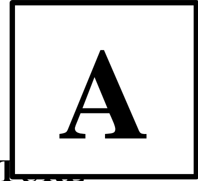
SNS COLLEGE OF TECHNOLOGY

(An Autonomous Institution)
Coimbatore – 641 035.

B.E / B.Tech – Internal Assessment Exam- I
Academic Year 2023-2024 (ODD)
FIFTH SEMESTER (REGULATION R2019)



19ITT202 – COMPUTER ORGANIZATION AND ARCHITECTURE

TIME: 1.5 HOURS **MAXIMUM MARKS: 50**

ANSWER KEY

PART A — (5 x 2 = 10 Marks)			

1.	Give a short sequence of machine instructions for the task "Add the contents of R1 and memory location 2500, and place the answer in location C"
	ADD [2500],R1
	Mov R1, C
		CO1	APP

2.	Give the advantage of cache memory. Cache memory is faster than main memory. It consumes less access time as compared to main memory. It stores the program that can be executed within a short period of time. It stores data for temporary use	CO1	REM
3.	List down the Various register. Instruction register(IR) Program counter(PC) Memory address register(MAR) Memory data register(MDR) General purpose registers (R_0 to R_{n-1})	CO1	REM
4.	What is half adder? Half Adder is a combinational logic circuit which is designed by connecting one EX-OR gate and one AND gate. The half adder circuit has two inputs: A and B, which add two input digits and generates a carry and a sum.	CO2	REM
5.	What is an overflow? When does it occur? The rules for detecting overflow in a two's complement sum are simple: If the sum of two positive numbers yields a negative result, the sum has overflowed. If the sum of two negative numbers yields a positive result, the sum has overflowed..	CO2	UND

PART B — (2 x 13 = 26 Marks)

Q.No		Question	Marks	CO	BT
6.	(a)	Discuss in detail about the various measures of performance of a computer.	13	CO1	UND

Performance of a computer can be measured by speed with which it can execute the program. Speed of the computer is affected by

- Hardware design
- Machine language instruction of the computer. Because the programs are usually written in high level language.
- Compiler, which translates high-level language into machine language.

For best performance, it is necessary to design a complier, machine instruction set, and the hardware in a coordinated way.

Consider a Time line diagram to describe how the operating system overlaps processing, disk transfers, and printing for several programs to make the best possible use of the resources available. The total time required to execute the program is t5 - t0. This is called elapsed time and it is the measure of the performance of the entire computer system.

It is affected by the speed of the processor, the disk and the printer. To discuss the performance of the processor we should only the periods during which the processor is active.

User program and OS routine sharing of the processor

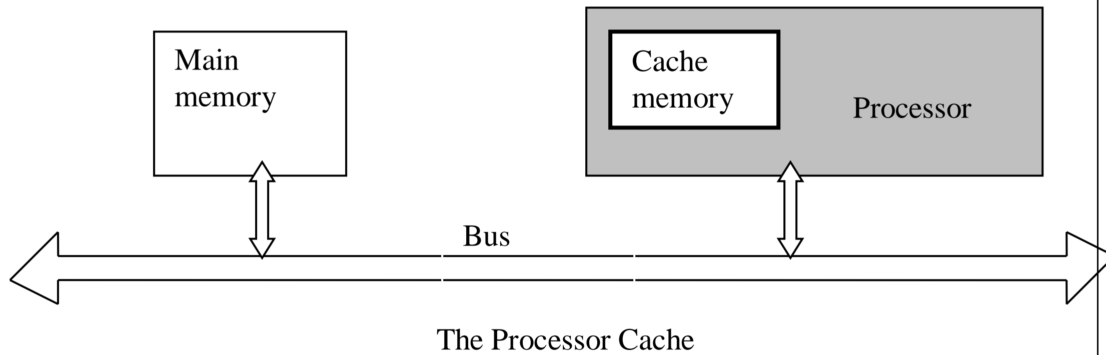
Pointer
Disk
OS routines
Program
t0 t1 t2 t3 t4 t4
Time

Elapsed time for the execution of the program depends on hardware involved in the execution of the program. This hardware includes processor and the memory which are usually connected by a BUS (As shown in the bus structure diagram.).

When the execution of the program starts, all program instructions and the required data are stored in the main memory. As execution proceeds, instructions are fetched from the main memory

one by one by the processor, and a copy is placed in the cache. When execution of the instruction calls for the data located in the main memory, the data are fetched and a copy is placed in the cache. If the same instruction or data is needed later, it is read directly from the cache. The processor and a small cache memory are fabricated into a single IC chip. The speed of such chip is relatively faster than the speed at which instruction and data can be fetched from the main memory. A program can be executed faster if the movement of the instructions and data between the main memory and the processor is minimized, which is achieved by using the cache.

To evaluate the performance, we can discuss about,



The Processor Cache

- ✓ Processor clock
- ✓ Basic performance equation
- ✓ Pipelining and Superscalar operation
- ✓ Clock Rate
- ✓ Instruction Set: CISC and RISC
- ✓ Compiler
- ✓ Performance Measurement

Processor clock

Processor circuits are controlled by timing signal called a clock. The clock defines regular time intervals, called clock cycle. To execute a machine instruction, the processor divides the action to be performed into sequence of basic steps, such that each step can be completed in one clock cycle.

Length of one clock cycle is P and this parameter P affects processor performance. It is inversely proportional to clock rate

$$R=1/P$$

This is measured in cycles per second.

Processors used in today's personal computers and workstations have clock rates from a few hundred millions to over a billion cycles per second is called hertz (Hz). The term "million" is denoted by the prefix Mega (M) and "billion" is denoted by prefix Giga (G). Hence, 500 million cycles per second is usually abbreviated to 500Mega Hertz (MHz). And 1250 million cycles per second is abbreviated to 1.25 Giga Hertz (GHz). The corresponding clock periods are 2 and 0.8 nano seconds (ns) respectively.

Basic performance equation

Let T be the time required for the processor to execute a program in high level language. The compiler generates machine language object program corresponding to the source program.

Assume that complete execution of the program requires the execution of N machine language instructions.

Assume that average number of basic steps needed to execute one machine instruction is S, where each basic step is completed in one clock cycle.

If the clock rate is R cycles per second, the program execution time is given by

$$T = (N \times S) / R$$

This is often called Basic performance equation.

To achieve high performance, the performance parameter T should be reduced. T value can be reduced by reducing N and S, and increasing R.

- Value of N is reduced if the source program is compiled into fewer number of machine instructions.
- Value of S is reduced if instruction has a smaller no of basic steps to perform or if the execution of the instructions is overlapped.
- Value of R can be increased by using high frequency clock, ie. Time required to complete a basic execution step is reduced.
- N, S and R are dependent factors. Changing one may affect another.

Pipelining and Superscalar operation

Pipelining

It is a technique of overlapping the execution of successive instructions. This technique improves performance.

Consider the instruction

Add R1, R2, R3

The above instruction adds the contents of registers R1 and R2, and places the sum to R3. The contents of R1 and R2 are first transferred to the inputs of the ALU. After addition is performed the result is transferred to register R3 from the processor.

Here processor can read the next instruction to be executed while performing addition operation of the current instruction and while transferring the result of addition to ALU, the operands required for the next instruction can be transferred to the processor. This process of overlapping the instruction execution is called **Pipelining**.

- ✓ If all the instructions are overlapped to the maximum degree, the effective value of S is 1. It is impossible always.
- ✓ Individual instructions require several clock cycles to complete but for the pupose of computing T, effective value of S is 1.

Superscalar operation

A higher degree of concurrency can be achieved if multiple instruction pipelines are implemented in the processor. This means that multiple functional units are used, creating parallel paths through which different instruction 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. So there is possibility of reducing the S value even

less than 1.

Parallel execution should preserve the logical correctness of programs. That is the result produced must be same as those produced by serial execution of program executions.

Clock Rate

There are two possibilities for increasing the clock rate, R.

- First, improving the integrated-circuit (IC) technology makes logic circuits faster, which reduces the time needed to complete a basic step. This allows the clock period, P, to be reduced and the clock rate, R, to be increased.
- Second, 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 instruction remain the same, the number of basic steps needed may increase.

Increases in the value of R by improvements in IC technology affect 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 a cache, the percentage of accesses to the main memory is small. Hence, much of the performance can be improved.

The value of T will be reduced by the same factor as R is increased because S and N are not affected.

Instruction Set: CISC and RISC

CISC: Complex Instructional Set Computers

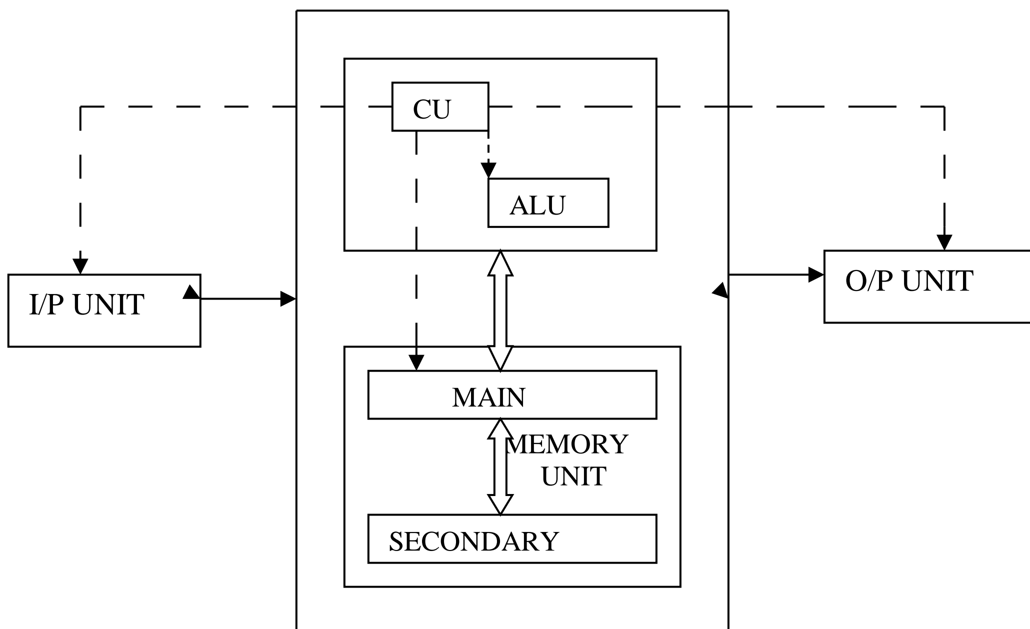
RISC: Reduced Instructional Set Computers

- 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 instructions,a large number of instructions may be needed to perform a given programming task.This could lead to a large value for N and a small value for S.
- On the other hand, if individual instructions perform more complex operations, 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.
- Processors with simple instructions are called Reduced Instruction Set Computers (RISC) and processors with more complex instructions are referred to as Complex Instruction Set Computers (CISC)
- The decision for choosing the instruction set is done with the use of pipelining. Because the effective value of S is close 1.

Compiler

A compiler translates a high-level language program into a sequence of machine instructions. To reduce N, we need to have a suitable machine instruction set and a compiler that makes good use of it.

An optimizing compiler takes advantage of various features of the target processor to reduce the product N x S, which is the total number of clock cycles needed to execute a program. The number of cycles is dependent not only on the choice of instructions, but also on the order in which they appear in the program. The compiler may rearrange program instructions to achieve better performance without changing the logic of the program.

		Complier and processor must be closely linked in their architecture. They should be designed at the same time. **Performance Measurement** The computer community adopted the idea of measuring computer performance using benchmark programs. To make comparisons possible, standardized programs must be used. The performance measure is the time it takes a computer to execute a given benchmark program. A nonprofit organization called System Performance Evaluation Corporation (SPEC). $$\text{SPEC rating} = \frac{\text{Running time on the reference computer}}{\text{Running time on the computer under test}}$$ The test is repeated for all the programs in the SPEC suite, and the geometric means of the results are computed. Let SPECi be the rating for program i in the suite. The overall SPEC rating for the computer is given by $$\text{SPEC rating} = \left(\prod_{i=1}^{n} SPEC_i \right)^{1/n}$$			
(OR)					
	(b)	Explain the various components of computer System with neat diagram. **A computer consists of five functionally independent main parts. They are,** • Input • Memory • Arithmetic and logic • Output • Control unit Basic functional units of a computer  The operation of a computer can be summarized as follows The computer accepts programs and the data through an input and stores them in the memory.The stored data are processed by the arithmetic and logic unit under program control.The processed data is delivered through the output unit.All above activities are directed by control unit.The	13	CO1	UND

information is stored either in the computer's memory for later use or immediately used by ALU to perform the desired operations.Instructions are explicit commands that

- Manage the transfer of information within a computer as well as between the computer and its I/O devices.
- Specify the arithmetic and logic operations to be performed.

To execute a program, the processor fetches the instructions one after another, and performs the desired operations.

The processor accepts only the machine language program.

To get the machine language program, Complier is used.

Note: Compiler is software (Translator) which converts the High Level Language program (source program) into Machine language program (object program)

1.Input unit:

The computer accepts coded information through input unit. The input can be from human operators, electromechanical devices such as keyboards or from other computer over communication lines.

Examples of input devices are

Keyboard, joysticks, trackballs and mouse are used as graphic input devices in conjunction with display.

Microphones can be used to capture audio input which is then sampled and converted into digital code for storage and processing.

Keyboard

- It is a common input device.
- Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted over cable to the memory of the computer.

2.Memory unit:

Memory unit is used to store programs as well as data.

Memory is classified into primary and secondary storage.

Primary storage:

It also called main memory.

It operates at high speed and it is expensive.

It is made up of large number of semiconductor storage cells, each capable of storing one bit of information.

These cells are grouped together in a fixed size called word. This facilitates reading and writing the content of one word (n bits) in single basic operation instead of reading and writing one bit for each operation

Each word is associated with a distinct address that identifies word location. A given word is accessed by specifying its address.

Word length:

The number of bits in each word is called word length of the computer.

Typical word lengths range from 16 to 64bits.

Programs must reside in the primary memory during execution.

RAM:

It stands for Random Access Memory. Memory in which any location can be reached in a short and fixed amount of time by specifying its address is called random-access memory.

Memory access time

- Time required to access one word is called Memory access time.
- This time is fixed and independent of the word being accessed.
- It typically ranges from few nano seconds (ns) to about 100ns.

Caches

They are small and fast RAM units.

They are tightly coupled with the processor.

They are often contained on the same integrated circuits(IC) chip to achieve high performance.

]Secondary storage:

It is slow in speed.

It is cheaper than primary memory.

Its capacity is high.

It is used to store information that is not accessed frequently.

Various secondary devices are magnetic tapes and disks, optical disks (CD-ROMs), floppy etc.

3.Arithmetic and logic unit:

Arithmetic and logic unit (ALU) and control unit together form a processor.

Actual execution of most computer operations takes place in arithmetic and logic unit of the processor.

Example:

Suppose two numbers located in the memory are to be added. They are brought into the processor, and the actual addition is carried out by the ALU.

Registers:

Registers are high speed storage elements available in the processor.

Each register can store one word of data.

When operands are brought into the processor for any operation, they are stored in the registers.

Accessing data from register is faster than that of the memory.

4.Output unit

The function of output unit is to produce processed result to the outside world in human understandable form.

Examples of output devices are Graphical display, Printers such as inkjet, laser, dot matrix and so on. The laser printer works faster.

5.Control unit:

Control unit coordinates the operation of memory, arithmetic and logic unit, input unit, and output unit in some proper way. Control unit sends control signals to other units and senses their states.

Example:

Data transfers between the processor and the memory are controlled by the control unit through timing signals.

Timing signals are the signals that determine when a given action is to take place.

Control units are well defined, physically separate unit that interact with other parts of the machine.

A set of control lines carries the signals used for timing and synchronization of events in all units

---

7.	(a)	Explain briefly on instruction set and its types with an example of each type.	13	CO1	UND

BASIC INSTRUCTION TYPES

The operation of addition of two numbers is a fundamental capability in any computer. The statement

C= A + B

in a high-level language program is a command to the computer to add the current values of the two variables called A and B, and to assign the sum to a third variable, C.

When the program containing this statement is compiled, the three variables, A,B,C are assigned to distinct location in the memory.

Hence the above high-level language statement requires the action

$C \leftarrow [A] + [B]$

to take place in the computer. Here [A] and [B] represents contents of A and B respectively.

To carry out this action, the contents of memory locations A and B are fetched from the memory and transferred into the processor where their sum is computed. This result is then sent back to the memory and stored in location C.

Performing a basic instruction is represented in many ways:

They are

- 3-address instruction
- 2 -address instruction
- 1-address instruction
- 0-address instruction

Let us first assume that this action is to be accomplished by a single machine instruction. Furthermore, assume that this instruction contains the memory addresses of the three operands - A, B, and C. This three-address instruction can be represented symbolically as

Add A,B,C

Operands A and B are called the source operands, C is called the destination operand, and Add is the operation to be performed on the operands. A general instruction of this type has the format

Operation Source1,Source2,Destination

- If k bits are needed to specify the memory address of each operand, the encoded form of the above instruction must contain 3k bits for addressing purposes in addition to the bits needed to denote the Add operation.
- For a modern processor with a 32-bit address space, a 3-address instruction is too large to fit in one word for a reasonable word length. Thus, a format that allows multiple words to be used for a single instruction would be needed to represent an instruction of this type.
- An alternative approach is to use a sequence of simpler instructions to perform the same task, with each instruction having only one or two operands. Suppose that two-address instructions of the form are available.

Operation Source,Destination

An Add instruction of this type is

Add A,B

which performs the operation B ← [A] + [B].

- When the sum is calculated, the result is sent to the memory and stored in location B, replacing the original contents of this location. This means that operand B is both a source and a destination.
- A single two-address instruction cannot be used to solve our original problem, which is to add the contents of locations A and B, without destroying either of them, and to place the sum in location C.

The problem can be solved by using another two address instruction that copies the contents of

one memory location into another. Such an instruction is

Move B,C

which performs the operation C ← [B], leaving the contents of location B unchanged. The word "Move" is a misnomer here; it should be "Copy."

However, this instruction name is deeply entrenched in computer nomenclature. The operation C ← [A] + [B] can now be performed by the two-instruction sequence

Move B,C

Add A,C

- In all the instructions given above, the source operands are specified first, followed by the destination. This order is used in the assembly language expressions for machine instructions in many computers.
- But there are also many computers in which the order of the source and destination operands is reversed. It is unfortunate that no single convention has been adopted by all manufacturers.
- In fact, even for a particular computer, its assembly language may use a different order for

different instructions. We have defined three- and two-address instructions. But, even two-address instructions will not normally fit into one word for usual word lengths and address sizes.

- Another possibility is to have machine instructions that specify only one memory operand.
- When a second operand is needed, as in the case of an Add instruction, it is understood implicitly to be in a unique location. A processor register, usually called the accumulator, may be used for this purpose. Thus, the one-address instruction

  Add A

  means the following: Add the contents of memory location A to the contents of the accumulator register and place the sum back into the accumulator. Let us also introduce the one-address instructions

  Load A

  and

  Store A

- The Load instruction copies the contents of memory location A into the accumulator, and the Store instruction copies the contents of the accumulator into memory location A. Using only one-address instructions, the operation C ([A] + [B] can be performed by executing the sequence of instructions

  Load A
  Add B
  Store C

- Note that the operand specified in the instruction may be a source or a destination, depending on the instruction.
- In the Load instruction, address A specifies the source operand, and the destination location, the accumulator, is implied.
- On the other hand, C denotes the destination location in the Store instruction, whereas the source, the accumulator, is implied.
- Some early computers were designed around a single accumulator structure. Most modern computers have a number of general-purpose processor registers - typically 8 to 32, and even considerably more in some cases.
- Access to data in these registers is much faster than to data stored in memory locations because the registers are inside the processor. Because the number of registers is relatively small, only a few bits are needed to specify which register takes part in an operation. For example, for 32 registers, only 5 bits are needed.
- This is much less than the number of bits needed to give the address of a location in the memory. Because the use of registers allows faster processing and results in shorter instructions, registers are used to store data temporarily in the processor during processing.

Let Ri represent a general-purpose register. The instructions

Load A,Ri

Store Ri,A

and

Add A,Ri

are generalizations of the Load, Store, and Add instructions for the single-accumulator case, in which register Ri performs the function of the accumulator.

- Even in these cases, when only one memory address is directly specified in an instruction, the instruction may not fit into one word.
- When a processor has several general-purpose registers, many instructions involve only operands that are in the registers. In fact, in many modem processors, computations can be performed directly only on data held in processor registers. Instructions such as

  Add Ri,Rj

  or

  Add Ri,Rj,Rk

  are of this type.
- In both of these instructions, the source operands are the contents of registers Ri and Rj. In the first instruction, Rj also serves as the destination register, whereas in the second instruction, a third register, Rk, is used as the destination. Such instructions, where only register names are contained in the instruction, will normally fit into one word.
- It is often necessary to transfer data between different locations. This is achieved with the instruction

  Move Source,Destination

  which places a copy of the contents of Source into Destination.
- When data are moved to or from a processor register, the Move instruction can be used rather than the Load or Store instructions because the order of the source and destination operands determines which operation is intended. Thus,

  Move A,Ri

  is the same as

  Load A,Ri

  and

  Move Ri,A

  is the same as

  Store Ri ,A

- In processors where arithmetic operations are allowed only on operands that are in processor registers, the C = A + B task can be performed by the instruction sequence

  Move A,Ri
  Move B,Rj
  Add Ri ,Rj
  Move Rj ,C

In processors where one operand may be in the memory but the other must be in a register, an instruction sequence for the required task would be

Move A,Ri
Add B,Ri
Move Ri,C

- The speed with which a given task is carried out depends on the time it takes to transfer

instructions from memory into the processor and to access the operands referenced by these instructions.

- Transfers that involve the memory are much slower than transfers within the processor. Hence, a substantial increase in speed is achieved when several operations are performed in succession on data in processor registers without the need to copy data to or from the memory.
- When machine language programs are generated by compilers from high-level languages, it is important to minimize the frequency with which data is moved back and forth between the memory and processor registers.

We used the task C ← [A] + [B] as an example instruction format. The diagram shows a possible program segment for this task as it appears in the memory of a computer. We have assumed that the computer allows one memory operand per instruction and has a number of processor registers. We assume that the word length is 32 bits and the memory is byte addressable. The three instructions of the program are in successive word locations, starting at location *i*. Since each instruction is 4 bytes long, the second and third instructions start at addresses *i* + 4 and *i* + 8.
For simplicity, we also assume that a full memory address can be directly specified in
a single-word instruction, although this is not usually possible for address space sizes
and word lengths of current processors.

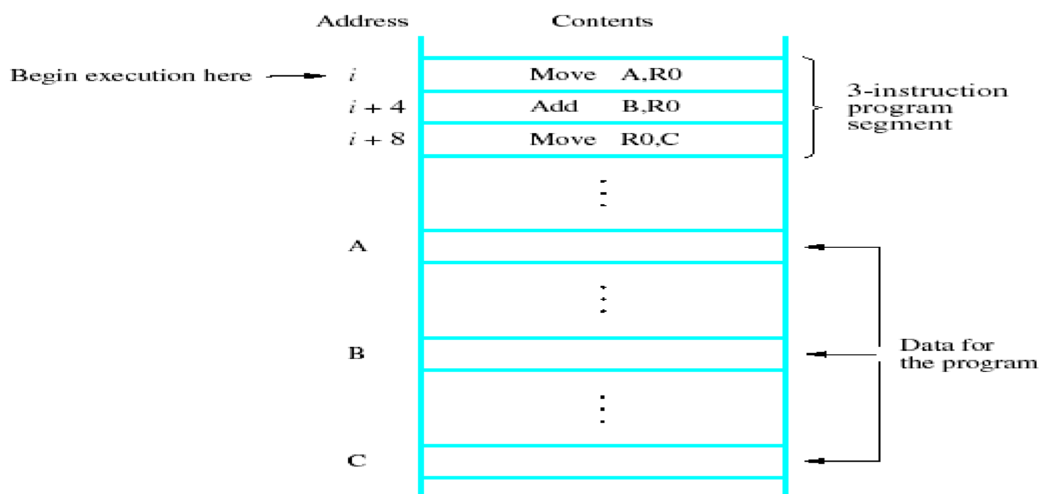


Fig: A program for C ← [A] + [B]

Execution steps of an above program:

- The processor contains a register called the *program counter* (PC), which holds the address of the instruction to be executed next.
- To begin executing a program, the address of its first instruction (*i* in our example) must be placed into the PC.
- Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called *straight-line sequencing*.
- During the execution of each instruction, the PC is incremented by 4 to point to the next instruction.
- Thus, after the Move instruction at location *i* + 8 is executed, the PC contains the value *i* + 12, which is the address of the first instruction of the next program segment.

- Executing a given instruction is a two-phase procedure.
- ✓ In the first phase, called ***instruction fetch***, the instruction is fetched from the memory location whose address is in the PC. This instruction is placed in the *instruction register* (IR) in the processor.
- ✓ At the start of the second phase, called ***instruction execute***, the instruction in IR is examined to determine which operation is to be performed.
- The specified operation is then performed by the processor. This often involves fetching operands from the memory or from processor registers, performing an arithmetic or logic operation, and storing the result in the destination location.
- At some point during this two-phase procedure, the contents of the PC are advanced to point to the next instruction. When the execute phase of an instruction is completed, the PC contains the address of the next instruction, and a new instruction fetch phase can begin.
- In most processors, the execute phase itself is divided into a small number of distinct phases corresponding to fetching operands, performing the operation, and storing the result.

(OR)

(b)	Define Addressing mode and Illustrate the basic addressing modes with an example for each.	13	CO1	RE

ADDRESSING MODES

The different ways in which the location of an operand is specified in an instruction are referred to as *addressing modes*.

1. IMPLEMENTATION OF VARIABLES AND CONSTANTS

Variables and constants are the simplest data types and are found in almost every Computer program. In assembly language, a variable is represented by allocating a register or a memory location to hold its value. Thus, the value can be changed as needed using appropriate instructions.

We accessed an operand by specifying the name of the register or the address of the memory location where the operand is located.

Register mode

The operand is the contents of a processor register; the name (address) of the register is given in the instruction. It is used to access the variables in the program.

Absolute mode

The operand is in a memory location; the address of this location is given explicitly in the instruction. It is also called as *Direct mode*. It also used to access the variables in the program.

Example instruction for register and absolute mode

```
Move LOC, R2
```

uses the register and absolute modes. The processor registers are used as temporary storage locations where the data in a register are accessed using the Register mode. The Absolute mode can represent global variables in a program. A declaration such as

```
Integer A, B;
```

In a high-level language program will cause the compiler to allocate a memory location to each of the variables A and B. Absolute mode can be used to access the variables in the program

Table 2.1 Generic addressing modes

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (Direct)	LOC	EA = LOC
Indirect	(R*i*) (LOC)	EA = [R*i*] EA = [LOC]
Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Base with index and offset	X(R*i*,R*j*)	EA = [R*i*] + [R*j*] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R*i*)+	EA = [R*i*]; Increment R*i*
Autodecrement	−(R*i*)	Decrement R*i*; EA = [R*i*]

EA = effective address
Value = a signed number

Immediate mode

Address and data constants can be represented in assembly language using the Immediate mode. The operand is given explicitly in the instruction.

For example, the instruction

Move 200*immediate,* R0

places the value 200 in register R0. Clearly, the Immediate mode is only used to specify the value of a source operand. Using a subscript to denote the Immediate mode is not appropriate in assembly languages. A common convention is to use the sharp sign (#) in front of the value to indicate that this value is to be used as an immediate operand.

Hence, we write the instruction above in the form

Move #200, R0

Constant values are used frequently in high-level language programs. For example, the statement

A = B + 6

contains the constant 6. Assuming that A and B have been declared earlier as variables and may be accessed using the Absolute mode, this statement may be compiled as follows:

```
Move B, R1
Add #6, R1
Move R1, A
```

Constants are also used in assembly language to increment a counter, test for some bit pattern, and so on.

INDIRECTION AND POINTERS

In the addressing modes that follow, the instruction does not give the operand or its address explicitly. Instead, it provides information from which the memory address of the operand can be determined. We refer to this address as the effective address (EA) of the operand.

Indirect mode

The effective address of the operand is the contents of a register or memory location whose address appears in the instruction. We denote indirection by placing the name of the register or the memory address given in the instruction in parentheses.

To execute the Add instruction the processor uses the value B, which is in register R1, as the effective address of the operand. It requests a read operation from the memory to read the contents of location B. The value read is the desired operand, which the processor adds to the contents of register R0. Indirect addressing through a memory location is also possible. In this

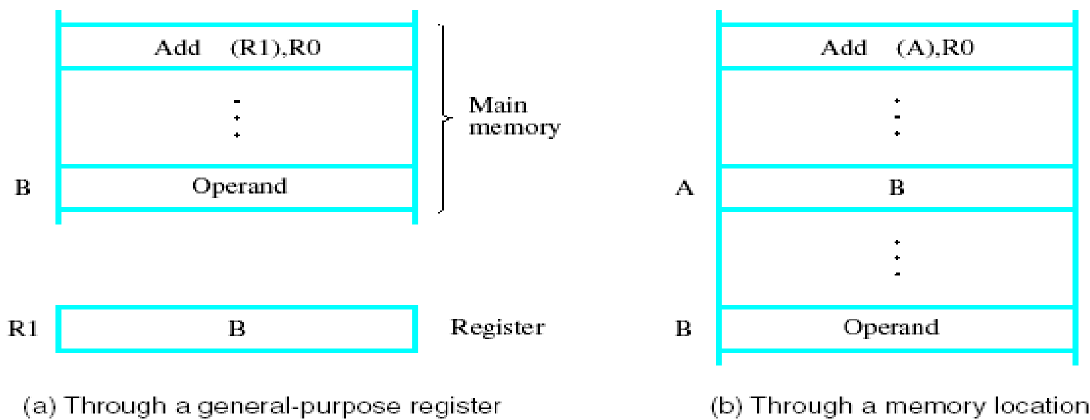


Fig: Indirect addressing.

Address	Contents		
	Move	N,R1	Initialization
	Move	#NUM1,R2	
	Clear	R0	
LOOP	Add	(R2),R0	
	Add	#4,R2	
	Decrement	R1	
	Branch>0	LOOP	
	Move	R0,SUM	

Fig: Use of indirect addressing in the program

The first time through the loop, the instruction

Add (R2), R0

fetches the operand at location NUM1 and adds it to R0. The second Add instruction adds 4 to the contents of the pointer R2, so that it will contain the address value NUM2 when the above instruction is executed in the second pass through the loop.

Consider the C-language statement

A= *B;

where B is a pointer variable. This statement may be compiled into

Move B, R1

Move (R1), A

Using indirect addressing through memory, the same action can be achieved with

Move (B), A

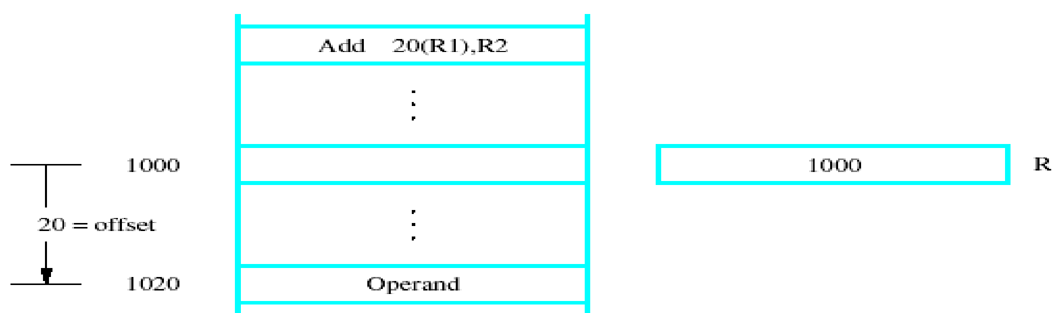
Despite its apparent simplicity, indirect addressing through memory has proven to be of limited usefulness as an addressing mode, and it is seldom found in modern computers. An instruction that involves accessing the memory twice to get an operand is not well suited to pipelined execution. Indirect addressing through registers is used extensively. The program shows the flexibility it provides. Also, when absolute addressing is not available, indirect addressing through registers makes it possible to access global variables by first loading the operand's address in a register.

INDEXING AND ARRAYS

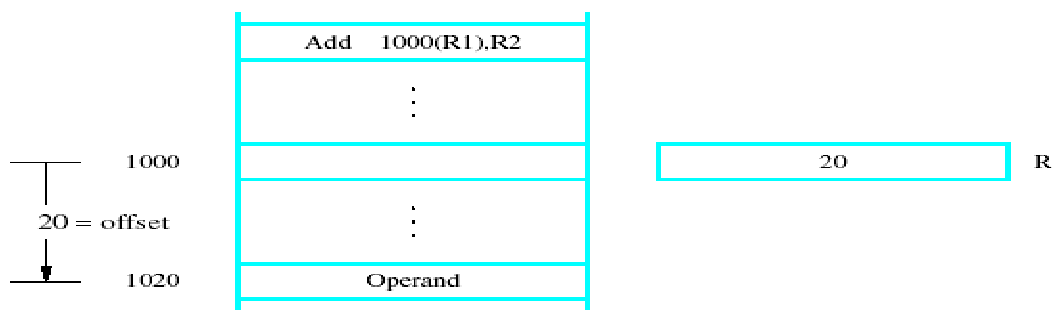
It is useful in dealing with lists and arrays.

Index mode

The effective address of the operand is generated by adding a constant value to the contents of a register. The register used may be either a special register provided for this purpose, or, more commonly; it may be any one of a set of general-purpose registers in the processor. In either case, it is referred to as an *index register*. We indicate the Index mode symbolically as X (R*i)* where X denotes the constant value contained in the instruction and R*i* is the name of the register involved. The effective address of the operand is given by EA = X + [R*i*].The contents of the index register are not changed in the process of generating the effective address. In an assembly language program, the constant X may be given either as an explicit number or as a symbolic name representing a numerical value. When the instruction is translated into machine code, the constant X is given as a part of the instruction and is usually represented by fewer bits than the word length of the computer. Since X is a signed integer, it must be sign-extended to the register length before being added to the contents of the register. The index register, R1, contains the address of a memory location, and the value X defines an *offset* (also called a *displacement*) from this address to the location where the operand is found. An alternative use: Constant X corresponds to a memory address, and the contents of the index register define the offset to the operand. In either case, the effective address is the sum of two values; one is given explicitly in the instruction, and the other is stored in a register.



(a) Offset is given as a constant



(b) Offset is in the index register

Indexed addressing

To see the usefulness of indexed addressing, consider a simple example involving a list of test scores for students taking a given course. Assume that the list of scores, beginning at location LIST. A four-word memory block comprises a record that stores the relevant information for each student. Each record consists of the student's identification number (ID), followed by the scores the student earned on three tests. There are *n* students in the class, and the value *n* is stored in location N immediately in front of the list. The addresses given in the figure for the student IDs and test scores assume that the memory is byte addressable and that the word length is 32 bits.

We should note that the list in n represents a two-dimensional array having *n* rows and four columns. Each row contains the entries for one student, and the columns give the IDs and test scores.

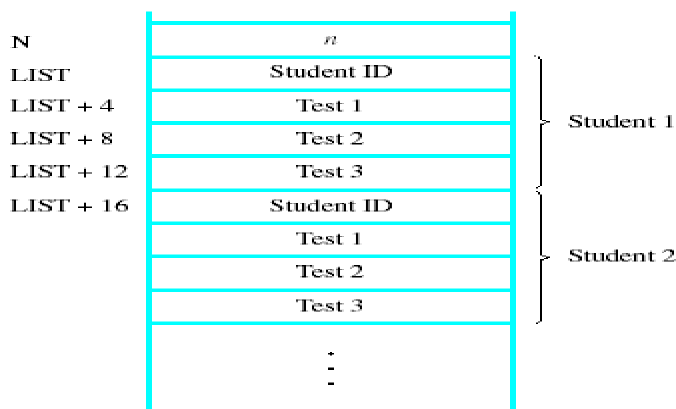


Fig: A list of students' marks

Suppose that we wish to compute the sum of all scores obtained on each of the tests and store these three sums in memory locations SUM1, SUM2, and SUM3. In the body of the loop, the program uses the Index addressing mode. *T*o access each of the three scores in a student's record, Register R0 is used as the index register. Before the loop is entered, R0 is set to point to the ID location of the first student record; thus, it contains the address LIST. On the first pass through the loop, test scores of the first student are added to the running sums held in registers R1, R2, and R3, which are initially cleared to 0. These scores are accessed using the Index addressing modes 4(R0), 8(R0), and 12(R0). The index register R0 is then incremented by 16 to point to the ID location of the second student. Register R4, initialized to contain the value *n*, is decremented by 1 at the end of each pass through the loop. When the contents of R4 reach 0, all student records have been accessed, and the loop terminates. Until then, the conditional branch instruction transfers control back to the start of the loop to process the next record. The last three instructions transfer the accumulated sums from registers R1, R2, and R3, into memory locations SUM1, SUM2, and SUM3, respectively. It should be emphasized that the contents of the index register, R0, are not changed when it is used in the Index addressing mode to access the scores. The contents of R0 are changed only by the last Add instruction in the loop, to move from one student record to the next.
In general, the Index mode facilitates access to an operand whose location is defined relative to a reference point within the data structure in which the operand appears. In the example just given, the ID locations of successive student records are the reference points, and the test scores are the operands accessed by the Index addressing mode.

We have introduced the most basic form of indexed addressing. Several variations of this basic form provide for very efficient access to memory operands in practical programming situations.

For example, a second register may be used to contain the offset X, in which case we can write the Index mode as

(Ri,R j)

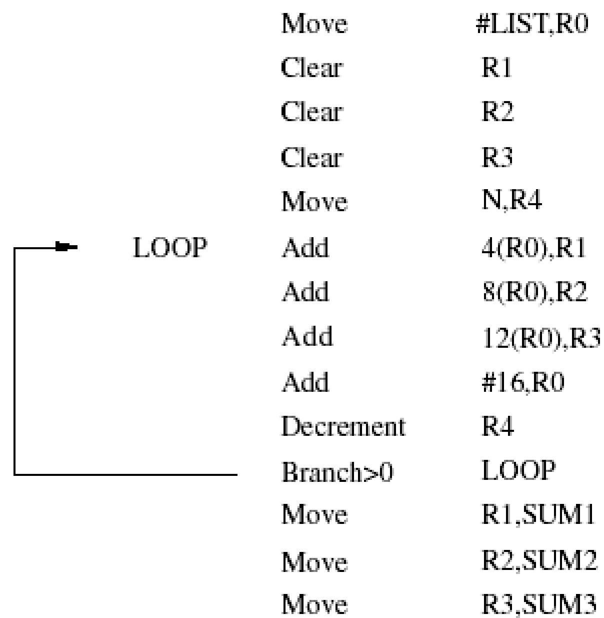


Fig: Indexed addressing used in accessing test scores in the list

The effective address is the sum of the contents of registers Ri and Rj . The second register is usually called the *base* register. This form of indexed addressing provides more flexibility in accessing operands, because both components of the effective address can be changed. As an example of where this flexibility may be useful, consider again the student record data structure shown in Figure. In the above program, we used different index values in the three Add instructions at the beginning of the loop to access different test scores. Suppose each record contains a large number of items, many more than the three test scores of that example. In this case, we would need the ability to replace the three Add instructions with one instruction inside a second (nested) loop. Just as the successive starting locations of the records (the reference points) are maintained in the pointer register R0, offsets to the individual items relative to the contents of R0 could be maintained in another register. The contents of that register would be incremented in successive passes through the inner loop.

Yet another version of the Index mode uses two registers plus a constant, which can be denoted as

X(Ri,R j)

In this case, the effective address is the sum of the constant X and the contents of registers Ri and Rj . This added flexibility is useful in accessing multiple components inside each item in a record, where the beginning of an item is specified by the (Ri,R j) part of the addressing mode. In other words, this mode implements a three-dimensional array.

RELATIVE ADDRESSING

We have defined the Index mode using general-purpose processor registers. A useful version of this mode is obtained if the program counter, PC, is used instead of a general purpose register. Then, X(PC) can be used to address a memory location that is X bytes away from the location presently pointed to by the program counter. Since the addressed location is identified

"relative" to the program counter, which always identifies the current execution point in a program, the name Relative mode is associated with this type of addressing.

Relative mode

The effective address is determined by the Index mode using the program counter in place of the general-purpose register R*i*. This mode can be used to access data operands. But, its most common use is to specify the target address in branch instructions. An instruction such as

Branch>0 LOOP

causes program execution to go to the branch target location identified by the name LOOP if the branch condition is satisfied. This location can be computed by specifying it as an offset from the current value of the program counter. Since the branch target may be either before or after the branch instruction, the offset is given as a signed number.

ADDITIONAL MODES

Auto increment mode

The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list. We denote the Autoincrement mode by putting the specified register in parentheses, to show that the contents of the register are used as the effective address, followed by a plus sign to indicate that these contents are to be incremented after the operand is accessed. Thus, the Autoincrement mode is written as

(R*i*)+

Implicitly, the increment amount is 1 when the mode is given in this form. But in a byte addressable memory, this mode would only be useful in accessing successive bytes of some list.

Autoincrement mode as (R*i*)+.

If the Autoincrement mode is available, it can be used in the first Add instruction and the second Add instruction can be eliminated. The modified program is shown in below Fig.

As a companion for the Autoincrement mode, another useful mode accesses the items of a list in the reverse order.

Autodecrement mode

The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand. We denote the Autodecrement mode by putting the specified register in parentheses, preceded by a minus sign to indicate that the contents of the register are to be decremented before being used as the effective address. Thus, we write -(R*i*)

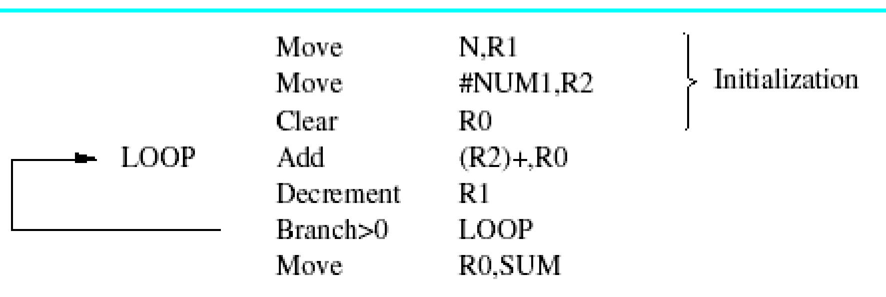


Fig: The Autoincrement addressing mode used in the program

		and incremented after it is used in the Autoincrement mode. The actions performed by the Autoincrement and Autodecrement addressing modes can obviously be achieved by using two instructions, one to access the operand and the other to increment or decrement the register that contains the operand address. Combining the two operations in one instruction reduces the number of instructions needed to perform the desired task.			

8	(a)	Discuss briefly about floating point addition and subtraction algorithms.	14	CO 2	U

In this section, we outline the general procedures for addition, subtraction, multiplication, and division of floating-point numbers. The rules given below apply to the single-precision IEEE standard format. These rules specify only the major steps needed to perform the four operations; for example, the possibility that overflow or underflow might occur is not discussed.

Furthermore, intermediate results for both mantissas and exponents might require more than 24 and 8 bits, respectively. These and other aspects of the operations must be carefully considered in designing an arithmetic unit that meets the standard. Although we do not provide full details in specifying the rules, we consider some aspects of implementation, including

rounding, in later sections. When adding or subtracting floating-point numbers, their mantissas must be shifted with respect to each other if their exponents differ. Consider a decimal example

in which we wish to add 2.9400 × 102 to 4.3100 × 104. We rewrite 2.9400 × 102 as 0.0294 × 104 and then perform addition of the mantissas to get 4.3394 × 104. The rule for addition and

subtraction can be stated as follows:

Add/Subtract Rule

1. Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.
2. Set the exponent of the result equal to the larger exponent.
3. Perform addition/subtraction on the mantissas and determine the sign of the result.
4. Normalize the resulting value, if necessary.

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

Multiply Rule

1. Add the exponents and subtract 127 to maintain the excess-127 representation.
2. Multiply the mantissas and determine the sign of the result.
3. Normalize the resulting value, if necessary.

Divide Rule

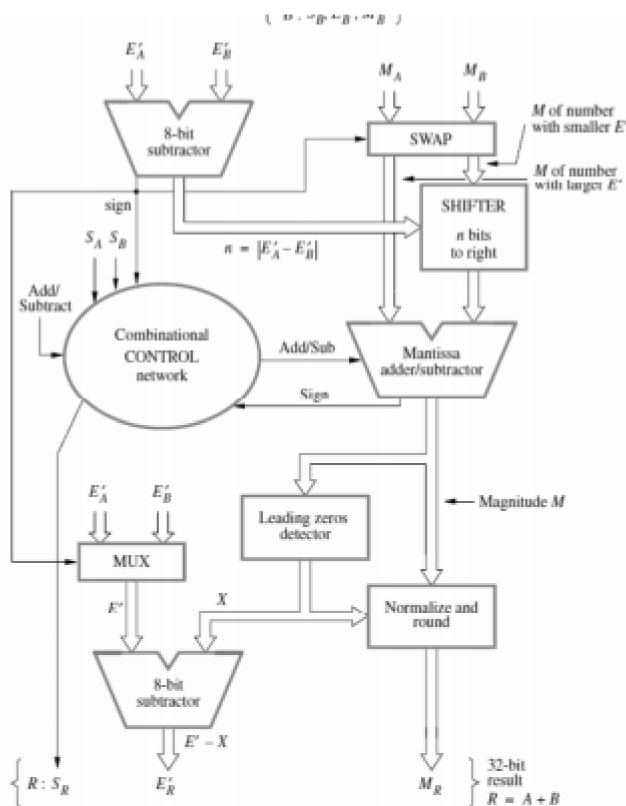
1. Subtract the exponents and add 127 to maintain the excess-127 representation.
2. Divide the mantissas and determine the sign of the result.

3. Normalize the resulting value, if necessary.

DECIMAL ARITHMETIC UNIT

The hardware implementation of floating-point operations involves a considerable amount of logic circuitry. These operations can also be implemented by software routines. In either case, the computer must be able to convert input and output from and to the user's decimal representation of numbers. In many general-purpose processors, floating-point operations are available at the machine-instruction level, implemented in hardware.

Implementing Floating-Point Operations An example of the implementation of floating-point operations is shown in Figure. This is a block diagram of a hardware implementation for the addition and subtraction of 32-bit floating-point operands that have the format shown in Figure a. Following the Add/Subtract rule, we see that the first step is to compare exponents to determine how far to shift the mantissa of the number with the smaller exponent. The shift-count value, n, is determined by the 8-bit subtractor circuit in the upper left corner of the figure. The magnitude of the difference EA− EB, or n, is sent to the SHIFTER unit. If n is larger than the number of significant bits of the operands, then the answer is essentially the larger operand (except for guard and sticky-bit considerations in rounding), and shortcuts can be taken in deriving



The sign of the difference that results from comparing exponents determines which mantissa is to be shifted. Therefore, in step 1, the sign is sent to the SWAP network in the upper right corner of Figure. If the sign is 0, then EA≥ EB and the mantissas MA and MB are sent straight through the SWAP network. This results in MB being sent to the SHIFTER, to be shifted n positions to the right. The other mantissa, MA, is sent directly to the mantissa adder/subtractor. If the sign is 1, then EA < EB and the mantissas are swapped before they are

sent to the SHIFTER. Step 2 is performed by the two-way multiplexer, MUX, near the bottom left corner of the figure. The exponent of the result, E, is tentatively determined as EA if EA≥ EB, or E B if E A
< EB, based on the sign of the difference resulting from comparing exponents in step 1.
Step 3 involves the major component, the mantissa adder/subtractor in the middle of the figure. The CONTROL logic determines whether the mantissas are to be added or subtracted. This is decided by the signs of the operands (SA and SB) and the operation (Add or Subtract) that is to be performed on the operands. The CONTROL logic also determines the sign of the result, SR. For example, if A is negative (SA = 1), B is positive (SB = 0), and the operation is A − B, then the mantissas are added and the sign of the result is negative (SR = 1). On the other hand , if A and B are both positive and the operation is A − B, then the mantissas are subtracted. The sign of the result, SR, now depends on the mantissa subtraction operation. For instance, if EA > EB, then M = MA − (shifted MB) and the resulting number is positive. But if EB > EA, then M = MB − (shifted MA) and the result is negative. This example shows that the sign from the exponent comparison is also required as an input to the CONTROL network. When EA= EB and the mantissas are subtracted, the sign of the mantissa adder/subtractor output determines the sign of the result. The reader should now be able to construct the complete truth table for the CONTROL network.
Step 4 of the Add/Subtract rule consists of normalizing the result of step 3 by shifting M to the right or to the left, as appropriate. The number of leading zeros in M determines the number of bit shifts, X , to be applied to M. The normalized value is rounded to generate the 24-bit mantissa, MR, of the result. The value X is also subtracted from the tentative result exponent E to generate the true result exponent, ER. Note that only a single right shift might be needed to normalize the result. This would be the case if two mantissas of the form 1.xx . . . were added. The vector M would then have the form 1x.xx . . . . We have not given any details on the guard bits that must be carried along with intermediate mantissa values. In the IEEE standard, only a few bits are needed, to generate the 24-bit normalized mantissa of the result.
Let us consider the actual hardware that is needed to implement the blocks in Figure.
The two 8-bit subtractors and the mantissa adder/subtractor can be implemented by combinational logic, as discussed earlier in this chapter. Because their outputs must be in sign and-magnitude form, we must modify some of our earlier discussions. A combination of 1's complement arithmetic and sign-and-magnitude representation is often used. Considerable flexibility is allowed in implementing the SHIFTER and the output normalization operation. The operations can be implemented with shift registers. However, they can also be built as combinational logic units for high-performance

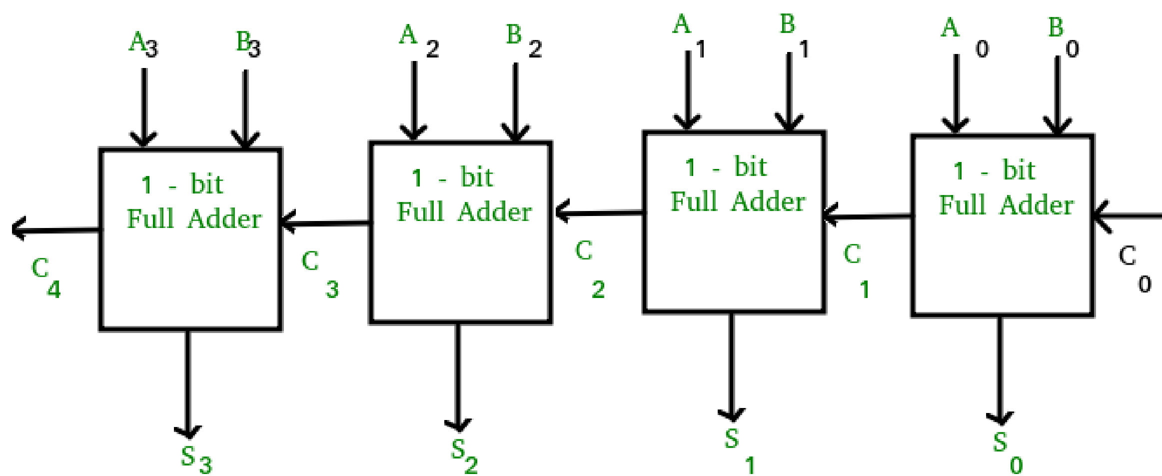
(OR)

(b)	Summarize briefly about carry-look ahead adder and also give the expression for full adder circuit to show carry generation and propagation.	14	CO 1	

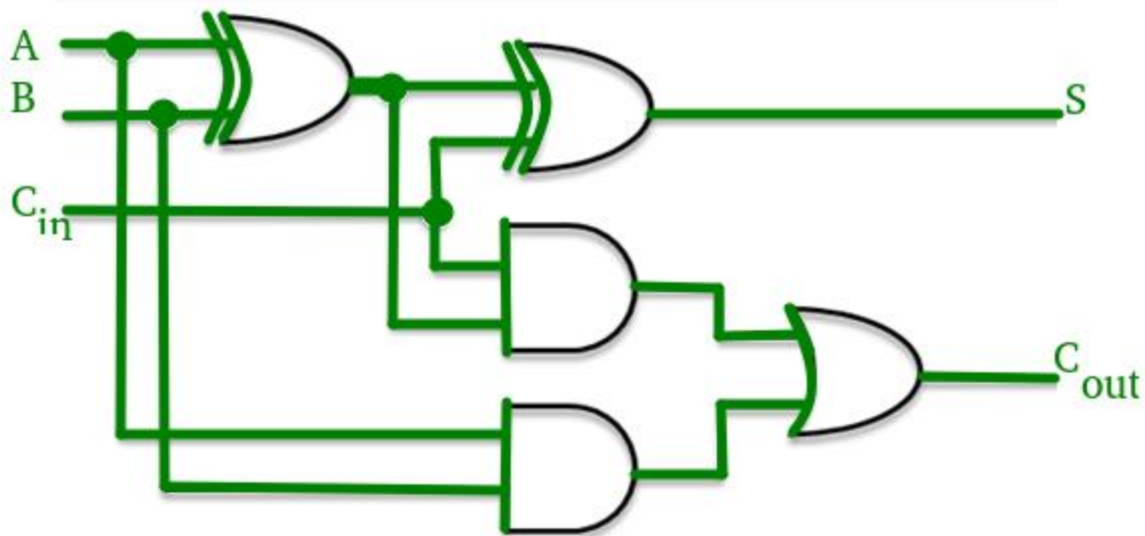
he adder produce carry propagation delay while performing other arithmetic operations like multiplication and divisions as it uses several additions or subtraction steps. This is a major problem for the adder and hence improving the speed of addition will improve the speed of all other arithmetic operations. Hence reducing the carry propagation delay of adders is of great importance. There are different logic design approaches that have been employed to overcome the carry propagation problem. One widely used approach is to employ a carry look-ahead

which solves this problem by calculating the carry signals in advance, based on the input signals. This type of adder circuit is called a carry look-ahead adder.

Here a carry signal will be generated in two cases:
Input bits A and B are 1
When one of the two bits is 1 and the carry-in is 1.
In ripple carry adders, for each adder block, the two bits that are to be added are available instantly. However, each adder block waits for the carry to arrive from its previous block. So, it is not possible to generate the sum and carry of any block until the input carry is known. The block waits for the block to produce its carry. So there will be a considerable time delay which is carry propagation delay.



Consider the above 4-bit ripple carry adder. The sum is produced by the corresponding full adder as soon as the input signals are applied to it. But the carry input is not available on its final steady-state value until carry is available at its steady-state value. Similarly depends on and on . Therefore, though the carry must propagate to all the stages in order that output and carry settle their final steady-state value.
The propagation time is equal to the propagation delay of each adder block, multiplied by the number of adder blocks in the circuit. For example, if each full adder stage has a propagation delay of 20 nanoseconds, then will reach its final correct value after 60 (20 × 3) nanoseconds. The situation gets worse, if we extend the number of stages for adding more number of bits.



A	B	C	C +1	Condition
0	0	0	0	No Carry Generate
0	0	1	0	
0	1	0	0	
0	1	1	1	No Carry Propogate
1	0	0	0	
1	0	1	1	
1	1	0	1	Carry Generate
1	1	1	1	

Consider the full adder circuit shown above with corresponding truth table. We define two variables as 'carry generate' and 'carry propagate' then,

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$

The sum output and carry output can be expressed in terms of carry generate and carry propagate as

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

		where produces the carry when both , are 1 regardless of the input carry. is associated with the propagation of carry from to . The carry output Boolean function of each stage in a 4 stage carry look-ahead adder can be expressed as $C_1 = G_0 + P_0 C_{in}$ $C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_{in}$ $C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{in}$ $C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_{in}$		