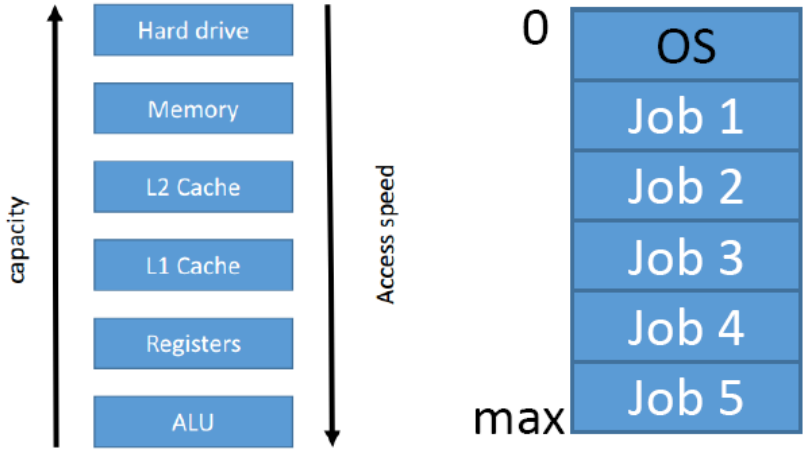
CSCI 509

OPERATING SYSTEMS INTERNALS

MULTIPROGRAMMING

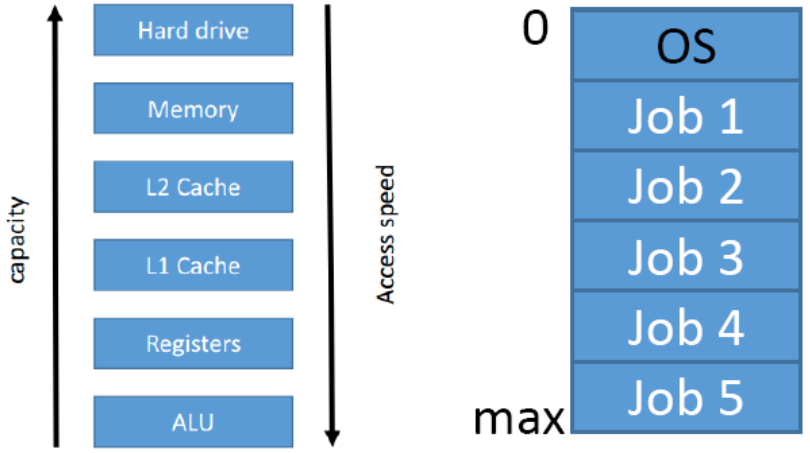
Worksheet Q1

- Q: What sort of stuff could go wrong when running multiple programs?



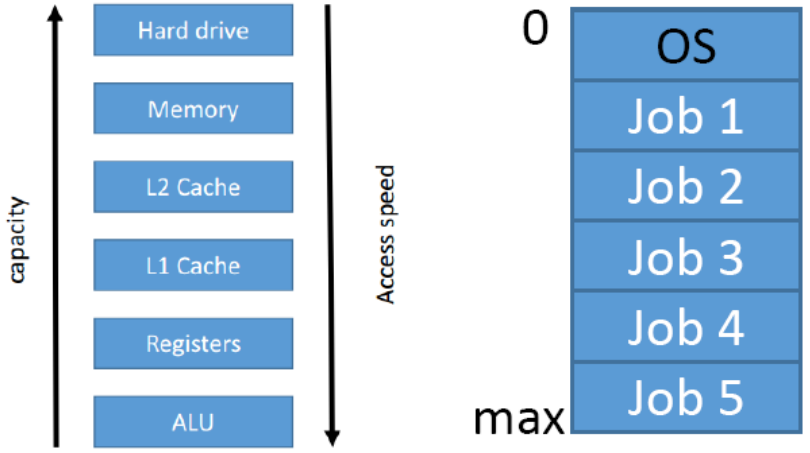
MULTIPROGRAMMING

- Q: What sort of stuff could go wrong when running multiple programs?
- Performance
  - Not enough memory for all jobs
  - Jobs taking too much or too little share of CPU
  - Multiple Jobs competing for same resource.



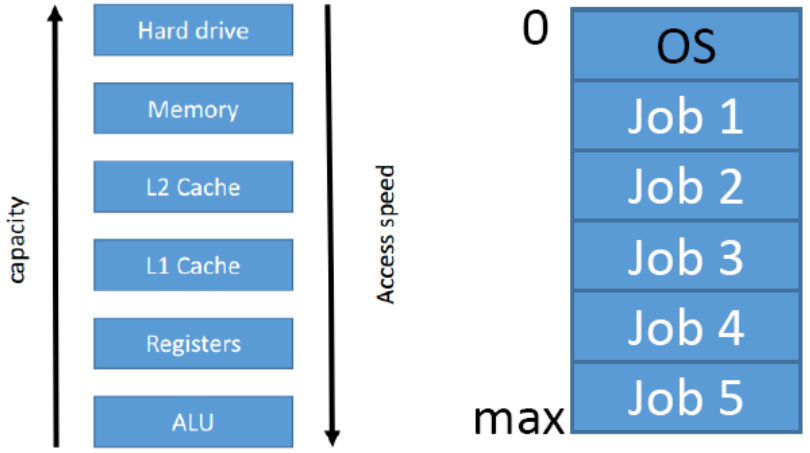
MULTIPROGRAMMING

- Q: What sort of stuff could go wrong when running multiple programs?
- Performance
  - Not enough memory for all jobs
  - Jobs taking too much or too little share of CPU
  - Multiple Jobs competing for same resource.
- Security:
  - Jobs modifying shared memory or file.



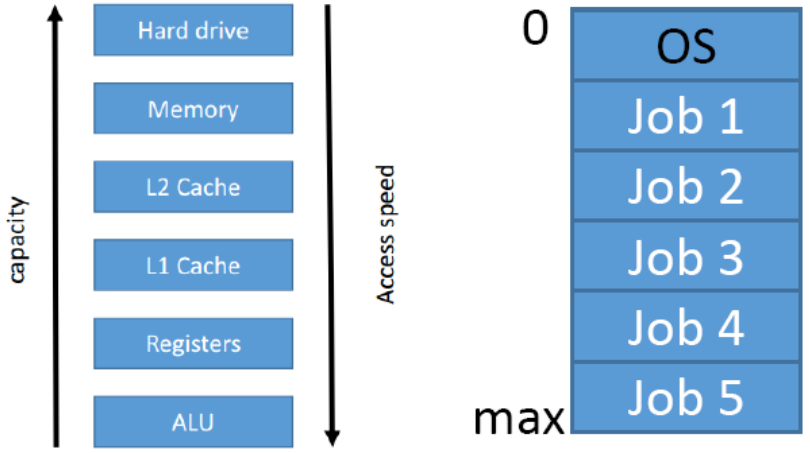
MULTIPROGRAMMING

- Q: What sort of stuff could go wrong when running multiple programs?
- Performance
  - Not enough memory for all jobs
  - Jobs taking too much or too little share of CPU
  - Multiple Jobs competing for same resource.
- Security:
  - Jobs modifying shared memory or file.
- Kernel / OS must manage all these issues.
- Another name for a 'job' is 'process'.



MULTIPROGRAMMING

- Q: What sort of stuff could go wrong when running multiple programs?
- Performance
  - Not enough memory for all jobs
  - Jobs taking too much or too little share of CPU
  - Multiple Jobs competing for same resource.
- Security:
  - Jobs modifying shared memory or file.
- Kernel / OS must manage all these issues.
- Another name for a 'job' is 'process'.
- Q:What is the difference between a program and a process?



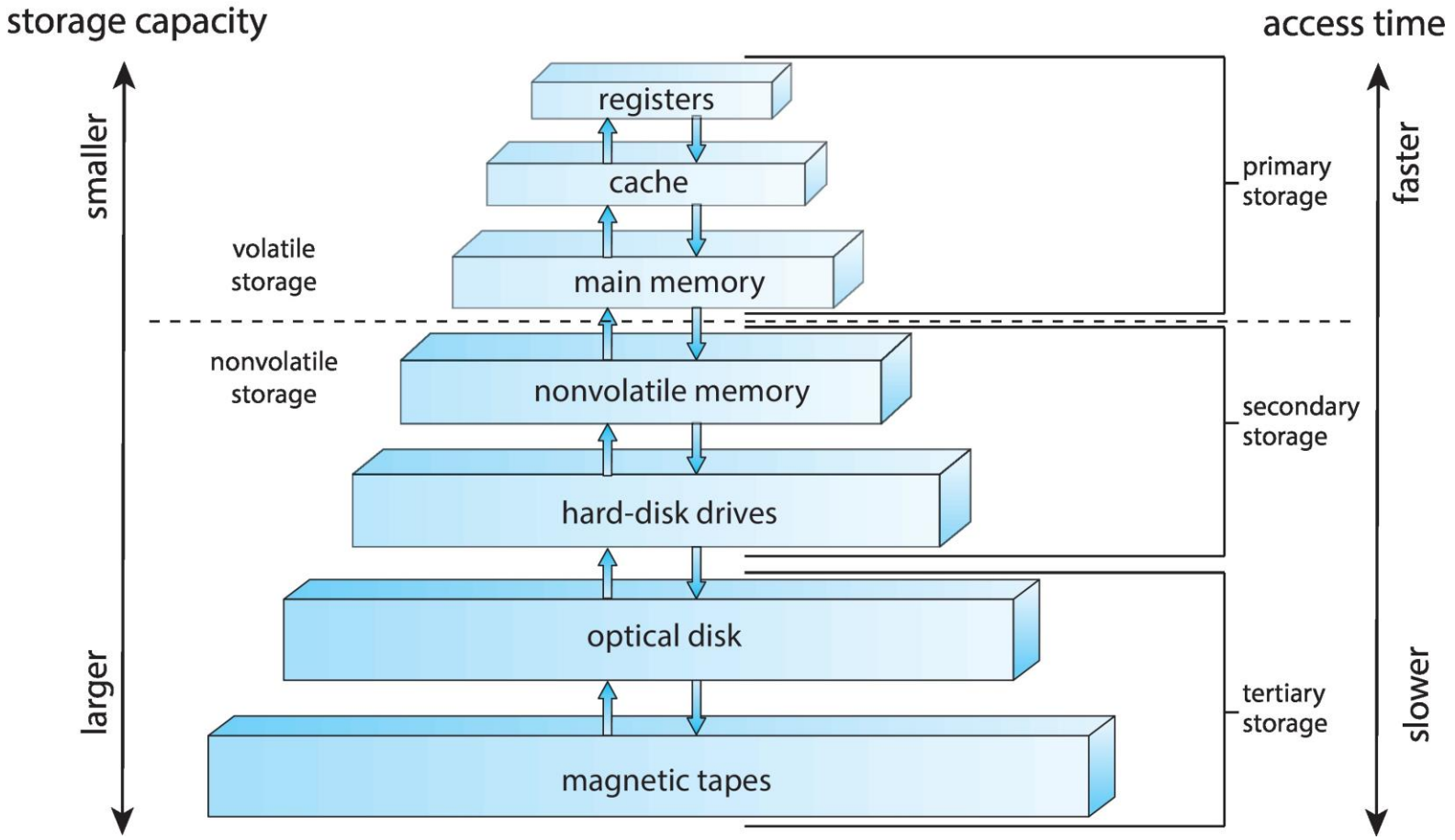
PROGRAM VS PROCESS

- A process is a program in execution. It is a unit of work within the system.
- Program is a ***passive entity***, process is an ***active entity***.

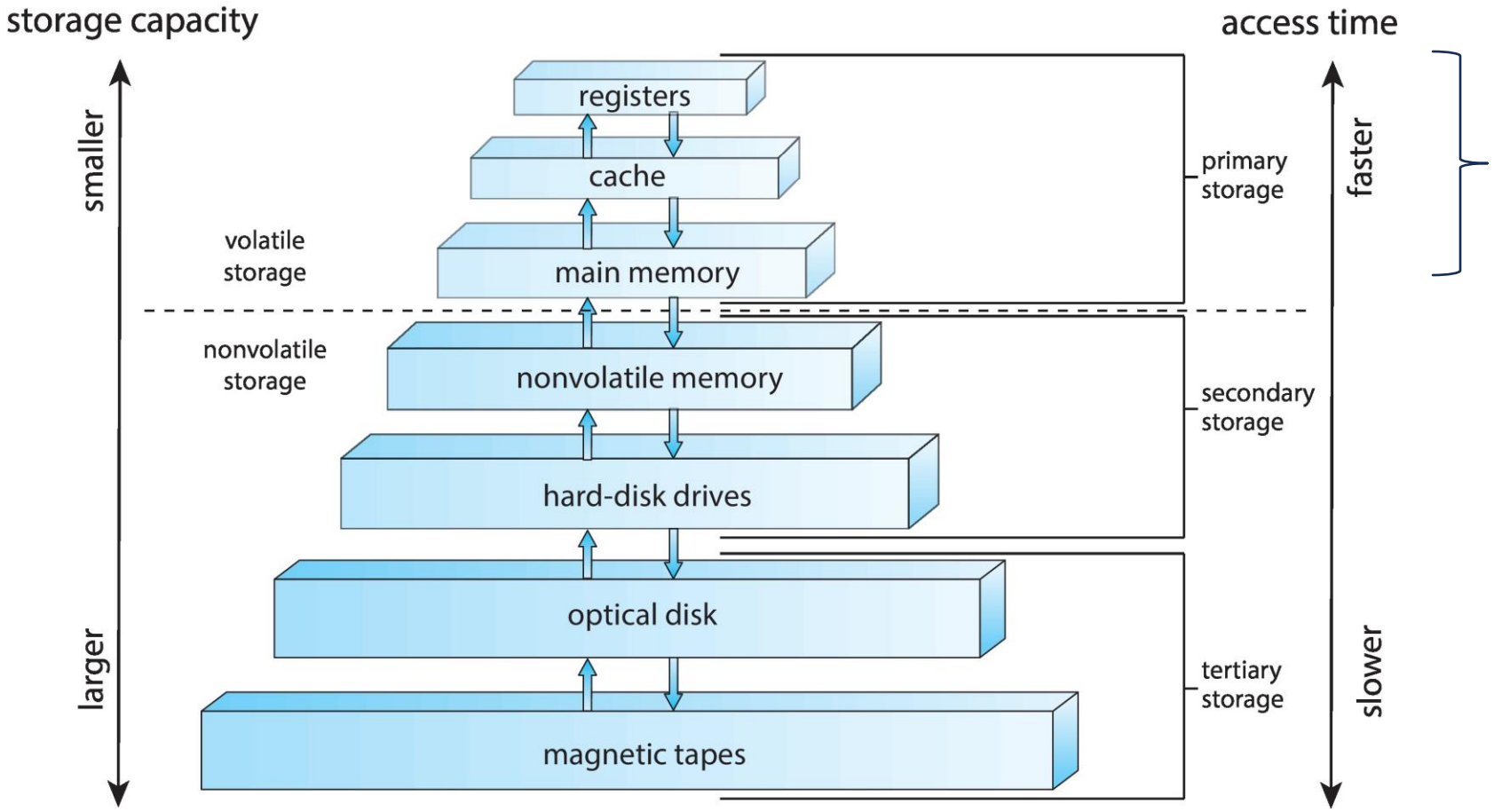
PROGRAM VS PROCESS

- A process is a program in execution. It is a unit of work within the system.
- Program is a ***passive entity***, process is an ***active entity***.
- Analogy:
  - If a program is the script of a play, the process is the play being performed.
  - You can have multiple processes running the same program.

MEMORY MANAGEMENT: MEMORY HIERARCHY

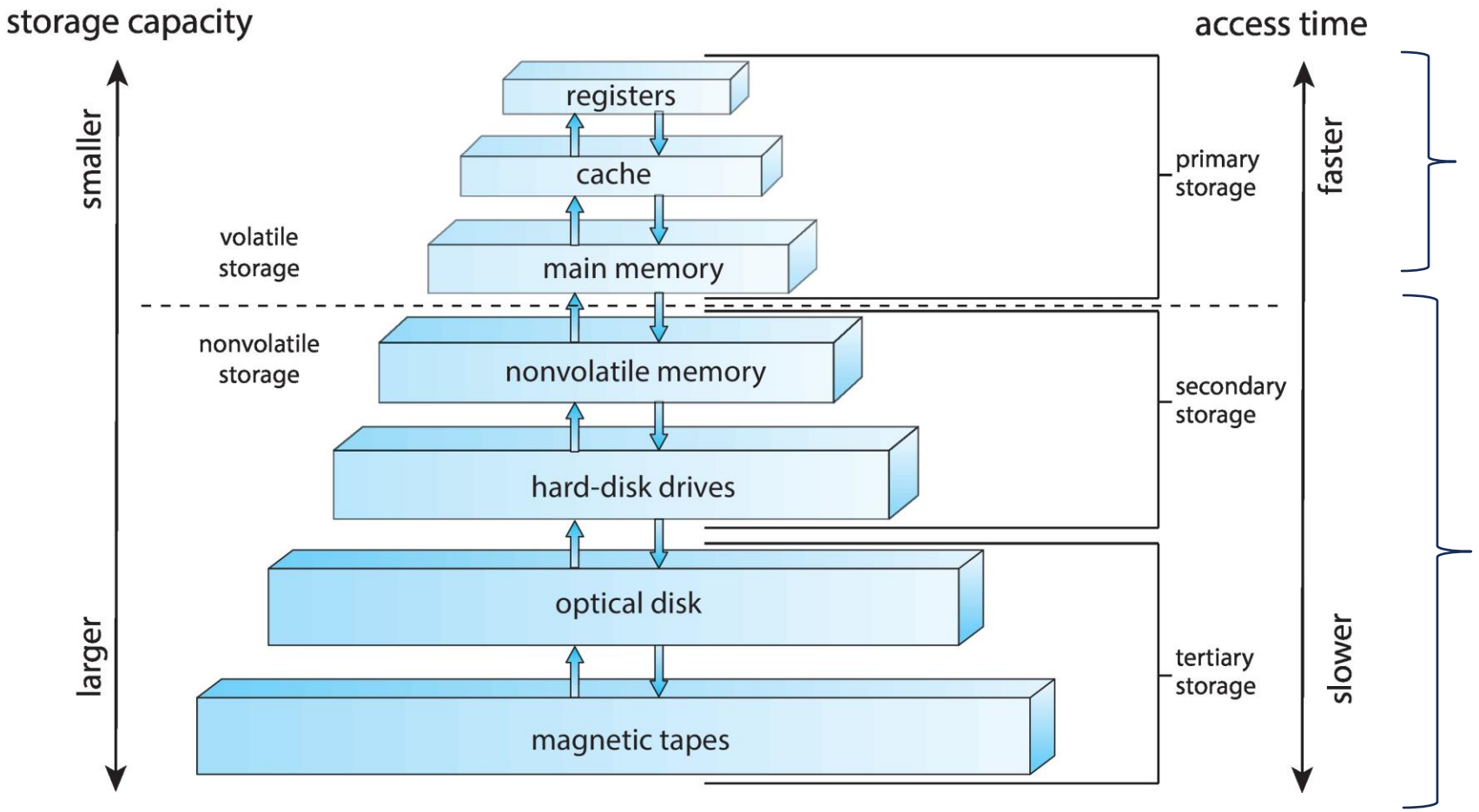


MEMORY MANAGEMENT: MEMORY HIERARCHY



- Fast
- Expensive
- Volatile: State is lost on power disconnect.

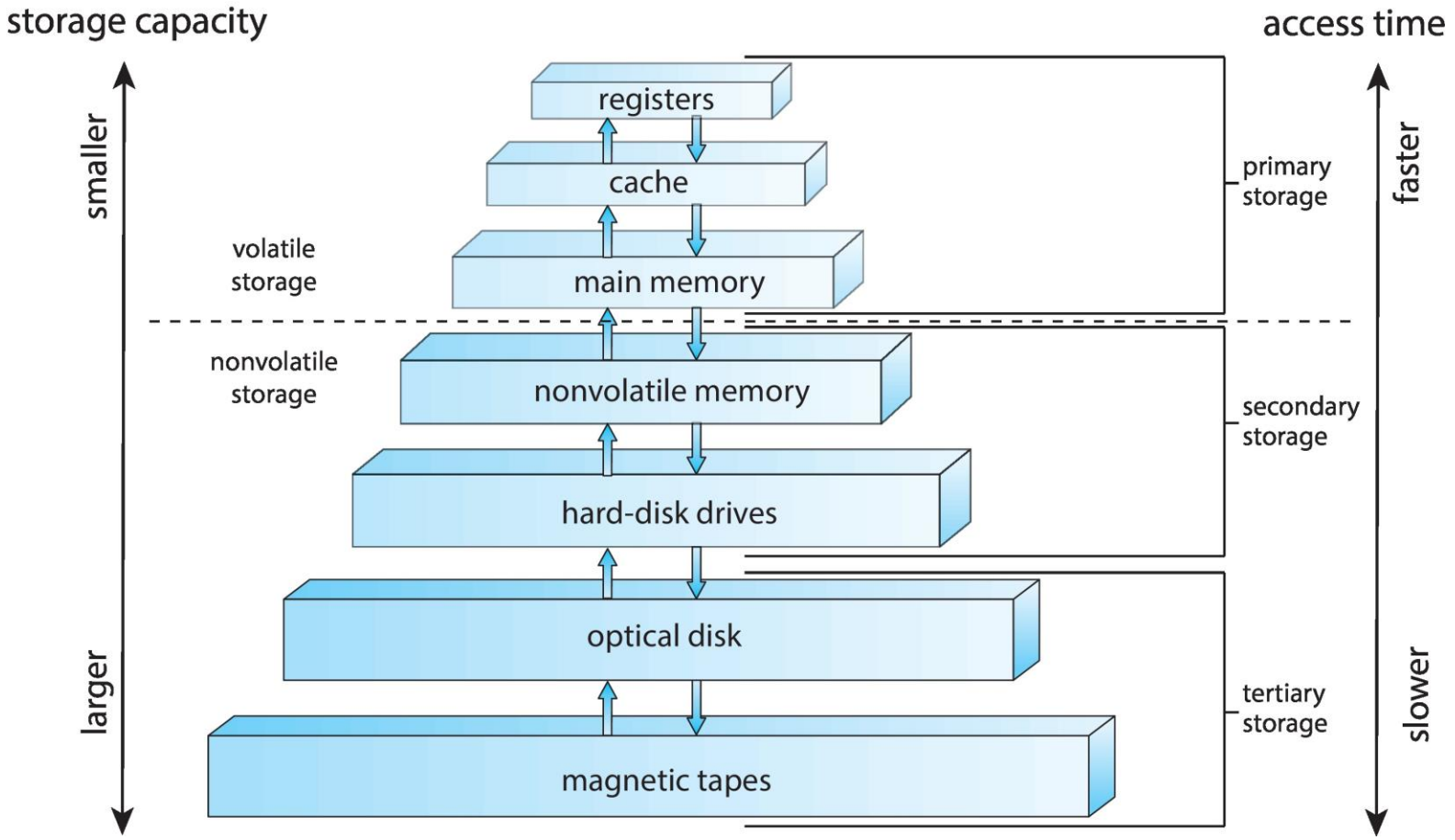
MEMORY MANAGEMENT: MEMORY HIERARCHY



- Fast
- Expensive
- Volatile: State is lost on power disconnect.

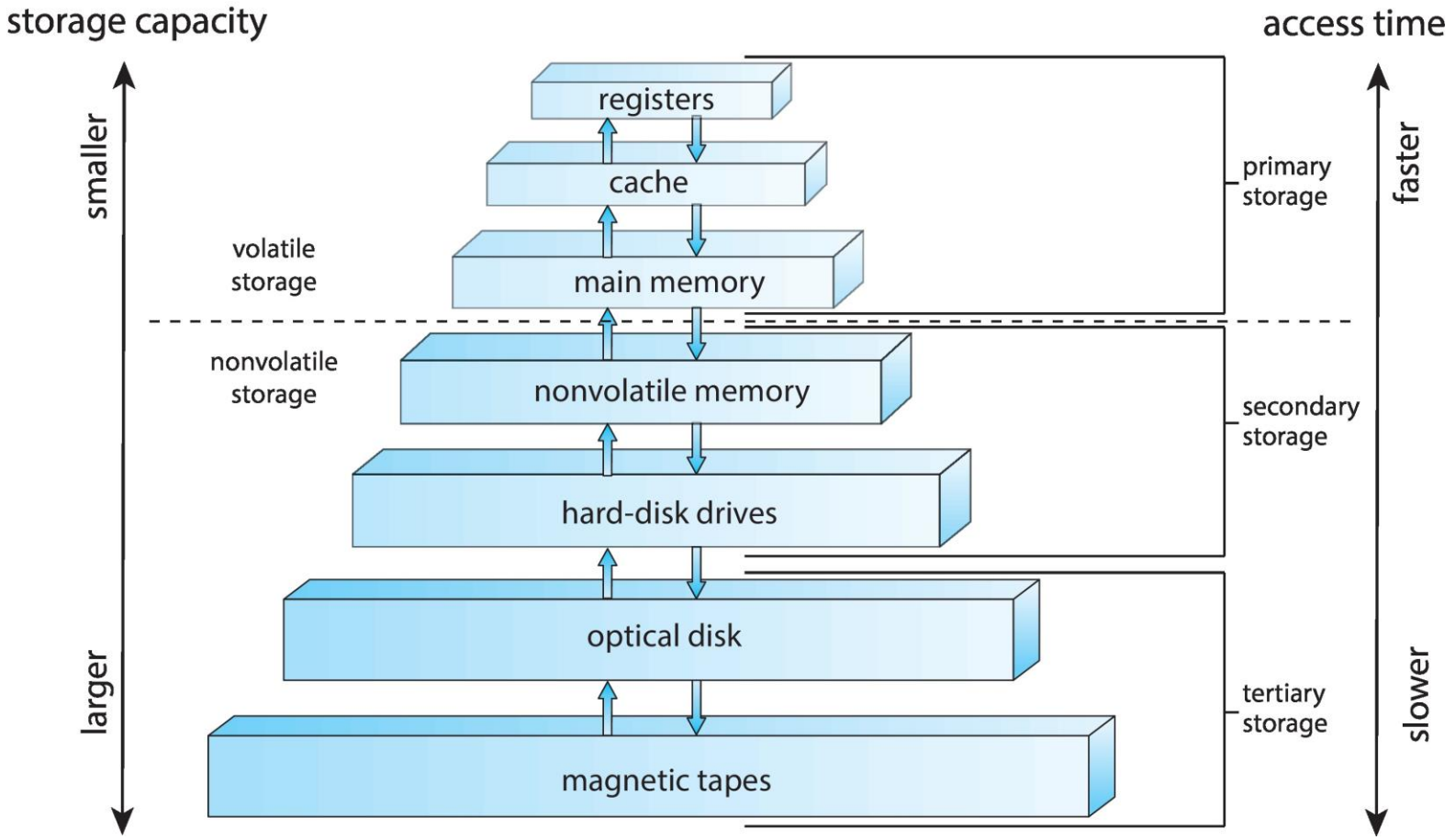
- Slow
- Non-volatile
- Cheap

MEMORY MANAGEMENT: MEMORY HIERARCHY



Q: Why does this work?

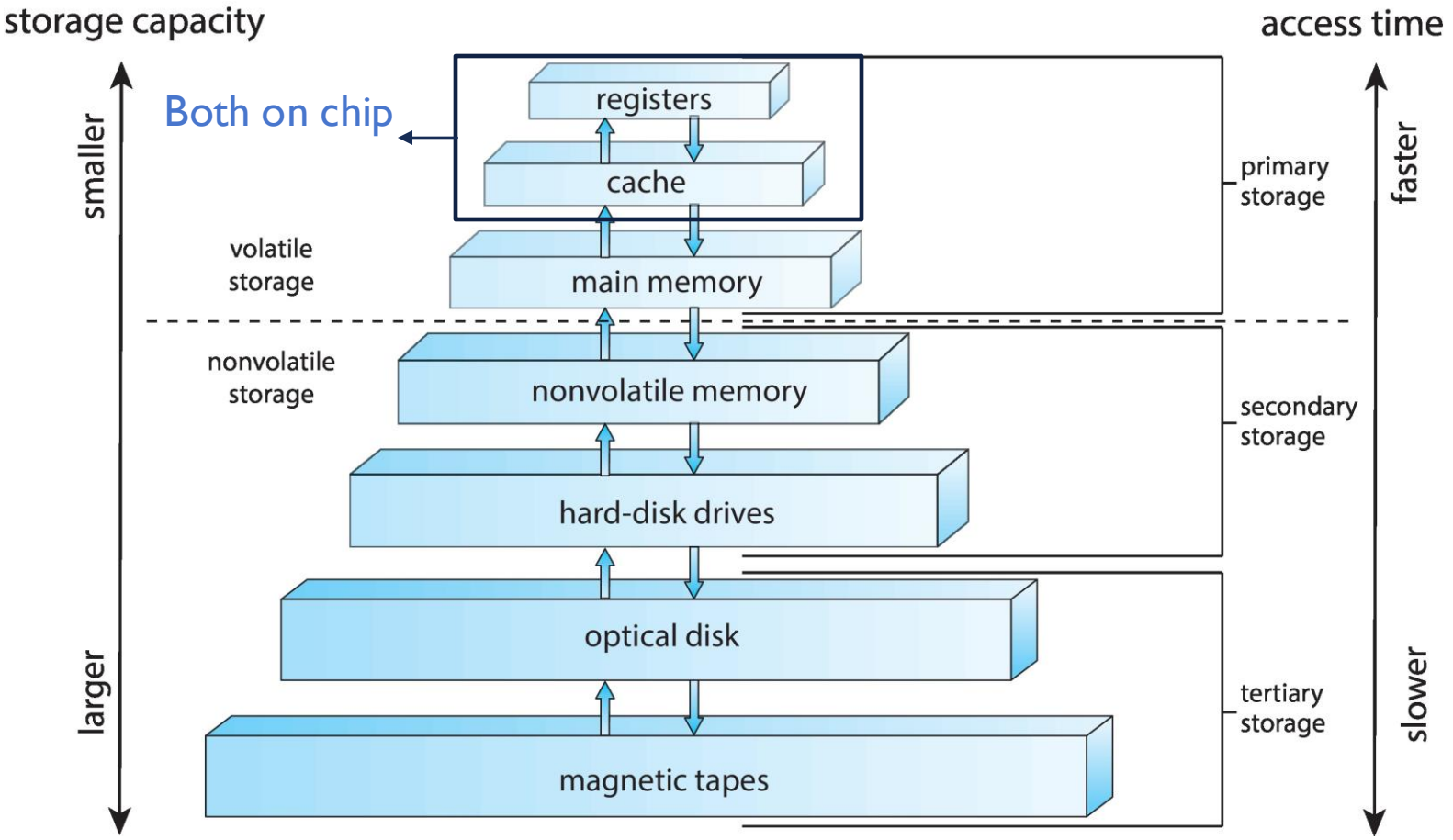
MEMORY MANAGEMENT: MEMORY HIERARCHY



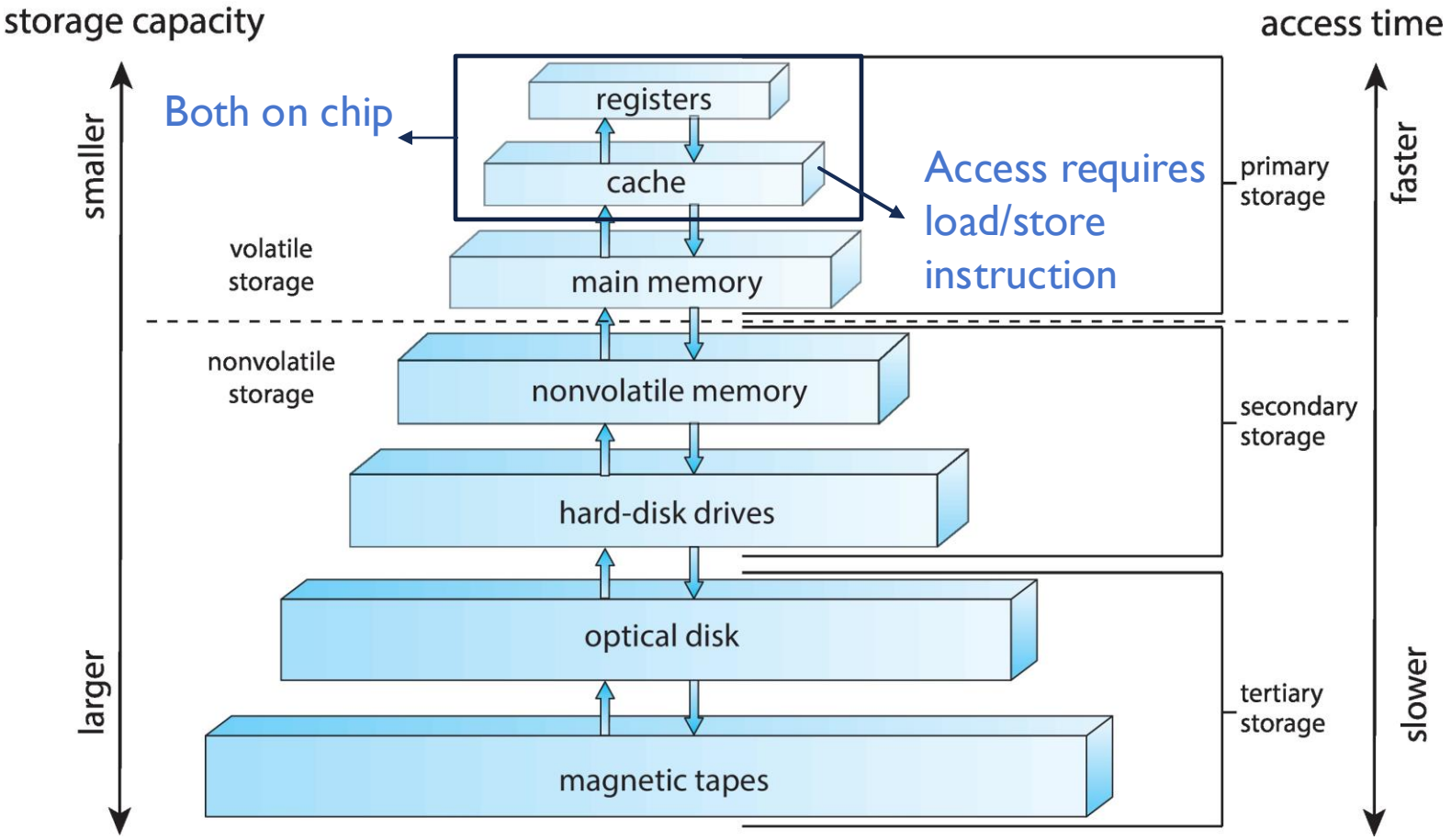
Q: Why does this work?

- CPU only operates on few words at a time …
- It never needs access to the full memory or any significant portion of it.

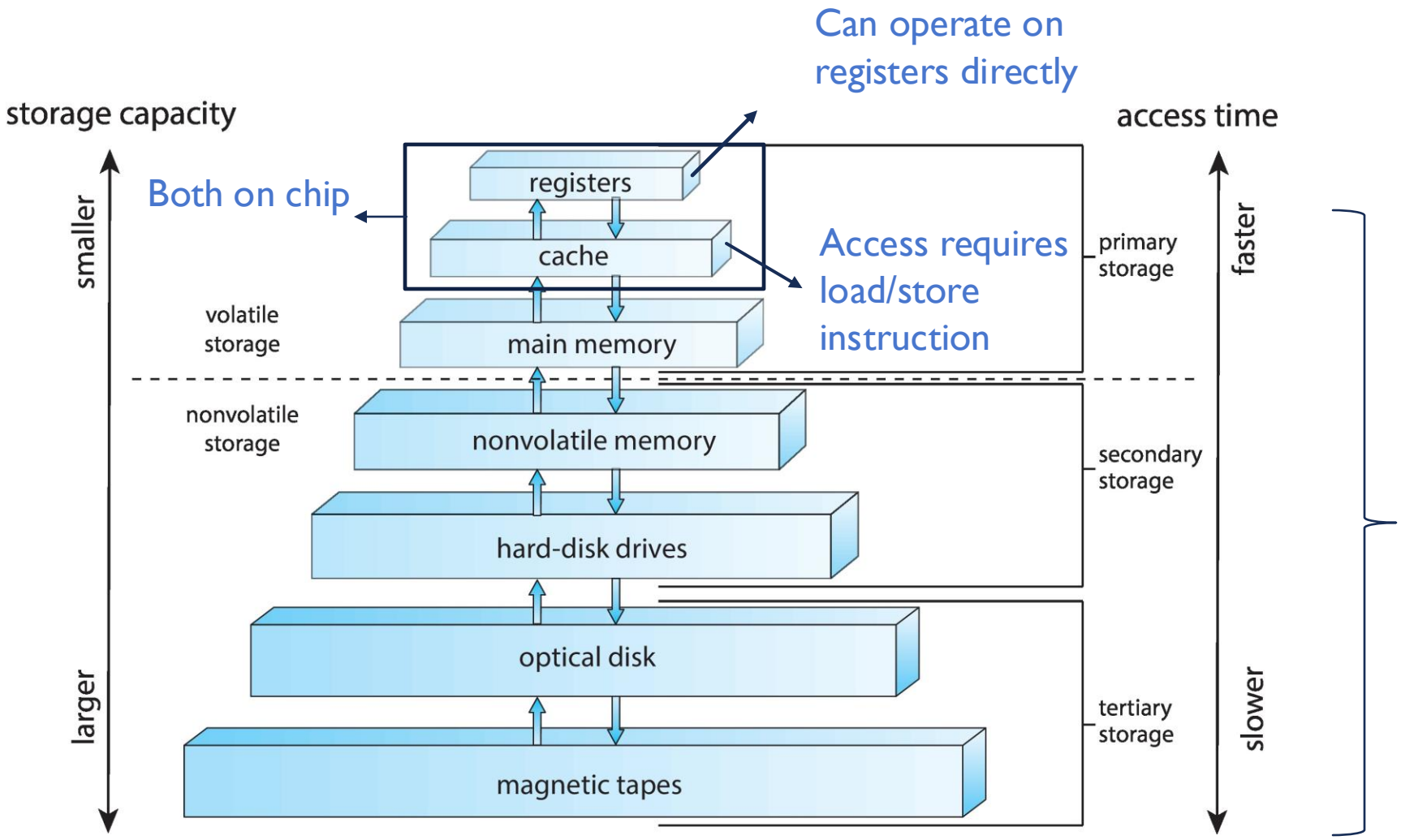
MEMORY HIERARCHY



MEMORY HIERARCHY



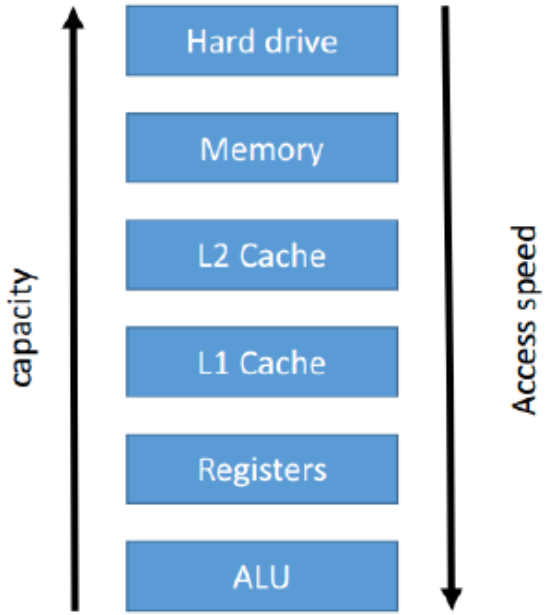
MEMORY HIERARCHY

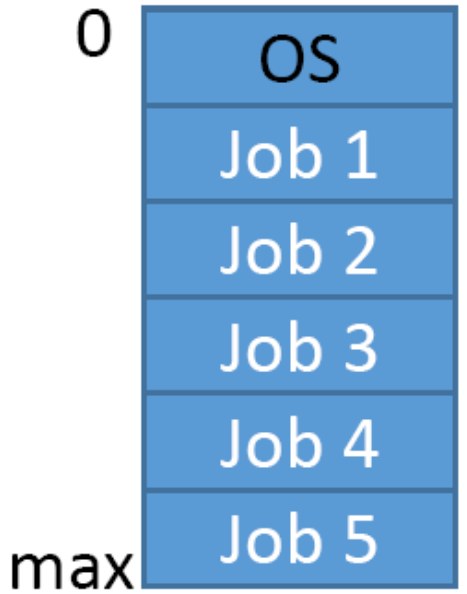


All is handled by the OS
The program doesn't need to worry about registers, cache, …

MEMORY MANAGEMENT

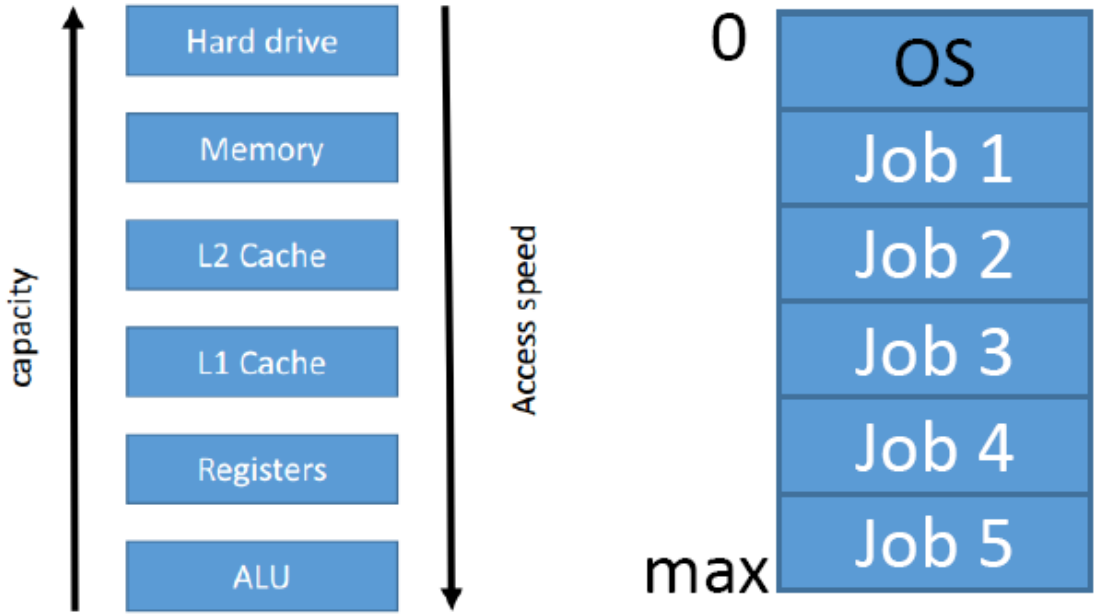
- How to divide limited memory across multiple processes.
- Not enough memory? Swap jobs to storage.





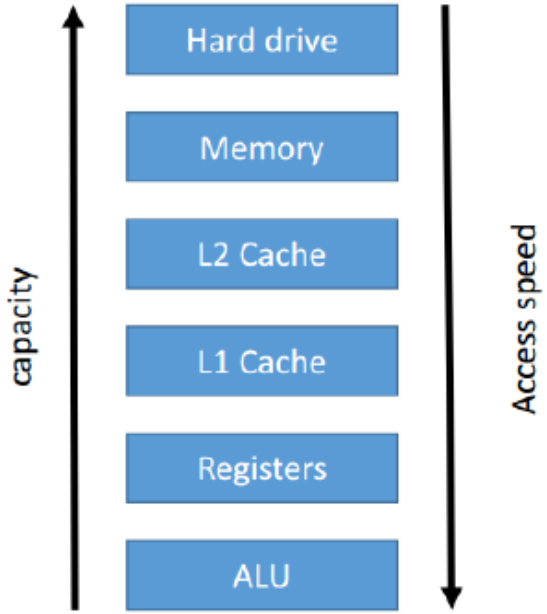
MEMORY MANAGEMENT

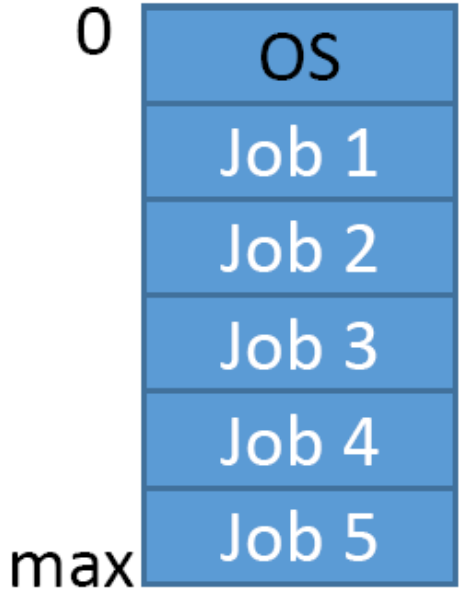
- How to divide limited memory across multiple processes.
- Not enough memory? Swap jobs to storage.
- Q:When a process is alive and running, where is it loaded?



MEMORY MANAGEMENT

- How to divide limited memory across multiple processes.
- Not enough memory? Swap jobs to storage.
- Q:When a process is alive and running, where is it loaded?
  - **A:** L1 Cache
  - **B:** L2 Cache
  - **C:** Main Memory
  - **D:** All of the above

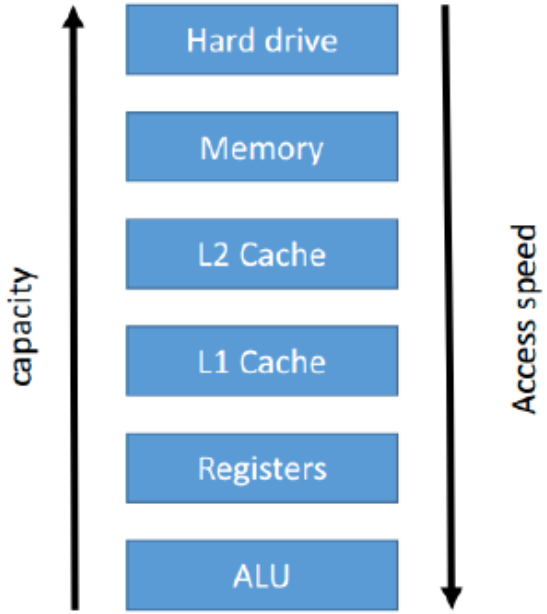


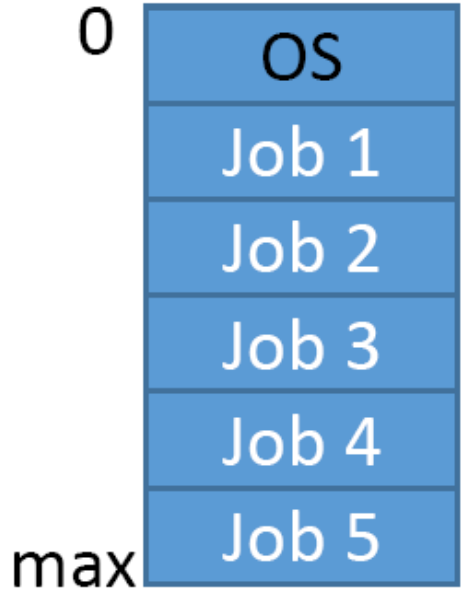


MEMORY MANAGEMENT

- How to divide limited memory across multiple processes.
- Not enough memory? Swap jobs to storage.
- Q:When a process is alive and running, where is it loaded?
  - **A:** L1 Cache
  - **B:** L2 Cache
  - **C:** Main Memory
  - **D:** All of the above

- Only a small portion is loaded in L1 cache.





PROTECTION

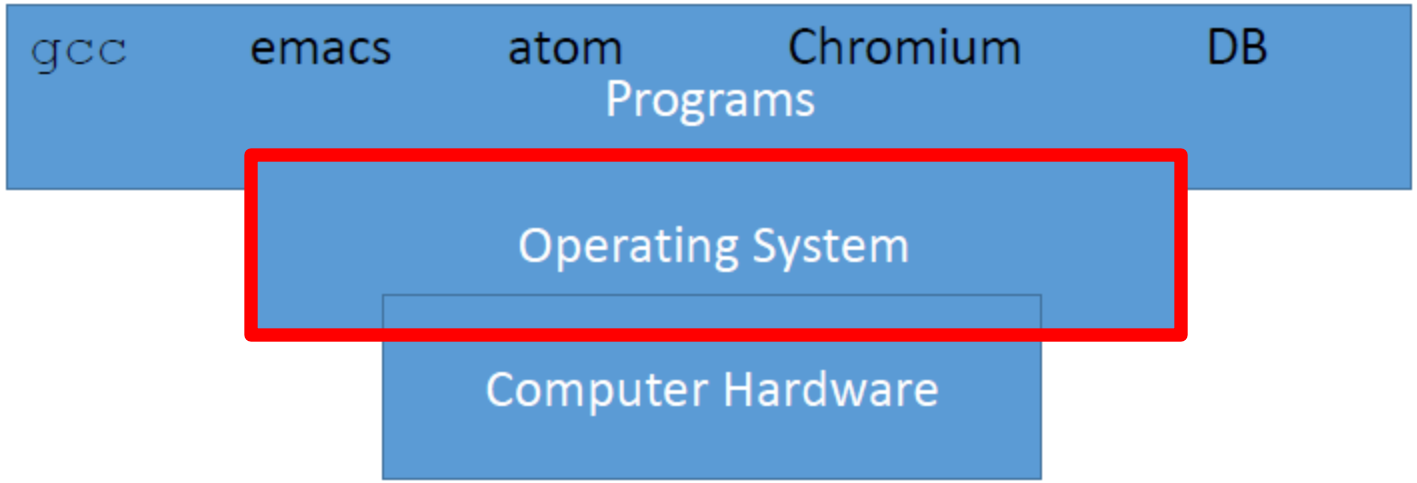
- **Protection** – any mechanism for controlling access of processes or users to resources defined by the OS
  - User identities (**user IDs**, security IDs) include name and associated number, one per user
  - User ID then associated with all files, processes of that user to determine access control
  - Group identifier (**group ID**) allows set of users to be defined and controls managed, then also associated with each process, file
  - **Privilege escalation** allows user to change to effective ID with more rights

SECURITY

- **Security** – defense of the system against internal and external attacks
  - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Can be handled by third party software.
- Protection Vs Security:
  - Protection: Managing access control among authentic users.
  - Security: Protecting the system from malicious software.

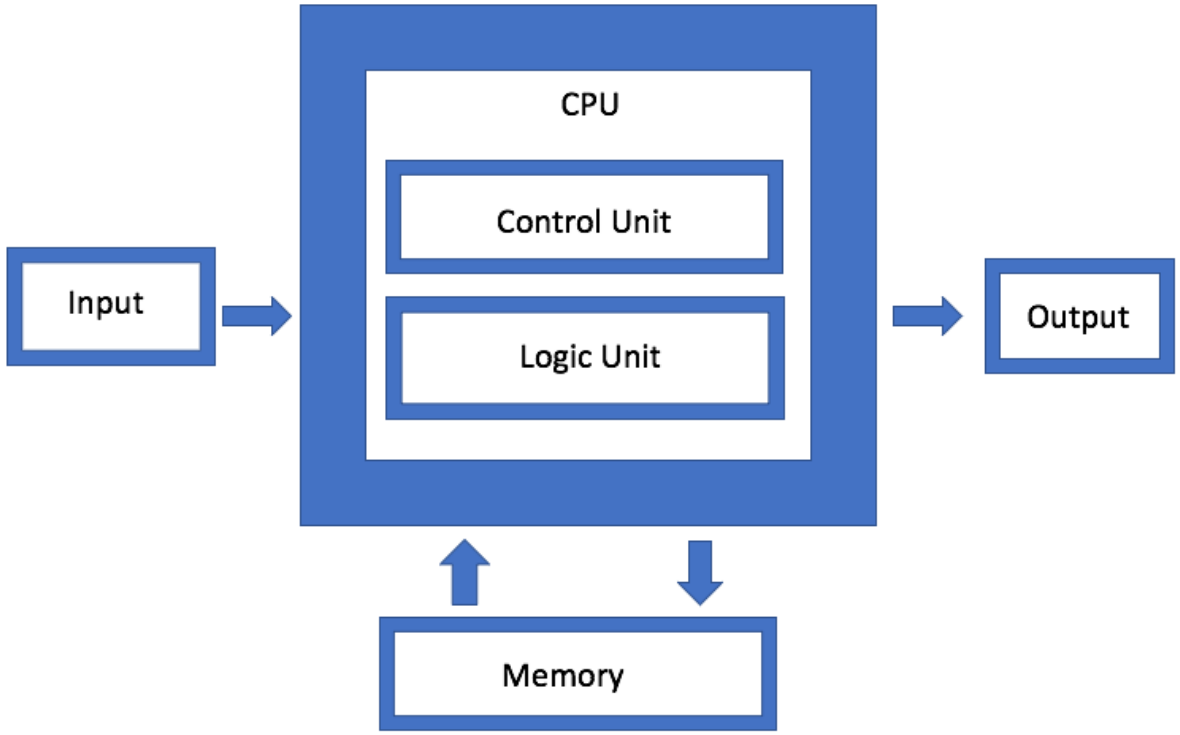
COMPUTER ARCHITECTURE REVIEW

- To understand OS better, we need to understand computer hardware.



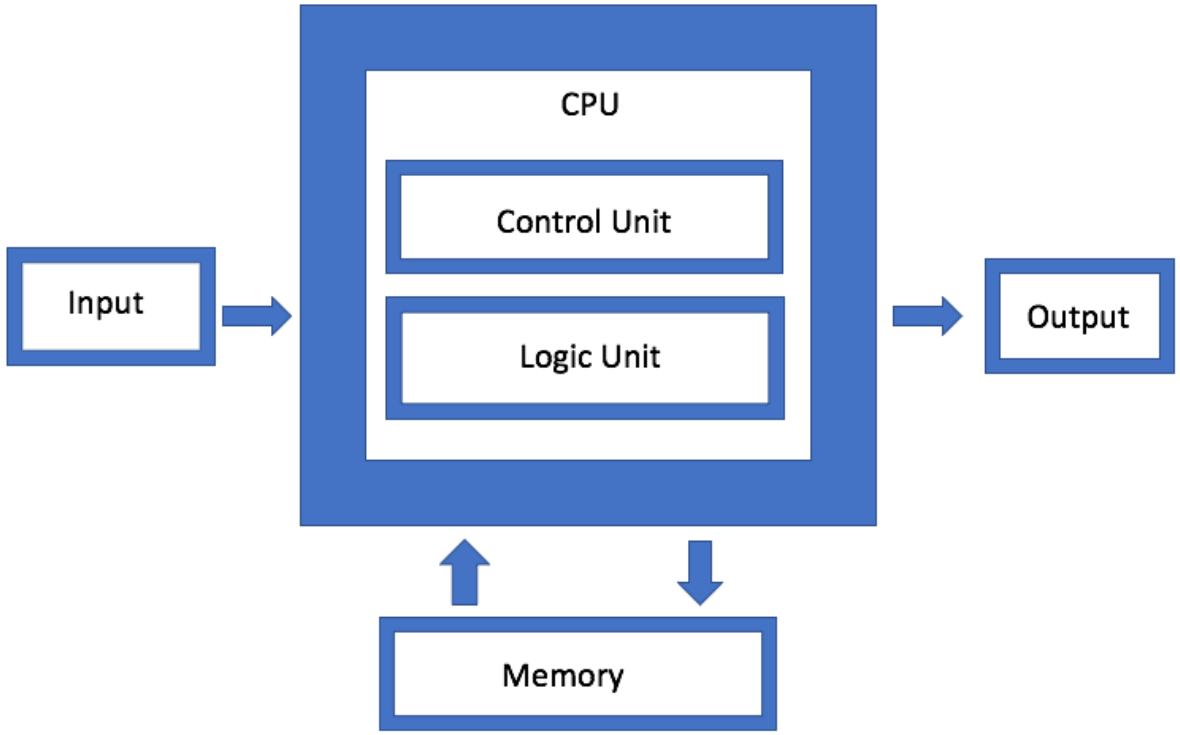
HARDWARE COMPONENTS

- Components:
  - CPU
  - Memory
  - Input
  - Output



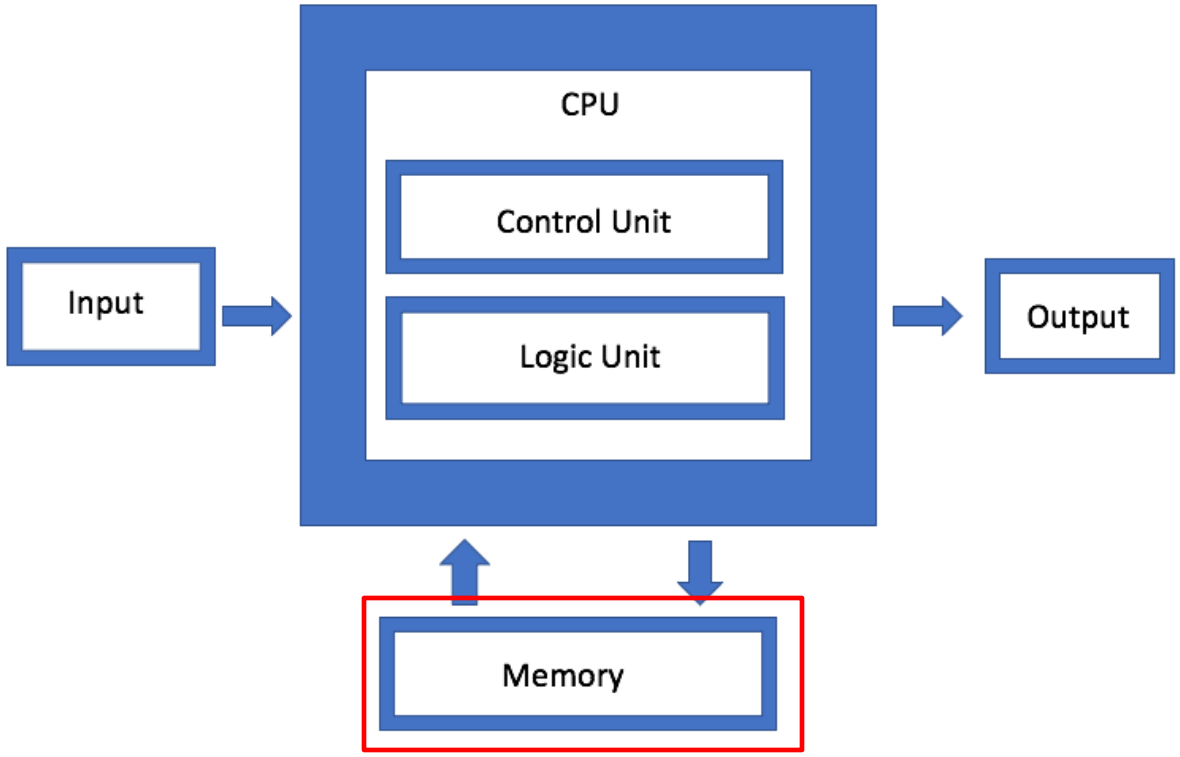
MODERN COMPUTER ARCHITECTURE

- Modern Computer Architecture:
  - Instruction and Data both stored in unified memory.
- At some point, in the early days of computing, program memory and data were separated and each had their own CPU interface.



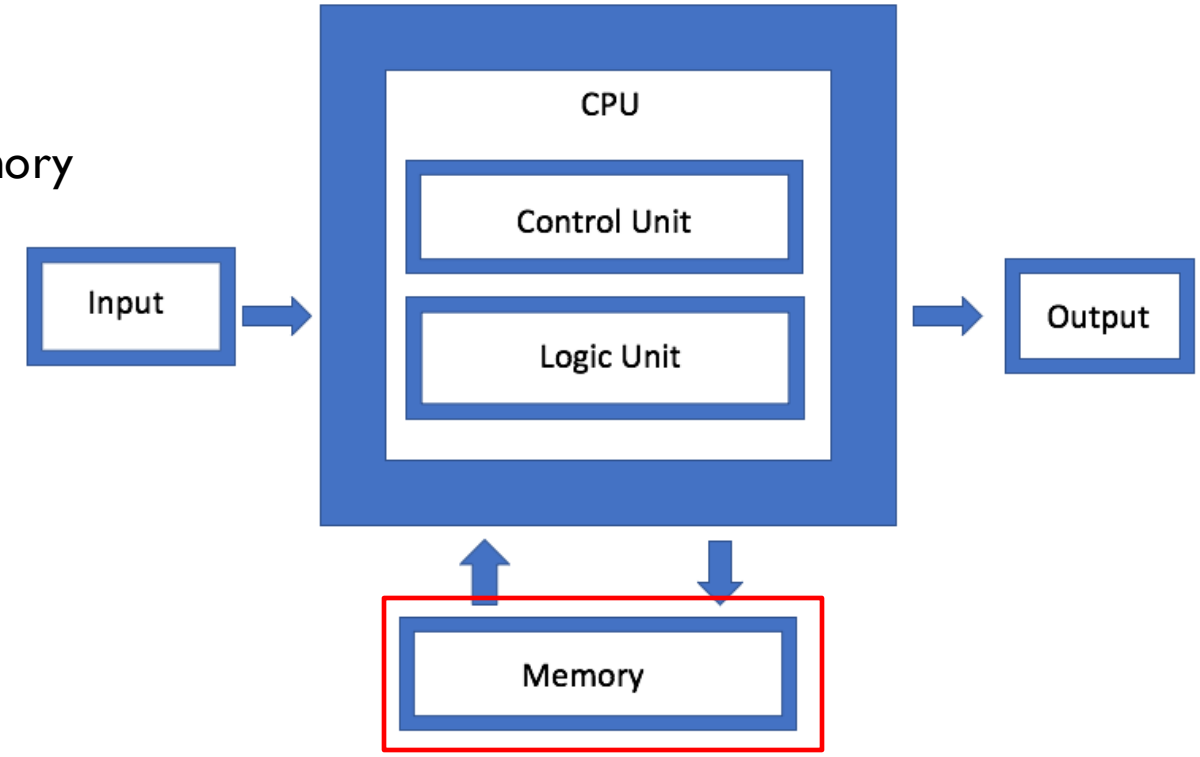
MEMORY

- Memory has many components:
  - Registers
  - Cache
  - Main Memory
  - Hard Disk
  - …

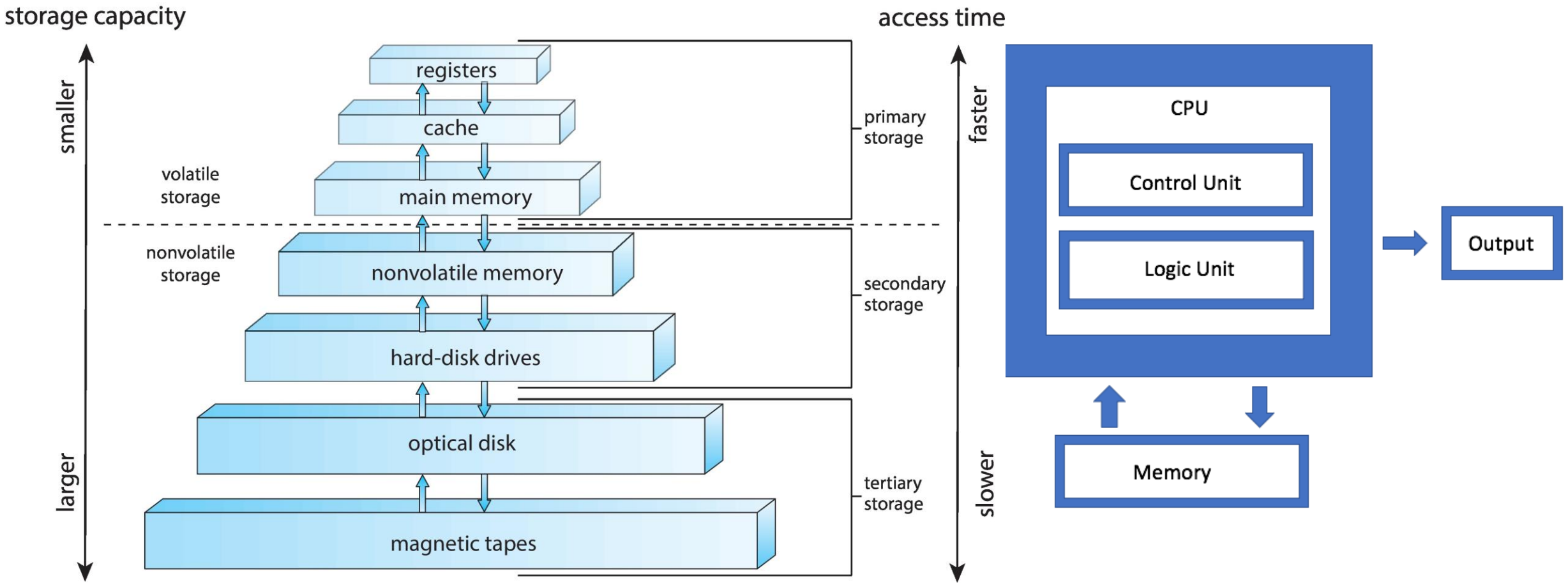


MEMORY

- Memory has many components:
  - Registers → Not really considered memory
  - Cache
  - Main Memory
  - Hard Disk
  - …



MEMORY HIERARCHY



PROGRAMS

Q: What is a program made of?

- A program is composed of a set of instructions.
- Computer hardware simply executes the instructions, one at a time (for single processor/core systems).

INSTRUCTION CYCLE

- Three steps in instruction cycle:

INSTRUCTION CYCLE

- Three steps in instruction cycle:
  - Fetch Instruction

Fetch the instruction

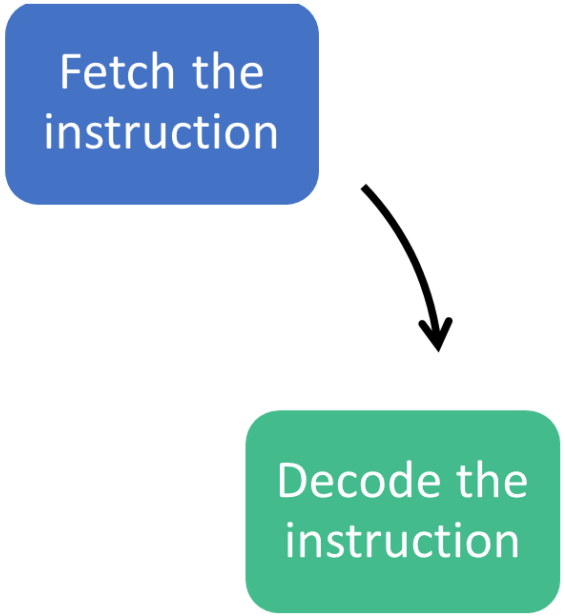
INSTRUCTION CYCLE

- Three steps in instruction cycle:
  - Fetch Instruction
    - Instruction is retrieved from memory

Fetch the instruction

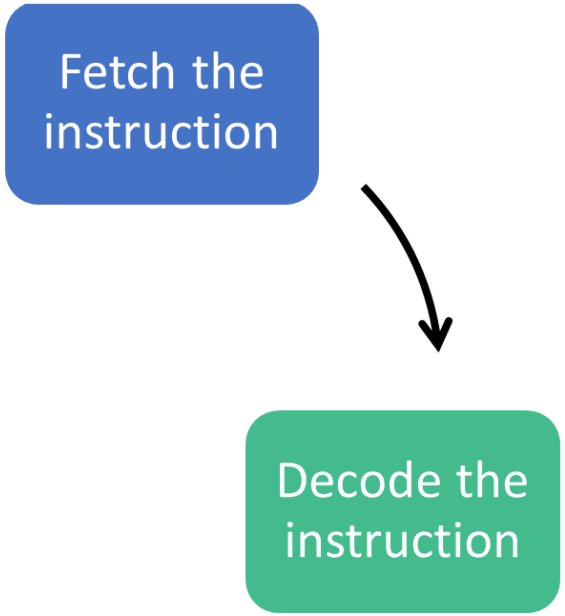
INSTRUCTION CYCLE

- Three steps in instruction cycle:
  - Fetch Instruction
  - Decode Instruction



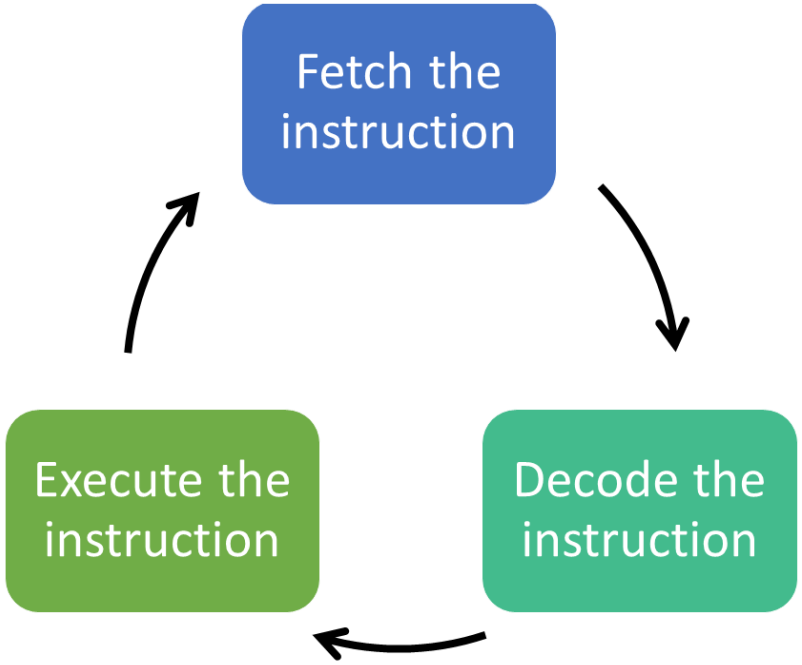
INSTRUCTION CYCLE

- Three steps in instruction
  - Fetch Instruction
  - Decode Instruction
    - Instruction is analyzed.

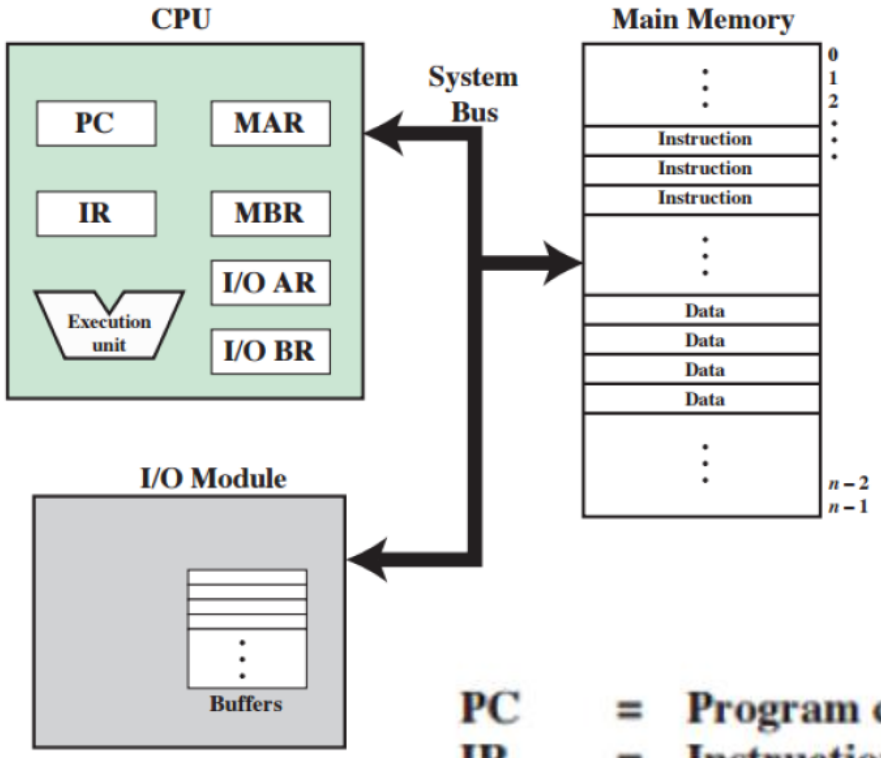


INSTRUCTION CYCLE

- Three steps in instruction cycle
  - Fetch Instruction
  - Decode Instruction
  - Execute Instruction

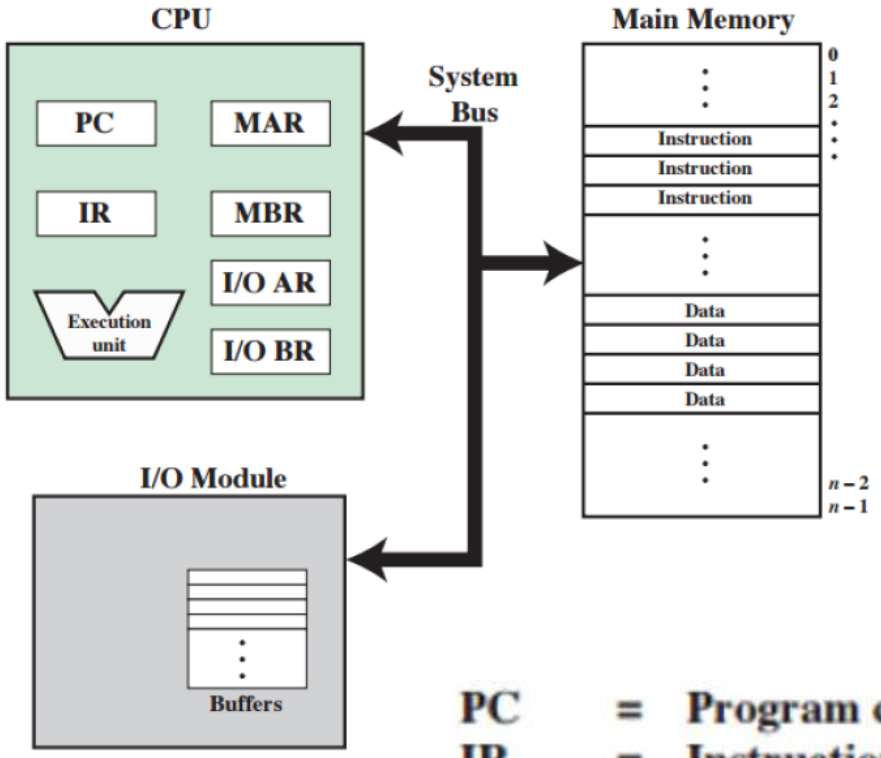


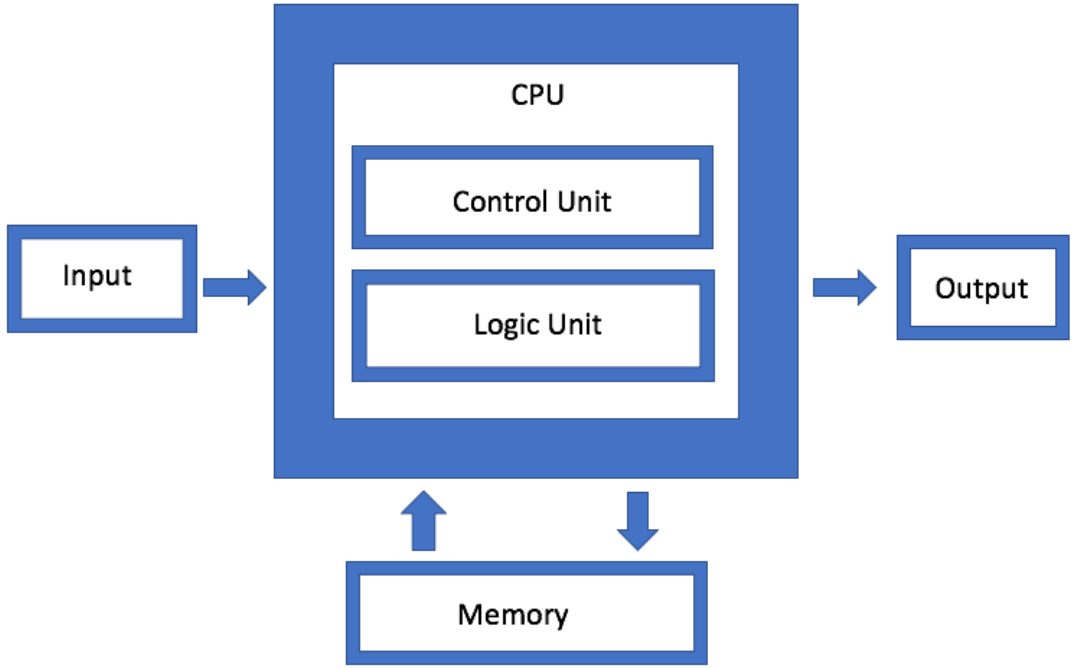
INSTRUCTION CYCLE



PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

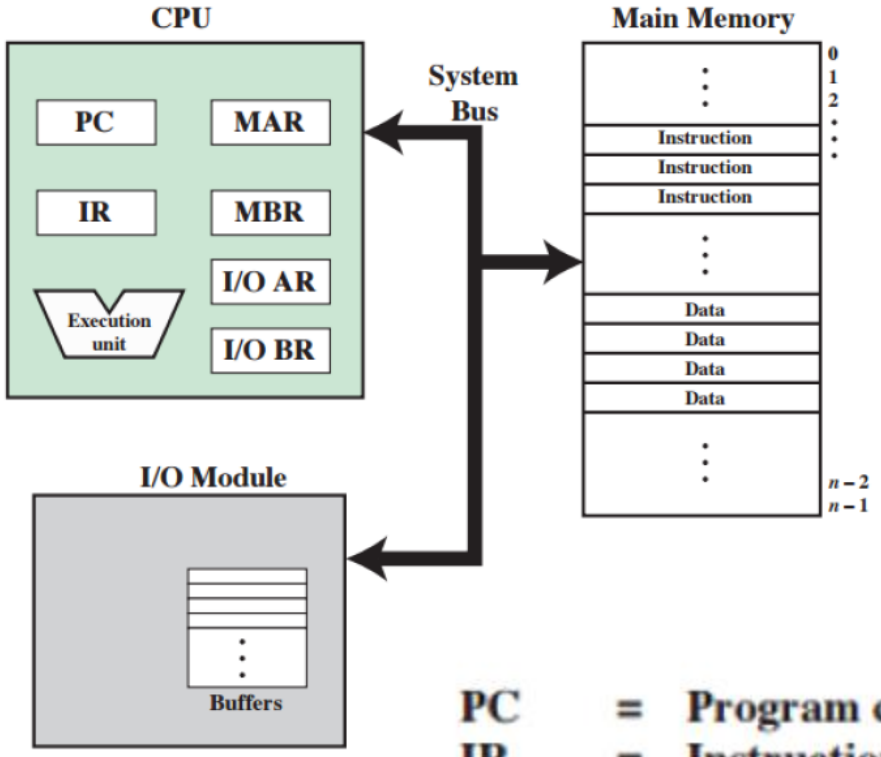
INSTRUCTION CYCLE

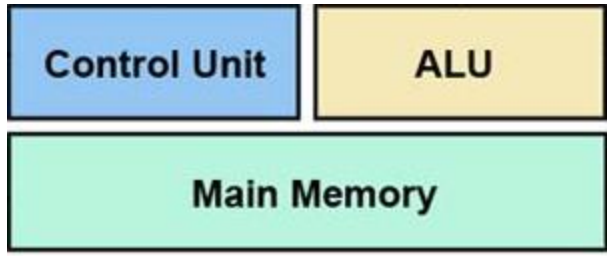




PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

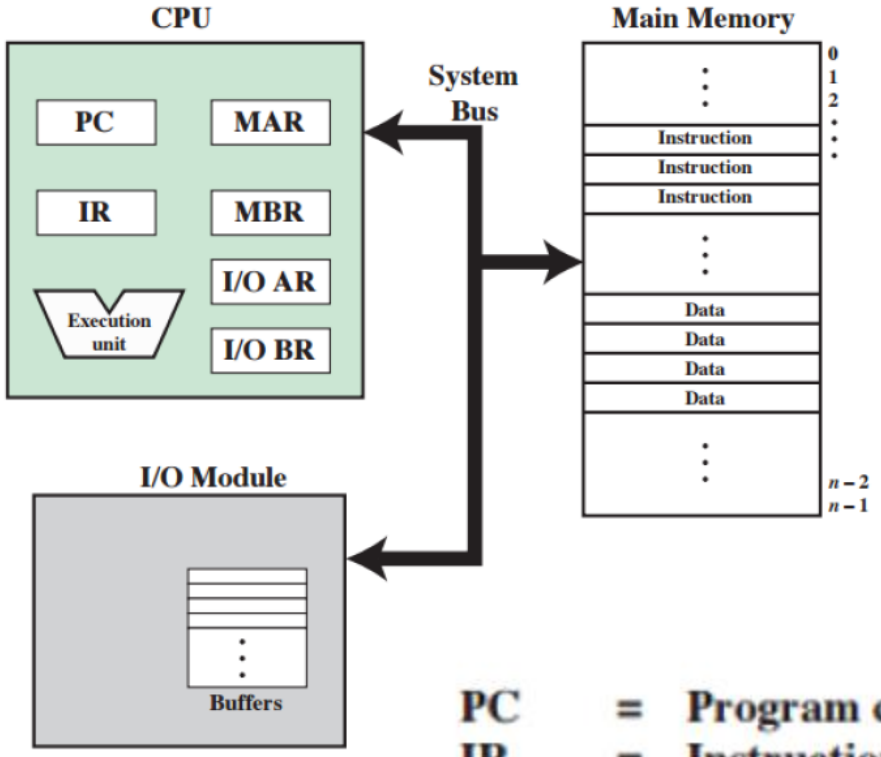
INSTRUCTION CYCLE

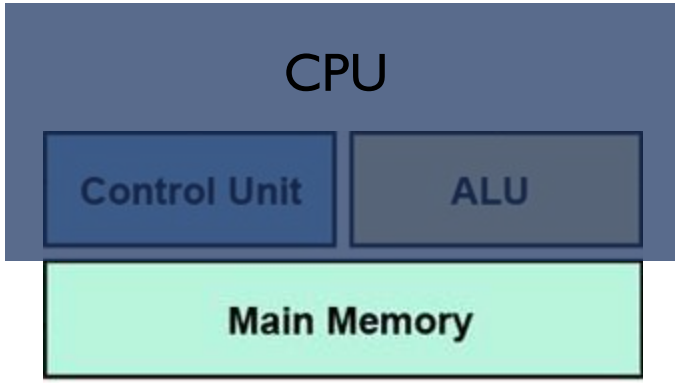




PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

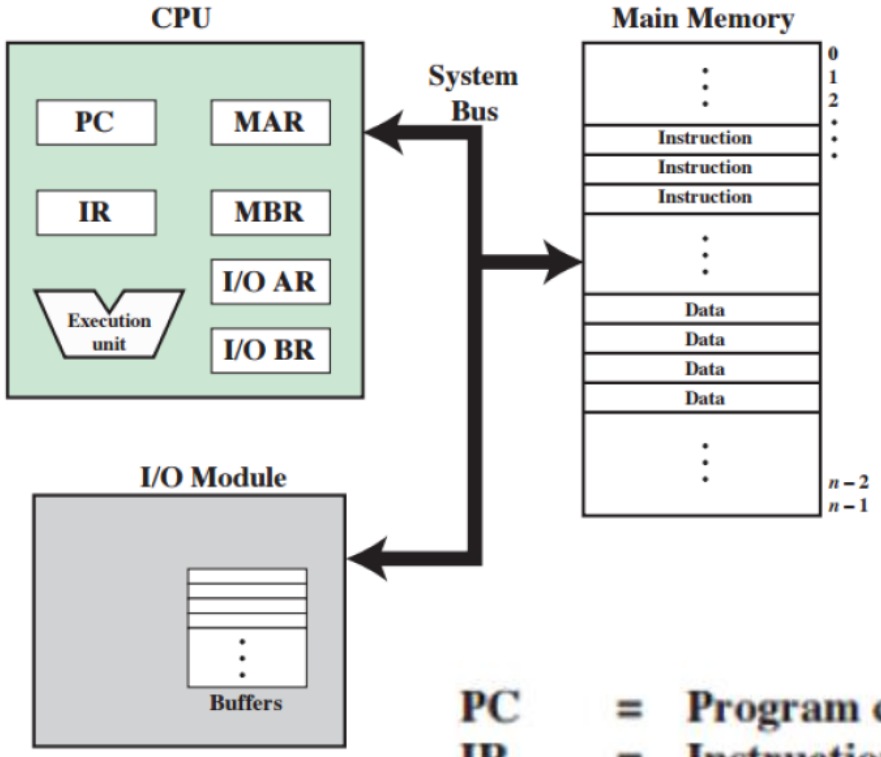
INSTRUCTION CYCLE

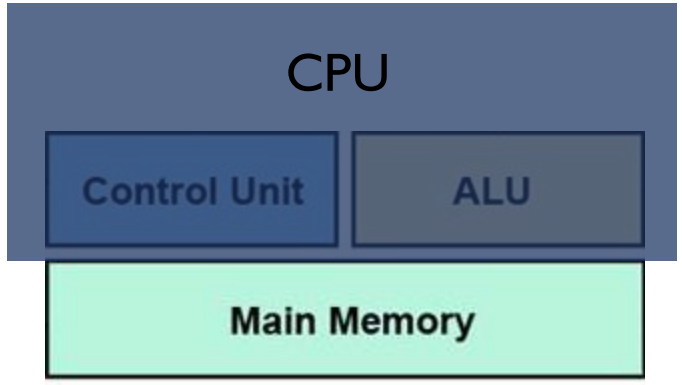




PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

INSTRUCTION CYCLE





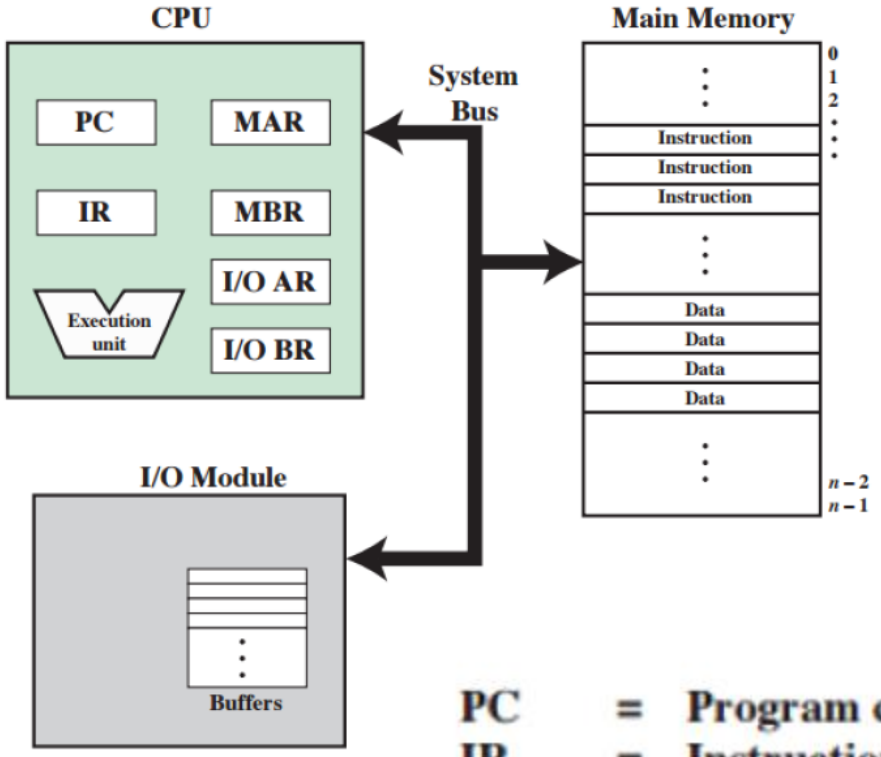
PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

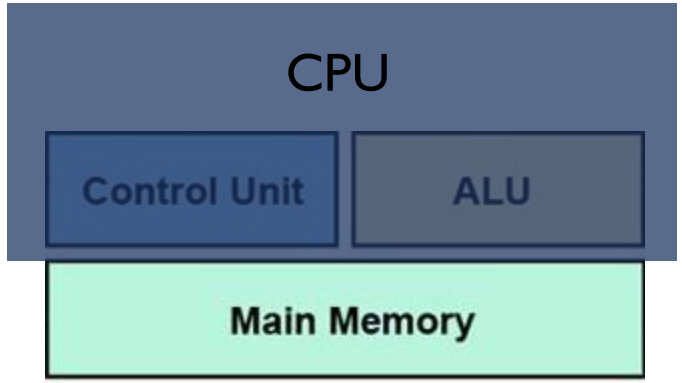
ALU: Execution Unit

Control Unit: Fetches Instructions

INSTRUCTION CYCLE

All execution is performed on registers.



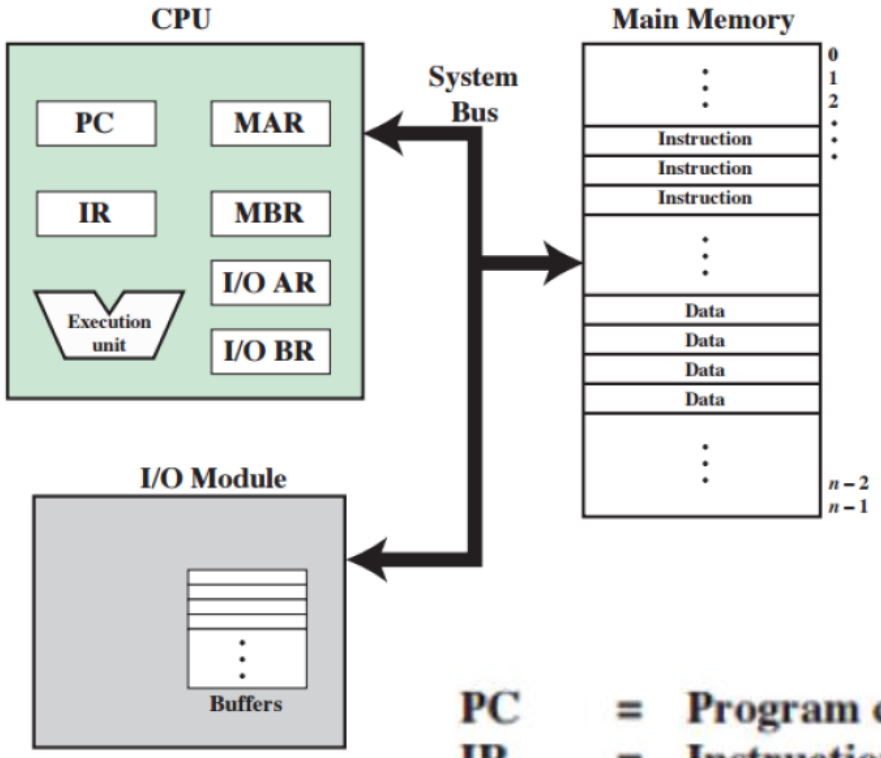


PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

ALU: Execution Unit

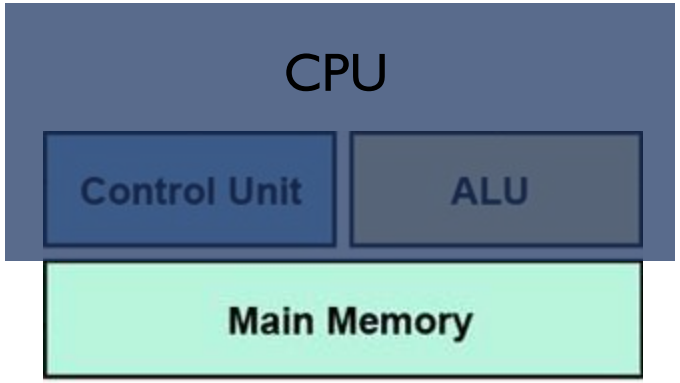
Control Unit: Fetches Instructions

INSTRUCTION CYCLE



PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

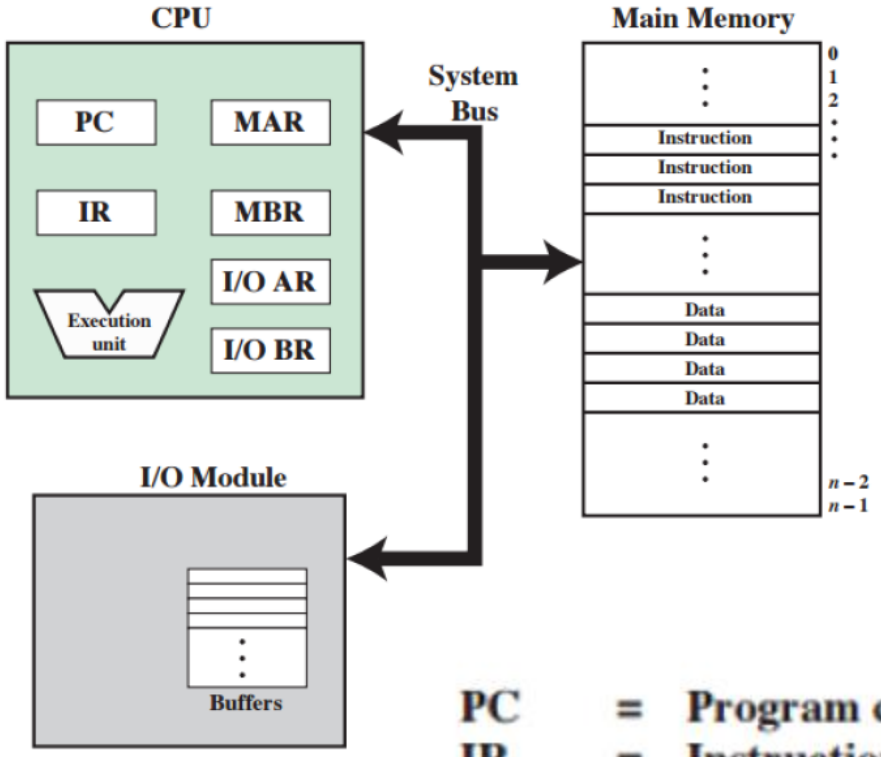
All execution is performed on registers.
Registered are labeled: r1, r2, r3 …
Floating point registers are separate: f1, f2, f3 …

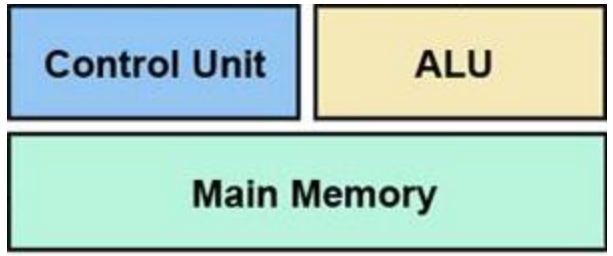


ALU: Execution Unit

Control Unit: Fetches Instructions

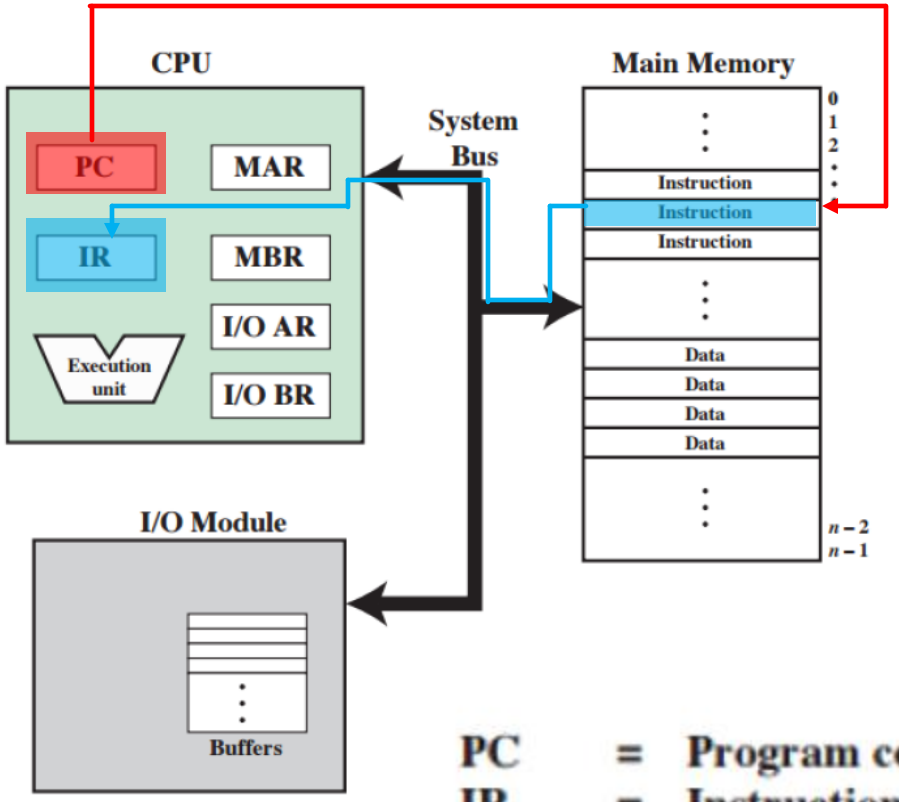
INSTRUCTION CYCLE

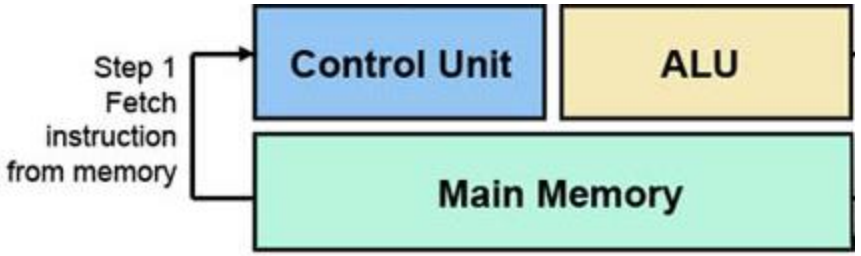




PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

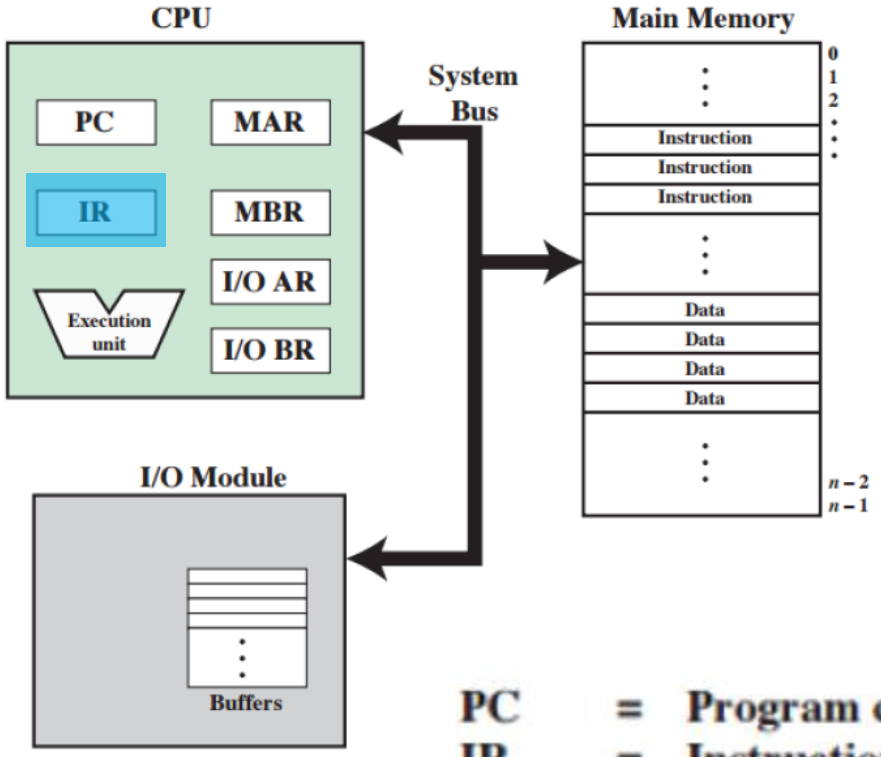
FETCH

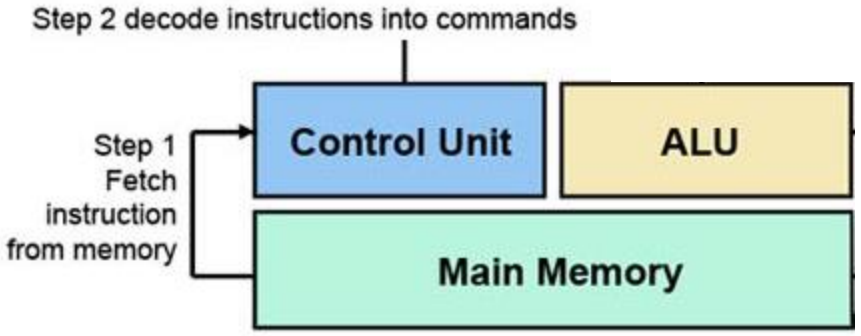




PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

DECODE

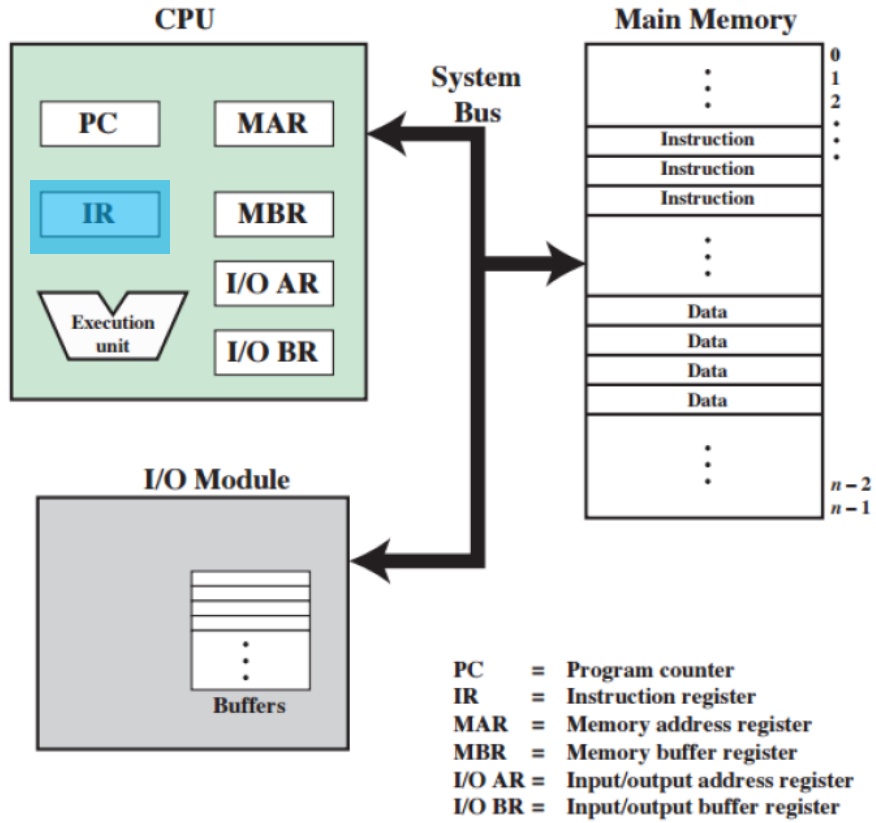




PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

INSTRUCTION FORMAT

- How is an instruction decoded?



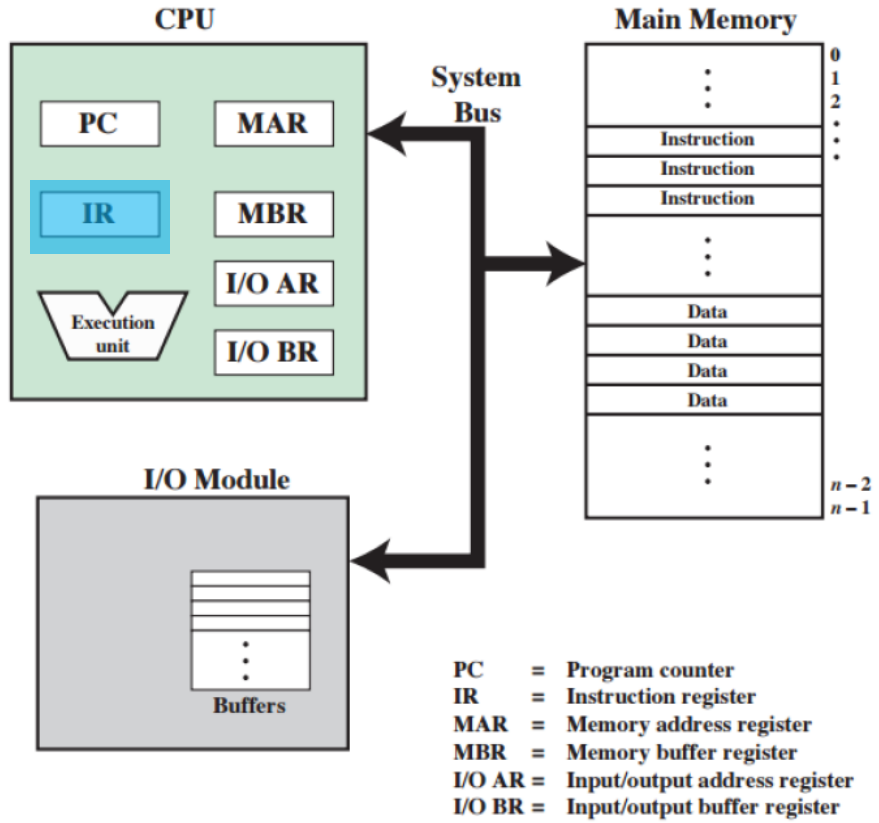
INSTRUCTION FORMAT

- How is an instruction decoded?

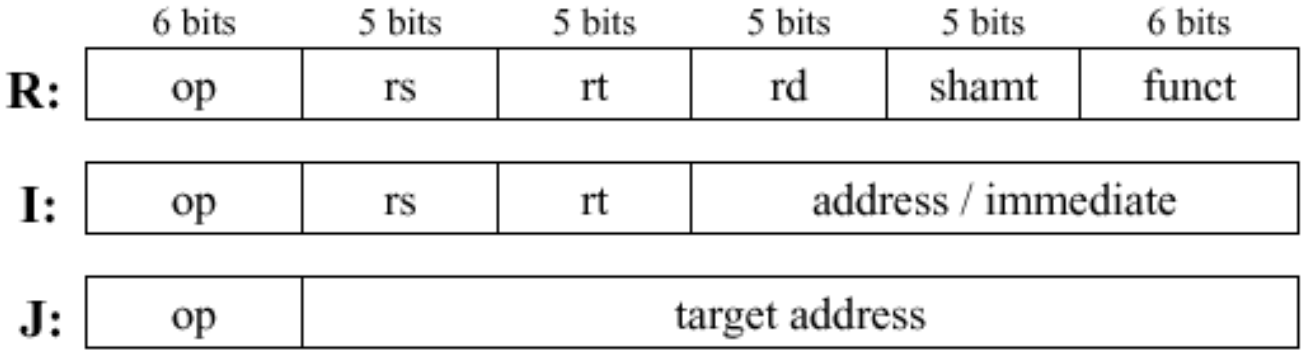
An instruction will look something like this:

001110 01001 01011 01010 00000 000000

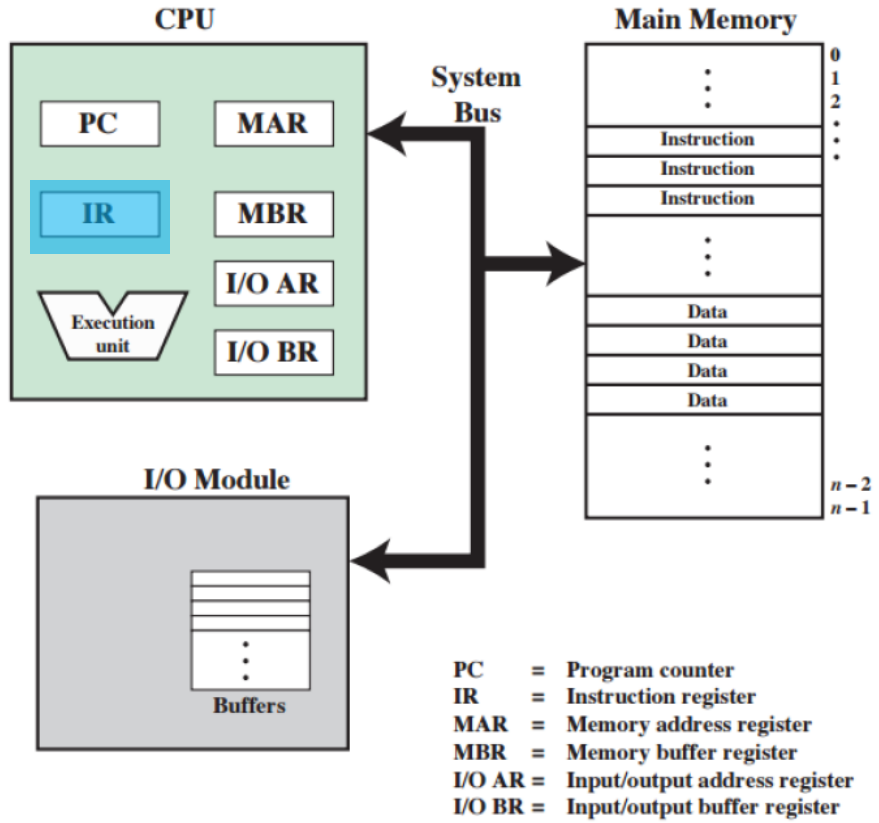
How can we "decode" that into a meaningful CPU operation?



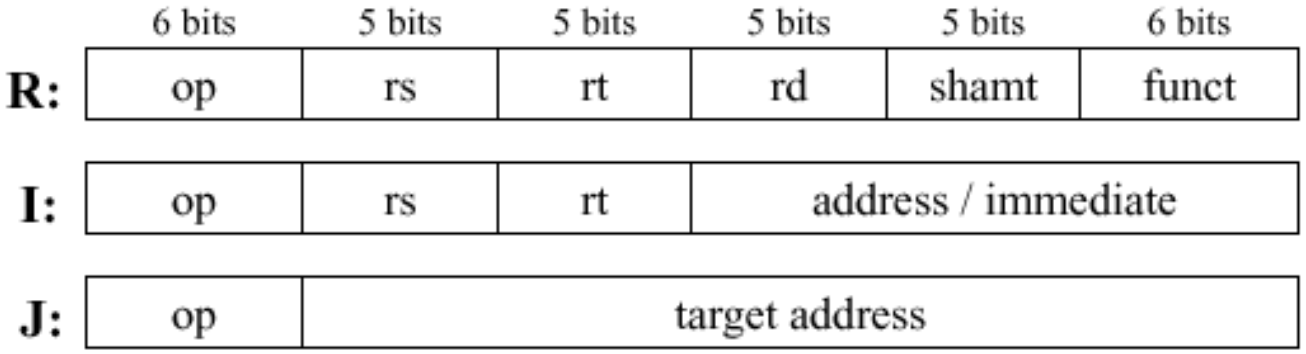
INSTRUCTION FORMAT



MIPS Architecture

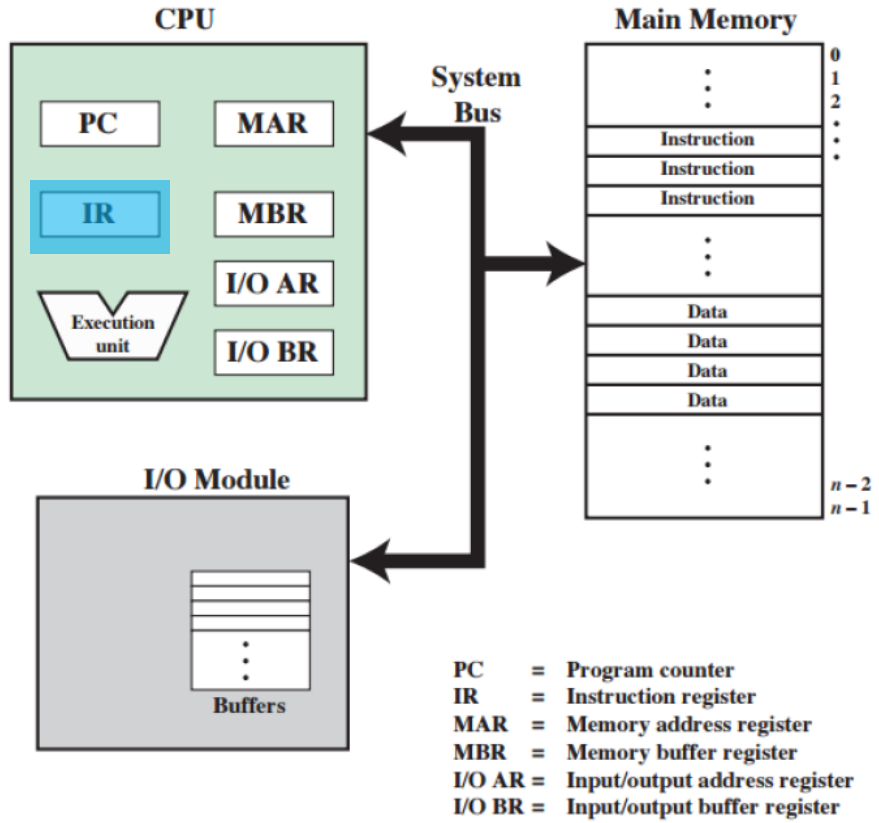


INSTRUCTION FORMAT

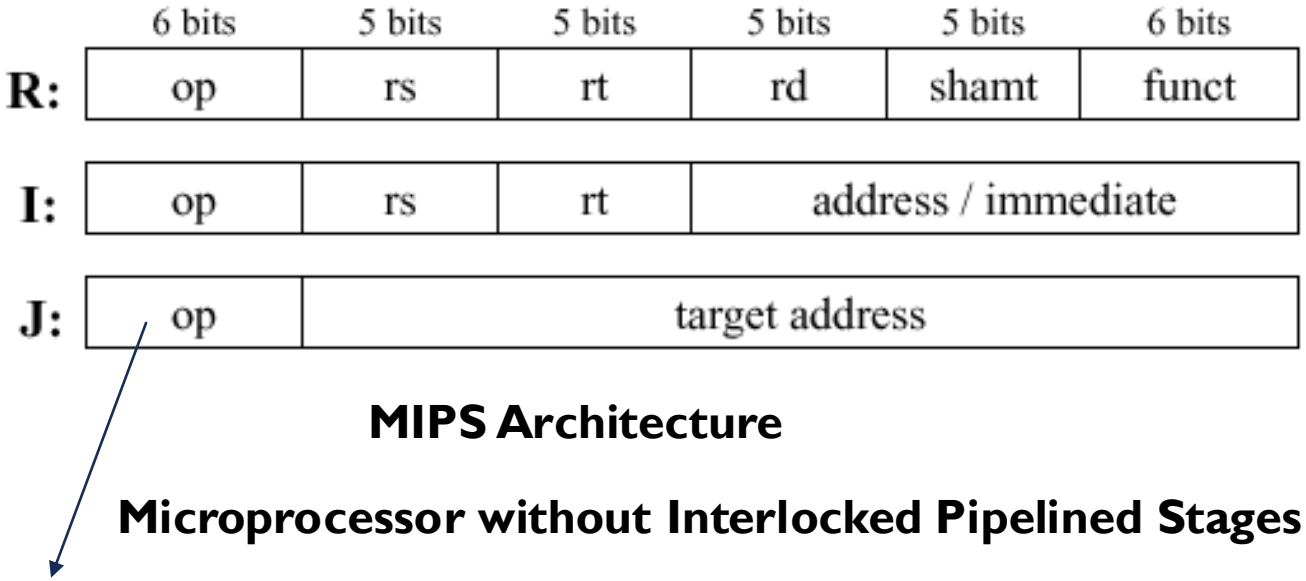


MIPS Architecture

Microprocessor without Interlocked Pipelined Stages



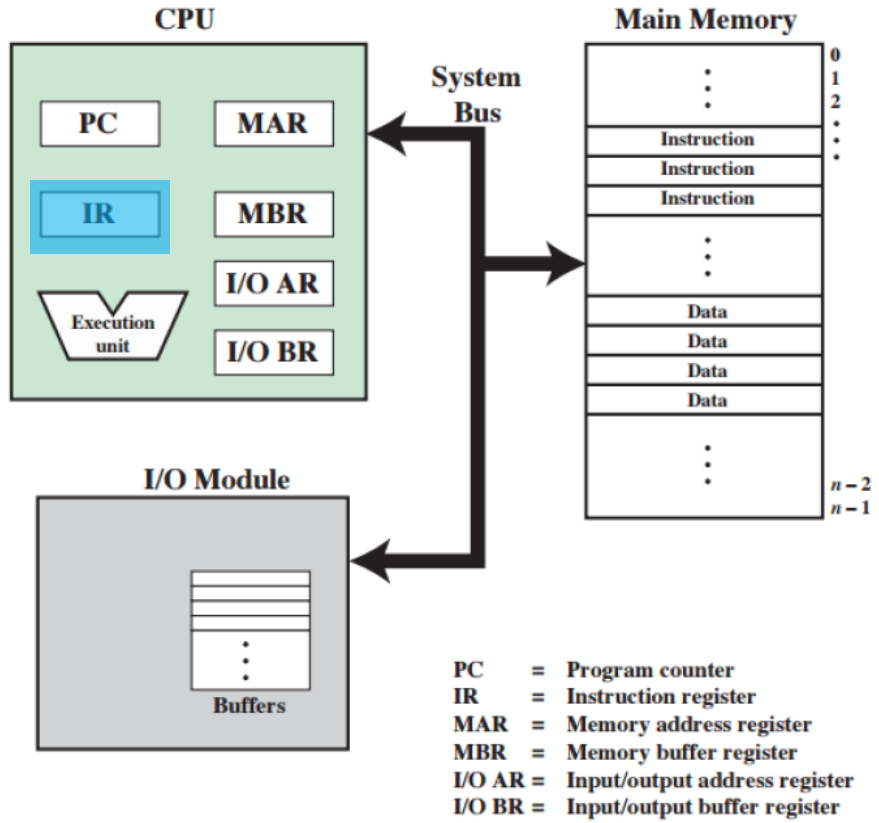
INSTRUCTION FORMAT



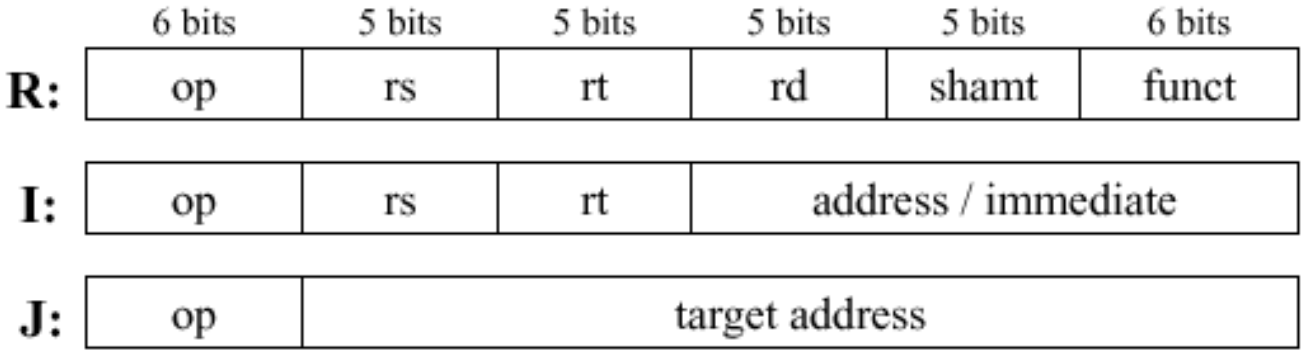
MIPS Architecture

Microprocessor without Interlocked Pipelined Stages

- The op code is the instruction 'ID'
- It indicates both the format and operation of the instruction



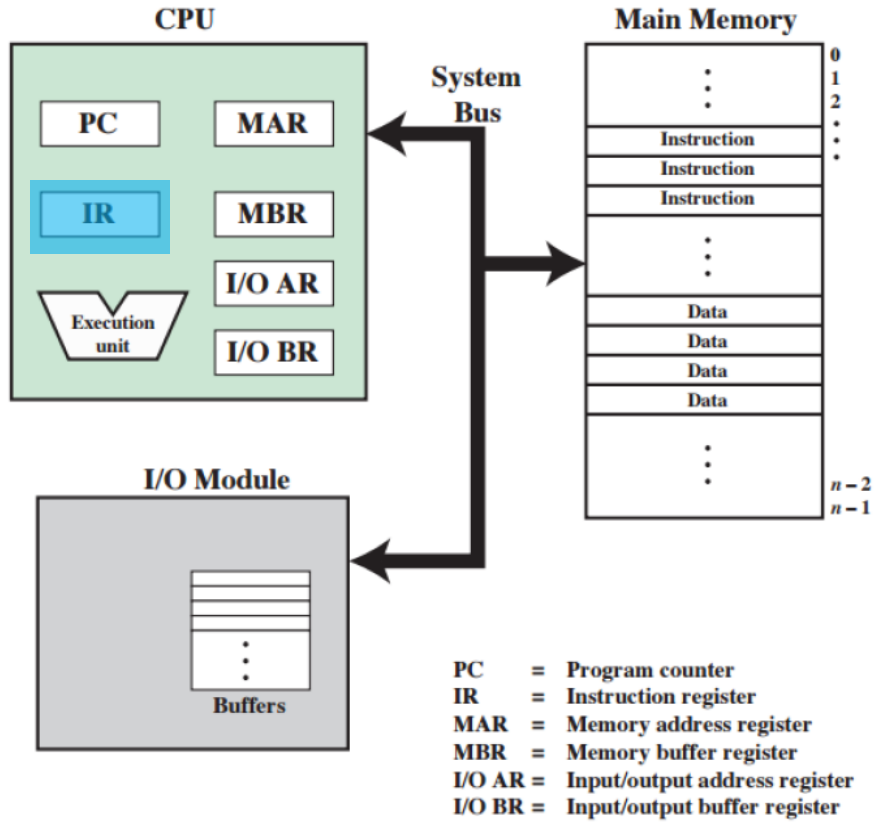
INSTRUCTION FORMAT



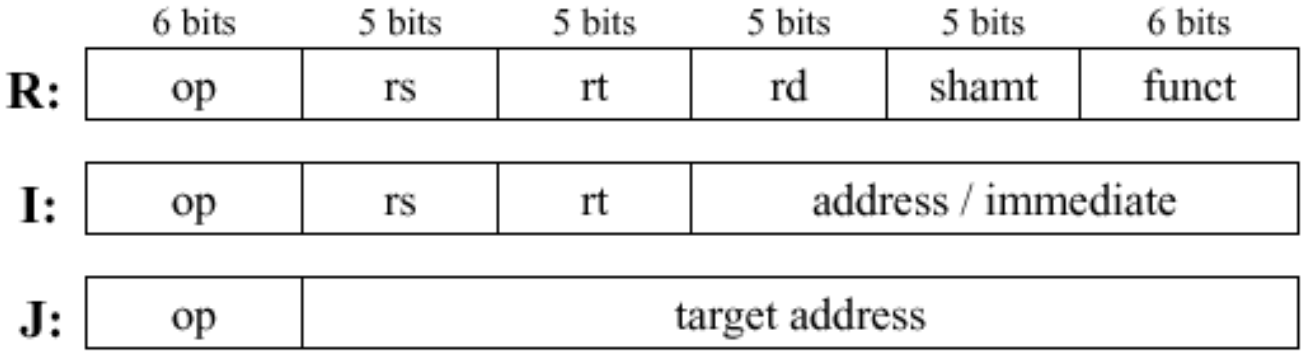
MIPS Architecture

Microprocessor without Interlocked Pipelined Stages

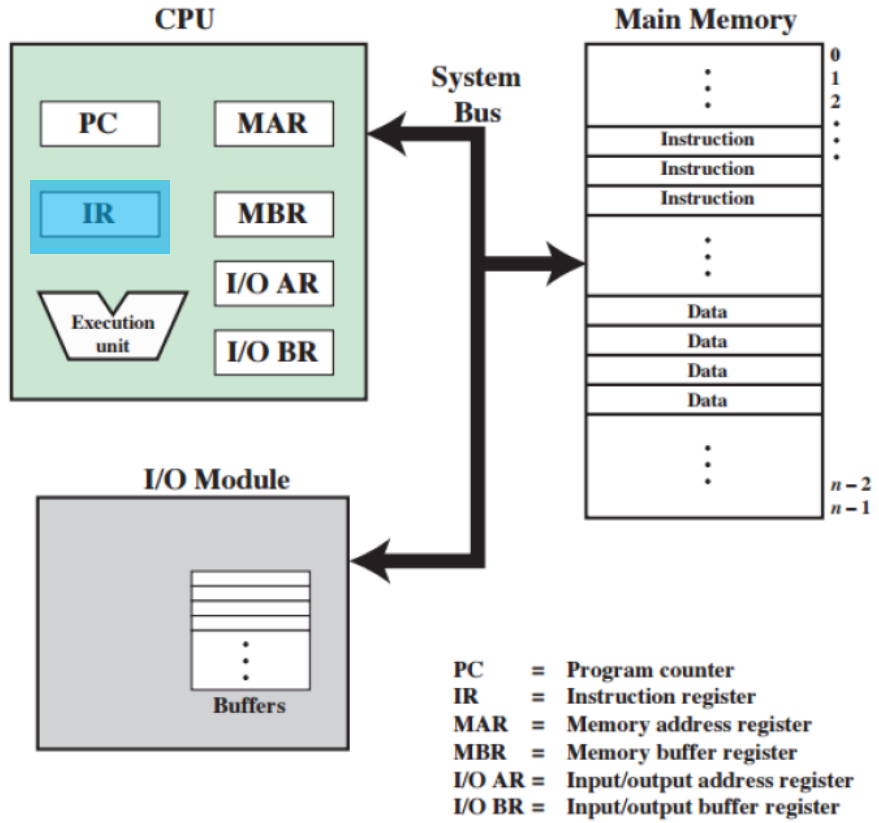
- R: Register type instruction, all operands are registers.
- I: Immediate: Contains immediate values without loading from registers.
- J: Special instruction for jump to new address



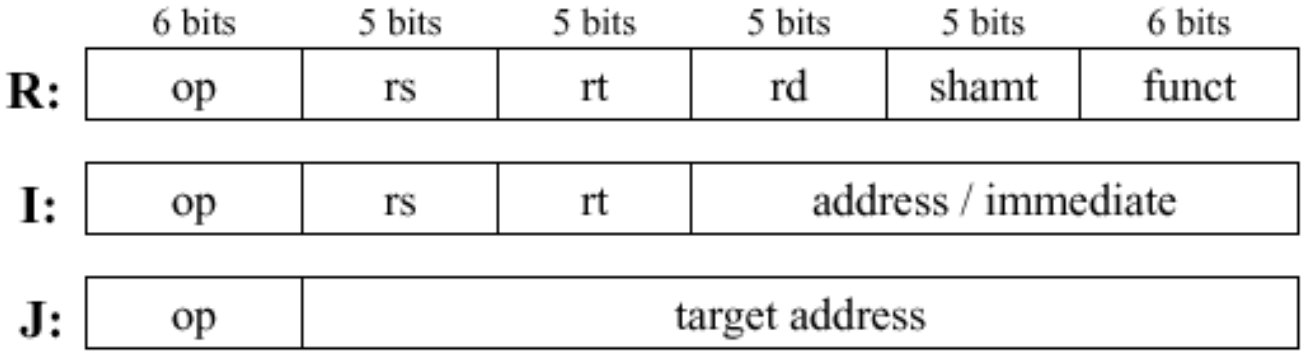
INSTRUCTION DECODE



op: basic operation of the instruction (opcode)
rs: first source operand register
rt: second source operand register
rd: destination operand register
shamt: shift amount
funct: selects the specific variant of the opcode (function code)
address: offset for load/store instructions ($+/-2^{15}$)
immediate: constants for immediate instructions



EXERCISE



- 001110: add R-Type

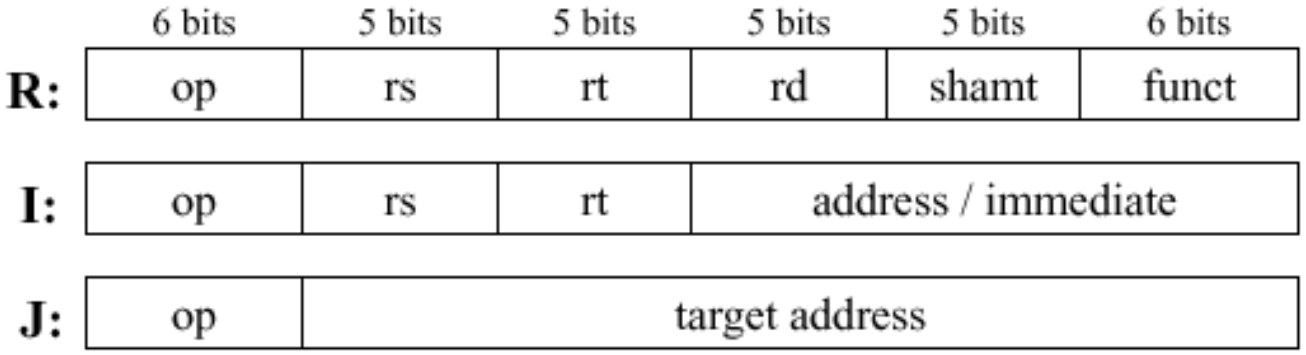
Before Execution

Register	Content
R9	125
R10	56
R11	196
R12	1323

After Execution

Register	Content
R9	???
R10	???
R11	???
R12	???

EXERCISE



- 001110: add R-Type

001110 01010 01100 01010 00000 000000

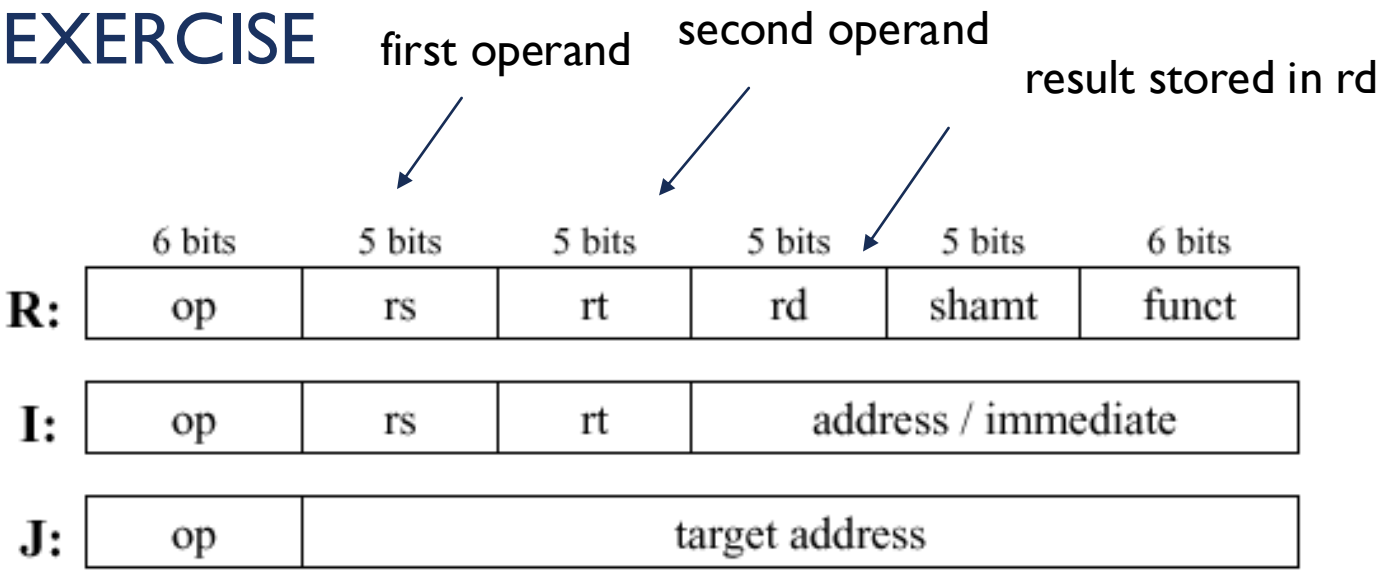
Before Execution

Register	Content
R9	125
R10	56
R11	196
R12	1323

After Execution

Register	Content
R9	???
R10	???
R11	???
R12	???

EXERCISE



- 001110: add R-Type

001110 01010 01100 01010 00000 000000

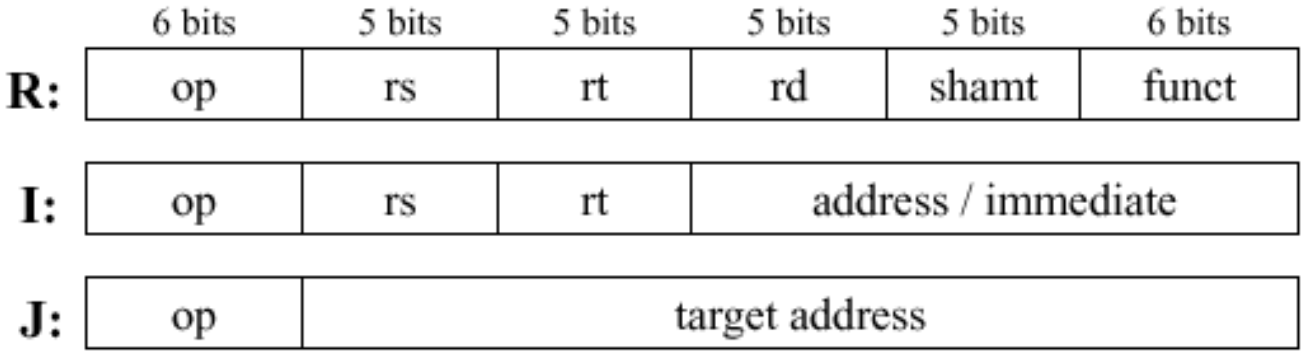
Before Execution

Register	Content
R9	125
R10	56
R11	196
R12	1323

After Execution

Register	Content
R9	???
R10	???
R11	???
R12	???

EXERCISE



- 001110: add

001110	01010	01100	01010	00000	000000
add	R10	R12	R10	N/A	N/A

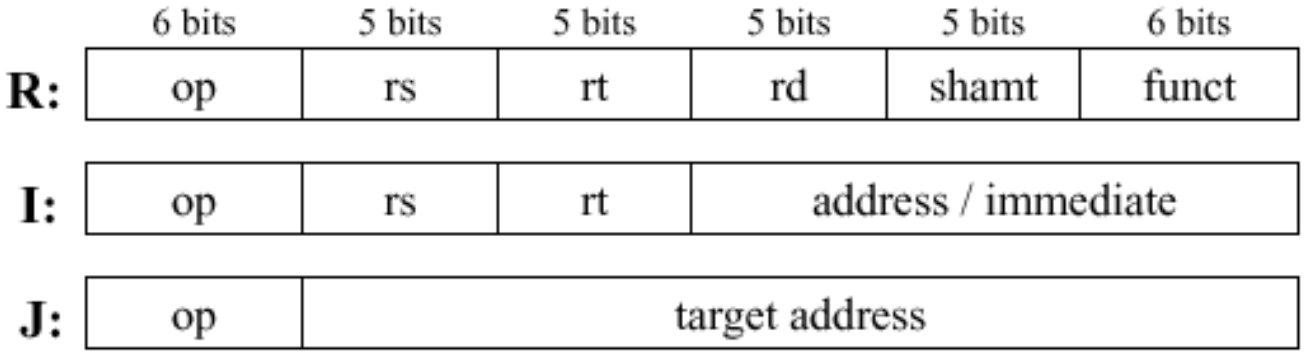
Before Execution

Register	Content
R9	125
R10	56
R11	196
R12	1323

After Execution

Register	Content
R9	???
R10	???
R11	???
R12	???

EXERCISE



- 001110: add

001110	01010	01100	01010	00000	000000
add	R10	R12	R10	N/A	N/A

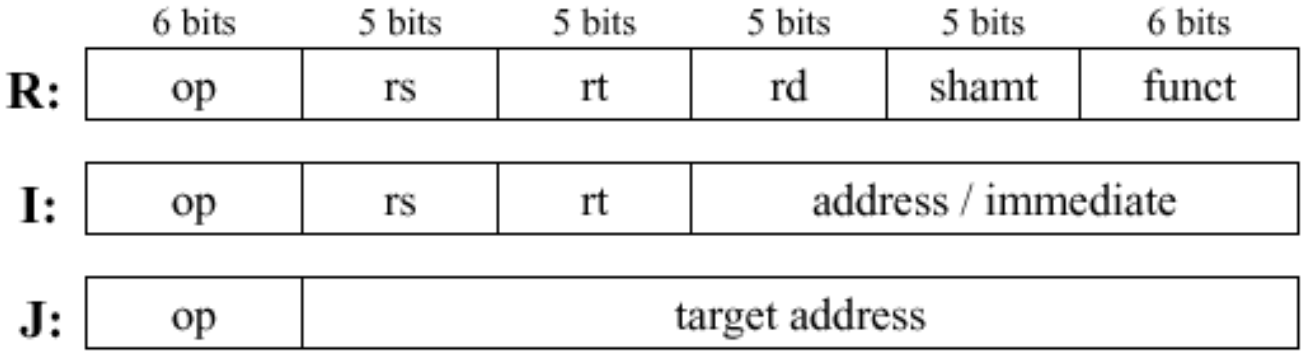
Before Execution

Register	Content
R9	125
R10	56
R11	196
R12	1323

After Execution

Register	Content
R9	125
R10	???
R11	???
R12	???

EXERCISE



- 001110: add

001110	01010	01100	01010	00000	000000
add	R10	R12	R10	N/A	N/A

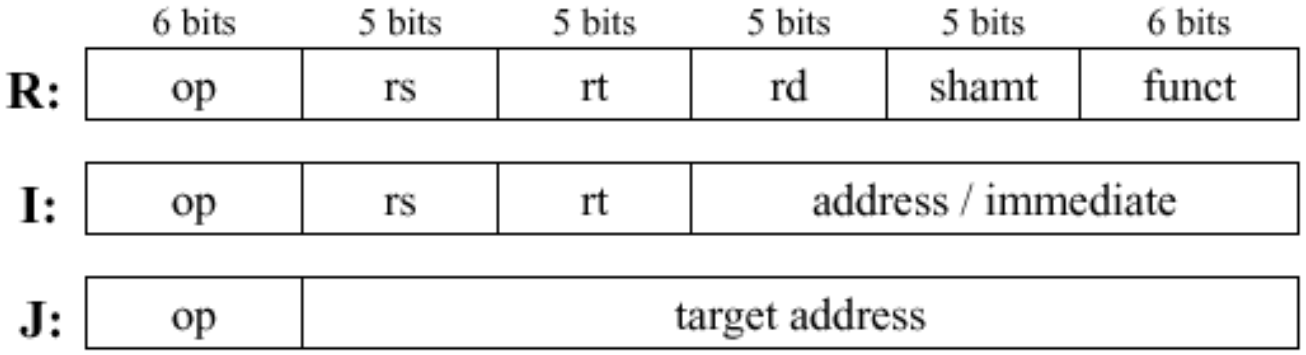
Before Execution

Register	Content
R9	125
R10	56
R11	196
R12	1323

After Execution

Register	Content
R9	125
R10	1379
R11	???
R12	???

EXERCISE



- 001110: add

001110	01010	01100	01010	00000	000000
add	R10	R12	R10	N/A	N/A

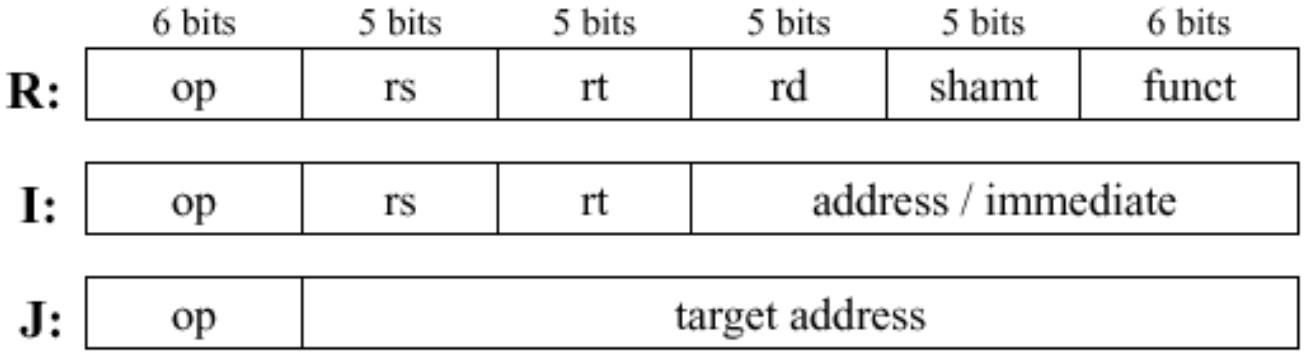
Before Execution

Register	Content
R9	125
R10	56
R11	196
R12	1323

After Execution

Register	Content
R9	125
R10	1379
R11	196
R12	???

EXERCISE



- 001110: add

001110	01010	01100	01010	00000	000000
add	R10	R12	R10	N/A	N/A

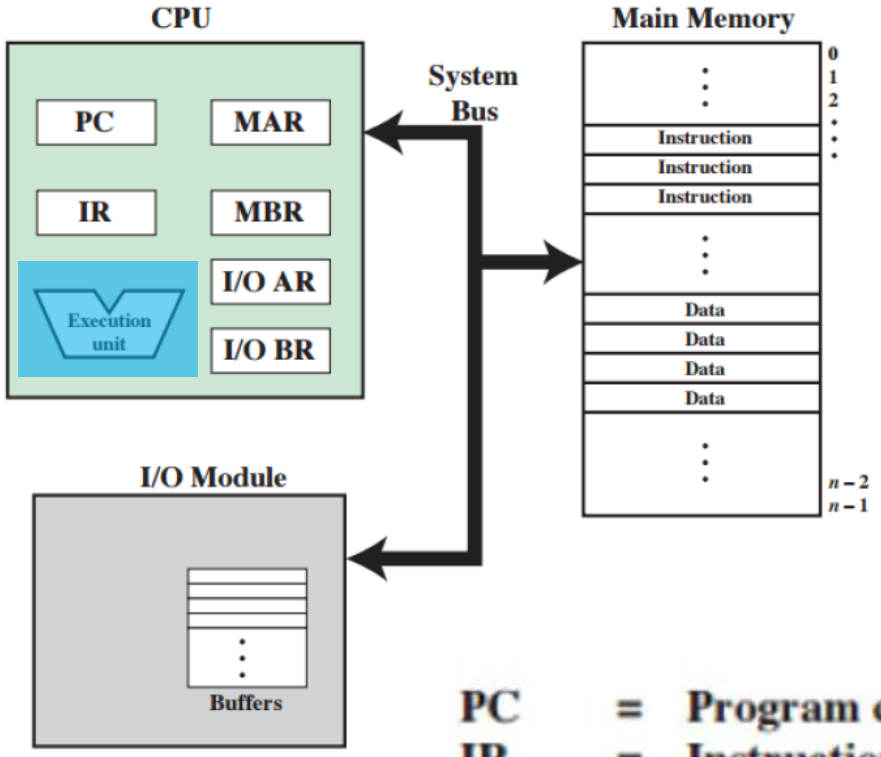
Before Execution

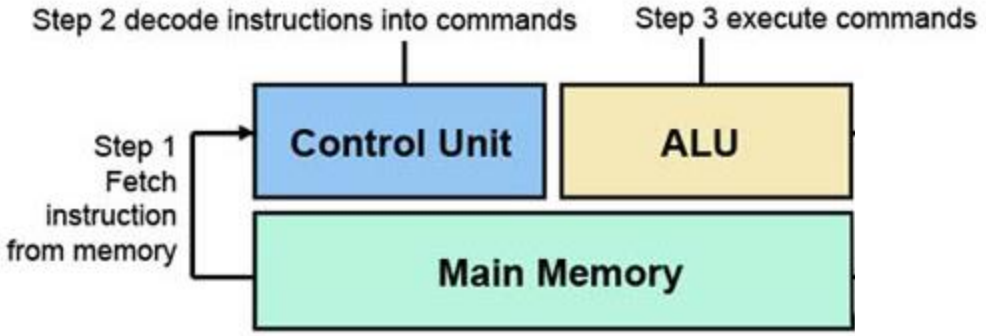
Register	Content
R9	125
R10	56
R11	196
R12	1323

After Execution

Register	Content
R9	125
R10	1379
R11	196
R12	1323

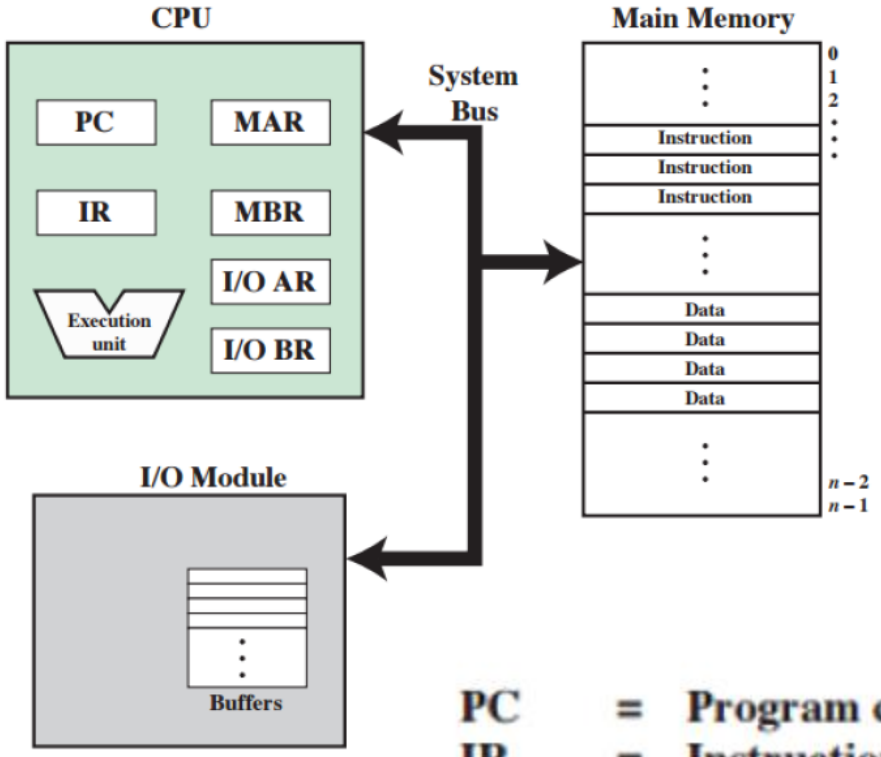
INSTRUCTION CYCLE: EXECUTE

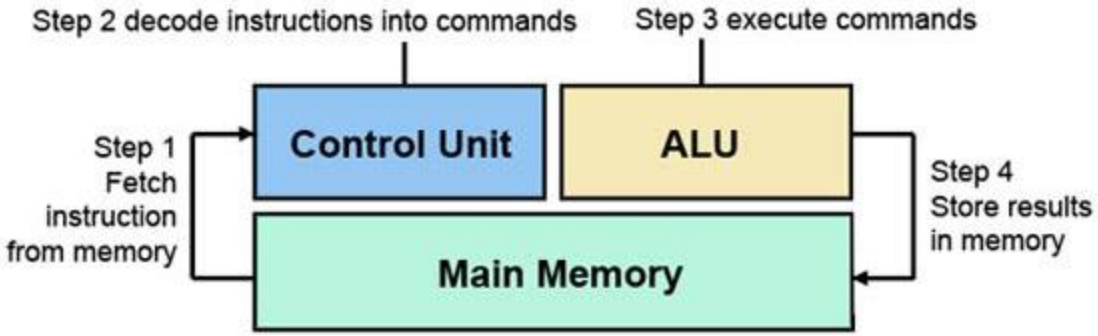




PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

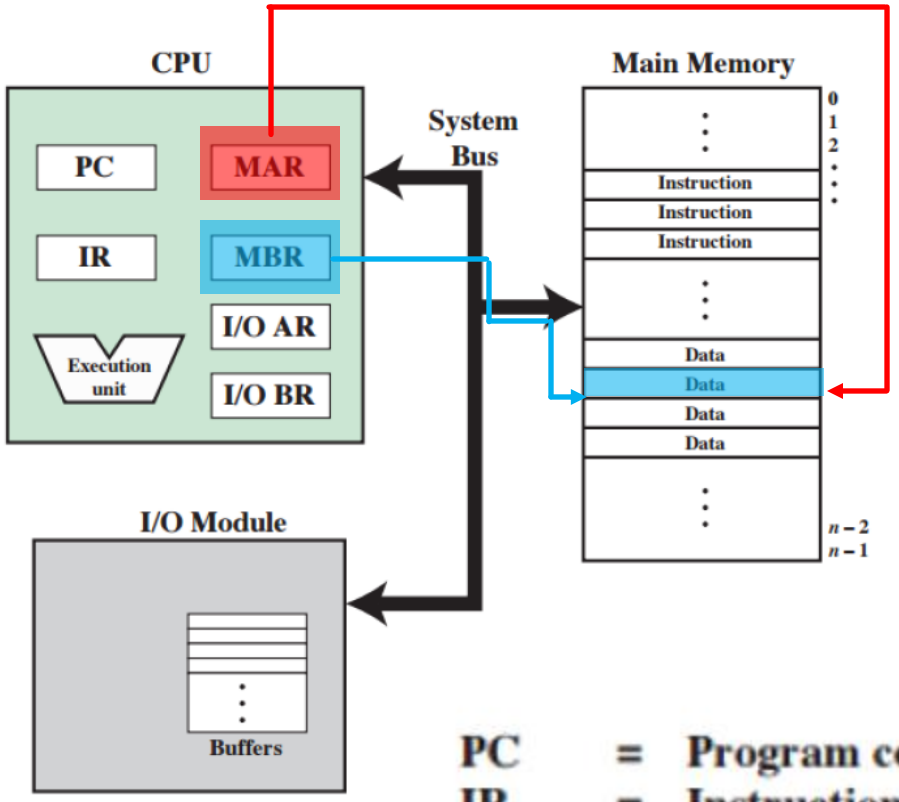
STEP 4: STORE IN MEMORY

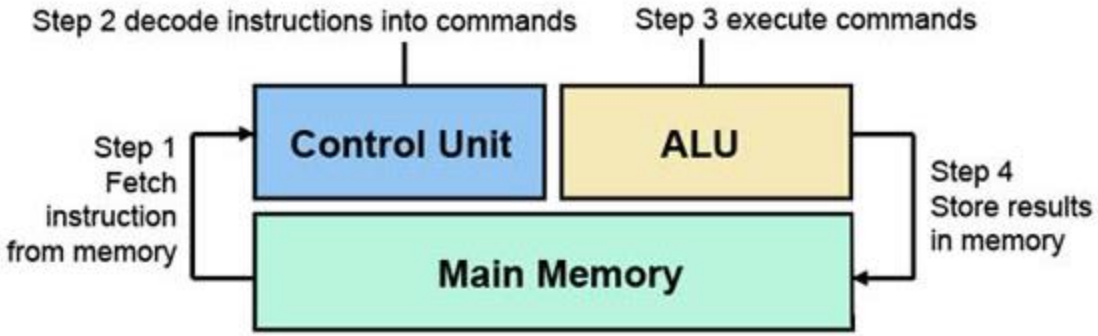




PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

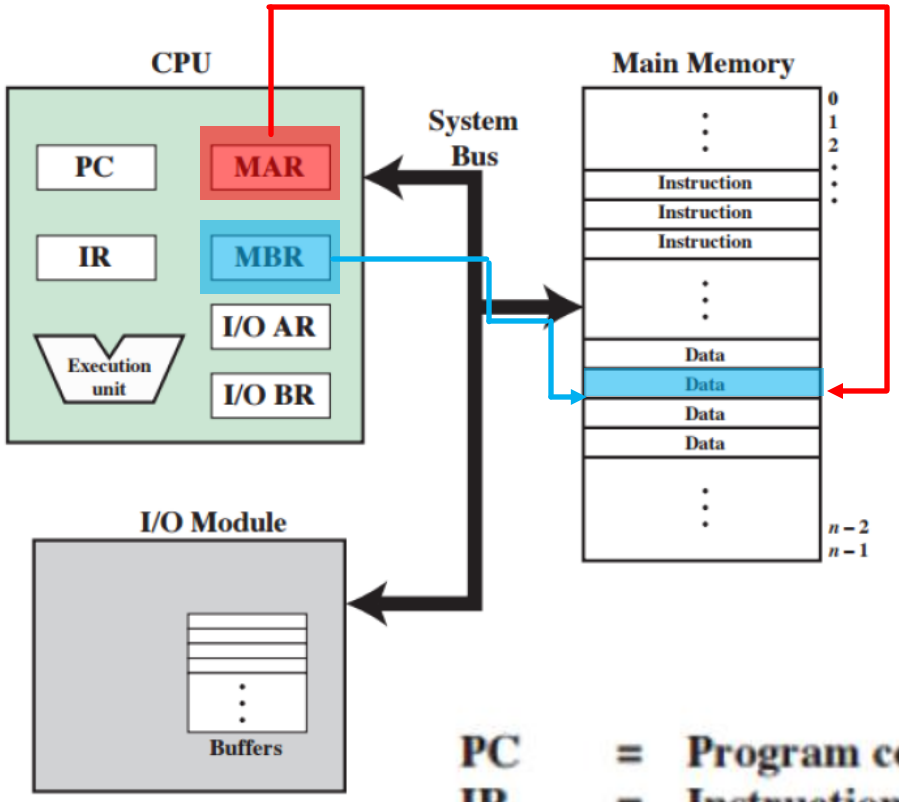
STEP 4: STORE IN MEMORY



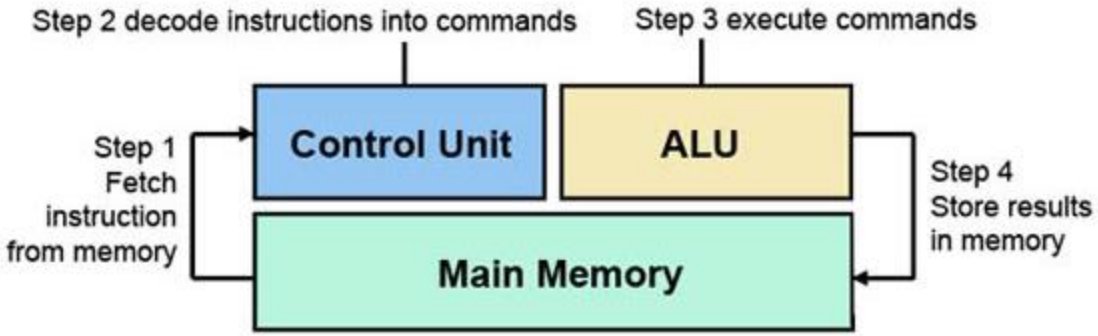


PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

STEP 4: STORE IN MEMORY

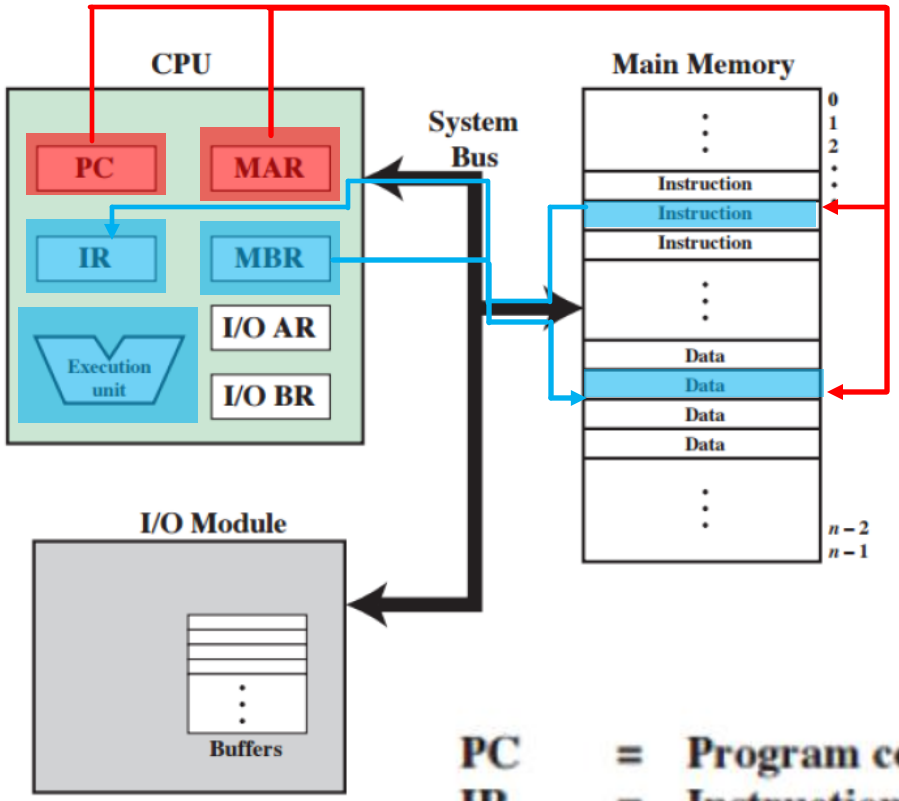


- Step 1 to 3 : Typical Instruction Cycle
- Step 4: Only when we need to store the result.

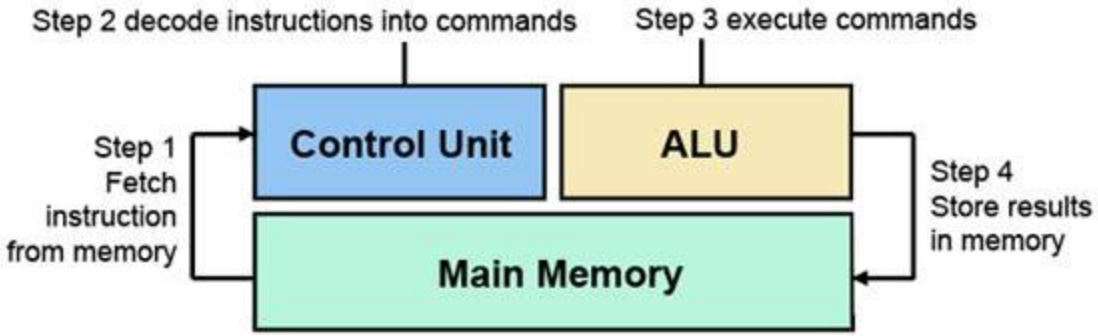


PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

OPERATING SYSTEM

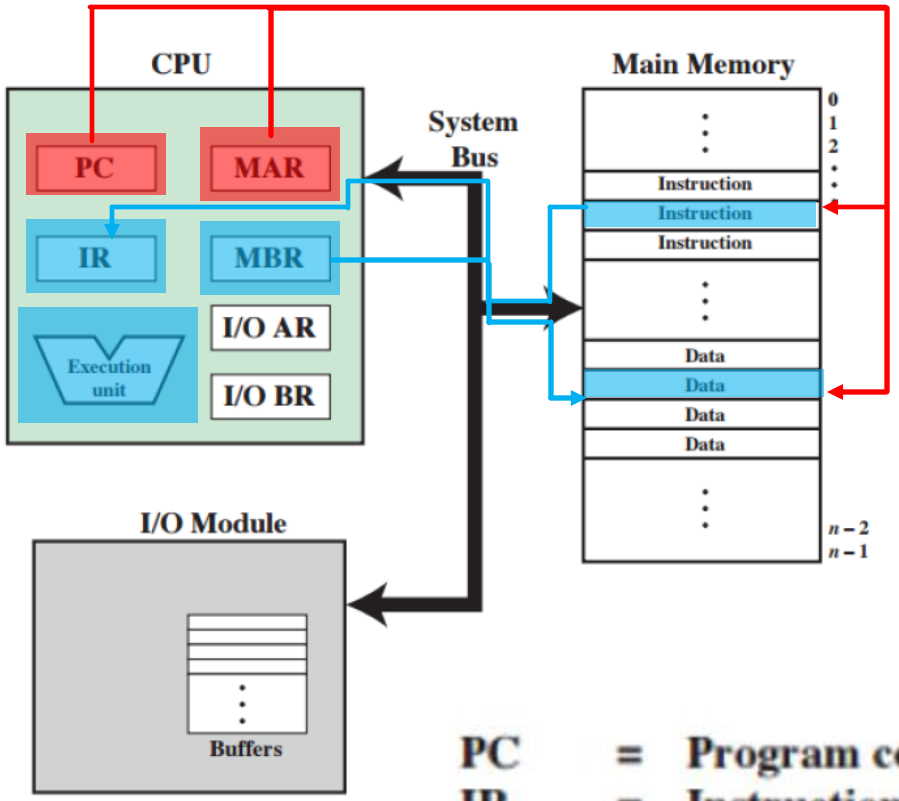


- So where is the Operating System?

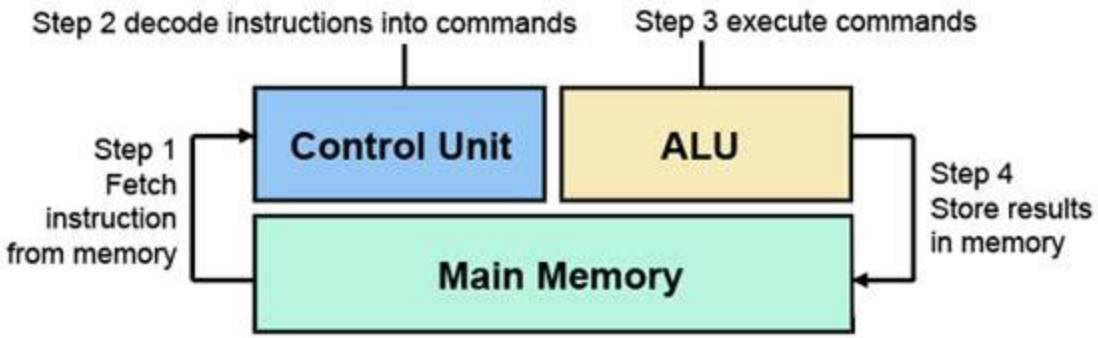


PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

OPERATING SYSTEM

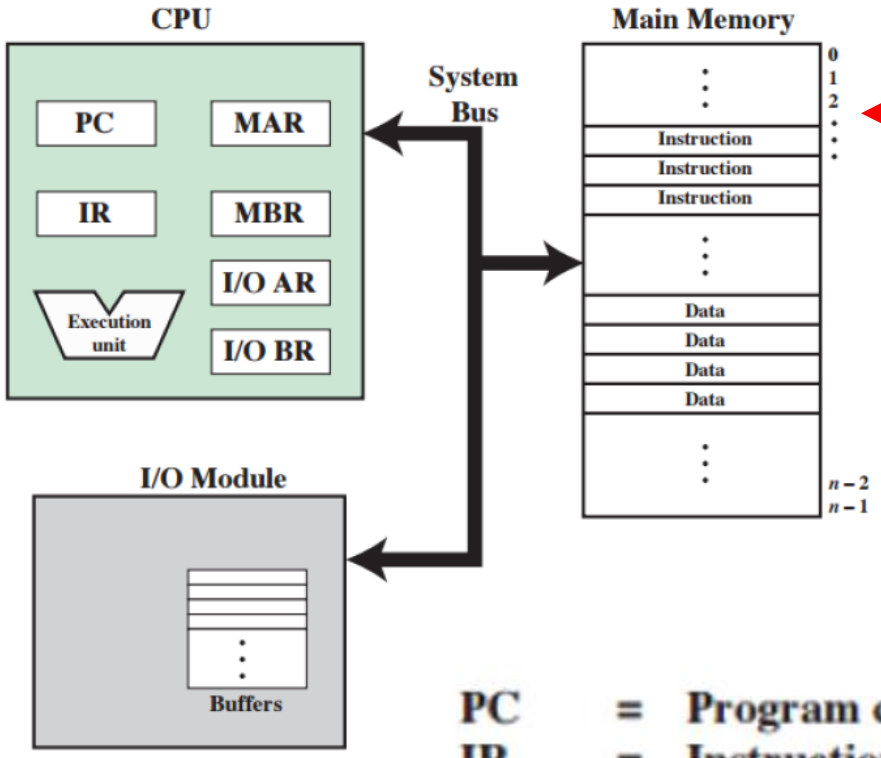


- What could cause the OS to step in?



PC = **Program counter**
IR = **Instruction register**
MAR = **Memory address register**
MBR = **Memory buffer register**
I/O AR = **Input/output address register**
I/O BR = **Input/output buffer register**

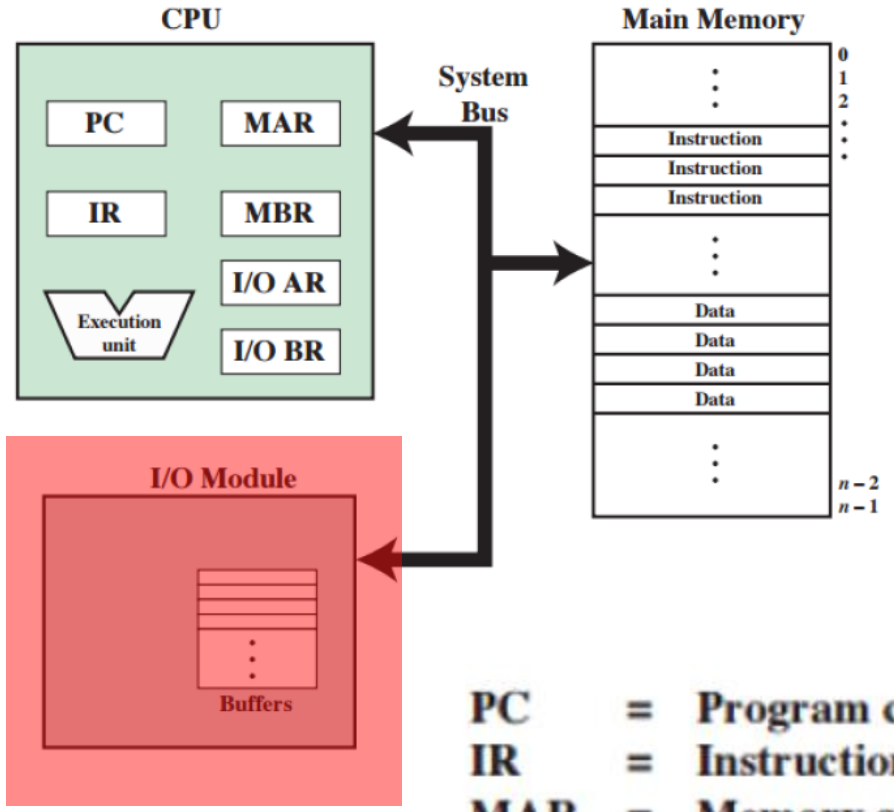
OPERATING SYSTEM FUNCTIONS



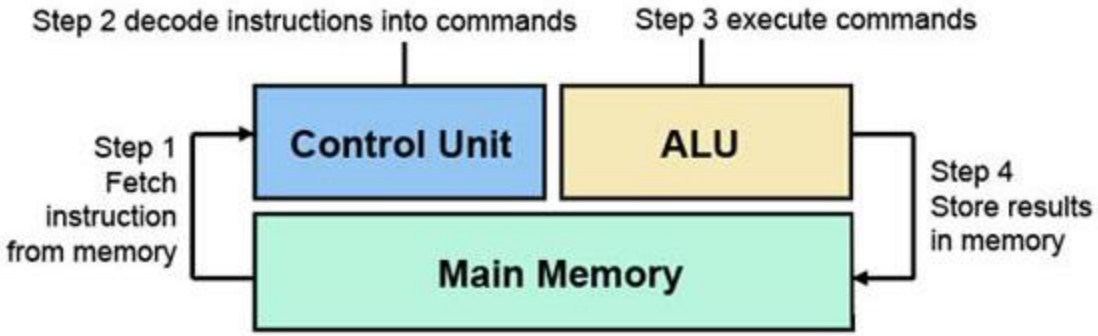
- Data/Instruction not in memory! Need to be retrieved from disk.

PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

OPERATING SYSTEM FUNCTIONS

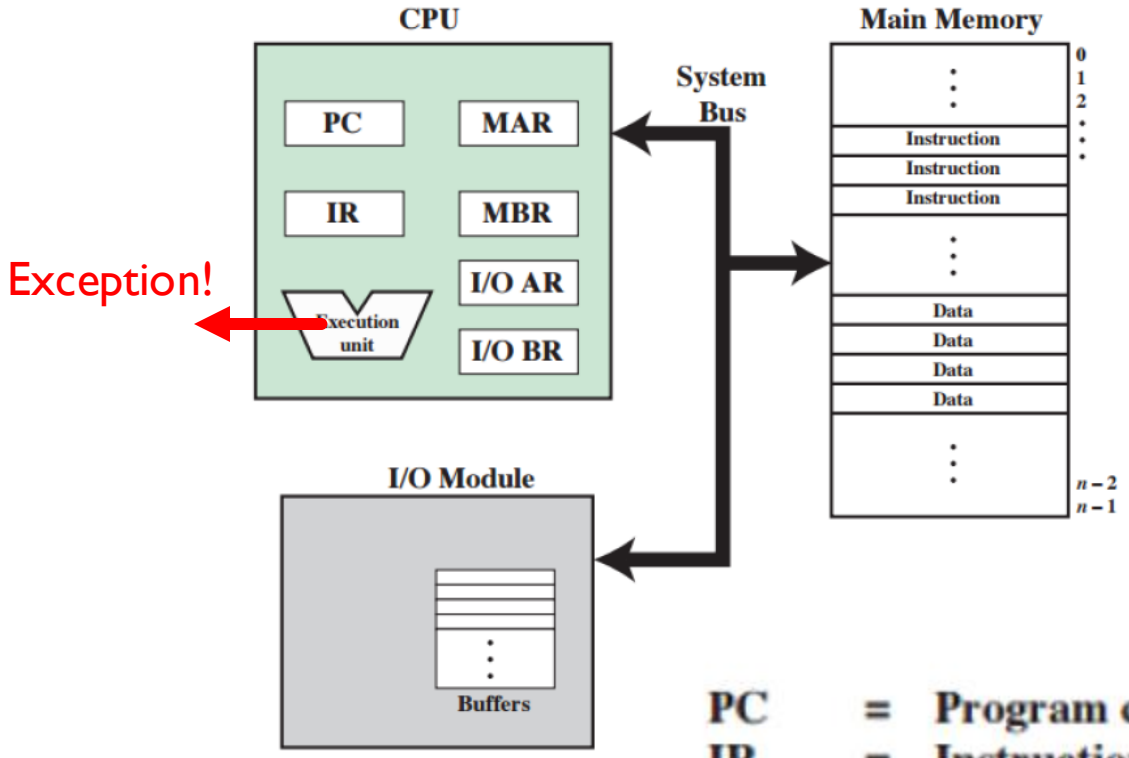


- What about I/O?
- System calls for I/O are made, and execution is handed to OS.



PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

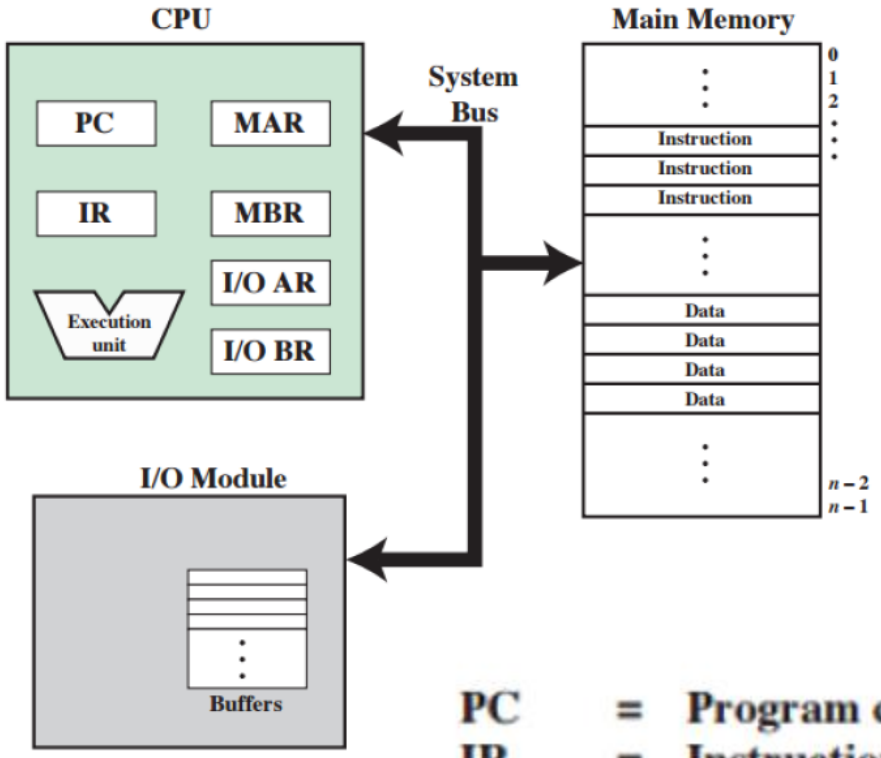
OPERATING SYSTEM FUNCTIONS



- Exceptions can be triggered by the execution unit.
- Ex. Division by zero.

PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

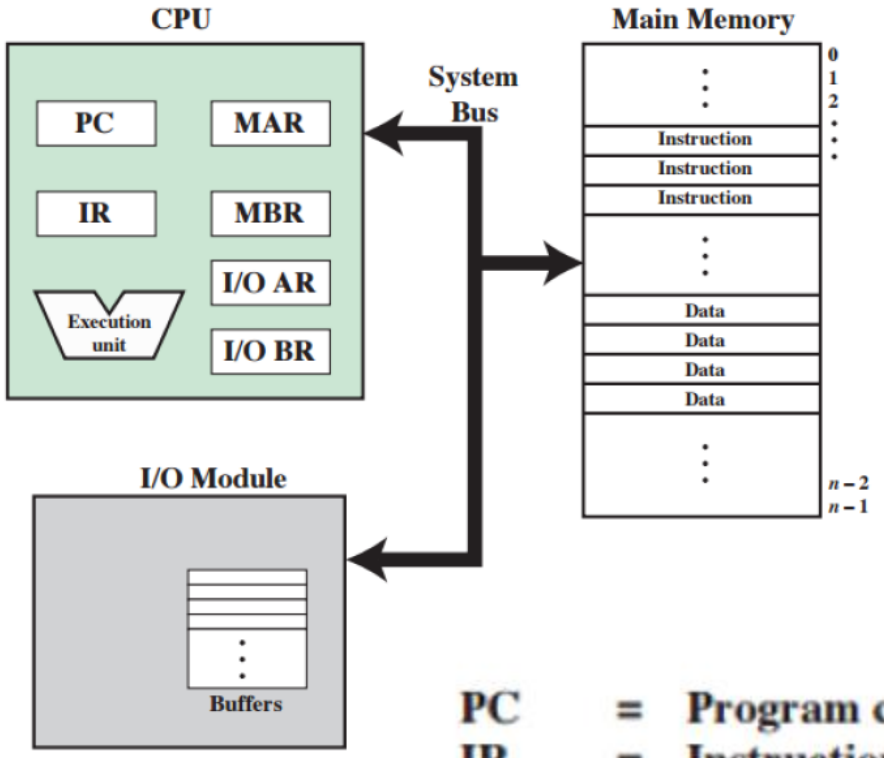
OPERATING SYSTEM FUNCTIONS



PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

- Hardware events:
  - User pressed keyboard key or moved mouse.
  - Network event has occurred.
  - A new device got connected.

OPERATING SYSTEM FUNCTIONS



PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

- Exception/Trap: internal event, caused by the process that is currently executing
- Interrupts: external events.
- Once triggered, control is handed to the **'kernel',** which is the core program running the operating system and running all system calls.

KERNEL MODE

KERNEL MODE

mode bit: 0 / 1

KERNEL MODE

mode bit: 0 / 1

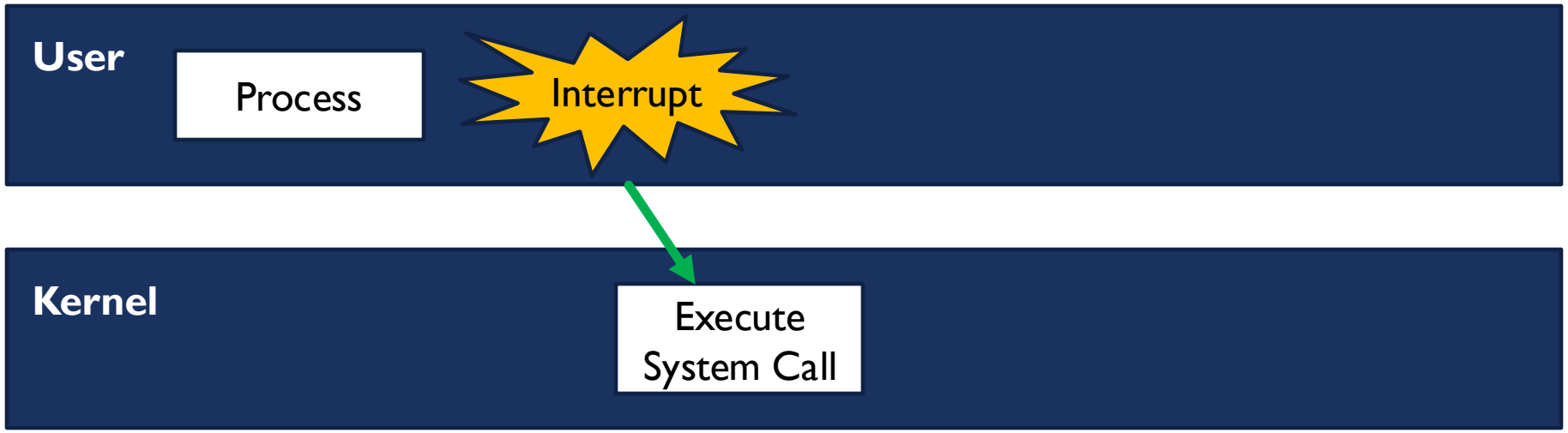
KERNEL MODE

mode bit: 0 / 1

KERNEL MODE

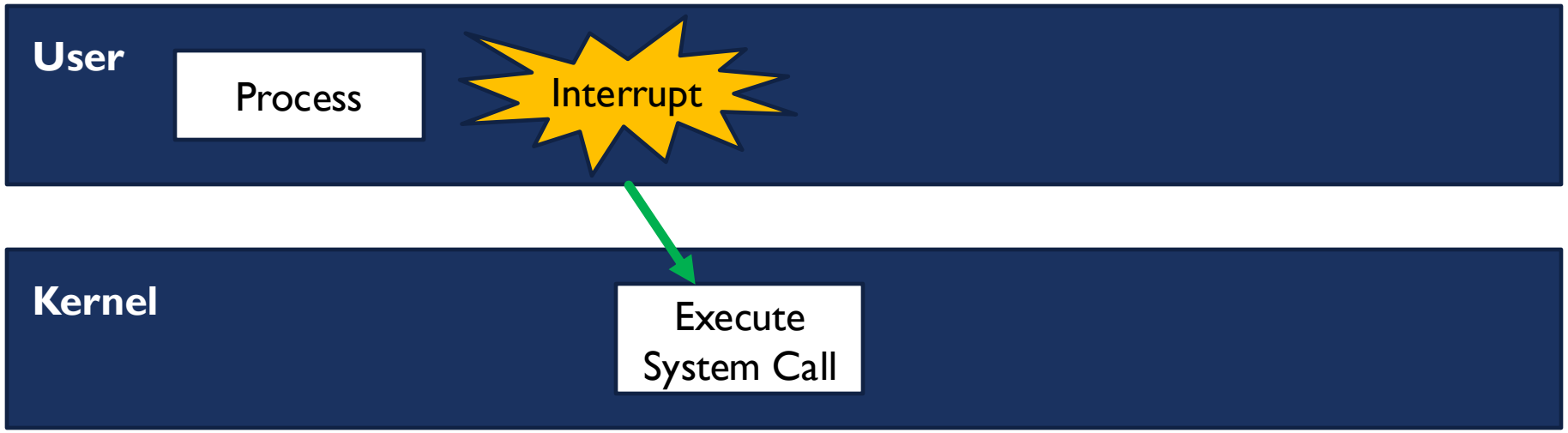
mode bit: 0 / 1

KERNEL MODE



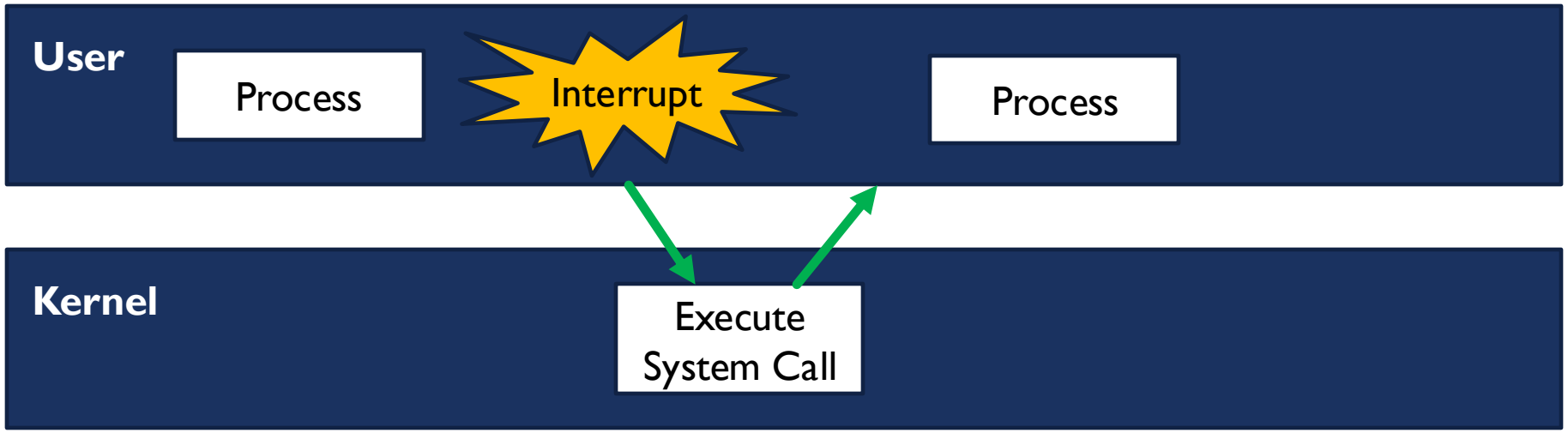
mode bit: 0 / 1

KERNEL MODE



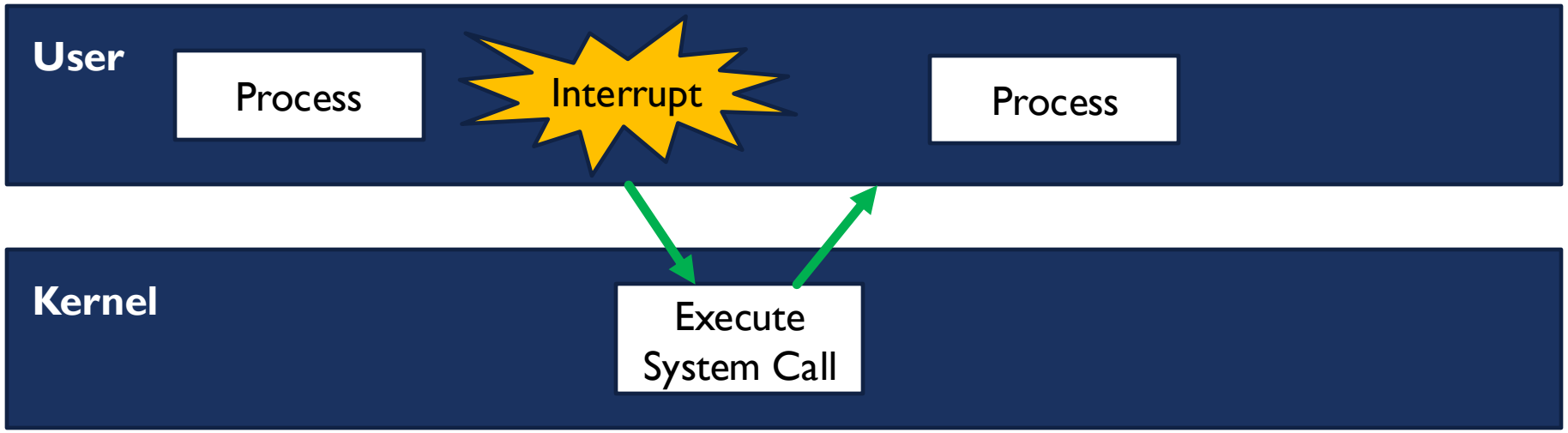
mode bit: 0 / 1

KERNEL MODE



mode bit: 0 / 1

KERNEL MODE



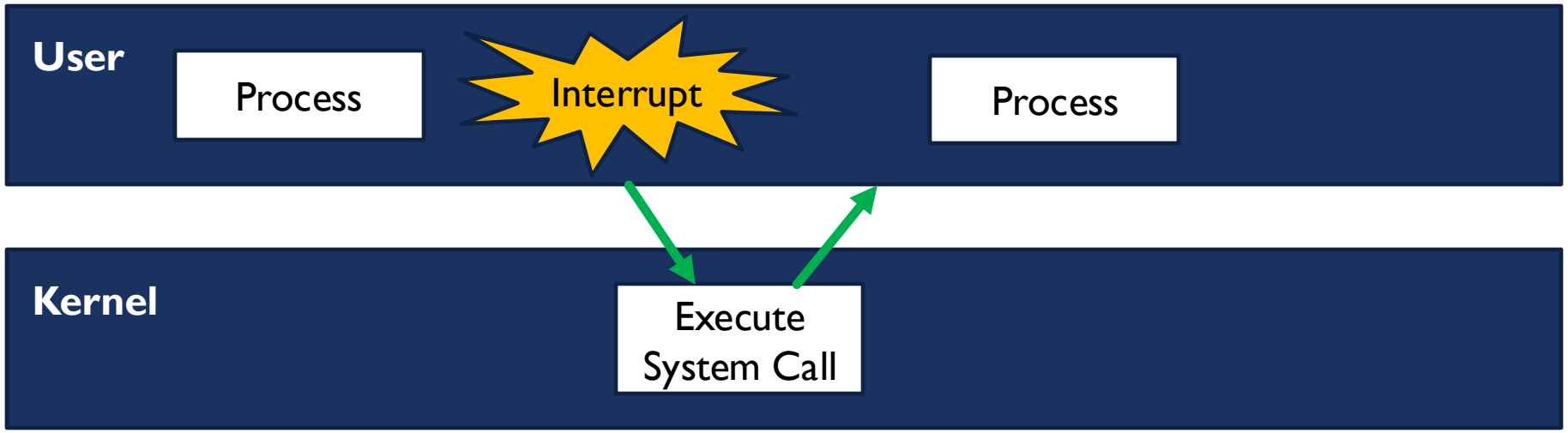
mode bit: 0 / 1

- Q: Running a program as a "Super User / Root" or "Administrator" would set mode bit to '0' i.e. Kernel Mode.

A: True
B: False

A B

KERNEL MODE



mode bit: 0 / 1

- Q: Running a program as a "Super User / Root" or "Administrator" would set mode bit to '0' i.e. Kernel Mode.

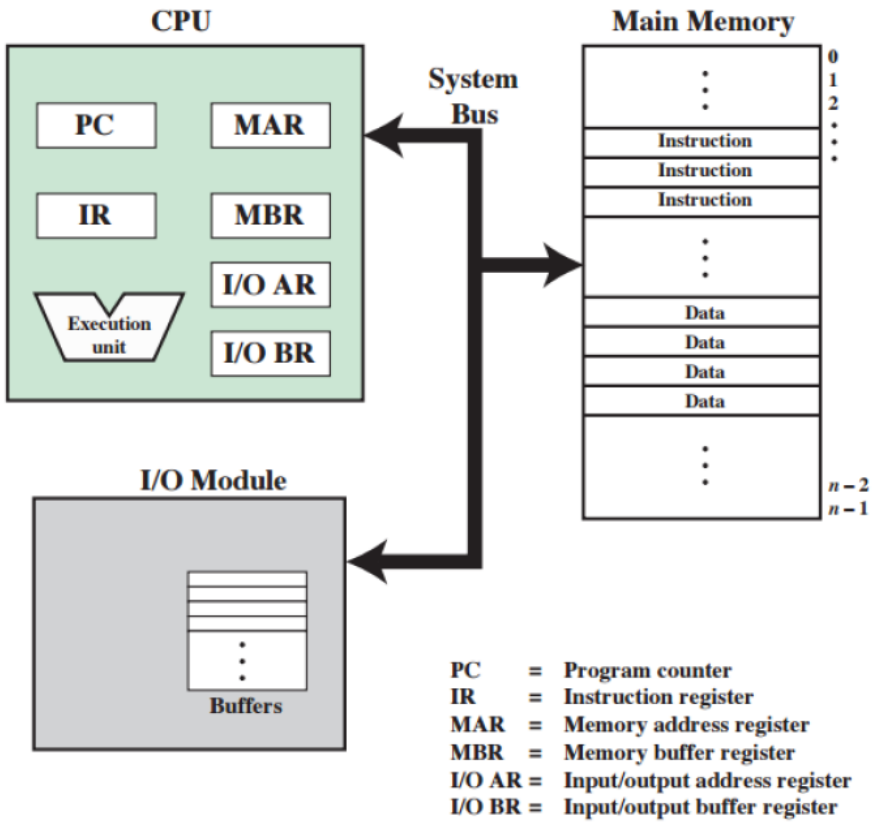
A: True
B: False

A B

KERNEL INSTRUCTIONS

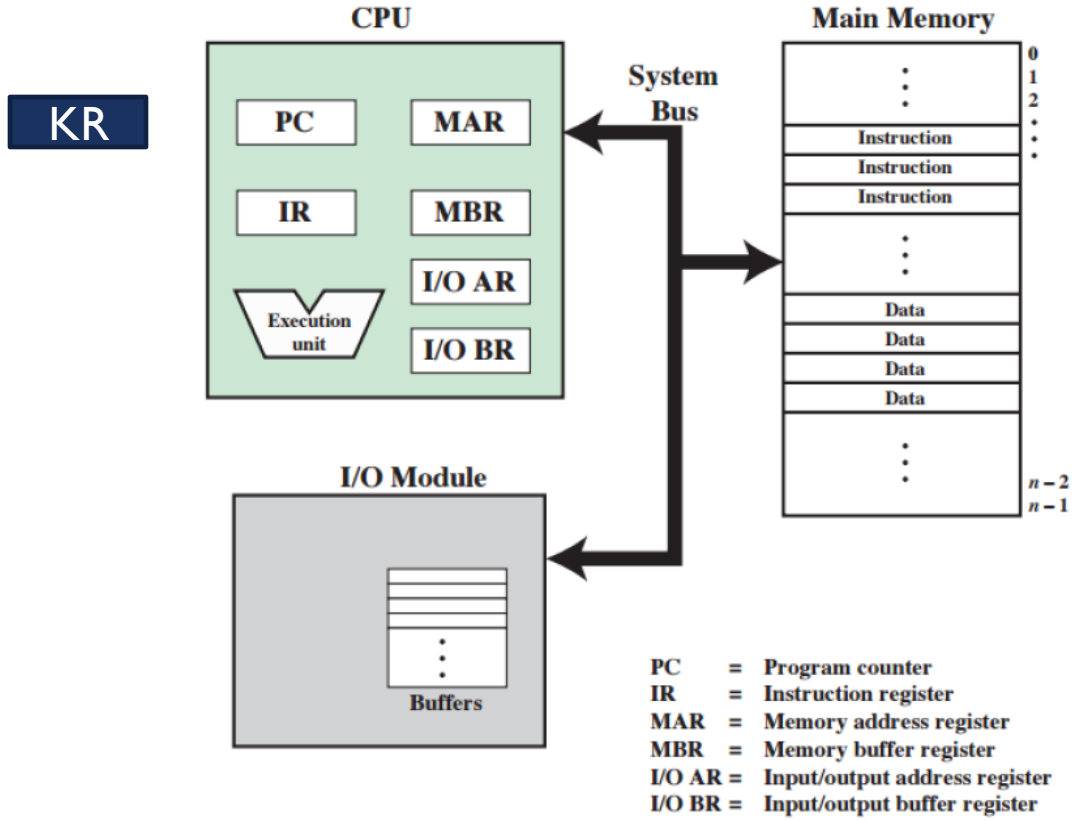
- So what happens exactly when we change the "kernel" mode bit?
- Access to Kernel Registers
- Privileged instructions, with unique OP code can be performed.
- Examples of Privileged Instructions

KR



KERNEL INSTRUCTIONS

- So what happens exactly when we change the "kernel" mode bit?
- Access to Kernel Registers
- Privileged instructions, with unique OP code can be performed.
- Examples of Privileged Instructions
  - I/O instructions and Halt instructions
  - Turn off all Interrupts
  - Set the Timer
  - Context Switching
  - Clear the Memory
  - Remove a process from the Memory
  - Modify entries in Device-status table



REVIEW EXERCISES

- What are the four major components of a computing system?
- What are the three core steps in an instruction cycle?
- What is difference between Interrupts and Exceptions/Traps?
- What does the kernel mode allow access to?
- Can we have multiple processes executing the same program?

REVIEW EXERCISES

- What are the four major components of a computing system?

REVIEW EXERCISES

- What are the four major components of a computing system?
  - Input, Output, Memory, CPU

REVIEW EXERCISES

- What are the three core steps in an instruction cycle?

REVIEW EXERCISES

- What are the three core steps in an instruction cycle?
  - Fetch, Decode, Execute

REVIEW EXERCISES

- What is the difference between Interrupts and Exceptions/Traps?

REVIEW EXERCISES

- What is the difference between Interrupts and Exceptions/Traps?
  - Interrupts: External
  - Exceptions: Internal

REVIEW EXERCISES

- What does the kernel mode allow access to?

REVIEW EXERCISES

- What does the kernel mode allow access to?
  - Privileged Instruction
  - Kernel Registers

REVIEW EXERCISES

- Can we have multiple processes executing the same program?

REVIEW EXERCISES

- Can we have multiple processes executing the same program?
  - Yes!

LOADING THE OPERATING SYSTEM

- The OS handles loading user programs.
- We know that the Operating System is ultimately just a program residing on disk.

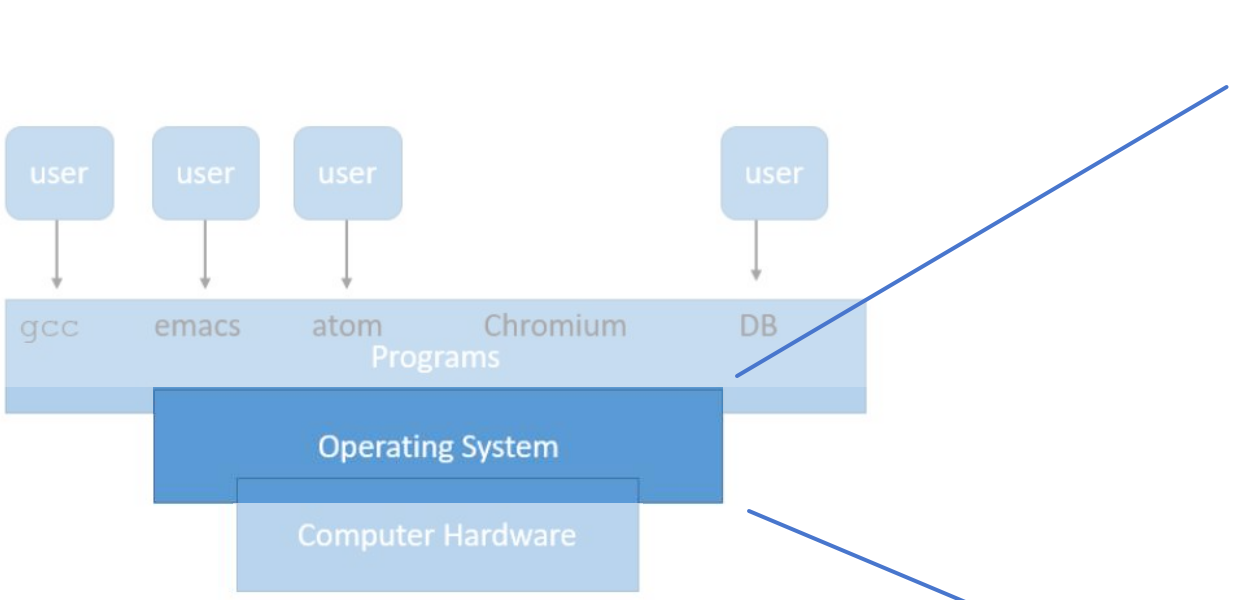
LOADING THE OPERATING SYSTEM

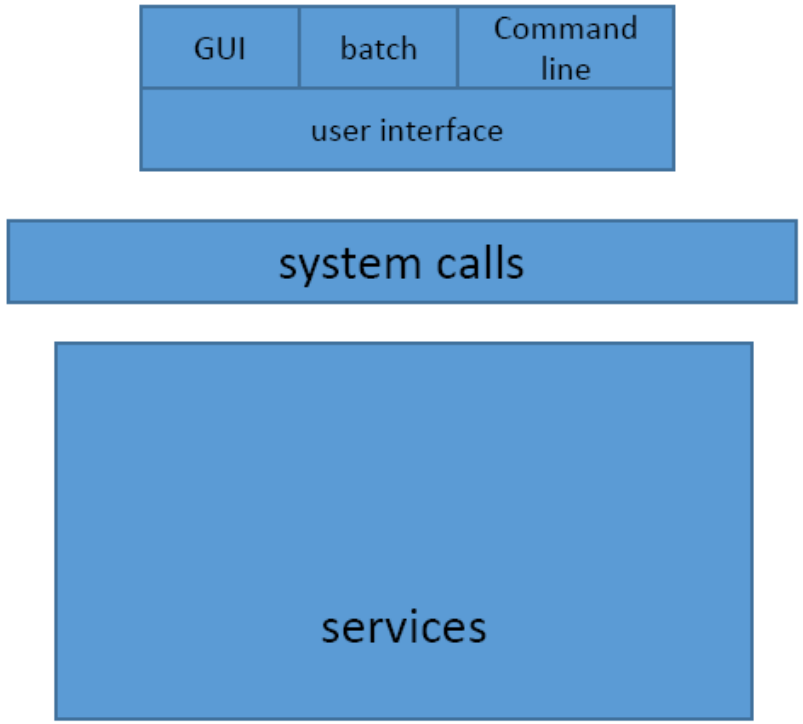
- The OS handles loading user programs.
- We know that the Operating System is ultimately just a program residing on disk.
- Therefore, there must be some other program that loads it:
  - Bootloader
  - Stored on board.

LOADING THE OPERATING SYSTEM

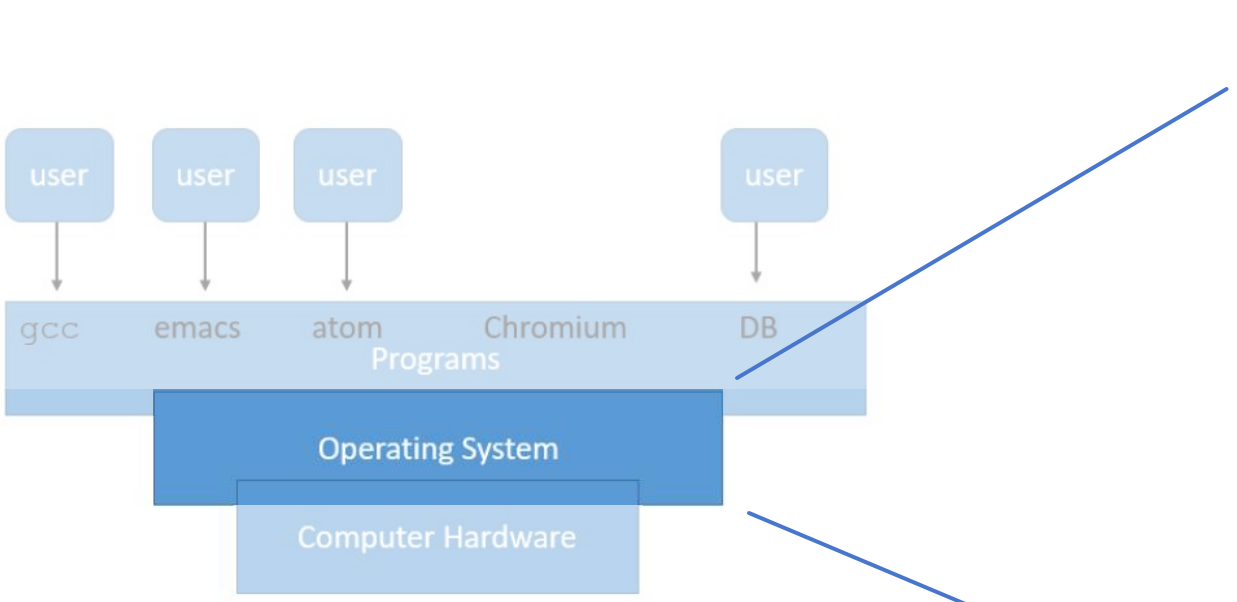
- The OS handles loading user programs.
- We know that the Operating System is ultimately just a program residing on disk.
- Therefore, there must be some other program that loads it:
  - Bootloader
  - Stored on board.
  - **Basic Instructions Operating System (BIOS)** or its modern replacement **Unified Extensible Firmware Interface** (**UEFI**).

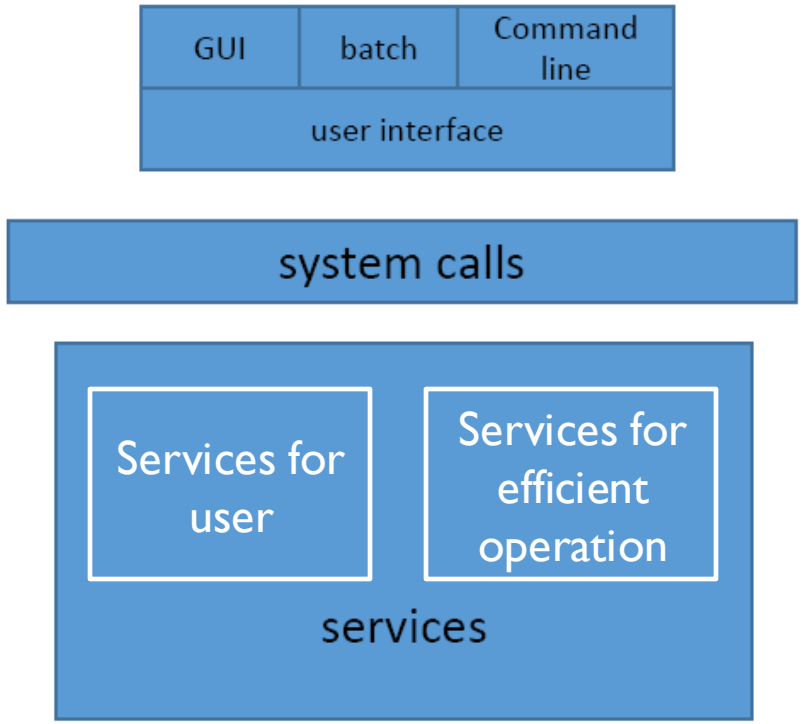
OPERATING SYSTEM SERVICES





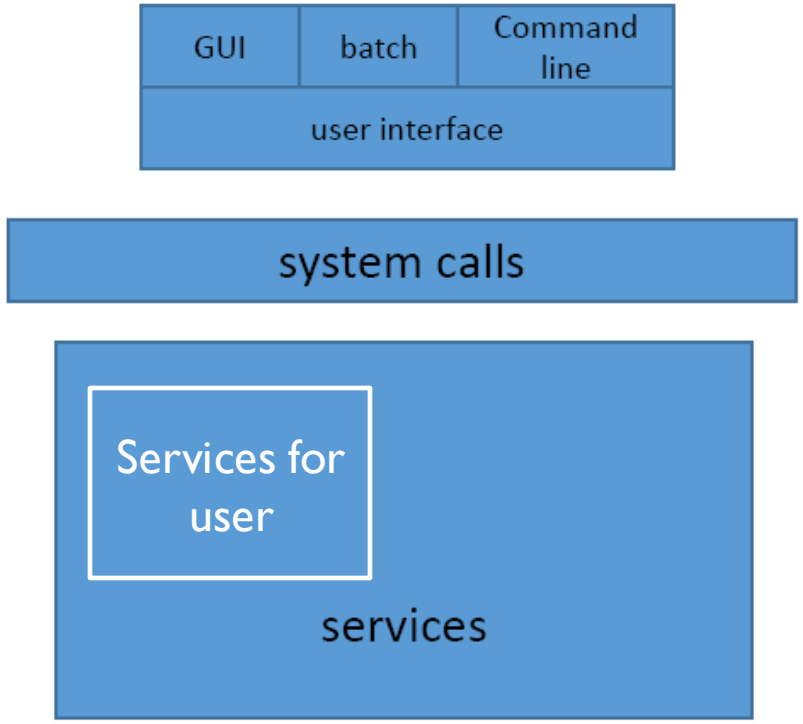
OPERATING SYSTEM SERVICES





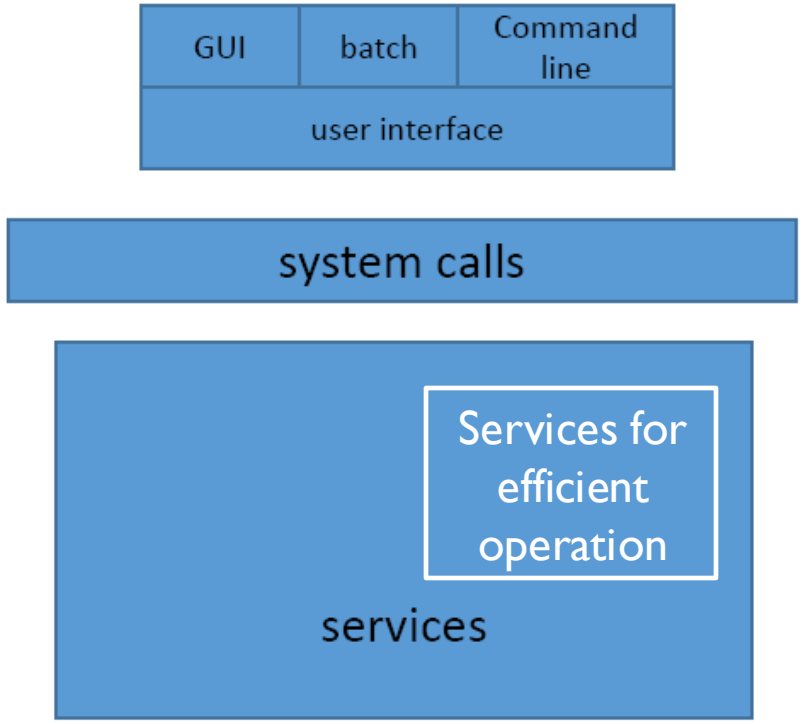
OPERATING SYSTEM SERVICES FOR USERS

- Program Execution: load the program and run it.
- I/O: Programs rarely communicate with I/O devices directly.
- Protection and Security.
  - Protection: User privileges
  - Security: Anti-malware systems
- Error detection.
- User Interface: CLI, GUI, touch



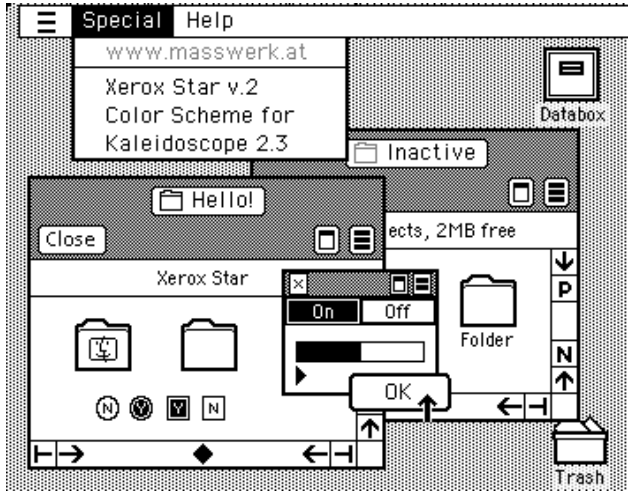
SERVICES FOR EFFICIENT SYSTEM OPERATION

- Resource Allocation
  - Scheduling Algorithms
  - Memory Allocation Algorithms
- Accounting
  - Which user/process uses how much of a specific resource.
  - Storage space, Memory, CPU access …
- Protection and Security



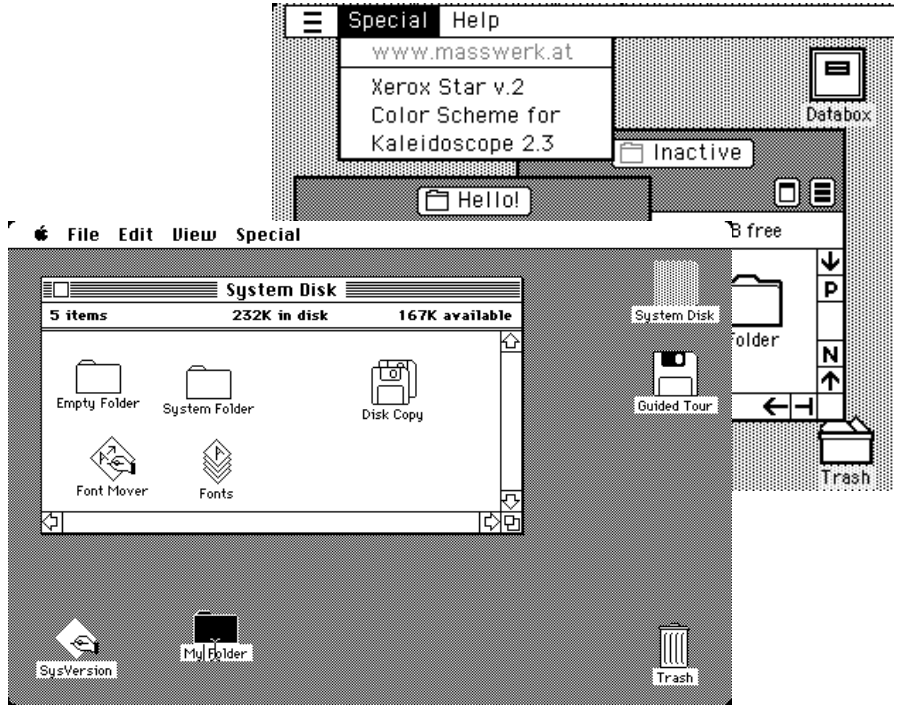
GUI HISTORY

- First commercial Computer offered with GUI was the Xerox Star.
- It failed mainly due to its very high cost and poor interest from buyers.



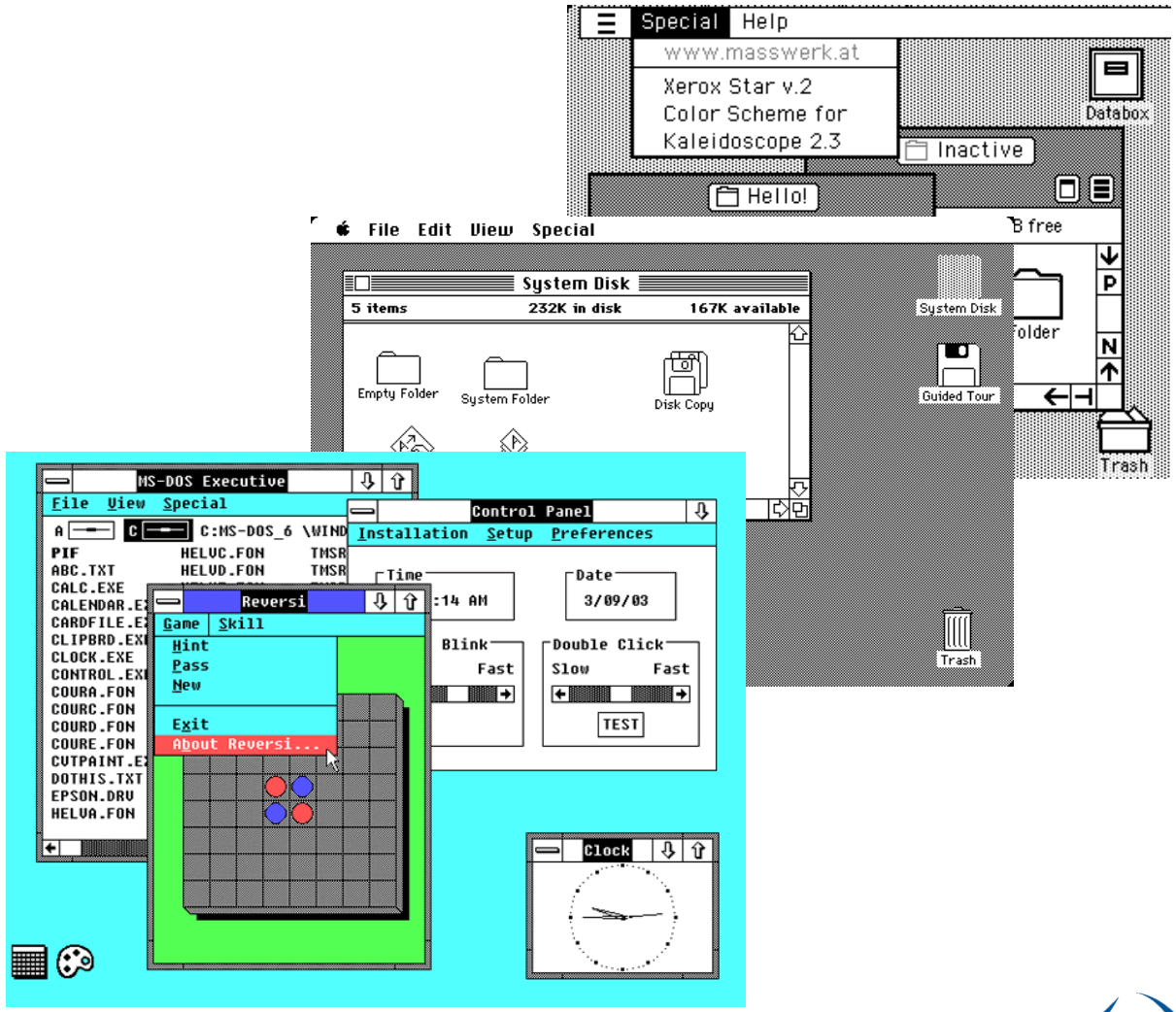
GUI HISTORY

- First commercial Computer offered with GUI was the Xerox Star.
- It failed mainly due to its very high cost and poor interest from buyers.
- Macintosh "System 1"offered a rather successful GUI system.
- Many other GUIs were offered by various manufacturers.



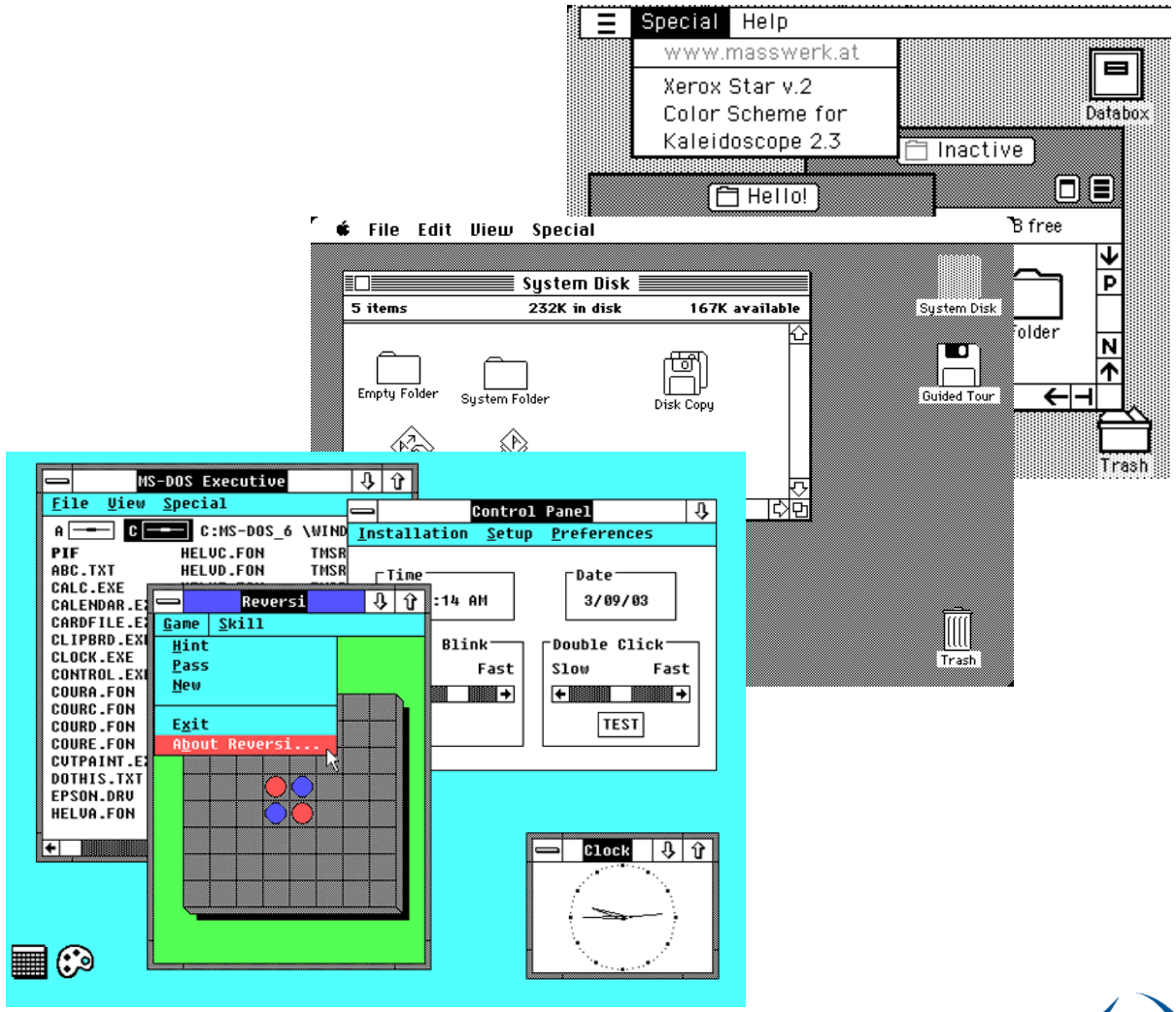
GUI HISTORY

- First commercial Computer offered with GUI was the Xerox Star.
- It failed mainly due to its very high cost and poor interest from buyers.
- Macintosh "System 1"offered a rather successful GUI system.
- Many other GUIs were offered by various manufacturers.
- Windows 2.0 offered the first “proper” GUI from Microsoft.



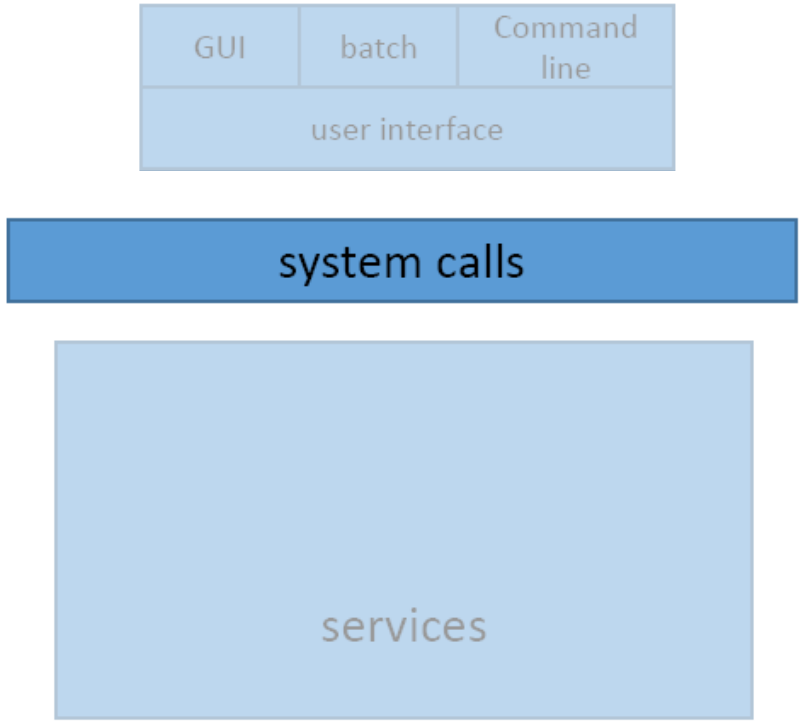
GUI HISTORY

- First commercial Computer offered with GUI was the Xerox Star.
- It failed mainly due to its very high cost and poor interest from buyers.
- Macintosh "System 1"offered a rather successful GUI system.
- Many other GUIs were offered by various manufacturers.
- Windows 2.0 offered the first “proper” GUI from Microsoft.
- Interesting fact: Apple sued Microsoft in 1988 for copying the “look and feel” of Mac. Apple lost the case.



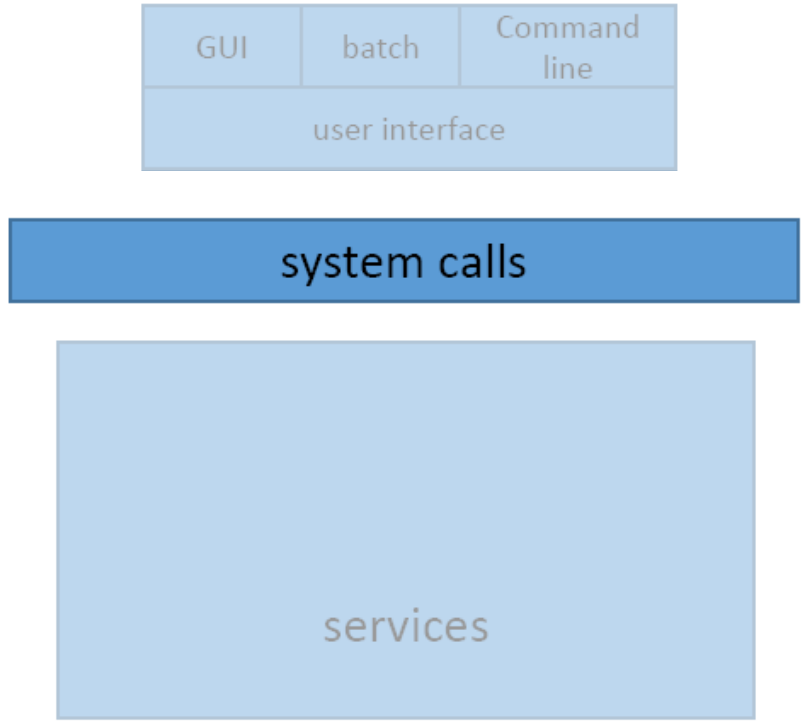
SYSTEM CALLS

- How are OS services performed?
- Services are performed through system calls.



SYSTEM CALLS

- How are OS services performed?
- Services are performed through system calls.
- Not very different than other programs except that most run in kernel mode.
- They are either called explicitly or invoked through interrupts/exceptions.



SYSTEM CALLS

- How are OS services performed?
- Services are performed through system calls.
- Not very different than other programs except that most run in kernel mode.
- They are either called explicitly or invoked through interrupts/exceptions.

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix

Process control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File management	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device management	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

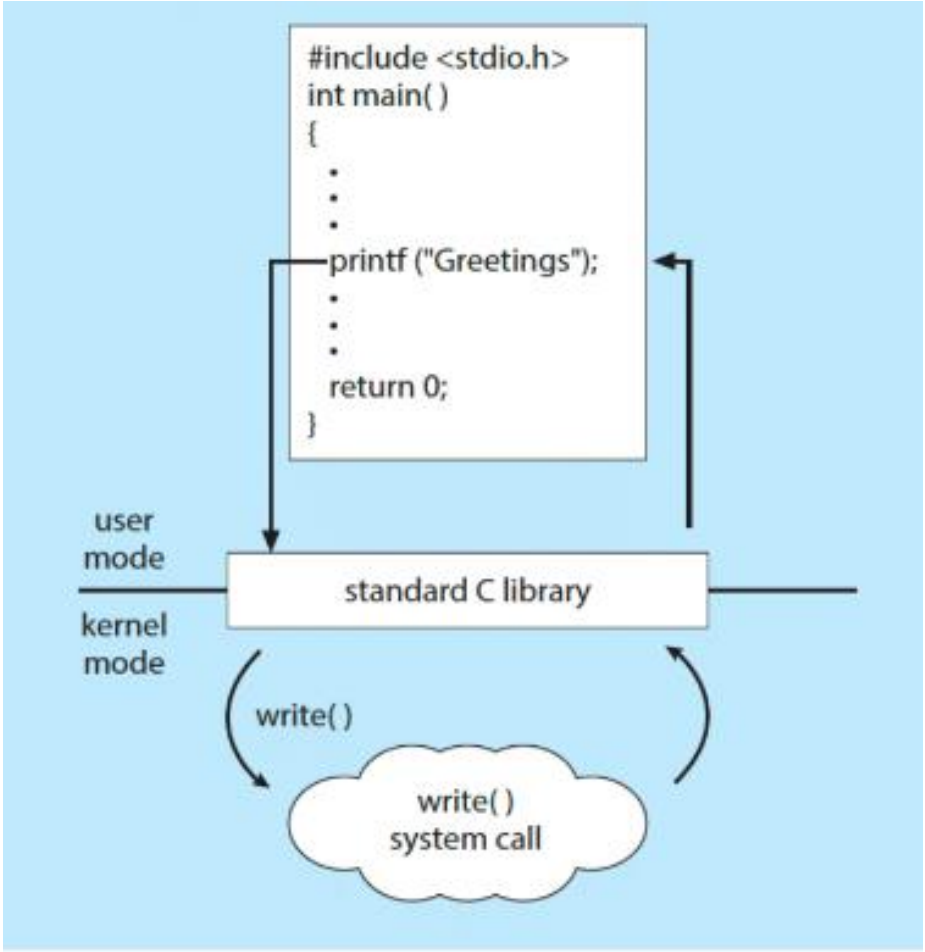
Information maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communications	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

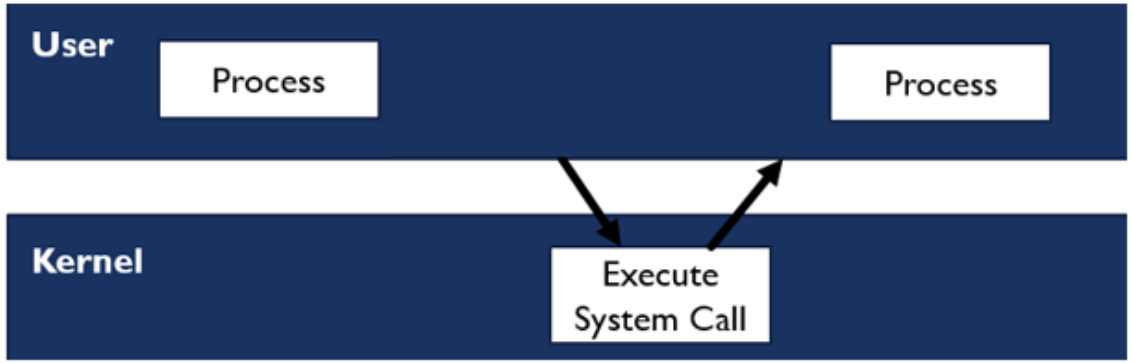
EXAMPLE OF IMPLICIT SYSTEM CALL

- Standard C library would call appropriate system call functions when needed.



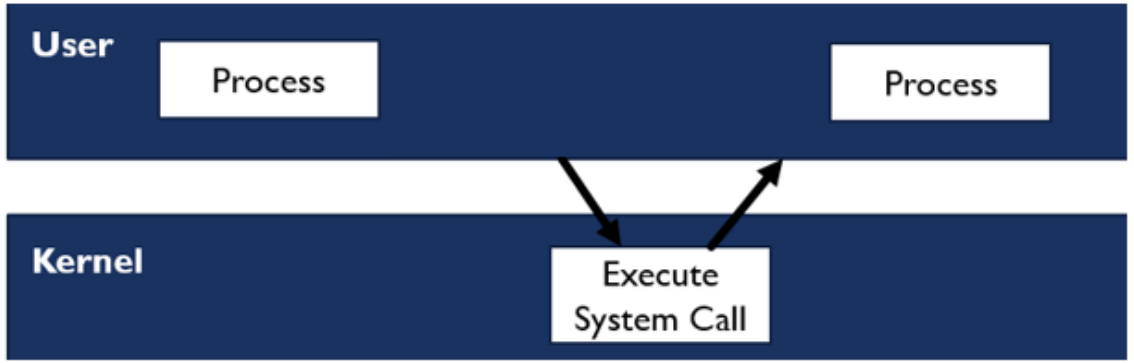
SYSTEM CALLS: PARAMETER PASSING

- Example: program initiates a system call to print text on screen.
- It has to pass the "text" to the kernel to do the printing.



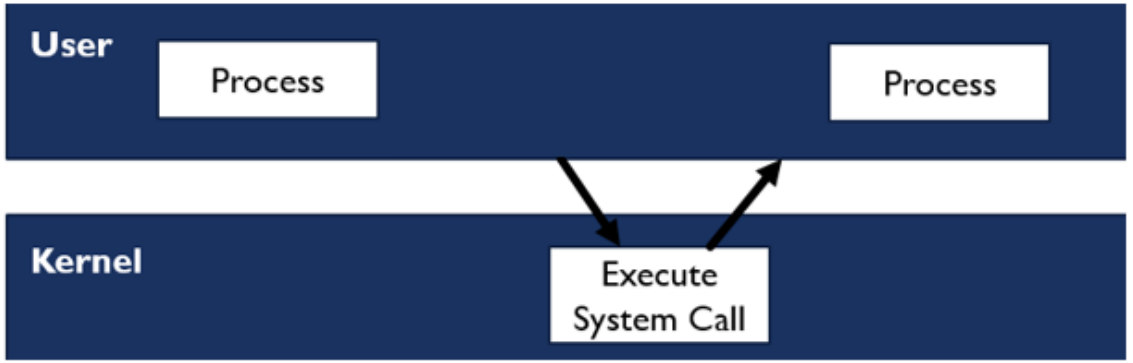
SYSTEM CALLS: PARAMETER PASSING

- Example: program initiates a system call to print text on screen.
- It has to pass the "text" to the kernel to do the printing.
- How is that done?



SYSTEM CALLS: PARAMETER PASSING

- Example: program initiates a system call to print text on screen.
- It has to pass the "text" to the kernel to do the printing.
- How is that done?




```
71  void addToList(int procNum, int initCount) {
72      newProc* node = (newProc*) malloc(sizeof(proc));
73      node->proc = procNum;
74      node->count = initCount;
75      if (list == NULL) {
76          node->next = node;
77          node->prev = node;
```

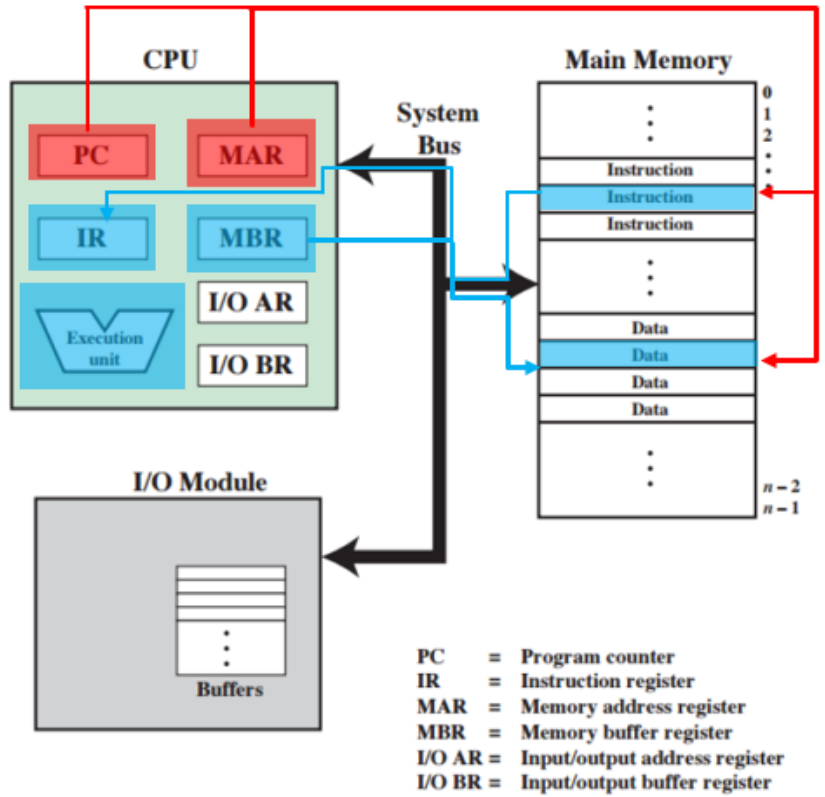
From a programmer perspective, "arguments" are passed very easily.

SYSTEM CALLS: PARAMETER PASSING

- Example: program initiates a system call to print text on screen.
- It has to pass the "text" to the kernel to do the printing.
- How is that done?

```
71  void addToList(int procNum, int initCount) {
72      newProc* node = (newProc*) malloc(sizeof(proc));
73      node->proc = procNum;
74      node->count = initCount;
75      if (list == NULL) {
76          node->next = node;
77          node->prev = node;
```

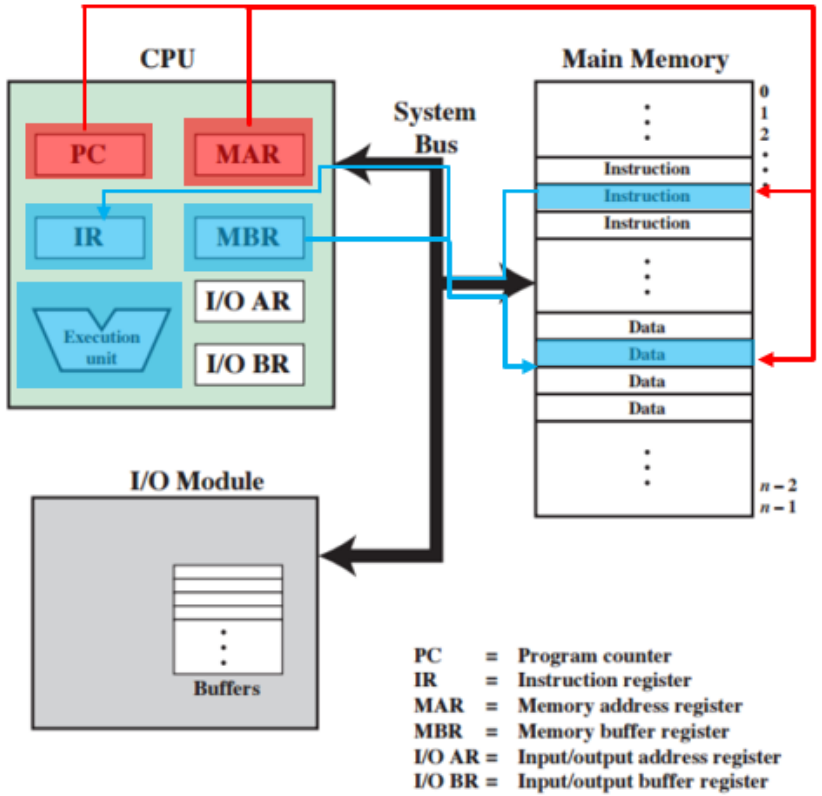
From a programmer perspective, "arguments" are passed very easily.



From a hardware perspective, things are a bit more complicated.

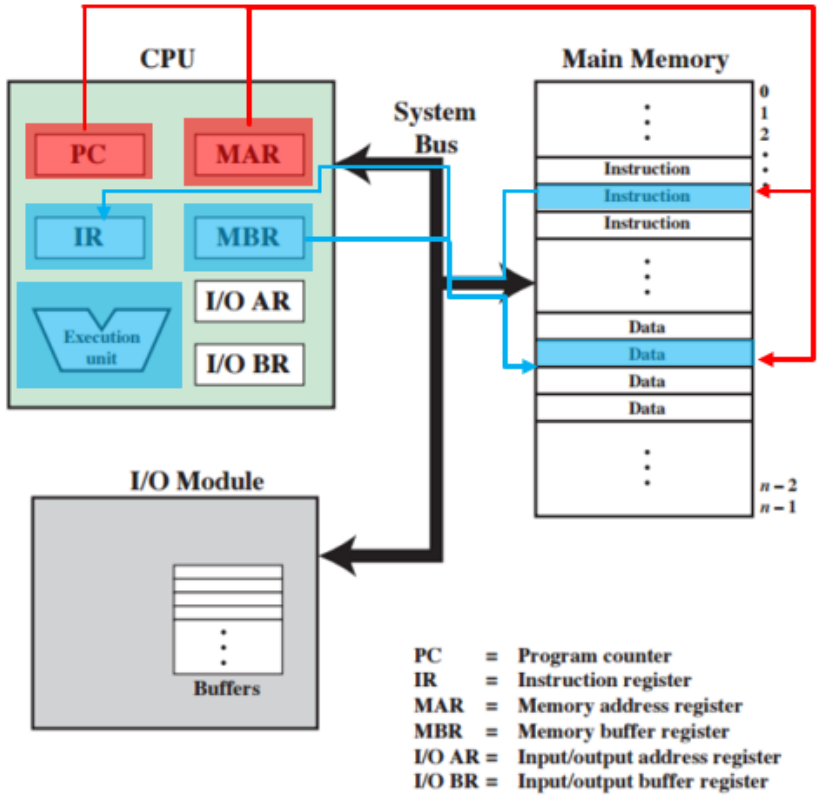
SYSTEM CALLS: PARAMETER PASSING

- Actual computer hardware: bunch of registers with access to memory … not enough "fields" for passing all parameters.
- Instructions are performed one at a time



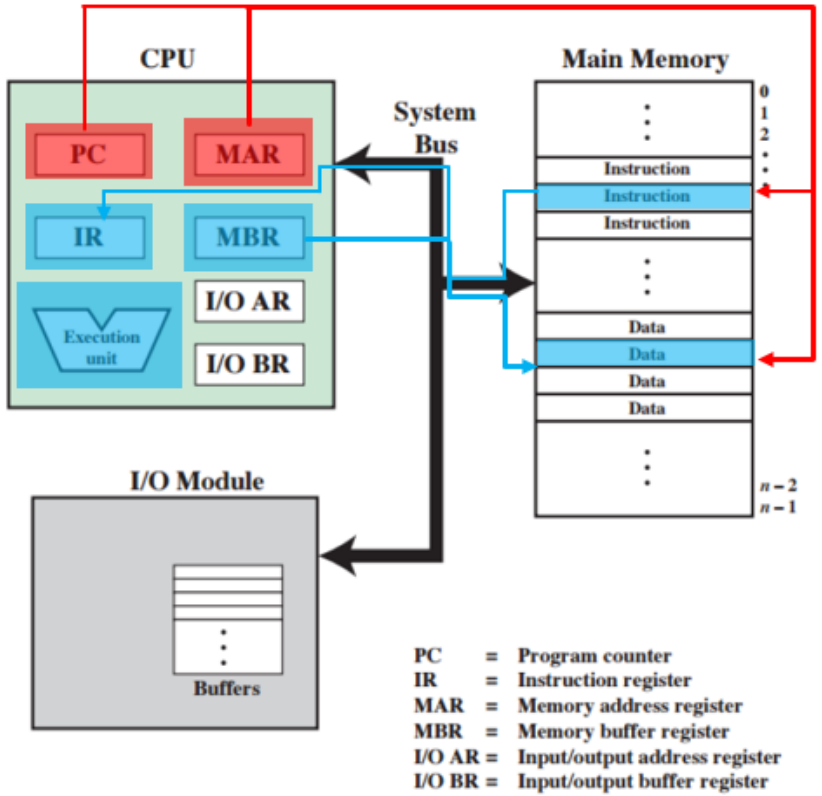
SYSTEM CALLS: PARAMETER PASSING

- Actual computer hardware: bunch of registers with access to memory … not enough "fields" for passing all parameters.
- Instructions are performed one at a time.
- Calling a "function" or doing a system call is simply "jumping" to different PC.



SYSTEM CALLS: PARAMETER PASSING

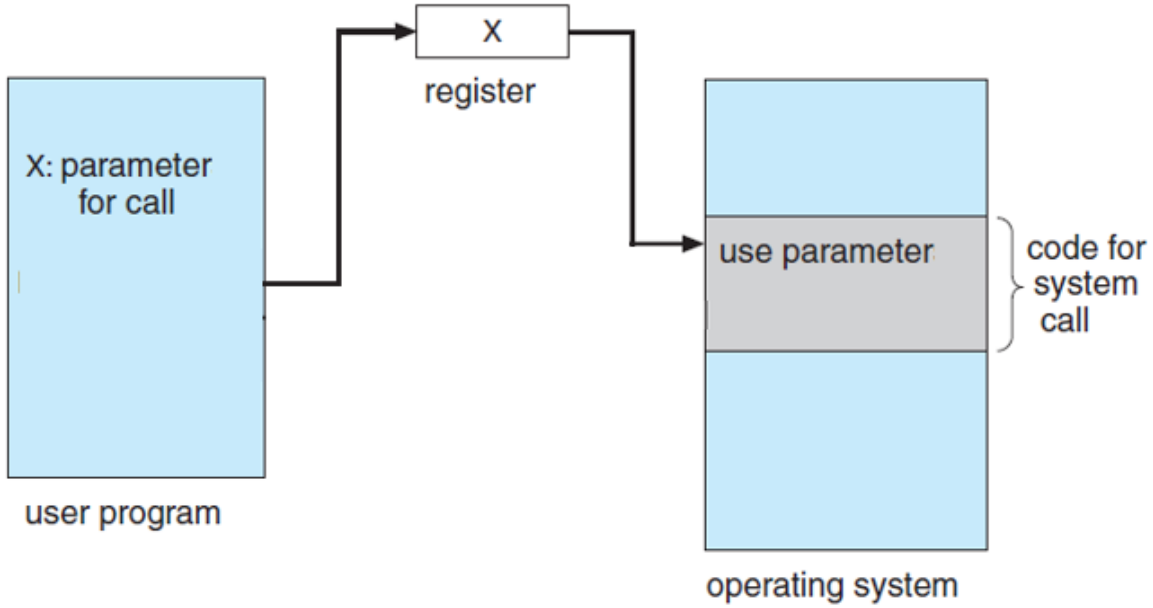
- Actual computer hardware: bunch of registers with access to memory … not enough "fields" for passing all parameters.
- Instructions are performed one at a time



- Worksheet Q1:
- Think and list a couple of approaches that we can use to pass parameters from program to kernel.

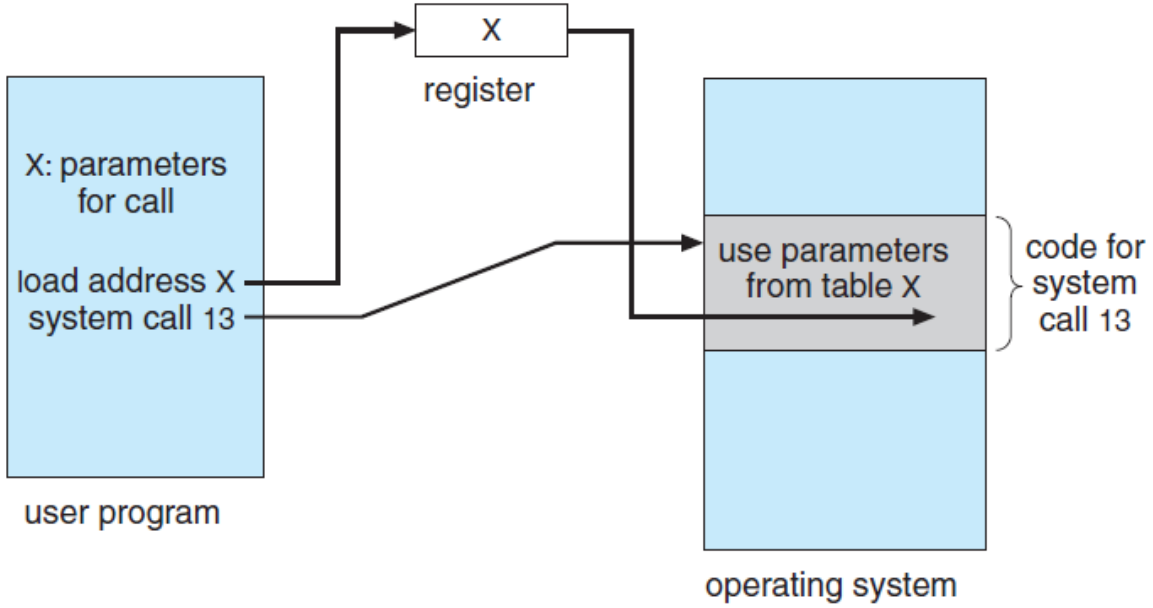
SYSTEM CALLS: PARAMETER PASSING

- Three methods for passing parameters:
- Saved into registers.



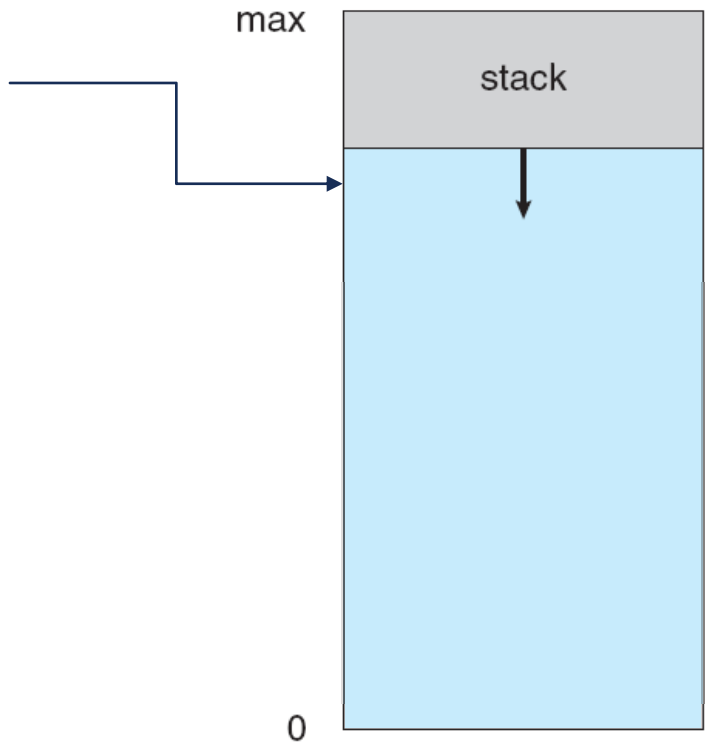
SYSTEM CALLS: PARAMETER PASSING

- Three methods for passing parameters:
- Saved into registers.
- As a table with table address passed through register.



SYSTEM CALLS: PARAMETER PASSING

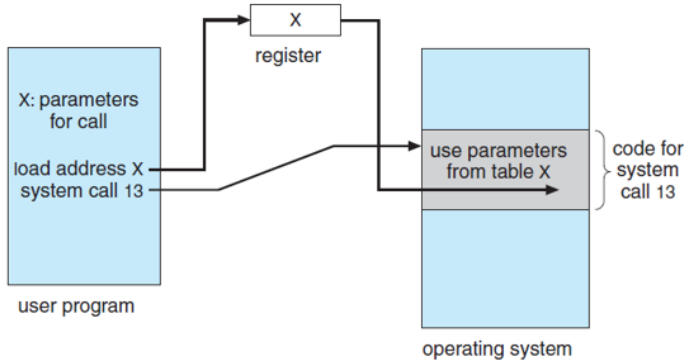
- Three methods for passing parameters:
- Saved into registers.
- As a table with table address passed through register.
- Pushed onto the stack where it is popped by the called function.

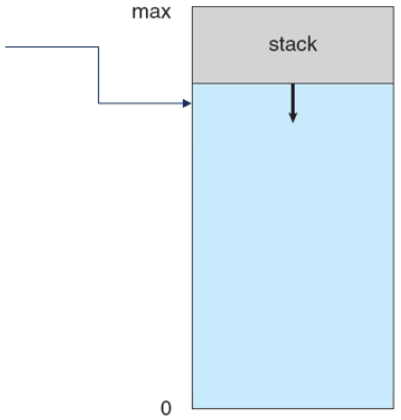


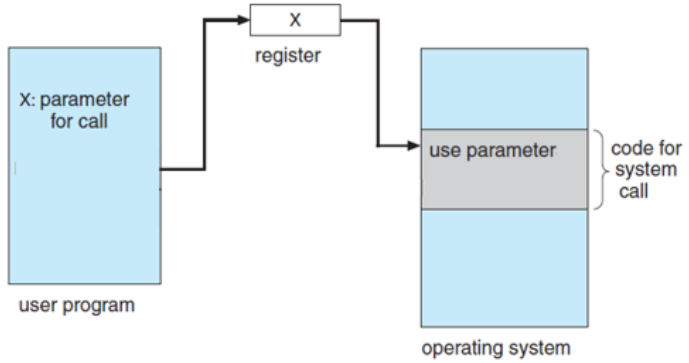
SYSTEM CALLS: PARAMETER PASSING

- Q: Who decides how to pass the parameters?
  - A: The program performing the system call.
  - B: The system call.

A B







SYSTEM CALLS: PARAMETER PASSING

- Q: Who decides how to pass the parameters?
  - A: The program performing the system call.
  - B: The system call.

A B