CHAPTER – 1
BASIC STRUCTURE OF COMPUTERS

Computer types: -

A computer can be defined as a fast electronic calculating machine that accepts the (data) digitized input information process it as per the list of internally stored instructions and produces the resulting information.

List of instructions are called programs & internal storage is called computer memory.

The different types of computers are

1. **Personal computers: -** This is the most common type found in homes, schools, Business offices etc., It is the most common type of desk top computers with processing and storage units along with various input and output devices.
2. **Note book computers: -** These are compact and portable versions of PC
3. **Work stations: -** These have high resolution input/output (I/O) graphics capability, but with same dimensions as that of desktop computer. These are used in engineering applications of interactive design work.
4. **Enterprise systems: -** These are used for business data processing in medium to large corporations that require much more computing power and storage capacity than work stations. Internet associated with servers have become a dominant worldwide source of all types of information.
5. **Super computers: -** These are used for large scale numerical calculations required in the applications like weather forecasting etc.,

Functional unit: -

A computer consists of five functionally independent main parts input, memory, arithmetic logic unit (ALU), output and control unit.

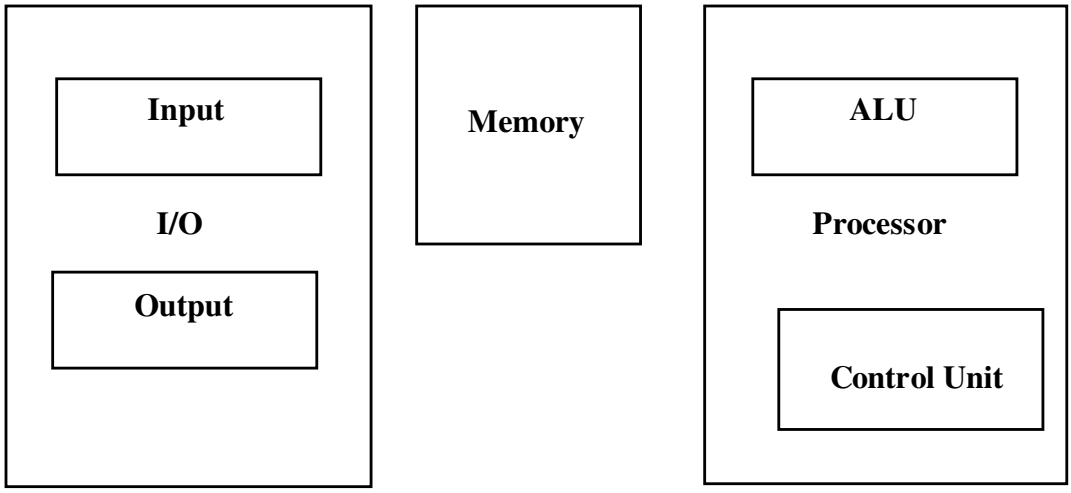


Fig a : Functional units of computer

Input device accepts the coded information as source program i.e. high level language. This is either stored in the memory or immediately used by the processor to perform the desired operations. The program stored in the memory determines the processing steps. Basically the computer converts one source program to an object program. i.e. into machine language.

Finally the results are sent to the outside world through output device. All of these actions are coordinated by the control unit.

Input unit: -

The source program/high level language program/coded information/simply data is fed to a computer through input devices keyboard is a most common type. Whenever a key is pressed, one corresponding word or number is translated into its equivalent binary code over a cable & fed either to memory or processor.

Joysticks, trackballs, mouse, scanners etc are other input devices.

Memory unit: -

Its function into store programs and data. It is basically to two types

1. **Primary memory**
2. **Secondary memory**

1. Primary memory: - Is the one exclusively associated with the processor and operates at the electronics speeds programs must be stored in this memory while they are being executed. The memory contains a large number of semiconductors storage cells. Each capable of storing one bit of information. These are processed in a group of fixed site called word.

To provide easy access to a word in memory, a distinct address is associated with each word location. **Addresses are** numbers that identify memory location.

Number of bits in each word is called word length of the computer. Programs must reside in the memory during execution. Instructions and data can be written into the memory or read out under the control of processor.

Memory in which any location can be reached in a short and fixed amount of time after specifying its address is called random-access memory (RAM).

The time required to access one word in called memory access time. Memory which is only readable by the user and contents of which can't be altered is called read only memory (ROM) it contains operating system.

Caches are the small fast RAM units, which are coupled with the processor and are aften contained on the same IC chip to achieve high performance. Although primary storage is essential it tends to be expensive.

2 Secondary memory: - Is used where large amounts of data & programs have to be stored, particularly information that is accessed infrequently.

Examples: - Magnetic disks & tapes, optical disks (ie CD-ROM's), floppies etc.,

Arithmetic logic unit (ALU):-

Most of the computer operators are executed in ALU of the processor like addition, subtraction, division, multiplication, etc. the operands are brought into the ALU from memory and stored in high speed storage elements called register. Then according to the instructions the operation is performed in the required sequence.

The control and the ALU are may times faster than other devices connected to a computer system. This enables a single processor to control a number of external devices such as key boards, displays, magnetic and optical disks, sensors and other mechanical controllers.

Output unit:-

These actually are the counterparts of input unit. Its basic function is to send the processed results to the outside world.

Examples:- Printer, speakers, monitor etc.

Control unit:-

It effectively is the nerve center that sends signals to other units and senses their states. The actual timing signals that govern the transfer of data between input unit, processor, memory and output unit are generated by the control unit.

Basic operational concepts: -

To perform a given task an appropriate program consisting of a list of instructions is stored in the memory. Individual instructions are brought from the memory into the processor, which executes the specified operations. Data to be stored are also stored in the memory.

Examples: - Add LOCA, R_0

This instruction adds the operand at memory location LOCA, to operand in register R_0 & places the sum into register. This instruction requires the performance of several steps,

1. First the instruction is fetched from the memory into the processor.
2. The operand at LOCA is fetched and added to the contents of R_0
3. Finally the resulting sum is stored in the register R_0

The preceding add instruction combines a memory access operation with an ALU Operations. In some other type of computers, these two types of operations are performed by separate instructions for performance reasons.

Load LOCA, R1
Add R1, R0

Transfers between the memory and the processor are started by sending the address of the memory location to be accessed to the memory unit and issuing the appropriate control signals. The data are then transferred to or from the memory.

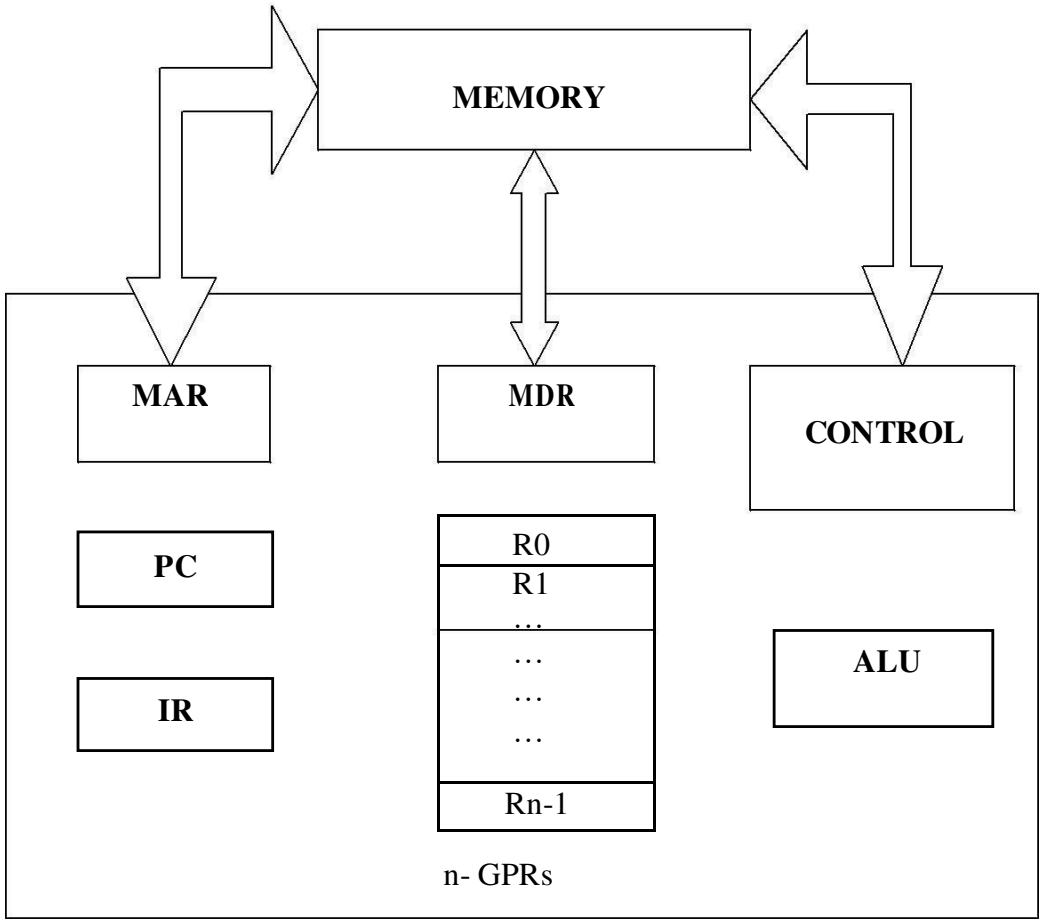


Fig b : Connections between the processor and the memory

The fig shows how memory & the processor can be connected. In addition to the ALU & the control circuitry, the processor contains a number of registers used for several different purposes.

The instruction register (IR):- Holds the instructions that is currently being executed. Its output is available for the control circuits which generates the timing signals that control the various processing elements in one execution of instruction.

The program counter PC:-

This is another specialized register that keeps track of execution of a program. It contains the memory address of the next instruction to be fetched and executed.

Besides IR and PC, there are n-general purpose registers R0 through R_{n-1}. The other two registers which facilitate communication with memory are: -

1. **MAR – (Memory Address Register):-** It holds the address of the location to be accessed.
2. **MDR – (Memory Data Register):-** It contains the data to be written into or read out of the address location.

Operating steps are

1. Programs reside in the memory & usually get these through the I/P unit.
2. Execution of the program starts when the PC is set to point at the first instruction of the program.
3. Contents of PC are transferred to MAR and a Read Control Signal is sent to the memory.
4. After the time required to access the memory elapses, the address word is read out of the memory and loaded into the MDR.
5. Now contents of MDR are transferred to the IR & now the instruction is ready to be decoded and executed.
6. If the instruction involves an operation by the ALU, it is necessary to obtain the required operands.
7. An operand in the memory is fetched by sending its address to MAR & Initiating a read cycle.
8. When the operand has been read from the memory to the MDR, it is transferred from MDR to the ALU.
9. After one or two such repeated cycles, the ALU can perform the desired operation.
10. If the result of this operation is to be stored in the memory, the result is sent to MDR.
11. Address of location where the result is stored is sent to MAR & a write cycle is initiated.
12. The contents of PC are incremented so that PC points to the next instruction that is to be executed.

Normal execution of a program may be preempted (temporarily interrupted) if some devices require urgent servicing, to do this one device raises an Interrupt signal.

An interrupt is a request signal from an I/O device for service by the processor. The processor provides the requested service by executing an appropriate interrupt service routine.

The Diversion may change the internal stage of the processor its state must be saved in the memory location before interruption. When the interrupt-routine service is completed the state of the processor is restored so that the interrupted program may continue.

Bus structure: -

The simplest and most common way of interconnecting various parts of the computer.

To achieve a reasonable speed of operation, a computer must be organized so that all its units can handle one full word of data at a given time.

A group of lines that serve as a connecting port for several devices is called a bus.

In addition to the lines that carry the data, the bus must have lines for address and control purpose.

Simplest way to interconnect is to use the single bus as shown

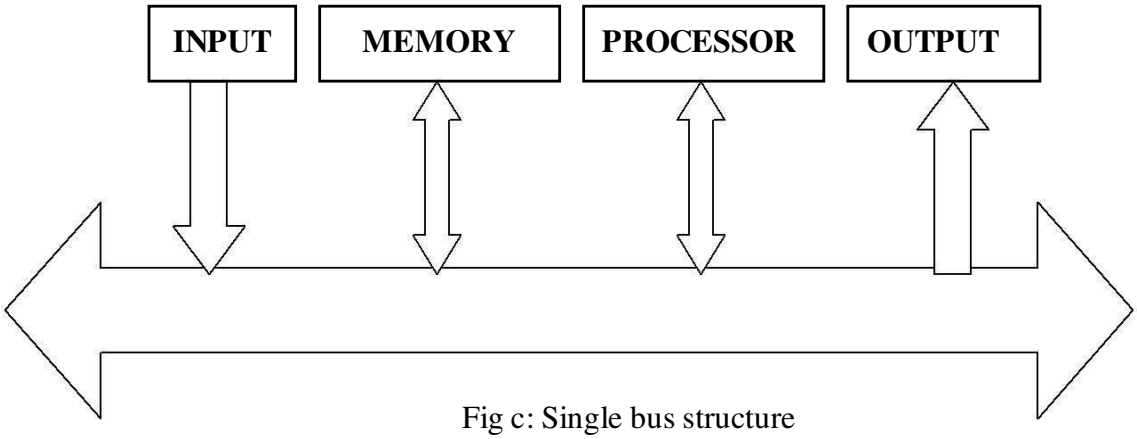


Fig c: Single bus structure

Since the bus can be used for only one transfer at a time, only two units can actively use the bus at any given time. Bus control lines are used to arbitrate multiple requests for use of one bus.

Single bus structure is

- Low cost
- Very flexible for attaching peripheral devices

Multiple bus structure certainly increases, the performance but also increases the cost significantly.

All the interconnected devices are not of same speed & time, leads to a bit of a problem. This is solved by using cache registers (ie buffer registers). These buffers are electronic registers of small capacity when compared to the main memory but of comparable speed.

The instructions from the processor at once are loaded into these buffers and then the complete transfer of data at a fast rate will take place.

Performance: -

The most important measure of the performance of a computer is how quickly it can execute programs. The speed with which a computer executes program is affected by the design of its hardware. For best performance, it is necessary to design the compiles, the machine instruction set, and the hardware in a coordinated way.

The total time required to execute the program is elapsed time is a measure of the performance of the entire computer system. It is affected by the speed of the processor, the disk and the printer. The time needed to execute a instruction is called the processor time.

Just as the elapsed time for the execution of a program depends on all units in a computer system, the processor time depends on the hardware involved in the execution of individual machine instructions. This hardware comprises the processor and the memory which are usually connected by the bus as shown in the fig c.

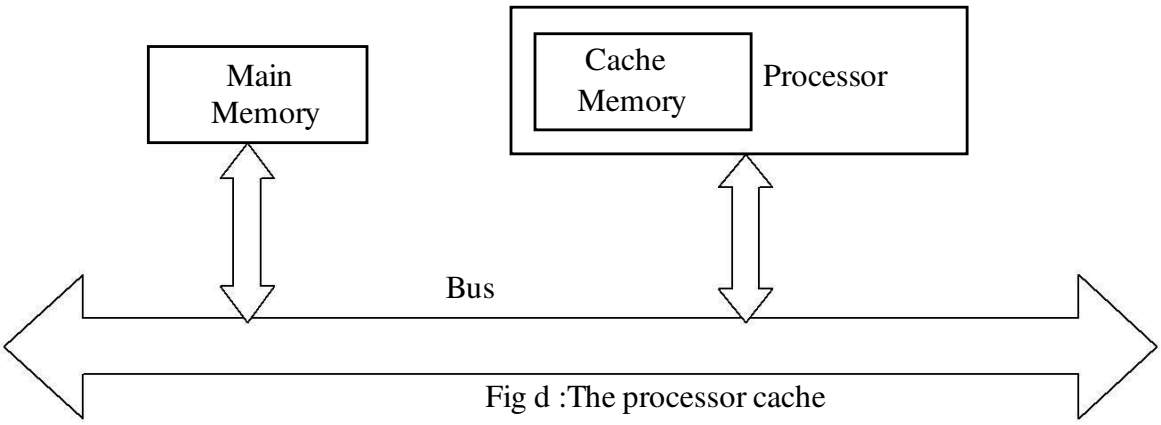


Fig d :The processor cache

The pertinent parts of the fig. c is repeated in fig. d which includes the cache memory as part of the processor unit.

Let us examine the flow of program instructions and data between the memory and the processor. At the start of execution, all program instructions and the required data are stored in the main memory. As the execution proceeds, instructions are fetched one by one over the bus into the processor, and a copy is placed in the cache later if the same instruction or data item is needed a second time, it is read directly from the cache.

The processor and relatively small cache memory can be fabricated on a single IC chip. The internal speed of performing the basic steps of instruction processing on

chip is very high and is considerably faster than the speed at which the instruction and data can be fetched from the main memory. A program will be executed faster if the movement of instructions and data between the main memory and the processor is minimized, which is achieved by using the cache.

For example:- Suppose a number of instructions are executed repeatedly over a short period of time as happens in a program loop. If these instructions are available in the cache, they can be fetched quickly during the period of repeated use. The same applies to the data that are used repeatedly.

Processor clock: -

Processor circuits are controlled by a timing signal called clock. The clock designer the regular time intervals called clock cycles. To execute a machine instruction the processor divides the action to be performed into a sequence of basic steps that each step can be completed in one clock cycle. The length P of one clock cycle is an important parameter that affects the processor performance.

Processor used in today's personal computer and work station have a clock rates that range from a few hundred million to over a billion cycles per second.

Basic performance equation: -

We now focus our attention on the processor time component of the total elapsed time. Let 'T' be the processor time required to execute a program that has been prepared in some high-level language. The compiler generates a machine language object program that corresponds to the source program. Assume that complete execution of the program requires the execution of N machine cycle language instructions. The number N is the actual number of instruction execution and is not necessarily equal to the number of machine cycle instructions in the object program. Some instruction may be executed more than once, which in the case for instructions inside a program loop others may not be executed all, depending on the input data used.

Suppose that the average number of basic steps needed to execute one machine cycle instruction is S, where each basic step is completed in one clock cycle. If clock rate is 'R' cycles per second, the program execution time is given by

$$T = \frac{N \times S}{R}$$

this is often referred to as the basic performance equation.

We must emphasize that N, S & R are not independent parameters changing one may affect another. Introducing a new feature in the design of a processor will lead to improved performance only if the overall result is to reduce the value of T.

Pipelining and super scalar operation: -

We assume that instructions are executed one after the other. Hence the value of

S is the total number of basic steps, or clock cycles, required to execute one instruction.
A substantial improvement in performance can be achieved by overlapping the execution of successive instructions using a technique called pipelining.

Consider Add R_1 R_2 R_3

This adds the contents of R_1 & R_2 and places the sum into R_3.

The contents of R_1 & R_2 are first transferred to the inputs of ALU. After the addition operation is performed, the sum is transferred to R_3. The processor can read the next instruction from the memory, while the addition operation is being performed. Then of that instruction also uses, the ALU, its operand can be transferred to the ALU inputs at the same time that the add instructions is being transferred to R_3.

In the ideal case if all instructions are overlapped to the maximum degree possible the execution proceeds at the rate of one instruction completed in each clock cycle. Individual instructions still require several clock cycles to complete. But for the purpose of computing T, effective value of S is 1.

A higher degree of concurrency can be achieved if multiple instructions pipelines are implemented in the processor. This means that multiple functional units are used creating parallel paths through which different instructions can be executed in parallel with such an arrangement, it becomes possible to start the execution of several instructions in every clock cycle. This mode of operation is called superscalar execution. If it can be sustained for a long time during program execution the effective value of S can be reduced to less than one. But the parallel execution must preserve logical correctness of programs, that is the results produced must be same as those produced by the serial execution of program instructions. Now a days may processor are designed in this manner.

Clock rate:- These are two possibilities for increasing the clock rate 'R'.

1. Improving the IC technology makes logical circuit faster, which reduces the time of execution of basic steps. This allows the clock period P, to be reduced and the clock rate R to be increased.
2. Reducing the amount of processing done in one basic step also makes it possible to reduce the clock period P. however if the actions that have to be performed by an instructions remain the same, the number of basic steps needed may increase.

Increase in the value 'R' that are entirely caused by improvements in IC technology affects all aspects of the processor's operation equally with the exception of the time it takes to access the main memory. In the presence of cache the percentage of accesses to the main memory is small. Hence much of the performance gain excepted from the use of faster technology can be realized.

Instruction set CISC & RISC:-

Simple instructions require a small number of basic steps to execute. Complex

instructions involve a large number of steps. For a processor that has only simple instruction a large number of instructions may be needed to perform a given programming task. This could lead to a large value of 'N' and a small value of 'S' on the other hand if individual instructions perform more complex operations, a fewer instructions will be needed, leading to a lower value of N and a larger value of S. It is not obvious if one choice is better than the other.

But complex instructions combined with pipelining (effective value of $S \approx 1$) would achieve one best performance. However, it is much easier to implement efficient pipelining in processors with simple instruction sets.

Performance measurements:-

It is very important to be able to access the performance of a computer, comp designers use performance estimates to evaluate the effectiveness of new features.

The previous argument suggests that the performance of a computer is given by the execution time T, for the program of interest.

Inspite of the performance equation being so simple, the evaluation of 'T' is highly complex. Moreover the parameters like the clock speed and various architectural features are not reliable indicators of the expected performance.

Hence measurement of computer performance using bench mark programs is done to make comparisons possible, standardized programs must be used.

The performance measure is the time taken by the computer to execute a given bench mark. Initially some attempts were made to create artificial programs that could be used as bench mark programs. But synthetic programs do not properly predict the performance obtained when real application programs are run.

A non profit organization called SPEC- system performance evaluation corporation selects and publishes bench marks.

The program selected range from game playing, compiler, and data base applications to numerically intensive programs in astrophysics and quantum chemistry. In each case, the program is compiled under test, and the running time on a real computer is measured. The same program is also compiled and run on one computer selected as reference.

The 'SPEC' rating is computed as follows.

$$\text{SPEC rating} = \frac{\text{Running time on the reference computer}}{\text{Running time on the computer under test}}$$

If the SPEC rating = 50

Means that the computer under test is 50 times as fast as the ultra sparc 10. This is repeated for all the programs in the SPEC suit, and the geometric mean of the result is computed.

Let SPECi be the rating for program 'i' in the suite. The overall SPEC rating for the computer is given by

$$\text{SPEC rating} = \left(\prod_{i=1}^{n} SPECi \right)^{\frac{1}{n}}$$

Where 'n' = number of programs in suite.

Since actual execution time is measured the SPEC rating is a measure of the combined effect of all factors affecting performance, including the compiler, the OS, the processor, the memory of comp being tested.

Multiprocessor & microprocessors:-

- Large computers that contain a number of processor units are called multiprocessor system.
- These systems either execute a number of different application tasks in parallel or execute subtasks of a single large task in parallel.
- All processors usually have access to all memory locations in such system & hence they are called shared memory multiprocessor systems.
- The high performance of these systems comes with much increased complexity and cost.
- In contrast to multiprocessor systems, it is also possible to use an interconnected group of complete computers to achieve high total computational power. These computers normally have access to their own memory units when the tasks they are executing need to communicate data they do so by exchanging messages over a communication network. This properly distinguishes them from shared memory multiprocessors, leading to name message-passing multi computer.

DATA REPRESENTATION

1.1 Data Representation

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

Data Types

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

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

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

Number Systems

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

Decimal Number system

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

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

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

Binary Number system

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

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

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

Octal and Hexadecimal Representation

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

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

```
1  5  5  4        6  5              Octal
1 1 0 11 0 1 1 0 0 1 1 0 1 0 1      Binary
 D  B     3        5     Hexadecimal
```

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

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

1-1.Binary –code Octal Numbers

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

1-2Binary - coded Hexadecimal Numbers

Hexadecimal Number	Binary – Coded Hexadecimal	Decimal Number	
0	0000	0	
1	0001	1	
2	0010	2	Code for
3	0011	3	single Hexadecimal
4	0100	4	digit
5	0101	5	
6	0110	6	
7	0111	7	
8	1000	8	
9	1001	9	
A	1010	10	
B	1011	11	
C	1100	12	
D	1101	13	
E	1110	14	
F	1111	15	
16	0001 0110	22	
32	0011 0010	50	
64	0110 0100	100	
F9	1111 1001	249	

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

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

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

Alphanumeric Representation

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

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

Table 1-3

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

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

1.2 Complements

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

(r-1)'s Complement

9's Complement

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

1's complement

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

(r's) complement

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

10's complement and 2's complement

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

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

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

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

1.3. Fixed point representation

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

- By giving it a fixed position

- By giving a floating point representation.

The fixed point method assumes that the binary point is always fixed in one position.

The two positions widely used are:

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

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

Integer representation

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

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

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

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

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

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

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

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

Arithmetic addition:

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

Examples:

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

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

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

Arithmetic subtraction

The subtraction of two signed binary numbers is very simple. Whennegative numbers are in 2‘s complement form. First take the 2‘s complement of the subtrahend, including the sign bit and add it to the minuend, including the sign bit. Finally discard any carry out of the sign bit. This procedure reveals the fact that a subtraction operation can be changed to an addition operation if the sign of the negative number is easily done by taking its 2‘s complement. The reverse is also true because the complement of a negative number in complement form produces the equivalent positive number.
Consider the subtraction of (-5) and (-

12)i.e. ((-5) – (-12)) = + 7

In binary with eight bits,

-5 = 11111011 and -12 = 11110100

This is obtained by changing all 1‘s to 0 and 0‘s to 1 of the binary numbers +5 and

+12 for all the eight bits and then adding one to the least bit.

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

It is important to note that binary numbers in the signed 2‘s complement system are

added and subtracted by the same basic addition and subtraction rules as unsigned numbers.

Overflow

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

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

indicate an overflow.

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

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

Decimal Fixed point representation

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

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

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

1.4Floating point representation

The floating point representation of a number has two parts.

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

The fixed point mantissa may be a fraction or an integer. For example, the decimal number +3245.698 is represented in floating point with a fraction as +0.3245698 and an exponent as +04. The value of the exponent indicates that the actual position of the decimal point is four positions to the right of the indicated decimal point in the fraction. This representation is equivalent to the scientific notation $+0.3245698 \times 10^{-4}$. Hence, a floating point is always represented as $m \times r^e$. In the register, only the mantissa _m‘ and the exponent

_e‘ are physically represented, including the signs. The radix _r‘ and the radix-point position of the mantissa are always assumed. The circuits that manipulate the floating point numbers in registers confirm with these two assumptions in order to provide the correct computational results.

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

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

A floating point number is said to be normalized if the most significant digit of the mantissa is non-zero. For example, the decimal number 430 is normalized but 00043 is not. In the same way, the 8-bit binary 00010101 is not normalized by shifting it three

positions to the left and discarding the leading 0‘s to obtain 10101000. The three shifts multiply the numbers by $2^3 = 8$. The exponent must be subtracted by 3, to keep the same value for the floating point number is provided by the normalized numbers. A zero cannot be normalized because it does not have a nonzero digit. It is represented in floating point as all 0‘s in the mantissa and exponent.

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

1.5 Error detection codes

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

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

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

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

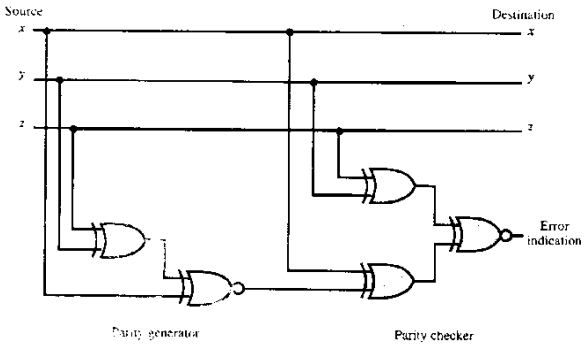
As an example, consider a 3-bit message to be transmitted with an odd parity bit. At the sending end, the odd-parity bit is generated by a parity generator circuit. As shown in Fig.1.1, this circuit consists of one exclusive-OR and one exclusive-NOR gate. Since P(even) is the exclusive-OR of x, y, z, and P(odd) is the complement of P(even), it

is necessary to employ an exclusive-NOR gate for the needed complementation. The message and-the odd-parity bit are transmitted to their destination where they are applied to a parity checker. An error has occurred during transmission if the parity of the four bits received is even, since the binary information transmitted was originally odd.

TABLE 1-4 Parity Bit Generation

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

Figure 1.1 Error detection with odd parity bit



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

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

2. REGISTER TRANSFER AND MICROOPERATIONS

2.1 Register Transfer Language

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

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

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

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

2.2 Register Transfer

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

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

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

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

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

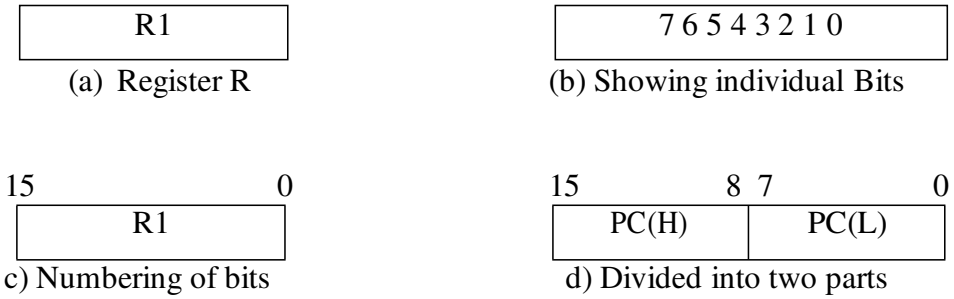
$$R2 \leftarrow R1$$

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

A statement that specifies a register transfer implies that circuits are available from

the outputs of the source register to the inputs of the destination register and that the destination register has a parallel load capability. Normally we want the transfer to occur only under a predetermined control condition. This can be shown by means of an if-then statement.

Figure 2.1 Block diagram of register



If (P = 1) then (R2←R1)

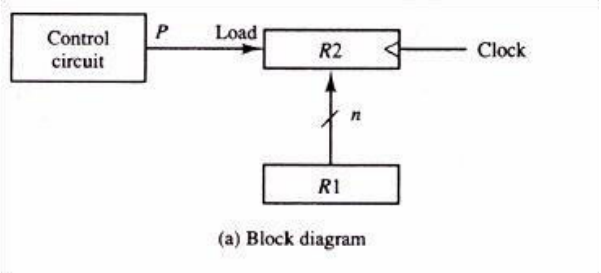
Where P is a control signal generated in the control section. It is sometimes convenient to separate the control variables from the register transfer operation by specifying a control function. A control function is a Boolean variable that is equal to 1 or 0. The control function is included in the statement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

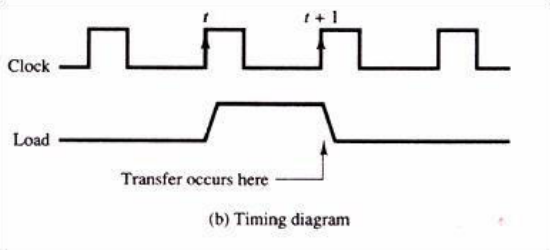
P: R2 ← R1

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

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

Figure 2.2 Transfer from R1 to R2 when P = 1





(b) Timing diagram

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

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

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

T: R2←R1, R1 ←R2

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

Table 2-1 Basic symbols for Register Transfers

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

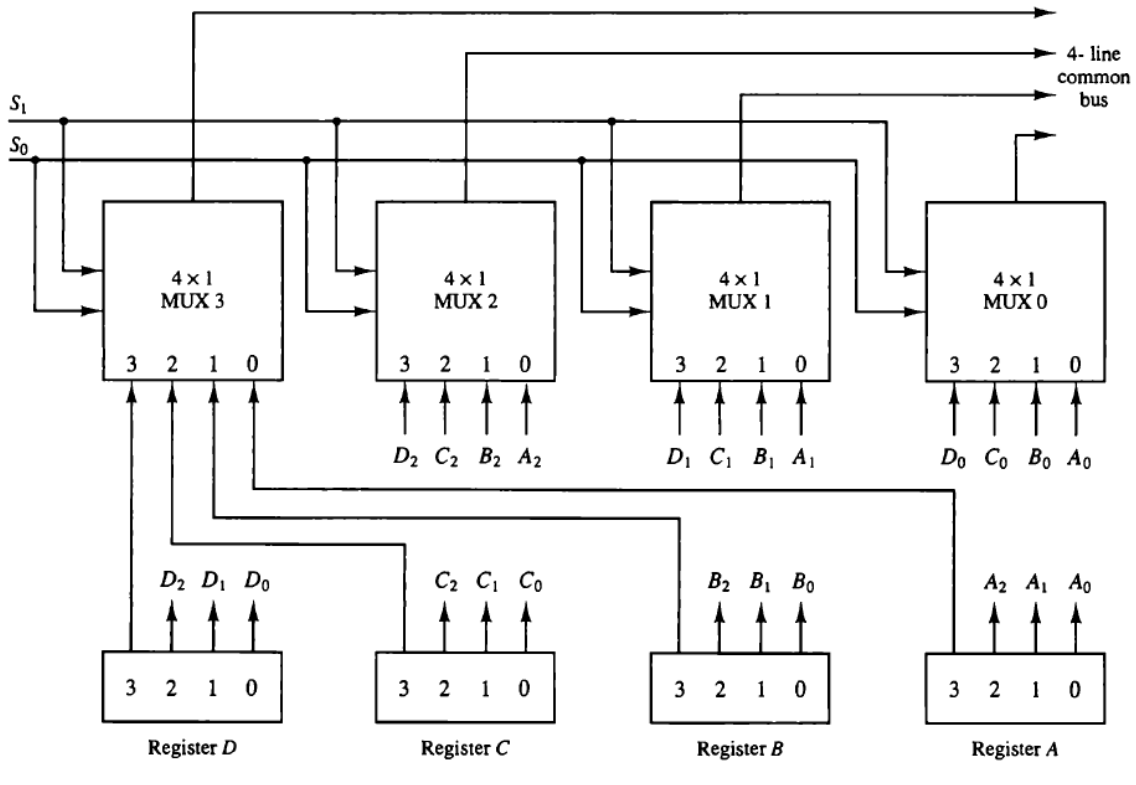
2.3 Bus and Memory Transfers

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

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

For example, output 1 of register A in connected to input 0 of MUX 1 because this input is labeled A_1. The diagram shows that the bits in the same significant position in each register are connected to the data inputs of one multiplexer to form one line of the bus. Thus MUX 0 multiplexes the four 0 bits of the register, MUX 1 multiplexes the four 1 bits registers, and similarly for the other two bits.

Figure 2.3 Bus system for four registers



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

Table 2-2 Function table for bus of fig 2.3

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

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

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

$$BUS \leftarrow C, \qquad R1 \leftarrow BUS$$

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.

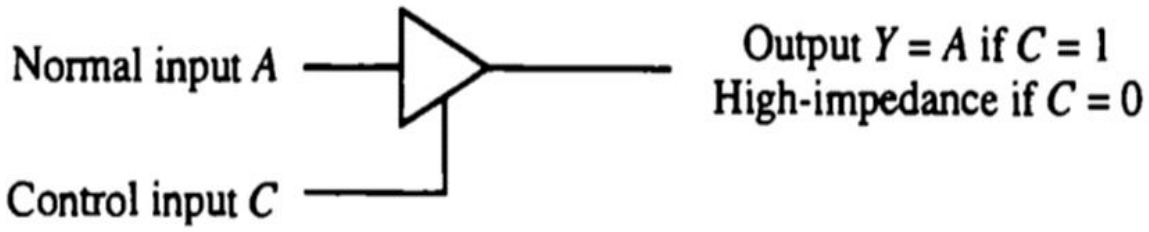
$$R1 \leftarrow C$$

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

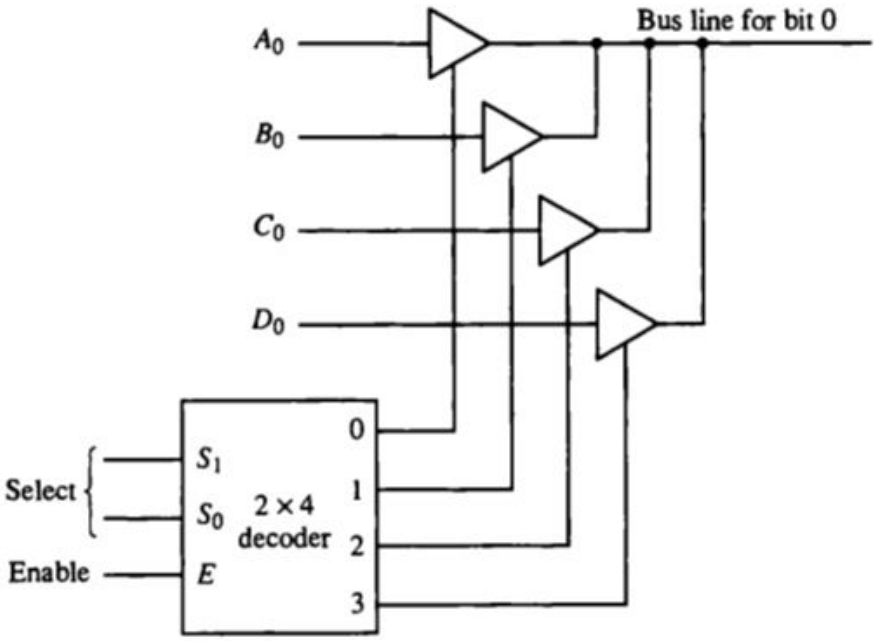
Three State Bus Buffers

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

Figure 2.4 Graphic symbols for three-state buffer



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



Bus system with three-state buffers

The outputs of the 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 lines. 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 others are in high impedance. Decoder can be used to control the active input at any given time. When enable input of decoder is 0 all the outputs are zero and the lines are in high impedance. When enable is 1 than one of the four output of decoder is active depending upon binary values of the selection lines.

Memory Transfer

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

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

Read: DR ← M (AR)

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

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

Write: M (AR) ← R1

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

2.4 Arithmetic Microoperations

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

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

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

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

R3 ← R1 + R2

specifies an add microoperation. It states that the contents of register R1 are added to the contents of register R2 and the sum transferred to register R3. To implement this statement with hardware we need three registers and the digital component that performs the addition operation. The other basic arithmetic microoperations are listed in Table 2-3. Subtraction is most often implemented through complementation and addition. Instead of using the minus operator, we can specify the subtraction by the following statement.

-

$$R3 \leftarrow R1 + R2 + 1$$

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

Table 2-3 Arithmetic Microoperation

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

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

Binary Adder

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

Figure 2.5 4 bit binary adder

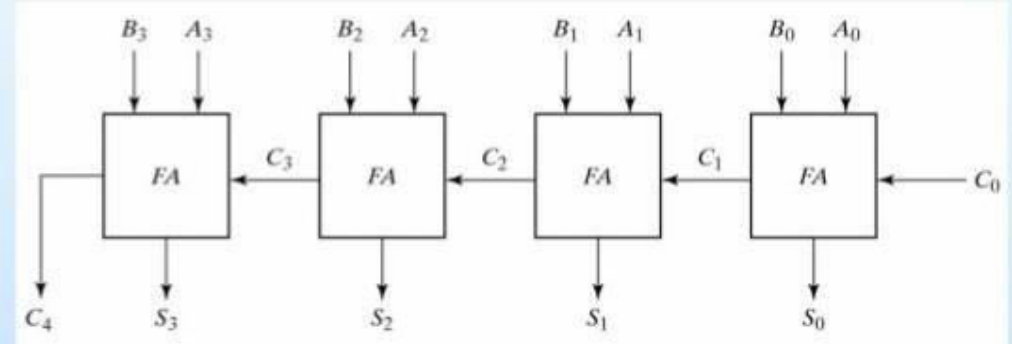


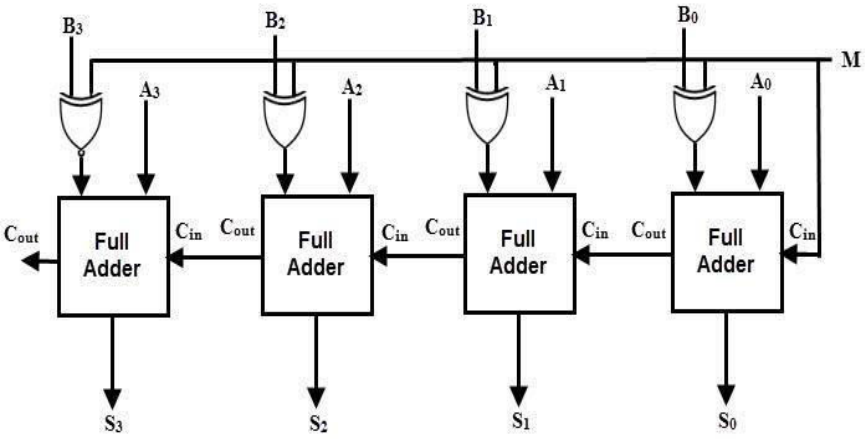
Figure 2.5 shows the interconnections of four full-adders (FA) to provide a 4-bit binary adder. The augend bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the low-order bit. The carries are connected in a chain through the full-adders.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

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

Binary Adder-Subtractor

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

Figure 2.6 4-bit adder – subtractor



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

2.5 Logic Microoperations

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

P: R1 ← R1⊕ R2

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

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

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

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

P + Q; R1 ← R2 + R3, R4 ← R5 v R6

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

List of Logic Microoperations

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

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

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

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

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

Hardware Implementation

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

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

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

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

```
1010   A before
1100   B (logic erand)
1110   A after
```

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

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

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

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

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

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

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

Again the two leftmost bits of B are 1‘s, so the corresponding bits of A are cleared to 0.

One can deduce that the Boolean operation performed on the individual bits is AB‘.

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

$A \leftarrow A \wedge B$

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

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

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

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

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

And then insert the new value:

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

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

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

1010 A
1010 B

0000 A ← A ⊕B

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

2.6. Shift Microoperations

Shift rnicrooperations are used for serial transfer of data. They are also used in conjunction with arithmetic, logic, and other data-processing operations. The contents of a register can be shifted to the left or the right. At the same time that the bits are shifted, the first flip-flop receives its binary information from the serial input. During a shift-left operation the serial input transfers a bit into the rightmost position. During a shift-right operation the serial input transfers a bit into the leftmost position. The information transferred through the serial input determines the type of shift. There are three types of shifts: logical, circular, and arithmetic.

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

R1 ← shl R1

R2 ← shr R2

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

The circular shift (also known as a **rotate operation**) circulates the bits of the register around the two ends without loss of information. This is accomplished by connecting the serial output of the shift register to its serial input. We will use the symbols c_{il} and c_{ir} for the circular shift left and right, respectively. The symbolic notation for the shift rnicrooperations is shown in Table 4-7.

TABLE 4-7 Shift Microoperations

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

An arithmetic shift is a rnicrooperation that shifts a signed binary number to the left or right. An arithmetic shift-left multiplies a signed binary number by 2. An arithmetic shift-right divides the number by 2. Arithmetic shifts must leave the sign bit unchanged because the sign of the number remains the same

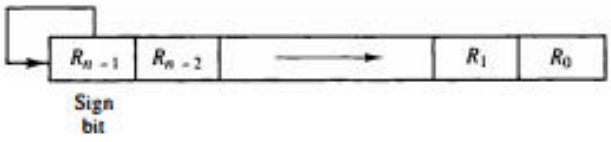


Figure 4-11 Arithmetic shift right.

when it is multiplied or divided by 2. The leftmost bit in a register holds the sign bit, and the remaining bits hold the number. The sign bit is 0 for positive and 1 for negative. Negative numbers are in 2's complement form. Figure 4-11 shows a typical register of n bits. Bit R_{n-1} in the leftmost position holds the sign bit. R_{n-2} is the most significant bit of the number and R_o is the least significant bit. The arithmetic shift-right leaves the sign bit unchanged and shifts the number (including the sign bit) to the right. Thus R_{n-1} remains the same, R_{n-2} receives the bit from R_{n-1} and so on for the other bits in the register. The bit in R_o is lost.

The arithmetic shift-left inserts a 0 into R_0 and shifts all other bits to the left. The initial bit of R_{n-1} is lost and replaced by the bit from R_{n-2}. A sign reversal occurs if the bit in R_{n-1} changes in value after the shift. This happens if the multiplication by 2 causes an overflow. An overflow occurs after an arithmetic shift left if initially, before the shift, R_{n-1} is not equal to R_{n-2}. An overflow flip-flop V_s can be used to detect an arithmetic shift-left overflow.

$V_s = R_{n-1} + R_{n-2}$

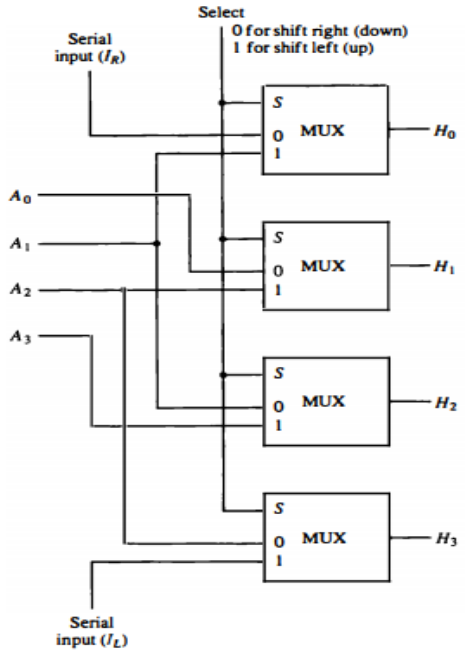
If $V_s = 0$, there is no overflow, but if $V_s = I$, there is an overflow and a sign reversal after the shift. V_s must be transferred into the overflow flip-flop with the same clock pulse that shifts the register.

Hardware implementation

A possible choice for a shift unit would be a bidirectional shift register with parallel load (see Fig. 2-9). Information can be transferred to the register in parallel and then shifted to the right or left. In this type of configuration, a clock pulse is needed for loading the data into the register, and another pulse is needed to initiate the shift. In a processor unit with many registers it is more efficient to implement the shift operation with a combinational circuit. In this way the content of a register that has to be shifted is first placed onto a common bus whose output is connected to the combinational shifter, and the shifted number is then loaded back into the register. This requires only one clock pulse for loading the shifted value into the register.

shifter:

A combinational circuit shifter can be constructed with multiplexers as shown in Fig. 4-12. The 4-bit shifter has four data inputs, A_0 through A_3 and four data outputs, H_0 through H_3. There are two serial inputs, one for shift left (IL) and the other for shift right (h).



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

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.

UNIT II

3. BASIC COMPUTER ORGANIZATION AND DESIGN

3.1 Instruction Codes

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

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

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

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

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

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

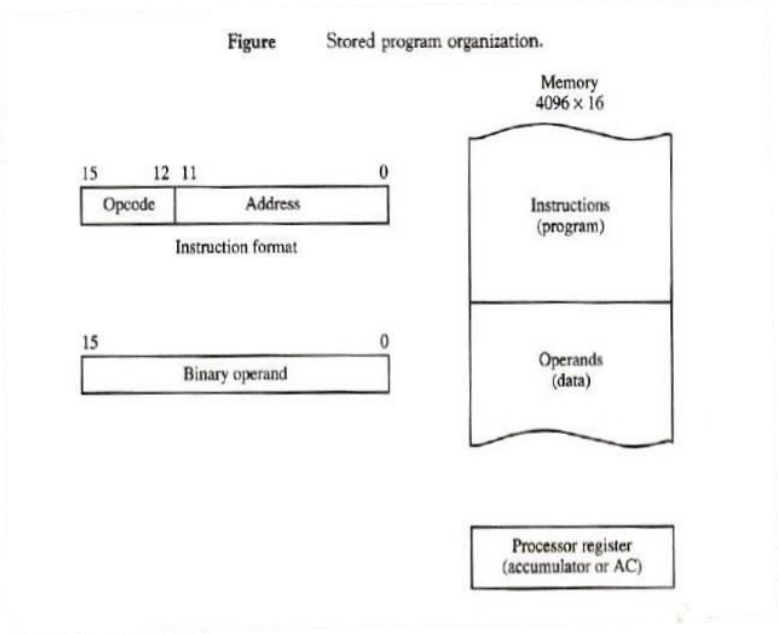
Stored program Organization

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

Figure 3.1 depicts this type of organization. Instructions are stored in one section of memory and a data in another. For a memory unit with 4096 words we need 12 bits to

specify an address since $2^{12} = 4096$. If we store each instruction code in one 16-bit memory word, we have available four bits for the operation code (abbreviated as opcode) to specify one out of 16 possible operation and 12 bits to specify the address of an operand. The control reads a 16-bit instruction from the program portion of memory. It uses the 12 bit address part of the instruction to read and 16 bit operand from the data portion of memory. It then executes the operation specified by the operation code.

Figure 3.1 Stored program organization



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

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

Indirect Address

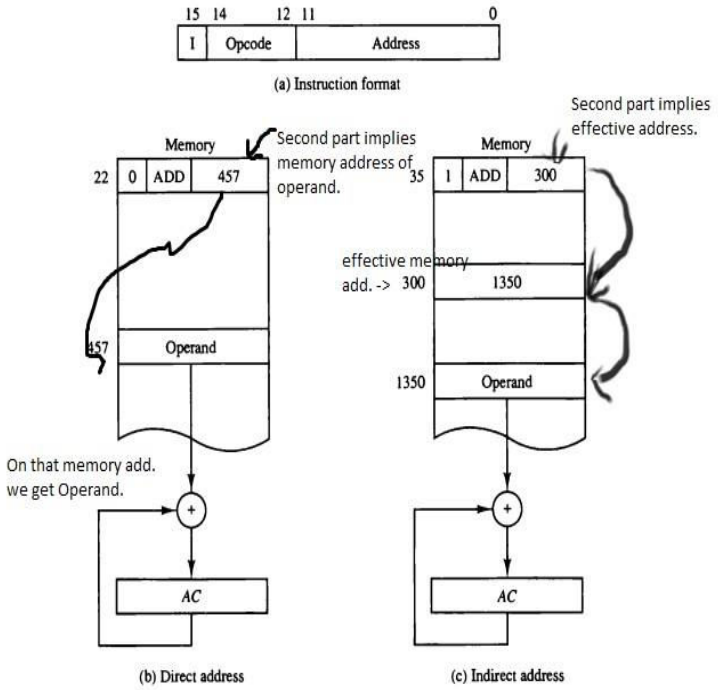
It is sometimes convenient to use the address bits of an instruction code not as an address but as the actual operand. When the second part of an operand, the instruction is said to have immediate operand. When the second part specifies the address of an operand, the instruction is said to have a direct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

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

A direct address instruction is shown in fig. 3.2.b. It is placed in address 22 in memory. The I bit is 0, so the instruction is recognized as a direct address instruction. The opcode specifies an ADD instruction, and the address part is the binary equivalent of 457.

The control finds the operand in memory at address 457 and adds it to the content of AC. The instruction in address 35 shown in fig.3.2.c.has a mode bit I = 1. Therefore, it is recognized as an indirect address instruction. The address part is the binary equivalent of 300. The control goes to address 300 to find the address instruction needs two references to memory to fetch and operand. The address of the operand in this case is 1350. The operand found in address 1350 is then added to the content of AC.

Figure 3.2 Demonstration of direct and indirect address



3.2 Computer Registers

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

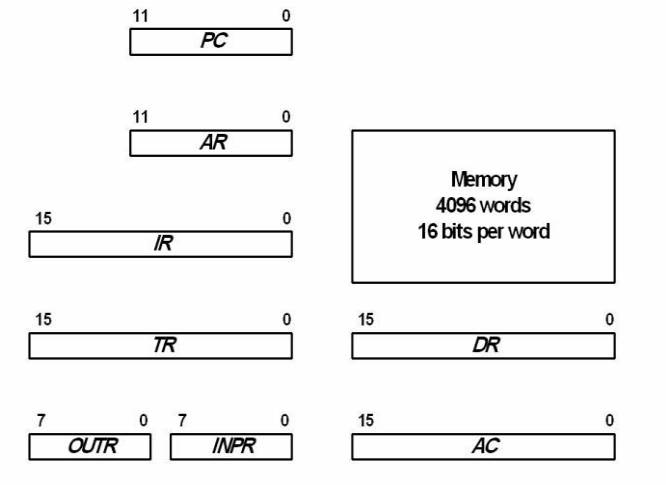
Table 3-1 List of registers for the Basic Computer

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

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

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

Figure 3. 3 Basic computer registers and memory



Common Bus System

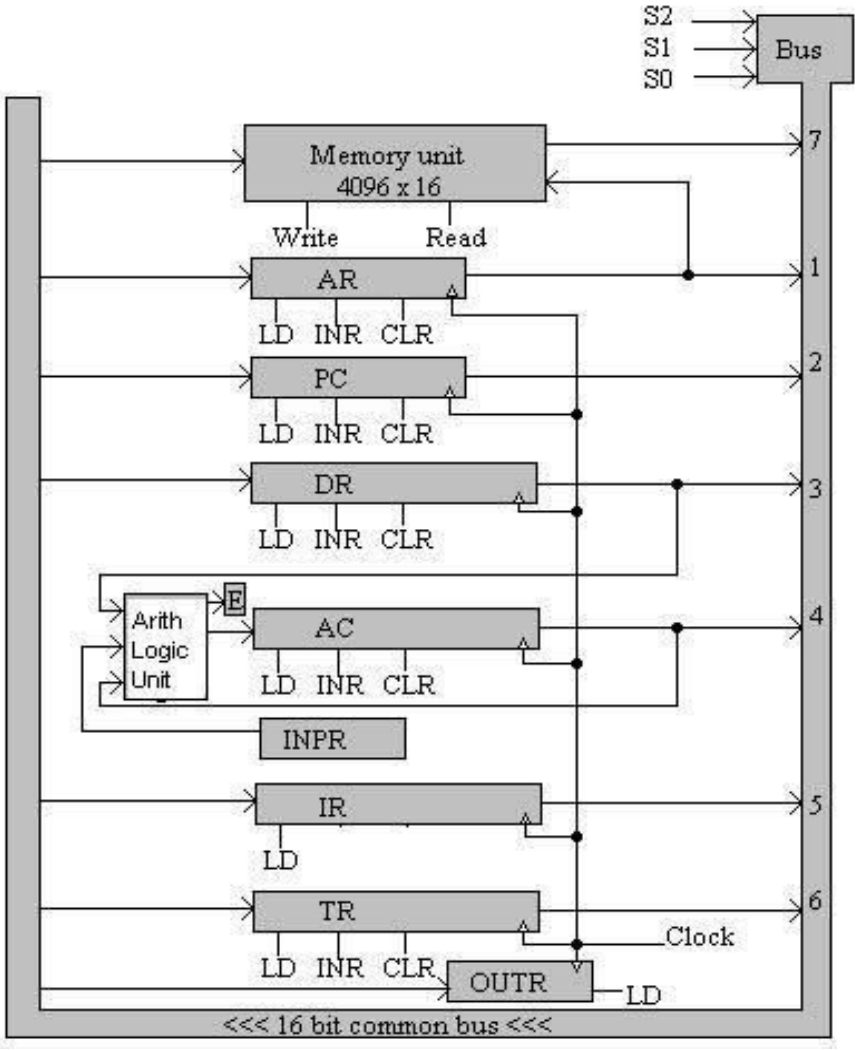
The basic computer has eight registers, a memory unit, and a control unit. Paths must be provided to transfer information from one register to another and between memory and register. The number of wires will be excessive if connections are made between the outputs of each register and the inputs of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

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

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

Four registers, DR, AC, IR, and TR, have 16 bits each. Two registers, AR and PC, have 12 bits each since they hold a memory address. When the contents of AR or PC are applied to the 16-bit common bus, the four most significant bits are set to 0‘s. When AR or PC receive information from the bus, only the 12 least significant bits are transferred into the register.

Figure 3.4 Common Bus system



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

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

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

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

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

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

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

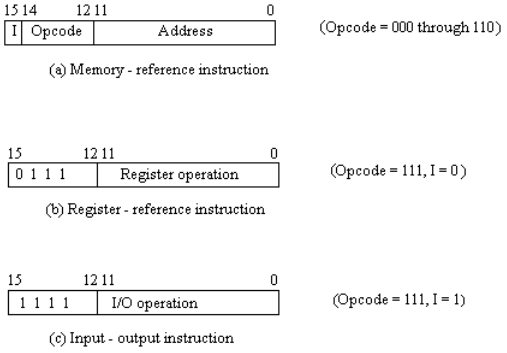
3.3 Computer Instructions

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

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

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

Figure 3.5 Basic computer Instruction formats



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

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

Table3-2 Basic Computer Instruction

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

Instruction Set Completeness

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

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

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

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

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

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

3.4 Timing and control

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

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

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

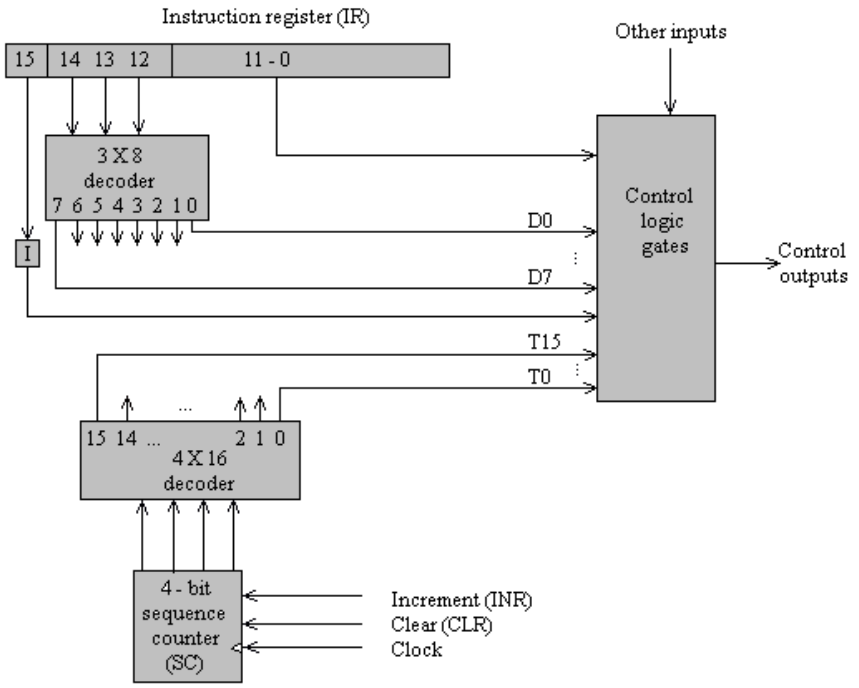
The instruction register is shown again in Fig, 3.6 where it is divided into three parts the I bit, the operation code and bits 0 through 11. The operation code in bits12 through 14 are decoded with a 3 x 8 decoder. The eight outputs of the decoder are designated by the binary value of the corresponding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

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

D_3T_4: SC ← 0

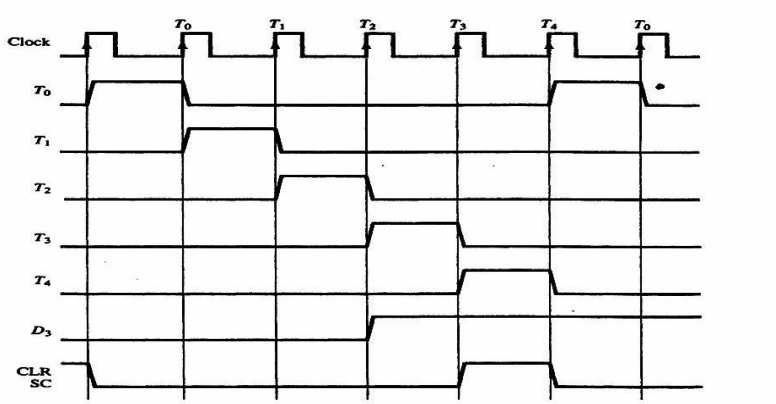
The timing diagram of Fig. 3.7 shows the time relationship of the control signals. The sequence counter SC responds to the positive transition of the clock. Initially, the CLR input of SC is active. The first positive transition of the Clock clears SC to 0, which in turn activates the timing signal T_0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

Figure 3.6 Control unit of basic computer



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

Figure 3.7 Example of control timing signals



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

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

3.5 Instruction Cycle

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

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

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

Fetch and decode

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

T_0: AR←PC

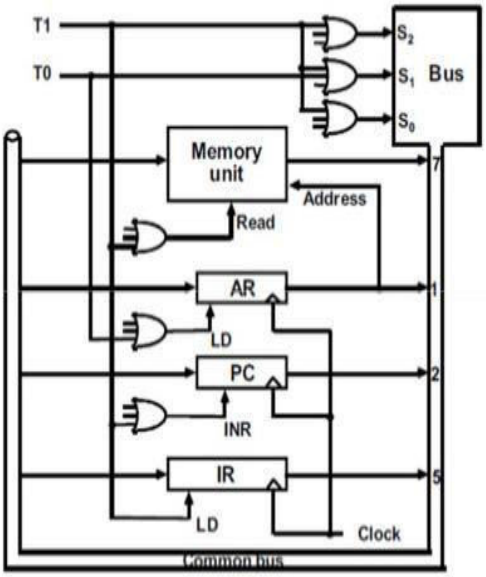
T_1: IR←M [AR] PC←PC +1

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

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

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

Figure 3.8 Register transfer for the fetch phase



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

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

T_1: IR←M (AR) PC←PC + 1

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

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

The next clock transition initiates the read and increment operations since T_1 = 1.

Figure 3.8 duplicates a portion of the bus system and shows how T_0 and T_1 are connected to the control

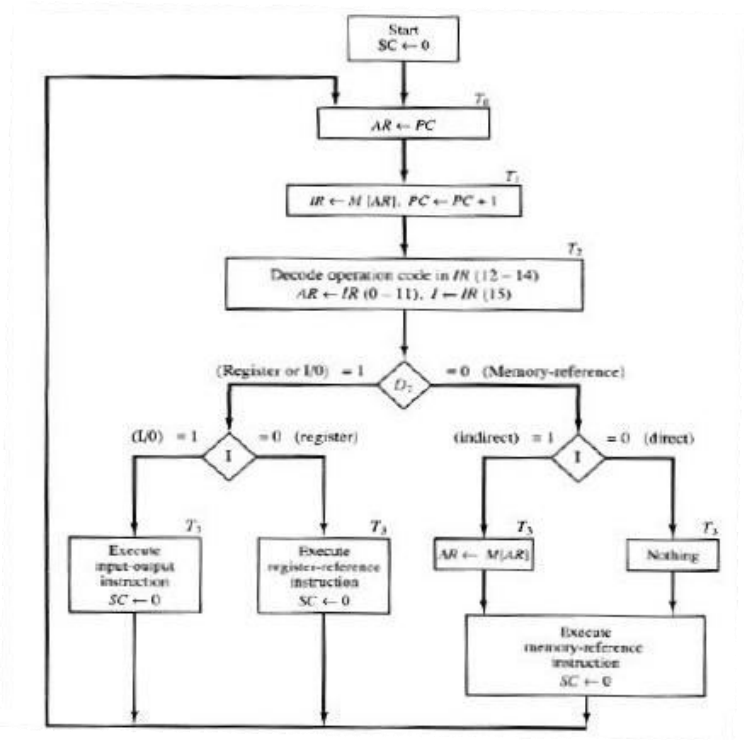
inputs of the registers, the memory, and the bus selection inputs. Multiple input OR gates are included in the diagram because there are other control functions that will initiate similar operations.

Determine the Type of Instruction

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

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

Figure 3.9 Flowchart for instruction cycle



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

AR ←M(AR)

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

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

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

Register-Reference Instructions

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

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

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

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

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

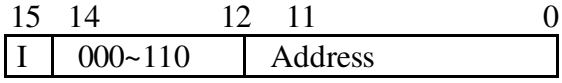
Table 3-3 Execution of register reference instructions

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

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

Memory reference instructions

- When the memory-reference instruction is decoded, D7 bit is set to 0.



- The following table lists seven memory-reference instructions.

Symbol	Operation Decoder	Symbolic Description
AND	D_0	AC ← AC ∧ M[AR]
ADD	D_1	AC ← AC + M[AR], E ← C_{out}
LDA	D_2	AC ← M[AR]
STA	D_3	M[AR] ← AC
BUN	D_4	PC ← AR
BSA	D_5	M[AR] ← PC, PC ← AR + 1
ISZ	D_6	M[AR] ← M[AR] + 1, if M[AR] + 1 = 0 then PC ← PC+1

- 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.

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.

D0T4: DR□M[AR]

D0T5: AC □□AC □□DR, SC □□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.

D1T4: DR □□M[AR]

D1T5: AC □□AC + DR, E □□Cout, SC □□0

LDA: Load to AC

This instruction transfers the memory word specified by the effective address to AC.

D2T4: DR □□M[AR]

D2T5: AC □□DR, SC □□0

STA: Store AC

This instruction stores the content of AC into the memory word specified by the effective address.

D3T4: M[AR] □□AC, SC □□0

BUN: Branch Unconditionally

This instruction transfers the program to instruction specified by the effective address. The BUN instruction allows the programmer to specify an instruction out of sequence and the program branches (or jumps) unconditionally.

D4T4: PC □□AR, SC □□0

BSA: Branch and Save Return Address

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.

M[AR] □□PC, PC □□AR + 1
M[135] □21, PC □135 + 1 = 136

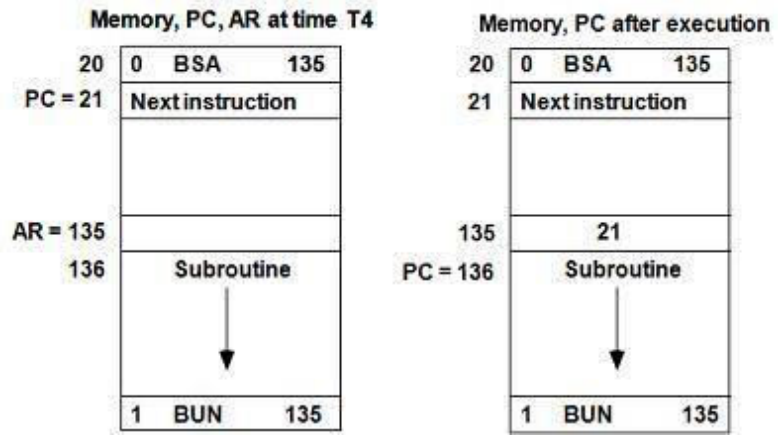


Figure2.10: Example of BSA instruction execution

It is not possible to perform the operation of the BSA instruction in one clock cycle when we use the bus system of the basic computer. To use the memory and the bus properly, the BSA instruction must be executed with a sequence of two microoperations:

D5T4: M[AR] □□PC, AR
□□AR + 1 D5T5: PC □□AR, SC
□□0

ISZ: Increment and Skip if Zero

These instruction increments the word specified by the effective address, and if the incremented value is equal to 0, PC is incremented by 1. 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.

D6T4: DR □□M[AR]
D6T5: DR □□DR + 1
D6T4: M[AR] □□DR, if (DR = 0) then (PC □□PC + 1), SC □□0

Control Flowchar

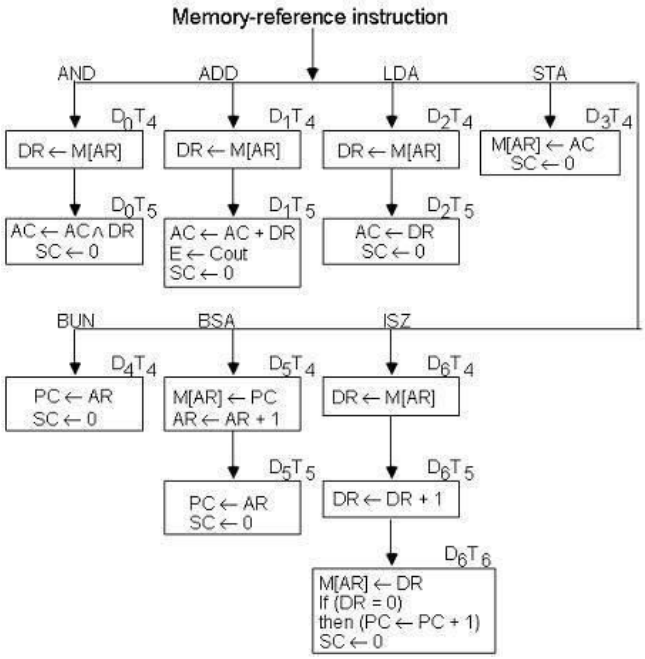


Figure 2.11: Flowchart for memory-reference instructions

3.6 Input-Output and Interrupt

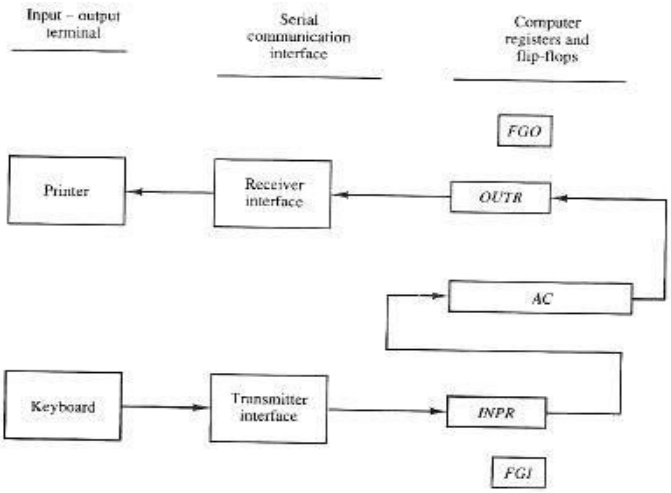
Instructions and data stored in memory must come from some input device. Computational results must be transmitted to the user through some output device. Commercial computers include many types of input and output devices

Input-Output Configuration

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

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

Figure 3.10 Input output configuration



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

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

Input – Output Instructions

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

Table 3-4 Input-output instructions

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

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

Program Interrupt

The process of communication just described is referred to as programmed control transfer. The computer keeps checking the flag bit, and when it finds it set, it

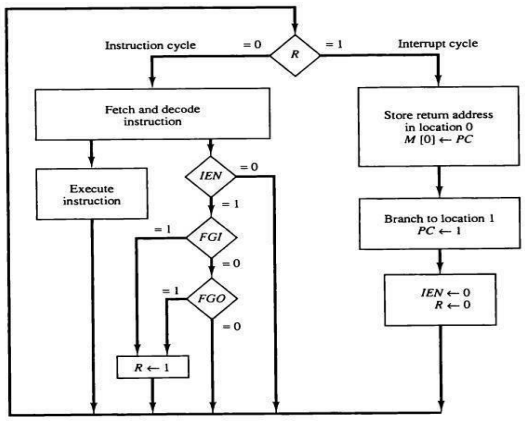
initiates an information transfer. The difference of information flow rate between the computer and that of the input-output device makes this type of transfer inefficient. To see why this is inefficient, consider a computer that can transfer information at a maximum rate of 10 characters per second.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

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

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

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

Figure 3.11 Flowchart for interrupt cycle



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

Interrupt Cycle

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

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

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

RT_0: AR←0. TR ← PC

RT_1: M(AR) ← TR,PC ← 0

RT_2: PC ← PC + 1, IEN ← 0, R←0, SC←0

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

The outputs of the control logic circuit are:

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

UNIT III

4. CENTRAL PROCESSING UNIT

4.1 Stack Organization

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

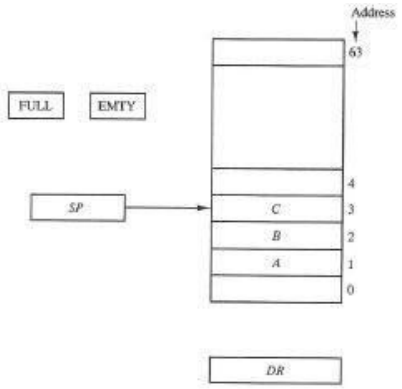
The stack in digital computers is essentially a memory unit with an address register that can count only (after initial value is loaded into it). The register that holds the address for the stack is called a stack pointer (SP) because its value always points at the top item in the stack.

The two operations of a stack are the insertion and deletion of items. The operation of insertion is called push (or push-down) because it can be thought of as the result of pushing a new item on top. The operation of deletion is called pop (or pop-up) because it can be thought of as the result of removing one item so that the stack pops up.

Register Stack

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

Figure 4.3 Block diagram of 64-word stack



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

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

with the following sequence of microoperations:

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

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

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

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

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

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

Memory Stack

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

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

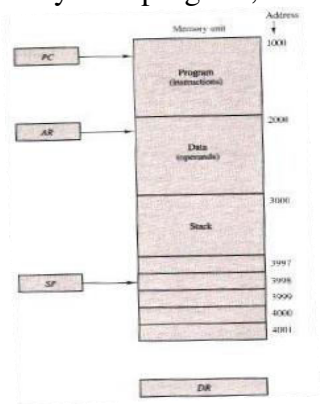
We assume that the items in the stack communicate with a data register DR. A new item is inserted with the push operation as follows.

SP←SP – 1

M[SP] ← DR

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

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



DR ←M[SP]

SP←SP + 1

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

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

Reverse Polish Notation

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

A*B + C*D

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

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

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

The reverse Polish notation is in a form suitable for stack manipulation. The expression A*B +

C* D

is written in reverse Polish notation as

AB * CD * +

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

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

(A*B) CD* +

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

(A*B) (C*D) +

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

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

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

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

AB + DE + C* F+*

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

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

(3*4) + (5*6)

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

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

4.2 Instruction Formats

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

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

1. An operation code field that specifics the operation to be performed.

2. An address field that designates a memory address or a processor register.
3. A mode field that specifies the way the operand or the effective address is determined.

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

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

Computers may have instructions of several different lengths containing varying number of addresses. The number of address fields in the instruction format of a computer depends on the internal organization of its registers. Most computers fall into one of three types of CPU organizations:

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

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

ADD X

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

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

R1, R2, R3

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

ADD R1, R2

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

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

MOV R1, R2

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

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

ADD R1, X

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

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

PUSH X

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

The instruction ADD

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

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

The arithmetic statement X = (A +

B) * (C + D)

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

Three – Address Instructions

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

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

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

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

Two–Address Instructions

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

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

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

One–Address Instructions

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

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

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

Zero – Address Instructions

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

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

RISC Instructions

The instruction set of a typical RISC processor is restricted to the use of load and store instructions when communicating between memory and CPU. All other instructions are executed within the registers of the CPU without referring to memory. A program for a RISC –type CPU consists of LOAD and STORE instructions that have one memory and one

memory and one register address, and computational–type instructions that have three addresses with all three specifying processor registers. The following is a program to evaluate X = (A+B) * (C + D)

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

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

4.5. Addressing Modes

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

1. To give programming versatility to the user by providing such facilities as pointers to

memory, counters for loop control, indexing of data, and program relocation.

2. To reduce the number of bits in the addressing field of the instruction.

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

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

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

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

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

Figure 4.5 Instruction format with mode field

Opcode	Mode	Address

Types of Addressing Modes:

1. Implied Mode
2. Immediate Mode
3. Register Mode
4. Register Indirect Mode:
5. Autoincrement or Autodecrement Mode
6. Direct Address Mode
7. Indirect Address Mode
8. Relative Address Mode
9. Indexed Addressing Mode
10. Base Register Addressing Mode

There are two modes that need **no address field at all**. These are the implied and immediate modes.

1. **Implied Mode:** In this mode the operands are specified implicitly in the definition ofthe instruction.

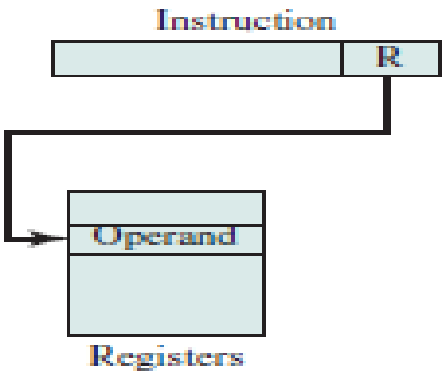
   For example, the instruction "complement accumulator (CMA)" is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction. In fact, all register reference instructions that use an accumulator are implied-mode instructions. Zero-address instructions in a stack-organized computer are implied-mode instructions since the operands are implied to be on top of the stack.

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

   EX: LDAC #34H
   LDAC loads data from memory to accumulator.
   Therefore, AC=00110100.

When the address field specifies a processor register, the instruction is said to be in the register mode.

3. **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.



E.g : LDAC R

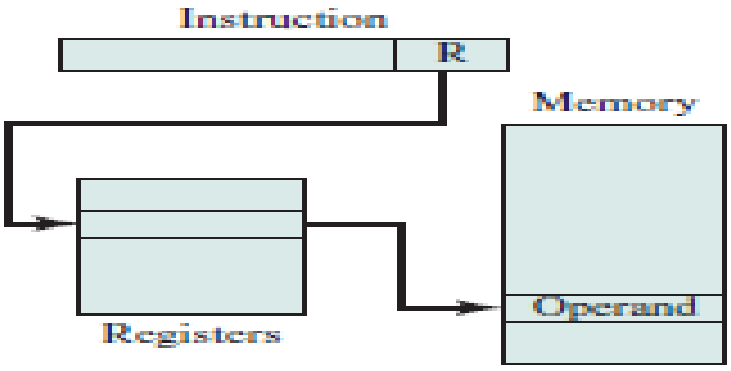
instruction reads data from register R

LDAC R

R : 5 → store value in Ac

∴ Ac = 5

4. **Register Indirect Mode:** In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory. In other words, the selected register contains the address of the operand rather than the operand itself.



EX: **LDAC (R1)**

If R1cotains the address of an operand in the memory, for example: address of an operand is 2000 which contains a value 350. Result: 350 is stored in the AC.

5. **Autoincrement 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 effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of the register are automatically incremented to the next value.

E.g : LDAC (R)

instruction reads address from register R.

R : 5

instruction reads data from memory location

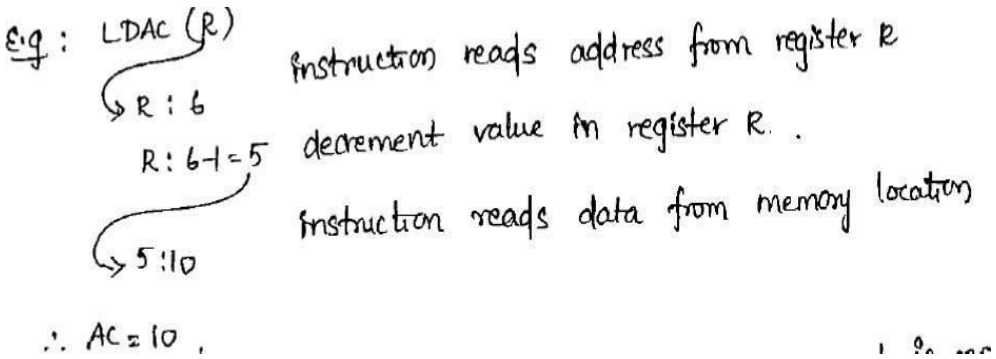
5 : 10 → store in AC

AC = 10

R : 5+1 = 6 .

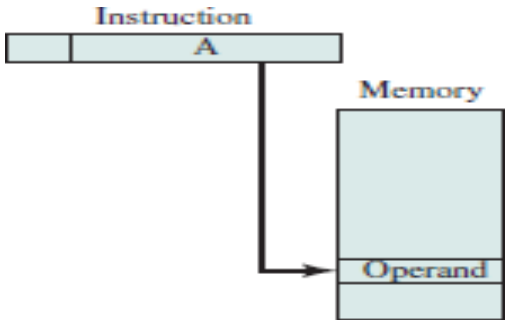
6. **Autodecrement Mode**

The effective address of the operand is the contents of a register specified in the instruction. Before accessing the operand, the contents of this register are automatically decremented and then the value is accessed.



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.

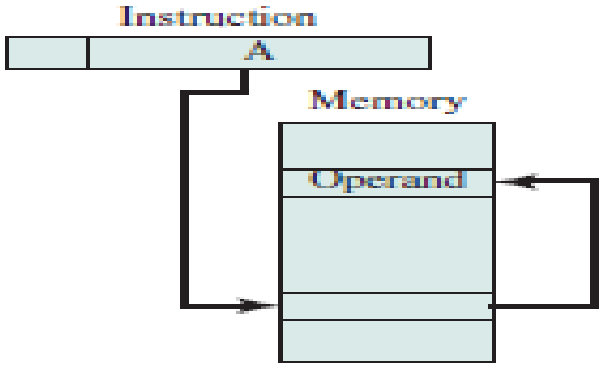
7. **Direct Address Mode:** In this mode the effective address is equal to the address part of the instruction. The operand resides in memory and its address is given directly by the address field of the instruction. In a branch-type instruction the address field specifies the actual branch address.



Ex**: LDAC 5000**

This instruction reads the operand from the Memory location 5000. if the memory location 5000 contains a value 250, then it will be stored in AC.

8. **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.



EX: ADD (A), R1

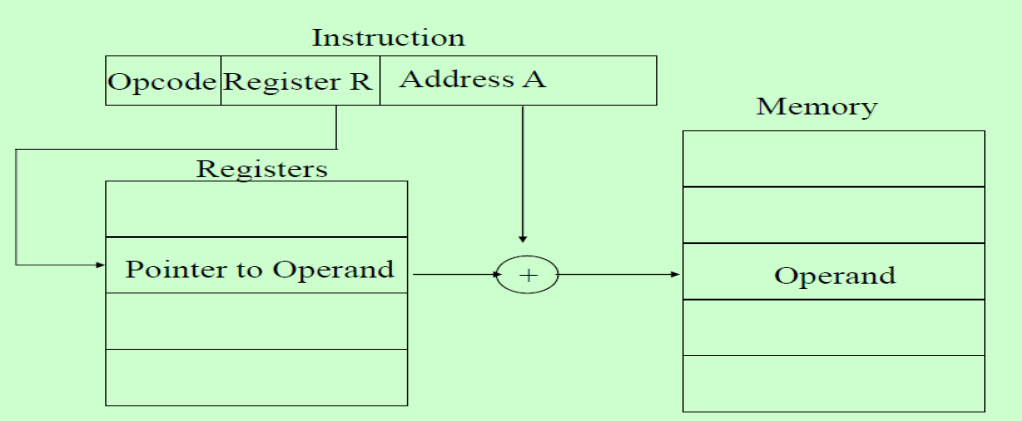
➔ If A is address of EA. For example: address of A is 1000 which contains 3000, 3000 is an address of an operand (EA).

➔ This instruction reads an operand from the location address 3000 and adds its contents to R1.

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:

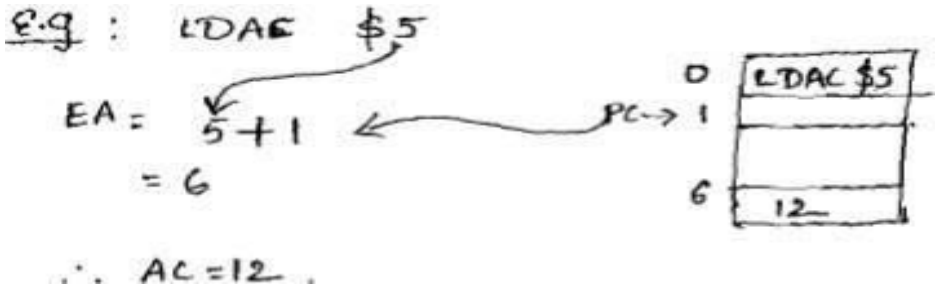
➔ **effective address = address part of instruction + content of CPU register**

The CPU register used in the computation may be the program counter, an index register, or a base register. In either case we have a different addressing mode which is used for a different application.



9. **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.



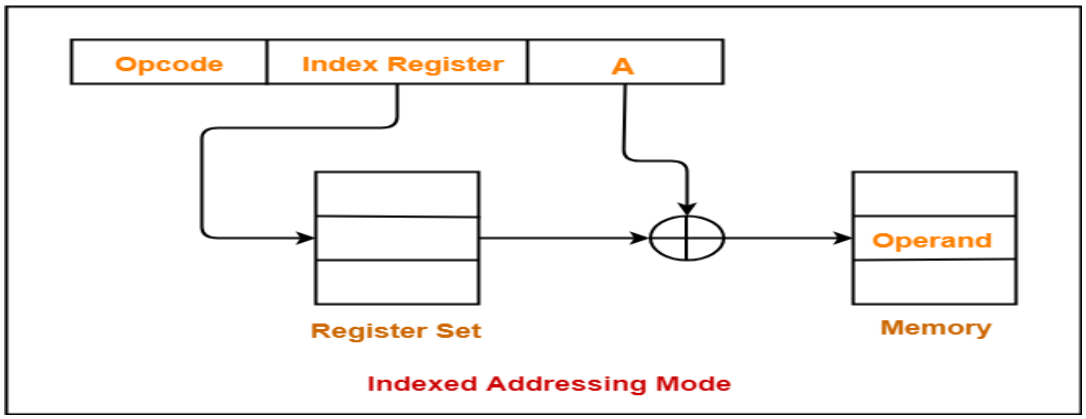
EX:

PC= address of next instruction , i.e., 1. The address given in the instruction is 5

Then EA= 5 + 1= 6 which contains a value 12. Finally AC contains 12.

10. **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.



Ex: LDAC A(XR)

Assume XR=100, A=500

This instruction reads the operand from the effective address (600)

i.e., EA= XR contents (index register) + 500

= 100 + 500=600

If memory location at 600 contains a value 55 (assume), This 55 will be stored in AC.

11. **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.

Ex: LDAC A(R)

Assume R=1000, A=50

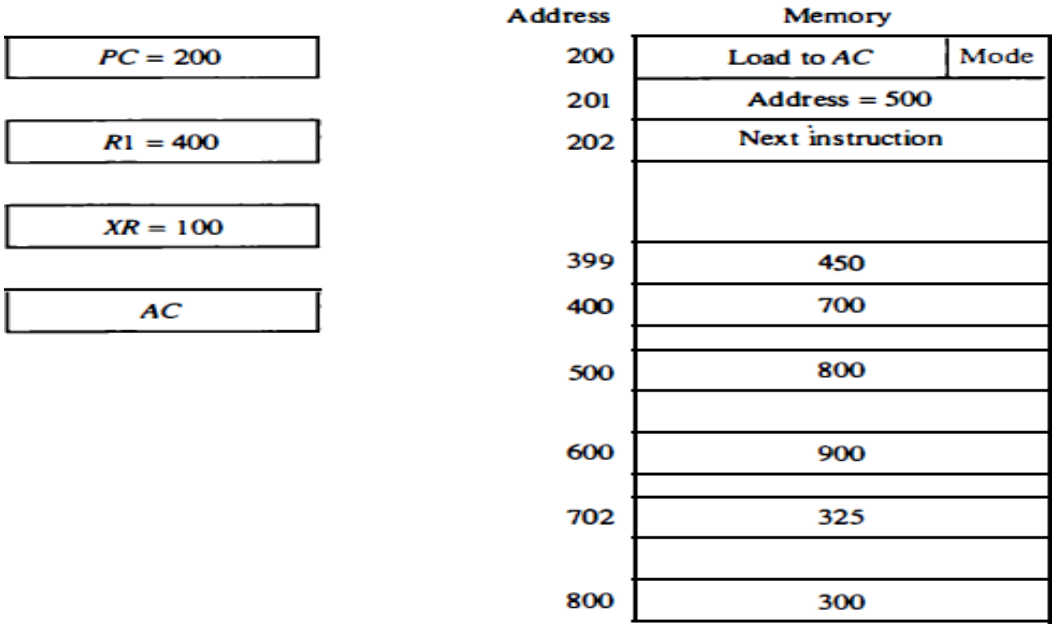
This instruction reads the operand from the effective address (1050)

i.e., EA= R contents (Base register) + 50

= 1000 + 50=1050

If memory location at 1050 contains a value 255 (assume), this 255 will be stored in AC.

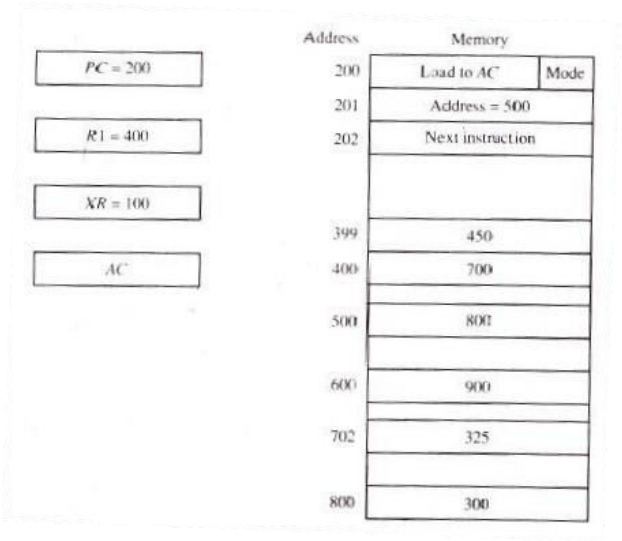
Numerical Example:



Numerical Example

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

Figure 4.6 Numerical example for addressing modes



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

4.6 Data Transfer and Manipulation:

Data Transfer and Manipulation Data transfer instructions cause transfer of data from one location to another without changing the binary information. The most common transfer are between the

- Memory and Processor registers
- Processor registers and input output devices
- Processor registers themselves

Typical Data Transfer Instructions

Name	Mnemonic
LOAD	LD
STORE	ST
MOVE	MOV
EXCHANGE	XCH
INPUT	IN
OUTPUT	OUT
PUSH	PUSH
POP	POP

Data manipulation Instructions

Data manipulation instructions perform operations on data and provide the computational capabilities for the computer. These instructions perform arithmetic, logic and shift operations.

Arithmetic Instructions

Arithmetic Instructions

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

Logical and Bit Manipulation Instructions

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

Shift Instructions

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

Program Control Instructions

The program control instructions provide decision making capabilities and change the path taken by the program when executed in computer. These instructions specify conditions for altering the content of the program counter. The change in value of program counter as a result of execution of program control instruction causes a break in sequence of instruction execution. Some typical program control instructions are:

Typical Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Skip next instruction	SKP
Call procedure	CALL
Return from procedure	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TEST

Subroutine call and Return A subroutine call instruction consists of an operation code together with an address that specifies the beginning of the subroutine. The instruction is executed by performing two tasks:

- The address of the next instruction available in the program counter (the return address) is stored in a temporary location (stack) so the subroutine knows where to return.
- Control is transferred to the beginning of the subroutine. The last instruction of every subroutine, commonly called return from subroutine; transfer the return address from the temporary location into the program counter. This results in a transfer of program control to the instruction where address was originally stored in the temporary location.

Interrupt

The interrupt procedure is, in principle, quite similar to a subroutine call except for three variations:

- The interrupt is usually initiated by an external or internal signal rather than from execution of an instruction.
- The address of the interrupt service program is determined by the hardware rather than from the address field of an instruction.

• An interrupt procedure usually stores all the information necessary to define the state of the CPU rather than storing only the program counter.

4.7 Program Control:

After the execution of a data transfer or data manipulation instruction, control returns to the fetch cycle with the program counter containing the address of the instruction next in sequence.

On the other hand, a program control type of instruction, when executed, may change the address value in the program counter and cause the flow of control to be altered. In other words, program control instructions specify conditions for altering the content of the program counter, while data transfer and manipulation instructions specify conditions for data-processing operations.

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 is an important feature in digital computers, as it provides control over the flow of program execution and a capability for branching to different program segments.

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

Table (9) : Program Control Instruction

Status Bit Conditions:

It is sometimes convenient to supplement the ALU circuit in the CPU with a status register where status bit conditions can be stored for further analysis. Status bits are also called condition-code bits or flag bits. Figure (6) shows the block diagram of an 8-bit ALU with a 4-bit status register. The four status bits are symbolized by C. S, Z, and V. The bits are set or cleared as a result of an operation performed in the ALU.

1. Bit **C (carry)** is set to 1 if the end carry C_8 is 1. It is cleared to 0 if the carry is 0.
2. Bit **S (sign)** is set to 1 if the highest-order bit F_7 is 1. It is set to 0 if the bit is 0.
3. Bit **Z (zero)** is set to 1 if the output of the ALU contains all 0's. It is cleared to 0 otherwise. In other words, Z = 1 if the output is zero and Z = 0 if the output is not zero.
4. Bit **V (overflow)** is set to 1 if the exclusive-OR of the last two carries is equal to 1, and cleared to 0 otherwise. This is the condition for an overflow when negative numbers are in 2's complement. For the 8-bit ALU, V = 1 if the output is greater than + 127 or less than – 128.

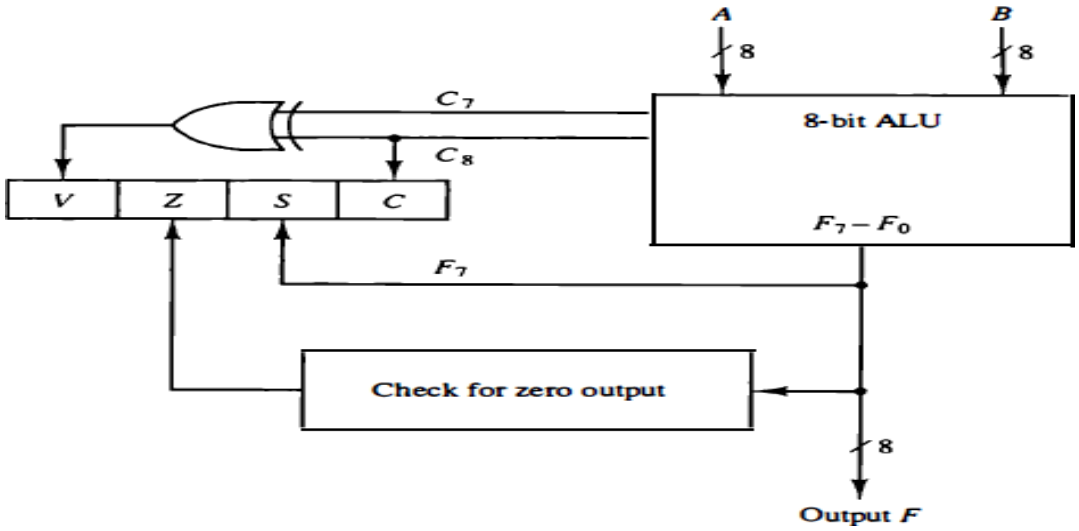


Figure (6): Status register bits

Conditional Branch Instructions:

- To change the flow of execution in the program we use some kind of branching instructions which are depending on the some conditions result.
- Each mnemonic is constructed with the letter **B** (for branch) and an abbreviation of the condition name. When the opposite condition state is used, the letter **N** (for no) is inserted to define the 0 state. Thus **BC** is Branch on Carry, and **BNC** is Branch on No Carry.
- If the stated condition is true, program control is transferred to the address

specified by the instruction. If not, control continues with the instruction that follows. The conditional instructions can be associated also with the jump, skip, call, or return type of program control instructions.

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	$Z = 1$
BNZ	Branch if not zero	$Z = 0$
BC	Branch if carry	$C = 1$
BNC	Branch if no carry	$C = 0$
BP	Branch if plus	$S = 0$
BM	Branch if minus	$S = 1$
BV	Branch if overflow	$V = 1$
BNV	Branch if no overflow	$V = 0$
	Unsigned compare conditions ($A - B$)	
BHI	Branch if higher	$A > B$
BHE	Branch if higher or equal	$A \geq B$
BLO	Branch if lower	$A < B$
BLOE	Branch if lower or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$
	Signed compare conditions ($A - B$)	
BGT	Branch if greater than	$A > B$
BGE	Branch if greater or equal	$A \geq B$
BLT	Branch if less than	$A < B$
BLE	Branch if less or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$

Table (10): Conditional Branch Instructions

Example: Consider an 8-bit ALU as shown in Figure (6). The largest unsigned number that can be accommodated in 8 bits is 255. The range of signed numbers is between + 127 and -128. Let A = 11110000 and B = 00010100. To perform A - B, the ALU takes the 2's complement of B and adds it to A.

$$\begin{array}{rl} A: & 11110000 \\ \overline{B} + 1: & +11101100 \\ \hline A - B: & 11011100 \end{array} \qquad C = 1 \quad S = 1 \quad V = 0 \quad Z = 0$$

The compare instruction updates the status bits as shown. C = 1 because there is a carry out of the last stage. S = 1 because the leftmost bit is 1. V = 0 because the last two carries are both equal to 1, and Z = 0 because the result is not equal to 0.

Subroutine Call and Return:

A subroutine is a self-contained sequence of instructions that performs a given computational task. During the execution of a program, a subroutine may be called to perform its function many times at various points in the main program. Each time a subroutine is called, a branch is executed to the beginning of the subroutine to start executing its set of instructions. After the subroutine has been executed, a branch is made back to the main program.

The instruction that transfers program control to a subroutine is known by different names. The most common names used are **call subroutine**, **jump to subroutine**, **branch to subroutine**, or **branch and save address**.

A call subroutine instruction consists of an operation code together with an address that specifies the beginning of the subroutine. The instruction 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***, and

(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 into the program counter. This results in a transfer of program control to the instruction whose address was originally stored in the temporary location.

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. The return from subroutine instruction causes the stack to pop and the contents of the top of the stack are transferred to the program counter. In this way, the return is always to the program that last called a subroutine. 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

If another subroutine is called by the current subroutine, the new return address is

pushed into the stack, and so on. 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

By using a subroutine stack, all return addresses are automatically stored by the hardware in one unit. The programmer does not have to be concerned or remember where the return address was stored.

→ A ***recursive subroutine*** is a subroutine that calls itself.

Program interrupt:

Program interrupt refers to the transfer of program control from a currently running program to another service program as a result of an external or internal generated request. Control returns to the original program after the service program is executed.

The interrupt procedure is, in principle, quite similar to a subroutine call except for three variations:

(1) The interrupt is usually initiated by an internal or external signal rather than from the execution of an instruction (except for software interrupt as explained later);

(2) The address of the interrupt service program is determined by the hardware rather than from the address field of an instruction; and

(3) An interrupt procedure usually stores all the information necessary to define the state of the CPU rather than storing only the program counter.

The state of the CPU at the end of the execute cycle (when the interrupt is recognized) is determined from:

1. The content of the program counter
2. The content of all processor registers
3. The content of certain status conditions

The collection of all status bit conditions in the CPU is sometimes called a **program status word or PSW**. The PSW is stored in a separate hardware register and contains the status information that characterizes the state of the CPU

The last instruction in the service program is a return from interrupt instruction. When this instruction is executed, the stack is popped to retrieve the old PSW and the return address. The PSW is transferred to the status register and the return address to the program counter. Thus the CPU state is restored and the original

program can continue executing.

There are three major types of interrupts that cause a break in the normal execution of a program. They can be classified as:

1. External interrupts
2. Internal interrupts
3. Software interrupts

External interrupts come from input-output (l/0) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source. Examples that cause external interrupts are l/0 device requesting transfer of data, l/0 device finished transfer of data etc.

Internal interrupts arise from illegal or erroneous use of an instruction or data. Internal interrupts are also called traps . Examples of interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation.

→External and internal interrupts are initiated from signals that occur in the hardware of the CPU.

A software interrupt is initiated by executing an instruction. Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program. The most common use of software interrupt is associated with a supervisor call instruction. This instruction provides means for switching from a CPU user mode to the supervisor mode.

4.8 Reduced Instruction Set

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

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

CISC Characteristics

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

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

The instructions in a typical CISC processor provide direct manipulation of operands residing in memory. For example, an ADD instruction may specify one operand in memory through index addressing and a second operand in memory through a direct

addressing. Another memory location may be included in the instruction to store the sum. This requires three memory references during execution of the instruction. As more instructions and addressing modes are incorporated into a computer, the more hardware logic is needed to implement and support them, and this may cause the computations to slow down. Major characteristics of CISC architecture are:

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

RISC Characteristics

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

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

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

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

Other characteristics attributed to RISC architecture are:

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

A large number of registers is useful for storing intermediate results and for

optimizing operand references. The advantage of register storage as op-posed to memory storage is that registers can transfer information to other registers much faster than the

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

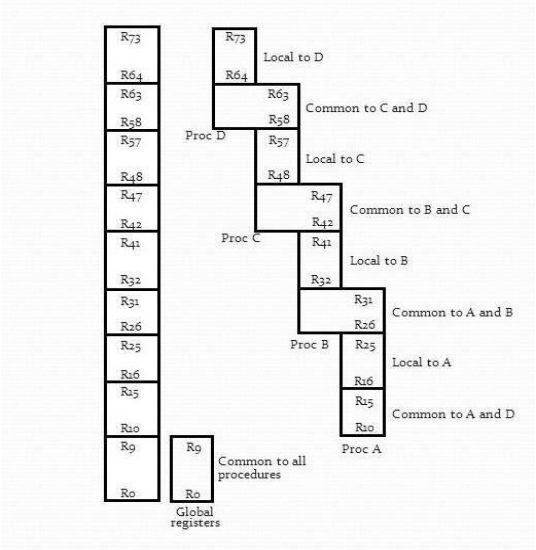
Overlapped Register Windows

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

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

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

Figure 4.7 Overlapped Register Windows



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

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

number of global registers = G

number of local registers in each window = L

number of registers common to two windows = C

number of windows = W

The number registers available for each window is calculated as folk

window size = L + 2C + G

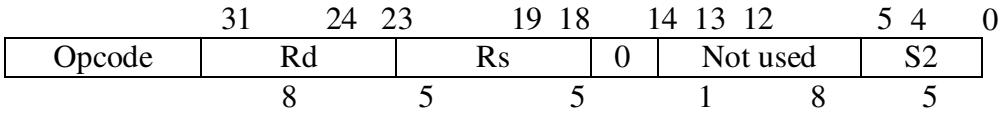
The total number of registers needed in the processor is

register file = (L + C)W + G

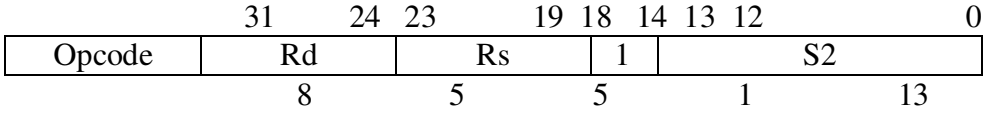
Berkeley RISC I

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

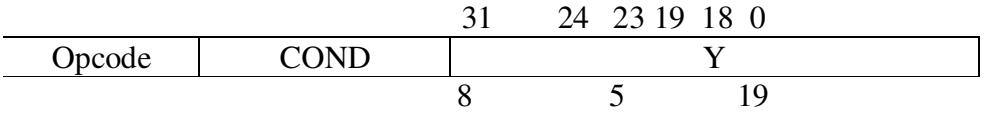
Figure 4.8 Berkeley RISC I instruction formats.



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



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



(C) P C relative mode

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

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

UNIT III

5. COMPUTER ARITHMETIC

5.1 Addition and Subtraction

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

Addition and Subtraction with Signed–Magnitude Numbers

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

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

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

Eight Conditions for Signed-Magnitude Addition/Subtraction

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

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

Hardware Implementation

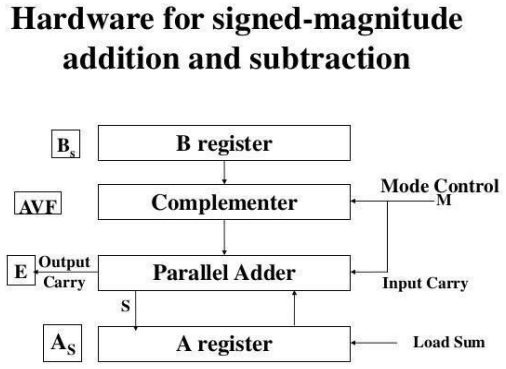
Let A and B be two registers that hold the magnitudes of the numbers, and $A_s$and B_s be two flip–flops that hold the corresponding signs. The result of the operation may be transferred to a third register,however, a saving is achieved if the result is transferred into A and As. Thus A and A_s together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

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

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

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

Figure 5.1 Hardware for signed magnitude addition and subtraction



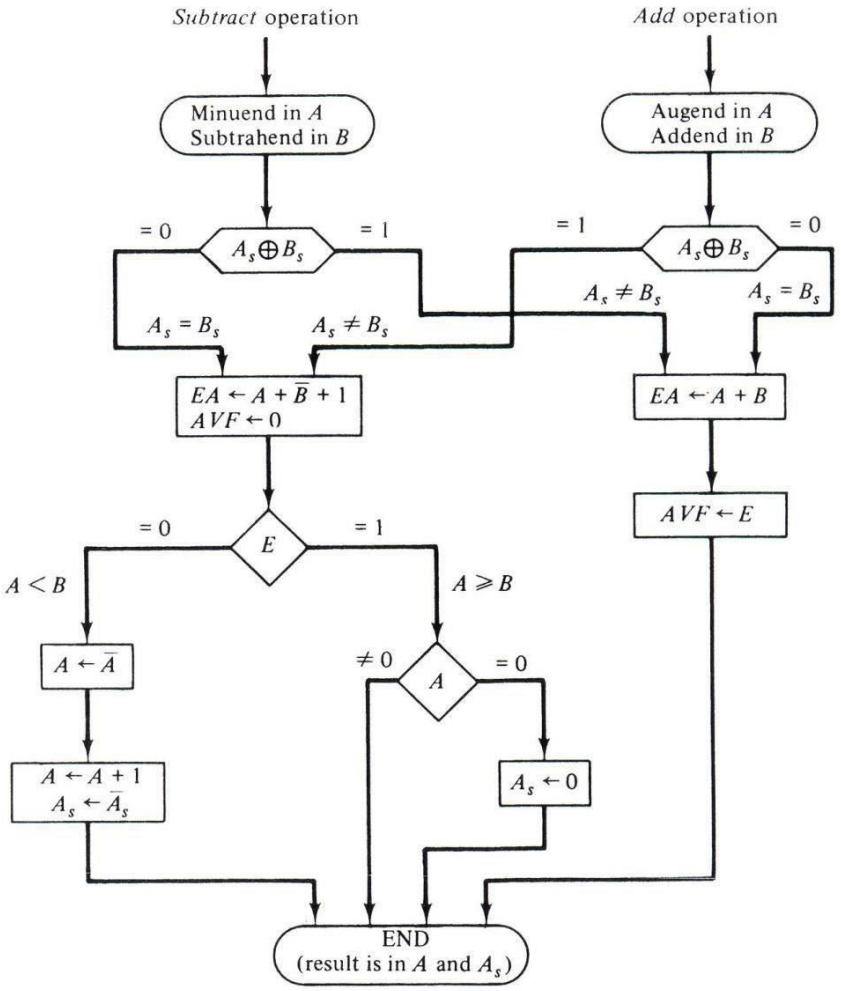
When M =1, the 1‘s complement of B is applied to the adder, the input carry is 1, and output S = A + B + 1. This is equal to A plus the 2‘s complement of B, which is equivalent to the subtraction A –B.

The flowchart for the hardware algorithm is presented in Fig. 5.2. The two signs A_s and B_s are compared by an exclusive–OR gate. If the output of the gate is 0, the signs are

identical; if it is 1, the signs are different. For an add operation, identical signs dictate that the magnitudes be added. For a subtract operation, different signs dictate that the magnitudes be added. The magnitudes are added with a microoperation EA ← A + B, where EA is a register that combines E and A. The carry in E after the addition constitutes an overflow if it is equal to 1. The value of E is transferred into the add-overflow flip – flop AVF.

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

Figure 5.2 Flow chart for add and subtract operations



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

The leftmost bit of a binary number represents the sign bit : 0 for positive and 1 for negative. If the sign bit is 1, the entire number is represented in 2‘s complement form. The addition of two numbers in signed -2‘s complement form consists of adding the numbers with the sign bits treated the same as the other bits of the number. A carry – out of the sign – bit position is discarded. The subtraction consists of first taking the 2‘s complement of the subtrahend and then adding it to the minuend. When two numbers of n digits each are added and the sum occupies n+1 digits, we say that an overflow occurred. An overflow can be detected by inspecting the last two carried out of the addition. When the two carries are applied to an exclusive–OR gate, the overflow is detected when the output of the gate is equal to 1.

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

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

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

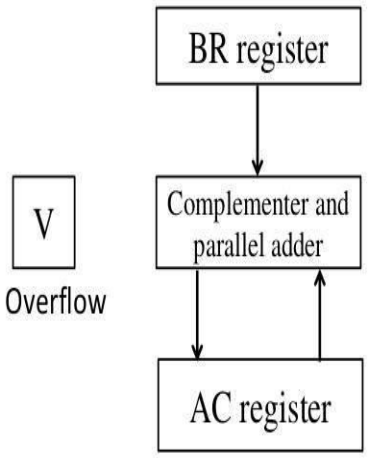
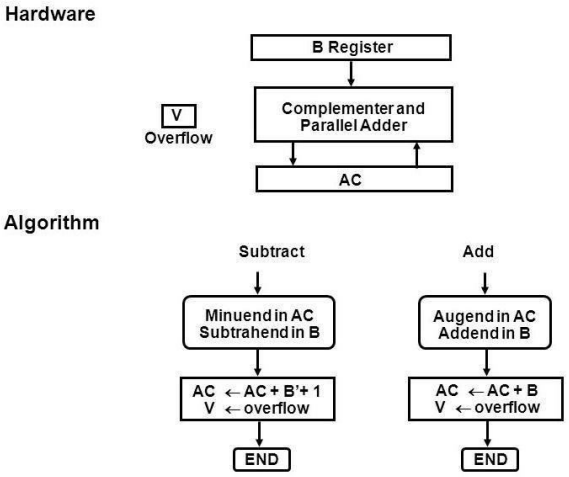


Figure 5.4 Algorithm for adding and subtracting numbers in signed -2‘s complement representation.



Comparing this algorithm with its signed – magnitude counterpart, we note that it is much simpler to add and subtract numbers if negative numbers are maintained in signed -2‘s complement representation. For this reason most computers adopt this representation over the more familiar signed–magnitude.

5.2 Multiplication Algorithms

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

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

If the multiplier bit is a 1, the multiplicand is copied down; otherwise, zeros are

copied down. The numbers copied down in successive lines are shifted one position to the left from the previous number. Finally, the numbers are added and their sum forms the product.

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

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

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

Figure 5.5 Hardware for multiply operation.

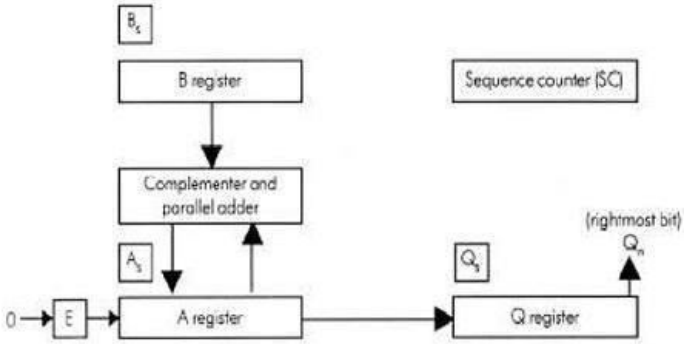


Figure 5.6 is a flowchart of the hardware multiply algorithm. Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in Bs and Q_s, respectively. The signs are compared , and both A and Q are set to correspond to the sign of the product since a double length product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

word will be occupied by the sign and the magnitude will consist on n -1 bits.

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

Figure 5.6 Flowchart for multiply operation

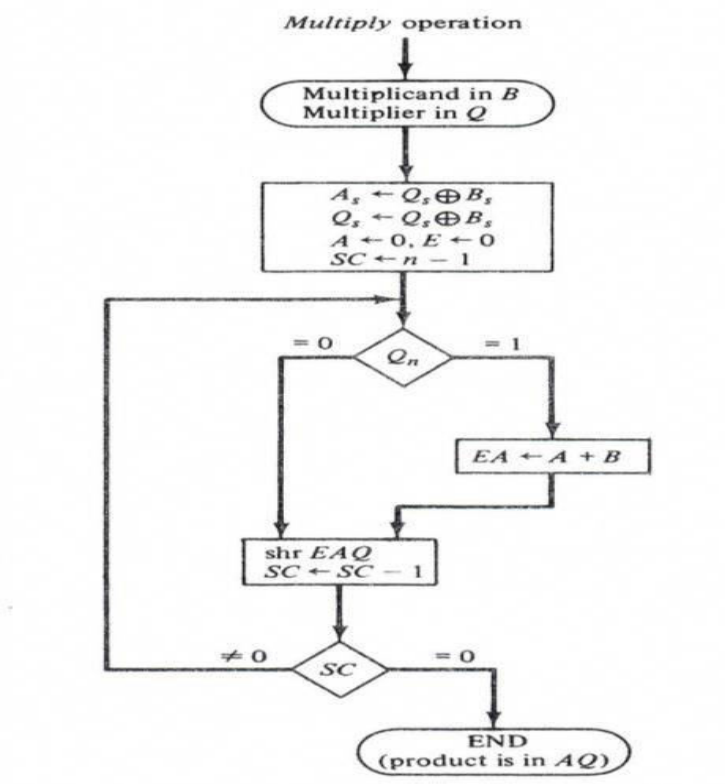


Table 5-2 Numerical example for binary multiplier

Multiplicand B = 10111

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

Booth Multiplication Algorithm

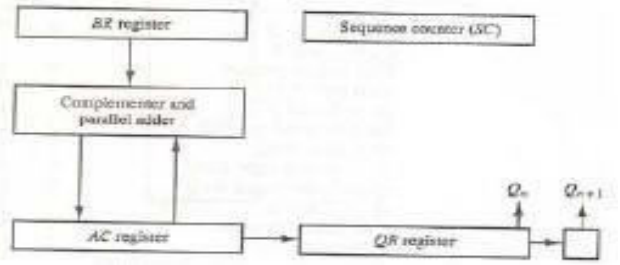
Booth algorithm gives a procedure for multiplying binary integers in signed -2‘s complement representation. It operates on the fact that strings of 0‘s in the multiplier require no addition but just shifting, and a string of 1‘s in the multiplier from bit weight 2^k to 2^m can be treated2^{k+1}-2^m.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

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

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

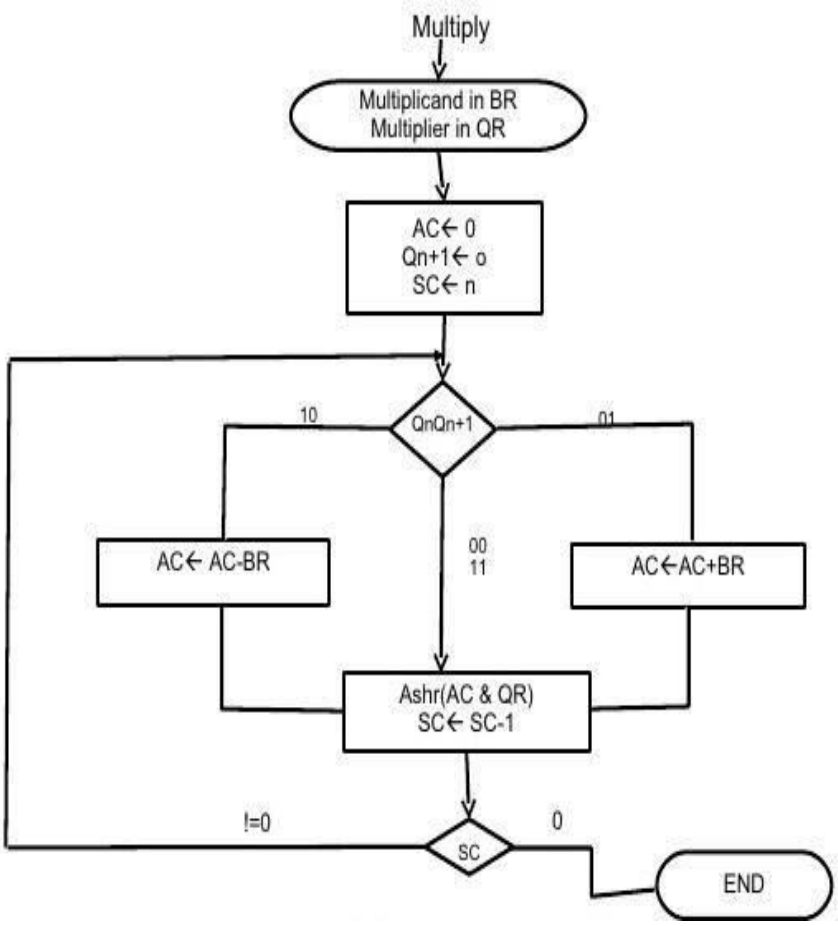
The hardware implementation of Booth algorithm requires the register configuration shown in Fig. 5.7. We rename registers A, B and Q, as AC, BR, and QR, respectively. Q_n designates the least significant bit of the multiplier in register QR. An extra flip – flop $Q_{n+1}$is appended to QR to facilitate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

Figure 5.7 Hardware for Booth algorithm



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

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



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

TABLE 5-3 Example of Multiplication with Booth Algorithm

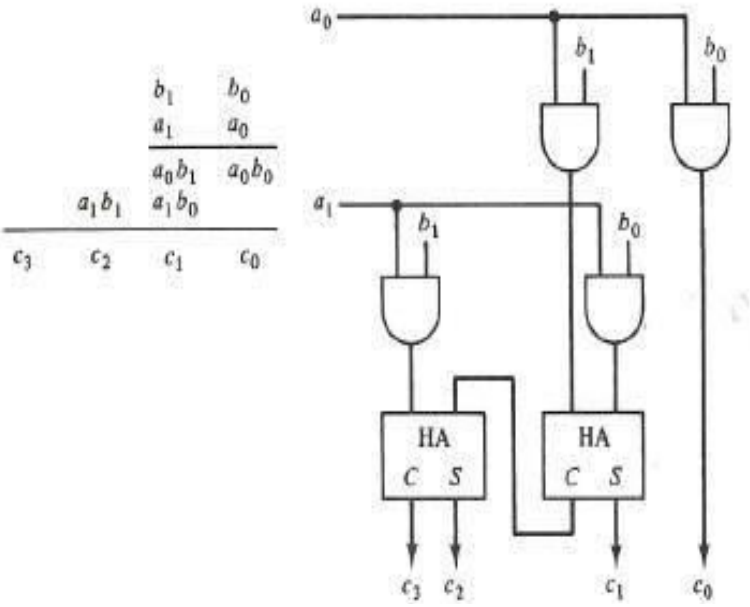
Q_nQ_{n+1}	BR=10111 BR+ 1=01001	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract BR	01001			
		01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	10111			
		11001			010
	ashr	11100	10110	0	001
0 0	ashr	11110	01011	0	
1 0	Subtract BR	01001			
		00111			
	ashr	00011	10101	1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

Array Multiplier

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

To see how an array multiplier can be implemented with a combinational circuit, consider the multiplication of two 2 – bit numbers as shown in Fig. 5.9. The multiplicand bits are b_1 and b_0, the multiplier bits are a_1 and a_0, and the product is $c_3c_2c_1c_0$. The first partial product is formed by multiplying a0 by b1b_0. The multiplication of two bits such as a_0 and b_0 produces a 1 if both bits are 1; otherwise, it produces a 0. This is identical to an AND operation and can be implemented with an AND gate. As shown in the diagram, the first partial product is formed by means of two AND gates. The second partial product is formed by multiplying a_1 by b_1b_0 and is shifted one position to the left. The two partial products are added with two half – adder (HA) circuits. Usually, there are more bits in the partial products and it will be necessary to use full–adders to produce the sum. Note that the least significant bit of the product does not have to go through an adder since it is formed by the output of the product does not have to go through an adder since it is formed by the output of the first AND gate.

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



5.3 Division Algorithms

Division of two fixed – point binary numbers in signed – magnitude representation is done with paper and pencil by a process of successive compare, shift, and subtract operations. Binary division is simpler than decimal division because the quotient digits are either 0 or 1 and there is no need to estimate how many times the dividend or partial remainder fits into

the divisor. The division process is illustrated by a numerical example in Fig. 5.10. The divisor B consists of five bits and the dividend A, of ten bits. The five most significant bits of the dividend are compared with the divisor. Since the 5-bits of A and compare this number with B. The 6 bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend. The difference is called a partial remainder because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder. The process is continued by comparing a partial remainder with the divisor. If the partial remainder is greater than or equal to the divisor, the quotient bit is equal to 1. The divisor is then shifted right and subtracted from the partial remainder. If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case. Note that the result gives both a quotient and a remainder.

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

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

Figure 5.10 Example of binary division

Divisor:

B=10001

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

Figure 5.11 Example of binary division with digital hardware

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

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

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

Divide Overflow

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

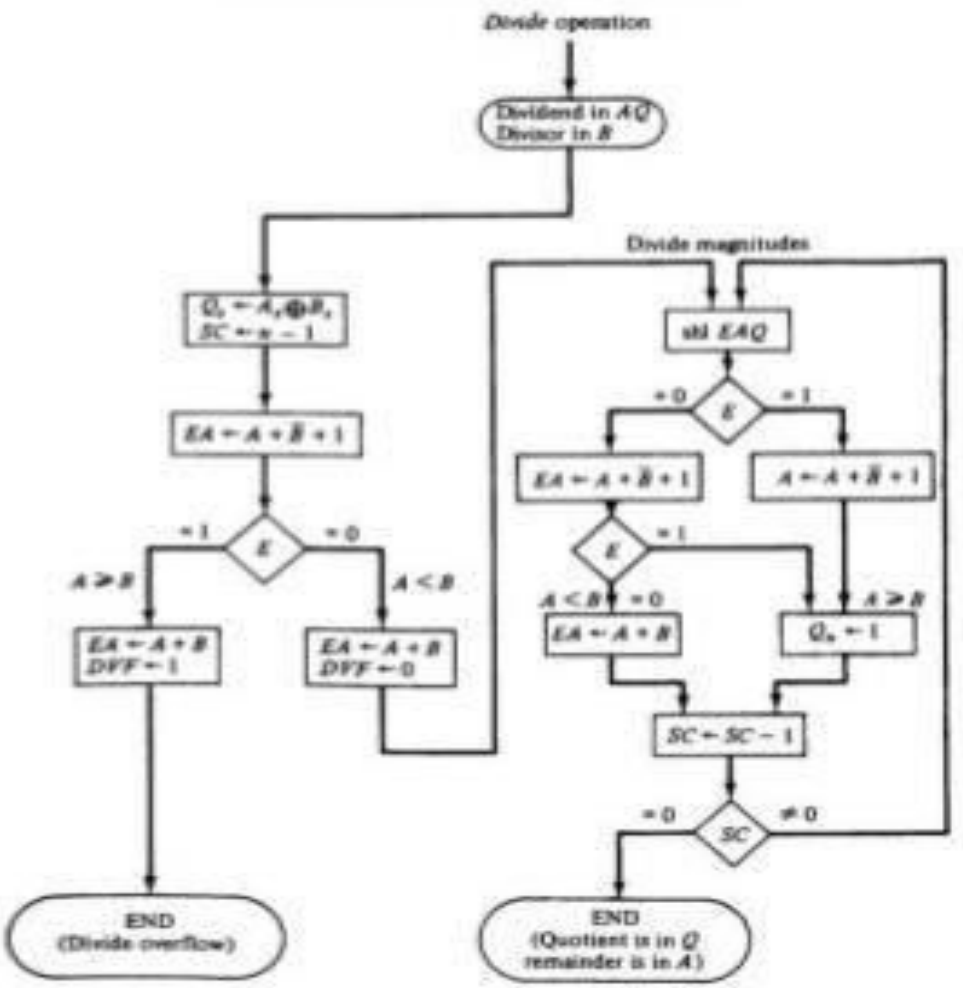
When the dividend is twice as long as the divisor, the condition for overflow can be stated as follows: A divide–overflow condition occurs if the high order half bits of the dividend constitute a number greater than or equal to the divisor. Another problem associated with division is the fact that a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

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

Hardware Algorithm

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

Figure 5.12 Flowchart for divide operation



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

dividend is restored by adding B to A.

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

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

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

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

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

Other Algorithms

The hardware method just described is called the restoring method. The reason for this name is that the partial remainder is restored by adding the divisor to the negative difference. Two other methods are available for dividing numbers, the comparison method and the non restoring method. In the comparison method A and B are compared prior to the

subtraction operation. Then If $A \geq B$, B is subtracted from A. If $A < B$ nothing is done.

The
partial remainder is shifted left and the numbers are compared again. The comparison can be
determined prior to the subtraction by inspecting the end–carry out of the parallel–adder prior to its transfer to register E.

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

5.4 Float Point Arithmetic Operations

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

Basic Considerations

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

$$m \times r^e$$

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

$$.53725 \times 10^3$$

A floating – point number is normalized if the most significant digit of the mantissa is nonzero. In this way the mantissa contains the maximum possible number of

significant digits. A zero cannot be normalized because it does not have a nonzero digits. A zero cannot be normalized because it does not have a nonzero digit. It is represented in floating–point by all 0‘s in the mantissa and exponent.

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

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

$$.5372400 \times 10^{2}$$
$$+ .1580000 \times 10^{-1}$$

It is necessary that the two exponents be equal before the mantissas can be added. We can either shift the first number three positions to the left, or shift the second number three positions to the right. When the mantissas are stored in registers, shifting to the left causes a
loss of most significant digits. Shifting to the right causes a loss of least significant digits. The second method is preferable because it only reduces the accuracy, while the first method may cause an error. The usual alignment procedure is to shift the mantissa that has the smaller exponent to the right by a number of places equal to the difference between the exponents. After this is done, the mantissas can be added :

$$\begin{array}{r} .5372400 \times 10^{2} \\ \underline{+.0001580 \times 10^{2}} \\ .5373980 \times 10^{2} \end{array}$$

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

$$\begin{array}{r} .56780 \times 10^{5} \\ \underline{-.56430 \times 10^{5}} \\ .00350 \times 10^{5} \end{array}$$

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

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

The operations performed with the exponents are compare and increment (for aligning the mantissas), add and subtract (for multiplication and division), and

decrement (to normalize the result). The exponent may be represent in any one of the three representation : signed – magnitude, signed – 2‘s complement, or signed -1‘s complement.

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

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

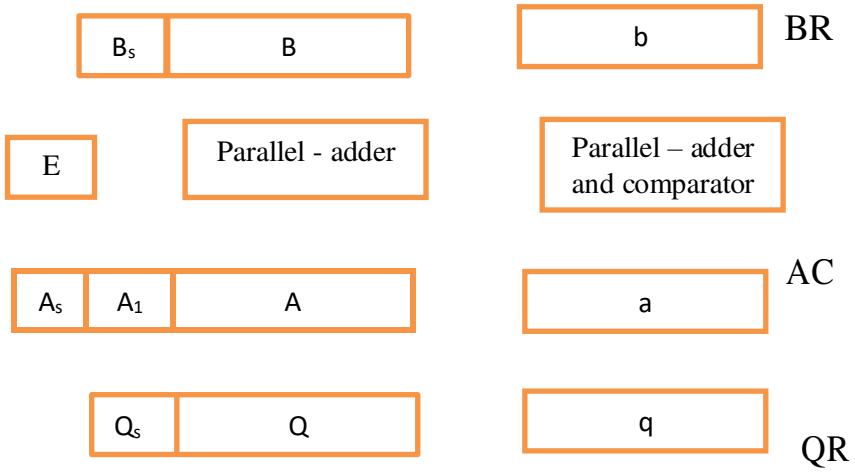
Register Configuration

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

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

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

Figure 5.13 Registers for floating–point arithmetic operation



whose sign is in A_s and a magnitude that is in A. The exponent is in the part of the register denoted by the lowercase letter symbol a. The diagram shows explicitly the most significant bit of A, labeled by A_1. The bit in this position must be a 1 for the number to be normalized. Note that the symbol AC represents the entire register, that is, the concatenation of A_s, A and
a. Register BR is subdivided into Bs, B and b, and QR into Qs, Q, and q. A parallel–adder adds the two mantissas and transfers the sum into A and the carry into E. A separate parallel adder is used for the exponents. Since the exponents are biased, they do not have a distinct sign bit but are represented as a biased positive quantity. The

floating–point numbers are so large that the chance of an exponent overflow is very remote, the exponent overflow will be neglected. The exponents are also connected to a magnitude comparator that provides three binary outputs to indicate their relative magnitude.

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

Addition and Subtraction

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

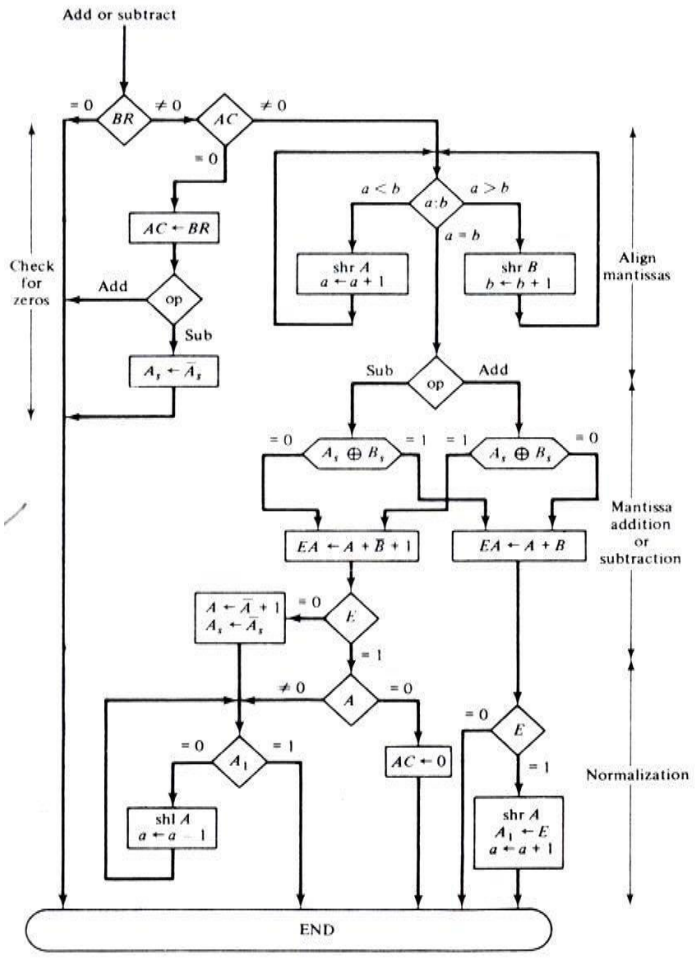
1. Check for zeros
2. Align the mantissas
3. Add or subtract the mantissas
4. Normalize the result

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

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

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

Figure 5.14 Addition and subtraction of floating point numbers



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

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

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

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

Multiplication

The multiplication of the mantissas is performed in the same way as in fixed–point to provide a double – precision product. The double–precision answer is used in fixed–point numbers to increase the accuracy of the product.

In floating–point, the range of a single precision mantissa combined with the exponent is usually accurate enough so that only single precision numbers are maintained. Thus the half most significant bits of the mantissa product and the exponent will be taken together to form a single precision floating–point product.

The multiplication algorithm can be subdivided into four parts.

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

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

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

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

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

The product may have an underflow, so the most significant bit in A is checked. If it is a 1, the product is already normalized. If it is a 0, the mantissa in AQ is shifted left and the exponent decremented. Note that only one normalization shift is necessary. The multiplier and multiplicand were originally normalized and contained fractions. The smallest normalized operand is 0.1, so the smallest possible product is 0.01. Therefore, only one leading zero may occur. Although the low – order half of the mantissa is in Q, we do not use it for the floating point product. Only the value in the AC is taken as the product.

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

If the dividend is greater than or equal to the divisor, the dividend fraction is shifted

to the right and its exponent incremented by 1.

The division algorithm can be subdivided into five parts

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

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

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

The dividend alignment is similar to the divide – overflow check in the fixed – point operation. The proper alignment requires that the fraction dividend be smaller than the divisor. The two fractions are compared by a subtraction test. The carry in E determines their relative magnitude. The dividend fraction is restored to its original value by adding the

divisor. If A $\geq$B, it is necessary to shift A once to the right and increment the dividend
exponent. Since both operands are normalized, this alignment ensures that A < B.

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

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

UNIT IV
THE MEMORY SYSTEM

4.1 BASIC CONCEPTS

The maximum size of the memory that can be used in any computer is determined by the addressing scheme.

For example, a 16-bit computer that generates 16-bit addresses is capable of addressing upto 2^{16} =64K memory locations. If a machine generates 32-bit addresses, it can access upto 2^{32} = 4G memory locations. This number represents the size of address space of the computer.

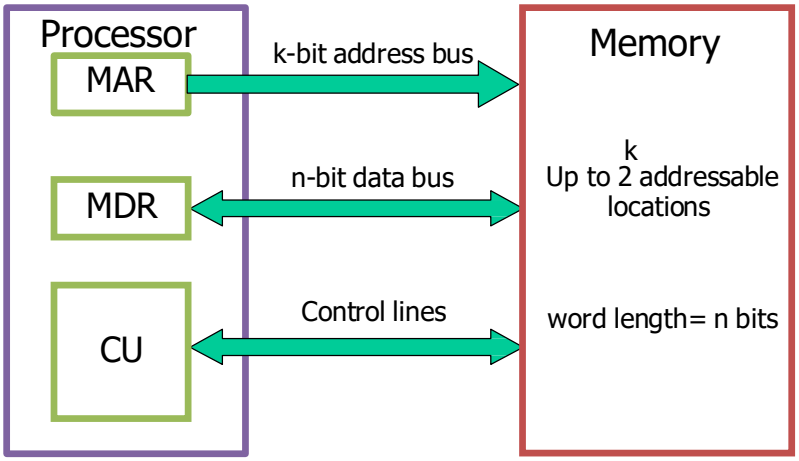
Address	Memory Locations
16 Bit	2^{16} = 64 K
32 Bit	2^{32} = 4G (Giga)
40 Bit	2^{64} = IT (Tera)

If the smallest addressable unit of information is a memory word, the machine is called word-addressable. If individual memory bytes are assigned distinct addresses, the computer is called byte-addressable. Most of the commercial machines are byte-addressable. For example in a byte-addressable 32-bit computer, each memory word contains 4 bytes. A possible word-address assignment would be:

Word Address	Byte Address			
0	0	1	2	3
4	4	5	6	7
8	8	9	10	11
.		…..		
.		…..		
.		…..		

With the above structure a READ or WRITE may involve an entire memory word or it may involve only a byte. In the case of byte read, other bytes can also be read but ignored by the CPU. However, during a write cycle, the control circuitry of the MM must ensure that only the specified byte is altered. In this case, the higher-order 30 bits can specify the word and the lower-order 2 bits can specify the byte within the word.

CPU-Main Memory Connection – A block schematic: -



From the system standpoint, the Main Memory (MM) unit can be viewed as a "block box". Data transfer between CPU and MM takes place through the use of two CPU registers, usually called MAR (Memory Address Register) and MDR (Memory Data Register). If MAR is K bits long and MDR is 'n' bits long, then the MM unit may contain upto 2^k addressable locations and each location will be 'n' bits wide, while the word length is equal to 'n' bits. During a "memory cycle", n bits of data may be transferred
between the MM and CPU. This transfer takes place over the processor bus, which has k address lines (address bus), n data lines (data bus) and control lines like Read, Write, Memory Function completed (MFC), Bytes specifiers etc (control bus). For a read operation, the CPU loads the address into MAR, set READ to 1 and sets other control signals if required. The data from the MM is loaded into MDR and MFC is set to 1. For a write operation, MAR, MDR are suitably loaded by the CPU, write is set to 1 and other control signals are set suitably. The MM control circuitry loads the data into appropriate locations and sets MFC to 1. This organization is shown in the following block schematic

Some Basic Concepts

Memory Access Times: -

It is a useful measure of the speed of the memory unit. It is the time that elapses between the initiation of an operation and the completion of that operation (for example, the time between READ and MFC).

Memory Cycle Time :-

It is an important measure of the memory system. It is the minimum time delay required between the initiations of two successive memory operations (for example, the time between two successive READ operations). The cycle time is usually slightly longer than the access time.

RANDOM ACCESS MEMORY (RAM):

A memory unit is called a Random Access Memory if any location can be accessed for a READ or WRITE operation in some fixed amount of time that is independent of the location's address. Main memory units are of this type. This distinguishes them from serial or partly serial access storage devices such as magnetic tapes and disks which are used as the secondary storage device.

Cache Memory:-

The CPU of a computer can usually process instructions and data faster than they can be fetched from compatibly priced main memory unit. Thus the memory cycle time become the bottleneck in the system. One way to reduce the memory access time is to use cache memory. This is a small and fast memory that is inserted between the larger, slower main memory and the CPU. This holds the currently active segments of a program and its data. Because of the locality of address references, the CPU can, most of the time, find the relevant information in the cache memory itself (cache hit) and infrequently needs access to the main memory (cache miss) with suitable size of the cache memory, cache hit rates of over 90% are possible leading to a cost-effective increase in the performance of the system.

Memory Interleaving: -

This technique divides the memory system into a number of memory modules and arranges addressing so that successive words in the address space are placed in different modules. When requests for memory access involve consecutive addresses, the access will be to different modules. Since parallel access to these modules is possible, the average rate of fetching words from the Main Memory can be increased.

Virtual Memory: -

In a virtual memory System, the address generated by the CPU is referred to as a virtual or logical address. The corresponding physical address can be different and the required mapping is implemented by a special memory control unit, often called the memory management unit. The mapping function itself may be changed during program execution according to system requirements.

Because of the distinction made between the logical (virtual) address space and the physical address space; while the former can be as large as the addressing capability of the CPU, the actual physical memory can be much smaller. Only the active portion of the virtual address space is mapped onto the physical memory and the rest of the virtual address space is mapped onto the bulk storage device used. If the addressed information is in the Main Memory (MM), it is accessed and execution proceeds. Otherwise, an exception is generated, in response to which the memory management unit transfers a contigious block of words containing the desired word from the bulk storage unit to the MM, displacing some block that is currently inactive. If the memory is managed in such a way that, such transfers are required relatively infrequency (ie the CPU will generally find the required information in the MM), the virtual memory system can provide a reasonably good performance and succeed in creating an illusion of a large memory with a small, in expensive MM.

INTERNAL ORGANIZATION OF MEMORY CHIPS:

Memory cells are usually organized in the form of array, in which each cell is capable of storing one bit of information. Each row of cells constitutes a memory word and all cells of a row are connected to a common line called as word line. The cells in each column are connected to Sense / Write circuit by two bit lines. Figure 3..1 shows the possible arrangements of memory cells.

The Sense / Write circuits are connected to data input or output lines of the chip. During a write operation, the sense / write circuit receives input information and stores it in the cells of the selected word. The data input and data output of each sense / write circuits are connected to a single bidirectional data line that can be connected to a data bus of the cpu.

R / W → specifies the required operation.

CS → Chip Select input selects a given chip in the multi-chip memory system

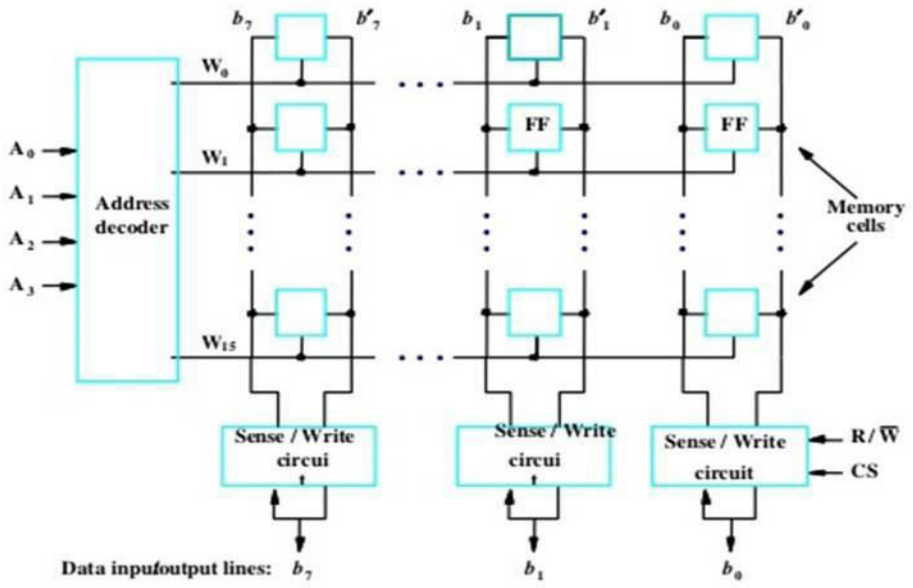


Figure Organization of bit cells in a memory chip.

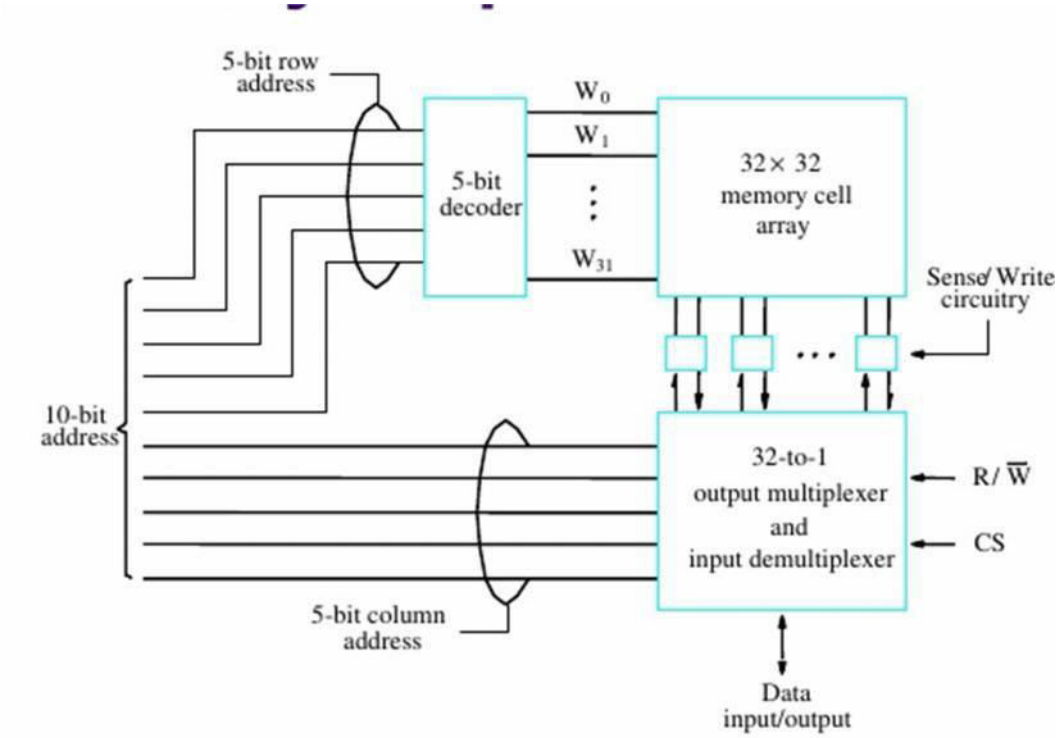


Figure 5.3. Organization of a 1K × 1 memory chip.

3.3 SEMI CONDUCTOR RAM MEMORIES:

Semi-Conductor memories are available is a wide range of speeds. Their cycle time ranges from 100ns to 10ns. When first introduced in the late 1960s, they were much more expensive. But now they are very cheap, and used almost exclusively in implementing main memories.

3.3.1STATIC MEMORIES:

Memories that consist of circuits capable of retaining their state as long as power is applied are known as static memory.

Static random-access memory (**SRAM**) is a type of semiconductor memory that uses bistable latching circuitry to store each bit. The term *static* differentiates it from *dynamic* RAM (DRAM) which must be periodically refreshed. SRAM exhibits data remanence, but is still *volatile* in the conventional sense that data is eventually lost when the memory is not powered. Figure 3.2 shows the implementation of static RAM.

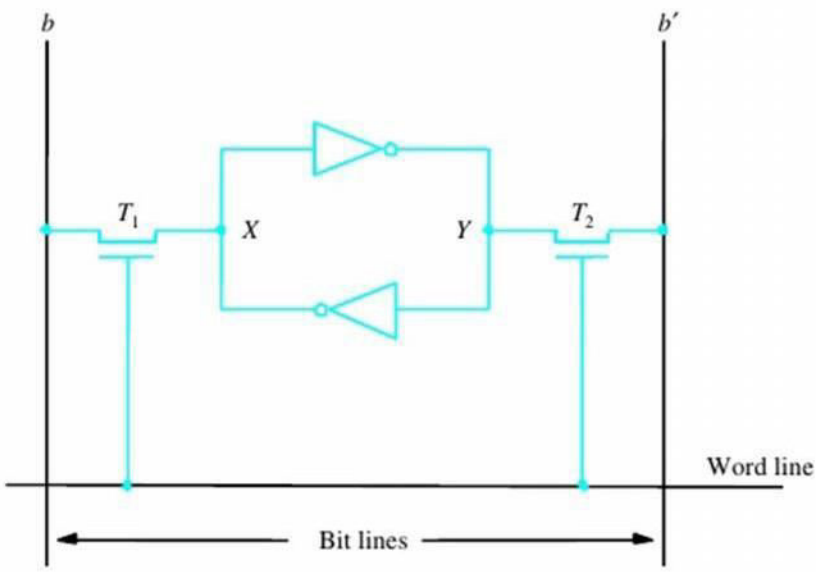


fig: 3.2Static RAM cell

Two inverters are cross connected to form a latch. The latch is connected to two bit lines by transistors T1 and T2. These transistors act as switches that can be opened / closed under the control of the word line. When the word line is at ground level, the transistors are turned off and the latch retains its state.

Read Operation:

In order to read the state of the SRAM cell, the word line is activated to close switches T1 and T2. If the cell is in state 1, the signal on bit line b is high and the signal on the bit line b is low. Thus b and $b^{'}$ are complement of each other. Sense / write circuit at the end of the bit line monitors the state of b and $b^{'}$ and set the output according.

Write Operation:

The state of the cell is set by placing the appropriate value on bit line b and its complement on b' and then activating the word line. This forces the cell into the corresponding state. The required signal on the bit lines are generated by Sense / Write circuit.

CMOS RAM CELL

Transistor pairs (T3, T5) and (T4, T6) form the inverters in the latch. In state 1, the voltage at point X is high by having T5, T6 on and T4, T5 are OFF. Thus T1 and T2 returned ON (Closed), bit line b and b' will have high and low signals respectively. The CMOS requires 5V (in older version) or 3.3.V (in new version) of power supply voltage.

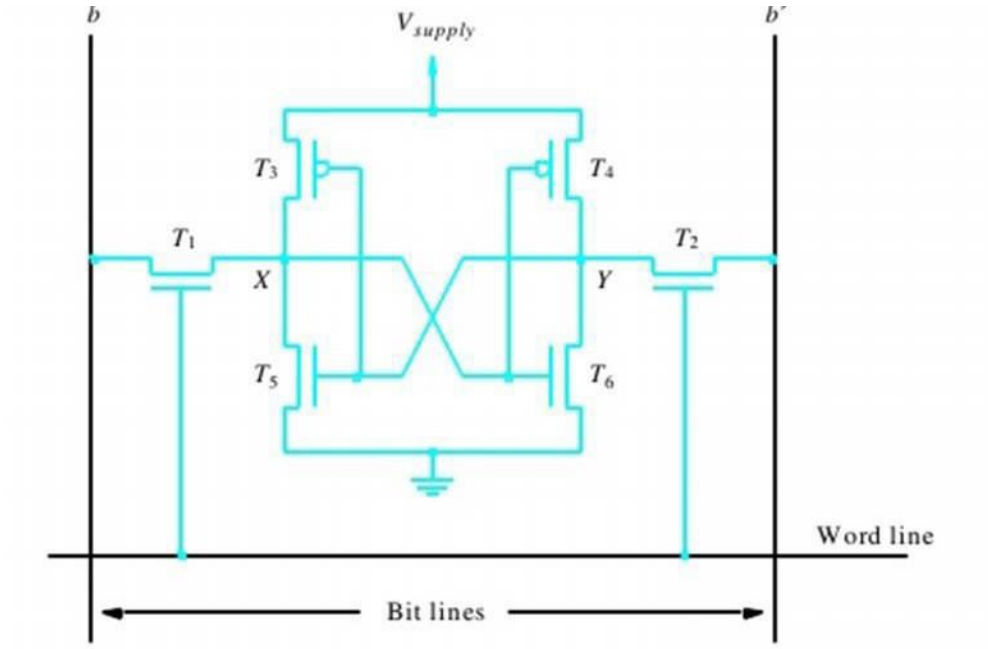


Figure3.3 : CMOS cell (Complementary Metal oxide Semi-Conductor):

Merit:

- It has low power consumption because the current flows in the cell only when the cell is being accessed.
- Static RAMs can be accessed quickly. It access time is few nanoseconds.

Demerit:

- SRAMs are said to be volatile memories because their contents are lost when the power is interrupted.

Asynchronous DRAMS:

Less expensive RAM"s can be implemented if simpler cells are used. Such cells cannot retain their state indefinitely. Hence they are called Dynamic RAM‘s (DRAM). The information stored in a dynamic memory cell in the form of a charge on a capacitor

and this charge can be maintained only for a few milliseconds. The contents must be periodically refreshed by restoring this capacitor charge to its full value.

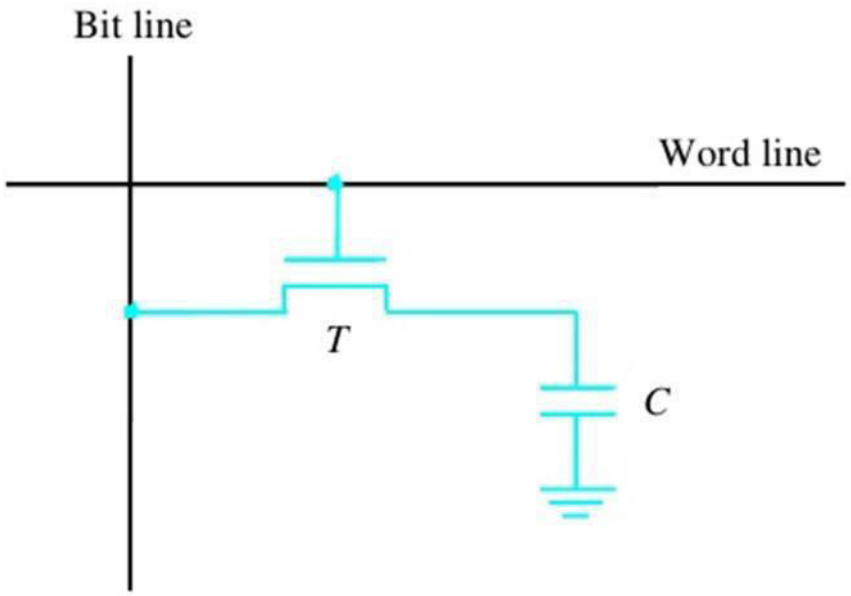


Figure 3.4: A single transistor dynamic Memory cell

In order to store information in the cell, the transistor T is turned on and the appropriate voltage is applied to the bit line, which charges the capacitor. After the transistor is turned off, the capacitor begins to discharge which is caused by the capacitor's own leakage resistance. Hence the information stored in the cell can be retrieved correctly before the threshold value of the capacitor drops down, as shown in Figure 3.4.

If charge on capacitor > threshold value →Bit line will have logic value **1**.

If charge on capacitor < threshold value → Bit line will set to logic value **0**.

During a read operation, the transistor is turned on and a sense amplifier connected to the bit line detects whether the charge on the capacitor is above the threshold value.

A 16-megabit DRAM chip configured as 2M x 8, is show n in Figure 3.5.

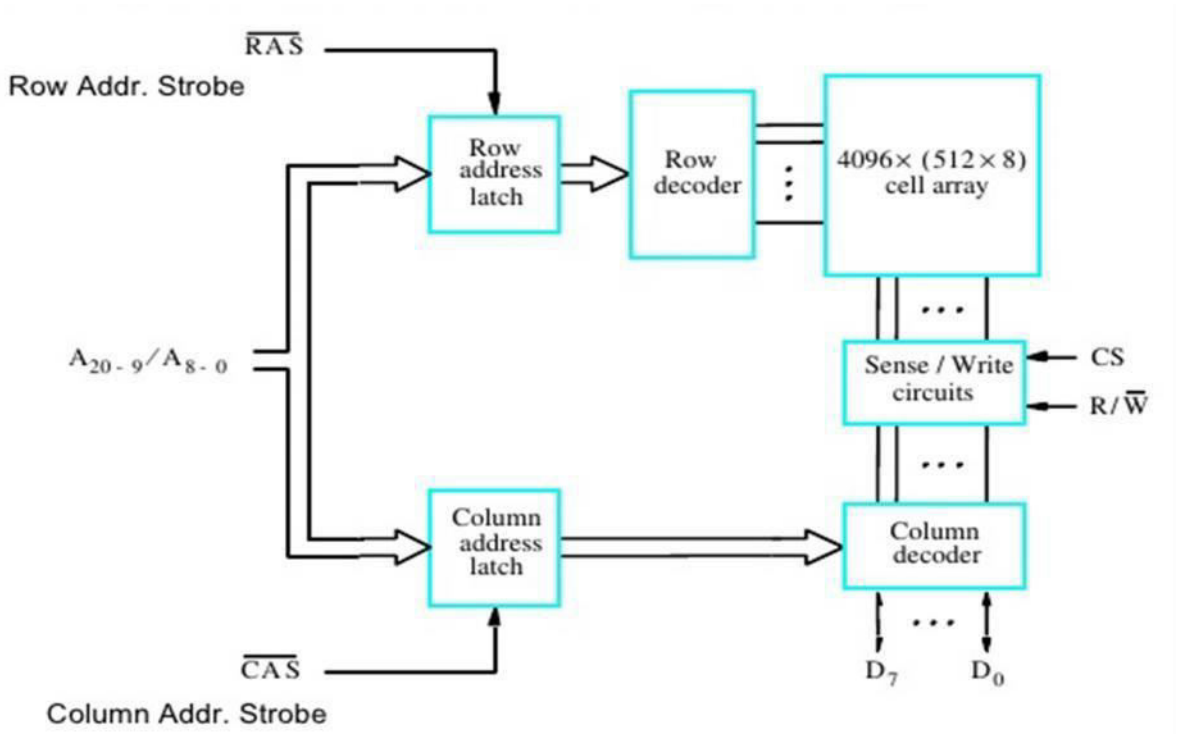


Figure 3.5: Internal organization of a 2M X 8 dynamic Memory chip.

DESCRIPTION:

The 4 bit cells in each row are divided into 512 groups of 8. 21 bit address is needed to access a byte in the memory (12 bit to select a row, and 9 bits specify the group of 8 bits in the selected row).

A (0-8) → Row address of a byte.

A (9-20) → Column address of a byte.

During Read/ Write operation, the row address is applied first. It is loaded into the row address latch in response to a signal pulse on Row Address Strobe (RAS) input of the chip. When a Read operation is initiated, all cells on the selected row are read and refreshed. Shortly after the row address is loaded, the column address is applied to the address pins and loaded into Column Address Strobe (CAS). The information in this latch is decoded and the appropriate group of 8 Sense/Write circuits is selected. R/W =1(read

The output values of the selected circuits are transferred to the data lines D0 - D7. R/W=0 (write operation). The information on D0 - D7 is transferred to the selected circuits.

RAS and CAS are active low so that they cause the latching of address when they change from high to low. This is because they are indicated by RAS and CAS. To ensure that the contents of a DRAM‘s are maintained, each row of cells must be accessed periodically. Refresh operation usually perform this function automatically. A specialized memory controller circuit provides the necessary control signals RAS and CAS that govern the timing. The processor must take into account the delay in the response of the memory. Such memories are referred to as Asynchronous DRAM‘s.

Fast Page Mode:

Transferring the bytes in sequential order is achieved by applying the consecutive sequence of column address under the control of successive CAS signals. This scheme allows transferring a block of data at a faster rate. The block of transfer capability is called as Fast Page Mode.

Synchronous DRAM:

Here the operations are directly synchronized with clock signal. The address and data connections are buffered by means of registers. The output of each sense amplifier is connected to a latch. A Read operation causes the contents of all cells in the selected row to be loaded in these latches. The Figure 3.6 shows the structure of SDRAM.

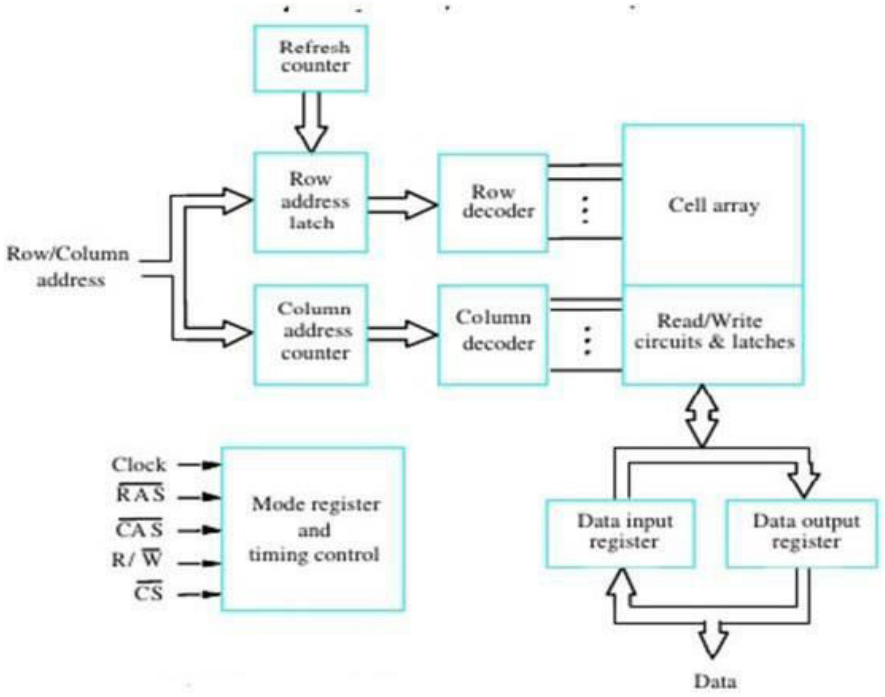


Figure 3.6: Synchronous DRAM

Data held in the latches that correspond to the selected columns are transferred into the data output register, thus becoming available on the data output pins.

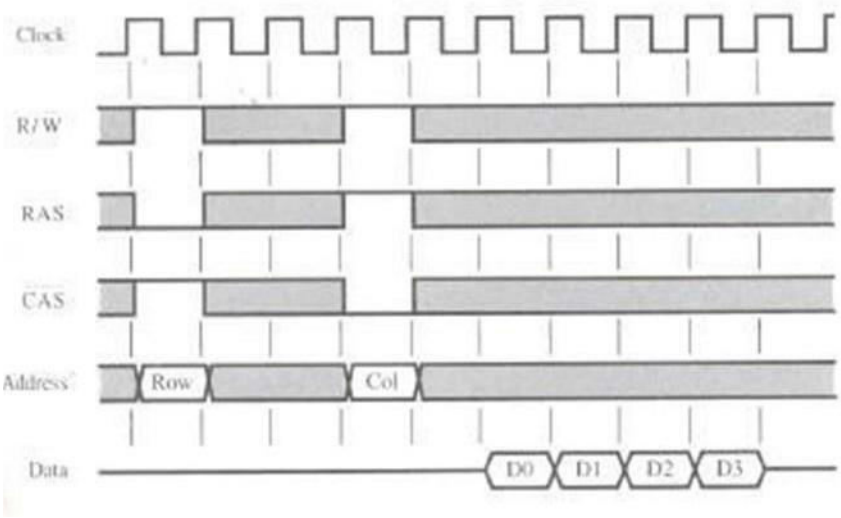


Figure 3.7: Timing Diagram Burst Read of Length 4 in an SDRAM

First, the row address is latched under control of RAS signal. The memory typically takes 2 or 3 clock cycles to activate the selected row. Then the column address is latched under the control of CAS signal. After a delay of one clock cycle, the first set of data bits is placed on the data lines. The SDRAM automatically increments the column address to access the next 3 sets of bits in the selected row, which are placed on the data lines in the next 3 clock cycles. A timing diagram for a typical burst of read of length 4 is shown in Figure 3.7.

Latency and Bandwidth:

A good indication of performance is given by two parameters. They are,

- Latency
- Bandwidth

Latency refers to the amount of time it takes to transfer a word of data to or from the memory. For a transfer of single word, the latency provides the complete indication of memory performance. For a block transfer, the latency denotes the time it takes to transfer the first word of data.

Bandwidth is defined as the number of bits or bytes that can be transferred in one second. Bandwidth mainly depends upon the speed of access to the stored data and on the number of bits that can be accessed in parallel.

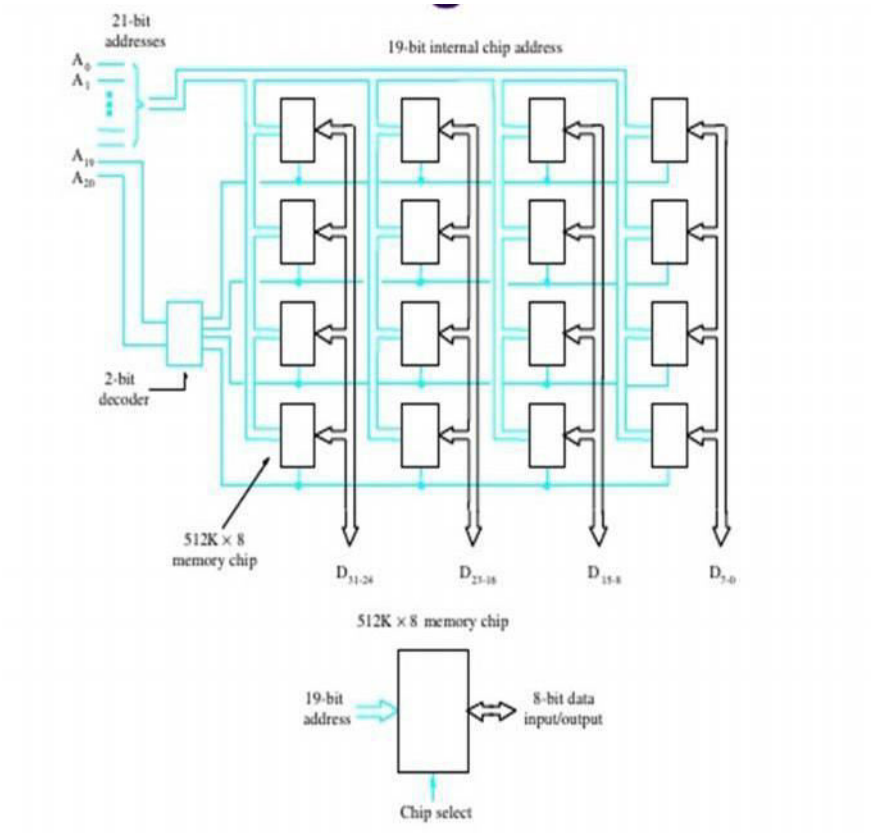
Double Data Rate SDRAM (DDR-SDRAM):

The standard SDRAM performs all actions on the rising edge of the clock signal. The double data rate SDRAM transfer data on both the edges (loading edge, trailing edge). The Bandwidth of DDR-SDRAM is doubled for long burst transfer. To make it possible to access the data at high rate, the cell array is organized into two banks. Each bank can be accessed separately. Consecutive words of a given block are stored in different banks. Such interleaving of words allows simultaneous access to two words that are transferred on successive edge of the clock.

3.3.2 Larger Memories: Dynamic Memory System:

The physical implementation is done in the form of Memory Modules. If a large memory is built by placing DRAM chips directly on the main system printed circuit board that contains the processor, often referred to as Motherboard; it will occupy large amount of space on the board. These packaging consideration have led to the development of larger memory units known as SIMM's and DIMM's

SIMM-Single Inline memory Module
DIMM-Dual Inline memory Module



MEMORY SYSTEM CONSIDERATION:

To reduce the number of pins, the dynamic memory chips use multiplexed address inputs. The address is divided into two parts. They are,

High Order Address Bit (Select a row in cell array and it is provided first and latched into memory chips under the control of RAS signal).

Low Order Address Bit (Selects a column and they are provided on same address pins and latched using CAS signals).

The Multiplexing of address bit is usually done by Memory Controller Circuit, as shown in Figure 3.8.

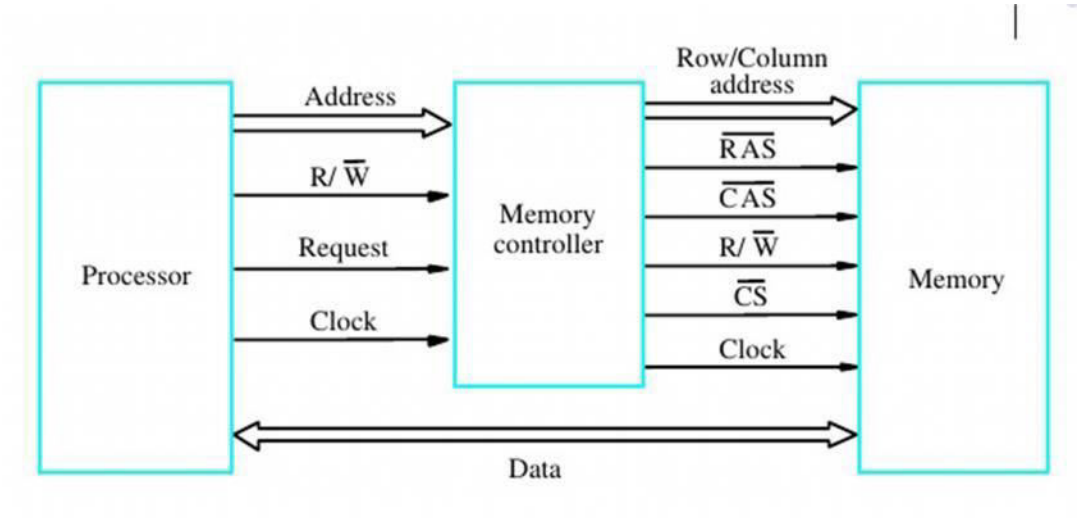


Figure 3.8: Use of Memory Controller

The Controller accepts a complete address and R/W signal from the processor, under the control of a request signal which indicates that a memory access operation is needed. The Controller then forwards the row and column portions of the address to the memory and generates RAS and CAS signals. It also sends R/W and CS signals to the memory. The CS signal is usually active low, hence it is shown as CS.

Refresh Overhead:

All dynamic memories have to be refreshed. In DRAM, the period for refreshing all rows is 16ms whereas 64ms in SDRAM.

Rambus Memory:

The usage of wide bus is expensive. Rambus developed the implementation of

narrow bus. Rambus technology is a fast signaling method used to transfer information between chips. Instead of using signals that have voltage levels of either 0 or Vsupply to represent the logical values, the signals consist of much smaller voltage swings around a reference voltage Vref. The reference Voltage is about 2V and the two logical values are represented by 0.3V swings above and below Vref.

This type of signaling is generally is known as Differential Signaling. Rambus provides a complete specification for the design of communication links (Special Interface circuits) called as Rambus Channel. Rambus memory has a clock frequency of 400MHZ. The data are transmitted on both the edges of the clock so that the effective data transfer rate is 800MHZ.

The circuitry needed to interface to the Rambus channel is included on the chip. Such chips are known as Rambus DRAMs (RDRAM). Rambus channel has,

- 9 Data lines (1-8 ⌜Transfer the data, 9th line → Parity checking).
- Control line
- Power line

A two channel Rambus has 18 data lines which have no separate address lines. It is also called as Direct RDRAM‘s. Communication between processor or some other device that can serves as a master and RDRAM modules are served as slaves, is carried out by means of packets transmitted on the data lines.
There are 3 types of packets. They are,

- Request
- Acknowledge
- Data

3.4 READ ONLY MEMORY (ROM):

Both SRAM and DRAM chips are volatile, which means that they lose the stored information if power is turned off. Many applications require Non-volatile memory (which retains the stored information if power is turned off).
E.g.: Operating System software has to be loaded from disk to memory which requires the program that boots the Operating System. i.e., it requires non-volatile memory. Non-volatile memory is used in embedded system. Since the normal operation involves only reading of stored data, a memory of this type is called ROM.

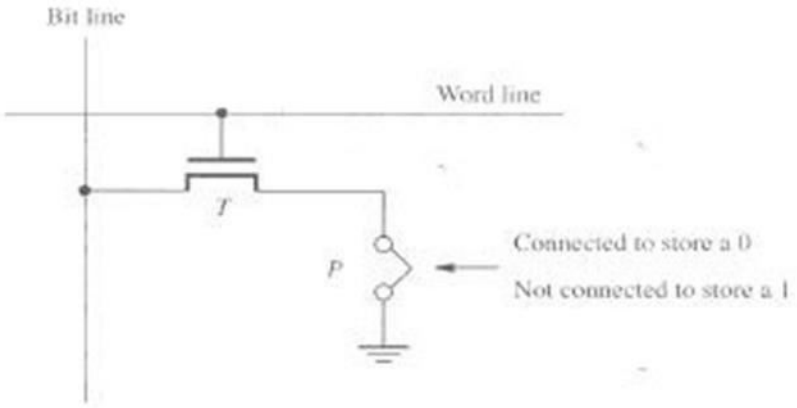


Figure 3.9: ROM cell

At Logic value _0‘ → Transistor (T) is connected to the ground point (P). Transistor switch is closed and voltage on bit line nearly drops to zero.

At Logic value _1‘ → Transistor switch is open. The bit line remains at high voltage. To read the state of the cell, the word line is activated. A Sense circuit at the end of the bit line generates the proper output value.

Different types of non-volatile memory are:

- PROM
- EPROM
- EEPROM
- Flash Memory

PROM: Programmable ROM:

PROM allows the data to be loaded by the user. Programmability is achieved by inserting a fuse at point P in a ROM cell. Before it is programmed, the memory contains all 0‘s. The user can insert 1‘s at the required location by burning out the fuse at these locations using high current pulse. This process is irreversible.

Merit :

- It provides flexibility.
- It is faster.
- It is less expensive because they can be programmed directly by the user.

EPROM - Erasable reprogrammable ROM:

EPROM allows the stored data to be erased and new data to be loaded. In an EPROM cell, a connection to ground is always made at _P‘ and a special transistor is used, which has the ability to function either as a normal transistor or as a disabled transistor that is always turned off. This transistor can be programmed to behave as a

permanently open switch, by injecting charge into it that becomes trapped inside.

Erasure requires dissipating the charges trapped in the transistor of memory cells. This can be done by exposing the chip to ultraviolet light, so that EPROM chips are mounted in packages that have transparent windows.

Merits:

- It provides flexibility during the development phase of digital system.
- It is capable of retaining the stored information for a long time.

Demerits:

- The chip must be physically removed from the circuit for reprogramming and its entire contents are erased by UV light.

EEPROM:- Electrically Erasable ROM:

EEPROM (also written **E^{2}PROM** and pronounced "e-e-prom," "double-e prom," "e-squared," or simply "e-prom") stands for **E**lectrically **E**rasable **P**rogrammable **R**ead- **O**nly **M**emory and is a type of non-volatile memory used in computers and other electronic devices to store small amounts of data that must be saved when power is removed, e.g., calibration tables or device configuration.

When larger amounts of static data are to be stored (such as in USB flash drives) a specific type of EEPROM such as flash memory is more economical than traditional EEPROM devices. EEPROMs are realized as arrays of floating-gate transistors. EEPROM is user modifiable read only memory (ROM) that can be erased and reprogrammed (written to) repeatedly through the application of higher than normal electrical voltage generated externally or internally in the case of modern EEPROMs.

EPROM usually must be removed from the device for erasing and programming, whereas EEPROMs can be programmed and erased in circuit. Originally, EEPROMs were limited to single byte operations which made them slower, but modern EEPROMs allow multi-byte page operations. It also has a limited life - that is, the number of times it could be reprogrammed was limited to tens or hundreds of thousands of times.

That limitation has been extended to a million write operations in modern EEPROMs. In an EEPROM that is frequently reprogrammed while the computer is in use, the life of the EEPROM can be an important design consideration. It is for this reason that EEPROMs were used for configuration information, rather than random access memory.

Merits:

It can be both programmed and erased electrically.

It allows the erasing of all cell contents selectively.

Demerits:

It requires different voltage for erasing, writing and reading the stored data.

FLASH MEMORY:

In EEPROM, it is possible to read and write the contents of a single cell. In Flash device, it is possible to read the contents of a single cell but it is only possible to write the entire contents of a block. Prior to writing, the previous contents of the block are erased.

E.g.: In MP3 player, the flash memory stores the data that represents sound. Single flash chips cannot provide sufficient storage capacity for embedded system application. There are 2 methods for implementing larger memory modules consisting of number of chips. They are,

- Flash Cards
- Flash Drives.

Merits:

- Flash drives have greater density which leads to higher capacity and low cost per bit.
- It requires single power supply voltage and consumes less power in their operation.

Flash Cards:

One way of constructing larger module is to mount flash chips on a small card. Such flash card have standard interface. The card is simply plugged into a conveniently accessible slot. Its memory size is of 8, 32,64MB. E.g.: A minute of music can be stored in 1MB of memory. Hence 64MB flash cards can store an hour of music.

Flash Drives:

Larger flash memory module can be developed by replacing the hard disk drive. The flash drives are designed to fully emulate the hard disk. The flash drives are solid state electronic devices that have no movable parts.

Merits:

- They have shorter seek and access time which results in faster response.

They have low power consumption which makes them attractive for battery driven application.

- They are insensitive to vibration.

Demerit:

- The capacity of flash drive (<1GB) is less than hard disk (>1GB).
- It leads to higher cost per bit.
- Flash memory will deteriorate after it has been written a number of times (typically at least 1 million times.)

SPEED, SIZE COST:

Characteristics	SRAM	DRAM	Magnetis Disk
Speed	Very Fast	Slower	Much slower than DRAM
Size	Large	Small	Small
Cost	Expensive	Less Expensive	Low price

Magnetic Disk:

A huge amount of cost effective storage can be provided by magnetic disk. The main memory can be built with DRAM which leaves SRA‘s to be used in smaller units where speed is of essence.

Memory	Speed	Size	Cost
Registers	Very high	Lower	Very Lower
Primary cache	High	Lower	Low
Secondary cache	Low	Low	Low
Main memory	Lower than Seconadry cache	High	High
Secondary Memory	Very low	Very High	Very High

3.5 CACHE MEMORY

Cache memory is an integral part of every system now. Cache memory is random access memory (RAM) that a computer microprocessor can access more quickly than it can access regular RAM. As the microprocessor processes data, it looks first in the cache memory and if it finds the data there (from a previous reading of data), it does not have to do the more time consuming reading of data from larger memory.

The effectiveness of cache mechanism is based on the property of Locality of reference.

Locality of Reference:

During some time period and remainder of the program is accessed relatively infrequently. It manifests itself in 2 ways. They are Temporal (The recently executed instruction are likely to be executed again very soon), Spatial (The instructions in close proximity to recently executed instruction are likely to be executed soon). If the active

segment of the program is placed in cache memory, then the total execution time can be reduced significantly.

If the active segment of a program can be placed in a fast cache memory, then the total execution time can be reduced significantly. The operation of a cache memory is very simple. The memory control circuitry is designed to take advantage of the property of locality of reference. The term Block refers to the set of contiguous address locations of some size. The cache line is used to refer to the cache block.

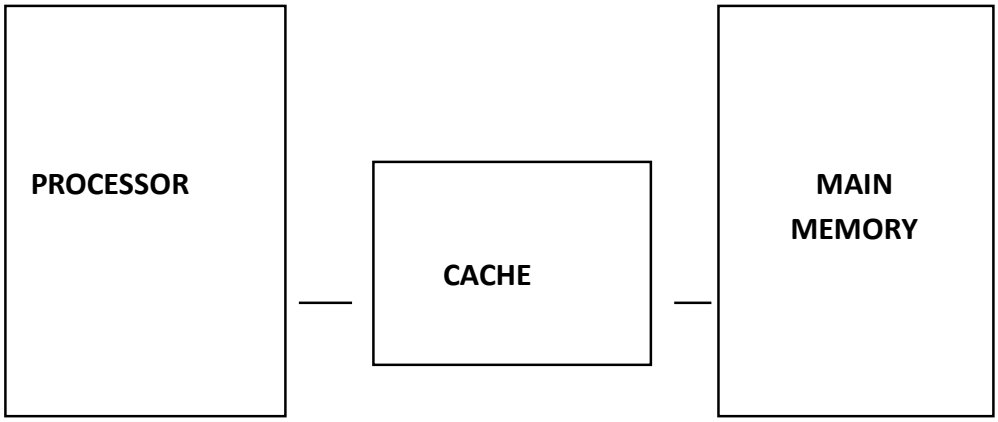


Figure : Use of Cache Memory

The Figure shows arrangement of Cache between processor and main memory. The Cache memory stores a reasonable number of blocks at a given time but this number is small compared to the total number of blocks available in Main Memory. The correspondence between main memory block and the block in cache memory is specified by a mapping function. The Cache control hardware decides that which block should be removed to create space for the new block that contains the referenced word. The collection of rule for making this decision is called the replacement algorithm. The cache control circuit determines whether the requested word currently exists in the cache. If it exists, then Read/Write operation will take place on appropriate cache location. In this case Read/Write hit will occur. In a Read operation, the memory will not be involved.

The write operation proceeds in 2 ways. They are:

- Write-through protocol
- Write-back protocol

Write-through protocol:

Here the cache location and the main memory locations are updated simultaneously.

Write-back protocol:

This technique is to update only the cache location and to mark it as with

associated flag bit called dirty/modified bit. The word in the main memory will be updated later, when the block containing this marked word is to be removed from the cache to make room for a new block. If the requested word currently does not exist in the cache during read operation, then read miss will occur. To overcome the read miss Load –through / early restart protocol is used.

Read Miss:

The block of words that contains the requested word is copied from the main memory into cache.

Load –through:

After the entire block is loaded into cache, the particular word requested is forwarded to the processor. If the requested word does exist in the cache during write operation, then Write Miss will occur. If Write through protocol is used, the information is written directly into main memory. If Write back protocol is used then blocks containing the addressed word is first brought into the cache and then the desired word in the cache is overwritten with the new information.

3.5.1CACHE MEMORY DESIGN PARAMETERS

There are two cache design parameters that dramatically influence the cache performance: the block size and the cache associability. There are also many other implementation techniques both hardware and software that improve the cache performance but they are not discussed here.

The simplest way to reduce the miss rate is to increase the block size. However increasing the block size also increases the miss penalty (which is the time to load a block from main memory into cache) so there is a trade–off between the block size and miss penalty. We can increase the block size up to a level at which the miss rate is decreasing but we also have to be sure that this benefit will not be exceeded by the increased miss penalty.

The second cache design parameter that reduces cache misses is the associability. There is an empirical result called the 2:1 rule of thumb which states that a direct mapped cache of size N has about the same miss rate as a 2 way set associative cache of size N/2. Unfortunately an increased associability will have a bigger hit time. More time will be taken to retrieve a block inside of an associative cache than in a direct mapped cache. To retrieve a block in an associative cache, the block must be searched inside of an entire set since there is more than one place where the block can be stored.

Based on the cause that determines a cache miss we can classify the cache misses as compulsory, capacity and conflict misses. This classification is called the 3C model. Compulsory misses are issued when a first access is done to a block that is not in the memory, so the block must be brought into cache. Increasing block size can reduce compulsory misses due to perfecting the other elements in the block. If the cache cannot

contain all the blocks needed during the execution of a program, capacity misses will occur due to blocks being discarded and later retrieved. If the block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Increasing the associability in general reduce the number of conflict misses and implicitly the runtime of the programs. However this is not true all the time. Minimizing the cache misses does not necessarily minimize the runtime. For example, there can be fewer cache misses with more memory accesses.

3.5.2Mapping Function:

Direct Mapping:

It is the simplest technique in which block j of the main memory maps onto block ‗J' modulo 128 of the cache. Thus whenever one of the main memory blocks 0, 128, 256 is loaded in the cache, it is stored in block 0. Block 1, 129, 257 are stored in cache block 1 and so on. The contention may arise when the cache is full, when more than one memory block is mapped onto a given cache block position. The contention is resolved by allowing the new blocks to overwrite the currently resident block. Placement of block in the cache is determined from memory address.

The memory address is divided into 3 fields, namely,

- **Low Order 4 bit field (word)** → Select one of 16 words in a block.
- **7 bit cache block field** → When new block enters cache, 7 bit determines the cache position in which this block must be stored.
- **5 bit Tag field** → The high order 5 bits of the memory address of the block is stored in 5 tag bits associated with its location in the cache.

As execution proceeds, the high order 5 bits of the address is compared with tag bits associated with that cache location. If they match, then the desired word is in that block of the cache. If there is no match, then the block containing the required word must be first read from the main memory and loaded into the cache. The direct mapping is shown in Figure 15.2.

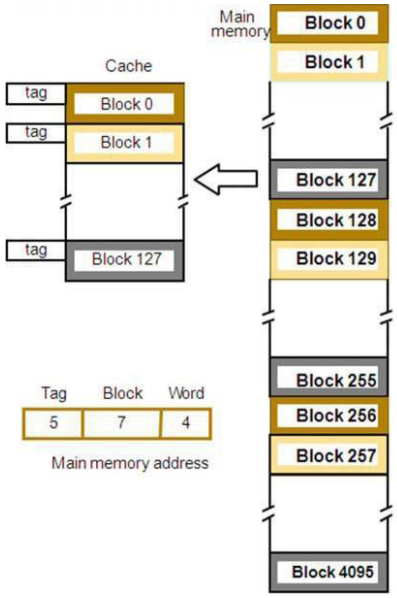


Figure 15.2: Direct Mapped Cache

Merit: It is easy to implement.

Demerit:

- It is not very flexible.

Associative Mapping:

Here, the main memory block can be placed into any cache block position.12 tag bits will identify a memory block when it is resolved in the cache. The tag bits of an address received from the processor are compared to the tag bits of each block of the cache to see if the desired block is present. This is called **associative mapping.** It gives complete freedom in choosing the cache location. A new block that has to be brought into the cache has to replace (eject) an existing block if the cache is full.

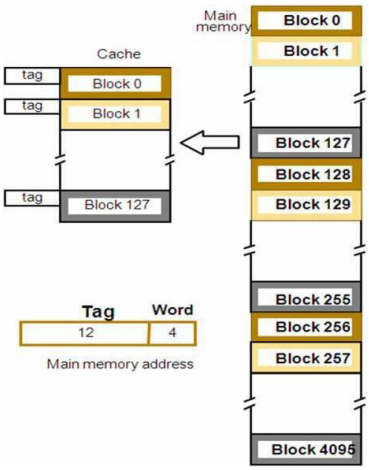


Figure 15.3: Associative Mapped Cache.

In this method, the memory has to determine whether a given block is in the cache. A search of this kind is called an associative Search. The associative-mapped cache is shown in Figure 15.3.

Merit:

- It is more flexible than direct mapping technique.

Demerit:

- Its cost is high.

Set-Associative Mapping:

It is the combination of direct and associative mapping. The blocks of the cache are grouped into sets and the mapping allows a block of the main memory to reside in any block of the specified set. In this case, the cache has two blocks per set, so the memory blocks 0, 64, 128……..4032 map into cache set to 0 and they can occupy either of the two block position within the set.

6 bit set field → Determines which set of cache contains the desired block.

6 bit tag field → the tag field of the address is compared to the tags of the two blocks of the set to clock if the desired block is present.

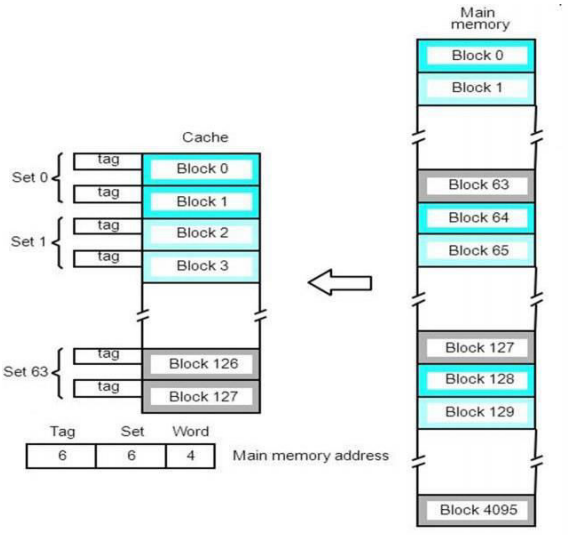


Figure 15.4: Set-Associative Mapping:

No of blocks per set	No of set field
2	6
3	5
8	4
128	no set field

The cache which contains 1 block per set is called direct Mapping. A cache that has ‗k' blocks per set is called as k-way set associative cache. Each block contains a control bit called a valid bit. The Valid bit indicates that whether the block contains valid data. The dirty bit indicates that whether the block has been modified during its cache residency. The set-associative mapping is shown in Figure 15.4.

Valid bit=0→When power is initially applied to system
Valid bit =1→When the block is loaded from main memory at first time.

If the main memory block is updated by a source and if the block in the source is already exists in the cache, then the valid bit will be cleared to ‗0'. If Processor and DMA use the same copies of data then it is called as the Cache Coherence Problem.

Merit:

- The Contention problem of direct mapping is solved by having few choices for block placement.
- The hardware cost is decreased by reducing the size of associative search.

3.6REPLACEMENT ALGORITHM:

In a direct-mapped cache, the position of each block is predetermined: hence, no replacement strategy exists. In associative and set-associative caches there exists some flexibility. When a new block is to be brought into the cache and all the positions that it may occupy are full, the cache controller must decide which of the old blocks to overwrite. This is an important issue, because the decision can be a strong determining factor in system performance. In general, the objective is to keep blocks in the cache that are likely to be referenced in the near future. However, it is not easy to determine which blocks are about to be referenced.

The property of locality of reference in programs gives a clue to a reasonable strategy. Because, programs usually stay in localized areas for reasonable periods of time, there is a high probability that the blocks that have been referenced recently will be referenced again soon. Therefore, when a block is to be overwritten, it is sensible to overwrite the one that has gone the longest time without being referenced. This block is called the least recently used (LW) block, and the technique is called the LRU replacement algorithm. To use the LRU algorithm, the cache controller must track references to all blocks as computation proceeds.

Suppose it is required to track the LRU block of a four block set in a set-associative cache. A 2-bit counter can be used for each block. When a hit occurs, the counter of the block that is referenced is set to 0. Counters with values originally lower than the referenced one are incremented by one, and all others remain unchanged. When a miss occurs and the set is not full,

the counter associated with the new block loaded from the main memory is set to 0, and the values of all other counters are increased by one. When a miss occurs and the set is full, the block with the counter value 3 is removed, the new block is put in its place, and its counter is set to 0.

The other three block counters are incremented by one (It can be easily verified that the counter values of occupied blocks are always distinct). The LRU algorithm has been used extensively. Although it performs well for many access patterns, it can lead to poor performance in some cases. For example, it produces disappointing results when accesses are made to sequential elements of an array that is slightly too large to fit into the cache. Performance of the LRU algorithm can be improved by introducing a small amount of randomness in deciding which block to replace.

Several other replacement algorithms are also used in practice. An intuitively reasonable rule would be to remove the "oldest" block from a full set when a new block must be brought in. However, because this algorithm does not take into account the recent pattern of access to blocks in the cache, it is generally not as effective as the LRU algorithm in choosing the best blocks to remove. The simplest algorithm is to randomly choose the block to be overwritten. Interestingly enough, this simple algorithm has been found to be quite effective in practice.

Example: Let us **c**onsider 4 blocks/set, in set associative cache, where 2 bit counter can be used for each block. When a ‗hit' occurs, then block counter = 0, the counter with values originally lower than the referenced one are incremented by 1 and all others remain unchanged. When a ‗miss' occurs and if the set is full, the blocks with the counter value 3 is removed, the new block is put in its place and its counter is set to ‗0' and other block counters are incremented by 1.

Merit: The performance of LRU algorithm is improved by randomness in deciding which

block is to be overwritten.

3.7 PERFORMANCE CONSIDERATION:

Two Key factors in the commercial success are the performance and cost where the best possible performance is at low cost. A common measure of success is called the Price Performance ratio.

Performance depends on how fast the machine instructions are brought to the processor and how fast they are executed. To achieve parallelism (i.e., both the slow and fast units are accessed in the same manner) interleaving is used.

3.7.1 Interleaving:

If the main memory is structured as a collection of physically separated modules, each with its own ABR (Address buffer register) and DBR(Data buffer register), memory access operations may proceed in more than one module at the same time. Thereby the aggregate rate of transmission of words to and from the main memory system can be increased.

Two methods of address layout are indicated in Figure 3.5. In the first case, memory address generated by the processor is decoded as shown in part (a) of the figure. The high-order k bits name one of n modules and the low-order m bits name a particular word in that module. When consecutive locations are accessed, only one module is involved. At the same time, devices with DMA ability may be accessing information in other modules.

In the second case, as shown in part (b) of the figure, which is called memory interleaving. The low-order k bits of the memory address select a module, and the high- order m bits name a location within the module. Thus, any component of the system that generates requests for access to consecutive memory locations can keep several modules busy at any one time which results in both faster access to a block of data and higher average utilization of the memory system as a whole.

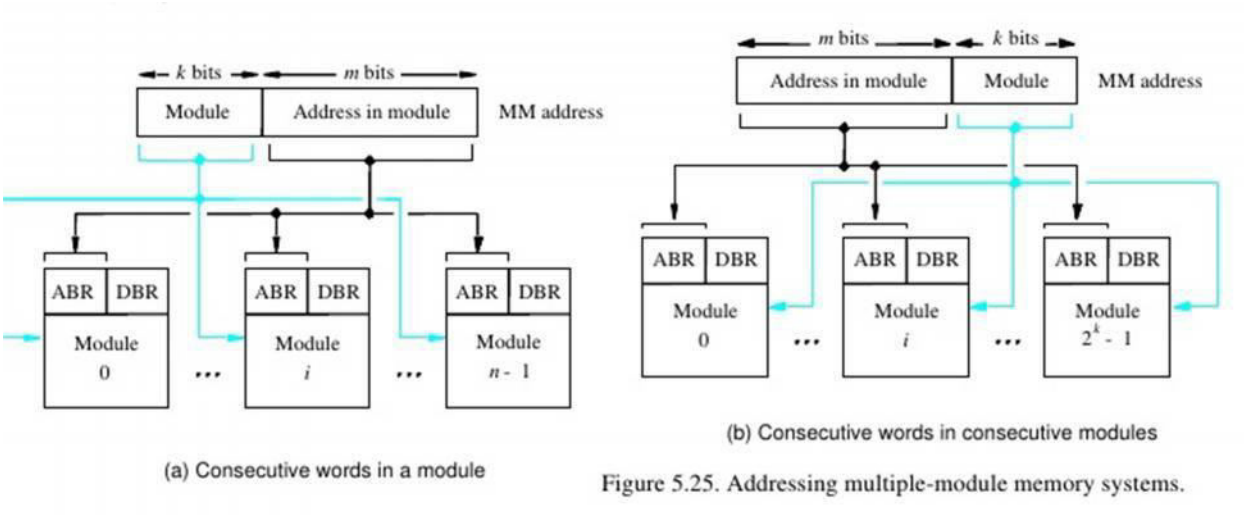


Figure : Addressing multiple-module memory system

3.7.2 HIT RATE AND MISS PENALTY

An excellent indicator of the effectiveness of a particular implementation of the memory hierarchy is the success rate in accessing information at various levels of the hierarchy. Recall that a successful access to data in a cache is called a hit. The number of hits stated as a fraction of all attempted accesses is called the hit rate, and the miss rate is the number of misses stated as a fraction of attempted accesses. Ideally, the entire memory hierarchy would appear to the CPU as a single memory unit that has the access time of a cache on the CPU chip and the size of a magnetic disk. How close we get to this ideal depends largely on the hit rate at different levels of the hierarchy. High hit rates, over 0.9, are essential for high performance computers. Performance is adversely affected by the actions that must be taken after a miss. The extra time needed to bring the desired information into the cache is called the Miss penalty. This penalty is ultimately reflected in the time that the CPU is stalled because the required instructions or data are not available for execution. In general, the miss penalty is the time needed to bring a block of data from a slower unit in the memory hierarchy to a faster unit. The miss penalty is reduced if efficient mechanisms for transferring data between the various units of the hierarchy are implemented. The previous section shows how an interleaved memory can reduce the miss penalty substantially. Consider now the impact of the cache on the overall performance of the computer. Let h be the hit rate, M the miss penalty, that is, the time to access information in the main memory, and C the time to access information in the cache. The average access time experienced by the CPU is $hC + (1 – h)\ M$.

3.7.3 CACHES ON PROCESSING CHIPS

When information is transferred between different chips, considerable delays are introduced in driver and receiver gates on the chips. Thus, from the speed point of view, the optimal place for a cache is on the CPU chip. Unfortunately, space on the CPU chip is needed for many other functions: this limits the size of the cache that can be accommodated. All high performance processor chips include some form of a cache. Some manufacturers have chosen to implement two separate caches, one for instructions and another for data, as in the 68040 and PowerPC 604 processors. Others have implemented a single cache for both instructions and data, as in the PowerPC 601 processor. A combined cache for instructions and data is likely to have a somewhat better hit rate, because it offers greater flexibility in mapping new information into the cache. However, if separate caches arc used, it is possible to access both caches at the same time, which leads to increased parallelism and, hence, better performance. The disadvantage of separate caches is that the increased parallelism comes at the expense of more complex circuitry. Since the Size of a cache on the CPU chip is limited by space constraints, a good strategy for designing a high performance system is to use such a cache as a primary cache. An external secondary cache, constructed with SRAM chips, is

When added to provide the desired capacity. If both primary and secondary caches are used, the primary cache should be designed to allow very fast access by the CPU, because its access time will have a large effect on the clock rate of the CPU. A cache cannot be accessed at the same speed as a register file, because the cache is much bigger and hence more complex. A practical way to speed up access in the cache is to access more than one word simultaneously and then let the CPU use them one at a time. The secondary cache can be considerably slower, but it should be much larger to ensure a high hit rate. Its speed is less critical, because it only affects the miss penalty of the primary cache. A workstation computer may include a primary cache with the capacity of tens of kilobytes and a secondary cache of several megabytes.

3.7.4 OTHER ENHANCEMENTS

In addition to the main design issues just discussed, several other possibilities exist for enhancing performance. We discuss three of them in this section.

Write Buffer

When the write-through protocol is used, each write operation results in writing a new value into the main memory. If the CPU must wait for the memory function to be completed, as we have assumed until now, then the CPU is slowed down by all write requests. Yet the CPU typically does not immediately depend on the result of a write operation, so it is not necessary for the CPU to wait for the write request to be completed. To improve performance, a write buffer can be included for temporary storage of write requests. The CPU places each write request into this buffer and continues execution of the next instruction. The write requests stored in the write buffer are sent to the main memory whenever the memory is not responding to read requests. Note that it is important that the read requests be serviced immediately, because the CPU usually cannot proceed without the data that is to be read from the memory. Hence, these requests are given priority over write requests. The

write buffer may hold a number of write requests. Thus, it is possible that a subsequent read request may refer to data that are still in the write buffer. To ensure correct operation, the addresses of data to be read

from the memory are compared with the addresses of the data in the write buffer. In case of a match, the data in the write buffer are used. This need for address comparison entails considerable cost. But the cost is justified by improved performance. A different situation occurs with the write-back protocol. In this case, the write operations are simply performed on the corresponding word in the cache. But consider what happens when a new block of data is to be brought into the cache as a result of a read miss, which replaces an existing block that has some dirty data. The dirty block has to be written into the main memory. If the required write-back is performed first, then the CPU will have to wait longer for the new block to be read into the cache. It is more prudent to read the new block first. This can be arranged by providing a fast write buffer for temporary storage of the dirty block that is ejected from the cache while the new block is being read. Afterward, the contents of the buffer are written into the main memory. Thus, the write buffer also works well for the write-back protocol.

Prefetching

In the previous discussion of the cache mechanism, we assumed that new data are brought into the cache when they are first needed. A read miss occurs, and the desired data are loaded from the main memory. The CPU has to pause until the new data

arrive, which is the effect of the miss penalty. To avoid stalling the CPU, it is possible to prefetch the data into the cache before they are needed. The simplest way to do this is through software. A special prefetch instruction may be provided in the instruction set of the processor. Executing this instruction causes the addressed data to be loaded into the cache, as in the case of a read miss. However, the processor does not wait for the referenced data. A prefetch instruction is inserted in a program to cause the data to be loaded in the cache by the time they are needed in the program. The hope is that prefetching will take place while the CPU is busy executing instructions that do not result in a read miss, thus allowing accesses to the main memory to be overlapped with computation in the CPU. Prefetch instructions can be inserted into a program either by the programmer or by the compiler. It is obviously preferable to have the compiler insert these instructions, which can be done with good success for many applications. Note that software prefetching entails a certain overhead; because inclusion of prefetch instructions increases the length of programs. Moreover, some prefetches may load into the cache data that will not be used by the instruction. This can happen if the prefetched data are ejected from the cache by a read miss involving other data. However, the overall effect of software prefetching on performance is positive, and many processors (including the PowerPC) have machine instructions to support this feature. Prefetching can also be done through hardware. This involves adding circuitry that attempts to discover a pattern in memory references, and then prefetches data according to this pattern.

Lockup-Free

Cache the software prefetehing scheme just discussed does not work well if it interferes significantly with the normal execution of instructions. This is the case if the action of prefetehing stops other accesses to the cache until the prefetch is completed. A cache of this type is said to be locked while it services a miss. We can solve this problem by modifying the basic cache structure to allow the CPU to access the cache while a miss is being serviced. In fact, it is desirable that more than one outstanding miss can be supported. A cache that can support multiple outstanding misses is called lockup- free. Since it can service only one miss at a time, it must include circuitry that keeps track of all outstanding misses. This may be done with special registers that hold the pertinent inFormation about these misses. Lockup-free caches were first used in the early l980s in the Cyber series of computers manufactured by Control Data Company.

We have used software prefetching as an obvious motivation for a cache that is not locked by a read miss. A much more important reason is that, in a processor that uses a pipelined organization, which overlaps the execution of several instructions, a read miss caused by one instruction could stall the execution of other instructions. A lockup-free cache reduces the likelihood of such stalling.

Unit-V

PIPELINE AND VECTOR PROCESSING

5.1. 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 computer system.

Instead of processing each instruction sequentially as in a conventional computer, a parallel processing system is able to perform concurrent data processing to achieve faster execution time.

For example, while an instruction is being executed in the ALU, the next instruction can be read from memory. The system may have two or more processors operating concurrently.

ADVANTAGES OF PARALLEL PROCESSING:

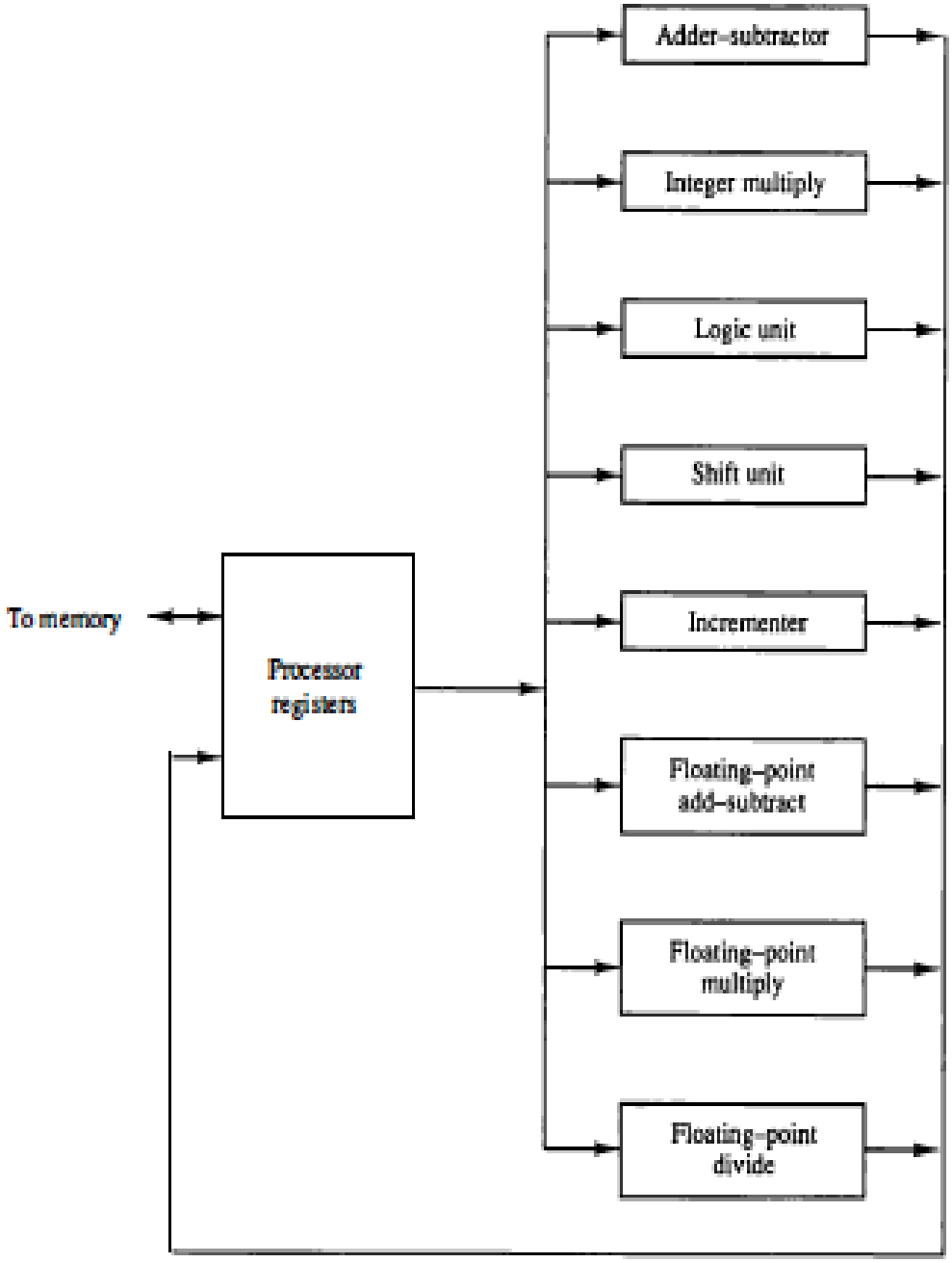
1. It speeds up the computer processing capability.
2. Increases its throughput, i.e., the amount of processing that can be accomplished during a given interval of time.
3. The amount of hardware increases with parallel processing and with it the cost of the system increases. However, technological developments have reduced hardware costs to the point where parallel processing techniques are economically feasible.

Parallel processing can be achieved by having a multiplicity of functional units that perform identical or different operations simultaneously. Parallel processing is established by distributing the data among the multiple functional units.

Fig shows one possible way of separating the execution time into 8 functional units operating in parallel. 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.



Flynn's classification divides computers into four major groups

- Single instruction stream, single data stream (SISD)
- Single instruction stream, multiple data stream (SIMD)
- Multiple instruction streams, single data stream (MISD)
- Multiple instruction stream, multiple data stream (MIMD)

SISD:

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 in this case may be achieved by means of multiple functional units or by pipeline processing.

SIMD:

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.

5.2. 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 segment in the pipeline. The final result is obtained after the data have passed through all segments.

5.2.1. Pipeline Organization

The simplest way of viewing the pipeline structure is to imagine that each segment consists of an input register followed by a combinational circuit. The register holds the data and the combinational circuit performs the sub operation in the particular segment. The output of the combinational circuit is applied to the input register of the next segment. A clock is applied to all registers after enough time has elapsed to perform all segment activity. In this way the information

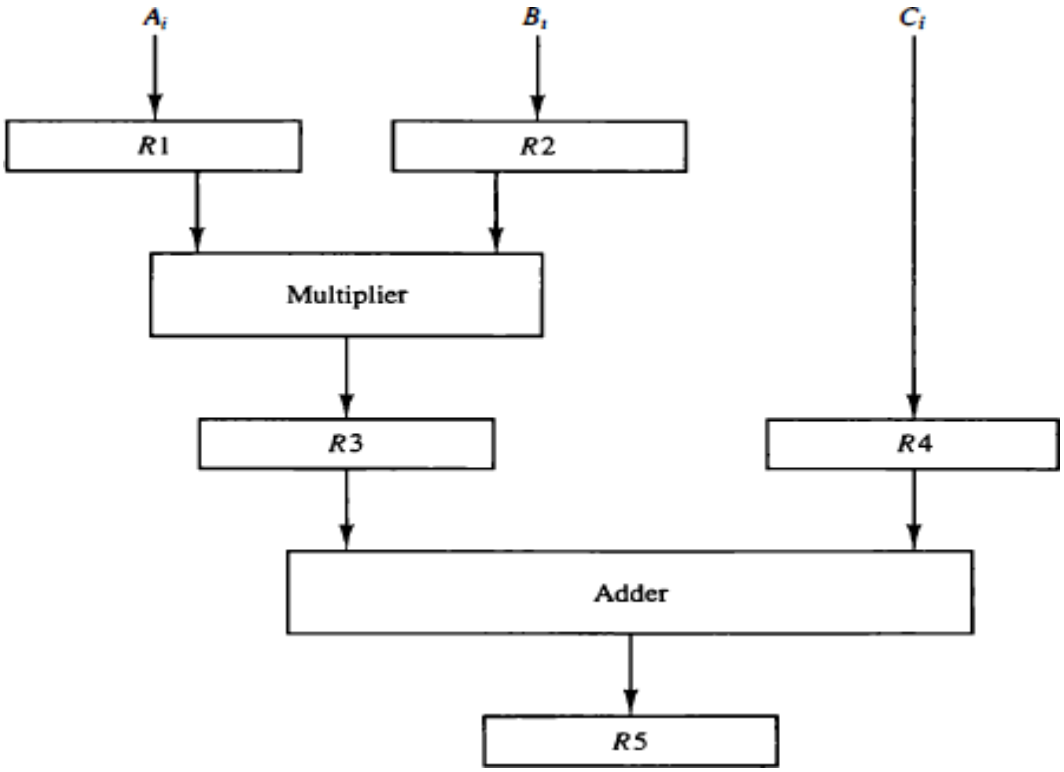
flows through the pipeline one step at a time.

Example demonstrating the pipeline organization

Suppose we want to perform the combined multiply and add operations with a stream of numbers.

$A_i * B_i + C_i$ for i=1, 2, 3 ….7

Each sub operation is to implemented in a segment within a pipeline. Each segment has one ortwo registers and a combinational circuit as shown in fig.



R1 through r5 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:

$R1 \leftarrow A_i$	$R2 \leftarrow B_i$	Input A_i and B_i
$R3 \leftarrow R1 * R2$	$R4 \leftarrow C_i$	multiply and input C_i
$R5 \leftarrow R3 + R4$		add C_i to product

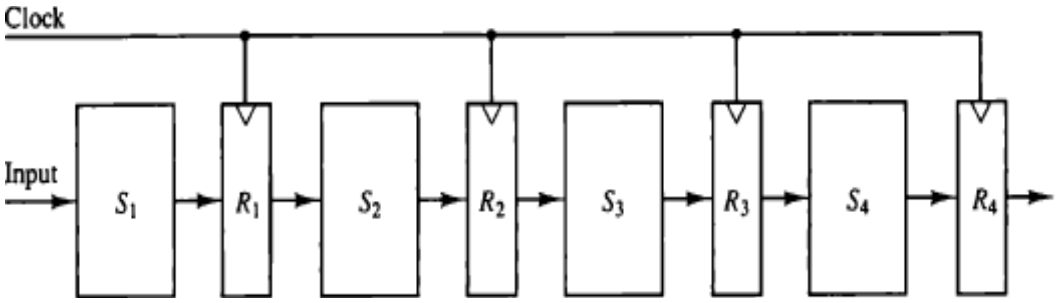
The five registers are loaded with new data every clock pulse.

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	—	—	—	—	$A_7 * B_7 + C_7$

The first clock pulse transfers A1 and B1 into R1 and R2. The second clock pulse transfers the product of R1 and R2 into R3 and C1 into R4. The same clock pulse transfers A2 and B2 into R1 and R2. The third clock pulse operates on all three segments simultaneously. It places A3 and B3 into R1 and R2, transfers the product of R1 and R2 into R3, transfers C2 into R4, and places the sum of R3 and R4 into R5. It takes three clock pulses to fill up the pipe and retrieve the first output from R5. From there on, each clock 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.

5.2.2. FOUR SEGMENT Pipeline

The general structure of four segment pipeline is shown in fig. the operands are passed through all four segments in affixed sequence. Each segment consists of a combinational circuit Si that performs a sub operation over the data stream flowing through the pipe. The segments are separated by registers Ri 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.



SPACE TIME DIAGRAM:

The behavior 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.

Fig The horizontal axis displays the time in clock cycles and the vertical axis gives the segment number. The diagram shows six tasks T1 through T6 executed in four segments. Initially, task T1 is handled by segment 1. After the first clock,

Segment:	1	2	3	4	5	6	7	8	9
1	T_1	T_2	T_3	T_4	T_5	T_6			
2		T_1	T_2	T_3	T_4	T_5	T_6		
3			T_1	T_2	T_3	T_4	T_5	T_6	
4				T_1	T_2	T_3	T_4	T_5	T_6

Clock cycles

segment 2 is busy with T1, while segment 1 is busy with task T2. Continuing in this manner, the first task T1 is completed after fourth clock cycle. From then on, the pipe completes a task every clock cycle.

Consider the case where a k-segment pipeline with a clock cycle time tp is used to execute n tasks. The first task T1 requires a time equal to ktp to complete its operation since there are k segments in a 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) tp. Therefore, to complete n tasks using a k segment pipeline requires

k+ (n-1) clock cycles.

Consider a non pipeline unit that performs the same operation and takes a time equal to tn to complete each task. The total time required for n tasks is n t_n. The speedup of a pipeline processing over an equivalent non pipeline processing is defined by the ratio

$$S=nt_n / (k+n-1)t_p$$

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 speed up ratio becomes

$$\mathbf{S=t_n/t_p}$$

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 $t_n=kt_p$. Including this assumption speed up ratio reduces to

$$S=kt_p/tp=k$$

5.3. ARITHMETIC PIPELINE

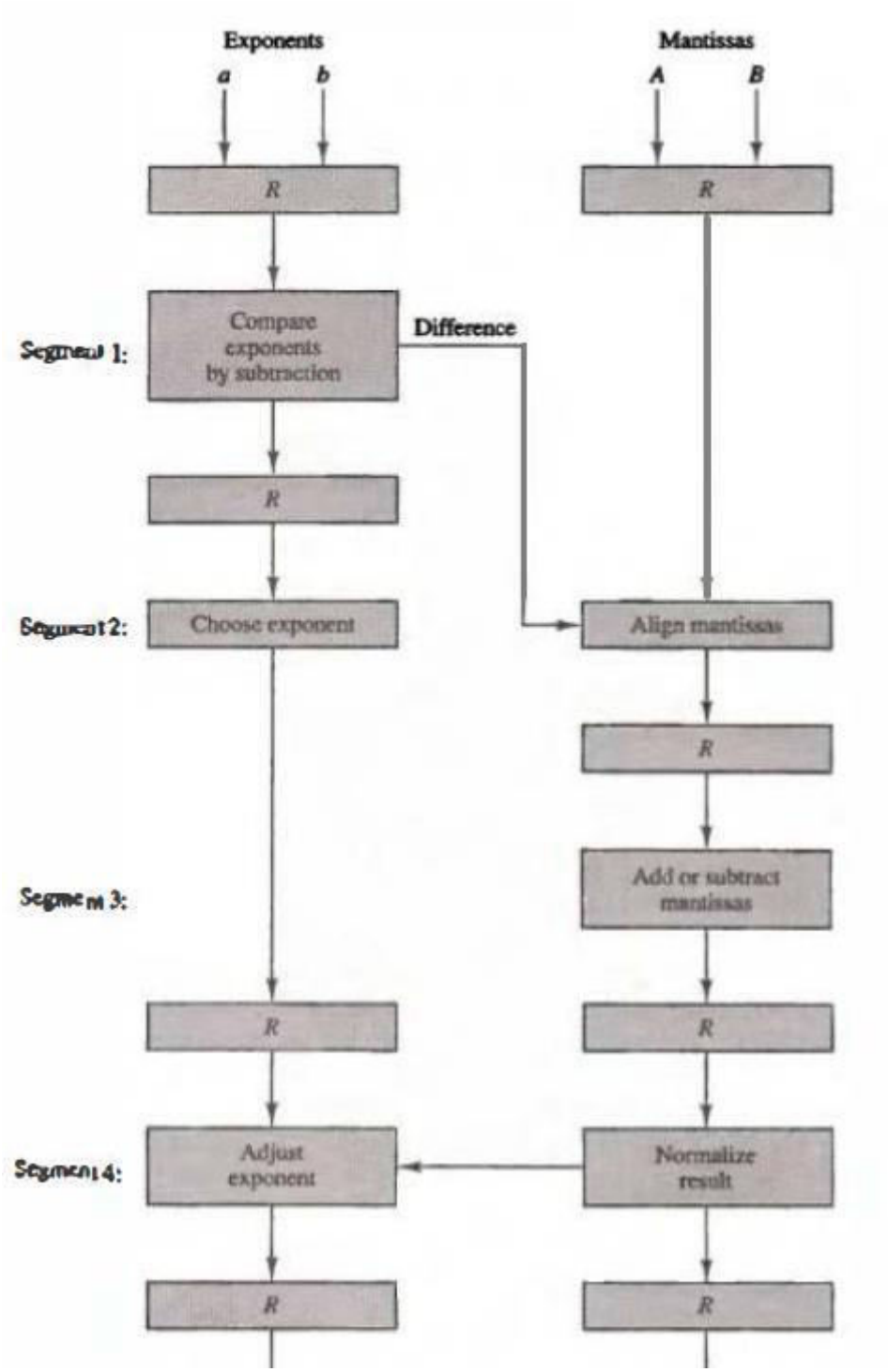
An arithmetic pipeline divides an arithmetic operation into sub operations for execution in the pipeline segments. Pipeline arithmetic units are usually found in very high speed computers. They are used to implement floating point operations, multiplication of fixed point numbers, and similar computations encountered in scientific problems.

5.3.1. Pipeline Unit For Floating Point Addition And Subtraction:

The inputs to the floating point adder pipeline are two normalized floating point binary numbers.

$X=A*2^a$

$Y=B*2^b$



A and B are two fractions that represent the mantissa and a and bare the exponents. The floating point addition and subtraction can be performed in four segments. The registers labeled are placed between the segments to store intermediate results. The sub operations that are performed in the four segments are:

1. Compare the exponents
2. Align the mantissa.
3. Add or subtract the mantissas.
4. Normalize the result.

The exponents are compared by subtracting them to determine their difference. The larger exponent is chosen as the exponent of the result. The exponent difference determines how many times the mantissa associated with the smaller exponent must be shifted to the right. This produces an alignment of the two mantissas.

The two mantissas are added or subtracted in segment3. The result is normalized in segment 4. When an overflow occurs, the mantissa of the sum or difference is shifted to right and the exponent incremented by one. If the underflow occurs, the number of leading zeroes in the mantissa determines the number of left shits in the mantissa and the number that must be subtracted from the exponent.

5.4. INSTRUCTION PIPELINE

An instruction pipeline operates on a stream of instructions by overlapping the fetch, decode, and execute phases of instruction cycle. An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments. This causes the instruction fetch and executes phases to overlap and perform simultaneous operations.

Consider a computer with an instruction fetch unit and an instruction execute unit designed to provide a two segment pipeline. The instruction fetch segment can be implemented by means of a first in first out (FIFO) buffer. Whenever the execution unit is not using memory, the control increments the program counter and uses it address value to read consecutive instructions from memory. The instructions are inserted into the FIFO buffer so that they can be executed on a first in first out basis. Thus an instruction stream can be placed in queue, waiting for decoding and processing by the execution segment.

In general the computer needs to process each instruction with the following sequence of steps.

1. Fetch the instruction.
2. Decode the instruction.

3. Calculate the effective address.

4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in the proper place.

5.4.1. Four Segment Instruction Pipeline

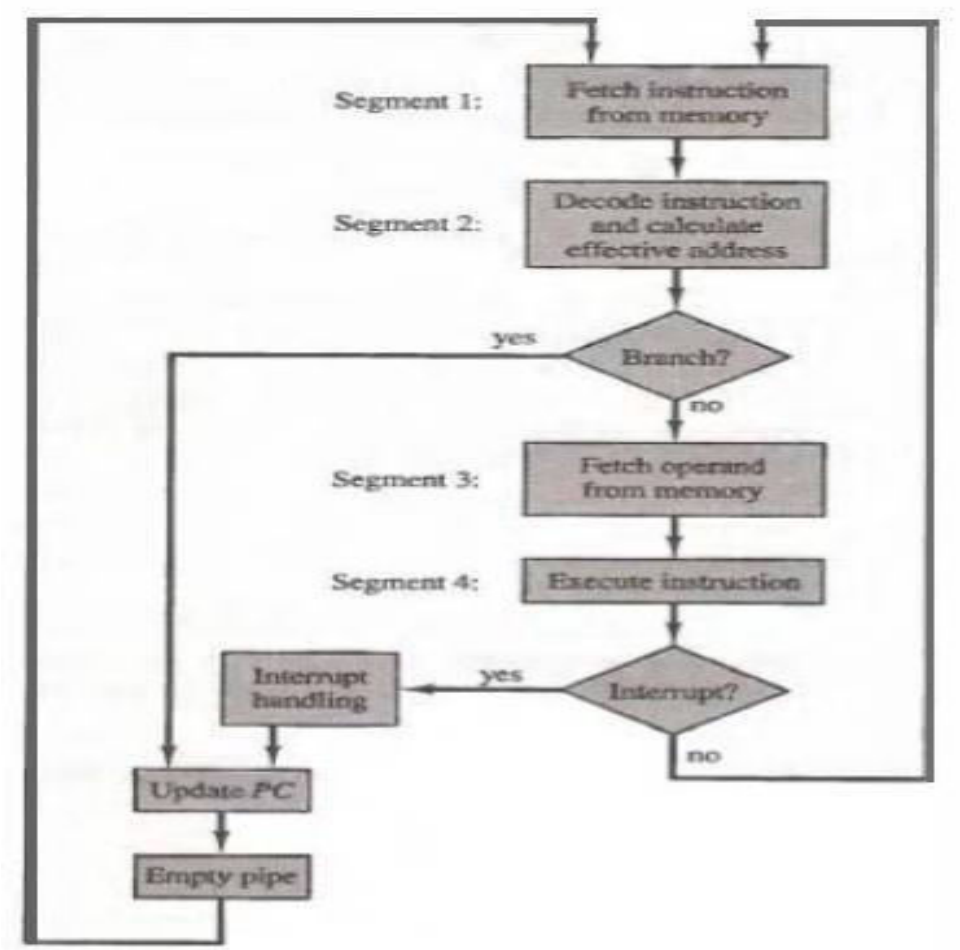


Fig shows the instruction cycle in the CPU can be processed with a four segment pipeline. While an instruction is being executed in segment 4, the next instruction in sequence is busy with fetching an operand from memory in segment 3. the effective address may be calculatedin a separate arithmetic circuit for the third instruction, and whenever the memory is available, the fourth and all subsequent instructions are placed in an instruction FIFO.

Fig 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.

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

1. FI is the segment that fetches an instruction.
2. DA is the segment that decodes the instruction and calculates the effective address.
3. FO is the segment that fetches the operand.
4. EX is the segment that executes the instruction.

It is assumed that the processor has separate instruction and data memories so that the operation in FI and FO 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 into segment FO; instruction 3 is being decoded in segment DA; and instruction 4 is being fetched from memory in segment FI.

Assume now this instruction is a branch instruction. As soon as this instruction is decoded in segment DA in step 4, the transfer from FI to DA of the other instructions are halted until the branch instruction is executed in step 6.

PIPELINE CONFLICTS:

1. RESOURCE CONFLICTS: They are caused by access to memory by two segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories.
2. DATA DEPENDENCY: these conflicts arise when an instruction depends on the result

of a previous instruction, but this result is not yet available.

3. BRANCH DIFFERENCE: they arise from branch and other instructions that change the value of PC.

5.4.2. DATA DEPENDENCY

A difficulty that may cause a degradation of performance in an instruction pipeline collision of data or address. A collision occurs when an instruction cannot proceed because previous instructions did not complete certain operations.

A data dependency occurs when an instruction needs data that are not yet available. For example, an instruction in the FO segment may need to fetch an operand that is being generated at the same time by the previous instruction in segment EX. Therefore, the second instruction must wait for the data to become available by the first instruction.

An address dependency may occur when an operand address cannot be calculated because the information needed by the addressing mode is not available. For example, an instruction with register indirect mode can not proceed to fetch the operand if the previous instruction is loading the address into the register. Therefore operand access to memory must be delayed until the required address is available.

Pipelined computers deal with such conflicts between data dependencies in a variety of ways.

The most straight forward method is to insert HARDWARE INTERLOCKS. An interlock is a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline. Detection of this situation causes the instruction whose source is not available to be delayed by enough clock cycles to resolve the conflict.

Another technique called OPERRND FORWADING uses special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline segments.

A procedure employed in some computers is to give the responsibility for solving data conflicts problems to the compiler. The compiler for such computers are designed to detect a data conflict and reorder the instruction as necessary to delay the loading of the conflicting data by inserting no-operation instructions. This method is referred to as DELAYED LOAD.

5.4.3. Handling of Branch Instructions

One of the major problems in operating the instruction pipeline is the occurrence of the branch instructions. A branch instruction can be conditional or unconditional. An unconditional branch always alters the sequential program flow by loading 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 is not satisfied.

Pipeline computers employ various hardware techniques to minimize the performance of degradation caused by instruction branching.One way of handling a conditional branch is to pre fetch the target instruction in addition to the instruction following the branch. Both are saved until the branch is executed. If the branch condition is successful, the pipeline continues from the branch target instruction

Another possibility is the use of BRANCH TARGET BUFFER or BTB. The BTB is an associative memory included in the fetch segment of the pipeline. Each entry in the BTB consists of the address of the previously executed branch instruction and the target instruction for that branch. It also stores the next few instructions after the branch target instruction. When the pipeline decodes a branch instruction, it searches the associative memory BTB for the address of the instruction. If it is in the BTB, the instruction is available directly and pre fetch continues from the new path. If the instruction is not in the BTB the pipeline shifts to a new instruction stream and stores the target instruction in the BTB.

5.5 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 an instruction pipeline using a small number of suboperations, with each being executed in one clock cycle. Because of the fixed-length instruction format, the decoding of the operation can occur at the same time as the register selection. All data manipulation instructions have register-to-register operations. Since all operands are in registers, there is no need for calculating an effective address or fetching of operands from memory. Therefore, the instruction pipeline can be implemented with two or three segments. One segment fetches the instruction from program memory, and the other segment executes the instruction in the ALU. A third segment may be used to store the result of the ALU operation in a destination register.

- The data transfer instructions in RISC are limited to load and store instructions.
    - These instructions use register indirect addressing. They usually need three or four stages in the pipeline.
    - To prevent conflicts between a memory access to fetch an instruction and to load or store an operand, most RISC machines use two separate buses with two 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 rate of one per clock cycle.

- In effect, it is to start each instruction with each clock cycle and to pipeline the 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 data conflicts and branch penalties.
  - RISC processors rely on the efficiency of the compiler to detect and minimize the delays encountered with these problems.

Example: Three-Segment Instruction Pipeline

- A typical set of instructions for a RISC processor are discussed earlier.
- Thee 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 instruction register.
  - The instruction is decoded at the same time that the registers needed for the execution 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 suboperations 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.

Delayed Load

- Consider the operation of the following four instructions:

  1. LOAD: R1 ← M[address 1]
  2. LOAD: R2 ← M[address 2]
  3. ADD: R3 ← R1 +R2
  4. STORE: M[address 3] ← R3

  There will be a *data conflict* in instruction 3 because the operand in R2 is not yet available in the A segment.
- This can be seen from the timing of the pipeline shown in Fig. 9-9(a).

The E segment in clock cycle 4 is in a process of placing the memorydata into R2. The A segment in clock cycle 4 is using the data from R2, but thevalue in R2 will not be the correct value since it has not yet been transferredfrom memory. It is up to the compiler to make sure that the instructionfollowing the load instruction uses the data fetched from memory. If thecompiler cannot find a useful instruction to put after the load, it inserts a no-op(no- operation) instruction. This is a type of instruction that is fetched frommemory but has no operation, thus wasting a clock cycle. This concept ofdelaying the use of the data loaded from memory is referred to as *delayed load*.

Figure 9-9(b) shows the same program with a no-op instruction insertedafter the load to R2 instruction. The data is loaded into R2 in clock cycle 4. Theadd instruction uses the value of R2 in step 5. Thus the no-op instruction isused to advance one clock cycle in order to compensate for the data conflict inthe pipeline. (Note that no operation is performed in segment A during clockcycle 4 or segment E during clock cycle 5.) The advantage of the delayed loadapproach is that the data dependency is taken care of by the compiler

ratherthan the hardware. This results in a simpler hardware segment since thesegment does not have to check if the content of the register being accessedis currently valid or not.

Clock cycles:	1	2	3	4	5	6
1. Load R1	I	A	E			
2. Load R2		I	A	E		
3. Add R1+R2			I	A	E	
4. Store R3				I	A	E

9 (a) Pipeline timing with data conflict

	1	2	3	4	5	6	7
1. Load R1	I	A	E				
2. Load R2		I	A	E			
3. No-operation			I	A	E		
4. Add R1+R2				I	A	E	
5. Store R3					I	A	E

9 (b) Pipeline timing with delayed load

Delayed Branch

- The method used in most RISC processors is to rely on the *compiler to redefine the branches* so that they take effect at the proper time in the pipeline. This method is referred to as *delayed branch*.
- The compiler is designed to analyze the instructions *before and after the branch* and

rearrange the program sequence by inserting useful instructions in the delay steps.

- It is up to the compiler to find useful instructions to put after the branch instruction. Failing that, the compiler can insert *no-op* instructions.

An Example of Delayed Branch

- The program for this example consists of five instructions.
  - Load from memory to R1
  - Increment R2
  - Add R3 to R4
  - Subtract R5 from R6
  - Branch to address X
- In Fig. 9-10(a) the compiler inserts *two no-op instructions* after the branch.
- The branch address X is transferred to PC in clock cycle 7.
- The program in Fig. 9-10(b) is rearranged by placing the add and subtract instructions *after the branch instruction*.
- PC is updated to the value of X in clock cycle 5.
-

Clock cycles:	1	2	3	4	5	6	7	8	9	10
1. Load	I	A	E							
2. Increment		I	A	E						
3. Add			I	A	E					
4. Subtract				I	A	E				
5. Branch to X					I	A	E			
6. No-operation						I	A	E		
7. No-operation							I	A	E	
8. Instruction in X								I	A	E

10 (a) Using no-operation instructions

Clock cycles:	1	2	3	4	5	6	7	8
1. Load	I	A	E					
2. Increment		I	A	E				
3. Branch to X			I	A	E			
4. Add				I	A	E		
5. Subtract					I	A	E	
6. Instruction in X						I	A	E

10 (b) Rearranging the instructions

In Fig. 9-10(a) the compiler inserts two no-op instructions after the branch.The branch address X is transferred to PC in clock cycle 7. The fetching of theinstruction at X is delayed by two clock cycles by the no-op instructions. Theinstruction at X starts the fetch phase at clock cycle 8 after the program counterPC has been updated.

The program in Fig. 9-10(b) is rearranged by placing theadd and subtractinstructions after the branch instruction instead of before as in the originalprogram. Inspection of the pipeline timing shows that PC is updated to thevalue of X in clock cycle 5, but the add and subtract instructions are fetchedfrom memory and executed in the proper sequence. In other words, if the loadinstruction is at address 101 and X is equal to 350, the branch instruction isfetched from address 103. The add instruction is fetched from address 104 andexecuted in clock cycle 6. The subtract instruction is fetched from address 105and executed in clock cycle 7. Since the value of X is transferred to PC with clockcycle 5 in the E segment, the instruction fetched

from memory at clock cycle6 is from address 350, which is the instruction at the branch address.

5.6 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 o Seismic data analysis o 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)
```

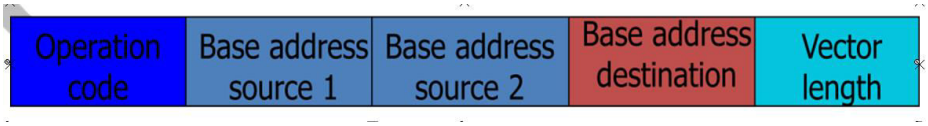
- This is implemented in machine language by the following sequence of operations.

```
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. 9-11.
  - 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 n^2 inner products or n^3 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

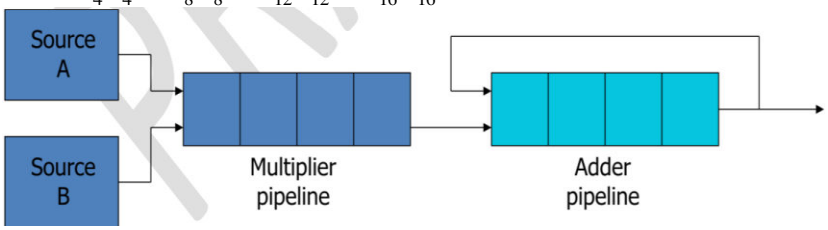
- c11= a11b11+ a12b21+ a13b31
- This requires three multiplication and (after initializing c11 to 0) three additions.

• 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.

• The inner product calculation on a pipeline vector processor is shown in Fig. 9-12.

$$\begin{aligned} C = {} & A_1B_1 + A_5B_5 + A_9B_9 + A_{13}B_{13} + \cdots \\ & + A_2B_2 + A_6B_6 + A_{10}B_{10} + A_{14}B_{14} + \cdots \\ & + A_3B_3 + A_7B_7 + A_{11}B_{11} + A_{15}B_{15} + \cdots \\ & + A_4B_4 + A_8B_8 + A_{12}B_{12} + A_{16}B_{16} + \cdots \end{aligned}$$

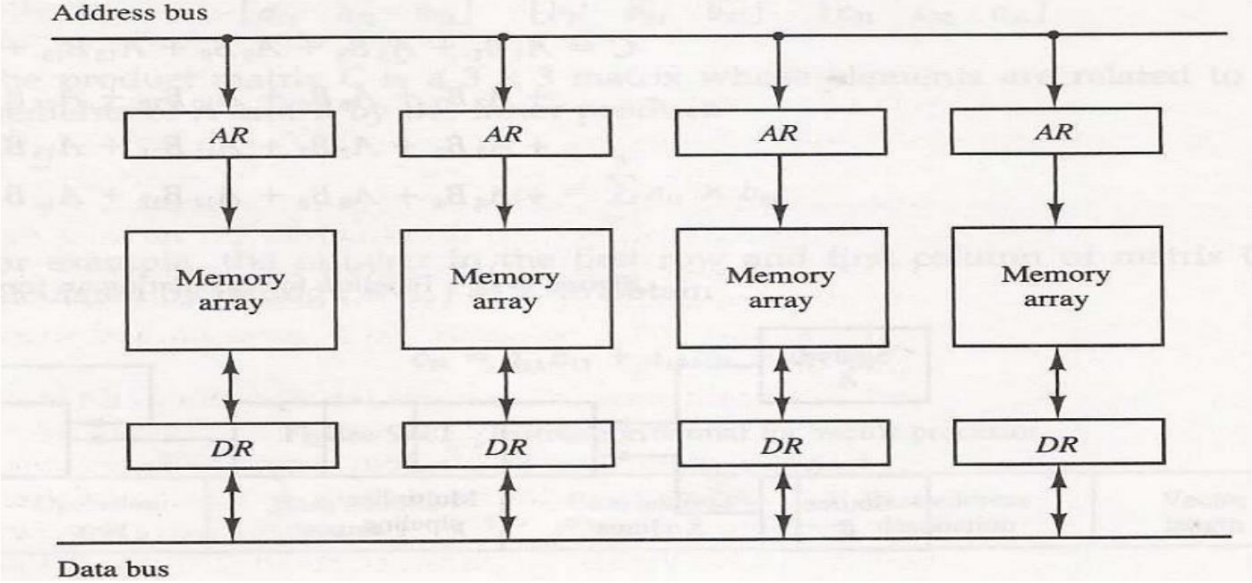


Memory Interleaving

- *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.

- Fig. 9-13 shows a memory unit with four modules.

Figure 9-13 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

5.7 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
- Fig. 9-14 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.

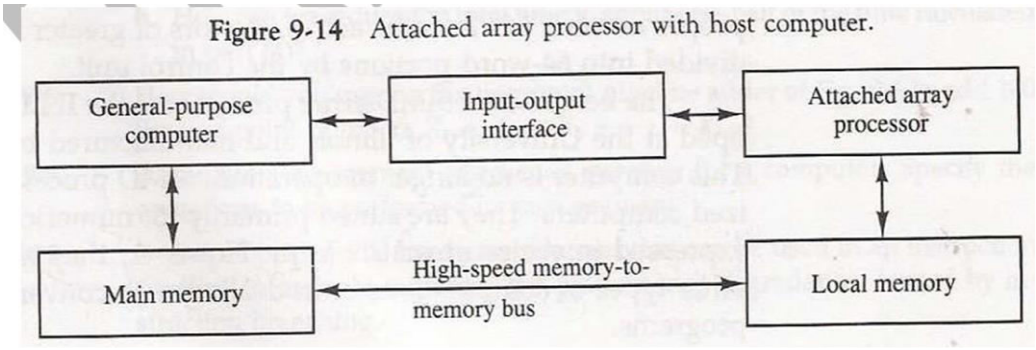


Figure 9-14 Attached array processor with host computer.

- 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

- An SIMD array processor is a computer with multiple processing units operating in parallel.
- A general block diagram of an array processor is shown in Fig. 9-15.
  - 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.

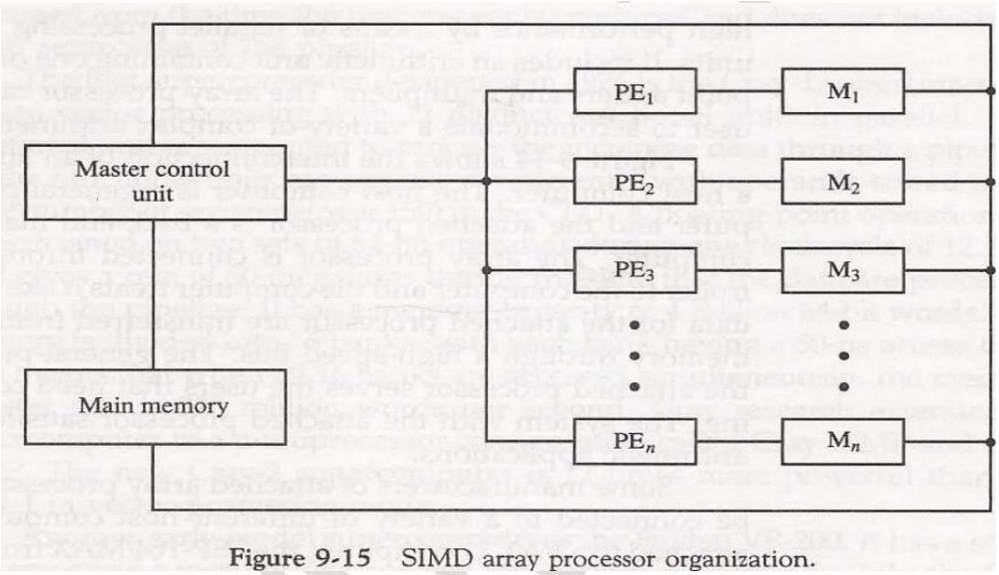


Figure 9-15 SIMD array processor organization.

- For example, the ILLIAC IV computer developed at the University of Illinois and manufactured by the Burroughs Corp.
  - Are highly specialized computers.
  - They are suited primarily for numerical problems that can be expressed in vector or matrix form.

Unit-VI
MULTIPROCESSORS

5.1Characteristics of multiprocessors

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 (MIMD) systems.

A multiprocessor system is controlled by one operating system that provides interaction between processors and all the components of the system. A multiprocessor improves the reliability of the system so that a failure or error in one part has a limited effect on the rest ofthe system. If a fault causes one processor to fail, a second processor can be assigned to perform the function of the disabled processor. This causes the loss in efficiency of the system but with no delay.

Multiprocessors can improve performance by decomposing the program into parallel executable tasks. This can be achieved in two ways.

*Multiple independent jobs can be made to operate in parallel.

*A single job can be partitioned into multiple parallel tasks.

Multiprocessors are classified by the way their memory is organized

1. Shared memory or tightly coupled multiprocessor:

A multiprocessor system with common shared memory is classified as shared memory. It provides a cache memory with each CPU. There will be a global common memory that all CPUs can access. Information can therefore be shared among the CPUs by placing it in the common global memory.

2. Disturbed memory or loosely coupled system:

Each processor element in a loosely coupled system has its own private local memory. The processors relay programs and data to other processors in packets, which consists of an address, the data content, and error detecting code.

5.2. Interconnection Structures

There are several physical forms available for establishing an interconnection network. Some of these schemes are

1. Time-shared common bus

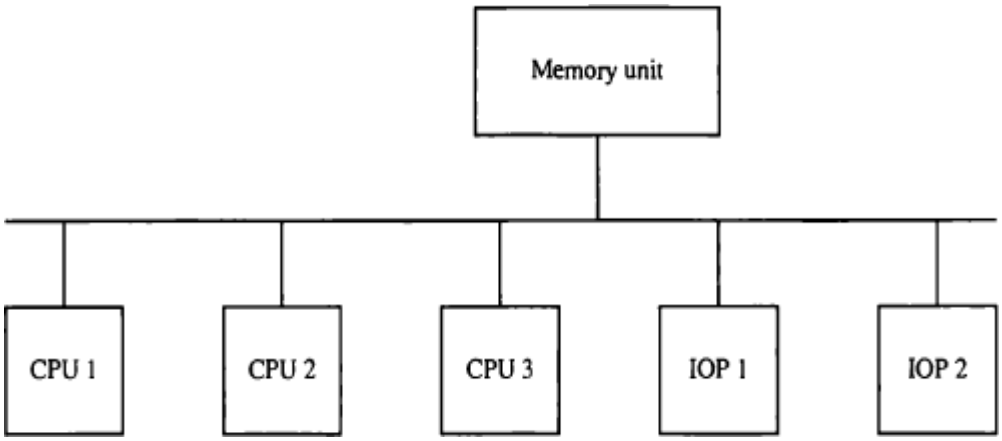
2. Multiport memory
3. Crossbar switch
4. Multistage switching network
5. Hypercube system

1. Time-shared common bus:

A common bus multi processor system consists of a number of processors connected through a common path to a memory unit. Only one processor can communicate with the memory or another processor at any given time. Any processor wishing to initiate a transfermust first determine the availability of the bus. A command is issued to inform the destination unit what operation is to be performed. The receiving unit recognizes its address and responds to the control signals from the sender, after this the transfer will be initiated.

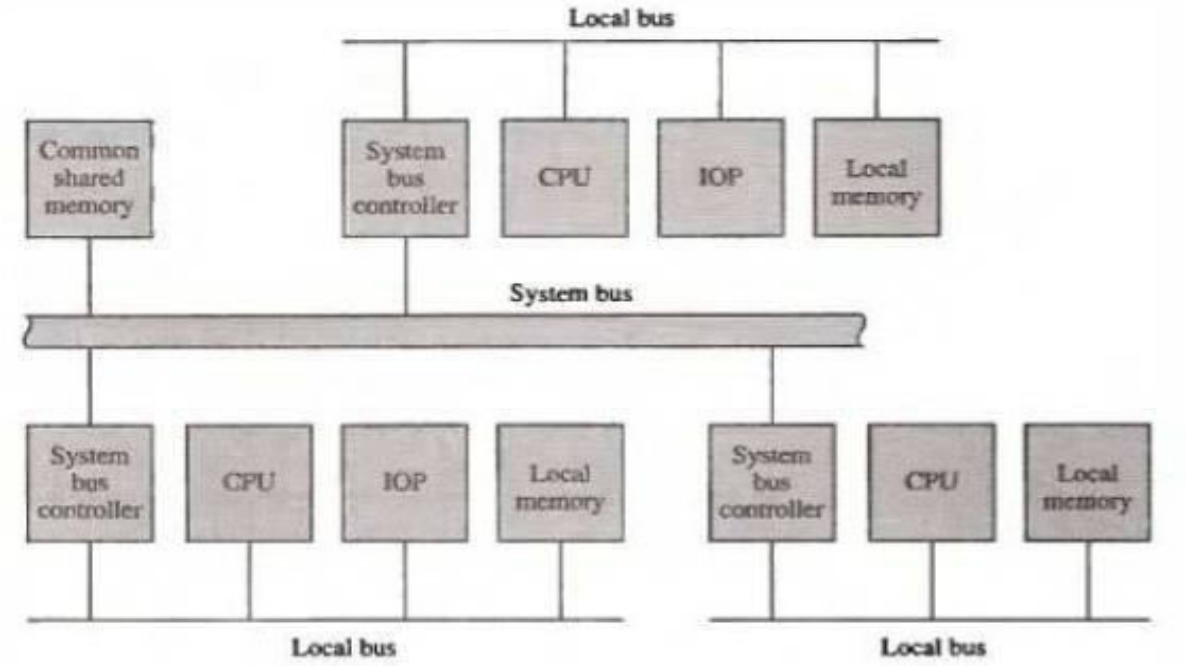
The system may exhibit transfer conflicts since all processors share one common bus. These conflicts must be resolved by putting a bus controller that establishes priorities among the requesting units.

The processors in the system are kept busy through the implementation of two or more buses to permit multiple simultaneous bus transfer. It increases the system cost and complexity.



Time shared common bus organization

Consider implementation of dual bus structure. In this a number of local buses connected to its own local memory and to one or more processors. Each local bus is connected to a CPU, an IOP or any combination of processors. A system bus controller links each local bus to common system bus. I/O devices connected to local memory as well as to local IOP are available to local processor



System bus structure for multiple processors

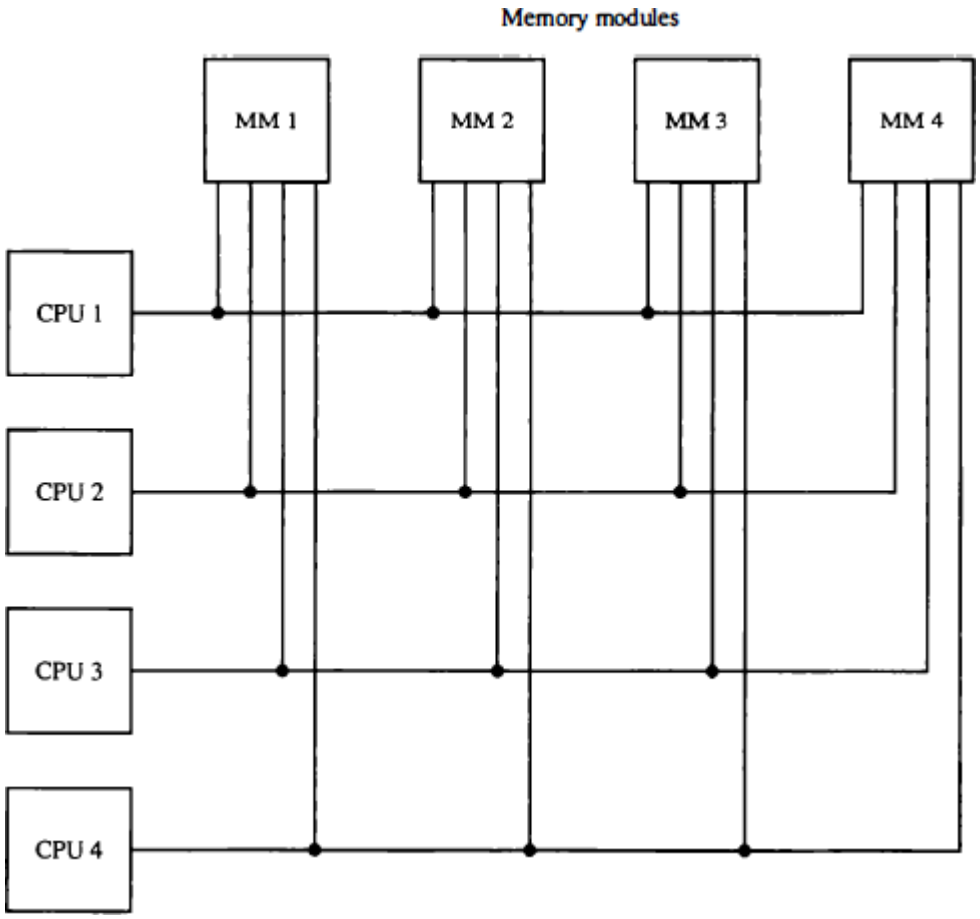
If an IOP is connected directly to the system bus, the I/O devices attached to it may be made available to the processors. Only one processor can communicate with the shared memory and other common resources through the system bus at any given time.

The other processors are kept busy communicating with their local memory and I/O devices. Part of local memory may be designed as a cache memory attached to CPU. In this way average access time of local memory can be made to approach the cycle time of CPU to which it is attached.

2. Multiport memory:

A multiport memory system employs separate buses between each memory module (MM) and each CPU. 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. MM is said to have 4ports and each port accommodates one of the buses. The module must have internal control logic to determine which port will have access to memory at any given time.

The advantage of multiport memory organization is the higher transfer rate because of multiple paths between processor and main memory. The disadvantage is it requires expensive control logic and a large number of cables and connectors. This structure is appropriate for a system with large number of processors.



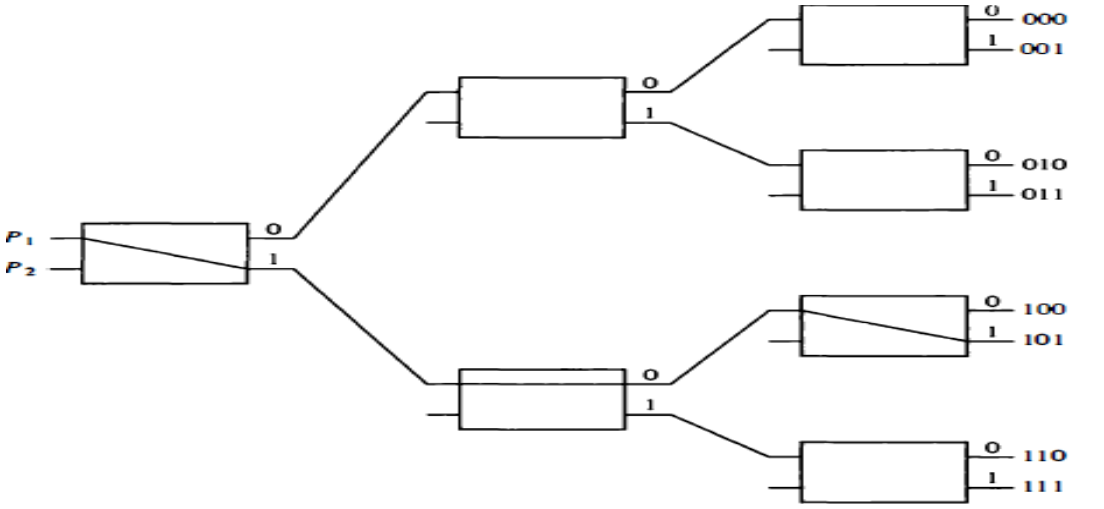
Multiport memory organization

4. Multistage Switching Network:

The basic component of a multistage network is a two-input, two-output interchange switch. A 2*2 switch has two inputs, labeled A and B and two-outputs labeled 0 and 1. The switch has the capability of connecting inputs A or B to either of the outputs. If inputs A and B both requests for the same output only one of them will be connected other will be blocked.

The two processors P1 and P2 are connected through switches to eight memory modules marked in binary from 000 through 111. The first bit of destination number determines the switch output in the first level.

For example to connect P1 to memory 101,it is necessary to form paths from P1 to o/p 1 in t he 1st level switch, o/p 0 in the 2nd level switch and o/p in the 3rd level.



Omega Network:

Omega switching network have been proposed for multistage switching network to control processor-memory communication. In this configuration there is exactly one path from each source to any particular destination. A particular request is initiated in the switching network by 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*2 switch setting. When the request arrives on either input of 2*2 switch, it is routed to the upper output if the specified bit is 0 or to the lower output if it is 1.

Omega Network in a tightly coupled multiprocessor:

In this the source is a memory module and the destination is a memory module. The first pass through the network sets up the path. Succeeding paths are used to transfer the address into memory and then transfer the data in read or write direction.

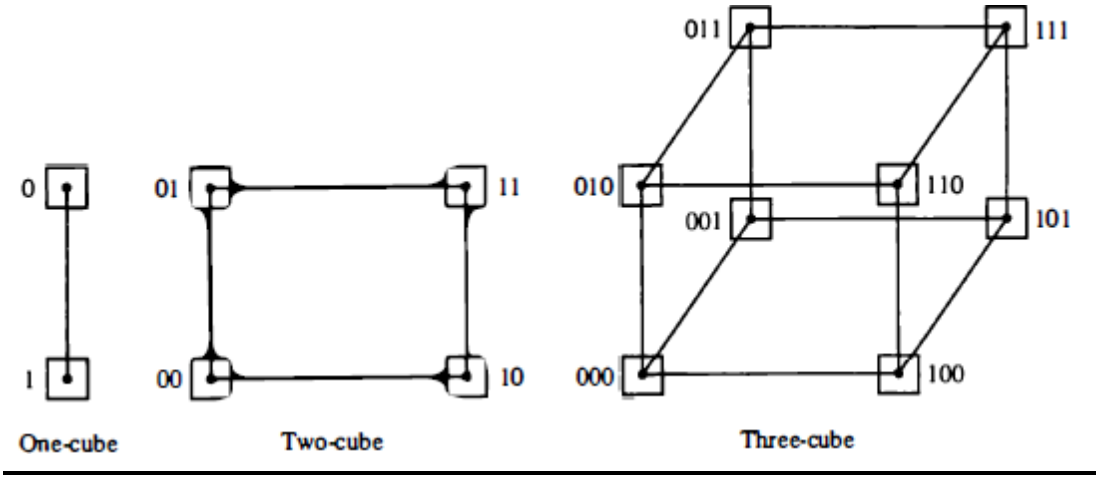
Omega Network in loosely coupled multiprocessor:

In this both the source and the destination are the processing elements. After the path is established the source processor transfers a message to the destination processor.

5. Hypercube Interconnection

The hypercube or binary n-cube multiprocessor structure is a loosely coupled system composed of 2 power of n processors interconnected in an n-dimensional binary cube. Each processor forms a node of the cube. Each processor has direct communication paths to n other neighbor processors. Each processor address differs from that of its neighbors by exactly one bit position.

Routing messages through an n-cube structure may take from one to n links from a source node to a destination node. For example, in a three-cube structure node 000 can communicate directly with node 001. It must cross at least two links to communicate with 011. Computing the ex-or of the source node address with the destination node address can develop a routing procedure.



5.3. Interprocessor Arbitration

Computer systems contain a number of buses at various levels to facilitate the transfer of information between components. A bus that connects the major components in a multiprocessor 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 through the system bus. The requesting processor may wait if another processor is currently utilizing the system bus. Arbitration must then be performed to resolve this multiple contention for the shared resources. Arbitration logic is a part of 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 lines.

Six bus arbitration signals

These are used for Interprocessor arbitration.

Bus request	BREQ
Common Bus request	CBRQ
Bus busy	BUSY
Bus clock	BCLK
Bus priority in	BPRN
Bus priority out	BPRO

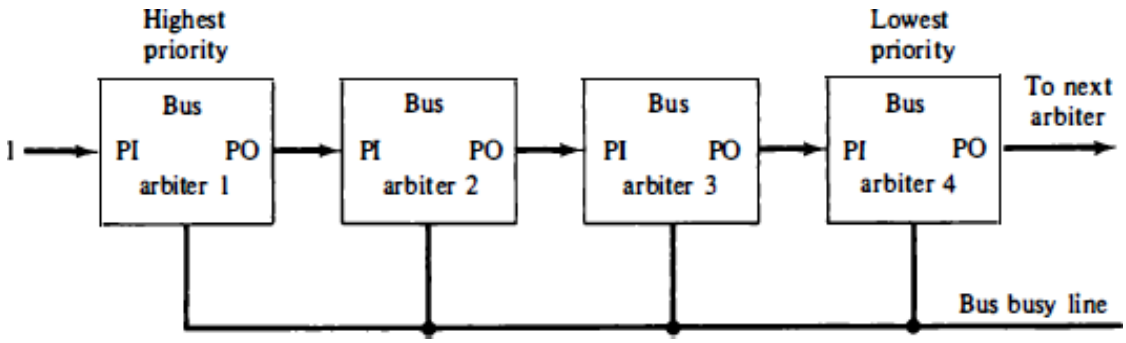
Functions of the Bus Arbitration Signals

The bus priority –in BPRN and the bus priority-out BPRO are used for a daisy-chain connection of bus arbitration circuits. The bus BUSY signal is an open collector output to instruct all arbiters when the bus is busy. The common bus request CBRQ is also an open collector output used to instruct the arbiter if there are any other arbiters of lower-priority requesting the use of bus. The signals used to connect parallel arbitration procedure are bus request BREQ and priority-in BRPN for request and ack signals respectively. The bus clock BCLK is used to synchronize all bus transactions

Serial Arbitration Procedure:

A hardware bus priority resolving techniques can be established by means of a serial or parallel connection of units requesting control of the system bus.

Serial priority resolving technique is obtained from a daisy-chain interconnection of bus arbitration circuits. Each processor has its own bus arbitration logic with priority in and priority out lines. The priority out (PO) of each arbiter is connected to priority in (PI) of next lower-priority. The PI of highest priority unit is maintained at logic 1.



Serial Arbitration

The processor whose arbiter has a PI=1 and PO=0 is one that is given the control of the system bus. The PO output for a particular arbiter is equal to 1 f its PI input is equal to 1 and the processor associated with the arbiter logic is not requesting the control of the bus.

Parallel Arbitration Logic:

The parallel arbitration logic uses an external priority encoder and a decoder. Each bus arbiter in this parallel scheme has a bus request output line and a bus ack input line. The processor takes control of the bus if its ack input line is enabled. First the request lines from the four arbiters 4*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*4 decoder, which enables the proper ack line to grant bus access to the highest priority unit.

Dynamic Arbitration Algorithms:

In a dynamic priority algorithm gives the system the capability for changing the priority of the devices while the system is in operation. The few arbitration procedures that follow dynamic priority algorithms are:

Time slice: It allocates a fixed length time slice of bus time that is offered sequentially to each processor. The service given to each component is independent of its location along the bus. No preference is given to any particular since each is given same amount of time to communicate with the bus.

Polling: In a bus system that uses polling, poll lines replace bus grant signal. The bus controller to define an address for each device connected to the bus uses Poll lines. After a number of bus cycles, the polling process continues by choosing different processor.

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. With this procedure no processor is favored over any other since the priorities are dynamically changed to give every device an opportunity to access the bus.

FIFO: In the first-come, first-serve scheme, requests are served in the order received. For this the bus controller establishes a queue according to the time that the bus requests arrive. Each processor must wait for its turn to use the bus on FIFO basis.

Rotating daisy-chain: This procedure is a dynamic extension of 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. Each arbiter priority for a given bus cycle is given 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 lowest priority.

5.4. Interprocessor Communication

The various processors in a multiprocessor system must be provided with a facility for communicating with each other.

In a shared memory multi processor system, the most common procedure is to set aside a portion of memory that is accessible to all processors. The primary use of common memory is to act as a message center similar to a 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 places it in the memory mail box, whether it has meaningful information, and which processor it is intended. The receiving processor can check the mailbox periodically if there are valid messages for it.

In addition to shared memory, a multiprocessor system may have other shared resources. To prevent conflict use of shared resources by several processors there must be a provision for assigning resources to these processors. This task is given to the operating system.

There are three organizations that have been used in the design of the operating system for multiprocessors:

1. **Master-slave configuration**: one processor designated the master, always executes the operating system functions. The remaining processors as slaves do not perform the 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.

2. **Separate operating system:** Each processor can execute the operating system routines its needs. This is more suitable for loosely coupled systems where every processor may have its own copy of the entire os.

3. **Distributed operating system:** In this each particular os has one processor at a time. This type of organization is also called floating os since the routine floats from one processor to another and the execution of routine may be assigned to different processors at different times.

In a loosely coupled multiprocessor system the memory is distributed among the processors and there is no shared memory for passing information. The communications between processors is by means of message passing through I/O channels. When the sending processor and the receiving processor name each other as source and destination, a channel of communication is established.

5.5. Interprocessor Synchronization

Synchronization is needed to enforce the correct sequence of processes and to ensure mutually exclusive access to shared writable data. A number of hardware mechanisms for mutual exclusion have been developed. One of the most popular methods is binary semaphore.

Mutual exclusion with a Semaphore: It is necessary 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 it is in a mutual section. It 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. It is a software-controlled flag that is stored in a memory location that all processors can access. When it is equal to 1, it means that a processor is executing a critical program, so that the shared memory is not available to other processors. When it is equal to 0, the shared memory is available to any requesting processor.

Testing and setting the semaphore is itself a critical operation and must be performed as a single indivisible operation. This action would allow simultaneous execution of a critical section at the same time, which can result in erroneous initialization of control parameters and a loss of essential information.

A semaphore can be initialized by means of a test and set instruction in 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 other processor changes the semaphore between the time that the processor is testing it and the time that it is setting it.

Cache Coherence:

The primary advantage of cache is to its ability to reduce the average access time in uniprocessors. When a processor finds a writ operation in cache during write operation there are two commonly used procedures to update memory.

1. write-through policy:

Both cache and maim memory are updated with every write operation

2. Write back policy:

Only cache is updated and the location is marked so that it can be copied later into the main memory.

A memory scheme is coherent if the value returned on a load instruction is always the value given by the latest store instruction with the same address. Cache coherence problems exist in multiprocessors with private caches because of the need to share writable data.

Conditions for cache incoherence

To illustrate the problem of cache coherence, consider three processor configurations with private caches. Sometime during the operation an element X from main memory is loaded into the three processors P1, P2, P3. It is also copied into the private caches of the three processors. Consider X as 52. The load on X to the three processors results in consistent copies in the cache and main memory.

A store to X 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 still they hold the same old value.

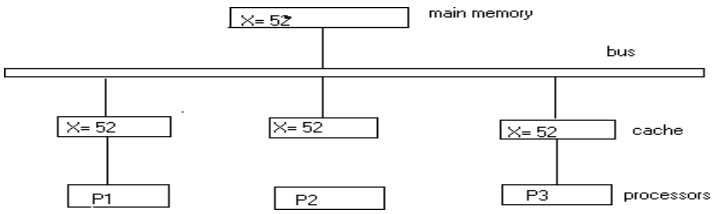
In writ back policy; main memory is not updated at the time of store. The copies in the other two caches and main memory are inconsistent. Memory is updated when the modified data is copied back into the main memory.

Solutions to the cache coherence problem:

Cache coherence problems can be solved by means of software and hardware combinations.

In the software solution, it is desirable to attach a private cache to each processor. If cache allows nonshared and read only data, such items are called cachable. Shared write only data

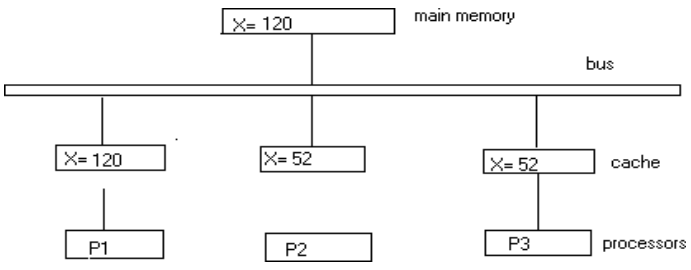
are non cachable which remains in main memory. This method restricts the type of data storedin caches



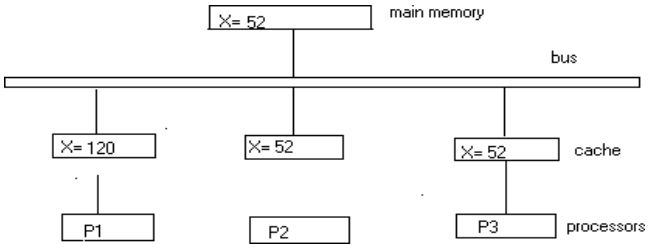
Cache configuration after a load X

Another scheme that allows writable data to exist in one cache is a method that employs centralized global table in its complier. Each block is identified as read only (RO) or write and read (WR). Only one copy of cache is RO. Thus if data is updated in RW block othercaches are not affected because they do not have a copy of this block.

In the hardware solution, a cache controller called snoopy 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 these network for possible write operations. It is basically a hardware unit designed to maintain a bus –watching mechanism over all the caches attaches to the bus. In this way inconsistent ways are prevented.



With write through cache policy



With write back cache policy