CS1100 – Introduction to Programming

Lecture 6

Instructor: Shweta Agrawal (shweta.a@cse.iitm.ac.in)

CS1100 – Introduction to Programming

- Programming : From Turtle to C.
- Data Types in C, representations, range of values for each type, Arithmetic operators, and operator precedence.
- Formatting the Input and the Output with various data types.

$\Big\}$ So far

CS1100 – Introduction to Programming

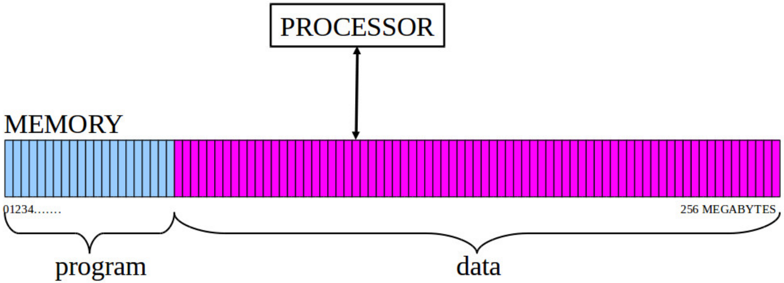
- Programming : From Turtle to C.
- Data Types in C, representations, range of values for each type, Arithmetic operators, and operator precedence.
- Formatting the Input and the Output with various data types.

So far

- Execution of Programs, Compilers.
- Modifying the control flow in Programs.
- `if-then-else`, `switch` statements.
- `loops` in C.

Up Next

The Computing Machine



- A program is a sequence of instructions assembled for some given task.
- Most instructions operate on data.
- Some instructions control the flow of the operations.

The questions ...

```
#include <stdio.h>

/* sum 2 integers */

int main() {
    int x = 98;
    int y = 99;
    int z;

    z = x+y;
    printf("%d\n", z);
    return 0;
}
```

The questions ...

```
#include <stdio.h>

/* sum 2 integers */

int main() {
    int x = 98;
    int y = 99;
    int z;

    z = x+y;
    printf("%d\n", z);
    return 0;
}
```

- How exactly does the computer execute a program?

The questions ...

```c
#include <stdio.h>

/* sum 2 integers */

int main() {
    int x = 98;
    int y = 99;
    int z;

    z = x+y;
    printf("%d\n", z);
    return 0;
}
```

- How exactly does the computer execute a program?
- What happens when you "compile" using "gcc"?

The questions ...

```
#include <stdio.h>

/* sum 2 integers */

int main() {
    int x = 98;
    int y = 99;
    int z;

    z = x+y;
    printf("%d\n", z);
    return 0;
}
```

- How exactly does the computer execute a program?
- What happens when you "compile" using "gcc"?
- While running the program, is this text of C-program stored in memory as it is?

The questions ...

```
#include <stdio.h>

/* sum 2 integers */

int main() {
    int x = 98;
    int y = 99;
    int z;

    z = x+y;
    printf("%d\n", z);
    return 0;
}
```

- How exactly does the computer execute a program?
- What happens when you "compile" using "gcc"?
- While running the program, is this text of C-program stored in memory as it is?
- How exactly does the computer know the type of some data is integer and some data is character etc?

Variables in Programs

- Each memory location is given a name.
- The name is the variable that refers to the data stored in that location. Eg: nsides,rollNo, classSize.
- Variables have types that define the interpretation data. e.g. integers (1, 14, 25649), or characters (a, f, G, H)
- All data is represented as binary strings. That is, it is a sequence of 0's and 1's (bits), of a predetermined size. Recall that a `byte` is made of `8 bits`.

Instructions

- Instructions - operate on data or changes the control flow of the program.

Instructions

- Instructions - operate on data or changes the control flow of the program.
- The instruction "`X` $\leftarrow$ `X+1`" on integer type says: "Take the integer stored in location named `X`, add 1 to it, and store it back in (location named) `X`"..

Instructions

- Instructions - operate on data or changes the control flow of the program.
- The instruction "`X` $\leftarrow$ `X+1`" on integer type says: "Take the integer stored in location named `X`, add 1 to it, and store it back in (location named) `X`"..
- Other instructions tell the processor to do something. For example, "jump" to a particular instruction next, or to exit.

Instructions

- Instructions - operate on data or changes the control flow of the program.
- The instruction "`X` $\leftarrow$ `X+1`" on integer type says: "Take the integer stored in location named `X`, add 1 to it, and store it back in (location named) `X`"..
- Other instructions tell the processor to do something. For example, "jump" to a particular instruction next, or to exit.

Programs and their Execution

Program in memory : sequence of instructions (known to CPU)

Programs and their Execution

Program in memory : sequence of instructions (known to CPU)

The processor(CPU) works as follows,

Programs and their Execution

Program in memory : sequence of instructions (known to CPU)

The processor(CPU) works as follows,
Step A: pick next instruction in the sequence.

Programs and their Execution

Program in memory : sequence of instructions (known to CPU)

The processor(CPU) works as follows,
Step A: pick next instruction in the sequence.
Step B: get data for the instruction to operate upon.

Programs and their Execution

Program in memory : sequence of instructions (known to CPU)

The processor(CPU) works as follows,
Step A: pick next instruction in the sequence.
Step B: get data for the instruction to operate upon.
Step C: execute instruction on data (or “jump”).

Programs and their Execution

Program in memory : sequence of instructions (known to CPU)

The processor(CPU) works as follows,
Step A: pick next instruction in the sequence.
Step B: get data for the instruction to operate upon.
Step C: execute instruction on data (or "jump").
Step D: store results in designated location (variable).

Programs and their Execution

Program in memory : sequence of instructions (known to CPU)

The processor(CPU) works as follows,
Step A: pick next instruction in the sequence.
Step B: get data for the instruction to operate upon.
Step C: execute instruction on data (or "jump").
Step D: store results in designated location (variable).
Step E: go to Step A.

Programs and their Execution

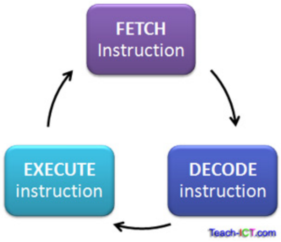
Program in memory : sequence of instructions (known to CPU)

The processor(CPU) works as follows,
Step A: pick next instruction in the sequence.
Step B: get data for the instruction to operate upon.
Step C: execute instruction on data (or “jump”).
Step D: store results in designated location (variable).
Step E: go to Step A.

Programs and their Execution

Program in memory : sequence of instructions (known to CPU)

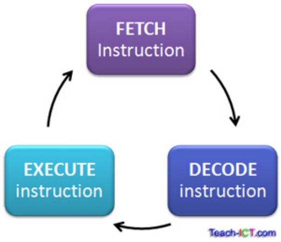
The processor(CPU) works as follows,
Step A: pick next instruction in the sequence.
Step B: get data for the instruction to operate upon.
Step C: execute instruction on data (or “jump”).
Step D: store results in designated location (variable).
Step E: go to Step A.

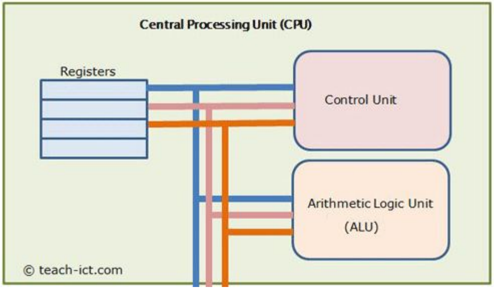


Programs and their Execution

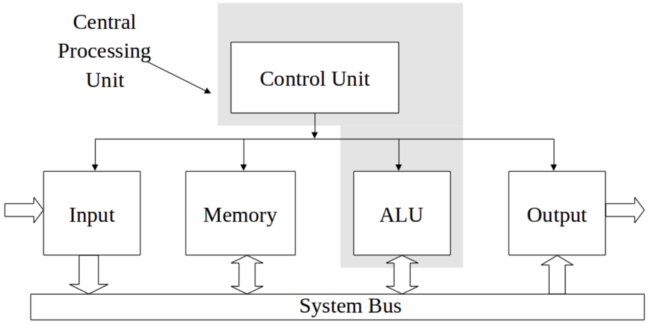
Program in memory : sequence of instructions (known to CPU)

The processor(CPU) works as follows,
Step A: pick next instruction in the sequence.
Step B: get data for the instruction to operate upon.
Step C: execute instruction on data (or "jump").
Step D: store results in designated location (variable).
Step E: go to Step A.





CPU interacts with other parts...



- Control Unit: Directs operation of processor. Tells other parts how to respond to received instructions.
- Arithmetic Logic Unit: Performs arithmetic (addition etc) and logical (OR, AND, etc) operations.

But there is a problem ...

Question : How does the CPU know what is to be done when it executes (say) :

- an assignment statement like "`X` $\leftarrow$ `X+1`"?
- the `printf`
- the `int x`.

Answer :

But there is a problem ...

Question : How does the CPU know what is to be done when it executes (say) :

- an assignment statement like "`X` $\leftarrow$ `X+1`"?
- the `printf`
- the `int x`.

Answer : It does not !!

But there is a problem ...

Question : How does the CPU know what is to be done when it executes (say) :

- an assignment statement like "`X` $\leftarrow$ `X+1`"?
- the `printf`
- the `int x`.

Answer : It does not !!
So what does it know?

But there is a problem ...

Question : How does the CPU know what is to be done when it executes (say) :

- an assignment statement like "`X` $\leftarrow$ `X+1`"?
- the `printf`
- the `int x`.

Answer : It does not !!
So what does it know? Only

- Addition and some basic arithmetic operations.
- Storage and retrieval from memory.
- A very elementary set of instructions - like `ADD`, `MOV`.
- There are specific codes for each of these instructions.

The Machine Language

- Here is an instruction that the machine understands :

  1011 0000 01100001

- It is an instruction that tells the machine MOV A 61h. That is, move, hexadecimal value "61" to the register named "A".
- Who said this is the meaning of this instruction?

The Machine Language

- Here is an instruction that the machine understands :

  1011 0000 01100001

- It is an instruction that tells the machine MOV A 61h. That is, move, hexadecimal value "61" to the register named "A".
- Who said this is the meaning of this instruction? fixed at the processor design stage. (Assembly Language)
- How to use it - combine several instructions like this to make something useful.
- The instruction **forward(100)** can really be represented by a sequence of instructions like that.

From C to Machine language

For example, `x = y + z` could require the following sequence.

- Get the contents of `y` into register R_1.
- Get the contents of `z` into R_2.
- Add contents of R_1 and R_2 and store it in R_1.
- Move contents of R_1 into location named `x`.

From C to Machine language

For example, `x = y + z` could require the following sequence.

- Get the contents of `y` into register R_1.
- Get the contents of `z` into R_2.
- Add contents of R_1 and R_2 and store it in R_1.
- Move contents of R_1 into location named `x`.

Are these written in English?

From C to Machine language

For example, `x = y + z` could require the following sequence.

- Get the contents of `y` into register R_1.
- Get the contents of `z` into R_2.
- Add contents of R_1 and R_2 and store it in R_1.
- Move contents of R_1 into location named `x`.

Are these written in English? No !!
in "machine language" like this :

1011 0000 01100001

From C to Machine language

For example, `x = y + z` could require the following sequence.

- Get the contents of `y` into register R_1.
- Get the contents of `z` into R_2.
- Add contents of R_1 and R_2 and store it in R_1.
- Move contents of R_1 into location named `x`.

Are these written in English? No !!
in "machine language" like this :

1011 0000 01100001

High level languages - Commands are human readable.
Eg : C, C++, Java, Python, FORTRAN, SimpleCPP.

A Demo

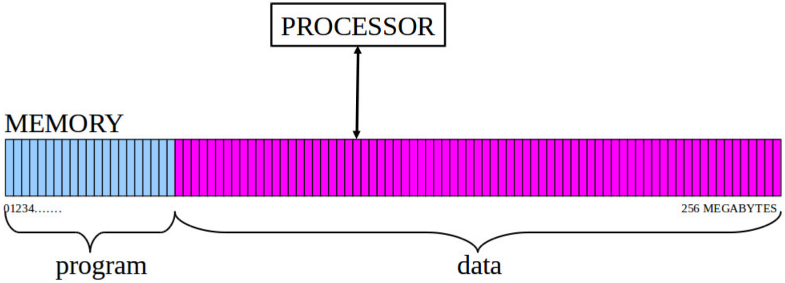
```
#include <stdio.h>

/* sum 2 integers */

int main() {
    int x = 98;
    int y = 99;
    int z;

    z = x+y;
    printf("%d\n", z);
    return 0;
}
```

A Demo

```c
#include <stdio.h>

/* sum 2 integers */

int main() {
    int x = 98;
    int y = 99;
    int z;

    z = x+y;
    printf("%d\n", z);
    return 0;
}
```

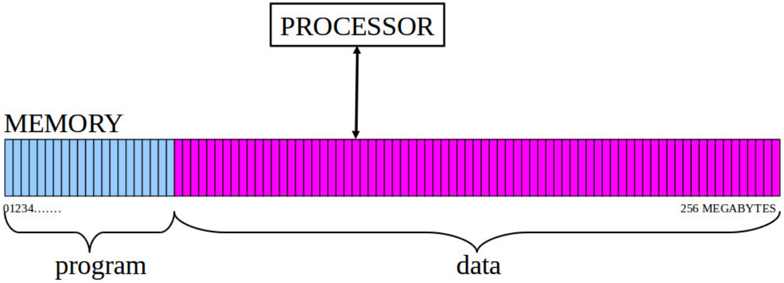
```
01110011 01101111 00101110 00110110 00000000 01110000
01110010 01101001 01101110 01110100 01100110 00000000
01011111 01011111 01101100 01101001 01100010 01100011
01011111 01110011 01110100 01100001 01110010 01110100
01011111 01101101 01100001 01101001 01101110 00000000
01011111 01011111 01100111 01101101 01101111 01101110
01011111 01110011 01110100 01100001 01110010 01110100
01011111 01011111 00000000 01000111 01001100 01001001
01000010 01000011 01011111 00110010 00101110 00110010
00101110 00110101 00000000 00000000 00000000 00000000
01011101 11000110 00000101 00111110 00001011 00100000
00000000 00000001 11110011 11000011 00001111 00011111
01000000 00000000 10111111 00100000 00001110 01100000
00000000 01001000 10000011 00111111 00000000 01110101
00000101 11101011 10010011 00001111 00011111 00000000
10111000 00000000 00000000 00000000 00000000 01001000
10000101 11000000 01110100 11110001 01010101 01001000
10001001 11100101 11111111 11010000 01011101 11101001
01111010 11111111 11111111 11111111 01010101 01001000
```

The Computing Machine



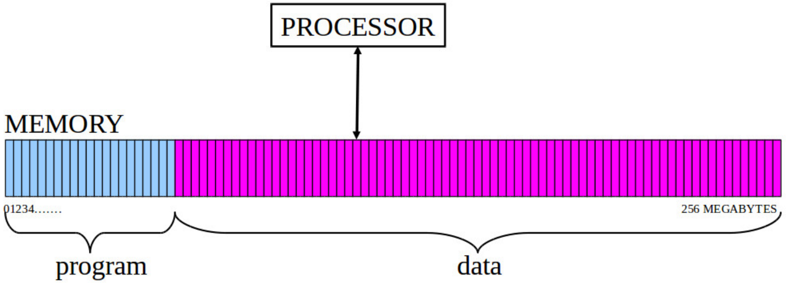
- The instructions are really in binary.

The Computing Machine



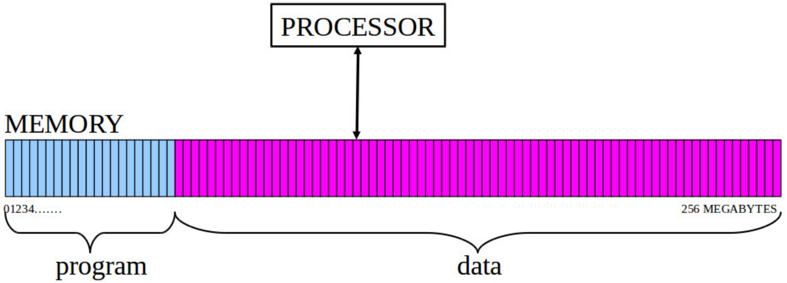
- The instructions are really in binary.
- They are **not** binary equivalents of the corresponding program characters.

The Computing Machine



- The instructions are really in binary.
- They are **not** binary equivalents of the corresponding program characters.
- They are "translations" of program instructions into the "machine language" which uses only very simple instructions.

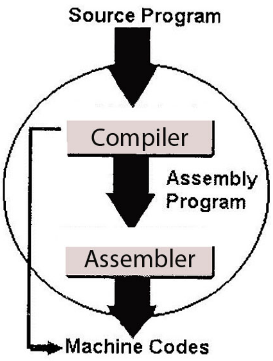
The Computing Machine



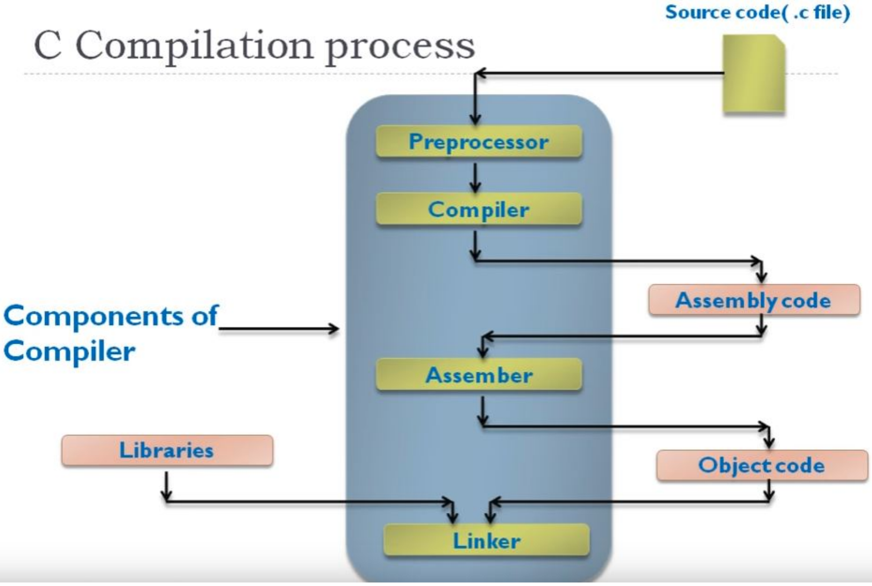
- The instructions are really in binary.
- They are **not** binary equivalents of the corresponding program characters.
- They are “translations” of program instructions into the “machine language” which uses only very simple instructions.
- But who does the translation?

Translators : "The Compiler"

- Source Program can be in C or SimpleCPP or any of the languages.
- A program called C-compiler takes in this program instructions and converts them into assembly language and finally into machine language.
- The final file produced is called the "executable file".



C Compilation process



Assembly versus Machine Language

- Machine language is a language that has a binary form. It can be directly executed by a computer.
- An assembly language is a low-level programming language that requires software called an assembler to convert it into machine code.

Example : Program to Sum Two numbers

program.c (9 lines)

```
#include <stdio.h>

/* sum 2 integers */

int main() {
    int x = 98;
    int y = 99;
    int z;

    z = x+y;
    printf("%d\n", z);
    return 0;
}
```

program.s (assembly language) (36 lines)

```
movq %rsp, %rbp
subq $16, %rsp
movl $98, -12(%rbp)
movl $99, -8(%rbp)
movl -12(%rbp), %edx
movl -8(%rbp), %eax
addl %edx, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
movl %eax, %esi
movl $.LC0, %edi
movl $0, %eax
```

and more ..

Example : Program to Sum Two numbers

a.out (binary) (1435 lines like this....)

```
01110011 01101111 00101110 00110110 00000000 01110000
01110010 01101001 01101110 01110100 01100110 00000000
01011111 01011111 01101100 01101001 01100010 01100011
01011111 01110011 01110100 01100001 01110010 01110100
01011111 01101101 01100001 01101001 01101110 00000000
01011111 01011111 01100111 01101101 01101111 01101110
01011111 01110011 01110100 01100001 01110010 01110100
01011111 01011111 00000000 01000111 01001100 01001001
01000010 01000011 01011111 00110010 00101110 00110010
00101110 00110101 00000000 00000000 00000000 00000000
01011101 11000110 00000101 00111110 00001011 00100000
00000000 00000001 11110011 11000011 00001111 00011111
01000000 00000000 10111111 00100000 00001110 01100000
00000000 01001000 10000011 00111111 00000000 01110101
00000101 11101011 10010011 00001111 00011111 00000000
10111000 00000000 00000000 00000000 00000000 01001000
10000101 11000000 01110100 11110001 01010101 01001000
10001001 11100101 11111111 11010000 01011101 11101001
01111010 11111111 11111111 11111111 01010101 01001000
```

Learnings so far..

- What are the steps involved in running a program?

Learnings so far..

- What are the steps involved in running a program?
- The role of a compiler and different parts of a compiler.

Learnings so far..

- What are the steps involved in running a program?
- The role of a compiler and different parts of a compiler.
- Assembly language, Machine language.

Learnings so far..

- What are the steps involved in running a program?
- The role of a compiler and different parts of a compiler.
- Assembly language, Machine language.
- What is coming up?

Learnings so far..

- What are the steps involved in running a program?
- The role of a compiler and different parts of a compiler.
- Assembly language, Machine language.
- What is coming up? More Programming !!