DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

III SEMESTER

COMPUTER ORGANIZATION AND ARCHITECTURE

SYLLABUS

UNIT – I

BASIC STRUCTURES OF COMPUTER: Functional Units, Multiprocessors and Multi computers, Memory Locations and Addresses, Memory operations, Instructions and Instruction Sequencing, Addressing modes, Assembly Language, Basic Input/output operations, Stacks and Queues, Subroutines, Shift and rotate Instructions, Byte-Sorting program.

UNIT – II

The IA-32 Pentium Example: Registers and Addressing, IA-32 Instructions, IA-32 Assembly Language, Program Flow Control, Logic and Shift/Rotate Instructions, I/O Operations, Subroutines, Other Instructions, Program Examples.

UNIT – III

INPUT/OUTPUT ORGANIZATION: Accessing I/O Devices, Interrupts, Interrupt Hardware, Enabling and Disabling Interrupts, Handling Multiple Devices, Controlling Device Requests, Exceptions, Use Of Interrupts in Operating Systems, Pentium Interrupt Structure, Direct Memory Access, Busses, Interface Circuits, Standard I/O Interfaces.

UNIT – IV

THE MEMORY SYSTEM: Some Basic Concepts, Semiconductor RAM Memories, Read-Only Memories, Speed, Size, and Cost, Cache Memories, Performance Considerations, Virtual memories, Memory Management requirements, Secondary Storage.

UNIT – V

BASIC PROCESSING UNIT : Some Fundamental Concepts, Execution Of a Complete Instruction, Multiple-Bus Organization, Hardwired Control, Microprogrammed Control, PIPELINING: Basic Concepts, Data Hazards, Instruction Hazards, Influence On Instructions Sets, Datapath and Control Considerations, Superscalar Operations, Performance Considerations

TEXT BOOKS

1. Carl Hamacher, Zvonko Vranesic and Safwat Zaky, “Computer Organization”, fifth edition, TataMcGraw Hill Education, 2011.

REFERENCES

1. John P. Hayes, “Computer Architecture and Organization”, Third edition, Tata McGraw Hill, 2013
2. William Stallings, “Computer organization and Architecture – Designing for performance”, 9th edition, Pearson education, 2012
3. Computer System Architecture – M.Moris Mano, IIIrd Edition, PHI / Pearson, 2006.

UNIT I

BASIC STRUCTURES OF COMPUTER: Functional Units, Multiprocessors and Multi computers, Memory Locations and Addresses, Memory operations, Instructions and Instruction Sequencing, Addressing modes, Assembly Language, Basic Input/output operations, Stacks and Queues, Subroutines, Shift and rotate Instructions, Byte-Sorting program.

2 Marks

1. What is Computer Organization?

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

2. What is meant by Computer Architecture?

- It is concerned with the structure and behaviour of the computer.
- It includes the information formats, the instruction set and techniques for addressing memory.

3. What is difference between Computer Architecture and Computer Organization? (Apr 12)

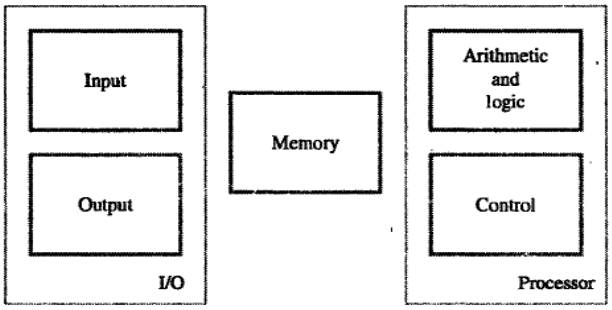
S.No	Computer Architecture	Computer Organization
1.	It refers to the attributes that have a direct impact on the logical execution of the program.	It refers to the operational units and their interconnections that realize the architectural specifications
2.	Architectural attributes includes the Instruction set, data types, no of bits used to represent the data, I/O mechanisms.	Organizational attributes include those h/w details such as control signals, interfaces b/w the computer memory & I/O peripherals

4. List the various functional units of a computer

A computer consists of 5 main parts.

- Input
- Memory
- Arithmetic and logic
- Output
- Control Units

5. Draw the structure of functional unit.



6. Write about Input unit

Computers accept coded information through input units, which read the data. Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted over a cable to either the memory or the processor.

7. Give some example of input devices

- Joysticks
- Trackballs
- Mouses
- Microphones (Capture audio input and it is sampled & it is converted into digital codes for storage and processing).

8. What is the use of memory unit and write its types

It stores the programs and data.

There are 2 types of storage classes

- Primary
- Secondary

9. What is primary storage?

It is a fast memory that operates at electronic speeds. Programs must be stored in the memory while they are being executed. The memory contains large no of semiconductor storage cells. Each cell carries 1 bit of information. The Cells are processed in a group of fixed size called Words.

10. Write the types of memory

There are 3 types of memory.They are

- RAM(Random Access Memory)
- Cache memory
- Main Memory

11. Write about RAM

Memory in which any location can be reached in short and fixed amount of time after specifying its address is called RAM. Time required to access 1 word is called Memory Access Time

12. Write about Cache Memory

The small,fast,RAM units are called Cache. They are tightly coupled with processor to achieve high performance.

13. What is the function of ALU?

Most of the computer operations (arithmetic & logic) are performed in ALU. The data required for the operation is brought by the processor and the operation is performed by the ALU.

14. What are basic operations of a computer memory?

The basic operations of the memory are READ and WRITE.
READ – read the data from input device to memory.
WRITE – writes data to the output device.

15. Write about Output Unit

Its function is to send the processed results to the outside world. Eg.Printer Printers are capable of printing 10000 lines per minute but its speed is comparatively slower than the processor

16. Write about control unit

The information received from the input unit is either stored in the computer's memory for later reference or immediately used by the arithmetic and logic circuitry to perform the desired operations. The processing steps are determined by a program stored in the memory. Finally the results are sent back to the outside world through the output unit. All of these actions are coordinated by the control unit.

17. What is multiprocessor?

Large computer systems may contain a number of processor units, in which case they are called multiprocessor. These systems either execute a number of different application tasks in parallel or they execute subtasks of a single large task in parallel.

18. What is meant by multi computer?

Multi computer is an interconnected group of complete computers to achieve high total computation power. The term generally refers to an architecture in which each processor has its own memory rather than multiple processors with a shared memory. Something similar to parallel computing

19. Write the difference between multi computer and multiprocessor

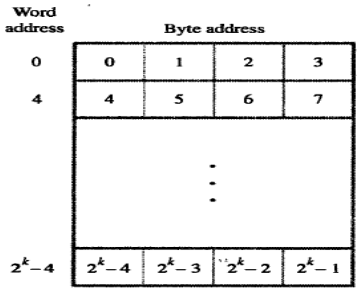
S.No	Multicomputer	Multiprocessors
1	A computer made up of several computers. similar to parallel computing	A multiprocessor system is simply a computer that has more than one CPU on its motherboard
2	Distributed computing deals with hardware and software systems containing more than one processing element, multiple programs, running under a loosely or tightly controlled regime.	Multiprocessing is the use of two or more central processing units (CPUs) within a single computer system
3	multicomputer have one physical address space per CPU	Multiprocessors have a single physical address space (memory) shared by all the CPUs
4	It can run faster	A multiprocessor would run slower, because it would be in ONE computer.
5	A multi-computer is multiple computers, each of which can have multiple processors. Used for true parallel processing	A multi-processor is a single system with multiple CPU's

20. What is byte addressability?

There are three basic information quantities to deal with: the bit, byte, and word .a byte is always 8 bits, but the word length typically ranges from 16 to 64 bits. It is impractical to assign n distinct addresses to individual bit locations in the memory. the most practical assignment is to have successive addresses refer to successive byte locations in the memory .this the assignment used in modern computers, called byte-addressable memory. Byte locations have addresses 0,1,2…….,thus if the word length of the machine is 32 bits ,successive words are located at addresses 0,4,8,….,with each word consisting of four bytes**.**

21. What is big –endian?

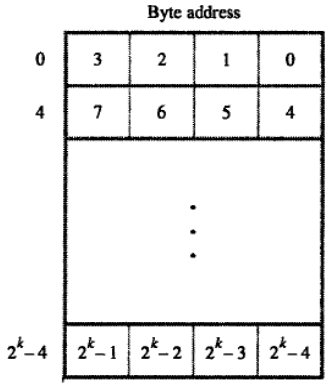
Big-endian and little-endian are terms that describe the order in which a sequence of bytes are stored in computer memory. The name big-endian is used when lower byte addresses are used for the more significant bytes (the leftmost bytes) of the word.



(a) Big-endian assignment

22. What is little-endian?

The name little0endian is used for the opposite ordering, where the lower byte addresses are used for the less significant bytes (the rightmost bytes)of the word. The words "more significant" and "less significant" are used in relation to the weights (powers of 2) assigned to bits when the word represents a number.



(b) Little-endian assignment

23. What is an instruction code?

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.

24. What is an operation code?

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 computer. The operation code must consist of at least n bits for a given 2^n (or less) distinct operations

25. What is an instruction format? (Apr 13)

The format of the instruction consists of two parts. The first part specifies the operation to be performed and the second specifies an 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.

Opcode

The operation itself is usually represented by a code called the opcode (for OPeration CODE)

Operands

All the other parts of an instruction are called operands.

Addresses

Some of the operands may be the actual addresses of data in memory.

Example

ADD AX,[102]

- The opcode is ADD
- There are two operands, AX and [102]

- This instruction contains one address - 102

Instructions are often categorized by the number of operands and addresses they contain. The above is a 2 operand 1 address instruction.

26. Write about types of computer instructions

A computer must have instructions capable of performing 4 types of operations:

- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

27. What is zero –address instruction?

An instruction that contains no address fields; operand sources and destination are both implicit. It may for example enable stack processing: a zero-address instruction implies that the absolute address of the operand is held in a special register that is automatically incremented (or decremented) to point to the location of the top of the stack.

Instruction format:

Opcode

Example: ADD

The above instruction consists of an operation code only. It has no address field. The operation has the effect of popping the two top numbers from the stack, adding the numbers and pushing the sum into the stack. Here all operands are performed within stack. To evaluate arithmetic expressions, they must be first converted into reverse polish notation. The operand at address X is pushed on to the top of the stack. Automatically the stack pointer is incremented.

Example:

X=(A + B) + (C + D)
PUSH A ; TOS <- A top of the stack
PUSH B ; T0S <- B
ADD ; TOS <- A + B
PUSH C ; TOS <- C
PUSH D ; TOS <- D
ADD ; TOS <- C + D
ADD ; TOS <- (C + D) + (A + B)
POP X ; M [X] <- TOS

28. What is one –address instruction?

One address instructions computers needs one address field. An implied accumulator (AC) register is used for all data manipulation. Here all the operations are carried out between the accumulator register and a memory operand.

Instruction format:

Opcode	Operand

ADD X

It denotes the operation,

AC <-- AC+ M[X]

Where

AC = Accumulator register

M[X] = Memory operand at address X.

Example:

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

LOAD A ; AC <- M [A]

ADD B ; AC <- AC + M[B]

STORE T ; M [T] <- AC

LOAD C ; AC <- M [C]

ADD D ; AC <- AC + M[D]

MUL T ; AC <- ACxM[T]

STORE X ; M[X] <- AC

29. What is two –address instruction?

In this format each address field specify either a processor register or a memory word i.e., they contain two address fields.

Instruction format:

Opcode	Source	Destination

ADD R1, R2

It denotes the operation,

R1 <-- R1 + R2

Here the destination register is the same as any one of the source registers.

Mov R1, R2

It denotes the operation,

R1 <-- R2

It transfers the content of register R2 to register R1.

ADD R1, X

It denotes the operation,

R1 <-- R1 + M[X]

Where R1 = Processor register

M[X] = Memory operand at address X.

Example:

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

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]

SUB R1,R2 ; R1 <- R1 - R2

MOV X,R1 ; M[X] <- R1

30. What is three –address instruction?

Three address instructions computer needs three register address fields. The register address field may be a processor register or a memory operand. Cyber 170 computer needs three address instructions .

Instruction format:

Opcode	Source1	Source2	Destinatin

ADD R1, R2, R3

The above instruction denotes the operations,

R1 <-- R2 + R3 (arithmetic addition)

Where

R2, R3 = Source registers

R1 = Destination register

Example:

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

```
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    */
```

Example of computer using this type of instructions Cyber 170.

31. Write about basic instruction types. (Apr 13)

S.No	Basic Instruction Types: Instruction Type	Syntax	Example	Description
1	Three Address	Operation Source1,Source2,Destination	Add A,B,C	Add values of variable A ,B & place the result into c.
2	Two Address	Operation Source,Destination	Add A,B	Add the values of A,B & place the result into B.
3	One Address	Operation Operand	Add B	Content of accumulator add with content of B.
4	Zero address	Operation	POP	Content of the top of the stack will be moved to accumulator

32. What is Addressing Mode?

It specifies a rule for interpreting or modifying the address field of the instruction (before the operand is actually referenced)

33. What is the need of Variety of Addressing Modes?

- To give programming flexibility to the user
- To use the bits in the address field of the instruction efficiently

34. What are the different types of addressing modes available?

The different types of addressing modes are:

1. Immediate addressing mode
2. Register addressing mode
3. Direct or absolute addressing mode
4. Indirect addressing mode
5. Indexed addressing mode
6. Base with index
7. Base with index and offset
8. Relative addressing mode
9. Auto increment
10. Auto decrement

35. What is implied addressing mode?

Address of the operands are specified implicitly in the definition of the instruction

- No need to specify address in the instruction
- EA = AC, or EA = Stack[SP]
- Examples from Basic Computer

CLA, CME, INP

36. What is Immediate Addressing Mode?

Instead of specifying the address of the operand, operand itself is specified

- No need to specify address in the instruction
- However, operand itself needs to be specified
- Sometimes, require more bits than the address
- Fast to acquire an operand

Operand = address field

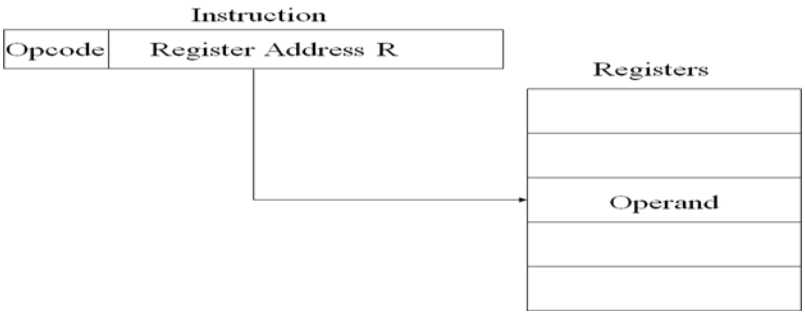
- e.g. ADD 5
  - Add 5 to contents of accumulator
  - 5 is operand
- No memory reference to fetch data
- Limited range

Instruction

37. What is Register Addressing Mode?

Address specified in the instruction is the register address

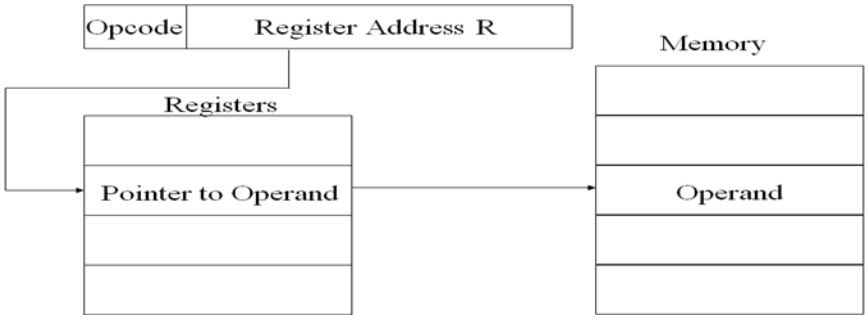
- Designated operand need to be in a register
- Shorter address than the memory address
- Saving address field in the instruction
- Faster to acquire an operand than the memory addressing
- EA = IR(R) (IR(R): Register field of IR)



38. What is Register Indirect Addressing Mode?

Instruction specifies a register which contains the memory address of the operand

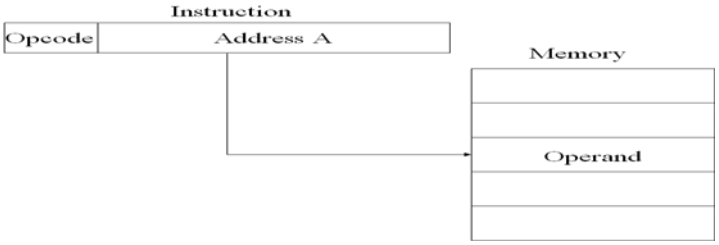
- Saving instruction bits since register address is shorter than the memory address
- Slower to acquire an operand than both the register addressing or memory addressing
- EA = [IR(R)] ([x]: Content of x)



39. What is Direct Addressing Mode?

Instruction specifies the memory address which can be used directly to access the memory

- Faster than the other memory addressing modes
- Too many bits are needed to specify the address for a large physical memory space
- EA = IR(addr) (IR(addr): address field of IR)



- **e.g. ADD A**

Add contents of cell A to accumulator

Look in memory at address A for operand

- Single memory reference to access data
- No additional calculations to work out effective address
- Limited address space

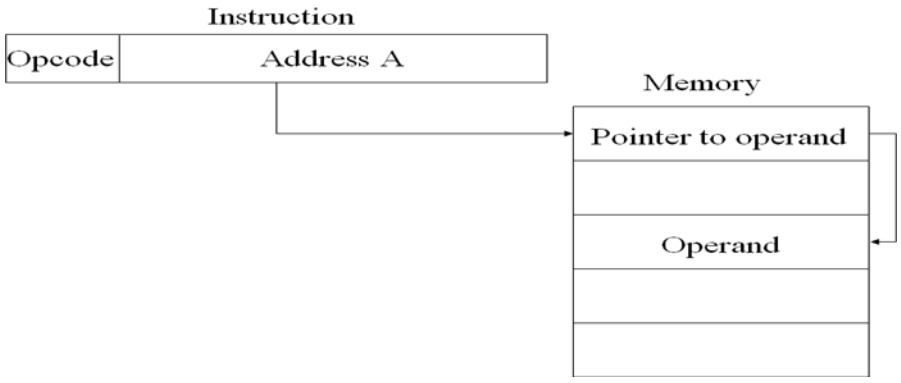
40. What is Indirect Addressing Mode?

The address field of an instruction specifies the address of a memory location that contains theaddress of the operand

- When the abbreviated address is used large physical memory can be addressed with a relatively small number of bits

- Slow to acquire an operand because of an additional memory access

- EA = M[IR(address)]



41. What is an effective address?

The effective address can be defined as address of the operand in a computation type instruction or the target address in a branch-type instruction.

42. What is a Program Counter? (Apr 13)

The program counter (PC) 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.

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.

A branch instruction calls for a transfer to a nonconsecutive instruction in the program. The address part of a branch instruction is transferred to PC to become the address of the next instruction.

43. What is an Instruction Register?

The instruction read from memory is placed in the **instruction register (IR).**

44. What is Data Register?

Data register holds the operand read from memory.

The computer needs processor registers for manipulating data.

45. What is Address Register?

Address register holds register for holding a memory address.

Memory **address register (AR)** has 12 bits since this is the width of a memory address.

46. What are the different types of Registers used for input and output?

Two registers are used for input and output.

Input register (INPR) receives an 8-bit character from an input device.

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

47. What is Relative Addressing Mode?

The Address fields of an instruction specifies the part of the address (abbreviated address) which can be used along with a designated register to calculate the address of the operand

- Address field of the instruction is short
- Large physical memory can be accessed with a small number of address bits
- EA = f(IR(address), R), R is sometimes implied

48. What are the different types of Relative Addressing Mode?

3 different Relative Addressing Modes depending on R

* PC Relative Addressing Mode (R = PC)
  - EA = PC + IR(address)
* Indexed Addressing Mode (R = IX, where IX: Index Register)
  - EA = IX + IR(address)
* Base Register Addressing Mode
  (R = BAR, where BAR: Base Address Register)
  - EA = BAR + IR(address)

49. What is an assembler?

Programming language processor that translates an assembly language program (the source program) to the machine language program (the object program) executable by a computer.

50. What are assembler directives?

In addition to providing a mechanism for representing instructions in a program ,the assembly language allows the programmer to specify other information needed to translate the source program into the object program. Suppose that the name SUM is used to represent the value 200 .this fact may be conveyed to the assembler program through a statement such as

SUM EQU 200

The above statement informs the assembler that the name SUM should be replaced by the value 200wherever it appears in the program, such statements, called assembler directives(or commands)

51. Write about program-controlled I/O (Nov 12)

The difference in speed between processor and I/O devices creates the need for mechanisms to synchronize the transfer of data between them. There will be several I/O devices connected to the processor; the processor checks the ``status'' input port periodically, under program control by the I/O handling procedure. If an I/O device requires service, it will signal this need by altering its input to the ``status'' port. When the I/O control program detects that this has occurred (by reading the status port) then the appropriate operation will be performed on the I/O device which requested the service.

52. List the advantages of program-controlled I/O

Program-controlled I/O has a number of advantages:

- All control is directly under the control of the program, so changes can be readily implemented.
- The order in which devices are serviced is determined by the program, this order is not necessarily fixed but can be altered by the program, as necessary. This means that the ``priority'' of a device can

be varied under program control. (The ``priority'' of a determines which of a set of devices which are simultaneously ready for servicing will actually be serviced first).

- It is relatively easy to add or delete devices.

53. What is pushdown stack?

A stack is a list of data elements, usually words or bytes with the accessing restriction that elements can be added or removed at one end of the list only. This end is called the top of the stack, and the other end is called the bottom. The structure is sometimes referred to as a pushdown stack.

54. Write short note on subroutine

It is necessary to perform a particular subtask many times on different data values. Such a subtask is usually called a subroutine.

55. What it is the use of link register?

The simplest subroutine linkage method is to save the return address in a specific location, which may be a register dedicated to this function. Such a register is called the link register.

56. What is parameter passing and how it is accomplished?

The exchange of information between a calling program and a subroutine is referred to as parameter passing. It may be accomplished in several ways. The parameter may be placed in registers or memory location, where they can be accessed by the subroutine. Alternatively, the parameters may be placed on the processor stack used for saving the return address.

57. Write about shift instruction

There are many applications that require the bits of an operand to be shifted right or left some specified number of bit positions. There are two types of shifts

- Logical shift left (LshiftL)
- Logical shift right (LshiftR)
- Arithmetic shift right

58. Write about condition code register or status register.

Result of various operation for user by subsequent conditional branch instruction is accomplished by recording the required information in individual bits often called **Condition code Flags.** These flags are usually grouped together in a special processor register called the **condition code register or status register.**

59. Write some commonly used flags

- **N(Negative)**→set to 1 if the result is –ve ,otherwise cleared to 0.
- **Z(Zero)**→ set to 1 if the result is 0 ,otherwise cleared to 0.
- **V(Overflow)**→ set to 1 if arithmetic overflow occurs ,otherwise cleared to 0.
- **C(Carry)**set to 1 if carry and results from the operation ,otherwise cleared to 0

60. What is Subroutines?

In a given program, it is often necessary to perform a particular task many times on different data values. It is prudent to implement this task as a block of instructions that is executed each time the task has to be performed. Such a block of instructions is usually called a subroutine.

61. What is subroutine linkage?

The way in which a computer makes it possible to call and return from subroutines is referred to as its subroutine linkage method. The simplest subroutine linkage method is to save the return address in a specific location, which may be a register dedicated to this function. Such a register is called the link register. When the subroutine completes its task, the Return instruction returns to the calling program by branching indirectly through the link register.

62. Write about parameter passing

When calling a subroutine, a program must provide to the subroutine the parameters, that is, the operands or their addresses, to be used in the computation. Later, the subroutine returns other parameters, which are the results of the computation. This exchange of information between a calling program and a subroutine is referred to as parameter passing.

63. Write about Logical Shifts instruction

Two logical shift instructions are needed, one for shifting left (LShiftL) and another for shifting right (LShiftR). These instructions shift an operand over a number of bit positions specified in a count operand contained in the instruction. The general form of a Logicalshift-left instruction is

LShiftL R*i*, R*j*, count

which shifts the contents of register R*j* left by a number of bit positions given by the count operand, and places the result in register R*i*, without changing the contents of R*j*.

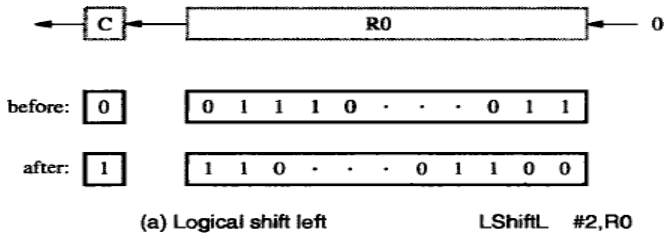
The count operand may be given as an immediate operand, or it may be contained in a processor register.

64. Write about the Logical-shift-left instruction

The instruction

LShiftL #2,R0

The above instruction shifts the contents of register R0 left by two bit positions is shown below.



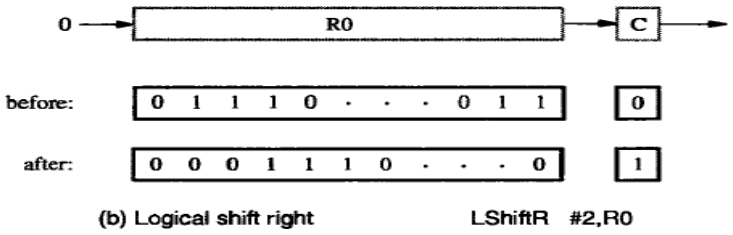
(a) Logical shift left LShiftL #2,R0

65. Write about the Logical-shift-right instruction

The instruction

LShiftR #2,R0

The above instruction shifts the contents of register R0 right by two bit positions is shown below.



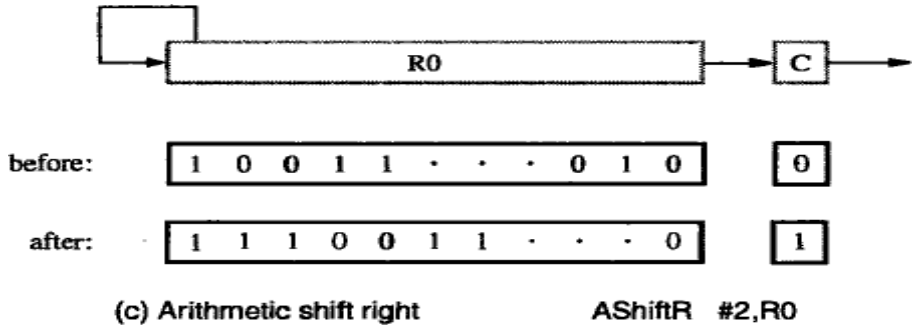
(b) Logical shift right LShiftR #2,R0

66. Write about Arithmetic Shifts instruction

In an arithmetic shift, the bit pattern being shifted is interpreted as a signed number. When a 2's-complement binary number shifted, a number one bit position to the left is equivalent to multiplying it by 2, and shifting it to the right is equivalent to dividing it by 2. Of course, overflow might occur on shifting left, and the remainder is lost when shifting right. Another important observation is that on a right shift the sign bit must be repeated as the fill-in bit for the vacated position as a requirement of the 2's-complement representation for numbers. This requirement when shifting right distinguishes arithmetic shifts from logical shifts in which the fill-in bit is always 0. Otherwise, the two types of shifts are the same. An example of an Arithmetic shift- right instruction, AShiftR, The Arithmetic-shift-left is exactly the same as the Logical-shift-left. For example ,

AShiftR #2,R0

The above instruction shifts the contents of register R0 right by two bit positions and insert's 1 in the vacated positions is shown below.



(c) Arithmetic shift right AShiftR #2,R0

67. Write about Rotate instruction

In the shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is retained in the Carry flag C. For situations where it is desirable to preserve all of the bits, rotate instructions may be used instead. These are instructions that move the bits shifted out of one end of the operand into the other end. Two versions of both the Rotate-left and Rotate-right instructions are often provided.

In one version, the bits of the operand are simply rotated. In the other version, the rotation includes the C flag. When the C flag is not included in the rotation, it still retains the last bit shifted out of the end of the register.

68. List out the various rotate instruction

In addition to the shift instructions, there are also four rotate instructions:

- RotateL (Rotate left without the carry flag CF)
- RotateR (Rotate right without the carry flag CF)

- RotateLC (Rotate left including the carry flag CF)
- RotateRC (Rotate right including the carry flag CF)

The OP codes RotateL, RotateLC, RotateR, and RotateRC, denote the instructions that perform the rotate operations.

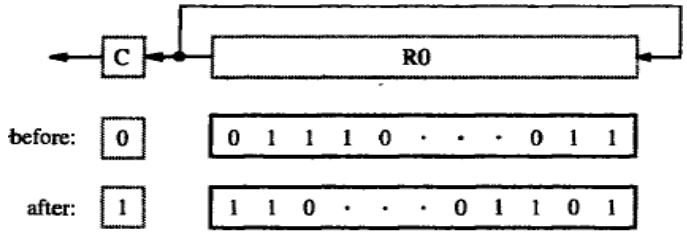
69. Write about RotateL instruction

This instruction rotates the content of the specified register on left by the specified number of bit position which is specified in the count variable.

For example,the instruction

RotateL #2,R0

Rotates the content of R0 to left by 2 bit positions but not through the carry is shown below.



(a) Rotate left without carry RotateL #2,R0

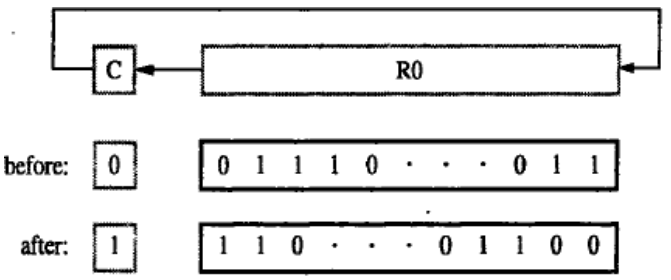
70. Write about RotateLC instruction

This instruction rotates the content of the specified register on left by the specified number of bit position through carry; the number of shift position will be specified in the count variable.

For example,the instruction

RotateLC #2,R0

Rotates the content of R0 to left by 2 bit positions through the carry is shown below.



(b) Rotate left with carry RotateLC #2,R0

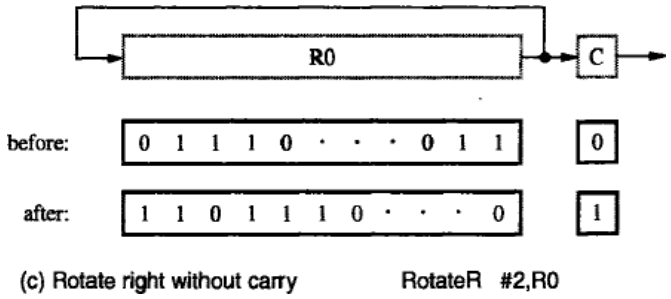
71. Write about RotateR instruction

This instruction rotates the content of the specified register on right by the specified number of bit position which is specified in the count variable.

For example,the instruction

RotateR #2,R0

Rotates the content of R0 to right by 2 bit positions but not through the carry is shown below.



(c) Rotate right without carry RotateR #2,R0

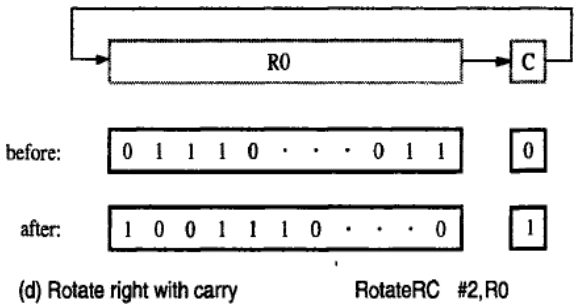
72. Write about RotateRC instruction

This instruction rotates the content of the specified register on right by the specified number of bit position through carry, the number of shift position will be specified in the count variable.

For example, the instruction

RotateRC #2,R0

Rotates the content of R0 to right by 2 bit a position through the carry is shown below.

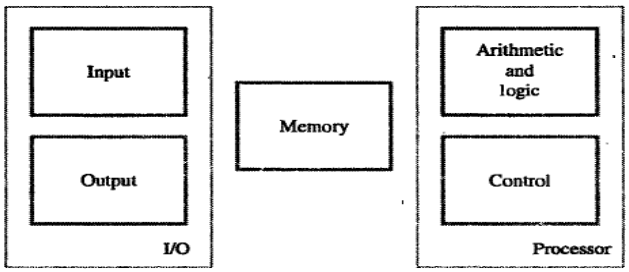


(d) Rotate right with carry RotateRC #2,R0

11 Marks

1. Write in detail about Functional units of a computer.

Functional units of a Computer



A computer consists of 5 main parts.

- Input
- Memory
- Arithmetic and logic
- Output
- Control Units

Input unit accepts coded information from human operators, from electromechanical devices such as keyboards, or from other computers over digital communication lines. The information received is either stored in the computer's memory for later reference or immediately used by the arithmetic and logic circuitry to perform the desired operations.

The processing steps are determined by a program stored in the memory. Finally the results are sent back to the outside world through the output unit. All of these actions are coordinated by the control unit. The list of instructions that performs a task is called a program. Usually the program is stored in the memory.

The processor then fetches the instruction that makes up the program from the memory one after another and performs the desire operations.

Input Unit:

- Computers accept coded information through input units, which read the data.
- Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted over a cable to either the memory or the processor.
- Some input devices are
  - ✓ Joysticks
  - ✓ Trackballs
  - ✓ Mouses
  - ✓ Microphones (Capture audio input and it is sampled & it is converted into digital codes for storage and processing).

Memory Unit:

It stores the programs and data.

There are 2 types of storage classes

- ✓ Primary
- ✓ Secondary

Primary Storage:

- It is a fast memory that operates at electronic speeds.
- Programs must be stored in the memory while they are being executed.
- The memory contains large no of semiconductor storage cells.
- Each cell carries one bit of information.
- The cells are processed in a group of fixed size called words. To provide easy access to any word in a memory, a distinct address is associated with each word location.
- Addresses are numbers that identify successive locations.
- The number of bits in each word is called the word length.
- The word length ranges from 16 to 64 bits.
- There are 3 types of memory. They are
  - Random Access Memory
  - Cache memory
  - Main Memory

RAM:

- Memory in which any location can be reached in short and fixed amount of time after specifying its address is called RAM. Time required to access 1 word is called Memory Access Time.

Cache Memory:

- The small,fast,RAM units are called Cache. They are tightly coupled with processor to achieve high performance.

Main Memory:

- The largest and the slowest unit is called the main memory.

ALU:

- Most computer operations are executed in ALU.
- Consider a example, Suppose 2 numbers located in memory are to be added. They are brought into the processor and the actual addition is carried out by the ALU. The sum may then be stored in the memory or retained in the processor for immediate use. Access time to registers is faster than access time to the fastest cache unit in memory.

Output Unit:

- Its function is to send the processed results to the outside world. eg.Printer Printers are capable of printing 10000 lines per minute but its speed is comparatively slower than the processor.

Control Unit:

- The operations of Input unit, output unit, ALU are co-ordinate by the control unit. The control unit is the Nerve centre that sends control signals to other units and senses their states. Data transfers between the processor and the memory are also controlled by the control unit through timing signals.
- The operation of computers are,
  - ✓ The computer accepts information in the form of programs and data through an input unit and stores it in the memory
  - ✓ Information stored in the memory is fetched, under program control into an arithmetic and logic unit, where it is processed.
  - ✓ Processed information leaves the computer through an output unit.
  - ✓ All activities inside the machine are directed by the control unit.

2. Write short notes on multi processor and multi computer

Multi computer:

- Multi computer is an interconnected group of complete computers to achieve high total computation power.. The term generally refers to an architecture in which each processor has its own memory rather than multiple processors with a shared memory. Something similar to parallel computing.
- Distributed computing deals with hardware and software systems containing more than one processing element or storage element, concurrent processes, or multiple programs, running under a loosely or tightly controlled regime.
- A multicomputer may be considered to be either a loosely coupled NUMA computer or a tightly coupled cluster. Multicomputer is commonly used when strong computer power is required in an environment with restricted physical space or electrical power.
- Common suppliers include Mercury Computer Systems, CSPI, and SKY Computers. Common uses include 3D medical imaging devices and mobile radar.

- In distributed computing a program is split up into parts that run simultaneously on multiple computers communicating over a network. Distributed computing is a form of parallel computing, but parallel computing is most commonly used to describe program parts running simultaneously on multiple processors in the same computer.
- Both types of processing require dividing a program into parts that can run simultaneously, but distributed programs often must deal with heterogeneous environments, network links of varying latencies, and unpredictable failures in the network or the computers

Multi processor:

- Large computer systems may contain a number of processor units, in which case they are called multiprocessor. These systems either execute a number of different application tasks in parallel or they execute subtasks of a single large task in parallel
- Multiprocessing is the use of two or more central processing units (CPUs) within a single computer system. The term also refers to the ability of a system to support more than one processor and/or the ability to allocate tasks between them. There are many variations on this basic theme, and the definition of multiprocessing can vary with context, mostly as a function of how CPUs are defined (multiple cores on one die, multiple chips in one package, multiple packages in one system unit, etc.).

3. Write in detail about memory location and addresses

- Number and character operands, as well as instructions are stored in the memory of a computer. The memory consist of many millions of storage cells, each of which can store a bit of information having the value 0 or 1.because a single bit represents a very small amount of information, bits are seldom handled individually. The usual approach is to deal with them in groups of fixed size.
- The memory is organized so that a group of n bits can be stored or retrieved in a single basic operation. Each group of n bits is referred to as a word of information, and n is called the word length. The memory of a computer can be represented as a collection of words shown below.

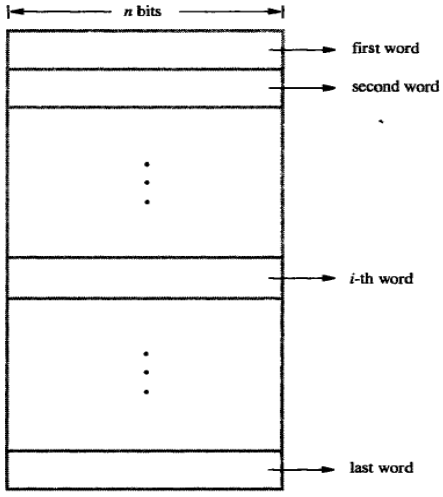


Figure 2.5 Memory words.

- Accessing the memory to store or retrieve a single item of information ,either a word or a byte, requires distinct names or addresses for each item location.

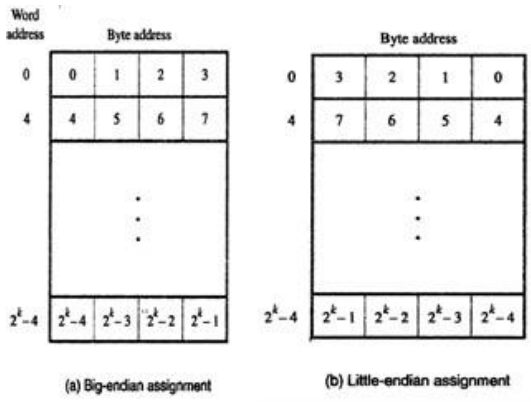
Byte addressability

There are three basic information quantities to deal with: the bit, byte, and word .a byte is always 8 bits, but the word length typically ranges from 16 to 64 bits. It is impractical to assign n distinct addresses to individual bit locations in the memory. the most practical assignment is to have successive addresses refer to

successive byte locations in the memory .this the assignment used in modern computers, called byte-addressable memory. Byte locations have addresses 0,1,2…….,thus if the word length of the machine is 32 bits ,successive words are located at addresses 0,4,8,….,with each word consisting of four bytes**.**

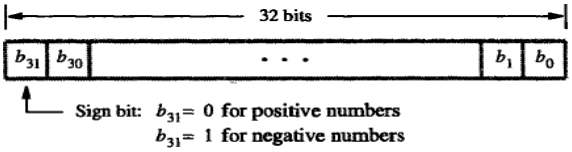
Big –endian and Little-endian assignments

- Big-endian and little-endian are terms that describe the order in which a sequence of bytes are stored in computer memory. The name big-endian is used when lower byte addresses are used for the more significant bytes (the leftmost bytes)of the word.
- The name little0endian is used for the opposite ordering, where the lower byte addresses are used for the less significant bytes (the rightmost bytes)of the word. The words "more significant" and "less significant" are used in relation to the weights (powers of 2) assigned to bits when the word represents a number. Both little-endian and big-endian assignments are used in commercial machines.
- In both cases ,byte addresses 0,4,8,….are taken as the addresses of successive words in the memory and are the addresses used when specifying memory read and write operations for words.



(a) Big-endian assignment (b) Little-endian assignment

- In addition to specifying the address ordering of bytes within a word, it is also necessary to specify the labeling of bits within a byte or a word. The most common convention is shown below



(a) A signed integer

- It is the most natural ordering for the encoding of numerical data. The same ordering is also used for labeling bits within a byte, that is b_7,b_6,…..b_0,from left to right.

Word alignment

In the case of a 32-bit word length, natural word boundaries occur at addresses 0,4,8…..the word locations have aligned addresses. In general, words are said to be aligned in memory if they begin at a byte address that is multiple of the number of bytes in a word. The number of bytes in a word is power of 2.hence,if the word length is 16(2 bytes),aligned words begin at byte addresses 0,2,4,…..and for a word length of 64(23 bytes),aligned words begin at byte addresses 0,8,16,….

Accessing numbers, characters, and character strings

A number usually occupies one word. It can be accessed in the memory by specifying its word address. Similarly, individual characters can be accessed by their byte address. In many applications, it is necessary to handle character strings of variable length .the beginning of the string is indicated by giving the address of the byte containing its first character. Successive byte locations contain successive characters of the string.

There are two ways to indicate the length of the string .a special control character with the meaning "end of the string "cam be used as the last character in the string, or a sequence memory word location or processor register can contain a number indicating the length of the string in bytes.

4. Write short notes on memory operations

The general- purpose computers use a set of instructions called a program to process data. A computer executes the program to create output data from input data. Both program instructions and data operands are stored in memory. Two basic operations requires in memory access Load operation (Read or Fetch)- Contents of specified memory location are read by processor. Store operation (Write)- Data from the processor is stored in specified memory location.

The load operation transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged. To start a load operation the processor sends the address of the desired location to the memory and requests that its contents be read. The memory reads the data stored at this address and sends them to the processor.

The store operation transfers an item of information from the processor to a specific memory location, destroying the former contents of that location. The processor sends the address of the desired location to the memory, together with the data to be written into that location.

5. Explain instructions and instruction sequencing

A computer must have instruction capable of performing the following operations. They are,

- Data transfer between memory and processor register.
- Arithmetic and logical operations on data.
- Program sequencing and control.
- I/O transfer.

Register Transfer Notation:

The possible locations in which transfer of information occurs are,

- Memory Location
- Processor register
- Registers in I/O sub-system.

Location	**Hardware Binary Address**	**Example**	**Description**
Memory	LOC,PLACE,A,VAR2	R1←[LOC]	The contents of memory location are transferred to. the processor register
Processor	R0,R1,….	[R3]←[R1]+[R2]	Add the contents of register R1 &R2 and places .their sum into register R3.It is .called Register Transfer Notation.
I/O registers	DATAIN,DATAOUT		Provides Status information

Assembly Language Notation:

It is another type of notation to represent machine instructions and programs.

Assembly Language Format	Description
Move LOC,R1	Transfers the contents of memory location to the processor register R1.
Add R1,R2,R3	Add the contents of register R1 & R2 and places their sum into register R3

Basic Instruction Types:

The operation of adding two numbers os 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. The contents of these locations represents the values of the 3 variables. Hence ,the above high –level language statement requires the action

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

The following table shows the basic instruction types:

Instruction Type	Syntax	Example	Description
Three Address	Operation Source1,Source2,Destination	Add A,B,C	Add values of variable A ,B & place the result into c.
Two Address	Operation Source,Destination	Add A,B	Add the values of A,B & place the result into B.
One Address	Operation Operand	Add B	Content of accumulator add with content of B.

Instruction Execution and Straight–line Sequencing:

Instruction Execution:

There are 2 phases for Instruction Execution. They are,

- Instruction Fetch
- Instruction Execution

Instruction Fetch:

The instruction is fetched from the memory location whose address is in PC.This is placed in IR.

Instruction Execution:

Instruction in IR is examined to determine whose operation is to be performed.

Program execution Steps:

- To begin executing a program, the address of first instruction must be placed in PC.
- The processor control circuits use the information in the PC to fetch & execute instructions one at a time in the order of increasing order.
- This is called Straight line sequencing. During the execution of each instruction, the PC is incremented by 4 to point the address of next instruction. Thus arte 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 second phase ,called instruction execute, the instruction in IR is examined to determine which operation is to be performed.
- The specified operation is the performed by the processor.

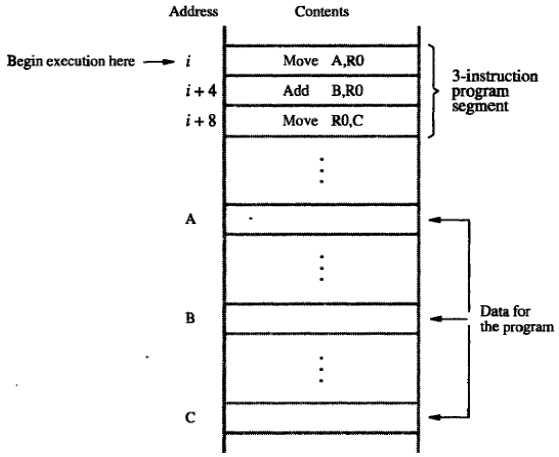


Fig. Program for C←[A]+[B]

Branching:

- Consider a task of adding a list of n numbers.The Address of the memory locations containing the n numbers are symbolically given as NUM_1,NUM_2…..NUM_n and a separate Add instruction is used to add each number to the contents of register R0. After all the numbers have been added,the result is placed in memory location SUM.

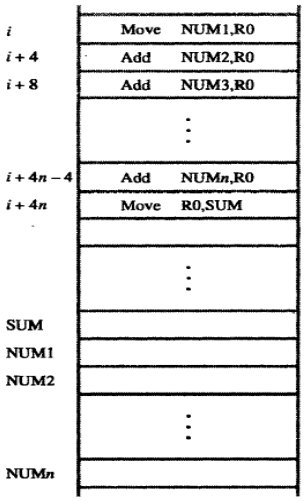


Fig. Straight line program for adding n numbers

Using loop to add 'n' numbers:

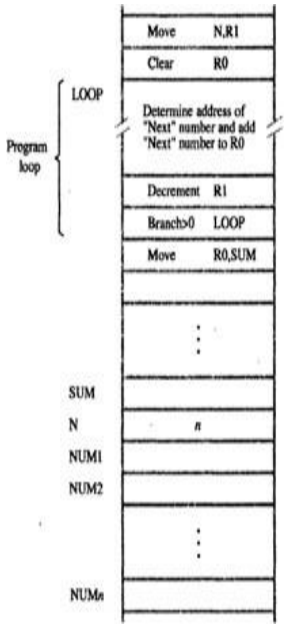
- Number of entries in the list „n" is stored in memory location M.Register R1 is used as a counter to determine the number of times the loop is executed.
- Content locations M are loaded into register R1 at the beginning of the program.
- It starts at location Loop and ends at the instruction. Branch>0.During each pass, the address of the next list entry is determined and the entry is fetched and added to R0.

Decrement R1

- It reduces the contents of R1 by 1 each time through the loop.

Brach > 0 Loop

- A conditional branch instruction causes a branch only if a specified condition is satisfied.



Conditional Codes:

- Result of various operation for user by subsequent conditional branch instruction is accomplished by recording the required information in individual bits often called **Condition code Flags.**
- These flags are usually grouped together in a special processor register called the **condition code register or status register.**
- Individual condition code flags are set to 1 or cleared to 0,depending on the outcome of the operation performed.

Commonly used flags:

- **N(Negative)**→set to 1 if the result is –ve ,otherwise cleared to 0.
- **Z(Zero)**→ set to 1 if the result is 0 ,otherwise cleared to 0.
- **V(Overflow)**→ set to 1 if arithmetic overflow occurs ,otherwise cleared to 0.
- **C(Carry)**set to 1 if carry and results from the operation ,otherwise cleared to 0

The N and Z flags indicate whether the result of arithmetic of logic operation is negative or zero. The N and Z flags may also be affected by instructions that transfer data, such as Move, Load or Store. This makes it possible for a later conditional branch instruction to cause a branch based on the sign and value of the operand that was moved.

6. Write in detail about addressing modes (Apr 13)

The different ways in which the location of an operand is specified in an instruction is called as Addressing mode.

Generic Addressing Modes are :

- Immediate mode
- Register mode
- Absolute mode

- Indirect mode
- Index mode
- Base with index
- Base with index and offset
- Relative mode
- Auto-increment mode
- Auto-decrement mode

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*]

Implementation of Variables and Constants:

Variables:

In assembly language, a variable is represented by allocating a register or a memory location to hold its value. The value can be changed as needed using the appropriate instructions. There are 2 accessing modes to access the variables. They are

- Register Mode
- Absolute Mode

Register Mode:

The operand is the contents of the processor register. The name(address) of the register is given in the instruction.

Absolute Mode (Direct Mode):

The operand is in memory location. The address of this location is given explicitly in the instruction. In some assembly languages, this mode is called Direct.

Eg: MOVE LOC,R2

The above instruction uses the register and absolute mode. The processor register is the temporary storage where the data in the register are accessed using register mode. The absolute mode can represent global variables in the program.

Mode	Assembler Syntax	Addressing Function
Register mode	Ri	EA=Ri
Absolute mode	LOC	EA=LOC

Where **EA**-Effective Address

Constants:

Address and data constants can be represented in assembly language using Immediate Mode.

Immediate Mode.

The operand is given explicitly in the instruction.

Eg: Move 200 $_{immediate}$, R0

It places the value 200 in the register R0.The immediate mode used to specify the value of source operand. In assembly language, the immediate subscript is not appropriate so # symbol is used. It can be re-written as Move #200,R0

Assembly Syntax:	**Addressing Function**
Immediate #value	Operand =value

Indirection and Pointers:

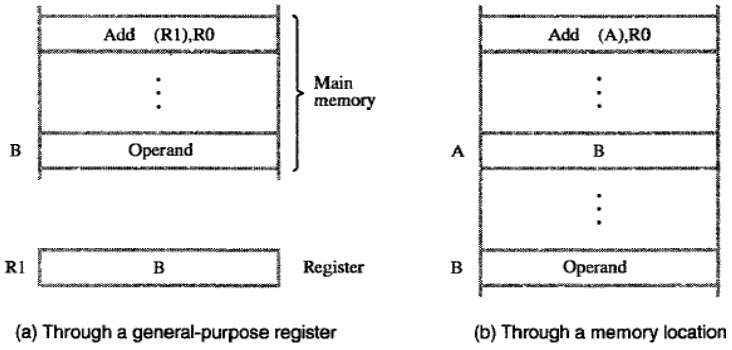
Instruction does not give the operand or its address explicitly. Instead it provides information from which the new address of the operand can be determined. This address is called effective Address(EA) of the operand

Indirect Mode:

The effective address of the operand is the contents of a register . We denote the indirection by the name of the register or new address given in the instruction. Address of an operand(B) is stored into R1 register. If we want this operand, we can get it through register R1(indirection). The register or new location that contains the address of an operand is called the **pointer.**

Mode	**Assembler syntax**	**Addressing function**
Indirect	Ri , LOC	EA=[Ri] or EA=[LOC]

Example:



(a) Through a general-purpose register (b) Through a memory location

Figure 2.11 Indirect addressing.

To execute the Add instruction fig-a ,the processor uses the value B which is in register R1,as the EA of the operand .it requests a read operation from the memory to read the contents of the 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 as shown in fig-b. in this case ,the processor first reads the contents of memory location A, then requests a second read operation using the value B as an address to obtain the operand. The register or memory location that contains the address of an operand is called a pointer.

Indexing and Arrays:

Index Mode:

The effective address of an operand is generated by adding a constant value to the contents of a register. The constant value uses either special purpose or general purpose register. The index mode is indicated symbolically as,

$X(R_i)$

Where

X – denotes the constant value contained in the instruction

R_i – It is the name of the register involved.

The Effective Address of the operand is,

EA=X + [Ri]

The index register R1 contains the address of a new location and the value of X defines an offset(also called a displacement).

To find operand,

First go to Reg R1 (using address)-read the content from R1-1000

Add the content 1000 with offset 20 get the result.

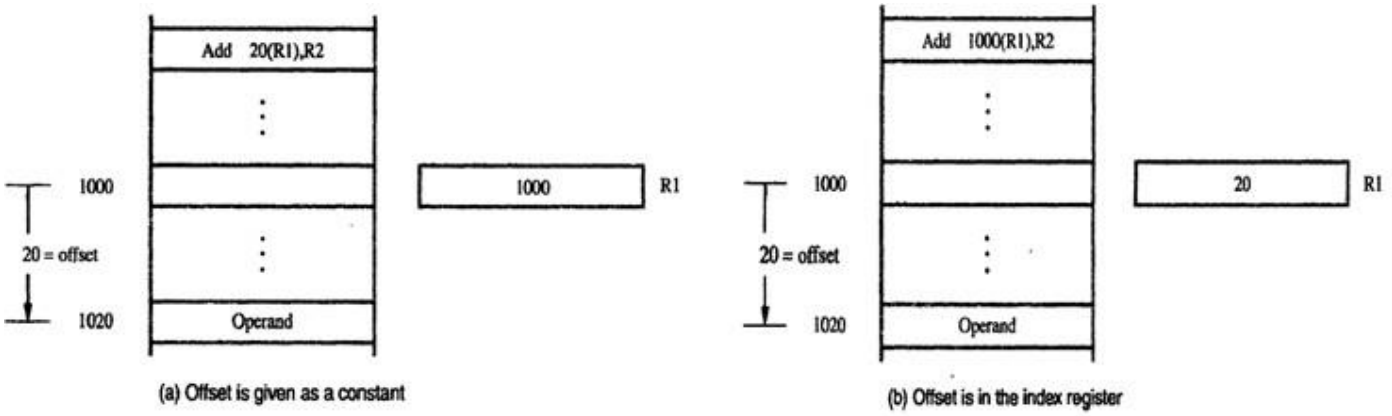
1000+20=1020

Here the constant X refers to the new address and the contents of index register define the offset to the operand. The sum of two values is given explicitly in the instruction and the other is stored in register.

Eg: Add 20(R1) , R2 (or) EA=>1000+20=1020

Index Mode	Assembler Syntax	Addressing Function
Index	X(Ri)	EA=[Ri]+X
Base with Index	(Ri,Rj)	EA=[Ri]+[Rj]
Base with Index and offset	X(Ri,Rj)	EA=[Ri]+[Rj] +X

Example



The above figure illustrates two ways of using the index mode .in fig –a ,the index register R1.contains the address of a memory location ,and the value X defined an offset(also called a displacement)from this address to the location where the operand is found.

An alternative in fig-b illustrates that the constant X corresponds 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.

Relative Addressing:

It is same as index mode. The difference is, instead of general purpose register, here we can use program counter(PC).

Relative Mode:

The Effective Address is determined by the Index mode using the PC in place of the general purpose register This mode can be used to access the data operand. But its most common use is to specify the target address in branch instruction.Eg. Branch>0 Loop It causes the program execution to goto the branch target location. It is identified by the name loop if the branch condition is satisfied.

Mode	Assembler syntax	Addressing function
Relative	X(PC)	EA=[PC]+X

Additional Modes:

There are two additional modes. They are

- Auto-increment mode
- Auto-decrement mode

These two modes are useful for accessing data items in successive locations in the memory.

Auto-increment mode:

The Effective Address of the operand is the contents of a register in the instruction. After accessing the operand, the contents of this register is automatically incremented to point to the next item in the list.

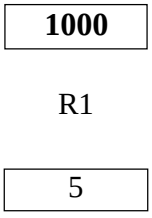
The auto increment mode is denoted by putting the specified register in parentheses to show that the contents of the register are used as the effective address ,followed by a plus sign to indicate that these contents are to be incremented after the operand is accessed. Thus the autoincrement mode is written as shown below

Mode	Assembler syntax	Addressing function
Auto-increment	(Ri)+	EA=[Ri]; Increment Ri

Example

Assume the content of R1 is 1000 and the location 1000 contains the value 5 ,which is nothing but the operand. Then the instruction

(R1)+



After accessing the location 1000, the content of R1 will be incremented by 1 for byte –sixed operands,2 for 16-bit operands ,and 4 for 32-bit operands. In this case suppose R1 is pointing integer value ,then R1 is incremented by 2.then R1 contains 1002.

Auto-decrement mode:

The Effective Address of the operand is the contents of a register in the instruction. After accessing the operand, the contents of this register is automatically decremented to point to the next item in the list.

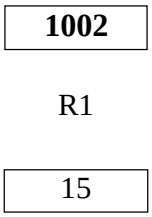
The auto decrement mode is denoted by putting the specified register in parentheses ,preceded by a minus sign to indicate that the contents of the register are to be decremented before being used as the effective address. Thus the autodecrement mode is written as shown below

Mode	Assembler syntax	Addressing function
Auto-decrement	-(Ri)	EA=[Ri]; Decrement Ri

Example

Assume the content of R1 is 1002 and the location 1002 contains the value 15 ,which is nothing but the operand. Then the instruction

-(R1)



Before accessing the operand, the content of R1 will be decremented by 1 for byte –sixed operands,2 for 16-bit operands ,and 4 for 32-bit operands. In this case suppose R1 is pointing integer value ,then R1 is decremented by 2.then R1 contains 1000,then this instruction access the value 15.

7. Write short notes assembly language and assembler directives

Assembly language:

Machine instructions are represented by patterns 0's and 1's.such patterns are awkward to deal with then preparing programs. Therefore symbolic names are used to represent the patterns, such as Move, Add, Increment and Branch .when writing programs for a specific computer, such words are replaced by acronym called mnemonics .such as MOV,ADD,INC ,and BR.A completes set of such symbolic names and rules for their use constitute a programming language, generally referred to as an assembly language.

Example instruction:

ADD #5,R3

Adds 5 to the contents of register R3 and puts the result back into register R3.

Assembler directives:

In addition to providing a mechanism for representing instructions in a program, the assembly language allows the programmer to specify other information needed to translate the source program into the object program. Suppose that the name SUM is used to represent the value 200 .this fact may be conveyed to the assembler program through a statement such as

SUM EQU 200

The above statement informs the assembler that the name SUM should be replaced by the value 200wherever it appears in the program, such statements, called assembler directives(or commands)

Example

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	

In order to run this program on a computer, it is necessary to write its source code in the required assembly language ,specifying all the information needed to generate the corresponding object program. And then the object program is loaded into main memory shown below.

	100	Move	N,R1
	104	Move	#NUM1,R2
	108	Clear	R0
LOOP	112	Add	(R2),R0
	116	Add	#4,R2
	120	Decrement	R1
	124	Branch>0	LOOP
	128	Move	R0,SUM
	132		
		⋮	
SUM	200		
N	204	100	
NUM1	208		
NUM2	212		
		⋮	
NUM*n*	604		

The below figure shows the assembly language representation for the above program. The assembler directive EQU ,informs the assembler about the value of SUM. The second assembler directive ORIGIN ,tells the assembler program where in the memory to place the data block that follows. In this case, the location specified has the address 204 Since this location is to be loaded with the value 100 ,a DATAWORD directive is used to inform the assembler of this requirement. It states that the data value 100 is to be placed in the memory word at address 204.

Any statement that results in instructions or data being placed in a memory location may be given a memory address label.the label is assigned a value equal to the address of that location .because the DATAWORD statement is given the label N .the name N is assigned the value 204.whenever N is encountered in the rest of the program ,it is replace with the value 204.using N as a label in this manner is equivalent to using the assembler directive

N EQU 204

The RESERVE assembler directive declares that a memory block of 400 bytes is to be reserved for data, and that the name NUM1 is to be associated with address 208.the second ORIGIN directive specifies that the instructions of the object program are to be loaded in the memory starting at address 100.the END directive ,which tells the assembler that this is the end of the source program text.

8. Explain with example about Shift and Rotate Instructions

There are many applications that require the bits of an operand to be shifted right or left some specified number of bit positions. The details of how the shifts are performed depend on whether the operand is a signed number or some more general binary-coded information. For general operands, we use a logical shift. For a signed number, we use an arithmetic shift, which preserves the sign of the number.

Logical Shifts

Two logical shift instructions are needed, one for shifting left (LShiftL) and another for shifting right (LShiftR). These instructions shift an operand over a number of bit positions specified in a count operand contained in the instruction. The general form of a Logicalshift-left instruction is

LShiftL R*i*, R*j*, count

which shifts the contents of register R*j* left by a number of bit positions given by the count operand, and places the result in register R*i*, without changing the contents of R*j*.

The count operand may be given as an immediate operand, or it may be contained in a processor register. Need to specify the bit values brought into the vacated positions at the right end of the destination operand, and to determine what happens to the bits shifted out of the left end. Vacated positions are filled with zeros.

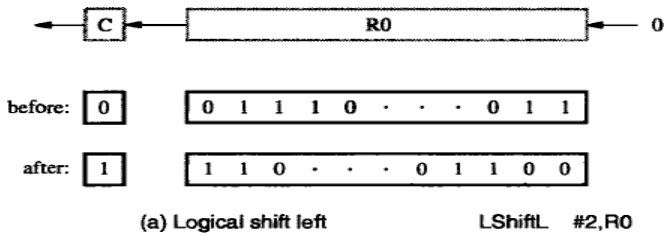
In computers that do not use condition code flags, the bits shifted out are simply dropped. In computers that use condition code flags, these bits are passed through the Carry flag, C, and then dropped. Involving the C flag in shifts is useful in performing arithmetic operations on large numbers that occupy more than one word.

The Logical-shift-left:

The instruction

LShiftL #2,R0

The above instruction shifts the contents of register R0 left by two bit positions is shown below.



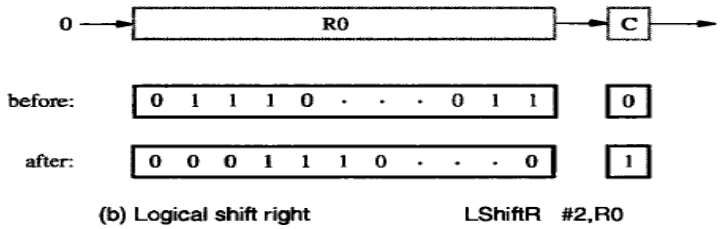
(a) Logical shift left LShiftL #2,R0

The Logical-shift-right:

The instruction

LShiftR #2,R0

The above instruction shifts the contents of register R0 right by two bit positions is shown below.



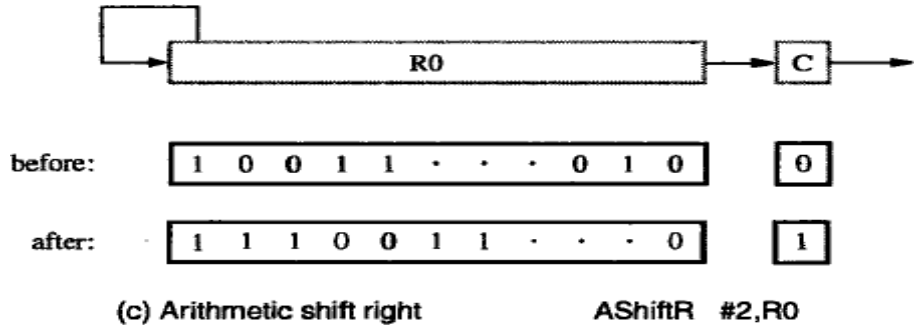
(b) Logical shift right LShiftR #2,R0

Arithmetic Shifts:

In an arithmetic shift, the bit pattern being shifted is interpreted as a signed number. When a 2's-complement binary number shifted, a number one bit position to the left is equivalent to multiplying it by 2, and shifting it to the right is equivalent to dividing it by 2. Of course, overflow might occur on shifting left, and the remainder is lost when shifting right. Another important observation is that on a right shift the sign bit must be repeated as the fill-in bit for the vacated position as a requirement of the 2's-complement representation for numbers. This requirement when shifting right distinguishes arithmetic shifts from logical shifts in which the fill-in bit is always 0. Otherwise, the two types of shifts are the same. An example of an Arithmeticshift- right instruction, AShiftR, The Arithmetic-shift-left is exactly the same as the Logical-shift-left.For example ,

AShiftR #2,R0

The above instruction shifts the contents of register R0 right by two bit positions and insert's 1 in the vacated positions is shown below.



(c) Arithmetic shift right AShiftR #2,R0

Rotate instruction

In the shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is retained in the Carry flag C. For situations where it is desirable to preserve all of the bits, rotate instructions may be used instead. These are instructions that move the bits shifted out of one end of the operand into the other end. Two versions of both the Rotate-left and Rotate-right instructions are often provided.

In one version, the bits of the operand are simply rotated. In the other version, the rotation includes the C flag. When the C flag is not included in the rotation, it still retains the last bit shifted out of the end of the register.

In addition to the shift instructions, there are also four rotate instructions:

- RotateL (Rotate left without the carry flag CF)
- RotateR (Rotate right without the carry flag CF)
- RotateLC (Rotate left including the carry flag CF)
- RotateRC (Rotate right including the carry flag CF)

The OP codes RotateL, RotateLC, RotateR, and RotateRC, denote the instructions that perform the rotate operations.

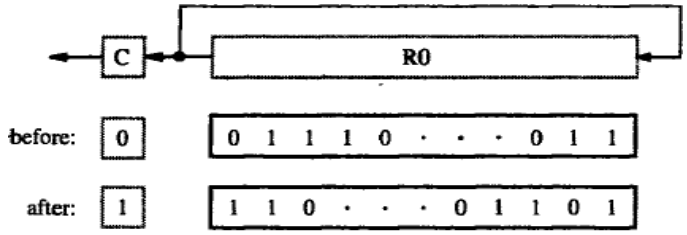
RotateL (Rotate left without the carry flag CF)

This instruction rotate the content of the specified register on left by the specified number of bit position which is specified in the count variable.

For example, the instruction

RotateL #2,R0

Rotates the content of R0 to left by 2 bit positions but not through the carry is shown below.



(a) Rotate left without carry RotateL #2,R0

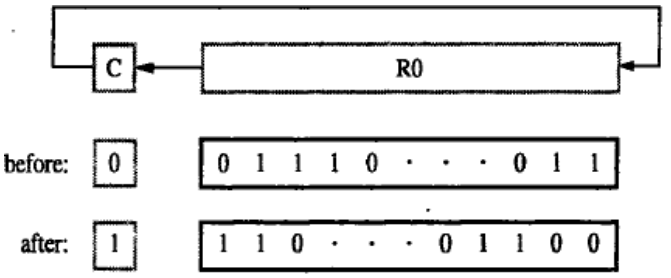
RotateLC (Rotate left including the carry flag CF)

This instruction rotate the content of the specified register on left by the specified number of bit position through carry, the number of shift position will be specified in the count variable.

For example, the instruction

RotateLC #2,R0

Rotates the content of R0 to left by 2 bit positions through the carry is shown below.



(b) Rotate left with carry RotateLC #2,R0

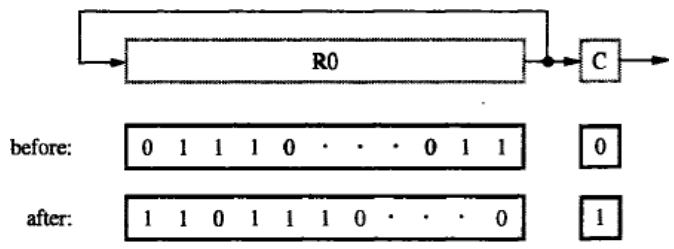
RotateR (Rotate right without the carry flag CF)

This instruction rotate the content of the specified register on right by the specified number of bit position which is specified in the count variable.

For example, the instruction

RotateR #2,R0

Rotates the content of R0 to right by 2 bit positions but not through the carry is shown below.



(c) Rotate right without carry RotateR #2,R0

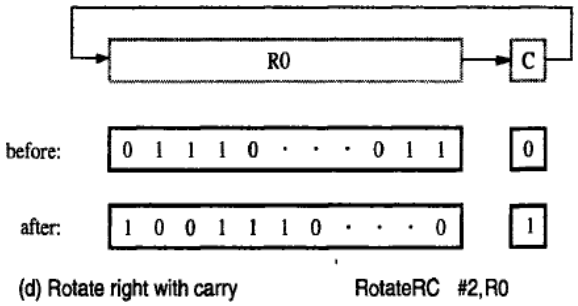
RotateRC (Rotate right including the carry flag CF)

This instruction rotate the content of the specified register on right by the specified number of bit position through carry, the number of shift position will be specified in the count variable.

For example, the instruction

RotateRC #2,R0

Rotates the content of R0 to right by 2 bit a position through the carry is shown below.



(d) Rotate right with carry RotateRC #2,R0

	Memory address label	Operation	Addressing or data information
Assembler directives	SUM	EQU	200
		ORIGIN	204
	N	DATAWORD	100
	NUM1	RESERVE	400
		ORIGIN	100
Statements that	START	MOVE	N,R1
generate		MOVE	#NUM1,R2
machine		CLR	R0
instructions	LOOP	ADD	(R2),R0
		ADD	#4,R2
		DEC	R1
		BGTZ	LOOP
		MOVE	R0,SUM
Assembler directives		RETURN	
		END	START

9. Describe about subroutines

Subroutines

In a given program, it is often necessary to perform a particular task many times on different data values. It is prudent to implement this task as a block of instructions that is executed each time the task has to be performed. Such a block of instructions is usually called a *subroutine*.

For example, a subroutine may evaluate a mathematical function, or it may sort a list of values into increasing or decreasing order. It is possible to reproduce the block of instructions that constitute a subroutine

at every place where it is needed in the program. However, to save space, only one copy of this block is placed in the memory, and any program that requires the use of the subroutine simply branches to its starting location. When a program branches to a subroutine we say that it is *calling* the subroutine. The instruction thatperforms this branch operation is named a Call instruction.

After a subroutine has been executed, the calling program must resume execution, continuing immediately after the instruction that called the subroutine. The subroutine is said to *return* to the program that called it, and it does so by executing a Return instruction. Since the subroutine may be called from different places in a calling program, provision must be made for returning to the appropriate location. The location where the calling program resumes execution is the location pointed to by the updated program counter (PC) while the Call instruction is being executed. Hence, the contents of the PC must be saved by the Call instruction to enable correct return to the calling program. The way in which a computer makes it possible to call and return from subroutines is referred to as its *subroutine linkage* method. The simplest subroutine linkage method is to save the return address in a specific location, which may be a register dedicated to this function. Such a register is called the *link register*. When the subroutine completes its task, the Return instruction returns to the calling program by branching indirectly through the link register.

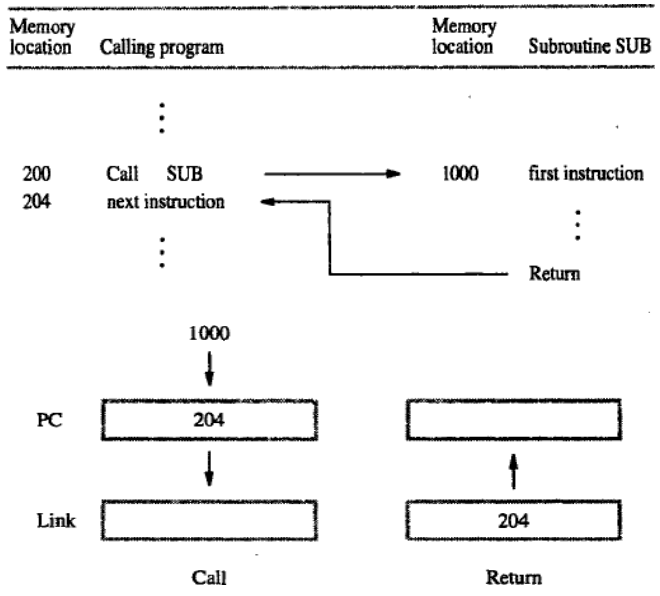
The Call instruction is just a special branch instruction that performs the following operations:

- Store the contents of the PC in the link register
- Branch to the target address specified by the Call instruction

The Return instruction is a special branch instruction that performs the operation

- Branch to the address contained in the link register

The below figure illustrates how the PC and the link register are affected by the Call and Return Instructions.



Subroutine Nesting and the Processor Stack

A common programming practice, called *subroutine nesting*, is to have one subroutine call another. In this case, the return address of the second call is also stored in the link register, overwriting its previous contents.

Hence, it is essential to save the contents of the link register in some other location before calling another subroutine. Otherwise, the return address of the first subroutine will be lost. Subroutine nesting can be carried out to any depth. Eventually, the last subroutine called completes its computations and returns to the subroutine that called it. The return address needed for this first return is the last one generated in the nested call sequence. That is, return addresses are generated and used in a last-in–first-out order. This suggests that the return addresses associated with subroutine calls should be pushed onto the processor stack.

Parameter Passing

When calling a subroutine, a program must provide to the subroutine the parameters, that is, the operands or their addresses, to be used in the computation. Later, the subroutine returns other parameters, which are the results of the computation. This exchange of information between a calling program and a subroutine is referred to as *parameter passing*. Parameter passing may be accomplished in several ways. The parameters may be placed in registers or in memory locations, where they can be accessed by the subroutine. Alternatively, the parameters may be placed on the processor stack. Passing parameters through processor registers is straightforward and efficient.

Calling program

```
            Move        N,R1          R1 serves as a counter.
            Move        #NUM1,R2      R2 points to the list.
            Call        LISTADD       Call subroutine.
            Move        R0,SUM        Save result.
            :
```

Subroutine

```
LISTADD     Clear       R0            Initialize sum to 0.
LOOP        Add         (R2)+,R0      Add entry from list.
            Decrement   R1
            Branch>0    LOOP
            Return                    Return to calling program.
```

The above figure shows how the program for adding a list of numbers can be implemented as a subroutine, LISTADD, with the parameters passed through registers.

- The size of the list, *n*, contained in memory location N, and the address, NUM1, of the first number, are passed through registers R2 and R4.
- The sum computed by the subroutine is passed back to the calling program through register R3. The first four instructions in the above figure constitute the relevant part of the calling program.
- The first two instructions load *n* and NUM1 into This instruction also saves the return address (i.e., the address of the Store instruction in the calling program) in the link register.
- The subroutine computes the sum and places it in R3. After the Return instruction is executed by the subroutine, the sum in R3 is stored in memory location SUM by the calling program.
- In addition to registers R2, R3, and R4, which are used for parameter passing, the subroutine also uses R5. Since R5 may be used in the calling program, its contents are saved by pushing them onto the processor stack upon entry to the subroutine and restored before returning to the calling program.

If many parameters are involved, there may not be enough general-purpose registers available for passing them to the subroutine. The processor stack provides a convenient and flexible mechanism for passing an arbitrary number of parameters.

LISTADD, which uses the processor stack for parameter passing.

- The address of the first number in the list and the number of entries are pushed onto the processor stack pointed to by register SP.
- The subroutine is then called. The computed sum is placed on the stack before the return to the calling program.

Assume that before the subroutine is called, the top of the stack is at level 1.

- The calling program pushes the address NUM1 and the value *n* onto the stack and calls subroutine LISTADD. The top of the stack is now at level 2.
- The subroutine uses four registers while it is being executed. Since these registers may contain valid data that belong to the calling program, their contents should be saved at the beginning of the subroutine by pushing them onto the stack. The top of the stack is now at level 3.
- The subroutine accesses the parameters *n* and NUM1 from the stack using indexed addressing with offset values relative to the new top of the stack (level 3). Note that it does not change the stack pointer because valid data items are still at the top of the stack.
- The value *n* is loaded into R2 as the initial value of the count and the address NUM1 is loaded into R4, which is used as a pointer to scan the list entries.
- At the end of the computation, register R3 contains the sum. Before the subroutine returns to the calling program, the contents of R3 are inserted into the stack, replacing the parameter NUM1, which is no longer needed.
- Then the contents of the four registers used by the subroutine are restored from the stack. Also, the stack pointer is incremented to point to the top of the stack that existed when the subroutine was called, namely the parameter *n* at level 2.
- After the subroutine returns, the calling program stores the result in location SUM and lowers the top of the stack to its original level by incrementing the SP by 8.

Observe that for subroutine LISTADD in Figure, we did not use a pair of instructions
Subtract SP, SP, #4
Store R*j*, (SP)
to push the contents of each register on the stack. Since we have to save four registers, this would require eight instructions. We needed only five instructions by adjusting SP immediately to point to the top of stack that will be in effect once all four registers are saved. Then, we used the Index mode to store the contents of registers. We used the same optimization when restoring the registers before returning from the subroutine.

10. Write short notes on Stacks & Queues

Stack

A *stack* is a list of data elements, usually words, with the accessing restriction that elements can be added or removed at one end of the list only. This end is called the top of the stack, and the other end is called the bottom. The structure is sometimes referred to as a *pushdown* stack. Imagine a pile of trays in a cafeteria; customers pick up new trays from the top of the pile, and clean trays are added to the pile by placing them onto the top of the pile.

Another descriptive phrase, *last-in–first-out* (LIFO) stack, is also used to describe this type of storage mechanism; the last data item placed on the stack is the first one removed when retrieval begins. The terms *push* and *pop* are used to describe placing a new item on the stack and removing the top item from the stack, respectively.

In modern computers, a stack is implemented by using a portion of the main memory for this purpose. One processor register, called the *stack pointer* (SP), is used to point to a particular stack structure called the *processor stack.*

Data can be stored in a stack with successive elements occupying successive memory locations. Assume that the first element is placed in location BOTTOM, and when new elements are pushed onto the stack, they are placed in successively lower address locations.

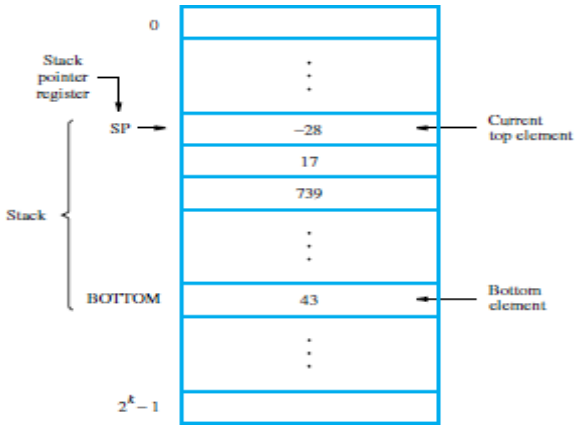


Fig. A stack of words in memory

The stack contains numerical values, with 43 at the bottom and −28 at the top. The stack pointer, SP, is used to keep track of the address of the element of the stack that is at the top at any given time. If we assume a byte-addressable memory with a 32-bit word length, the push operation can be implemented as

Subtract SP, SP, #4
Store R*j*, (SP)

where the Subtract instruction subtracts 4 from the contents of SP and places the result in SP. Assuming that the new item to be pushed on the stack is in processor register R*j*, the Store instruction will place this value on the stack. These two instructions copy the word from R*j* onto the top of the stack, decrementing the stack pointer by 4 before the store (push) operation. The pop operation can be implemented as Load R*j*, (SP)
Add SP, SP, #4
These two instructions load (pop) the top value from the stack into register R*j* and then increment the stack pointer by 4 so that it points to the new top element.

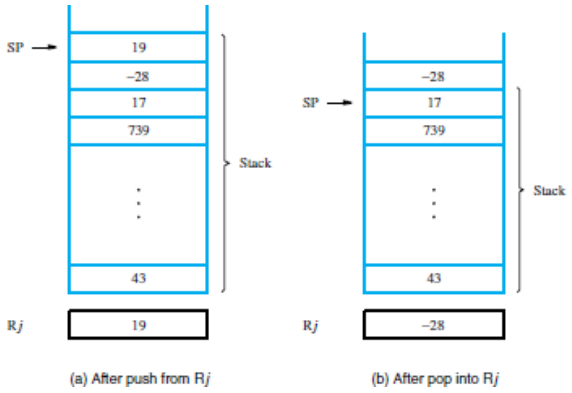


Fig. Effect of stack operations

Queue

Another useful data structure that is similar to stack is called a queue. Data are stores and retrieved from a queue on a first-in-first-out (FIFO) basis. Thus new data are added at the back (high-address end) and retrieved from the front (low-address end) of the queue.

There are two important differences in how a stack and a queue are implemented.

- One end of the stack is fixed (bottom), while other end rises and falls as data are pushed and popped.
- Both ends of the queue move to higher addresses as data are added at the back and removed from the front. So two pointers are needed to keep track of the two ends of the queue.
- Another difference between a stack and a queue is that, without further control, a queue would continuously move through the memory of a computer in the direction of higher addresses.
- One way to limit the queue to a fixed region in memory is to use a circular buffer.

11. Explain with example the byte sorting program

Consider a program for sorting a list of bytes stored in memory into ascending alphabetic order. Assume that the lists consists of n bytes, not necessarily distinct, and that each byte contains the ASCII code for a character from the set of letters A through Z. when an ASCII character is stored in a byte location, it is customary to set the most significant bit position to 0. Using this code, we can sort a list of characters alphabetically by sorting their codes in increasing numerical order, considering them positive numbers.

Let the list be stored in memory locations LIST, through LIST+n-1, and let n be a 32-bit value stored at address N. We sort using a straight-selection sort algorithm. The largest number is found and placed at the end of the list in location LIST+n-1. Then the largest number in the remaining sublist of n-1 numbers is placed at the end of the sublist in location LIST+n-2. The procedure is repeated until the list is sorted.

```
for   (j = n−1; j > 0; j = j − 1)
      { for ( k = j−1; k >= 0; k = k − 1 )
          { if   (LIST[k] > LIST[j])
                 {  TEMP = LIST[k];
                    LIST[k] = LIST[j];
                    LIST[j] = TEMP;
                 }
          }
      . }
```

(a) C-language program for sorting

	Move	#LIST,R0	Load LIST into base register R0.
	Move	N,R1	Initialize outer loop index
	Subtract	#1,R1	register R1 to $j = n - 1$.
OUTER	Move	R1,R2	Initialize inner loop index
	Subtract	#1,R1	register R2 to $k = j - 1$.
	MoveByte	(R0,R1),R3	Load LIST(j) into R3, which holds current maximum in sublist.
INNER	CompareByte	R3,(R0,R2)	If LIST(k) $\leq$ [R3],
	Branch≤0	NEXT	do not exchange.
	MoveByte	(R0,R2),R4	Otherwise, exchange LIST(k)
	MoveByte	R3,(R0,R2)	with LIST(j) and load
	MoveByte	R4,(R0,R1)	new maximum into R3.
	MoveByte	R4,R3	Register R4 serves as TEMP.
NEXT	Decrement	R2	Decrement index registers R2 and
	Branch≥0	INNER	R1, which also serve as
	Decrement	R1	as loop counters, and branch
	Branch>0	OUTER	back if loops not finished.

(b) Assembly language program for sorting

Control flow is handled differently in the two programs for purposes of efficiency in the assembly language program.

Using the if-then control statement in the C-language program causes the three-line then clause to exchange LIST(k) and LIST(j) if LIST(k)>LIST(j).

In the assembly language program, a branch is taken around the four-instruction exchange code if LIST(k)<=LIST(j).

If the machine instruction set allows a move operation from one memory location directly to another memory location, then the four-instruction exchange code in the inner loop can be replaced by the three-instruction sequence.

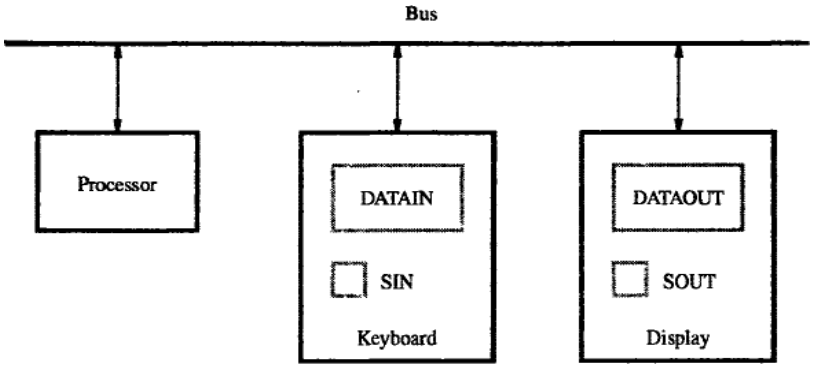
```
MoveByte   (R0,R2),(R0,R1)
MoveByte   R3,(R0,R2)
MoveByte   (R0,R1),R3
```

The above program works correctly only if the list has at least two elements because the check for loop termination is done at the end of each loop. Hence, there is at least one pass through the loop, regardless of the value of n.

12. Explain Basic input /output operation

Consider a task that reads in character input from a keyboard and produces character output on a display screen. A simple way of performing such I/O reads is to use a method known as program controlled I/O.The rate of data transfer from the keyboard to a computer is limited by the typing speed of the user, which is unlikely to exceed a few characters per second. The rate of output transfers from the computer to the display is much higher. It is determined by the rate at which characters can be transmitted over the link between the computer and the display device, typically several thousand characters per second. however, this is still much slower than the speed of a processor that can execute many millions of instructions per second. The difference between the processor and I/O devices creates the need for mechanisms to synchronize the transfer of data between them.

A solution to this problem is ,the processor sends the first character and then waits for a signal from the display that the character has been received. It then sends the second character and so on. Input is sent from the keyboard in a similar way ,the processor waits for a signal indicating that a character key has been stuck and that its code is available in some buffer register associated with the keyboard. Then the processor proceeds to read that code. The keyboard and the display are separate devices as shown below



The action of striking a key on the keyboard does not automatically cause the corresponding character to be displayed on the screen. One block of instructions in the I/O program transfers the character into the processor, and another associated block of instruction causes the character to be displayed.

Consider the problem of moving a character code from the keyboard to the processor, striking a key stores the corresponding character cede in a 8-bit buffer register associated with the keyboard.

This register called DATAIN .To inform the processor that a valid character is in DATAIN ,a status control flag ,SIN ,is set to 1.A program monitors SIN , and when SIN is set to 1,the processor reads the contents of DATAIN.When the character is transferred to the processor, SIN is automatically cleared to 0.If a second character is entered at the keyboard, SIN is again set to 1 and the process repeats.

An analogous process takes place when characters are transferred from the processor to the display. A buffer register, DATAOUT ,and a status control flag, SOUT ,are used for this transfer .When SOUT equals 1,the display is ready to receive a character. Under program control, the processor monitors SOUT ,and when SOUT is set to 1,the processor transfers a character code to DATAOUT..The transfer of a character to DATAOUT clears SOUT to 0;when the display device is ready to receive a second character, OUT is again set to 1.The buffer registers DATAIN and DATAOUT and the status flags SIN and SOUT are part of circuitry commonly known as a device interface. The circuitry for each device is connected to the processor via a bus is shown above.