ARQCP Course

Arquitetura de Computadores
Licenciatura em Engenharia Informática

2023/24

Paulo Baltarejo Sousa
pbs@isep.ipp.pt

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Disclaimer

Material and Slides

Some of the material/slides are adapted from various:

- Presentations found on the internet;
- Books;
- Web sites;
- ...

Outline

Definitions and Concepts

RISC *vs.* CISC

- RISC (**Reduced Instruction Set Computer**) and CISC (**Complex Instruction Set Computer**) refer to the processor architectures that utilize different data processing instruction sets to perform basic logical and input/output operations.
- CISC
  - This architecture was introduced in the 1970s by Intel Corporation when the earliest computers focused on enhancing CPU speed by **minimizing the number of instructions** per program.
    - It can perform **multiple operations in a single instruction**.
- RISC
  - These were introduced in the 1980s by David Patterson and John Hennessy to overcome the complexities of CISC processors.
    - RISC processors work with more instructions (simple).
    - It performs **a single operation in an instruction**.

CPU Architecture

- An Instruction Set Architecture (ISA) **is part of the abstract model of a computer that defines how the CPU is controlled by the software.**
- The ISA acts as an **interface between the hardware and the software, specifying both what the processor is capable of doing as well as how it gets done.**
- Example ISAs:
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: used in almost all mobile phones, Apple M1
  - RISC V: new open-source ISA
  - MIPS

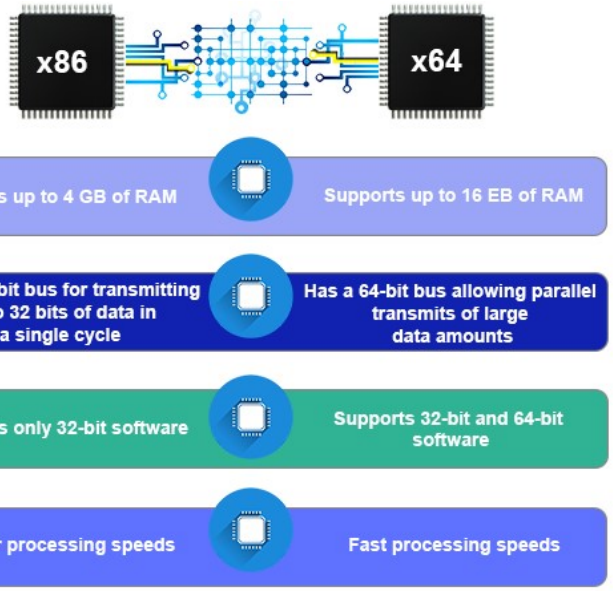
x86 Architecture

- **x86 is a type of ISA** for computer processors originally developed by Intel in 1978.
- The x86 architecture is based on Intel's 8086 (hence the name) microprocessor.
- At first, it was a 16-bit instruction set for 16-bit processors, and later it grew to 32-bit instruction sets.
- The number of bits signifies how much information the CPU can process per cycle.
  - For example, a 32-bit CPU transfers up to 32 bits of data per clock cycle.
- The x86 architecture's most significant limitation is that it can handle a maximum of 4GB of RAM.

x64 Architecture

- **x64 (short for x86-64) is an instruction set architecture based on x86, extended to enable 64-bit code**.
- It was first released in 2000, introducing two modes of operation - the 64-bit mode and the compatibility mode, which allows users to run 16-bit and 32-bit applications as well.
  - Since the entire x86 instruction set remains implemented in the x64 one, the older executables run with practically no performance penalties.
- The x64 architecture supports much greater amounts of virtual and physical memory than the x86 architecture, allowing applications to store large data amounts in memory.
- Additionally, x64 expands the number of general-purpose registers to 16, providing further enhancements and functionality.
- The x64 architecture allows the CPU to process 64-bits of data per clock cycle.

x86 *vs.* x86-64 Architectures

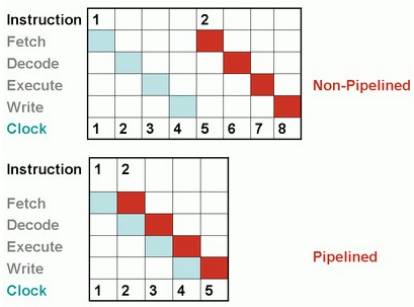


Microarchitecture

- A microarchitecture **is the digital logic that allows an instruction set to be executed**.
- It is the combined **implementation of registers, memory, arithmetic logic units, multiplexers, and any other digital logic blocks**.
  - All of this, together, **forms the processor**.
- A microarchitecture **combined with an instruction set architecture (ISA) makes up the system's computer architecture as a whole.**
- Different microarchitectures **can implement the same ISA, but with trade-offs in things like power efficiency or execution speed.**

CPU Pipeline

- Pipelining is a technique used **to improve the execution throughput of a CPU by using the processor resources in a more efficient manner**.
- The basic idea **is to split the processor instructions into a series of small independent stages**.
- Each stage is designed to perform a certain part of the instruction**.**



- **Fetch** an instruction from memory
- **Decode** the instruction to be executed
- **Execute** the instruction
- **Write** the result back to register or memory

CPU Branch prediction

- Branch prediction is a technique used in CPU design that attempts **to guess the outcome of a conditional operation and prepare for the most likely result**.
- A digital circuit that performs this operation is known as a **branch predictor**.
- How does it work?
  - When a conditional operation such as an `if...else` statement needs to be processed, the branch predictor "speculates" which condition most likely will be met.
  - It then executes the operations required by the most likely result ahead of time.
    - This way, they are already complete if and when the guess was correct.
  - At runtime, if the guess turns out not to be correct, the CPU executes the other branch of operation, incurring a slight delay.
  - But if the guess was correct, speed is significantly increased.

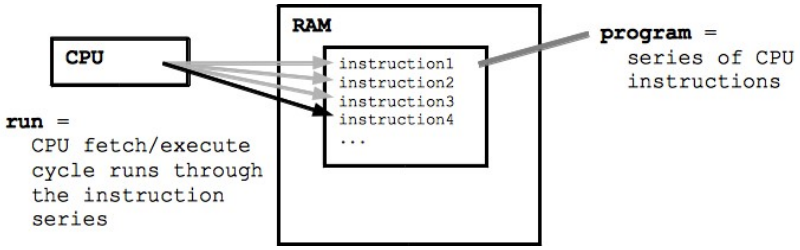
CPU Out-of-order Execution

- A processor that executes the instructions one after the other, may use the resources inefficiently that leads to poor performance of the processor.
- Out-of-order execution can improve the performance of the processor.
- Out-of-order execution can be achieved by executing the instruction in an different from the original order they appear.
- This approach efficiently uses instruction cycles (fetch, decode, execute and write) and reduces costly delay.
- A processor will execute the instructions in an order of availability of data or operands instead of original order of the instructions in the program.
- By doing so the processor will avoid being idle while data is retrieved for the next instruction in a program.

Computer program

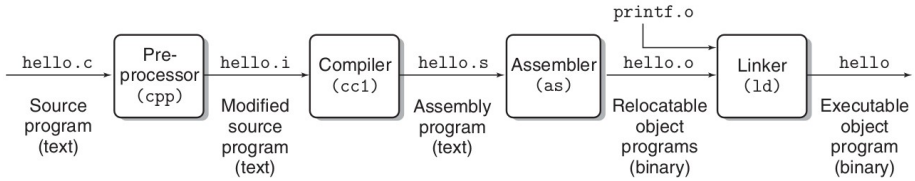
Execution

- A computer **program is a collection of instructions (and data)** that performs a specific task when executed by a computer.
- The **program instructions are loaded into main memory** (RAM).
- The CPU must look into **memory and fetch in the instructions and data and act upon them according to what the instructions are**.



Compilation (I)

- The gcc C `compiler` generates its output in the form of **assembly code**, a textual representation of the machine code giving the individual instructions in the program.
- gcc then invokes both an `assembler` and a `linker` to generate the **executable machine code** from the assembly code.



Compilation (II)

- **Machine language**, or **machine code**, is the most basic set of instructions that a computer can execute.
- **Each type of processor** has its own set of **machine language instructions** (ISA).

C

```c
#include<stdio.h>
int main(){
  printf("Hello world!\n");
  return 0;
}
```

Assembly

```
.section .data
.LC0:
  .string "Hello world!"
.section .text
  .global main
main:
  pushl %ebp
  movl %esp, %ebp
  movl $.LC0, %edi
  call puts
  movl $0, %eax
  popl %ebp
  ret
```

Machine code

```
01111111 01000101 01001100 01000110 00000010 00000001
00000001 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000001 00000000
...
```

Assembly

- Assembly language **is abstraction layer on machine code instructions.**
- An assembly language program is **stored as text file** (just as a higher level language program).
- Each **assembly instruction represents exactly one machine instruction** (according to the ISA).
- The general form of an assembly instruction is:

```
mnemonic operands
```

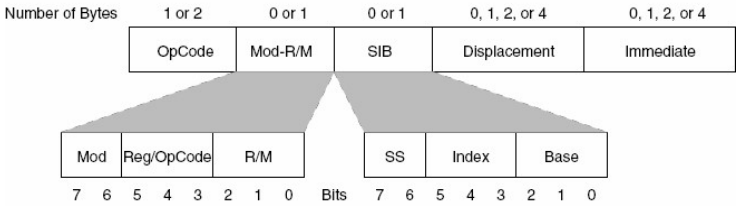
  - For example, an addition operation could be coded in assembly language as:

```
addl %eax, %ebx
```

    - The word `addl` is a **mnemonic** for the addition instruction.
- An assembler **is a program that reads a text file with assembly instructions and converts the assembly into machine code.**

Machine code (I)

- The general format of **machine code instruction**:



  - An **OpCode** (abbreviated from operation code) is the portion of a machine language instruction that specifies the operation to be performed.
  - The **MOD-R/M** byte specifies instruction operands and their addressing mode.
  - ...

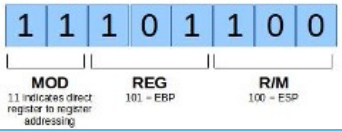
Machine code (II)

Instruction to disassemble: **55 → PUSH EBP**

55 **OPCODE BYTE**

Instruction to disassemble: **8B EC → MOV EBP, ESP**

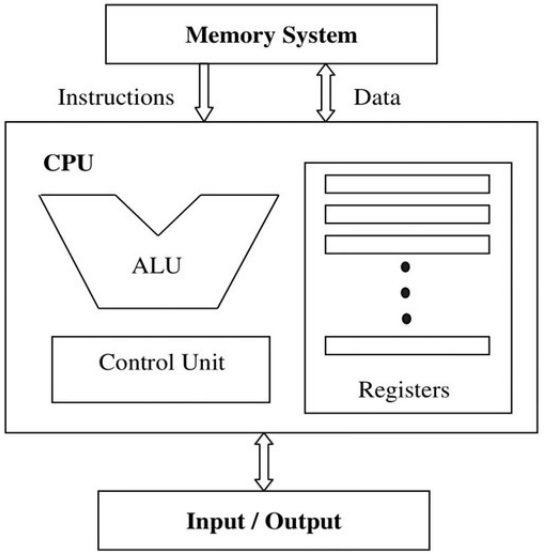
8B **OPCODE BYTE**



EC **MOD-REG-R/M BYTE**

Central Processing Unit (CPU)

CPU

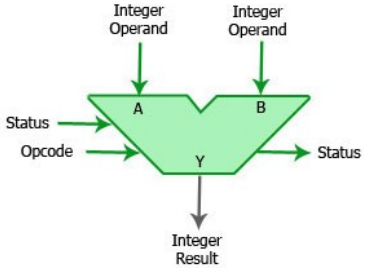

Memory System
Instructions
Data
CPU
ALU
Control Unit
Registers
Input / Output

CPU Control Unit

- The Control Unit is the part of the computer's CPU, which **directs the operation of the processor**.
- It coordinates the flow of data out of, into, and between the various ()sub)units of a processor.
- It **understands commands and instructions**.
- It **regulates the flow of data within the processor**.
- It accepts external commands or instructions, which it turns into a series of control signals.
- It is in charge of a CPU's multiple execution units (such as ALUs, data buffers, and registers).
- It also performs a variety of activities, including fetching, decoding, handling execution, and storing results.

CPU Arithmetic and Logic Unit

- The Arithmetic and Logic Unit is a unit of central processing unit **where all arithmetic and logical operations are carried out.**

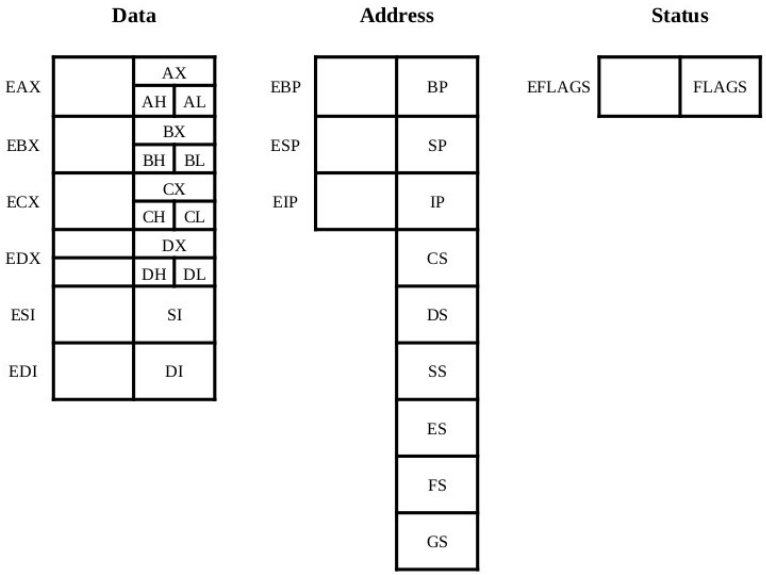


- The basic ALU operates on integer numbers.
  - Two numbers (operands), `A` and `B`, are presented to the input of the ALU, and also an instruction - formally called an **opcode**, such as "add", "subtract", ...
  - The ALU carries out the requested operation on `A` and `B` and produces an output result.
    - Along with the result, a set of status flags are set which include 'Zero', 'Carry', 'Negative', 'Overflow'.

CPU x86 Registers (I)

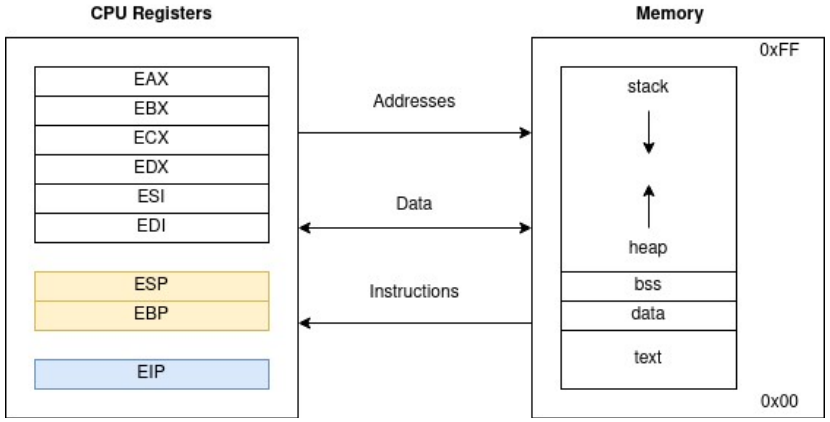
- The **x86 architecture contains fourteen registers**.
- **Data Registers**: Holds data for operations.
  - Accumulator Register (AX), Base Register (BX), Count Register (CX), and Data Register (DX).
- **Address Register**: Holds Address of instruction.
  - Segment Registers
    - Code Segment (CS), Data Segment (DS), Stack Segment (SS), and Extra Segment (ES, FS, GS)
  - Pointer Registers
    - Stack Pointer (SP), Base Pointer (BP), and Instruction Pointer (IP)
  - Index Registers
    - Source Index (SI) and Destination Index (DI)
- **Status Registers**: Store the current status of the processor.
  - Status Flags
  - Control Flags

CPU x86 Registers (II)



CPU and Memory

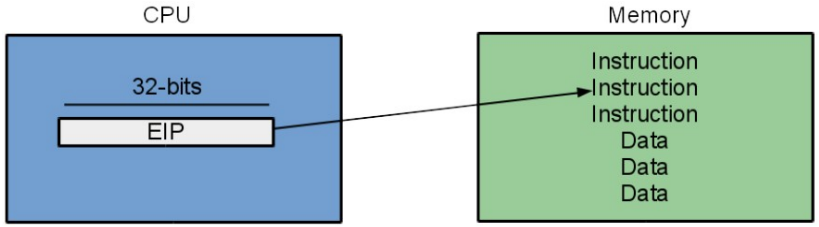
- CPU and Memory **are the core** of computational system.



- Multipurpose Registers (EAX, EBX, ..., EDI)
- Special-purpose Registers (ESP, EBP, and EIP)

Instruction Execution Register

- The **Instruction Pointer** (EIP) register **contains the memory address of the next instruction to be executed**



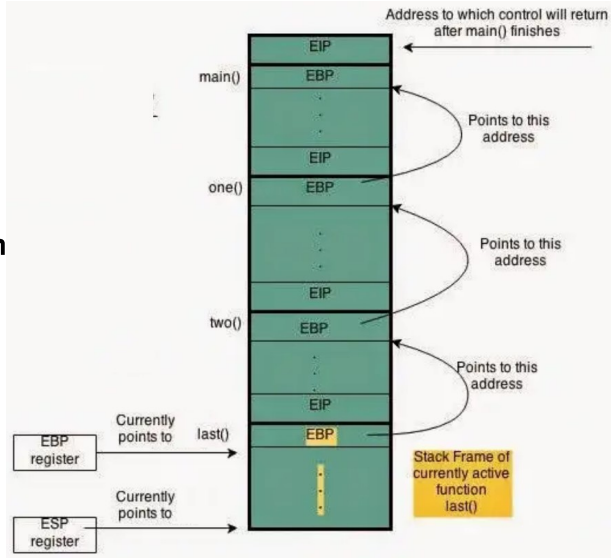
- **After the execution of the instruction**, the EIP register is **automatically increased to the address of next instruction**, unless it ... jumps

```
    movl $1, %eax
    jmp label
    movl $3, %ebx
label:
    int $0x80
```

Stack (I)

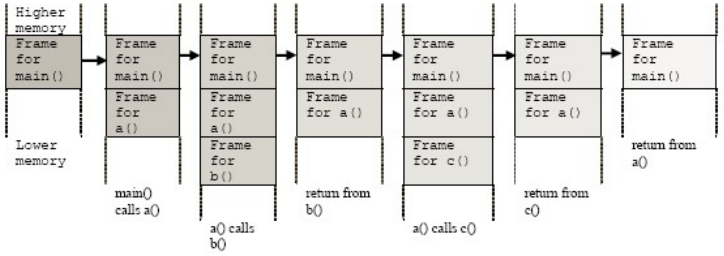
- **Stack Pointer** (ESP) register contains the **address of the top of the stack.**
- **Base Pointer** (EBP) register contains the **address of the bottom of the stack frame**

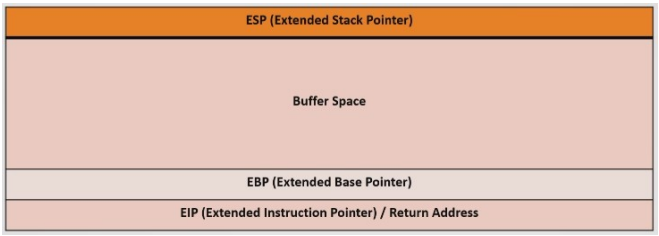
```
last(){
}
two(){
  last();
}
one(){
  two();
}
main(){
  one();
}
```



Stack (II)

```
int c(){
  return 0;
}
int b(){
  return 0;
}
int a(){
  b();
  c();
  return 0;
}
int main(){
  a();
  return 0;
}
```





Stack (III)

```
b:
  pushl %ebp
  movl %esp,%ebp
  movl $0, %eax
  movl %ebp,%esp
  popl %ebp
  ret
a:
  pushl %ebp
  movl %esp,%ebp
  ...
  call b
  ...
  movl %ebp,%esp
  popl %ebp
  ret
```

- The **Return address is the address of the next instruction right after `call`**
- `call <label>`
  - Push **Return address** on stack.
  - Changes the EIP register with the address represented by `label`
- `ret` instruction **pops the address from stack and set EIP register with popped value**.
  - It should be the **Return address** previously pushed by `call` instruction.

Why Assembly?

Why is learning assembly language still important?

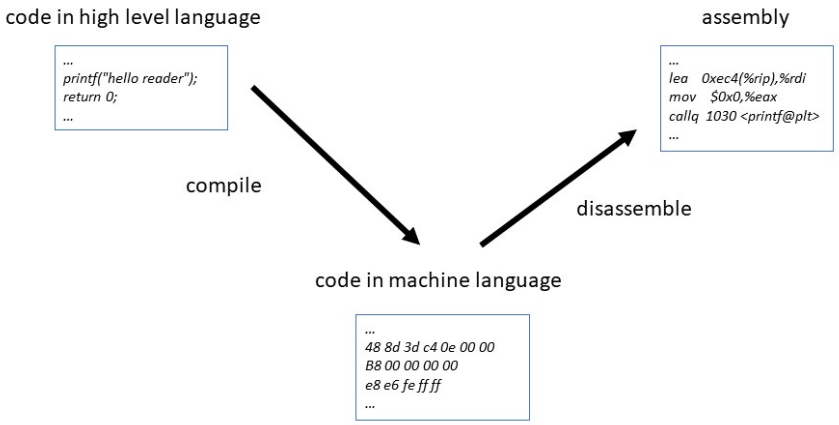
- Assembly machine code **is the foundation of all software**, the fundamental language that computer hardware understands.
  - As programmers and developers, it's essential to grasp how it works.
- The need for programmers **to learn assembly code has shifted over the years** from one of being able to **write programs** directly in assembly to one of being able to read and understand the **code generated by compilers**.
- Understand the **compiler optimizations.**
- Better understanding of architecture issues.
  - Understand how **computer systems perform**.
  - Understanding processor and memory function.
- Improve **algorithm development** skills.
- And so on.

Reverse Engineering (I)

- Reverse engineering is the **inverse process to the normal development of a product**, i.e. the process **begins with a finished product and it is dismantled into its individual parts**.
- Reverse engineering refers **to taking an already compiled code and converting it back into a human-readable format**.
  - In most cases, reverse engineering allows us to understand the program's functionality better and determine how it runs.
- One of the programming languages that are used in reverse engineering is **Assembly Language**

Reverse Engineering (II)

- Disassembler
  - Converting the program **from machine code to assembly code** in order to analyze the program.



Reverse Engineering (III)

- Disassemble a Binary File: `objdump` command
- `> gcc hello.c -o hello`

```c
#include<stdio.h>
int main(){
  printf("Hello world!\n");
  return 0;
}
```

- `> objdump -d hello`

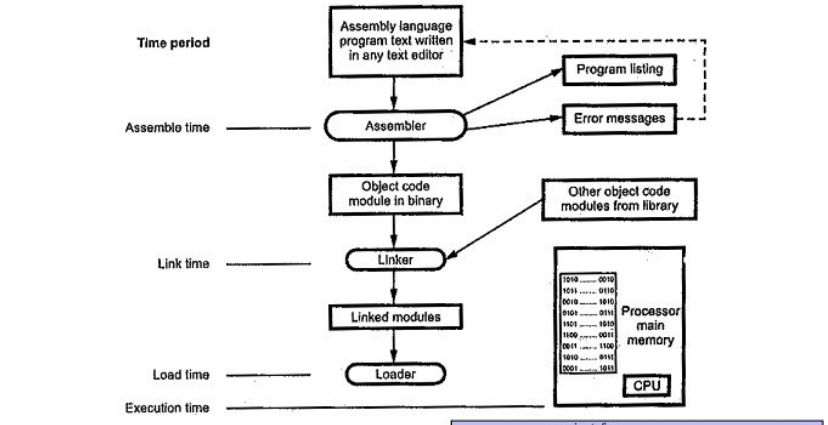
```
...
0000000000001149 <main>:
1149: f3 0f 1e fa  endbr64
114d: 55  push %rbp
114e: 48 89 e5  mov %rsp,%rbp
1151: 48 8d 3d ac 0e 00 00  lea 0xeac(%rip),%rdi # 2004 <_IO_stdin_used+0x4>
1158: e8 f3 fe ff ff  callq 1050 <puts@plt>
115d: b8 00 00 00 00  mov $0x0,%eax
1162: 5d  pop %rbp
1163: c3  retq
1164: 66 2e 0f 1f 84 00 00  nopw %cs:0x0(%rax,%rax,1)
116b: 00 00 00
116e: 66 90  xchg %ax,%ax
...
```

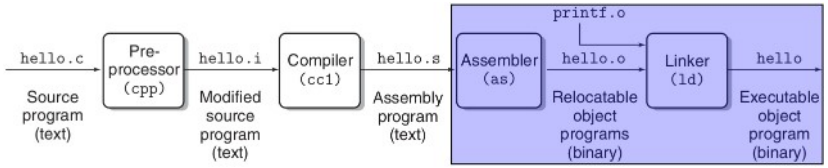
Coding in Assembly

Assembly

- Assembly language is a **low-level programming language that closely corresponds to the machine code instructions of a specific CPU type**.
- It acts as an intermediary layer between high-level languages and machine code, providing a more human-readable representation of the machine code instructions.
- Each assembly language **is designed for exactly one specific computer architecture**.
- Unlike high-level languages, **assembly language is not portable**.
- A program written **in assembly language can only be run on the same type of machine that it was written for**.

Coding & Compiling





Code

- Writing in assembly language involves using **mnemonic codes to represent each low-level machine operation**.
- These mnemonics **are then translated directly into machine code** by an assembler.
- The syntax of assembly language consists of instructions that are made up of an operation code (opcode) followed by a list of arguments.
  - The **opcode**, or mnemonic, is an abbreviated name for an operation (e.g., ADD for addition, MOV for moving data).
  - The arguments are typically **memory addresses**, **constants**, or **CPU registers**.
- Assembly language allows programmers to use symbolic addresses (**labels)**, which means that they don't need to know the exact memory location they are coding for.
  - This makes the code more readable and easier to debug and maintain.
- All of the lines beginning with "`.`" are directives to guide the assembler and linker.

Structure of Assembly Program

- An assembly program can be divided into three sections:
  - The `data` section
    - It is used for declaring initialized data or constants.
    - This data does not change at runtime.
    - It is used for declaring constant values, file names or buffer size etc.
  - The `bss` section
    - It is used for declaring uninitialized variables
  - The `text` section
    - It is used for code (instructions and logic)