Assembly Language

What Can a Computer Understand ?

* Computer can clearly NOT understand instructions of the form
  * Compute the determinant of a matrix
  * Find the shortest path between Mumbai and Delhi

McGraw Hill Education

The Language of Instructions

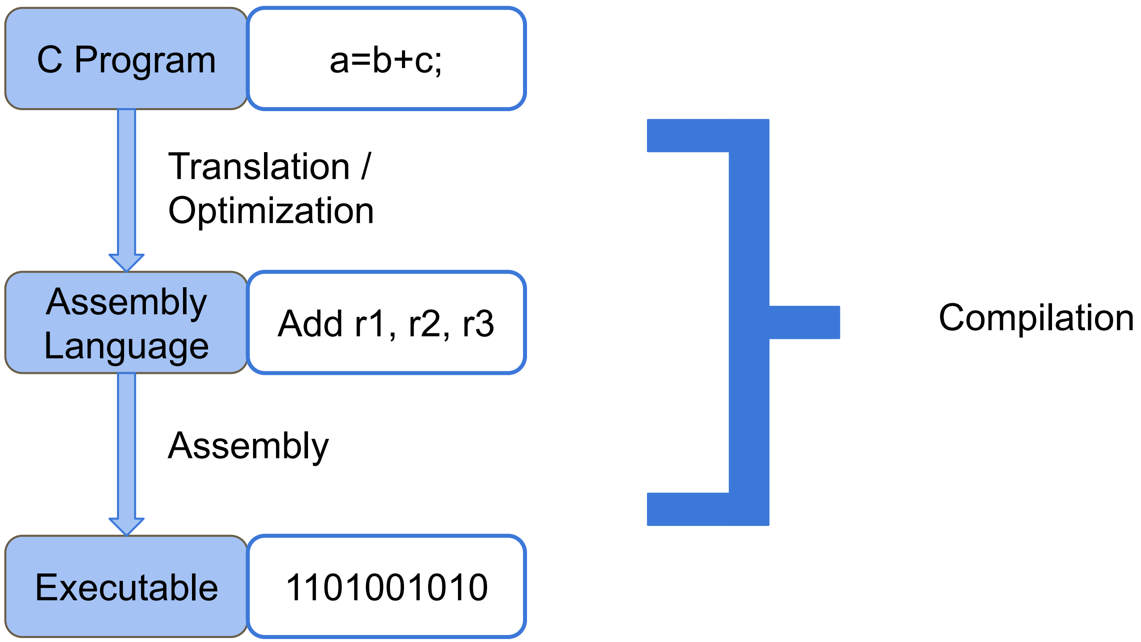
- Humans can understand complicated sentences
- Computers can understand
  - Very simple instructions
  - The semantics of all the instructions supported by a processor is known as its ***instruction set architecture*** (ISA). This includes the semantics of the instructions themselves, along with their operands, and interfaces with peripheral devices.

How to Instruct a Computer ?

$\text{Program} \xrightarrow{\text{compile}} \text{Executable} \xrightarrow{\text{execute}} \text{Output}$

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

Compilation and Assembly



Compilation and Assembly

- Write a simple program, and compile it using gcc
- The resultant *a.out* is in machine language, and in binary form
- Try *gcc -S test.c -o test.s* . *test.s* is the assembly program
- To get the binary from the assembly program, run *gcc test.s*

Popular Instruction Set Architectures (ISAs)

x86	Laptops, desktops, servers
ARM	Mobiles, Raspberry Pi
SPARC v8	Leon3
PowerPC	IBM machines
RISC-V	IITM’s Shakti processors
SimpleRISC	CS301
ToyRISC	CS311

Features of an ISA

* Complete
  * 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.

Features of an ISA – II

* Generic
  * Instructions should not be too specialized, e.g. add14 (adds a number with 14) is too specialized
* Simple
  * Should not be very complicated.

Designing an ISA

* Important questions that need to be answered :
  * How many instructions should we have ?
  * What should they do ?
  * How complicated should they be ?

RISC vs CISC

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

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

Let us now design an ISA …

* Single Instruction ISA
  * sbn – subtract and branch if negative

```
1: sbn a, b, 3    // a = a - b; if a < 0, jump to 3
2: sbn c, d, 3
3: sbn e, f, 1
...
```

Let us now design an ISA …

* Add (a + b) (assume temp = 0)

```
1: sbn temp, b, 2
2: sbn a, temp, exit
```

Single Instruction ISA - II

*

Initialization:

one = 1
index = 10
sum = 0

```
1: sbn temp, temp, 2
2: sbn temp, index, 3
3: sbn sum, temp, 4
4: sbn index, one, exit
5: sbn temp, temp, 6
6: sbn temp, one, 1
```

exit

Single Instruction ISA - II

*

```
Initialization:
        one  = 1
        index = 10
        sum = 0
```

```
1: sbn temp, temp, 2        // temp = 0
2: sbn temp, index, 3       // temp = -1 * index
3: sbn sum, temp, 4         // sum += index
4: sbn index, one, exit     // index -= 1
5: sbn temp, temp, 6        // temp = 0
6: sbn temp, one, 1         // (0 - 1 < 0), hence goto 1
```

exit

Single Instruction ISA - II

* Add the numbers – 1 … 10

```
Initialization:
        one  = 1
        index = 10
        sum = 0
```

```
1: sbn temp, temp, 2        // temp = 0
2: sbn temp, index, 3       // temp = -1 * index
3: sbn sum, temp, 4         // sum += index
4: sbn index, one, exit     // index -= 1
5: sbn temp, temp, 6        // temp = 0
6: sbn temp, one, 1         // (0 - 1 < 0), hence goto 1
```

exit

Single Instruction ISA - III

* Find whether a number is positive

- The given number is at address ‘number’
- If num is positive, write ‘0’ to address ‘result’; if negative, write ‘-1’
- Initialization: ‘one’ = 1

Single Instruction ISA - III

* Find whether a number is positive

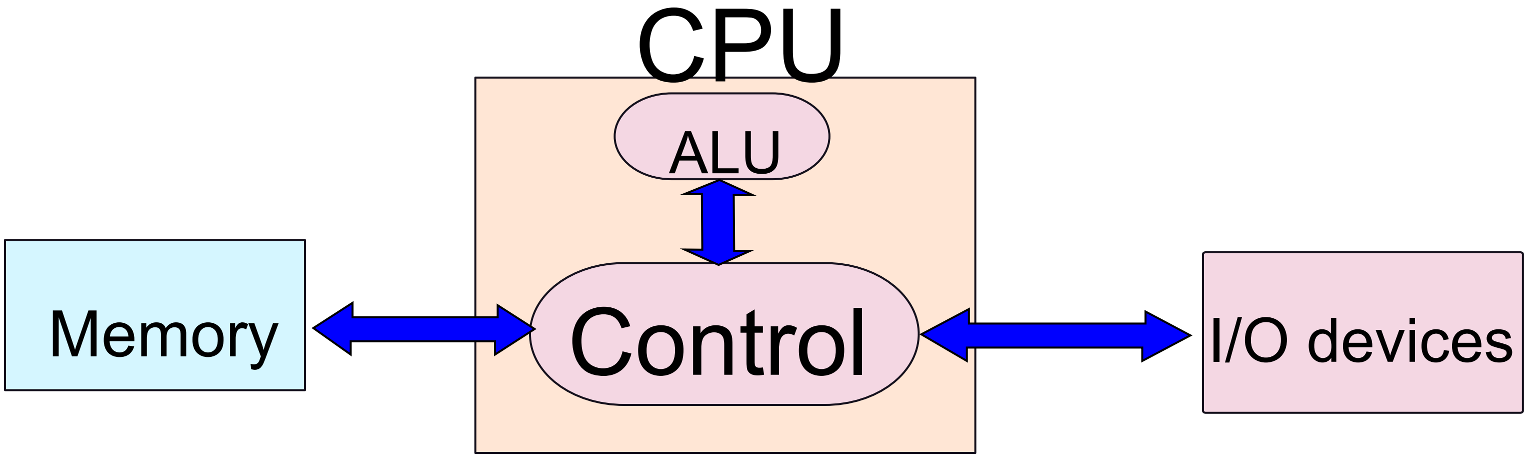
- The given number is at address ‘number’
- If num is positive, write ‘0’ to address ‘result’; if negative, write ‘-1’
- Initialization: ‘one’ = 1

```
1: sbn result, result, 2        // result = 0
2: sbn temp, temp, 3            // temp = 0
3: sbn temp, number, 5          // if positive, then branch
4: sbn result, one, 5           // if negative, result = -1
5: exit
```

Multiple Instruction ISA

* Arithmetic and Logical Instructions
  * add, subtract, multiply, divide, 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

Von-Neumann Architecture



Problems with Harvard/ Von-Neumann Architectures

* The memory is assumed to be one large array of bytes

General Rule: Larger is a structure, slower it is

* Solution:
  * Have a small array of named locations **(registers)** that can be used by instructions

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
- Finally, data is stored back into their memory locations.

Example of a Program in Machine Language with Registers

```
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

Machine with Registers