Computer Architecture (Overview of ISA)

विद्याधनं सर्वधन प्रधानम्

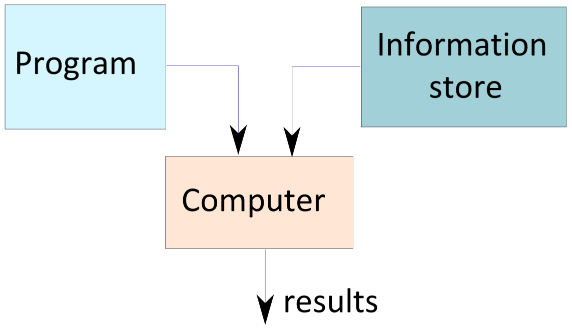
Subhasis Bhattacharjee
Department of Computer Science and Engineering,
Indian Institute of Technology, Jammu

September 1, 2023

Outline I

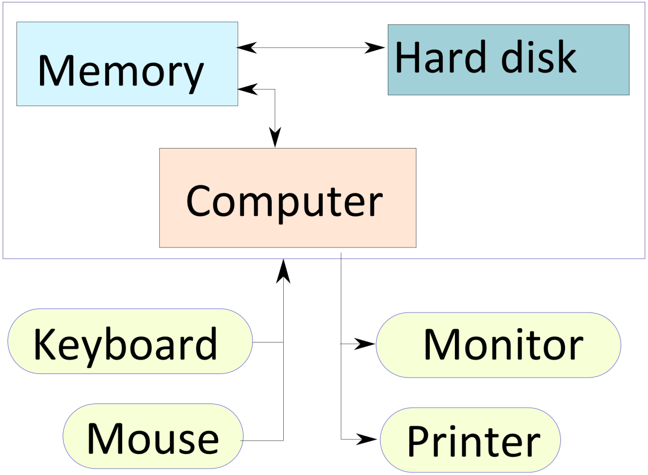
Working of Computer Systems

Computer System Working

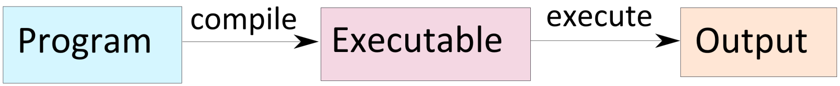


- **Program**: List of instructions given to the computer
- **Information store**: data, images, files, videos
- **Computer**: Process the information store according to the instructions in the program

Computer System with Peripheral Devices



How to Instruct a Computer


$$\begin{tikzcd}\boxed{\text{Program}} \arrow[r, "\text{compile}"] & \boxed{\text{Executable}} \arrow[r, "\text{execute}"] & \boxed{\text{Output}}\end{tikzcd}$$

- Write a program in a high level language – C, C++, Java
- Compile it into a format that the computer understands
- Execute the program

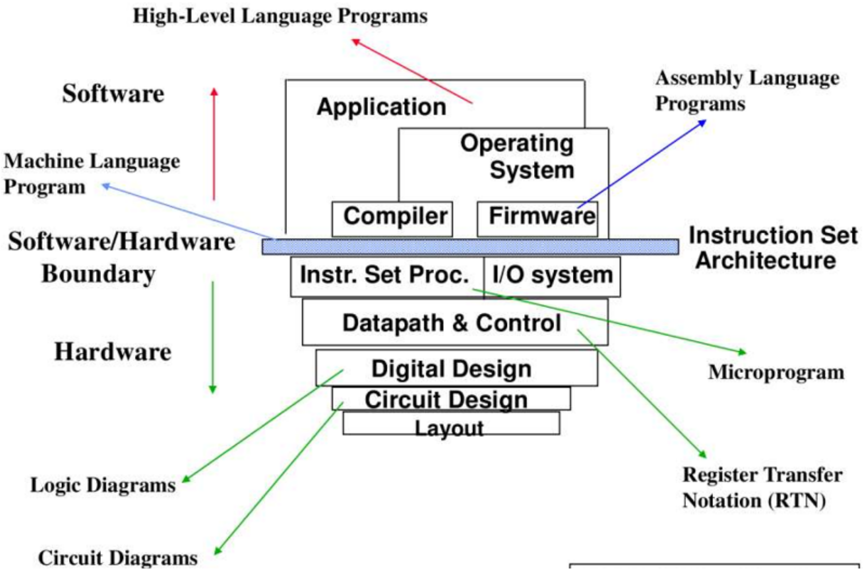
What Can a Computer Understand

- Computer can NOT understand High-Level instructions of the form
  - Multiply two matrices
  - Compute the determinant of a matrix
  - Find the shortest path between Mumbai and Delhi
- Computer understands:
  - Add a + b to get c
  - Multiply a * b to get c

The Language of Instructions

- Humans can understand:
  - Complicated sentences (in various languages eg., English, French, Spanish)
- Computers can understand:
  - Very simple instructions.
  - Informally that is a part of ISA.

Hierarchy in Computer Architecture



Instruction Set Architecture

Instruction Set Architecture (ISA)

An instruction set architecture (ISA) is an abstract model of a computer. A device that executes instructions described by that ISA, such as a **Processor**, is called an implementation.

Elements of an Instruction Set Architecture (ISA)

ISA Contains:

- defines the instructions / Instruction Set:
  - Syntax and semantics of all the instructions
  - Operands of instructions
  - Defines the data types
  - Encoding of instruction
- Fundamental features:
  - Registers - their sizes and usage
  - Amount of memory support
  - Addressing modes
  - Virtual memory
  - Hardware support for managing main memory
  - Input/output model

Features of an ISA – Complete, Concise, Generic, Simple

Usual Instruction Set in an ISA

Example of instructions in an ISA

- Arithmetic instructions : add, sub, mul, div
- Logical instructions : and, or, not
- Data transfer/movement instructions

Features of an ISA – II

Completeness

It should be able to implement all the programs that users may write.

Concise

- The instruction set should have a limited size.
- Typically an ISA contains 32-1000 instructions.

Generic

- Instructions should not be too specialized, e.g. add14 (adds a number with 14) instruction is too specialized

Simple

Should not be very complicated.

Classification of ISA

Classification of ISA - Based on Complexity / Size

- How many instructions should we have ?
- What should they do ?
- How complicated should they be ?

Two different paradigms

The division is NOT very Exclusive / Distinctive.

- Reduced Instruction Set Computer (RISC)
- Complex Instruction set Computer (CISC)

RISC vs CISC

RISC

A reduced instruction set computer (RISC) implements simple instructions that have a simple and regular structure. The number of instructions is typically a small number (64 to 128).
Examples: ARM, IBM PowerPC, HP PA-RISC

CISC

A complex instruction set computer (CISC) implements complex instructions that are highly irregular, take multiple operands, and implement complex functionalities. Secondly, the number of instructions is large (typically 500+).
Examples: Intel x86, VAX

Completeness of an ISA

Completeness of an ISA

Complete means:

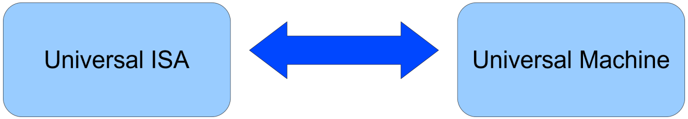
- Can implement all types of programs.
- How to ensure that we have just enough instructions such that we can implement every possible program that we might want to write?

Incomplete Instruction Set - Example

- Let IS contains only **add** instruction.
- We cannot **subtract**
- So, **NOT Complete**.

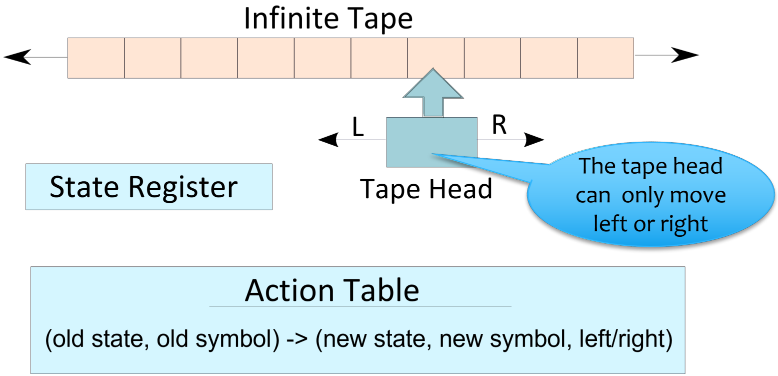
Is there any Theoretical Result on Completeness?

- Is there an universal ISA ?
- The universal machine has a set of basic actions, and each such action can be interpreted as an instruction.



Turing Machines

Turing Machine - developed by Alan Turing


Infinite Tape
L
R
Tape Head
The tape head can only move left or right
State Register
Action Table
(old state, old symbol) -> (new state, new symbol, left/right)

Operation of a Turing Machine

- There is an infinite tape that extends to the left and right (consists of an infinite number of cells)
- The tape head points to a cell, and can either move 1 cell to the left or right
- Based on the symbol in the cell, and its current state, the Turing machine computes the transition:
  - Computes the next state
  - Overwrites the symbol in the cell (or keeps it the same)
  - Moves to the left or right by 1 cell
- The action table records the rules for the transitions.

Solving Problems on Turing Machine

Example of Problem Solving using Turing Machine

You may also see: **Examples of Turing Machines - Online**

- This machine is extremely simple, and extremely powerful
- We can solve all kinds of problems – mathematical problems, engineering analyses, protein folding, computer games, etc.
- But may be a bit difficult to formulate the solution!!!

ISA Completeness - rely on Church-Turing Thesis

Church-Turing Thesis - by Alonzo Church & Alan Turing

Any real-world computation can be translated into an equivalent computation involving a Turing machine.

- Note : It is a thesis, not a theorem.
- Nobody has found a counter-example.

Definition: Turing-complete / Turing completeness / computationally universal

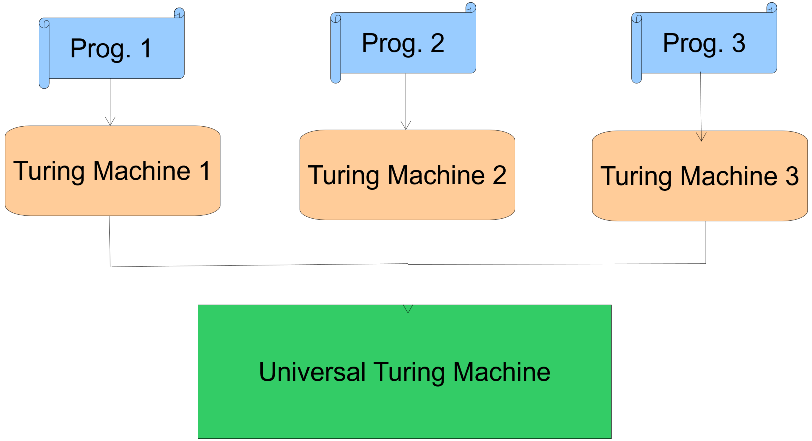
Any computing system that is equivalent to a Turing machine is said to be Turing complete.

Universal Machines

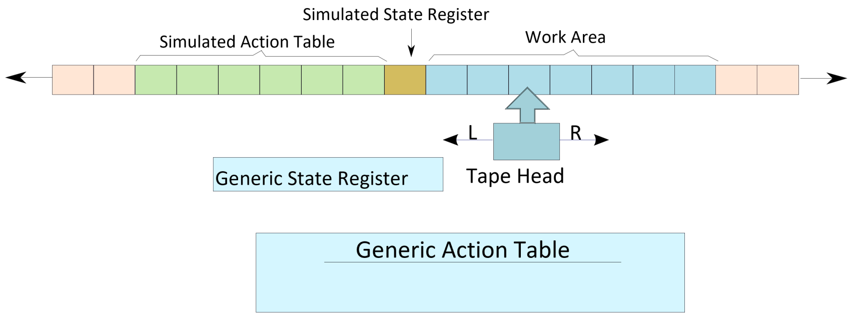
Universal Turing Machine

- For every problem in the world, we can design a Turing Machine (Church-Turing thesis)
- Can we design a universal Turing machine that can simulate any Turing machine. This will make it a **universal machine (UTM)**
- Why not ? The logic of a Turing machine is really simple. We need to move the tape head left, or right, and update the symbol and state based on the action table. **A UTM can easily do this.**
- A UTM needs to have an action table, state register, and tape that can simulate any arbitrary Turing machine.

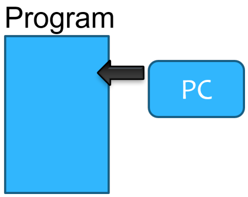
Universal Turing Machine



A Universal Turing Machine

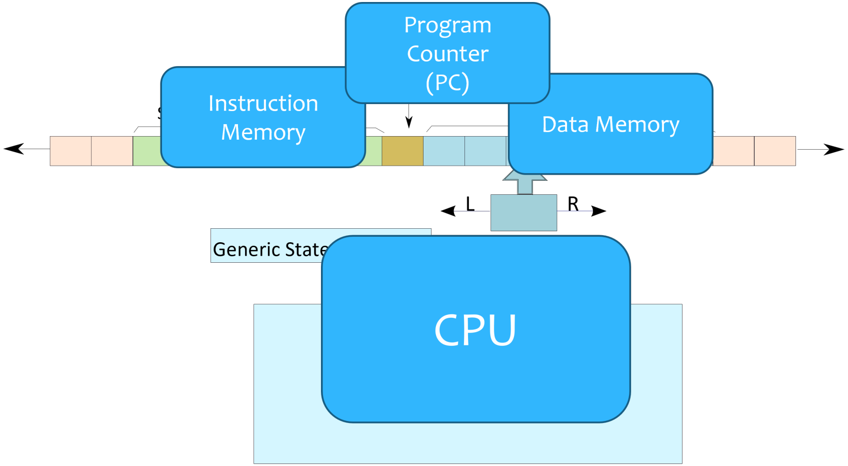


Program Execution

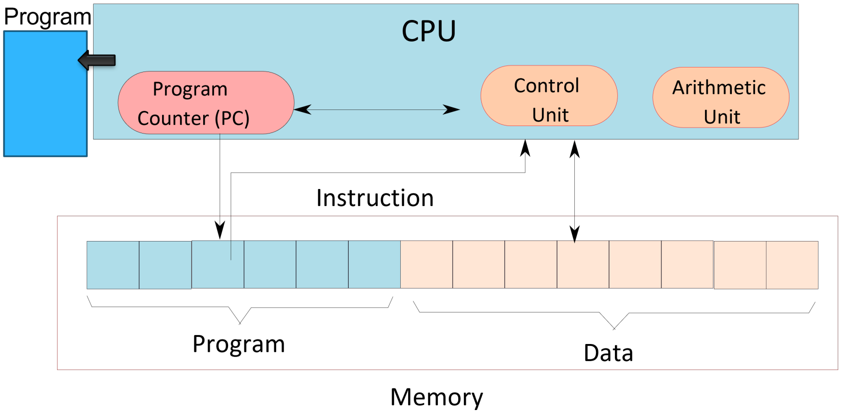


- Program: It is a list of instruction
- Program Counter (PC): It tracks the current instruction being processed.
  - Also called Instruction Pointer (IP) in Intel x86 and Itanium microprocessors
  - Sometimes called the instruction address register (IAR), the instruction counter.
  - Usually, the PC is incremented after fetching an instruction, and holds the memory address of ("points to") the next instruction that would be executed.

UTM with CPU, RAM and Program



Computer Inspired from the Turing Machine



Elements of a Computer

Memory (array of bytes) contains

- The program, which is a sequence of instructions
- The program data $\rightarrow$ variables, and constants

The program counter (PC) points to an instruction in a program

- After executing an instruction, it points to next instruction by default
- A branch instruction makes the PC point to another instruction (not in sequence)

CPU (Central Processing Unit) contains

- Program counter
- instruction execution units / Arithmetic Logic Unit (ALU)
- Control Unit (CU)

Designing Instruction Sets

IS Design - Example 1

IS-1 $\rightarrow$ Only Three instructions - Boolean Logic Operations

Instruction Snapshot	**Explanation**
AND c, a, b	c $\leftarrow$ a & b
OR c, c, b	c $\leftarrow$ a $\mid$ b
NOT b, a	b $\leftarrow$ ã

What can we say about this IS-1

- Complete: Yes. We can implement any Boolean function using only AND, OR and NOT.
- Concise: Yes. Only 3 Instructions
- Generic: Yes.
- Simple:
  - Yes. Each instruction is easy (directly coming from Boolean Algebra).
  - No. Very difficult / cumbersome to write other operations, eg, arithmetic operations. We need to develop them.

IS Design - Example 2

IS-2 $\rightarrow$ Only one instruction - Boolean Logic Operations

Instruction Snapshot	**Explanation**
NAND c, a, b	c $\leftarrow \overline{(a \,\&\, b)}$

What can we say about this IS-2

- Complete: Yes. We can implement any Boolean function using only NAND.
- Concise: Yes. Only 1 Instruction.
- Generic: Yes.
- Simple:
  - Yes. Instruction is easy (only NAND logic in Boolean Algebra).
  - No. More difficult / cumbersome than IS-1 for other operations, eg, arithmetic/and/or/not operations. We need to develop them.

IS Design - Example 3

IS-3 → Only one instruction - Arithmetic Operation

Instruction Snapshot	**Explanation**
ADD c, a, b	c ← a + b

What can we say about this IS-2

- Complete: NO.
  - How to do subtraction? Not possible.
  - How to do logical operations (AND, OR, NOT, NAND)? Not Possible.
- Concise, Generic, Simple: They are NOT important when IS is NOT Complete.

IS Design - Example 4

IS-4 → Only one instruction - Arithmetic Operation

Instruction	**Explanation**	**Meaning**
SBN a, b, L	Two tasks: (1) a ← a − b, (2) if Result is −**ve**, then jump to L	**S**ubtract and **B**ranch if **N**egative

What can we say about this IS-4

- Complete: Yes. We can implement any arithmetic function. We will see it soon.
- Concise: Yes. Only 1 Instruction.
- Generic: Yes.
- Simple:
  - Not Really. But not difficult also. A bit tricky one.
  - No. More difficult / cumbersome than IS-1 for other operations, eg, arithmetic operations. We need to develop them.

IS-4 Design - Check for Completeness - P1

Dow to do addition? a + b?

Using only **SBN** instruction

Program / Instructions	**Comments**
L1: SBN t, t, L2	set $t = 0$; $t \leftarrow 0$;
L2: SBN t, b, L3	$t \leftarrow -b$;
L3: SBN a, t, LExit	$a \leftarrow a - t$; $\equiv$ $a \leftarrow a - (-b)$;

IS-4 Design - Check for Completeness - P1

Add number $1 \cdots 10$

Using only **SBN** instruction

Program / Instructions	**Comments**
L1: sbn t, t, 2	t = 0
L2: sbn t, index, 3	t = -1 * index
L3: sbn sum, t, 4	sum += index
L4: sbn index, one, exit	index -= 1
L5: sbn t, t, 6	t = 0
L6: sbn t, one, 1	(0 - 1 < 0), hence goto L1

Single Instruction / Multi-Instruction ISA

Multiple Instruction ISA

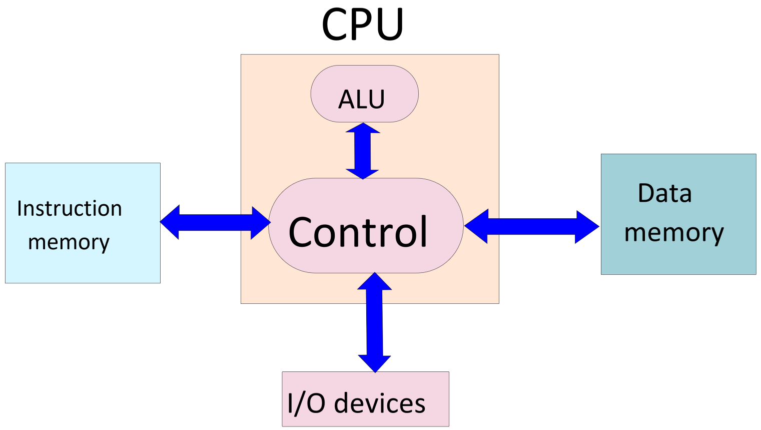
- **Arithmetic Instructions**
  - add, subtract, multiply, divide
- **Logical Instructions**
  - or, and, not
- **Move instructions**
  - Transfer values between memory locations
- **Branch instructions**
  - Move to a new program location, based on the values of some memory locations

Practical Machine Architectures

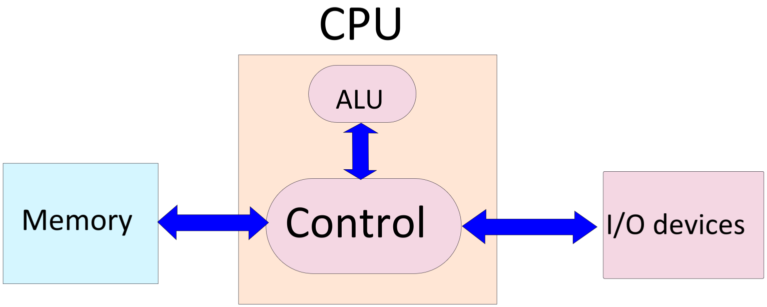
Designing Practical Machines

- Harvard Architecture
- Von-Neumann Architecture

Harvard Architecture



Von-Neumann Architecture



Problems with Harvard/ Von-Neumann Architectures

- The memory is assumed to be one large array of bytes
    - It is very very slow

General Rule: Larger is a structure, slower it is

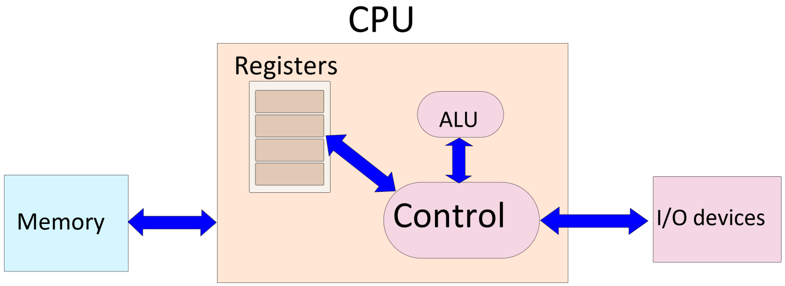
- Solution:
    - Have a small array of named locations (registers) that can be used by instructions
    - This small array is very fast

Insight: Accesses exhibit locality (tend to use the same variables frequently in the same window of time)

Uses of Registers

- **A CPU (Processor)** contains set of registers (16-64)
- These are named storage locations.
- Typically values are **loaded** from memory to registers.
- Arithmetic / logical instructions use registers as input operands.
- Output of operations are also kept in registers.
  - This is more advantageous. Why?
  - If previous output is required as inputs for subsequent instruction - we do not need to fetch operand again.
- Finally, data is **stored** back into their memory locations.

Machine with Registers



Example of a Program in Machine Language with Registers

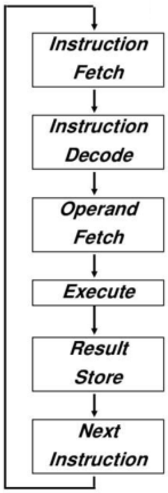
Program with registers

Program / Instructions	**Comments**
1: r1 = mem[b]	load b
2: r2 = mem[c]	load c
3: r3 = r1 + r2	add b and c.
4: mem[a] = r3	save the result

- r1, r2, and r3, are registers
- mem —>array of bytes representing memory

Instruction Processing on CPU

Instructions Processing by CPU



- **Instruction Fetch**: CPU retrieves instruction from memory.
  - PC knows where is the instruction.
- **Instruction Decode**: CPU interprets the instruction and determines
  - what operands are needed?
  - where are the operands?
  - what operation to be performed?
- **Operand Fetch**: CPU fetches the operand(s) from memory / registers.
- **Execute**: CPU performs the operation specified in the instruction.
- **Result Store**: CPU stores result into register / memory
- **Next Instruction**: Update PC to points to next instruction.

Few Details on Instruction Processing

- Instruction Format or Encoding: How is it decoded?
- Location of operands and result (addressing modes):
  - What kind of operands?
  - Where are the operands?
  - How many explicit operands?
  - How are memory operands located?
  - Which can or cannot be in memory?
- Operations: What operations are supported? (i.e., **Instruction Set**)
- Next Instruction: How to update PC?
  - Is it the next memory location?
  - Jumps, conditions, branches.

Instruction Processing - Some Alternatives

What could be the minimum?

- Instruction Fetch
- Instruction Decode
- Execute
- Next Instruction

What is happening here?

- Possibly **No Operand**.
- Or, Operands are at some fixed locations (Memory / Registers)
- Possibly **Result is also stored in Fixed location**.

Can we design such an ISA?

Yes. But - We will discuss after few weeks.

n-Address Machine

Opcode & n-Address Instruction

opcode

An **opcode** (i.e., **operation code**) is the portion of a machine language instruction that specifies the operation to be performed. Examples:

- mov r1, #15 → **mov** is an opcode
- add r1, r2, r3 → **add** is an opcode

n-Address Instruction

An n-Address Instruction is a form of instruction that consists of one opcode and *n* fields for operands.

Usually no restriction on the kind of operands

- Any of the *n* fields can be register, or memory, or, immediate value.
- Some can be immediate, some can be register
- Some instruction may operate only on registers

n-Address Machine

n-Address Machine / ISA

If Instruction Set (IS) contains only n-Address Instructions.

What type of machine is IS-4 - See Page 38

3 Address Machine.

This is supposed to be the definition / intention. But it is a bit different. Why?

Reasons:

- Difficult to design IS - where all instructions have same number of operands
- It may not be useful, or, It may reduce efficiency
- Usually an ISA contains instructions of different number of addresses

n-Address Machine - P2

n-Address Machine / ISA

If Instruction Set (IS) contains any instruction with at most n-Address Instructions.

3-Address Machine

IS can contain:

- mostly 3-Address Instructions
- (optional) some 0-Address Instruction, 1-Address Instruction, 2-Address Instruction

3-Address Instruction - Example

3-Address Instruction can be as follows:

Three-address is a form of machine-specific instruction and consists of one opcode and three fields for address. A single address field can indicate destination, and two address fields are for the source operands.
Example:

- add r0, r1, r2
- Explanation:
  - r0 = r1 + r2,
  - r0 is the destination,
  - r1, r2 are source operands

2-Address Instruction - Example

2-Address Machine can be as follows:

Two-address is a form of machine-specific instruction and consists of one opcode and two fields for address. A single address field can indicate a source operand as well as the destination, and another address field is the the source operand.
Example:

- add r0, r1
- Explanation:
  - r0 = r0 + r1,
  - r0 is the destination,
  - r0, r1 are source operands

1-Address Instruction - Example

1-Address Machine can be as follows:

One-address is a form of machine-specific instruction and consists of one opcode and only 1 field for address. A single address field can indicate one source operand.

- **How can we write add operation?**
- We can assume a special register (usually called **Accumulator**) that acts as both (1) destination, and (2) one source operand.
- Let AC denotes the special register **Accumulator**.

Example:

- add r1
- Explanation:
  - AC = AC + r1,
  - AC is the destination,
  - AC, r1 are source operands

1-Address Machine - How to Write Program

Let us first define IS

Assume 3 instructions are in IS. Here "b" can be a register or memory.

Instruction Snapshot	**Explanation**
LOAD b	AC $\leftarrow$ b
STORE b	b $\leftarrow$ AC
ADD b	AC $\leftarrow$ AC + b

How to do z = x + y

Program	**Explanation**
LOAD x	AC $\leftarrow$ x
ADD y	AC $\leftarrow$ AC + y
STORE z	z $\leftarrow$ AC

We can verify that we can write any program if IS is complete.

1-Address Machine is also called as **Accumulator Machine**

0-Address Instruction - Example

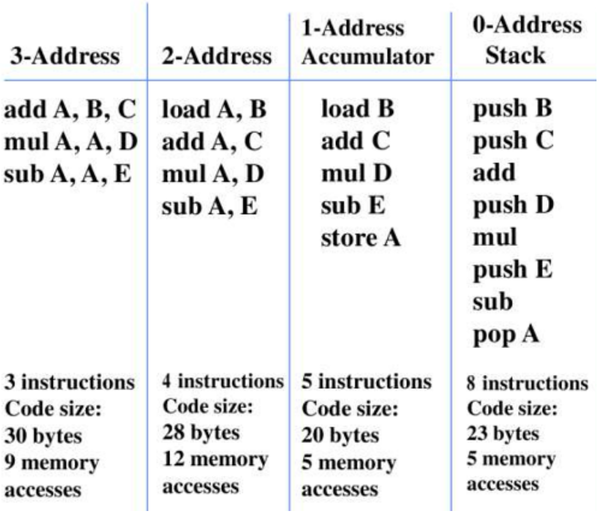
0-Address Machine / Stack Machine

- Zero-address is a form of machine-specific instruction and consists of only the opcode.
- A stack is used in the CPU.

We will NOT study this. I will give some idea later on.

Expression Evaluation on Various n-Address ISA

Evaluate: $A = (B - C) * D - E$



3-Address	2-Address	1-Address Accumulator	0-Address Stack
add A, B, C	load A, B	load B	push B
mul A, A, D	add A, C	add C	push C
sub A, A, E	mul A, D	mul D	add
	sub A, E	sub E	push D
		store A	mul
			push E
			sub
			pop A
3 instructions Code size: 30 bytes 9 memory accesses	4 instructions Code size: 28 bytes 12 memory accesses	5 instructions Code size: 20 bytes 5 memory accesses	8 instructions Code size: 23 bytes 5 memory accesses

ISA n-Address Trade-offs

- 3-address machine:
  - shortest code sequence
  - a large number of bits per instruction
  - large number of memory accesses.
- 0-address (stack) machine:
  - Longest code sequence
  - shortest individual instructions
  - more complex to program.

GPR & Load-Store ISA

Some More Variations on ISA

- General purpose register machine (GPR)
- Load-Store Machine

General purpose register machine (GPR)

- A few general purpose registers (GPRs)
- instruction contain address by specifying among GPR using a short register address
- Most of the address(es) in n-address are registers

Advantages of GPR

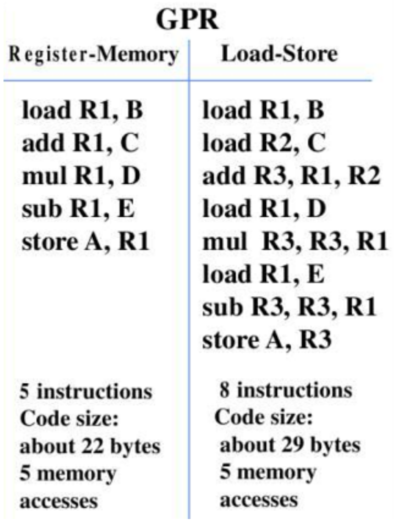
- Low number of memory accesses.
- Faster, since register access is much faster than memory access.
- Registers are easier for compilers to use
- Shorter, simpler instructions.

Load-Store Machine

- It is a GPR machine with some additional properties
- Memory addresses are only included in data movement instructions (loads / stores)
- All other instructions use only registers as source(s) & destination.

Expression Evaluation on GPR & Load-Store ISA

Evaluate: $A = (B - C) * D - E$



GPR	
Register-Memory	Load-Store
load R1, B add R1, C mul R1, D sub R1, E store A, R1	load R1, B load R2, C add R3, R1, R2 load R1, D mul R3, R3, R1 load R1, E sub R3, R3, R1 store A, R3
5 instructions Code size: about 22 bytes 5 memory accesses	8 instructions Code size: about 29 bytes 5 memory accesses

Summary of Various ISA

Complex Instruction Set Computer (CISC)

- Emphasizes doing more with each instruction.
- Motivated by the high cost of memory and hard disk capacity
- Original CISC architectures evolved with faster more complex CPU design
- Wide variety of addressing modes:
- Variable-length or hybrid instruction encoding is used.
- Examples: x86, x64, VAX

Reduced Instruction Set Computer (RISC)

- Focuses on reducing the number and complexity of instructions of the ISA.
- Reduced number of cycles needed per instruction
  - At least one instruction completed per clock cycle.
- Designed with CPU instruction **pipelining** in mind
- (Usually) Fixed-length instruction encoding
- Only load and store instructions access memory.
- Simplified addressing modes supported
  - Usually limited to immediate, register indirect, register displacement, indexed
- Delayed loads and branches
- Pre-fetch and speculative execution - Usually supported.
- Examples: ARM, MIPS, HP PA-RISC, SPARC, Alpha, POWER, PowerPC