# COMP 310 Operating Systems

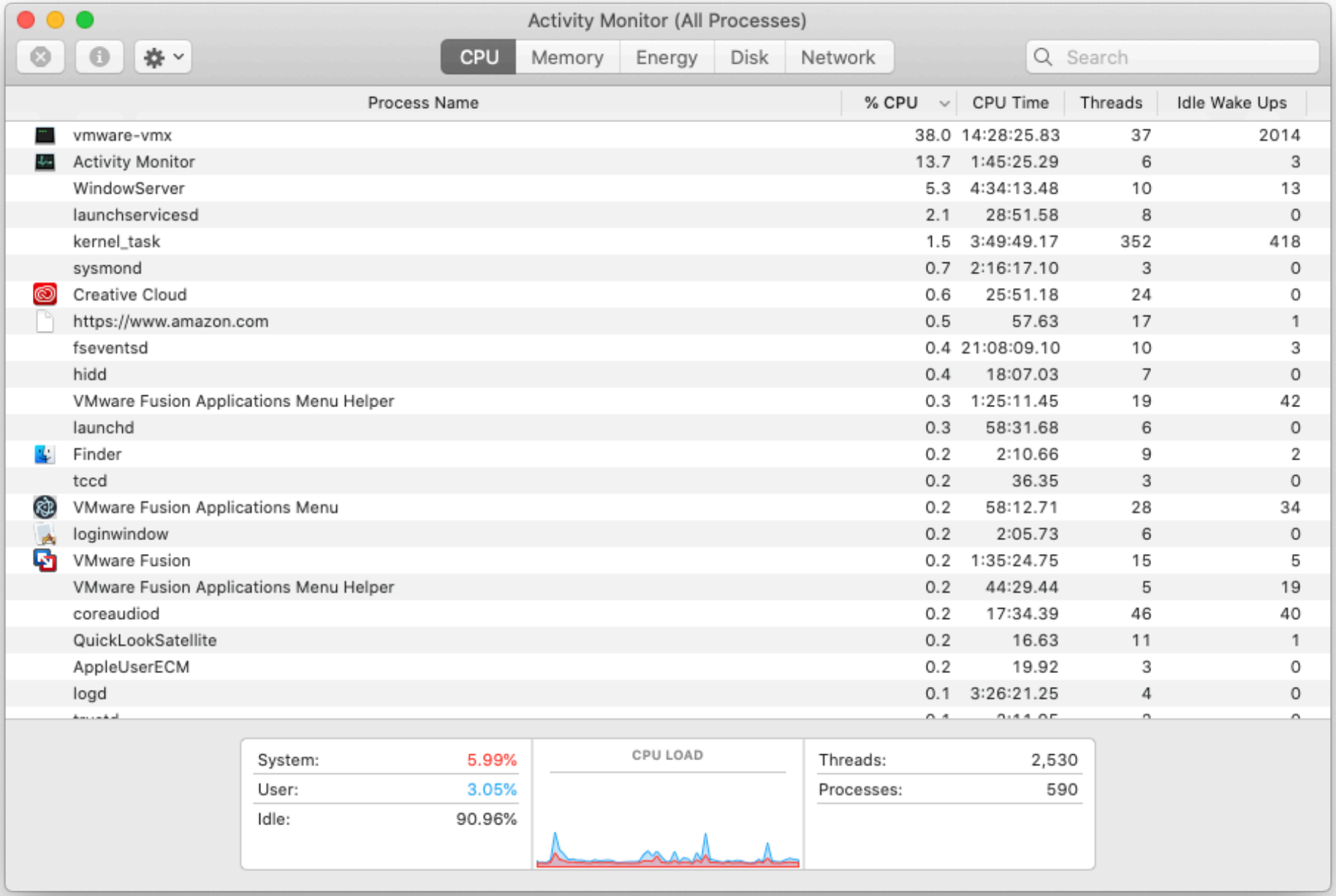
https://neilklingensmith.com/teaching/loyola/cs310-s2025/

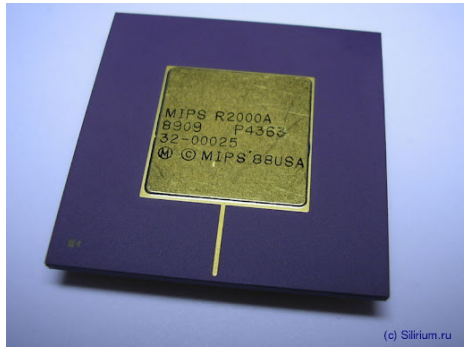
NEIL KLINGENSMITH, SPRING 2025

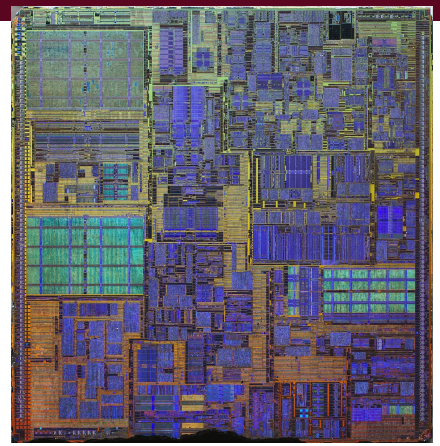

LOYOLA
UNIVERSITY CHICAGO

# WHAT IS THIS GUY DOING?

UNIVAC, 1951

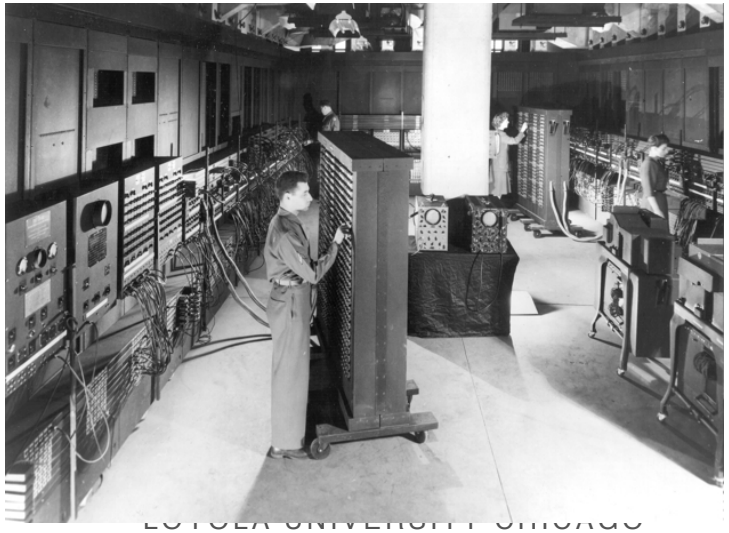

Activity Monitor (All Processes)
CPU
Memory
Energy
Disk
Network
Search
Process Name
% CPU
CPU Time
Threads
Idle Wake Ups
vmware-vmx 38.0 14:28:25.83 37 2014
Activity Monitor 13.7 1:45:25.29 6 3
WindowServer 5.3 4:34:13.48 10 13
launchservicesd 2.1 28:51.58 8 0
kernel_task 1.5 3:49:49.17 352 418
sysmond 0.7 2:16:17.10 3 0
Creative Cloud 0.6 25:51.18 24 0
https://www.amazon.com 0.5 57.63 17 1
fseventsd 0.4 21:08:09.10 10 3
hidd 0.4 18:07.03 7 0
VMware Fusion Applications Menu Helper 0.3 1:25:11.45 19 42
launchd 0.3 58:31.68 6 0
Finder 0.2 2:10.66 9 2
tccd 0.2 36.35 3 0
VMware Fusion Applications Menu 0.2 58:12.71 28 34
loginwindow 0.2 2:05.73 6 0
VMware Fusion 0.2 1:35:24.75 15 5
VMware Fusion Applications Menu Helper 0.2 44:29.44 5 19
coreaudiod 0.2 17:34.39 46 40
QuickLookSatellite 0.2 16.63 11 1
AppleUserECM 0.2 19.92 3 0
logd 0.1 3:26:21.25 4 0
System: 5.99%
User: 3.05%
Idle: 90.96%
CPU LOAD
Threads: 2,530
Processes: 590



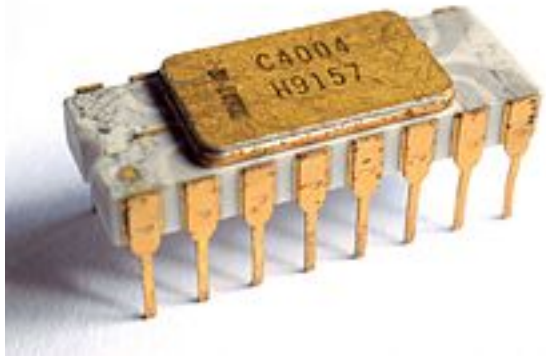



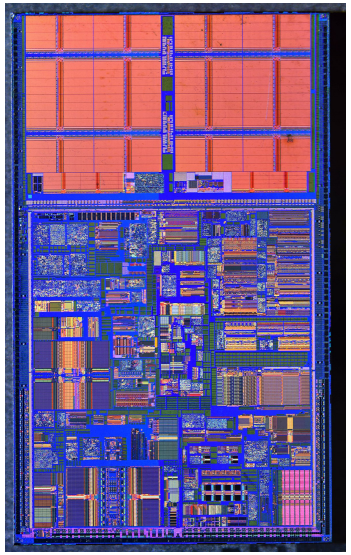
1945 1958 1964 1971 1986 1997 2000 2007 2022

ENIAC FORTRAN IBM 360 Intel 4004 Pentium II Pentium 4 CUDA RTX 4090



MIPS R2k

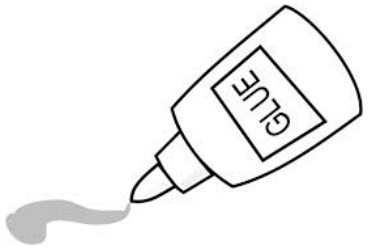


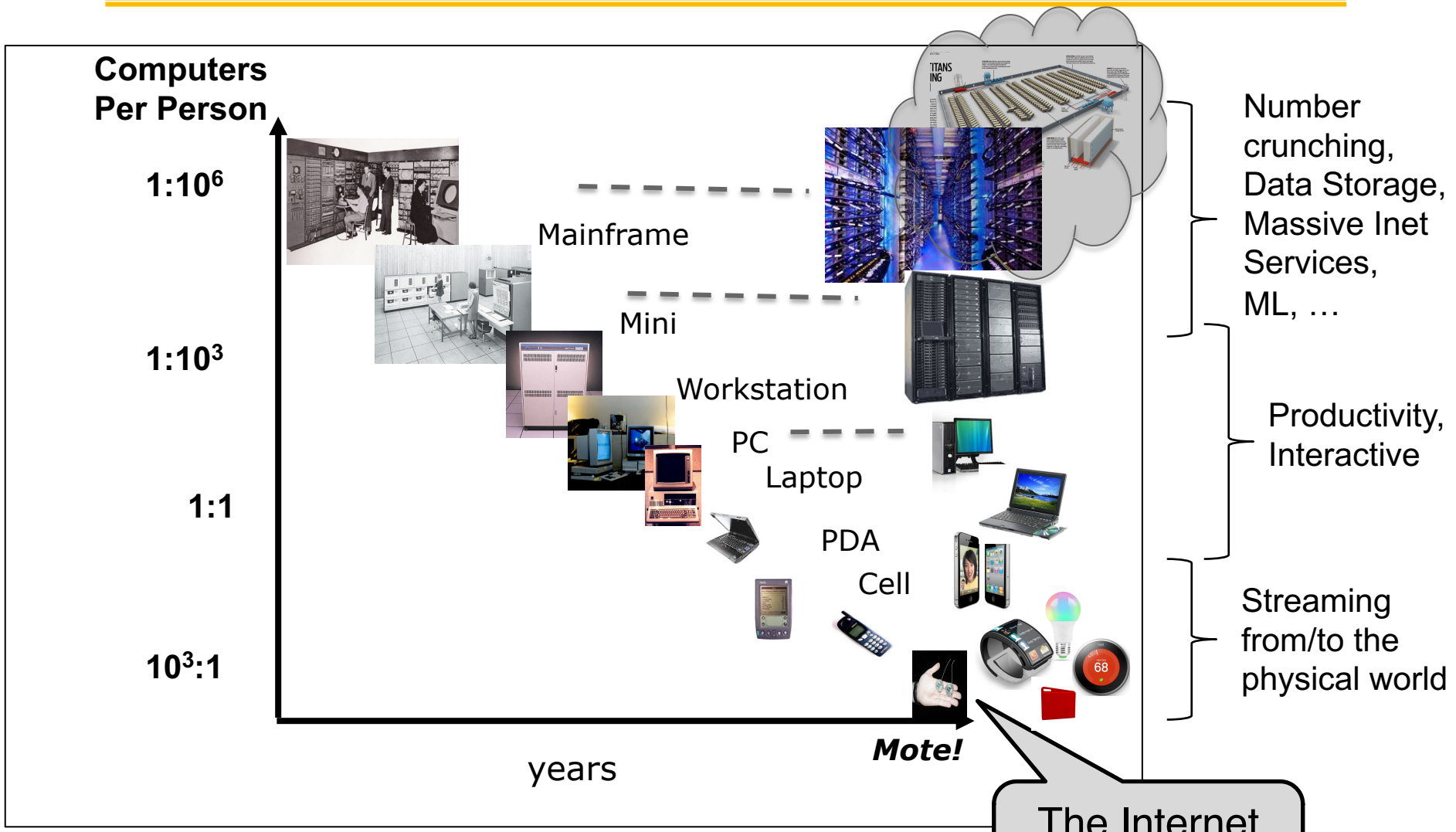
# What is an Operating System?

- **Referee**
  - Manage sharing of resources, Protection, Isolation
    - » Resource allocation, isolation, communication
- **Illusionist**
  - Provide clean, easy to use abstractions of physical resources
    - » Infinite memory, dedicated machine
    - » Higher level objects: files, users, messages
    - » Masking limitations, virtualization

- **Glue**
  - Common services
    - » Storage, Window system, Networking
    - » Sharing, Authorization
    - » Look and feel



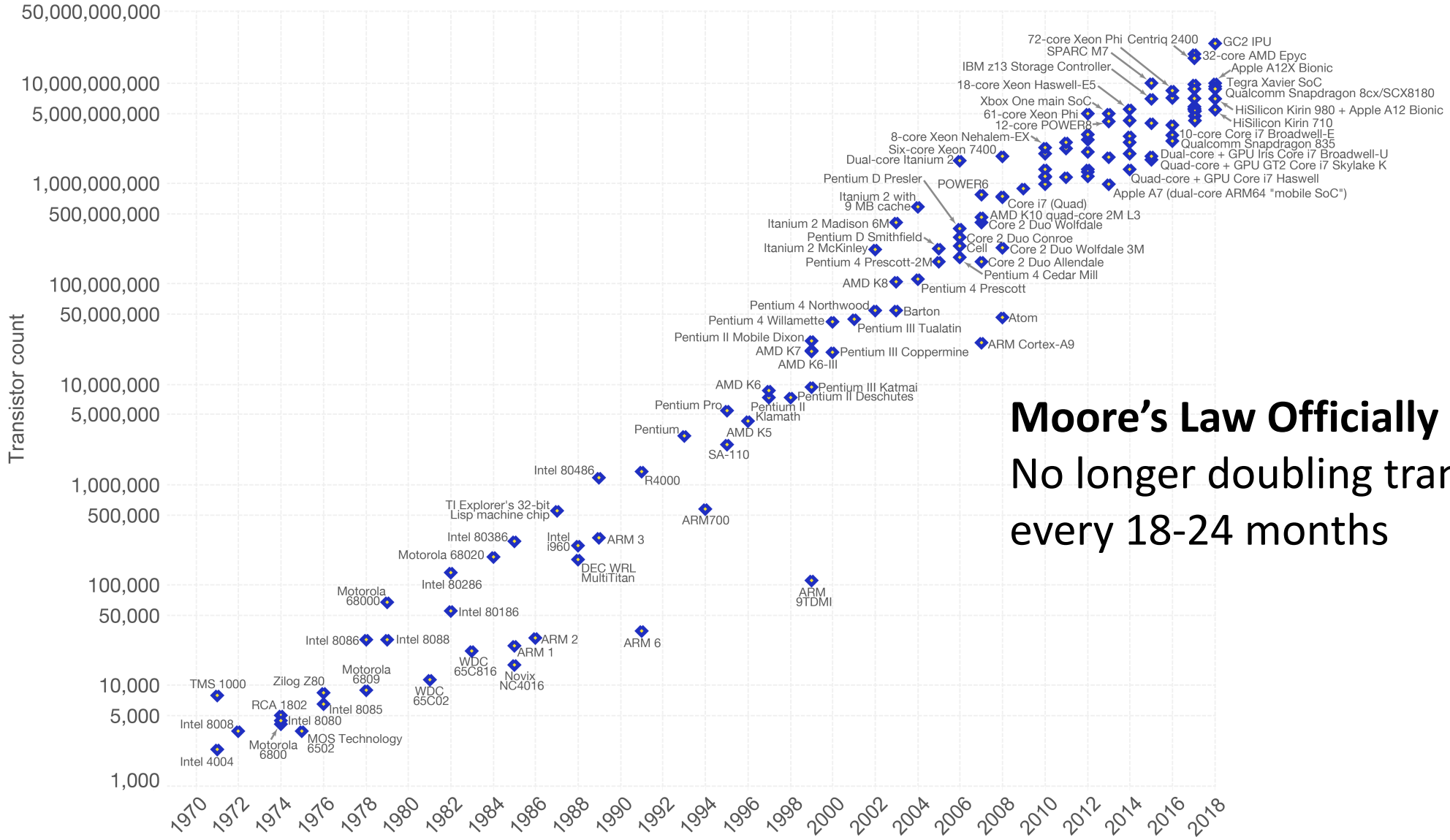
# Across incredibly diversity




Bell's Law: new computer class per 10 years

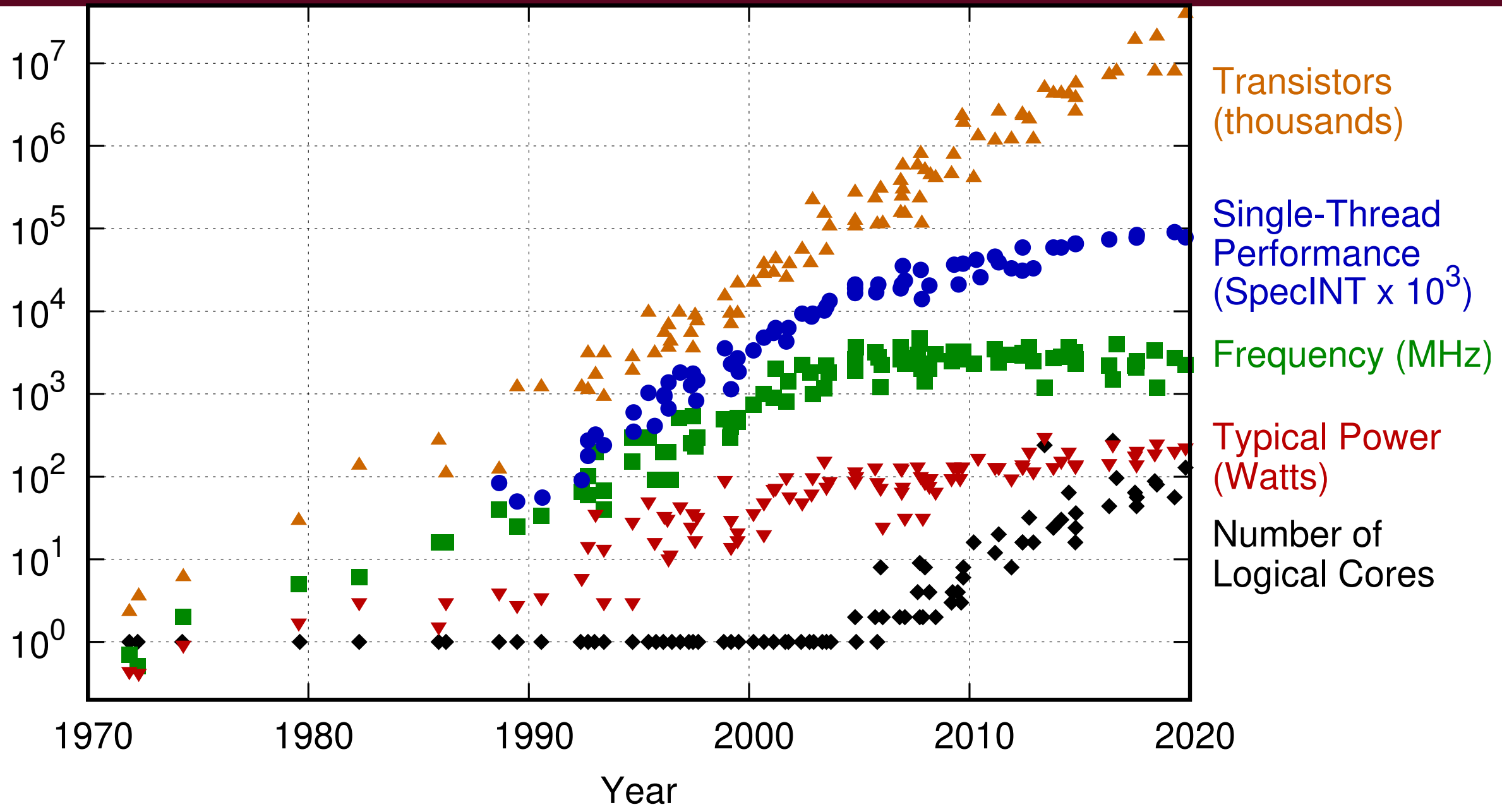
# Moore’s Law – The number of transistors on integrated circuit chips (1971-2018)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.




**Moore’s Law Officially Ended in 2016:**

No longer doubling transistor density every 18-24 months




Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

# Vast Range of Timescales

**Jeff Dean's "Numbers Everyone Should Know"**

| | |
|---|---|
| L1 cache reference | 0.5 ns |
| Branch mispredict | 5 ns |
| L2 cache reference | 7 ns |
| Mutex lock/unlock | 25 ns |
| Main memory reference | 100 ns |
| Compress 1K bytes with Zippy | 3,000 ns |
| Send 2K bytes over 1 Gbps network | 20,000 ns |
| Read 1 MB sequentially from memory | 250,000 ns |
| Round trip within same datacenter | 500,000 ns |
| Disk seek | 10,000,000 ns |
| Read 1 MB sequentially from disk | 20,000,000 ns |
| Send packet CA->Netherlands->CA | 150,000,000 ns |

# Operating Systems Help Manage Complexity

Advances in hardware make programming difficult

- OS Provides Consistent Abstractions
- OS Manages Resource Sharing

Key Building Blocks:

- Processes
- Threads, Concurrency, Scheduling, Coordination
- Address Spaces
- Protection, Isolation, Security
- Communication
- Persistent Storage, transactions, consistency, resilience
- Interfaces to Devices

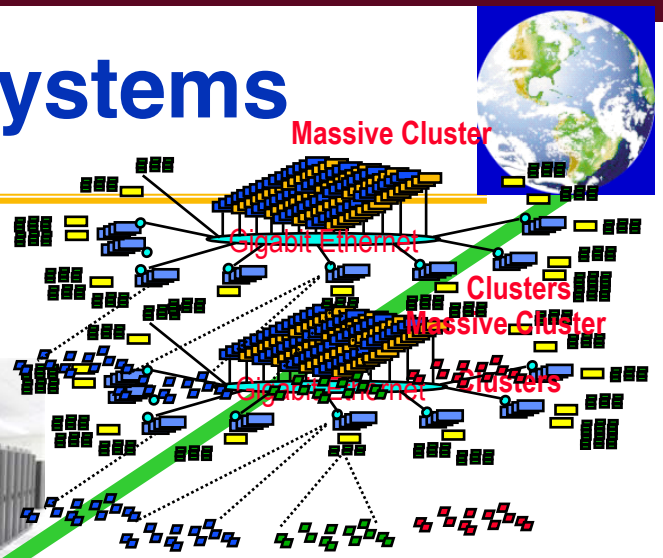
# Not Only PCs connected to the Internet

- **In 2011, smartphone shipments exceeded PC shipments!**
- **2011 shipments:**
  - **487M smartphones**
  - **414M PC clients**
    - » **210M notebooks**
    - » **112M desktops**
    - » **63M tablets**
  - **25M smart TVs**


- **4 billion phones in the world → smartphones over next few years**
- **Then…**

# Societal Scale Information Systems

- **The world is a large distributed system**
  - **Microprocessors in everything**
  - **Vast infrastructure behind them**



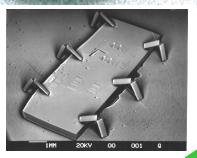


Internet Connectivity

Scalable, Reliable, Secure Services



Databases
Information Collection
Remote Storage
Online Games
Commerce
…



MEMS for Sensor Nets

# What is the OS?

- Components:
  - Memory Management
  - I/O Management
  - CPU Scheduling
  - Communications? (Email?)
  - Multitasking?
- What About:
  - File System?
  - Multimedia Support?
  - User Interface/Windowing?
  - Internet Browser?


- There's no universally-accepted definition.
- The one program that runs all the time is the kernel.
- Maybe you can say "everything that comes with a fresh OS install"
- *Studying OSes is really about the Hardware/Software interface (API)* - John Kubiatowicz

# Policy/Mechanism

Goal:

- Keep user programs from crashing the OS
- Keep user programs from crashing each other

Policy:

- Programs are not allowed to read/write memory of ther programs or of the OS

Mechanism:

- Address translation
- Dual-mode operation


1. Rosetta
2. mac OS Port

1. OS
2. Driver Support

SiFive

Given we have a single processor cache that is

- 32-bit address space
- Word addressed (addresses are left shifted by 2 by adding “00” to end of address inside the processor, this implies that it can address 2^32*4 = 16GBytes of memory)
- Cache is 16KByte in size
- Cache block size (aka cache line size) = 16 words (64 bytes = 16*4)
    - # of cache blocks = 256
- direct mapped (1-way associative)

From the above information, we can infer that the offset requires 4 bits (2^4=16), the index requires 8 bits (2^8=256), and tag is 20 bits.

From the testing perspective, what are the interesting cases we would want to test about the operation of the cache? What may be some corner cases?

Write a test program that generates addresses to access the cache while hitting the interesting cases and corner cases of the cache.

What makes a good test?

- Random traffic
- Hits corner cases (interesting scenarios a totally random test will not activate)
- Hits the corner cases randomly rather than explicitly
- Come up with a reasonable number of cycles to test with each type of random traffic to get a good tradeoff between compute resource and test thoroughness.

For the sake of simplicity, we will not put the checker code in this test program (assume the correctness will be checked elsewhere), and for the sake of this problem, we will not be testing the data part of the program. In another word, this is a cache traffic driver program. The checker code will be placed elsewhere.

Take as much time as you want, but I’m expecting people to only spend 20-40 minutes on this.

Example Code in C - you can use any programming language you are comfortable with

---

```
#include <stdint.h>

#define TAG_WIDTH 20
#define INDEX_WIDTH 8
#define NUM_INDEX (1<<INDEX_WIDTH)
#define OFFSET_WIDTH 4
#define NUM_OFFSET (1<<OFFSET_WIDTH)

int main() {
   /* Enter your code here.

      Your code needs to use these two procedures to perform operations on the cache.
      These two procedures are already defined:

      call WriteToMemory(addr) to write to address and
      call ReadFromMemory(addr) to read from address


      Remembering we are simplifying the problem so don't worry about the write data or read
data
    */

   /* Code example:
      This is a bad test in more ways than one, you will need to replace or add to this
      test.  It's here to show you how to generate addresses to read and write to caches.
      Do not assume the solution will be similar to this snippit of test code.
      */
   for(int i=0;i<10000;i++) {
      uint32_t addr = rand();
      uint32_t data = rand();
      /* uncomment this line for debugging, but final code should be commented out
         Note: If this is uncommented, the test will fail */
      //       printf("Generated Addr: %8x\n", addr);
      if(rand() % 2) {
         WriteToMemory(addr, data);
      } else {
         data = ReadFromMemory(addr);
      }
   }
   return 0;
```

# Turning in Assignments

- We will use GitHub Classroom. See course webpage for link.
- Fill out the survey on the course website (see schedule for today).

# Coding Guidelines

- Make sure you test code a bit at a time—split into functions.
- Build pieces one at a time.
- Plan first.

# Homework

- Class will be front-loaded with homework
- Each week you will have two assignments

**Homework Assignment**

Adding a feature to your kernel

**“In-Class” Activity**

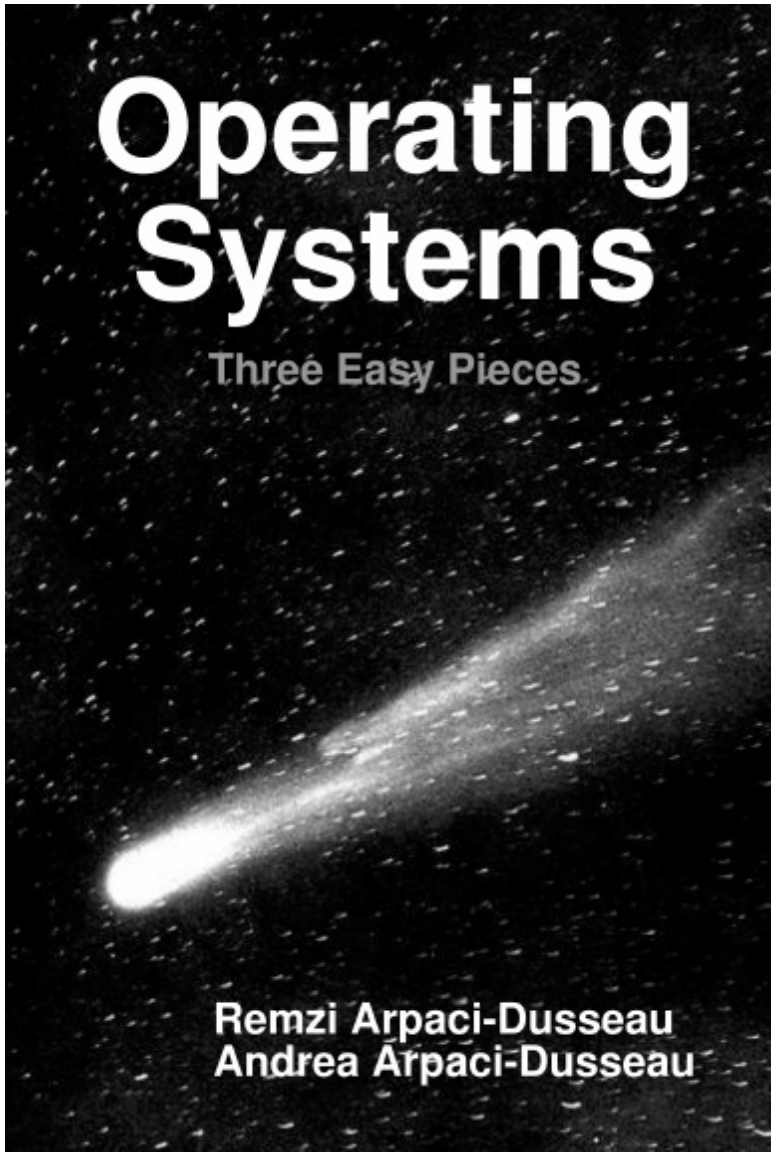
Informal coding practice

# Programming in C

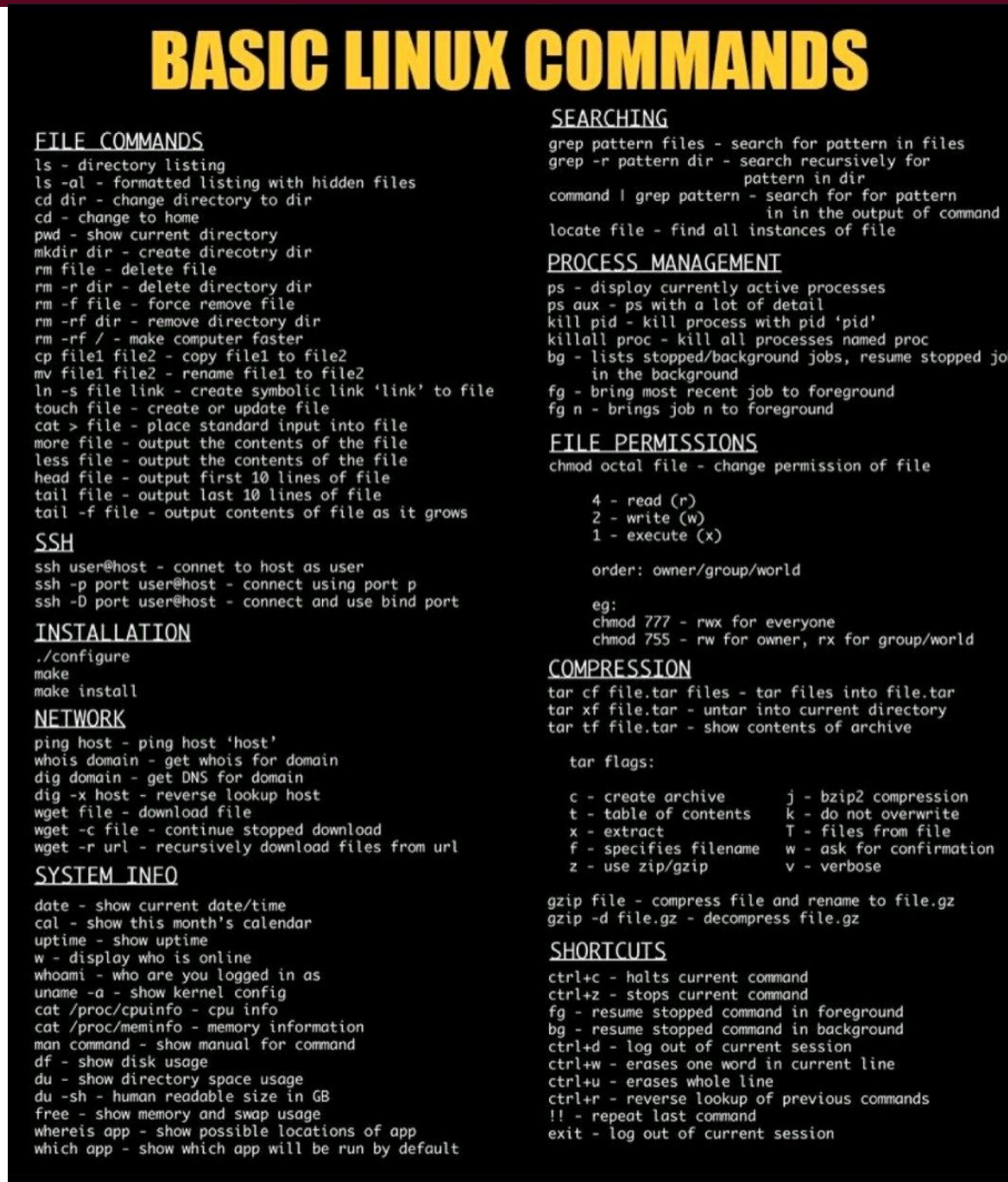
- You're supposed to kinda know how to write C code
- You need to get good at writing C fast
- C refresher available at: `https://os.neilklingensmith.com`

# The Textbook

- Free @ `http://ostep.org`
- Links to relevant chapters on course webpage schedule



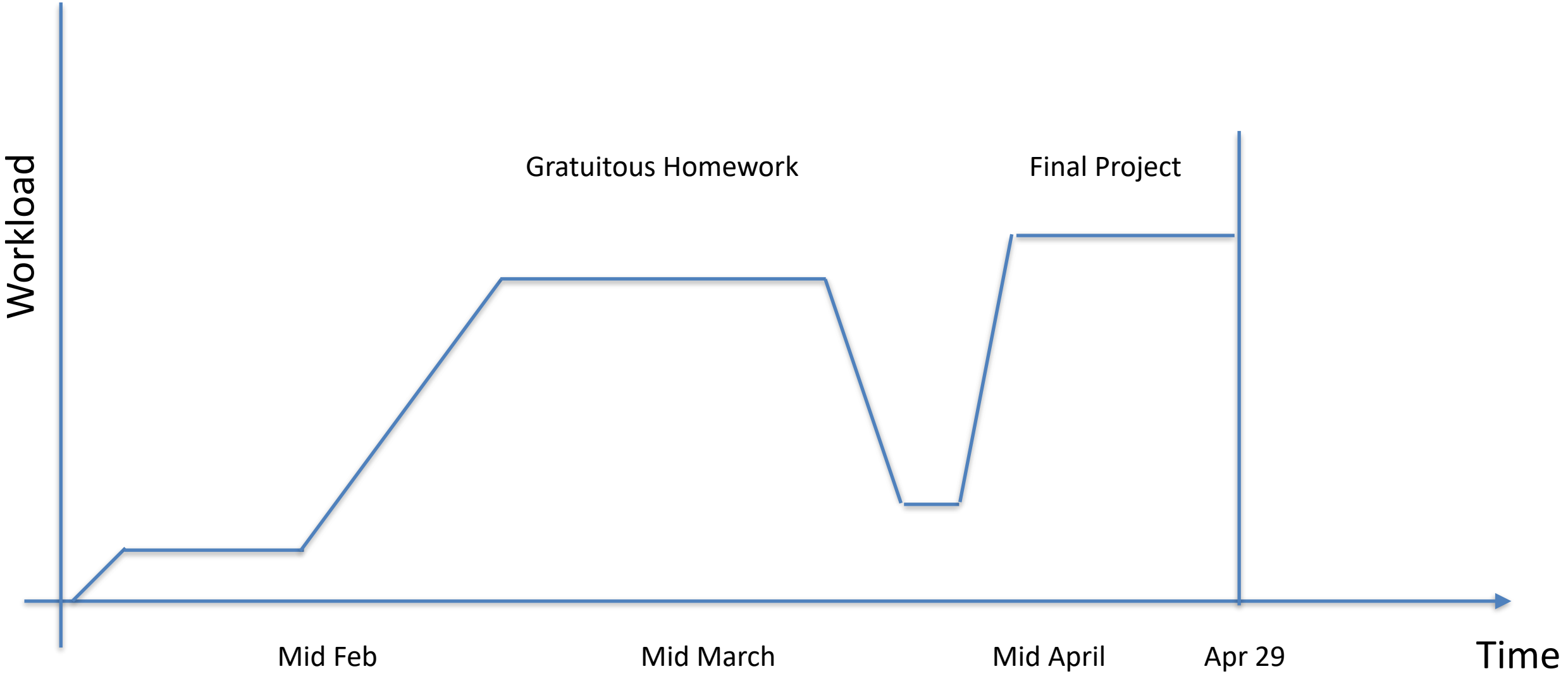
# Check Course Website



# Grading

| Category | Weight |
|---|---|
| Homework | 40% |
| Quizzes | 20% |
| Final Project | 40% |

- No exams. Your grade is based on homework and final project.
- No partial credit for code that doesn't compile.
- Start homework on Tuesday/Wednesday so you can get help on Thursday in lab if you get stuck.
- NO LATE WORK WILL BE ACCEPTED
- I will drop the lowest two quizzes and homeworks at the end of the semester.


Workload
Gratuitous Homework
Final Project
Mid Feb
Mid March
Mid April
Apr 29
Time

*Do what is easy and your life will be hard.*
*Do what is hard and your life will be easy.*

- Motivational kitchen magnet

# Course Website

https://neilklingensmith.com/teaching/loyola/cs310-s2025/

# Lab

Thursday 4--6 PM
Doyle Makerspace

# Lab Kits

- VMware Fusion (mac) or Workstation (PC)
  - Check Wayback Machine link if you're having trouble downloading.
- Or a burner laptop

Install Ubuntu this week.

# Microsoft Teams Channel

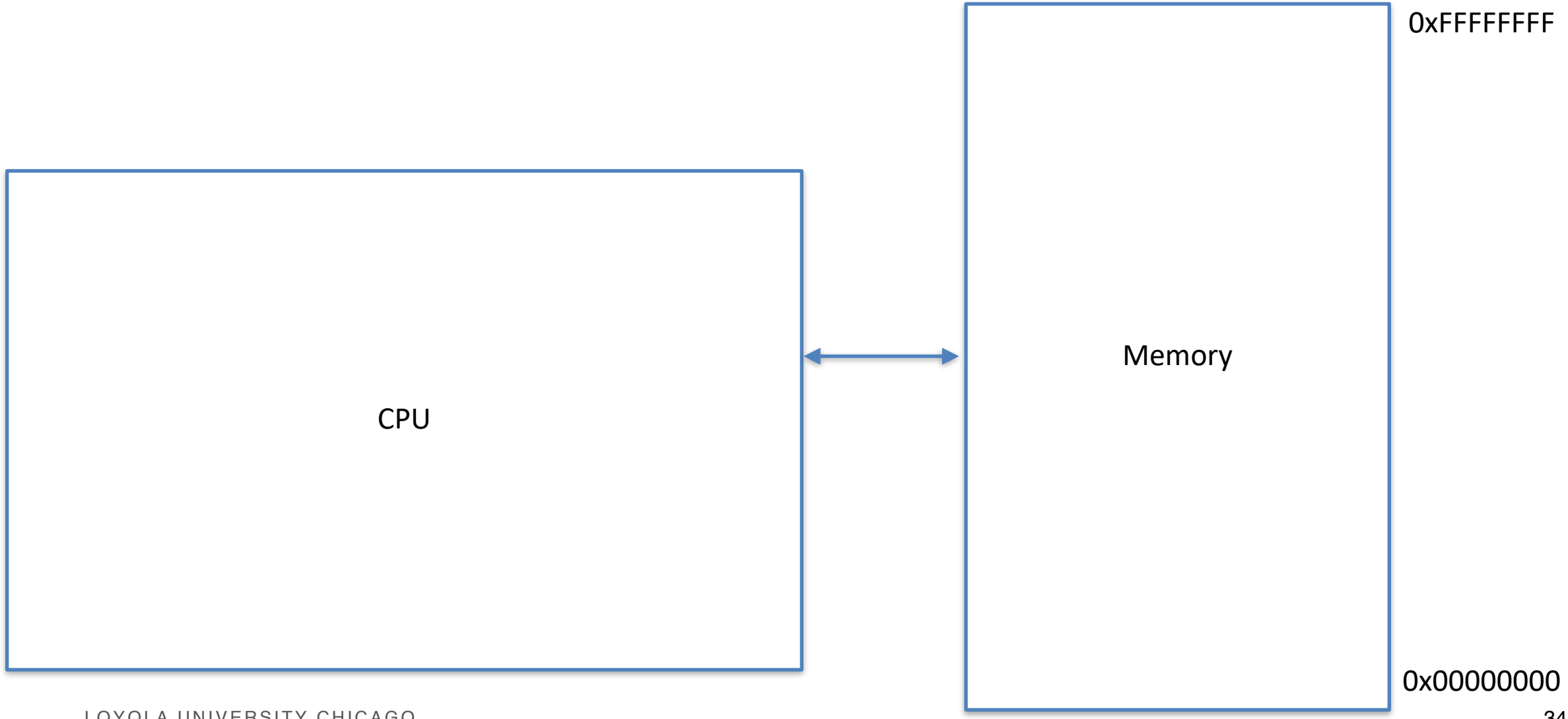
Link in welcome email.

# DoD Cyber Service Academy Scholarship

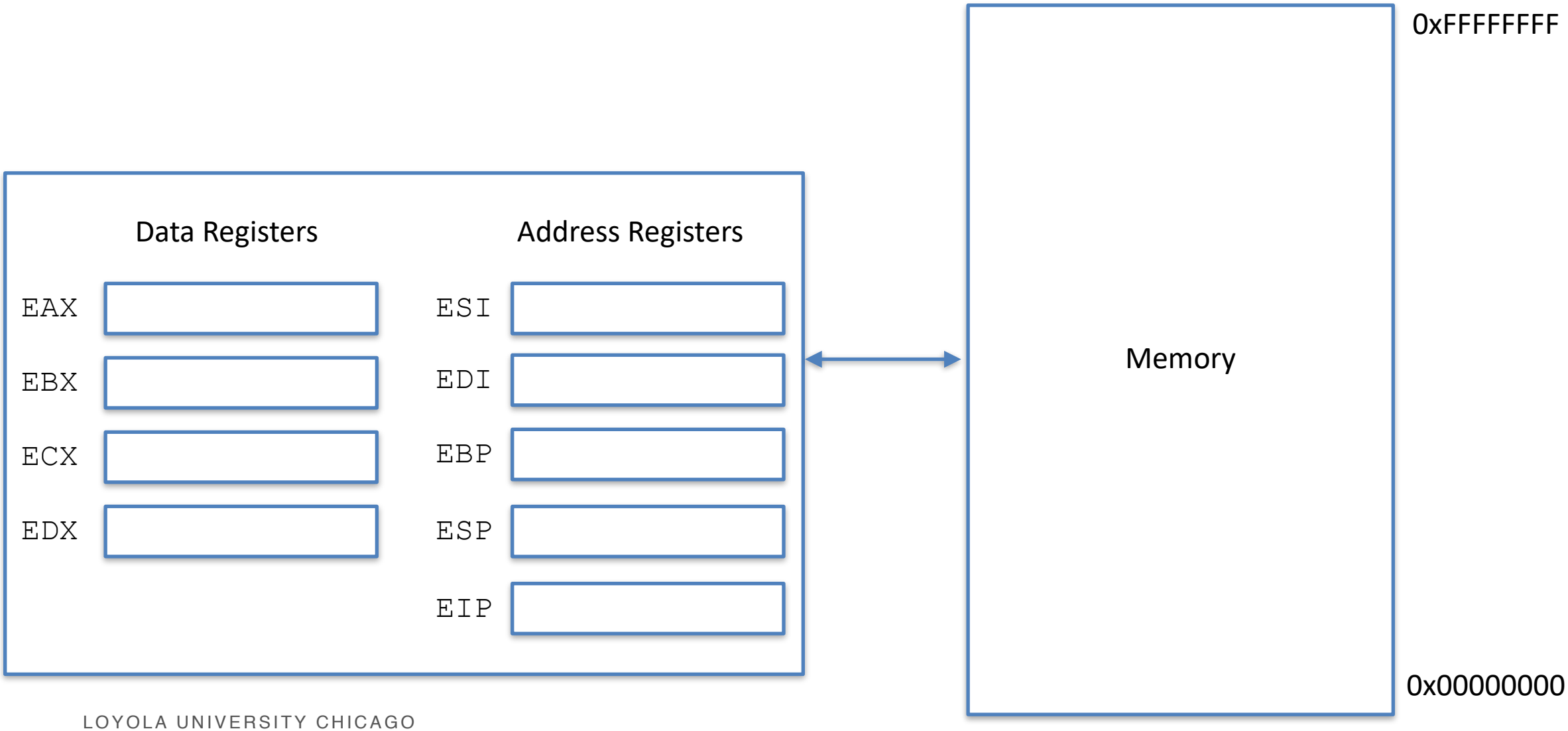
- Full year scholarship
- Plus internship or job offer
- Open to juniors and seniors
- Applications open until Feb 1

# BOOTLOADERS

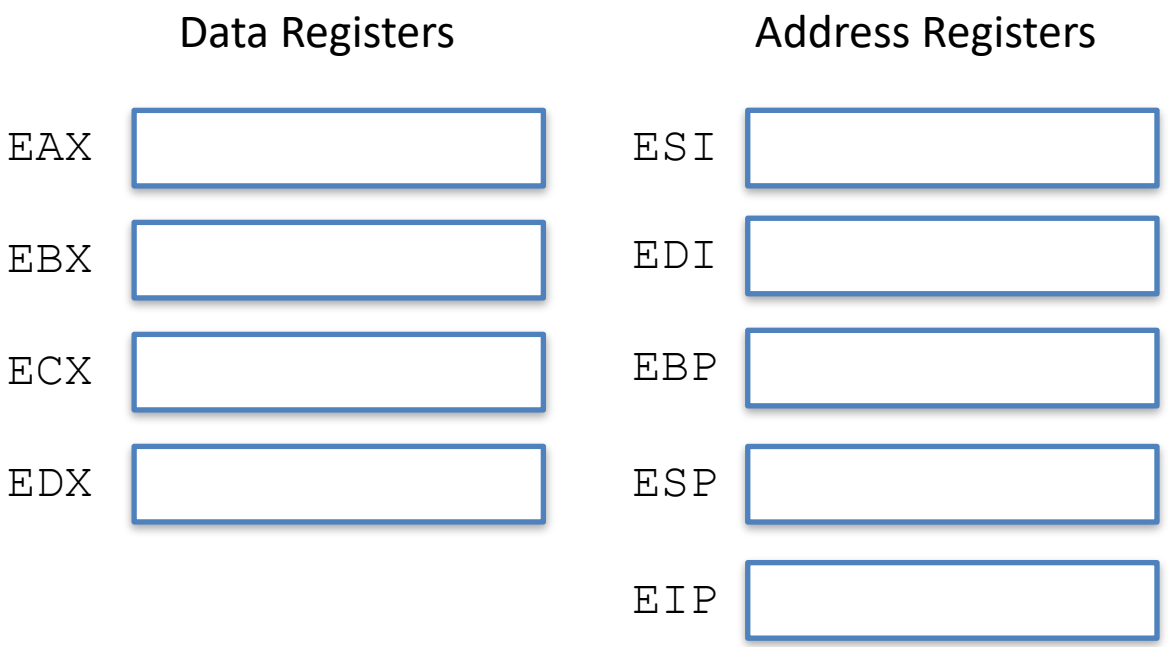
# Programmer's Model of 386



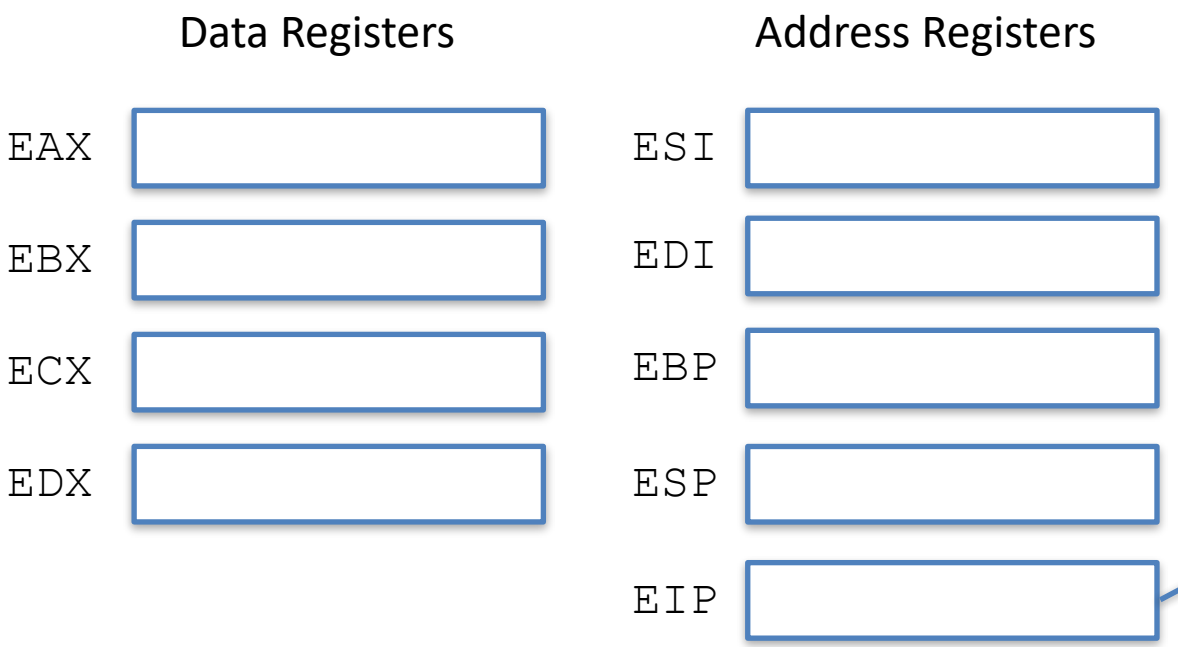
# Programmer's Model of 386



# PROGRAMMER'S MODEL OF 386: INSIDE THE CPU

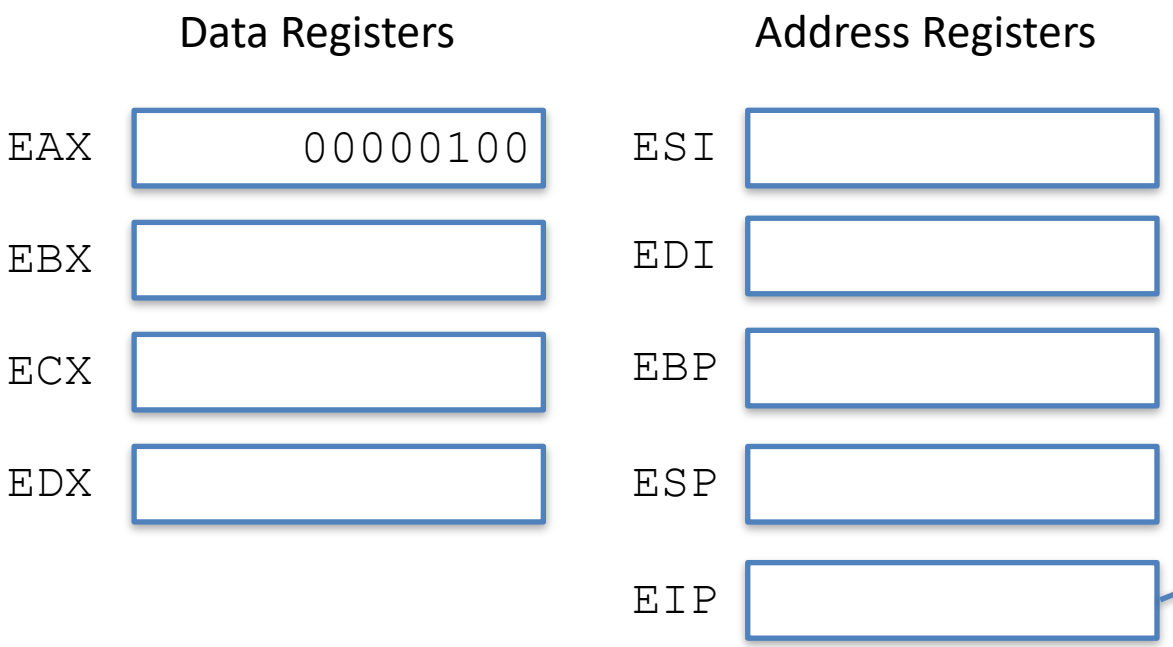


# PROGRAMMER'S MODEL OF 386: INSIDE THE CPU



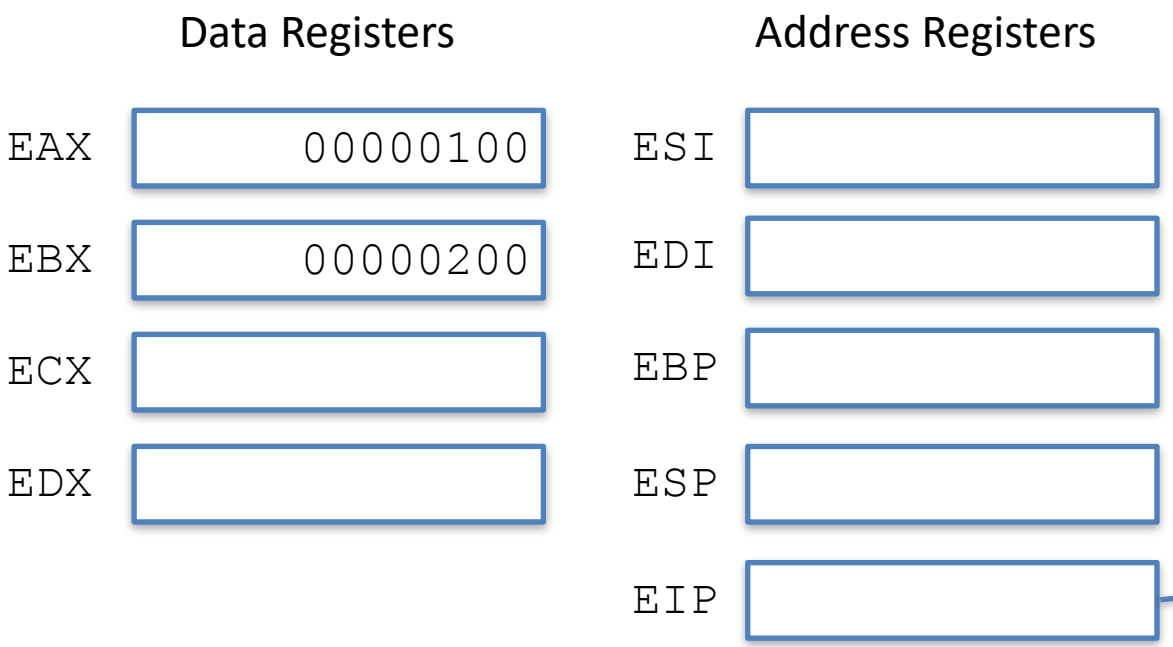

```
main:
  mov eax,00000100h
  mov ebx,00000200h
loop:
  add eax,ebx
  cmp eax,00000400h
  jlt loop
  ret
```

# PROGRAMMER'S MODEL OF 386: INSIDE THE CPU



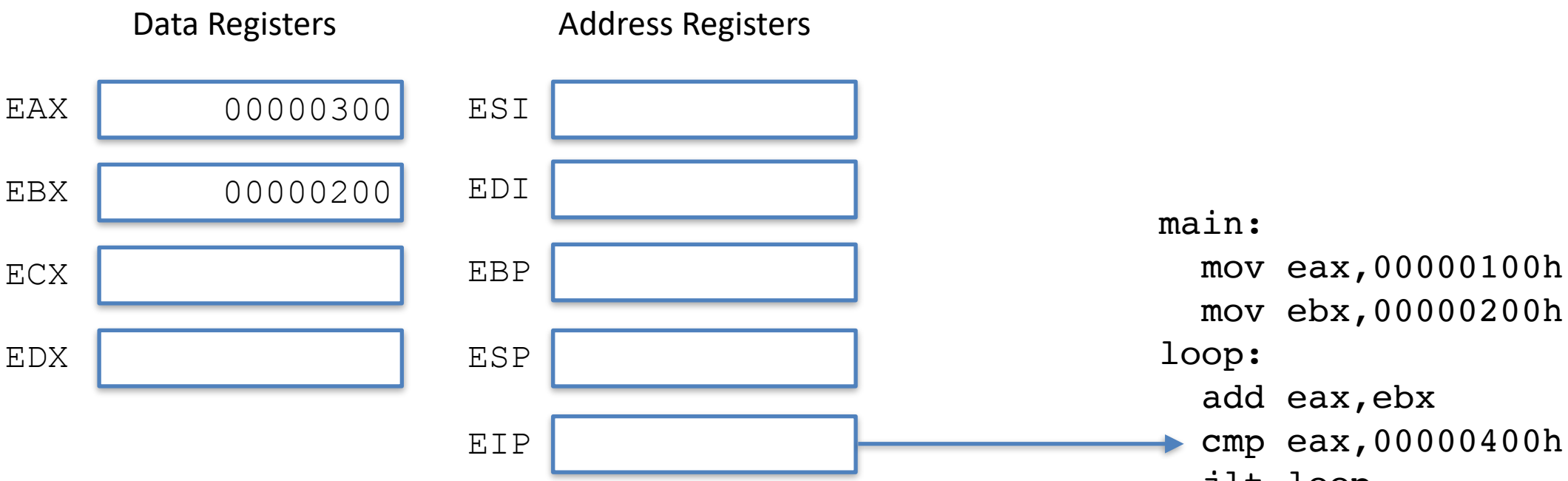

```
main:
   mov eax,00000100h
   mov ebx,00000200h
loop:
   add eax,ebx
   cmp eax,00000400h
   jlt loop
   ret
```

# PROGRAMMER'S MODEL OF 386: INSIDE THE CPU



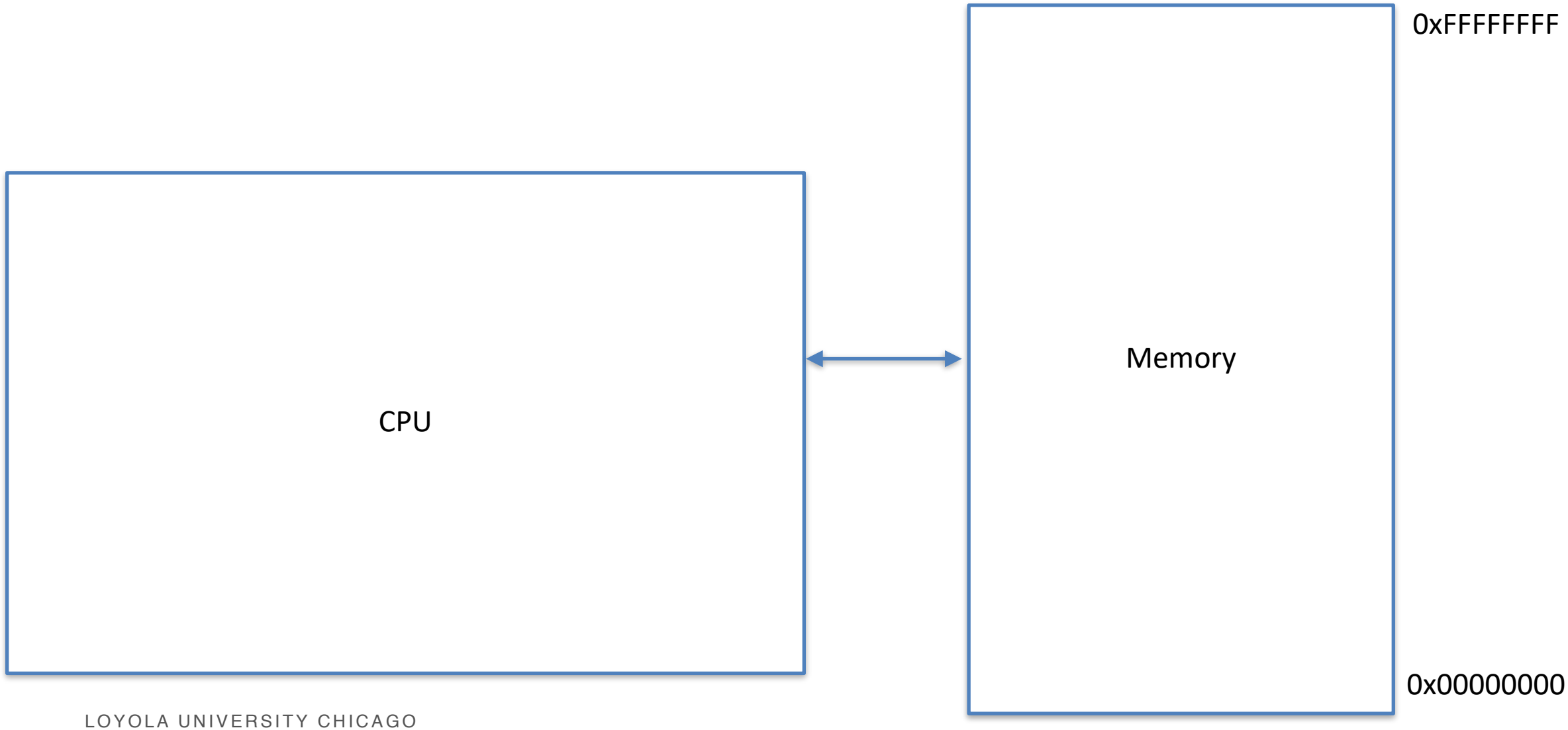

```
main:
   mov eax,00000100h
   mov ebx,00000200h
loop:
   add eax,ebx
   cmp eax,00000400h
   jlt loop
   ret
```

# PROGRAMMER'S MODEL OF 386: INSIDE THE CPU

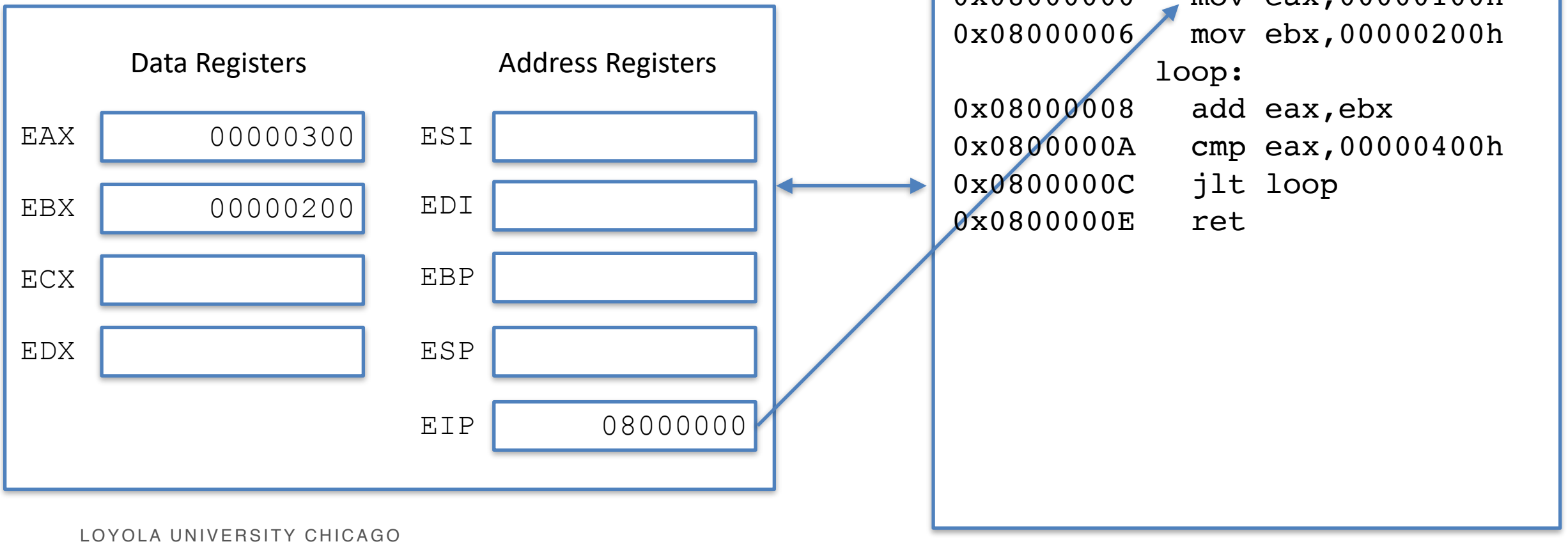



```
main:
   mov eax,00000100h
   mov ebx,00000200h
loop:
   add eax,ebx
   cmp eax,00000400h
   jlt loop
   ret
```

# Where should the program live in memory?



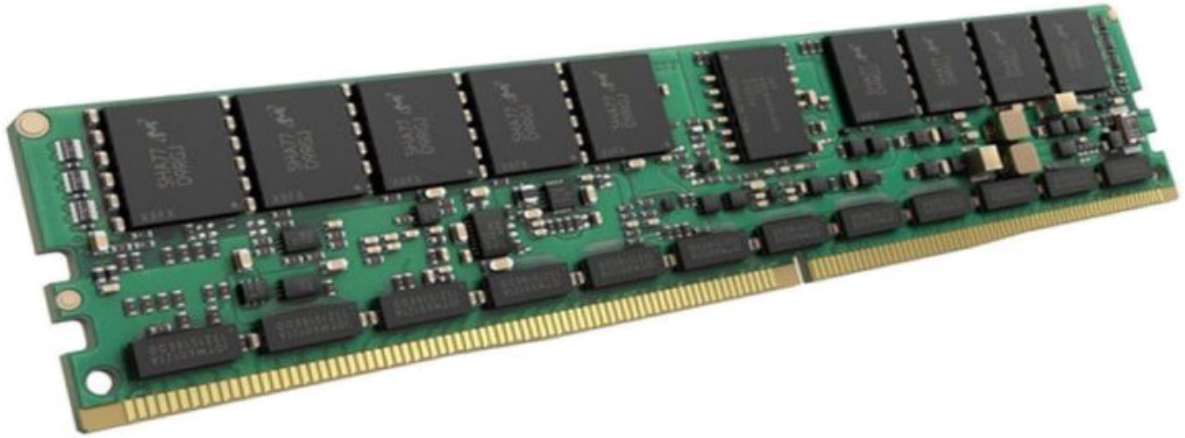
# Say I decide to put my program at 0x8000000 How does it get there?



# SO HOW DOES THE OS GET INTO MEMORY?

AS WE'LL SEE, OUR HARDWARE CHOICES ARE NOT AWESOME.

## DDR SDRAM (Main Memory)



Volatile👎

Loses its contents on poweroff

Must be re-initialized on each boot

Read/Write👍

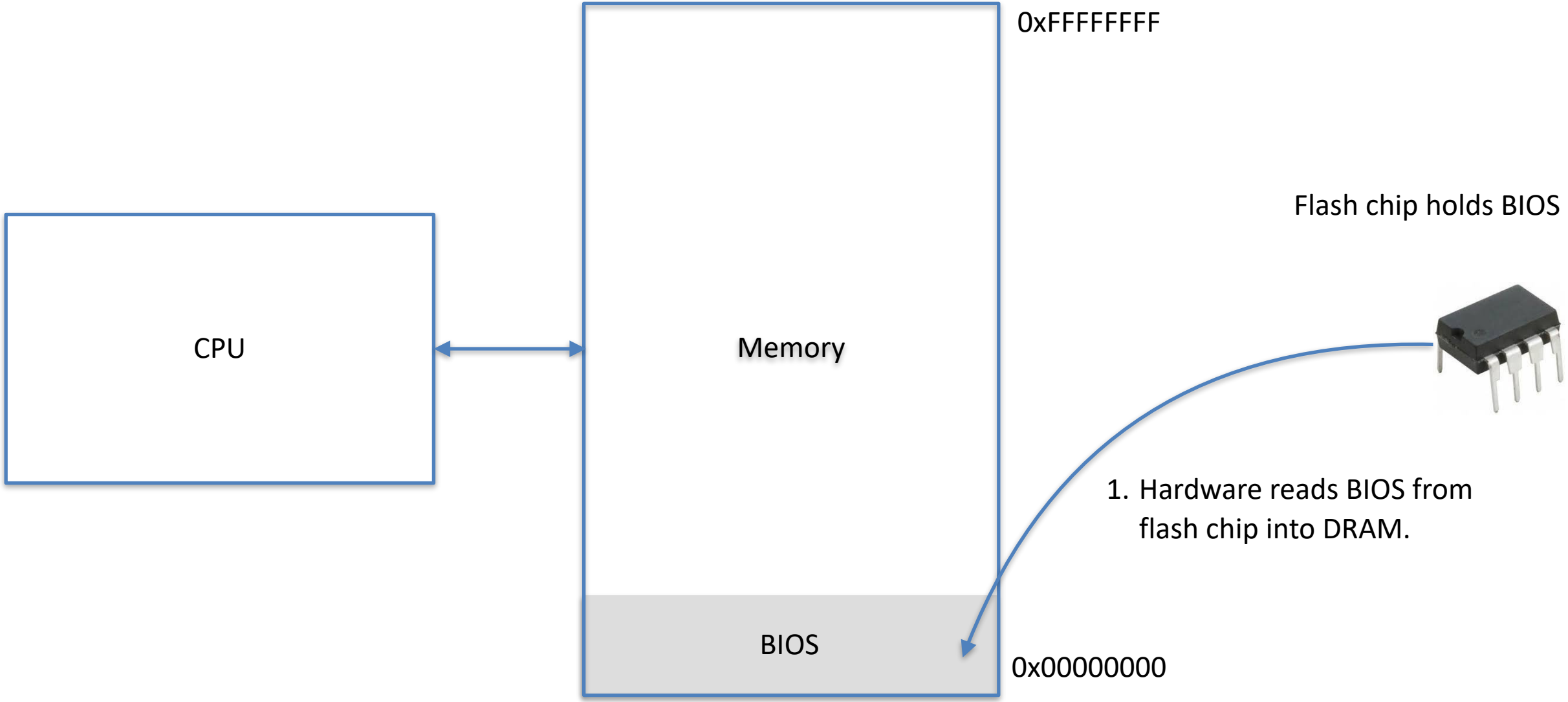
## Flash Memory

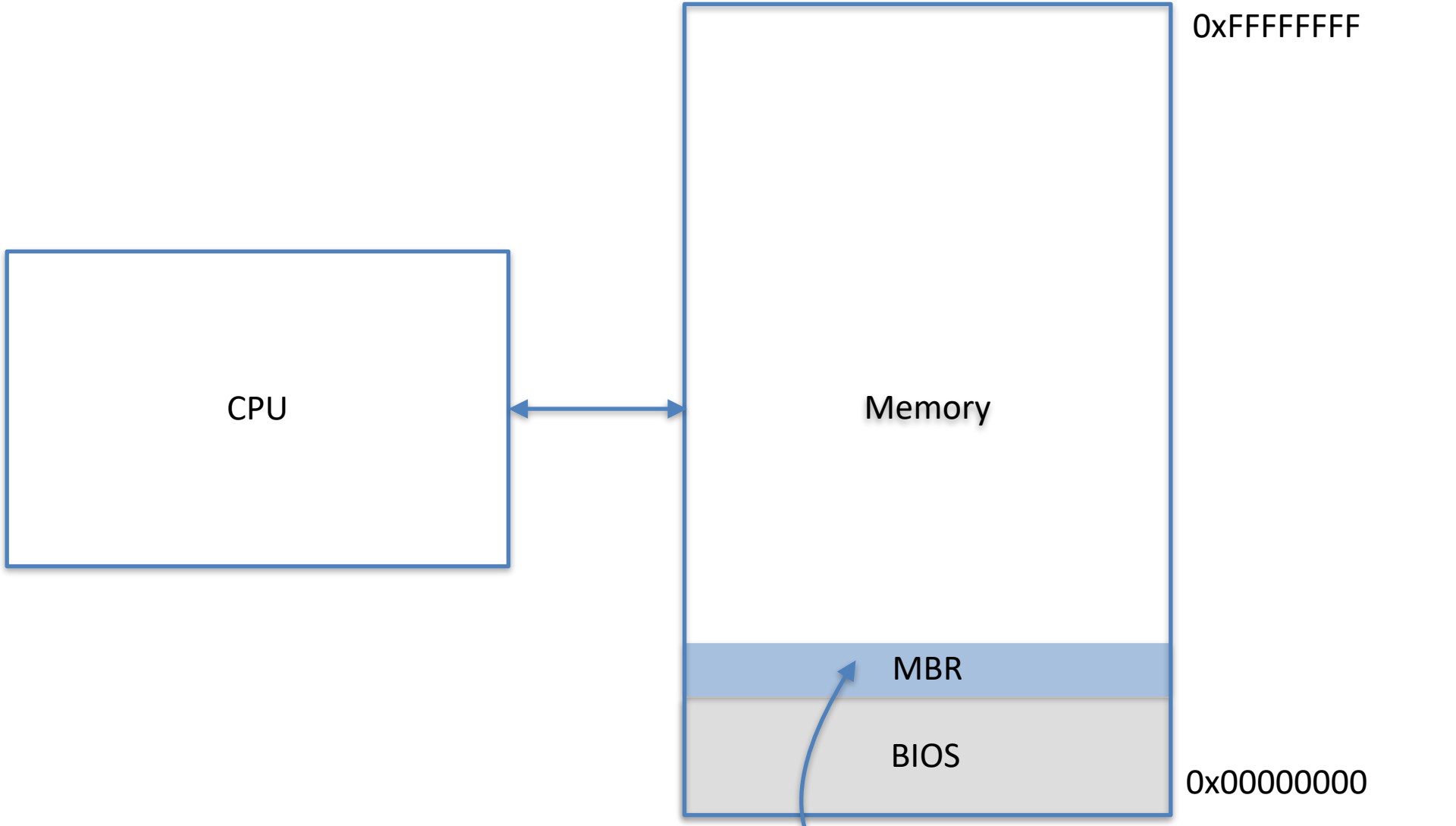

Nonvolatile👍

Retains its contents on poweroff

Read Only👎

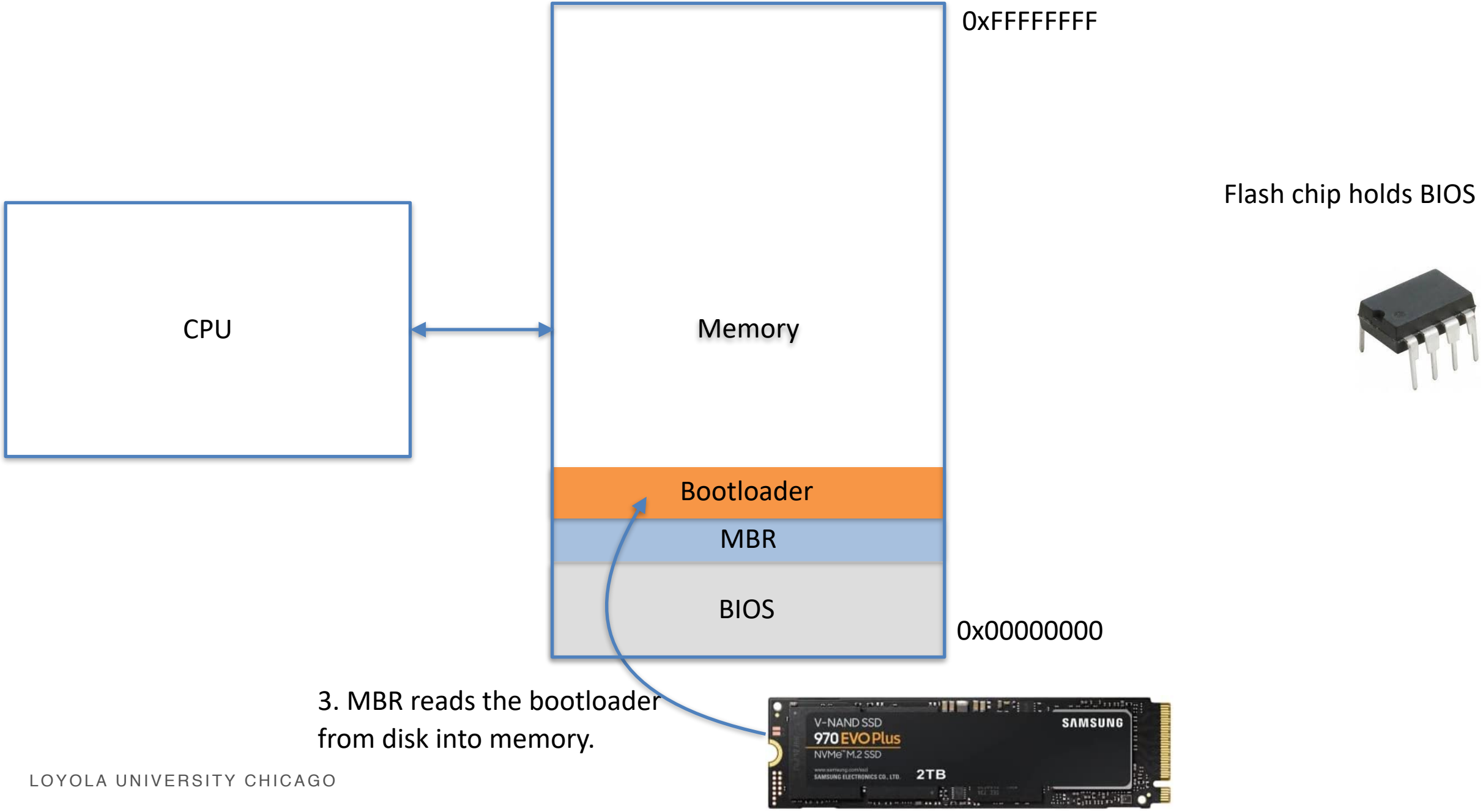
Can’t use for variable storage


0xFFFFFFFF
CPU
Memory
BIOS
0x00000000
Flash chip holds BIOS
1. Hardware reads BIOS from flash chip into DRAM.
V-NAND SSD
970 EVO Plus
NVMe M.2 SSD
2TB
SAMSUNG

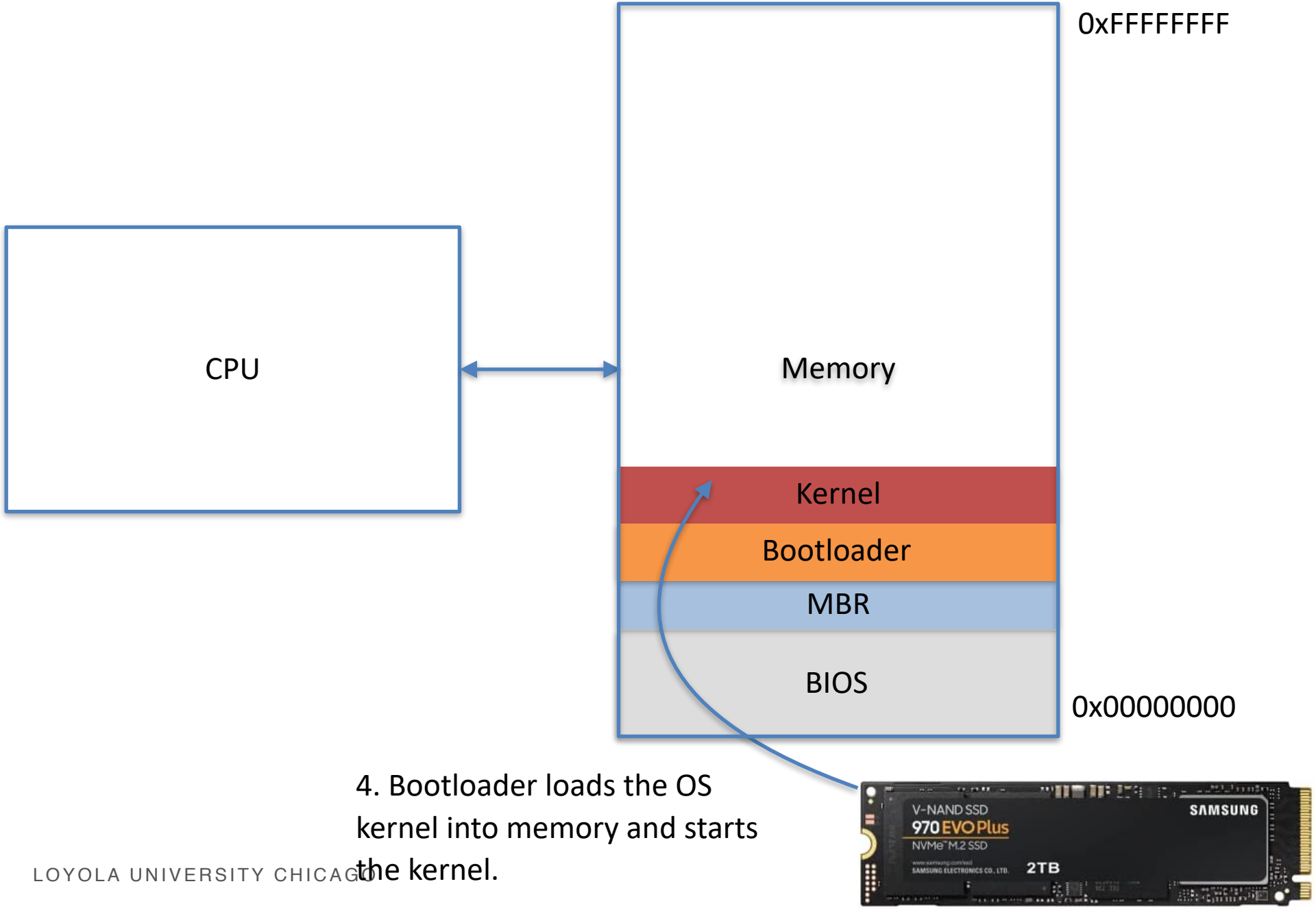



Flash chip holds BIOS

2. BIOS reads MBR from disk into DRAM

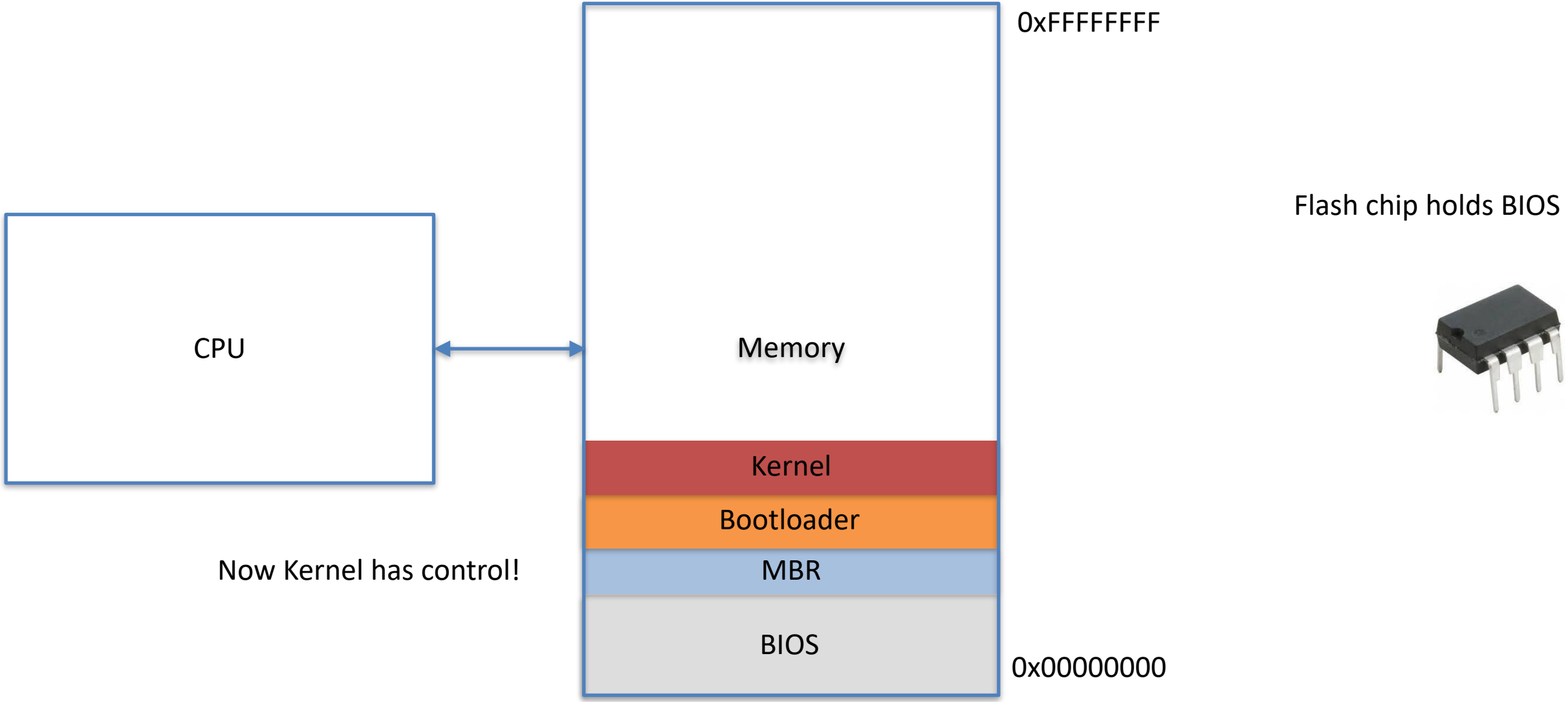



Flash chip holds BIOS



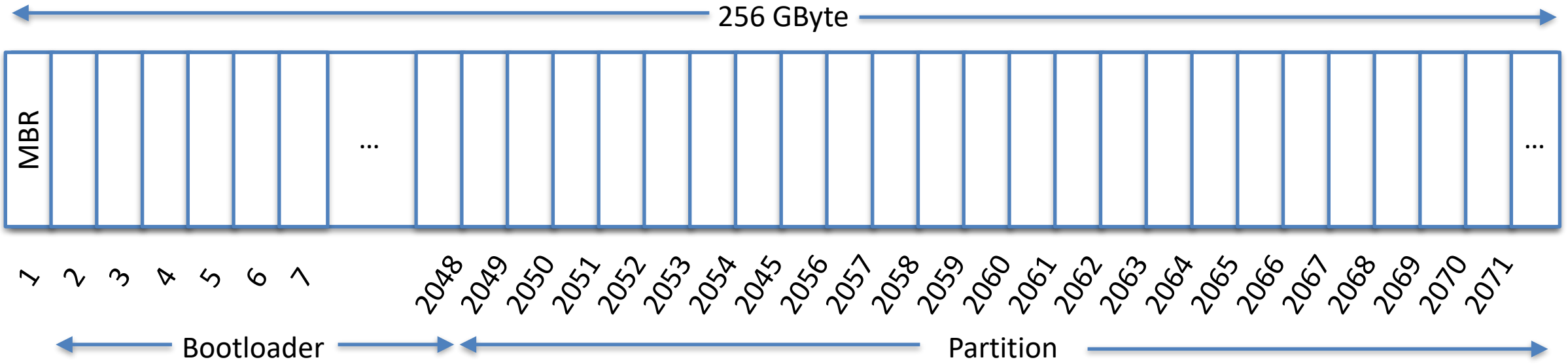

Flash chip holds BIOS

4. Bootloader loads the OS kernel into memory and starts the kernel.




Flash chip holds BIOS

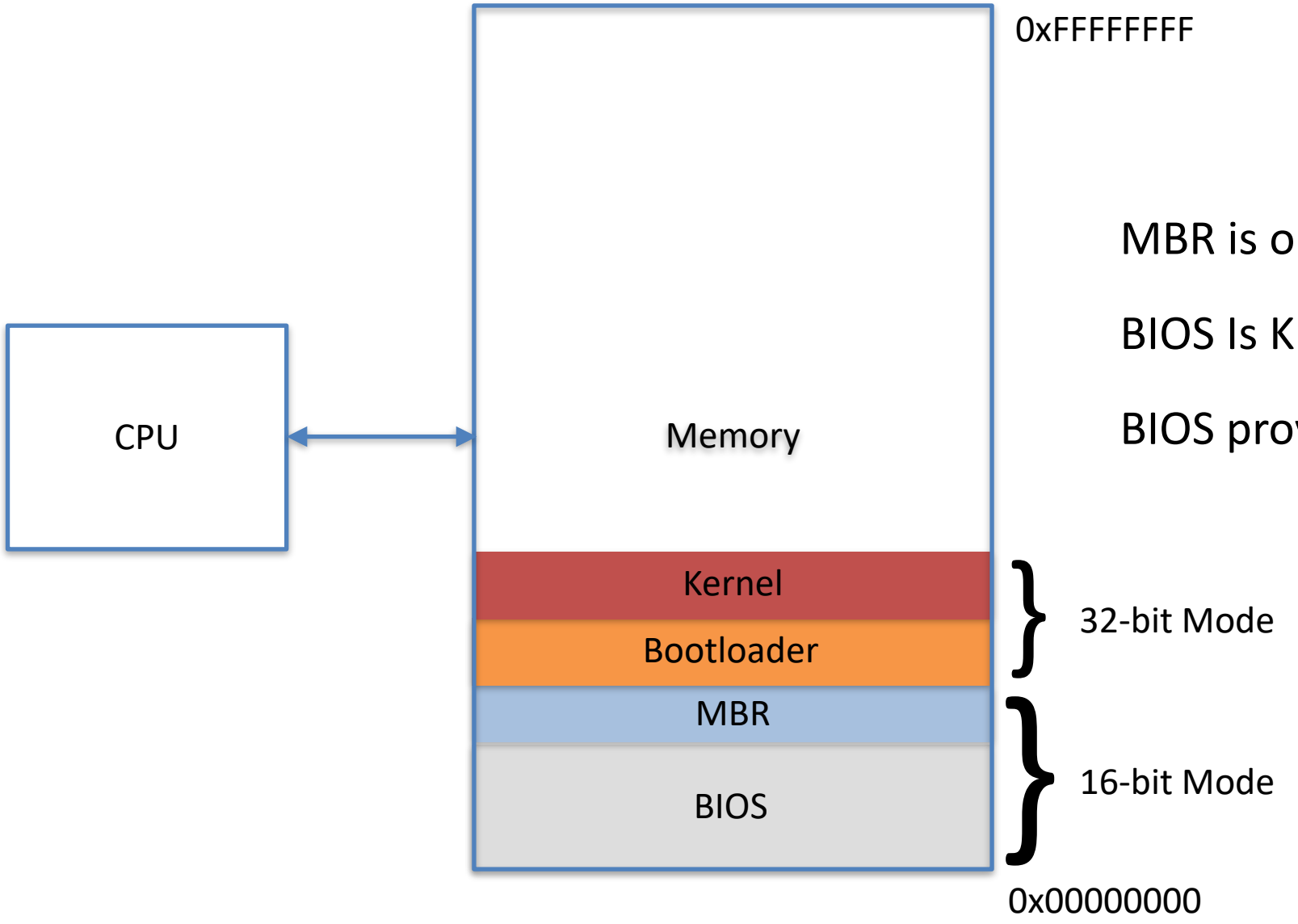



Disk is divided into 512-byte sectors  256 GByte $\times \frac{\text{Sector}}{512 \text{ Bytes}} =$ 2,097,152 Sectors

First 2048 sectors (1 Mbyte) store bootloader

# WRITING AN MBR




MBR is only 512 bytes!

BIOS Is Kinda Like a Set of Drivers for MBR

BIOS provides functions for the MBR to call.

# Normal Function Calls

**FUNCTION ADDRESS IS SPECIFIED BY THE SOFTWARE.**

```
void main() {
  printf("hello\n");
}
```

```
main:
  push hello_str
  call printf
  add sp,2
  ret

hello_str: db "hello", 0xa, 0xd, 0
```

1. Push function args onto stack.
2. Call function.
3. Clean up the stack.

In step 2, **we specify the address** of the function we want to call.

We can do this if `printf()` is a part of our program because the compiler knows its address.

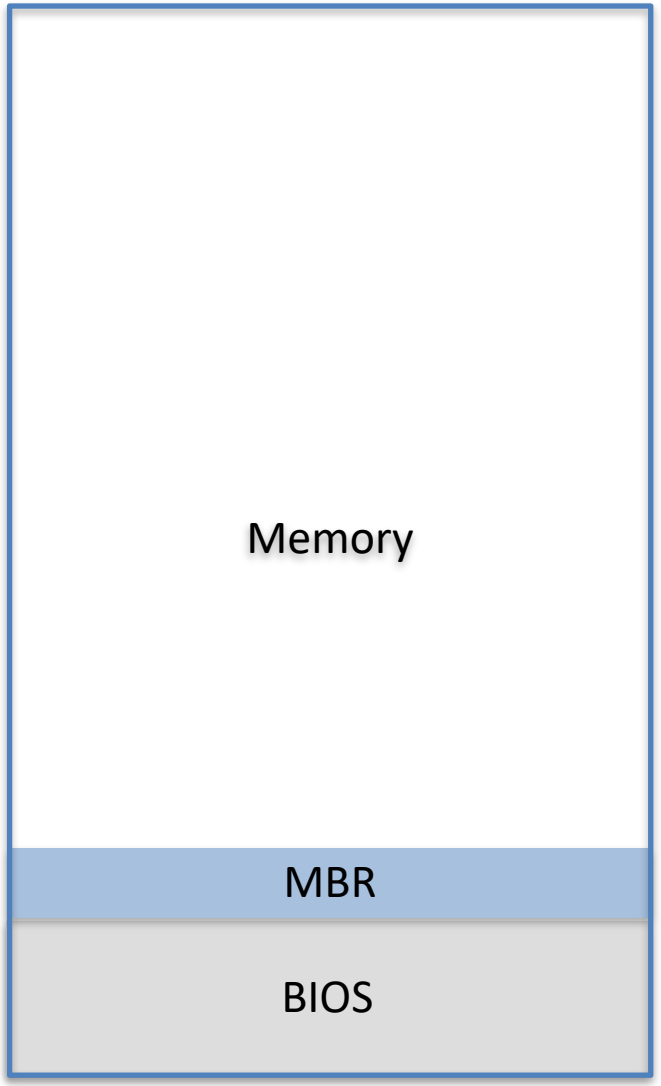
# Calls to the BIOS

**WE DON'T KNOW THE FUNCTION ADDRESS IN THE BIOS.**

The MBR and the BIOS are separate programs written by different companies.

MBR wants to call functions in the BIOS, but...

The compiler of the MBR does not know the addresses of functions in the BIOS.



# Calls to the BIOS

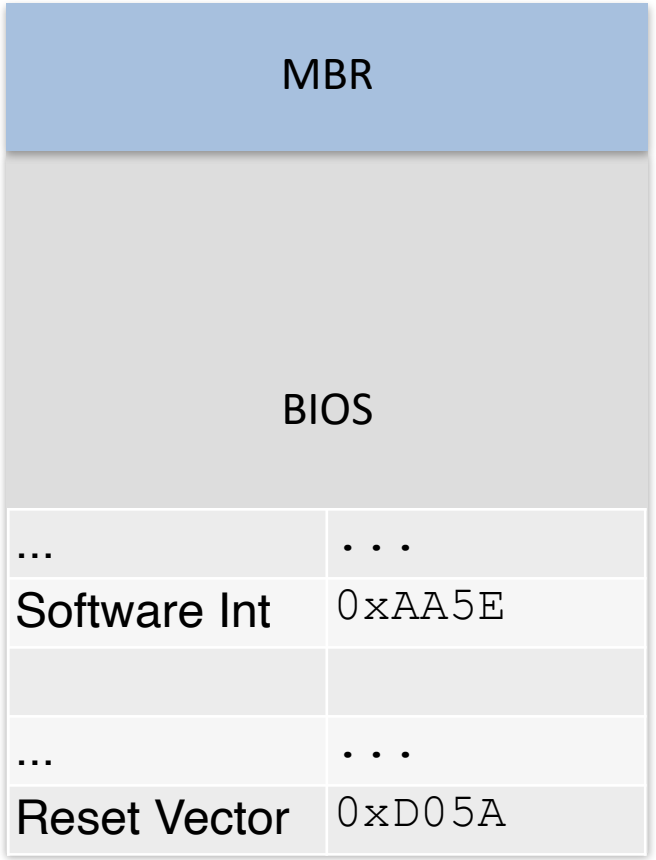
**WE DON'T KNOW THE FUNCTION ADDRESS IN THE BIOS.**

An **interrupt vector table** located at address 0 in memory holds the addresses of functions in the BIOS.

The `int` instruction executes a software interrupt, which calls one of the functions in the vector table.

The int instruction needs:
1. A function code (to tell BIOS what to do)
2. A parameter list

Parameters to BIOS calls passed in the CPU registers (not stack)



# BIOS Function Calls

**USE `INT` INSTRUCTION WITH PARAMETERS IN REGISTERS.**

```
void main() {
  printf("hello\n");
}
```

```
main:
  mov ah, 0x0e
  xor bh,bh
  mov bl,0
  mov bp,hello_str
  mov cx,5
  mov dx,0
  int 0x10
  ret

hello_str: db "hello", 0xa, 0xd, 0
```

1. Set up parameters in registers.

2. Call BIOS.

# The only thing a computer knows how to do is execute instructions.

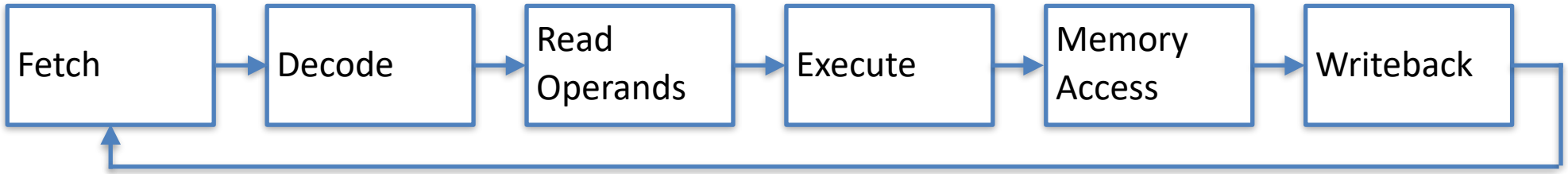
```
if( a < 5 ) {
  b += a;
  a++;
}
```

```
cmp ax,5
jge .not_less_than

add bx,ax

inc ax

.not_less_than:
   ...
```

# Kinds of instructions

- Arithmetic
  - Add, subtract, multiply, divide
- Logic
  - AND, OR, NOT, XOR
- Shifts
  - Left shift, right shift, rotate, etc.
- Control
  - Branch/Jump
  - Procedure calls
- Memory Accesses
  - Load/store

# The only thing a computer knows how to do is execute instructions.



# RASPBERRY PI BOOT PROCESS

CPU

GPU

Memory

CPU

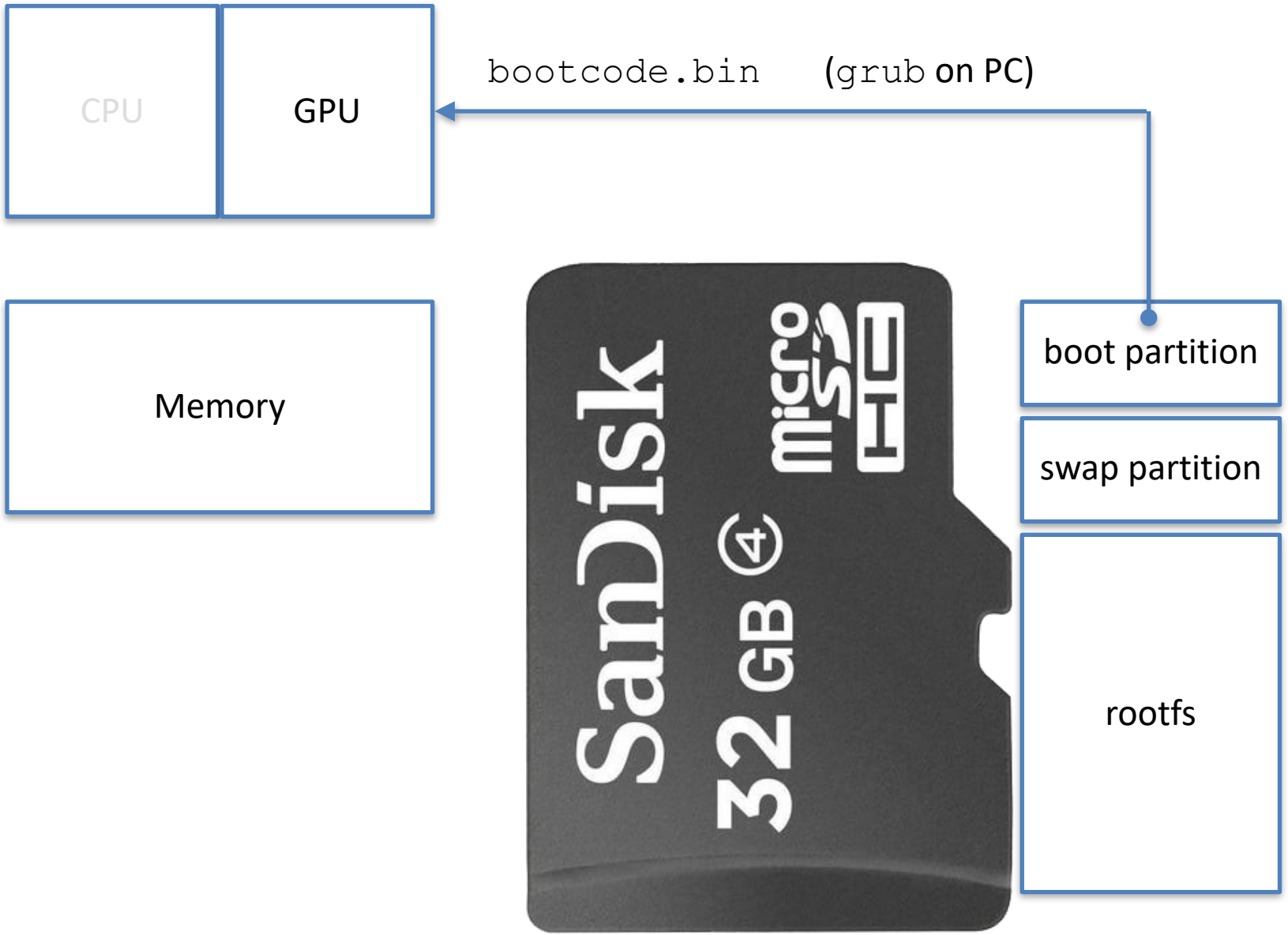
GPU

Memory


boot partition

swap partition

rootfs


CPU
GPU
bootcode.bin (grub on PC)
Memory
SanDisk
32 GB
boot partition
swap partition
rootfs

CPU

GPU

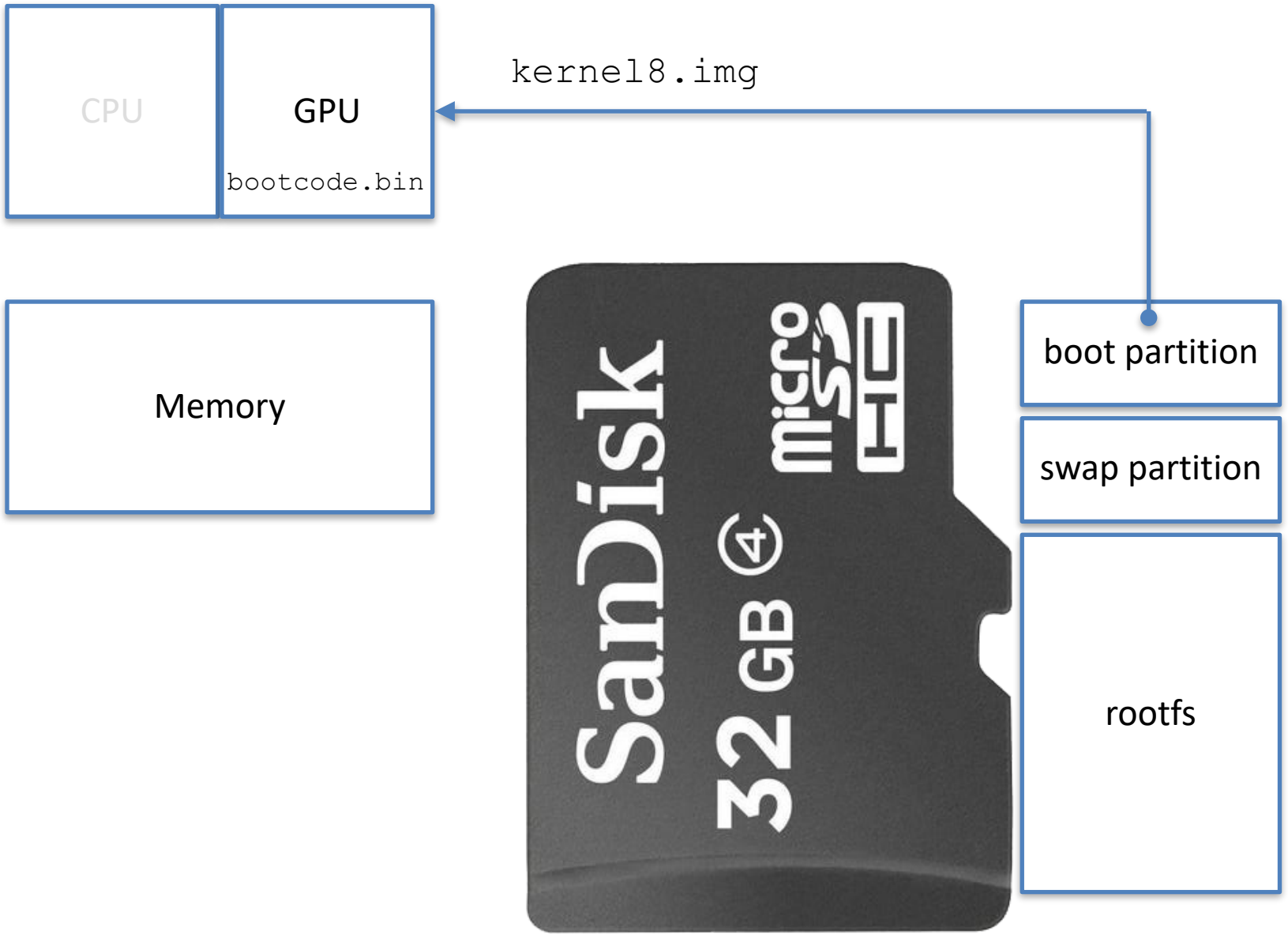
`bootcode.bin`

Memory


boot partition

swap partition

rootfs


CPU
GPU
bootcode.bin
kernel8.img
Memory
boot partition
swap partition
rootfs
SanDisk
32 GB
micro SD HC

CPU

`kernel8.img`

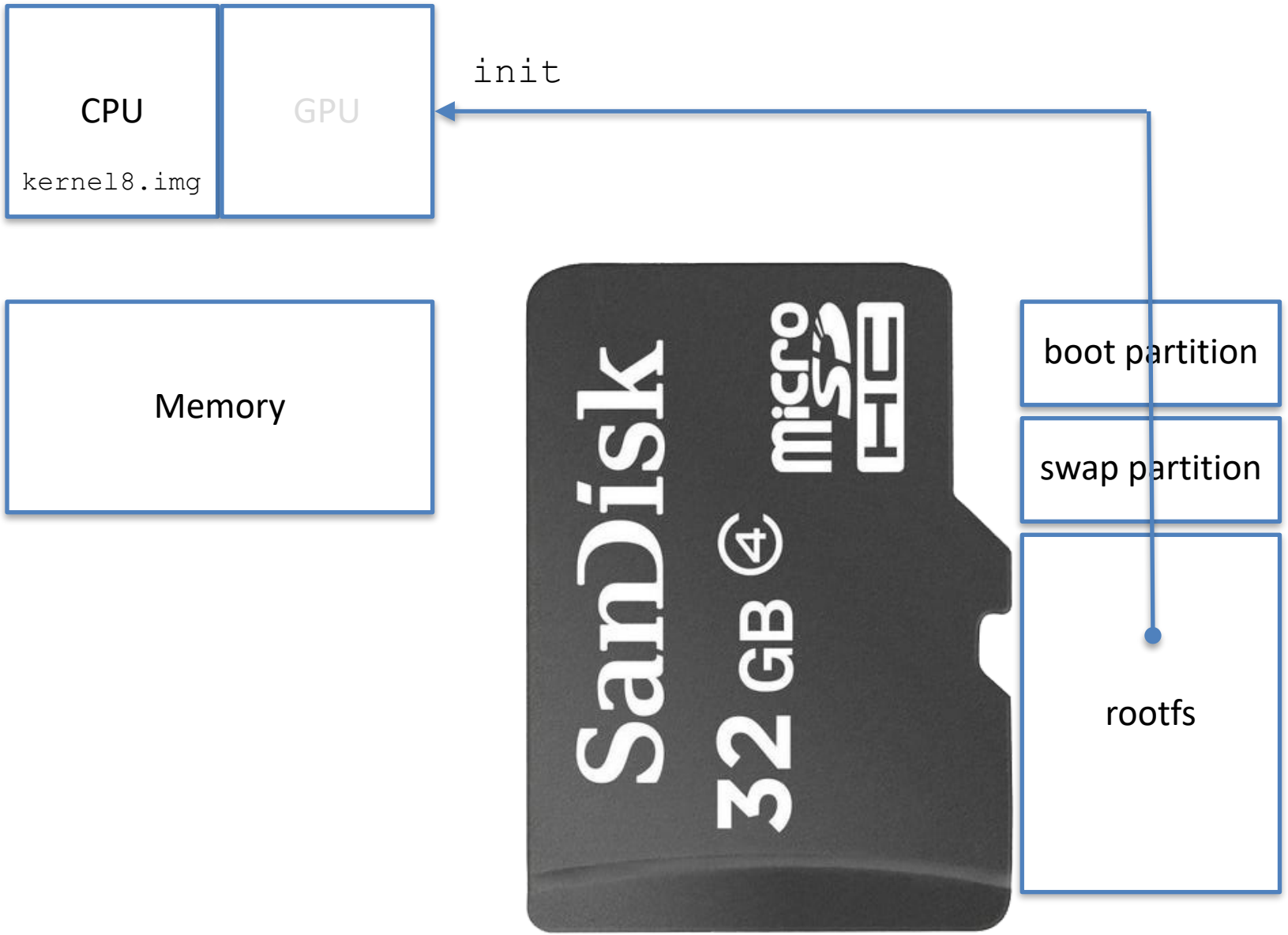
GPU

Memory


boot partition

swap partition

rootfs


CPU
kernel8.img
GPU
init
Memory
SanDisk
32 GB
boot partition
swap partition
rootfs