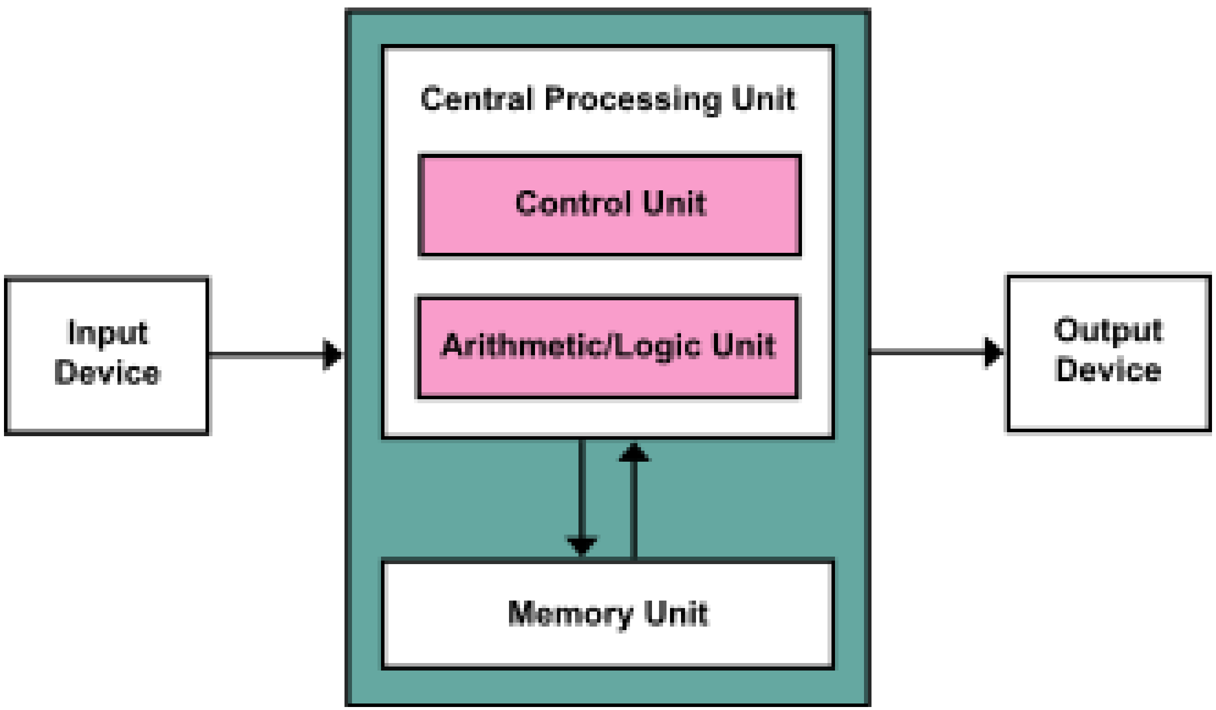
OPERATING SYSTEMS: INTRODUCTION, PROGRAM LOADING, HISTORY

Instructor: William Zheng
Email:
william.zheng@digipen.edu
PHONE EXT: 1745

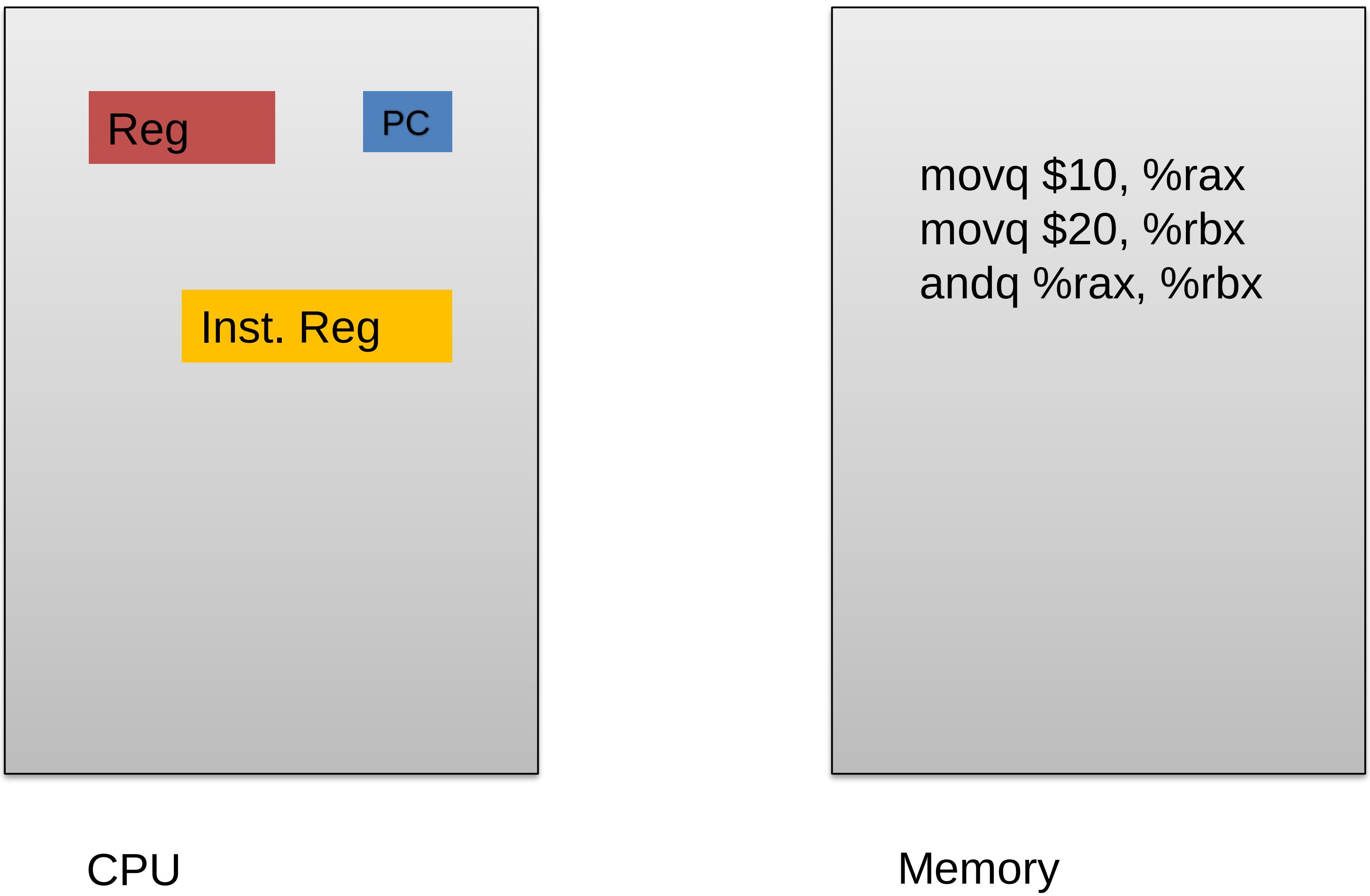
OUTLINE

- Execution of a program
- Boot Sequence
- Roles of an OS
- History of OS

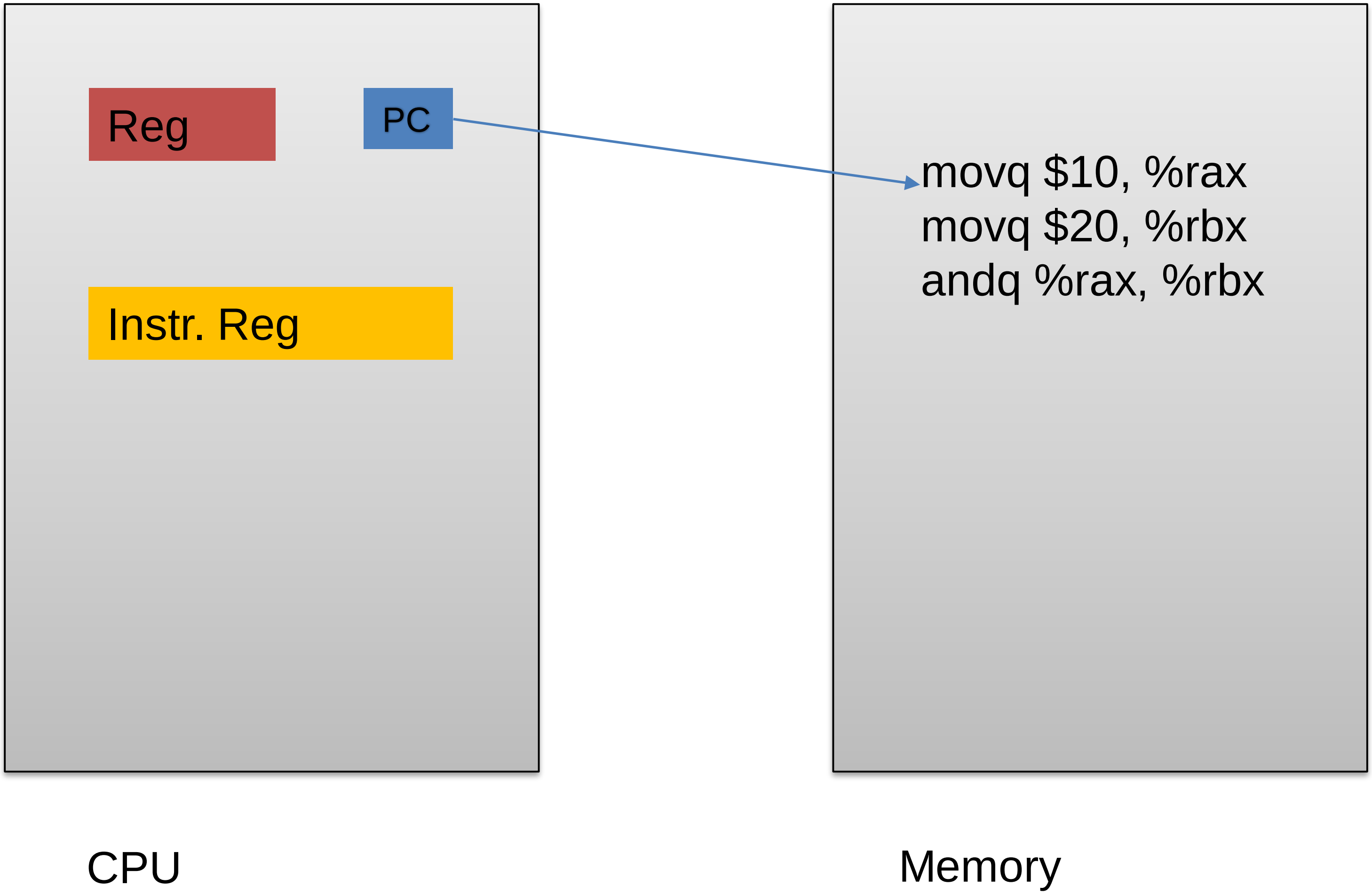
Von Neumann Model



Execution

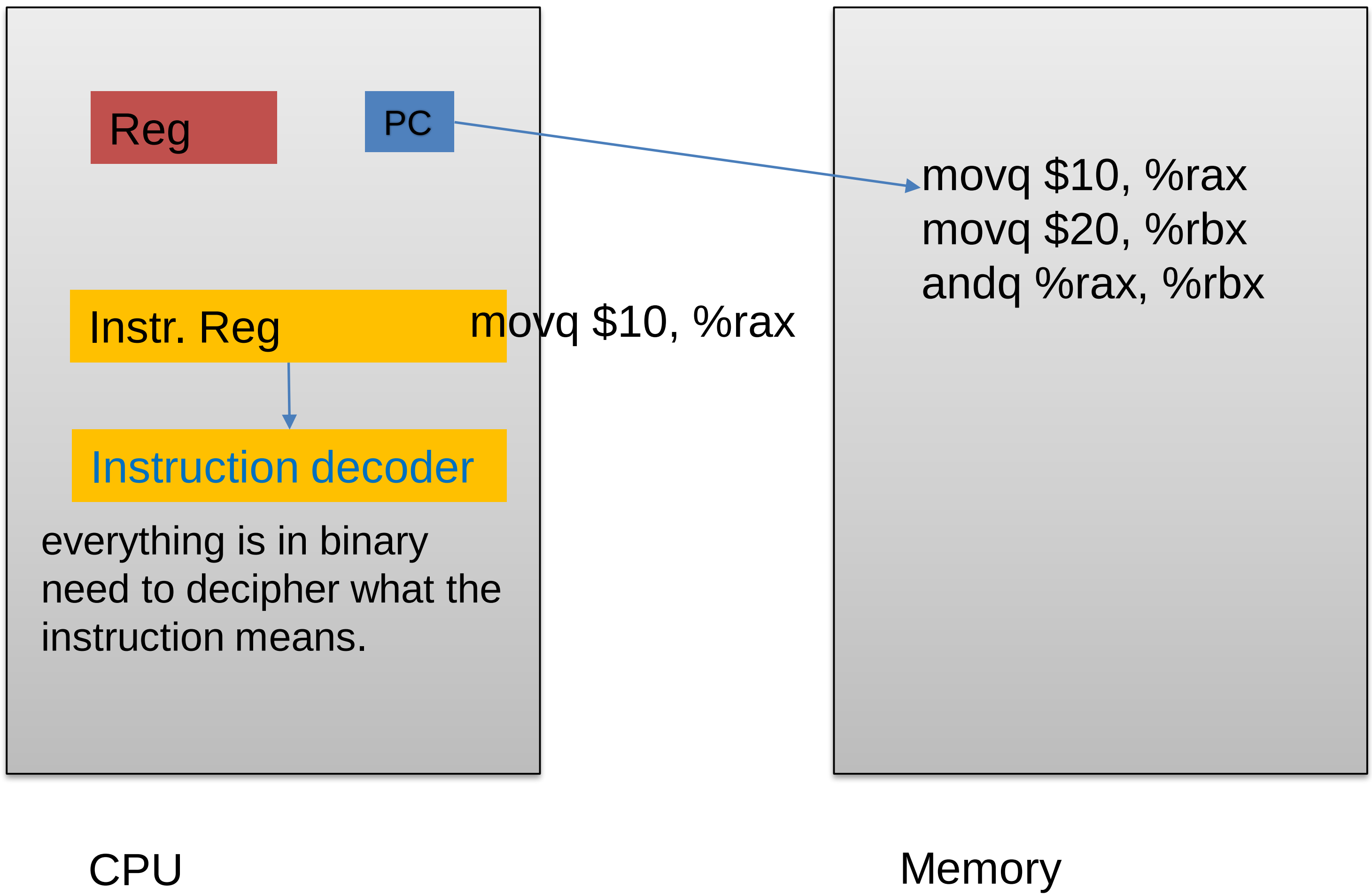


Fetch



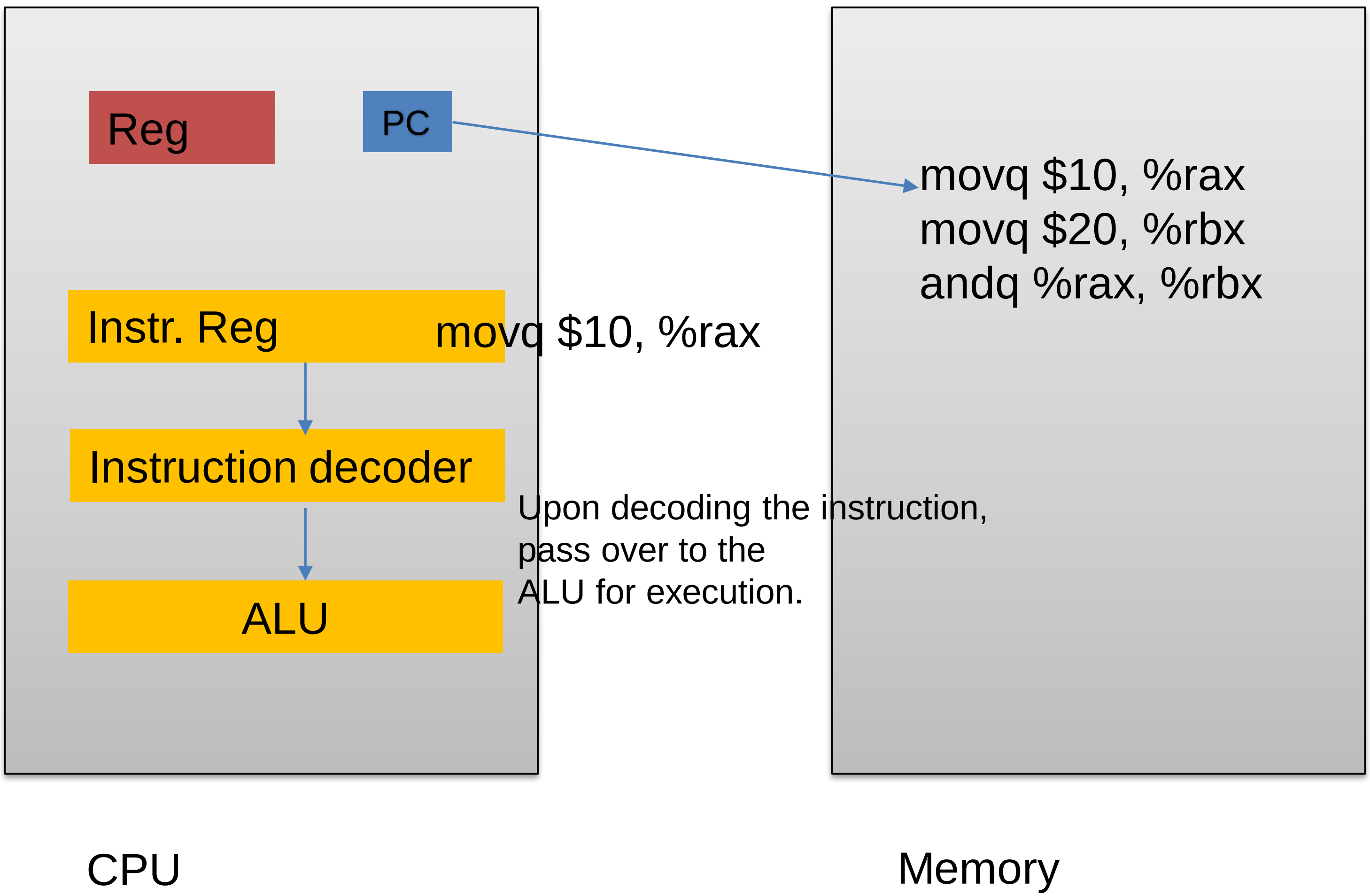
Reg
PC
Instr. Reg
movq $10, %rax
movq $20, %rbx
andq %rax, %rbx
CPU
Memory

Decode

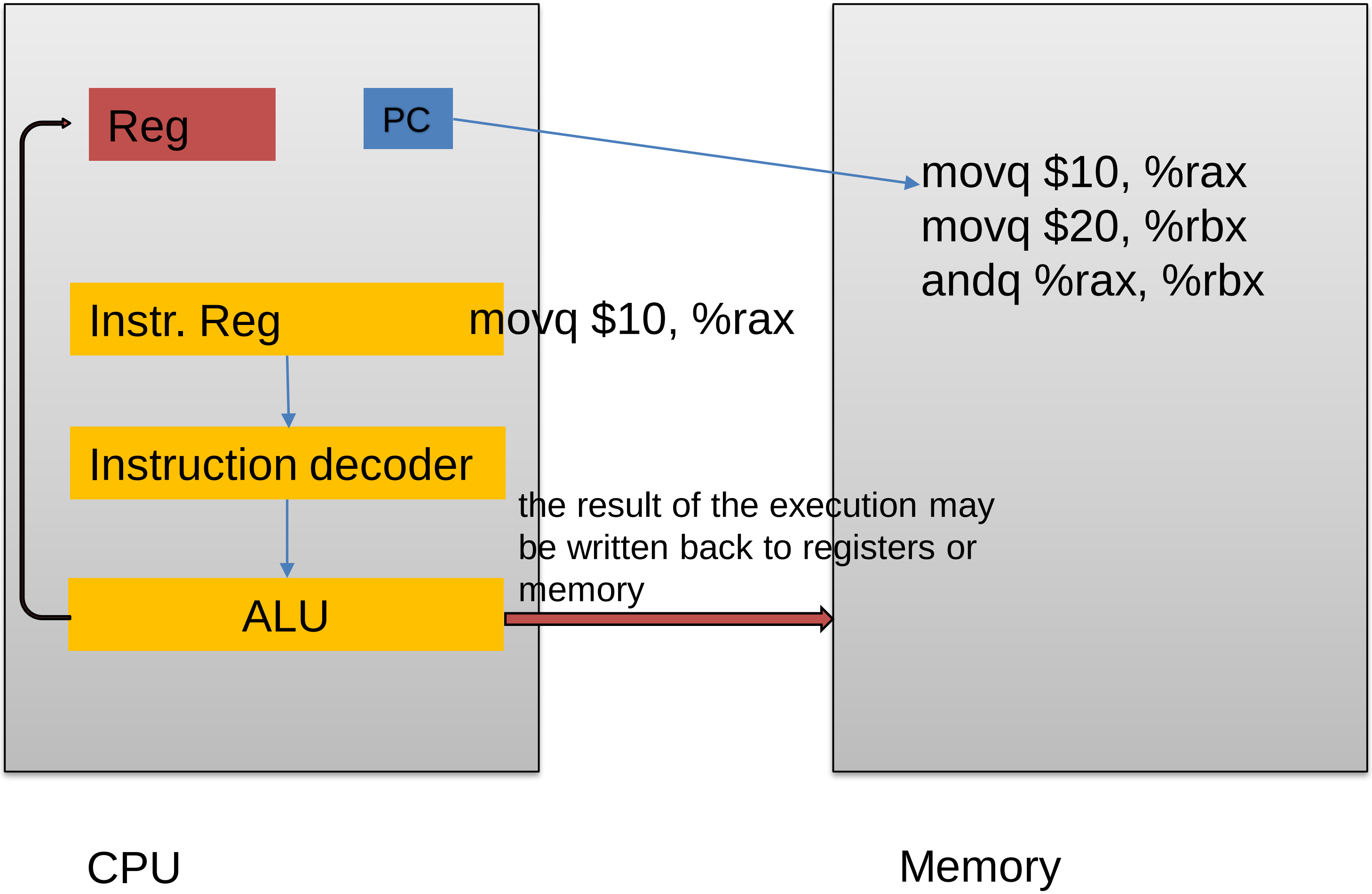


Reg
PC
movq $10, %rax
movq $20, %rbx
andq %rax, %rbx
Instr. Reg
movq $10, %rax
Instruction decoder
everything is in binary
need to decipher what the
instruction means.
CPU
Memory

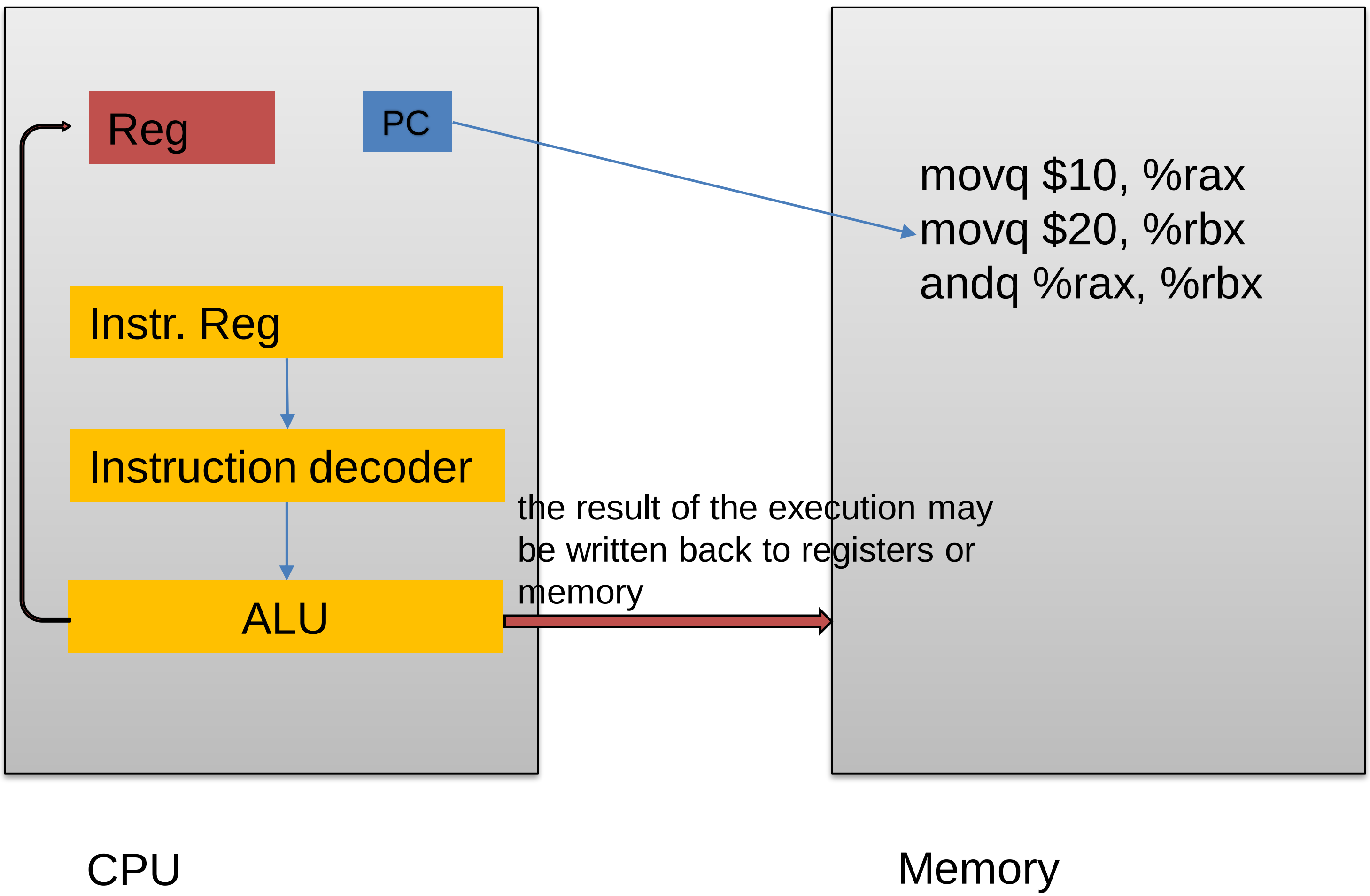
Execute


Reg
PC
movq $10, %rax
movq $20, %rbx
andq %rax, %rbx
Instr. Reg
movq $10, %rax
Instruction decoder
Upon decoding the instruction,
pass over to the
ALU for execution.
ALU
CPU
Memory

Writeback


Reg
PC
movq $10, %rax
movq $20, %rbx
andq %rax, %rbx
Instr. Reg
movq $10, %rax
Instruction decoder
the result of the execution may be written back to registers or memory
ALU
CPU
Memory

Next instruction…Repeat the cycle



Program execution

- So how does a program execute?
  - Naïve answer: load the program into memory and point the PC to the start of the program.
- Outstanding questions:
  - Where to store the program in memory?
  - Which memory address is the 1st instruction of the program?
  - How much memory should I reserve for the running program?
  - How about stack and heap?

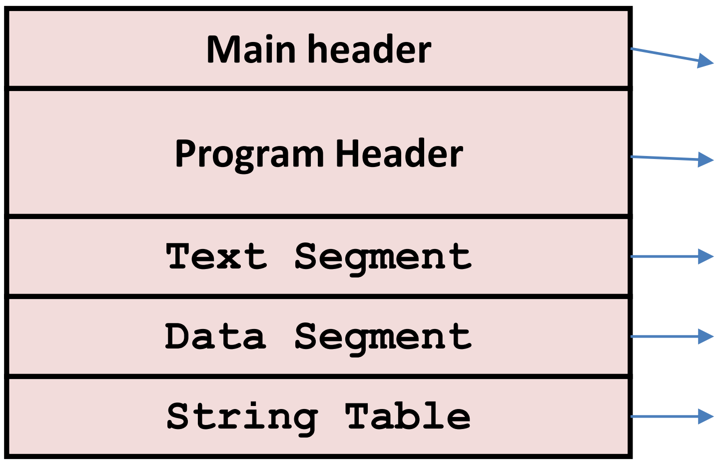
Executable and loading

- Demo (Using WSL)
  - Hexdump to
  - Readelf to check the ELF format header
    - readelf –h a.out
  - Use objdump program to examine an executable
    - `g++ hello-world.cpp –o hello-world`
    - `objdump –D  hello-world (to disassemble program)`
    - `objdump -s -j .data  hello-world (to examine the data section)`
    - `objdump –x hello-world (to see all section headers)`

Loading a Program

- Need a program called a loader
  - Able to read the executable format
  - Copy the text and data segments into the correct memory addresses.
  - Allocate space for Stack and heap for the new running program
  - Set the Program Counter value to the address of the starting instruction of the loaded program.
  - The newly loaded program runs

Layout of an executable



How many sections,
address of start instruction etc

How much memory needed,
these sections need to be in these addresses etc

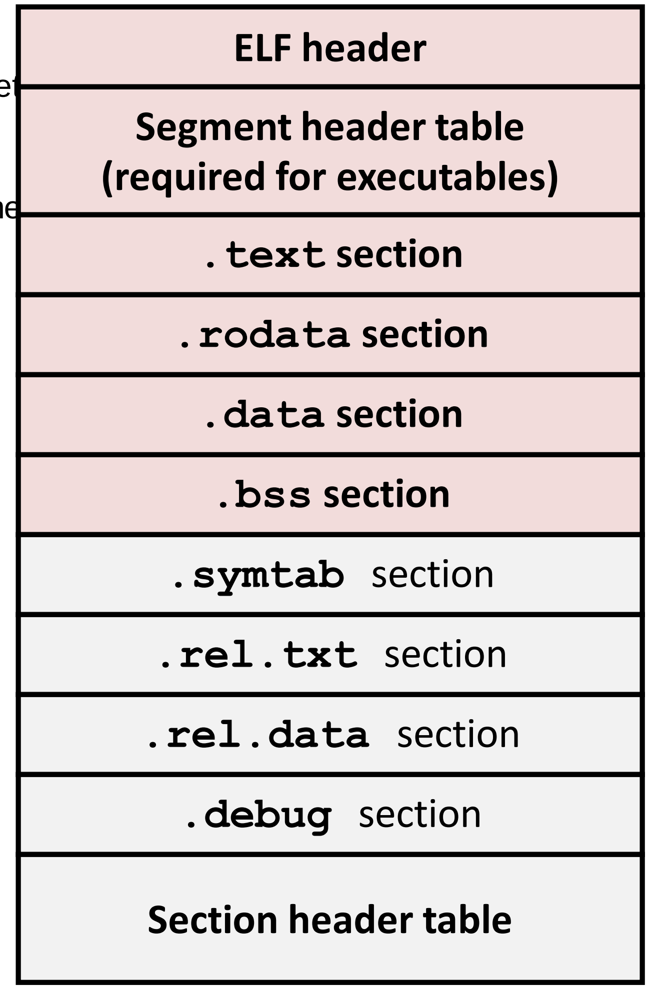
Code

Data used in your program i.e., global variables

Really an extension of data segment. Strings such as “hello world”.

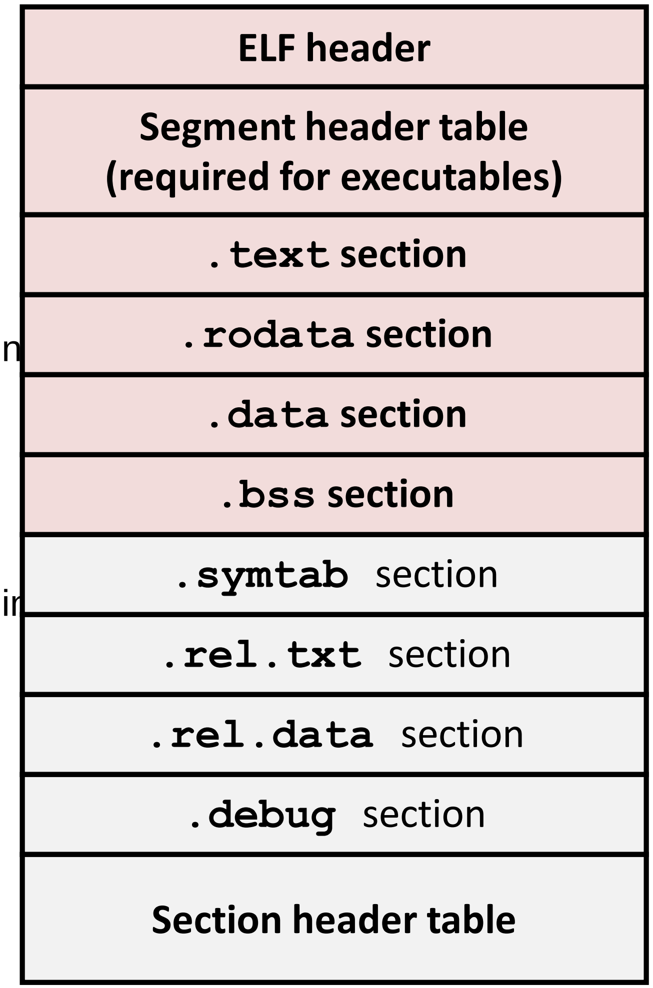
ELF Object File Format

- Elf header
  - Word size, byte ordering, file type (.o, exec, .so), machine type, et
- Segment header table
  - Page size, virtual addresses memory segments (sections), segme sizes.
- `.text` section
  - Code
- `.rodata` section
  - Read only data: jump tables, ...
- `.data` section
  - Initialized global variables
- `.bss` section
  - Uninitialized global variables
  - “Block Started by Symbol”
  - “Better Save Space”
  - Has section header but occupies no space



ELF Object File Format

- `.symtab` section
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- `.rel.text` section
  - Relocation info for **`.text`** section
  - Addresses of instructions that will need to be modified in the executable
  - Instructions for modifying.
- `.rel.data` section
  - Relocation info for **`.data`** section
  - Addresses of pointer data that will need to be modified in the merged executable
- `.debug` section
  - Info for symbolic debugging (**`gcc -g`**)
- Section header table
  - Offsets and sizes of each section



Symbol Resolution

Referencing a global…

…that's defined here

```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

main.c

```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Defining a global

Linker knows nothing of `val`

Referencing a global…

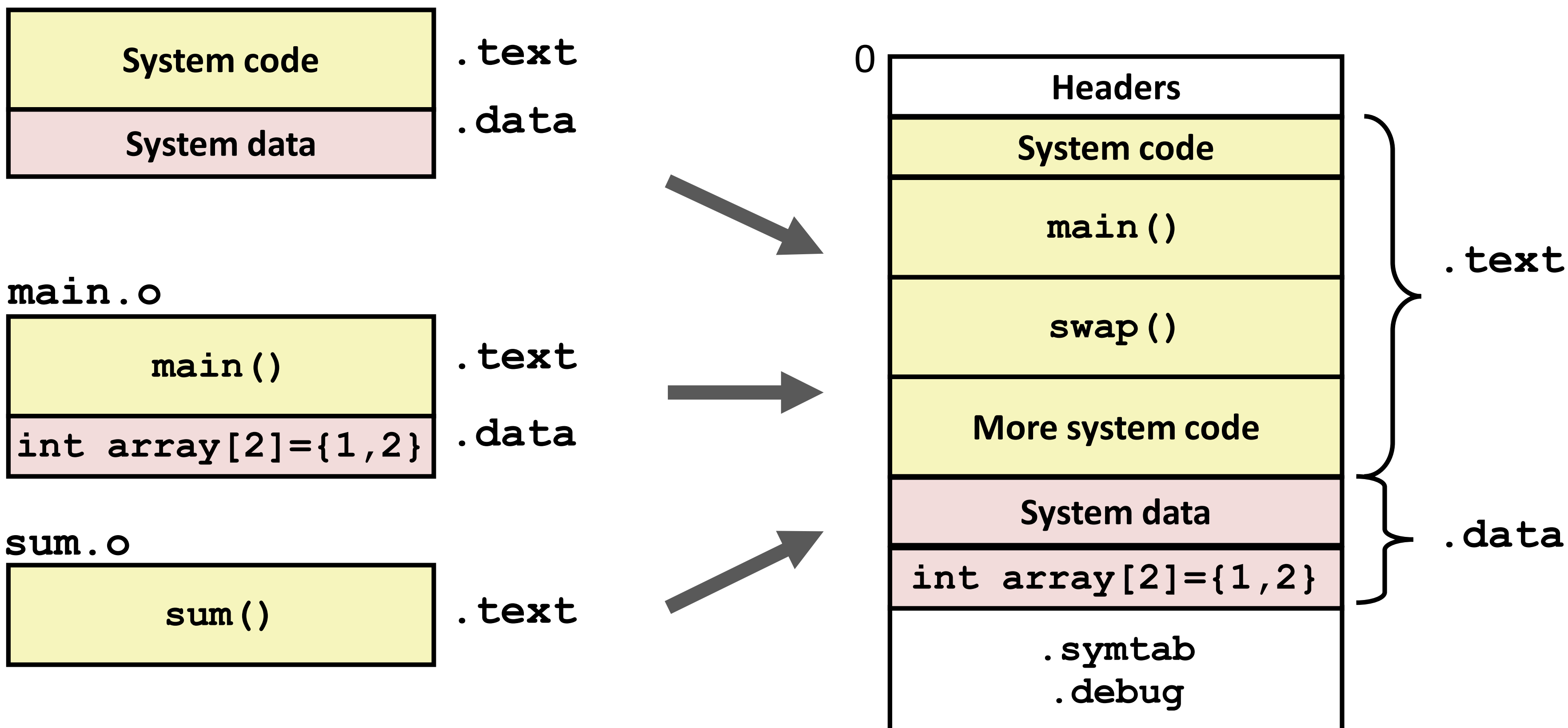
…that's defined here

Linker knows nothing of `i` or `s`

Relocation

Relocatable Object Files

Executable Object File



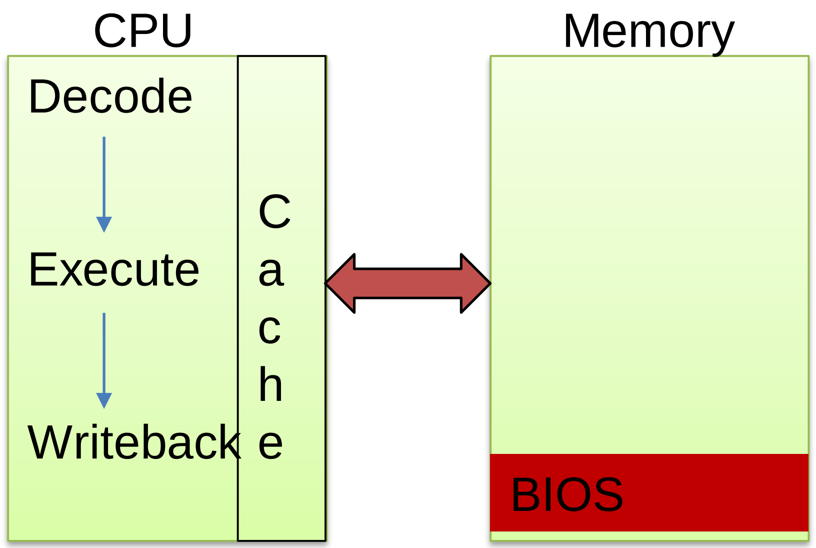
Loader

- Chicken and egg problem
- Who loads the loader?
- Need to talk about the boot sequence

Boot-up process - I

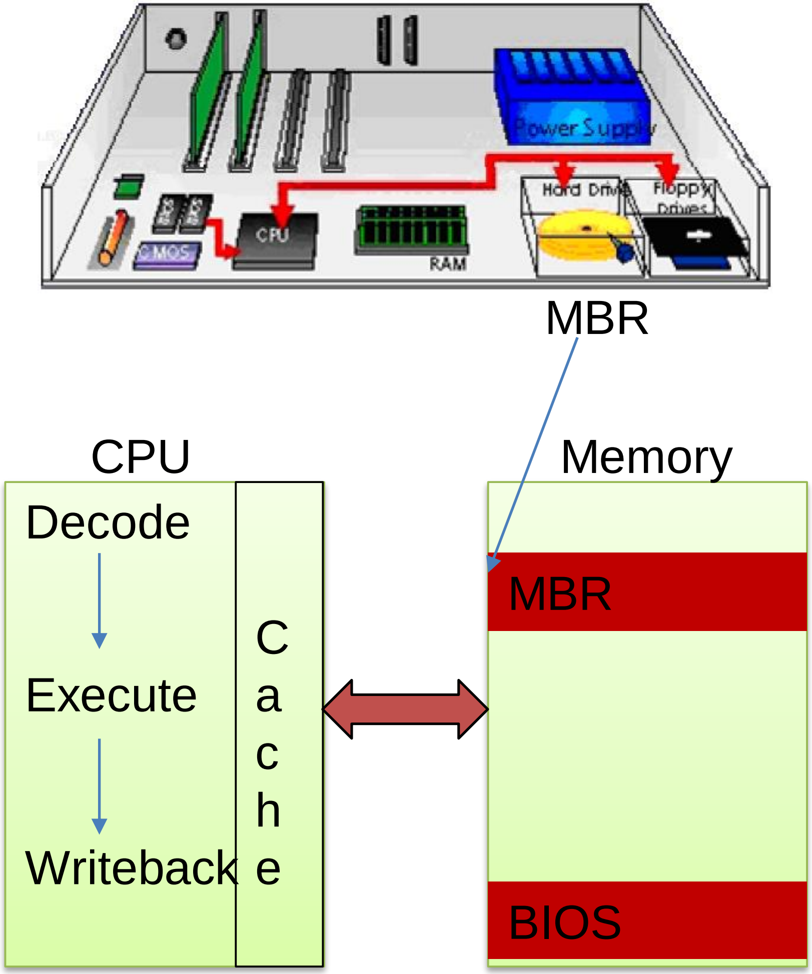
1. Power-On (HW up & running)
2. Run BIOS

- Memory-mapped to FFFFFFF0h
- Performs POST test
- Initialize peripherals etc
- Search through the secondary storages i.e., hard disks for bootable drive



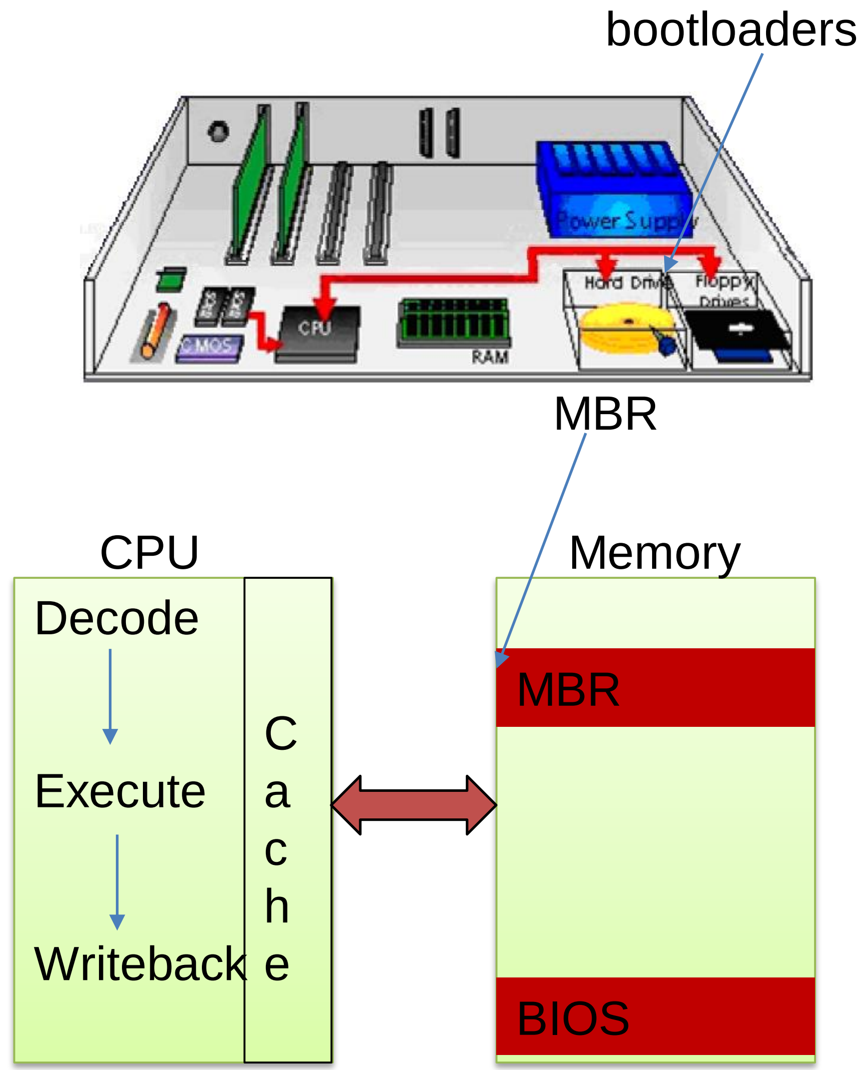
Boot-up process - II

1. Power-On (HW up & running)
2. Run BIOS
3. BIOS load and run
MBR (Master Boot Record)
   - Boot Record
   - 1st Sector Boot Record
   - Small (512 Bytes)

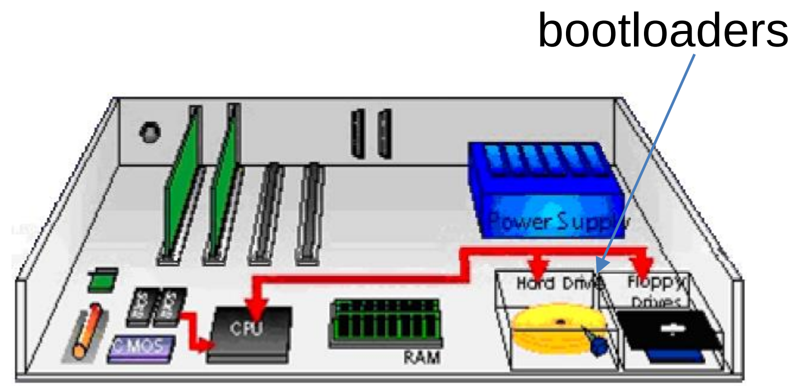


Boot-up process - III

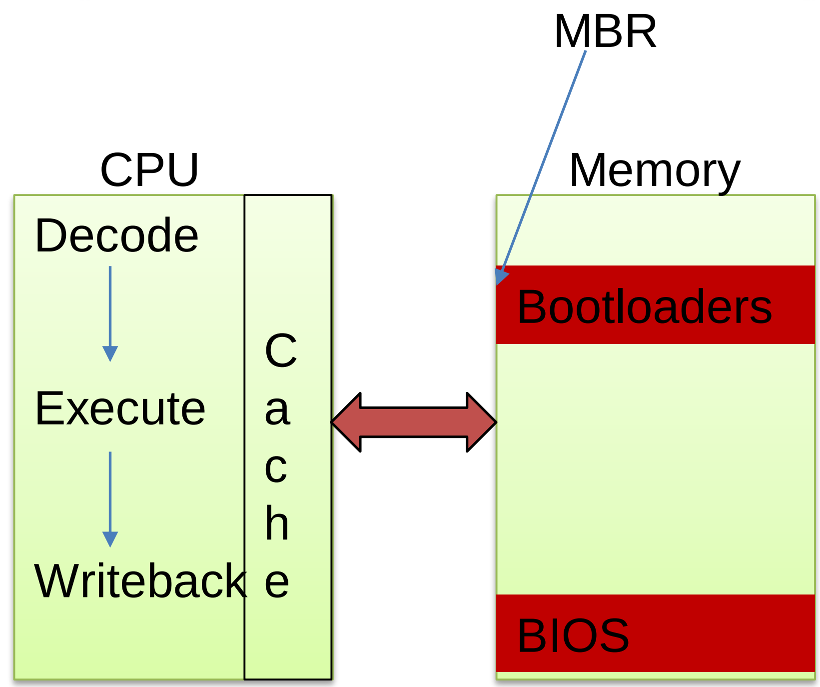
1. Power-On (hardware up & running)
2. Run BIOS
3. BIOS load and run MBR
4. MBR may load boot loaders (chain-loading)



Boot-up process - IV

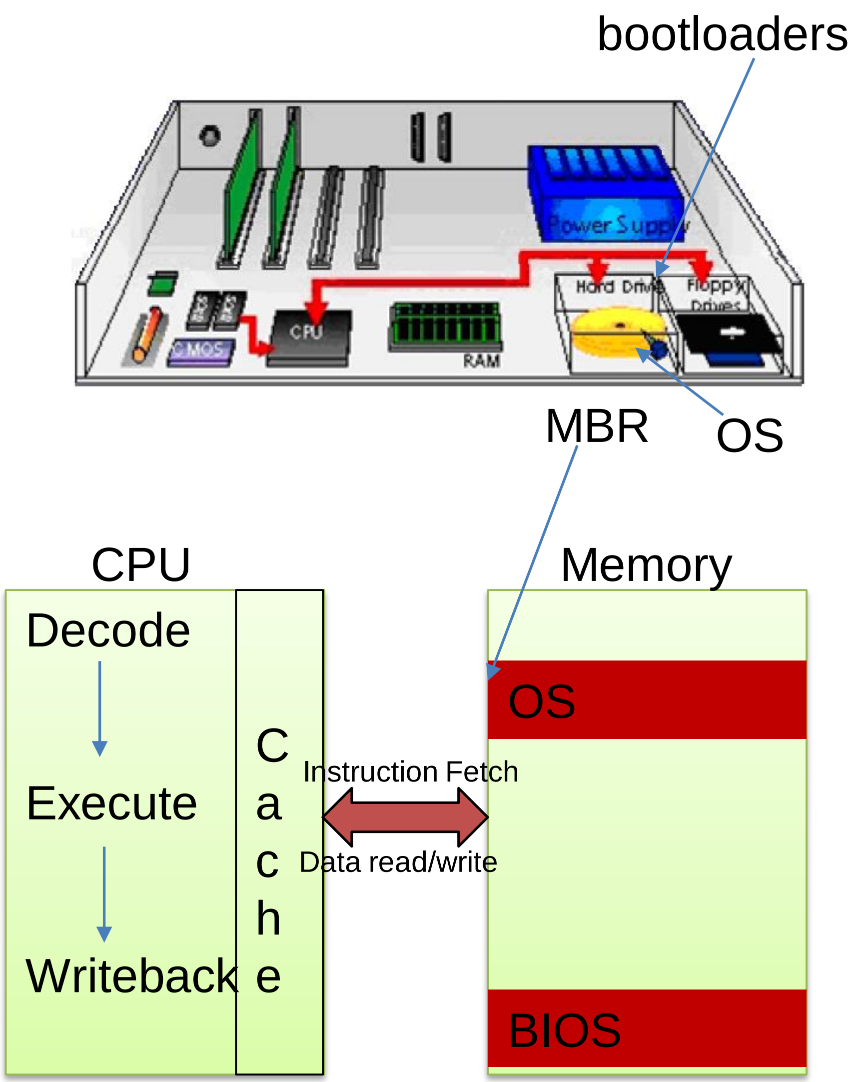


1. Power-On (hardware up & running)
2. Run BIOS
3. BIOS load and run MBR
4. MBR may load boot loaders (chain-loading)



Boot-up process - V

1. Power-On (hardware up & running)
2. Run BIOS
3. BIOS load and run MBR
4. MBR may load boot loaders (chain-loading)
5. Load and run OS



Why Need BIOS?

Initialize and Test HW Components

ensure that the components are attached, functional and accessible to the Operating System (OS)

Load bootloader or OS

BIOS loads the OS directly or loads the bootloader and then passes control to the bootloader

Provide an abstraction layer for I/O devices

BIOS facilitates the interaction btw OS and application by providing an abstraction layer for I/O devices.

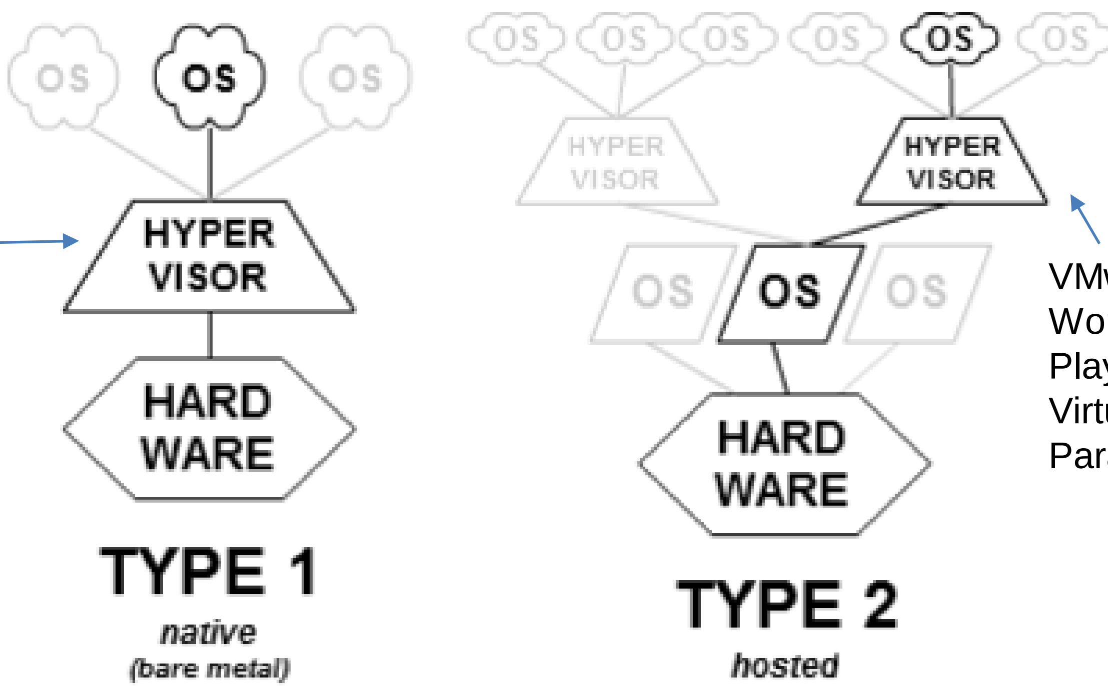
What does an OS do?

- Interface/Abstraction
  - API for programmers
  - Remove need for low-level details
- Portability
- Resource Management
  - Virtualization
- Security and Protection

Virtualization

Hypervisor/Virtual Machine Monitor (VMM)

Xen
Hyper V
VMware ESX



VMware
Workstation/
Player
VirtualBox
Parallels Desktop for Mac

Before there were computers

- “Computers” are more like super-sized calculators

<1950s: Initial computing machines

Cambridge Differential Analyzer

Before there were computers

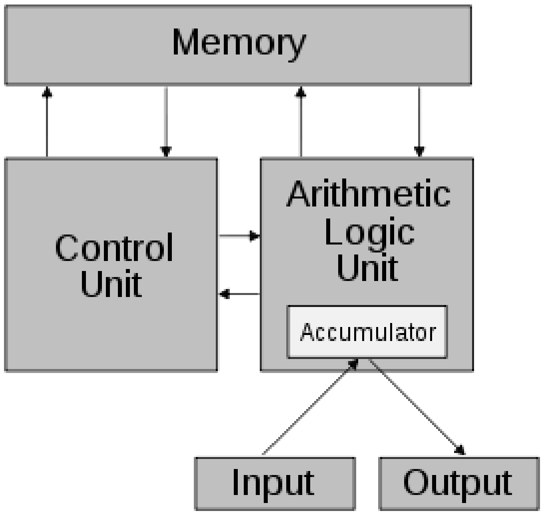
- “Computers” are more like super-sized calculators
- Non-programmable
  - Only 1 function
  - To change the function, a massive re-engineering project

Something's brewing in 1940s…

Von Neumann wrote a paper titled
"First Draft of a report on the EDVAC"

Computers consists of 5 Parts:

1. CA
2. CC
3. M
4. I
5. O



Connected by address bus, data bus and control bus.

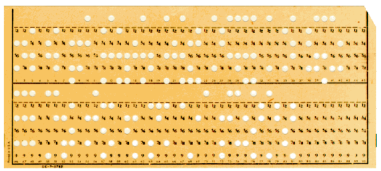
All these proposed in 1945! And the model still fits till now.

Mainframes

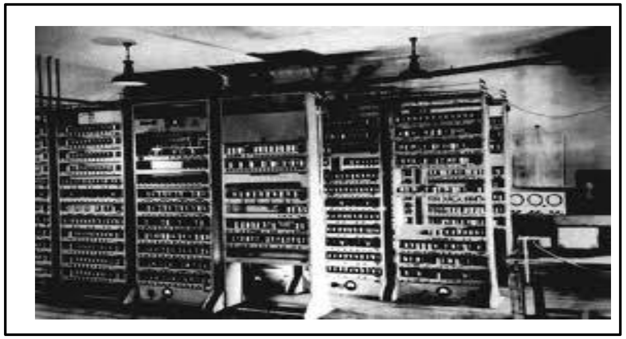
UNIVAC I (1951), 1000 cu. feet, 2000 additions per second

Lifetime of a program – in early 1950s

1. Write the program(in assembly by hand!)

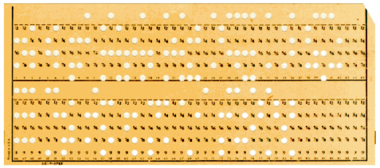


2. Write the program in the form of punch-cards/tapes



3. Load the program manually from tapes/punched cards

4. If error occurs, (how is an error detected?), programmer examine memory and registers directly.



5. Output was in form of punched cards or tapes

Problems

- Assembly Programming
  - Error prone
  - Labor intensive
- Reinventing the wheel
- I/O Peripherals
- Computer expensive, cheap labor
  - Need to keep the computer running as busy as possible.

Partly solved by high-level programming languages. FORTRAN, COBOL

Introduction of library routines

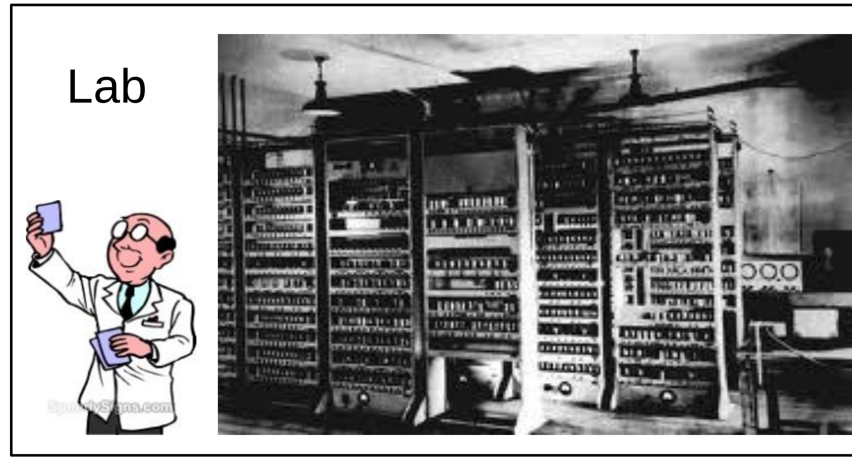
Running a program after FORTRAN…

1. Loading the FORTRAN compiler tape (with FORTRAN program as input)
2. Running the compiler
3. Unloading the compiler tape (assembly code is printed as output)
4. Loading the assembler tape (with assembly code as input)
5. Running the assembler
6. Unloading the assembler tape (object program is printed as output)
7. Loading the object program tape
8. Running the object program

Big Problem:
While all the loading and unloading is being done, the CPU is idle!

Initial Solution

1. Get a computer operator (better and faster at loading/unloading tapes)
2. Batch the same jobs together.



Student A:
Help me run my FORTRAN program

Student B:
Help me run my COBOL program

Student C:
Help me run my FORTRAN program

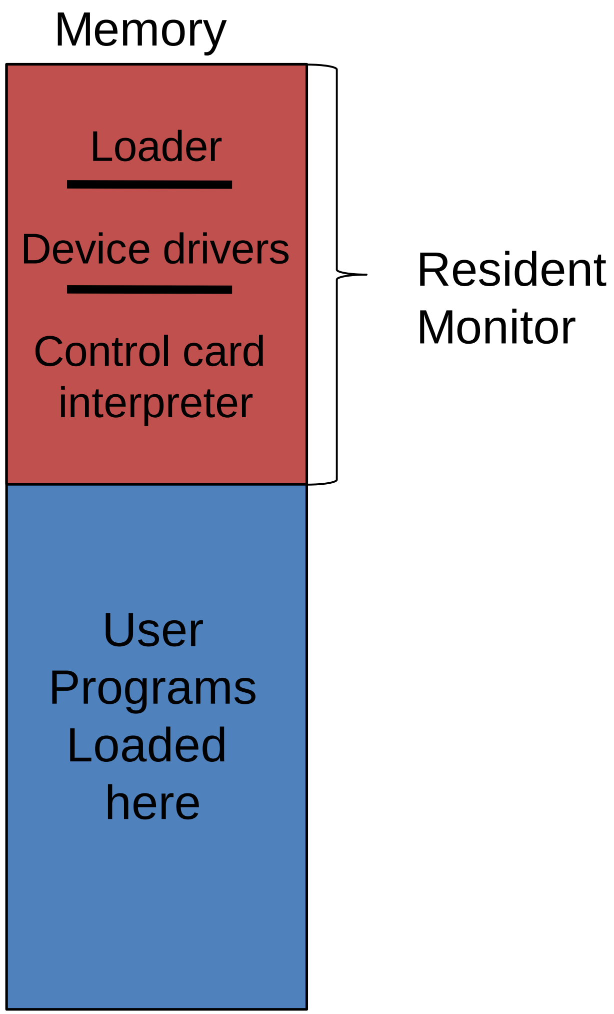
Lab Operator:
Why don’t I load the FORTRAN compiler only once and compile the FORTRAN jobs before I deal with the COBOL job?

Not good enough

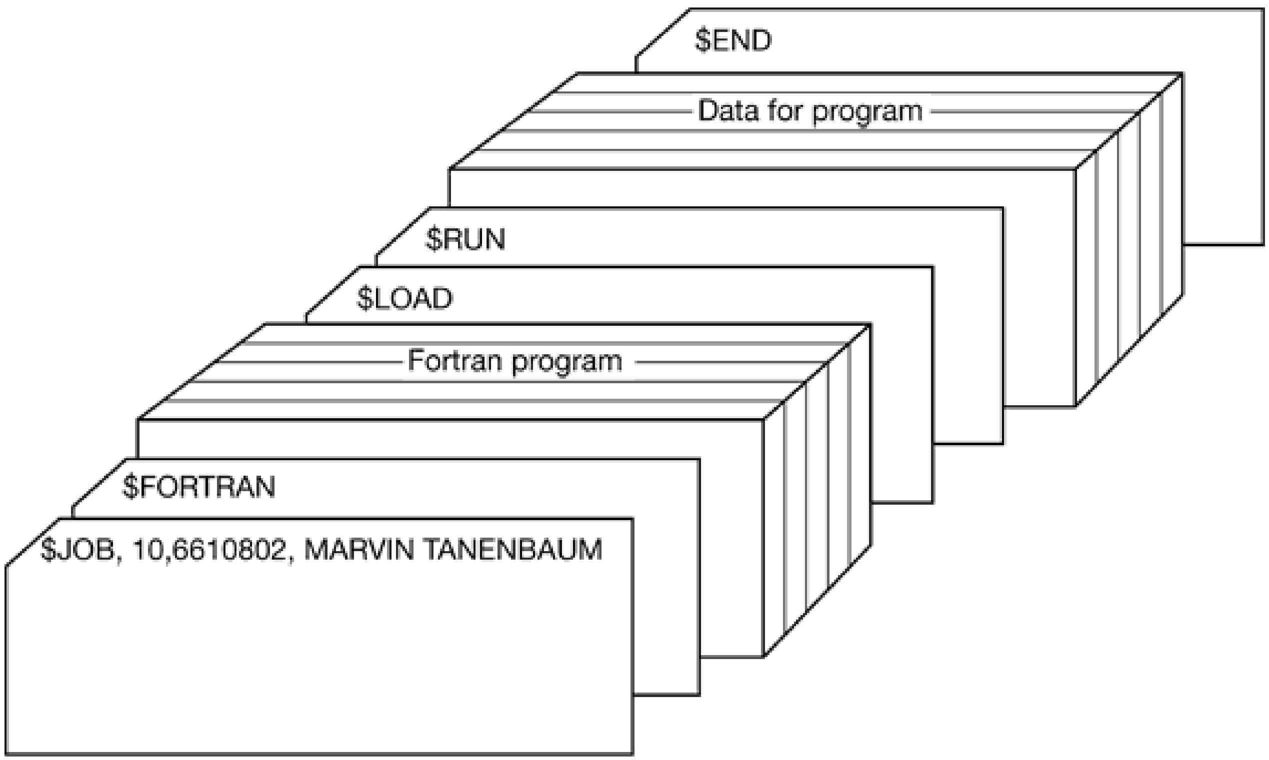
- When a job stops…
  - Who knows? Maybe the lab operator is sleeping or outside the lab.
- In between jobs
  - Loading and unloading is still slow.

Resident Monitor

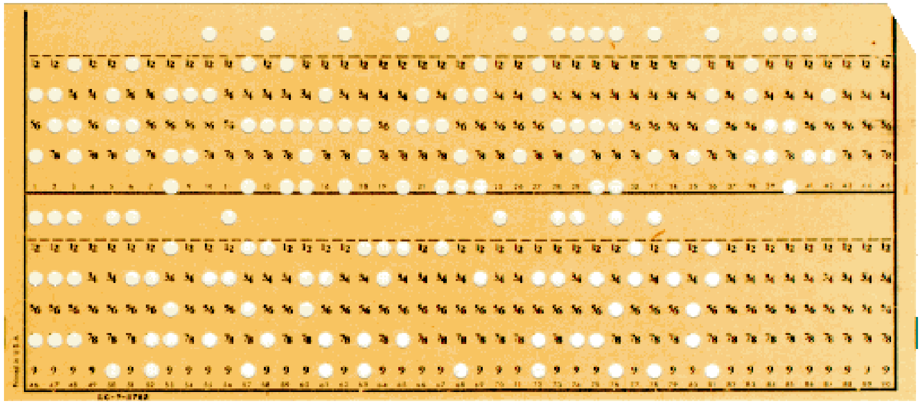
- Idea
  - Computers fast. Humans slow.
  - Automatic job loader in memory.



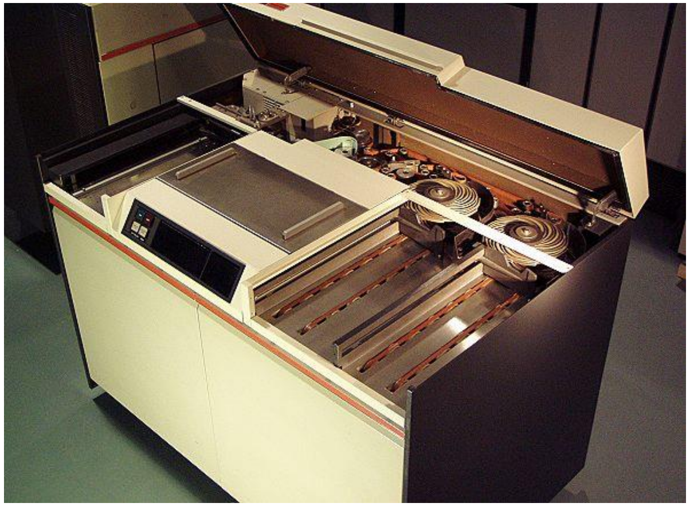
An example batch job cards



Hardware I/O libraries



Punched cards



Card reader 10 cards/s

635kg! printer

So how are we doing in dealing with this problem?

- Computer expensive, cheap labor
  - Need to keep the computer running as busy as possible.
- What kind of bottlenecks do we have so far?

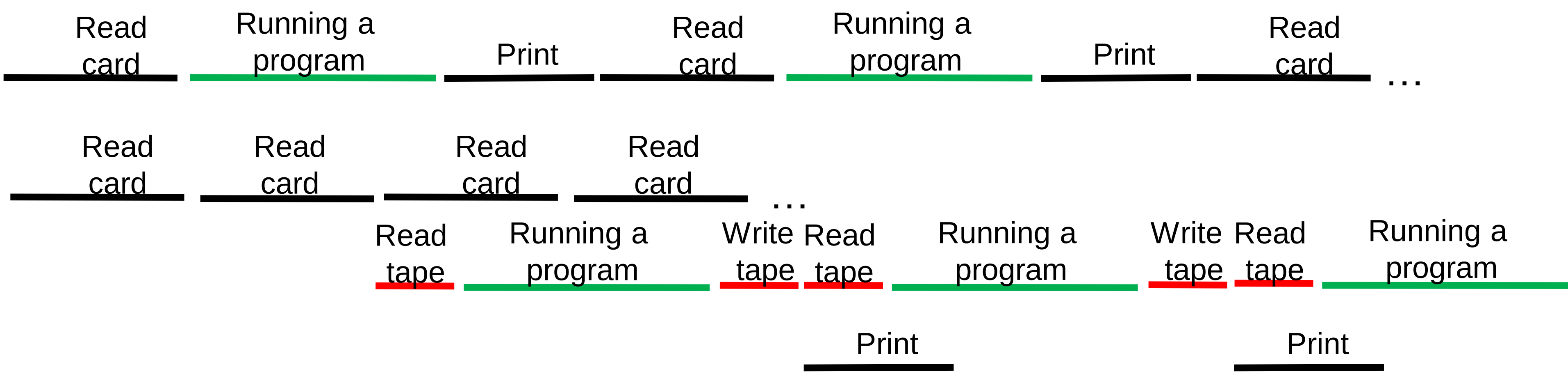
Loading a program

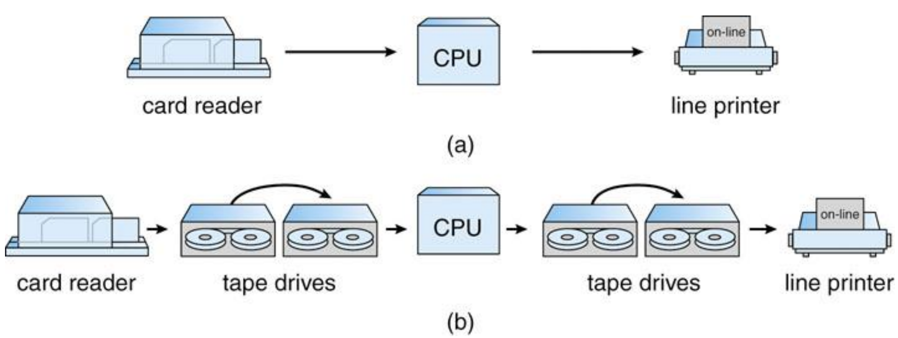
Running a program

Getting the Output

Overlapped I/O

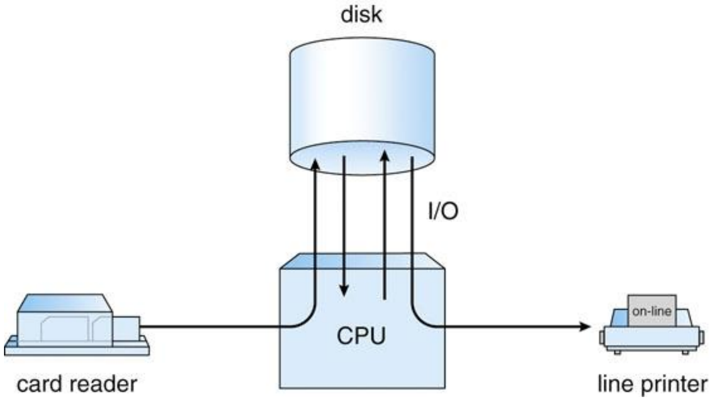
- Card Reader /line printer slower than Tape Drives





Tapes versus disks

- Sequential versus random access
- Fast to read “card” and write “card”
- SPOOL (Simultaneous Peripheral Operation On-Line)
- Leads to multiprogramming



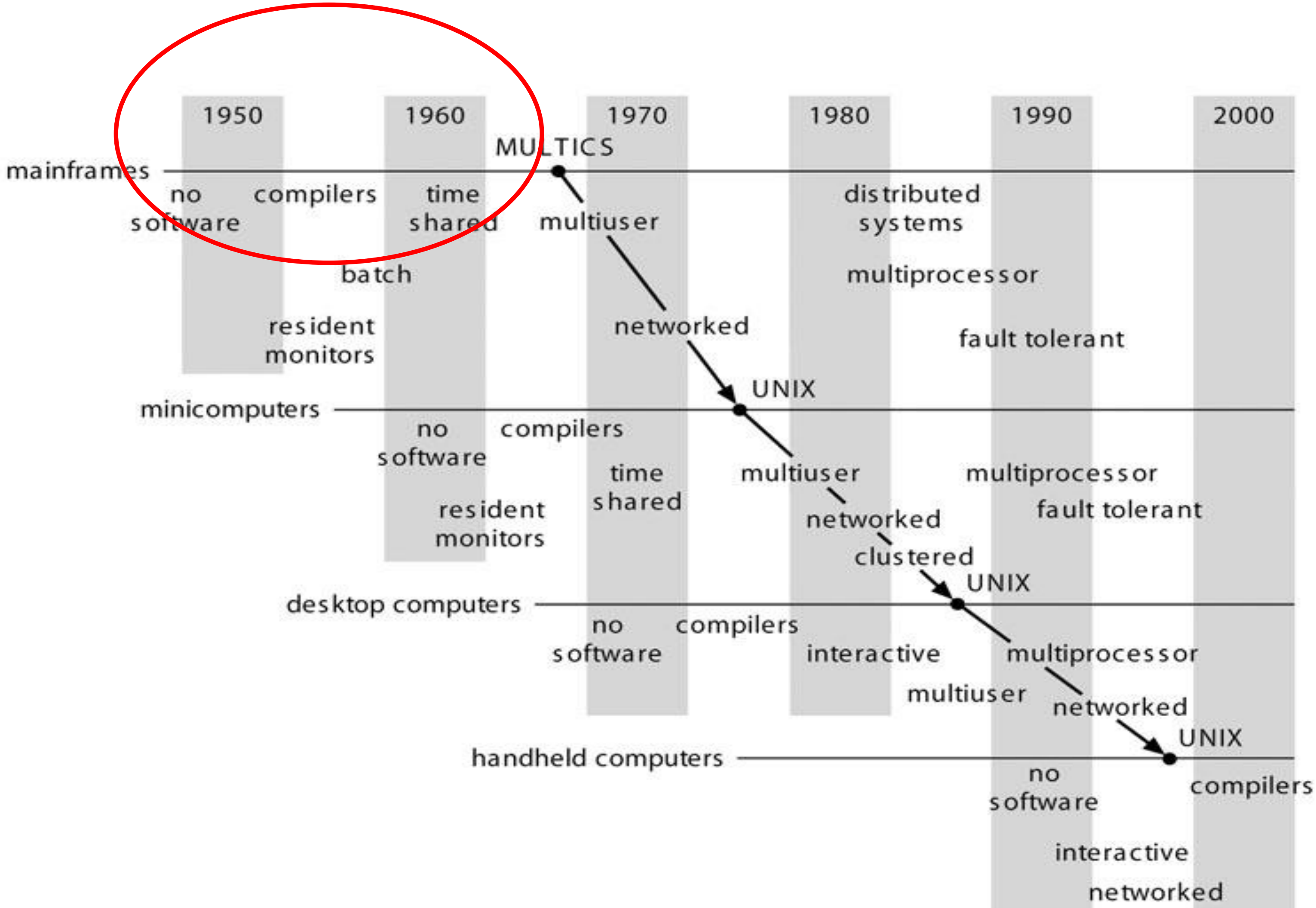
Minicomputers Desktops, Handhelds

Minicomputers: 8 cu. Feet,
330000 additions per second

Desktops: ? cu ft,
6 billion additions per second

Handhelds: in ur hand!,
600 million additions per second

History of Computers



5 Phases

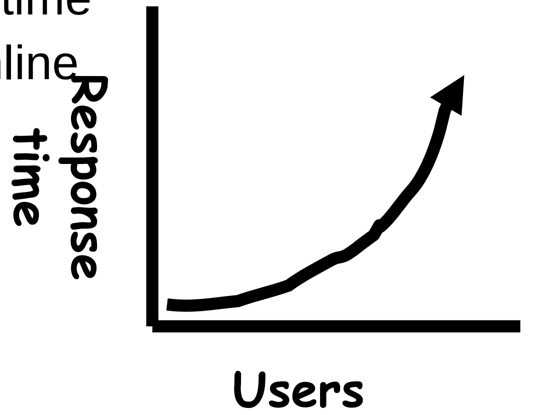
- Phase 1 (1948—1970)
- Phase 2 (1970 – 1985)
- Phase 3 (1981-)
- Phase 4 (1988 -): Distributed Systems
- Phase 5 (1995 -): Mobile Systems

Phase 1 (1948—1970)

- Hardware Expensive, Humans Cheap
- When computers cost millions of $’s, optimize for more efficient use of the hardware!
  - Lack of interaction between user and computer
- User at console: one user at a time
- Batch monitor: load program, run, print
- Optimize to better use hardware
  - When user thinking at console, computer idle$\Rightarrow$BAD!
  - Feed computer batches and make users wait
- *No protection:* what if batch program has bug?

Phase 2 (1970 – 1985)

- Hardware Cheaper, Humans Expensive
- Computers available for tens of thousands of dollars instead of millions
- OS Technology maturing/stabilizing
- *Interactive* timesharing:
  - Use cheap terminals (~$1000) to let multiple users interact with the system at the same time
  - Sacrifice CPU time to get better response time
  - Users do debugging, editing, and email online
- Problem: Thrashing
  - Performance very non-linear response with load
  - Thrashing caused by many factors including
    - Swapping, queueing

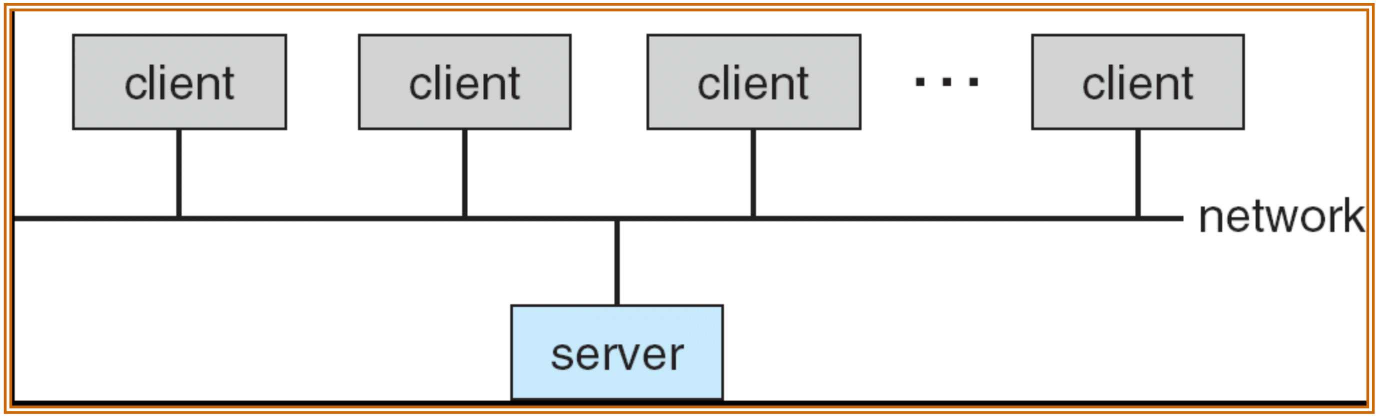


Phase 3 (1981-)

- Hardware Very Cheap, Humans Very Expensive
- Computer costs $1K, Programmer costs $100K/year
  - If you can make someone 1% more efficient by giving them a computer, it's worth it!
  - Use computers to make people more efficient
- Personal computing:
  - Computers cheap, so give everyone a PC
- Limited Hardware Resources Initially:
  - OS becomes a subroutine library
  - One application at a time (MSDOS, CP/M, …)
- Eventually PCs become powerful:
  - OS regains all the complexity of a “big” OS
  - multiprogramming, memory protection, etc (NT,OS/2)
- Question: As hardware gets cheaper does need for OS go away?

Phase 4 (1988 -): Distributed Systems

- Networking (Local Area Networking)
  - Different machines share resources
  - Printers, File Servers, Web Servers
  - Client – Server Model
- Services
  - Computing
  - File Storage



Phase 4 (1988 -): Internet

- Developed by the research community
  - Based on open standard: Internet Protocol
  - Internet Engineering Task Force (IETF)
- Technical basis for many other types of networks
  - Intranet: enterprise IP network
- Services Provided by the Internet
  - Shared access to computing resources: telnet (1970's)
  - Shared access to data/files: FTP, NFS, AFS (1980's)
  - Communication medium over which people interact
    - email (1980's), on-line chat rooms, instant messaging (1990's)
    - audio, video (1990's, early 00's)
  - Medium for information dissemination
    - USENET (1980's)
    - WWW (1990's)
    - Audio, video (late 90's, early 00's) – replacing radio, TV?
    - File sharing (late 90's, early 00's)

Phase 5 (1995 -): Mobile Systems

- Ubiquitous Mobile Devices
  - Laptops, PDAs, phones
  - Small, portable, and inexpensive
    - Recently twice as many smart phones as PDAs
    - Many computers/person!
  - Limited capabilities (memory, CPU, power, etc…)
- Wireless/Wide Area Networking
  - Leveraging the infrastructure
  - Huge distributed pool of resources extend devices
  - Traditional computers split into pieces. Wireless keyboards/mice, CPU distributed, storage remote
- Peer-to-peer systems
  - Many devices with equal responsibilities work together
  - Components of “Operating System” spread across globe