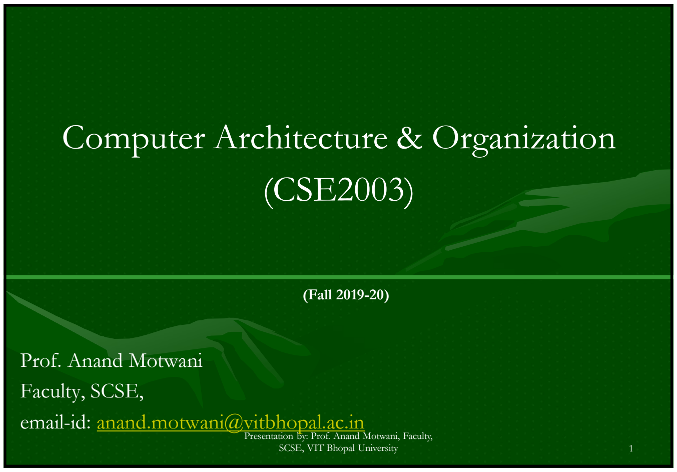
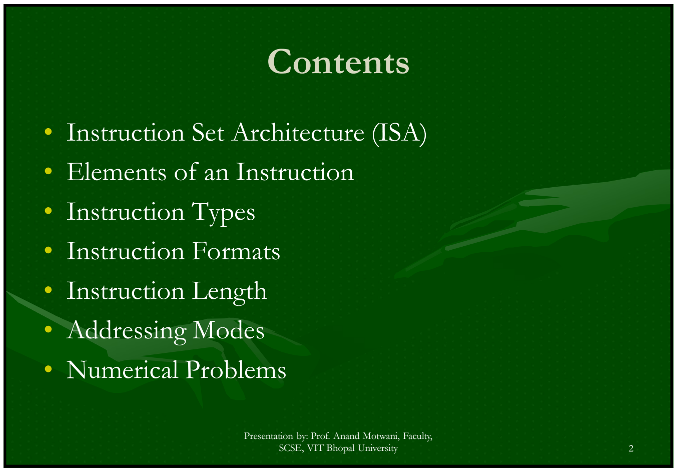


# Computer Architecture & Organization (CSE2003)

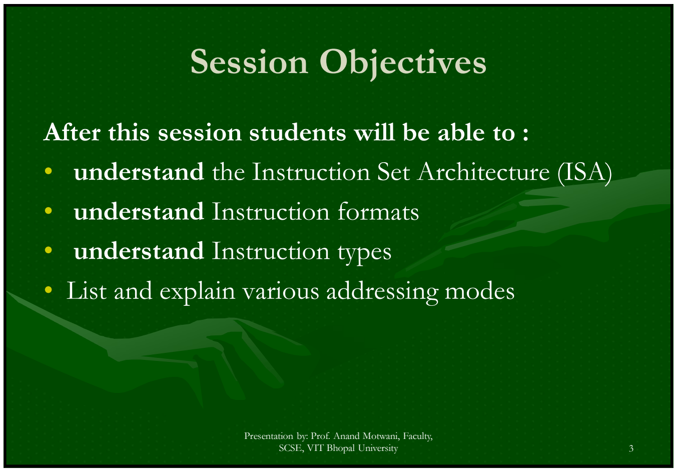
**(Fall 2019-20)**

Prof. Anand Motwani
Faculty, SCSE,
email-id: anand.motwani@vitbhopal.ac.in



## Contents

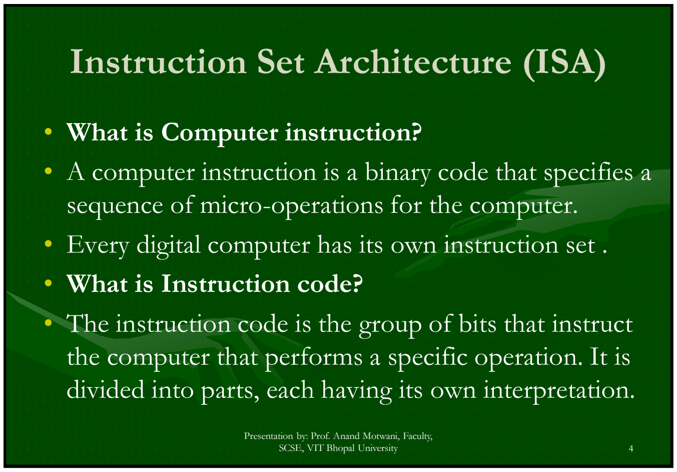
- Instruction Set Architecture (ISA)
- Elements of an Instruction
- Instruction Types
- Instruction Formats
- Instruction Length
- Addressing Modes
- Numerical Problems



## Session Objectives

**After this session students will be able to :**

- **understand** the Instruction Set Architecture (ISA)
- **understand** Instruction formats
- **understand** Instruction types
- List and explain various addressing modes



## Instruction Set Architecture (ISA)

- **What is Computer instruction?**
- A computer instruction is a binary code that specifies a sequence of micro-operations for the computer.
- Every digital computer has its own instruction set .
- **What is Instruction code?**
- The instruction code is the group of bits that instruct the computer that performs a specific operation. It is divided into parts, each having its own interpretation.

- The ISA serves as the boundary between software and hardware. We will briefly describe the instruction sets found in many of the microprocessors used today.
- The ISA of a processor can be described using 5 categories:

**Operand Storage in the CPU**
Where are the operands kept other than in memory?

**Number of explicit named operands**
How many operands are named in a typical instruction.

**Operand location**
Can any ALU instruction operand be located in memory? Or must all operands be kept internally in the CPU?

**Operations**
What operations are provided in the ISA.

**Type and size of operands**
What is the type and size of each operand and how is it specified?

Of all the above the most distinguishing factor is the first.
The 3 most common types of ISAs are:
***1.Stack*** - The operands are implicitly on top of the stack.
***2.Accumulator*** - One operand is implicitly the accumulator.
*3.General Purpose Register (GPR)* - All operands are explicitly mentioned, they are either registers or memory locations.

We will look at the assembly code of
C = A + B;
in all 3 architectures:

- Lets look at the assembly code of
- C = A + B; in all 3 architectures:

| **Stack** | **Accumulator** | **GPR** |
|---|---|---|
| PUSH A | LOAD A | LOAD R1,A |
| PUSH B | ADD B | ADD R1,B |
| ADD | STORE C | STORE C,R1 |
| POP C | - | - |

# Advantages and disadvantages of each of these approaches?

**Stack:**
**Advantages:** Simple Model of expression evaluation (reverse polish). Short instructions.
**Disadvantages:** A stack can't be randomly accessed This makes it hard to generate efficient code. The stack itself is accessed every operation and becomes a bottleneck.

**Accumulator:**
**Advantages:** Short instructions.
**Disadvantages:** The accumulator is only temporary storage so memory traffic is the highest for this approach.

**GPR:**
**Advantages:** Makes code generation easy. Data can be stored for long periods in registers.
**Disadvantages:** All operands must be named leading to longer instructions.

- Earlier CPUs were of the first 2 types but in the last 15 years all CPUs made are GPR processors.
- The 2 major reasons are that registers are faster than memory, the more data that can be kept internally in the CPU the faster the program will run.
- The other reason is that registers are easier for a compiler to use.

- As we mentioned before most modern CPUs are of the GPR (General Purpose Register) type. A few examples of such CPUs are the IBM 360, DEC VAX, Intel 80x86 and Motorola 68xxx.
- But while these CPUS were clearly better than previous stack and accumulator based CPUs they were still lacking in several areas:
- Instructions were of varying length from 1 byte to 6-8 bytes. This causes problems with the pre-fetching and pipelining of instructions.
- ALU (Arithmetic Logical Unit) instructions could have operands that were memory locations. Because the number of cycles it takes to access memory varies so does the whole instruction. This isn't good for compiler writers, pipelining and multiple issue.
- Most ALU instructions had only 2 operands where one of the operands is also the destination. This means this operand is destroyed during the operation or it must be saved before somewhere.

# ISA

- The ISA defines the functions performed by the CPU. The instruction set is the programmer's means of controlling the CPU. Thus programmer requirements must be considered in designing the instruction set.
- Most important and fundamental design issues:
- **Operation repertoire : How many and which operations to provide, and how complex operations** should be.
- **Data Types : The various type of data upon which operations are performed.**
- **Instruction format : Instruction length (in bits), number of addresses, size of various fields and so on.**
- **Registers : Number of CPU registers that can be referenced by instructions and their** use.
- **Addressing : The mode or modes by which the address of an operand is specified.**

# ISA - Elements of an Instruction

Each instruction must contain following information required by the CPU for execution.

**Operation Code:**

- Specifies the operation to be performed (e.g., add, move etc.). The operation is specified by a binary code, know as the *operation code or opcode.*

**Source operand reference:**

- The operation may involve one or more source operands; that is, operands that are inputs for the operation.

**Result operand reference:**

- The operation may produce a result.

**Next instruction reference:**

- This tells the CPU where to fetch the next instruction after the execution of this instruction is complete.

# ISA

- Operation Code:
  - – Most basic part of instruction is its operation part aka operation code.
  - – Operation code defines operations  such as ADD, SHIFT etc.
  - – No. Of bits in Opcode depends on available operations for that particular computer
  - – Opcode with n-bits can specify $2^n$ instructions.

- Source and result operands can be in one of the three areas:
  - – main or virtual memory,
  - – CPU register or
  - – I/O device.

# Instruction Types

**1. Data Processing:**

- Arithmetic and Logic instructions Arithmetic instructions provide computational capabilities for processing numeric data. Logic (Boolean) instructions operate on the bits of a word as bits rather than as numbers. Logic instructions thus provide capabilities for processing any other type of data. There operations are performed primarily on data in CPU registers.

**2. Data Storage:**

- Memory instructions Memory instructions are used for moving data between memory and CPU registers.

**3. Data Movement:**

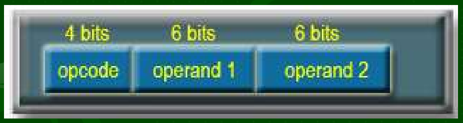
- I/O instructions I/O instructions are needed to transfer program and data into memory from storage device or input device and the results of computation back to the user.

**4. Control:**

- Test and branch instructions Test instructions are used to test the value of a data word or the status of a computation. Branch instructions are then used to branch to a different set of instructions depending on the decision made.

# Instruction Format:

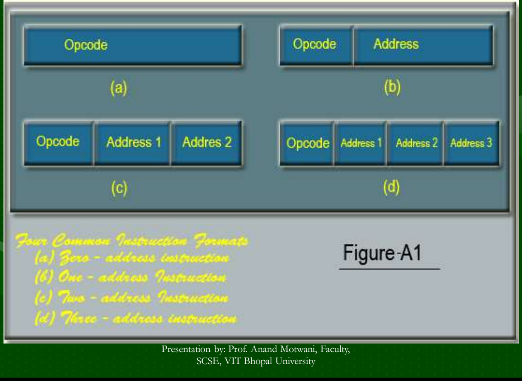
- Size and meaning of fields within the instruction.
- Or Layout of the bits of an instruction
- Example: A simple instruction format



## Instruction Format

- An instruction format must include an opcode and, implicitly or explicitly, zero or more operands.
- Each explicit operand is referenced using one of the addressing mode that is available for that machine. The format must, implicitly or explicitly, indicate the addressing mode of each operand.
- For most instruction sets, more than one instruction format is used.
- Four common instruction format are there.

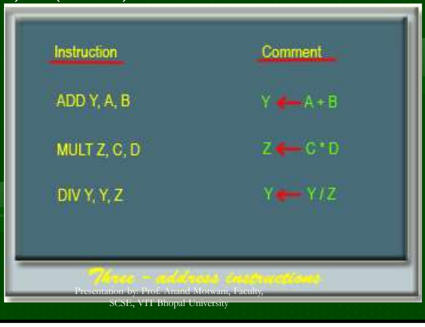
## 4 common Instruction formats



Four Common Instruction Formats
(a) Zero - address instruction
(b) One - address Instruction
(c) Two - address Instruction
(d) Three - address instruction

Figure A1

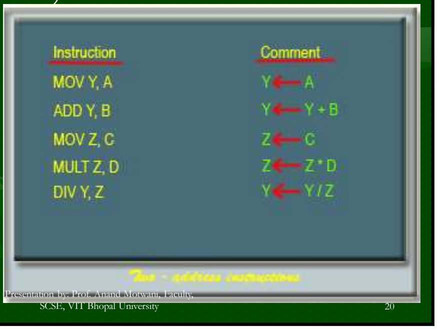
- Y= (A + B) / (C * D)



| Instruction | Comment |
|---|---|
| ADD Y, A, B | Y ← A + B |
| MULT Z, C, D | Z ← C * D |
| DIV Y, Y, Z | Y ← Y / Z |

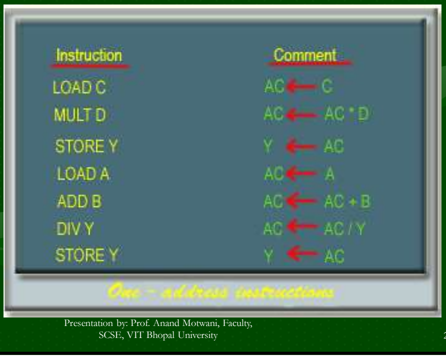
Three - address instructions

- Y= (A + B) / (C * D)



| Instruction | Comment |
|---|---|
| MOV Y, A | Y ← A |
| ADD Y, B | Y ← Y + B |
| MOV Z, C | Z ← C |
| MULT Z, D | Z ← Z * D |
| DIV Y, Z | Y ← Y / Z |

Two - address instructions



| Instruction | Comment |
|---|---|
| LOAD C | AC ← C |
| MULT D | AC ← AC * D |
| STORE Y | Y ← AC |
| LOAD A | AC ← A |
| ADD B | AC ← AC + B |
| DIV Y | AC ← AC / Y |
| STORE Y | Y ← AC |

One – address instructions

# Zero Address Instruction

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

| | |
|---|---|
| LOAD A | AC <- M[A] |
| PUSH A | TOS <- A |
| PUSH B | TOS <- B |
| ADD | TOS <- (A + B) |
| PUSH C | TOS <- C |
| PUSH D | TOS <- D |
| ADD | TOS <- (C + D) |
| MUL | TOS <- (C + D) x (A + B) |
| POP X | M[X] <- TOS |

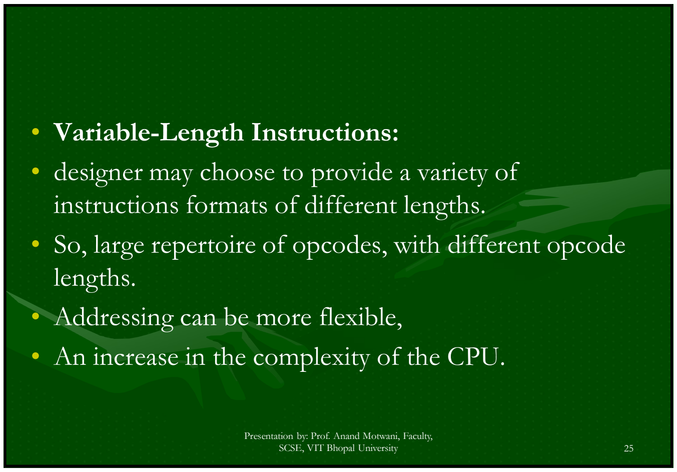
# Three address instruction

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

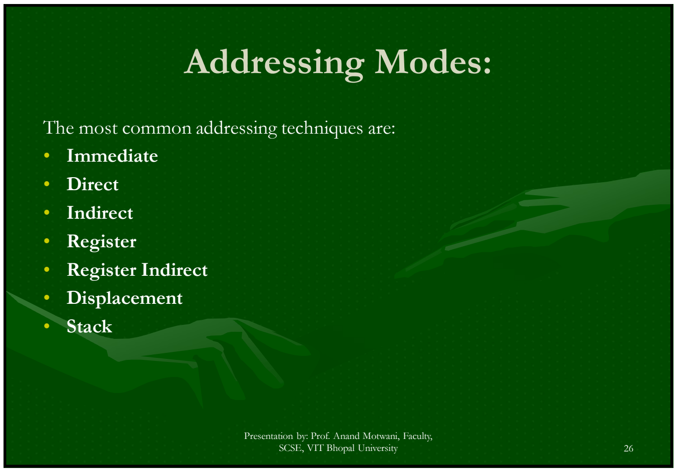
| | |
|---|---|
| ADD R1, A, B | R1 <- M[A] + M[B] |
| ADD R2, C, D | R2 <- M[C] + M[D] |
| MUL X, R1, R2 | M[X] <- R1 x R2 |

# Instruction Length

- We already discussed.
- in most of the case there is a correlation between memory transfer length and word length of the machine.



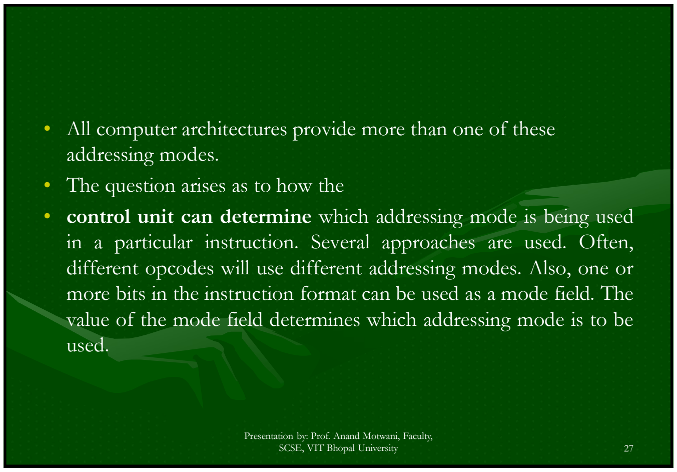
- **Variable-Length Instructions:**
- designer may choose to provide a variety of instructions formats of different lengths.
- So, large repertoire of opcodes, with different opcode lengths.
- Addressing can be more flexible,
- An increase in the complexity of the CPU.



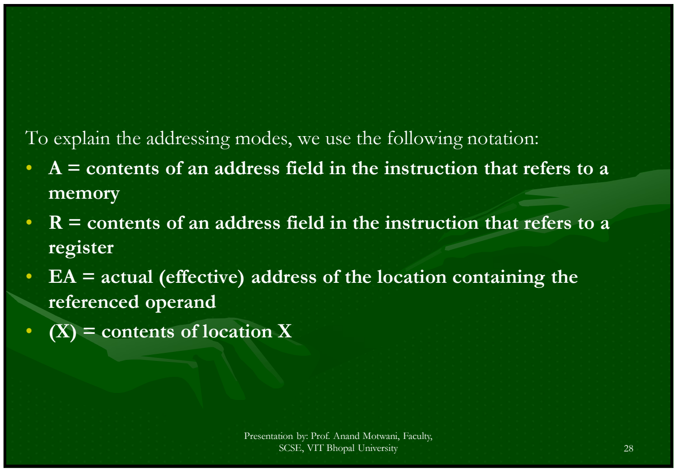
## Addressing Modes:

The most common addressing techniques are:

- **Immediate**
- **Direct**
- **Indirect**
- **Register**
- **Register Indirect**
- **Displacement**
- **Stack**

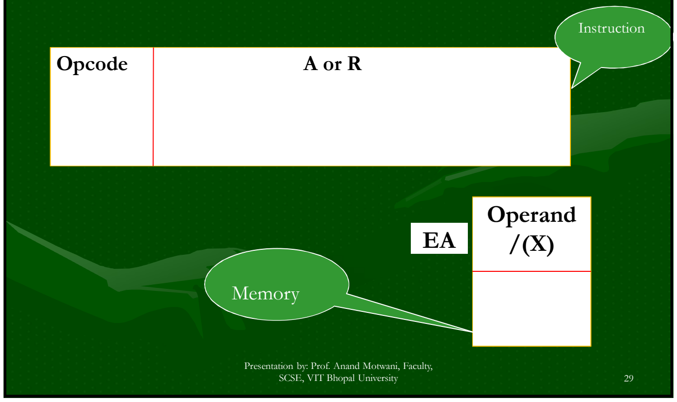


- All computer architectures provide more than one of these addressing modes.
- The question arises as to how the
- **control unit can determine** which addressing mode is being used in a particular instruction. Several approaches are used. Often, different opcodes will use different addressing modes. Also, one or more bits in the instruction format can be used as a mode field. The value of the mode field determines which addressing mode is to be used.



To explain the addressing modes, we use the following notation:

- **A = contents of an address field in the instruction that refers to a memory**
- **R = contents of an address field in the instruction that refers to a register**
- **EA = actual (effective) address of the location containing the referenced operand**
- **(X) = contents of location X**



Instruction

Opcode | A or R

EA

Operand /(X)

Memory

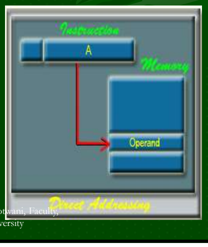
# Immediate Addressing:

Instruction
Operand

- The simplest form of addressing is immediate addressing, in which the operand is actually present in the instruction:
- OPERAND = A
- This mode can be used to define and use constants or set initial values of variables. The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand. The disadvantage is that the size of the number is restricted to the size of the address field, which, in most instruction sets, is small compared with the world length.

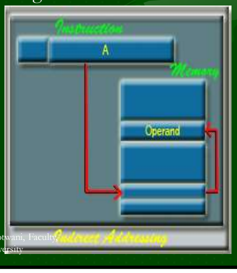
# Direct Addressing:

- A very simple form of addressing is direct addressing, in which the address field contains the effective address of the operand:
- EA = A
- It requires only one memory reference and no special calculation.



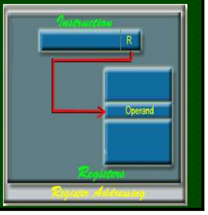
# Indirect Addressing:

- With direct addressing, the length of the address field is usually less than the word length, thus limiting the address range. One solution is to have the address field refer to the address of a word in memory, which in turn contains a full-length address of the operand. This is know as indirect addressing:
- EA = (A)



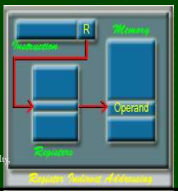
# Register Addressing:

- Register addressing is similar to direct addressing. The only difference is that the address field refers to a register rather than a main memory address:
- EA = R
- The advantages of register addressing are that only a small address field is needed in the instruction and no memory reference is required. The disadvantage of register addressing is that the address space is very limited.



# Register Indirect Addressing:

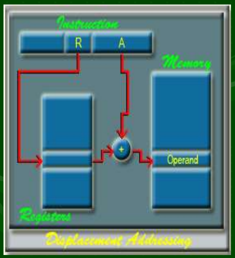
- Register indirect addressing is similar to indirect addressing, except that the address field refers to a register instead of a memory location. It requires only one memory reference and no special calculation.
- EA = (R)
- Register indirect addressing uses one less memory reference than indirect addressing. Because, the first information is available in a register which is nothing but a memory address. From that memory location, we use to get the data or information. In general, register access is much more faster than the memory access.



# Displacement Addressing:

- A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing, which is broadly categorized as displacement addressing:
- EA = A + (R)
- Displacement addressing requires that the instruction have two address fields, at least one of which is explicit. The value contained in one address field (value = A) is used directly. The other address field, or an implicit reference based on opcode, refers to a register whose contents are added to A to produce the effective address. Three of the most common use of displacement addressing are:

# Displacement Addressing:



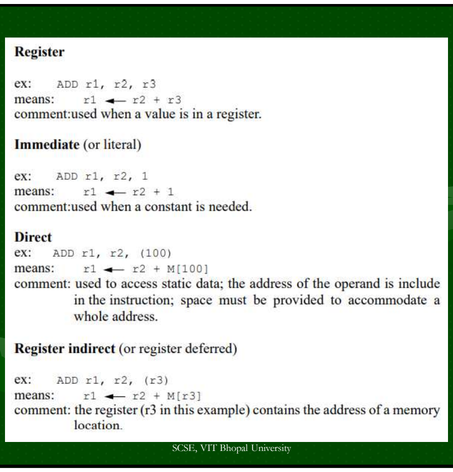
# Displacement Addressing

Depending upon the use and implementation this addressing scheme may be known as:

- Relative addressing
- Base-register addressing
- Indexing

- **Relative Addressing:**
- For relative addressing, the implicitly referenced register is the *program counter (PC). That is, the* current instruction address is added to the address field to produce the EA. Thus, the effective address is a displacement relative to the address of the instruction.
- **Base-Register Addressing:**
- The reference register contains a memory address, and the address field contains a displacement from that address. The register reference may be *explicit or implicit.*
- In some implementation, a single segment/base register is employed and is used implicitly. In others, the programmer may choose a register to hold the base address of a segment, and the instruction must reference it explicitly.

- In general-purpose-register (GPR) machines, the ones we are concerned at most, an addressing mode may specify:
- a constant;
- a register;
- a memory location.



**Register**

ex: `ADD r1, r2, r3`
means: `r1 ← r2 + r3`
comment:used when a value is in a register.

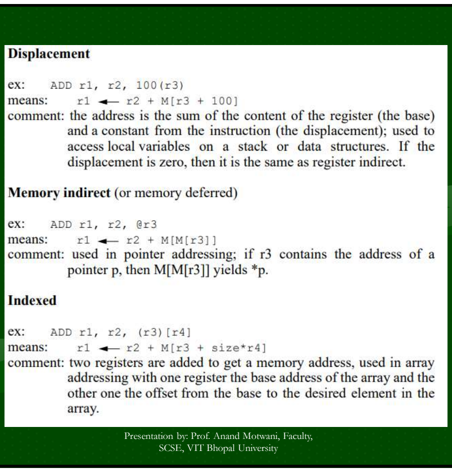
**Immediate** (or literal)

ex: `ADD r1, r2, 1`
means: `r1 ← r2 + 1`
comment:used when a constant is needed.

**Direct**
ex: `ADD r1, r2, (100)`
means: `r1 ← r2 + M[100]`
comment: used to access static data; the address of the operand is include in the instruction; space must be provided to accommodate a whole address.

**Register indirect** (or register deferred)

ex: `ADD r1, r2, (r3)`
means: `r1 ← r2 + M[r3]`
comment: the register (r3 in this example) contains the address of a memory location.



**Displacement**

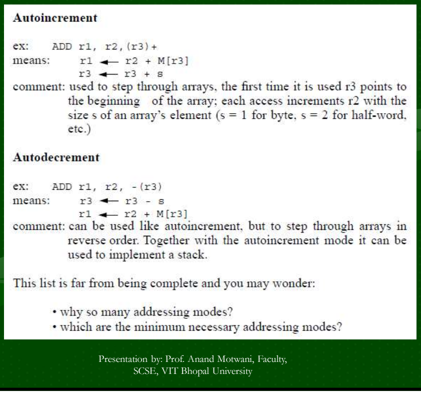
ex: `ADD r1, r2, 100(r3)`
means: `r1 ← r2 + M[r3 + 100]`
comment: the address is the sum of the content of the register (the base) and a constant from the instruction (the displacement); used to access local variables on a stack or data structures. If the displacement is zero, then it is the same as register indirect.

**Memory indirect** (or memory deferred)

ex: `ADD r1, r2, @r3`
means: `r1 ← r2 + M[M[r3]]`
comment: used in pointer addressing; if r3 contains the address of a pointer p, then M[M[r3]] yields *p.

**Indexed**

ex: `ADD r1, r2, (r3)[r4]`
means: `r1 ← r2 + M[r3 + size*r4]`
comment: two registers are added to get a memory address, used in array addressing with one register the base address of the array and the other one the offset from the base to the desired element in the array.



**Autoincrement**

ex: `ADD r1, r2,(r3)+`
means: `r1 ← r2 + M[r3]`
`r3 ← r3 + s`
comment: used to step through arrays, the first time it is used r3 points to the beginning of the array; each access increments r2 with the size s of an array's element (s = 1 for byte, s = 2 for half-word, etc.)

**Autodecrement**

ex: `ADD r1, r2, -(r3)`
means: `r3 ← r3 - s`
`r1 ← r2 + M[r3]`
comment: can be used like autoincrement, but to step through arrays in reverse order. Together with the autoincrement mode it can be used to implement a stack.

This list is far from being complete and you may wonder:

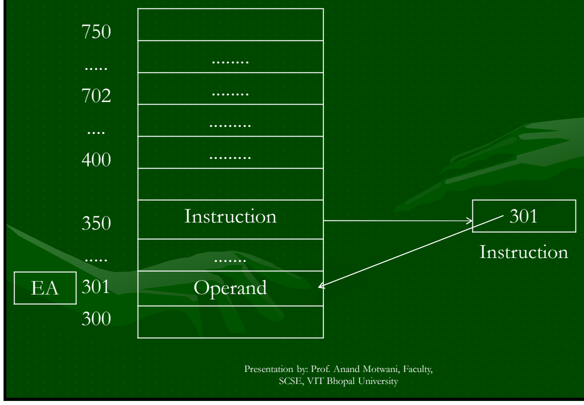
- why so many addressing modes?
- which are the minimum necessary addressing modes?

- why so many addressing modes?
- The addressing modes significantly reduce instruction count of programs (keep in mind that this comes with the increased hardware complexity).
- • which are the minimum necessary addressing modes?
- the answer depends upon the architecture; for a load-store architecture register, immediate, and register deferred are sufficient. From this three we can create the equivalent of other addressing modes

# 1. Numerical Problem

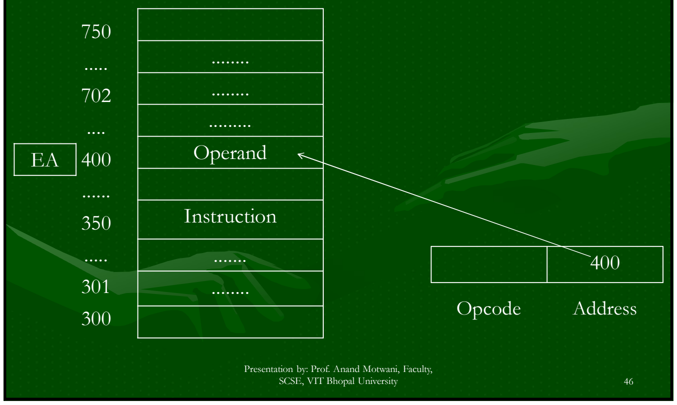
- An instruction is stored at location 350 with its address field at location 301. The address field has the value 400. A processor Register R1 contains the number 200. Evaluate the EA if the addressing mode of the instruction is:
- 1. Immediate 2. Direct 3. Register Indirect
- 4. Relative 5. Index with R1 as the index register.

## 1. Direct



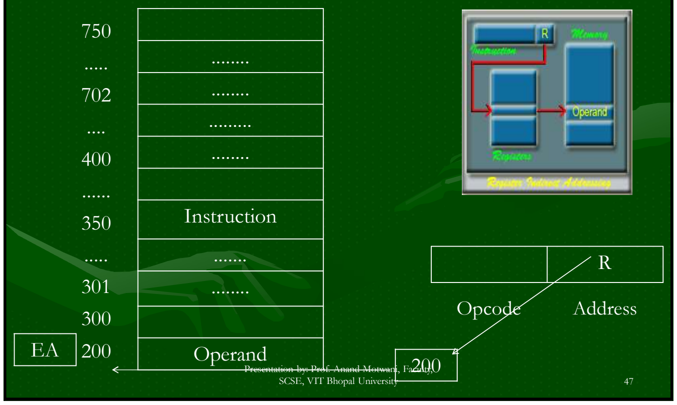
| | | |
|---|---|---|
| | 750 | |
| | ..... | ........ |
| | 702 | ........ |
| | .... | ......... |
| | 400 | ......... |
| | | |
| | 350 | Instruction |
| | ..... | ....... |
| EA | 301 | Operand |
| | 300 | |

## 2. Immediate



| | | |
|---|---|---|
| | 750 | |
| | ..... | ........ |
| | 702 | ........ |
| | .... | ......... |
| EA | 400 | Operand |
| | ...... | |
| | 350 | Instruction |
| | ..... | ....... |
| | 301 | ........ |
| | 300 | |

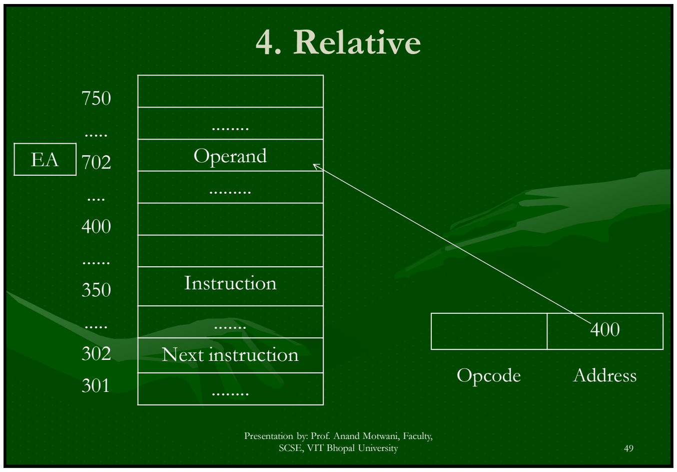
## 3. Register Indirect



| | | |
|---|---|---|
| | 750 | |
| | ..... | ........ |
| | 702 | ........ |
| | .... | ......... |
| | 400 | ........ |
| | ...... | |
| | 350 | Instruction |
| | ..... | ....... |
| | 301 | ........ |
| | 300 | |
| EA | 200 | Operand |

## 4. Relative

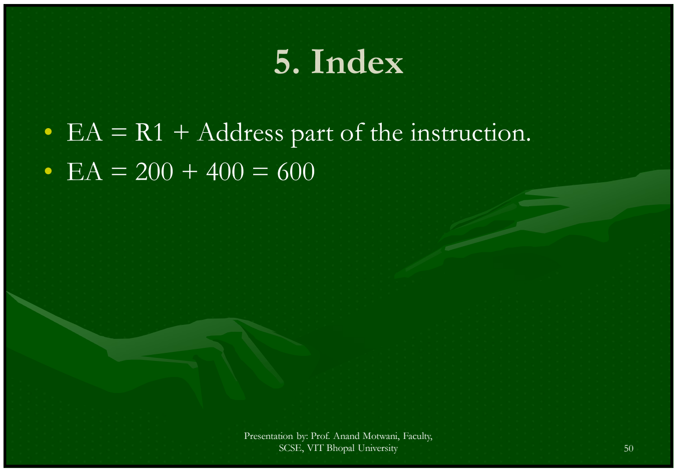
- EA = PC + address part in the instruction
- EA = 302 + 400 = 702



## 4. Relative

| EA |  |  |
|---|---|---|
|  | 750 |  |
|  | ..... | ........ |
| EA | 702 | Operand |
|  | .... | ......... |
|  | 400 |  |
|  | ...... |  |
|  | 350 | Instruction |
|  | ..... | ....... |
|  | 302 | Next instruction |
|  | 301 | ........ |

| Opcode | Address |
|---|---|
|  | 400 |



## 5. Index

- EA = R1 + Address part of the instruction.
- EA = 200 + 400 = 600

# 2. Numerical Problem

- A computer has 32 bit instruction and 12 bit address. If there are 250 two address instructions.
- Draw instruction format.
- How may one address instructions can be formed?

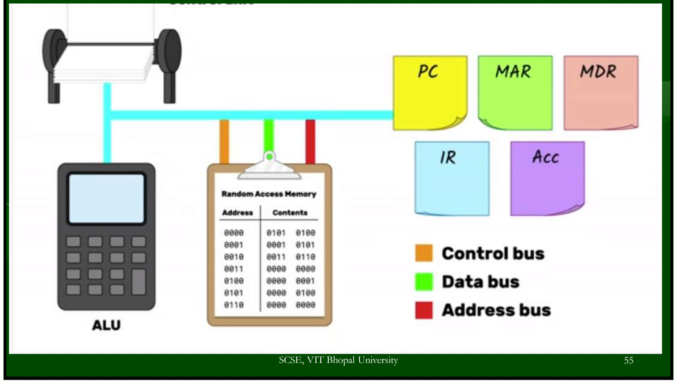
- Solution on next slide

- Solution
- Instruction is two address instruction format

| 8 bit<br>Opcode | 12 bit<br>Address | 12 bit<br>Address |
|---|---|---|

- Answer: One address instructions = 6

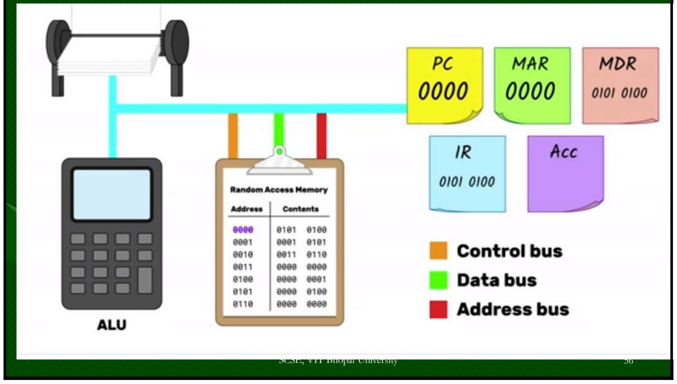
# Instruction Cycle: Fetch



Random Access Memory

| Address | Contents | |
|---|---|---|
| 0000 | 0101 | 0100 |
| 0001 | 0001 | 0101 |
| 0010 | 0011 | 0110 |
| 0011 | 0000 | 0000 |
| 0100 | 0000 | 0001 |
| 0101 | 0000 | 0100 |
| 0110 | 0000 | 0000 |

# Instruction Cycle: Fetch

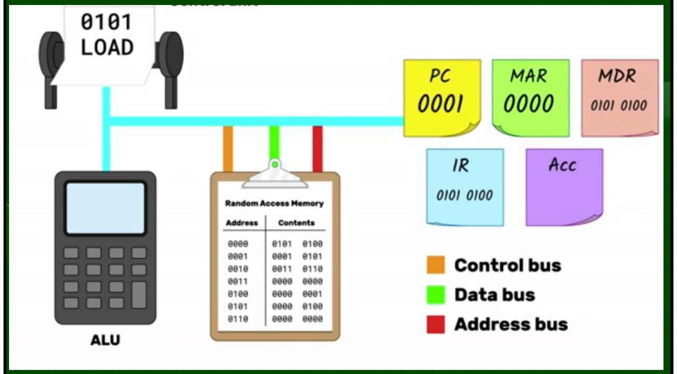


Random Access Memory

| Address | Contents | |
|---|---|---|
| 0000 | 0101 | 0100 |
| 0001 | 0001 | 0101 |
| 0010 | 0011 | 0110 |
| 0011 | 0000 | 0000 |
| 0100 | 0000 | 0001 |
| 0101 | 0000 | 0100 |
| 0110 | 0000 | 0000 |

- After completion of 1 fetch cycle:
- PC = 0001

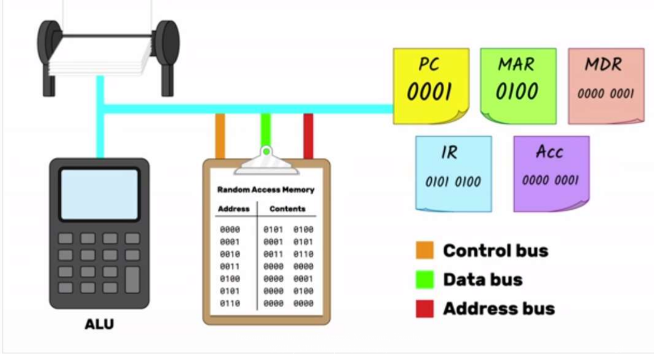
# Instruction Cycle: Decode



Random Access Memory

| Address | Contents |
|---|---|
| 0000 | 0101 0100 |
| 0001 | 0001 0101 |
| 0010 | 0011 0110 |
| 0011 | 0000 0000 |
| 0100 | 0000 0001 |
| 0101 | 0000 0100 |
| 0110 | 0000 0000 |

# Instruction Cycle: Execute



Random Access Memory

| Address | Contents |
|---|---|
| 0000 | 0101 0100 |
| 0001 | 0001 0101 |
| 0010 | 0011 0110 |
| 0011 | 0000 0000 |
| 0100 | 0000 0001 |
| 0101 | 0000 0100 |
| 0110 | 0000 0000 |