Prepared by Dr K Badari Nath, Associate Professor,
Computer Science & Engineering, RV College Of Engineering

FCSD – Foundations Of Computer System Design

Unit 4 : Structure of Computers and Instruction Set Architecture

Basic Structure of Computers: Functional Units, Basic Operational Concepts, Performance – Technology and Parallelism. **Instruction Set Architecture**: Memory Locations and Addresses, Memory Operations, Instructions and Instruction Sequencing, Addressing Modes, Assembly Language- Assembler Directives, Assembly and Execution of Programs. Stacks, Subroutines- Subroutine Nesting and the Processor Stack, Parameter Passing, The Stack Frame.

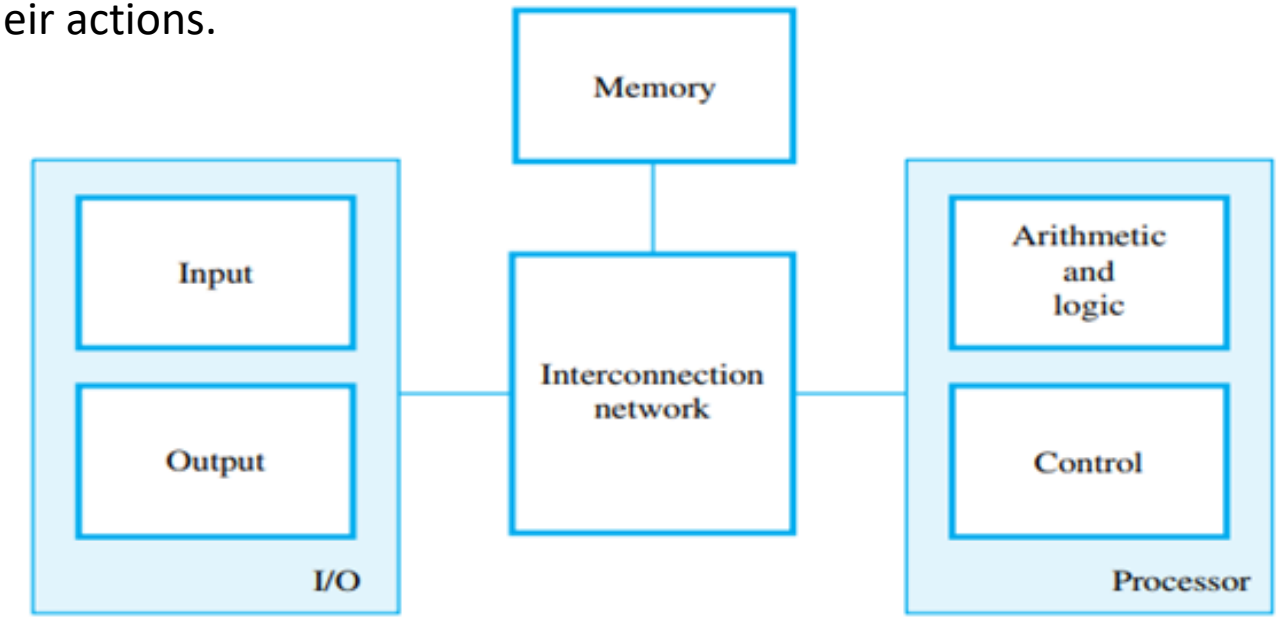
- CIE/SEE Question solutions are there from SLIDE 44

- Reference: Chapter 1 – Basic Structure of Computers,
  Chapter 2 - Instruction Set Architecture
  (Book:"Computer Organization and Embedded Systems" by Carl Hamacher)

*https://www.youtube.com/@drkbadarinath4636
(for lecture videos)

Basic Structure of Computers: Functional Units…

A computer consists of five functionally independent main parts: input, memory, arithmetic and logic, output, and control units,. The input unit accepts coded information from human operators using devices such as keyboards, or from other computers over digital communication lines. The information received is stored in the computer's memory, either for later use or to be processed immediately by the arithmetic and logic unit. The processing steps are specified by a program that is also 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. An interconnection network provides the means for the functional units to exchange information and coordinate their actions.



The operation of a computer can be summarized as follows:

- 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 in the computer are directed by the control unit

Information handled by a computer is categorize as either instructions or data. Instructions, or machine instructions, are explicit commands that, i) Govern the transfer of information within a computer as well as between the computer and its I/O devices. ii) Specify the arithmetic and logic operations to be performed.

A program is a list of instructions which performs a task. Programs are stored in the memory. The processor fetches the program instructions from the memory, one after another, and performs the desired operations. The computer is controlled by the stored program, Data are numbers and characters that are used as operands by the instructions.

Data are also stored in the memory. The instructions and data handled by a computer must be encoded in a suitable format. Most present-day hardware employs digital circuits that have only two stable states. Each instruction, number, or character is encoded as a string of binary digits called bits, each having one of two possible values, 0 or 1

The operation of a computer can be summarized as follows:

- 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 in the computer are directed by the control unit

Basic Structure of Computers: Basic Operational Concepts….

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

A typical instruction might be :

Load R2, LOC ,

This instruction reads the contents of a memory location whose address is represented symbolically by the label LOC and loads them into processor register R2. The original contents of location LOC are preserved, whereas those of register R2 are overwritten. Execution of this instruction requires several steps. First, the instruction is fetched from the memory into the processor. Next, the operation to be performed is determined by the control unit. The operand at LOC is then fetched from the memory into the processor. Finally, the operand is stored in register R2. After operands have been loaded from memory into processor registers, arithmetic or logic operations can be performed on them.

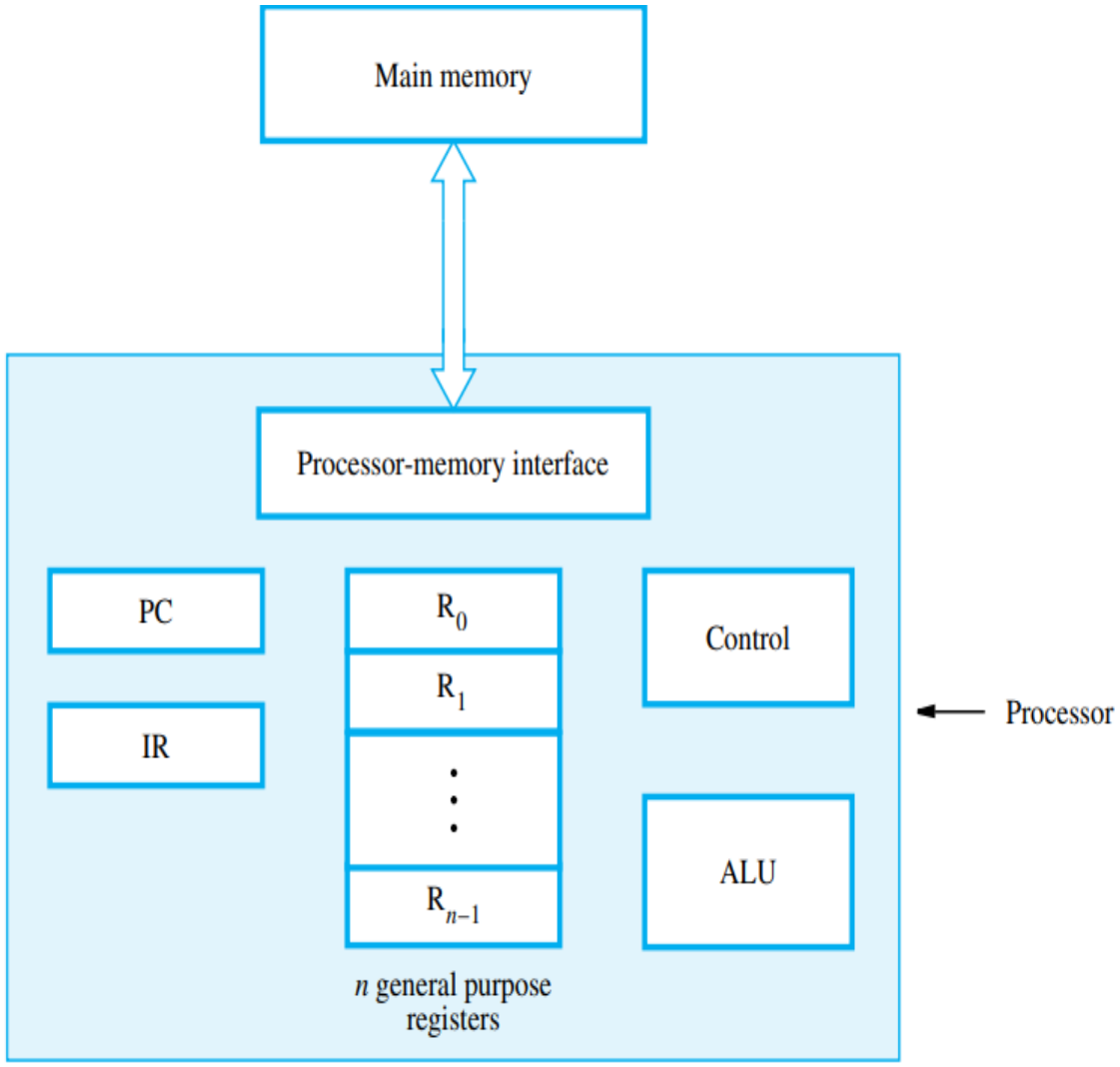
For example, the instruction : Add R4, R2, R3

adds the contents of registers R2 and R3, then places their sum into register R4. The operands in R2 and R3 are not altered, but the previous value in R4 is overwritten by the sum. After completing the desired operations, the results are in processor registers.

They can be transferred to the memory using instructions such as

Store R4, LOC

This instruction copies the operand in register R4 to memory location LOC. The original contents of location LOC are overwritten, but those of R4 are preserved. For Load and Store instructions, transfers between the memory and the processor are initiated by sending the address of the desired memory location to the memory unit and asserting the appropriate control signals. The data are then transferred to or from the memory.



Connection between the processor and the main memory.

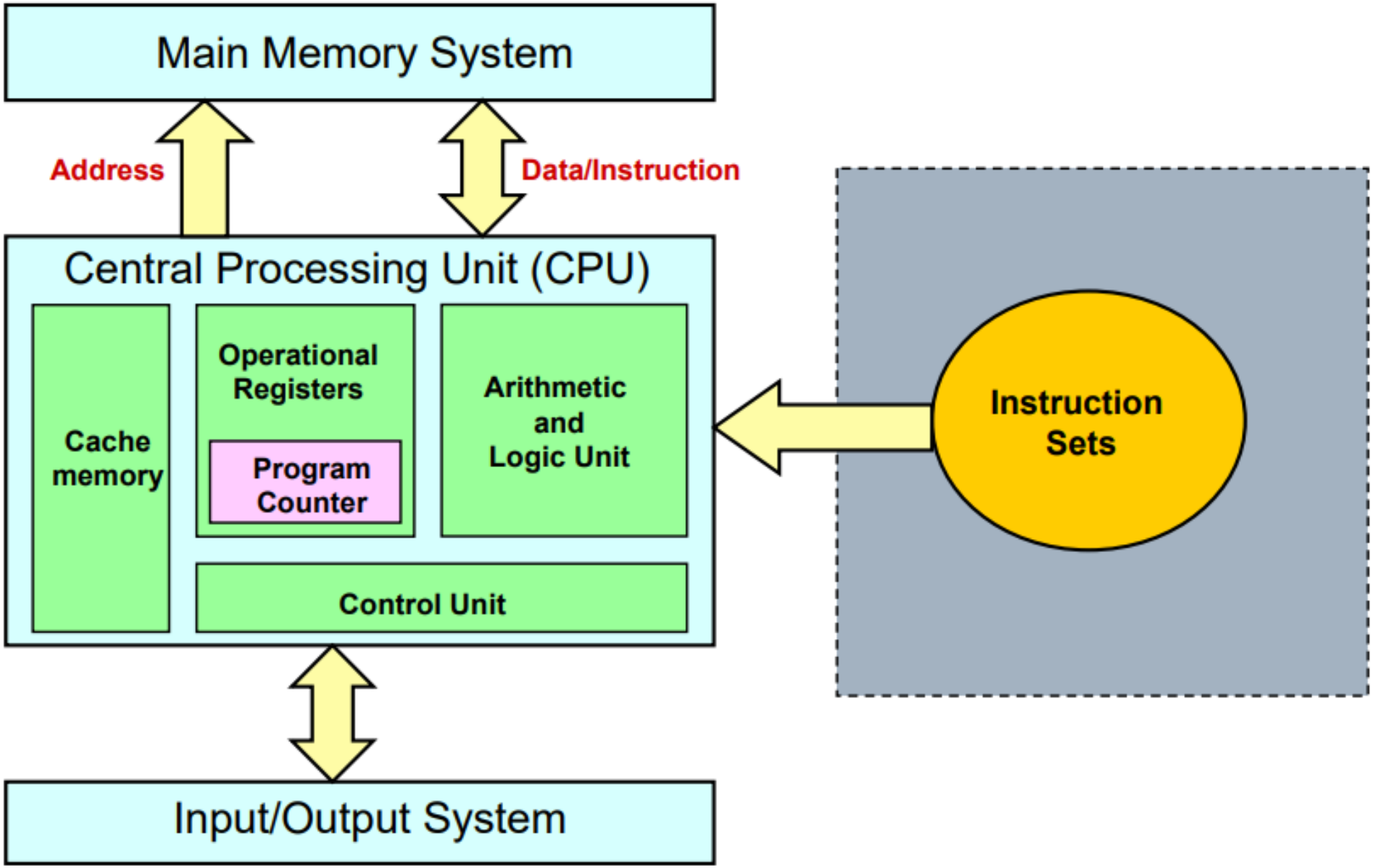
Basic Operational Concepts…

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

The instruction register (IR) holds the instruction that is currently being executed. Its output is available to the control circuits, which generate the timing signals that control the various processing elements involved in executing the instruction.

The program counter (PC) is another specialized register. It contains the memory address of the next instruction to be fetched and executed. During the execution of an instruction, the contents of the PC are updated to correspond to the address of the next instruction to be executed. It is customary to say that the PC points to the next instruction that is to be fetched from the memory.

In addition to the IR and PC, Figure shows general-purpose registers R0 through Rn−1, often called processor registers. They serve a variety of functions, including holding operands that have been loaded from the memory for processing


Main Memory System
Address
Data/Instruction
Central Processing Unit (CPU)
Cache memory
Operational Registers
Program Counter
Arithmetic and Logic Unit
Control Unit
Instruction Sets
Input/Output System

Basic Structure of Computers: Performance – Technology and Parallelism

- A key design objective of a computer system is to achieve the best possible performance at the lowest possible cost.
  - Price/performance ratio is a common measure of success.
- Performance of a processor depends on:
  - How fast machine instructions can be brought into the processor for execution.
  - How fast the instructions can be executed.
  - The speed with which a computer executes programs is affected by the design of its instruction set, its hardware and its software, including the operating system, and the technology in which the hardware is implemented. Because programs are usually written in a high-level language, performance is also affected by the compiler that translates programs into machine language.
  - The technology of Very Large Scale Integration (VLSI) that is used to fabricate the electronic circuits for a processor on a single chip is a critical factor in the speed of execution of machine instructions. The speed of switching between the 0 and 1 states in logic circuits is largely determined by the size of the transistors that implement the circuits. Smaller transistors switch faster. Advances in fabrication technology over several decades have reduced transistor sizes dramatically. This has two advantages: instructions can be executed faster, and more transistors can be placed on a chip, leading to more logic functionality and more memory storage capacity

Performance can be increased by performing a number of operations in parallel. Parallelism can be implemented on many different levels.

- Instruction-level Parallelism (Pipelining) The simplest way to execute a sequence of instructions in a processor is to complete all steps of the current instruction before starting the steps of the next instruction. If we overlap the execution of the steps of successive instructions, total execution time will be reduced. For example, the next instruction could be fetched from memory at the same time that an arithmetic operation is being performed on the register operands of the current instruction. This form of parallelism is called pipelining.
- Multicore Processors Multiple processing units can be fabricated on a single chip. In technical literature, the term core is used for each of these processors. The term processor is then used for the complete chip. Hence, we have the terminology dual-core, quad-core, and octo-core processors for chips that have two, four, and eight cores, respectively
- Multiprocessors Computer systems may contain many processors, each possibly containing multiple cores. Such systems are called multiprocessors. These systems either execute a number of different application tasks in parallel, or they execute subtasks of a single large task in parallel. All processors usually have access to all of the memory in such systems, shared-memory.

Basic Performance Equation

Let T be the processor time required to execute a program that has been prepared in some high level language. The compiler generates a machine language object program, that corresponds to the source program

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

N – Actual Number of Machine Language Instructions,
S – Average number of basic steps needed to execute one machine instruction
R – Clock rate, No. of clock cycles per second (refer next slide for details)

Computer Performance Using Bench Mark Programs

(more practical way to compute performance, then using performance eq.)

SPEC rating = (Running time on the reference computer) /
(Running time on the computer under test)

Benchmark -> Standard program used to measure performance of a computer. The performance measure is the time it takes a computer to execute a given benchmark.

A non-profit organization called System Performance Evaluation Corporation (SPEC) selects and publishes representative application programs for different application domains, together with test results for many commercially available computers. The programs selected range from game playing, compiler, and database applications to numerically intensive programs in astrophysics and quantum chemistry. In each case, the program is compiled for the computer under test, and the running time on real computer is measured. The same program is also compiled and run on one computer selected as reference. For SPEC2000, the reference computer is an Ultra-SPARC10 workstation with a 300-MHz UltraSPARC-Iii processor.

Computer Performance Using Bench Mark Programs

Thus a SPEC rating of 50 means that the computer under test is 50 times as fast as the the UltraSPARC10 for this particular benchmark.

The test is repeated for all the programs in the SPEC suite, and the geometric mean of the results is computed. Let SPECi be the rating for program i in the suite. The overall SPEC rating for the computer is given by
SPEC rating =

$$\text{SPEC rating} = \frac{\text{running time on reference computer}}{\text{running time on computer undertest}}$$

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

https://www.spec.org/

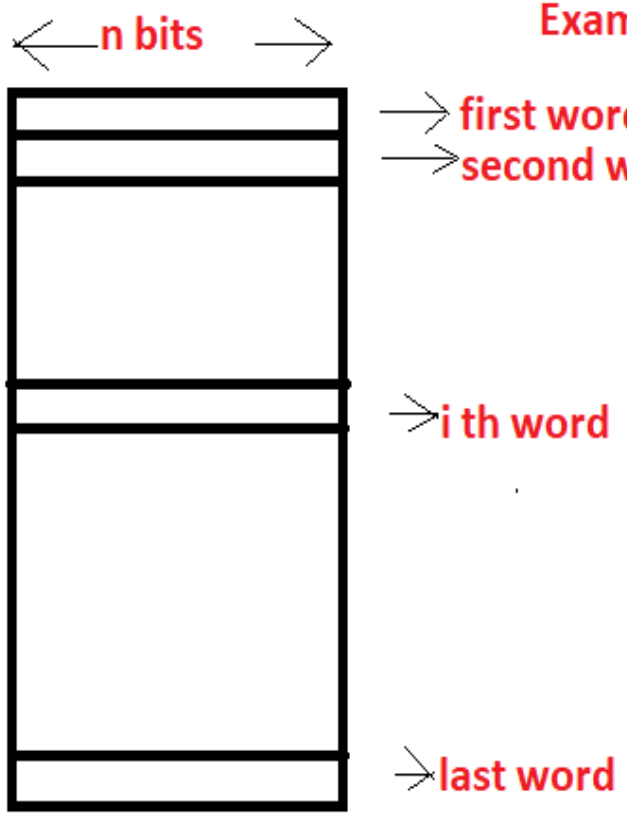
Instruction Set Architecture

Memory Locations & Addresses

Instructions (in binary form) and Data (numbers and characters in binary or BCD or ASCII) are stored in the memory.

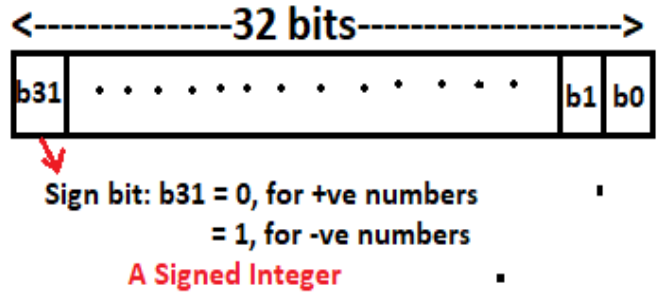
Memory is organized so that a group of ‘n’ bits can be stored or retrieved in a single, basic operation. Each group of n bits is referred to as a word of information, and n is called the word length. Popular computer word lengths are 16/32/64/128 bits.

Ex. 32 bit word length computer can store a 32 bit signed number in a single word or four ASCII characters. Machine instructions may required one to many words.



Memory Words

Examples of Encoded information in 32 bit words



A Signed Integer

Four ASCII Characters

Byte Addressability

Memory is made of 1bit memory cells, organized into bytes and words. Unique address is required to access them. Most of the modern computers allocate one unique address for set of 8 bits in the memory. Each memory location, representing one byte has given one unique address.(this is called as byte addressability). Address is numbered in binary from 0 to N-1, where N is the total number of memory locations (bytes), is based on the address lines provided by the processor. If M is the number of address lines, then

2^M = N, in bytes

(where M is no. of address lines, of the processor)

Examples:

2^{10} = 1K 1 kilobytes

2^{20} = 1M 1 megabytes

2^{30} = 1G 1 gigabytes

2^{40} = 1T 1 terabytes

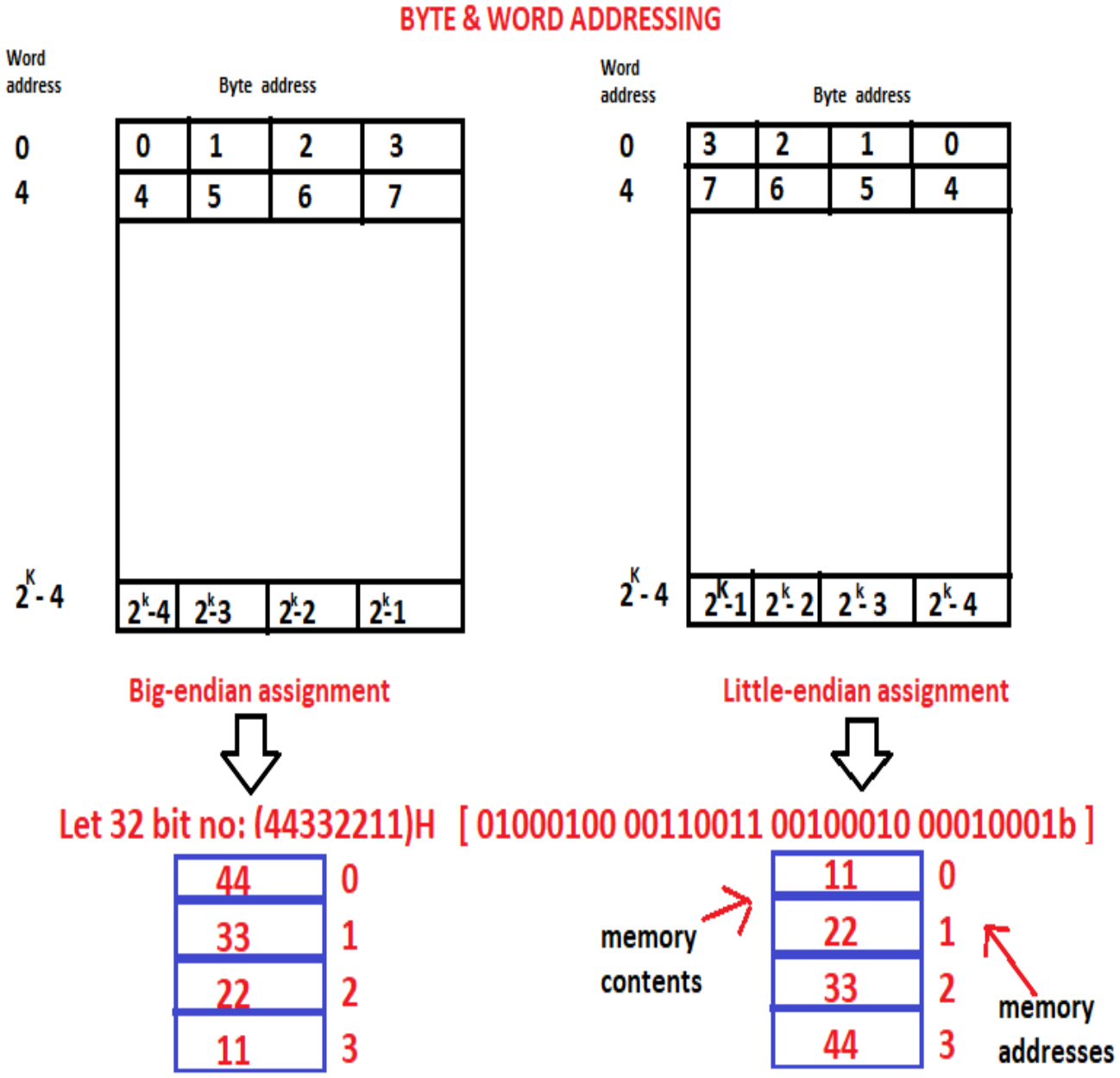
2^{16} = 64 K bytes 2^{24} = 16 M bytes 2^{32} = 4G bytes

Note: In 64 bit windows, the theoretical amount of virtual address space is 2^64 bytes (16 exabytes), but only a small portion of the 16 exabyte range is actually used. 1exabyte = 1billion gigabyte. Windows XP x64 is currently limited to

BIG-ENDIAN,LITTLE-ENDIAN assignments

There are two ways that byte addresses can be assigned across words.
In big-endian format, lower byte addresses are used for the more significant bytes (the leftmost bytes) of the word. Ex: Motorolla processors
In little-endian format, lower byte addresses are used for the less significant bytes (the rightmost bytes) of the word. Ex: Intel Processors
(ARM supports both)

In both cases, byte addresses 0,4,8,…, are taken as the addresses of successive words in the memory and those are used when performing memory read/write operations.



Word Alignment

In the case of a 32 bit word length, natural word boundaries occur at addresses 0,4,8,…. We say that 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 a multiple of the number of bytes in a word.

Memory Operation

- Random access memories must have two basic operations
  - Write: writes a data into the specified location
  - Read: reads the data stored in the specified location
- In machine language program, the two basic operations usually are called
  - Store: write operation
  - Load: read operation
- The Load operation transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged
- The Store operation transfers an item of information from the processor to a specific memory location, destroying the former contents of that location

Instructions

- A computer must have instructions capable of performing four 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
- Register transfer notation
  - The contents of a location are denoted by placing square brackets around the name of the location
  - For example, R1←[LOC] means that the contents of memory location LOC are transferred into processor register R1
  - As another example, R3←[R1]+[R2] means that adds the contents of registers R1 and R2, and then places their sum into register R3

Assembly Language Notation

- Types of instructions
  - Zero-address instruction
  - One-address instruction
  - Two-address instruction
  - Three-address instruction
- Zero-address instruction
  - For example, store operands in a structure called a **pushdown stack**
- One-address instruction
  - Instruction form: **Operation Destination**
  - For example, **Add A**: add the contents of memory location A to the contents of the accumulator register and place the sum back into the accumulator
  - As another example, **Load A**: copies the contents of memory location A into the accumulator

Assembly Language Notation

- Two-address instruction
  - Instruction form: **Operation Source, Destination**
  - For example, **Add A, B**: performs the operation B←[A]+[B]. When the sum is calculated, the result is sent to the memory and stored in location B
  - As another example, **Move B, C**: performs the operation C←[B], leaving the contents of location B unchanged
- Three-address instruction
  - Instruction form: **Operation Source1, Source2, Destination**
  - For example, **Add A, B, C**: adds A and B, and the result is sent to the memory and stored in location C
  - If k bits are needed to specify the memory address of each operand, the encoded form of the above instruction must contain 3k bits for addressing purposes in addition to the bits needed to denote the Add operation.

Condition Codes

- The processor keeps track of information about the results of various operations for use by subsequent conditional branch instructions. This is accomplished by recoding required information in individual bits, often called *condition code flags*
- Four commonly used flags are
  - N (negative): set to 1 if the results is negative; 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 a carry-out results from the operation; otherwise, cleared to 0

Addressing Modes

- Programmers use data structures to represent the data used in computations. These include lists, linked lists, array, queues, and so on
- A high-level language enables the programmer to use constants, local and global variables, pointers, and arrays
- When translating a high-level language program into assembly language, the compiler must be able to implement these constructs using the facilities in the instruction set of the computer
- The different ways in which the location of an operand is specified in an instruction are referred to as *addressing modes*

Name	Example	Assembler syntax	Addressing function
	EXAMPLES		
Immediate	MOV #20,R0 move the value 20, to R0	#Value	Operand=Value
Register	MOV R1,R0 move the contents of R1 to R0	Ri	EA=Ri
Absolute (Direct)	MOV SUM,R2 add the number stored in memory location "SUM" TO R2 and store the answer in R2	LOC	EA=LOC
Indirect	ADD (R2),R0 add the number stored in memory location, whose address is stored in R2, with R0 and store ans in R0. R2 acts as pointer	(Ri)	EA=[Ri]
		(LOC)	EA=[LOC]
Index	MOV 4(R2),R0 move the contents of memory location, whose address is computed by addng 4 and contents of R2, to the register R0	X(Ri)	EA=[Ri]+X
Base with index	ADD (R0,R1),R2 add the contents of memory location, whose address is computed by adding the contents of R0 and R1, with R2 and store the answer in R2	(Ri, Rj)	EA=[Ri]+[Rj]
Base with index and offset	ADD 10(R0,R1),R2	X(Ri, Rj)	EA=[Ri]+[Rj]+X
Relative	BNE LOOP (BRANCH NOT EQUAL) program jumps/branches to the location, LOOP, whose address is computed by adding the contents of PC with the offset value i.e distance of LOOP relative to PC	X(PC)	EA=[PC]+X
Autoincrement	ADD (R2)+,R0 same as Indirect, but here pointer is incremented, after the operation	(Ri)+	EA=[Ri]; Increment Ri
Autodecrement	ADD -(R2),R0 same as Indirect, but here pointer is decremented, before the operation	-(Ri)	Decrement Ri; EA=[Ri]

- Register mode: the operand is the contents of a processor register; the name (address) of the register is given in the instruction
  - For example, **Add Ri, Rj** (adds the contents of Ri and Rj and the result is stored in Rj)
- Absolute mode: the operand is in a memory location; the address of this location is given explicitly in the instruction. (In some assembly languages, this mode is called Direct)
  - For example, **Move LOC, R2** (moves the content of the memory with address LOC to the register R2)
  - The Absolute mode can represent global variables in a program. For example, a declaration such as Integer A, B;
- Immediate mode: the operand is given explicitly in the instruction

Indirection and Pointers

- Indirect mode: the effective address of the operand is the contents of a register or memory location whose address appears in the instruction
- Indirection is denoted by placing the name of the register or the memory address given in the instruction in parentheses
- The register or memory location that con of an operand is called a pointer

Indexing and Arrays

- Index mode: the effective address of the operand is generated by adding a constant value to the contents of a register
  - The register used may be either a special register provided for this purpose, or, more commonly, it may be any one of a set of general-purpose registers in the processor. It is referred to as an index register
  - The index mode is useful in dealing with lists and arrays
  - We denote the Index mode symbolically as X(Ri), where X denotes the constant value contained in the instruction and Ri is the name of the register involved. The effective address of the operand is given by EA=X+(Ri). The contents of the index register are not changed in the process of generating the effective address

Variations of Indexed Addressing Mode

- A second register may be used to contain the offset X, in which case we can write the Index mode as (Ri,Rj)
  - The effective address is the sum of the contents of registers Ri and Rj
  - The second register is usually called the base register
  - This mode implements a two-dimensional array
- Another version of the Index mode use two registers plus a constant, which can be denoted as X(Ri,Rj)
  - The effective address is the sum of the constant X and the contents of registers Ri and Rj
  - This mode implements a three-dimensional array

Additional Modes

- Autoincrement mode: the effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list
  - The Autoincrement mode is denoted as (Ri)+
- Autodecrement mode: the contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand
  - The Autodecrement mode is denoted as –(Ri)

Assembly Language

- A complete set of symbolic names and rules for their use constitute a programming language, generally referred to as an assembly language
- Programs written in an assembly language can be automatically translated into a sequence of machine instructions by a program called an *assembler*
- When the assembler program is executed, it reads the user program, analyzes it, and then generates the desired machine language program
- The user program in its original alphanumeric text format is called a *source program*, and the assembled machine language program is called an *object program*

Assembler Directives

In addition to providing a mechanism for representing instructions in a program, assembly language allows the programmer to specify other information needed to translate the source program into the object program. Such statements, called assembler directives (or commands), are used by the assembler while it translates a source program into an object program. Some popular directives are provided below.

EQU : used to name the constant values, like #define statements in C

```
N  EQU   10
```

Wherever N is referred in the program, it will replaced by 10, before assembly program is converted to object program

ORIGIN: which tells the assembler program, where in the memory to place the instructions that follow.

```
ORG 100
```

;It specifies that the instructions of the object program are to be loaded in the memory starting at address 100

RESERVE: used to reserve the required space, in terms of bytes

```
SUM:  RESERVE  4
```

A 4-byte space for the sum variable is reserved by means of the assembler directive RESERVE

DATAWORD: used to declare a variable, which can hold a word, with the initial value

```
N:   DATAWORD   150
```

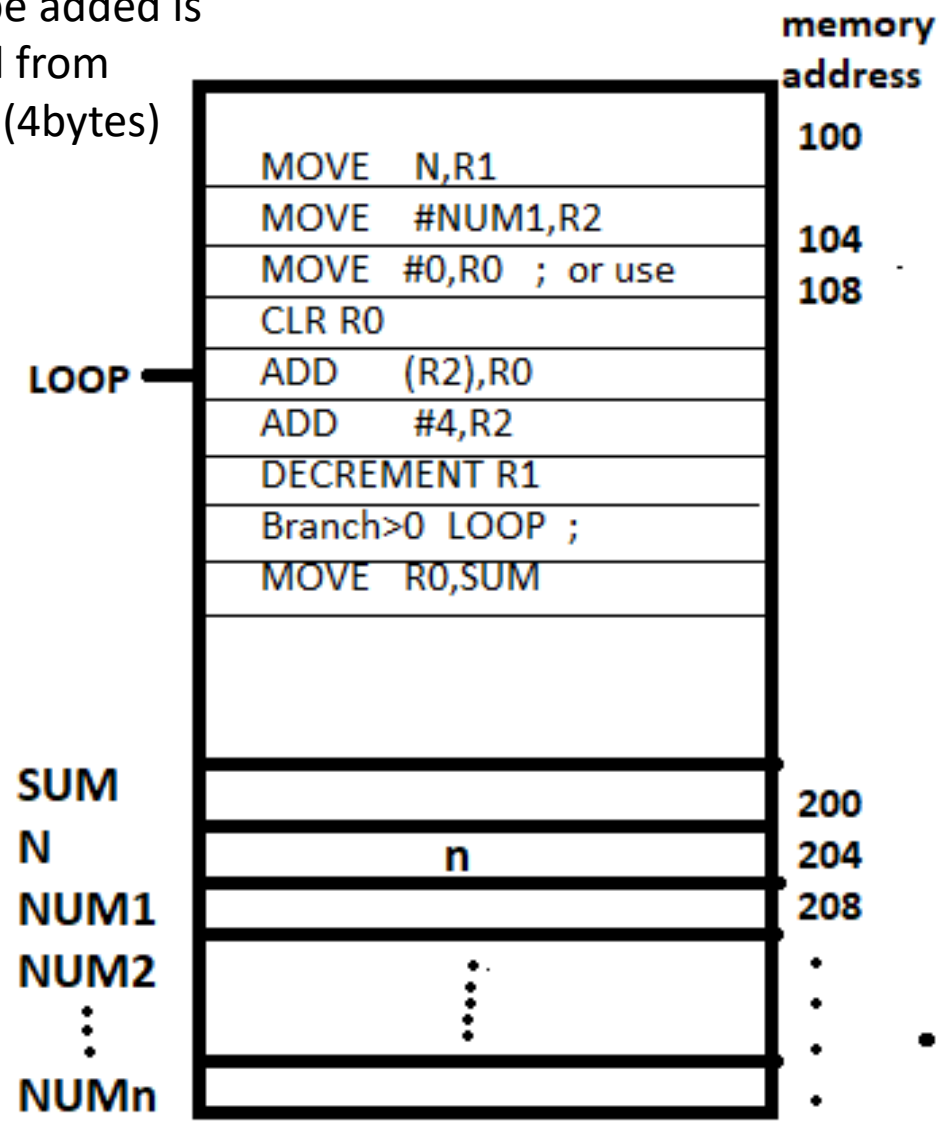
Value 150 is stored in the variable N, of size one word, and name of the variable is N

END: end of the assembly program

Write ALP program to add N numbers stored in the memory, indicate the memory layout

Program is stored from the memory address, 100.
Store the answer at memory 200, the total numbers (here 150 is taken) to be added is stored at the memory N, i.e at the address 204, and the numbers are stored from location Num1 onwards (from address, 208…). Here, numbers are of 32 bit, (4bytes)

```
ORIGIN 100

MOVE    N,R1
MOVE    #NUM1,R2
MOVE   #0,R0   ;  or use CLR R0
ADD     (R2),R0
ADD      #4,R2
DECREMENT R1
Branch>0  LOOP  ; or use  BGT
LOOP
MOVE   R0,SUM

ORIGIN 200
SUM:   RESERVE    4
   N:    DATAWORD 150
NUM1: RESERVE 600
END
```

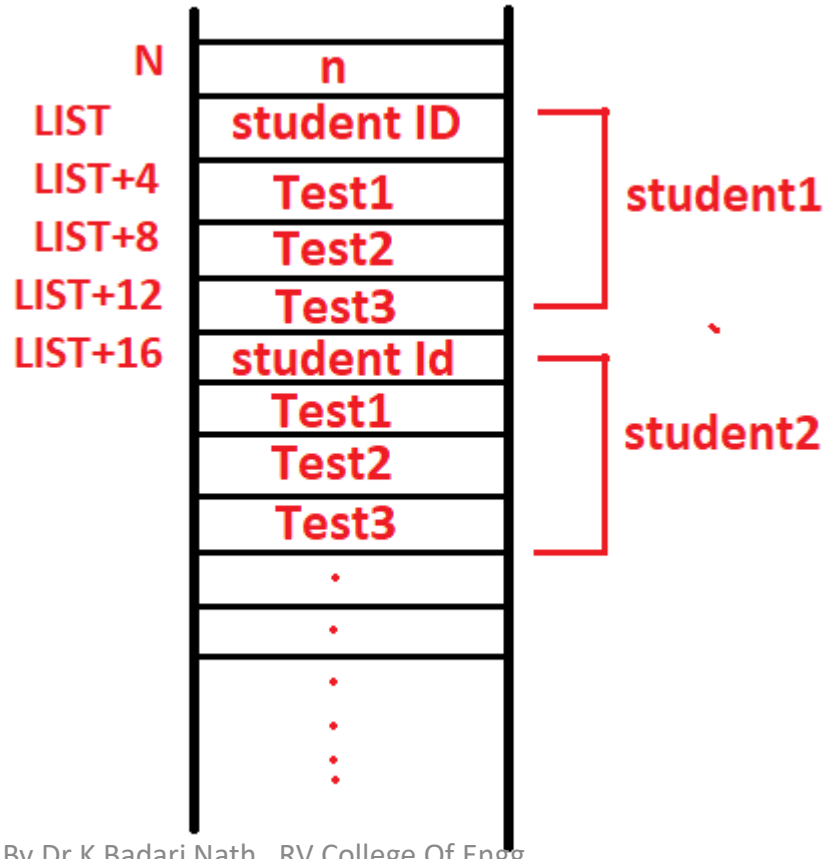


Write ALP program to compute the sum of all test scores obtained on each of the tests and store in memory for N students, for three tests indicate the memory layout

```
ORIGIN 100
MOVE #LIST, R0
CLEAR  R1
CLEAR  R2
CLEAR  R3
MOV    N,R4
ADD    4(R0),R1
ADD    8(R0),R2
ADD   12(R0),R3
ADD    #16,R0
DECREMENT R4
BRANCH>0 LOOP  ; or BGT LOOP
MOVE R1, SUM1
MOVE R2, SUM2
MOVE R3, SUM3

ORIGIN 200
SUM1:   RESERVE   4
SUM2:   RESERVE   4
SUM3:   RESERVE   4
     N:  DATAWORD 10
  LIST:  RESERVE 160
END
```

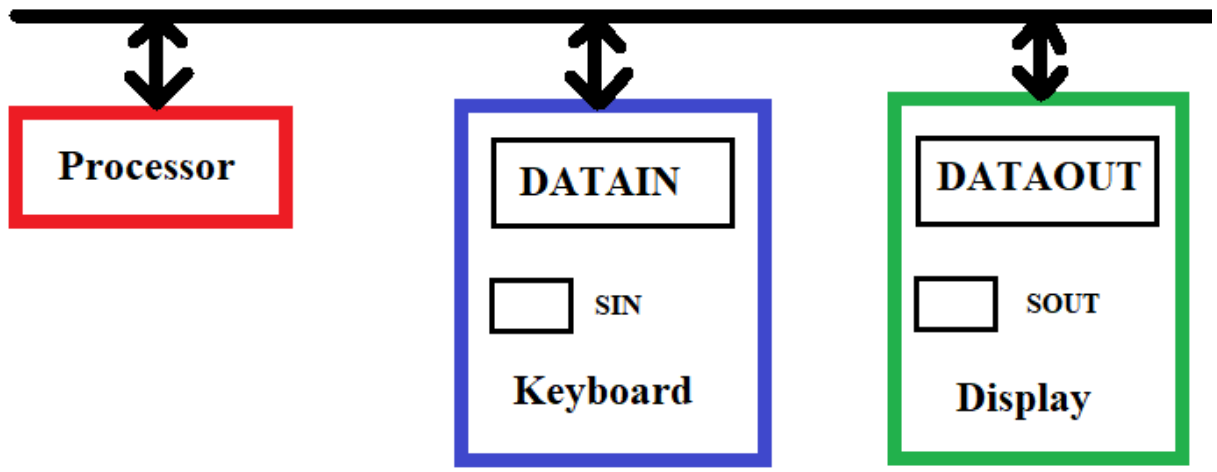
(4 words for one record, 4x4=16 bytes, 16x10=160,for 10 studens)

Consider a simple example involving a list of test scores for students taking a given course. Assume that the list of scores beginning at location LIST, is structured as four word memory block for each student. Each record consists of the students identification number (ID), followed by the scores the student earned on three tests. There are n students in the class, and the value n is stored in location N immediately in front of the list. Assume memory is byte addressable and the word length is 32 bits



ALP – Basic Input/Output Operations

Consider a task that reads in character input from a keyboard and produces a character output on a display screen using program controlled I/O. The difference in speed between the processor and I/O devices creates the need for mechanism to synchronize the transfer of data between them. For that following buffer registers and flags are used in the device circuitry.
DATAIN – an 8bit buffer register associated with keyboard, SIN – Keyboard status flag, indicating key is present in buffer, (bit 3)
DATAOUT – buffer register, for printer, SOUT – status flag, indicating printer is ready to receive the character,(bit 3)

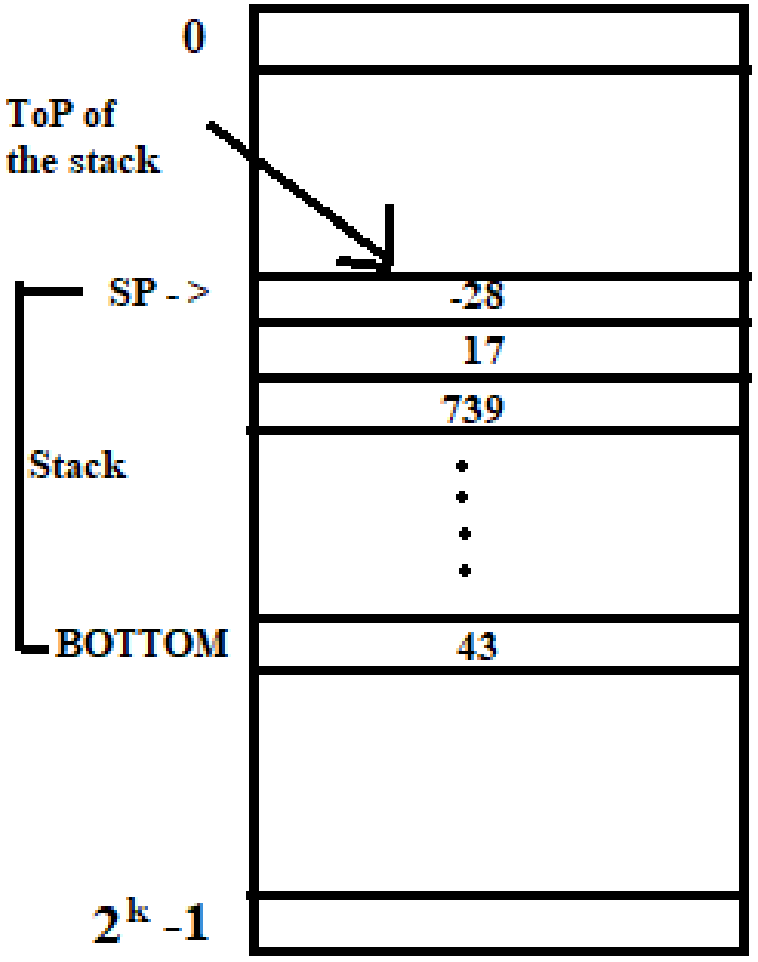


BUS CONNECTION FOR PROCESSOR, KEYBOARD, AND DISPLAY

```
;Initialize the pointer R0, to point to memory, to store characters
        Move        #LOC, R0
;wait for character entered in Kb buffer DATAIN. SIN==1,
;then read data, on read SIN becomes 0
READ  TestBit       #3,INSTATUS
        BRANCH=0    READ
;store the data in the memory
        MoveByte    DATAIN, (R0)
;write the data to display, if SOUT==1,
;on writing SOUT becomes 0, when display is ready it becomes 1
ECHO  TestBit       #3, OUTSTATUS
        Brach = 0   ECHO
        MoveByte    (R0), DATAOUT
;stop doing this when character recieved in NEWLINE character
        Compare     #CR, (R0)+
        Branch != 0 READ
```

Note: Program controlled I/O requires continuous involvement of the Processor in I/O activities, this can be improved using Interrupt driven I/O , DMA transfers.

ALP : Implementation of Stacks



PUSH OPERATION..

```
Compare     #2000, SP
Branh > 0   EMPTY_ERROR
Subtract    #4, SP
Move        NEWITEM, (SP)
```

```
; OR, autodecrement is supported
 Move     NEWITEM, -(SP)
```

POP OPERATION...

```
Compare          #1500 , SP
Branch <= 0      FULL_ERROR
Move             (SP), ITEM
Add              #4, SP
```

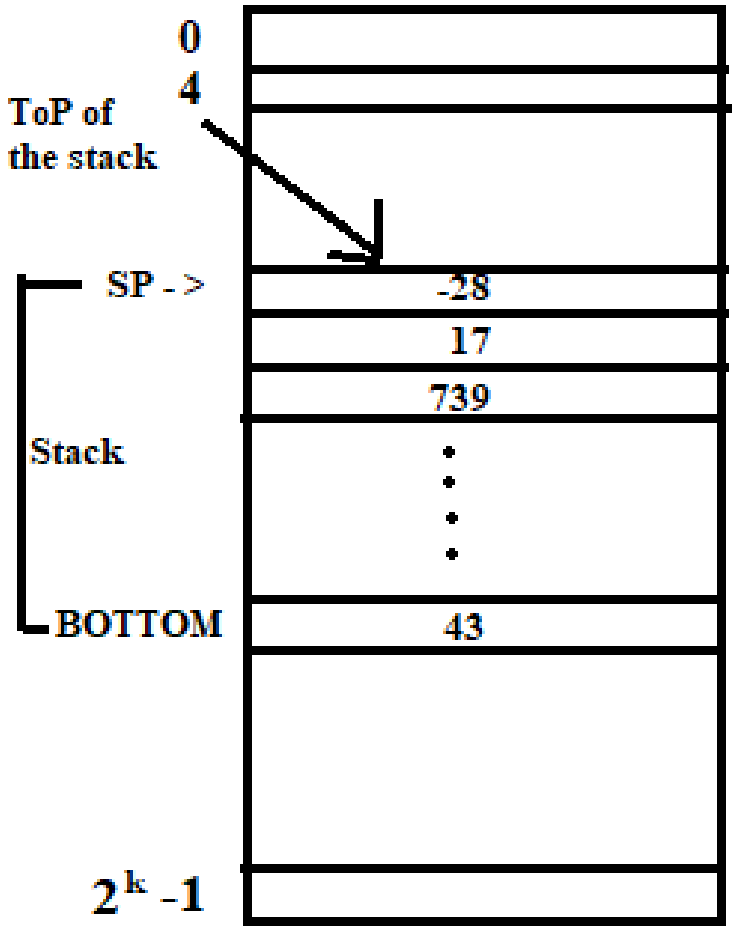
```
;OR, if autoincrement is supported
Move      (SP)+, ITEM
```

A Stock of Words in Memory (stack grows - Higher to Lower addresses

A stack is a list of words/bytes, operates with the LIFO principle, elements can be added/removed using PUSH and POP operations.

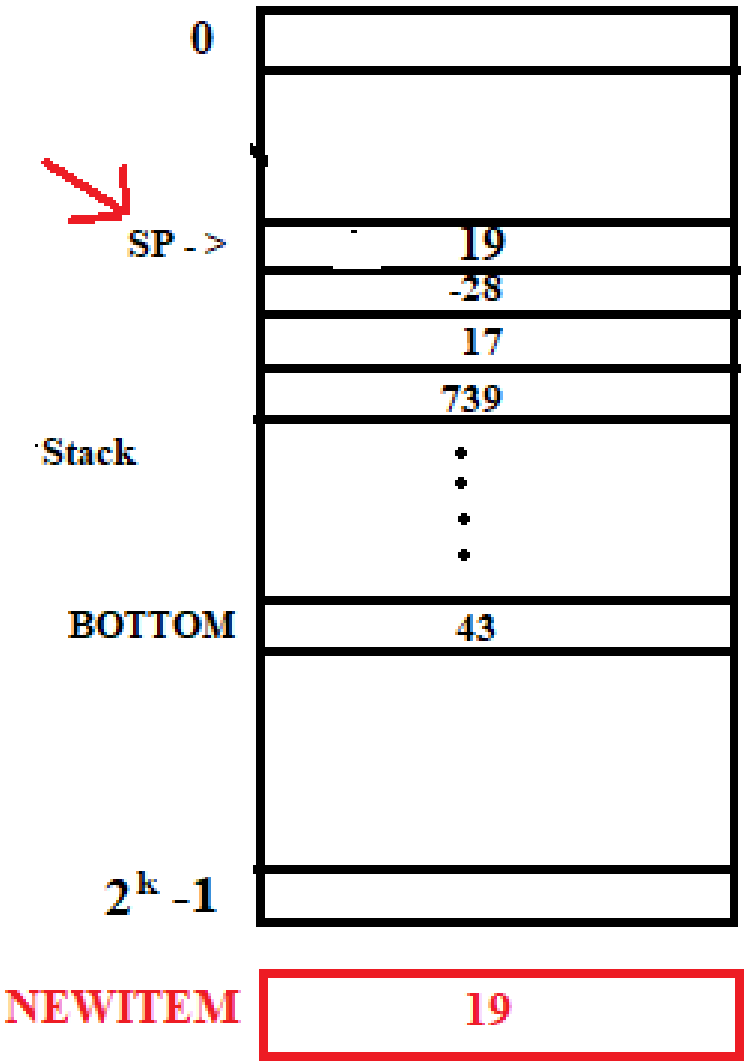
Shown in the figure, the stack of word data items in the memory (assuming byte addressable memory with 32 bit word length), with fixed stack size of (BOTTOM :2000 TO 1500), with data element 43 at the BOTTOM and -28 at the top, With top of the stack is pointed by special register SP –Stack Pointer.

It is common practice, stack grows in the direction of decreasing memory addresses, PUSH decrements SP by 4, POP increments SP by 4.



STACK, initial values
ex: SP = 1000

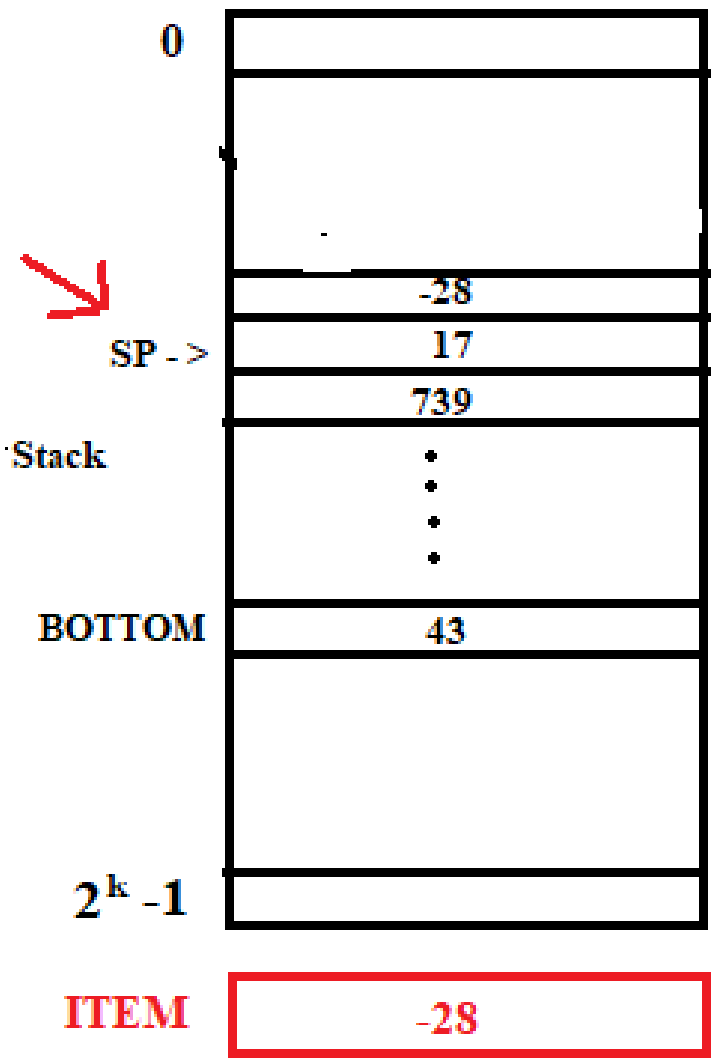
MOVE NEWITEM, -(SP)



NEWITEM 19

after PUSH ing NEWITEM
SP = 996

MOVE (SP)+, ITEM



ITEM -28

after POP ing into ITEM
SP = 1004

ALP : Implementation of QUEUEs

Data are stored in and retrieved from a queue on FIFO basis. Common practice is, queue grows in the increasing addresses in memory, new data are added at the back (high addresses end) and data are retrieved from the front (low-addresses end). In case of stack, one end of the stack is fixed (the bottom), while the other end rises and falls as data are pushed or popped, a single pointer is sufficient to point to the top of the stack. On the other hand, both ends of a 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. No special registers are provided for this, unlike SP. As queue would continuously move through the memory of a computer in the direction of higher addresses, to limit the queue to a fixed region in memory, is to use a circular buffer.

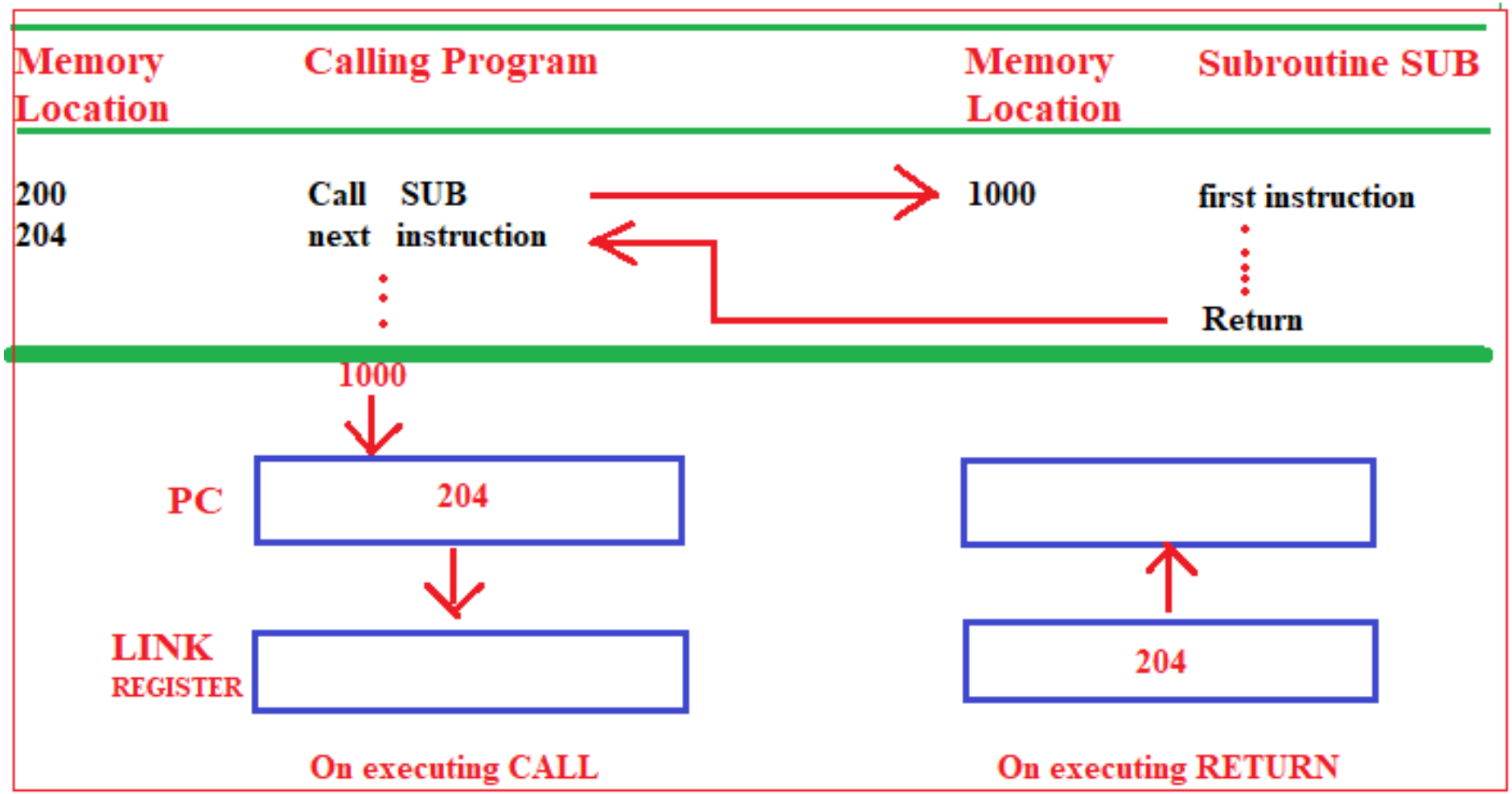
Let us assume that memory addresses from BEGINNING to END are assigned to the queue. The first entry in the queue is entered into location BEGINNING, and successive entries are appended to the queue by entering them at successively higher addresses. By the time the back of the queue reaches END, space will have been created at the beginning if some items have been removed from the queue. Hence the back pointer is reset to the value BEGINNING and the process continues. Care must be taken, to detect when the region assigned to the data structure is either completely full or completely empty.

ALP: SUBROUTINES

Subroutines are used when a particular sub task is to be executed many times on different values, at different times in the program.

CALL instruction is used, for calling the subroutine, the calling program must resume execution continuing immediately after the instruction that called the subroutine. RETURN instruction is executed by the subroutine, to return to the calling program.

The way in which a computer makes it possible to CALL and RETURN from subroutine is referred to as its subroutine linkage method. The simplest subroutine method is to save the return address in s specific location, which may be a register dedicated to this function, called as LINK REGISTER. When CALL is executed, PC (it points to the next instruction) contents are stored in LINK REGISTER, when RETURN is executed, PC will be re-loaded with LINK REGISTER contents.



1. Store PC to Link register
2. Branch to target address specified in the instruction

Branch to the address contained in LinkReg

SUB ROUTINE LINKAGE USING LINK REGISTER

Subroutine Nesting…

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, destroying its previous contents. Hence, it is essential to save the contents of 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 the 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 is done using processor stack.

Many Processors do this automatically as one of the operations performed by the call instruction. The Stack pointer points to a stack called the processor stack. The call instruction pushes the contents of PC on to the processor stack and loads the subroutine address into the PC. The return instruction pops the return address from the processor stack into the PC.

ALP: Subroutines – Parameter Passing

The exchange of information between a calling program and a subroutine is referred as parameter passing. The calling program may place the parameters using,

1. Using Registers (straightforward/easy and efficient)
2. Using Memory Locations
3. Using the Processor Stack (flexible, can handle large number of parameters)

Parameter Passing by Value and by Reference : Note the nature of the two parameter, NUM1 and n, passed to the subroutine shown in the figure. The purpose of the subroutine is to add a list of numbers. Instead of passing the actual list entries, the calling program passes the address of the first number in the list (passing by reference). The second parameter is passed by value, that is, the actual number of entries, n , is passed to the subroutine.

```
; passing the operands using registers R1,R2
; receives the answer using register R0
            Move    N,R1
            Move    #NUM1, R2
            Call    LISTADD ; callibg a subroutine
            Move    R0,SUM


;Subroutine, to add the list of N numbers
LISTADD     Clear   R0
            Add     (R2)+,R0
            Decrement  R1
            Branch > 0  LOOP
            Return  ; return to the main program
```

ALP:Subroutines Parameter Using Stack

Assume, before the stack is used for parameter passing, it is at LEVEL 1

In MainProgram :

The parameters (N and NUM1) are pushed to stack, SP is decremented by 8

The subroutine LISTADD is called, hence return address is saved on stack,

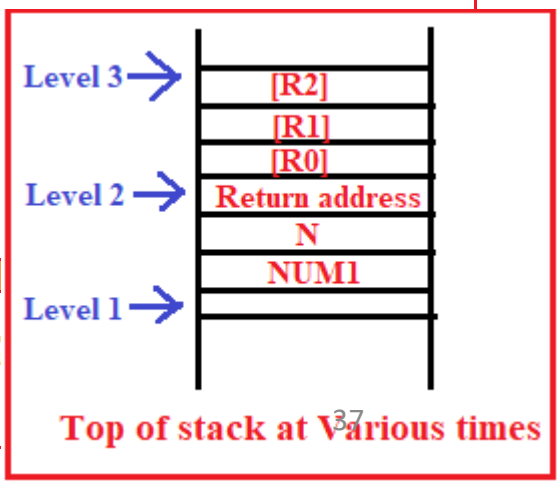
now SP is further decremented by 4, and now stack is at LEVEL 2.

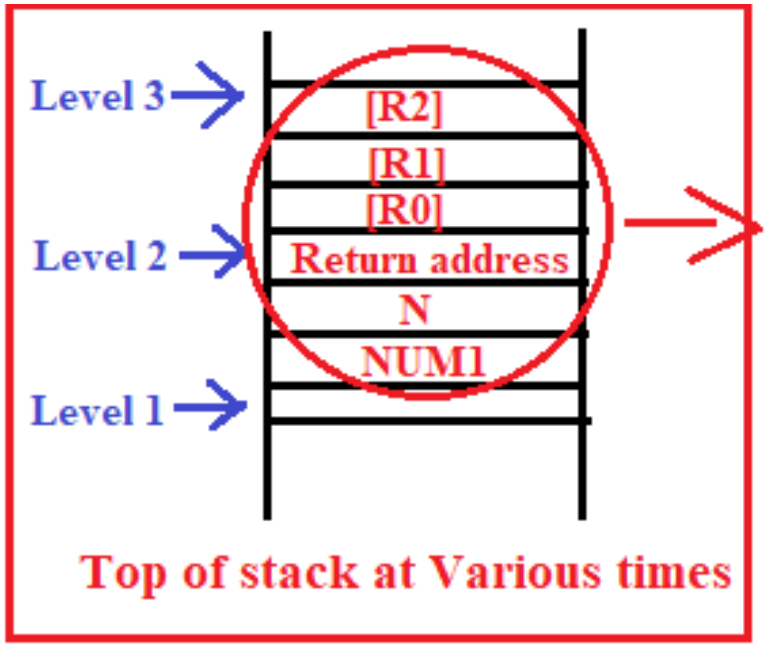
In SubRoutine :

The subroutine uses three registers. Since these registers may contain valid data that belong to the calling program, their contents should be saved by pushing them onto the stack. MoveMultiple, is used to store the contents of registers R0,R1,R2 on stack. Now, stack is at LEVEL 3.

Subroutine accesses the parameters N and NUM1 from the stack using indexed addressing (hence SP value will not change). Before the subroutine returns to the calling program, the answer stored in R0 are placed on the stack replacing the parameter NUM1, which is no longer needed. Then, the contents of Registers R0-R2used by subroutine are restored from the stack. Now the top item on the stack is the return address 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.

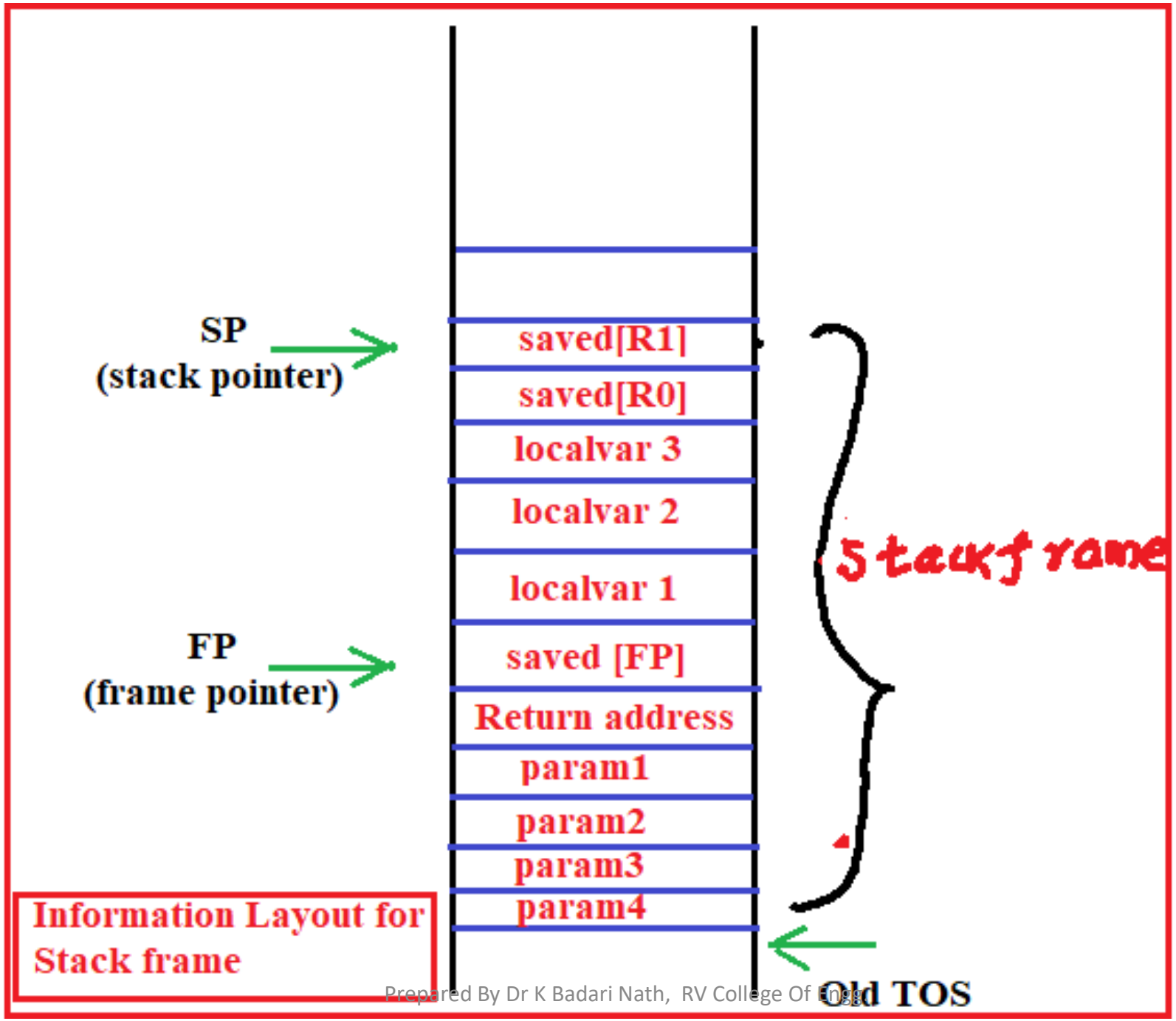
```
; main Program  ( top of stack is at level 1)
        Move    #NUM1, -(SP)
        Move    N, - (SP)
        Call    LISTADD; after calling, top of stack is at level 2)
;restore top of the stack (level 1)
        Add     #8, SP
                .
                .
                .
                .

; sub routine, to add list of numbers, using parameter passing using stack
LISTADD MoveMultiple  R0-R2, - (SP) ; (level 3)
        ; above instn. saves the prev. contents, before usage
        Move        16 (SP), R1   ; Initialize R1 to n
        Move        20 (SP), R2   ; Initialize the ponter R2 to list
        Clear       R0
        Add         (R2)+,R0
        Decrement R1
        Branch > 0 LOOP
        Move        R0, 20 (SP)    ; put result on t
        MoveMultiple  (SP)+,R0-R2   ; restore re
        Return
```





During execution of the subroutine, six locations at the top of the stack contain entries that are needed by the subroutine. These locations constitute a private workspace for the subroutine, created at the time the subroutine is entered and freed up when the subroutine returns control to the calling program. Such space is called a stack frame. If the subroutine requires more space for local memory variables, they can also be allocated on the stack.

Stack frame is created with subroutine call and freed up during return.


SP
(stack pointer)
saved[R1]
saved[R0]
localvar 3
localvar 2
localvar 1
FP
(frame pointer)
saved [FP]
Return address
param1
param2
param3
param4
Old TOS
Stackframe
Information Layout for
Stack frame

ALP:Subroutine – stack frame

With reference to the figure, SP& FP are manipulated as follows, when the stack frame is built,used and dismantled, when subroutine called.

-> Assume SP points to the old top-of-stock (TOS) element.

-> Before the subroutine is called, the calling program pushes the four parameters on to the stack.

->The call instruction is executed, resulting in the return address being pushed on to the stack. Now, SP points to this return address, and the first instruction of the subroutine is about to be executed.

->This is the point at which the frame pointer FP is set to contain the proper memory address. Since, FP is usually a general-purpose register, it may contain information of use to the calling program. Therefore, its contents are saved by pushing them onto the stack. Since SP now points to this position, its contents are copied into FP. Hence the first two instructions executed in the subroutine are:

```
Move    FP, -(SP)
Move    SP, FP
```

Now both SP,FP point to the saved FP contents.

ALP:Subroutine – stack frame

->Space for the three local variables is now allocated on the stack by executing the instruction

Subtract #12, SP

->Finally, the contents of processor registers R0 and R1 are saved by pushing them on to the stack, as indicated in the figure.

->The subroutine now executes its task. When the task is completed, the subroutine pops the saved values of R1 and R0 back into those registers, Removes the local variables from the stack frame by executing the instruction

Add #12,SP

-> And pops the saved old value of FP back into FP. At this point, SP points to the return address, so the return instruction can be executed, transferring control back to the calling program.

->The calling program is responsible for removing the parameters from the stack frame, some of which may be results passed by the subroutine. The stack pointer now points to the old TOS, where it was, before stack frame was created.

main program

```
2000        Move    PARAM2, -(SP)
2004        Move    PARAM1, -(SP)
2008        Call    SUB1
2012        Move    (SP), RESULT
2016        Add     #8, SP
2020        next instructionn
                    :
```

first subroutine

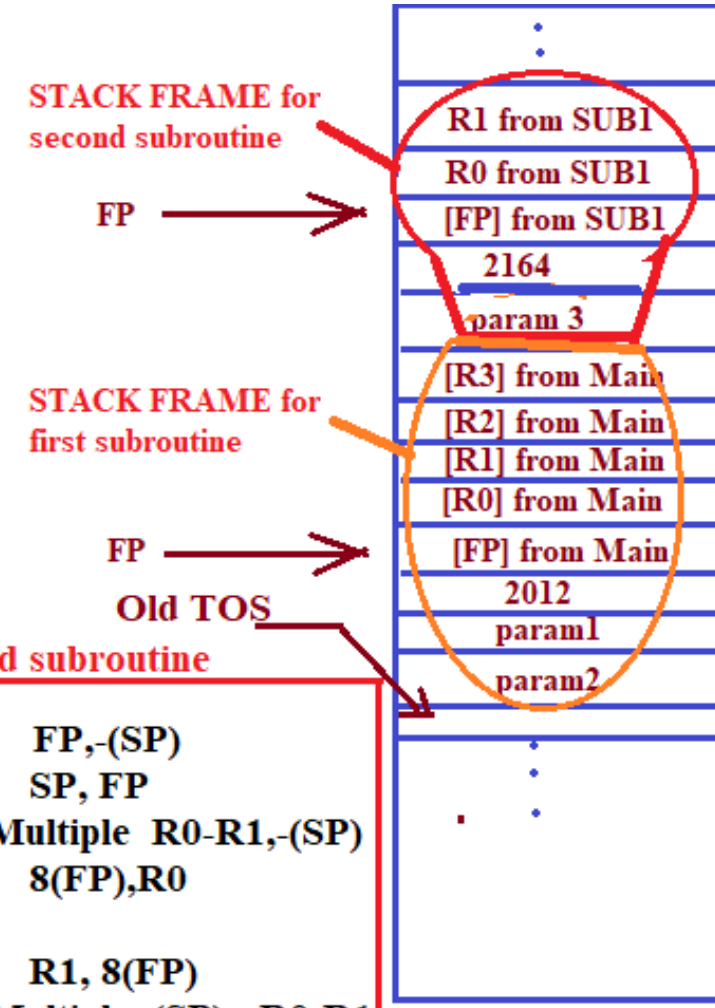
```
2100  SUB1  Move    FP, -(SP)
            Move    FP, SP
            MoveMultiple RO-R3,-(SP)
            Move    8(FP),R0
            Move    12(FP), R1
                    :
            Move    PARAM3,-(SP)
2160        Call    SUB2
2164        Move    (SP)+,R2

            Move    R3,8(FP)
            MoveMultiple  (SP)+, R0-R3
            Move    (SP)+,FP
            Return
```

second subroutine

```
3000 SUB2   Move    FP,-(SP)
            Move    SP, FP
            MoveMultiple  R0-R1,-(SP)
            Move    8(FP),R0
                    :
            Move    R1, 8(FP)
            MoveMultiple  (SP)+,R0-R1
            Move    (SP)+, FP
            Return
```

->place the two parameters on stack, SP <- SP -8
-> SUB1 Called, return address 2012, pushed to stack, SP<- SP-4
-> after returned from SUB1, answer is received in place of param1(is on top of the stack), hence
Move (SP),RESULT
->Add 8 to SP, to resore stack level (i.e the space allocated for param1,param2)

Push registers R0 to R3 used for computation(SP<-SP-12)
Access the param1,param2 passed from main program using FP and copy it to R0,R1 for use
Now push param3(SP<-SP-4), call SUB2: return adds 2164 pushed to stack (SP<-SP-4)



ALP:Subroutine – Nesting…

The stack is the proper data structure for holding return addresses when subroutines are nested. Stack frames for nested subroutines build up on the processor stack, as they are called.

As shown in the figure, main program calling a first subroutine SUB1, which then calls a second subroutine SUB2. All Parameters are passed on the stack. The flow of execution is as follows:

The main program pushes two parameters param2 and param1 onto the stack, then calls SUB1.

The first subroutine is responsible for computing a single answer and passing it back to the main program on the stack. During the course of its computations, SUB1 calls the second subroutine SUB2, in order to perform some subtask.

SUB1 passes a single parameter param3 to SUB2 and gets a result passed back to it. After SUB2 executes its RETURN instruction, this result is stored in register R2 by SUB1. SUB1 then continues its computations and eventually passes the required answer back to the main program on the stack. When SUB1 executes its return to the main program, the main program stores this answer in memory location RESULT and continues execution..

Q1) Derive the basic Performance equation and discuss in detail any two ways to enhance the performance of the computing system.

T = Processor Time
N = Actual No. of instructions
S = Average No. of basic steps to execute one instruction
R = Clock rate

The ways to enhance the performance of the system : Pipelining & Super scalar execution(explain in detail).

- $T = \frac{N*S}{R}$

The Performance can also improved by clock rate/instruction set/compiler…(explain in brief)

Q2) A program contains 100 instructions, out of that 50% of instructions requires 4 clock cycles and remaining requires 3 clock cycles for execution. Find the total time required to execute the program running on a 1 MHz machine.

50% of 100 instructions = 50 x 4 clks = 200

50% of 100 instructions = 50 x 3 clks = 150

total = 350 clks

1 CLK time = 1/1MHz = 1micro second

Total time = 350 clks x 1 micro second = 350 microsecond

Q3)A program contains 1000 instructions, out of that 25% of instructions requires 4 clock cycles, 40% instructions requires 5 clock cycles and remaining requires 3 clock cycles for execution. Find the total time required to execute the program running on a 1GHz machine.

25% - 250x 4cks -> 1000

40% - 400 x 5cks -> 2000

35% - 350 x 3cks -> 1050

---

total time = 4050 cks = 4050 x (1/1GHz) = 4050 / (10 power 9) = 4.06 MicroSeconds

Q4) For the Processor running at 1MHz, taking 3 clock cycles for the execution of one complete instruction (fetch & execute) and requiring one memory word (32bits) for storing one instruction, calculate the following for the given program:

i)Total memory required to store the program (in bytes)

ii) Total time required to execute the program (in seconds)

```
MOVE #10,R0
MOVE #15,R1
ADD  R0,R1,R2
```

Total time:

Frequency =1MHz clock , hece T = 1/frequency = 1Microsecond (i.e time for one clock cycle)

Total clocks for the complete program = 3 Instructions x 3 Clock Cycles (because 3clks/instrn)

= 9 clock cycles = 9 x 1Microsecond= 9 Microseconds.

Memory size = 3 instructions x 32 bits = 3 x 4 bytes/word = 12 bytes

(because each instruction requires, 32 bits i.e 32/8= 4 bytes)

Q5) Represent the 32-bit integer 20406080H in Little endian and Big endian formats

Assuming byte addressing by the processor,

Little endian – 80h,60h,40h,20h

Big endian - 20h,40h,60h,80h

Q6) Represent the 32bit integer 12345678h in Little Endian and Big Endian formats

Addresses	0	1	2	3
Big endian	12	34	56	78
Little endian	78	56	34	12

Q7) Indicate the value of condition flags after the execution of the following instructions.

MOVE #0F H,R0 (note: 0F is represented in Hex)

MOVE #F0 H,R1

ADD R0, R1,R2

0F – 0000 1111

F0 - 1111 0000

ANS 1111 1111

Condition Flags: Z=0, CY=0,OF=0, SIGN/N FLAG = 1

- Answer is not zero, hence Z=0
- No carry is generated after adding 0F and F0 numbers, hence CY = 0
- ExOr of Carry coming from D6 bit and D7 bit gives Overflow flag, here 0.exor.0 =0
- Sign flag is the last bit of the answer, it is 1, hence Sign flag(also called Negative flag) = 1

Q8)Perform the addition of two numbers 48H and 78H and indicate the value of the flags CY,Z,OF and Zero.

48 H ------ 0100 1000

78 H ------ 0111 1000

ANS - 1100 0000 CY=0,Z=0,OF=1,SIGN/N=1

What are the conditional code flags, indicate the value after the addition 98h and 68h

98H - 1001 1000

68H - 0110 1000

0000 0000 CY=1,Z=0,0F=0,N=0

Q9)Write a assembly program to evaluate the expression, A x B + C x D

MOVE A,R0

MOVE B,R1

ADD R0,R1,R2

MOVE C,R0

MOVE D,R1

ADD R0,R1,R3

ADD R2,R3,R3 ---- R3 contains the final answer

Q10) Give a short sequence of machine instructions for the task: ”Add the contents of memory location A to those of location B, and place the answer in location C”. Instructions LOAD LOC, Ri and STORE Ri, LOC are the only instructions available to transfer data between the memory and general purpose register Ri. Do not destroy the contents of either location A or B.

Assuming the instruction formats given in the question and RISC type processor.

LOAD A,R0
LOAD B,R1
ADD R0,R1,R2 ; (ADD R0,R1 – add R1 & R0 and place answer back in R1)
STORE R2,C

Q11) Register R1 and R2 of a computer contain the decimal values 3500 and 2800. Find the effective address of the memory operand in each of the following instruction

i. STORE R3, 50(R1,R2)

ii. ADD –(R1),R5

i.) Effective Address = 50 + (R1) + (R2) = 50 + 3500 +2800 = 6350

ii.) Effective Address = (3500-4) = 3496 (assumed word length is 32 bit , 4 bytes. In this instruction memory operand is –(R1), i.e R1 contents contains the address of the memory location, and ‘-’ indicates, pointer is decremented by 4, and is used to refer to the memory)

Q12) For an RISC machine with effective value of S as 1.25 and the average value of N as 200, Clock rate as 800MHz. Calculate the total program execution time required.

T = (N x S) / R

T = (200 x 1.25)/ 800MHz = 250/800 micro seconds (note 1/1MHz = 1 micro second)
= .0.3125 micro seconds

Q13) A processor has 40 distinct instructions and 24 general purpose registers. A 32-bit instruction word has an opcode, two registers operands and an immediate operand. Give the number of bits available for the immediate operand field.

- 2(power)6 = 64 , hence 6 bits are required to represent what type of instruction (5 bits=32)
- 2(power)5 =32, hence 5 bits are required to represent one register operand (4bits=16)
- Total = 10 bits are required to represent two register operands.
- Remaining bits = 32 – (6+10) = 16 bits, is available to represent the immediate operand)

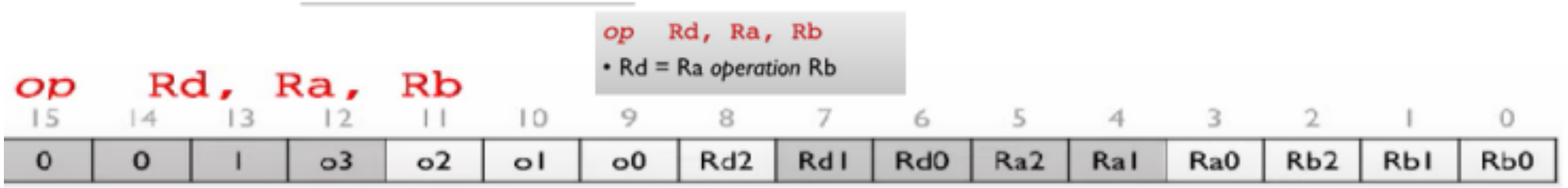
Q14) Write the meaning of the following instructions and contents of the registers R4,R2 after the execution of the following instructions/program. Assume following values before the execution of the program, R4=0X0005, R2=0X0006, [0X0005] = 0X02.

```
ADD   R4 , R2
ADD   (R4) , R2
```

- ADD R4 , R2 means, Add contents of R4 and R2 and store the answer in R2
 = 5+6 = 11 (in decimal)
 = 0x0B (in hexadecimal)

- ADD (R4) , R2 means, Add the contents of memory, pointed by R4 (i.e [0x0005]-> 2) and contents of R2 (i.e 11D) and store the answer in R2 (answer is 13D)
 = 13D

Q15). Prepare the machine code for the ADD instruction, as per the information given in the following figure




```
ADD ,R1,R4,R6
```

Ans:

The given instruction assumes: ADD R1,R4,R6 -> add contents of R4 (i.e Ra-100) and contents of R6 (i.e Rb-110) and then performs the operation, using the opcode indicated by 4bits O3,O2,O1,O0 (assuming 0000 is for ADD) and store the answer in R1 (i.e Rd-001)

001 0000 001 100 110

Grouping as nibbles

0010 0000 0110 0110

2 0 6 6 h or 0x2066 (in hexadecimal)

Q16) Write a sequence of instructions that will compute the value of y = X(power)4 + 2x + 6

3 address instruction:

```
MULT   Y,X,2
MULT   X,X,X
MULT   X,X,X
ADD    Y,Y,X
ADD    Y,Y,6
```

$$Y = X^4 + 2X + 6$$

2 address instruction:

```
MOV Z,X
MULT Z,X
MULT Z,X
MULT Z,X
MOV Y,6
ADD Y,Z
MOV Z,X
MUL  Z,3
ADD Y,Z
```

1 address instruction

```
MOVE X
MULT X
MULT X
MULT X
STORE Z
MOVE X
MULT 2
ADD Z
ADD 6
STORE Y
```

Q17)Write the Assembly language program to add n numbers with suitable comments.

Refer the slide 28

Q17)Demonstrate with code snippet, Machine Language/Assembly Language support to "Subroutine Linkage".

CALL and RET to be indicated
PC contents are loaded to Link Register on CALL,
While LR contents are loaded to PC on Return.

Note: students can also use Processor stack instead of LR register for storing..

When a subroutine is called, the address of the instruction following the CALL instructions stored in_____

Ans: Link Register (in some processors, it is on top of the stack, pointed by SP)

Q18) List all the generic addressing modes supported by the processor instruction set, with an example instruction.

Name	Example	Assembler syntax	Addressing function
	EXAMPLES		
Immediate	MOV #20,R0 move the value 20, to R0	#Value	Operand=Value
Register	MOV R1,R0 move the contents of R1 to R0	Ri	EA=Ri
Absolute (Direct)	MOV SUM,R2 add the number stored in memory location "SUM" TO R2 and store the answer in R2	LOC	EA=LOC
Indirect	ADD (R2),R0 add the number stored in memory location, whose address is stored in R2, with R0 and store ans in R0. R2 acts as pointer	(Ri)	EA=[Ri]
		(LOC)	EA=[LOC]
Index	MOV 4(R2),R0 move the contents of memory location, whose address is computed by addng 4 and contents of R2, to the register R0	X(Ri)	EA=[Ri]+X
Base with index	ADD (R0,R1),R2 add the contents of memory location, whose address is computed by adding the contents of R0 and R1, with R2 and store the answer in R2	(Ri, Rj)	EA=[Ri]+[Rj]
Base with index and offset	ADD 10(R0,R1),R2	X(Ri, Rj)	EA=[Ri]+[Rj]+X
Relative	BNE LOOP (BRANCH NOT EQUAL) program jumps/branches to the location, LOOP, whose address is computed by adding the contents of PC with the offset value i.e distance of LOOP relative to PC	X(PC)	EA=[PC]+X
Autoincrement	ADD (R2)+,R0 same as Indirect, but here pointer is incremented, after the operation	(Ri)+	EA=[Ri]; Increment Ri
Autodecrement	ADD -(R2),R0 same as Indirect, but here pointer is decremented, before the operation	-(Ri)	Decrement Ri; EA=[Ri]