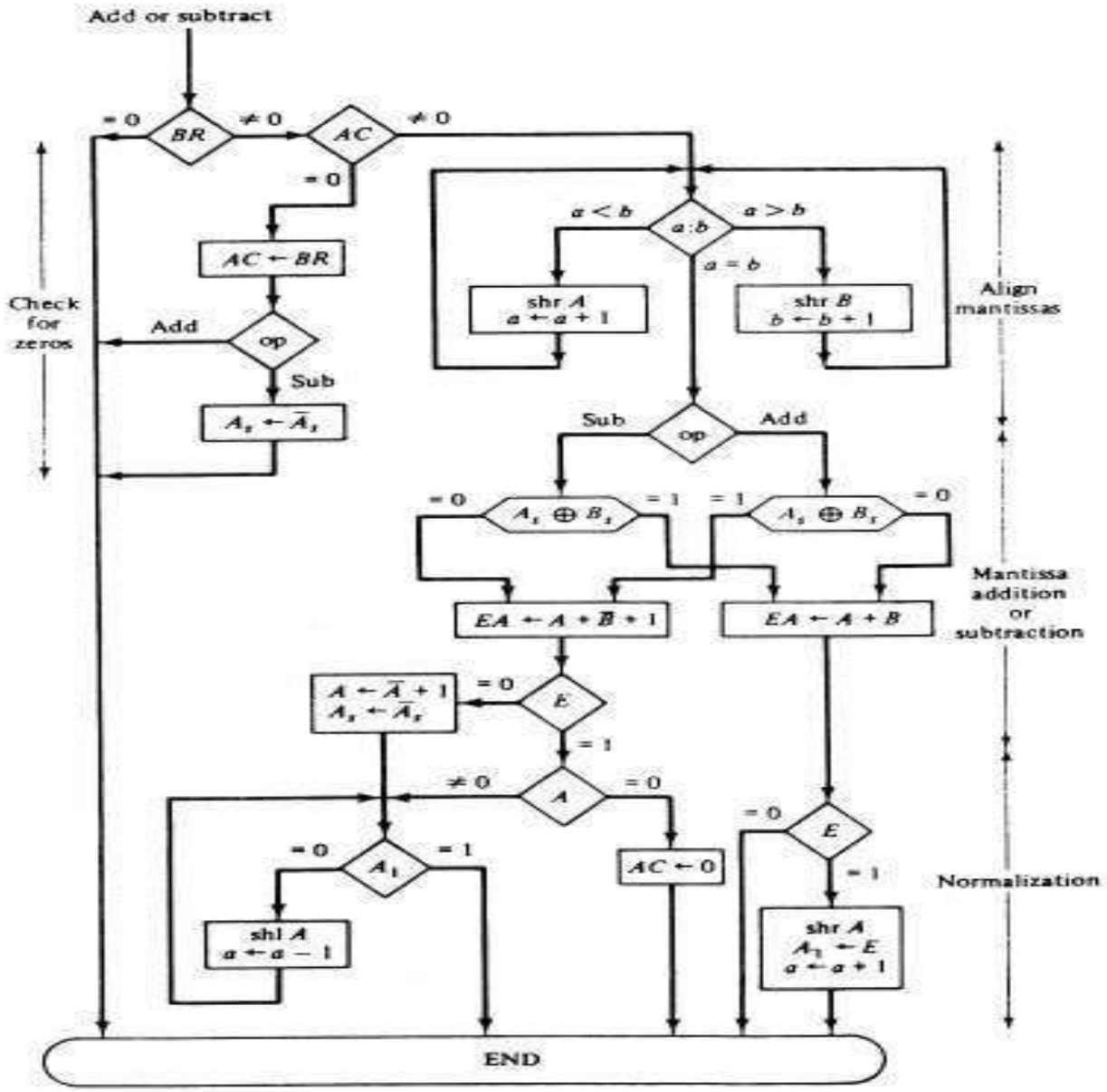
During addition or subtraction, the two floating-point operands are kept in AC and BR. The sum or difference is formed in the AC. The algorithm can be divided into four consecutive parts:

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

A floating-point number cannot be normalized, if it is 0. If this number is used for 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 un-normalized. The normalization procedure ensures that the result is normalized before it is transferred to memory.

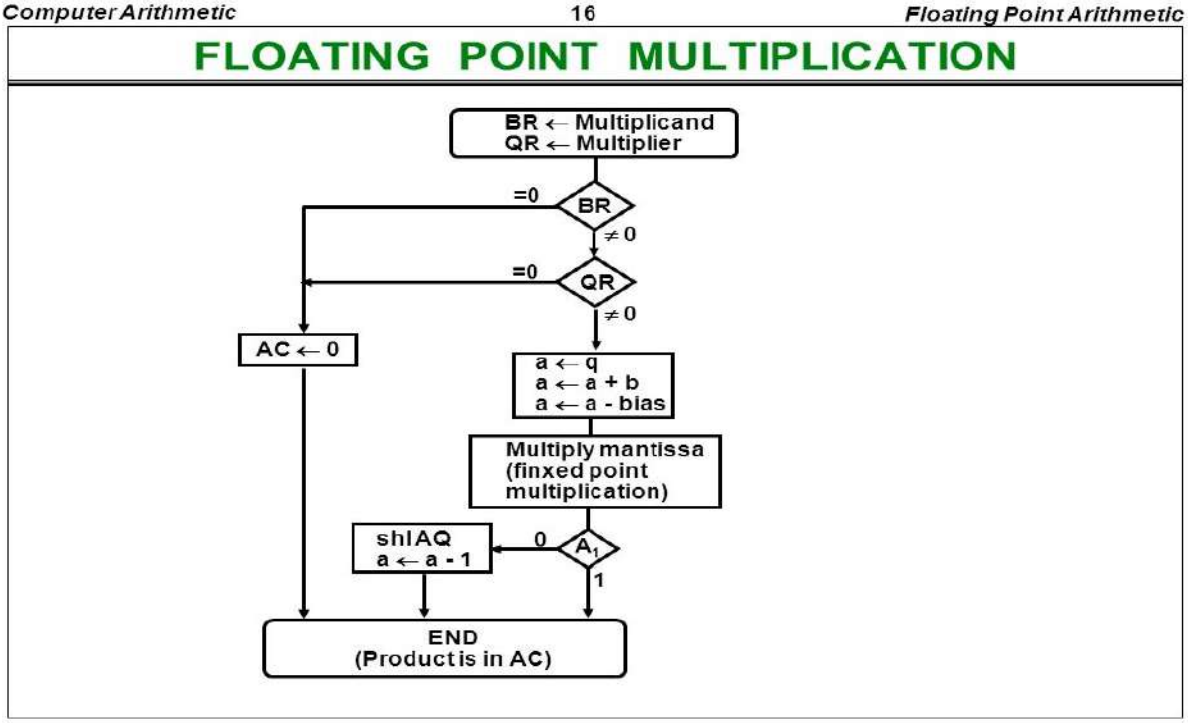
If the magnitudes were subtracted, there may be zero or may have an underflow in the result. If the mantissa is equal to zero the entire floating-point number in the AC is cleared to zero. Otherwise, the mantissa must have at least one bit that is equal to 1. The mantissa has an underflow if the most significant bit in position A1, is 0. In that case, the mantissa is shifted left and the exponent decremented. The bit in A1 is checked again and the process is repeated until A1 = 1. When A1 = 1, the mantissa is normalized and the operation is completed.

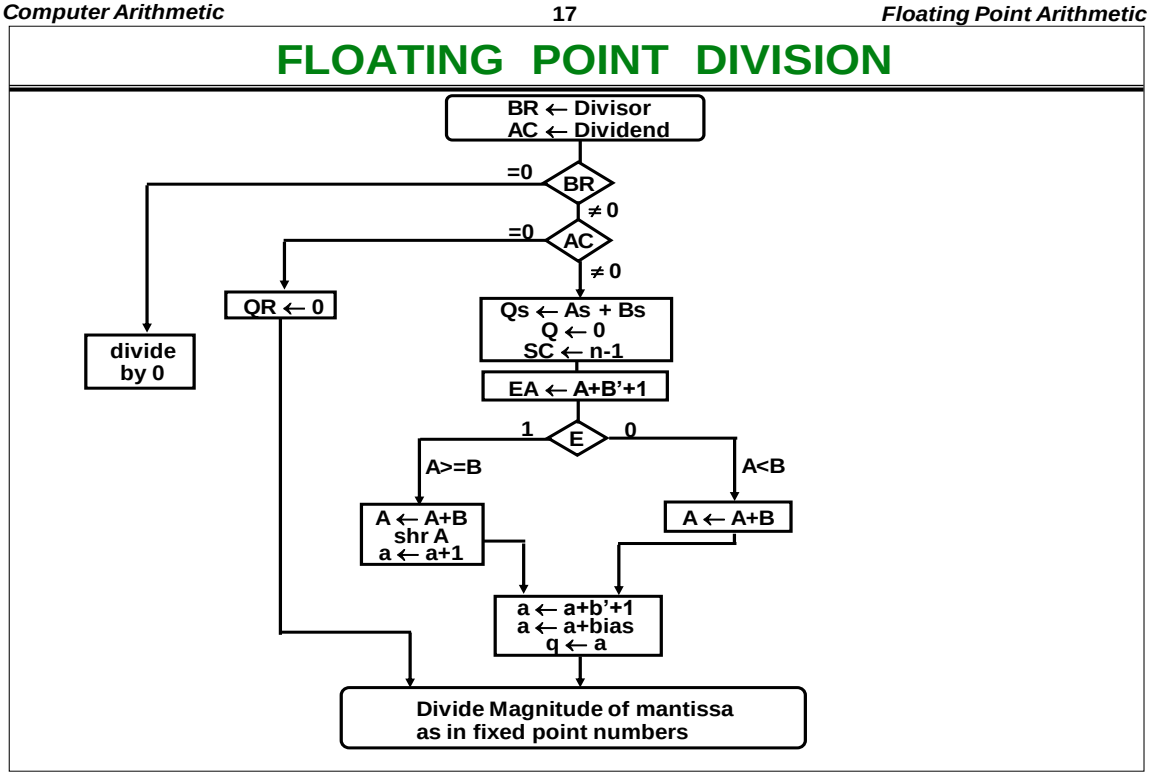




Algorithm for Floating Point Addition and Subtraction

Multiplication:



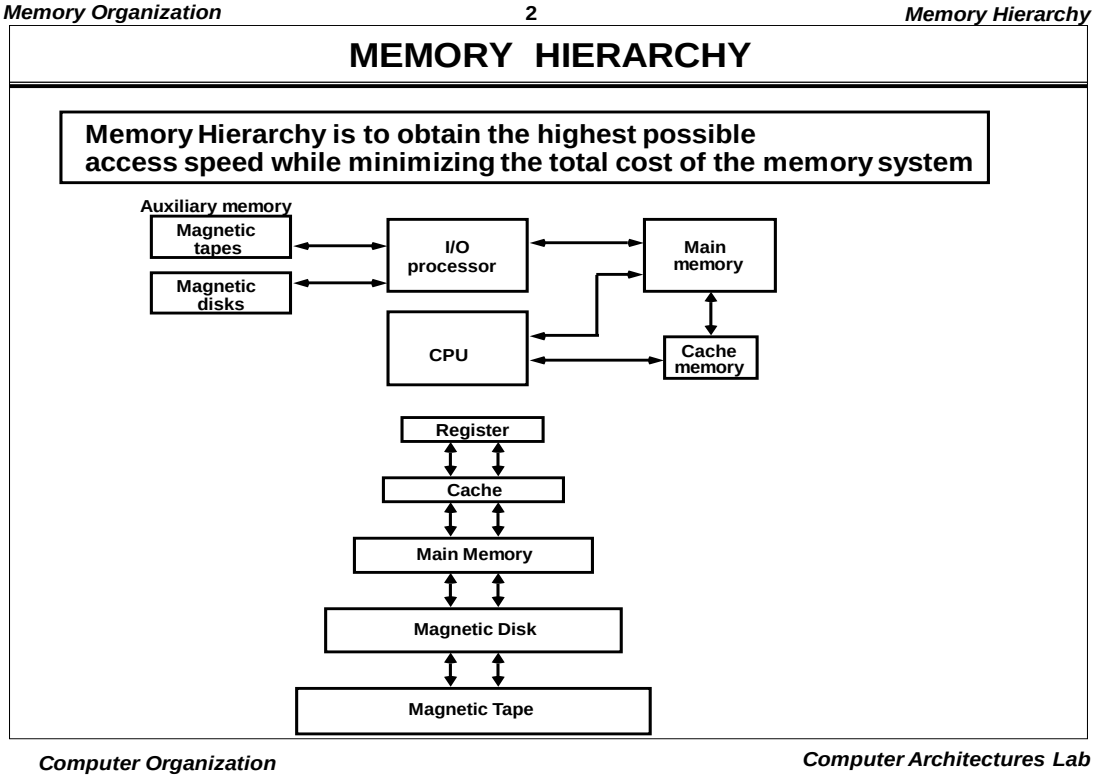


MEMORY ORGANIZATION

: Memory Hierarchy, Main Memory, Auxiliary memory, Associative

Memory, Cache Memory, Virtual Memory

Memory Hierarchy :



memory address map of RAM and ROM.

Main Memory

- The main memory is the central storage unit in a computer system.
- Primary memory holds only those data and instructions on which computer is currently working.
- It has limited capacity and data is lost when power is switched off.
- It is generally made up of semiconductor device.
- These memories are not as fast as registers.
- The data and instruction required to be processed reside in main memory.
- It is divided into two subcategories RAM and ROM.

Memory address map of RAM and ROM

- 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, shown in table 9.1.

- To demonstrate with a particular example, assume that a computer system needs 512 bytes of RAM and 512 bytes of ROM.
- The RAM and ROM chips to be used are specified in figure 9.1 and figure 9.2.

Memory address map of RAM and ROM

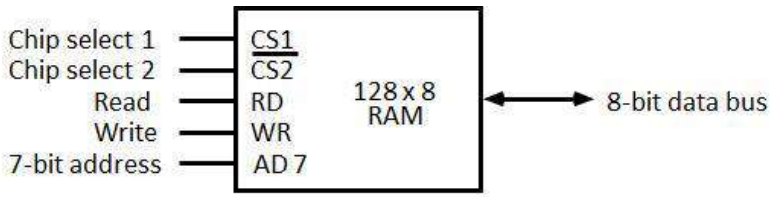


Figure 9.1: Typical RAM chip

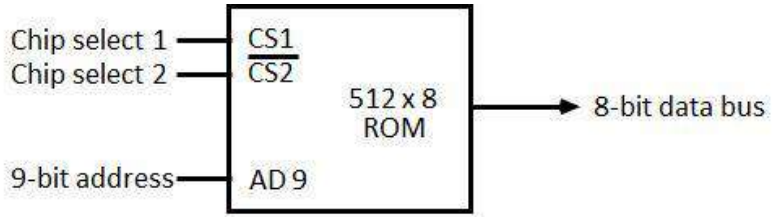


Figure 9.2: Typical ROM chip

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

- 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.
- The small 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.

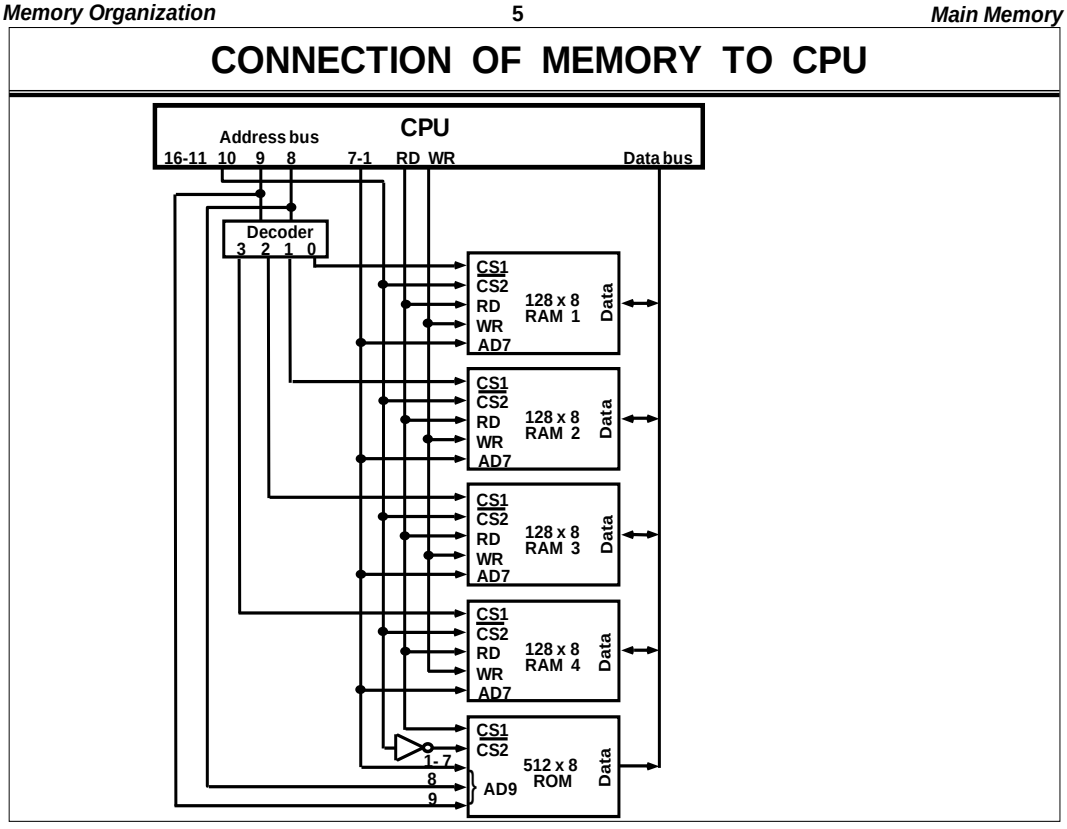
- It is now necessary to distinguish between four RAM chips by assigning to each a different address. For this particular example we choose bus lines 8 and 9 to represent four distinct binary combinations.
- The table clearly shows that the nine low-order bus lines constitute a memory space for RAM equal 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

Memory connections 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.



Auxiliary Memory :

- **Magnetic Tape:** Magnetic tapes are used for large computers like mainframe computers where large volume of data is stored for a longer time. In PC also you can use tapes in the form of cassettes. The cost of storing data in tapes is inexpensive. Tapes consist of magnetic materials that store data permanently. It can be 12.5 mm to 25 mm wide plastic film-type and 500 meter to 1200 meter long which is coated with magnetic material. The deck is connected to the central processor and information is fed into or read from the tape through the processor. It's similar to cassette tape recorder.

Magnetic tape is an information storage medium consisting of a magnetisable coating on a thin plastic strip. Nearly all recording tape is of this type, whether used for video with a video cassette recorder, audio storage (reel-to-reel tape, compact audio cassette, digital audio tape (DAT), digital linear tape (DLT) and other formats including 8-track cartridges) or general purpose digital data storage using a computer (specialized tape formats, as well as the above-mentioned compact audio cassette, used with home computers of the 1980s, and DAT, used for backup in workstation installations of the 1990s).

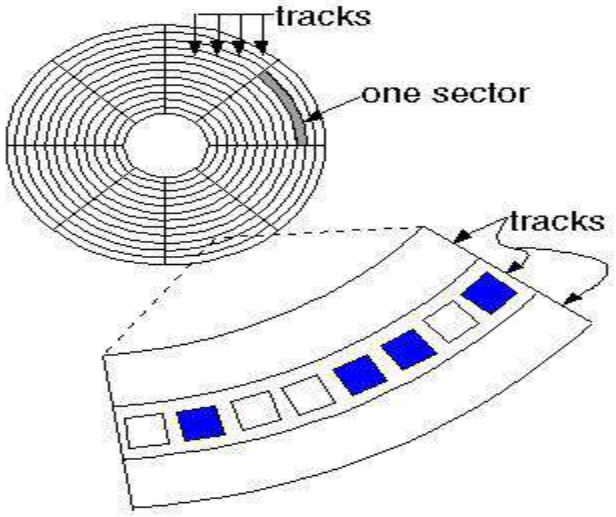
- Magneto-optical and optical tape storage products have been developed using many of the same concepts as magnetic storage, but have achieved little commercial success.

- **Magnetic Disk:** You might have seen the gramophone record, which is circular like a disk and coated with magnetic material. Magnetic disks used in computer are made on the same principle. It rotates with very high speed inside the computer drive. Data is stored on both the surface of the disk. Magnetic disks are most popular for direct access storage device. Each disk consists of a number of invisible concentric circles called tracks. Information is recorded on tracks of a disk surface in the form of tiny magnetic spots. The presence of a magnetic spot represents one bit and its absence represents zero bit. The information stored in a disk can be read many times without affecting the stored data. So the reading operation is non-destructive. But if you want to write a new data, then the existing data is erased from the disk and new data is recorded. For Example-Floppy Disk.

The primary computer storage device. Like tape, it is magnetically recorded and can be re-recorded over and over. Disks are rotating platters with a mechanical arm that moves a read/write head between the outer and inner edges of the platter's surface. It can take as long as one second to find a location on a floppy disk to as little as a couple of milliseconds on a fast hard disk. See hard disk for more details.

The disk surface is divided into concentric tracks (circles within circles). The thinner the tracks, the more storage. The data bits are recorded as tiny magnetic spots on the tracks. The smaller the spot, the more bits per inch and the greater the storage.

Sectors
Tracks are further divided into sectors, which hold a block of data that is read or written at one time; for example, READ SECTOR 782, WRITE SECTOR 5448. In order to update the disk, one or more sectors are read into the computer, changed and written back to disk. The operating system figures out how to fit data into these fixed spaces. Modern disks have more sectors in the outer tracks than the inner ones because the outer radius of the platter is greater than the inner radius



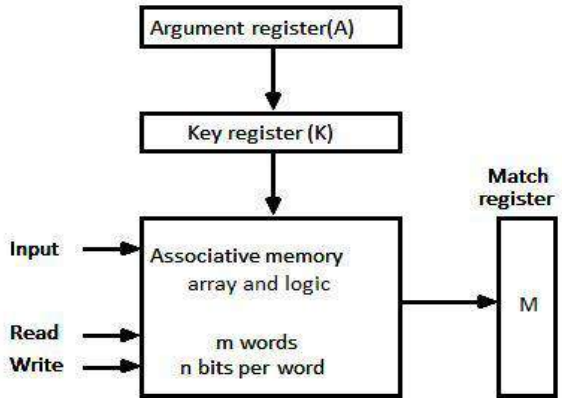
Block diagram of Magnetic Disk

Optical Disk: With every new application and software there is greater demand for memory capacity. It is the necessity to store large volume of data that has led to the development of optical disk storage medium. Optical disks can be divided into the following categories:

1. **Compact Disk/ Read Only Memory (CD-ROM**
2. **Write Once, Read Many (WORM)**
3. **Erasable Optical Disk**

Associative Memory :Content Addressable Memory (CAM).

- The time required to find an item stored in memory can be reduced considerably if stored data can be identified for access by the content of the data itself rather than by an address.
- 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.
- The block diagram of an associative memory is shown in figure 9.3.



- It consists of a memory array and logic form 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.

Hardware Organization

- The key register provides a mask for choosing a particular field or key in the argument word.
- The entire argument is compared with each memory word if the key register contains all 1's.

- Otherwise, only those bits in the argument that have 1st in their corresponding position of the key register are compared.
- Thus the key provides a mask or identifying piece of information which specifies how the reference to memory is made.
- 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 1's in these position.

A	101 111100	
K	111 000000	
Word1	100 111100	no match
Word2	101 000001	match

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

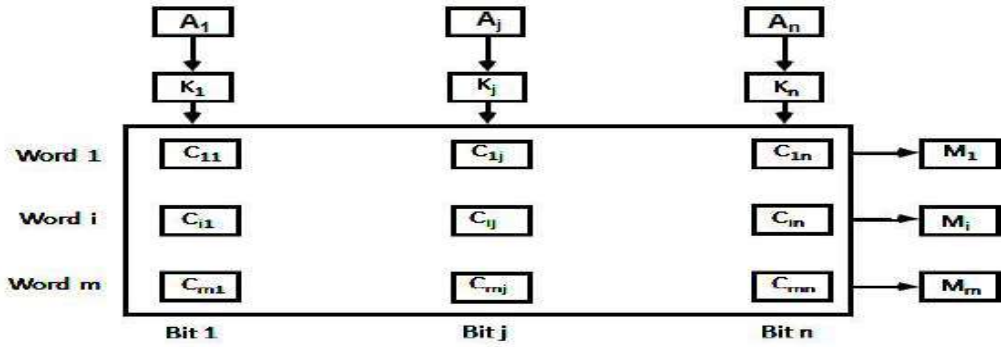


Figure 9.4: Associative memory of *m* word, n cells per word.

- The relation between the memory array and external registers in an associative memory is shown in figure 9.4.
- 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 Cij is the cell for bit j in words i.
- A bit Aj in the argument register is compared with all the bits in column j of the array provided that Kj =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 Mi in the match register is set to 1.
- If one or more unmasked bits of the argument and the word do not match, Mi is cleared to 0.

Cache Memory :

- Cache is a fast small capacity memory that should hold those information which are most likely to be accessed.

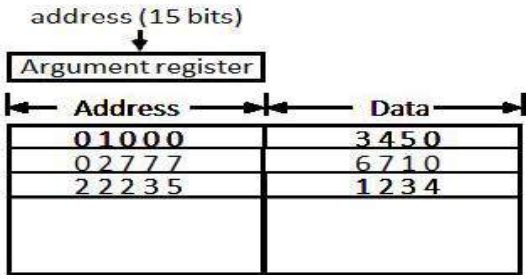
- The basic operation of the cache is, when the CPU needs to access memory, the cache is examined.
- If the word is found in the cache, it is read from the fast memory. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word.
- The transformation of data from main memory to cache memory is referred to as a **mapping process**.

Associative mapping

- Consider 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 miss, the CPU reads the word from main memory and the word is then transferred to cache.

Figure 9.5: Associative mapping cache

(all numbers in octal)



- 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 figure 9.5 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 CPU.
- If no match occurs, the main memory is accessed for the word.
- The address data pairs 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.
- This constitutes a first-in first-one (FIFO) replacement policy.

direct mapping in organization of cache memory:

- 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 from the tag field.
- The figure 9.6 shows that main memory needs an address that includes both the tag and the index.

Figure 9.6: Addressing relationships between main and cache memories

- The number of bits in the index field is equal to the number of address bits required to access the cache memory.
- The internal organization of the words in the cache memory is as shown in figure 9.7.

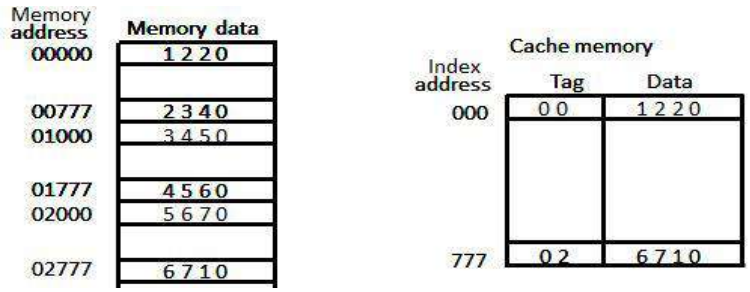


Figure 9.7: Direct mapping cache organization

- Each word in cache consists of the data word and its associated tag.
- When a new word is first brought into the cache, the tag bits are stored alongside the data bits.
- 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.
- 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 word at index address 000 is then replaced with a tag of 02 and data of 5670.

  The **disadvantage** of direct mapping is that two words with the same index in their address but with different tag values cannot reside in cache memory at the same time.

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".

When a miss occurs in a set-associative cache and the set is full, it is necessary to replace one of the tag-data items with a new value.

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

Write-through and Write-back cache write method.

Write Through

- The simplest and most commonly used procedure is to update main memory with every memory write operation.
- The cache memory being updated in parallel if it contains the word at the specified address. This is called the *write-through* method.
- This method has the advantage that main memory always contains the same data as the cache.
- This characteristic is important in systems with direct memory access transfers.
- It ensures that the data residing in main memory are valid at all times so that an I/O device communicating through DMA would receive the most recent updated data.

Write-Back (Copy-Back)

- 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 reason for the write-back method is that during the time a word resides in the cache, it may be updated several times.
- However, as long as the word remains in the cache, it does not matter whether the copy in main memory is out of date, since requests from the word are filled from the cache.
- It is only when the word is displaced from the cache that an accurate copy need be rewritten into main memory.

Virtual Memory

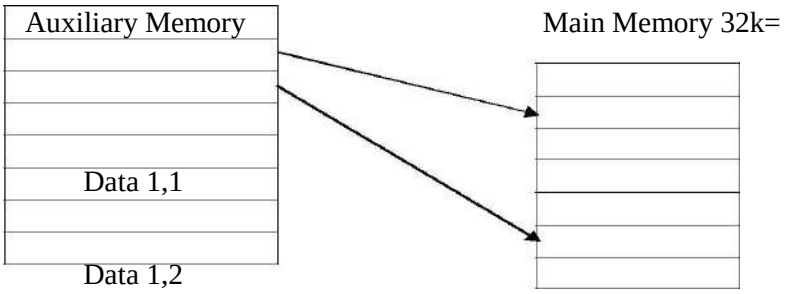
- Virtual memory is used to give programmers the illusion that they have a very large memory at their disposal, even though the computer actually has a relatively small main memory.
- A virtual memory system provides a mechanism for translating program-generated addresses into correct main memory locations.

Address space

An address used by a programmer will be called a virtual address, and the set of such addresses is known as address space.

Memory space

An address in main memory is called a location or physical address. The set of such locations is called the memory space.



Address space $1024k=2^{10}$

- As an illustration, 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 multiprogramming 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 figure 9.9.
- 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 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 too long.

Address mapping using pages.

- AThe table implementation of the address mapping is simplified if 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.
- Consider a 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 figure 9.9
- At any given time, up to four pages of address space may reside in main memory in any one of the four blocks.

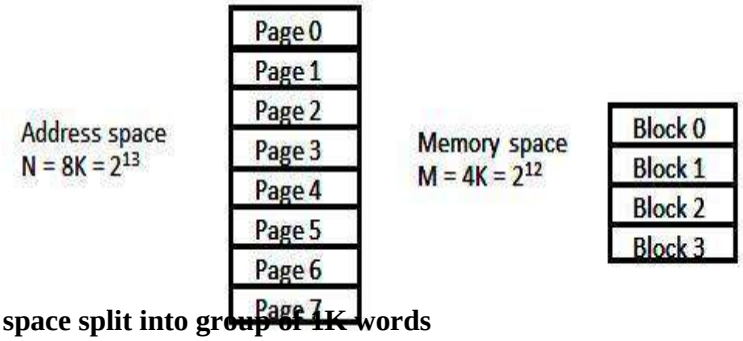


Figure 9.10 Address and Memory space split into group of 1K words

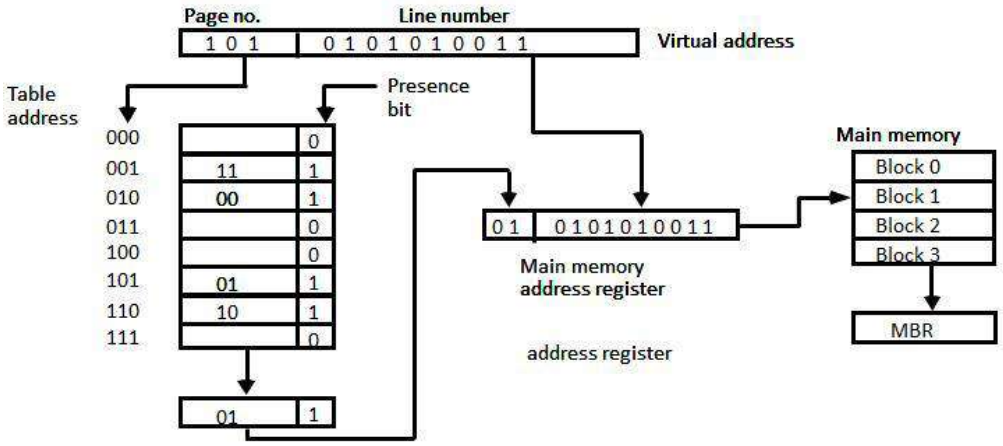


Figure 9.11: Memory table in paged system

- The organization of the memory mapping table in a paged system is shown in figure 9.10.
- 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 give 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.
- A 0 in the presence bit indicates that this page is not available in main memory. The CPU references a word in memory with a virtual address of 13 bits.

  The three 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.

  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.

  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.

Segment

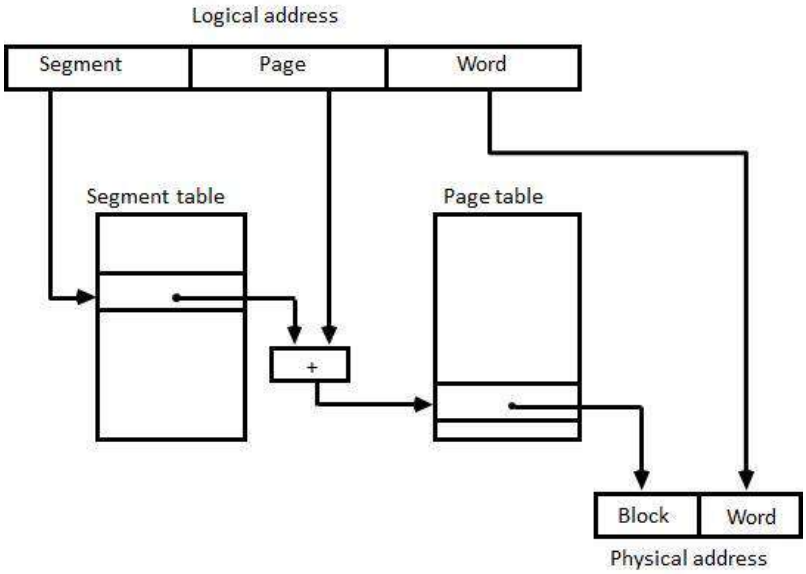
A segment is a set of logically related instructions or data elements associated with a given name.

Logical address

The address generated by segmented program is called a logical address.

Segmented page mapping

- The length of each segment is allowed to grow and contract according to the needs of the program being executed. Consider logical address shown in figure 9.12.
  Figure 9.12: Logical to physical address mapping



- The logical address is partitioned into three fields.
- The segment field specifies a segment number.
- The page field specifies the page within the segment and word field gives specific word within the page.
- A page field of k bits can specify up to 2k pages.
- A segment number may be associated with just one page or with as many as 2k pages.
- Thus the length of a segment would vary according to the number of pages that are assigned to it.
- The mapping of the logical address into a physical address is done by means of two tables, as shown in figure 9.12.
- The segment number of the logical address specifies the address for the segment table.
- The entry in the segment table is a pointer address for a page table base.
- The page table base is added to the page number given in the logical address.
- The sum produces a pointer address to an entry in the page table.

- The concatenation of the block field with the word field produces the final physical mapped address.
- The two mapping tables may be stored in two separate small memories or in main memory.
- In either case, memory reference from the CPU will require three accesses to memory: one from the segment table, one from the page table and the third from main memory.
- This would slow the system significantly when compared to a conventional system that requires only one reference to memory.

REFERENCE :

1. COMPUTER SYSTEM ARCHITECTURE, MORRIS M. MANO, 3RD EDITION, PRENTICE HALL INDIA.