COMPUTER ORGANIZATION AND ARCHITECTURE

MODULE 1

DIGITAL COMPUTERS

1.1 INTRODUCTION

The digital computer is a digital system that performs various computational tasks. The word digital implies that the information in the computer is represented by variables that take a limited number of discrete values. These values are processed internally by components that can maintain a limited number of discrete states. The decimal digits 0, 1, 2, ..., 9, for example, provide 10 discrete values. The first electronic digital computer was developed in the late 1940s and was used primarily for numerical computations.

Digital computers use the binary number system, which has two digits: o and 1. A Binary digit is called a bit. In computers, information is represented in "Group of bits".

1.1.1 COMPUTER SYSTEM

A computer system is sometimes subdivided into two functional entities:

- Hardware
- Software

Hardware: The hardware of the computer consists of all the electronic components and electromechanical devices that comprise the physical entity of the device.

The hardware of the computer is usually divided into three major parts, as shown in Fig. 1-1.

- The Central Processing Unit (CPU) contains an arithmetic and logic unit for manipulating data, a number of registers for storing data, and control circuits for fetching and executing instructions.
- The memory of a computer contains storage for instructions and data. It is called a Random-Access Memory (RAM) because the CPU can access any location in memory at random and retrieve the binary information within a fixed interval of time.
- The Input and Output Processor (IOP) contains electronic circuits for communicating and controlling the transfer of information between the computer and the outside world.

- The input and output devices connected to the computer include keyboards, printers, terminals, magnetic disk drives, and other communication devices.

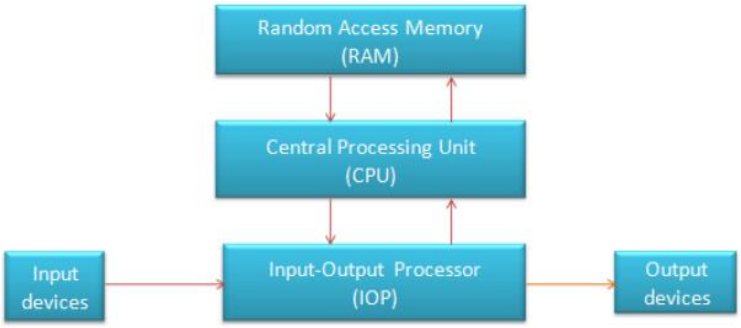


Figure 1.1 - Block diagram of a digital computer

Software: Computer software consists of the instructions and data that the computer manipulates to perform various data-processing tasks.

The system software of a computer consists of a collection of programs whose purpose is to make more effective use of the computer.

The programs included in a systems software package are referred to as the **Operating System**.

Application Software: It is software that performs specific tasks for an end-user.

For example, A High-level language program written by user to solve particular data processing needs is an **Application Program**.

The compiler that translates the high-level language program to machine language is a **System Program.**

1.1.2 COMPUTER ORGANIZATION

Computer Organization is concerned with the way the hardware components operate and the way they are connected together to form the computer system. The various components are assumed to be in place and the task is to investigate the organizational structure to verify that the computer parts operate as intended.

1.1.3 COMPUTER DESIGN

Computer Design is concerned with the hardware design of the computer. Once the computer specifications are formulated, it is the task of the designer to develop hardware for the system.

1.1.4 COMPUTER ARCHITECTURE

Computer Architecture is concerned with the structure and behaviour of the computer as seen by the user. It includes the information formats, the instruction set, and techniques for addressing memory.

The **Architectural Design** of a computer system is concerned with the specifications of the various functional modules, such as processors and memories, and structuring them together into a computer system.

1.2 DATA REPRESENTATION:

- It refers to the form in which data is stored, processed, and transmitted.
- Devices such as smartphones, iPods, and computers store data in digital formats that can be handled by electronic circuitry (motherboard).
- It refers to the symbols that represent people, events, things, and ideas. Data can be a name, a number, the colors in a photograph, or the notes in a musical composition.
- The term "data" refers to factual information used for analysis or reasoning.
- Registers are made up of flip-flops and flip-flops are two-state devices that can store only 1's and 0's.

Numbering Systems		
System	Base	Digits
Binary	2	0 1
Octal	8	0 1 2 3 4 5 6 7
Decimal	10	0 1 2 3 4 5 6 7 8 9
Hexadecimal	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

There are many methods or techniques which can be used to convert numbers from one base to another.

- Decimal to Other Base System
- Other Base System to Decimal
- Other Base System to Non-Decimal
- Shortcut method−Binary to Octal
- Shortcut method−Octal to Binary
- Shortcut method−Binary to Hexadecimal
- Shortcut method−Hexadecimal to Binary

- Decimal to Other Base System

Steps

- **Step1** −Divide the decimal number to be converted by the value of the new base.
- **Step 2**− Get the remainder from Step 1 as the rightmost digit (least significant digit) of new base number.
- **Step3**−Divide the quotient of the previous divides by the new base.
- **Step4**−Record the remainder from Step3 as the next digit (to the left) of the new base number.

Repeat Steps 3 and 4, getting remainders from right to left, until the quotient becomes zero in Step 3.

The last remainder thus obtained will be the Most Significant Digit (MSD) of the new base number.

Example: Decimal number: 29_{10}

Calculating Binary Equivalent

Step	Operation	Result	Remainder
Step 1	29/2	14	1
Step 2	14/2	7	0
Step 3	7/2	3	1
Step 4	3/2	1	1
Step 5	1/2	0	1

As mentioned in Steps 2 and 4, the remainders have to be arranged in the reverse order so that the first remainder becomes the Least Significant Digit (LSD) and the last remainder becomes the Most Significant Digit (MSD).

Decimal number : 29_{10}

Binary number : 11101_2

Other Base System to Decimal System

Steps

- **Step 1**− Determine the column (positional) value of each digit (this depends on the position of the digit and the base of the number system).
- **Step2**−Multiply the obtained column values (inStep1) by the digits in the

corresponding columns.

- **Step3**−Sum the products calculated in Step2.The total is the equivalent value in decimal.

Step	Binary Number	Decimal Number
Step 1	11101_2	$((1\times2^4)+(1\times2^3)+(1\times2^2)+(0\times2^1)+(1\times2^0))_{10}$
Step 2	11101_2	$(16+8+4+0+1)_{10}$
Step 3	11101_2	29_{10}

Example

Binary Number – 11101_2

Calculating Decimal Equivalent−

Binary Number−11101_2=Decimal Number−29_{10}

Other Base System to Non-Decimal System

Steps:

Step 1: Convert the original number to a decimal number (base10).

Step 2 : Convert the decimal number obtained to the new base number.

Example

Octal Number−25_8

Calculating Binary Equivalent− Step 1 − Convert to Decimal

Step	Octal Number	Decimal Number
Step 1	25_8	$((2\times8^1)+(5\times8^0))_{10}$
Step 2	25_8	$(16+5)_{10}$
Step 3	25_8	21_{10}

Octal Number−25_8=Decimal Number−21_{10}

Step 2 – Convert Decimal to Binary

Step	Operation	Result	Remainder
Step 1	21/2	10	1
Step 2	10/2	5	0
Step 3	5/2	2	1
Step 4	2/2	1	0
Step 5	1/2	0	1

Decimal Number−21_{10}=Binary Number−10101_2

Octal Number − 25_8= Binary Number – 10101_2

Shortcut method –Binary to Octal

Steps:

- **Step1**−Divide the binary digits in to groups of three (starting from the right).
- **Step2**−Convert each group of three binary digits to one octal digit.

Example

Binary Number − 10101

Calculating Octal Equivalent –

Step	Binary Number	Octal Number
Step 1	10101_2	010101
Step 2	10101_2	$2_8 5_8$
Step 3	10101_2	25_8

Binary Number−10101_2 = Octal Number – 25_8 Shortcut method-Octal to Binary

Steps

- **Step 1**− Convert each octal digit to a 3 digit binary number (the octal digits may be treated as decimal for this conversion).
- **Step 2**− Combine all the resulting binary groups (of 3 digits each) into a single binary number.

Example

Octal Number−25_8

Calculating Binary Equivalent−

Step	Octal Number	Binary Number
Step 1	25_8	$2_{10}5_{10}$
Step 2	25_8	010_2101_2
Step 3	25_8	010101_2

Octal Number − 25_8 = Binary Number − 101012

Shortcut method – Binary to Hexadecimal

Steps

- **Step1** − Divide the binary digits into groups of four (starting from the right).
- **Step2** − Convert each group of four binary digits to one hexadecimal symbol.

Example

Binary Number−101012

Calculating hexadecimal Equivalent−

Step	Binary Number	Hexa decimal Number
Step 1	10101_2	0001 0101
Step 2	10101_2	$1_{10}5_{10}$
Step 3	10101_2	15_{16}

Binary Number −101012 = Hexadecimal Number−1516

Shortcut method – Hexadecimal to Binary

Steps

- **Step 1**− Convert each hexadecimal digit to a 4 digit binary number (the hexadecimal digits may be treated as decimal for this conversion).
- **Step 2**− Combine all the resulting binary groups (of 4 digits each) into a single binary number.

Example

Hexadecimal Number – 1516

Calculating Binary Equivalent–

Step	Hexa decimal Number	Binary Number
Step 1	15_{16}	$1_{10}5_{10}$
Step 2	15_{16}	0001_20101_2
Step 3	15_{16}	00010101_2

HexadecimalNumber–15_{16}=BinaryNumber–10101_2

1.2.1 DATA TYPES:

The data types found in the registers of digital computers may be classified as being one of the following categories:

1. Numbers used in arithmetic computation,
2. Letters of the alphabet used in data processing,
3. Other discrete symbols used for specific purposes.

Data are numbers and other binary-coded information that are operated on to achieve required computational results.

All types of data, except binary numbers, are represented in binary-coded form.

1.2.2 COMPLEMENTS

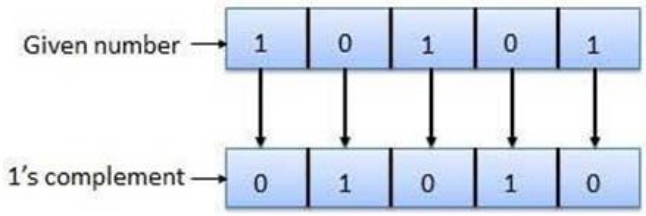
Complements are used in the digital computers in order to simplify the subtraction operation and for the logical manipulations. For each radix-r system (radix r represents base of number system) there are two types of complements.

S.N.	Complement	Description
1	Radix Complement	The radix complement is referred to as the r' s complement
2	Diminished Radix Complement	The diminished radix complement is referred to as the(r-1)'s complement

As the binary system has base r = 2. So the two types of complements for the binary system are 2's complement and 1's complement.

1's complement

The 1's complement of a number is found by changing all 1's to 0's and all 0's to 1's. This is called as taking complement or 1's complement.



For example: +9 will be represented as 00001001 in eight-bit notation and -9 will be represented as 11110110, which is the 1's complement of 00001001.

Example

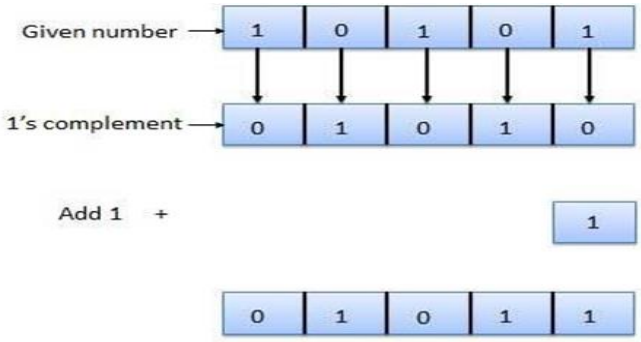
```
1's complement of "0111" is "1000"
1's complement of "1100" is  "0011"
```

2's complement

The 2's complement of binary number is obtained by adding 1 to the Least Significant Bit (LSB) of 1's complement of the number.

2's complement = 1's complement + 1

Example of 2's Complement is as follows



Binary Arithmetic

Binary arithmetic is essential part of all the digital computers and much other digital system.

Binary Addition

It is a key for binary subtraction, multiplication, division. There are four rules of binary addition.

Case	A + B	Sum	Carry
1	0 + 0	0	0
2	0 + 1	1	0
3	1 + 0	1	0
4	1 + 1	0	1

In fourth case, a binary addition is creating a sum of (1 + 1 = 10) i.e. 0 is written in the given column and a carry of 1 over to the next column.

Example – Addition

0011010 + 001100 = 00100110

```
  1 1          carry
  0 0 1 1 0 1 0  = 26₁₀
+ 0 0 0 1 1 0 0  = 12₁₀
-----------------
  0 1 0 0 1 1 0  = 38₁₀
```

Binary Subtraction

Subtraction and Borrow, these two words will be used very frequently for the binary subtraction. There are four rules of binary subtraction.

Case	A - B	Subtract	Borrow
1	0 - 0	0	0
2	1 - 0	1	0
3	1 - 1	0	0
4	0 - 1	0	1

Example – Subtraction

0011010 - 001100 = 00001110

```
    1 1        borrow
  0 0 1 1 0 1 0  = 26₁₀
- 0 0 0 1 1 0 0  = 12₁₀
-----------------
  0 0 0 1 1 1 0  = 14₁₀
```

Binary Multiplication

Binary multiplication is similar to decimal multiplication. It is simpler than decimal multiplication because only 0s and 1s are involved. There are four rules of binary multiplication.

Case	A x B	Multiplication
1	0 x 0	0
2	0 x 1	0
3	1 x 0	0
4	1 x 1	1

Example – Multiplication

Example:

0011010 x 001100 = 100111000

```
      0 0 1 1 0 1 0   = 26₁₀
    x 0 0 0 1 1 0 0   = 12₁₀
  -------------------
      0 0 0 0 0 0 0
    0 0 0 0 0 0 0
  0 0 1 1 0 1 0
0 0 1 1 0 1 0
-------------------
0 1 0 0 1 1 1 0 0 0   = 312₁₀
```

Binary Division

Binary division is similar to decimal division. It is called as the long division procedure.

Example – Division

101010 / 000110 = 000111

```
              1 1 1      = 7₁₀
          ------------
000110 ) 1 0 1 0 1 0     = 42₁₀
       - 1 1 0           = 6₁₀
       -------
         1 0 0 1
       - 1 1 0
       -------
           1 1 0
         - 1 1 0
         -------
               0
```

1.2.3 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	$(0000\ 0001\ 0011\ 0101)_{BCD}$

1.2.4 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 x 10+4

• A floating-point number is always interpreted to represent m x r

Example: the binary number +1001.11 (with 8-bit fraction and 6-bit exponent)

- ❖ Fraction: 01001110
- ❖ Exponent: 000100
- ❖ Equivalent to +(.1001110)2 x 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.

Other Alphanumeric codes:

EBCDIC

- In 1964, BCD was extended to an 8-bit code, Extended Binary-Coded Decimal Interchange Code (EBCDIC).
- EBCDIC was one of the first widely-used computer codes that supported upper and lowercase alphabetic characters, in addition to special characters, such as punctuation and control characters.
- EBCDIC and BCD are still in use by IBM mainframes today.

Character Types	Language	Number of Characters	Hexadecimal Values
Alphabets	Latin, Greek, Cyrillic, etc.	8192	0000 to 1FFF
Symbols	Dingbats, Mathematical, etc.	4096	2000 to 2FFF
CJK	Chinese, Japanese, and Korean phonetic symbols and punctuation.	4096	3000 to 3FFF
Han	Unified Chinese, Japanese, and Korean	40,960	4000 to DFFF
	Han Expansion	4096	E000 to EFFF
User Defined		4095	F000 to FFFE

ASCII

- ASCII: American Standard Code for Information Interchange.
- In 1967, a derivative of this alphabet became the official standard that we now call ASCII.

Unicode

- Both EBCDIC and ASCII were built around the Latin alphabet.
- In 1991, a new international information exchange code called Unicode.
- Unicode is a 16-bit alphabet that is downward compatible with ASCII and Latin-1 character set.

- Because the base coding of Unicode is 16 bits, it has the capacity to encode the majority of characters used in every language of the world.
- Unicode is currently the default character set of the Java programming language.

Table - Unicode Codespace

- The Unicode codespace is divided into six parts. The first part is for Western alphabet codes, including English, Greek, and Russian.
- The lowest-numbered Unicode characters comprise the ASCII code.
- The highest provide for user-defined codes.

Error Detection and Correction Codes

- No communications channel or storage medium can be completely error-free.

➢ Cyclic Redundancy Check:

- Cyclic redundancy check (CRC) is a type of checksum used primarily in data communications that determines whether an error has occurred within a large block or stream of information bytes.
- Arithmetic Modulo 2The rules are as follows:

$0 + 0 = 0$

$0 + 1 = 1$

$1 + 0 = 1$

$1 + 1 = 0$

EXAMPLE 2.35 Find the sum of 1011_2 and 110_2 modulo 2.

$1011_2 + 110_2 = 1101_2$ (mod 2)

EXAMPLE 2.36 Find the quotient and remainder when 1001011_2 is divided by 1011_2.

Quotient 1010_2 and Remainder 101_2

1.3 COMPUTER ARITHMETIC

Arithmetic instructions in digital computers manipulate data to produce results necessary for the solution of computational problems. These instructions perform arithmetic calculations and are responsible for the bulk of activity involved in processing data in a computer. The four basic arithmetic operations are addition, subtraction, multiplication, and division.

The arithmetic operation in the digital computer manipulate data to produce results. It is necessary to design arithmetic procedures and circuits to program arithmetic operations using algorithm. The algorithm is a solution to any problem and it is stated by a finite number of well-defined procedural steps. The algorithms can be developed for the following types of data.

1. Fixed point binary data in signed magnitude representation
2. Fixed point binary data in signed 2's complement representation.

Operation	Add 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)

3. Floating point representation
4. Binary Coded Decimal (BCD) data

1.3.1 ADDITION AND SUBTRACTION WITH SIGNED MAGNITUDE

Consider two numbers having magnitude A and B. When the signed numbers are added or subtracted, there can be 8 different conditions depending on the sign and the operation performed as shown in the table below:

From the table, we can derive an algorithm for addition and subtraction as follows:

Addition (Subtraction) Algorithm:

- When the signs of A & B are identical, add the two magnitudes and attach the sign of A to the result.

- When the sign of A & B are different, compare the magnitude and subtract the smaller number from the large number. Choose the sign of the result to be same as A if A > B, or the complement of the sign of A if A < B. If the two numbers are equal, subtract B from A and make the sign of the result positive.

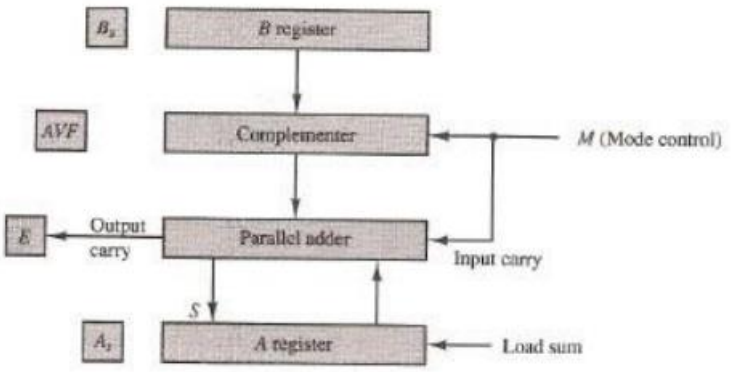


Figure 1.2 - Hardware for signed magnitude addition and subtraction

The hardware consists of two registers A and B to store the magnitudes, and two flipflops As and Bs to store the corresponding signs. The results can be stored in the register A and As which acts as an accumulator. The subtraction is performed by adding A to the 2's complement of B. The output carry is transferred to the flip-flop E. The overflow may occur during the add operation which is stored in the flip-flop A E F.When m = 0, the output of E is transferred to the adder without any change along with the input carry of '0".

The output of the parallel adder is equal to A + B which is an add operation. When m = 1, the content of register B is complemented and transferred to parallel adder along with the input carry of 1. Therefore, the output of parallel is equal to A + B' + 1 = A – B which is a subtract operation.

Hardware Algorithm

As and Bs are compared by an exclusive-OR gate. If output=0, signs are identical, if 1 signs are different.

- For Add operation, identical signs dictate addition of magnitudes and for operation identical signs dictate addition of magnitudes and for subtraction, different magnitudes dictate magnitudes be added. Magnitudes are added with a micro-operation EA.

- Two magnitudes are subtracted if signs are different for add operation and identical for subtract operation. Magnitudes are subtracted with a micro-operation EA = B and number (this number is checked again for 0 to make positive 0 [As=0]) in A is correct result. E = 0 indicates A < B, so we take 2's complement of A.

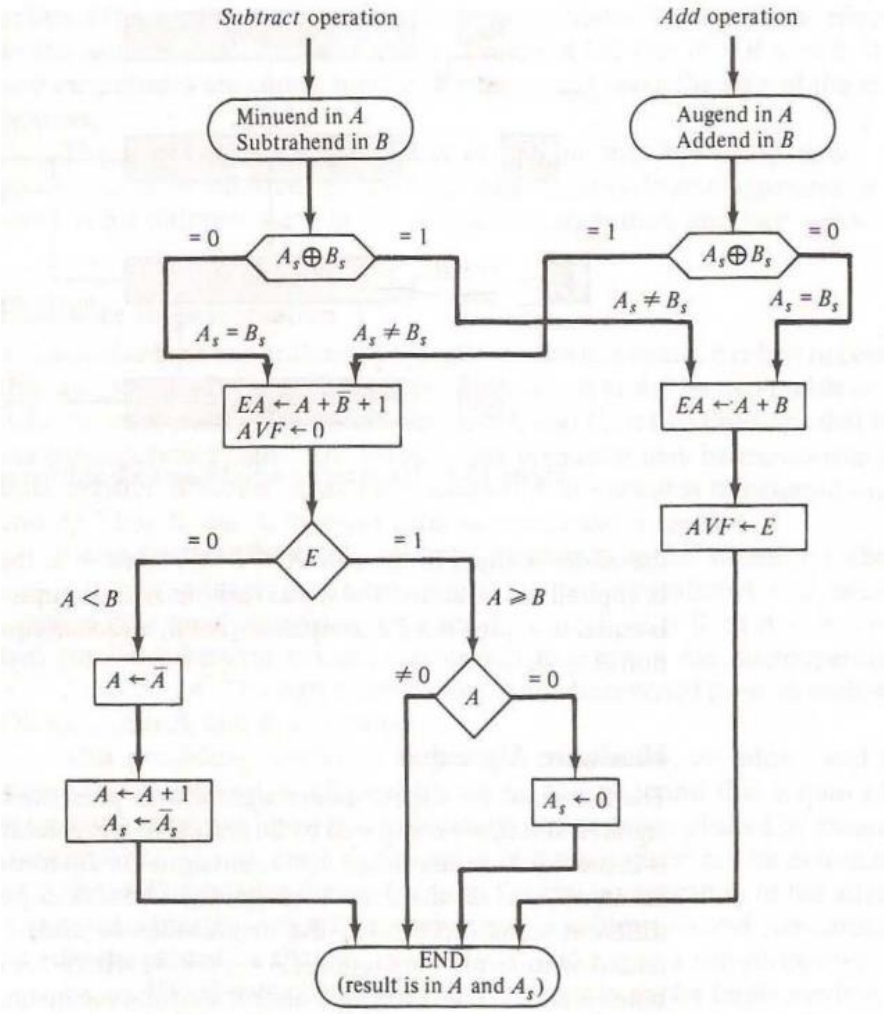


Figure 1.3 - Flowchart for add and subtract operations

1.3.2 MULTIPLICATION

Hardware Implementation and Algorithm

- Generally, the multiplication of two final point binary number in signed magnitude representation is performed by a process of successive shift and ADD operation. The process consists of looking at the successive bits of the multiplier (least significant bit first). If the multiplier is 1, then the multiplicand is copied down otherwise, 0's are

copied. The numbers copied down in successive lines are shifted one position to the left and finally, all the numbers are added to get the product.

- But, in digital computers, an adder for the summation (∑) of only two binary numbers are used and the partial product is accumulated in register. Similarly, instead of shifting the multiplicand to the left, the partial product is shifted to the right. The hardware for the multiplication of signed magnitude data is shown in the figure below.

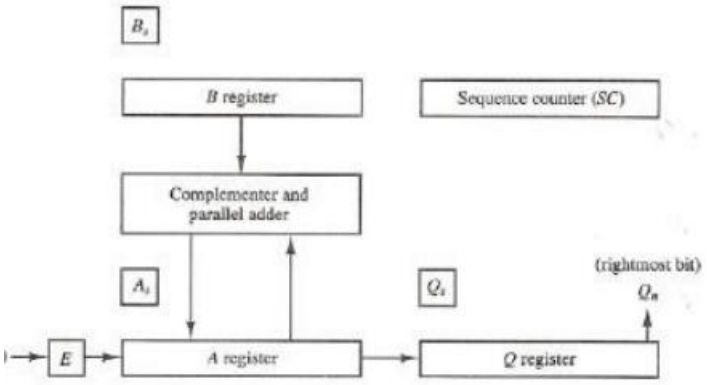


Figure 1.4 - Hardware for multiply operation

- Initially, the multiplier is stored q register and the multiplicand in the B register. A register is used to store the partial product and the sequence counter (SC) is set to a number equal to the number of bits in the multiplier. The sum of A and B form the partial product and both shifted to the right using a statement "Shr EAQ" as shown in the hardware algorithm. The flip flops As, Bs & Qs store the sign of A, B & Q respectively. A binary '0" inserted into the flip-flop E during the shift right.

Hardware Algorithm

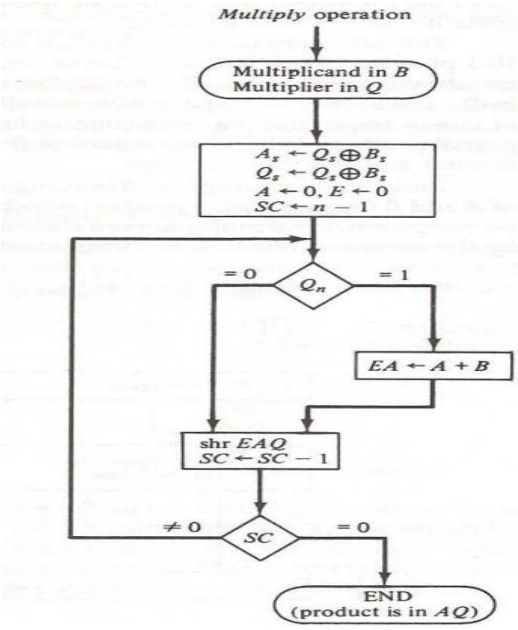


Figure 1.5 -Flowchart for multiply algorithm

Booth Algorithm

The algorithm that is used to multiply binary integers in signed 2's complement form is called booth multiplication algorithm. It works on the principle that the string 0's in the multiplier doesn't need addition but just the shifting and the sting of 1's from bit weight 2k to 2m can be treated as 2k+1 – 2m (Example, +14 = 001110 = 23=1 – 21 = 14). The product can be obtained by shifting the binary multiplication to the left and subtraction the multiplier shifted left once.

According to booth algorithm, the rule for multiplication of binary integers in signed 2's complement form are:

- The multiplicand is subtracted from the partial product of the first least significant bit is 1 in a string of 1's in the multiplicand.
- The multiplicand is added to the partial product if the first least significant bit is 0 (provided that there was a previous 1) in a string of 0's in the multiplier.
- The partial product doesn't change when the multiplier bit is identical to the previous multiplier bit.

This algorithm is used for both the positive and negative numbers in signed 2's complement form. The hardware implementation of this algorithm is in figure below:

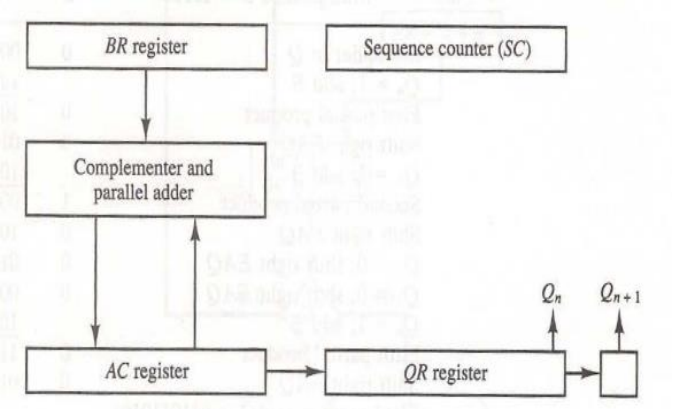


Figure 1.6 - Hardware for Booth algorithm.

The flowchart for booth multiplication algorithm is given below:

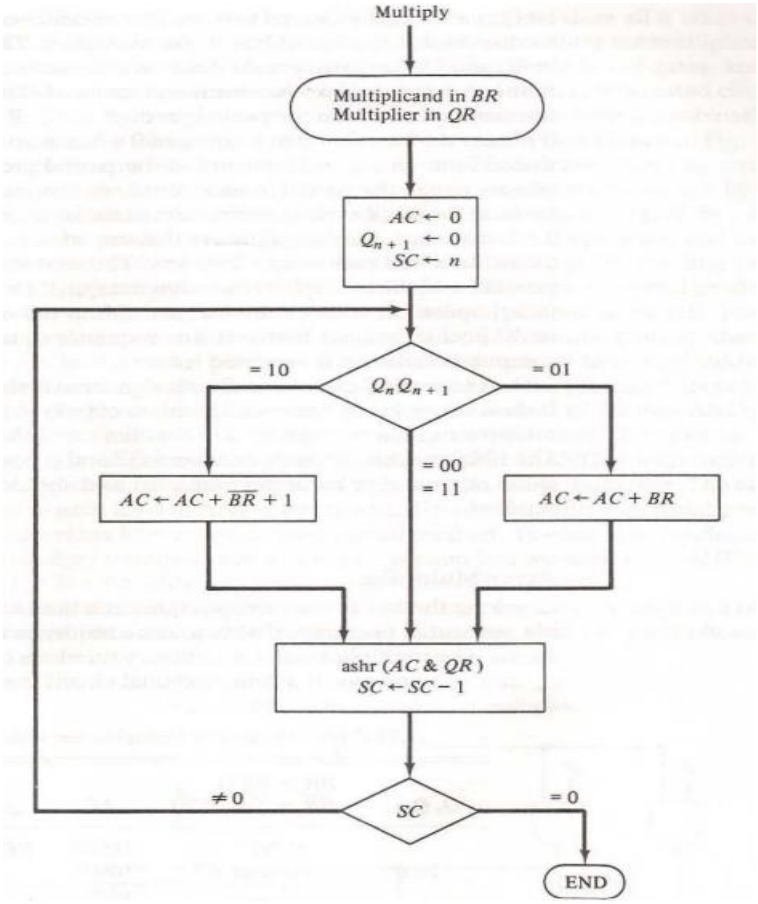


Figure 1.7 - Booth algorithm for multiplication of signed,2's complement numbers.

Numerical Example: Booth algorithm

BR=10111(Multiplicand) QR=10011(Multiplier)

Array Multiplier

The multiplication algorithm first check the bits of the multiplier one at time and form partial product. This is a sequential process that requires a sequence of add and shift micro-operation. This method is complicated and time consuming. The multiplication of 2 binary numbers can also be done with one micro-operation by using combinational circuit that provides the product all at once.

Example: Consider that the multiplicand bits are b1 and b0 and the multiplier bits are a1 and a0. The partial product is c3c2c1c0. The multiplication two bits a0 and a1 produces a binary 1 if both the bits are 1, otherwise it produces a binary 0. This is identical to the AND operation and can be implemented with the AND gates as shown in figure.

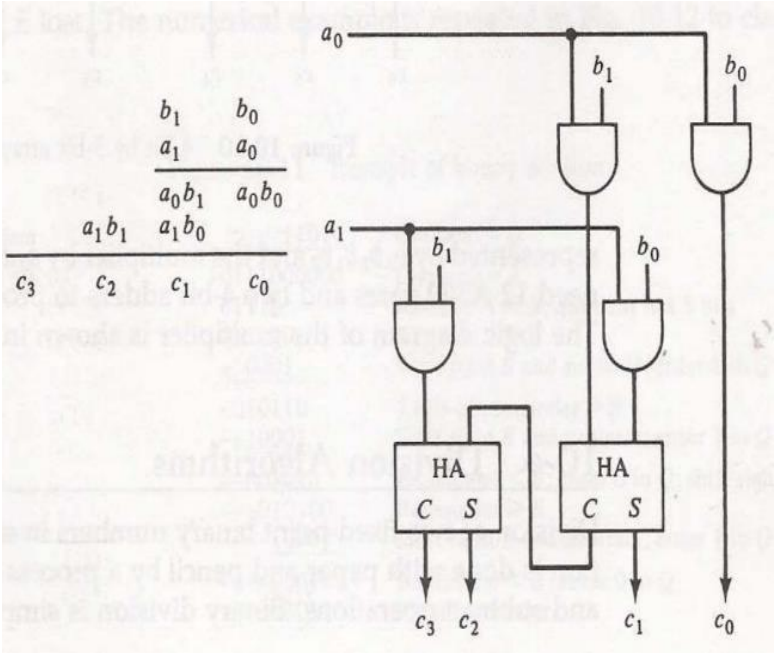


Figure 1.8 - 2-bit by 2-bit array multiplier

1.3.3 DIVISION ALGORITHM

The division of two fixed point signed numbers can be done by a process of successive compare shift and subtraction. When it is implemented in digital computers, instead of shifting the divisor to the right, the dividend or the partial remainder is shifted to the left. The subtraction can be obtained by adding the number A to the 2's complement of number B. The information about the relative magnitudes of the information about the relative magnitudes of numbers can be obtained from the end carry.

The divisor is stored in register B and a double length dividend is stored in register A and Q. the dividend is shifted to the left and the divider is subtracted by adding twice complement of the value. If E = 1, then A >= B. In this case, a quotient bit 1 is inserted into Qn and the partial remainder is shifted to the left to repeat the process. If E = 0, then A > B. In this case, the quotient bit Qn remains zero and the value of B is added to restore the partial remainder in A to the previous value. The partial remainder is shifted to the left and approaches continues until the sequence counter reaches to 0. The registers E, A & Q are shifted to the left with 0 inserted into Qn and the previous value of E is lost as shown in the flow chart for division Algorithm.

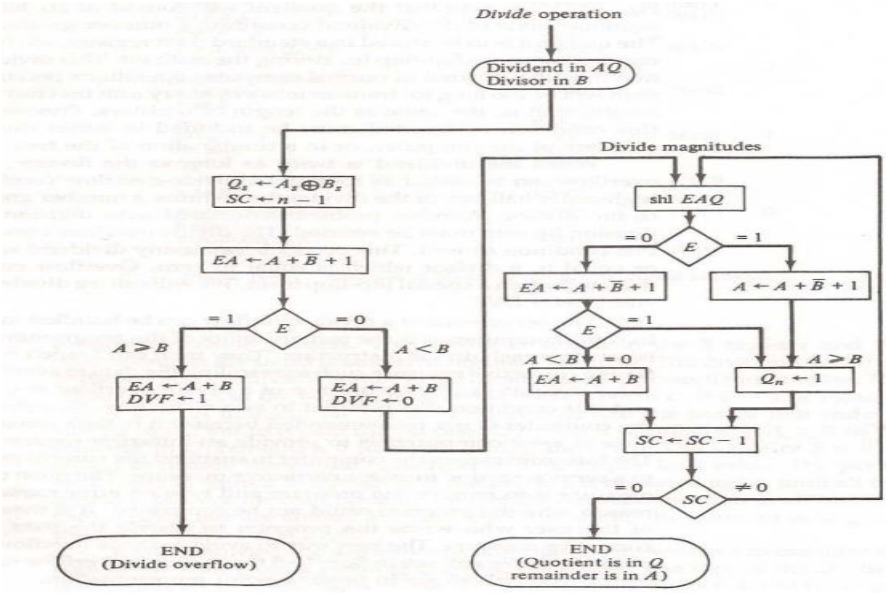


Figure 1.9 - Flowchart for divide operation

This algorithm can be explained with the help of an example.

Consider that the divisor is 10001 and the dividend is 01110.

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 $Q_n = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	11001	00110	
Add B		10001		
Restore remainder	1	01010		2
shl EAQ	0	10100	01100	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00011		
Set $Q_n = 1$	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\overline{B} + 1$		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	

Figure 1.10 - Example of binary division with digital hardware

Arithmetic Operations on Floating-Point Numbers

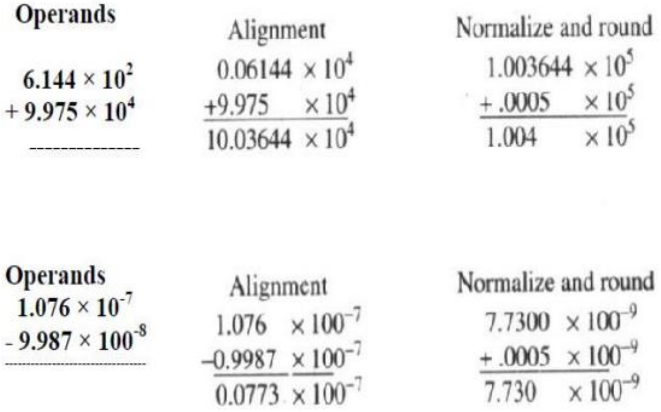
The rules apply to the single-precision IEEE standard format. These rules specify only the major steps needed to perform the four operations. Intermediate results for both mantissas and exponents might require more than 24 and 8 bits, respectively & overflow or an underflow may occur. These and other aspects of the operations must be carefully considered in designing an arithmetic unit that meets the standard. If their exponents differ, the mantissas of floating-point numbers must be shifted with respect to each other before they are added or subtracted. Consider a decimal example in which we wish to add 2.9400 x 10^2 to 4.3100 x 10^4. We rewrite 2.9400 x 10^2 as 0.0294 x 10^4 and then perform addition of the mantissas to get 4.3394 x 10^4. The rule for addition and subtraction can be stated as follows:

Add/Subtract Rule

The steps in addition (FA) or subtraction (FS) of floating-point numbers (s1, eˆ, f1) fad {s2, eˆ 2, f2) are as follows.

1. Unpack sign, exponent, and fraction fields. Handle special operands such as zero, infinity, or NaN(not a number).
2. Shift the significand of the number with the smaller exponent right by bits.

3. Set the result exponent er to max (e1, e2).
4. If the instruction is FA and s1= s2 or if the instruction is FS and s1 ≠ s2 then add the significands; otherwise subtract them.
5. Count the number z of leading zeros. A carry can make z = -1. Shift the result significand left z bits or right 1 bit if z = -1.
6. Round the result significand, and shift right and adjust z if there is rounding overflow, which is a carry-out of the leftmost digit upon rounding.
7. Adjust the result exponent by er = er - z, check for overflow or underflow, and pack the result sign, biased exponent, and fraction bits into the result word.



Multiplication and division are somewhat easier than addition and subtraction, in that no alignment of mantissas is needed.

Decimal Arithmetic Unit

Decimal arithmetic unit refers to a digital function that does decimal micro-operations. This function adds or subtracts decimal numbers by forming 9's or 10's complement of the subtrahend. This decimal arithmetic unit first accepts coded decimal numbers and then generates output in the binary form. Since four bits are necessary to represent every coded decimal digit, a single stage decimal arithmetic unit comprises nine binary input variables and five binary output variables.

Every stage has to comprise four sets of input for the augend digit, four sets of input for the addend digit, and an input-carry. The output comprises four terminals for the sum digit and one for the output-carry.

BCD Adder

BCD adder refers to a 4-bit binary adder that can add two 4-bit words of BCD format. The output of the addition is a BCD-format 4-bit output word, which represents the decimal sum of the addend and augend and a carry that is generated in case this sum exceeds a decimal value of 9. Therefore, BCD adders can perform decimal addition.

Let us examine an arithmetic operation consisting of two decimal digits in BCD, along with a possible carry from a previous stage. The output of the arithmetic operation cannot be more that 9 + 9 + 1 = 19, because no input digit exceeds 9. In case two BCD numbers are applied to a 4-bit binary adder, the adder forms the sum in binary and gives an output ranging from 0 to 19. Here, 1 refers to an output carry.

Table discusses the construction of a BCD adder, wherein the binary numbers are labelled using symbols K, Z8, Z4, Z2, and Z1.

Sum of Binary Digits					Sum of BCD Digits					
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	Decimal
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

Table 1.1 - Derivation of BCD Adder

In the above table, K is the carry. The subscripts below the letter Z represent the weights. The weights, according to the table, are 8, 4, 2, and 1. These weights can be allotted to the four bits in BCD code. The first column contains the binary sum as in the outputs of a 4-bit binary adder. The second column contains of the output sum of two decimal numbers that is

represented in BCD. Now, a simple rule is essential to convert the binary numbers in the first column to the correct BCD digit representation in the second column.

It is evident from table 10.1 that if the binary sum is less than or equal to 1001, then the corresponding BCD number is identical and therefore, there is no need for conversion. However, in case the binary sum is more than 1001, a non-valid BCD representation is obtained. Therefore, the binary 6 (0110) has to be added to the binary sum to convert it to the correct BCD representation and to produce an output carry.

The decimal numbers in BCD are added by employing one 4-bit binary adder and by performing arithmetic operation one digit at a time. To produce a binary sum, first addition is performed on the low-order pair of BCD digits.

In case the output is equal to or greater than 1010, it can be set right by adding 0110 to the binary sum. This would produce an output-carry automatically for the next pair of significant numbers. Then, the subsequent high-order pair of numbers along with input-carry is added to produce their binary sum. In case this output is greater than or equal to 1010, it is set right by adding 0110. This process is repeated until every decimal digit is added.

The entries in table 10.1 help derive the logic circuit that detects the required corrections. When the binary sum has an output carry K = 1, a correction is required. The other six combinations starting from 1010 to 1111 that require corrections have a 1 in position Z8. To differentiate them from binary 1000 and 1001, which also have a 1 in position Z8 , it is specified that either Z4 or Z2 must have a 1.

The following Boolean function is used to express the condition for a correction and an output carry:

$$C = K + Z_8Z_4 + Z_8Z_2$$

In case C = 1, 0110 is added to the binary sum and an output-carry is provided for the next stage.

Figure below depicts a BCD adder circuit to add two BCD numbers in parallel and to produce a sum digit, which is also in BCD. The internal construction of the BCD adder must include the correction logic.

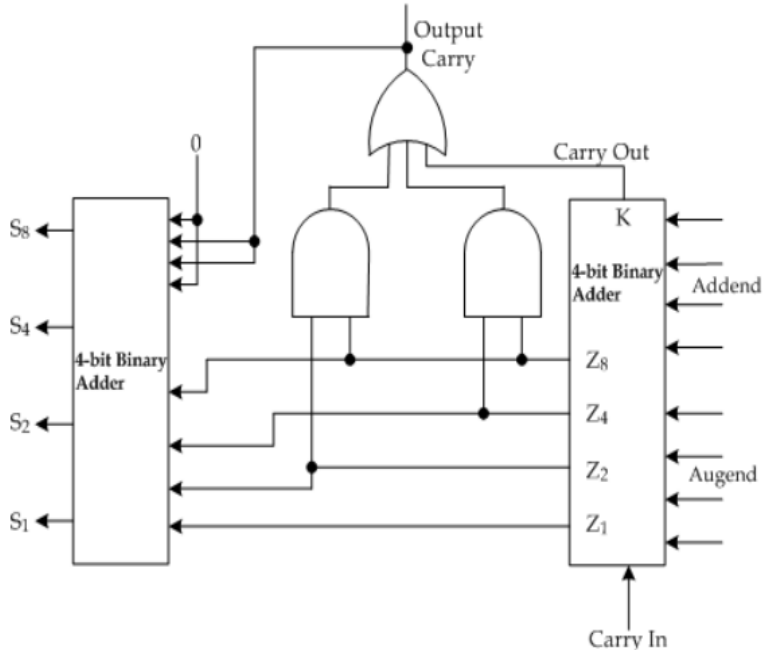


Figure 1.11 - Block diagram of BCD adder

As depicted in figure, a second 4-bit binary adder is used to add 0110. To produce the binary sum, the two decimal digits along with the input-carry are added in the top 4-bit binary adder.

In case the output-carry is equal to 0, no binary number is added to the binary sum. However, in case the output-carry is equal to 1, binary number 0110 is added to the binary sum through the 4- bit binary adder on the left side of figure. The output-carry that is generated from the binary adder on the left side of figure can be ignored, because it provides information that is already present in the output-carry terminal.

A decimal parallel-adder adding n decimal numbers requires n BCD adder stages along with the output-carry from one stage connected to the input-carry of the next-higher order stage. BCD adders comprise the required circuits for carry look-ahead to achieve shorter propagation delays. It should be noted that the adder circuit for correction may not require all four full-adders. It is also possible to optimize the adder circuits.

BCD Subtraction

A subtractor circuit is required to perform a subtraction operation on two decimal numbers. BCD subtraction is slightly different from BCD addition. Performing subtraction operation by taking the 9's or 10's complement of the subtrahend and adding it to the minuend is economical. It is not possible to obtain the 9's complement by complementing every bit in the code because the BCD is not a self- complementing code. The 9's complement has to be formed by a circuit that subtracts Notes every BCD number from 9.

The 9's complement of a decimal digit that is represented in BCD can be obtained by complementing the bits in the coded representation of the digit. However, it is essential that a correction is included. There are two methods of correction. They are:

1. **First Method**: The binary 1010 is added to every complemented digit. The carry is discarded after performing the addition.
2. **Second Method**: The binary 0110 is added before the digit is complemented.

For instance, the 9's complement of BCD 0111 is calculated by complementing every bit to get 1000. The value 0010 is obtained by adding binary 1010 and ignoring the carry. Using the second method, 0110 and 0111 can be added to obtain 1101. The required output, that is, 0010 can be obtained by complementing every bit.

Complementing every bit of a 4-bit binary digit N is the same as subtracting the digit from 1111. When the decimal equivalent of 10 is added, the value obtained is 15 - N + 10 = 9 - N + 16. However, the digit 16 signifies the carry that is discarded, hence, the result equals to 9 - N as required. Adding and then complementing the binary equivalent of decimal 6 provides 15 - (N + 6) = 9 - N as needed.

A combination circuit can also be used to obtain the 9's complement of a BCD digit. When this combination circuit is attached to a BCD adder, it results in a BCD adder or subtractor.

Consider that the subtrahend digit is denoted by the four binary variables B8, B4, B2 and B1. Also consider M to be a mode bit that controls add or subtract operation. Therefore, when M = 0, the two digits are added and when M = 1, the digits are subtracted.

Consider the binary variables x8 , x4 , x2 and x1 to be the outputs of the 9's complementer circuit. According to the truth table for circuits:

1. B_1 needs to be complemented at all times.

2. B_2 is the same every time in 9's complement as in original number.

3. x4 is 1 if the exclusive OR of B_2 and B_4 is 1.

4. x8 is 1 if $B_8B_4B_2 = 000$.

For the 9's complement circuit, the Boolean functions are as follows:

1. $X_1 = B_1M_1 + B_{11}M$
2. $X_2 = B_2$
3. $X_4 = B_4M_1 + (B_{14}B_2 + B_4B_{12})\ M$
4. $X_8 = B_8M_1 + B_{18}B_{14}B_{12}M$

In the above equations it can be observed that x = B if M= 0. If M = 1, the x outputs produce the 9's complement of B.

Decimal Arithmetic Operation

Algorithms that are used for arithmetic operations with decimal data and binary data are alike. If the micro-operations symbol is interpreted correctly the same flowchart can be used for both multiplication and division. The decimal numbers in BCD are stored in groups of four bits in the computer registers. When performing decimal micro-operations, every 4-bit group represents a decimal digit and has to be taken as a group.

For convenience, the same symbol can be used for binary and decimal arithmetic micro-operations with different interpretation. Table depicts symbols for decimal arithmetic micro-operations.

Symbolic Designation	Description
$A \leftarrow A + B$	Add decimal numbers and transfer sum into A
$\overline{B}$	9's complement of B
$A \leftarrow A + \overline{B} + 1$	Content of A plus 10's complement of B into A
$Q_L \leftarrow Q_L + 1$	Increment BCD number in Q_L
dshr A	Decimal shift-right register A
dshl A	Decimal shift-left register A

Table 1.2 - Decimal Arithmetic Microoperation Symbols

In table above, we can see a bar over the symbol for the register letter. This refers to the 9's complement of decimal number that is stored in the register. When 1 is added to the 9's complements the 10's complement is produced. Therefore, the symbol $A \leftarrow A + \overline{B} + 1$ for decimal digits denotes, transfer of decimal sum that was formed by adding the original content A to the 10's complement of B. It may be confusing to use similar symbols for 9's complement and 1's

complement in case both types of data are used in the same system. Therefore, it would be better to implement a different symbol for the 9's complement. In case only one type of data is taken into consideration, the symbol would apply to the type of data used.

Consider that a register A holds a decimal 7860 in BCD. The bit pattern of the 12 flip-flops equal to:

0111 1000 0110 0000.

The micro-operation dshr A moves the decimal number one digit to the right to provide 0786. This shift is over the four bits and as such, changes the content of register to 0000 0111 1000 0110.

MODULE 2
INPUT-OUTPUT ORGANIZATION

2.1 INTRODUCTION

The input-output subsystem of a computer, referred to as I/O, provides an efficient mode of communication between the central system and the outside environment. Programs and data must be entered into computer memory for processing and results obtained from computations must be recorded or displayed for the user.

Input or output devices that are connected to computer are called peripheral devices. These devices are designed to read information into or out of the memory unit upon command from the CPU and are considered to be the part of computer system. These devices are also called peripherals.

For example: Keyboards, display units and printers are common peripheral devices.

There are three types of peripherals:

1. Input peripherals: Allows user input, from the outside world to the computer. Example: Keyboard, Mouse etc.

2. Output peripherals: Allows information output, from the computer to the outside world. Example: Printer, Monitor etc

3. Input-Output peripherals: Allows both input (from outside world to computer) as well as, output (from computer to the outside world).

Example: Touch screen etc.

Interfaces

Interface is a shared boundary between two separate components of the computer system which can be used to attach two or more components to the system for communication purposes.

There are two types of interfaces:

1. CPU Interface
2. I/O Interface

Let's understand the I/O Interface in details,

2.1.1 INPUT-OUTPUT INTERFACE

Input-output interface provides a method for transferring information between internal storage and external I/O devices. Peripherals connected to a computer need special communication links for interfacing them with the central processing unit.

The major differences are:

1. Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory, which are electronic devices. Therefore, a conversion of signal values may be required.
2. The data transfer rate of peripherals is usually slower than the transfer rate of the CPU, and consequently, a synchronization mechanism may be needed.
3. Data codes and formats in peripherals differ from the word format in the CPU and memory.
4. The operating modes of peripherals are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU

A typical communication link between the processor and several peripherals is shown in **Figure 2.1**. The I/O bus consists of data lines, address lines, and control lines. The magnetic disk, printer, and terminal are employed in practically any general-purpose computer. The interface selected responds to the function code and proceeds to execute it. The function code is referred to as an I/O command and is in essence an instruction that is executed in the interface and its attached peripheral unit.

There are three ways that computer buses can be used to communicate with memory and I/O:

1. Use two separate buses, one for memory and the other for I/O.
2. Use one common bus for both memory and I/O but have separate control lines for each.
3. Use one common bus for memory and I/O with common control lines.

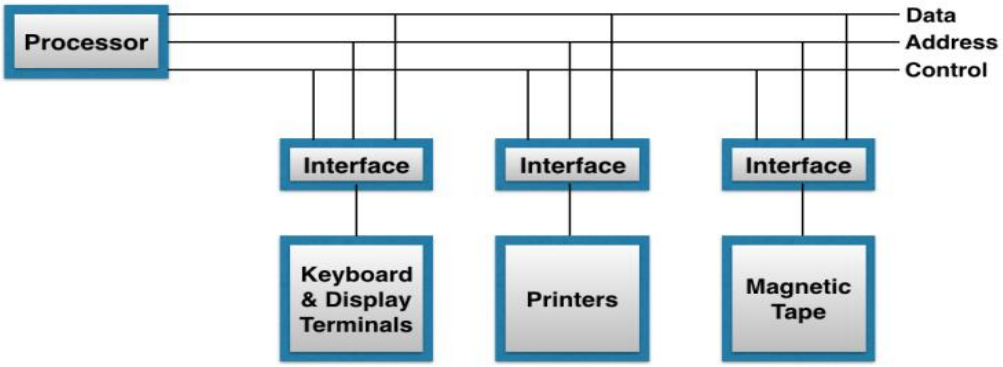
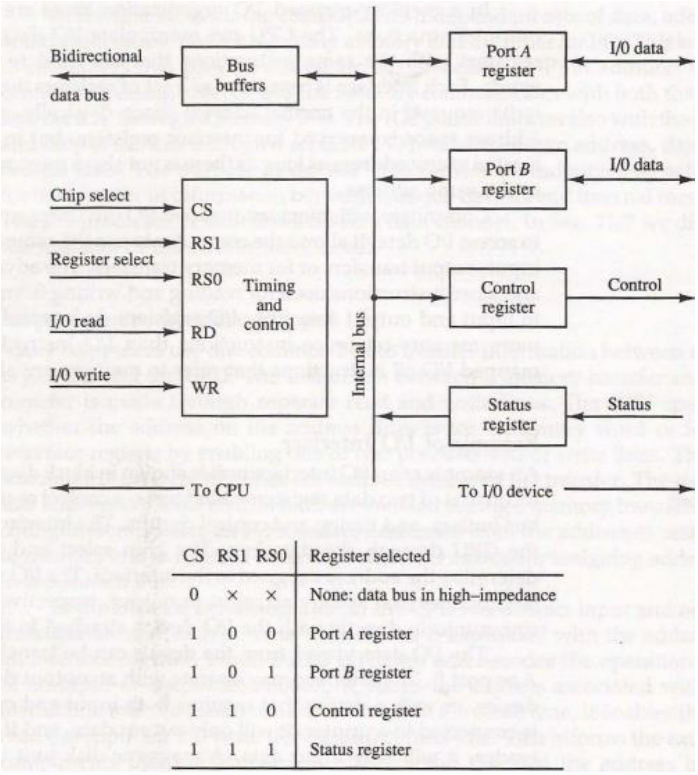


Figure 2.1 - Connection of I/O bus to input-output devices

Isolated I/O versus Memory-Mapped I/O

Many computers use one common bus to transfer information between memory or I/O and the CPU. In the isolated I/O configuration, the CPU has distinct input and output instructions, and each of these instructions is associated with the address of an interface register. The isolated I/O method isolates memory and I/O addresses so that memory address values are not affected by interface address assignment since each has its own address space. The other alternative is to use the same address space for both memory and I/O.



CS	RS1	RS0	Register selected
0	×	×	None: data bus in high–impedance
1	0	0	Port *A* register
1	0	1	Port *B* register
1	1	0	Control register
1	1	1	Status register

Figure 2.2 - Example of I/O Interface Unit

This is the case in computers that employ only one set of read and write signals and do not distinguish between memory and I/O addresses. This configuration is referred to as memory- mapped I/O. In a memory-mapped I/O organization there is no specific input or output instructions. Computers with memory-mapped I/O can use memory-type instructions to access I/O data. An example of an I/O interface unit is shown in block diagram form in Fig.37. It consists of two data registers called ports, a control register, a status register, bus buffers, and timing and control circuits. The interface communicates with the CPU through the data bus. The chip select and register select inputs determine the address assigned to the interface. The I/O read and write are two control lines that specify an input or output, respectively. The four registers communicate directly with the I/O device attached to the interface.

2.1.2 ASYNCHRONOUS DATA TRANSFER

The internal operations in a digital system are synchronized by means of clock pulses supplied by a common pulse generator. If the registers in the interface share a common clock with the CPU registers, the transfer between the two units is said to be synchronous. In most cases, the internal timing in each unit is independent from the other in that each uses its own private clock for internal registers. In that case, the two units are said to be asynchronous to each other. This approach is widely used in most computer systems. Asynchronous data transfer between two independent units requires that control signals be transmitted between the communicating units to indicate the time at which data is being transmitted. Two way of achieving this:

- The strobe: pulse supplied by one of the units to indicate to the other unit when the transfer has to occur.
- The handshaking: The unit receiving the data item responds with another control signal to acknowledge receipt of the data.

The strobe pulse method and the handshaking method of asynchronous data transfer are not restricted to I/O transfers.

The strobe may be activated by either the source or the destination unit. **Figure 2.2** shows a source-initiated transfer and the timing diagram.

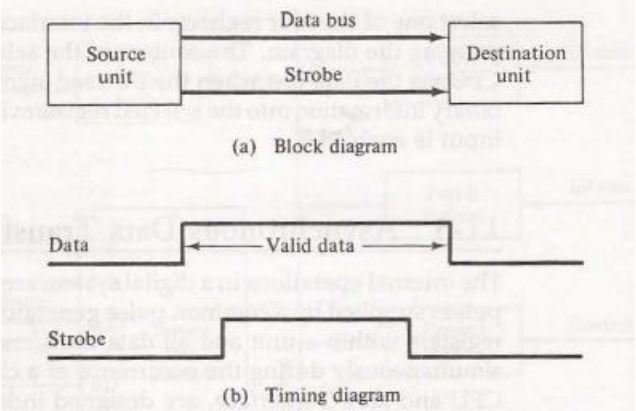


Figure 2.2 - Source, initiated strobe for data transfer

Figure 2.3 shows the strobe of a memory-read control signal from the CPU to a memory.

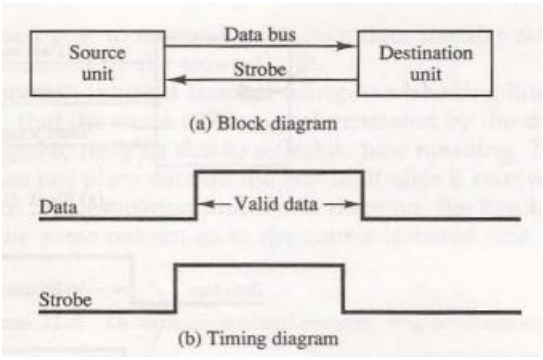


Figure 2.3 – Destination-initiated strobe for data transfer

The disadvantage of the strobe method is that the source unit that initiates the transfer has no way of knowing whether the destination unit has actually received the data item that was placed in the bus. The handshake method solves this problem by introducing a second control signal that provides a reply to the unit that two-wire control initiates the transfer.

Figure 2.4 shows the data transfer procedure when initiated by the source. The two handshaking lines are data valid, which is generated by the source unit, and data accepted, generated by the destination unit. The timing diagram shows the exchange of signals between the two units.

Figure 2.5 the destination-initiated transfer using handshaking lines. Note that the name of the signal generated by the destination unit has been changed to ready for data to reflect its new meaning.

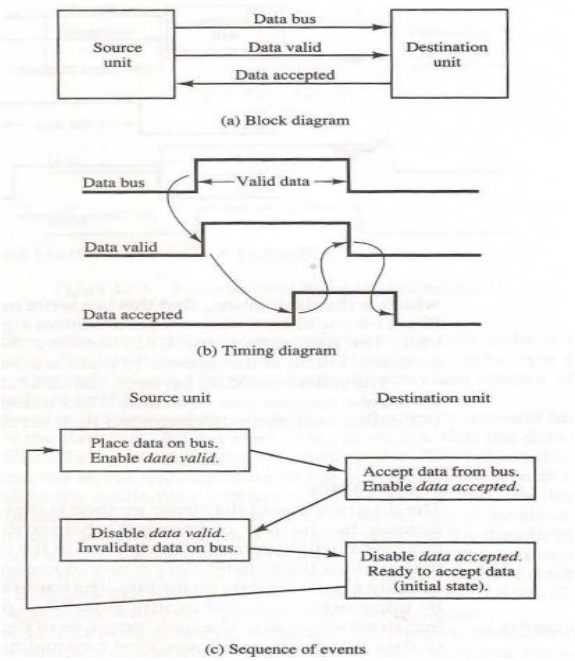


Figure 2.4 - Source-initiated transfer using handshaking

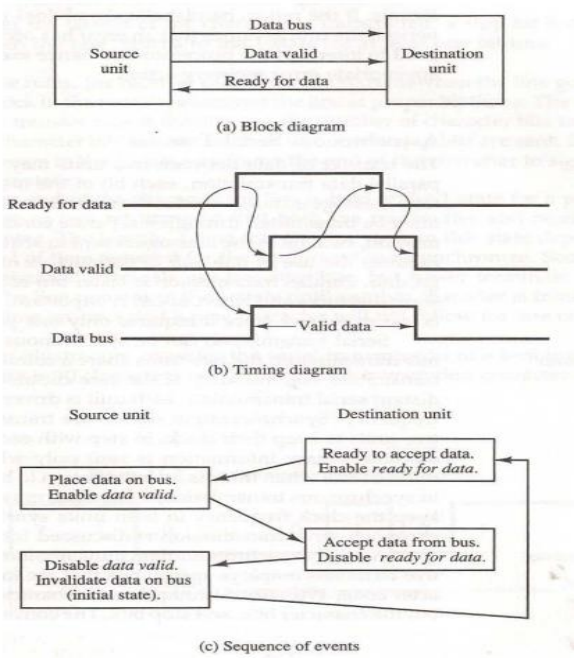


Figure 2.5 - Destination-initiated transfer using handshaking

2.1.3 MODES OF TRANSFER

- Binary information received from an external device is usually stored in memory for later processing. Information transferred from the central computer into an external device originates in the memory unit. The CPU merely executes the I/O instructions and may accept the data temporarily, but the ultimate source or destination is the memory unit.
- Data transfer between the central computer and I/O devices may be handled in a variety of modes. Some modes use the CPU as an intermediate path; other transfer the data directly to and from the memory unit.
- Data transfer to and from peripherals may be handled in one of three possible modes:
    1. Programmed I/O
    2. Interrupt-initiated I/O
    3. Direct memory access (DMA)
- Programmed I/O operations are the result of I/O instructions written in the computer program. Each data item transfer is initiated by an instruction in the program. Usually, the transfer is to and from a CPU register and peripheral. Other instructions are needed to transfer the data to and from CPU and memory. Transferring data under program control

requires constant monitoring of the peripheral by the CPU. Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made. It is up to the programmed instructions executed in the CPU to keep close tabs on everything that is taking place in the interface unit and the I/O device.

- In the programmed I/O method, the CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer. This is a time-consuming process since it keeps the processor busy needlessly. It can be avoided by using an interrupt facility and special commands to inform the interface to issue an interrupt request signal when the data are available from the device. In the meantime, the CU can proceed to execute another program. The interface meanwhile keeps monitoring the device. When the interface determines that the device is ready for data transfer, it generates an interrupt request to the computer. Upon detecting the external interrupt signal, the CPU momentarily stops the task it is processing, branches to a service program to process the /O transfer, and then returns to the task it was originally performing Transfer of data under programmed I/O is between CPU and peripheral.
- In direct memory access (DMA), the interface transfers data into and out of the memory unit through the memory bus. The CPU initiates the transfer by supplying the interface with the starting address and the number of words needed to be transferred and then proceeds to execute other tasks. When the transfer is made, the DMA requests memory cycles through the memory bus. When the request is granted by the memory controller, the DMA transfers the data directly into memory. The CPU merely delays its memory access operation to allow the direct memory TO transfer. Since peripheral speed is usually slower than processor speed, I/O-memory transfers are infrequent compared to processor access to memory.
- Many computers combine the interface logic with the requirements for direct memory access into one unit and call it an I/O processor (IOP). The IOP can handle many peripherals through a DMPA and interrupt facility. In such a system, the computer is divided into three separate modules: the memory unit, the CPU, and the IOP.

Example Of Programmed I/O

- In the programmed I/O method, the I/O device does not have direct access to memory. A transfer from an I/O device to memory requires the execution of several instructions by

the CPU, including an input instruction to transfer the data from the device to the CPU, and a store instruction to transfer the data from the CPU to memory. Other instructions may be needed to verify that the data are available from the device and to count the numbers of words transferred.

- An example of data transfer from an /O device through an interface into the CPU is shown in **Figure 2.6.**

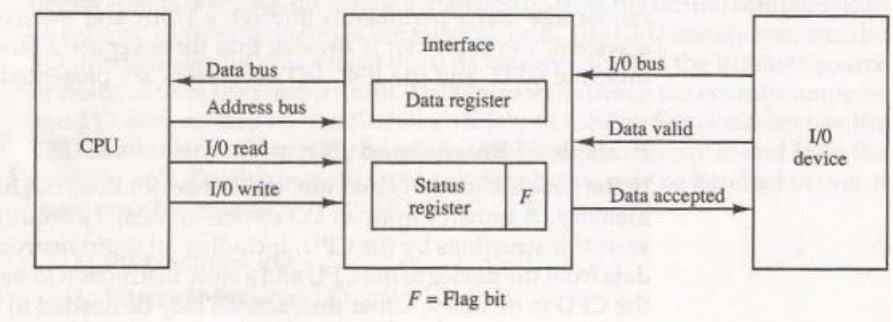


Figure 2.6 - Data transfer from I/O device to CPU

- The device transfers bytes of data one at a time as they are available. When a byte of data is available, the device places it in the I/O bus and enables its data valid line. The interface accepts the byte into its data register and enables the data accepted line. The interface sets a bit in the status register that we will refer to as an F or "flag" bit. The device can now disable the data valid line, but it will not transfer another byte until the data accepted line is disabled by the interface.
- A program is written for the computer to check the flag in the status register to determine if a byte has been placed in the data register by the I/O device. This is done by reading the status register into a CPU register and checking the value of the flag bit. If the flag is equal to 1, the CPU reads the data from the data register. The flag bit is then cleared to 0 by either the CPU or the interface, depending on how the interface circuits are designed. Once the flag is cleared, the interface disables the data accepted line and the device can then transfer the next data byte.

- A flowchart of the program that must be written for the CPU is shown in **Figure 2.7.**

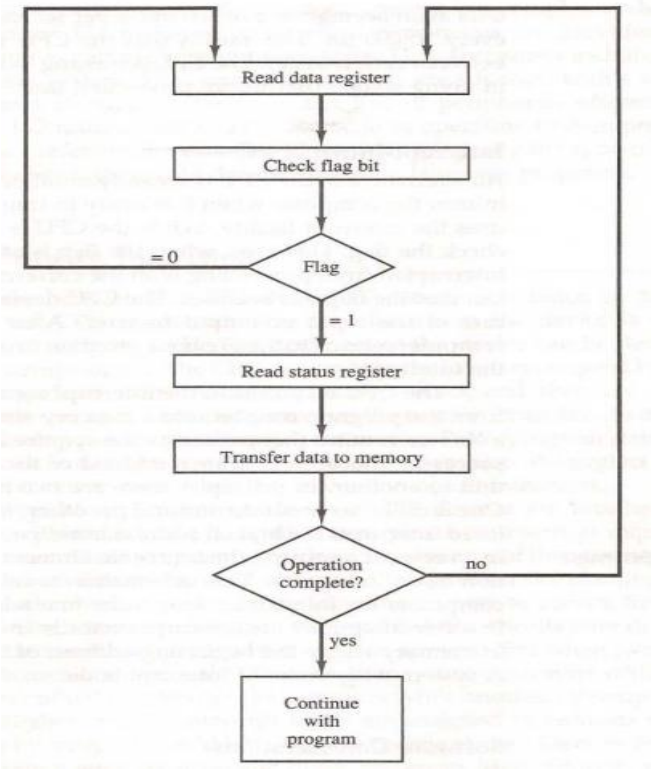


Figure 2.7 - Flowchart for CPU Program to Input Data

It is assumed that the device is sending a sequence of bytes that must be stored in memory. The transfer of each byte requires three instructions:

1. Read the status register.
2. Check the status of the flag bit and branch to step 1 if not set or to step 3 if set.
3. Read the data register.

- Each byte is read into a CPU register and then transferred to memory with a store instruction. A common I/O programming task is to transfer a block of words form an I/O device and store them in a memory buffer.
- The programmed IO method is particularly useful in small low-speed computers or in systems that are dedicated to monitor a device continuously. The difference in information transfer rate between the CPU and the I/O device makes this type of transfer inefficient.

Interrupt-Initiate I/O

- An alternative to the CPU constantly monitoring the flag is to let the interface inform the computer when it is ready to transfer data. This mode of transfer uses the interrupt facility.

'While the CPU is running a program, it does not check the flag. However, when the flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that the flag has been set.

- The CPU deviates from what it is doing to take care of the input or output transfer. After the transfer is completed, the computer returns to the previous program to continue what it was doing before the interrupt.
- The CPU responds to the interrupt signal by storing the return address from the program counter into a memory stack and then control branches to a service routine that processes the required I/O transfer. The way that the processor chooses the branch address of the service routine varies from tone unit to another. In principle, there are two methods for accomplishing this. One is called vectored interrupt and the other, no vectored interrupt. In a non-vectored interrupt, the branch address is assigned to a fixed location in memory. In a vectored interrupt, the source that interrupts supplies the branch information to the computer. This information is called the interrupt vector. In some computers the interrupt vector is the first address of the I/O service routine. In other computers the interrupt vector is an address that points to a location in memory where the beginning address of the I/O service routine is stored.

Software Considerations

- The previous discussion was concerned with the basic hardware needed to interface I/O devices to a computer system. A computer must also have software routines for controlling peripherals and for transfer of data between the processor and peripherals. /O routines must issue control commands to activate the peripheral and to check the device status to determine when it is ready for data transfer. Once ready, information is transferred item by item until all the data are transferred. In some cases, a control command is then given to execute a device function such as stop tape or print characters. Error checking and other useful steps often accompany the transfers.
- In interrupt-controlled transfers, the /O software must issue commands to the peripheral to interrupt when ready and to service the interrupt when it occurs. In DMA transfer, the /O software must initiate the DMA channel to start its operation.

- Software control of input-output equipment is a complex undertaking. For this reason I/O routines for standard peripherals are provided by the manufacturer as part of the computer system. They are usually included within the operating system. Most operating systems are supplied with a variety of I/O programs to support the particular line of peripherals offered for the computer. I/O routines are usually available as operating system procedures and the user refers to the established routines to specify the type of transfer required without going into detailed machine language programs.

2.1.4 PRIORITY INTERRUPT

- Data transfer between the CPU and an I/O device is initiated by the CPU. The CPU cannot start the transfer unless the device is ready to communicate with the CPU. The readiness of the device can be determined from an interrupt signal.
- Numbers of /O devices are attached to the computer; several sources will request service simultaneously. The first task of the interrupt system is to identify the source of the interrupt and decide which device to service first.
- A priority interrupts is a system to determine which interrupt is to be served first when two or more requests are made simultaneously. Also determines which interrupts are permitted to interrupt the computer while another is being serviced. Higher priority interrupts can make requests while servicing a lower priority interrupt.
- Establishing the priority of simultaneous interrupts can be done by software or hardware.
- Priority Interrupt by Software (Polling)

- Priority is established by the order of polling the devices (interrupt sources)
- Flexible since it is established by software
- Low cost since it needs a very little hardware - Very slow

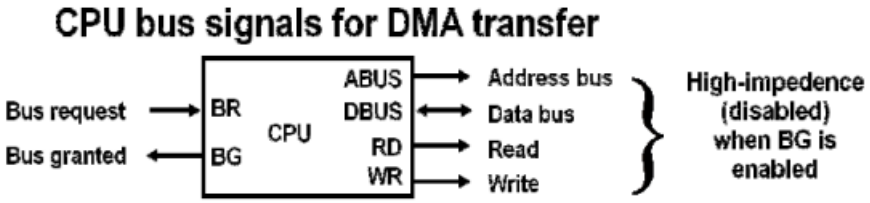
- Priority Interrupt by Hardware

- Require a priority interrupt manager which accepts all the interrupt requests to determine the highest priority request
- Fast since identification of the highest priority interrupt request is identified by the hardware. Each interrupt source has its own interrupt vector to access directly to its own service routine

- The hardware priority function can be established by either a serial or a parallel connection of interrupt lines. The serial connection is also known as the daisy chaining method.

2.1.5 DIRECT MEMORY ACCESS (DMA)

- The transfer of data between a fast storage device such as magnetic disk and memory is often limited by the speed of the CPU. Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer. This transfer technique is called **Direct Memory Access (DMA).**



- The unit communicates with the CPU via the data bus and control lines. The registers in the DMA are selected by the CPU through the address bus by enabling the DS (DMA select) and RS (register select) inputs. The RD (read) and WR (write) inputs are bidirectional.
- 'When the BG (bus grant) input is 0, the CPU can communicate with the DMA registers through the data bus to read from or write to the DMA registers.
- When BG = 1, the CPU has relinquished(ceased) the buses and the DMA can communicate directly with the memory by specifying an address in the address bus and activating the RD or 'WR control.
- The DMA communicates with the external peripheral through the request and acknowledge lines by using a prescribed handshaking procedure.
- The DMA controller has three registers: an address register, a word count register, and a control register. The address register contains an address to specify the desired location in memory. The address bits go through bus buffers into the address bus. The address register is incremented after each word that is transferred to memory.
- The DMA controller has three registers: an address register, a word county register, and a control register. The address register contains an address to specify the desired location in memory. The address bits go through bus buffers into the address bus. The address register is incremented after each word that is transferred to memory.

- The control register specifies the mode of transfer. All registers in the DMA appear to the CPU as I/O interface registers. Thus, the CPU can read from or write into the DMA registers under program control via the data bus.
- The DMA is first initialized by the CPU. After that, the DMA starts and continues to transfer data between memory and peripheral unit until an entire block is transferred. The initialization process is essentially a program consisting of /O instructions that include the address for selecting particular DMA registers. The CPU initializes the DMA by sending the following information through the data bus:
  1. The starting address of the memory block where data are available (for read) or where data are to be stored (for write)
  2. The word count, which is the number of words in the memory block
  3. Control to specify the mode of transfer such as read or write
  4. 4. A control to start the DMA transfer
- The starting address is stored in the address register. The word count is stored in the word count register, and the control information in the control register.
- Once the DMA is initialized, the CPU stops communicating with the DMA unless it receives an interrupt signal or if it wants to check how many words have been transferred.

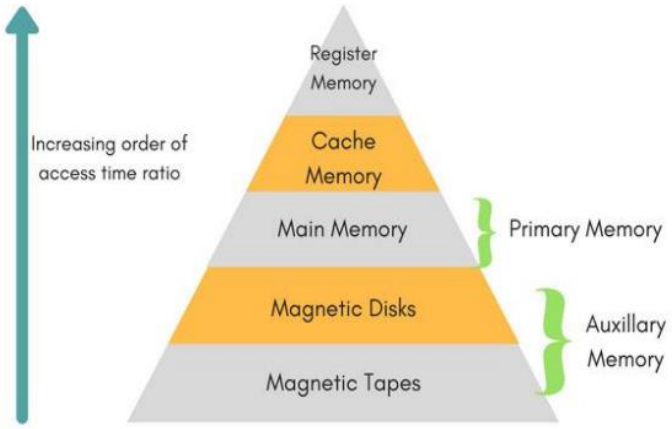
- During DMA transfer, the CPU is idle and has no control of the memory buses.
- A DMA controller takes over the buses to manage the transfer directly between the I/O device and memory.
- The CPU may be placed in an idle state in a variety of ways. One common method extensively used in microprocessors is to disable the buses through special control signals.
- The bus request (BR) input is used by the DMA controller to request the CPU to cease control of the buses. When this input is active, the CPU terminates the execution of the current instruction and places the address bus, the data bus, and the read and write lines into a high impedance state behaves like an open circuit, which means that the output is disconnected and does not have a logic significance.
- The CPU activates the Bus grant (BG) output to inform the external DMA that the buses are in the high-impedance state. The DMA that originated the bus request can now take control of the buses to conduct memory transfers without processor intervention. When the

DMA terminates the transfer, it disables the bus request line. The CPU disables the bus grant, takes control of the buses, and returns to its normal operation.

- 'When the DMA takes control of the bus system, it communicates directly with the memory. The transfer can be made in several ways. In DMA **burst transfer**, a block sequence consisting of a number of memory words is transferred in a continuous burst while the DMA controller is master of the memory buses. This mode of transfer is needed for fast devices such as magnetic disks, where data transmission cannot be stopped or slowed down until an entire block is transferred.
- An alternative technique called **cycle stealing** allows the DMA controller to transfer one data word at a time after which it must return control of the buses to the CPU. The CPU merely delays its operation for one memory cycle to allow the direct memory I/O transfer to "steal" one memory cycle.

2.2 MEMORY ORGANIZATION

2.2.1 MEMORY HIERARCHY



The total memory capacity of a computer can be visualized by hierarchy of components. The memory hierarchy system consists of all storage devices contained in a computer system from the slow Auxiliary Memory to fast Main Memory and to smaller Cache memory.

Auxiliary memory access time is generally 1000 times that of the main memory, hence it is at the bottom of the hierarchy.

The main memory occupies the central position because it is equipped to communicate directly with the CPU and with auxiliary memory devices through Input/output processor (I/O).

When the program not residing in main memory is needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space in main memory for other programs that are currently in use.

The cache memory is used to store program data which is currently being executed in the CPU. Approximate access time ratio between cache memory and main memory is about 1 to 7~10.

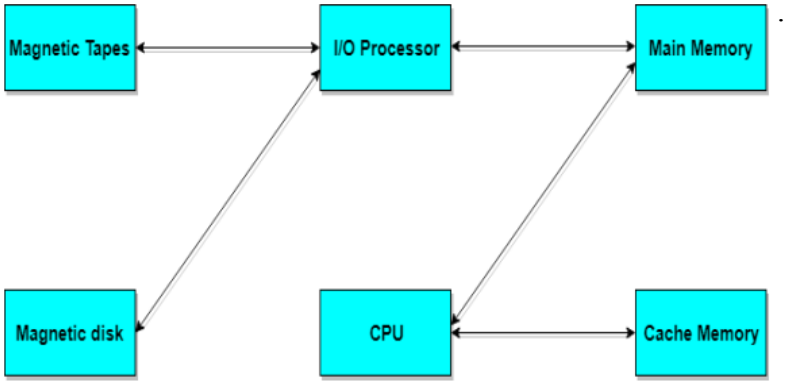


Figure 2.8 - Memory hierarchy in a computer system

Memory Access Methods

Each memory type, is a collection of numerous memory locations. To access data from any memory, first it must be located and then the data is read from the memory location. Following are the methods to access information from memory locations:

1. Random Access: Main memories are random access memories, in which each memory location has a unique address. Using this unique address any memory location can be reached in the same amount of time in any order.
2. Sequential Access: This method allows memory access in a sequence or in order.
3. Direct Access: In this mode, information is stored in tracks, with each track having a separate read/write head.

2.2.2 MAIN MEMORY

The memory unit that communicates directly within the CPU, Auxiliary memory and Cache memory, is called main memory. It is the central storage unit of the computer system. It is a large and fast memory used to store data during computer operations. Main memory is made up of RAM and ROM, with RAM integrated circuit chips holing the major share.

- RAM: Random Access Memory
  - DRAM: Dynamic RAM, is made of capacitors and transistors, and must be refreshed every 10~100 ms. It is slower and cheaper than SRAM.
  - SRAM: Static RAM, has a six-transistor circuit in each cell and retains data, until powered off.
  - NVRAM: Non-Volatile RAM, retains its data, even when turned off. Example: Flash memory.

- ROM: Read Only Memory, is non-volatile and is more like a permanent storage for information. It also stores the bootstrap loader program, to load and start the operating system when computer is turned on. PROM(Programmable ROM), EPROM(Erasable PROM) and EEPROM(Electrically Erasable PROM) are some commonly used ROMs.

2.2.3 Auxiliary Memory

The most common auxiliary memory devices used in computer systems are magnetic disks and magnetic 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. 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.

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 drums and disks are quite similar in operation. Both consist of high-speed rotating surfaces coated with a magnetic recording medium. The rotating surface of the drum is a cylinder and that of the disk, a round flat plate. Bits are recorded as magnetic spots on the surface as it passes a stationary mechanism called a write head. Stored bits are detected by a

change in magnetic field produced by a recorded spot on the surface as it passes through a read head.

Magnetic Disks

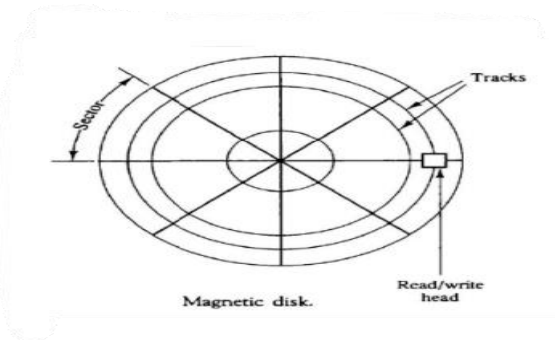
- A magnetic disk is a circular plate constructed of metal or plastic coated with magnetized material. Often both sides of the disk are used and several disks may be stacked on one spindle with read/write heads available on each surface.
- All disks rotate together at high speed and are not stopped or started from 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.
- Some units use a single read/write head from each disk surface. 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 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.
- 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.

Magnetic Tape

- The Magnetic 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 be stopped, started to move forward or in reverse, or can be rewound.
- 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 of characters in the record. Records may be of fixed or variable length.



2.2.4 ASSOCIATIVE MEMORY

- Many data-processing applications require the search of items in a table stored in memory. An assembler program searches the symbol address table in order to extract the symbol's binary equivalent.
- The number of accesses to memory depends on the location of the item and the efficiency of the search algorithm. Many search algorithms have been developed to minimize the number of accesses while searching for an item in a random or sequential access memory.
- 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).**
- When a word is to be read from an associative memory, the content of the word, or part of the word, is specified. The memory locates all words which match the specified content and marks them for reading.
- An associative memory is more expensive than a random-access memory because each cell must have storage capability as well as logic circuits for matching its content with an external argument. For this reason, associative memories are used in applications where the search time is very critical and must be very short.

Hardware Organization

- The block diagram of an associative memory is shown Argument register (4) in **Figure.**

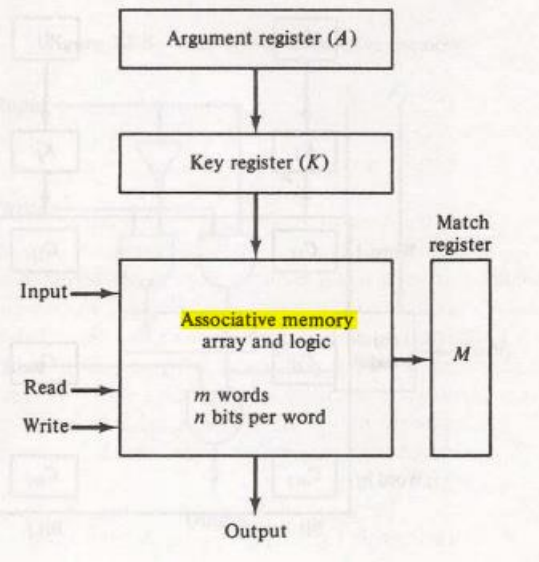


Figure 2.9 - Block diagram of associative memory.

- It consists of a memory array and logic for 'm' words with # 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 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 1's in their corresponding position of the key register is compared.
- 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 positions.

A 101 111100

K 111 000000

Word 1 100 111100 no match

Word 2 101 000001 match

- The relation between the memory array and external registers in an associative memory is shown in **Figure 2.10.**

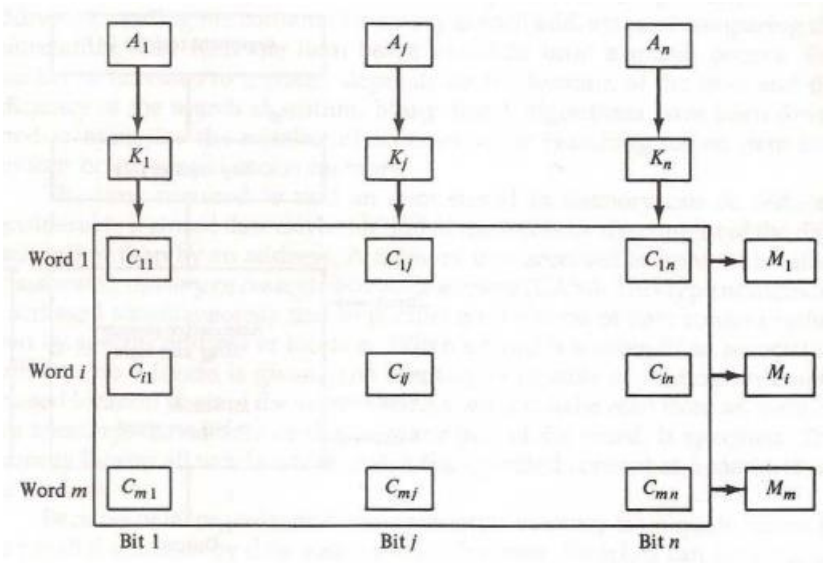


Figure 2.10 - Associative Memory of M Word, N Cells Per Word.

- 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 Cjj is the cell for bit j in word i.
- A bit Aj in the argument register is compared with l 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.
- The internal organization of a typical cell Cij is shown in **Figure 2.11.**

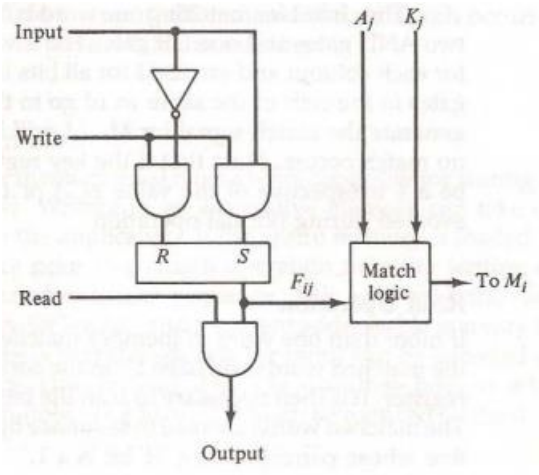


Figure 2.11 - One Cell of Associative Memory

- It consists of a flipflop storage element Fij and the circuits for reading, writing, and matching the cell.
- The input bit is transferred into the storage cell during a write operation. The bit stored is read out during a read operation.
- The match logic compares the content of the storage cell with the corresponding unmasked bit of the argument and provides an output for the decision logic that sets the bit in Mi.

2.2.5 CACHE MEMORY

Locality of Reference: The references to memory at any given time interval tends to be confined within a localized area.

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 is fetched from memory. Thus, loops and subroutines tend to localize the references to memory for fetching instructions.

Iterative procedures refer to common memory locations and array of numbers are confined within a local portion of memory.

If the active portions of the program and data are placed in a fast small memory, the average memory access time can be reduced, thus reducing the total execution time of the program. 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.

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 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. The ratio of the number of hits divided by the total CPU references to memory (hits plus misses) is the hit ratio. The average memory access time of a computer system can be improved considerably by use of a cache.

The transformation of data from main memory to cache memory is referred to as a mapping process. Three types of mapping procedures are:

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

Consider the following memory organization:

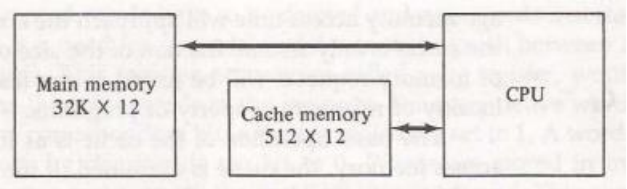


Figure 2.12 - Example of Cache Memory.

Associative Mapping

- The faster and most flexible cache organization use an associative memory. 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.
- 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.

- The decision as to what pair is replaced is determined from the replacement algorithm that the designer chooses for the cache. A simple procedure is to replace cells of the cache in round-robin order whenever a new word is requested from main memory. This constitutes a first-in first-out (FIFO) replacement policy.

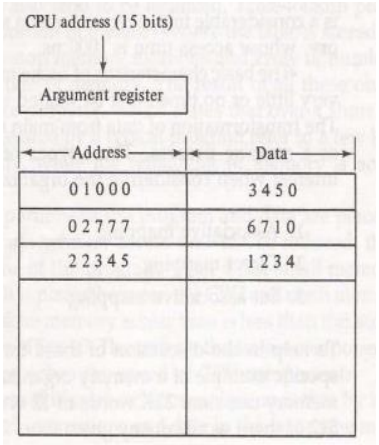


Figure 2.13 - Associative Mapping Cache (All Numbers In Octal)

Direct Mapping

- Associative memories are expensive compared to random-access memories because of the added logic associated with each cell.
- Direct mapping uses RAM instead of CAM.
- 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.
- The internal organization of the words in the cache memory is as shown in **Figure 2.14.**

- Each word in cache consists of the data word and its associated tag. When a new word is first brought in to 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.

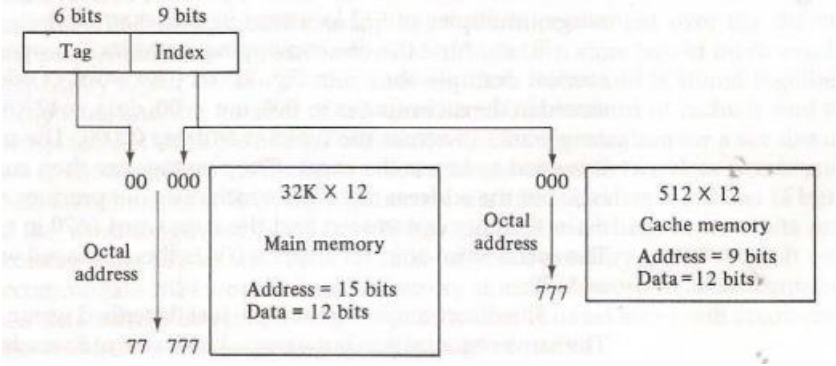


Figure 2.14 - Addressing Relationships Between Main and Cache Memories

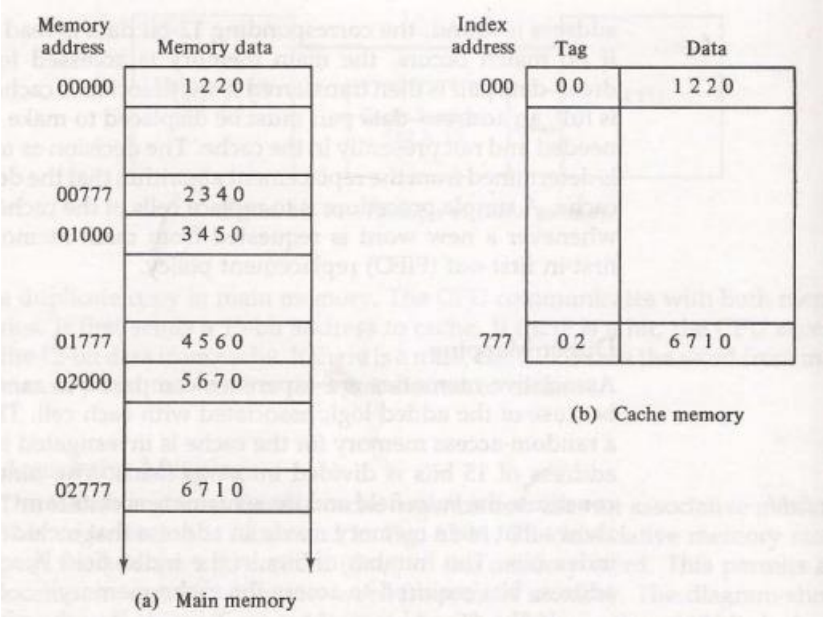


Figure 2.15 - Direct Mapping Cache Organization

- 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 disadvantage of direct mapping is that the hit ratio can drop considerably if two or more words whose addresses have the same index but different tags are accessed repeatedly.

- Suppose that the CPU now wants to access the word at address 02000. The index address is 000, so it is sued 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 direct-mapping uses a block size of one word. The same organization but using a block size of 8 words is shown in **Figure 2.16.**

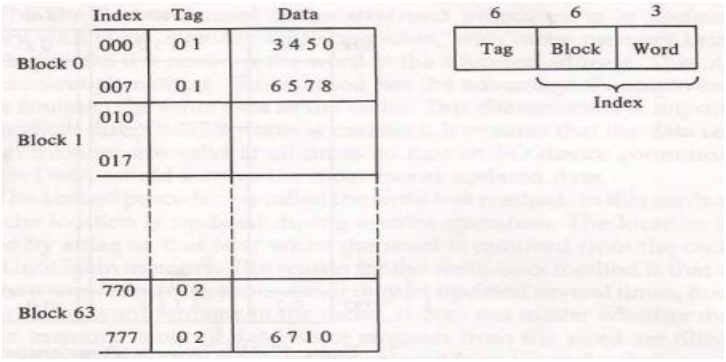


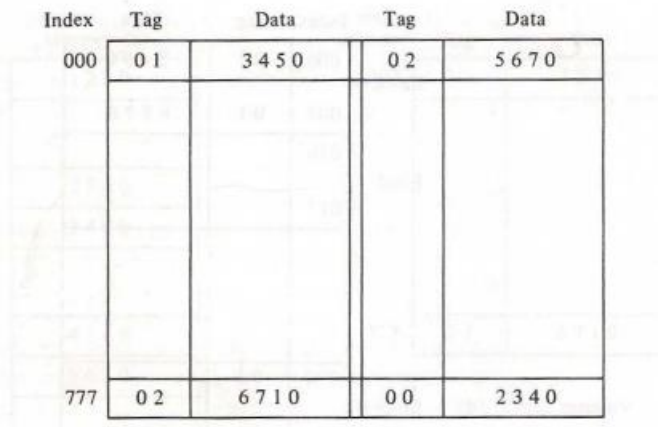
Figure 2.16 - Direct Mapping Cache With Block Size of 8 Words

- The index field is now divided into two parts: the blockfield and the word 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. Although this takes extra time, the hit ratio will most likely improve with a larger block size because of the sequential nature of computer programs.

Set Associative Mapping

- 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 index address refers to two data words and their associated tags. Each tag requires six bits and each data word has 12 bits, so the word length is 2(6 + 12) = 36 bits. An index address of nine bits can accommodate 512 words. Thus, the size of cache memory is 512 x 36. It can accommodate 1024.

- 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 hit ratio will improve as the set size increases because more words with the same index but different tags can reside in cache.
- 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).



Index	Tag	Data	Tag	Data
000	01	3450	02	5670
777	02	6710	00	2340

Figure 2.17 - Two,Way Set ... Associative Mapping Cache

Writing Into Cache

- An important aspect of cache organization is concerned with memory write requests. If the operation is a write, there are two ways that the system can proceed.
- The simplest and most commonly used procedure is to up data main memory with every memory write operation, with 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.
- 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 words are 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.

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. After initialization the cache is considered to be empty, built in effect it contains some non-valid data. It is customary to include with each word in cache a valid bit to indicate whether or not the word contains valid data.
- 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 introduction of the valid bit means that a word in cache is initialization condition has the effect of forcing misses from the cache until it fills with valid data.

MODULE 3
MICROPROGRAMMED CONTROL

3.1 MICROPROGRAMMED CONTROL

Control Unit - The function of the control unit in a digital computer is to initiate sequences of microoperations. The complexity of the digital system is derived from the number of sequences of microoperations that are performed.

Two major types of Control Unit:

1. Hardwired Control Unit
2. Micro Programmed Control Unit

I. Hardwired Control Unit:

a. When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be hardwired.

II. Micro Programmed Control Unit:

A control unit whose binary control variables are stored in memory is called a micro programmed control unit.

3.1.1 CONTROL MEMORY

Control Memory is the storage in the microprogrammed control unit to store the microprogram.

Writeable Control Memory: Control Storage whose contents can be modified, allow the change in microprogram and Instruction set can be changed or modified is referred as Writeable Control Memory.

Control Word: The control variables at any given time can be represented by a control word string of 1 's and 0's called a control word.

A control unit whose binary control variables are stored in memory is called a ***Microprogrammed Control Unit***. Each word in control memory contains within it a ***Microinstruction***. A sequence of microinstructions constitutes a ***Microprogram***. The control memory can be a ***Read-Only Memory (ROM).*** ROM words are made permanent during the

hardware production of the unit.The content of the word in ROM at a given address specifies a microinstruction.

Microoperations:

- In computer central processing units, micro-operations (also known as a micro-ops or μops) are detailed low-level instructions used in some designs to implement complex machine instructions (sometimes termed macro-instructions in this context).

Microinstruction:

- A symbolic microprogram can be translated into its binary equivalent by means of an assembler.
- Each line of the assembly language microprogram defines a symbolic microinstruction.
- Each symbolic microinstruction is divided into five fields: label, microoperations, CD, BR, and AD.

Microprogram:

- A sequence of microinstructions constitutes a microprogram.
- Since alterations of the microprogram are not needed once the control unit is in operation, the control memory can be a read-only memory (ROM).
- ROM words are made permanent during the hardware production of the unit.
- The use of a micro program involves placing all control variables in words of ROM for use by the control unit through successive read operations.
- The content of the word in ROM at a given address specifies a microinstruction.

Microcode:

- Microinstructions can be saved by employing subroutines that use common sections of microcode.
- For example, the sequence of microoperations needed to generate the effective address of the operand for an instruction is common to all memory reference instructions.

- This sequence could be a subroutine that is called from within many other routines to execute the effective address computation.

Dynamic Microprogramming:

A more advanced development known as dynamic microprogramming permits a microprogram to be loaded initially from an auxiliary memory such as a magnetic disk. Control units that use dynamic microprogramming employ a writable control memory. This type of memory can be used for writing(to change the microprogram) but is used mostly for reading. A memory that is part of a control unit is referred to as a ***Control Memory.***

A computer that employs a microprogrammed control unit will have two separate memories:

a. Main Memory
b. Control Memory

The **Main Memory** is available to the user for storing the programs.

the **Control Memory** holds a fixed microprogram that cannot be altered by the occasional user.

Organization of Micro Programmed Control Unit:

- The general configuration of a micro-programmed control unit is demonstrated in the block diagram of **Figure 3.1**.
- The control memory is assumed to be a ROM, within which all control information is permanently stored.

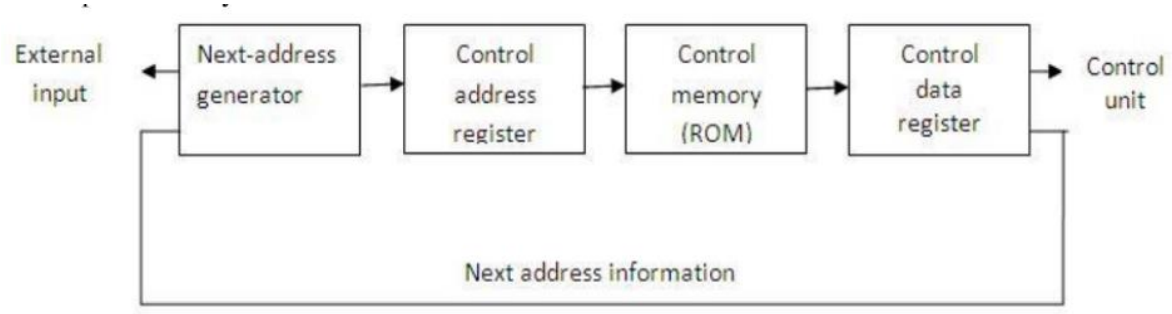


Figure 3.1 - Micro-Programmed Control Organization

- The control memory address register specifies the address of the microinstruction, and the control data register holds the microinstruction read from memory.

- The microinstruction contains a control word that specifies one or more microoperations for the data processor. Once these operations are executed, the control must determine the next address.
- The location of the next microinstruction may be the one next in sequence, or it may be located somewhere else in the control memory.
- While the microoperations are being executed, the next address is computed in the next address generator circuit and then transferred into the control address register to read the next microinstruction.
- Thus, a microinstruction contains bits for initiating microoperations in the data processor part and bits that determine the address sequence for the control memory.
- The next address generator is sometimes called a ***micro-program sequencer***, as it determines the address sequence that is read from control memory.
- Typical functions of a micro-program sequencer are incrementing the control address register by one, loading into the control address register an address from control memory, transferring an external address, or loading an initial address to start the control operations.
- The control data register holds the present microinstruction while the next address is computed and read from memory.
- The data register is sometimes called a ***pipeline register.***
- It allows the execution of the microoperations specified by the control word simultaneously with the generation of the next microinstruction.
- This configuration requires a two-phase clock, with one clock applied to the address register and the other to the data register.
- The main advantage of the micro programmed control is the fact that once the hardware configuration is established; there should be no need for further hardware or wiring changes.
- If we want to establish a different control sequence for the system, all we need to do is specify a different set of microinstructions for control memory.

3.1.2 ADDRESS SEQUENCING

- Microinstructions are stored in control memory in groups, with each group specifying a routine.
- Each computer instruction has its own microprogram routine in control memory to generate the microoperations that execute the instruction.
- The hardware that controls the address sequencing of the control memory must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another.
- To appreciate the address sequencing in a micro-program control unit, let us enumerate the steps that the control must undergo during the execution of a single computer instruction.

Step-1:

- An initial address is loaded into the control address register when power is turned on in the computer.
- This address is usually the address of the first microinstruction that activates the instruction fetch routine.
- The fetch routine may be sequenced by incrementing the control address register through the rest of its microinstructions.
- At the end of the fetch routine, the instruction is in the instruction register of the computer.

Step-2:

- The control memory next must go through the routine that determines the effective address of the operand.
- A machine instruction may have bits that specify various addressing modes, such as indirect address and index registers.
- The effective address computation routine in control memory can be reached through a branch microinstruction, which is conditioned on the status of the mode bits of the instruction.
- When the effective address computation routine is completed, the address of the operand is available in the memory address register.

Step-3:

- The next step is to generate the microoperations that execute the instruction fetched from memory. The microoperation steps to be generated in processor registers depend on the operation code part of the instruction.
- Each instruction has its own micro-program routine stored in a given location of control memory.
- The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a mapping process.
- A mapping procedure is a rule that transforms the instruction code into a control memory address.

Step-4:

- Once the required routine is reached, the microinstructions that execute the instruction may be sequenced by incrementing the control address register.
- Micro-programs that employ subroutines will require an external register for storing the return address.
- Return addresses cannot be stored in ROM because the unit has no writing capability.
- When the execution of the instruction is completed, control must return to the fetch routine.
- This is accomplished by executing an unconditional branch microinstruction to the first address of the fetch routine.

The address sequencing capabilities required in a control memory are:

1. Incrementing of the control address register.
2. Unconditional branch or conditional branch, depending on status bit conditions.
3. A mapping process from the bits of the instruction to an address for control memory.
4. A facility for subroutine call and return.

The **Figure 3.2** shows a block diagram of a control memory and the associated hardware needed for selecting the next microinstruction address.

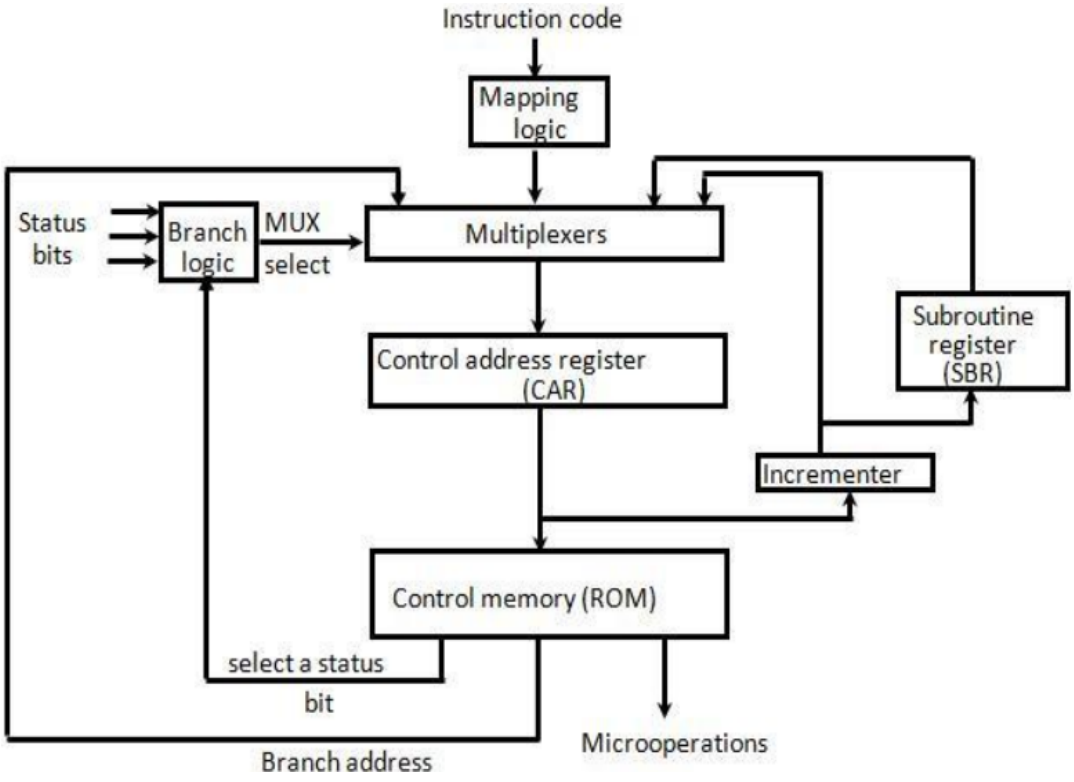


Figure 3.2 - Selection of Address for Control Memory

- The microinstruction in control memory contains a set of bits to initiate microoperations in computer registers and other bits to specify the method by which the next address is obtained.
- The diagram shows four different paths from which the control address register (CAR) receives the address.
- The incrementer increments the content of the control address register by one, to select the next microinstruction in sequence.
- Branching is achieved by specifying the branch address in one of the fields of the microinstruction.
- Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition.
- An external address is transferred into control memory via a mapping logic circuit.

- The return address for a subroutine is stored in a special register whose value is then used when the micro-program wishes to return from the subroutine.

Conditional Branching

- The branch logic of **Figure 3.2** provides decision-making capabilities in the control unit.
- The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and input or output status conditions.
- The status bits, together with the field in the microinstruction that specifies a branch address, control the conditional branch decisions generated in the branch logic.
- A 1 output in the multiplexer generates a control signal to transfer the branch address from the microinstruction into the control address register.
- A 0 output in the multiplexer causes the address register to be incremented.
- In this configuration, the microprogram follows one of two possible paths, depending on the value of the selected status bit.
- An ***Unconditional Branch*** microinstruction can be implemented by loading the branch address from control memory into the control address register.
    - This can be accomplished by fixing the value of one status bit at the input of the multiplexer, so it is always equal to 1.

Mapping of an Instruction

- A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine for an instruction is located.
- The status bits for this type of branch are the bits in the operation code part of the instruction. For example, a computer with a simple instruction format as shown in **Figure 2.1.2** has an operation code of four bits which can specify up to 16 distinct instructions.
- Assume further that the control memory has 128 words, requiring an address of seven bits.

- One simple mapping process that converts the 4-bit operation code to a 7-bit address for control memory is shown in **Figure 3.3**.
- This mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the two least significant bits of the control address register.
- This provides for each computer instruction a microprogram routine with a capacity of four microinstructions.
- If the routine needs more than four microinstructions, it can use addresses 1000000 through 1111111. If it uses fewer than four microinstructions, the unused memory locations would be available for other routines.

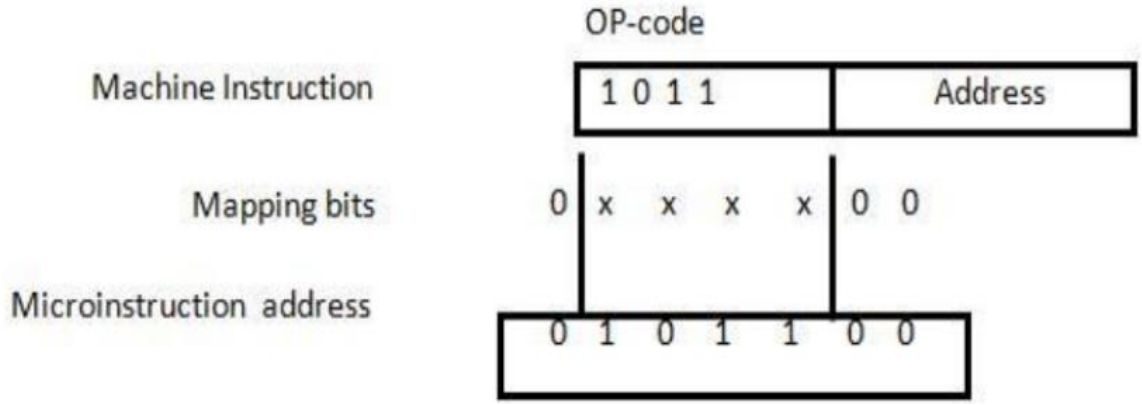


Figure 3.3 - Mapping from Instruction Code to Microinstruction Address

- One can extend this concept to a more general mapping rule by using a ROM to specify the mapping function.
- The contents of the mapping ROM give the bits for the control address register.
- In this way the microprogram routine that executes the instruction can be placed in any desired location in control memory.
- The mapping concept provides flexibility for adding instructions for control memory as the need arises.

3.1.3 MICROPROGRAM EXAMPLE

Computer Hardware Configuration

The block diagram of the computer is shown in **Figure 3.4.**

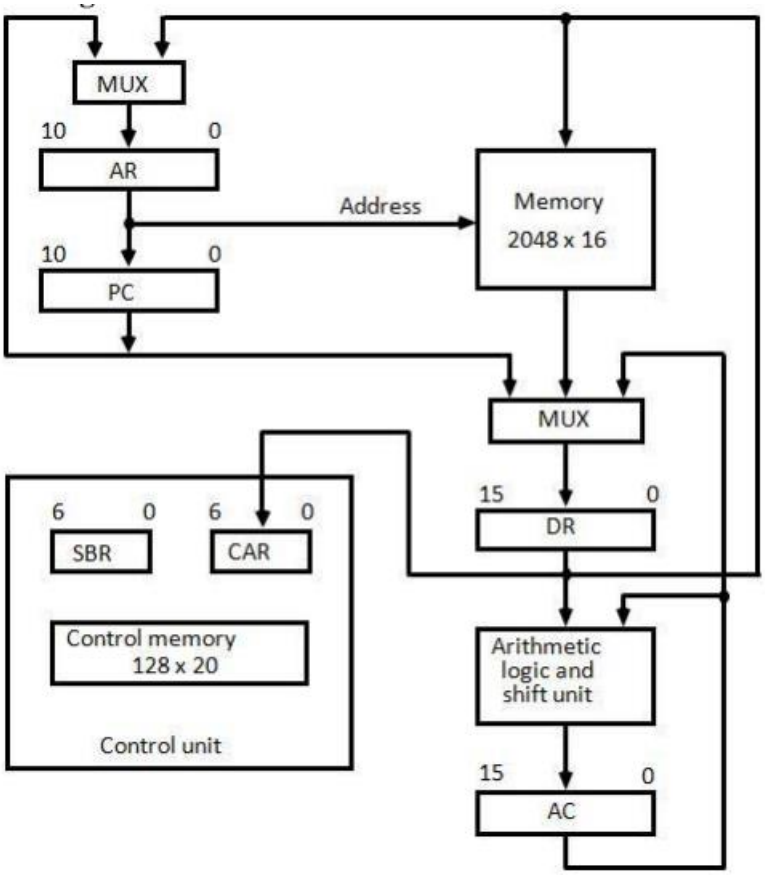


Figure 3.4 Computer Hardware Configuration

1. It consists of two memory units:

   Main memory - for storing instructions and data

   Control memory - for storing the microprogram.
2. Six Registers:

   Processor unit register: AC (accumulator),PC (Program Counter), AR (Address Register), DR (Data Register)

   Control unit register: CAR (Control Address Register), SBR (Subroutine Register)
3. Multiplexers:

   The transfer of information among the registers in the processor is done through multiplexers rather than a common bus.
4. The arithmetic, logic, and shift unit perform microoperations with data from AC and DR and places the result in AC.
5. DR can receive information from AC, PC, or memory.
6. AR can receive information from PC or DR.

7. PC can receive information only from AR.
8. Input data written to memory come from DR, and data read from memory can go only to DR.

Microinstruction Format

The microinstruction format for the control memory is shown in **Figure 3.5**. The 20 bits of the microinstruction are divided into four functional parts as follows:

- The 20 bits of the microinstruction are divided into four functional parts. The three fields Fl, F2, and F3 specify microoperations for the computer.
- The CD field selects status bit conditions.
- The BR field specifies the type of branch to be used.
- The AD field contains a branch address. The address field is seven bits wide, since the control memory has 128 = 2∧7 words.

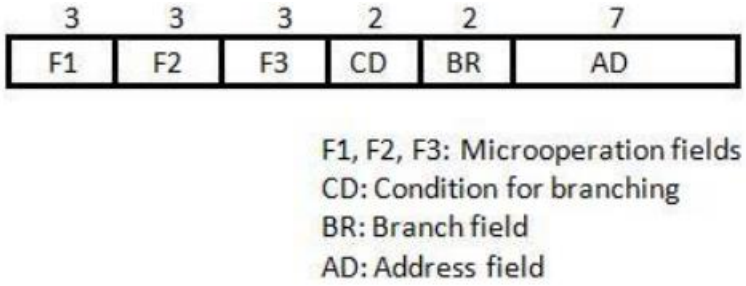


Figure 3.5 - Microinstruction Format

- The microoperations are subdivided into three fields of three bits each The nine bits of the microoperation fields will then be 000 100 101.

- The three bits in each field are encoded to specify seven distinct micro-operations as listed in **Table 3.1**.
- As an example, a microinstruction can specify two simultaneous microoperations from F2

$$DR \leftarrow M[AR] \quad \text{with F2} = 100$$
$$\text{and} \quad PC \leftarrow PC + 1 \quad \text{with F3} = 101$$

and F3 and none from F1.

- All transfer-type microoperations symbols use five letters. The first two letters designate the source register, the third letter is always a T, and the last two lett·ers designate the destination register.

  Example: DRTAC means DR to AC

  - The CD (condition) field consists of 2 bits needed to specify 4 status bit conditions.
  - We will use the symbols U,I,S and Z for the 4 status bits when we write microprograms in symbolic form.

F1	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC + DR$	ADD
010	$AC \leftarrow 0$	CLRAC
011	$AC \leftarrow AC + 1$	INCAC
100	$AC \leftarrow DR$	DRTAC
101	$AR \leftarrow DR(0\text{–}10)$	DRTAR
110	$AR \leftarrow PC$	PCTAR
111	$M[AR] \leftarrow DR$	WRITE

F2	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC - DR$	SUB
010	$AC \leftarrow AC \vee DR$	OR
011	$AC \leftarrow AC \wedge DR$	AND
100	$DR \leftarrow M[AR]$	READ
101	$DR \leftarrow AC$	ACTDR
110	$DR \leftarrow DR + 1$	INCDR
111	$DR(0\text{-}10) \leftarrow PC$	PCTDR

F3	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC \oplus DR$	XOR
010	$AC \leftarrow \overline{AC}$	COM
011	$AC \leftarrow$ shl AC	SHL
100	$AC \leftarrow$ shr AC	SHR
101	$PC \leftarrow PC + 1$	INCPC
110	$PC \leftarrow AR$	ARTPC
111	Reserved	

CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	$DR(15)$	I	Indirect address bit
10	$AC(15)$	S	Sign bit of AC
11	$AC = 0$	Z	Zero value in AC

BR	Symbol	Function
00	JMP	$CAR \leftarrow AD$ if condition = 1 $CAR \leftarrow CAR + 1$ if condition = 0
01	CALL	$CAR \leftarrow AD$, $SBR \leftarrow CAR + 1$ if condition = 1 $CAR \leftarrow CAR + 1$ if condition = 0
10	RET	$CAR \leftarrow SBR$ (Return from subroutine)
11	MAP	$CAR(2\text{–}5) \leftarrow DR(11\text{–}14)$, $CAR(0,1,6) \leftarrow 0$

Table 3.1 - Symbols and Binary Code for Microinstruction Fields

- The BR (branch) field consists of two bits. It is used, in conjunction with the address field AD, to choose the address of the next microinstruction.

Symbolic Microinstruction

- A symbolic microprogram can be translated into its binary equivalent by means of an assembler.
- The Simplest and most straightforward way to formulate an assembly language for a microprogram is to define symbols for each field of the microinstruction and to give users the capability for defining their own symbolic addresses.
- Each line of the assembly language microprogram defines a symbolic microinstruction.
- Each symbolic microinstruction is divided into five fields: label, microoperations, CD, BR, and AD. The fields specify the following **Table 3.2**.

1.	Label	The label field may be empty or it may specify a symbolic address. A label is terminated with a colon (:).
2.	Microoperations	It consists of one, two, or three symbols, separated by commas, from those defined in Table 5.3. There may be no more than one symbol from each F field. The NOP symbol is used when the microinstruction has no microoperations. This will be translated by the assembler to nine zeros.
3.	CD	The CD field has one of the letters U, I, S, or Z.
4.	BR	The BR field contains one of the four symbols defined in Table 5.2.
5.	AD	The AD field specifies a value for the address field of the microinstruction in one of three possible ways: i. With a symbolic address, this must also appear as a label. ii. With the symbol NEXT to designate the next address in sequence. iii. When the BR field contains a RET or MAP symbol, the AD field is left empty and is converted to seven zeros by the assembler.

Table 3.2 - Symbolic Microinstruction

- We will use also the pseudo-instruction ORG to define the origin, or first address, of a microprogram routine.Thus, the symbol ORG 64 informs the assembler to place the next microinstruction in control memory at decimal address 64, which is equivalent to the binary address 1000000.

Fetch Routine

- The control memory has 128 words, and each word contains 20 bits. To microprogram the control memory, it is necessary to determine the bit values of each of the 128 words. The first 64 words (addresses 0 to 63) are to be occupied by the routines for the 16 instructions. The last 64 words may be used for any other purpose.
- The microinstructions needed for the fetch routine are,

$$AR \leftarrow PC$$
$$DR \leftarrow M[AR], \quad PC \leftarrow PC + 1$$
$$AR \leftarrow DR(0\text{–}10), \quad CAR(2\text{–}5) \leftarrow DR(11\text{–}14), \quad CAR(0,1,6) \leftarrow 0$$

- The address part is transferred to AR and then control is transferred to one of 16 routines by mapping the operation code part of the instruction from DR into CAR.
- The fetch routine needs three microinstructions which are placed in control memory at addresses 64, 65, and 66.

```
        ORG 64
FETCH:  PCTAR          U    JMP    NEXT
        READ, INCPC    U    JMP    NEXT
        DRTAR          U    MAP
```

- The translation of the symbolic microprogram to binary produces the following binary microprogram. The bit values are obtained from **Table 3.3.**

Binary Address	F1	F2	F3	CD	BR	AD
1000000	110	000	000	00	00	1000001
1000001	000	100	101	00	00	1000010
1000010	101	000	000	00	11	0000000

Table 3.3 - Symbolic Microprogram to Binary

- The three microinstructions that constitute the fetch routine have been listed in three different representations.
- The symbolic representation is useful for writing microprograms in an assembly language format.
- The binary representation is the actual internal content that must be stored in control memory.

Symbolic Microprogram

- The execution of the third (MAP) microinstruction in the fetch routine results in a branch to address OxxxxOO, where XXXX are the four bits of the operation code.
- The first address for the BRANCH and STORE routines are 0 0001 00 (decimal 4) and 0 0010 00 (decimal 8), respectively. The first adchess for the other 13 routines are at address values 12, 16, 20, ... , 60.
- The indirect address mode is associated with all memory-reference instructions.
- The subroutine, symbolized by INDRCT, is located right after the fetch routine, as shown below.

```
INDRCT:   READ    U   JMP   NEXT
          DRTAR   U   RET
```

- The transfer and return from the indirect subroutine occurs, assume that the MAP microinstruction at the end of the fetch routine caused a branch to address 0, where the ADD routine is stored.
- The first microinstruction in the ADD routine calls subroutine INDRCT, conditioned on status bit I.
- The INDRCT subroutine ha two microinstructions.

```
          ORG 64
FETCH:    PCTAR          U   JMP   NEXT
          READ, INCPC    U   JMP   NEXT
          DRTAR          U   MAP
INDRCT:   READ           U   JMP   NEXT
          DRTAR          U   RET
```

Binary Microprogram

- The symbolic microprogram is useful format for expressing microprograms in a comprehensible manner. However, this is not how the microprogram is kept in memory. If the microprogram is basic enough, as it is in this case, the symbolic code must be translated to binary using either an assembler programme or by the user.
- In the symbolic microprogram address 3 has no equivalent in the symbolic microprogram since the ADD routine has only three microinstructions at addresses 0, 1, and 2.

- The next routine starts at address 4. Even though address 3 is not used, some binary value must be specified for each word in control memory.
- Since the position would never be utilized, we might have defined a word with only zeroes in it. However, it would be good to jump to address 64, which is the start of the fetch function, if an unforeseen problem occurs or if a noise signal sets CAR to the value of 3.

3.1.4 DESIGN OF CONTROL UNIT

- The bits of the microinstruction are usually divided into fields, with each field defining a distinct, separate function.
- The various fields encountered in instruction formats provide:
    - Control bits to initiate microoperations in the system
    - Special bits to specify the way that the next address is to be evaluated
    - An address field for branching
- The number of control bits that initiate microoperations can be reduced by grouping mutually exclusive variables into fields by encoding the k bits in each field to provide 2 k microoperations.
- Each field requires a decoder to produce the corresponding control signals.
    - Reduces the size of the microinstruction bits
    - Requires additional hardware external to the control memory
    - Increases the delay time of the control signals
- **Figure 3.6** shows the three decoders and some of the connections that must be made from their outputs.
- Outputs 5 or 6 of decoder F1 are connected to the load input of AR so that when either one of these outputs is active; information from the multiplexers is transferred to AR.
- The transfer into AR occurs with a clock pulse transition only when output 5 (from DR (0-10) to AR i.e. DRTAR) or output 6 (from PC to AR i.e. PCTAR) of the decoder are active.
- The arithmetic logic shift unit can be designed instead of using gates to generate the control signals; it comes from the outputs of the decoders.

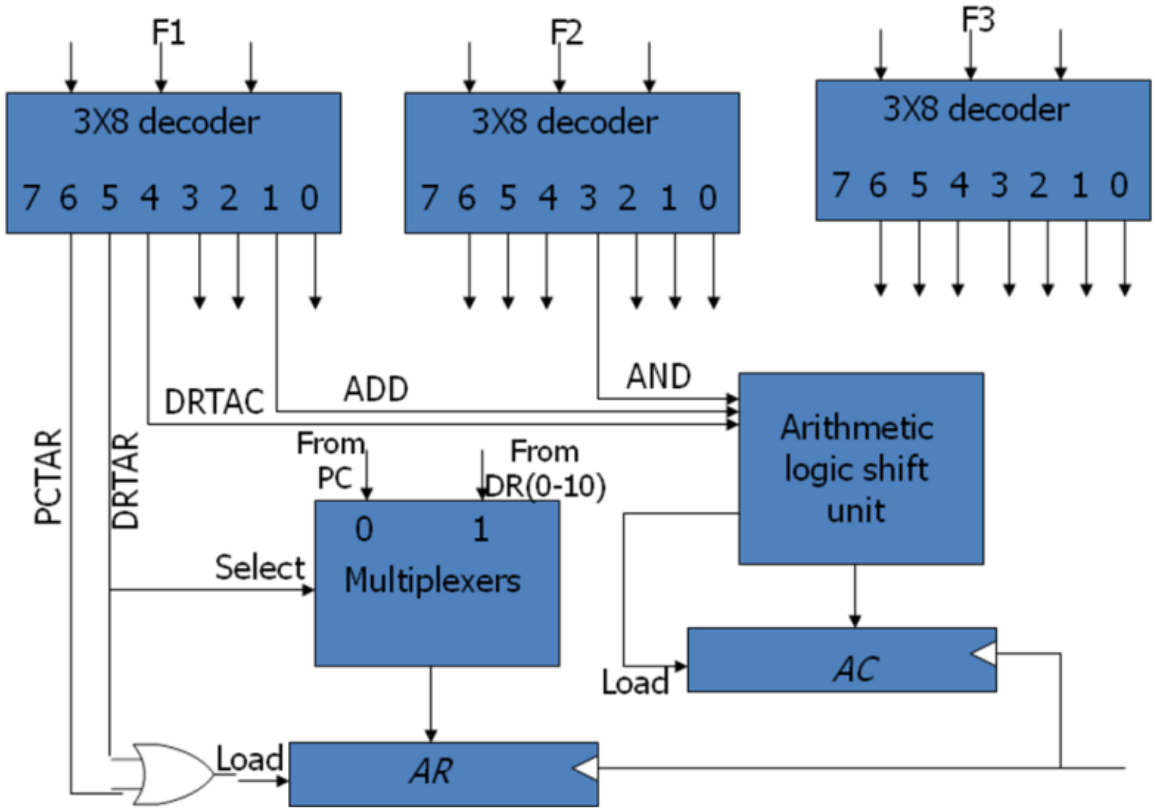


Figure 3.6 - Decoding of Microoperation Fields

Microprogram Sequencer

- The basic components of a microprogrammed control unit are the control memory and the circuits that select the next address.
- The address selection part is called a microprogram sequencer.
- A microprogram sequencer can be constructed with digital functions to suit a particular application
- To guarantee a wide range of acceptability, an integrated circuit sequencer must provide an internal organization that can be adapted to a wide range of application.

- The purpose of a microprogram sequencer is to present an address to the control memory so that a microinstruction may be read and executed.

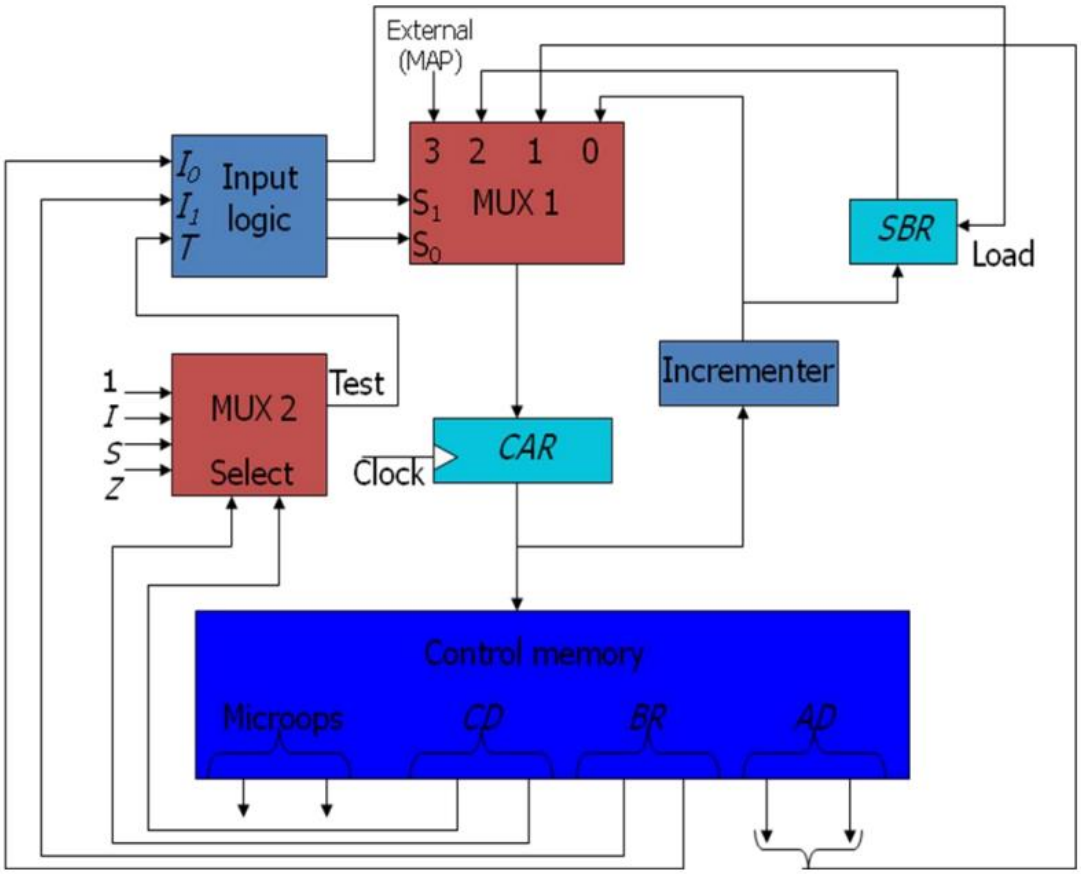


Figure 3.7 - Microprogram Sequencer for a Control Memory

- The block diagram of the microprogram sequencer is shown in **Figure 3.7.**
    - The control memory is included to show the interaction between the sequencer and the attached to it.
    - There are two multiplexers in the circuit; first multiplexer selects an address from one of the four sources and routes to CAR, second multiplexer tests the value of the selected status bit and result is applied to an input logic circuit.
    - The output from CAR provides the address for control memory, contents of CAR incremented and applied to one of the multiplexers inputs and to the SBR.
    - Although the diagram shows a single subroutine register, a typical sequencer will have a register stack about four to eight levels deep. In this way, a push,

pop operation and stack pointer operate for subroutine call and return instructions.

- The CD (Condition) field of the microinstruction selects one of the status bits in the second multiplexer.
- The Test variable (either 1 or 0) i.e. T value together with the two bits from the BR (Branch) field go to an input logic circuit.
- The input logic circuit determines the type of the operation.

Design of Input Logic

- The input logic in a particular sequencer will determine the type of operations that are available in the unit.
- Typical sequencer operations are: increment, branch or jump, call and return from subroutine, load an external address, push or pop the stack, and other address sequencing operations.
- Based on the function listed in each entry was defined in **Table 3.1**, the truth table for the input logic circuit is shown in **Table 3.4**.
- Therefore, the simplified Boolean functions for the input logic circuit can be given as:

$$S_1 = I_1$$
$$S_0 = I_1 I_0 + I_1' T$$
$$L = I_1' I_0 T$$

BR Field	Input I_1	I_0	T	MUX 1 S_1	S_0	Load *SBR* L
0	0	0	0	0	0	0
0	0	0	1	0	1	0
0	1	0	1	0	0	0
0	1	0	1	0	1	1
1	0	1	0	1	0	0
1	1	1	X	1	1	0

Table 3.4 - Input Logic Truth Table for Microprogram Sequencer

- The bit values for S1 and S0 are determined from the stated function and the path in the multiplexer that establishes the required transfer.
- Note that the incrementer circuit in the sequencer of **Figure 3.7** is not a counter constructed with flip-flops but rather a combinational circuit constructed with gates.

3.2 CENTRAL PROCESSING UNIT

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 **Figure 3.8.** Its purpose is to interpret instruction cycles received from memory and perform arithmetic, logic and control operations with data stored in internal register, memory words and I/O interface units. A CPU is usually divided into two parts namely processor unit (Register Unit and Arithmetic Logic Unit) and control unit.

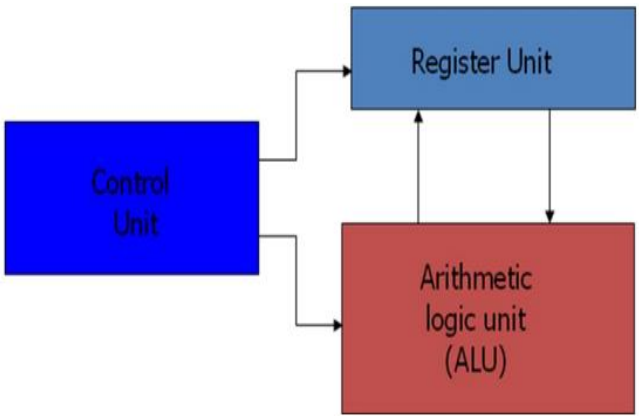


Figure 3.8 - Major Components of CPU

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.

3.2.1 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 micro-operations. Hence it is necessary to provide a common unit that can perform all the arithmetic, logic and shift microoperation in the processor.

The need for memory locations arises for storing pointers, counters, return address, temporary results and partial products. Memory access consumes the most of the time off an operation in a computer. It is more convenient and more efficient to store these intermediate values in processor registers.

A common bus system is employed to contact registers that are included in the CPU in a large number. Communications between registers is not only for direct data transfer but also for performing various micro-operations. A bus organization for such CPU register shown in **Figure 3.9**, is connected to two multiplexers (MUX) to form two buses A and B. The selected lines in each multiplexers select one register of the input data for the particular bus.

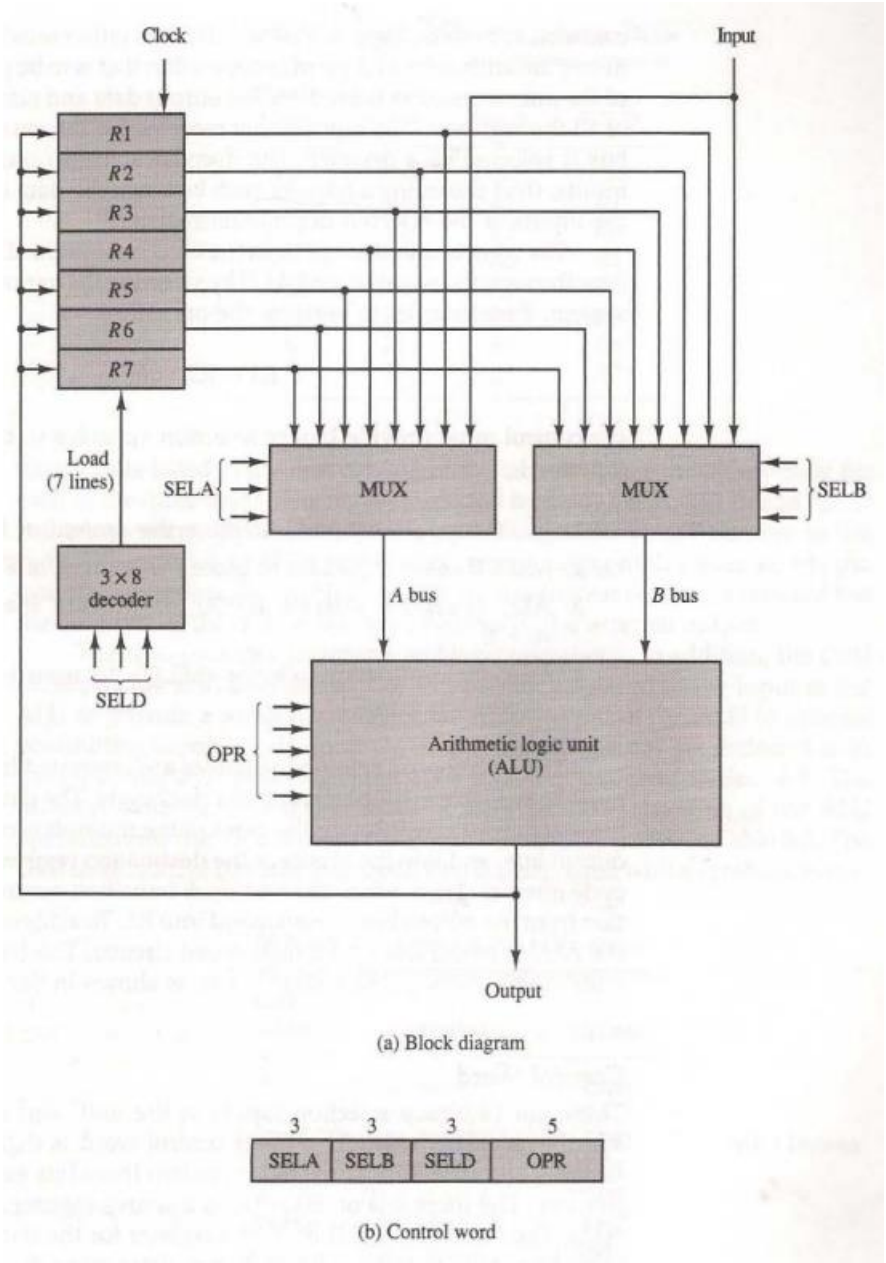


Figure 3.9 - Register Set with Common ALU

Operation of Control Unit:

The control unit directs the information flow through ALU by: -

- Selecting various Components in the system
- Selecting the Function of ALU

Example: R1 <- R2 + R3

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
4. Decoder destination selector (SELD): to transfer the content of the output bus into R1

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 **Figure 3.9(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 of 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 3.5.**

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 3.5 - Encoding of Register Selection Fields

ALU

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.

The function table for this ALU is listed in **Table 3.6**.

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

Table 3.6 - Encoding of ALU Operations

Examples of Microoperations

The subtract microoperation given by the statement 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. Thus, the control word is specified by the four fields and the corresponding

$$R1 \leftarrow R2 - R3$$

binary value for each field.

The binary control word for the subtract microoperation is 010 011 001 00101 and is obtained as follows:

The control word for this microoperation and a few others are listed in **Table 3.7**. The increment and transfer microoperations do not use the B input of the ALU.

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

Microoperation	Symbolic Designation SELA	SELB	SELD	OPR	Control Word
$R1 \leftarrow R2 - R3$	R2	R3	R1	SUB	010 011 001 00101
$R4 \leftarrow R4 \vee R5$	R4	R5	R4	OR	100 101 100 01010
$R6 \leftarrow R6 + 1$	R6	—	R6	INCA	110 000 110 00001
$R7 \leftarrow R1$	R1	—	R7	TSFA	001 000 111 00000
Output $\leftarrow R2$	R2	—	None	TSFA	010 000 000 00000
Output $\leftarrow$ Input	Input	—	None	TSFA	000 000 000 00000
$R4 \leftarrow$ sh1 $R4$	R4	—	R4	SHLA	100 000 100 11000
$R5 \leftarrow 0$	R5	R5	R5	XOR	101 101 101 01100

Table 3.7 - Examples of Microoperations for the CPU

3.2.2 Instruction Formats

- 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 specifies the operation to be perform
  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
- Computers may have instructions of several different lengths containing varying number of addresses.
- The number of address fields in the instruct 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.

Single Accumulator Organization

- In an accumulator type organization, all the 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 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.

Stack Organization

- The stack-organized CPU has 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 stack pointer is updated automatically.
- Operation-type instructions do not need an address field in stack-organized computers.
- This is because the operation is performed on the two items that are on 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 effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack.
- There is no need to specify operands with an address field since all operands are implied to be in the stack.
- Most computers fall into one of the three types of organizations.
- Some computers combine features from more than one organizational structure.
- The influence of the number of addresses on computer programs, we will evaluate the arithmetic statement

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

- Using zero, one, two, or three address instructions and using the symbols ADD, SUB, MUL and DIV for four arithmetic operations; MOV for the transfer type operations; and LOAD and STORE for transfer to and from memory and AC register.
- Assuming that the operands are in memory addresses A, B, C, and D and the result must be stored in memory ar address X and also the CPU has general purpose registers R1, R2, R3 and R4.

Three Address Instructions

- Three-address instruction formats can use each address field to specify either a processor register or a memory operand.
- The program assembly language that evaluates **X = (A+B) * (C+D)** is shown below, together with comments that explain the register transfer 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
```

- The symbol M [A] denotes the operand at memory address symbolized by A.
- The advantage of the three-address format is that it results 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 formats use each address field can specify either a processor register or 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     M[X]←R1
```

- The MOV instruction moves or transfers the operands to and from memory and processor registers.
- The first symbol listed in an instruction is assumed be both a source and the destination where the result of the operation transferred.

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. But for the basic discussion 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    AC ← M[A]
ADD     B    AC ← AC + M[B]
STORE   T    M[T] ← AC
LOAD    C    AC ← M[C]
ADD     D    AC ← AC + M[D]
MUL     T    AC ← AC * M[T]
STORE   X    M[X] ← AC
```

- All operations are done between the AC register and a memory operand.
- T is the address of a temporary memory location required for storing the intermediate result.

Zero Address Instructions

- A stack-organized computer does not use an address field for the instructions ADD and MUL.
- The PUSH and POP instructions, however, 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
```

- To evaluate arithmetic expressions in a stack computer, it is necessary to convert the expression into reverse Polish notation.
- The name "zero-address" is given to this type of computer because of the absence of an address field in the computational instructions

RISC Instructions

- The instruction set of a typical RISC processor is use only load and store instructions for communicating between memory and CPU.
- All other instructions are executed within the registers of CPU without referring to memory.
- LOAD and STORE instructions that have 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)The load instructions transfer the operands from memory to CPU register.

```
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, R2     R3 ← R3 + R4
MUL     R1, R1, R3     R1 ← R1*R3
STORE   X, R1          M[X] ← R1
```

- The add and multiply operations are executed with data in the register without accessing memory.
- The result of the computations is then stored memory with a store in instruction.

3.2.3 ADDRESSING MODES

- The way the operands are chosen during program execution is dependent on the addressing mode of the instruction.
- Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:
    - 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. o
    - To reduce the number of bits in the addressing field of the instruction.
- Most addressing modes modify the address field of the instruction; there are two modes that need no address field at all. These are 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 because the operand in the accumulator register is implied in the definition of the instruction.
- All register reference instructions that use an accumulator are implied mode instructions.
- Zero address in a stack organization computer is implied mode instructions.

Immediate Mode

- In this mode the operand is specified in the instruction itself.
- In other words, an immediate-mode instruction has an operand rather than an address field.
- Immediate-mode instructions are useful for initializing registers to a constant value.
- The address field of an instruction may specify either a memory word or a processor register.
- When the address specifies a processor register, the instruction is said to be in the register mode.

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 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 few bits to select a register than would have been required to specify a memory address directly.

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.
- The address field of an instruction is used by the control unit in the CPU to obtain the operand from memory.
- Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated.
- The basic two mode of addressing used in CPU are direct and indirect address mode.

Direct Address Mode:

- In this mode the effective address is equal to the address part of the instruction.
- The operand resides in memory and its address is given directly by the address field of the instruction.
- In a branch-type instruction the address field specifies the actual branch address.

Indirect Address Mode:

- In this mode the address field of the instruction gives the address where the effective address is stored in memory.
- Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.
- A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU.
- The effective address in these modes is obtained from the following computation: Effective address =address part of instruction + content of CPU register.
- The CPU register used in the computation may be the program counter, an index register, or a base register.
- We have a different addressing mode which is used for a different application.

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.
- An index register is a special CPU register that contains an index value.

Base 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.
- This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register.

Numerical Example:

To show the differences between the various modes, we will show the effect of the addressing modes on the instruction defined in **Figure 3.10.**

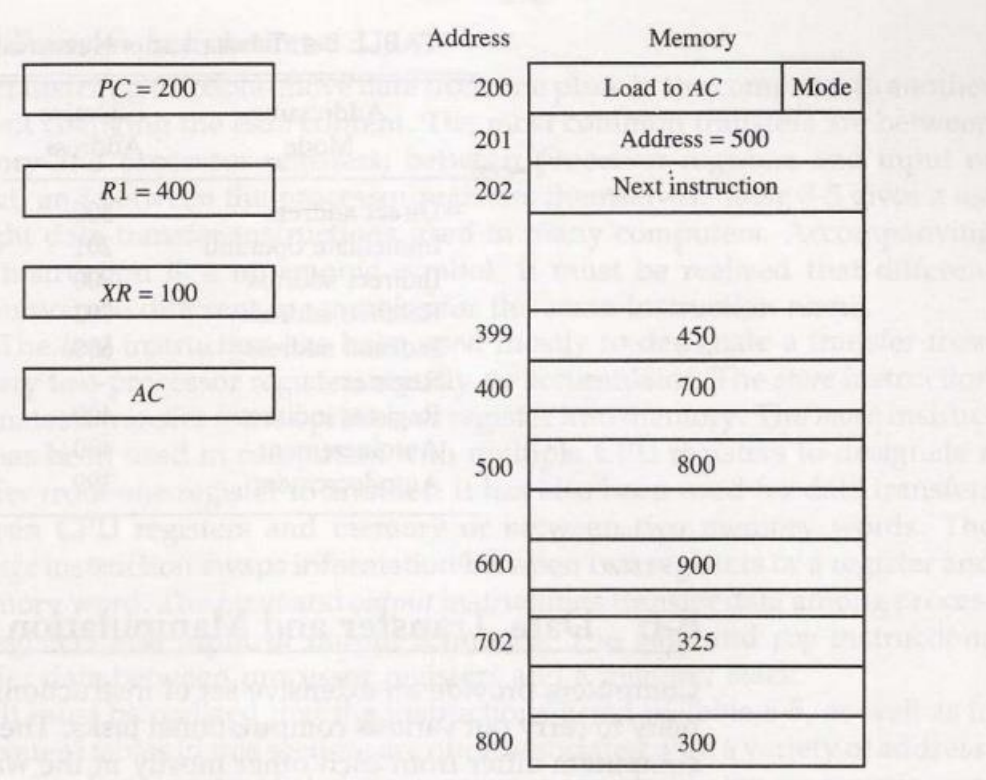


Figure 3.10 - Numerical Example for Addressing Modes

- 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 specifies 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.
- 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 500.
- 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.
- 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 the effective address is 500 + 202 =702 and the operand is 325. (The value in PC after the fetch phase and during the execute phase is 202.)

- 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 400 is 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.
- **Table 3.8** lists the values of the effective address and the operand loaded into AC for the nine addressing modes.

Addressing Mode	Effective Address	Content of *AC*
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

Table 3.8 - Tabular List of Numerical Example

3.2.4 DATA TRANSFER AND MANIPULATION

Most computer instructions can be classified into three categories:

1. Data transfer instructions
2. Data manipulation instructions
3. Program control instructions

Data Transfer Instructions

- Data transfer instructions move data from one place in the computer to another without changing the data content.

- The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves.
- **Table 3.9** gives a list of eight data transfer instructions used in many computers.
- The load instruction has been used mostly to designate a transfer from memory to a processor register, usually an accumulator.
- The store instruction designates a transfer from a processor register into memory.

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

Table 3.9 - Typical Data Transfer Instructions

- The move instruction has been used in computers with multiple CPU registers to designate a transfer from one register to another and also between CPU registers and memory or between two memory words.
- The exchange instruction swaps information between two registers or a register and a memory word.
- The input and output instructions transfer data among processor registers and input or output terminals.
- The push and pop instructions transfer data between processor registers and a memory stack.
- Different computers use different mnemonics symbols for differentiate the addressing modes.
- As an example, consider the load to accumulator instruction when used with eight different addressing modes.

- **Table 3.10** shows the recommended assembly language convention and actual transfer accomplished in each case

Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	$AC \leftarrow M[ADR]$
Indirect address	LD @ADR	$AC \leftarrow M[M[ADR]]$
Relative address	LD $ADR	$AC \leftarrow M[PC + ADR]$
Immediate operand	LD #NBR	$AC \leftarrow NBR$
Index addressing	LD ADR(X)	$AC \leftarrow M[ADR + XR]$
Register	LD R1	$AC \leftarrow R1$
Register indirect	LD (R1)	$AC \leftarrow M[R1]$
Autoincrement	LD (R1)+	$AC \leftarrow M[R1], R1 \leftarrow R1 + 1$

Table 3.10 - Eight Addressing Modes for the Load Instruction

- **ADR** stands for an address.
- **NBA** a number or operand.
- **X** is an index register.
- **AC** is the accumulator register.
- The **@** character symbolizes an indirect addressing.
- The **$** character before an address makes the address relative to the program counter PC.
- The **#** character precedes the operand in an immediate-mode instruction.
- An indexed mode instruction is recognized by a register that placed in parentheses after the symbolic address.
- The register mode is symbolized by giving the name of a processor register.
- In the register indirect mode, the name of the register that holds the memory address is enclosed in parentheses.
- The auto-increment mode is distinguished from the register indirect mode by placing a plus after the parenthesized register. The auto-decrement mode would use a minus instead.

Data Manipulation Instructions

- Data manipulation instructions perform operations on data and provide the computational capabilities for the computer.

- The data manipulation instructions in a typical computer are usually divided into three basic types:
    1. Arithmetic instructions
    2. Logical and bit manipulation instructions
    3. Shift instructions

1. Arithmetic Instructions

- The four basic arithmetic operations are addition, subtraction, multiplication and division.
- Most computers provide instructions for all four operations.
- Some small computers have only addition and possibly subtraction instructions. The multiplication and division must then be generated by mean software subroutines.
- A list of typical arithmetic instructions is given in **Table 3.11**.

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate (2's complement)	NEG

Table 3.11 - Typical Arithmetic Instructions

- The increment instruction adds 1 to the value stored in a register or memory word.
- A number with all 1's, when incremented, produces a number with all 0's.
- The decrement instruction subtracts 1 from a value stored in a register or memory word.
- A number with all 0's, when decremented, produces number with all 1's.
- The add, subtract, multiply, and divide instructions may be use different types of data.
- The data type assumed to be in processor register during the execution of these arithmetic operations is defined by an operation code.
- An arithmetic instruction may specify fixed-point or floating-point data, binary or decimal data, single-precision or double-precision data.

- The mnemonics for three add instructions that specify different data types are shown below. ADDI Add two binary integer numbers ADDF Add two floating-point numbers ADDD Add two decimal numbers in BCD
- A special carry flip-flop is used to store the carry from an operation.
- The instruction "add carry" performs the addition on two operands plus the value of the carry the previous computation.
- Similarly, the "subtract with borrow" instruction subtracts two words and borrow which may have resulted from a previous subtract operation.
- The negate instruction forms the 2's complement number, effectively reversing the sign of an integer when represented it signed-2's complement form.

2. Logical and Bit Manipulation Instructions

- Logical instructions perform binary operations on strings of bits store, registers.
- They are useful for manipulating individual bits or a group of that represent binary-coded information.

- The logical instructions consider each bit of the operand separately and treat it as a Boolean variable.
- By proper application of the logical instructions, it is possible to change bit values, to clear a group of bits, or to insert new bit values into operands stored in register memory words.
- Some typical logical and bit manipulation instructions are listed in **Table 3.12**.
- The clear instruction causes the specified operand to be replaced by 0’s.
- The complement instruction produces the 1's complement by inverting all bits of the operand.
- The AND, OR, and XOR instructions produce the corresponding logical operations on individual bits of the operands.
- The logical instructions can also be used to performing bit manipulation operations.

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

Table 3.12 - Typical Logical and Bit Manipulation Instructions

- There are three-bit manipulation operations possible: a selected bit can clear to 0, or can be set to 1, or can be complemented.
    - o The AND instruction is used to clear a bit or a selected group of bits of an operand.
    - o The OR instruction is used to set a bit or a selected group of bits of an operand.
    - o Similarly, the XOR instruction is used to selectively complement bits of an operand.
- Other bit manipulations instructions are included in above table perform the operations on individual bits such as a carry can be cleared, set, or complemented.
- Another example is a flip-flop that controls the interrupt facility and is either enabled or disabled by means of bit manipulation instructions.

3. Shift Instructions

- Shifts are operations in which the bits of a word are moved to the left or right.
- The bit shifted in at the end of the word determines the type of shift used.
- Shift instructions may specify logical shifts, arithmetic shifts, or rotate-type operations.
- In either case the shift may be to the right or to the left.
- **Table 3.13** lists four types of shift instructions.

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

Table 3.13 - Typical Shift Instructions

- The logical shift inset to the end bit position.
- The end position is the leftmost bit position for shift rights the rightmost bit position for the shift left.
- Arithmetic shifts usually conform to the rules for signed-2's complement numbers.
- The arithmetic shift-right instruction must preserve the sign bit in the leftmost position.
- The sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unchanged.
- This is a shift-right operation with the end bit remaining the same.
- The arithmetic shift-left instruction inserts 0 to the end position and is identical to the logical shift-instruction.
- The rotate instructions produce a circular shift. Bits shifted out at one of the words are not lost as in a logical shift but are circulated back into the other end.
- The rotate through carry instruction treats a carry bit as an extension of the register whose word is being rotated.
- Thus, a rotate-left through carry instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit position into the carry, and at the same time, shift the entire register to the left.

3.2.5 PROGRAM CONTROL

- Program control instructions specify conditions for altering the content of the program counter.

- The change in value of the program counter as a result of the execution of a program control instruction causes a break in the sequence of instruction execution.
- This instruction provides control over the flow of program execution and a capability for branching to different program segments.
- Some typical program control instructions are listed in **Table 3.14**. Branch and jump instructions may be conditional or unconditional.

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TST

Table 3.14 - Typical Program Control Instructions

- An unconditional branch instruction causes a branch to the specified address without any conditions.
- The conditional branch instruction specifies a condition such as branch if positive or branch if zero.
- The skip instruction does not need an address field and is therefore a zero-address instruction.
- A conditional skip instruction will skip the next instruction if the condition is met. This is accomplished by incrementing program counter.
- The call and return instructions are used in conjunction with subroutines.
- The compare instruction forms a subtraction between two operands, but the result of the operation not retained. However, certain status bit conditions are set as a result of operation.
- Similarly, the test instruction performs the logical AND of two operands and updates certain status bits without retaining the result or changing the operands.

Subroutine Call And Return

- A subroutine is self-contained sequence of instructions that performs a given computational task.
- The most common names used are call subroutine, jump to subroutine, branch to subroutine, or branch and save return address.
- A subroutine is executed by performing two operations

  1. The address of the next instruction available in the program counter (the return address) is stored in a temporary location so the subroutine knows where to return
  2. Control is transferred to the beginning of the subroutine.
- The last instruction of every subroutine, commonly called return from subroutine, transfers the return address from the temporary location in the program counter.
- Different computers use a different temporary location for storing the return address.
- The most efficient way is to store the return address in a memory stack.
- The advantage of using a stack for the return address is that when a succession of subroutines is called, the sequential return addresses can be pushed into the stack.
- A subroutine call is implemented with the following microoperations:

$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PC$	Push content of PC onto the stack
$PC \leftarrow$ effective address	Transfer control to the subroutine

- The instruction that returns from the last subroutine is implemented by the microoperations:

$PC \leftarrow M[SP]$	Pop stack and transfer to PC
$SP \leftarrow SP + 1$	Increment stack pointer

MODULE 4
REGISTER TRANSFER LANGUAGE AND MICRO OPERATIONS

4.1.1 REGISTER TRANSFER LANGUAGE

- Digital systems are composed of modules that are constructed from digital components, such as registers, decoders, arithmetic elements, and control logic.
- The modules are interconnected with common data and control paths to form a digital computer system.
- 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.
- Examples are shift, count, clear, and load.
- Some of the digital components from before are registers that implement microoperations.
- The internal hardware organization of a digital computer is best defined by specifying.
    - The set of registers it contains and their functions.
    - The sequence of microoperations performed on the binary information stored.
    - The control that initiates the sequence of microoperations.
- Use symbols, rather than words, to specify the sequence of microoperations.
- The symbolic notation used is called a register transfer language.
- A programming language is a procedure for writing symbols to specify a given computational process.
- Define symbols for various types of microoperations and describe associated hardware that can implement the microoperations.

Registers

- Designate computer registers by capital letters to denote its function.
- The register that holds an address for the memory unit is called MAR.
- The program counter register is called PC.
- IR is the instruction register and R1 is a processor register.

- The individual flip-flops in an n-bit register are numbered in sequence from 0 to n-1.
- Figure 2-1 shows the representation of registers in block diagram form.

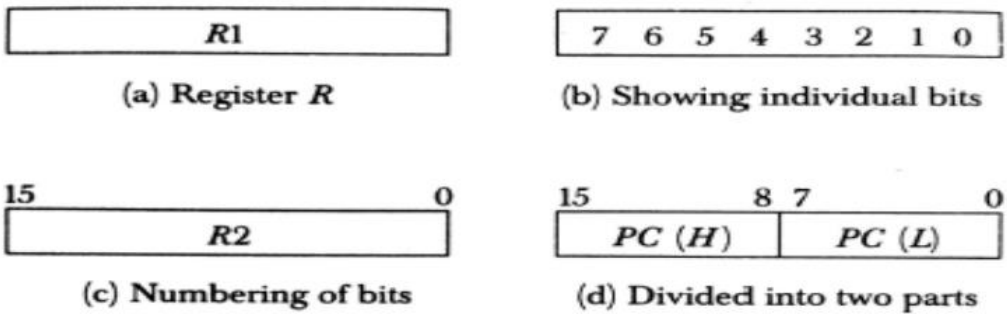


Figure 4.1 - Block Diagram of Register

- The most common way to represent a register is by a rectangular box with the name of the register inside, as in **Figure 4.1(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).**
- 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.

4.1.2 REGISTER TRANSFER

- Information transfer from one register to another is designated in symbolic form by means of a replacement operator.
- The statement R2← 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 R 1 does not change after the transfer.
- If we want the transfer to occur only under a predetermined control condition then it can be shown by an if-then statement.

If (P=1) then R2← R1

- P is the control signal generated by a control section.
- Every statement written in a register transfer notation implies a hardware construction for implementing the transfer.
- **Figure 4.2** shows the block diagram that depicts the transfer from R1 to R2.

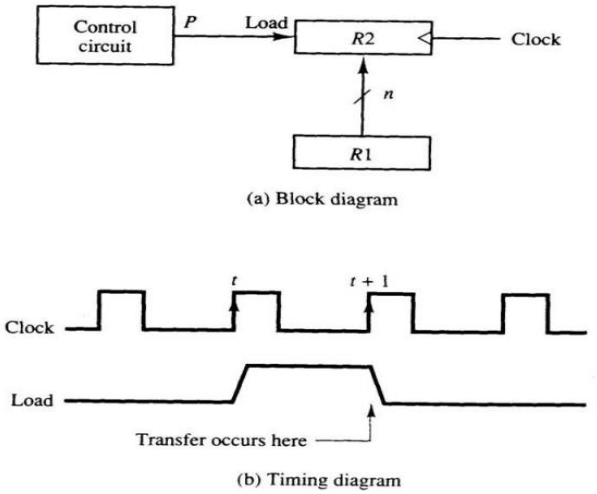


Figure 4.2 - Transfer from R1 to R2 When P=1

- The n outputs of register R1 are connected to the n inputs of register 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 P.
- It is assumed that the control variable is synchronized with the same clock as the one applied to the register.
- As shown in the timing diagram, P is activated in the control section by the rising 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.
- 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 time t +1.

- The basic symbols of the register transfer notation are listed in below Table 2-1.

Symbol	Description	Examples
Letters(and numerals)	Denotes a register	MAR, R2
Parentheses ( )	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

Table 4.1 - Basic Symbols of Register Transfers

- A comma is used to separate two or more operations that are executed at the same time.
- All microoperations written on a single line are to be executed at the same time

T: R2 ← R1, R1 ←R2

4.1.3 BUS AND MEMORY TRANSFERS

- A more efficient scheme for transferring information between registers in a **multiple-register** configuration is a **Common Bus System**.
- A common bus consists of a set of common lines, one for each bit of a register.
- Control signals determine which register is selected by the bus during each particular register transfer.
- Different ways of constructing a Common Bus System.
    - Using Multiplexers
    - Using Tri-state Buffers

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 registers is shown in below **Figure 4.3**.

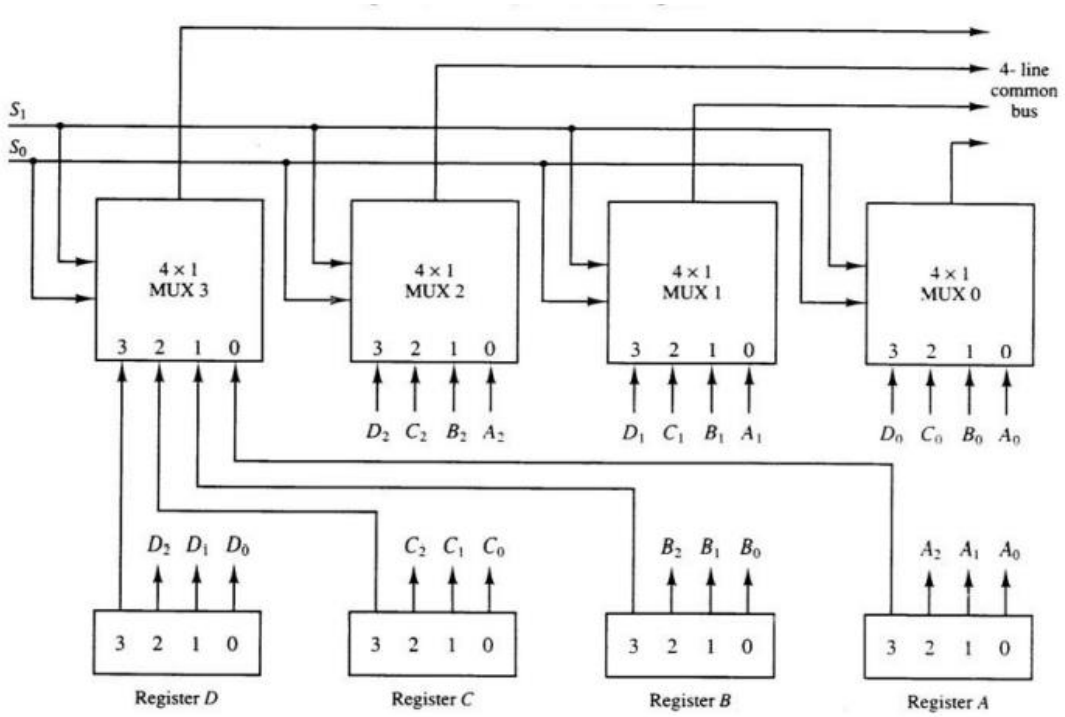


Figure 4.3 - Bus System for Four Registers

- 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 is connected to input 0 of MUX 1 because this input is labelled A1.
- 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 registers, MUX 1 multiplexes the four 1 bits of the registers, and similarly for the other two bits.
- The two selection lines Si and So 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 S1S0 = 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 S1S0 = 01, and so on.

- **Table 4.2** shows the register that is selected by the bus for each of the four possible binary value of the selection lines.

S1	S0	Register Selected
0	0	A
0	1	B
1	0	C
1	1	D

Table 4.2 - Function Table for Bus Fig 4.3

- In general, a bus system has
    - Multiplex "k" Registers
    - each register of "n" bits
    - each register of "n" bits
    - no. of multiplexers required = n
    - no. of multiplexers required = n
- When the bus is includes 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

Three-State Bus Buffers

- Instead of using multiplexers, three-state gates can be used to construct the bus system.
- A three-state gate is a digital circuit that exhibits three states.
- Two of the states are signals equivalent to logic 1 and 0.
- The third state is a **high-impedance state**- this behaves like an open circuit, which means the output is disconnected and does not have a logic significance.
- The graphic symbol of a three-state buffer gate is shown in **Figure 2-3**.

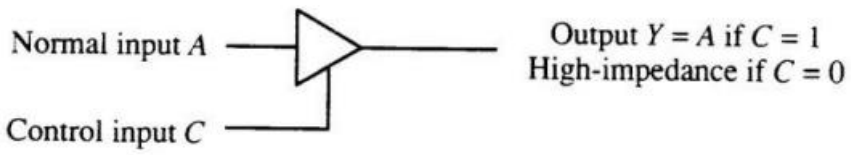


Figure 4.4 - Graphic Symbol of Three-State Buffer

- The three-state buffer gate has a normal input and a control input which determines the output state.
- With control 1, the output equals the normal input.
- With control 0, the gate goes to a high-impedance state.
- This enables a large number of three-state gate outputs to be connected with wires to form a common bus line without endangering loading effects.
- The construction of a bus system with three-state buffers is shown in **Figure 4.5.**

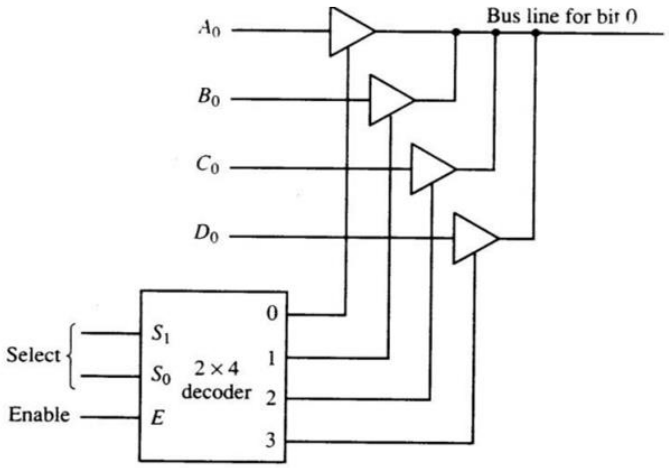


Figure 4.5 - Bus Line with Three-State Buffer

- The outputs of four buffers are connected together to form a single bus line.
- The control inputs to the buffers determine which of the four normal inputs will communicate with the bus line.
- No more than one buffer may be in the active state at any given time. The connected buffers must be controlled so that only one three-state buffer has access to the bus line while all other buffers are maintained in a high impedance state.
- One way to ensure that no more than one control input is active at any given time is to use a decoder, as shown in the diagram.
- When the enable input of the decoder is 0, all of its four outputs are 0, and the bus line is in a high-impedance state because all four buffers are disabled.
- When the enable input is active, one of the three-state buffers will be active, depending on the binary value in the select inputs of the decoder.

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**.
- It is necessary to specify the address of M when writing memory transfer operations.
- Designate the address register by **AR** and the data register by **DR**.
- The read operation can be stated as follows:

DR ← M[AR]

- The write operation can be stated as:

Write: M[AR] ← R1

TYPES OF MICRO-OPERATIONS

- **Register Transfer Micro-operations**: Transfer binary information from one register to another.
- **Arithmetic Micro-operations**: Perform arithmetic operation on numeric data stored in registers.
- **Logical Micro-operations**: Perform bit manipulation operations on data stored in registers.
- **Shift Micro-operations**: Perform shift operations on data stored in registers.

Register Transfer Micro-operation doesn't change the information content when the binary information moves from source register to destination register. Other three types of micro-operations change the information change the information content during the transfer.

4.1.4 ARITHMETIC MICRO-OPERATIONS

- The basic arithmetic microoperations are addition, subtraction, increment, decrement, and shift.
- Example of addition: R3 ← R1+R2.
- Subtraction is most often implemented through complementation and addition.

- Example of subtraction: R3 ← R1 + $\overline{R2}$+ 1 (strikethrough denotes bar on top – 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 subtracting.

Binary Adder

- A binary adder is a digital circuit that generates the arithmetic sum of two binary numbers of any length.
- A binary added is constructed with full-adder circuits connected in cascade.
- An n-bit binary adder requires n full-adders.
- **Figure 4.6** shows the interconnections of four full-adders (FA) to provide a 4-bit binary adder.

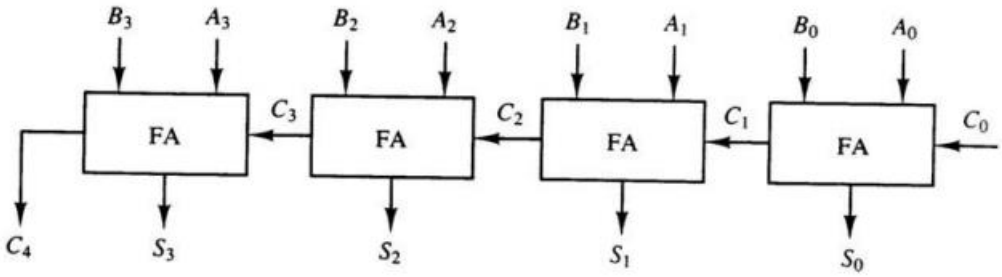


Figure 4.6 - 4-Bit Binary Adder

- The augends 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 Co 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.

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 **Figure 4.7**.

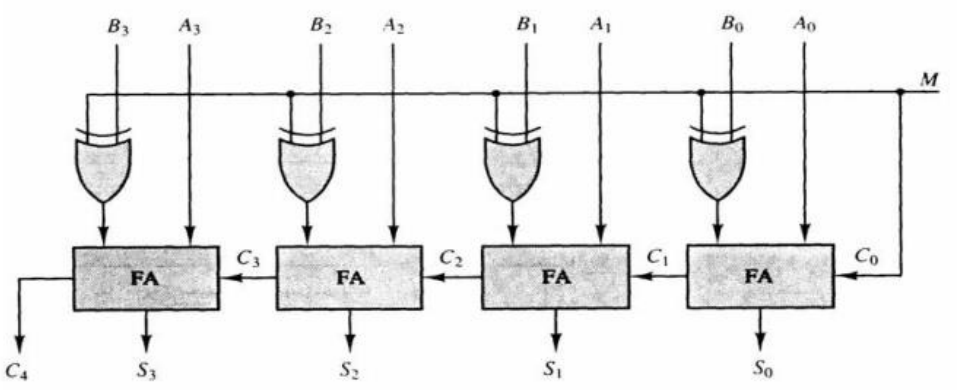


Figure 4.7 - 4-Bit Adder-Subtractor

- The mode input M controls the operation. When M = 0 the circuit is an 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 XOR 0 = B. The full-adders receive the value of B, the input carry is 0, and the circuit performs A plus B.
- When M = 1, we have B xor 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 2's complement of B.

Binary Incrementer

- The increment microoperation adds one to a number in a register.
- For example, if a 4-bit register has a binary value 0110, it will go to 0111 after it is incremented.
- This can be implemented by using a binary counter – every time the count enable is active, the count is incremented by one.
- If the increment is to be performed independent of a particular register, then use half-adders connected in cascade.
- An n-bit binary incrementor requires n half-adders.

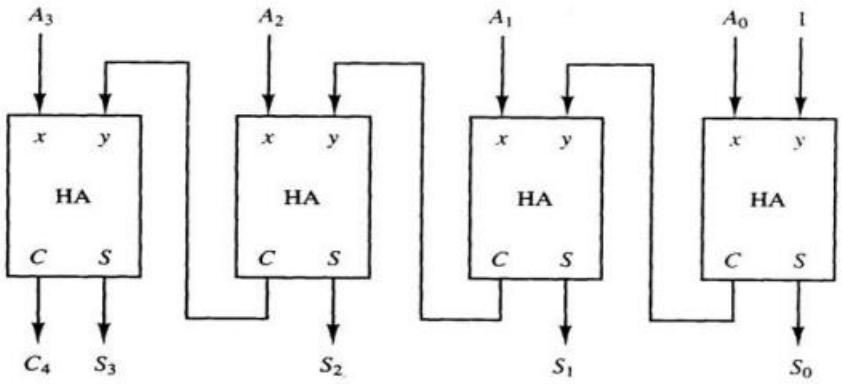


Figure 4.8 - 4-Bit Binary Incrementer

- One of the inputs to the least significant half-adder (HA) is connected to logic-1 and the other input is connected to the least significant bit of the number to be incremented.
- The output carry from one half-adder is connected to one of the inputs of the next-higher-order half-adder.
- The circuit receives the four bits from A0 through A3, adds one to it, and generates the incremented output in S0 through S3.
- The output carry C4 will be 1 only after incrementing binary 1111. This also causes outputs S0 through S3 to go to 0.
- The circuit of Fig. 2-8 can be extended to an n -bit binary incrementer by extending the diagram to include n half-adders.
- The least significant bit must have one input connected to logic-1. The other inputs receive the number to be incremented or the carry from the previous stage.

Arithmetic Circuit

- The basic component of an arithmetic circuit is the parallel adder.
- By controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations.
- The diagram of a 4-bit arithmetic circuit is shown in **Figure 4.9**. It has four full-adder circuits that constitute the 4-bit adder and four multiplexers for choosing different operations.

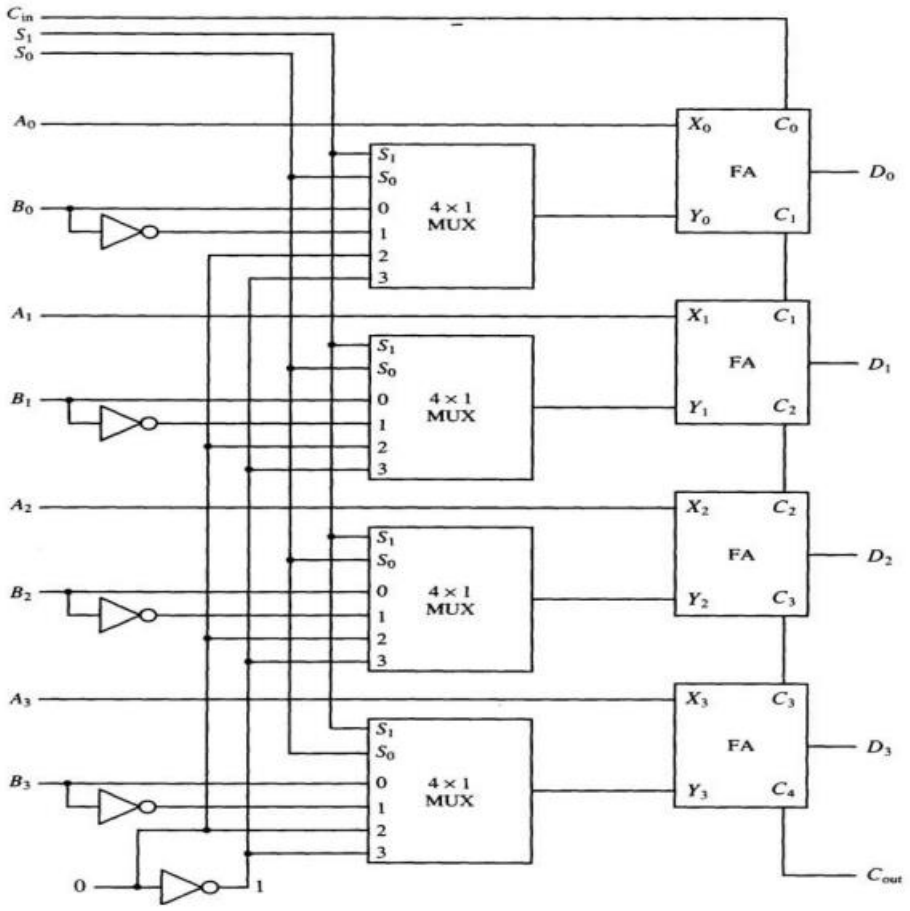


Figure 4.9 - Arithmetic Circuit

- There are two 4-bit inputs A and B and a 4-bit output D.
- The four inputs from A go directly to the X inputs of the binary adder.
- Each of the four inputs from B are connected to the data inputs of the multiplexers.
- The multiplexers data inputs also receive the complement of B.
- The other two data inputs are connected to logic-0 and logic-1.
- The four multiplexers are controlled by two selection inputs S1 and S0. The input carry Cin, goes to the carry input of the FA in the least significant position. The other carries are connected from one stage to the next.
- By controlling the value of Y with the two selection inputs S1 and S0 and making Cin equal to 0 or 1, it is possible to generate the eight arithmetic microoperations listed in **Table 4.3**.

Select			Input	Output	
S_1	S_0	C_{in}	Y	$D = A + Y + C_{in}$	Microoperation
0	0	0	B	$D = A + B$	Add
0	0	1	B	$D = A + B + 1$	Add with carry
0	1	0	$\overline{B}$	$D = A + \overline{B}$	Subtract with borrow
0	1	1	$\overline{B}$	$D = A + \overline{B} + 1$	Subtract
1	0	0	0	$D = A$	Transfer A
1	0	1	0	$D = A + 1$	Increment A
1	1	0	1	$D = A - 1$	Decrement A
1	1	1	1	$D = A$	Transfer A

Table 4.3 - Arithmetic Circuit Function Table

Addition:

- When S1S0= 00, the value of B is applied to the Y inputs of the adder.
    - If Cir, = 0, the output D =A+B.
    - If Cin = 1, output D=A+B + 1.
- Both cases perform the add microoperation with or without adding the input carry.

Subtraction:

- When S1S0 = 01, the complement of B is applied to the Y inputs of the adder.
    - If Cin = 1, then D = A + B + 1. This produces A plus the 2's complement of B, which is equivalent to a subtraction of A -B.
    - When Cin = 0 then D = A + B. This is equivalent to a subtract with borrow, that is, A-B-1.

Increment:

- When S1S0 = 10, the inputs from B are neglected, and instead, all 0's are inserted into the Y inputs. The output becomes D = A + 0 + Cin. This gives D = A when Cin = 0 and D = A + 1 when Cin = 1.
- In the first case we have a direct transfer from input A to output D.
- In the second case, the value of A is incremented by 1.

Decrement:

- When S1S0= 11, all l's are inserted into the Y inputs of the adder to produce the decrement operation D = A -1 when Cin = 0.

- This is because a number with all 1's is equal to the 2's complement of 1 (the 2's complement of binary 0001 is 1111). Adding a number A to the 2's complement of 1 produces F = A + 2's complement of 1 = A — 1. When Cin = 1, then D = A -1 + 1=A, which causes a direct transfer from input A to output D.

4.1.5 LOGIC MICROOPERATIONS

- Logic operations specify binary operations for strings of bits stored in registers and treat each bit separately.
- Example: the XOR of R1 and R2 is symbolized by

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.

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 truth tables obtained with two binary variables as **Table 4.4**.

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

Table 4.4 - Truth Tables for 16 Functions of Two Variables

- The 16 Boolean functions of two variables x and y are expressed in algebraic form in the first column of Table 4-6.
- 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 micro-operations listed in the second column represent a relationship between the binary content of two registers A and B.
- Example: R1 = 1010 and R2 = 1100

1010 Content of R1

1100 Content of R2

0110 Content of R1 after P = 1

- Symbols used for logical microoperations:
    - OR:V
    - AND: ^
    - XOR: $\oplus$
- The + sign has two different meanings: logical OR and summation
- When + is in a microoperation, then summation
- When + is in a control function, then OR
- Example:

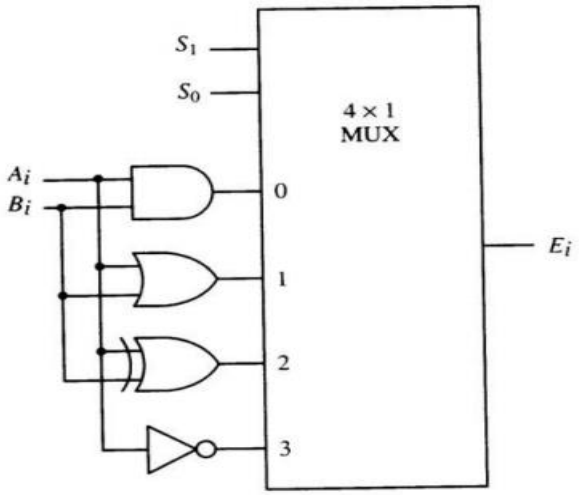
  P + Q: R1 ←R2 + R3, R4 ← R5 V R6
- There are 16 different logic operations that can be performed with two binary variables.

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR
$F_8 = (x + y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \overline{B}$	Complement B
$F_{11} = x + y'$	$F \leftarrow A \vee \overline{B}$	
$F_{12} = x'$	$F \leftarrow \overline{A}$	Complement A
$F_{13} = x' + y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow$ all 1's	Set to all 1's

Table 4.5 - Sixteen Logic Microoperations

- The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers.
- All 16 microoperations can be derived from using four logic gates.

- Figure 2-10 shows one stage of a circuit that generates the four basic logic microoperations. ¬ It consists of four gates and a multiplexer. Each of the four logic operations is generated through a gate that performs the required logic.
- The outputs of the gates are applied to the data inputs of the multiplexer. The two selection inputs S1 and S0 choose one of the data inputs of the multiplexer and direct its value to the output.



(a) Logic diagram

S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \overline{A}$	Complement

(b) Function table

Figure 4.10 - One Stage of Logic Circuit

Some Applications:

- Logic micro-operations are very useful for manipulating individual bits or a portion of a word stored in a register.
- They can be used to change bit values, delete a group of bits or insert new bits values into a register.
- The following example shows how the bits of one register (designated by A) are manipulated by logic microoperations as a function of the bits of another register (designated by B).

Selective Set

- The selective-set operation sets to 1 the bits in register A where there are corresponding l'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 OR microoperation can be used to selectively set bits of a register.

Selective Complement

- The selective-complement operation complements bits in A where there are corresponding 1's 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

- The exclusive-OR microoperation can be used to selectively complement bits of a register.

Selective Clear

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

1010 A before

1100 B (Logic Operand)

0010 A after

- The corresponding logic microoperation is $A \leftarrow \overline{A} \wedge B$.

Mask

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

1010 A before

1100 B (mask)

1000 A After Masking

Insert

- The insert operation inserts a new value into a group of bits

- This is done by first masking the bits to be replaced and then Oring them with the bits to be inserted

0110 1010	A before	0000 1010	A before
0000 1111	B (mask)	1001 0000	B (insert)
0000 1010	A after masking	1001 1010	A after insertion

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

Clear

- The clear operation compares the bits in A and B and produces an all 0‟s result if the two number are equal.

1010 A

1010 B

0000 A ← A⊕ B

4.1.6 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.
- There are three types of shifts: **logical, circular, and arithmetic**.
- The symbolic notation for the shift microoperations is shown in **Table 4.6**.

Symbolic designation	Description
$R \leftarrow \text{shl } R$	Shift-left register R
$R \leftarrow \text{shr } R$	Shift-right register R
$R \leftarrow \text{cil } R$	Circular shift-left register R
$R \leftarrow \text{cir } R$	Circular shift-right register R
$R \leftarrow \text{ashl } R$	Arithmetic shift-left R
$R \leftarrow \text{ashr } R$	Arithmetic shift-right R

Table 4.6 - Shift Microoperations

1. A **logical shift** is one that transfers 0 through the serial input.

- The symbols shl and shr are for logical shift-left and shift-right by one position R1 ← shlR1.

2. The **circular shift** (aka rotate) circulates the bits of the register around the two ends without loss of information.
    - The symbols cil and cir are for circular shift left and right.
3. The **arithmetic shift** shifts a signed binary number to the left or right.
    - To the left is multiplying by 2, to the right is dividing by 2.
    - Arithmetic shifts must leave the sign bit unchanged.
    - A sign reversal occurs if the bit in Rn-1 changes in value after the shift.
    - This happens if the multiplication causes an overflow.
    - An overflow flip-flop Vs can be used to detect

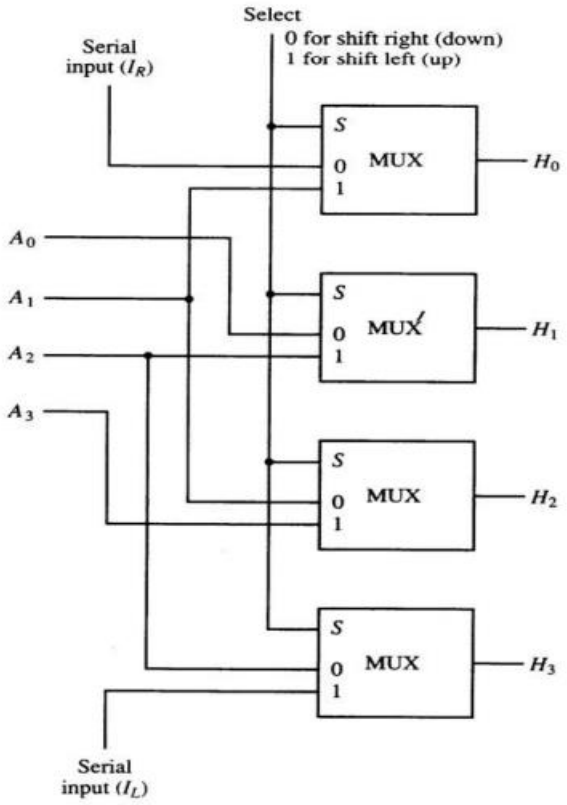
The overflow Vs = Rn-1 ⊕Rn-2

Figure 4.11 - Arithmetic Shift Right

HARDWARE IMPLEMENTATION:

- A combinational circuit shifter can be constructed with multiplexers as shown in **Figure 4.12**.
- The 4-bit shifter has four data inputs, A0 through A3, and four data outputs, H0 through H3.
- There are two serial inputs, one for shift left (IL) and the other for shift right (IR).
- When the selection input S=0 the input data are shifted right (down in the diagram).
- When S = 1, the input data are shifted left (up in the diagram).
- The function table in Fig. 4-12 shows which input goes to each output after the shift.
- A shifter with n data inputs and outputs requires n multiplexers.

- The two serial inputs can be controlled by another multiplexer to provide the three possible types of shifts.



Function table

Select	Output			
S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

Figure 4.12 - 4-Bit Combinational Circuit Shifter

4.1.7 ARITHMETIC LOGIC SHIFT UNIT

- The arithmetic logic unit (ALU) is a common operational unit connected to a number of storage registers.
- To perform a microoperation, the contents of specified registers are placed in the inputs of the ALU.
- The ALU performs an operation and the result 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.

- The arithmetic, logic, and shift circuits introduced in previous sections can be combined into one ALU with common selection variables. One stage of an arithmetic logic shift unit is shown in **Figure 4.13**.

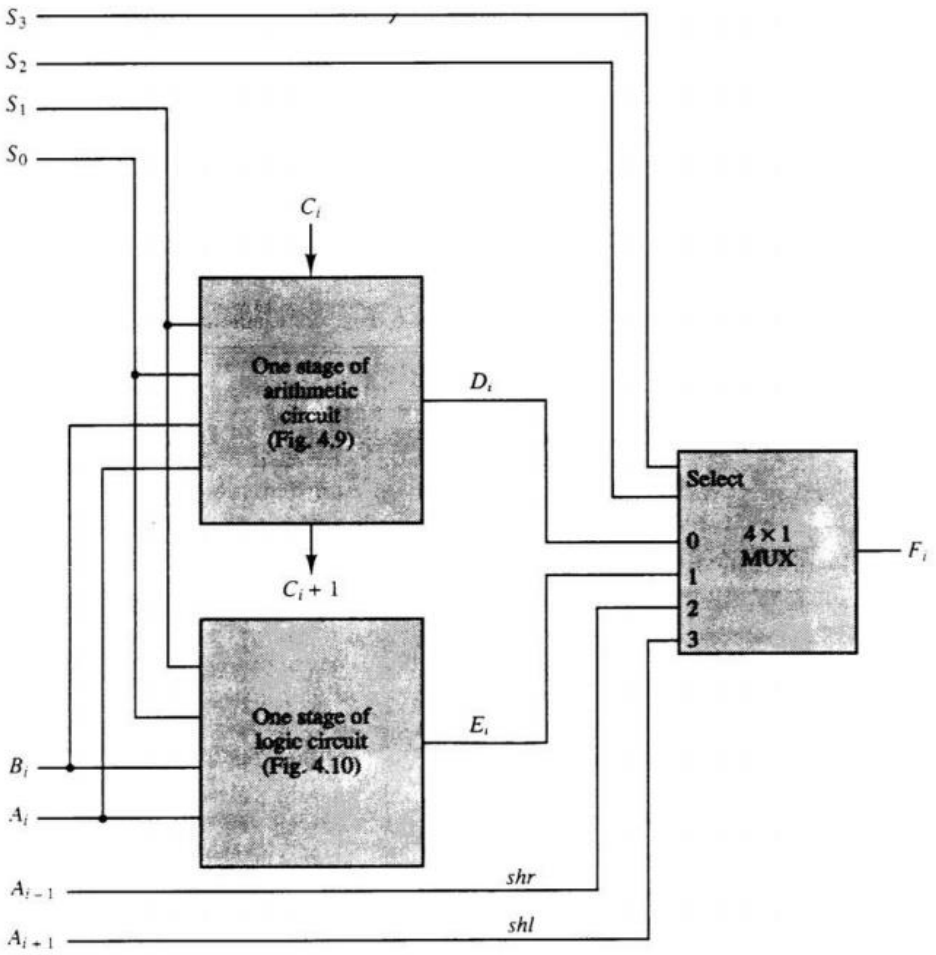


Figure 4.13 - One Stage of Arithmetic Logic Shift Unit

- Particular microoperation is selected with inputs S1 and S0. A 4 x 1 multiplexer at the output chooses between an arithmetic output in Di and a logic output in Ei.
- The data in the multiplexer are selected with inputs S3 and S2. The other two data inputs to the multiplexer receive inputs Ai-1 for the shift-right operation and Ai+1 for the shift-left operation.
- The circuit whose one stage is specified in Fig. 4-13 provides eight arithmetic operation, four logic operations, and two shift operations.
- Each operation is selected with the five variables S3, S2, S1, S0 and Cin.

- The input carry Cin is used for selecting an arithmetic operation only.

Operation select						
S_3	S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + \overline{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \overline{B} + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	×	$F = A \wedge B$	AND
0	1	0	1	×	$F = A \vee B$	OR
0	1	1	0	×	$F = A \oplus B$	XOR
0	1	1	1	×	$F = \overline{A}$	Complement A
1	0	×	×	×	$F =$ shr A	Shift right A into F
1	1	×	×	×	$F =$ shl A	Shift left A into F

Table 4.7 - Functional Table for Arithmetic Logic Shift Unit

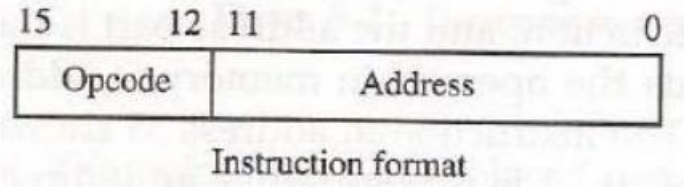
- **Table 4.7** lists the 14 operations of the ALU. The first eight are arithmetic operations and are selected with S3S2 = 00.
- The next four are logic and are selected with S3S2 = 01.
- The input carry has no effect during the logic operations and is marked with don't-care x's.
- The last two operations are shift operations and are selected with S3S2= 10 and 11.
- The other three selection inputs have no effect on the shift.

4.2. BASIC COMPUTER ORGANIZATION AND DESIGN

4.2.1 INSTRUCTION CODES:

- The organization of the computer is defined by its internal registers, the timing and control structure, and the set of instructions that it 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.
- **Program**: set of instructions that specify the operations, operands, and the sequence by which processing has to occur.
- **Instruction**: a binary code that specifies a sequence of micro-operations for the computer.
- 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 micro-operations. – Instruction Cycle.
- **Instruction Code**: group of bits that instruct the computer to perform specific operation.
- Instruction code is usually divided into two parts: Opcode and address(operand).



Instruction format

➢ **Operation Code (opcode):**

- Group of bits that define the operation.
- Eg: add, subtract, multiply, shift, complement.
- No. of bits required for opcode depends on no. of operations available in computer.
- n bit opcode >= 2 n (or less) operations.

➢ **Address (operand):**

- Specifies the location of operands (registers or memory words).
- Memory words are specified by their address.
- Registers are specified by their k-bit binary code.
- k-bit address >= 2 k registers.

Stored Program Organization:

- The ability to store and execute instructions is the most important property of a general-purpose computer. That type of stored program concept is called stored program organization.
- The simplest way to organize a computer is to have one processor register and an instruction code format with two parts. The first part specifies the operation to be performed and the second specifies an address.
- The below **Figure 4-14** shows the stored program organization.

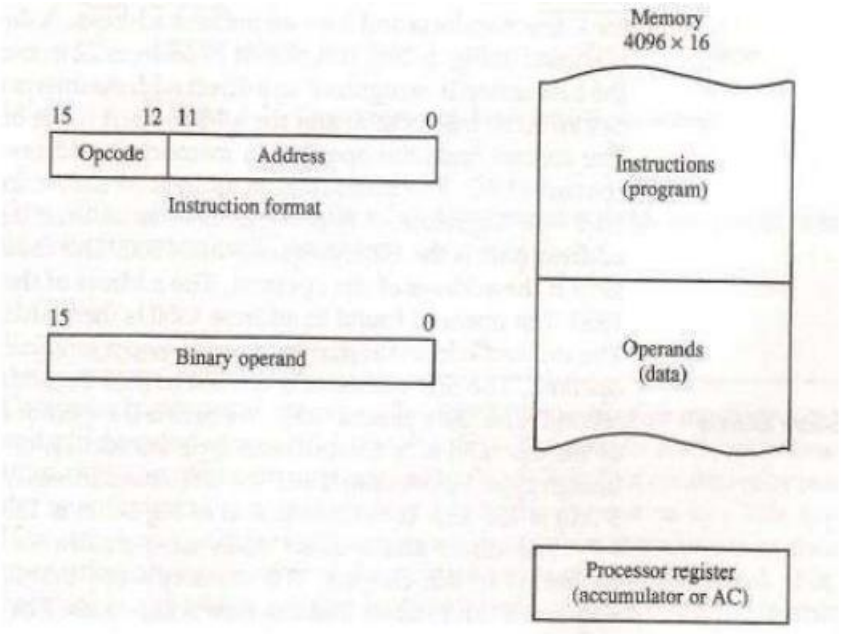


Figure 4.14 - Stored Program Organization

- Instructions are stored in one section of memory and data in another.
- For a memory unit with 4096 words, we need 12 bits to specify an address since 212 = 4096.
- If we store each instruction code in one 16-bit memory word, we have available four bits for the operation code (abbreviated opcode) to specify one out of 16 possible operations, and 12 bits to specify the address of an operand.

Accumulator (AC):

- Computers that have a single-processor register usually assign to it the name accumulator and label its AC.

- The operation is performed with the memory operand and the content of AC.

Addressing of Operand:

- 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 instruction code specifies an operand, the instruction is said to have an immediate operand.
- When the second part specifies the address of an operand, the instruction is said to have a direct address.
- When second part of the instruction designate an address of a memory word in which the address of the operand is found such instruction have indirect address.
- One bit of the instruction code can be used to distinguish between a direct and an indirect address.
- The instruction code format shown in **Figure 4.15(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 address and 1 for an indirect address.

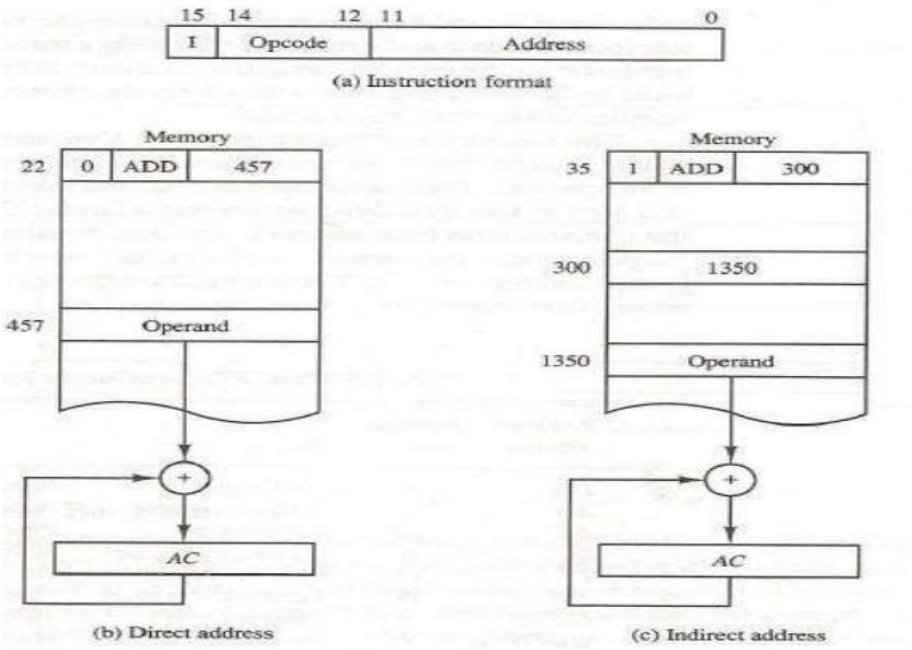


Figure 4.15 - Demonstration of Direct and Indirect Address

- A direct address instruction is shown in **Figure 4.15(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 **Figure 4.15(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 of the 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.
- The effective address to be the address of the operand in a computation-type instruction or the target address in a branch-type instruction.
- Thus, the effective address in the instruction of **Figure 4.15(b)** is 457 and in the instruction of **Figure 4.15(c)** is 1350.

4.2.2 COMPUTER REGISTERS

- The need of the registers in computer for:
    - Instruction sequencing needs a counter to calculate the address of the next instruction after execution of the current instruction is completed (PC).
    - Necessary to provide a register in the control unit for storing the instruction code after it is read from memory (IR).
    - Needs processor registers for manipulating data (AC and TR) and a register for holding a memory address (AR).
- The above requirements dictate the registers, their configuration, number of bits and their uses.

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 output character

Table 4.8 - List of Registers for the Basic Computer

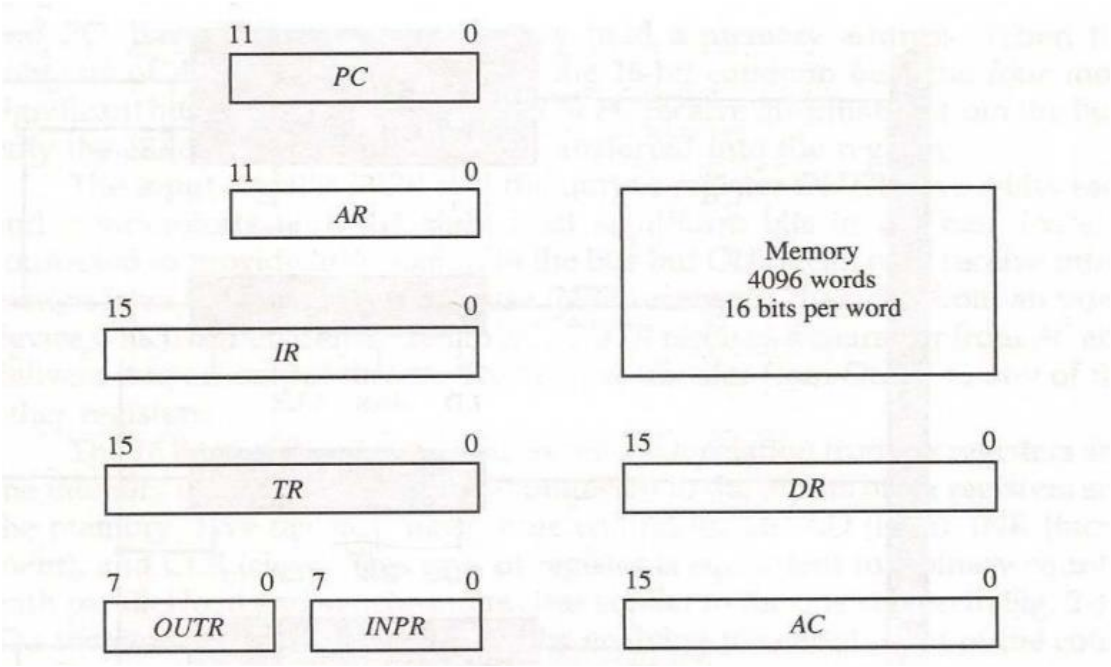


Figure 4.16 - Basic Computer Registers and Memory

- 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.
- 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.
- Two registers are used for input and output.
    - Two registers are used for input and output.

- The output register (OUTR) holds an 8-bit character for an output device.

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 registers.
- 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 **Figure 4.17.**

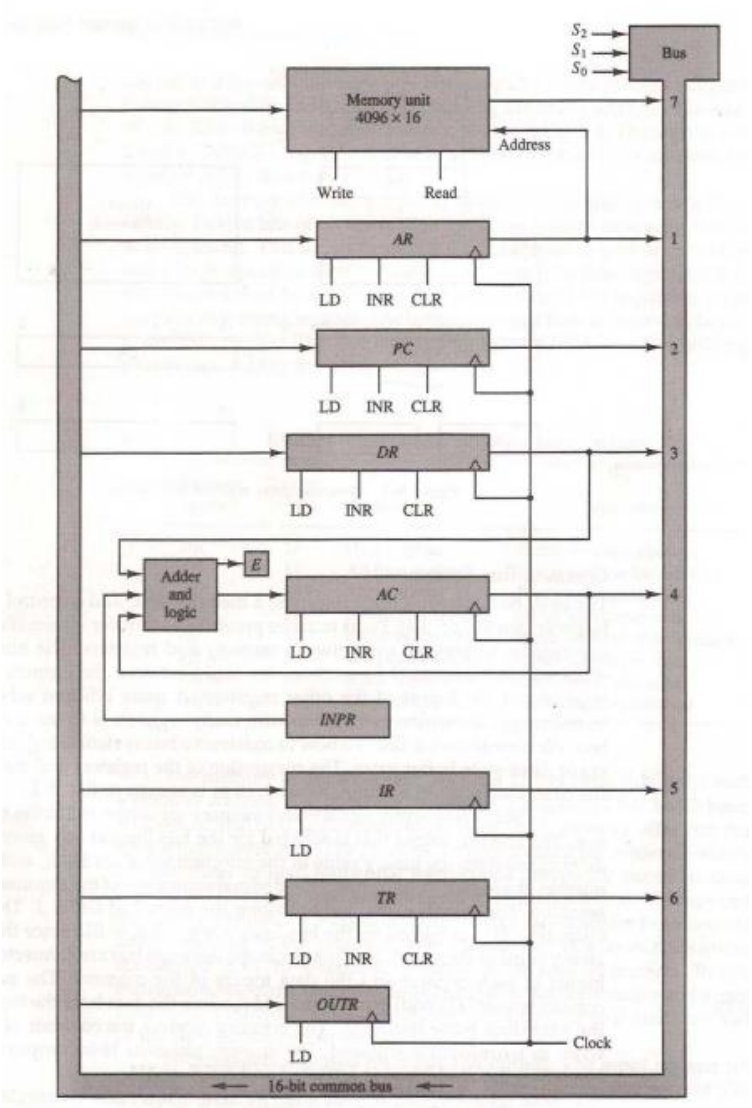


Figure 4.17 - Basic Computer Registers Connected to a Common Bus

- 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.
- The lines from the common bus are connected to the inputs 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
- 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 receives information from the bus, only the 12 least significant bits are transferred into the register.
- The input register INPR and the output register OUTR have 8 bits each.
- They 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 from an input device which is then transferred to AC.
- OUTR receives a character from AC and delivers it to an output device.
- Five registers have three control inputs: LD (load), INR (increment), and CLR (clear).
- This type of register is equivalent to a binary counter with parallel load and synchronous clear.
- Two registers have only a LD input.

- The input data and output data 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.
- 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.
    - One set of 16-bit inputs come from the outputs of AC.
    - The result of an addition is transferred to AC and the end carry-out of the addition is transferred to flip-flop E (extended AC bit).
    - A third set of 8-bit inputs come from the input register INPR.
- 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.
- For example, the two microoperations DR⇓AC and AC ⇓ 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 adder and logic circuit into AC, and enabling the LD (load) input of AC, all during the same clock cycle.

4.2.3 COMPUTER INSTRUCTIONS

- The basic computer has three instruction code formats, as shown in **Figure 4.18**. Each format has 16 bits.

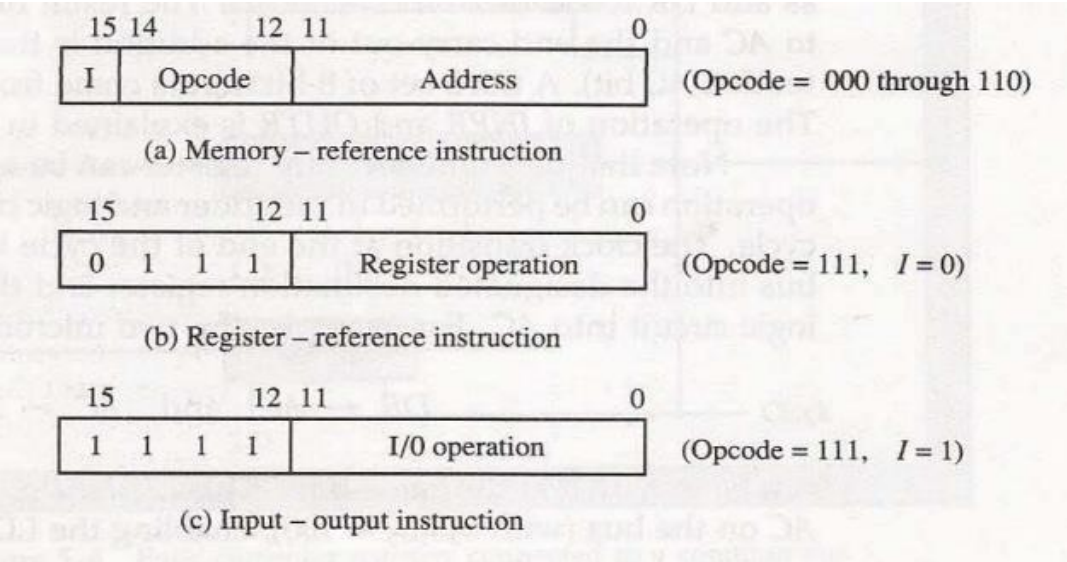


Figure 4.18 - Basic Computer Instruction Formats

- 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 an 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 1.11 with a 0 in the leftmost bit (bit 15) of the instruction.
- A register-reference instruction specifies an operation on the AC register. So, an operand from memory is not needed. Therefore, the other 12 bits are used to specify the operation to be executed.
- 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. ¬ The instructions for the computer are listed in **Table 4.9**.
- The symbol designation is a three-letter word and represents an abbreviation intended for programmers and users.
- The hexadecimal code is equal to the equivalent hexadecimal number of the binary code used for the instruction.

Symbol	Hexadecimal code $I = 0$	$I = 1$	Description
AND	0xxx	8xxx	AND memory word to *AC*
ADD	1xxx	9xxx	Add memory word to *AC*
LDA	2xxx	Axxx	Load memory word to *AC*
STA	3xxx	Bxxx	Store content of *AC* in 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 instruction if *AC* positive
SNA	7008		Skip next instruction if *AC* negative
SZA	7004		Skip next instruction if *AC* zero
SZE	7002		Skip next instruction if *E* is 0
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

Table 4.9 - Basic Computer Instructions

Instruction Set Completeness:

- A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function.
- The set of instructions are said to be complete if the computer includes a sufficient number of instructions in each of the following categories:
    - Arithmetic, logical, and shift instructions
    - Instructions for moving information to and from memory and processor registers

- Program control instructions together with instructions that check status conditions
- Input and output instructions

- 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-2's complement representation.
- The circulate instructions, CIR and CIL; can be used for arithmetic shifts as well as any other type of shifts desired.
- There are three logic operations: AND, complement AC (CMA), and clear AC(CLA). The ANDand 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 AC (STA) instruction.
- The branch instructions BUN, BSA, and ISZ, together with the four skip instructions, provide capabilities for program control and checking of status conditions.
- The input (INP} and output (OUT) instructions cause information to be transferred between the computer and external devices.

4.2.4 TIMING AND CONTROL

All sequential circuits in the Basic Computer CPU are driven by a master clock, with the exception of the INPR register. At each clock pulse, the control unit sends control signals to control inputs of the bus, the registers, and the ALU. Control unit design and implementation can be done by two general methods:

- A **hardwired control** unit is designed from scratch using traditional digital logic design techniques to produce a minimal, optimized circuit. In other words, the control unit is like an ASIC (application-specific integrated circuit).
- A **microprogrammed control** unit is built from some sort of ROM. The desired control signals are simply stored in the ROM, and retrieved in sequence to drive the microoperations needed by a particular instruction.

The differences between hardwired and microprogrammed control are

Hardwired control	Microprogrammed control
✓ The control logic is implemented with gates, flip-flops, decoders, and other digital circuits.	✓ The control information is stored in a control memory. The control memory is programmed to initiate the required sequence of microoperations.
✓ The advantage that it can be optimized to produce a fast mode of operation.	✓ Compared with the hardwired control operation is slow.
✓ Requires changes in the wiring among the various components if the design has to be modified or changed.	✓ Required changes or modifications can be done by updating the microprogram in control memory.

- The block diagram of the hardwired control unit is shown in **Figure 4.19**.
- 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). It is divided into three parts: The I bit, the operation code, and bits 0 through 11.
- The operation code in bits 12 through 14 are decoded with a 3 x 8 decoder. The eight outputs of the decoder are designated by the symbols D0 through D7.
- Bit 15 of the instruction is transferred to a flip-flop designated by the symbol I.
- Bits 0 through 11 are applied to the control logic gates.
- The 4-bit sequence counter can count in binary from 0 through 15.

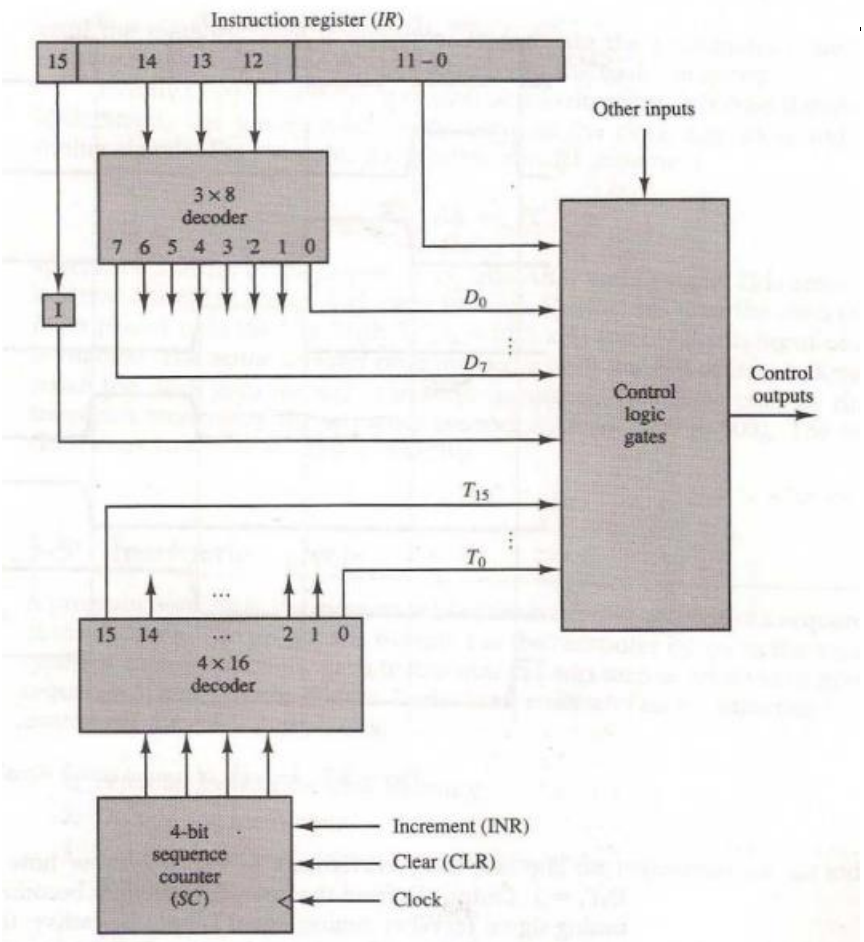


Figure 4.19 - Control Unit of Basic Computer

- The outputs of the counter are decoded into 16 timing signals T0 through T15.
- The sequence counter SC can be incremented or cleared synchronously.
- The counter is incremented to provide the sequence of timing signals out of the 4 x 16 decoder.
- As an example, consider the case where SC is incremented to provide timing signals T0, T1, T2, T3 and T4 in sequence. At time T4, SC is cleared to 0 if decoder output D3 is active.
- This is expressed symbolically by the statement D3T4.
- The timing diagram of **Figure 4.20** 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 T0 out of the decoder. T0 is active during one clock cycle.

- SC is incremented with every positive clock transition, unless its CLR input is active.
- This produces the sequence of timing signals T0, T1, T2, T3, T4and so on, as shown in the diagram.
- The last three waveforms in **Figure 4.20** show how SC is cleared when D3T4 = 1.
- Output D3 from the operation decoder becomes active at the end of timing signal T2.
- When timing signal T4 becomes active, the output of the AND gate that implements the control function D3T4 becomes active.
- This signal is applied to the CLR input of SC. On the next positive clock transition (the one marked T4 in the diagram) the counter is cleared to 0.
- This causes the timing signal T0 to become active instead of T5 that would have been active if SC were incremented instead of cleared.

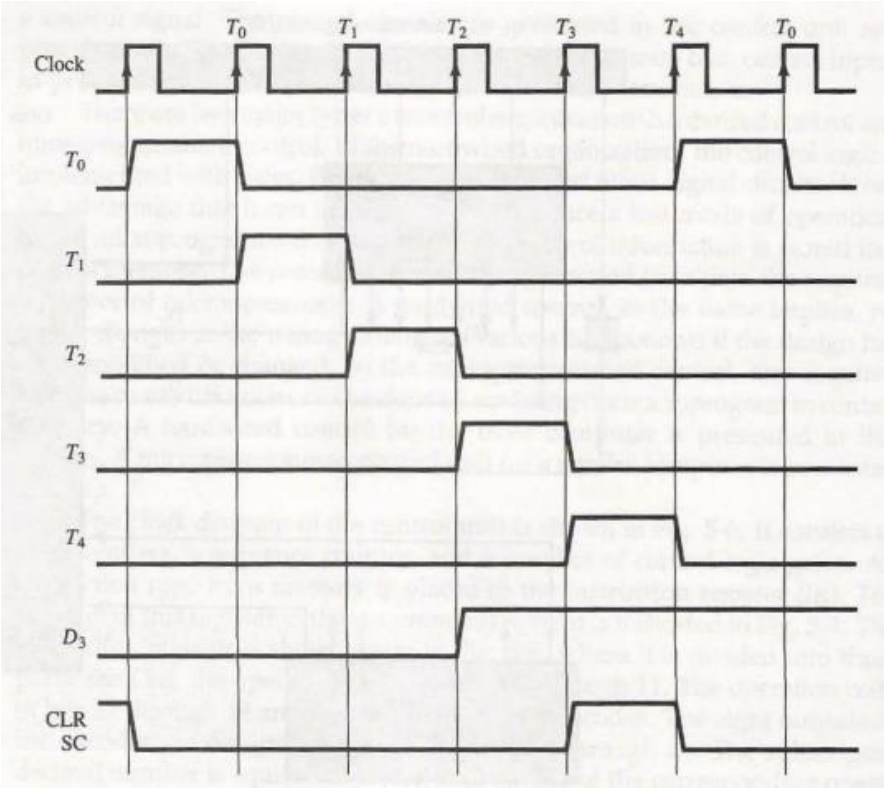


Figure 4.20 - Example of control timing signals

4.2.5 INSTRUCTION CYCLE

The CPU performs a sequence of microoperations for each instruction. The sequence for each instruction of the Basic Computer can be refined into 4 abstract phases:

1. Fetch instruction
2. Decode
3. Fetch operand
4. Execute

Program execution can be represented as a top-down design:

1. Program execution
a. Instruction 1
i. Fetch instruction
ii. Decode
iii. Fetch operand
iv. Execute
b. Instruction 2
i. Fetch instruction
ii. Decode
iii. Fetch operand
iv. Execute
c. Instruction 3 ...

Program execution begins with:

PC ← address of first instruction, SC ← 0

After this, the SC is incremented at each clock cycle until an instruction is completed, and then it is cleared to begin the next instruction. This process repeats until a HLT instruction is executed, or until the power is shut off.

Instruction Fetch and Decode

- The instruction fetch and decode phases are the same for all instructions, so the control functions and microoperations will be independent of the instruction code.
- Everything that happens in this phase is driven entirely by timing variables T0, T1 and T2. Hence, all control inputs in the CPU during fetch and decode are functions of these three variables alone.

T0: AR ← PC

T1: IR ← M[AR], PC ← PC + 1

T2: D0-7 ← decoded IR(12-14), AR ← IR(0-11), I ← IR(15)

For every timing cycle, we assume SC ← SC + 1 unless it is stated that SC ← 0.

4.2.6 MEMORY-REFERENCE INSTRUCTIONS

- **Table 4.10** lists the seven memory-reference instructions.
- The decoded output Di for i = 0, 1, 2, 3, 4, 5, and 6 from the operation decoder that belongs to each instruction is included in the table.
- The effective address of the instruction is in the address register AR and was placed there during timing signal T2 when I= 0, or during timing signal T3 when I = 1.
- The execution of the memory-reference instructions starts with timing signal T4.
- The symbolic description of each instruction is specified in the table in terms of register transfer notation.

Symbol	Operation decoder	Symbolic description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR], \quad E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC, \quad PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1,$ If $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$

Table 4.10 – Memory Reference Instructions

AND to AC:

- This is an instruction that performs the AND logic operation on pairs of bits in AC and the memory word specified by the effective address.
- The result of the operation is transferred to AC.
- The microoperations that execute this instruction are:

D_1T_4: $DR \leftarrow M[AR]$
D_1T_5: $AC \leftarrow AC + DR, \quad E \leftarrow C_{out}, \quad SC \leftarrow 0$

ADD to AC:

- This instruction adds the content of the memory word specified by the effective address to the value of AC.

- The sum is transferred into AC and the output carry Cout is transferred to the E (extended accumulator) flip-flop.
- The microoperations needed to execute this instruction are

$$D_1T_4: \quad DR \leftarrow M[AR]$$
$$D_1T_5: \quad AC \leftarrow AC + DR, \quad E \leftarrow C_{out}, \quad SC \leftarrow 0$$

LDA: Load to AC

- This instruction transfers the memory word specified by the effective address to AC.
- The microoperations needed to execute this instruction are

$$D_2T_4: \quad DR \leftarrow M[AR]$$
$$D_2T_5: \quad AC \leftarrow DR, \quad SC \leftarrow 0$$

BUN: Branch Unconditionally

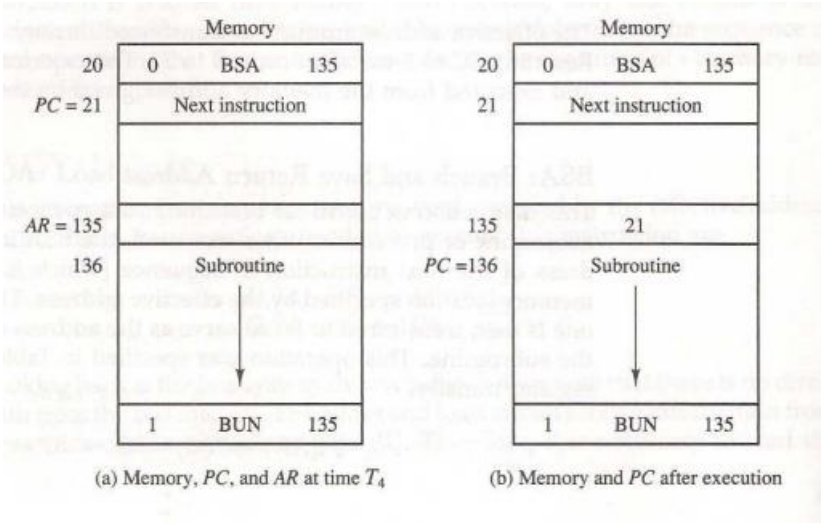
- This instruction is useful for branching to a portion of the program called a subroutine or procedure.
- When executed, the BSA instruction stores the address of the next instruction in sequence (which is available in PC) into a memory location specified by the effective address.
- The effective address plus one is then transferred to PC to serve as the address of the first instruction in the subroutine.
- This operation was specified with the following register transfer:

$$M[AR] \leftarrow PC, \quad PC \leftarrow AR + 1$$

- A numerical example that demonstrates how this instruction is used with a subroutine is shown in **Figure 4.21.**

Figure 4.21 - Example Of BSA Instruction Execution

- The BSA instruction is assumed to be in memory at address 20.



- The I bit is 0 and the address part of the instruction has the binary equivalent of 135.
- After the fetch and decode phases, PC contains 21, which is the address of the next instruction in the program (referred to as the return address). AR holds the effective address 135.
- This is shown in part (a) of the figure.
- The BSA instruction performs the following numerical operation:

$$M[135] \leftarrow 21, \quad PC \leftarrow 135 + 1 = 136$$

- The result of this operation is shown in part (b) of the figure.
- The return address21 is stored in memory location 135 and control continues with the subroutine program starting from address 136.
- The return to the original program (at address 21) is accomplished by means of an indirect BUN instruction placed at the end of the subroutine.
- When this instruction is executed, control goes to the indirect phase to read the effective address at location 135, where it finds the previously saved address 21.
- When the BUN instruction is executed, the effective address 21 is transferred to PC.
- The next instruction cycle finds PC with the value 21, so control continues to execute the instruction at the return address.

- The BSA instruction must be executed with a sequence of two microoperations:

$$D_5T_4: \quad M[AR] \leftarrow PC, \quad AR \leftarrow AR + 1$$
$$D_5T_5: \quad PC \leftarrow AR, \quad SC \leftarrow 0$$

ISZ: Increment and Skip if Zero

- This instruction increments the word specified by the effective address, and if the incremented value is equal to 0, PC is incremented by 1 to skip the next instruction in the program.
- Since it is not possible to increment a word inside the memory, it is necessary to read the word into DR, increment DR, and store the word back into memory.
- Since it is not possible to increment a word inside the memory, it is necessary to read the word into DR, increment DR, and store the word back into memory.

$$D_6T_4: \quad DR \leftarrow M[AR]$$
$$D_6T_5: \quad DR \leftarrow DR + 1$$
$$D_6T_6: \quad M[AR] \leftarrow DR, \quad \text{if } (DR = 0) \text{ then } (PC \leftarrow PC + 1), \quad SC \leftarrow 0$$

Control Flowchart:

- A flowchart showing all microoperations for the execution of the seven memory-reference instructions is shown in **Figure 4.22**.

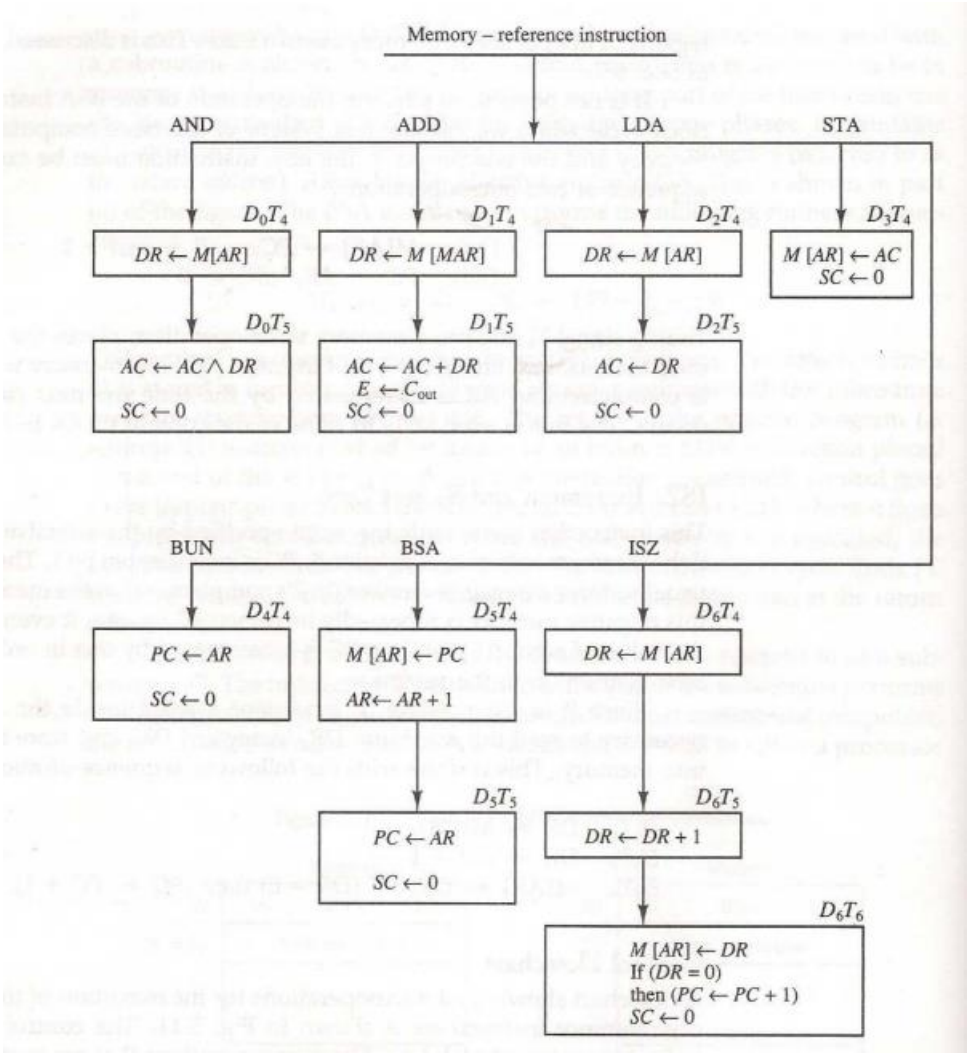


Figure 4.22 - Flowchart for Memory Reference Instructions

4.2.7 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.
- To demonstrate the most basic requirements for input and output communication, we will use as an illustration a terminal unit with a keyboard and printer.

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 **Figure 4.23**.

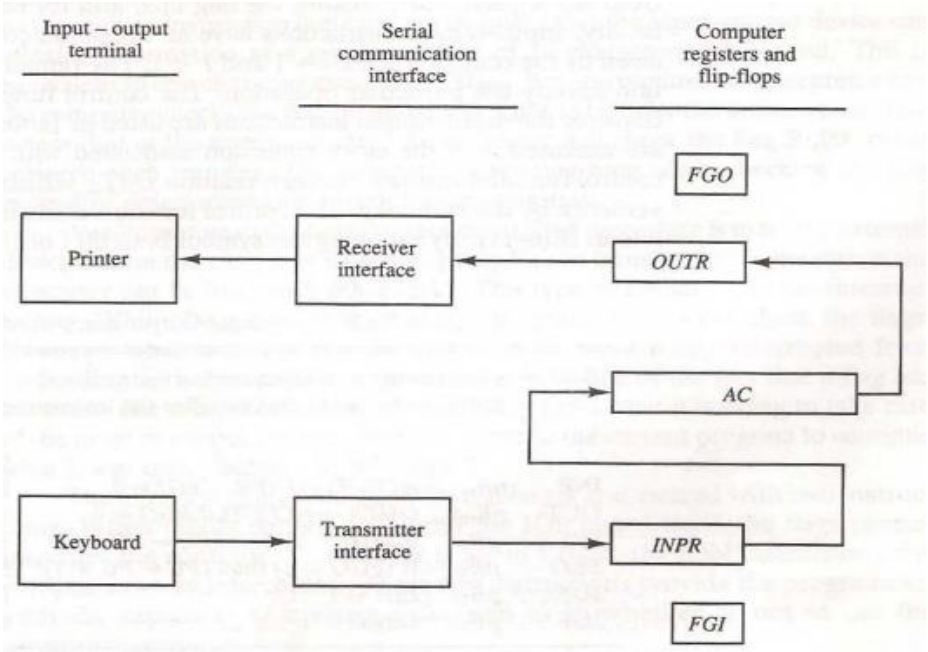


Figure 4.23 - Input-Output Configuration

- The input register INPR consists of eight bits and holds alphanumeric input information.
- The 1-bit input flag FGI is a control 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.
- 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.

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 4.11.**

$D_7IT_3 = p$ (common to all input–output instructions)
$IR(i) = B_i$ [bit in $IR(6–11)$ that specifies the instruction]

	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0–7) \leftarrow INPR$, $FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC(0–7)$, $FGO \leftarrow 0$	Output character
SKI	pB_9:	If ($FGI = 1$) then ($PC \leftarrow PC + 1$)	Skip on input flag
SKO	pB_8:	If ($FGO = 1$) then ($PC \leftarrow PC + 1$)	Skip on output flag
ION	pB_7:	$IEN \leftarrow 1$	Interrupt enable on
IOF	pB_6:	$IEN \leftarrow 0$	Interrupt enable off

Table 4.11 - Input-Output Instructions

- These instructions are executed with the clock transition associated with timing signal T3.
- Each control function needs a Boolean relation D7IT3, which we designate for convenience by the symbol p.
- The control function is distinguished by one of the bits in IR (6-11).
- By assigning the symbol Bi to bit i of IR, all control functions can be denoted by pBi for i = 6 though 11.
- The sequence counter SC is cleared to 0 when p = D7IT3 = 1.
- The last two instructions set and clear an interrupt enable flip-flop IEN.

Program Interrupt:

- 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.
- 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.
- When a flag is set, the computer is momentarily interrupted from the current program.
- The computer deviates momentarily from what it is doing to perform of the input or output transfer.
- Tt 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 (with the ION instruction), the computer can be interrupted.
- The way that the interrupt is handled by the computer can be explained by means of the flowchart of **Figure 4.24**.
- 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 1EN = 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.

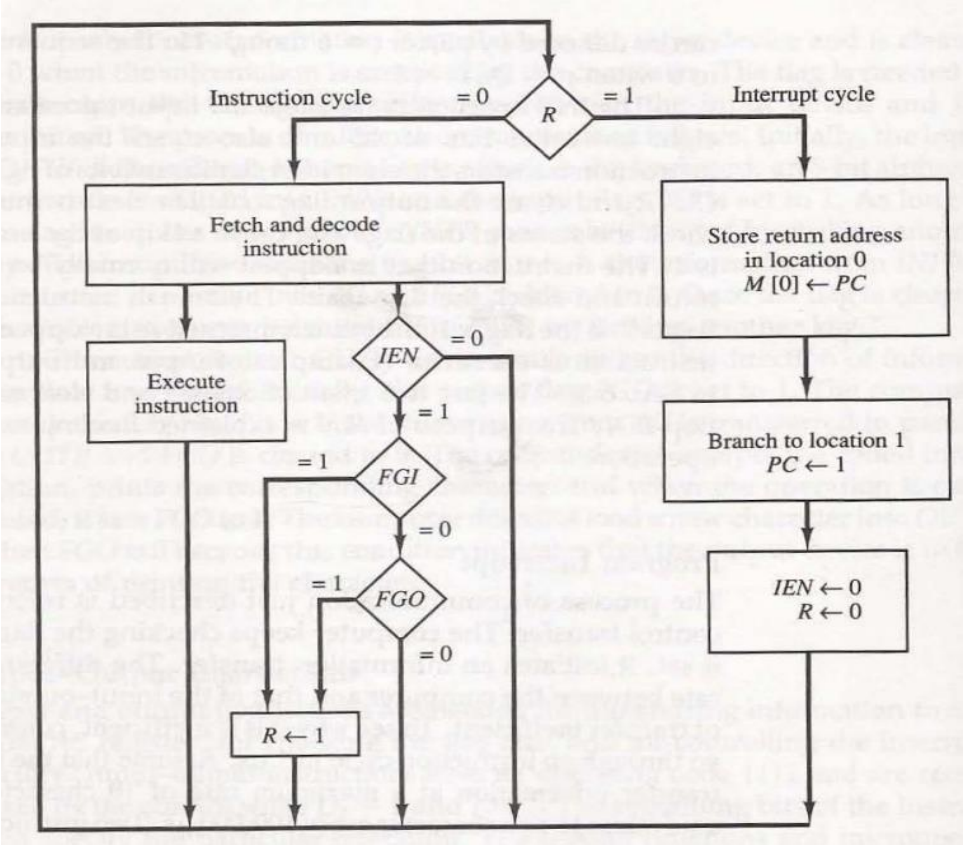


Figure 4.24 - Flowchart for Interrupt Cycle

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.
- This location may be a processor register, a memory stack, or a specific memory location.

- An example that shows what happens during the interrupt cycle is shown in **Figure 4.25.**

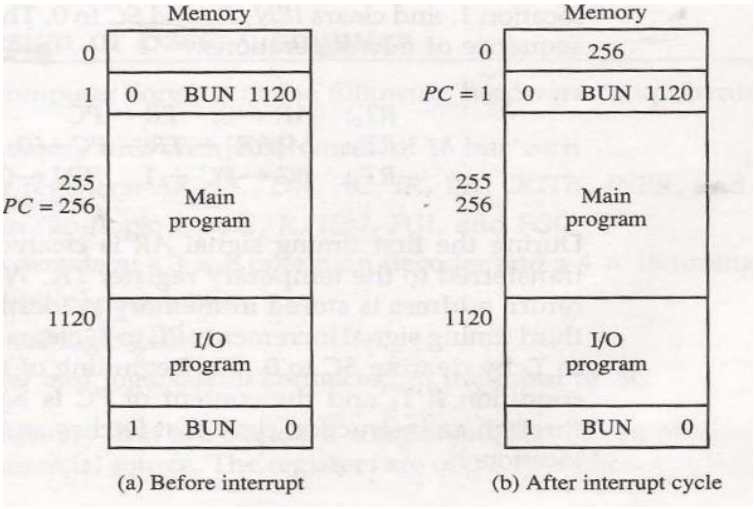


Figure 4.25 - Demonstration of the Interrupt Cycle

- When an interrupt occurs and R is set to 1 while the control is executing the instruction at address 255.
- At this time, the returns address 256 is in PC.
- The programmer has previously placed an input—output service program in memory starting from address 1120 and a BUN 1120 instruction at address 1. This is shown in **Figure 4.25(a)**.
- When control reaches timing signal T0and finds that R = 1, it proceeds with the interrupt cycle. The content of PC (256) is stored in memory location 0, PC is set to 1, and R is cleared to 0.
- The branch instruction at address 1 causes the program to transfer to the input—output service program at address 1120.
- This program checks the flags, determines which flag is set, and then transfers the required input or output information.
- Once this is done, the instruction ION is executed to set IEN to 1 (to enable further interrupts), and the program returns to the location where it was interrupted.
- This is shown in **Figure 4.25(b)**.

MODULE 5

REDUCED INSTRUCTION SET COMPUTER

INTRODUCTION OF RISC & 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. In this section we introduce the major characteristics of CISC and RISC architectures and then present the instruction set and instruction format of a RISC processor.

5.1 CISC CHARACTERISTICS

- The design of an instruction set for a computer must take into consideration not only machine language constructs, but also the requirements imposed on the use of high-level programming languages.
- The translation from high-level to machine language programs is done by means of a compiler program. One reason for the trend to provide a complex instruction set is the desire to simplify the compilation and improve the overall computer performance.
- The task of a compiler is to generate a sequence of machine instructions for each high-level language statement. The task is simplified if there are machine instructions that implement the statements directly.
- The essential goal of a CISC architectures to attempt to provide a single machine instruction for each statement that is written in a high-level language. Examples of CISC architectures are the Digital Equipment Corporation VAX computer and the IBM 370 computer.
- 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. If the computer has 32-bit words (four bytes), the first instruction occupies half a word, while the second instruction needs one word in addition to one byte in the next word.
- Packing variable instruction formats in a fixed-length memory word requires special decoding circuits that count bytes within words and frame the instructions according their byte length. 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.

- Although CISC processors have instructions that use only processor registers, the availability of other modes of operations tend to simplify high-level language compilation. However, 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. In summary

The 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 in frequently
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.

Reduced Instruction Set Computer (RISC)

- An important aspect of computer architecture is the design of the instruction set for the processor. The instruction set chosen for a particular computer determines the way that machine language programs are constructed. Early computers had small and simple instruction sets, forced mainly by the need to minimize the hardware used to implement them. As digital hardware became cheaper with the advent of integrated circuits, computer instructions tended to increase both in number and complexity.
- Many computers have instruction sets that include more than 100 and sometimes even more than 200 instructions. These computers also employ a variety of data types and a large number of addressing modes. The trend into computer hardware complexity was influenced by various factors, such as upgrading existing models to provide more customer applications, adding instructions that facilitate the translation from high-level language into machine language programs, and striving to develop machines that move functions from

software implementation into hardware implementation. A computer with a large number of instruction is classified as a complex instruction set computer, abbreviated CISC.

5.2 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 instructions
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 of register-to- register operations, with only simple load and store operations for memory access. Thus each operand is brought into a processor register with a load instruction. All computations are done among the data stored in processor registers. Results are transferred to memory by means of store instructions. This architectural feature simplifies the instruction set and encourages the optimization of register manipulation. The use of only a few addressing modes results from the fact that almost all instructions have simple register addressing. Other addressing modes may be included, such as immediate operands and relative mode.

5.3 PIPELINE AND VECTOR PROCESSING

5.3.1 PARALLEL PROCESSING:

- Parallel processing is a term used to denote a large class of techniques that are used to provide simultaneous data-processing tasks for the purpose of increasing the computational speed of a computer system.

- The purpose of parallel processing is to speed up the computer processing capability and increase its throughput, that is, the amount of processing that can be accomplished during a given interval of time.
- The amount of hardware increases with parallel processing, and with it, the cost of the system increases.
- Parallel processing can be viewed from various levels of complexity.
- At the lowest level, we distinguish between parallel and serial operations by the type of registers used. e.g. shift registers and registers with parallel load
- At a higher level, it can be achieved by having a multiplicity of functional units that perform identical or different operations simultaneously.
- Fig. below shows one possible way of separating the execution unit into eight functional units operating in parallel.
- A multifunctional organization is usually associated with a complex control unit to coordinate all the activities among the various components.

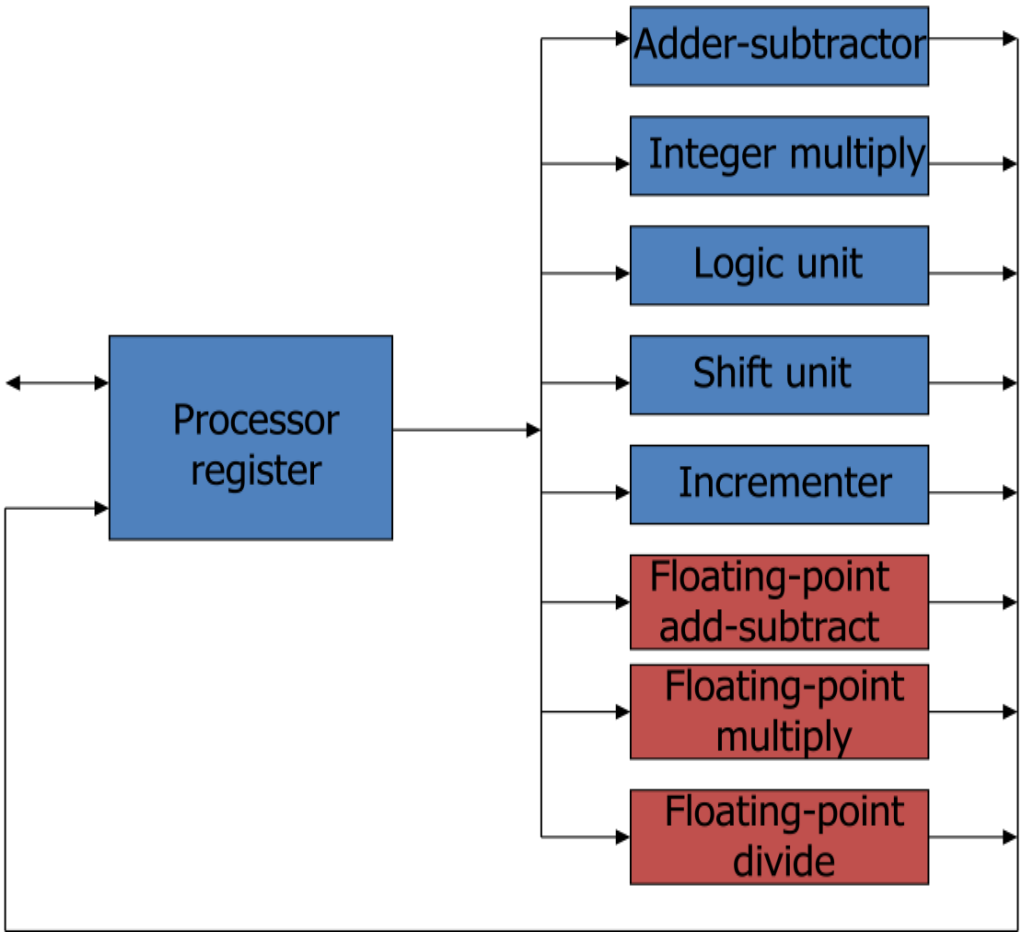


Figure 5.1 - Parallel Processing

The operands in the registers are applied to one of the units depending on the operation specified by the instruction associated with the operands. The operation performed in each functional unit is indicated in each block of the diagram. The adder and integer multiplier perform the arithmetic operations with integer numbers. The floating-point operations are separated into three circuits operating in parallel. The logic, shift, and increment operations can be performed concurrently on different data. All units are independent of each other, so one number can be shifted while another number is being incremented. A multifunctional organization is usually associated with a complex control unit to coordinate all the activities among the various components.

There are a variety of ways that parallel processing can be classified. it can be considered from the

- Internal organization of the processors
- Interconnection structure between processors
- The flow of information through the system

One classification introduced by M. Flynn considers the organization of a computer system by the number of instructions and data items that are manipulated simultaneously.

The normal operation of a computer is to fetch instructions from memory and execute them in the processor.

The sequence of instructions read from memory constitutes an *instruction stream.* The operations performed on the data in the processor constitute a *data stream*. Parallel processing may occur in the instruction stream, in the data stream, or in both. Flynn's classification divides computers into four major groups as follows:

- **Single instruction stream, single data stream (SISD)**
- **Single instruction stream, multiple data stream (SIMD)**
- **Multiple instruction stream, single data stream (MISD)**
- **Multiple instruction stream, multiple data stream (MIMD)**
- **Single instruction stream, single data stream (SISD)**
    - Represents the organization of a single computer containing a control unit, a processor unit, and a memory unit.

- Instructions are executed *sequentially* and the system may or may not have internal parallel processing capabilities.
- parallel processing may be achieved by means of *multiple functional units* or by
- *pipeline processing*.

➢ **Single instruction stream, multiple data stream (SIMD)**

- Represents an organization that includes many processing units under the supervision of a common control unit.
- All processors receive the same instruction from the control unit but operate on different items of data.
- The shared memory unit must contain *multiple modules* so that it can communicate with all the processors simultaneously.

➢ **Multiple instruction stream, single data stream (MISD)**

- MISD structure is only of theoretical interest since no practical system has been constructed using this organization.

➢ **Multiple instruction stream, multiple data stream (MIMD)**

- MIMD organization refers to a computer system capable of processing several programs at the same time.

e.*g. Multiprocessor and multicomputer system*

We consider parallel processing under the following main topics:

- **Pipeline processing:** Is an implementation technique where arithmetic sub operations or the phases of a computer instruction cycle overlap in execution.
- **Vector processing: Deals** with computations involving large vectors and matrices.
- **Array processing:** Perform computations on large arrays of data.

5.3.2 PIPELINING

- Pipelining is a technique of decomposing a sequential process into sub operations, with each sub process being executed in a special dedicated segment that operates

concurrently with all other segments.

- A pipeline can be visualized as a collection of processing segments through which binary information flows. Each segment performs partial processing dictated by the way the task is partitioned. The result obtained from the computation in each segment is transferred to the next segmenting the pipeline.
- The final result is obtained after the data have passed through all segments. The name "pipeline" implies a flow of information analogous to an industrial assembly line. It is characteristic of pipelines that several computations can be in progress in distinct segments at the same time.
- The overlapping of computation is made possible by associating a register with each segment in the pipeline isolation between each segment so that each can operate on distinct data simultaneously
- **The pipeline organization will be demonstrated by means of a simple**
- Example. Suppose that we want to perform the combined multiply and add operations with a stream of numbers

$$A_i * B_i + C_i \quad \text{for } i = 1, 2, 3, \ldots, 7$$

- Each sub operation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as **shown in Fig.** R 1 through RS are registers that receive new data with every clock pulse.
- The multiplier and adder are combinational circuits. The sub operations performed in each segment of the pipeline are as follows:

$R1 \leftarrow A_i, \quad R2 \leftarrow B_i$	Input A_i and B_i
$R3 \leftarrow R1 * R2, \quad R4 \leftarrow C_i$	Multiply and input C_i
$R5 \leftarrow R3 + R4$	Add C_i to product

- Input A, and B, Multiply and input C, Add C; to product The five registers are loaded with new data every clock pulse. The effect of each clock is shown in **Table 5.1**. The first clock pulse transfers A1 and B1 into R 1 and R2. The second dock pulse transfers the product of R 1 and R2 into R3 and C1 into R4. The same clock pulse transfers A2 and B2 into R 1 and R2. The third clock pulse operates on all three segments simultaneously. It places A, and B, into R1 and R2, transfers the product of R1 and R2

into R3, transfers C, into R4, and places the sum of R3 and R4 into RS. It takes three clock pulses to fill up the pipe and retrieve the first output from RS. From there on, each dock produces a new output and moves the data one step down the pipeline. This happens as long as new input data flow into the system. When no more input data are available, the clock must continue until the last output emerges out of the pipeline.

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

Table 5.1 – Content of Registers in Pipeline Example

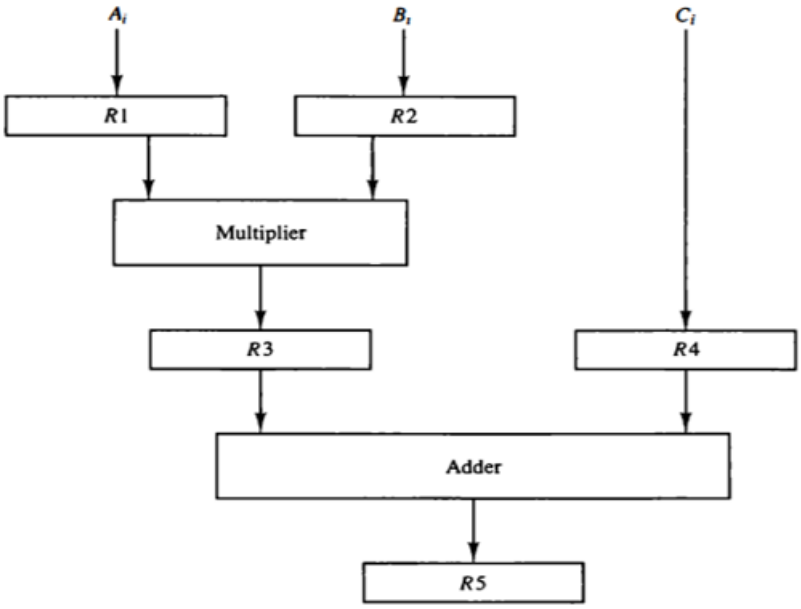


Figure 5.2 - Example of Pipeline Processing

5.3.3 FOUR-SEGMENT PIPELINE

The general structure of a four-segment pipeline is illustrated in Fig. 9-3. The operands pass through all four segments in a fixed sequence. Each segment consists of a

combinational circuit S; that performs a sub operation over the data stream flowing through the pipe. The segments are separated by registers R; that hold the intermediate results between the stages. Information flows between adjacent stages under the control of a common clock applied to all the registers simultaneously. We task define a task as the total operation performed going through all the segments in the pipeline The behaviour of a pipeline can be illustrated with a space-time diagram. This is a diagram that shows the segment utilization as a function of time. The space-time diagram of a four-segment pipeline is demonstrated in **Figure 5.3**. The horizontal axis displays the time in clock cycles and the vertical axis gives the segment number.

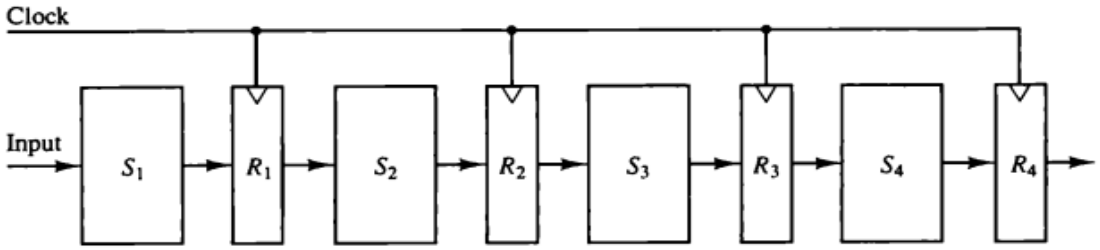


Figure 5.3 - Four-Segment Pipeline

The diagram shows six tasks T1 through T6 executed in four segments. Initially, task 1i is handled by segment 1. After the first clock, segment 2 is busy with T_1 while segment 1 is busy with task T2• Continuing in this manner, the first task T1 is completed after the fourth clock cycle. From then on, the pipe completes a task every clock cycle. No matter how many segments there are in the system, once the pipeline is full, it takes only one clock period to obtain an output.

Now consider the case where a k-segment pipeline with a clock cycle time t, is used to execute n tasks. The first task T1 requires a time equal to k_{tp}, to complete its operation since there are k segments in the pipe. The remaining n - 1 tasks emerge from the pipe at the rate of one task per clock cycle and they will be completed after a time equal to (n - 1)t,. Therefore, to complete n tasks using a k-segment pipeline requires k + (n - 1) clock cycles. For example, the diagram of **Figure 5.4** shows four segments and six tasks. The time required to complete all the operations is 4 + (6 - 1) = 9 clock cycles, as indicated in the diagram.

Next consider a non-pipeline unit that performs the same operation and takes a time equal to t. to complete each task. The total time required for n tasks is n_{tn}. The speedup of a pipeline processing over an equivalent non pipeline processing is defined by the ratio

$$S = \frac{nt_n}{(k + n - 1)t_p}$$

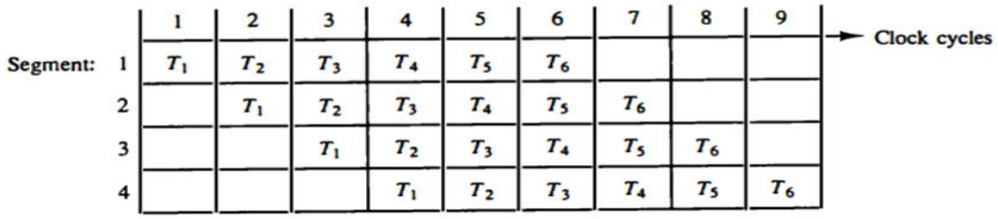


Figure 5.4 – Space Time Diagram of Pipeline

As the number of tasks increases, n becomes much larger than k - 1, and k + n - 1 approaches the value of n. Under this condition, the speedup becomes

$$S = \frac{t_n}{t_p}$$

If we assume that the time it takes to process a task is the same in the pipeline and non-pipeline circuits, we will have t, = kt,. Including this assumption, the speedup reduces to

$$S = \frac{kt_p}{t_p} = k$$

This shows that the theoretical maximum speedup that a pipeline can provide is k, where k is the number of segments in the pipeline.

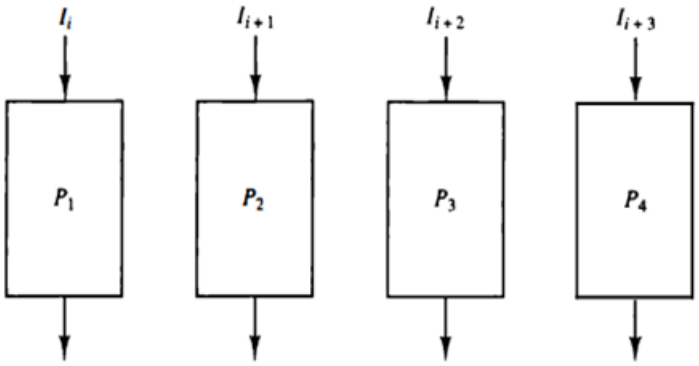


Figure 5.5 - Multiple Functional Units in Parallel

5.3.4 ARITHMETIC PIPELINE:

Pipeline arithmetic units are usually found in very high-speed computers.

They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems.

There are various reasons why the pipeline cannot operate at its maximumtheoretical rate.

- Different segments may take different times to complete their sub operation.
- It is not always correct to assume that a nonpipe circuit has the same time delay as that of an equivalent pipeline circuit.
- There are two areas of computer design where the pipeline organization is applicable.
- Arithmetic pipeline
- Instruction pipeline

Arithmetic Pipeline: Introduction

- Pipeline arithmetic units are usually found in very high-speed computers

Floating–point operations, multiplication of fixed-point numbers, and similar computations in scientific problem

- Floating–point operations are easily decomposed into sub operations as demonstrated in Sec. 10-5.

- An example of a pipeline unit for floating-point addition and subtraction is showed in the following:

o The inputs to the floating-point adder pipeline are two normalized floating- point binary number

$X = A \times 2^a$

$Y = B \times 2^b$

- A and B are two fractions that represent the mantissas, a and b are the exponents.
- The floating-point addition and subtraction can be performed in four segments, as shown in **Figure 5.6**.
- The sub operations that are performed in the four segments are:
    - *Compare the exponents*
    - The larger exponent is chosen as the exponent of the result.
    - *Align the mantissas*

- The exponent difference determines how many times the mantissa associated with the smaller exponent must be shifted to the right.
    - *Add or subtract the mantissas*
    - *Normalize the result*

- When an overflow occurs, the mantissa of the sum or difference is shifted right and the exponent incremented by one.
- If an underflow occurs, the number of leading zeros in the mantissa determines the number of left shifts in the mantissa and the number that must be subtracted from the exponent.

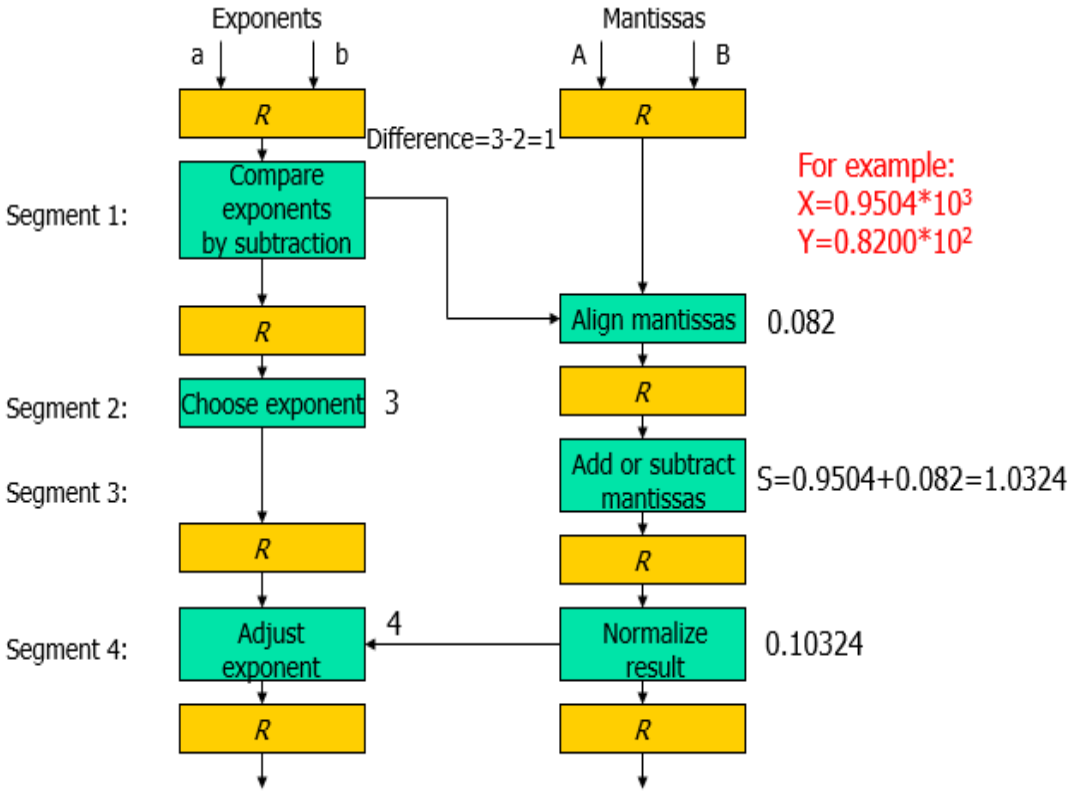


Figure 5.6 - Arithmetic Pipeline

5.3.5 INSTRUCTION PIPELINE

- Pipeline processing can occur not only in the *data stream* but in the *instruction* as well.
- Consider a computer with an instruction fetch unit (FIFO)and an instruction execution unit (PC) designed to provide a *two-segment* pipeline.
- Computers with complex instructions require other phases in addition to above phases to process an instruction completely.
- In the most general case, the computer needs to process each instruction with the following sequence of steps.
- Fetch the instruction from memory.
- Decode the instruction.
- Calculate the effective address.
- Fetch the operands from memory.

- Execute the instruction.
- Store the result in the proper place.
- There are certain difficulties that will prevent the instruction pipeline from operatingat its maximum rate.
- Different segments may take different times to operate on the incoming information.
- Some segments are skipped for certain operations.
- Two or more segments may require memory access at the same time, causing one segment to wait until another is finished with the memory.

Example: four-segment instruction pipeline: Assume that:

- The decoding of the instruction can be combined with the calculation of the effective address into one segment.
- The instruction execution and storing of the result can be combined into one segment. **Figure** shows how the instruction cycle in the CPU can be processed with a four- segment pipeline.
- Thus, up to four sub operations in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time.

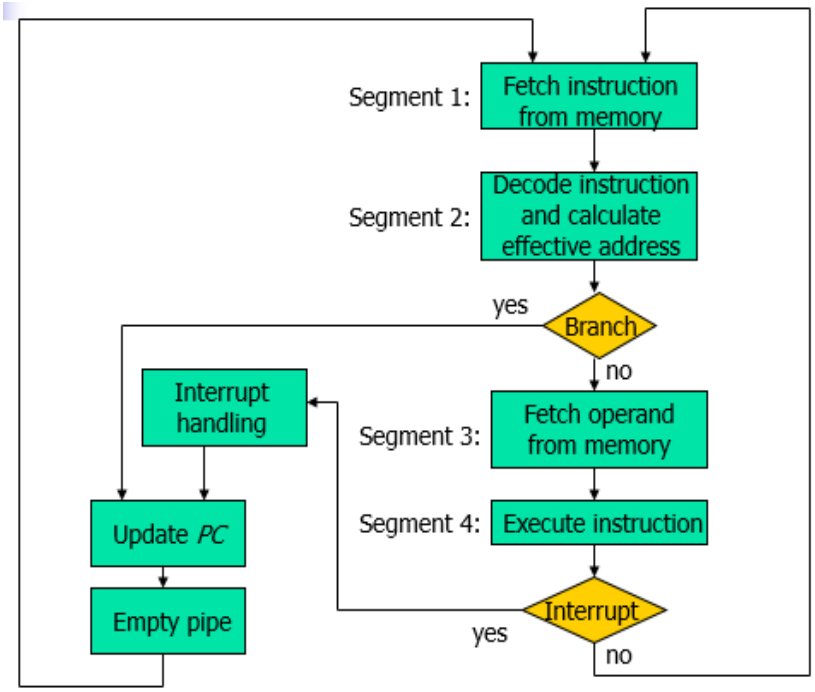


Figure 5.7 - Four-Segment Instruction Pipeline

- An instruction in the sequence may be causes a branch out of normal sequence.
- In that case the pending operations in the last two segments are completed and all information stored in the instruction buffer is deleted.
- Similarly, an interrupt request will cause the pipeline to empty and start again from a new address value. Figure shows the operation of the instruction pipeline
- The time in the horizontal axis is divided into steps of equal duration. The four segments are represented in the diagram with an abbreviated symbol.
  - F1 is the segment that fetches an instruction.
  - DA is the segment that decodes the instruction and calculates the effective address.
  - Fo is the segment that fetches the operand.
  - EX is the segment that executes the instruction.

	Step:	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	FI	DA	FO	EX									
Instruction:	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	—	—	FI	DA	FO	EX			
	5					—	—	—	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

FI: the segment that fetches an instruction
DA: the segment that decodes the instruction and calculate the effective address
FO: the segment that fetches the operand
EX: the segment that executes the instruction

It is assumed that the processor has separate instruction and data memories so that the operation in F1 and PC can proceed at the same time. In the absence of a branch instruction, each segment operates on different instructions. Thus, in step 4, instruction
1 is being executed in segment EX; the operand for instruction 2 is being fetched in segment FO; instruction 3 is being decoded in segment DA; and instruction 4 is being fetched from memory in segment FI.

Assume now that instruction 3 is a branch instruction. As soon as this instruction is decoded in segment DA in step 4, the transfer from F1 to DA of the other instructions is halted until the branch instruction is executed in step6. If the branch is taken, a new instruction is fetched

in step 7. If the branches not taken, the instruction fetched previously in step 4 can be used. The pipeline then continues until a new branch instruction is encountered.
Another delay may occur in the pipeline if the EX segment needs to store the result of the operation in the data memory while the FO segment needs to fetch an operand. In that case, segment FO must wait until segment EX has finished its operation.

- In general, there are three major difficulties that cause the instruction pipeline to deviate from its normal operation.
- *Resource conflicts* caused by access to memory by two segments at the same time.
- Can be resolved by using separate instruction and data memories
- *Data dependency conflicts* arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
- *Branch difficulties* arise from branch and other instructions that change the value of PC.

Data dependency: A difficulty that may cause a degradation of performance in an instruction pipeline is due to possible collision of data or address. A data dependency occurs when an instruction needs data that are not yet available.

- An address dependency may occur when an operand address cannot be calculated because the information needed by the addressing mode is not available. Pipelined computers deal with such conflicts between data dependencies in a variety of ways.
- ***Hardware interlocks*:** an interlock is a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline. This approach maintains the program sequence by using hardware to insert the required delays.
- Operand forwarding: uses special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline segments. This method requires additional hardware paths through multiplexers as well as the circuit that detects the conflict.
- ***Delayed load*:** the *compiler* for such computers is designed to detect a dataconflict and reorder the instructions as necessary to delay the loading of the conflicting data by inserting no-operation instructions.

Handling of branch instructions

- One of the major problems in operating an instruction pipeline is the occurrenceof branch instructions.
    - An unconditional branch always alters the sequential program flow byloading the program counter with the target address.
    - In a conditional branch, the control selects the target instruction if the condition is satisfied or the next sequential instruction if the condition isnot satisfied.

- Pipelined computers employ various hardware techniques to minimize theperformance degradation caused by instruction branching.
- **Prefetch target instruction:** To prefetch the target instruction in addition to theinstruction following the branch. Both are saved until the branch is executed.
- **Branch target buffer (BTB):** The BTB is an associative memory included in thefetch segment of the pipeline.
    - Each entry in the BTB consists of the address of a previously executed branch instruction and the target instruction for that branch.
    - It also stores the next few instructions after the branch target instruction.
- **Loop buffer:** This is a small very high-speed register file maintained by theinstruction fetch segment of the pipeline.
- **Branch prediction:** A pipeline with branch prediction uses some additional logic toguess the outcome of a conditional branch instruction before it is executed.
- **Delayed branch:** in this procedure, the compiler detects the branch instructions and rearranges the machine language code sequence by *inserting useful instructions* that keep the pipeline operating without interruptions.
    - A procedure employed in most RISC processors.
    - e.g. no-operation instruction

5.3.4 RISC PIPELINE

Among the characteristics attributed to RISC is its ability to use an efficient instruction

pipeline. The simplicity of the instruction set can be utilized to implement aninstruction pipeline using a small number of sub operations, with each being executed in oneclock cycle. Because of the fixed-length instruction format, the decoding of the operation canoccur at the same time as the register selection. All data manipulation instructions haveregister-to-register operations. Since all operands are in registers, there is no needfor calculating an effective address or fetching of operands from memory. Therefore, theinstruction pipeline can be implemented with two or three segments. One segment fetchesthe instruction from program memory, and the other segment executes the instruction in theALU. A third segment may be used to store the result of the ALU operation in a destinationregister.

- The data transfer instructions in RISC are limited to load and store instructions.
  - These instructions use register indirect addressing. They usually need three orfour stages in the pipeline.
  - To prevent conflicts between a memory access to fetch an instruction and toload or store an operand, most RISC machines use two separate buses withtwo memories.
  - Cache memory: operate at the same speed as the CPU clock

- One of the major advantages of RISC is its ability to execute instructions at the rateof one per clock cycle.
  - In effect, it is to start each instruction with each clock cycle and to pipelinethe processor to achieve the goal of single-cycle instruction execution.
  - RISC can achieve pipeline segments, requiring just one clock cycle.

- *Compiler* supported that translates the high-level language program into machine language program.
  - Instead of designing hardware to handle the difficulties associated with dataconflicts and branch penalties.
  - RISC processors rely on the efficiency of the compiler to detect and minimizethe delays encountered with these problems.

Example: Three-Segment Instruction Pipeline

- A typical set of instructions for a RISC processor are discussed earlier.

- These are three types of instructions:
  - The data manipulation instructions: operate on data in processor registers
  - The data transfer instructions:
  - The program control instructions:

Now consider the hardware operation for such a computer.

- The *control section* fetches the instruction from program memory into an instructionregister.
- The instruction is decoded at the same time that the registers needed for theexecution of the instruction are selected.
- The processor unit consists of a number of registers and an arithmetic logic unit (ALU).
- A data memory is used to load or store the data from a selected register in the register file.

- The instruction cycle can be divided into three sub operations and implemented in three segments:
- **I: Instruction fetch**
  - Fetches the instruction from program memory
- **A: ALU operation**
  - The instruction is decoded and an ALU operation is performed.
  - It performs an operation for a data manipulation instruction.
  - It evaluates the effective address for a load or store instruction.
  - It calculates the branch address for a program control instruction.
- **E: Execute instruction**
  - Directs the output of the ALU to one of three destinations, depending on the decoded instruction.
  - It transfers the result of the ALU operation into a destination register in the register file.
  - It transfers the effective address to a data memory for loading or storing.
  - It transfers the branch address to the program counter.

5.4 VECTOR PROCESSING

- In many science and engineering applications, the problems can be formulated in terms of vectors and matrices that lend themselves to vector processing.

- Computers with vector processing capabilities are in demand in specialized applications. e.g.
- *Long-range weather forecasting*
- *Petroleum explorations*
- *Seismic data analysis*
- *Medical diagnosis*
- *Artificial intelligence and expert systems*
- *Image processing*
- *Mapping the human genome*
- To achieve the required level of high performance it is necessary to utilize the *fastest and most reliable hardware* and apply innovative procedures from *vector and parallel processing techniques*.
- **Vector Operations**
- Many scientific problems require arithmetic operations on large arrays of numbers.
- A vector is an ordered set of a one-dimensional array of data items.
- A vector V of length n is represented as a row vector by V=[v1,v2,…,Vn].
- To examine the difference between a conventional scalar processor and a vector processor, consider the following Fortran DO loop:

DO 20 I = 1, 100

20 C(I) = B(I) + A(I)

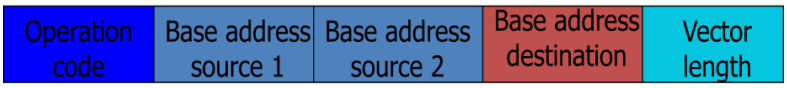
- This is implemented in machine language by the following sequence of operations.

```
      Initialize I = 0
20    Read A(I)
      Read B(I)
      Store C(I) = A(I) + B(I)
      Increment I = I + 1
      If I ≤ 100 go to 20
      Continue
```

- A computer capable of vector processing eliminates the overhead associated with the time it takes to fetch and execute the instructions in the program loop.

C(1:100) = A(1:100) + B(1:100)

- A possible instruction format for a vector instruction is shown in Fig. .
  - This assumes that the vector operands reside in *memory*.



- It is also possible to design the processor with a large number of *registers* and store all operands in registers prior to the addition operation.
  - The base address and length in the vector instruction specify a group of CPU registers.

Matrix Multiplication

- The multiplication of two n x n matrices consists of n2 inner products or n3 multiply-add operations.
- Consider, for example, the multiplication of two 3 x 3 matrices A and B.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

The product matrix C is a 3×3 matrix whose elements are related to the elements of A and B by the inner product:

$$c_{ij} = \sum_{k=1}^{3} a_{ik} \times b_{kj}$$

For example, the number in the first row and first column of matrix C is calculated by letting $i = 1, j = 1$, to obtain

$$c_{11} = a_{11} b_{11} + a_{12} b_{21} + a_{13} b_{31}$$

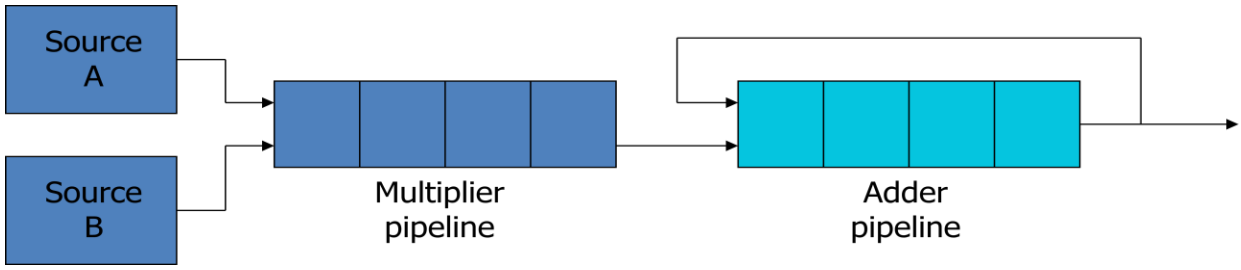
- This requires three multiplication and (after initializing c11 to 0) three additions.
- In general, the inner product consists of the sum of *k* product terms of the form

C = A1B1+A2B2+A3B3+…+AkBk.

- In a typical application k may be equal to 100 or even 1000.

$$\begin{aligned} C = {} & A_1 B_1 + A_5 B_5 + A_9 B_9 + A_{13} B_{13} + \cdots \\ & + A_2 B_2 + A_6 B_6 + A_{10} B_{10} + A_{14} B_{14} + \cdots \\ & + A_3 B_3 + A_7 B_7 + A_{11} B_{11} + A_{15} B_{15} + \cdots \\ & + A_4 B_4 + A_8 B_8 + A_{12} B_{12} + A_{16} B_{16} + \cdots \end{aligned}$$

- **The inner product calculation on a pipeline vector processor is shown in Figure.**



Memory Interleaving

- *Pipeline* and *vector processors* often require simultaneous access to memory from two or more sources.
- An instruction pipeline may require the fetching of an instruction and an operand at the same time from two different segments.
  - An arithmetic pipeline usually requires two or more operands to enter the pipeline at the same time.
  - Instead of using two memory buses for simultaneous access, the memory can be partitioned into a number of modules connected to a common memory address and data buses.
  - A memory module is a memory array together with its own address and data registers.
  - **Figure 5.8** shows a memory unit with four modules.

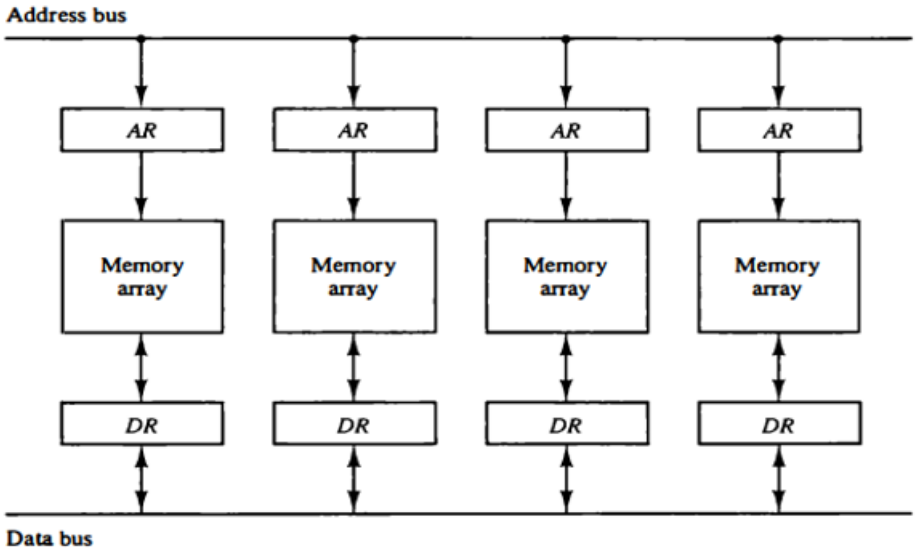


Figure 5.8 - Multiple Module Memory Organization

- The advantage of a modular memory is that it allows the use of a technique called

interleaving.

- In an interleaved memory, different sets of addresses are assigned to different memory modules.
- By staggering the memory access, the effective memory cycle time can be *reduced by a factor close to the number of modules*.

Supercomputers

- A commercial computer with vector instructions and pipelined floating-point arithmetic operations is referred to as a *supercomputer*.
- To speed up the operation, the components are *packed tightly* together to minimize the distance that the electronic signals have to travel.
- This is augmented by instructions that process vectors and combinations of scalars and vectors.
- A supercomputer is a computer system best known for its high computational speed, fast and large memory systems, and the extensive use of parallel processing.
- It is equipped with *multiple functional units* and each unit has its own *pipeline* configuration.
- It is specifically optimized for the type of numerical calculations involving vectors and matrices of floating-point numbers.
- They are limited in their use to a number of scientific applications, such *as numerical weather forecasting, seismic wave analysis, and space research*.
- A measure used to evaluate computers in their ability to perform a given number of floating-point operations per second is referred to as *flops*.
- A typical supercomputer has a basic cycle time of 4 to 20 ns.
- The examples of supercomputer: Cray-1: it uses vector processing with 12 distinct functional units in parallel; a large number of registers (over 150); multiprocessor configuration (Cray X- MP and Cray Y-MP)
- Fujitsu VP-200: 83 vector instructions and 195 scalar instructions; 300 megaflops.

5.4 ARRAY PROCESSORS:

Introduction

- An array processor is a processor that performs computations on large arrays of data.

- The term is used to refer to two different types of processors.
- Attached array processor:
  - Is an auxiliary processor.
  - It is intended to improve the performance of the host computer in specific numerical computation tasks.
- SIMD array processor:
  - Has a single-instruction multiple-data organization.
  - It manipulates vector instructions by means of multiple functional units responding to a common instruction.

Attached Array Processor

- Its purpose is to enhance the performance of the computer by providing vector processing for complex scientific applications.
- Parallel processing with multiple functional units
- **Figure 5.9** shows the interconnection of an attached array processor to a host computer.
- The host computer is a general-purpose commercial computer and the attached processor is a back-end machine driven by the host computer. The array processor is connected through an input-output controller to the computer and the computer treats it like an external interface.
- The data for the attached processor are transferred from main memory to a local memory through a high-speed bus. The general-purpose computer without the attached processor serves the users that need conventional data processing. The system with the attached processor satisfies the needs for complex arithmetic applications.

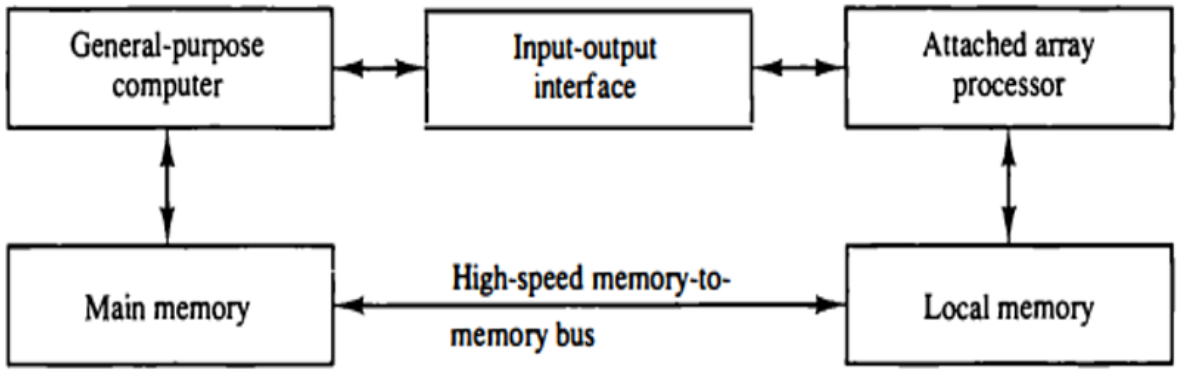


Figure 5.9 - Attached Array Processor with Host Computer

- For example, when attached to a VAX 11 computer, the FSP-164/MAX from Floating- Point

Systems increases the computing power of the VAX to 100megaflops.

- The objective of the attached array processor is to provide *vector manipulation capabilities* to a conventional computer at a fraction of the cost of supercomputer.

SIMD Array Processor

- A SIMD array processor is a computer with multiple processing units operating in parallel.
- A general block diagram of an array processor is shown in Figure 5.10.
- It contains a set of identical processing elements (PEs), each having a local memory *M*.
- Each PE includes an ALU, a floating-point arithmetic unit, and working registers.
- Vector instructions are broadcast to all PEs simultaneously.
- Masking schemes are used to control the status of each PE during the execution of vector instructions.
- Each PE has a flag that is set when the PE is active and reset when the PE is inactive.
- For example, the ILLIAC IV computer developed at the University of Illinois and manufactured by the Burroughs Compare highly specialized computers. They are suited primarily for numerical problems that can be expressed invector or matrix form.

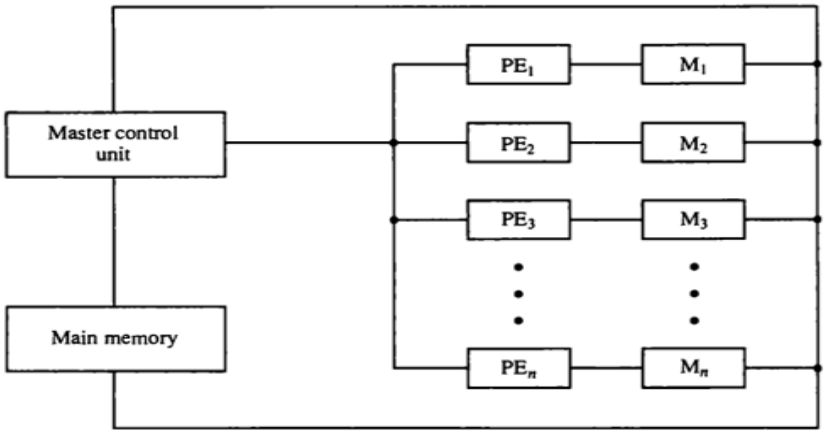


Figure 5.10 - SIMD Array Processor Organization

5.5 MULTI PROCESSORS:

5.5.1 CHARACTERISTICS OF MULTIPROCESSOR:

- A multiprocessor system is an interconnection of two or more CPUs with memory and input-output equipment. The term "processor" In multiprocessor can mean either a central

processing unit (CPU) or an input-output processor (IOP).

- Multiprocessors are classified as multiple instruction stream, multiple data stream (MIMO) systems.
- Computers are interconnected with each other by means of communication lines to form a computer network. The network consists of several autonomous computers that may or may not communicate with each other.
- A multiprocessor system is controlled by one operating system that provides interaction between processors and all the components of the system cooperate in the solution of a problem.
- Multiprocessing improves the reliability of the system so that a failure or error in one part has a limited effect on the rest of the system. If a fault causes one processor to fail, a second processor can be assigned to perform the functions of the disabled processor.
- Multiprocessing can improve performance by decomposing a program into parallel executable tasks. This can be achieved in one of two ways.
- The user can explicitly declare that certain tasks of the program be executed in parallel.
- The other is a compiler with multiprocessor software that can automatically detect parallelism in a user's program.
- Multiprocessors are classified by the way their memory is organized.
- A multiprocessor system with common shared memory is classified as a shared memory or tightly coupled multiprocessor. This does not preclude each processor from having its own local memory. In fact, most commercial tightly coupled multiprocessors provide a cache memory with each CPU.
- An alternative model of microprocessor is the distributed-memory or loosely coupled system. Each processor element in a loosely coupled system has its own private local memory. The processors are tied together by a switching scheme designed to route information from one processor to another through a message-passing scheme.
- Loosely coupled systems are most efficient when the interaction between tasks is minimal, whereas tightly coupled systems can tolerate a higher degree of interaction between tasks.

5.6 INTERCONNECTION STRUCTURES:

- The components that form a multiprocessor system are CPUs, IOPs connected to input-

output devices, and a memory unit that may be partitioned into a number of separate modules.

- The interconnection between the components can have different physical configurations, depending on the number of transfer paths and memory in a shared memory system.

There are several physical forms available for establishing an interconnection network. Some of these schemes are presented in this section:

1. Time-shared common bus
2. Multiport memory
3. Crossbar switch
4. Multistage switching network
5. Hypercube system

Time-Shared Common Bus: A common-bus multiprocessor system consists of a number of processors connected through a common path to a memory unit. A time-shared common bus for five processors is shown in **Figure 5.11**. Only one processor can communicate with the memory or another processor at any given time.

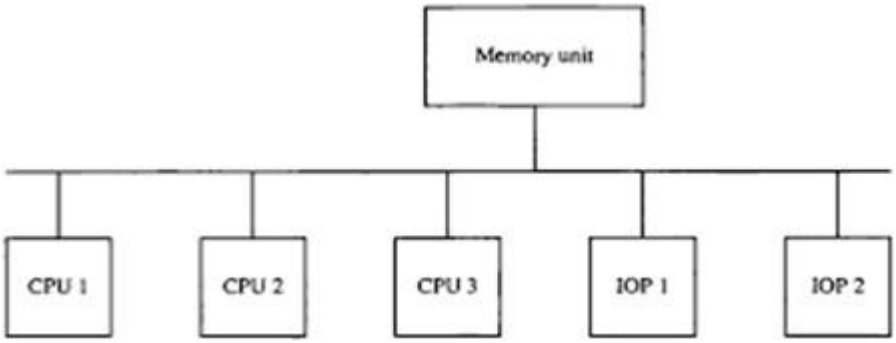


Figure 5.11: Time Shared Common Bus Organization

- Transfer operations are conducted by the processor that is in control of the bus at the time.
- If any other processor wishing to initiate a transfer must first determine the availability status of the bus, and only after the bus becomes available can the processor address the destination unit to initiate the transfer.
- A command is issued to inform the destination unit what operation is to be performed.
- The receiving unit recognizes its address in the bus and responds to the control signals from the sender, after which the transfer is initiated.
- The system may exhibit transfer conflicts since one common bus is shared by all processors.

These conflicts must be resolved by incorporating a bus controller that establishes priorities among the requesting units.

- A single common-bus system is restricted to one transfer at a time. This means that when one processor is communicating with the memory, all other processors are either busy with internal operations or must be idle waiting for the bus. However, this increases the system cost and complexity.

A more economical implementation of a dual bus structure is depicted in Figure.

- In this method a number of local buses each connected to its own local memory and to one or more processors.
- Each local bus may be connected to a CPU, an lOP, or any combination of processors. A system bus controller links each local bus to a common system bus.
- The I/O devices connected to the local lOP, as well as the local memory, are available to the local processor.
- The memory connected to the common system bus is shared by all processors. Only one processor can communicate with the shared memory and other common resources through the system bus at any given time.
- The other processors are kept busy communicating with their local memory and VO devices. Part of the local memory may be designed as a cache memory attached to the CPU.
- In this way, inconsistent versions the average access time of the local memory can be made to approach the cycle time of the CPU to which it is attached.

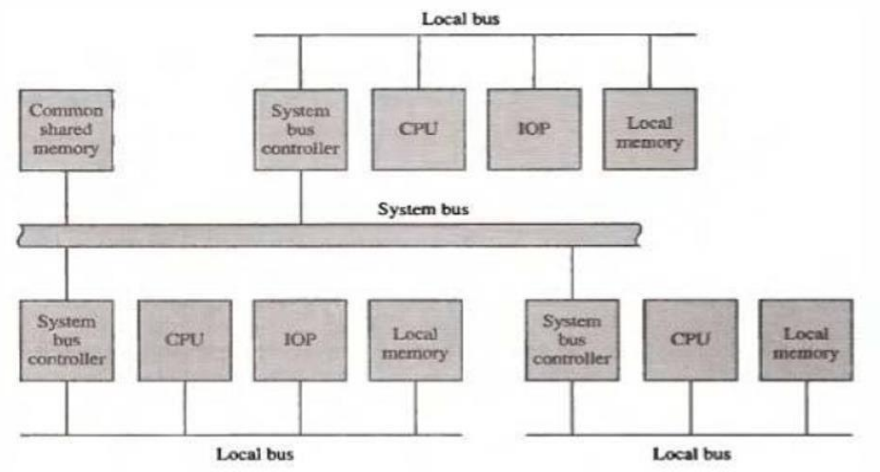


Figure 5.12: System Bus Structure for Multiprocessors

Multiport Memory

- A multiport memory system employs separate buses between each memory module and each CPU. This is shown in Fig .3 for four CPUs and four memory modules (MMs). Each processor bus is connected to each memory module.
- A processor bus consists of the address, data, and control lines required to communicate with memory. l
- The module must have internal control logic to determine which port will have access to memory at any given time. Memory access conflicts are resolved by assigning fixed priorities to each memory port.
- The priority for memory access associated with each processor may be established by the physical port position that its bus occupies in each module. Thus CPU 1 will have priority over CPU 2, CPU 2 will have priority over CPU 3, and CPU 4 will have the lowest priority.
- The advantage of the multiport memory organization is the high transfer rate that can be achieved because of the multiple paths between processors and memory.
- The disadvantage is that it requires expensive memory control logic and a large number of cables and connectors.

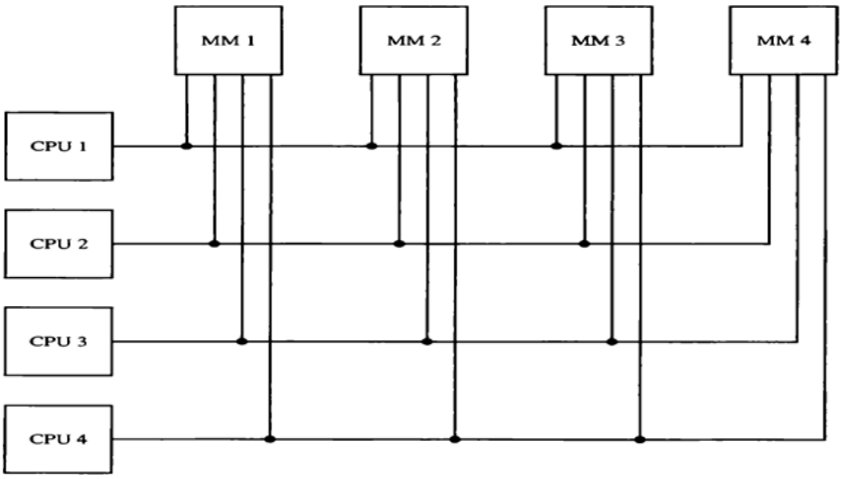


Figure 5.13 - Multiport Memory Organization

Crossbar Switch

- The crossbar switch organization consists of a number of cross points that are placed at intersections between processor buses and memory module paths.
- Figure 4 shows a crossbar switch interconnection between four CPUs and four memory

modules.

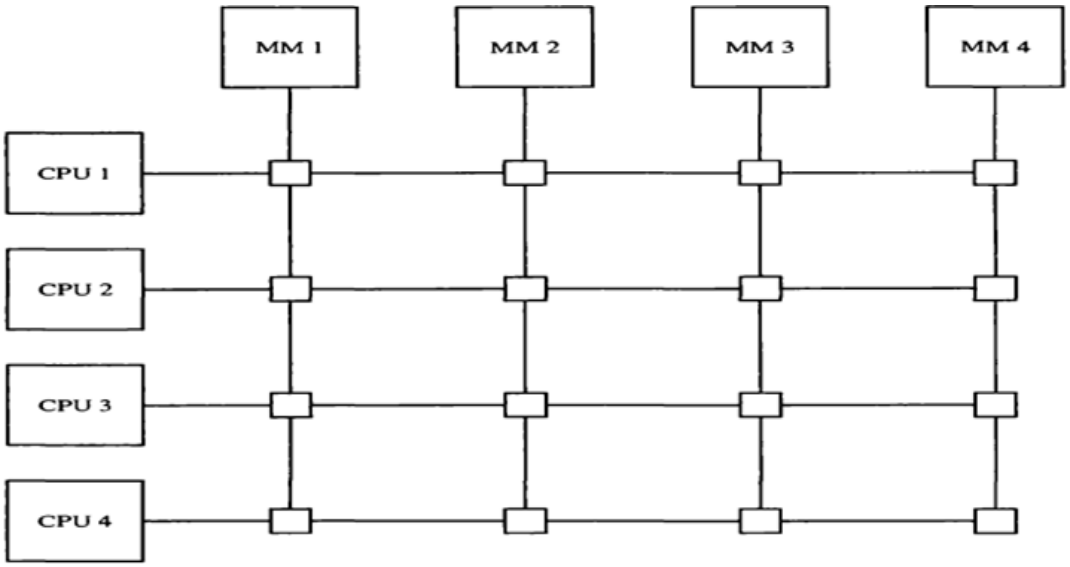


Figure 5.14 - Crossbar Switch Interconnection Between Four CPU

- The small square in each cross point is a switch that determines the path from a processor to a memory module.
- Each switch point has control logic to set up the transfer path between a processor and memory. It examines the address that is placed in the bus to determine whether its particular module is being addressed.
- It also resolves multiple requests for access to the same memory module on a predetermined priority basis.
- **Figure 5.15** shows the functional design of a crossbar switch connected to one memory module.

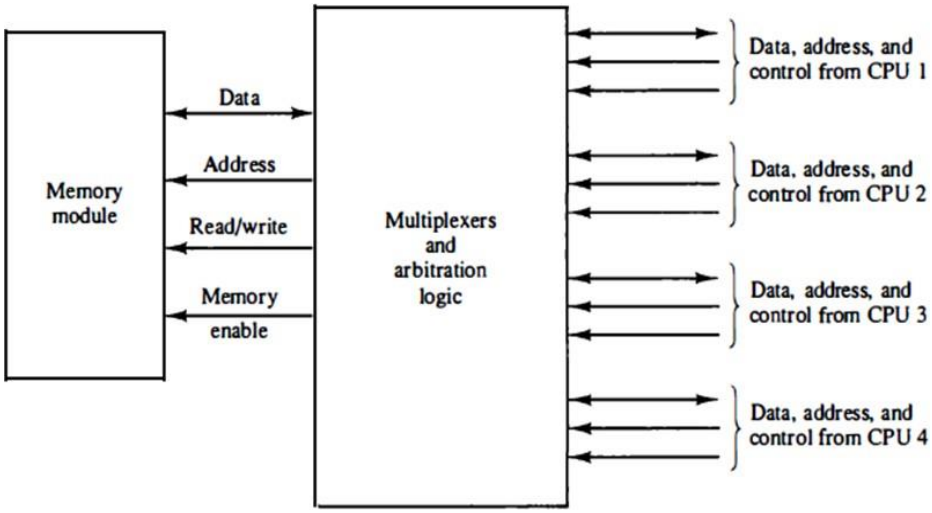


Figure 5.15: Functional Design of a Crossbar Switch

- The circuit consists of multiplexers that select the data, address, and control from one CPU for communication with the memory module.
- Priority levels are established by the arbitration logic to select one CPU when two or more CPUs attempt to access the same memory.
- The multiplexers are controlled with the binary code that is generated by a priority encoder within the arbitration logic.
- A crossbar switch organization supports simultaneous transfers from memory modules because there is a separate path associated with each module.
- The hardware required to implement the switch can become quite large and complex.

Multistage Switching Network:

- The basic component of a multistage network is a two-input, two-output interchange switch.
- As shown in **Figure 5.16**. the 2 X 2 switch has two inputs labeled A and B, and two outputs, labeled 0 and 1.
- There are control signs (not shown) associated with the switch that establish the interconnection between the input and output terminals.

The switch has the capability connecting input A to either of the outputs. Terminal B of the switch behave in a similar fashion.

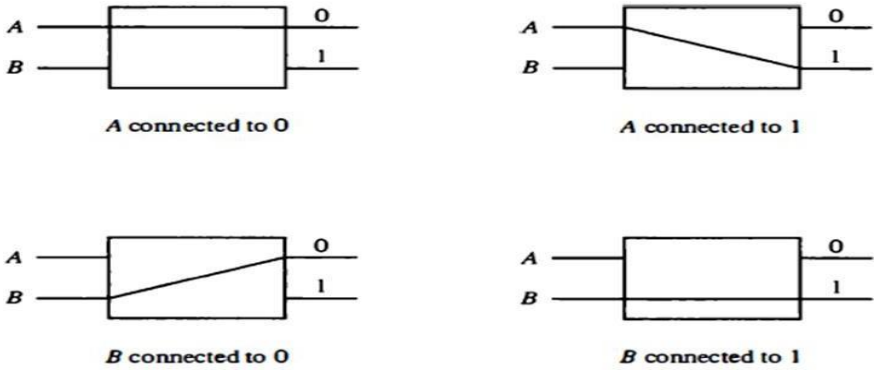


Figure 5.16: The 2 X 2 Switch has Two Inputs Labeled A and B

- The switch also has the capability to arbitrate between conflicting requests. If inputs A and B both request the same output terminal only one of them will be connected; the other will be blocked.

- Using the 2 x 2 switch as a building block, it is possible to build multistage network to control the communication between a number of souroce and destinations. To see how this is done, consider the binary tree shown in **Figure 5.17**.

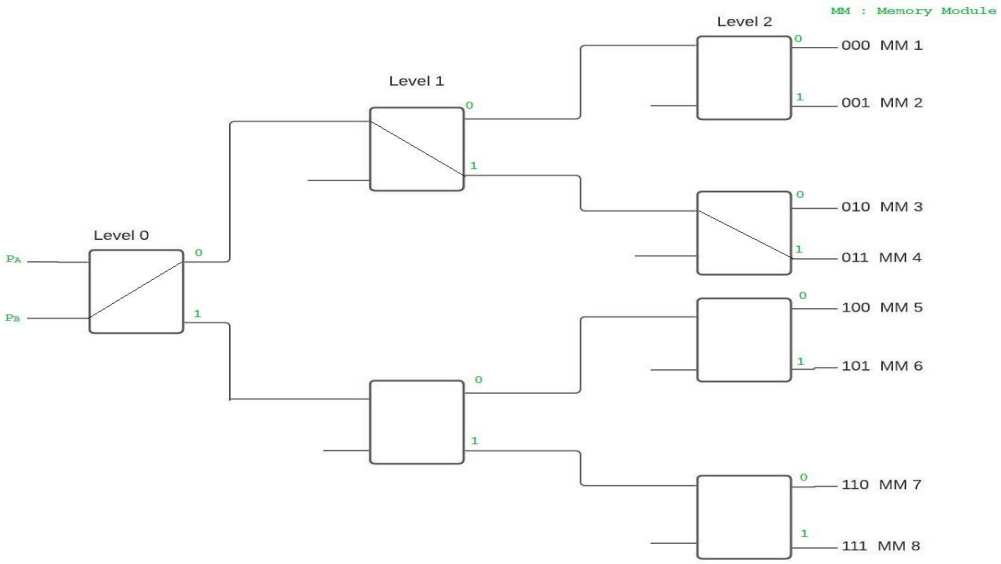


Fig 5.17: Binary Tree with 2 X 2 Switches

- The two processors P1 and P2 are connected through switches to eight memory modules marked in binary from 000 through 111. The path from source to a destination is determined from the binary bits of the destination number.
- The first bit of the destination number determines the switch output in the first level. The second bit specifies the output of the switch in the second level, and the third bit specifiesthe output of the switch in the third level.
- For example, to connect P1 to memory 101, it is necessary to form a path from P1 to output1 in the first-level switch, output 0 in the second-level switch, and output 1 in the third- level switch.

Omega switching network: In this configuration, there is exactly one path from each source to any particular destination. For example, any two sources cannot be connected simultaneously to destinations 000 and 001.

- A particular request is initiated in the switching network by the source, which sends a 3- bit pattern representing the destination number.
- As the binary pattern moves through the network, each level examines a different bit to determine the 2 x 2 switch setting. Level 1 inspects the most significant bit, level 2 inspects the

middle bit, and level 3 inspects the least significant bit.

- When the request arrives on either Input of the 2 x 2 switch, it is routed to the upper output if the specified bit is 0 or to the lower output if the bit is 1.

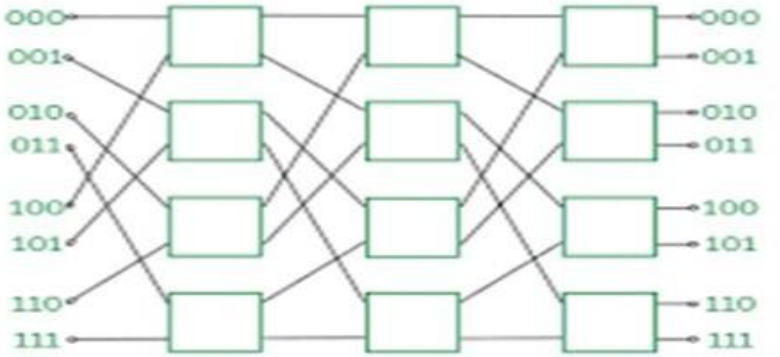


Figure 5.18 - 8 * 8 Omega Switching Network

- A particular request is initiated in the switching network by the source, which sends a 3- bit pattern representing the destination number.
- As the binary pattern moves through the network, each level examines a different bit to determine the 2 x 2 switch setting. Level 1 inspects the most significant bit, level 2 inspects the middle bit, and level 3 inspects the least significant bit.
- When the request arrives on either Input of the 2 x 2 switch, it is routed to the upper output if the specified bit is 0 or to the lower output if the bit is 1.

Hypercube Interconnection:

- Structure represents a loosely coupled system made up of $\mathbf{N=2^n}$ processors interconnected in an n-dimensional binary cube.
- Each processor makes a node of the cube. Therefore, it is customary to refer to each node as containing a processor, in effect it contains not only a CPU but also local memory and I/O interface.
- Each processor has direct communication paths to n other neighbor processors. These paths correspond to the cube edges.
- There are 2 distinct n-bit binary addresses which can be assigned to the processors. Each processor address differs from that of each of its n neighbors by exactly one bit position.
- Hypercube structure for n= 1, 2 and 3.

- A one cube structure contains n = 1 and $2^n = 2$. It has two processors interconnected by a single path.
- A two-cube structure contains n=2 and $2^n=4$. It has four nodes interconnected as a cube.
- An n-cube structure contains 2n nodes with a processor residing in each node.
- Each node is assigned a binary address in such a way that the addresses of two neighbors differ in exactly one bit position.

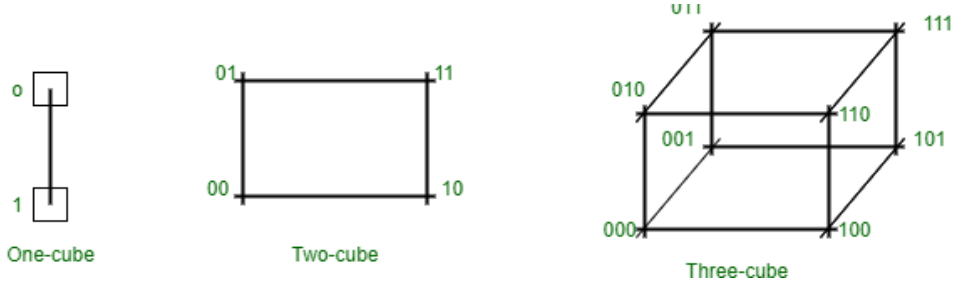


Figure 5.18 - Hypercube Structures for n=1,2,3

5.7 INTERPROCESSOR ARBITRATION:

- The CPU contains a number of internal buses for transferring information between processor registers and ALU.
- A memory bus consists of lines for transferring data, address, and read/write information.
- An I/O bus is used to transfer information to and from input and output devices. A bus that connects major components in a multisystem bus processor system, such as CPUs, IOPs, and memory, is called a system bus.
- The processors in a shared memory multiprocessor system request access to common memory or other common resources through the system bus.
- If no other processor is currently utilizing the bus, the requesting processor may be granted access immediately. However, the requesting processor must wait if another processor is currently utilizing the system bus. Furthermore, other processors may request the system bus at the same time.
- Arbitration must then be performed to resolve this multiple contention for the shared resources.
- The arbitration logic would be part of the system bus controller placed between the local bus and the system bus.

System Bus A typical system bus consists of approximately 100 signal lines. These lines are divided into three functional groups: data, address, and control. In addition, there are power distribution lines that supply power to the components.

- The data lines provide a path for the transfer of data between processors and common memory. The number of data lines is usually a multiple of 8, with 16 and 32 being most common.
- The address lines are used to identify a memory address or any other source or destination, such as input or output ports. The number of address lines determines the maximum possible memory capacity in the system.
- For example, an address of 24 lines can access up to 224 (16 mega) words of memory.
- Data transfers over the system bus may be synchronous or asynchronous.
- In a synchronous bus, each data item is transferred during a time slice known in advance to both source and destination units.
- In an asynchronous bus, each data item being transferred is accompanied by handshaking control signals to indicate when the data are transferred from the source and received by the destination.
- The control lines provide signals for controlling the information transfer between units. Timing signals indicate the validity of data and address information. Command signals specify operations to be performed.
- Typical control lines include transfer signals such as memory read and write, acknowledge of a transfer, interrupt requests, bus control signals such as bus request and bus grant, and signals for arbitration procedures.
- Table 13-1 lists the 86 lines that are available in the IEEE standard 796 multibus. It includes16 data lines and 24 address lines.

IEEE Standard 796 Multibus Signals

	Signal name
Data and address	
Data lines (16 lines)	DATA0–DATA15
Address lines (24 lines)	ADRS0–ADRS23
Data transfer	
Memory read	MRDC
Memory write	MWTC
IO read	IORC
IO write	IOWC
Transfer acknowledge	TACK
Interrupt control	
Interrupt request (8 lines)	INT0–INT7
Interrupt acknowledge	INTA
Miscellaneous control	
Master clock	CCLK
System initialization	INIT
Byte high enable	BHEN
Memory inhibit (2 lines)	INH1–INH2
Bus lock	LOCK
Bus arbitration	
Bus request	BREQ
Common bus request	CBRQ
Bus busy	BUSY
Bus clock	BCLK
Bus priority in	BPRN
Bus priority out	BPRO
Power and ground (20 lines)	

Serial Arbitration Procedure:

Arbitration procedures service all processor requests on the basis of established priorities.

- A hardware bus priority resolving technique can be established by means of a serial or parallel connection of the units requesting control of the system bus.
- The serial priority resolving technique is obtained from a daisy-chain connection of bus arbitration circuits.
- The processors connected to the system bus are assigned priority according to their position along the priority control line.
- The device closest to the priority line is assigned the highest priority. When multiple devices concurrently request the use of the bus, the device with the highest priority is granted access to it.

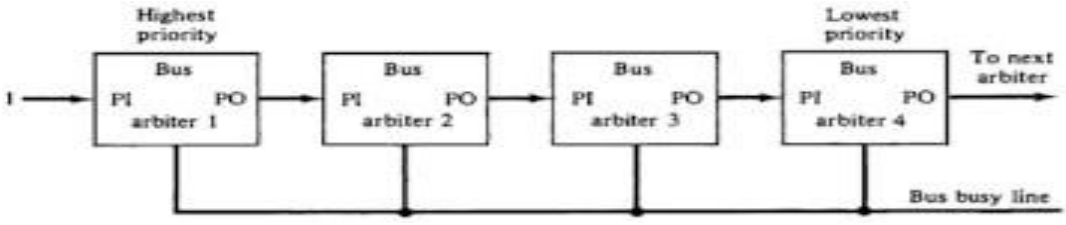


Figure 5.18 - The Daisy-Chain Connection of Four Arbiter

Figure shows the daisy-chain connection of four arbiters. It is assumed that each processor has its own bus arbiter logic with priority-in and priority-out lines.

- The priority out (PO) of each arbiter is connected to the priority in (PI) of the next-lower-priority arbiter. The PI of the highest priority unit is maintained at logic 1 value.
- The highest-priority unit in the system will always receive access to the system bus when it requests it. The Po output for a particular arbiter is equal to 1 if its PI input is equal to 1 and the processor associated with the arbiter logic is not requesting control of the bus.
- This is the way that priority is passed to the next unit in the chain. It the processor requests control of the bus and the corresponding arbiter finds its PI input equal to 1, it sets its PO output to 0.
- Lower priority arbiters receive a 0 in PI and generate a 0 in Po. Thus, the processor whose arbiter has a PI = 1 and Po = 0 is the one that is given control of the system bus.
- A processor may be in the middle of a bus operation when a higher-priority processor requests the bus. The lower-priority processor must complete its bus operation before it relinquishes control of the bus.
- The busy line comes from open-collector circuits in each unit and provides a wired-OR logic connection.
- When an arbiter receives control of the bus (because its PI = 1 and Po = 0) it examines the busy line. If the line is inactive, it means that no other processor is using the bus.
- The arbiter activates the busy line and its processor takes control of the bus. However, if the arbiter finds the busy line active, it means that another processor is currently using the bus.
- The arbiter keeps examining the busy line while the lower-priority processor that lost control of the bus completes its operation.
- When the bus busy line returns to its inactive state, the higher- priority arbiter enables the busy line, and its corresponding processor can then conduct the required bus transfers.

Parallel Arbitration Logic:

The parallel bus arbitration technique uses an external priority encoder and a decoder as shown in **Figure 5.19.**

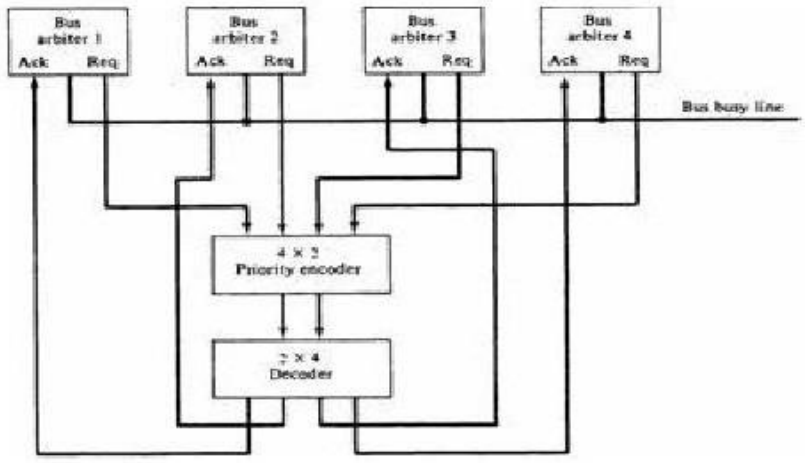


Figure 5.19 - Parallel Arbitration

- Each bus arbiter in the parallel scheme has a bus request output line and a bus acknowledge input line.
- Each arbiter enables the request line when its processor is requesting access to the system bus. The processor takes control of the bus if it acknowledges input line is enabled.
- The bus busy line provides an orderly transfer of control, as in the daisy-chaining case.
- Figure 11 shows the request lines from four arbiters going into a 4 X 2 priority encoder.
- The output of the encoder generates a 2-bit code which represents the highest- priority unit among those requesting the bus.
- The 2-bit code from the encoder output drives a 2 X 4 decoder which enables the proper acknowledge line to grant bus access to the highest-priority unit.

Dynamic Arbitration Algorithms

- The two bus arbitration procedures just described use a static priority algorithm since the priority of each device is fixed by the way it is connected to the bus.
- In contrast, a dynamic priority algorithm gives the system the capability for changing the priority of the devices while the system is in operation.
- **Time slice**: The time slice algorithm allocates a fixed-length time slice of bus time that is offered sequentially to each processor, in round-robin fashion.

- The service given to each system component with this scheme is independent of its location along the bus. No preference is given to any particular device since each is allotted the same amount of time to communicate with the bus.
- **Polling**: In a bus system that uses polling, the bus grant signal is replaced by a set of lines called poll lines which are connected to all units. These lines are used by the bus controller to define an address for each device connected to the bus. The bus controller sequences through the addresses in a prescribed manner. When a processor that requires access recognizes its address, it activates the bus busy line and then accesses the bus. After a number of bus cycles, the polling process continues by choosing a different processor. The polling sequence is normally programmable, and as a result, the selection priority can be altered under program control.
- **LRU**: The least recently used (LRU) algorithm gives the highest priority to the requesting device that has not used the bus for the longest interval. The priorities are adjusted after a number of bus cycles according to the LRU algorithm.
- **FIFO**: In the first-come, first-serve scheme, requests are served in the order received. To implement this algorithm, the bus controller establishes a queue arranged according to the time that the bus requests arrive. Each processor must wait for its turn to use the bus on a first-in, first-out (FIFO) basis.
- **Rotating Daisy-Chain** The rotating daisy-chain procedure is a dynamic extension of the daisy-chain algorithm. In this scheme there is no central bus controller, and the priority line is connected from the priority-out of the last device back to the priority-in of the first device in a closed loop. Whichever device has access to the bus serves as a bus controller for the following arbitration. Each arbiter priority for a given bus cycle is determined by its position along the bus priority line from the arbiter whose processor is currently controlling the bus. Once an arbiter releases the bus, it has the lowest priority.

5.8 INTERPROCESSOR COMMUNICATION AND SYNCHRONIZATION:

- The various processors in a multiprocessor system must be provided with a facility for communicating with each other. A communication path can be established through common input output channels.
- In a shared memory multiprocessor system, the most common procedure is to setaside a

portion of memory that is accessible to all processors.

- The primary use of the common memory is to act as a message center similar toa mailbox, where each processor can leave messages for other processors and pick up messages intended for it.
- The sending processor structures a request, a message, or a procedure, and placesit in the memory mailbox.
- The receiving processor can check the mailbox periodically to determine if thereare valid messages for it.
- A more efficient procedure is for the sending processor to alert the receiving processor directly by means of an interrupt signal.
- This can be accomplished through a software-initiated interprocessor interrupt by means of an instruction in the program of one processor which when executedproduces an external interrupt condition in a second processor.
- To prevent conflicting use of shared resources by several processors there must be a provision for assigning resources to processors. This task is given to the operating system.
- There are three organizations that have been used in the design of operating system for multiprocessors: **master-slave configuration, separate operating system, and distributed operating system**. In a master-slave mode, one processor, designated the master, always executes the operating system functions. The remaining processors, denoted as slaves, do not perform operating system functions.
- If a slave processor needs an operating system service, it must request it by interrupting the master and waiting until the current program can be interrupted.
- In the separate operating system organization, each processor can execute the operating system routines it needs.
- In the distributed operating system organization, the operating system routines are distributed among the available processors. However, each particular operating system function is assigned to only one processor at a time.
- In a loosely coupled multiprocessor system the memory is distributed among theprocessors and there is no shared memory for passing information.
- The communication between processors is by means of message passing through 1/0 channels.

- The communication is initiated by one processor calling a procedure that residesin the memory of the processor with which it wishes to communicate.
- When the sending processor and receiving processor name each other as a source and destination, a channel of communication is established.
- A message is then sent with a header and various data objects used to communicate between nodes.
- The operating system in each node contains routing information indicating the alternative paths that can be used to send a message to other nodes.
- The communication efficiency of the interprocessor network depends on the communication routing protocol, processor speed, data link speed, and the topology of the network.

Inter processor Synchronization:

- The instruction set of a multiprocessor contains basic instructions that are used to implement communication and synchronization between cooperating processes. Communication refers to the exchange of data between different processes.
- Synchronization refers to the special case where the data used to communicate between processors is control information.
- Synchronization is needed to enforce the correct sequence of processes and to ensure mutually exclusive access to shared writable data.
- Multiprocessor systems usually include various mechanisms to deal with the synchronization of resources. Low-level primitives are implemented directly by the hardware.
- A number of hardware mechanisms for mutual exclusion have been developed. One of the most popular methods is through the use of a binary semaphore.

Mutual Exclusion with a Semaphore:

- To protect data from being changed simultaneously by two or more processors. This mechanism has been termed mutual exclusion.
- Mutual exclusion must be provided in a multiprocessor system to enable one processor to exclude or lock out access to a shared resource by other processors when critical section it

is in a critical section.

- A critical section is a program sequence that, once begun, must complete execution before another processor accesses the same shared resource.
- A binary variable called a semaphore is often used to indicate whether or not a processor is executing a critical section.
- A semaphore is a software- controlled flag that is stored in a memory location that all processors can access.
- When the semaphore is equal to1, it means that a processor is executing a critical program, so that the shared memory is not available to other processors.
- When the semaphore is equal to 0, the shared memory is available to any requesting processor. Processors that share the same memory segment agree by convention not to use the memory segment unless the semaphore is equal to 0, indicating that memory is available.
- They also agree to set the semaphore to 1 when they are executing a critical section and to clear it to 0 when they are finished.
- Testing and setting the semaphore is itself a critical operation and must be performed as a single indivisible operation.
- A semaphore can be initialized by means of a test and set instruction in hardware lock conjunction with a hardware lock mechanism.
- A hardware lock is a processor-generated signal that serves to prevent other processors from using the system bus as long as the signal is active.
- The test and-set instruction tests and sets a semaphore and activates the lock mechanism during the time that the instruction is being executed.
- This prevents other processors from changing the semaphore between the time that the processor is testing it and the time that it is setting it.
- Assume that the semaphore is a bit in the least significant position of a memory word whose address is symbolized by SEM.
- Let the mnemonic TSL designate the "test and set while locked" operation. The instruction

TSL SEM

will be executed in two memory cycles (the first to read and the second to write) without interference as follows:

R←M [SEM] Test semaphore

M[SEM]←1 Set semaphore

- The semaphore is tested by transferring its value to a processor register R and then it is set to 1.
- The value in R determines what to do next. If the processor finds that R = 1, it knows that the semaphore was originally set. That means that another processor is executing a critical section, so the processor that checked the semaphore does not access the shared memory.
- If R = 0, it means that the common memory is available. The semaphore is set to 1 to prevent other processors from accessing memory.
- The processor can now execute the critical section. The last instruction in the program must clear location SEM to zero to release the shared resource to otherprocessors.
- The lock signal must be active during the execution of the test and set instruction.

5.9 CACHE COHERENCE:

- The primary advantage of cache is its ability to reduce the average access time in uniprocessors. When the processor finds a word in cache during a read operation, the main memory is not involved in the transfer.
- If the operation is to write, there are two commonly used procedures to update memory.
- In the **write-through policy**, both cache and main memory are updated with every write operation.
- In the **write-back policy**, only the cache is updated, and the location is marked so that it can be copied later into main memory.

Conditions for Incoherence:

- Cache coherence problems exist in multiprocessors with private caches because of the need to share writable data.
- Read-only data can safely be replicated without cache coherence enforcement mechanisms. Consider the three-processor configuration with private caches shown in **Figure 5.20**.
- Sometime during the operation an element X from main memory is loaded into the three processors, P1, P2, and P3. As a consequence, it is also copied into the private caches of the three processors.

- For example we assume that X contains the value of 52. The load on X to the three processors results in consistent copies in the caches and main memory.

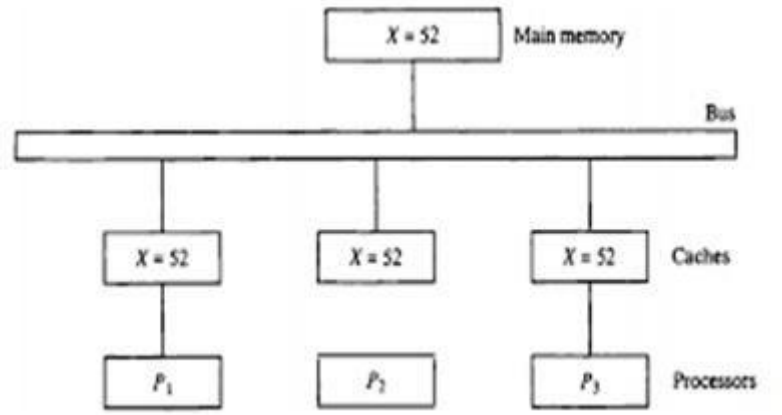


Figure 5.20: Cache Configuration after a Load on

- If one of the processors performs a store to X, the copies of X in the caches become inconsistent. A load by the other processors will not return the latest value.
- Depending on the memory update policy used in the cache, the main memory may also be inconsistent with respect to the cache. This is shown in Fig.13. A store to X (of the value of 120) into the cache of processor P1 updates memory to the new value in a write-through policy.
- A write-through policy maintains consistency between memory and the originating cache, but the other two caches are inconsistent since they still hold the old value.
- In a write-back policy, main memory is not updated at the time of the store. Thecopies in the other two caches and main memory are inconsistent.
- Another configuration that may cause consistency problems is a direct memory access (DMA) activity in conjunction with an IOP connected to the system bus.
- In the case of input, the DMA may modify locations in main memory that also reside in cache without updating the cache.
- During a DMA output, memory locations may be read before they are updated from the cache when using a write-back policy.
- I/O-based memory incoherence can be overcome by making the IOP a participant in the cache coherent solution that is adopted in the system.

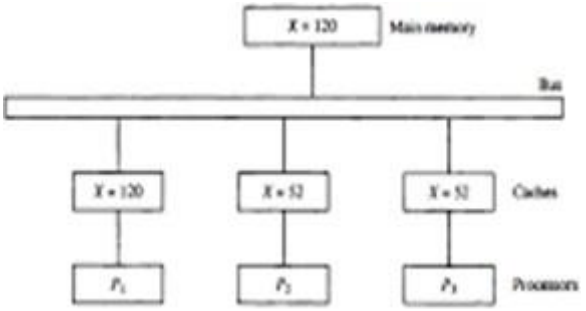


Figure 5.21(a) - With Write Through Cache Policy

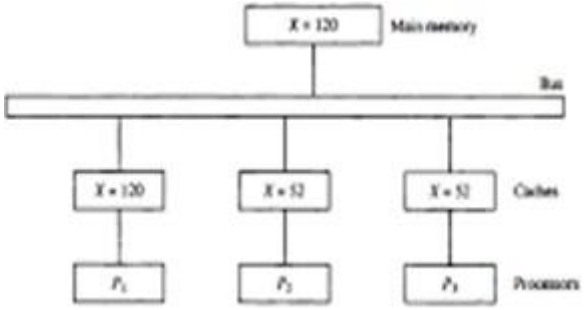


Figure 5.21(b) - With Write Back Cache Policy

Solutions to the Cache Coherence Problem

- A simple scheme is to disallow private caches for each processor and have a shared cache memory associated with main memory. Every data access is made to the shared cache. In effect, this scheme solves the problem by avoiding it.
- For performance considerations it is desirable to attach a private cache to each processor. One scheme that has been used allows only nonshared and read-only data to be stored in caches. Such items are called cachable. Shared writable data are non cachable.
- The compiler must tag data as either cachable or noncachable, and the system hardware makes sure that only cachable data are stored in caches. The noncachable data remain in main memory.
- A scheme that allows writable data to exist in at least one cache is a method that employs a centralized global table in its compiler. The status of memory blocks is stored in the central global table.
- Each block is identified as read-only (RO) or read and write (RW). All caches can have copies of blocks identified as RO. Only one cache can have a copy of an RW block.
- Thus, if the data are updated in the cache with an RW block, the other caches are not affected because they do not have a copy of this block.

- The cache coherence problem can be solved by means of a combination of software and hardware or by means of hardware-only schemes.
- The two methods mentioned previously use software-based procedures that require the ability to tag information to disable caching of shared writable data.
- Hardware-only solutions are handled by the hardware automatically and have the advantage of higher speed and program transparency.
- In the hardware solution, the cache controller is specially designed to allow it to monitor all bus requests from CPUs and IOPs. All caches attached to the bus constantly monitor the network for possible write operations.
- Depending on the method used, they must then either update or invalidate their own cache copies when a match is detected. The bus controller that monitors this action is referred to snoopy cache as a snoopy cache controller.
- This is basically a hardware unit designed to maintain a bus-watching mechanism over all the caches attached to the bus.
- All the snoopy controllers watch the bus for memory store operations.
- When a word in a cache is updated by writing into it, the corresponding location in main memory is also updated.
- The local snoopy controllers in all other caches check their memory to determine if they have a copy of the word that has been overwritten.
- If a copy exists in a remote cache, that location is marked invalid. Because all caches snoop on all bus writes, whenever a word is written, the net effect is to update it in the original cache and main memory and remove it from all other caches.
- In another variant of the snoopy cache coherence protocol. whenever a processor writes to a block in a write through scheme, the cache controllers at all processor s match the address you check if they have a copy of the block.
- If they have copy of the block, then they update the local cache.