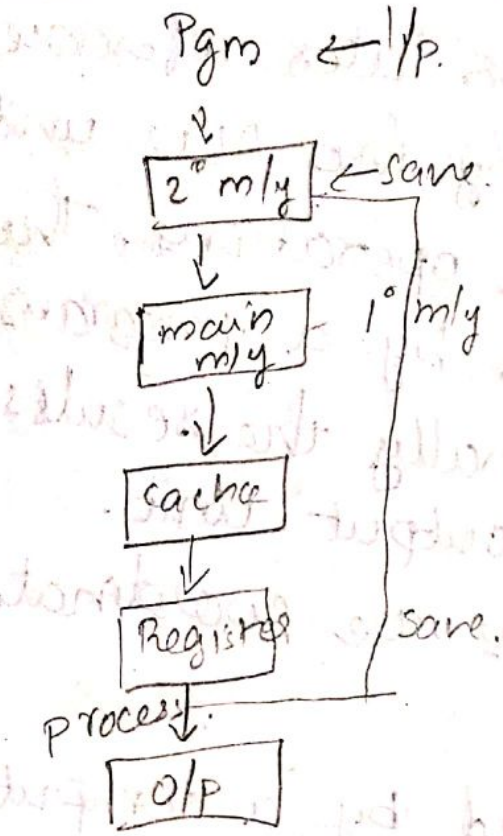
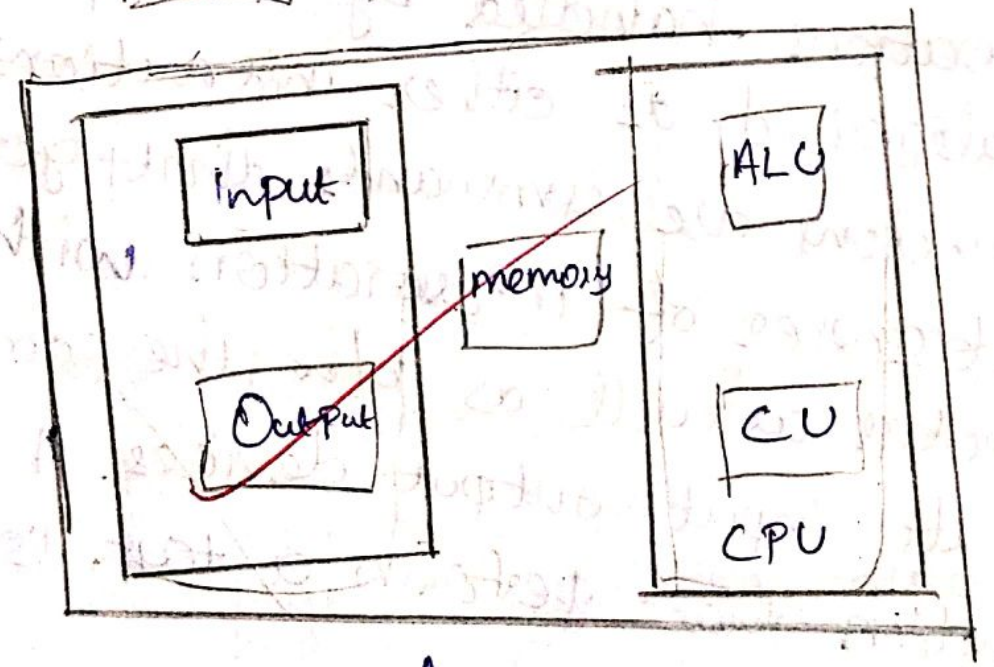
MODULE - 1

BASIC FUNCTIONAL UNITS OF A COMPUTER



A computer consist of 5 independant units.

1) Input.
2) Output
3) Memory
4) Arithmetic & Logic
5) Control units.



Input :- keyboard, mouse, Touch screen.

Output :- Speaker, Printer, Monitor, touch screen.

Input unit accepts information from human operators, from electro mechanical devices

such as keyboards or mouses (other computers) over digital communicational lines. The information recieved is either stored in the computer's memory for later reference or immediately used by the ALU unit to perform the desired operations. The Processing steps are determine by a program stored in the memory. Finally the results are send back to the output unit.

All of these actions are coordinated by the control unit (CU)

Informations handled by a computer can be categorized as either instructions or data. Instructions are commands that governed the transfer of information within a computer as well as b/w the computer and its input-output devices. A list of instruction that perform a task is called a program.

Programs are stored in the memory the processor fetches the instruction that make up the program from the memory one after another and performs the desired operation-

the computer is completely controlled by the stored program except for possible external interruption by an operator or by i/p - o/p devices connected to the machine

Information handled by a computer must be encoded in a suitable format, each number, character or instruction is encoded as a string of binary digits called bits, each having one of two possible values 0 or 1.

Alpha numeric characters are also expressed in terms of binary codes such as ASCII, (7bit) Unicodes, etc.

Input unit

Computers accept codded information through i/p units which read the data. Most well known i/p device is the 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 to the memory or the processor

~~Adem~~

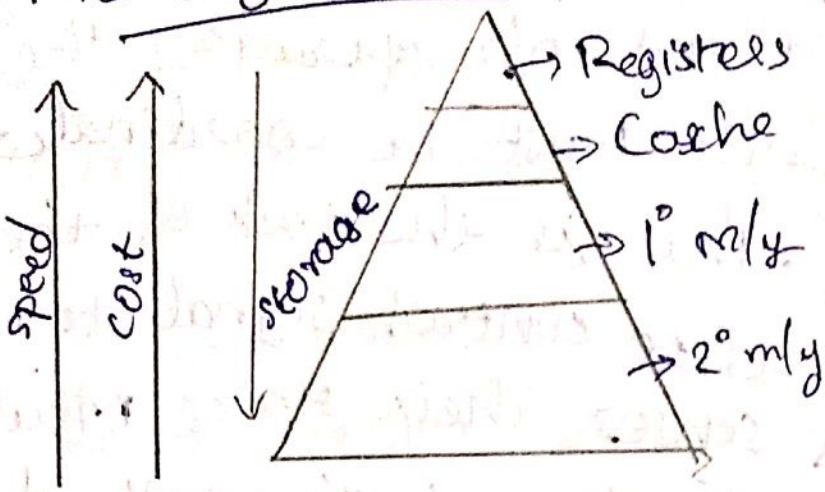
m/y unit

Function of m/y unit is to store Prgms and data. There 2 classes ~~and~~ of storage called 1° and 2°.

1° storage is a fast memory that operates at electronic speeds programs must be stored in the memory while they are being executed. The memory contains a large number of semiconductor storage cells, each capable of storing 1 bit of information These cells are rarely read or written as individual cells byt instead are processed in groups of fixed size called words. The number of bits in each word is often referred to as word length ~~or~~ of the computer. Typical word lengths range from 16 to 64 bits.

The memory of a computer is normally implemented as a m/y hierarchy of 3 or 4 levels of semiconductor RAM units with different speeds and sizes.

Memory Hierarchy



m/y → 1° m/y RAM, cache
→ 2° m/y Regis
↳ Hard dis

CD ROM
RAM — D.FF Static RAM (FF). (Fast)
— Dynamic RA
(capacitor

ALU &

Most computer operations are executed in the ALU of processor. Suppose 2 numbers located in the m/y are to be added, they are brought to the processor and actual addition is carried out by ALU, The sum may then be stored in the m/y or retained to the processor for immediate use

Output Unit

It's the function in the sense Processed result to the o/p world. Most familiar example of a such a device is printer. Some units such as graphic displays provide both an o/p function and i/p function.

Control Unit

The m/y, Arithmetic & Logic and i/p-o/p

units store and process information and perform input and o/p operations, the operation of these units must be coordinated in some way. This is the task of the CU. The CU sense control signals to other units and senses their states. Much of the control circuitary is physically distributed throughout the machine. A large set of control line (wires) carries the signals used for timing and synchronization of events in all units.

31/1/19

Basic Operational concepts.

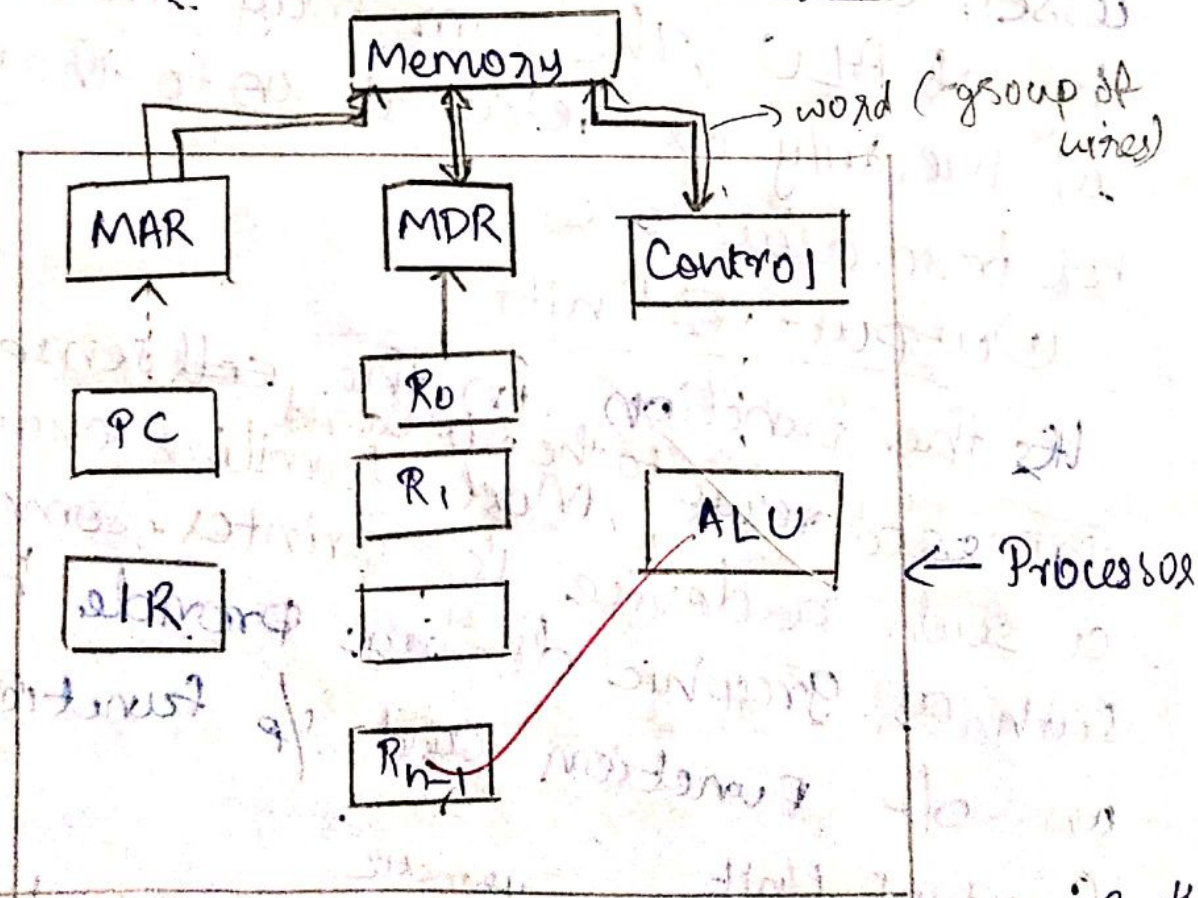


Fig :- Connections between the processor & the memory

Assembly Language / Middle level Programming Langua

Add LOCA, R0	Load LOCA, R1 Add R1, R0

IR → Instruction Register

PC → Program Counter

$R_0, R_1, \ldots R_{n-1}$ ⇒ General purpose Registers

MAR ⇒ Memory address register

MDR ⇒ Memory data Register

Interrupt signal

↓

Interrupt service routine

↓

Processor state or saved

Any activity in a computer is governed by instructions to perform a given task and appropriate program consisting of a list of instructions is stored in the memory individual instructions are brought from the memory into the processor which executes the specified operations. Data to be used as operands are also stored in the memory

A typical instruction may be load LOC A, R_1
Add R_1, R_0. The first of these instructions transfers the contents of mly location LOC A into processor register R_1 and 2nd instruction Adds the contents of register R_1 and R_0 and places the sum into R_0. This destroys the former contents of register R_1 as well as R_0 whereas the original contents of mly location LOC A are preserved. Transfers b/w the mly and processor are started by sending the address of the mly location to be accesed to the memory unit and issuing the appropriate control signal. In addition to the ALU and control unit, processor contains a no. of registers used for several different purposes. The instruction register (IR) cause hold the instruction that is currently being executed. Its output is available to the control circuits which generate the timing signals that control the various processing elements involved in executing the instruction. The PC is another register that keeps track of execution of a program, it contains the mly address of the next instruction to be fetched and executed

during the execution of an instruction, the contents of PC of are updated to correspond to the address of next instructions to be executed, besides the IR and PC, there are n general purpose registers R0, ~~R1~~ through Rn-1. Finally 2 registers facilitate communication with the m/y. These are the m/y address registes (MAR) and the m/y data register (MDR). The MAR contains the address of location to be accessed. The MDR contains the data to be written into or read out of the addressed location.

Execution of the program starts when the PC is said to point to the first instruction of the program. The contents of the PC are transfer to the MAR and a read control signal is ~~said~~ end to the memory. And the content is loaded into the MDR. Next the contents of the MDR are transfer to the IR. At this point the instruction is ready to be executed, if the instruction involves an operation to be performed by the ALU. It is necessary to obtained the required operands. ~~We can~~ If an operand

resides in the m/g. it has to be fetched by sending its address to the MAR and initiating the read cycle. When the operand has ~~to be~~ been read by the MDR. It is transfered from MDR to the ALU. After one or more operands are fetched in this way. The ALU can perform the desired operation, if the result of this operation is to be stored in the m/y then the result is send to the MDR. The address of the location where the result is to be stored is ~~s~~send to the MAR. And the write cycle is initiated.

1/02/19

BUS STRUCTURES

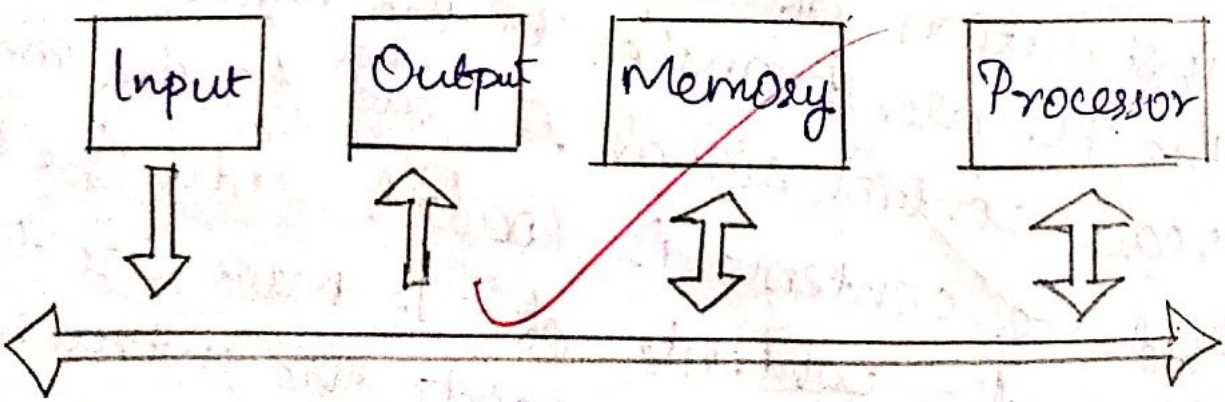


fig:- Single-bus structure

Register.

Buffer ~~pointer~~ → for slow devices

eg:- Printer, keyboards

BUS → group of wires.

→ "word" operation / transfer.

Address of single bus.

↳ Buffer register is used to speed the devices like printer, keyboard etc.

Disks

Software

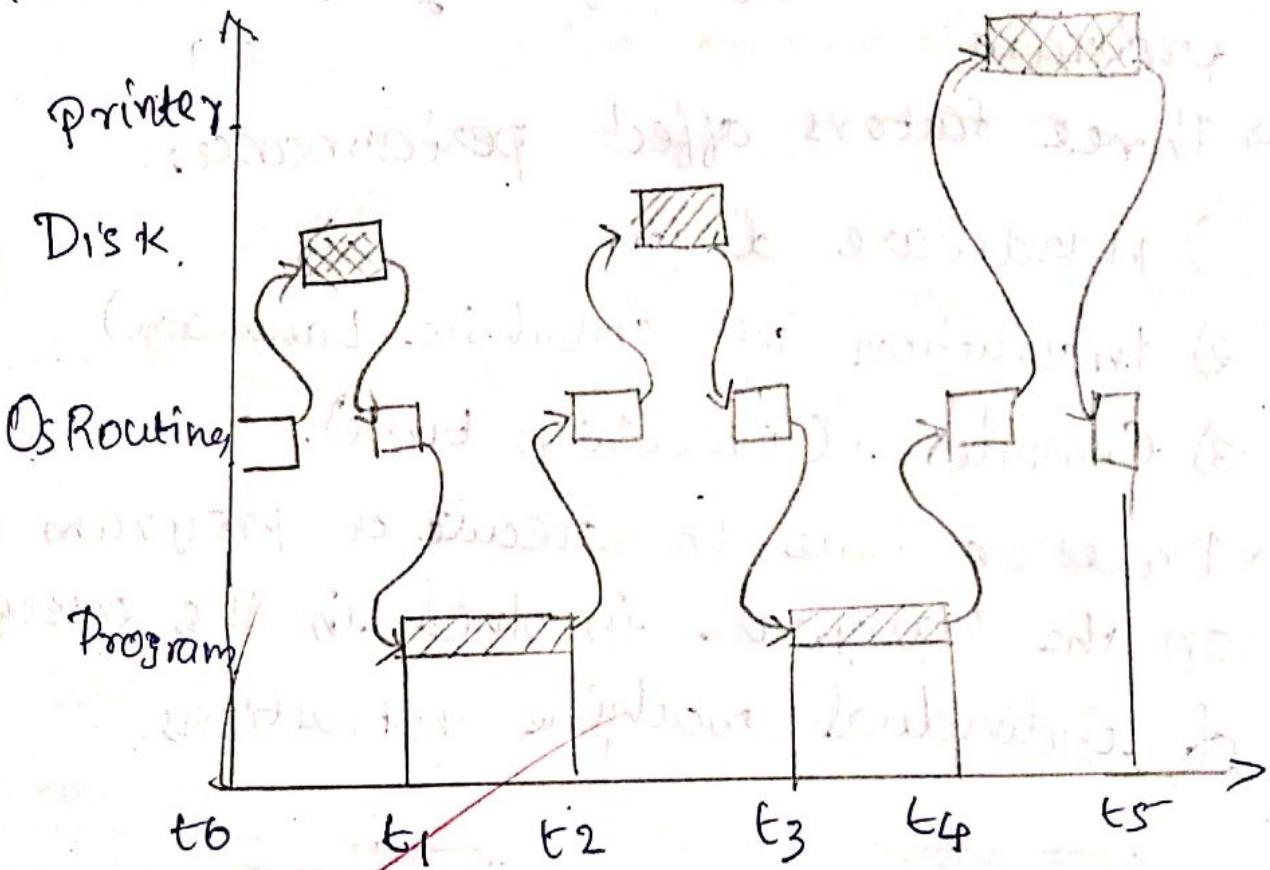


Fig:- User program and Os routine sharing of the processor.

a) system software b) Application s/w.

eg :- Os, compiler, editors.

multiprogramming or multitasking.

Sample program

↓

loads a file from Disk to processor and prints result to printer.

2/2/19

Performance of System

→ The most important measures of a computer is how quickly it can execute program.

→ Three factors affect performance:

1) Hardware design
2) Instruction set (Machine Language)
3) Compiler (Execution time).

→ Processor time to execute a program depends on the hardware involved in the execution of individual machine instructions.

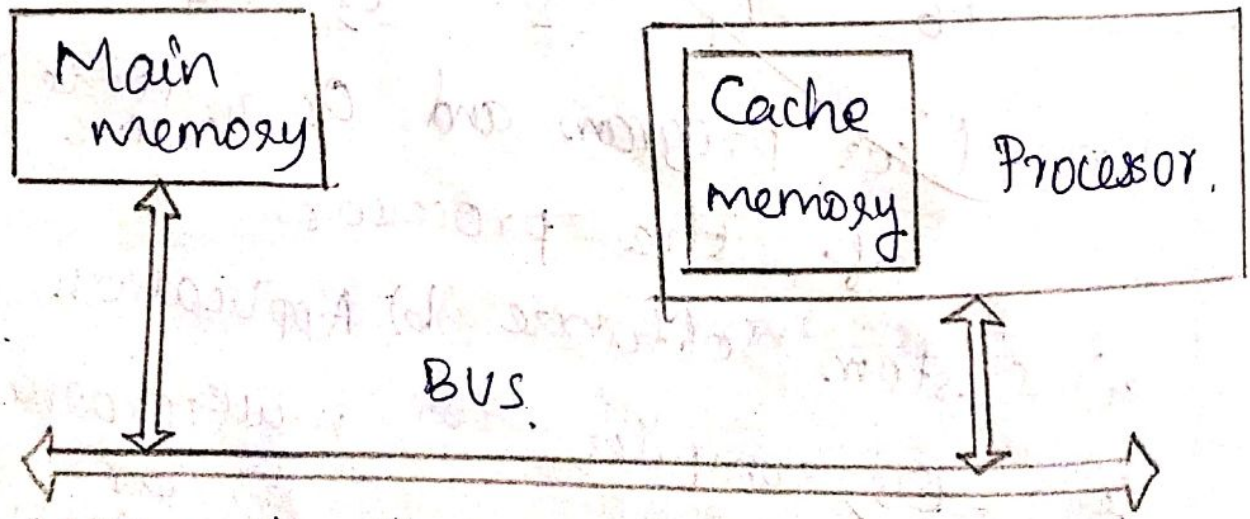


Fig: The Processor Cache.

Basic performance equation

$$T = \frac{N \times S}{R}$$

T – Processor time required to execute a program that has been prepared in HLL.

N – Number of actual machine language instruction needed to complete the execution (note: loop)

S – Average number of basic steps needed to execute one machine instruction. Each step completes in one clock cycle.

R – clock rate

Note: there are not independent to each other

? How to improve T.

Pipeline and superscalar operation.

→ Instructions are not necessarily executed one after another.

→ the value of S doesnot have to be the number of clock cycles to execute one instruction.

→ Pipelining: Overlapping the execution of successive instruction.

→ Add R_1, R_2, R_3.

→ Superscalar operation : Multiple instruction pipelines are implemented in the processing.

→ Goal - Reduce S (could become < 1 !)

CISC and RISC

→ Tradeoff between N & S

→ A key consideration is the use of Pipelining

i) S is close to 1 then though the number of basic steps per instruction may be considerably large,

ii) It is much easier to implement efficient pipelining in processor with symbol instruction sets.

→ Reduced instruction set computer [RISC]

→ Complex instruction set computer [CISC]

RISC - Ex - intel X86, AMD

CISC - Ex - ARM, MIPS, SPARC

Compiler

→ A compiler translates a HLL program into a sequence of machine instructions.

→ To reduce N, we need a suitable machine instruction set and a compiler that makes good use of it.

→ Goal - Reduce N x S

→ A compilor may not be designed for a specific processor; however, a high quality compilor is usually designed for, and with a specific processor

Performance Measurement

→ It is difficult to computer

→ Measure computer performance using bench-mark programs.

→ System performance Evaluation cooperation (SPEC) ~~se~~ selects and publishes representative application programs.

5/2/19

Memory Location, Addressers, & Operation.

• Memory consists of many millions of storage cells, each of which can store 1 bit.

• Data is usually accessed in n-bit groups. n is called word length.

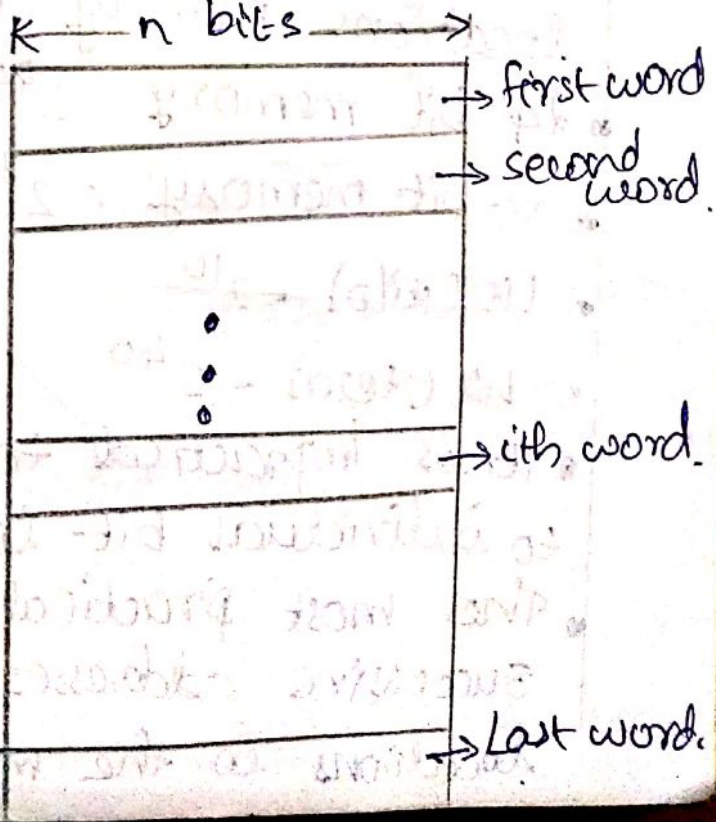
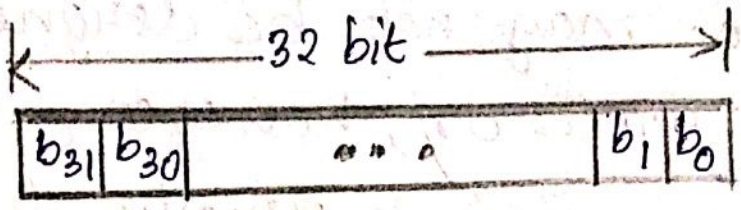


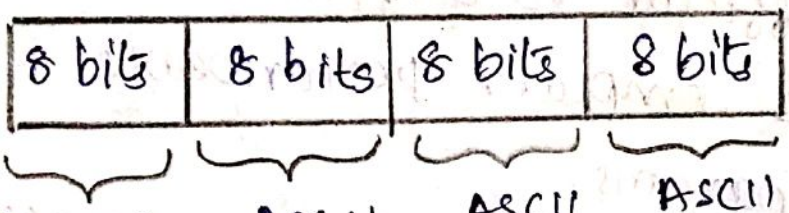
Fig:- Memory words

- 32-bit word length example.



Sign bit: $b_{31} = 0$ for positive numbers
$b_{31} = 1$ for negative numbers

a) A signed integer.



- To retrieve information from memory; either for one word or one byte (8-bit), addresses for each location are needed.
- A k-bit address memory has 2^k memory locations, namely $0 - 2^k - 1$, called memory space
- 24-bit memory : $2^{24} = 16,777,216 = 16M$ ($1M = 2^{20}$)
- 32-bit memory : $2^{32} = 4G$ ($1G = 2^{30}$)
- 1K (kilo) $= 2^{10}$
- 1T (tera) $= 2^{40}$
- It is impractical to assign distinct addresses to individual bit locations in the memory.
- The most practical assignment is to have successive addresses refer to successive byte locations in the memory - byte-addressable

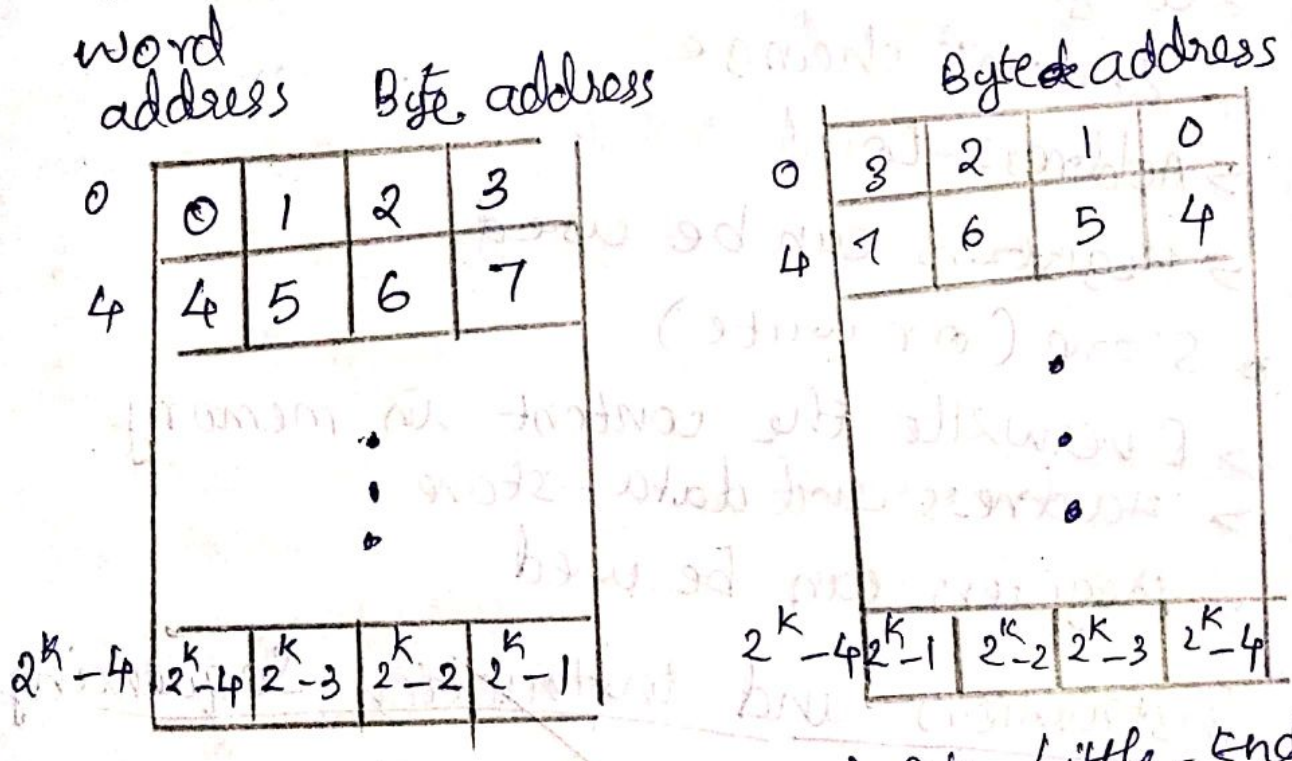
memory

- Byte locations have addresses 0, 1, 2, ...
If word length is 32 bits, they successive words are located at addresses 0, 4, 8, ...

Big-Endian and Little-Endian Assignments.

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

Little Endian: Opposite ordering. Lower byte addresses are used for the less significant bytes of the word



a) Big-endian assignment.

b) Big-Little-Endian assignment

Fig:- Byte and word addressing

- Address ordering of bytes
- Word alignment.
  - Words are said to be aligned in memory

if they begin at a byte address that is a multiple of the num of bytes in a word

→ 16-bit word : word addresses : 0, 2, 4, ...

→ 32-bit word : word addresses : 0, 4, 8, ...

→ 64-bit word : word addresses : 0, 8, 16, ...

- Access numbers, characters, and character string

Memory Operation.

- Load (or Read or Fetch)
  - > Copy the content. The memory content doesn't change
  - > Address - Load
  - > Registers can be used
- Store (or write)
  - > Overwrite the content in memory.
  - > Address and data - store
  - > Registers can be used

6/2/19

Instruction and Instruction Sequencing.

"Must-Perform" Operations.

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

Register Transfer Notation

- Identify a location by a symbolic name standing for its hardware binary address (LOC, R0, ...)
- Contents of a location are denoted by placing square brackets around the name of the location ($R_1 \leftarrow [LOC]$, $R3 \leftarrow [R1] + [R_2]$)
- Register Transfer Notation (RTN)

Assembly Language Notation

- Represent machine instructions and programs
- Move LOC, R_1 = $R1 \leftarrow [LOC]$
- Add R_1, R_2, R_3 = $R_3 \leftarrow [R_1] + [R_2]$

CPU Organization

- Single Accumulator.
  - > Result usually goes to the Accumulator
  - > Accumulator has to be saved to memory quite often.
- General Register
  - > Registers hold operands thus reduce memory traffic.
  - > Register bookkeeping
- Stack
  - > Operands and result are always in the stack.

Instruction Formats

TOS - TOP Of Stack

- Three-Address Instructions
  - > ADD R_1, R_2, R_3 $R_1 \leftarrow R_2 + R_3$
- Two-Address Instructions
  - > ADD R_1, R_2 $R_1 \leftarrow R_1 + R_2$
- One-Address Instructions
  - > ADD M $AC \leftarrow AC + M[AR]$
- Zero-Address Instructions
  - > ADD $TOS \leftarrow TOS + (TOS - 1)$
- RISC Instructions
  - > Lots of registers. Memory is restricted to Load & store

Instruction Formats

Example : Evaluate (A+B) * (C+D)

- Three-Address

1. ADD R_1, A, B ; $R_1 \leftarrow M[A] + M[B]$
2. ADD R_2, C, D ; $R_2 \leftarrow M[C] + M[D]$
3. MUL X, R_1, R_2 ; $M[X] \leftarrow R_1 * R_2$

- Two-Address

1. MOV R_1, A ; $R_1 \leftarrow M[A]$
2. ADD R_1, B ; $R_1 \leftarrow R_1 + M[B]$
3. MOV R_2, C ; $R_2 \leftarrow M[C]$

4. ADD R2, D ; R2 ← R2 + M[D]

5. MUL R1, R2 ; R1 ← R1 * R2.

6. MOV X, R1 ; M[X] ← R1

- One-Address.

Accumulator registers.

1. LOAD A ; AC ← M[A]

2. ADD B ; AC ← AC + M[B]

3. STORE T ; M[T] ← AC

4. LOAD C ; AC ← M[C]

5. ADD D ; AC ← AC + M[D]

6. MUL T ; AC ← AC * M[T]

7. STORE X ; M[X] ← AC.

- Zero-Address

1. PUSH A ; TOS ← A

2. PUSH B ; TOS ← B

3. ADD ; TOS ← (A + B)

4. PUSH C ; TOS ← C

5. PUSH D ; TOS ← D

6. ~~MUL~~ ADD ; TOS ← (C + D).

7. MUL ; TOS ← (C+D) * (A+B)

8. POP X ; M[X] ← TOS.

7/2/19

- RISC

1. LOAD R1, A ; $R_1 \leftarrow M[A]$
2. LOAD R2, B ; $R_2 \leftarrow M[B]$
3. LOAD R3, C ; $R_3 \leftarrow M[C]$
4. LOAD R4, D ; $R_4 \leftarrow M[D]$
5. ADD R1, R1, R2 ; $R_1 \leftarrow R_1 + R_2$
6. ADD R3, R3, R4 ; $R_3 \leftarrow R_3 + R_4$
7. MUL R1, R1, R3 ; $R_1 \leftarrow R_1 * R_3$
8. STORE X, R1 ; $M[X] \leftarrow R_1$

Using Registers

- Registers are faster
- shorter instructions.
  > The number of registers is smaller (eg :- 32 registers need 5 bits)
- Potential speed up
- Minimize the frequency with which data is moved back and forth between the memory and processor registers.

Instruction Execution and Straight-Line Sequencing.

PC
↓
IR
↓

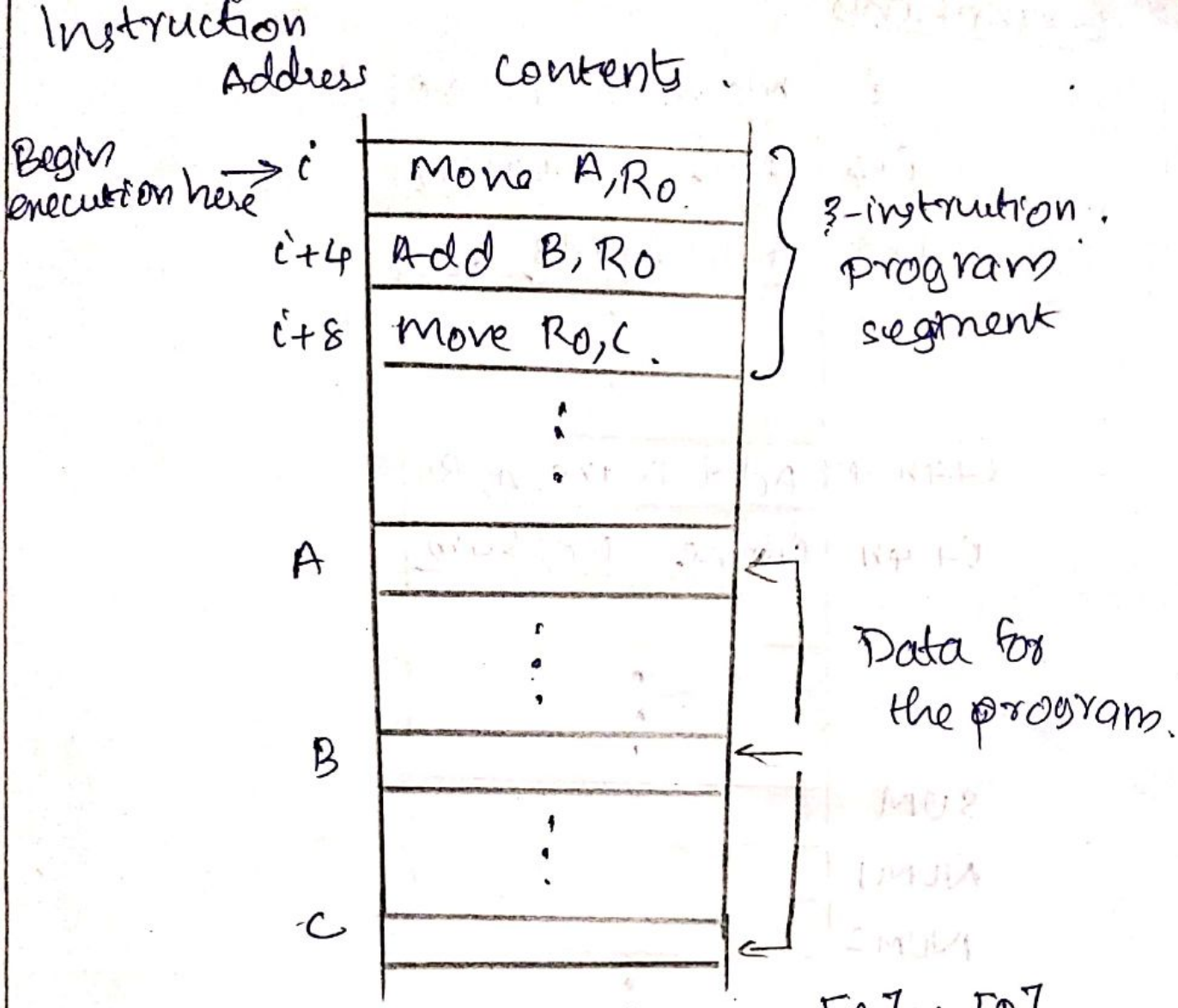


fig:- Program for C ← [A] + [B]

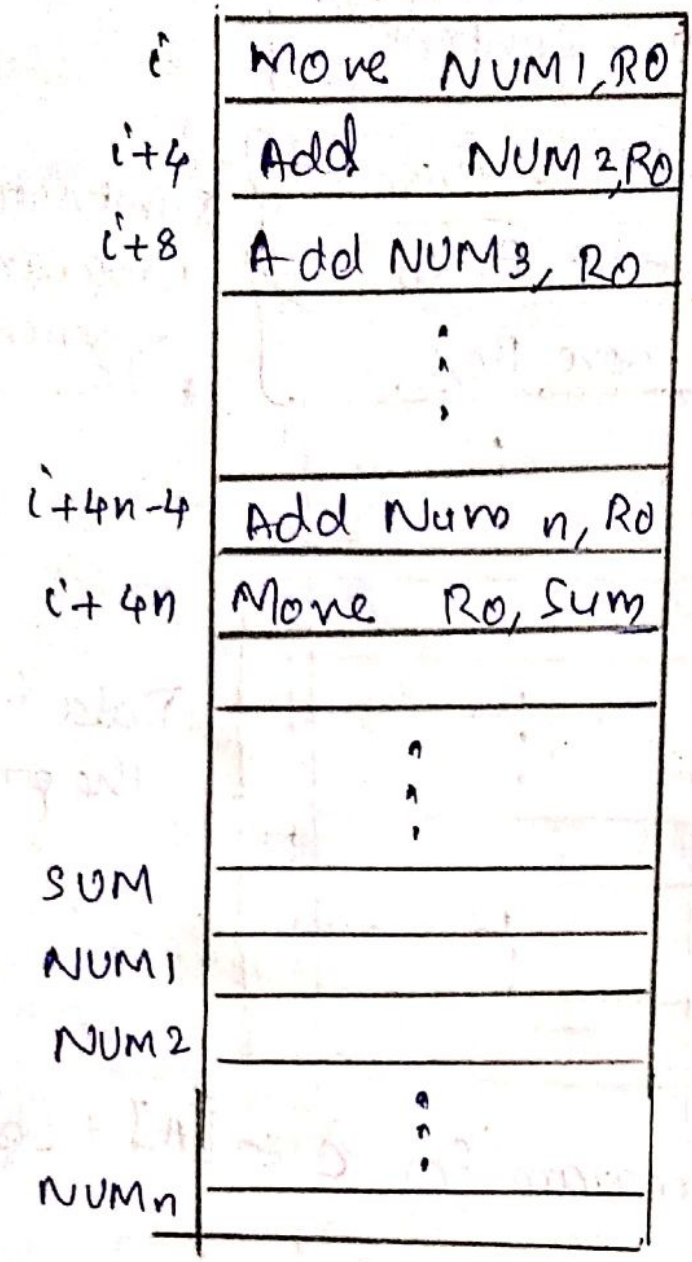
Assumptions:

- One memory operand per instruction.
- 32 bit word length.
- Memory is byte addressable
- Full memory address can be directly specified in a single-word instruction.

Two-phase procedure.

- Instruction fetch
- Instruction execute.

Branching



A straight-line program for adding n numbers.

Condition Codes

- Condition code flags.
- Condition code register/status register
- N (negative)
- Z (zero)
- V (overflow)
- C (carry)
- Different instructions affect different flags.

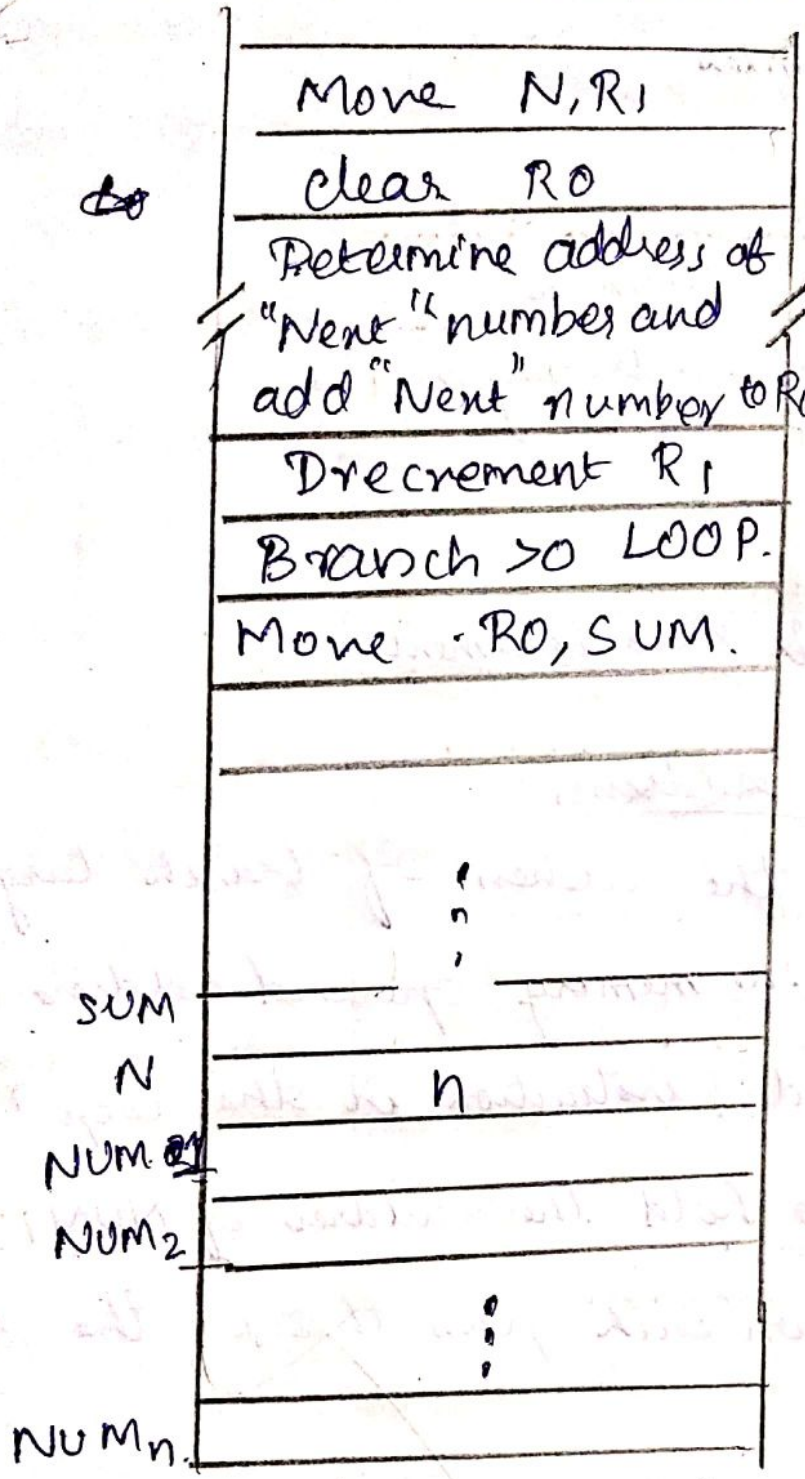


Fig :- Using a loop to add n numbers

14/2/19

Addressing Modes

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

Basic Addressing modes

1. Immediate
2. Register
3. Absolute / Direct
4. Indirect
5. Index
6. Auto Increment & Auto Decrement

} Common Addressing modes.

~~Generating Memory Addresses~~

⇒ How to specify the address of branch target?

→ Can we give the memory operand address directly in a single Add instruction in the loop?

→ Use a register to hold the address of NUM1; then increment by 4 on each pass through the loop.

→ Implied

- AC is implied in "ADD M[AR]" in "One-Address" instructions
- TOS is implied in "ADD" in "Zero-Address" instruction

→ Immediate

The use of constraint in "MOV R1,5", i.e. R1←5

→ Register

- Indicate which register hold the operand.

→ Register Indirect

- Indicate the register that holds the number of the register that holds the operands.

MOV R1,(R2)

→ Auto increment / Auto decrement

- Access & update in 1 instruction

→ Direct Address

- Use the given address to access a memory location

~~→ Addressing Modes~~

→ Indirect Address

- Indicate the memory location that holds the address of the memory location that holds the data

→ Relative Address

- EA = PC + Relative Address

12/2/19 Effective Address

Some instruction doesnot give the operand or its address explicitly instead it provides information from which the only address of the operands can be determined. This address is known as Effective address.

1. Register Addressing mode

The operand is the content of a

process register, the name (address) The name of the register is given in the instr.

2. Absolute addressing Mode

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

eg:- MOV LOC, R2 uses these 2 modes.

3. Immediate Addressing Mode

The operand is given explicitely in the instruction

eg:- MOV #200, R0. places the value 200 in register R0. A common convension is to use the sharp sign (#) infront of the value to indicate that, this value is to be used as immediate operand.

4. Indirect Addressing mode

The effective address of the operand is the contents of the registers or memory location whose address appears in the instruction

eg:- Add (R1), R0 we denote indirection?

by Placing the name of the register or the memory address given in the instruction in paranthesis : to execute the add instruction the processor uses the value B which is in register R_1 as the effective address of the operand.

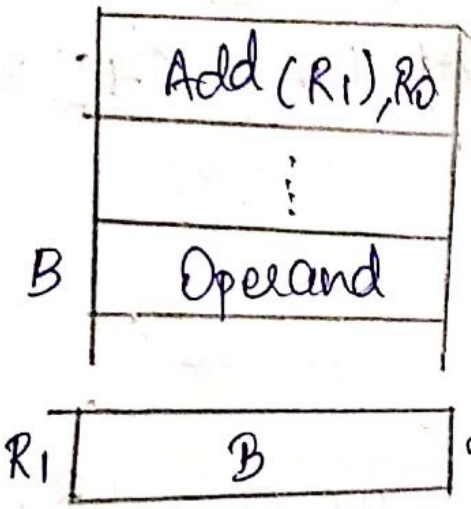


Fig (a):- Register.

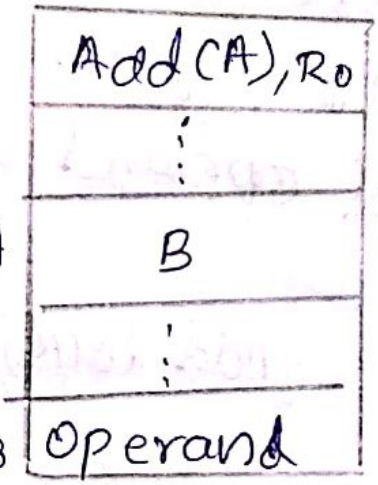


fig:-(b) Memory

The register or memory location that contains the address of an operand is called a pointer.

5. Index Addressing Mode

The effective address of the operand is generated by adding the constant value to the contents of ~~the~~ a register. It is useful in dealing with list and arrays. The register used may be either a special register or more commonly it may be any one of

a set of general purpose registers in the processor. In either case it is referred to as an index register; we indicate the index mode as $X(R_i)$, where X denotes the constant value contained in the instruction and R_i is the name of the register involved. The effective address of operand is given by $EA = X + [R_i]$.

→ Value of R_i

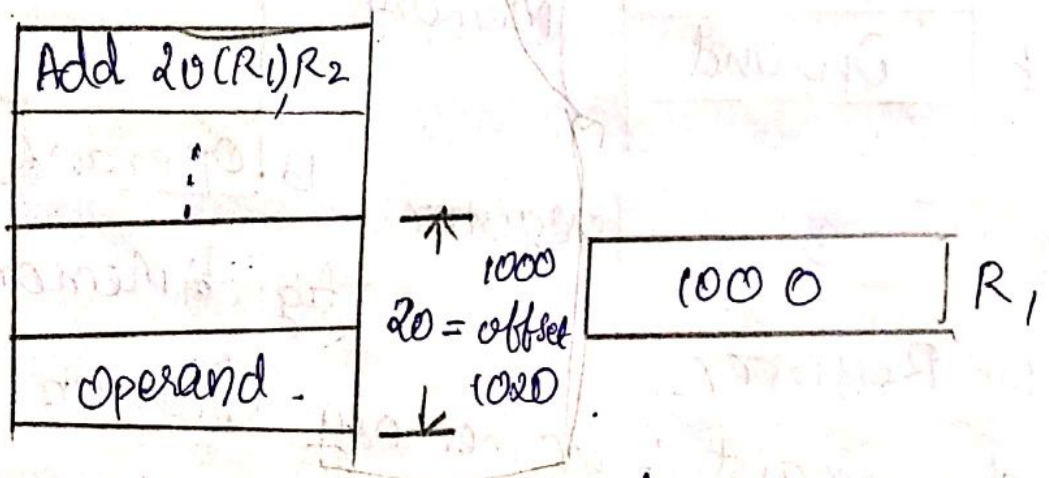


fig 2(a): offset is given as a constant.

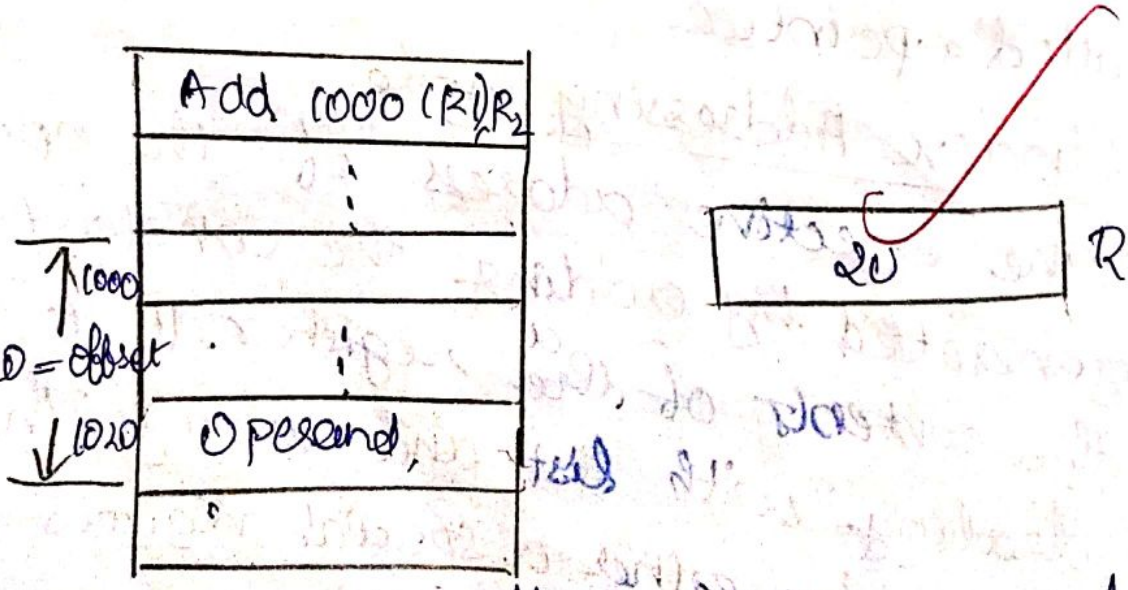


fig 2(b): Offset is in the index register

In figure 2.(a) the index register R_i contains the address of a memory location and the value X defines an offset (displacement) from this address to the location where the operand is found.

In fig 2.(b), the constant X corresponds to a memory address and the contents of the index register define the offset to the operand. In either case the effective address is the sum of 2 values, 1 is given explicitely in the instruction and the other is stored in a register. Several variations of the basic index form provide for very efficient access to mly operands in practical programming situations.

Eg:- A second register may be used to contain the offset X in which case we can write the index mode as (Ri, Rj). The EA is the sum of the contents of the registers Ri and Rj. The second register is called the base register.

Another version of the index mode uses 2 registers plus a constant which can be denoted as X (Ri, Rj).

In this case the EA is the sum of the constant X and the contents of the register R_i and R_j

Relative mode

The effective address is determined by the index mode using the program counter (PC) in place of the general purpose register R_i. This mode can be used to access data operands, the most common use is to specify the target address in branch instruction

eg:- Branch >0 LOOP

6. Auto Increment & Auto Decrement AM

1. Auto increment mode

The EA of the operand is the contents of the registers specified in the register after accessing the operand, the contents of this registers are automatically incremented to point to the next item in a list or an array. We denote the auto increment mode

by putting the specified register in parantheses to show that the contents of the registers are used as the effective address followed by a (+) sign to indicate that these contents are to be incremented after the operand is accessed.

The auto increment mode is written as

(Ri)+.

2. Auto Decrement mode (AD mode.)

The contents of a register specified in the instruction are ~~specified~~ first automatically decremented and are then used as the EA of the operand. We denote the AD mode by -(Ri).

12/9 Addressing Modes

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

Name	Assembler Syntax	Addressing Function
Immediate	#value	Operand = value
Register	Ri	EA = Ri
Absolute (Direct)	LOC	EA = LOC
Indirect	(Ri) (LOC)	EA = [Ri] EA = [LOC]
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	X(PC)	EA = [PC] + X
Auto increment	(Ri)+	EA = [Ri]; Increment Ri
Auto Decrement	-(Ri)	Decrement Ri; EA = [Ri]

1/10

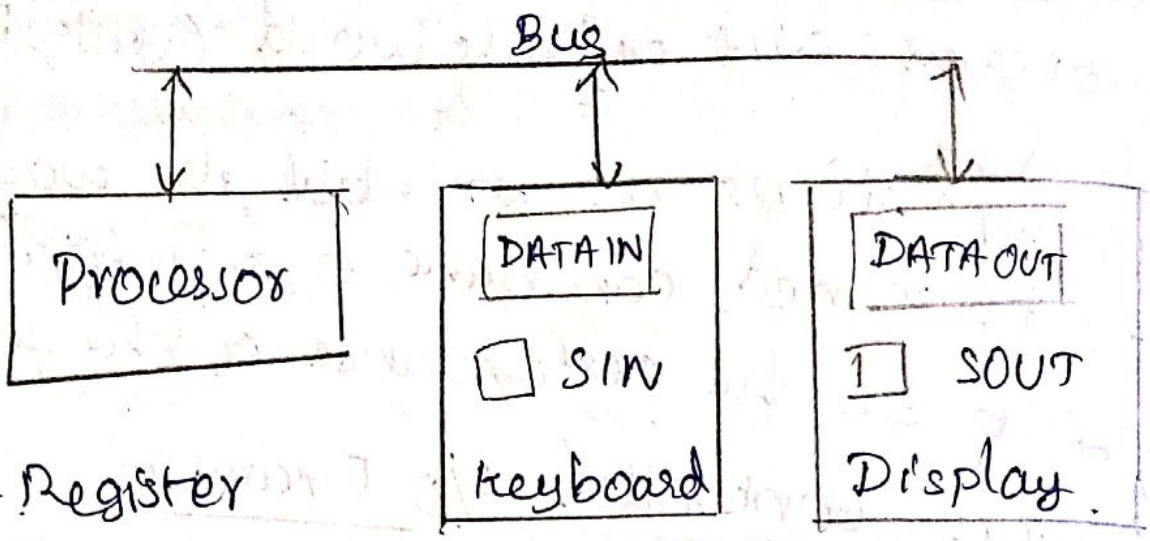
The data on which the instructions operate are not necessarily already stored in memory

- Data need to be transfered between processor and outside world (disk, keyboard, etc.)
- I/O operations are essential, the way they are performed can have a significant effect on the performance of the computer.

Program - Controlled I/O Example.

- Read in character input from a keyboard and produce character output on a display screen.
  - > Rate of data transfer (keyboard, display, processor)
  - > Difference in speed between processor and I/O device creates the need for mechanisms to synchronize the transfer of data.
  - > A solution: an output, the processor sends the first character and then waits for a signal from the display that the character has been recieved. It then sends the second character. Input is sent from the keyboard in a similar way.

Program-Controlled I/O example



- Register
- Flag
- Dual interface.

- Machine instructions that can check the state of the status flags and transfer data:

READ WAIT Branch to READ WAIT if SIN = 0
Input from DATAIN to R_1

WRITE WAIT Branch to WRITE WAIT if
SOUT = 0
Output from R_1 to DATAOUT.

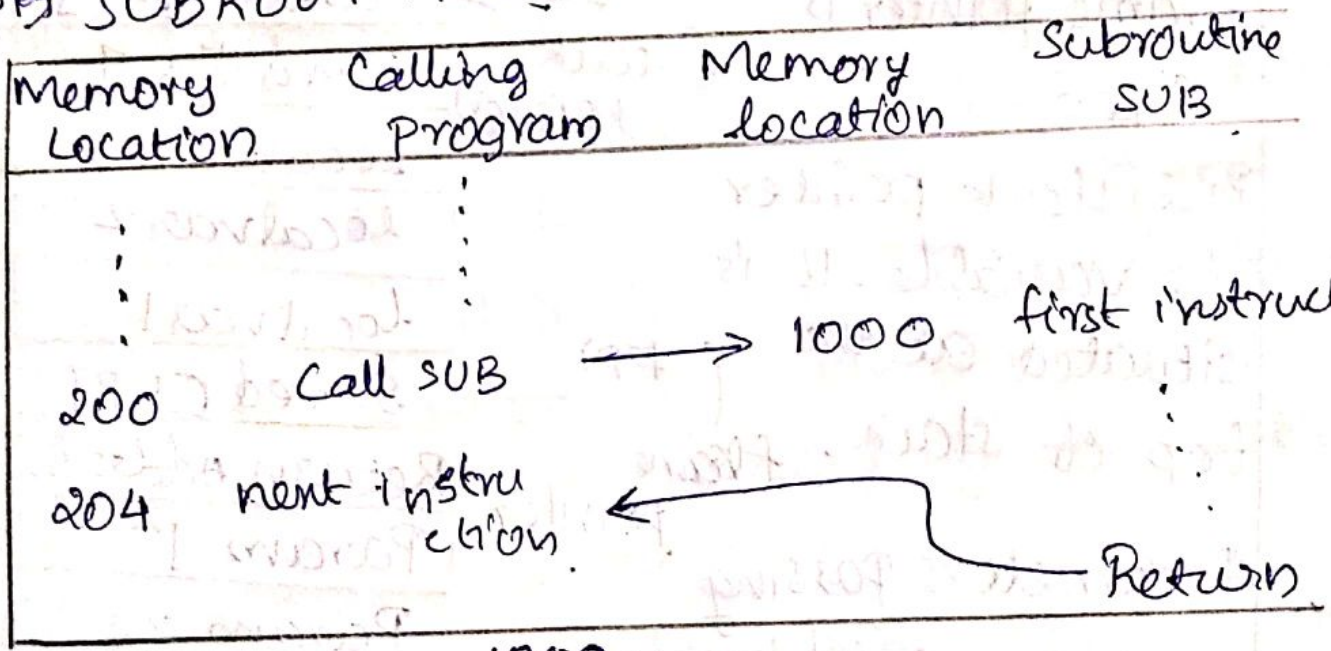
- Memory-Mapped I/O – some memory address values are used to refer to peripheral device buffer registers. No special instructions are needed. Also use device status registers

READ WAIT Testbit #3, INSTATUS
Branch = 0 READ WAIT

MoveByte DATAIN, R1

- Assumption - the initial state of SIN is 0 and the initial state of SOUT is 1.
- Any drawback of this mechanism in terms of efficiency?
  > Two wait loops → Processor execution time is wasted.
- Alternate solution?
  > Interrupt.

14/2/19 SUBROUTINES



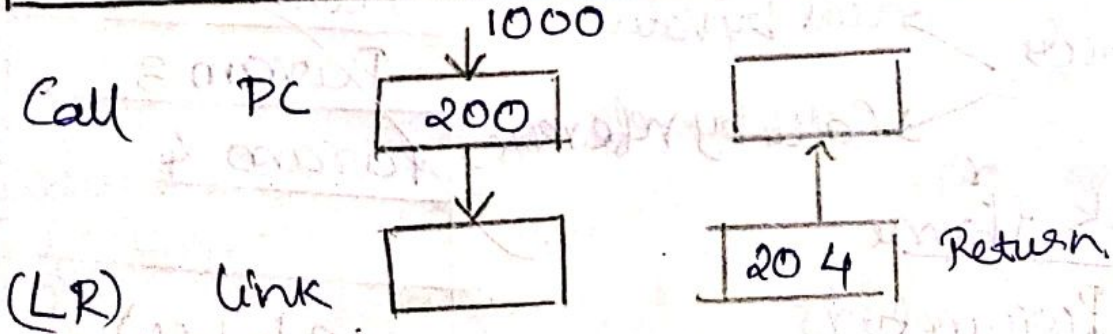


fig:- Subroutine linkage using a link register (LR)

Subroutine is the name in Assembly language.

In C, subroutine is function.

In JAVA, " is methods.

Subroutine nesting:- using processor stack.

If more call is occur, it is not save in link register. But it is stored in the processor stack.

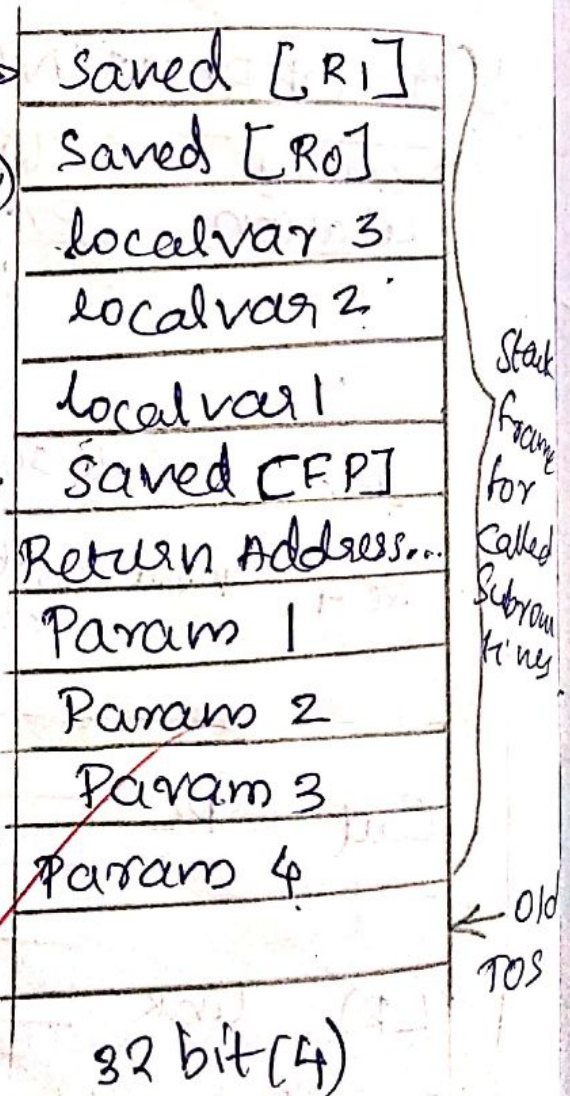
The STACK FRAME

Frame pointer is fixed

But stack pointer is variable. It is situated at the top of stack.

Parameters passing methods → Call by value
→ Call by reference.

Stack frame
- Parameters
- return address
- local variable
- Saved register



Subroutines

In a given program it is often necessary to perform a particular subtask many times on different data values. Such a subtask is called a subroutines. The way in which the computer makes it possible to call and return from subroutines is referred to as its subroutine linkage method. This simplest subroutine linkage method is to save the return address in a specific location, which may be a register dedicated to this function, such a register is called the link register. When the subroutine completes its task the return instruction returns to the calling program by branching indirectly through the link register.

A common programming practice called subroutine nesting is to have one subroutine call another in this case the return address of the second call is also stored in the link register, destroying its previous contents. Hence it is essential to save

the contents of the link register to some other location, before calling another sub-routine. Eventually, the last subroutine call completes its computation and returns to the subroutine that called it. the return address needed for the Ist return is the last one generated in the nested call sequence i.e, the return addresses are generated and used in a LIFO order. This suggest the return addresses associated with subroutine calls, should be pushed on to a stack. A particular register is designated as stack pointer (SP) to be used in this operation, the SP points to a stack called the processor stack

19/2/19 eg:- Assembly program to add N numbers.

```
Move N, R1   — Absolute, Register
                Source → destination
Move #NUM1, R2   — Immediate
Clear R0
Loop Add (R2)+, R0
     Decrement R1
     Branch >0 LOOP
     Move R0, SUM
```

i.e, R1 = 0, (if 1) check Z (if 0) come out of loop

Parameter Passing in Subroutines

Calling Pgm

```
MOVE  N, R1
Move. #NOM 1, R2
Call  LISTADD
Move  R0, SUM.
```

call by value / address

R1 & R2 is passed

Subroutine.

```
LISTADD  clear R0
LOOP     Add (R2)+, R0
         Decrement R1
         Branch>0  LOOP.
         Return
```

Using Stack (for large no. of values)

due to limited registers in CPU

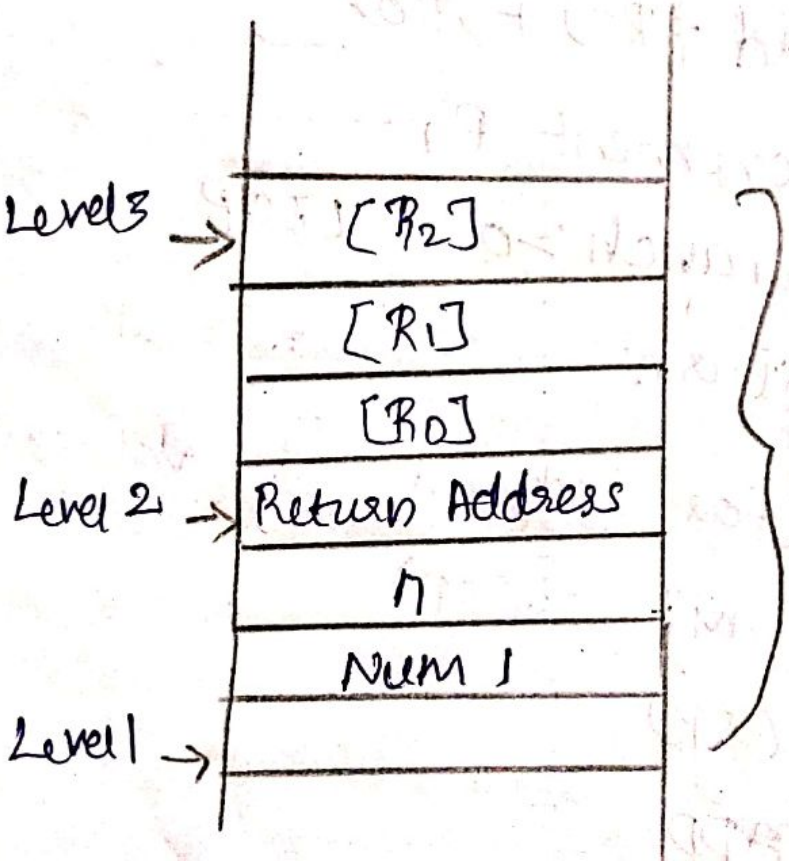
```
Move. #NUM1, -(SP)
Move  N -(SP).
Call  LISTADD
Move  4(SP), SUM
Add   #8, SP
```

(address to stack more than 1 regis. per push or POP at a time.

```
LISTADD  Move Multiple R0-R2, -SP)
         Move 16(SP), R1
         Move 20(SP), R2.
```

```
        Clear R0
LOOP →  Add (R2)+, R0
        Decrement R1
        Branch >0 LOOP
        Move R0, Z0(SP)
        Move Multiple (SP)+, R0-R2
        Return
```

Stack to register (POP)



stack frame for subroutine LIST ADD
→ SP
→ FP (Frame Point)