

1.1 Basic Structure of Computers

A Computer is a fast electronic machine that accepts input information in digital form, process the input according to a set of stored instructions (called programs) and outputs the resulting information.

Computer Types

Based on size, cost, computational power and intended use computers can be classified as below:

- **Personal computer**

The most common form of desktop computers is personal computers. It has processing and storage units, visual display, audio output units and a keyboard that can be placed easily on office/home desk. Storage media include hard disk, CD ROM and diskettes.

- **Notebook Computers**

These are compact version of personal computer. All components are packaged into a single unit, with the size of thin briefcase.

- **Workstations**

Dimensions of workstations are same as that of desktop computers. They have high resolution graphics input/output capability and have more computational power than personal computers.

- **Mainframes(Enterprise systems)**

These types of systems are used for business data processing. These have more computing power and storage capacity than workstations.

- **Servers**

Contain sizable database storage units and are capable of handling large volumes of requests to access the data. These are widely accessible to the education, business and personal user communities. Requests and responses are usually transported over internet communication facilities.

- **Supercomputers**

Super computers are high end powerful computer systems. Used for large scale numerical calculations required in applications such as weather forecasting. India's first Supercomputer is PARAM 8000 developed by CDAC (Centre for Development of Advanced Computing).

1.1.1 Functional Units

A computer consist of mainly five independent functional parts

- **Input Unit**

This unit accepts information from human operators with the help of electromechanical devices such as keyboard. Whenever a key is pressed the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted over a cable to either the memory or processor. The received information can be stored in computers memory for later references otherwise it can be immediately used by the ALU circuitry to perform the desired operations.

- **Memory Unit**

It is the storage unit of the computer system. The input information is processed based on the stored instructions called programs; these programs are stored in memory unit. The

memory contains a large number of semiconductor storage cells where each cell can store one bit of information. The cells are processed in groups of fixed sizes called words, with a distinct address for each word. Addresses are simply numbers that can identify successive locations. The number of bits in each word determines word length of the computer (16 to 64 bits). Memory units are mainly classified into two types:

✓ **Primary Memory**

It is a fast memory operating at electronic speeds. Programs are stored in this memory during their execution period. Primary memory is divided into two types:

○ **RAM(Random Access Memory)**

It is a volatile memory that means the contents will be lost when the power is switched off. RAM can again be divided into two types.

▪ **Static Memory(SRAM)**

Memories that consist of circuits that are capable of retaining their states as long as the power is applied are known as static memories. These memories are designed by using transistors and inverters.

▪ **Dynamic Memory(DRAM)**

These are less expensive RAMs. The cost of static RAMs are high because of the usage of the several transistors. Dynamic RAMs are implemented with the help of a capacitor and a single transistor. Such cells don't retain their state indefinitely hence they are called dynamic RAMs.

○ **ROM(Read Only Memory)**

This is one of the major types of memory used in personal computers. ROM is a type of memory that normally can only be read as opposed to RAM which can be both read and written. It is a non-volatile memory that its contents will retain even when the power is switched off. ROM can again be divided into three types:

▪ **Programmable ROM(PROM)**

This is a type of ROM that can be programmed using special equipment; it can be written to, but only once. This is useful for companies that make their own ROMs from software they write, because when they change their code they can create new PROMs without requiring expensive equipment. This is similar to the way a CD-ROM recorder works by letting you "burn" programs onto blanks once and then letting you read from them many times. In fact, programming a PROM is also called burning, just like burning a CD-R, and it is comparable in terms of its flexibility.

▪ **Erasable Programmable ROM(EPROM)**

An EPROM is a ROM that can be erased and reprogrammed. A little glass window is installed in the top of the ROM package, through which you can actually see the chip that holds the memory. Ultraviolet light of a specific frequency can be shined through this window for a specified period of time, which will erase the EPROM and allow it to be reprogrammed again. Obviously this is much more useful than a regular PROM, but it does require the erasing light. Continuing the "CD" analogy, this technology is analogous to a reusable CD-RW.

- **Electrically Erasable Programmable ROM(EEPROM)**

The next level of erasability is the EEPROM, which can be erased under software control. This is the most flexible type of ROM, and is now commonly used for holding BIOS programs. When you hear reference to a "flash BIOS" or doing a BIOS upgrade by "flashing", this refers to reprogramming the BIOS EEPROM with a special software program. Here we are blurring the line a bit between what "read-only" really means, but remember that this rewriting is done maybe once a year or so, compared to real read-write memory (RAM) where rewriting is done often many times per second.

✓ **Secondary Memory**

This memory is used when large amount of data and programs have to be stored. Compared to the primary memory these are cheaper but performs at low speed. Examples are CD ROMS, Magnetic disk, USB drives etc.

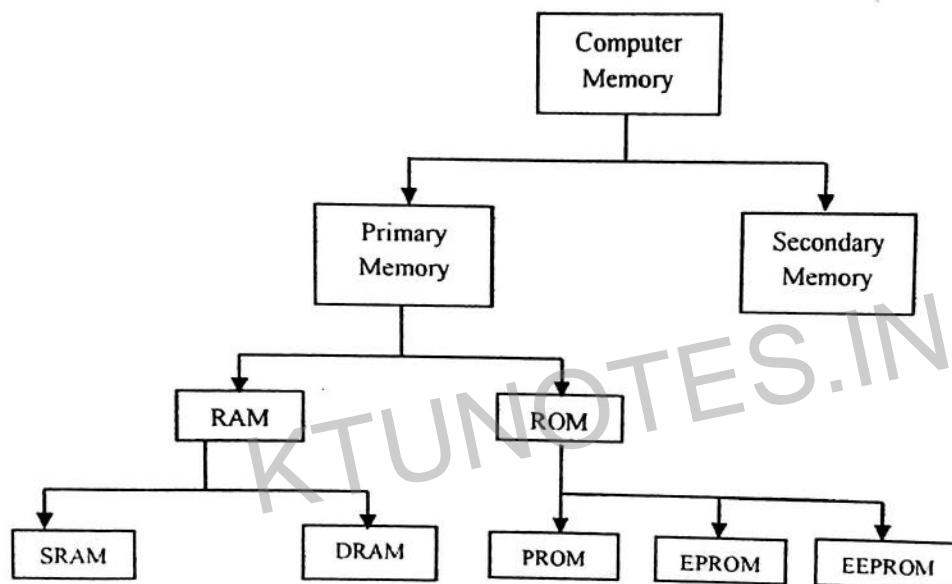


Fig 1.1: Classification of Memories

- **Arithmetic and Logic unit**

This is the main part of the processing unit of a computer. The desired operations are performed by this unit. Arithmetic and logic operations can be separated with the help of a mode selector. When the mode selector bit is zero it performs arithmetic operations and performs logical operations when the bit is one. To perform the operation the required operands have to be brought into the processor, where they are stored in registers.

- **Control unit**

This is one of another core part of a processing unit of a computer. It is known as the nerve centre of a computer system. It coordinates the operation of all other units in computer system. It sends the control signals to other units and senses their states. Example of control signals are read, write etc.

- **Output unit**

The processed results are sent to the outside world through output device. Example: Printer.

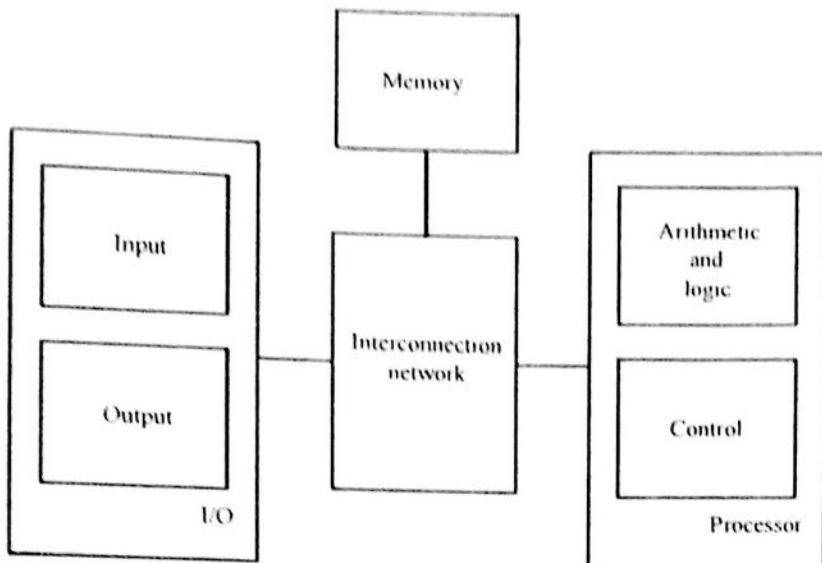


Fig 1.2: Basic Functional units of a computer

1.1.2 Basic Operational Concepts

In order to execute an operation in a processor the required instructions have to be brought out from the memory to the processor. Transfers between memory and processor are started by sending the address of the memory location to be accessed to the memory unit and issuing the appropriate control signals. Then the data are transferred to or from the memory. The following figure shows the connection between the memory and the processor.

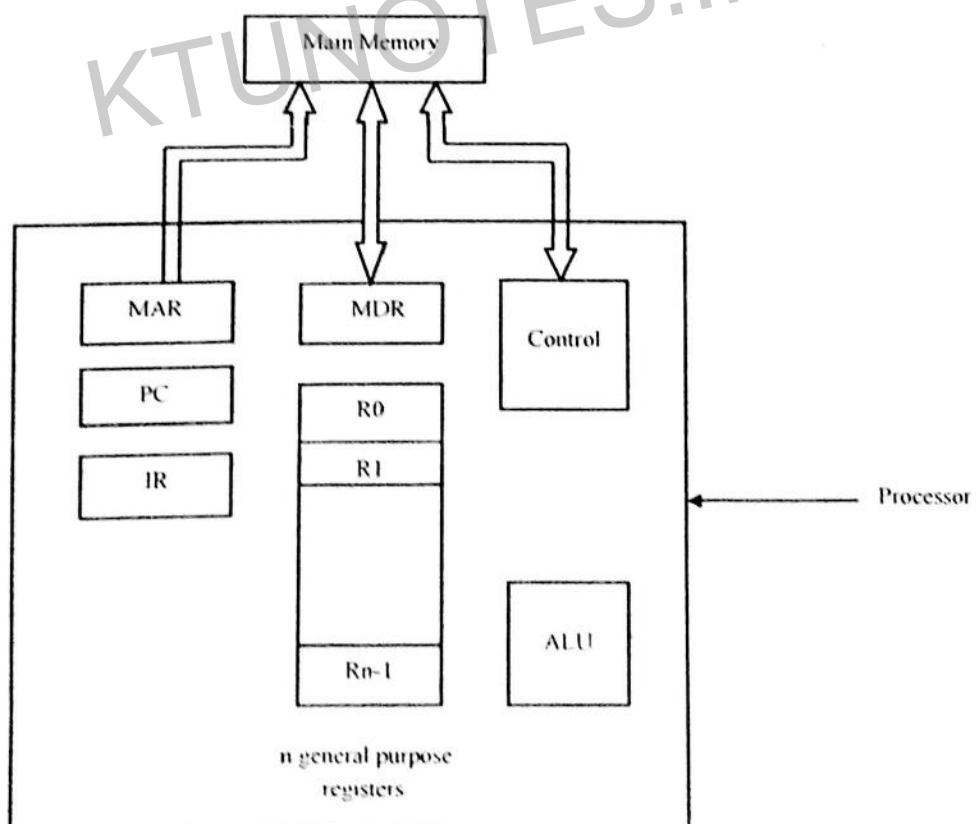


Fig 1.3: Connection between the processor and main memory

Memory Address Register (MAR) and Memory Data Register (MDR) are the two registers which are facilitating the communication between the processor and memory. In order to read an information from memory the address of the memory location in which the information is residing have to be put in MAR and a Read control signal will be sent to memory unit. When the memory unit sees the address in address line of the bus and read signal in control line of the bus memory will start the read operation from the concerned address and the result will be sent to the MDR. From there the information can be transferred to any other registers inside the processor through internal processor bus. Similarly for Write operation initially the data to be written into the memory has to be placed in MDR and the address in which the desired information have to be kept in memory will be placed in MAR and a Write control signal will be sent to the memory unit. When the memory unit sees the address, data and write signal in the external memory bus it will start the corresponding write operation.

Other than MAR and MDR few registers like PC(Program Counter), IR(Instruction Register) and some general purpose registers R_0, R_1, \dots, R_{n-1} are there within the processor unit. PC is a register which holds the address of the next instruction to be fetched. Initially PC will be assigned by the starting program address and after fetching of each instruction it will be incremented by a size of word byte. IR is a register which holds the decoded instruction to be executed.

Normal execution of programs may be pre-empted if some device requires urgent servicing. In order to deal with the situation immediately the normal execution of the current program must be interrupted. To do this the device raises an interrupt signal. An interrupt is nothing but it is a request from an I/O device for service by the processor. The processor executes an interrupt service routine (ISR) to service the same. Before servicing an interrupt the current state of the processor must be saved in memory locations. After ISR is completed the state of the processor is restored so that the interrupted program may continue.

1.1.3 Bus Structures

Different functional units of a computer can be connected using a structure called bus system structure. There are external bus structures and internal bus structures. Bus interface between memory and processor is called external memory bus and bus structure inside the processor is called internal bus structures. Bus consists of three lines of carries, one for holding address, one for holding data and one for carrying control information.

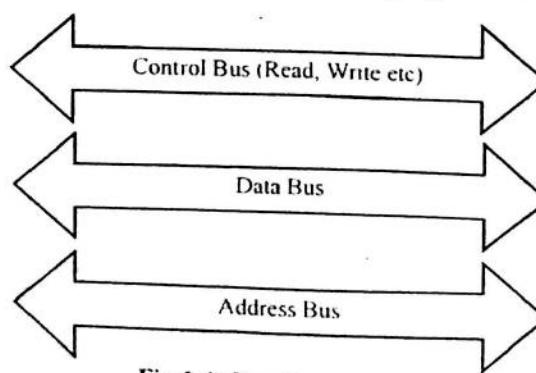


Fig 1.4: Bus Structure

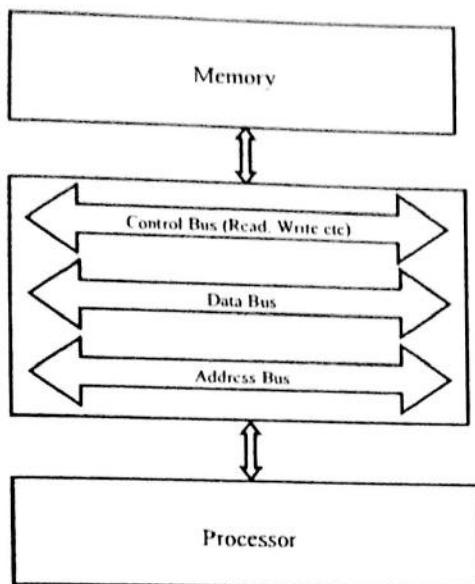


Fig 1.5: External processor bus

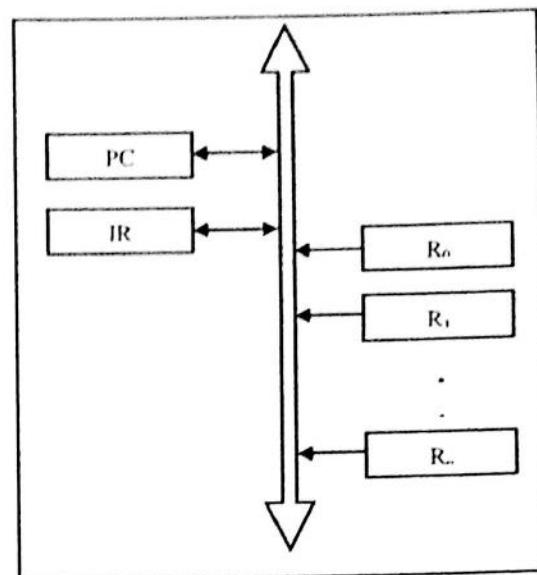


Fig 1.6: Internal processor bus

Bus Structures can be classified into two:

- **Single bus structure**

Only two units can actively participate at a time, that is only one transfer takes place at a time.

Advantages

- Low cost
- Flexibility for attaching peripheral devices

Disadvantages

- Only one transfer at a time

- **Multiple bus structure**

It contains multiple buses, so that more than one transfer can take place.

Advantages

- More concurrency in operations.
- Better performance

Disadvantages

- Increased cost

The different units connected to the processor are having different speeds. These timing differences can be smoothed by including buffer registers with the devices to hold information during transfers. Once the buffer is full, the device can start the operation without further intervention by the bus and the processor.

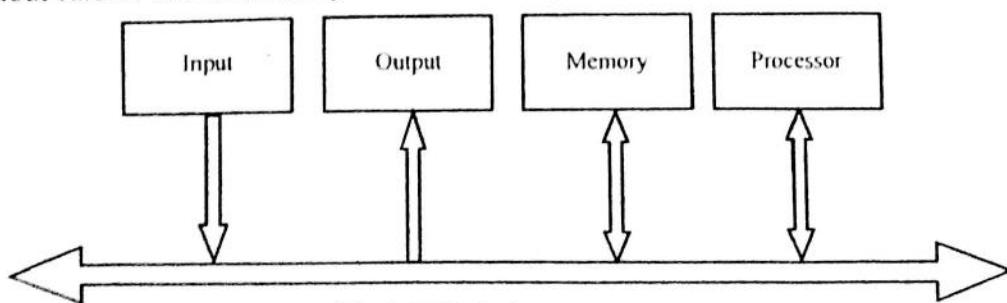


Fig 1.7: Single bus structure

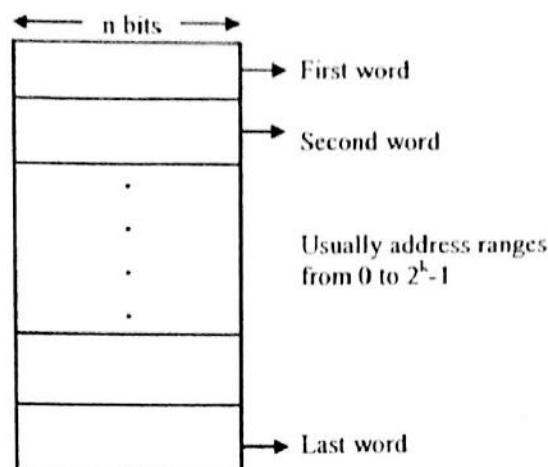


Fig 1.9: Memory Address

Note

16 bit address: - Creates an address space of 2^{16} addresses

32 bit address: - Creates an address space of 2^{32} addresses

Byte Addressability

Basic information quantities are, bit, byte and word. Byte is of size 8 bit (always this is constant). Word length may range from 16 to 64 bits. So normally distinct addresses will be assigned to each byte location. This is known as byte addressability.

For example, if word length is 16 bits, successive words are located at addresses 0 and 4. If 32 bits, successive words will be located at addresses 0, 4, 8 ..., with each word consisting of four bytes.

Big-Endian and Little-Endian Assignments

Byte addresses can be assigned across words in two ways:

- **Big – Endian**

Lower byte addresses are used for most significant bytes of the word.

- **Little – Endian**

Lower byte addresses are used for least significant bytes of the word.

Word Address	Byte Address			
0	0	1	2	3
4	4	5	6	7

2^{k-4}	2^{k-4}	2^{k-3}	2^{k-2}	2^{k-1}

Fig 1.10(a): Big-Endian Assignment

Word Address	Byte Address			
0	3	2	1	0
4	7	6	5	4

2^{k-4}	2^{k-1}	2^{k-2}	2^{k-3}	2^{k-4}

Fig 1.10(b): Little-Endian Assignment

Word Alignment

The words are said to be aligned in memory if they begin at a byte address that is a multiple of number of bytes in a word. The number of bytes in a word is power of 2.

Example

If word length is 16 bits, aligned words begin at byte addresses 0, 4, 8....

If word length is 64 bits, aligned words begin at byte addresses 0,8,16....

Words are said to have unaligned addresses if the words begin at arbitrary byte address.

Accessing Numbers, Characters and Character Strings

A number can be accessed in the memory by specifying its word address. Individual characters can be accessed by their byte address. Character Strings can be of variable length. The beginning of the string is indicated by giving the address of the byte containing the first character. A successive byte location contains successive characters of the string. End of string (a special control character) can be used as the last character in the string.

1.3 Memory Operations

To execute an instruction, the words containing the instruction have to be brought out to the processor from memory. Operands and results also have to be moved in between of main memory and processor. Main operations are:

- **Load(Read or Fetch)**

Transfer copy of the contents of a specific memory location to the processor. The memory contents remain unchanged.

- **Store(Write)**

It transfers information from processor to memory. It will destroy the earlier content of the memory location. Information can be transferred between processor and memory in terms of bytes or words. One byte or one word can be transferred in a single operation.

1.4 Instructions and Instruction Sequencing

A computer must have instructions capable of performing four type of operations:

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

➤ Register- Transfer Notation

Information can be transferred to and fro between memory locations, processor registers and registers with I/O. We can identify a location by a symbolic name.

Example

LOC, PLACE, A (Address of memory locations)

R0, R1, R5 (Processor register names)

DATA IN, OUT STATUS (I/O Registers)

Register- Transfer Examples

$$R1 \leftarrow [LOC] \quad (1)$$

The contents of the memory location are transferred to processor register R1. Square bracket indicates the contents of the location.

$$R3 \leftarrow [R1] + [R2] \quad \text{--- (2)}$$

Adds the contents of registers R1 and R2 and places the sum in register R3. The type of notations (1) and (2) are known as register transfer notation.

➤ Assembly Language Notation

The register transfer notation $R1 \leftarrow [LOC]$ can be notated in assembly language as:

Move LOC, R1

Here the content of LOC is unchanged, R1 will be over written. Similarly,

$$R3 \leftarrow [R1] + [R2]$$

This can be notated in assembly language as

Add R1, R2, R3

➤ Basic Instruction Types

Consider the statement $C = A + B$. To execute this statement, the operands A and B have to be fetched from memory to the processor. ALU computes the operation and the result will be sent back to memory and stored in location C.

➤ Three Address Instructions

Syntax

Operation source1, source2, destination

Example

Add A, B, C

Where, A and B are source operands and C is destination operand. Add is the operation to be performed on operands. Suppose that K bits are needed to specify the memory address of each operand, and then totally $3K$ bits are needed totally to specify the memory address of all operands in the above sample case. In addition, some bits are needed to denote Add instruction also. Three address instruction is too large to fit in one word for most cases. An alternative approach is to use two address instructions.

➤ Two Address Instructions

Syntax

Operation source, destination

Example

Add A, B

Which performs the operation $B \leftarrow [A] + [B]$. Square bracket indicates the contents of the location specified. That is, it adds the contents of A and B and the result is stored back to B. Here the value of location B will be overwritten. To preserve the value of B, we can go for another instruction Move B, C. It moves the content of B to C, leaving the

contents of location B unchanged. We couldn't find an alternative approach by using a single two address instruction.

Add A, B → Move B, C	
Add A, C	
(Single two address instruction)	(Leave B unchanged)

Here, in all instructions we are adopting a scheme such that source operand is specified first then destination operand. This may not be the case with all architectures. In some of the machine instructions, it will follow a scheme of destination first, and then source. There is no uniform scheme for specifying operands. Even two address instructions also, may not be fit into one word for usual word length. Another possibility is to have machine instructions with single operand. These are called One Address Instructions.

➤ One Address Instructions

Here one of the register called Accumulator is implicit in all cases.

Example

Add A

Add the content of memory location A to the content of the accumulator register and places the sum back to accumulator.

Question

Represent $C \leftarrow [A] + [B]$ in terms of one address instruction

Ans

Load A
Add B
Store C

Load instruction copies the contents of memory location A into accumulator. Add B, adds the content of B to Accumulator and Store C, and stores the content of accumulator to memory location C. Early computers have designed with only a single accumulator. But now, modern computers are coming with so many general purpose registers (R1, R2,.....R_i,....R_n). To perform an operation, normally the data can store in registers rather than taking from memory, so that faster processing may occur. Instructions size can also be shorten, because register addressing can be done with fewer bits. Means,

32 bit computer → 2^{32} memory locations as possible

That is one word location can be identified by a 32 bit address. But for register addressing, consider 32 general purpose registers only 5 bits are needed to identify a register ($2^n=32$, n=5).

reduced to

32 bit → 5 bits (If register is holding operand in an instruction)

Example

Add R _i , R _j
Add R _i , R _j , R _k

These types of instructions may normally fit into one word.

➤ Data transfer between different locations

Instructions used for data transfer between different locations are **Move**.

Syntax

Move Source, Destination

Example

Move A, Ri → same as Load A, Ri

Move Ri, A → same as Store Ri, A

Question

Write the instruction sequence for $C = A + B$ (suppose that arithmetic operations are allowed only on register operands)

Answer

Move A, Ri

Move B, Rj

Add Ri, Rj

Move Rj, C

Question

Write the instruction sequence for $C = A + B$ (suppose that one operand in memory, other in register)

Answer

Move A, Ri

Add B, Ri

Move Ri, C

➤ Zero Address Instructions

Instructions can be used with zero operands in which the locations of all operands are specified implicitly. (It is possible by storing the operands in a structure called pushdown stack).

➤ Instruction Execution and Straight line sequencing

This section deals with the flow of execution of a program. Consider a sample memory space for the program $C \leftarrow [A] + [B]$

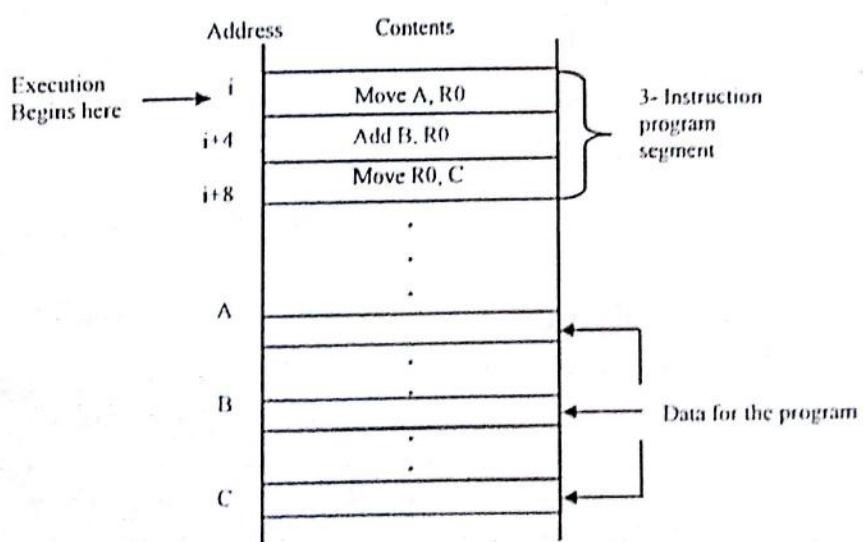


Fig 1.11: Program execution

Initially PC (Program Counter) contains the address of the next instruction to be executed. In this example initially address i is placed in PC. The processor control circuitry use the information in PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called straight line sequencing .While the instruction is being executed, the value of PC will be updated by incrementing 4 bytes, (because here length of instruction is 4 bytes). Instruction execution consists of 2 phases:

- **Instruction Fetch**

Instruction is fetched from memory location whose address is in PC. This instruction is placed in IR (Instruction Register).

- **Instruction Execute**

Instruction in IR (Instruction Register) is analyzed to see which operation have to be performed by the processor. This may include several operations like fetching of operands from memory (or from processor registers), performing an ALU operation, storing of result into memory etc.

After the execution phase, PC will contain the address of the next instruction to be executed. Then a new instruction fetch phase will begin.

➤ **Branching**

Consider a program of adding a list of n numbers ($\text{num1}, \text{num2}... \text{num } n$) and stores the result in memory location sum. By straight line sequencing approach, it can be done as follows:

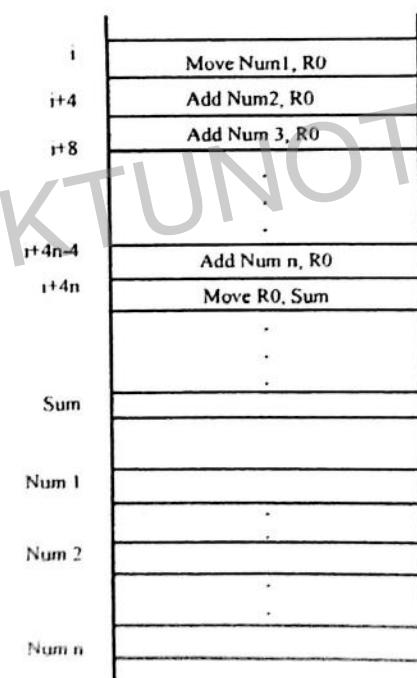


Fig 1.12: Straight Line Sequencing Program for adding 'n' numbers

Initially the value of first number (Num1) is moved to R0. Then it will be added with Num2 and result is stored in R0.Num3 will be read next and added with R0 (R0 now contains num1+num2+num3).The process will be continued with n numbers. After this, R0 contains a value (Num1+Num2+.....Num n).We have to store the result into location Sum. So, we are in need of an instruction,

Move R0, Sum

Here, a long list of add instruction is there. To remove this, we are going for a looping concept. The loop is a straight line sequence of instructions executed as many times as needed. It starts at the location Loop and ends at the instruction Branch>0.

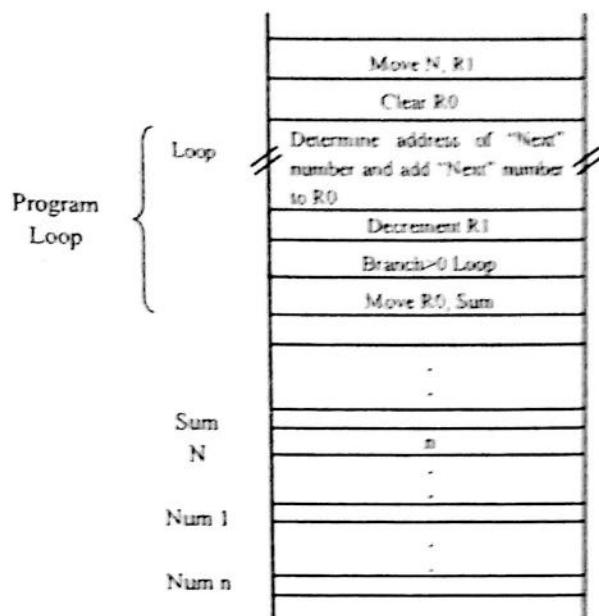


Fig 1.13: Using loop to add 'n' numbers

No of entries in the list (n) is stored at memory location N. After each addition operation this value is decremented by one. This value is a number greater than zero that means again there are numbers to be added to the list.

Branch Instruction loads a new value into the program counter .The processor fetches and executes the instructions from these addresses (known as branch target) instead of loading the instruction from address of PC. A conditional branch instruction causes a branch only if the specified condition is satisfied. In other cases, PC is incremented in the normal way and the next instruction in sequential address order is fetched and executed.

➤ Condition Codes

Whenever a conditional branch instruction is executed, it may require the results of various operations (to check the condition). Processors keeping this information in individual bits called condition code flags. These flags are grouped together in a special processor register called condition code Register or Status register.

Four commonly used flags are:

- N (Negative) → Set to 1 if result is negative, otherwise zero.
- Z (Zero) → Set to 1 if result is zero, otherwise cleared to zero.
- V (Overflow) → Set to 1 if arithmetic overflow occur, otherwise cleared to zero.
- C (Carry) → Set to 1 if a carry after the operation, otherwise cleared to zero.

In the above example, Branch>0 tests the condition code flags N and Z .That is ,the branch is taken only if register R either contain a negative or zero value.

1.5 Addressing Modes

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

Generic Addressing Modes

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

➤ Implementation of Variables and Constants

▪ Variables

The value can be changed as needed using the appropriate instructions. There are 2 accessing modes to access the variables. They are

- **Register Mode**
The operand is the contents of the processor register. The name (address) of the register is given in the instruction.
- **Absolute Mode (Direct Mode)**
The operand is in new location. The address of this location is given explicitly in the instruction.

Example

MOVE LOC, R2

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

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

Where, EA is Effective Address

▪ Constants

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

➤ Immediate mode

The operand is given explicitly in the instruction.

Example

Move 200 immediate, R0

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

Move #200, R0

Assembly Syntax

Immediate #value

Addressing Function

Operand =value

▪ **Indirection and Pointers**

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

➤ **Indirect Mode**

The effective address of the operand is the contents of a register. We denote the indirection by the name of the register or new address given in the instruction.

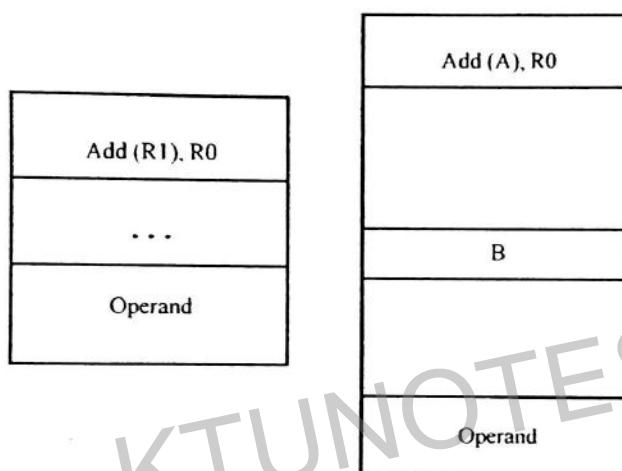


Fig 1.14: Indirect mode

Address of an operand (B) is stored into R1 register. If we want this operand, we can get it through register R1 (indirection). The register or new location that contains the address of an operand is called the pointer.

Mode

Indirect

Assembler Syntax

Ri, LOC

Addressing Function

EA=[Ri] or EA=[LOC]

➤ **Indexing and Arrays**

▪ **Index Mode**

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

$X(Ri)$

Where,

X – denotes the constant value contained in the instruction

Ri – It is the name of the register involved

The Effective Address of the operand is,

$$EA=X + |Ri|$$

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

To find operand,

1. First go to Reg R1 (using address)-read the content from $R1=1000$
2. Add the content 1000 with offset 20 get the result.

$$1000+20=1020$$

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

Example

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

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

➤ **Relative Addressing**

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

- **Relative Mode**

The Effective Address is determined by the Index mode using the PC in place of the general purpose register (gpr). This mode can be used to access the data operand. But its most common use is to specify the target address in branch instruction.

Example

Branch>0 Loop

It causes the program execution to goto the branch target location. It is identified by the name loop if the branch condition is satisfied.

Mode	Assembler Syntax	Addressing Function
Relative	X(PC)	EA=[PC]+X

➤ **Additional Modes**

There are two additional modes. They are

- **Auto-increment mode**

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

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

- **Auto-decrement mode**

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

Mode	Assembler Syntax	Addressing Function
Auto-decrement	(-Ri)	EA-[Ri]; Decrement Ri

1.6 ARM

Advanced RISC Machine (ARM) Limited has designed a family of microprocessors. All ARM processors share the same machine instruction set.

Registers, Memory access and Data transfer

In ARM architectures, memory is byte addressable using 32 bit addresses. Processor registers are also 32 bits long. In moving of data between processor registers and memory, operand length may be 8 bit (byte) or words (32 bit). Both little Endian and big Endian addressing schemes are supported. Memory is accessed only by Load and Store instructions. All arithmetic and logic instructions operate only on data in processor registers. This arrangement is a basic feature of RISC Architectures.

Register Structure

There are sixteen 32 bit registers labelled R0 to R15. R0 to R14 are general purpose registers and one is dedicated as a program counter (PC). General purpose registers can hold either memory operands or data operands. The Current Program Status Register (CPSR) or simply status register holds the condition code flags, interrupt disable flags and processor mode bits, as described in the following figure.

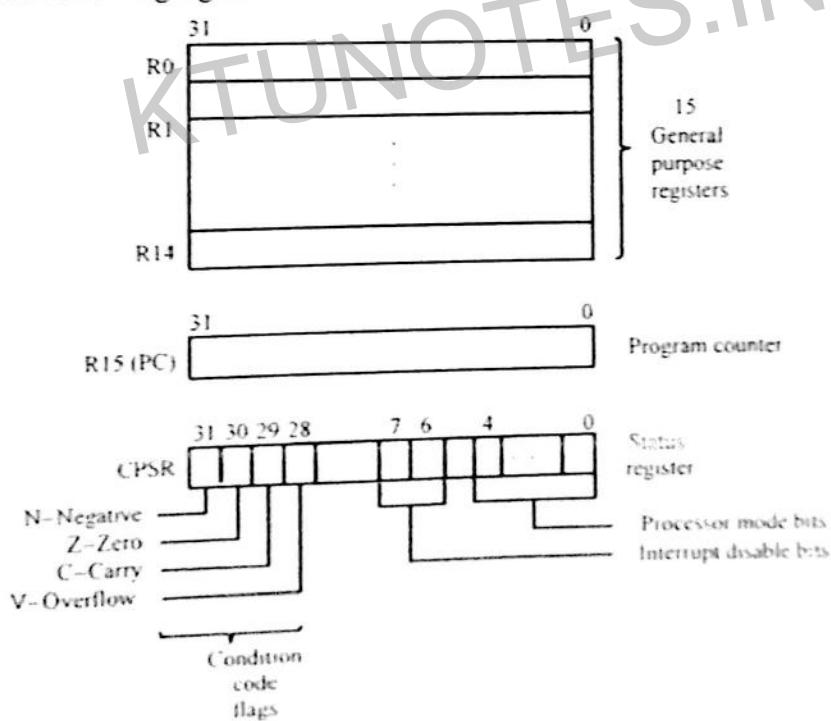


Fig 1.15: ARM Register Structure

There are 15 additional general purpose registers called the banked registers. They are duplicates of some of the R0 to R14 registers. They are used when the processor switches into supervisor mode.

Memory Access Instructions and addressing modes

In ARM, access to memory is provided with only Load and Store instructions. The basic encoding format is shown as in the following figure:

31	28 27	20 19	16 15	12 11	4 3	0
Condition	OP code	Rn	Rd	Other info		Rm

Fig 1.16: ARM instruction format

- Conditional execution of instructions

Unlike others, in ARM processors all instructions are conditionally executed, depending on the condition specified in the instruction. Instruction is executed only when the condition flag is true. Otherwise the processor proceeds to the next instruction. One of the conditions is used to indicate that the instruction is always executed.

- Memory addressing modes

For addressing memory operands one of the basic method is generate an EA(Effective Address) of the operand by adding a signed offset to the contents of the base register Rn(which is specified in the instruction).The magnitude of offset may be either an immediate value or the contents of the register Rm.

Examples

LDR Rd, [Rn, #offset]

It performs the operation

$Rd \leftarrow [[Rn] + \text{offset}]$

LDR Rd, [Rn, Rm]

It performs the operation

$Rd \leftarrow [[Rn] + [Rm]]$

If a negative offset is used, Rm must be preceded by a minus sign. An offset of zero doesn't have to specify explicitly. That is,

LDR Rd, [Rn]

It performs the operation

$Rd \leftarrow [[Rn]]$

A byte operand can be moved by using the Opcode LDRB. Similarly Store has the mnemonics STR and STRB.

Example

STR Rd, [Rn]

It performs the operation

$[Rn] \leftarrow [Rd]$

Generally we can define three addressing modes in ARM processors.

- Pre-indexed mode:

Effective address of the operand is the sum of contents of base register Rn and an offset value.

- Pre-indexed with write back mode:

It is working in the same way as pre-indexed mode except that effective address is written back to Rn.

- Post-indexed mode:

The effective address of the operand is the contents of Rn. The offset is then added to this address and the result is written back into Rn.

Register Move Instructions

To copy the contents of register Rm into register Rd, ARM uses the following instruction

MOV Rd, Rm

To load an immediate value in the register Rd, the instruction will be

MOV Rd, #immediate value

Example

MOV R0, #70

Places the value 70 in register R0.

Arithmetic and Logic Instructions

ARM instruction set has a number of arithmetic and logic operations. The operands may be in general purpose registers or may give as an immediate operand. Memory operands are not allowed in these instructions.

- **Arithmetic Instructions**

The general format for arithmetic instruction is,

OP code Rd, Rn, Rm

Operation specified by the OP code is performed on operands in general purpose registers Rn and Rm. The result is placed in register Rd.

Example

ADD R0, R2, R4

Adds the content of R2 and R4 and places the sum in register R0.

It performs the operation, $R0 \leftarrow [R2] + [R4]$

ADD R0, R3, #17

Adds the content of R3 and 17 and stores the sum in R0.

It performs the operation, $R0 \leftarrow [R3] + 17$

The immediate value is contained in the 8 bit field on bits b₇₋₀ of the instruction. The second operand can be shifted or rotated before being used in the instruction. When a shift or rotation is required, it is specified last in the assembly language expression for the instruction.

Example

ADD R0, R1, R5, LSL #4

The second operand contained in register R5 is shifted left 4 bit positions and it is then added to the contents of register R1 and sum is placed in Register R0. Two versions of multiply instructions are there.

1. Multiplies the contents of two registers and places the low order 32 bits of the product in a third register. Higher order bits of the product, if any, are discarded.

MUL R0, R1, R2

It performs the operation, $R0 \leftarrow [R1] * [R2]$

2. Second version called Multiply Accumulate specifies a fourth register whose contents are added to the product before storing the result in the destination register.

MLA R0, R1, R2, R3

It performs the operation, $R0 \leftarrow [R1] * [R2] + [R3]$

This method is often used in numerical algorithms for digital signal processing.

1.7 Basic I/O Operations

Input (I)/Output (O) operations are essential in a computer system, because data have to be transferred from memory of a computer to the outside world. The way in which I/O is performed has a significant role in the performance of a computer system.

Consider a task that reads a character from the keyboard and produces character output on a display screen. A simple way of performing such I/O task is to use a method known as program controlled I/O. Whenever a key is pressed on the keyboard, that character code has to be moved to the processor. Similarly, for display the same, that character code have to be moved from processor to display device. But when this transfer takes place, processor is very fast compared to the I/O device (keyboard and display). So, some sort of synchronization mechanism we are in need off. A solution to this problem is as follows:

The processor waits for a signal from the keyboard indicating that a character key has been struck and that its code is available in some buffer register associated with the keyboard. This register is known as DATAIN (8 bit buffer register). A status flag SIN is used to signal the processor. A program monitors SIN value, and when SIN is set to 1, the processor reads the contents of DATAIN. When the character is transferred to the processor, SIN is automatically cleared to 0.

Similarly, with the output device, processor sends the first character and then waits for a signal from the display to know that character has been received. A buffer register DATAOUT and a status control flag SOUT is used for this transfer. When SOUT is 1, the display is ready to receive the character. A program monitors the value of SOUT and when this is equal to 1, processor transfers a character code to DATAOUT. Transfer of a character to DATAOUT clears SOUT to 0. When the display device is ready to receive a second character, SOUT is again set to 1. The above specified I/O transfers are accomplished with the help of machine instructions.

A processor can monitor SIN flag and transfer character from DATAIN to register R1 with the following sequence of operations.

READWAIT	Branch to READWAIT if SIN=0 Input from DATAIN to R1
----------	--

Analogous sequence of operations for transferring output to the display is

WRITEWAIT	Branch to WRITEWAIT if SOUT=0 Output from R1 to DATAOUT
-----------	--

Branch operation is normally implemented by two machine instructions. The first instruction checks the status flag and the second instruction performs branch. Status flags are monitored by executing a short wait loop. We assume that initial state of SIN is 0 and SOUT is 1. This will be done by the device control circuits.

If the scheme used is memory mapped I/O instead of program control I/O, it is possible to use the same instruction set used for memory access. In such cases the contents of DATAIN and DATAOUT can be moved to R1 by using the instruction Move. (Load, store etc can also be used) as in the following instruction.

Movebyte DATAIN, R1

Similarly contents of DATAOUT can be moved to R1 as,

Movebyte R1, DATAOUT

Status flag SIN and SOUT are automatically cleared when the buffer registers are referenced. (Note: Difference between Move and Movebyte is in Move operand is word operands and for Movebyte operand size is byte.) Status flags also can addressed as part of memory address space by assigning distinct addresses. But the common practice is including SIN and SOUT in device status registers. Bit b3 in registers INSTATUS and OUTSTATUS is used for this purpose.

Read operation can now be implemented with the machine instruction sequence as shown below:

READWAIT	Testbit #3, INSTATUS
	Branch=0 READWAIT
	Movebyte DATAIN, R1

Write operation may implemented as,

WRITEWAIT	Testbit #3, OUTSTATUS
	Branch=0 WRITEWAIT
	Movebyte R1, DATAOUT

Testbit instruction checks the value of status flags. If the value of test bit is 0, then the condition of the branch instruction is true, and a branch is made to the beginning of the wait loop. When the device is ready (at that time bit tested will be equal to 1) the data are read from the input or written into the output buffer.

	Move	#LOC, R0	Initialize pointer register R0 to point to the address of the first location in memory Where the characters are to be stored
READ	TestBit Branch=0 MoveByte	#3,INSTATUS READ DATAIN, (R0)	Wait for a character to be entered in the keyboard buffer DATAIN Transfer the character from DATAIN into the memory (this clears SIN to 0)
ECHO	TestBit Branch=0 MoveByte Compare Branch/0	#3, OUTSTATUS ECHO (R0), DATAOUT #CR, (R0) RFAD	Wait for the display to become ready. Move the character just read to the output buffer register (this clears SOUT to 0) Check if the character just read is CR (carriage return) If it is not CR, then branch back and read another character Also increment the pointer to store the next character

Table 1.1: A program that reads a line of characters and displays it

1.8 Stack Subroutine Calls

In this section we are dealing with the concepts of subroutines, stacks and how a subroutine call can be performed with the help of stack data structure.

Stacks

Stack is a data structure which contains a list of data elements usually words or bytes, with the accessing restriction that elements can be added or removed at one end of the list only. This end is called top of the stack, and the other end is called the bottom. The structure is also referred as push down stack or LIFO (Last In First Out) stack. The operations on stack are push (inserting an item into the stack) and pop (remove an item from the stack). Data stored in the memory of a computer can be organized as a stack, with successive elements occupying successive memory locations. First element is placed in location BOTTOM and when new data are added they are placed in successively lower address locations. Here we are considering a stack which grows in the direction of decreasing memory addresses.

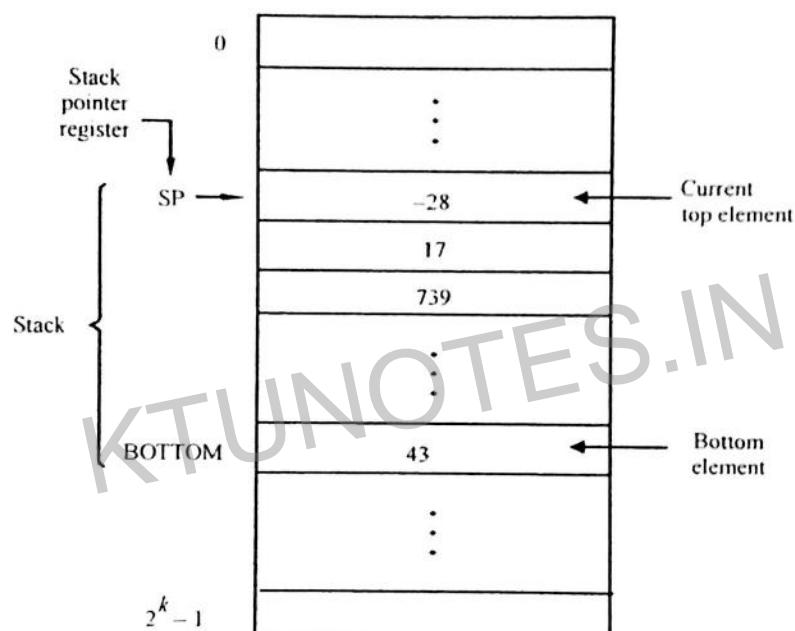


Fig 1.18: A stack of words in the memory

A processor register is used to keep track of the address of the element of the stack that is at top at any given time. This register is called stack pointer (SP). Push and Pop operations can be implemented as follows (Assume that 32 bit word length that is 4 bytes word length)

Push:	Subtract	#4, SP
	Move	NEWITEM, (SP)
Pop:	Move	(SP), ITEM
	Add	#4, SP

If the processor has auto increment and auto decrement addressing modes, the above operations can be replaced with a single instruction as follows:

Push:	Move	NEWITEM, - (SP)
Pop:	Move	(SP) +, ITEM

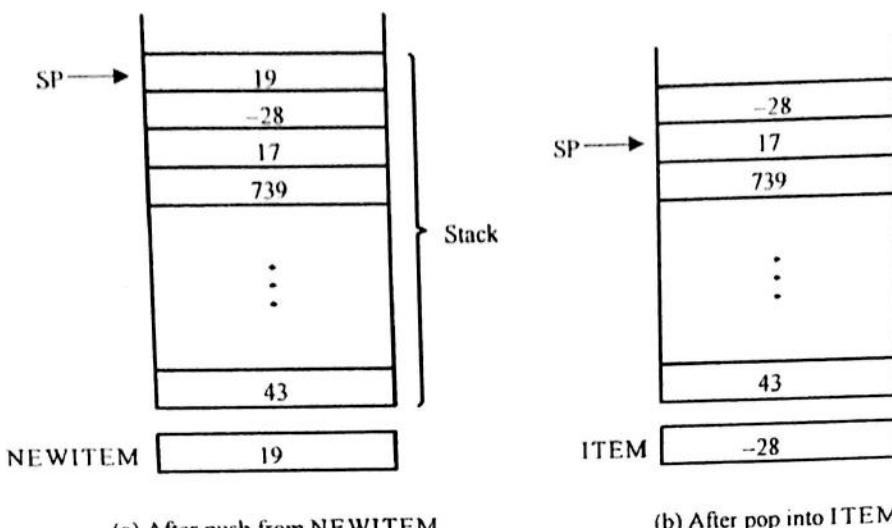


Fig 1.19: Effect of stack operations on the stack in fig 1.18

When a stack is used in a program, it is usually allocated a fixed amount of space in the memory. In this case, we have to avoid pushing a data item when the stack has reached its maximum size. Similarly always have to avoid a pop operation from an empty stack (otherwise it will lead to program error).

Subroutines

A computer program often needs to perform a particular subtask many times with different data values. This subtask is known as subroutines. Subroutine may consist of a block of instructions. At every place of the program where this subroutine is needed we can place these entire block of instructions. But this will waste more space. So, to save space only one copy of the instructions that constitute the subroutine is placed in the memory. Any program that requires the use of this subroutine simply branches to the starting location of subroutine. When a program branches to a subroutine it is known as calling a subroutine. The instruction that performs this branch operation is known as CALL instruction.

After a subroutine has been executed, the calling program has to resume the execution continuing immediately after the instruction that called the subroutine. At the end of the subroutine a RETURN instruction will be there, which transfer the flow of control to the appropriate location. In order to do so, these location addresses have to store somewhere. Normally this will be the address of the location pointed to by the updated PC (Program Counter) while the CALL instruction is being executed. Hence, the contents of the PC must be saved by the CALL instruction to enable correct return to the calling program.

- **Subroutine Linkage**

The way in which computers make it possible to call and return from a subroutine is referred as subroutine linkage. Here, a register named as link register is used to store the

return address. When a subroutine completes its task, return instruction return to the calling program by branching indirectly through link register.

CALL instruction:

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

RETURN instruction:

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

Sub Routine Nesting and Processor Stack

One subroutine call in another subroutine is known as subroutine nesting. 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 happen at any depth. The return address are generated and used in a last in first out order. So, a stack may be the best option to store the return address associated with subroutine calls. Stack pointer register is used for this purpose. Stack pointer points to a stack called processor stack. The call instruction pushes the contents of PC onto the processor stack and loads the subroutine address into PC. The Return instruction pops the return address from processor stack into the PC.

Parameter Passing

While calling a subroutine, calling program has to pass the necessary operands or address to subroutine for processing the information. Similarly subroutines may return the results of computation to the calling program. This exchange of information between calling program and subroutine is termed as parameter passing.

Parameter passing can be done in several ways.

1. Parameters may place in registers.
2. Parameters may place in memory locations
3. Parameters may place on processor stack.

Parameters may pass with two mechanisms:

1. Passed by value: actual data is passed to the subroutine
2. Passed by reference: address of actual data is passed to the subroutine.

Stack Frame

A private work space (in stack) for the subroutine, which is created at the time the subroutine is entered and freed up when the subroutine returns control to the calling program is called stack frame. If the subroutine requires more space for local memory variables, they can also be allocated on the stack. A common layout of stack frame is shown in below figure.

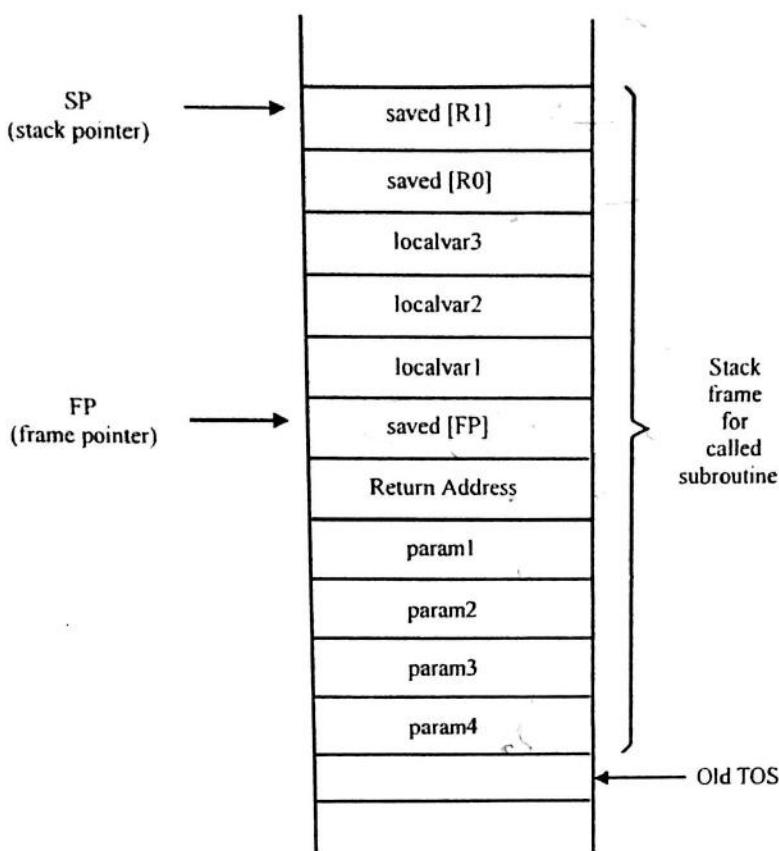


Fig 1.20: A subroutine stack frame example

In addition to stack pointer SP, it is useful to have another pointer FP (Frame Pointer). FP is useful for the accessing of parameters passed to the subroutine and the local memory variables used by the subroutine.

The contents of FP remain fixed throughout the execution of the subroutine. But stack pointer SP always point to the current top element of the stack which may vary. As FP is fixed we can use the indexed addressing mode to access the parameters and local variables. (For example, -4(FP), 8(FP))

Initially SP is pointed to the old TOS. Before the subroutine is called the calling program pushes the four parameters onto the stack. Then the CALL instruction is executed. Then return address is pushed onto the stack. Now SP points to this return address, and the first instruction of the subroutine is about to be executed. At this point FP is set to contain the proper memory address. Since FP is a general purpose register, it may contain information of use to the calling program; its contents are saved by pushing them on to the stack. Since SP now points to this position, its contents are copied into FP. Thus, the first two instruction executed in the subroutine are,

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

After these instructions are executed, both SP and FP point to the saved FP contents. Space for three local variables are allocated by executing the instruction

```
Subtract #12, SP
```