UNIT I – Chapter 2

2. MACHINE INSTRUCTIONS & PROGRAMS

Chapter Outcome:

- ➔ Gaining knowledge over basic machine instructions and program execution..
- ➔ Different addressing methods for accessing register and memory locations.
- ➔ Syntax of assembly language.
- ➔ Concepts of subroutine and their execution.
- ➔ Usage of stacks in executing program.

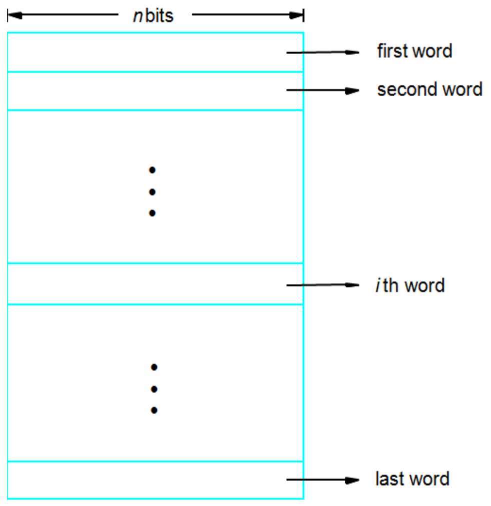
2.1 Memory locations and addresses

→Main functionality of the memory unit is to store programs and data operands that are required for program execution.

→Each memory unit is made up of a large storage cells were each cell has the capacity of storing 1 bit in terms of binary value 0 or 1

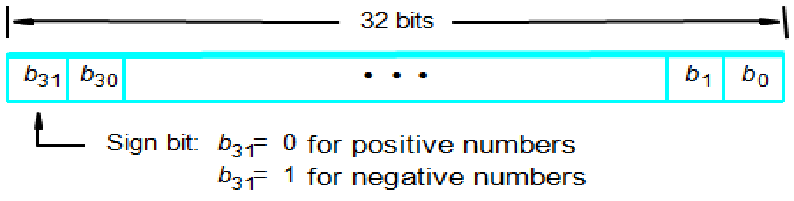
→Even if the memory unit is made of single bit storage cells, while accessing it accesses in terms of group of n bits called **word.** n bits representing the word is called as **word length.** Word length varies from system to system and the usual length varies from 16 to 64 bits.

→When representing memory schematically, it is represented as a collection of words arranged verticallyas shown in the figure below:



→Next lets concentrate on the structure of each word, how exactly it stores the information. From now on we will discuss the memory unit having 32 bit word length as a standard.

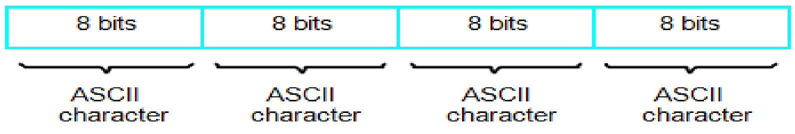
Structure of each word


32 bits
b_{31}
b_{30}
• • •
b_1
b_0
Sign bit: b_{31} = 0 for positive numbers
b_{31} = 1 for negative numbers

(a) A signed integer

This is one of the format for the structure of word which shows the arrangement of bits according to the word length.

There is another way for storing the information in encoded form. In encoded form, the information will be stored in terms of ASCII value, were each ASCII character is made up of 8 bits (1 byte). Then a word of 32 bit length is divided into 4 ASCII character or 4 bytes length. Below shown figure represents the format of encoded 32 bit word.



(b) Four characters

Accessing memory items:

After knowing about the proper structure of memory and its representation, next we need to know how exactly data items stored in memory location can be accessed. To access any item from memory location, each location is assigned with a unique address.

Address locations are directly related to number of address lines that is present in a system. If there are **k** bits of address lines, then the address value ranges from **0 to 2^k - 1**locations.

2.2.1 Byte Addressability

Assigning address to successive bytes of the memory location is called as **Byte addressability.** Most of the modern computers use this assignment system. If the word length is 32 bits, then successive words are located in 0, 4, 8… byte positions in memory system.

2.2.2 Big Endean and Little Endean Assignments

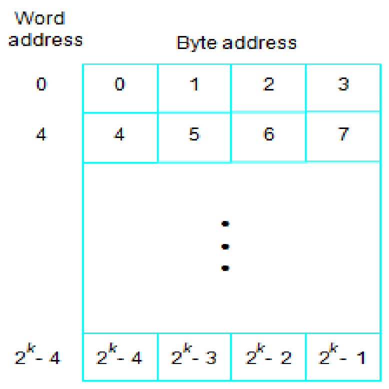
There are basically two different methods for assigning byte addresses across the words in a memory location:

1. Big Endean form

2. Little Endean form

1. Big Endean Form:

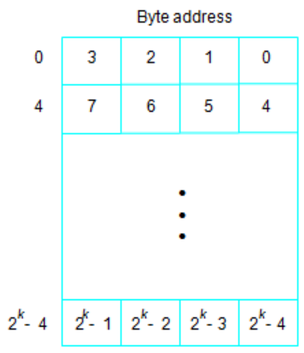
In this form of representation, lower byte addresses are used for Most Significant Bytes of the word. I.e. the leftmost byte of the word. To get the better understanding of this form, consider the pictorial representation shown below:



(a) Big-endian assignment

2. Little Endean Form:

In this form of representation, lower byte addresses are used for Least Significant Byte of the word. I.e. the rightmost byte of the word. The representation is done exactly opposite of the Big Endean format. Even then when accessing the successive words, successive words are located in the same location as that of Big Endean format. To get the better understanding of the form, consider the pictorial representation shown below:



(b) Little-endian assignment

2.3 Memory Operations

There are two basic memory operations:

1. Load (Read or Fetch)

2. Store (Write)

1. Load: Transfers a copy of contents of a specific memory location into processor registers. On performing this operation, contents of memory location remain unchanged. To accomplish this operation, address of memory location is sent to the memory unit along with the read signal. Hence also called as **read** operation.

2. Store: Transfers an item of information from the processor register into a specific memory location destroying former contents that is already stored inside the memory location. This operation is accomplished by sending desired address of the memory location along with write signal into memory unit. Hence it is also called as **write** operation.

2.4 Instructions and Instruction sequencing

Computer system must have instructions that are capable of performing mainly four types of basic operations:

1. Data transfer between processor registers and memory.

2. Arithmetic and Logic operations on data.

3. Program sequencing and control.

4. I/O transfers.

2.4.1 Register Transfer Notation (RTN)

When it comes to transfer of information in the computer system, transfer of information involves transfer of data from one location into another location. Possible locations are memory locations, processor registers and I/O registers.

To represent the transfer of information between these locations, a standard notation is used and this standard notation is called as **Register Transfer Notation.** It uses different nomenclatures to represent these locations.

- ➔ Symbolic names for the memory locations are represented as variable names such as LOC, A, VAR1, NUM etc.
- ➔ Symbolic names for registers are usually represented by placing a alphabet or number suffix to the letter 'R'. For ex R0, R1, R2, …..etc
- ➔ I/O registers are represented by the names such as DATAIN, DATAOUT etc.

Contents of these locations are denoted by placing a square braces between the location names. For example

- ➔ A notation [LOC] means contents present inside the memory location LOC.
- ➔ A notation [R0] means contents of the register R0

To get a clear knowledge on Register Transfer Notation, we will look into some of the example notations and their meaning which is shown below:

Example 1:

R1 ← [LOC]

This means contents of memory location LOC is copied into the processor register R1.

Example 2:

R3 ← [R1] + [R2]

This means contents of register R1 and R2 are added and sum is placed inside the processor register R3.

These types of notations help in understanding the problem and makes writing of machine instructions in an easier way. Every time left side has to be the name of the location and right side has to be contents of some location.

2.4.2 Assembly Language Notation

A notation used to represent the machine and programs very closely are called as **Assembly Language Notation**.

For example consider an operation that is represented using Register Transfer Notation

R1 ← [LOC]

Above operation loads the contents of memory location LOC into the register R1. In machine instruction, this can be represented as:

Move LOC, R1

This instruction copies the data from memory location **LOC** and loads into a register **R1.** In this notation first we need to write the operation to be performed which is **Move**. Then destination location **R1** is written at the end and sources must be written first immediately after the operation. Here source is **LOC.** Source means it will fetch the contents of the specified location which can be a register or a memory location.

Consider another Register Transfer Notation

R3 ← [R1] + [R2]

This instruction as we know adds contents of register R1 and registers R2 and stores the value inside register R3. In that case R1 and R2 are sources and R3 is a destination and the operation involved is add operation. In a machine instruction, this can be represented as shown below:

Add R1, R2, R3

2.4.3 Basic Instruction Types

Based on the number of operands that can be used in a single instruction, machine instructions can be written in three different formats:

1. Three address instruction
2. Two address instruction
3. One address instruction

To discuss the format of all the above instructions, we will consider a common example of the expression **C = A+B** as an example

Expression C = A+B in Register Transfer Notation is represented as:

C 🡐 [A] + [B]

1. **Three address instruction**: An instruction type which consists of memory address of three operands are called as three address instruction. Below shown is the syntax of three address instruction format

Syntax

Operation Source1, Source2, Destination

Were,

Operation 🡒Represents the type of operation to be performed such as add, sub etc

Source1 🡒 Represents operand1 memory location used in the operation

Source2 🡒 Represents operand2 memory location used in the operation

Destination🡒Represents the destination memory location were result is stored

For the Considered expression can be represented as:

Operation 🡒 Add

Source1 🡒 A

Sourec2 🡒 B

Destination 🡒 C

Instruction Format is:

Add A, B, C

2. Two address instruction: An instruction type which consists of memory address of only two operands is called **two address instruction types.** In this format, One of the location acts both as a source and destination.

Syntax for two address format is:

Operation Source, Destination

Were,

Operation →Operation performed such as Add, Sub etc

Source →Operand1 required for the operation fetched from memory location

Destination→This acts both as a source and destination. To do the operation, it fetches operand2 from the memory location and after performing the operation it acts as destination location for storing the result.

To implement C = A+B:

1. Move one of the operands to destination location C
   C ← [B]
2. Then add the operands A and C and store it in a memory location C
   C ← [A] + [C]

In Two address instruction, it can be represented as:

Move B, C

Add A, C

3. One address instruction: An instruction type which contains memory address of only one operand is called as **One address instruction.**

In that case second operand is considered to be in a location called as processor register. Some systems have only one unique processor register called **accumulator** and some systems have many registers called **general purpose registers.** Based on this there are two classifications for one address instruction:

i. **One address instruction using accumulator:** In this format, instruction contains memory address of only one operand and the second operand is considered to be in a unique location called **Accumulator.**

For example:

Add A

It means to add the contents of memory location A with contents present in the accumulator and store the result back in the accumulator.

Load A

It means to load the contents from memory location A into the accumulator register.

Store A

It means to store the contents of **accumulator** into the memory location **A**

C = A + B (C← [A] + [B]) using single instruction format is performed as shown below:

Load	**A**	**(acc← [A])**
Add	**B**	**(acc← [acc] + [B])**
Store	**C**	**(C ← [acc])**

ii. One address instruction using general purpose registers: Modern computers have n number of processor registers and are called as **general purpose registers (High speed processor memory).** In that case one of the operand is fetched from the memory location and the other one is fetched from the general purpose register specified.

For example, instruction of this format can be written as:

Move A, Ri

This means loads the contents of memory location A into the register Ri

Move Rj, A

This means stores the contents of register Rj into the memory location A

Processors that allow arithmetic operations to be performed on the registers represents the instruction **C = A+B** in the following type:

Move	**A, R0**	**(R0 ← [A])**
Move	**B, R1**	**(R1 ← [B])**
Add	**R0, R1**	**(R1 ←[R0] + [R1])**
Move	**R1, C**	**(C ← [R1])**

Instruction format in which one memory operand is allowed on the memory location:

Move	**A, R0**	**(R0 🡐 [A])**
Add	**B, R0**	**(R0 🡐 [B] + [R0])**
Move	**R0, C**	**(C 🡐 [R0])**

[Note: Please do solve this example problem (A+B)*C and in case any doubts please do revert back]

2.4.4 Instruction Execution and Straight line sequencing

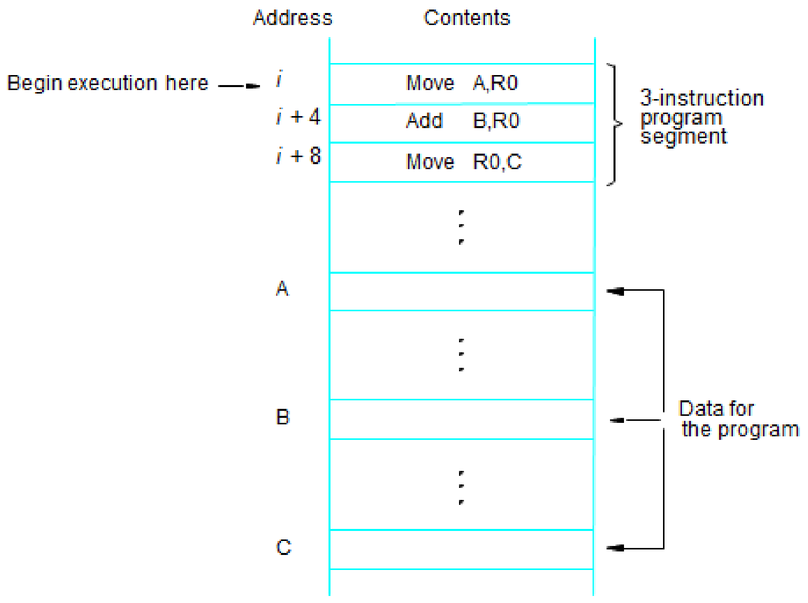
In this section we will know about the arrangement of program instructions in memory and how they are executed by the processor to accomplish a task.

To explain the instruction execution concepts in detail, let's consider the execution of following set of instruction sequence by the processor:

Move	**A, R0**
Add	**B, R0**
Move	**R0,C**

Program instructions and data required for the program execution needs to be brought into main memory before they start their execution. We will see how these instructions and data are arranged in the main memory unit for the set of instructions considered above

Arrangement of program instructions in memory system



Instructions when stored in the memory are stored in a sequence as shown in the above schematic representation.

System memory is assumed to be a 32 bit memory system and instructions are stored in successive words having 4 byte length in the sequence I, i+4 and i+8.

Memory location A, B and C correspond to the data required for the operation.

Next we will look how execution steps are carried out in the processor for the above considered set of instructions

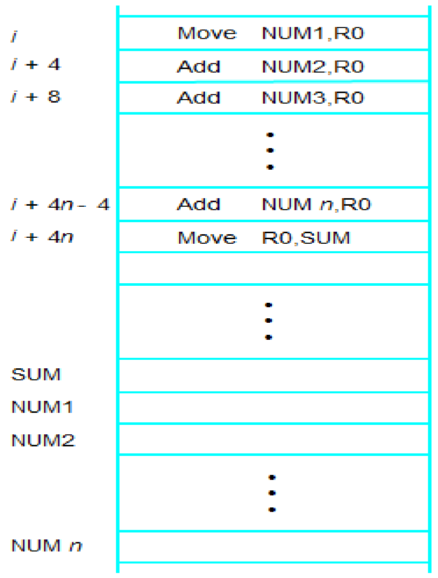
1. To begin with the execution of a program, register called **Program Counter (PC)** must hold the address of first instruction that is present in the location I if the program.
2. Next processor control circuits use the information in the **PC** to fetch and execute instructions one at a time in the order of increasing address, this method of execution is called as **Straight line sequencing.**
3. During execution of each instruction, value of **PC** is incremented by 4 to point to the next instruction in the program sequence.

Altogether execution of a given instruction is divided into two phases:

1. **Instruction fetch:** In this phase instruction is fetched from the memory location whose address is in **PC** and fetched instruction is placed in a**Instruction Register (IR).**
2. **Instruction execute:** In this phase instruction in IR is executed to determine the operation to be performed. This phase often involves fetching of operands either from memory or processor registers and performing operation on them and storing result back in register or memory location.
   Some point during these two phases PC is incremented to point to the next instruction.

2.**4.5 Branching**

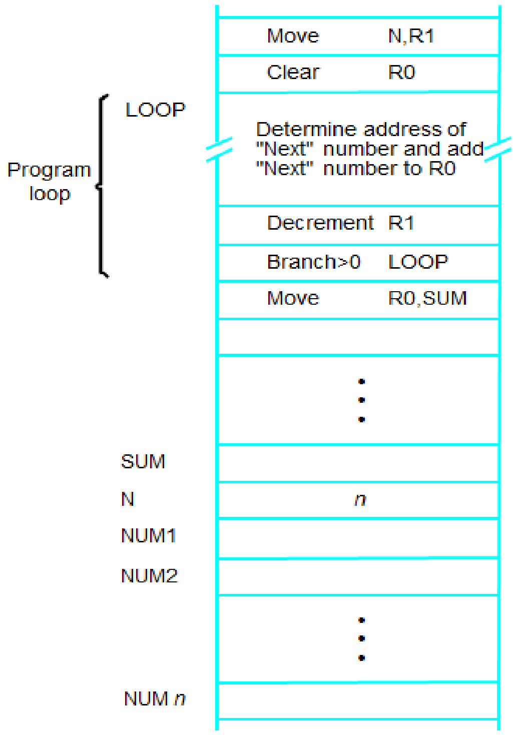
To explain branching technique, let us consider the task of adding N numbers through machine instructions:
Assume that the address of **N** numbers to be **Num1, Num2,….,Numn** which are sequentially arranged in the memory location one after the other in successive words. Let us use the format of the instruction that allows only one operand address to fetch from the memory location and the other from any general purpose register. After doing the require operation final result is saved in the memory location called **sum**
According to straight line sequencing technique, machine instruction for the considered problem can be implemented as shown in the figure below.



As and when instruction accesses a new number, it is added to the **sum** series were register **R0**is keeping track of the sum series and final result is saved in the location sum.
If the implementation is done through straight line sequencing as shown in the above figure, it requires a very large number of add instructions to be placed in the memory which acquires lot of space in the memory. I.e. if there are n numbers there are n-1 add instructions. To overcome this problem, instead of placing so many add instructions a single add instruction can be place inside the loop which can be made to repeat n times with a different data value.
Logic of implementing looping concept through machine instruction is called **Branching** technique.
To understand the concept of branching for the above instructions, we will understand the logic through 'C' program pseudocode written as shown below:

```
sum = 0;
loop: sum = sum + next number in the address location;
n=n-1;
if(n>0) goto loop
```

Here value **n** is the total numbers to be added in the series which keeps track of the total number of iterations. It starts from the highest value and gets decremented for every iteration and has to stop when it reaches 0. In every iteration the number is added to sum series and finally when the loop terminates, sum series is saved in the location **sum. goto**is used to implement the looping concept which goes back to a location called **loop** if number is not equal to 0.
Using loop technique in the machine instruction technique, it can be written as shown in the figure below:



Let us understand the above written instructions by comparing it with pseudo code written in C.

→Initially register R1 is set to value of Nthrough the instruction move N, R1. This register is used to keep track of total number of iterations for the loop.

→Clear R0 instruction empties the register R0 or simply sets the NULL (0) value to the register so that each number is added to R0 and stored back in R0. To keep it simple it sets the value R0=0.

→To determine the address of a next number, there is a method and we will study how to do that when we discuss the concept called addressing modes.

→Decrement R1 decrements the contents of register R1 by 1. Were R1 has the value of total numbers which determines the count of a loop n times

→Next we will see processing of branch instructions:

- Branch instructions load a new value into a program counter (PC) apart from the sequential execution.
- Branch instructions specifies the new address that is to be loaded into the program counter and this address is called as **Branch Target address.** Program counter is loaded with the branch target address only if the specified condition in the branch instruction is satisfied.
- For example in the above written set of instructions, **Branch** > **0** Loop means the result of immediately preceding instruction is checked and if it is greater than 0 then program counter will be loaded with the address 0 or else carries out the execution in sequence. In that case result of **decrement R1** is checked which the preceding instruction to branch instruction is. If it is greater than zero or not and decision of branching is done based on its value.

2.4.6 Condition Codes

→Processor keeps track of information about the results of various instructions for the usage of subsequent conditional branch instructions.

→This is done by recording the required information of the most recently executed instruction in a individual bits. These individual bits are called as **Condition Code flags.** These flags are grouped together in a special processor register called **Condition Code register** or **Status register.**

There are four commonly used flags for any operations:

1. **N (Negative flag):** This flag sets a bit value to 1 if the executed result is less than zero or else sets the value to 0.
2. **Z (Zero flag):** This flag sets the bit value to 1 if the executed result is zero or else sets the value to 0.
3. **V (Overflow flag):** This flag sets the bit value to 1 if the executed result has caused arithmetic overflow or else sets the value to 0.
4. **C (Carry flag):** This flag sets the bit value to 1 if the executed result has caused carry out from the operation or else sets the value to 0.

Typical register structure would be arranged as shown below:

N	Z	C	V

For example an instruction Branch > 0 will check if the bit N and Z are 0 or not to ensure that value of the previous result is greater than zero.

2.5 Addressing modes

What are addressing modes?

Different ways for specifying the address of a memory location in the instruction for fetching the operands are called as **addressing modes.**

Based on the type of operand need to be accessed, addressing modes are categorized under the following types:

1. Implementation of variables and constants
2. Indirection and pointers
3. Indexing and arrays
4. Relative addressing
5. Additional modes

2.5.1 Implementation of variables and constants

1. Representation of variables: A variable in the instruction is represented by allocating a register or memory location to hold its value (contents). There are two different types based on the location type it is stored:

i. Register mode: Operand to be fetched is the contents of processor register were name of the register is specified in the instruction

Example: Add R0, R1

In this example R0 and R1 contents acts as an operands to be fetched.

ii. Absolute mode (Direct mode): Operand to be fetched is the contents of memory location and address of memory location is directly stated in the instruction in terms of the memory location name. This mode is also called as direct mode as it directly accesses the memory location for fetching the operands.

Example: Move Loc1, Loc2

In this example, Loc1 and Loc2 are the locations of the memory to be accessed which moves the contents of Loc1 into Loc2 of the memory.

As an example consider below instruction:

Move Loc, R2

In this example, contents of memory location Loc is loaded into the processor register R2. In this case both register mode and direct mode are used were R2 is a register and Loc is a memory location.

2.Representation of constants

Immediate mode: To represent constants there is only one addressing mode called as immediate mode. In this addressing mode, operand value is explicitly given in the instruction. Since the values can be explicitly given only into the registers it is also called as **Register Immediate mode.** Note that when using this addressing mode the values can be stated only to the register and memory location cannot be used in this mode.

Example: Move #200, R0

This places the value 200 into the register R0 and symbol '#' is used to specify the immediate mode.

Example: Move #NUM1, R0

This fetches the address value of memory location named #NUM1 and loads into the register R0.

Let's look into another example. To implement **A = B+6** in the machine instruction, it can be implemented as:

In register transfer notation, it is represented as A ← [B] + 6

Move	**B, R1**	**(R1← [B])**
Move	**#6, R1**	**(R1← 6)**
Move	**R1, A**	**(A← [R1])**

2.5.2 Indirection and Pointers

Following addressing modes that will be discussed does not provide the memory address of the operand directly. Rather than it gives the information from which the memory address of the operand can be determined. The address of the meory location to fetch the operand is called as **Effective address.**

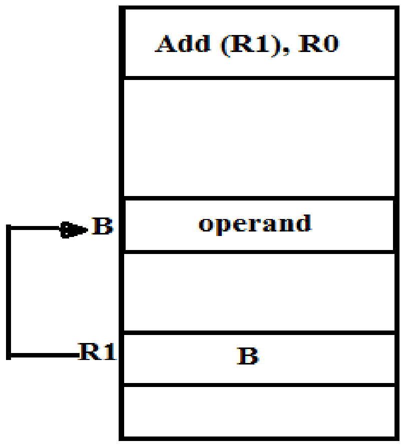
Basically in this type, memory is accessed indirectly from another memory. This technique of addressing is called as **Indirect mode.**

Indirect mode: Effective address of the operand is the contents of register or memory location whose address is specified in the instruction. We denote the memory to be accessed in this mode by placing the name of the register or memory location in parenthesis.

For example (R1) means operand address is the contents of register R1 and (LOC) means operand address is the contents of the memory location.

We will look into the following examples to get the clear understanding:

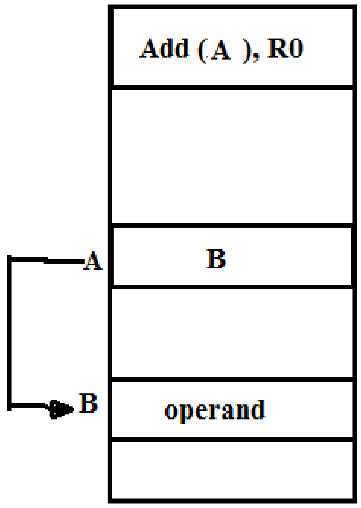
Example 1:Add (R1), R0



In this example, R1 is specified in the braces and indicates the indirect mode. That means R1 contains the address of the memory location were the operand is placed. The above instruction uses indirect mode through general purpose register.

Here in this example content of R1 is the address from were operand needs to be fetched. R1 is having the value B which means B is the effective address from where the operand needs to be fetched.

Example 2: Add (A), R0



This example is bit different in the sense it uses indirect mode through memory location. Here the address of the memory location is the contents of the memory location A which is nothing but B. So here in this case effective address is the memory location B.

These registers or memory location that contains the address of another memory location is called as pointer.

Example: Finding sum of N numbers using indexed addressing mode

In the last section we have left the method of finding the address of the operand in the example of adding n numbers. Instruction written below shows the method of finding the address of a memory location using indexed mode.

```
        Move     N, R1          //load the value of N into register R1
        Move     #NUM1, R2      //load register R2 with NUM1 address value
        Clear    R0             //set register R0 with null value
Loop:   Add      (R2), R0       //add number pointed to by the location
```

```
Add         #4, R2      //increment value of R2 to point to next address location
Decrement   R1          //decrement the value of counter
Branch>0    Loop        //Check if R1>0 if so goto Loop
Move        R0,Sum      //store the result in location Sum
```

Logic of the above program:

1. First instruction loads the value of N to the register R1 (N contains total numbers to be added in the series)
2. Next **#NUM1, R2** is a**immediate mode** which will extract the address of first number and loads the value into register R2.
3. Clear R0 sets the value of register R0 to 0 which is required for the continues addition of the sum series.
4. Loop instructions:
   a. **Add (R2), R0** is a indirect mode which adds the operand from the address present in R2 (for the first iteration it is NUM1) with the value of R0 and loads the result into the register R0
   b. **Add #4, R2,** this instruction adds a value 4 to the value of R2 were R2 is incremented by 4 bytes. This increments the value by one word to access the next number present in the memory location.
   c. **Decrement R1**decrements the value stored in R1 by one I.e. decrements the value by 1 after adding a number which tracks the number of iterations needs to be done.
   d. **Branch > 0** will move the control to loop if the condition is true I.e. it will check if the result of preceding instruction is greater than zero.
   e. Finally once after terminating the loop result is stored in the memory location **Sum.**

2.5.3 Indexing and Arrays

Indexed mode: Effective address of the operand is generated by adding a constant value to the contents of the register. General syntax of indexed mode is given by:

X(Ri)

Were,

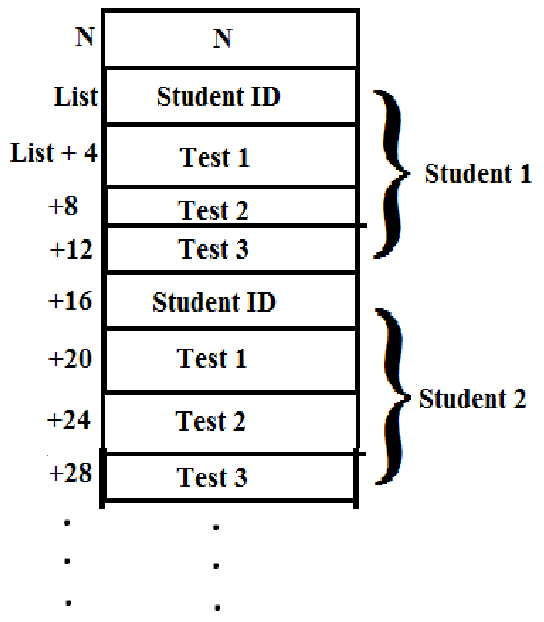
Ri→Index register having an address value often called as a Base register.

X →A constant value added into the contents of register Ri called as **offset value.**

Effective Address (EA) is calculated by the formula:

EA = X + [Ri]

An example program to exhibit the usage of Indexed mode:



To exhibit the usage of indexed n\mode, we will consider the example of student marks entry of three tests which is shown accordingly in the figure above.

Input to this program is student’s ID and the marks of three tests and the output is the sum of test1, test2 and test3 of all the students calculated separately.

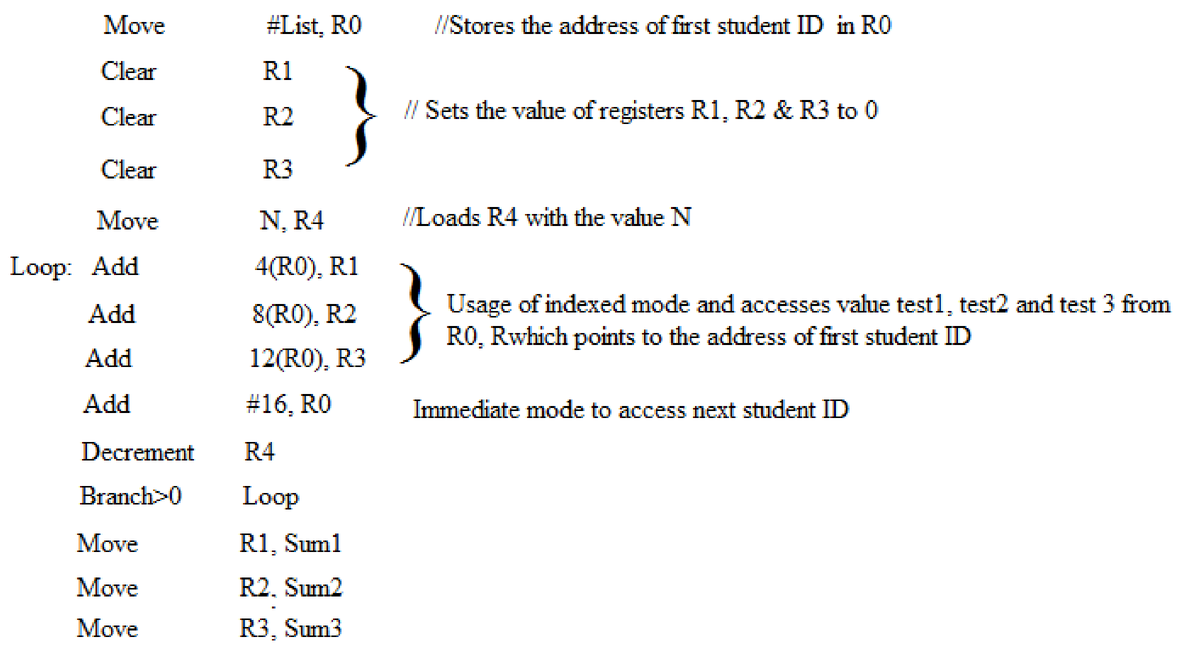
By observing the above figure, we will get a clear picture on the structure of data arrangement and following observation can be made:

1. Every preceding student ID is found after 4 memory locations (16 bytes) from the previous student ID
2. Test 1 is after 1 word from a particular student ID (4 bytes)
3. Test 2 is after 2 words from a particular student ID (8 bytes)
4. Test 3 is after 3 word from a particular student ID (12 bytes)

So above data can be accessed by referring to the first student ID and then incrementing its address

According to the data structure shown above, memory location List is the memory location containing the student 1 ID and all the other data can be accessed from that point.

Let us see how to write the program accordingly



```
        Move        #List, R0      //Stores the address of first student ID  in R0
        Clear       R1   }
        Clear       R2   }         // Sets the value of registers R1, R2 & R3 to 0
        Clear       R3   }
        Move        N, R4          //Loads R4 with the value N
Loop:   Add         4(R0), R1   }
        Add         8(R0), R2   }  Usage of indexed mode and accesses value test1, test2 and test 3 from
        Add         12(R0), R3  }  R0, Rwhich points to the address of first student ID
        Add         #16, R0        Immediate mode to access next student ID
        Decrement   R4
        Branch>0    Loop
        Move        R1, Sum1
        Move        R2, Sum2
        Move        R3, Sum3
```

Types of indexed mode: There are some variations in the indexed mode addressing by the way it is being written:

1. **Base with index register:**Here one of the register itself is used as offset value X. General syntax of this type of indexed mode is written as show:

   (Ri, Rj)

   Here second register Rj acts is the offset value and effective address EA is given by:

   EA = [Ri] + [Rj]

2. **Base with index and offset value:** Here constant value X, contents ofindex register Rj and Base register Ri are added together to get the Effective Address value.
   General syntax of this mode is given by:

   X(Ri, Rj)

   EA = X + [Ri] + [Rj]

[Problem: Please try to implement the column sum using machine instructions of matrix 3X3. In case of any confusion get it cleared.]

2.5.4 Relative addressing mode

In the last section we have discussed about the indexed addressing mode using any general purpose register as a base register. An useful version of this is derive by using **Program Counter** in place of general purpose register. Hence it is also called as PC relative addressing mode.

General form of this mode is denoted by **X(PC)**which addresses a memory location that is X bytes away from location pointed to b the PC. Here X is a constant value that acts as a displacement from PC which is a base register in this case.

The addressing mode is called as relative mode because the address of location is identified relative to the program counter. Effective address of this mode is given by:

EA = X + [PC]

Most common use of this mode is to specify target address in branch instructions such as:

Branch > 0 Loop

Here Loop is the branch target address and when processed internally specifies the displacement value from the program counter.

To get better understanding of the processing of this mode, consider an example program written below:

```
                Move         N, R1

                Move         #NUM1, R2

                Clear        R0

1000            Loop:  Add           (R2), R0

1004            Add          #4, R2

1008            Decrement    R1

1012            Branch>0     Loop

1016            Move         R0,Sum
```

Above program is a set of instructions to add N numbers in the loop. Let us assume that instruction belonging to the loop starts from the location with address 1000.

We know that Program Counter (PC) register will be having the address of next instruction to be fetched and executed in the processor when the current instruction is getting executed in the processor.

When the instruction Branch > 0 at address 1012 is getting executed, at that time PC will be having the value of the address 1016 which is the next instruction to be fetched in the sequence.

But if the Branch > 0 is evaluated to true, PC should load the value of the address that points to Loop. I.e. the address 1000 has to be loaded into PC. This means that PC value should be decremented by 16 bytes in the address location to point into address value 1000.

When Branch is getting executed, PC will be having the value 1016 and if evaluated to true it has to load a new value 1000. Mathematically it can be given as:

X + PC = 1000

X + 1016 = 1000

X = 1000-1016

X = -16

X value has to be -16 if it has to go back to the Loop address

2.5.5 Additional Modes

So far discussed are the basic modes of addressing which are present in almost all the computer systems. Following discussions are on additional modes and they are called as additional modes because these modes are derived from basic modes which are not present in all the systems and they are intended to aid certain programming tasks in fewer steps that combine basic modes together in a single instruction.

Two types of additional modes are discussed here:

1. **Auto increment mode**
2. **Auto decrement mode**

1. Auto increment mode: In this addressing mode effective address of the operand is the contents of the register specified in the instruction. After accessing the operands, the contents of the register are automatically incremented to point to the next item in the memory location.

When it is incremented, it is incremented according to the value of accessed operand. For example if the accessed operand is integer, memory location of 1 word (4 bytes) is incremented in 32 bits system. If the operand type is character, location value is incremented by 1 byte (8bits).

The general form of representation of this addressing mode is **(Ri)+**which automatically increments the value to the next location corresponding to the operand type accessed.

For example: Consider the following instructions:

Add (R2), R0

Add #4, R2

In the above set of instructions, operand is accessed from the location pointed to by the contents of the register R2 and added with R0. After adding it with R0, contents of the register R2 is incremented by 4 bytes to point to the next location. This can be written in single instruction using the auto increment mode as shown below

Add (R2)+, R0

Above instruction after accessing the operand, increases value by 4 bytes to point to the next location in the memory.

This mode can be used for adding N numbers in the series. Below set of instructions shows the usage of this mode for adding N numbers.

```
      Move        N, R1        //load the value of N into register R1
      Move        #NUM1, R2    //load register R2 with NUM1 address value
      Clear       R0           //set register R0 with null value
Loop: Add    (R2)+, R0         // Access the operand and increment to the next location
      Decrement   R1           //decrement the value of counter
      Branch>0    Loop         //Check if R1>0 if so goto Loop
      Move        R0,Sum       //store the result in location Sum
```

2.Auto decrement mode: This addressing mode is exactly opposite to the auto increment mode. This mode first decreases the memory location to value to point to next location and then accesses the operand value from that memory loaction

General form of this mode is given by: **-(Ri)**

2.5.6 Summary of addressing modes

Name	Assembler syntax	Addressingfunction
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
Basewith index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Basewith 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*]

2.6 Subroutines

In a program particular subtask needs to be performed many a times with different data values. Such subtask is called as **Subroutines.**

For example adding of two numbers, finding square root etc.

To save space in memory system, a copy of subroutine instructions is place d in the memory and any programs that require subroutine simply branches control to starting location of the subroutine. This process of branching into the subroutine is called as **Calling Subroutine** and the instruction that performs branch operation is called as **call instruction.**

After subroutine execution, calling program must resume its execution continuing immediately after call instruction. Instruction that causes control to go back to Calling program is called as **return** instruction.

Locations were calling program resumes its execution after returning from the subroutine is the value of the updated PC while call instruction is being executed. Hence contents of PC must be saved by **call instruction** before the control branches into the starting location of the subroutine.

2.6.1 Subroutine linkage method

The way in which computer makes it possible to call and return from the subroutine is referred to as **Subroutine linkage method.**

One of the simplest subroutine linkage methods is to save return address value in a specific location which may be a dedicated register called **Link register.**

In the subroutine linkage method, **call** and **return** instructions play a major role. Let us look into the operations performed by them:

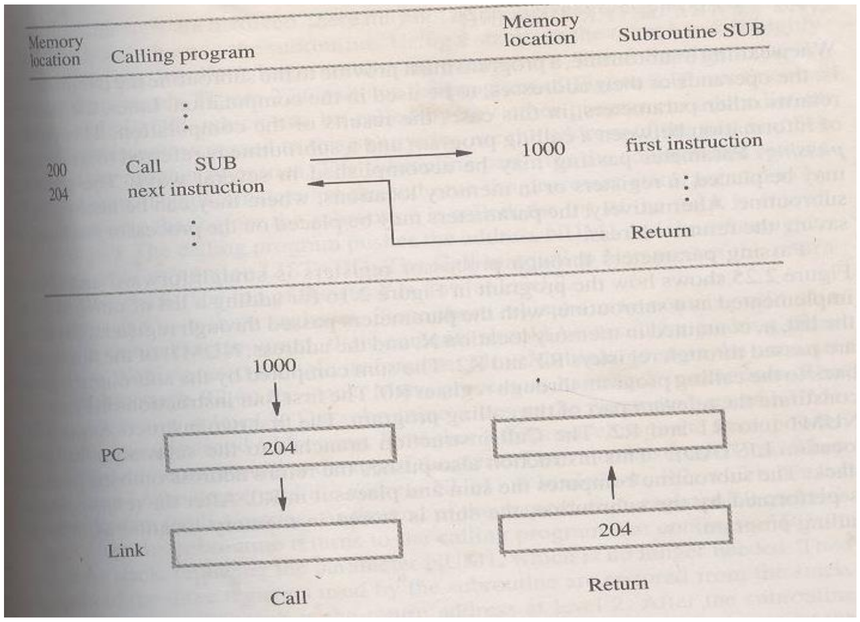
Operations performed by call instruction

1. Stores the contents of PC in link register
2. Branch to the target address specified by the call instruction.

Operation performed by Return instruction

1. Branch to the address contained in the link register.

Operations performed by the above instructions are best explained through the example shown below:



Here when the call instruction is executed, resuming address is 204 which is saved in the link register. Then when return instruction is executed, PC is loaded with the value saved in the link register.

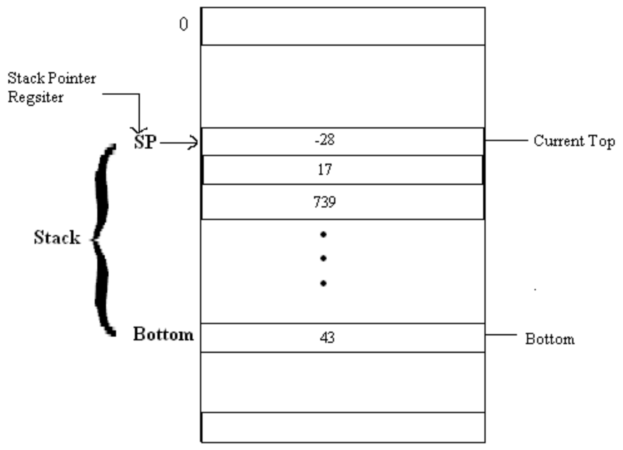
2.7 Stacks and Queues

We know that a computer program often needs to perform a particular subtask many a times using **subroutine Structure.**

In order to organize control and information linkage between **subroutine** and **main program**, a data structure called **stack** is used. A stack is a list of data elements were elements can be added or removed at one end of the list only. The end from which items are added or removed is called as **TOP of stack**and the other end is called **bottom of the stack.** The last entered element is the first one that is removed from the list. Hence it is also called as **Last In First Out (LIFO).**

Term **Push** is used to add the elements and term **Pop** is used to remove the item from the list.

Representation of the Stack in memory



Data that needs to be stored in computer memory can be organized as a stack shown in the above figure.

Assume that very first element is placed in the bottom of the stack and when new elements are pushed, they are placed in the successive lower address location. Here attack grows in a decreasing address value and shrinks in increasing address value.

A processor register called **Stack Pointer (SP)** is used to keep track of the Top of the stack. It could be one of the general purpose register or a dedicated register.

2.7.1 Operations on Stack

Being familiar with the concept of stack, it doesn't make that difficult to implement the operations of the stack such as **Push** and **Pop** operations.

1. **Push Operation**

Push operation means decrementing the stack pointer (SP) i.e. top of the stack and then adding an element into the location that is pointed to by **SP**

The operation on this can be implemented as shown below:

```
Subtract    #4, SP            //Decrement SP by one word to point into next location
Move        Newitem, (SP)     //Add the item to the address pointed by SP
```

The implementation can be written using the single instruction through auto dcrement addressing mode :

Move Newitem, -(SP)

Whenever it comes to the push operation on stack, above instruction is used to implement it.

2.Pop Operation

Pop operation means remove the value from the stack top i.e. the item pointed by **SP** and then increment the stack pointer value to the next location to point into the new top of stack.

The operation can be implemented using below shown instructions:

Move (SP), Item

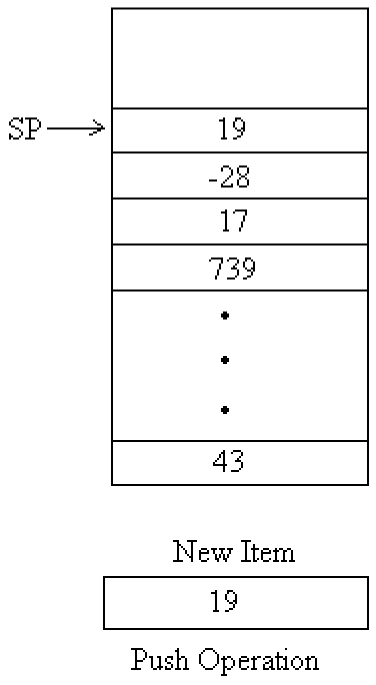
Add #4, SP

The above instruction can be replaced by single instruction using auto increment mode:

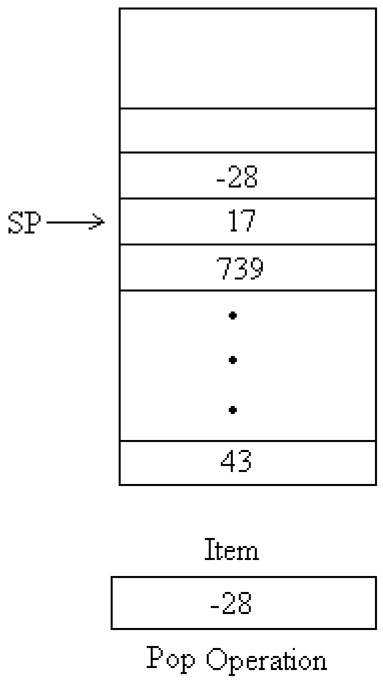
Move (SP)+, Item

From now on pop operation is implemented as shown in the above instruction.

Summarization of Push and Pop:



Push Operation



Pop Operation

2.8 Subroutine Nesting and Processor stack

One subroutine calling another subroutine is called as **Subroutine Nesting.**

In that case if we use link register to store the return address of the calling program, second call will be stored in the link register destroying previous contents of the first call instruction. Hence it is very much required to save the contents of link register in some other location as new call instructions are executed.

We know that subroutines can be carried out to any level. But while returning, first address to return is the last one generated in the calling sequence and return address generated first is last needed in the return sequence which clearly shows the concept of **LIFO** for its operation. Hence to implement the concept of LIFO into subroutine nesting, data structure stack is used were return addresses are pushed from **PC** on call instruction and popped out from the stack into **PC** on return instruction. Stack Pointer (SP) is used to keep track of pushed and popped return address.

With this **call** and **return** instructions implement two new operations:

Call: Pushes the contents of PC onto processor stack and loads subroutine address onto PC.

Return: Pops addresses from the stack pointed to by SP and loads into **PC.**

2.8.1 Parameter Passing

When calling a subroutine, calling program must provide subroutine the operands or their addresses required to do the operation. Later once after doing the required operation, subroutine returns the parameters in the form of result back to the calling program.

This exchange of information between a calling program and subroutine is called as **Parameter Passing.**

Parameter passing can be done in three different ways:

1. Registers
2. Memory location
3. Stack

Parameter passing through registers and memory location is similar. Only difference is that memory location is used in place of register. So let us look how parameter passing is doe through registers:

Example program for parameter passing through register

```
        Move        N, R1       //load the value of N into register R1
        Move        #NUM1, R2   //load register R2 with NUM1 address value
        Call        LISTADD
        Move        R0, Sum     //store the result in location Sum
```

Subroutine:

```
LISTADD Clear       R0          //set register R0 with null value
  Loop: Add    (R2)+, R0        // Access the operand and increment
        Decrement   R1          //decrement the value of counter
        Branch>0    Loop        //Check if R1>0 if so goto Loop
        Return                  //Return back to calling program
```

Here R1 and R2 are registers for parameters passed from main to subroutine as R1 and R2 are accessing the value assigned in the main program

R0 register is passed parameter from subroutine to main as the value for R0 assigned in subroutine is used in the main program

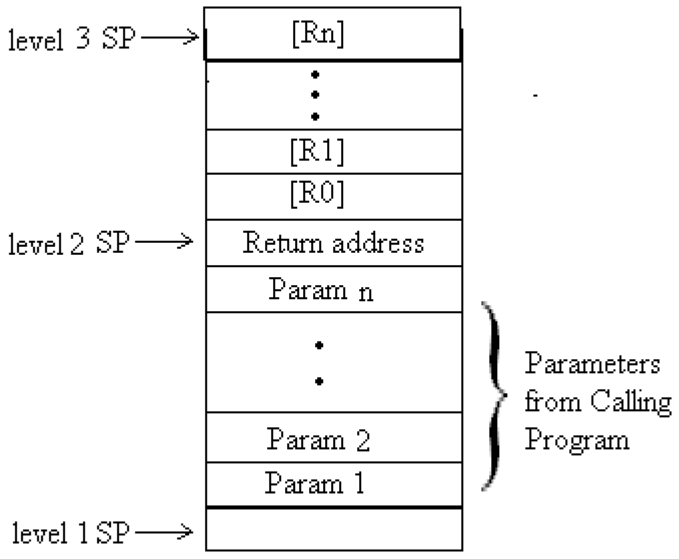
2.8.2 Parameter passing using stacks

Next we will look into the concepts of parameter passing using stacks and how exactly it is useful. Usage stack for parameter passing is more efficient than parameter passing through registers.

If too many parameters are involved, parameter passing through registers is not that efficient as there may not be enough general purpose registers. In that case it is highly flexible to use stack for passing parameters of any number. All the parameters are accessed through Stack Pointer register.

We have already seen that usage of stack providing a greater flexibility in subroutine nesting as there are multiple return addresses needs to be used. Hence it is very much essential to use a standard structure for the stack which provides access to parameters as well as to the return addresses.

2.8.3 Structure of Stack for the usage of parameter passing



1. According to the layout Stack Pointer starts from level1 which is the original top of stack. Initially parameters that are required for the subroutine from the calling program are pushed on to the stack. Instruction for that is:
   Move param, - (SP)
   Were param is parameter required for the subroutine and is pushed onto the stack using Push operation
2. Next when a call instruction calls the subroutine, it automatically pushes the return address on to the stack. At this point **SP** will be pointing at level2 as shown in the figure.
3. Once after call instruction, control enters the subroutine. In the subroutine all registers that are needed for performing the subroutine operations are selected and their contents are stored in stack for the later usage.
   For example, if registers R0, R1 and R2 are required for subroutine operation, then their contents are pushed onto the stack by the instructions shown below:
   Move R0, - (SP)
   Move R1, - (SP)
   Move R2, - (SP)

   The above instructions push the contents of R0, R1 and R2 onto the stack. This can be written using a single instruction called **Movemultiple** which pushes the contents of R0, R1 and R2 in sequence as shown below:

   Movemultiple R0 – R2, - (SP)

   At this point Stack Pointer (SP) will be pointing to the top most location of the stack which is the saved contents of register R1.

4. Then required operations are performed and the result that needs to be passed into calling program is stored in some unused location of the stack which is a location were parameters from calling program are stored.
5. Once after completing the required operation, control can return back to calling program. But before returning back, saved contents are restored back into original registers. i.e. as an example we have stored the contents of three registers R0, R1 and R2. These contents are restored back by popping the required contents and storing it on to the original register. This operation is done through the instruction shown below:
   Move (SP)+, R2
   Move (SP)+, R1
   Move (SP)+, R0
   Above instructions pop the contents of the stack and places the value in the register R0, R1 and R2 in sequence. This instruction can be replace by single instruction called **Movemultiple** which pops the contents of R0, R1 and R2 respectively. The instruction is shown below:

   Movemultiple (SP)+, R0 – R2

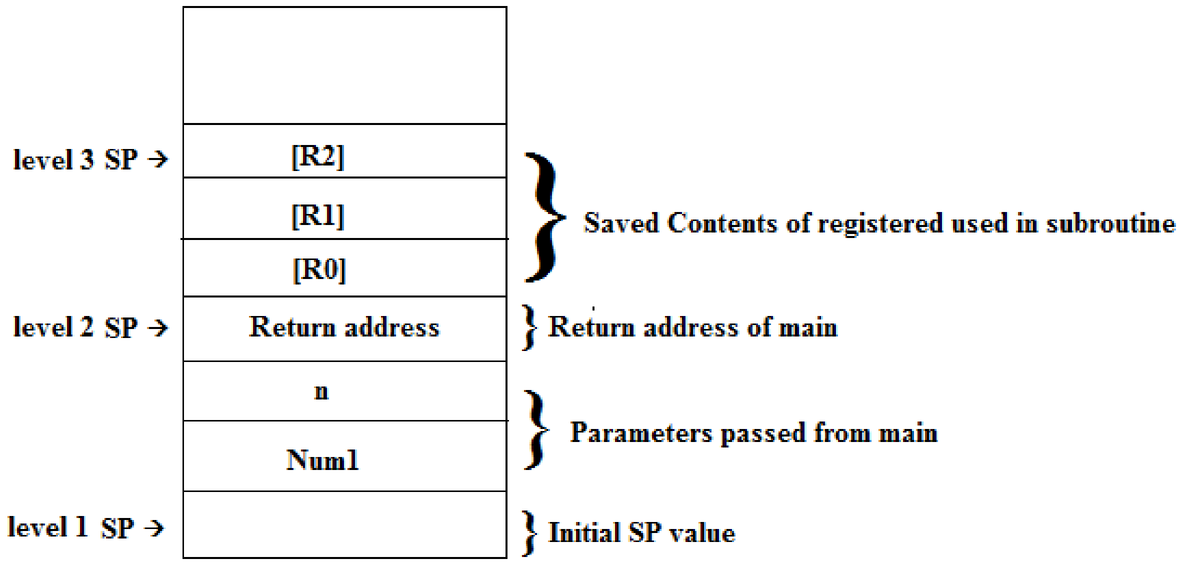
   Here first popped will be saved in R2, next R1 and finally it is into R0. When this instruction is executed SP will be pointing to the location were return address is saved.
6. Once after popping all the required contents and restoring the contents and when SP is pointing to return address, return instruction is executed in the subroutine which automatically pops the return address and SP will be pointing to the location below the return address. Now the control goes back to the calling program.
7. Once after control transfers back to the calling program, very first thing to do is accessing the result using Stack Pointer (SP) and once after accessing the result Stack Pointer (SP) is set to the original value.

2.8.4 Example

To get to know how it works we will consider the example program of adding N numbers which uses subroutine LISTADD for performing the addition of N numbers Before writing the instructions, first let us plot the structure of stack and then we will see how to write instructions according to that.

(Stack Structure in Next Page)

Stack Structure for the program adding of N numbers



→SP mentioned in three different levels specify the push operations done at different levels based on the sequence of operations for the structure of the stack.

→According to the structure, we know first parameter that needs to be passed from calling function into subroutine is pushed on to the stack. In this example, Num1 and n are the parameters that are passed from main into subroutine LISTADD.

→Once after pushing the required parameters subroutine can be called any time perform the required operation using call instruction. When subroutine is called, it automatically pushes the return address of main into the stack. This level 2 of stack operation.

→On call instruction, control transfers into the subroutine and very first thing to do in subroutine is to save the previous contents of the register that needs to be used in subroutine. In this example, we are using register R0, R1 and R2 in subroutine LISTADD. Hence push the contents of all the three registers. This is level3 of stack operation.

→Next to access any parameter required for subroutine, it is done through SP. For example, in this program to access the parameter n and NUM1 it has to make displacement of 16 and 20 bytes respectively.

→Program according to the layout is shown in next page:

Main Program

```
        Move        #Num1, - (SP)   //Push address of Num1 to stack which is a parameter
        Move        N, - (SP)       //Push value of N to stack as another parameter
        Call        LISTADD         //Call subroutine LISTADD
        Move        4(SP), Sum      //Store the result in location Sum
        Add         #8, (SP)        //Set stack pointer back to its original value
```

Subroutine LISTADD

```
LISTADD  MoveMultiple     R0 –R2, - (SP) // Save the contents of register R0 to R2
         Move             16(SP), R1     //access value of Parameter1
         Move             20(SP), R2     // access value of parameter2
         Clear            R0             //Set R0 to zero
    Loop Add              (R2)+, R0
         Decrement        R1             //decrement the value of counter
         Branch>0         Loop           //Check if R1>0 if so goto Loop
         Move             R0, 20(SP)     // Push result onto unused stack location
         MoveMultiple (SP)+, R0—R2 //Restore the contents of the register
         Return //Go back to the main program
```

2.8.5 Stack Frame

A private workspace of the subroutine in stack which includes parameters passed onto the subroutine and its local variables are called as **stack Frame.**

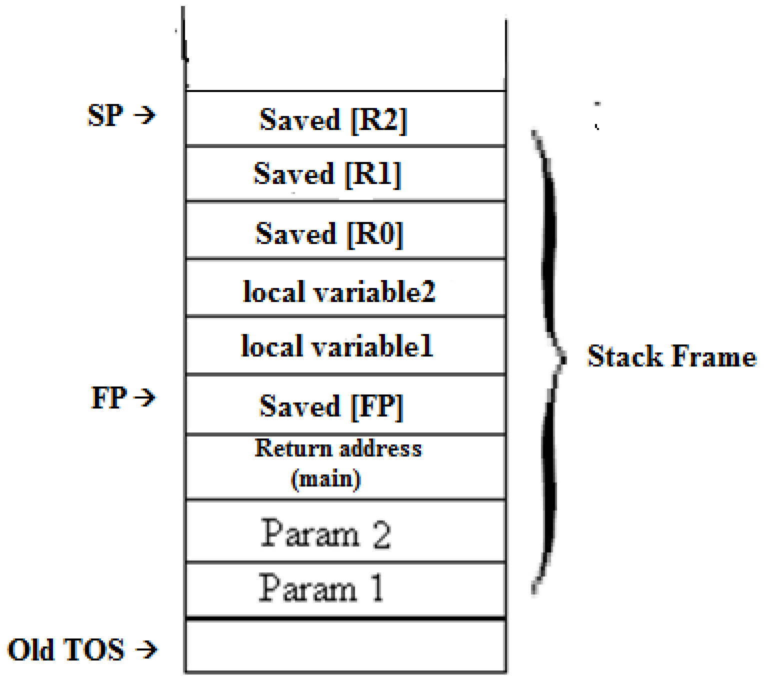
If a subroutine requires more space for local memory, then even that can be allocated a space in the stack memory which will exist till the subroutine completes its execution. This facility was not provided in previous concept of stack but is an improvement in the concept of Stack frame.

In this method for the convenient access of the parameters passed to subroutines and local variables used, it has one more pointer register other than stack pointer which is called as **Frame Pointer (FP).** FP is usually one of the general purpose register.

In this method all the required parameters and local variables are accessed with respect to Frame Pointer (FP) and not using Stack Pointer (SP)

2.8.6 Layout of Stack Frame along with Frame Pointer (FP)

To discuss the layout of the stack, let us consider that two parameters param1 and param2 are passed from **main** into subroutine. Let us assume that Subroutine uses two local variables and there registers for its operation R0, R1 and R2. In that case layout of the stack frame is shown below:



→Structure represented above shows the structure of stack using frame pointer when all the required elements are pushed onto the stack and Stack Pointer (SP) will be pointing to the top most position. Even shows the Frame Pointer pointing to one of the location of the stack. Unlike Stack Pointer, Frame Pointer doesn't change in regular intervals rather than remains fixed and changes only when a subroutine is called every time. All the required elements over here are accessed using Frame Pointer and not using Stack Pointer. Let us see the sequence of instructions according to the structure.

Note: If there are no local variables space need not be allocated for them

1. Initially parameters are pushed onto the stack from calling program in the way that is been discussed in the previous section. Here there are two parameters param1 and param2. So the instructions to push onto the stack are :

   Move param1, - (SP)

   Move param2, - (SP)

2. Next when a call instruction calls the subroutine, it automatically pushes the return address on to the stack.

3. After pushing the return address, control will automatically branch into the starting location of the subroutine and this is the point where we have to set Frame Pointer (FP) immediately when it enters the subroutine. Since FP is one of the general purpose register, its is very important to save the contents of the FP for the later usage in the program. So first we have to save their contents and then we have to set the value of SP to FP in the beginning. The instructions for doing this are:

   Move FP, -- (SP) //Save the contents of FP in the stack
   Move SP, FP //Set the Pointer FP to the value of SP

4. Next space for the local variables is allocated in the stack by reserving memory locations in the stack. Here we are using two local variables in the subroutine and 2 memory locations i.e. 8 bytes have to be kept reserved in the stack. Instruction for doing the same is:

   Subtract #8, SP // allocates space of 2 memory locations

5. Next in the subroutine all registers that are needed for performing the subroutine operations are selected and their contents are stored in stack for the later usage. Here we are assuming to use two regisetrs R0 and R1. Instruction for this operation is:

   Movemultiple R0 – R2, - (SP)

6. Then required operations are performed and the result that needs to be passed into calling program is stored in some unused location of the stack which is a location were parameters from calling program are stored.

7. Once after completing the required operation, control can return back to calling program. But before returning back, saved contents are restored back into original registers. i.e. we have stored the contents of two registers R0 and R1. These contents are restored back by popping the required contents and storing it on to the original register. This operation is done through the instruction shown below:

   Movemultiple (SP)+, R0 – R1

8. After restoring the contents of register, it deallocates local variable space allocated in the stack memory. To do this instruction written is:
   Add #12, SP

9. Now SP will be pointing to the location were contents of Frame Pointer are stored and at this point contents of Frame Pointer are restored back into FP register deactivating the Frame Pointer. This is done by the instruction
   Move (SP)+, FP

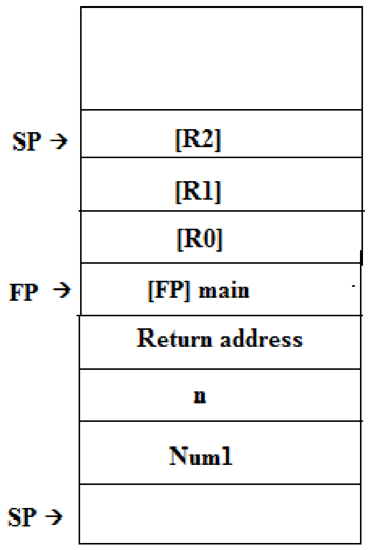
10. Now SP is pointing to return address, return instruction is executed in the subroutine which automatically pops the return address and SP will be pointing to the location below the return address. Now the control goes back to the calling program.

11. Once after control transfers back to the calling program, very first thing to do is accessing the result using Stack Pointer (SP).

12. Finally once after accessing the result Stack Pointer (SP) is set to the original value.

2.8.3 Example

To get to know how it works we will consider the example program of adding N numbers which uses subroutine LISTADD for performing the addition of N numbers Before writing the instructions, first let us plot the structure of stack using Frame Pointer and then we will see how to write instructions according to that.



→According to the structure, we know first parameter that needs to be passed from calling function into subroutine is pushed on to the stack. In this example, Num1 and n are the parameters that are passed from main into subroutine LISTADD.

→Once after pushing the required parameters subroutine can be called any time perform the required operation using call instruction. When subroutine is called, it automatically pushes the return address of main into the stack.

→On call instruction, control transfers into the subroutine and very first thing to do in subroutine is to save the Frame Pointer contents and set the Frame Pointer (FP) value.

→ Next we need to allocate space for local variables if present. In this program we do not have any local variables so there is no need to set space for them.

→ Next we need to save the previous contents of the register that needs to be used in subroutine. In this example, we are using register R0, R1 and R2 in subroutine LISTADD. Hence push the contents of all the three registers.

→Next to access any parameter required for subroutine, it is done through FP. For example, in this program to access the parameter n and NUM1 it has to make displacement of 8 and 12 bytes respectively.

→Program according to the layout is shown below:

Main Program

```
        Move        #Num1, - (SP)   //Push address of Num1 to stack which is a parameter
        Move        N, - (SP)       //Push value of N to stack as another parameter
        Call        LISTADD         //Call subroutine LISTADD
        Move        (SP), Sum       //Store the result in location Sum
        Add         #8, (SP)        //Set stack pointer back to its original value
```

Subroutine LISTADD

```
LISTADD     Move            FP, -- (SP)     //Save contents of FP in Stack
            Move            SP, FP          //Set Frame pointer
            MoveMultiple    R0 –R2, - (SP) // Save the contents of register R0 to R2
            Move            8(FP), R1       //access value of Parameter1
```

```
        Move            12(FP), R2      // access value of parameter2
        Clear           R0              //Set R0 to zero
Loop    Add             (R2)+, R0
        Decrement       R1              //decrement the value of counter
        Branch>0        Loop            //Check if R1>0 if so goto Loop
        Move            R0, 8(FP)       // Push result onto unused stack location
        MoveMultiple    (SP)+, R0—R2 //Restore the contents of the register
        Move            (SP)+, FP       //Restore contents of FP
        Return //Go back to the main program
```

***************************ALL THE VERY BEST******************************