Introduction to Operating Systems
Lect. 1 - Agenda

- Course outline and policies
- Computer-System Organization - review

General Information:

Instructor : Omar Mansour, Ph.D.

Email : omansour@nyu.edu

Office : 370 Jay street, room 844

Office Hours : by appointment.

Credit Hours : 3

Class sessions - section 1: Mondays, 6-8:30 pm, rm 909

Class sessions – section 12: Wednesdays 6-8:30 pm, rm 801

Teaching assistants:

TBA

Required Text Book:

Silberschatz, Galvin and Gagne, ***Operating System Concepts***, 9th edition, Wiley.

Course Description:

This course covers the functions and organization of operating systems, including process management, memory management, resource allocation, input/output systems, and information protection.

Required pre-requisites:

1. C or C++ language programming (CS-UG 2124)

Recommended pre-requisites:

1. Computer architecture (CS-UG 2214 or CS – GY 6133)
2. Data structures and algorithms (CS-UG 1134 or CS – GY 5403), particularly stacks, queues, trees and linked lists.

Course Objectives:

1. Acquiring fundamental knowledge and proficiency in modern operating system **design**.
2. Learning how to **use utilities provided** by modern operating systems in developing reliable applications that can interact with the system and with other local or remote applications.

Grading (graduate sections):

Assignments and quizzes : 30%
Mid-term exam : 35%
Final exam : 35%

Grading range:

Grade letter	Percentage of available points
A	93-100
A-	88-92
B+	82-87
B	76-81
B-	70-75
C+	60-69
C	50-59

Attendance and participation policy:

- Attendance and participation includes attendance, class participation (e.g. answering questions posed by the instructor), quizzes, in-class assignments, etc.
- If I notice a significant amount of class absence or lack of participation in assignments, quizzes, etc. , I may notify you by email, and it may result in failing or being withdrawn from the course.

Assignments policy:

1. Assignments must be submitted on or before 11:59 pm on the day they are due.
2. Late assignments will not be permitted.
3. Students are required to perform the work pertaining to the assignments **alone.** This includes programming assignments. Students however are encouraged to discuss the concepts pertaining to the course or the assignments with other students or with their teaching assistants, while doing the actual work themselves.
4. Copying of code or answers to homework questions is an act of plagiarism. If the teaching assistant suspects any type of cheating or plagiarism, he/she may contact the student involved and discuss his/her work.

Syllabus (tentative):

Week	Description
1	Introduction
2	Operating systems architecture
3	Processes and Operating System data structures – 1
4	Processes and Operating System data structures – 2
5	Inter-process communications
6	Threads
7	Synchronization
8	Deadlocks
9	Midterm exam – **tentative schedule**
10	Scheduling
11	Memory management
12	Virtual memory
13	Disk Management
14	I/O and file systems
15	Final exam

Academic Honesty:

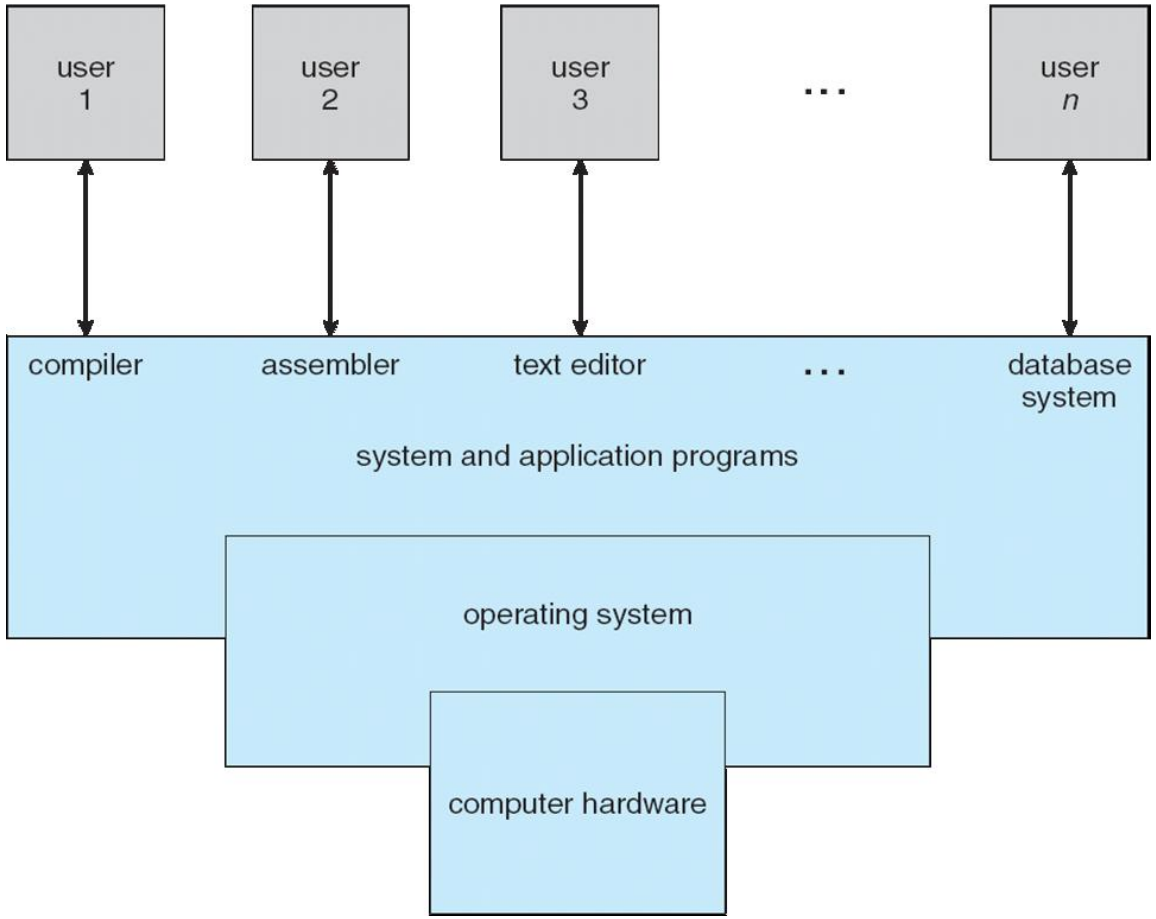
- Students at NYU are expected to be **honest** and forthright in their academic endeavors.
- Academic dishonesty includes cheating, unapproved collaboration, coercion, inventing false information or citations, plagiarism, tampering with computers, destroying other people's coursework, lab or studio property, theft of course materials, or other academic misconduct. If you have questions regarding this policy, contact your professor *prior* to submitting the work for evaluation. See your academic catalogue for a full explanation.
- All students must adhere to the NYU Tandon school of engineering's "Student Code of Conduct", https://engineering.nyu.edu/campus-and-community/student-life/office-student-affairs/policies/student-code-conduct

Academic Honesty (cont.):

- All assignments, unless otherwise explicitly listed, are to be done independently, unless explicit permission from the instructor is provided.
- Anyone caught cheating in this course will receive a “0” on the assignment/assessment and the professor additionally retains the option of significantly reducing the final grade. If a student is caught a second time in, he/she shall fail the course.

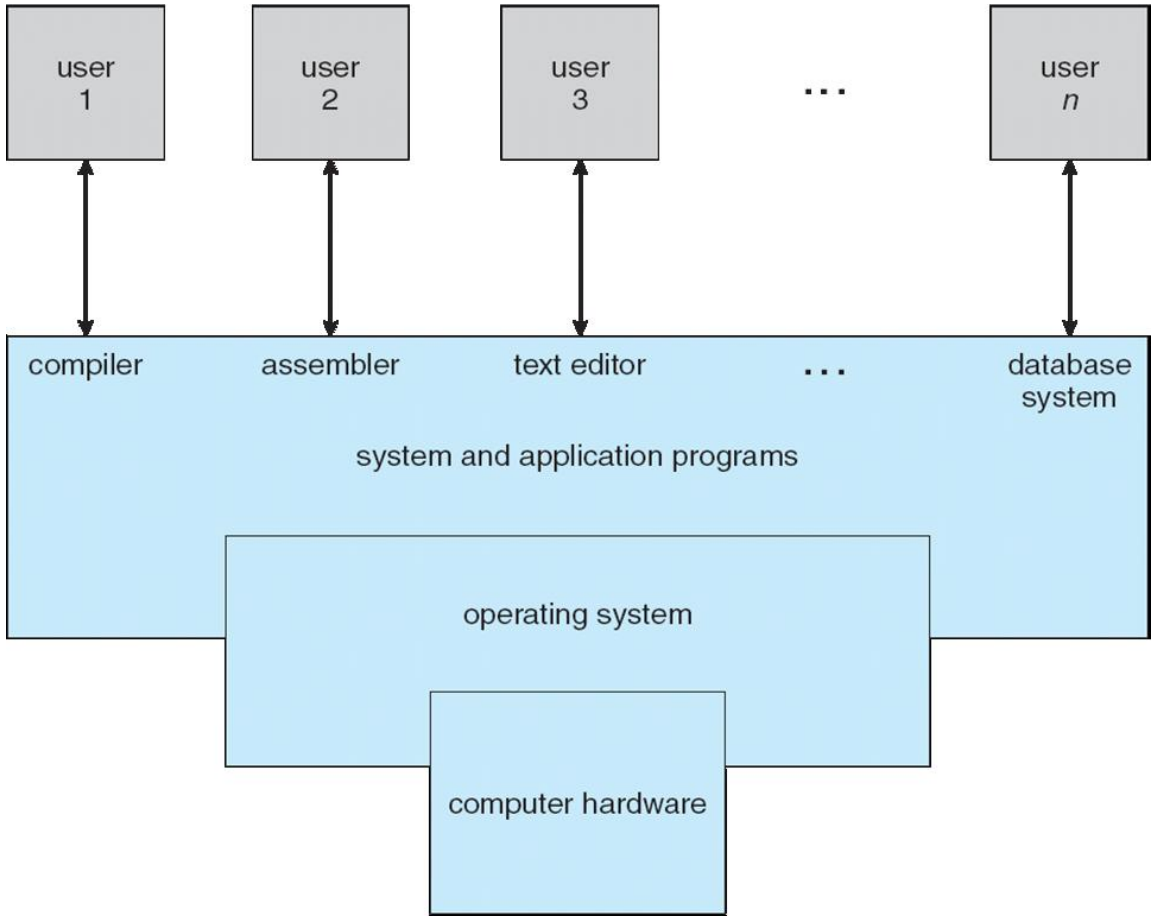
Computer System Structure

- A Computer system can be divided into **three** components:
  - **Hardware** – provides basic computing resources
    - CPU, memory, I/O devices
  - **Operating system**
    - Controls and coordinates use of hardware among various applications and users



Computer System Structure

- **System and Application programs** – Use system resources (e.g. CPU, memory, etc.) to solve the computing problems of the users
  - Example application programs: Word processors, web browsers, database systems, video games
  - Example of system programs: compilers, shell programs, window managers, etc.



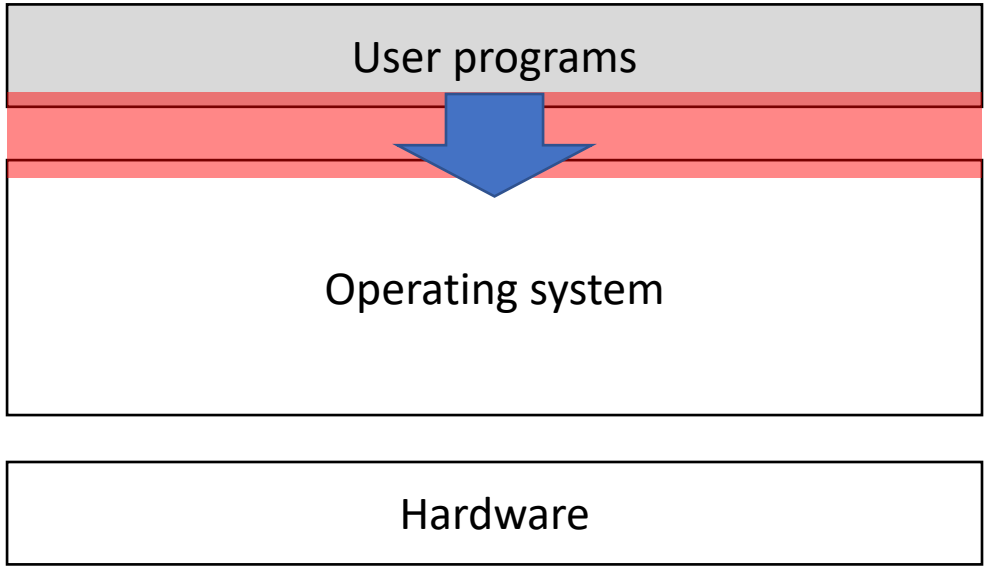
What is an Operating System?

- A program that acts as an intermediary between **application/system programs** and the **computer hardware**
- Users may interact with the computer via the command-line interface (CLI) or Graphics user interface (GUI). Both may be considered **system programs**.
- Applications, and system programs, are also be referred to as **user-mode programs**,
  - Because they run in an **unprivileged mode** (**user mode**), whereas the operating system runs in a **privileged mode** (or **kernel mode**).

- Operating system goals:
  1. Providing an environment for executing user programs and facilitating their design.
  2. Use the computing resources (hardware and software) in an efficient and secure manner.

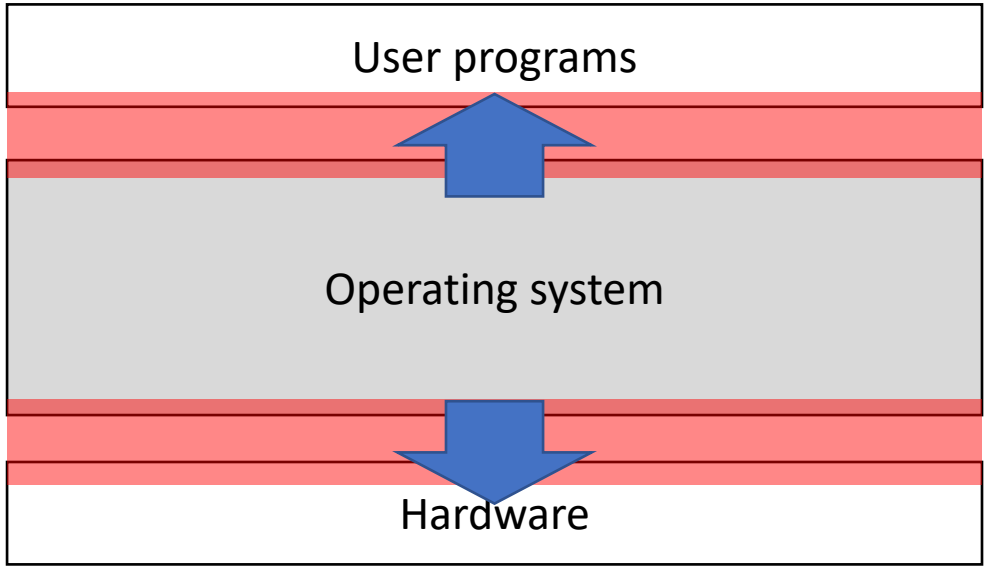
What Operating Systems Do? – user view

- Provides convenience, **ease of use** and **good responsiveness**
  - Don't care about **resource utilization**



What Operating Systems Do? – system view

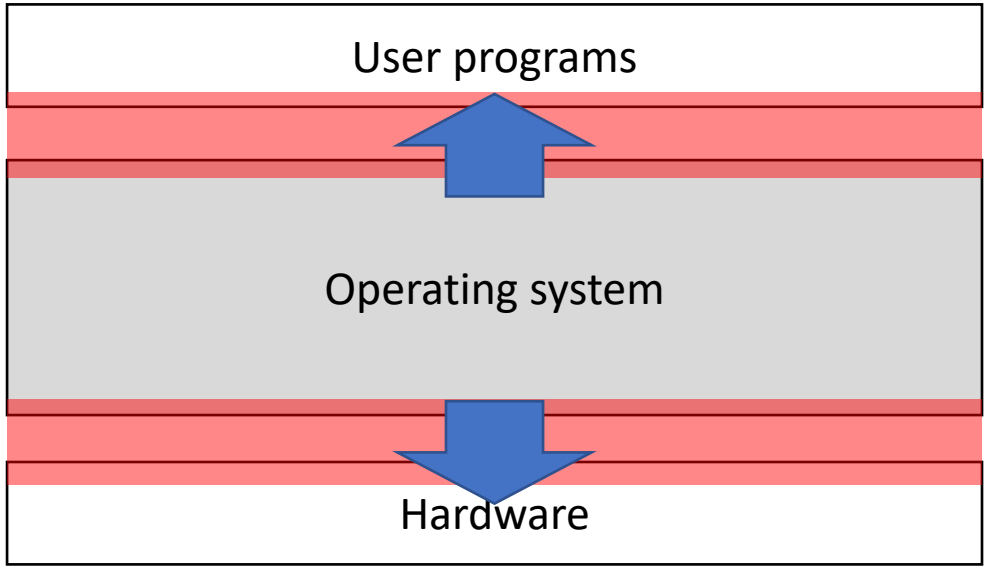
- OS is a **control program**
  - Controls execution of programs to prevent errors and improper use of the computer
- OS is a **resource allocator**
  - Manages all resources
  - Decides between conflicting requests for **efficient** and **fair** resource use



What Operating Systems Do? – system view – cont.

- In fact the most important resource to manage is the CPU;
  - The OS needs to share the CPU amongst multiple tasks and threads.

- One may argue that if you don't need multi-tasking or mult-threading, then perhaps you don't need an operating system!



What Operating Systems Do?

Operating systems may deploy on various computing platforms:

- Users of dedicated systems such as **workstations** (desktops or laptops) have dedicated resources but frequently use shared resources from **servers**
- **Servers** must allocate hardware resources in a manner that keeps all its clients happy. Historically, they were referred to as main-frames or mini-computers.
- **Handheld or mobile computers** are resource poor, optimized for usability and **battery life**
- Some computers have little or no user interface, such as **embedded computers** in devices and automobiles

Operating System Definition

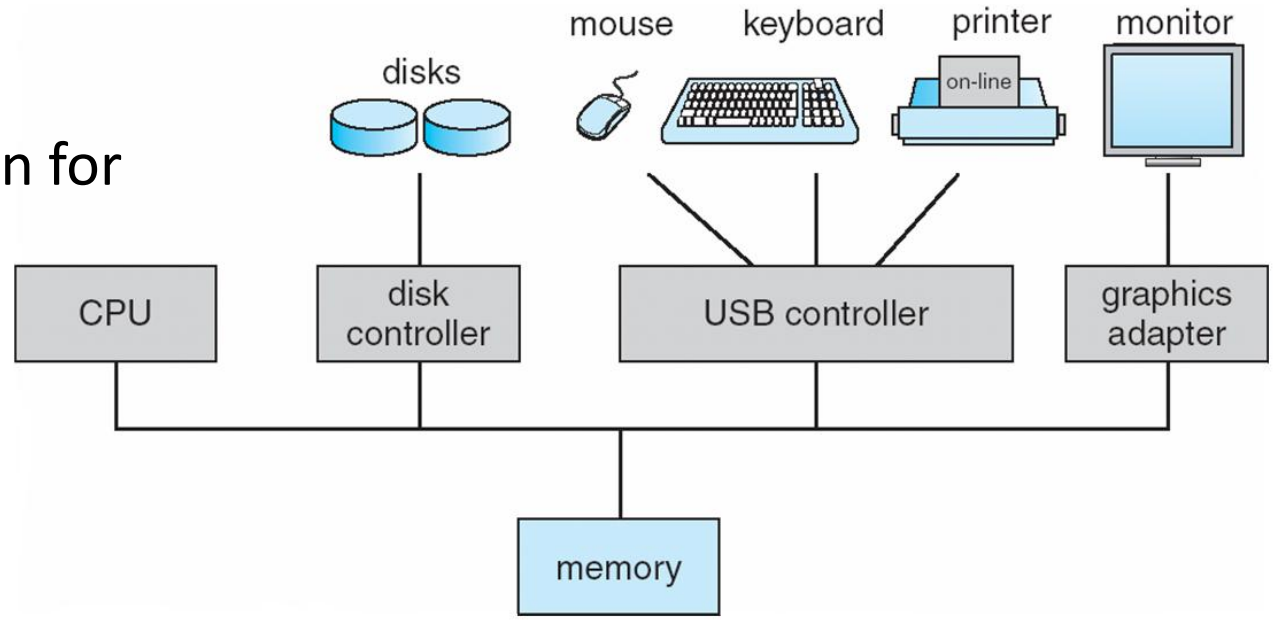
- No universally accepted definition
- “Everything a vendor ships when you order an operating system” is a good approximation
  - But varies wildly
- “The one program **running at all times** on the computer” is the **kernel**. The kernel runs while the CPU is in ***privileged or kernel mode:***
  - It can use all available CPU modes
  - Can access any memory location or hardware resource.
  - Can handle interrupts
- All other programs run in ***user mode*** (i.e. cannot freely access all memory locations, or hardware resources) and is either
  - a system program (ships with the operating system) , or
  - an application program.

Computer System Organization, **review**

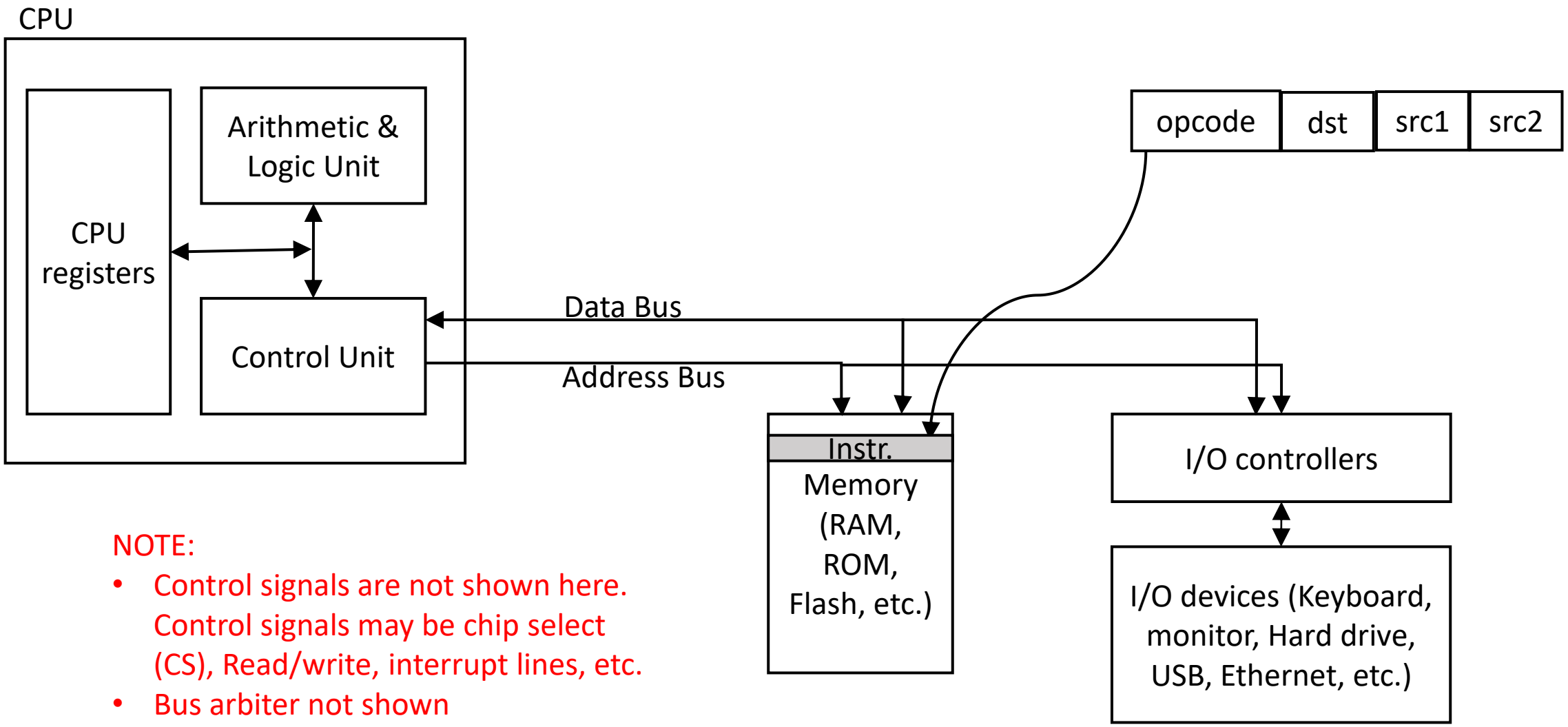
- Computer-system operation
  - A computer system hardware has 3 main components:
    - One or more CPUs.
    - Memory (may be shared amongst CPUs and devices)
    - I/O Device controllers and I/O devices
  - They connect through common bus providing access to shared memory
  - Concurrent execution of CPUs

  and devices results in competition for

  memory cycles.



Computer System Organization, **review** – cont.



NOTE:

- Control signals are not shown here. Control signals may be chip select (CS), Read/write, interrupt lines, etc.
- Bus arbiter not shown

Central Processing Unit (CPU)

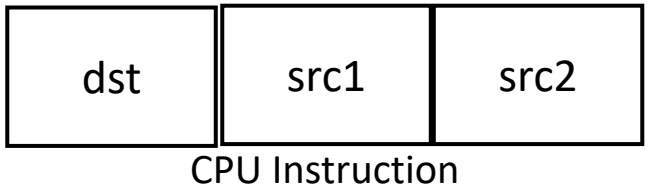
- The CPU's job is to run programs

- It does so by running a fetch-and-execute loop:

```
Repeat:
    Fetch next_instruction
    Execute
```

- The CPU contains a control unit, an Arithmetic and logic unit (ALU) and a register file

CPU instructions



- Also known as **machine instructions**
- Commands to be executed by the CPU
- Have at least 3 fields:
  - **Instruction (e.g. add, subtract, multiply, compare, etc.)**
  - **Source register**
  - **Destination register**
  - **Optionally: some flags (e.g. a condition, such as branch only if nonzero, or multiply only if negative, etc.)**

Central Processing Unit (CPU) – cont.

- **Control unit:** directs the overall operation of the CPU to **fetch and execute** instructions (it is the maestro of the orchestra, that keeps things together!)

NYU Orchestra
(NYU Steinhardt)

Central Processing Unit (CPU) – cont.

- **Control unit:** directs the overall operation of the CPU to **fetch and execute** instructions (it is the maestro of the orchestra, that keeps things together!)
  - Keeps track of where the next instruction resides (using the **program counter**, PC)
    - PC is initialized to zero at reset.
  - Issues the signals needed to read the next instruction from memory
  - Executes ALU instructions by issuing signals to the arithmetic and logic unit.
  - Executes Load and store instructions by issuing signals to the main memory for reading data to a CPU register (Load) or writing data from a CPU register to the system memory (store).
  - Executes Load/store instructions to **memory-mapped device registers** in much of the same way as load/store to main memory.

NYU Orchestra
(NYU Steinhardt)

Central Processing Unit (CPU) – cont.

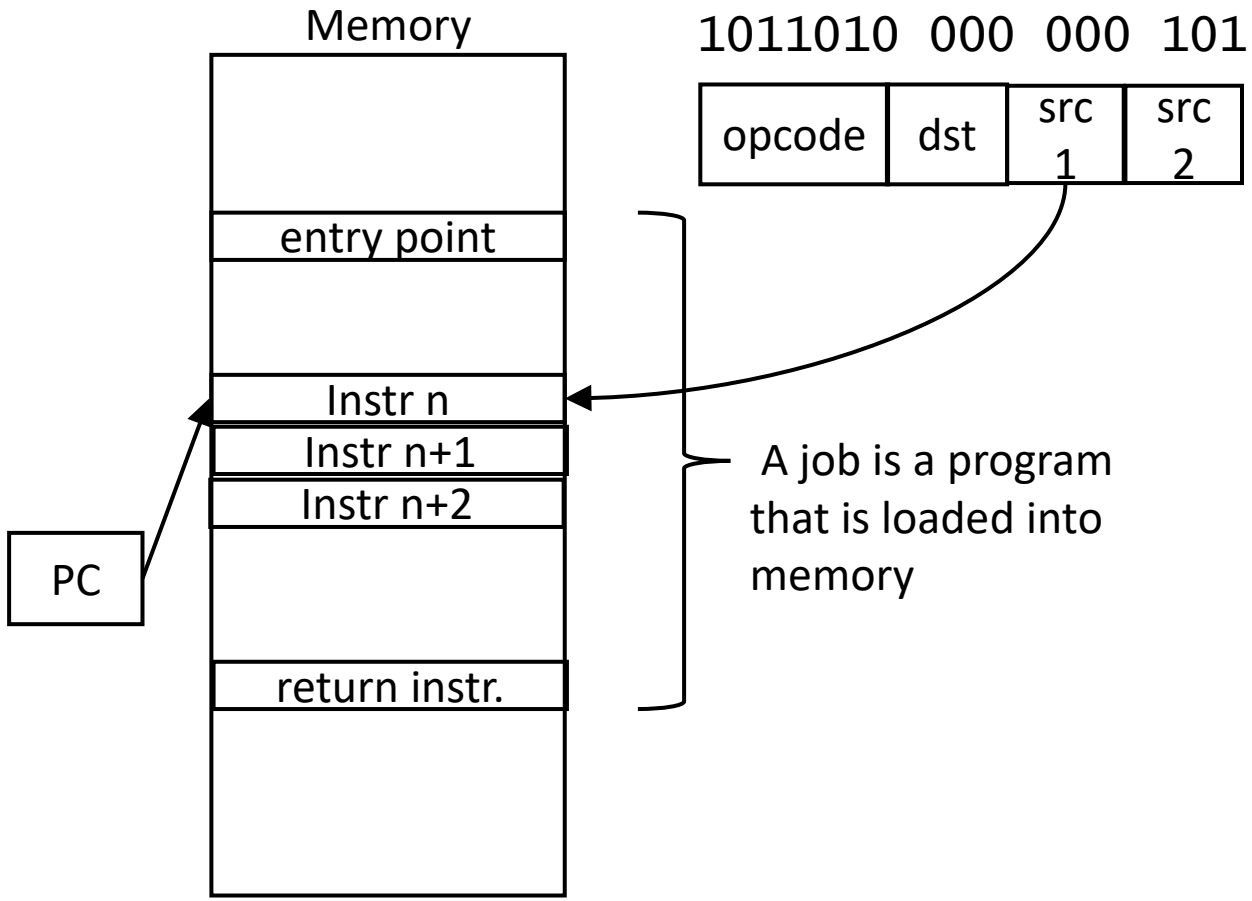
- **Arithmetic and Logic Unit (ALU):** performs all of the elementary computations, such as addition, subtraction, comparisons, and so on, that a computer provides.

- **CPU registers:** may be used as source operands and as destinations for the result. (Also contain the program counter – PC)

- Instructions may use Reduced instruction set architecture (RISC) vs complex instruction set architecture (CISC)

Programs and jobs

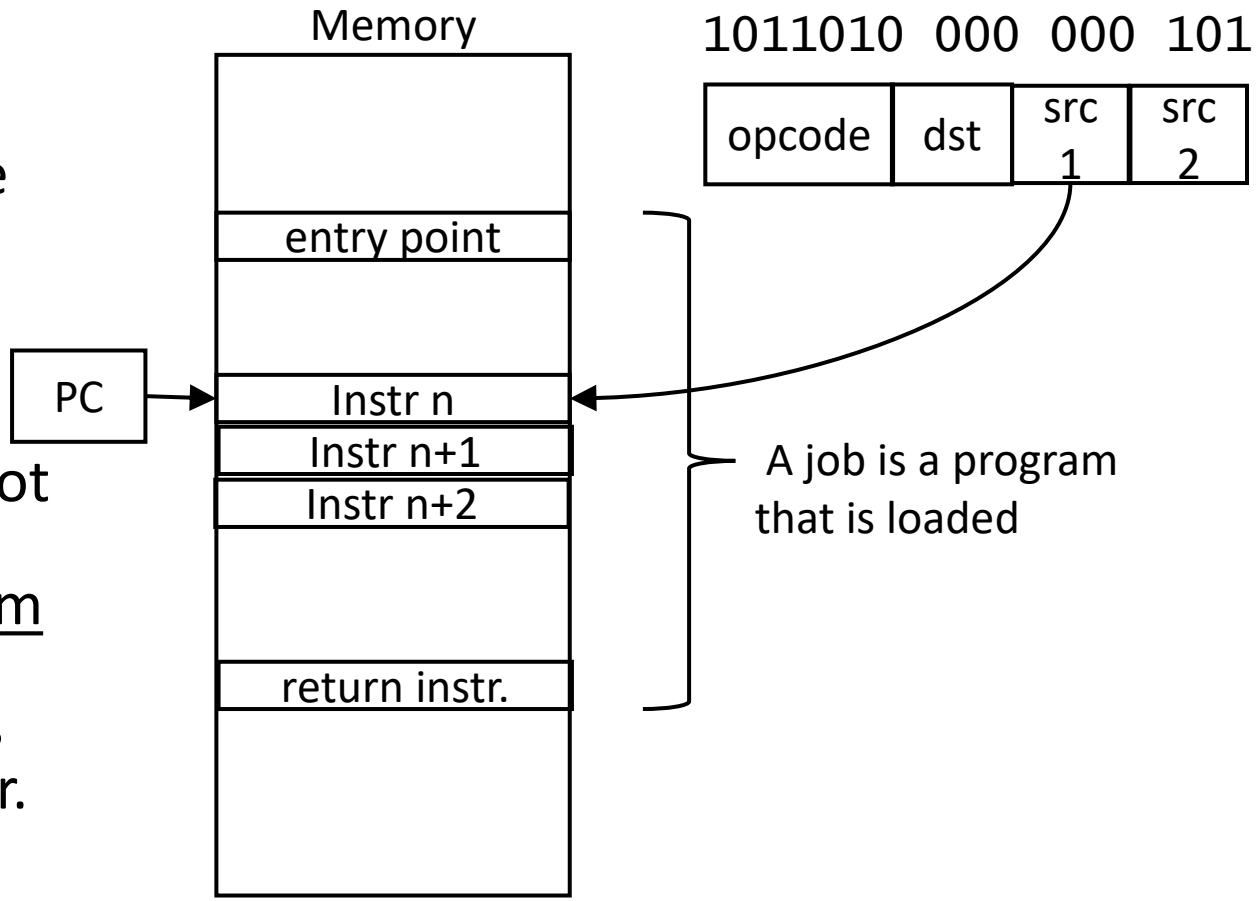
- A **program** is a collection of machine instructions (binary code), usually residing in the hard disk.
  - Typically grouped into a main routine and a number of subroutines.
- A **job** or **process** is a program that is loaded into the system memory (code+data)
- The first instruction to execute is called the **entry point** (not necessarily located at the beginning of your program memory)



Programs and jobs – cont.

- The last instruction to execute is the **exit** or **return** instruction (not necessarily located at the end), where the PC goes back to the initial caller (e.g. the shell program).

- **NOTE:** The control unit can only fetch instructions from memory, i.e. it cannot directly access data in a hard disk,
  - A subroutine or a separate program is thus needed to access the hard disk's information. This program is often referred to as a device driver.



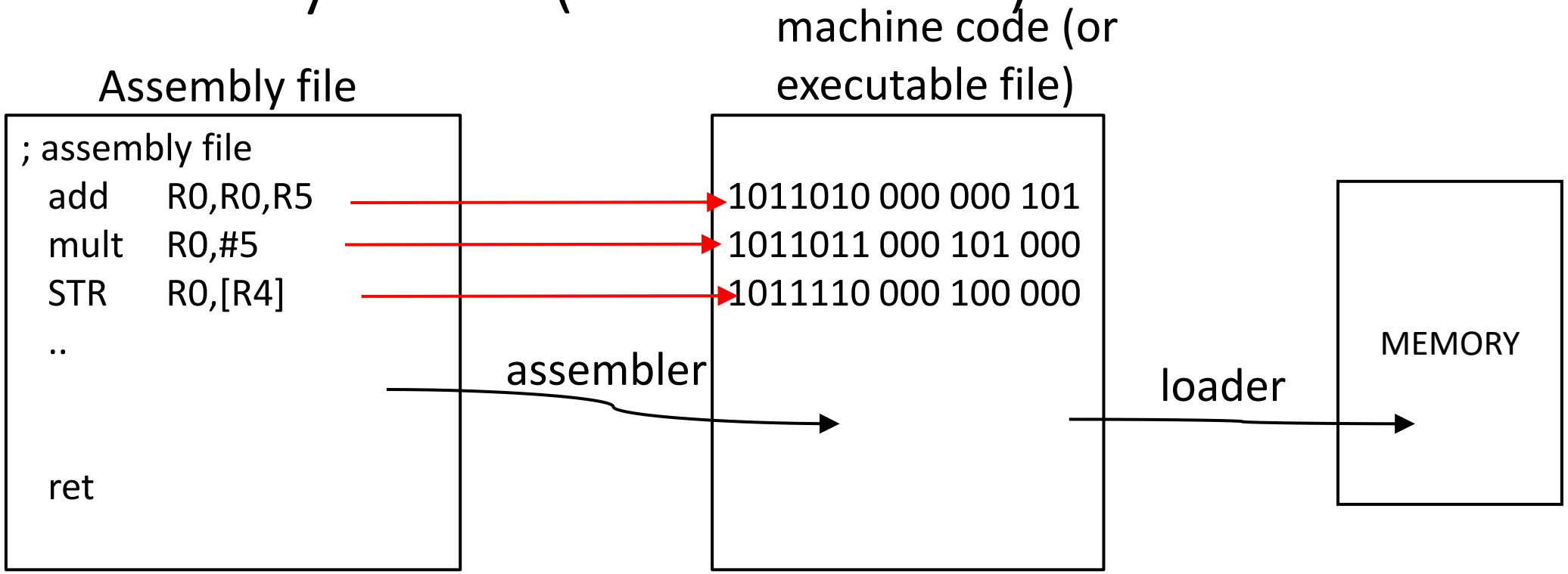
Machine Code

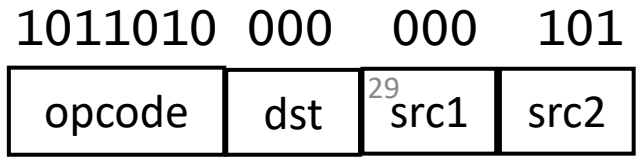
- A program may be stored in a hard-drive when it is not running but must be loaded (by a “loader”) into the main memory in order to be executed by the CPU (and thus becomes a job or process).

- In the early days of computing (1960’s), programs (or jobs) used to be written using machine code (i.e. binary ones and zeros such as 1011010000000101) into punch cards. The punch cards were then loaded into computers to be executed.

- **NOTE:** the hardware (i.e. the CPU, memory, etc.) ONLY understands 1’s and 0’s.

Assembly code (source code)





Assembly code

- Instead of using machine codes (ones/zeros), **English-like** names (e.g. ADD), often **abbreviated** (e.g. SUB for subtract), are used for instructions (opcode field), whereas **symbolic names** are used for source/destination registers (e.g. R1, R9 or AX).
  - Each CPU platform has its own assembly language.
  - Assembly instructions map one-for-one with machine instructions
  - Operands may be **immediate values** (i.e. not a src/dst register but rather a value) and may be coded in binary, decimal, hexadecimal, etc.

Assembly code – cont.

Instructions may be grouped into three general categories:

- Move instructions – Allow your program to move data from main memory into a CPU register (load) and vise versa (store), as well as between CPU registers.

- Arithmetic and Logic instructions – allow the program to perform arithmetic operations (e.g. add, subtract, multiply, etc.), logic operations (AND, OR, etc.), and relational operations (compare, less than, etc.) on data stored in CPU registers

- Program control instructions – Typically your program executes instructions sequentially, but may **branch** and execute an instruction that is not the next instruction in memory, and may be far away from the current instruction.
    - May be conditional or unconditional
    - May save the return address (function calls) or not (branch, aka jump or goto)

Assembly code – cont.

- **Assembly Pseudo-instructions** do not translate to machine code however, they are instructions to the assembler (i.e. the compiler)
  - **Labels** may be used to specify certain program locations (addresses) and can be then used within instructions instead of an actual address - useful in accessing memory locations containing variables **and** in branch instructions.
  - **Macros** allows grouping of a group of machine instructions so they can be re-used many multiple times.
    - So what is the difference between a macro and a subroutine then?
  - Allows grouping of code or data into **sections**
  - Allows allocating bytes (within the data sections) with or without initial values

```
                .section .data          # data section typically contains initialized global vars
mychar:
                .BYTE  10               # arr holds the address of that byte – NOTE THE TAB
myint:          .int 24                 # assembly is NOT case-sensitive
                .section .bss           # bss section typically contains uninitialized global vars
myarr:          .space 80
                .equ devid, 1           # devid is a symbol that now has a value of 1
L1:
                .section .text
                mov r1, r2              # L1 holds the address of the mov instruction
```

C/C++ source code

- Machine code and assembly are often called first and second generation languages.
- C and C++ use **English-like statements** and are considered to be third generation languages. They work at much higher level than assembly.
  - A single C/C++ statement typically maps to multiple machine instructions.
- C/C++ source code may access any specific memory location using **pointers**.
- Code written in C or C++ **cannot** access **CPU registers**, however most vendors of C and C++ compilers offer facilities to access assembly instructions as if they are function calls, e.g.

```
asm(" add r2,1");
```

- We will use C for most of our assignments.

Assembly/C code - example

Incrementing a variable

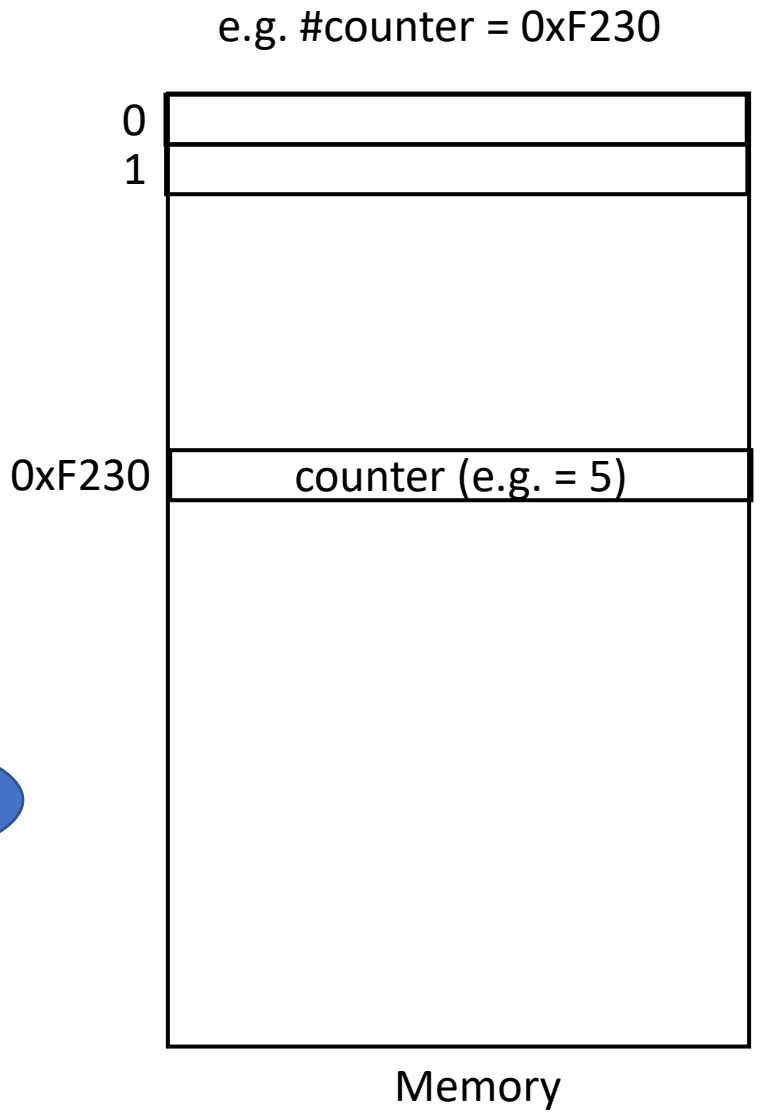
C/C++

```
counter++;
```

.equ is used to define a symbol, “counter”

assembly

```
.equ counter F230h
mov r2, #counter
ld  r1,[r2]     ; r1 = counter
inc r1          ; r1 = r1 + 1
st  r1,[r2]     ; counter = r1
```



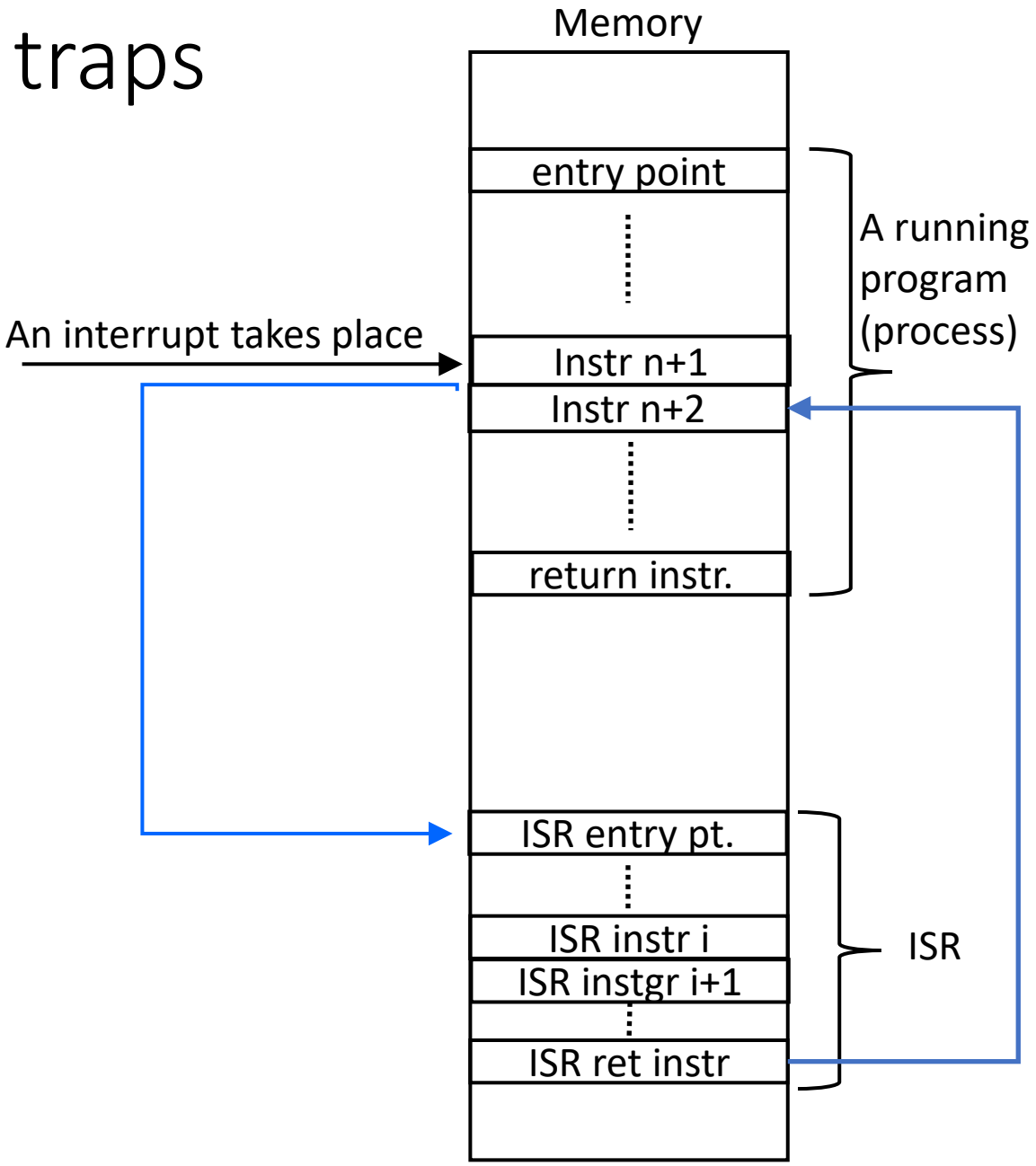
Interrupts, exceptions and traps

- The terms interrupt and exception are often used interchangeably. Interrupts may occur due to 3 categories of causes:
  - **Hardware Interrupts:** Caused by an I/O device controller. These are usually the result of an event external to the CPU.
  - **Software Interrupts**:
    - The software (usually unintentionally) attempting to perform an **illegal operation**, e.g.:
      - Divide by zero
      - Accessing a memory region it's not supposed to access
      - An invalid opcode.
    - A **system call**, **a software trap** or **trap instruction** is a software interrupt/exception **intentionally** issued by a user program to request system services.

Interrupts, exceptions and traps

When an exception occurs (due to any of the 3 causes)

- The CPU does not execute the next instruction within the current routine (instr n+2),
- Instead, it starts executing an interrupt service routine (ISR).
- Upon completing the ISR, the CPU may resume the interrupted routine when it encounters a “return from interrupt” machine instruction.



The Interrupt Vector Table

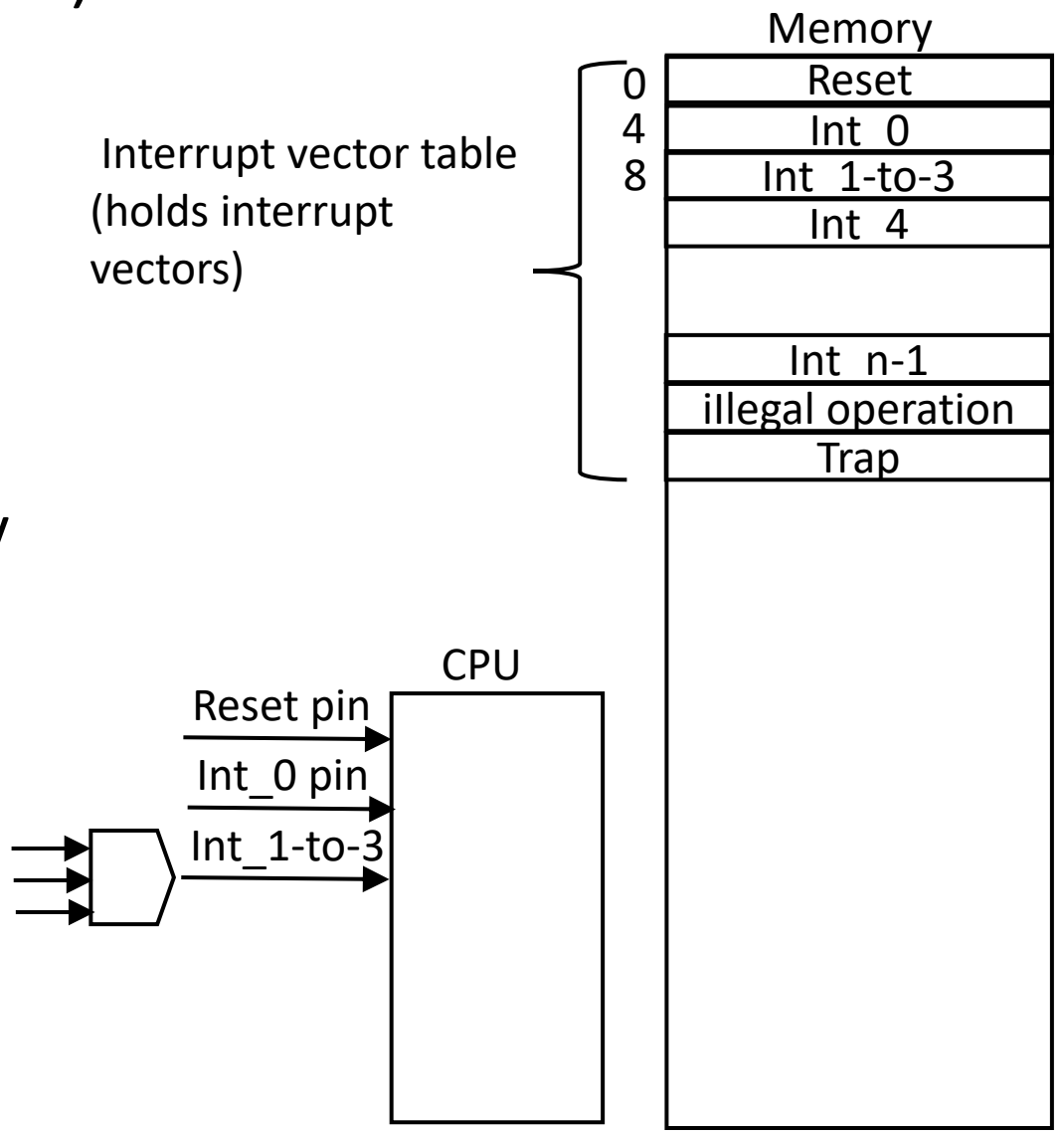
- There are many possible sources of interrupts/exceptions → how does the CPU know where is the interrupt service routine for each exception?

Via the **Interrupt Vector Table**!

The Interrupt Vector Table (IVT)

- A table that stores either
  - Branch instructions to the **interrupt service routines (ISR)** – i.e. routines that can handle interrupts
  - Addresses of entry points for ISRs.

  (choice is processor dependent)
- The interrupt vector table may be located at address 0, or at the end of the memory address space (depending on the CPU configuration).
- It may be remapped after booting and starting the OS kernel (**why?**)
- Generally, each interrupt line maps to a specific interrupt vector.
  - In a 32-bit system, how many bytes in each vector, assuming the IVT contains ISR addresses (not branch instructions)?

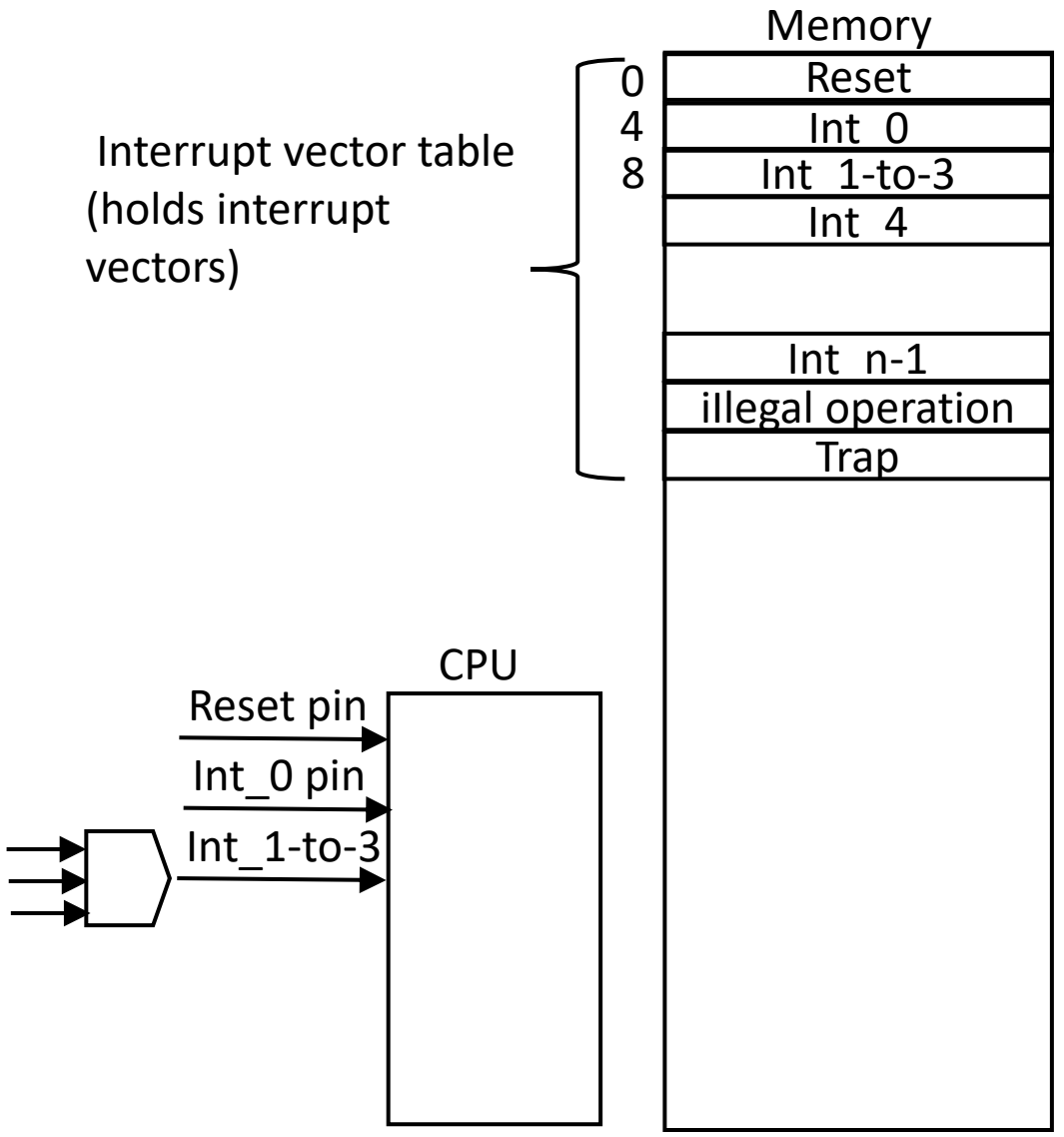


The Interrupt Vector Table – cont.

Interrupt vector table remapping:

- **Typically implemented by taking advantage of virtual memory**, which we will discuss how to manage **later in the course**.
- Alternatively, some CPU's (e.g. ARM) may have a **Vector Table Base register**.

NOTE: that some primitive CPU's may have none of the prior two options and the system may need to mimic the functionality in software.

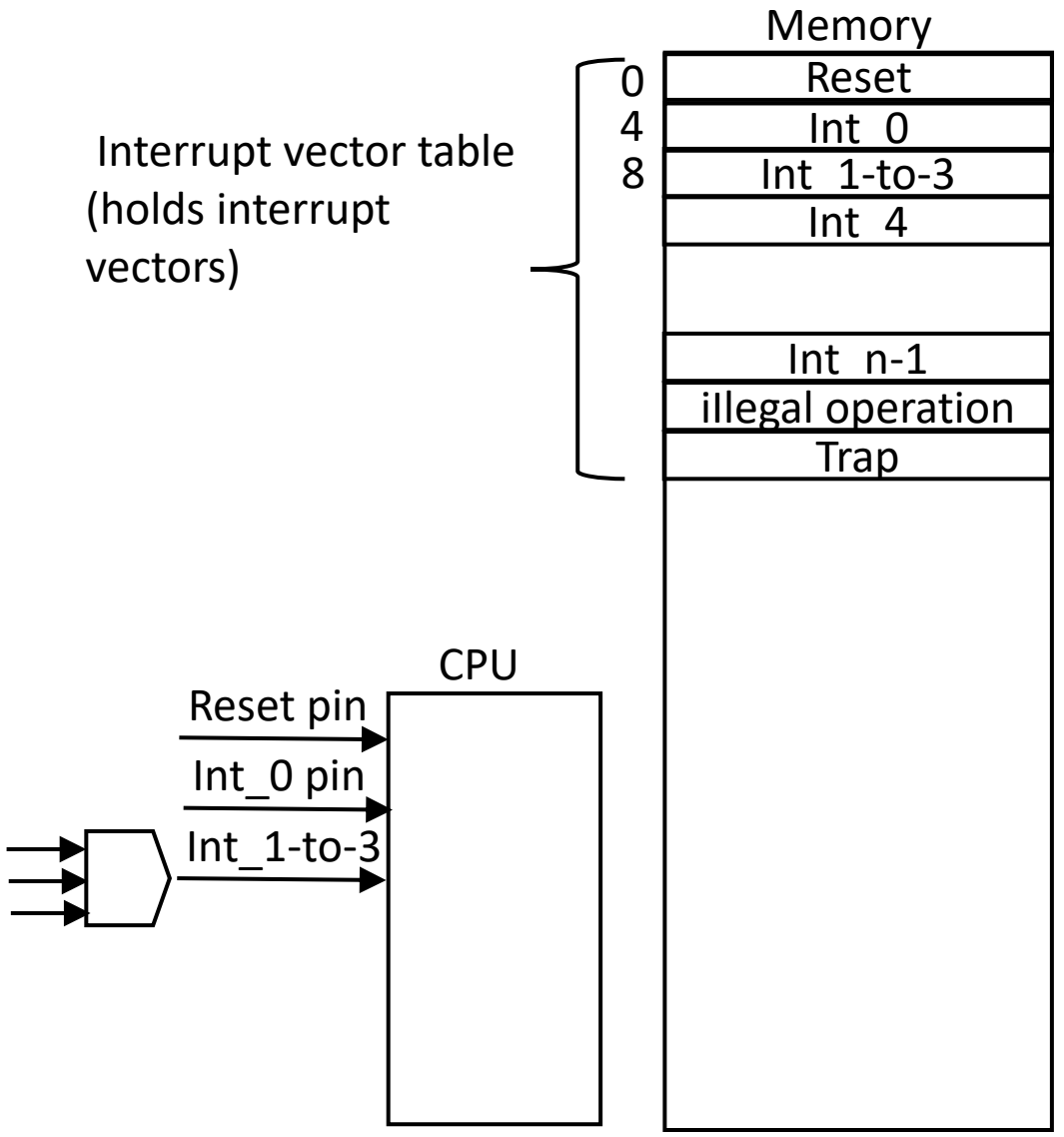


The Interrupt Vector Table – cont.

- There are usually vectors for the system call and for illegal operations.
- What is stored in the reset vector?

NOTE:

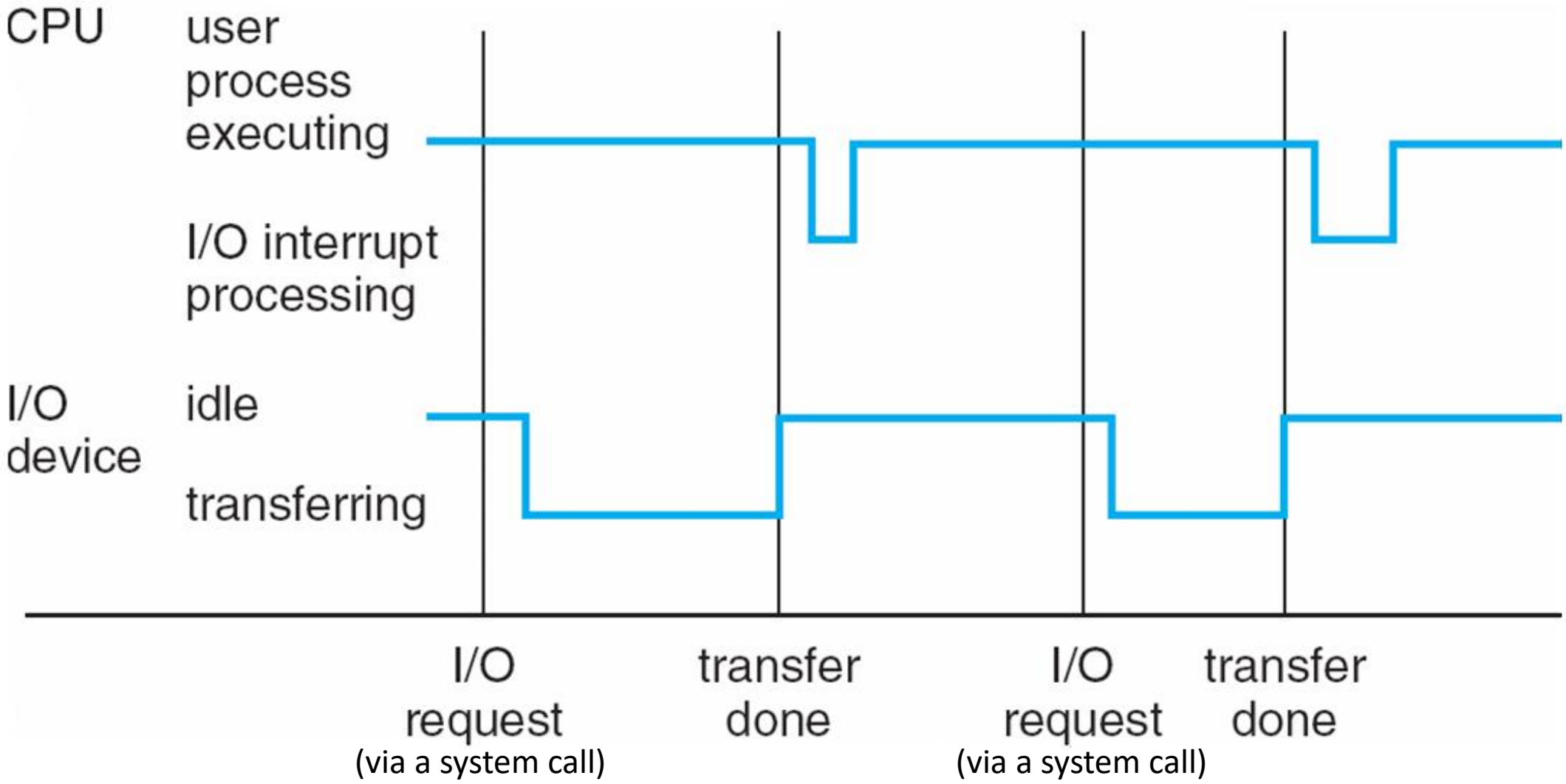
- Multiple interrupt lines may **share** a single interrupt vector (as in interrupt lines 1 to 3 in the example to the right)
  - In such a case the ISR would need to detect which particular interrupt pin is responsible and then call the corresponding interrupt handler.



Interrupt Handling

- Interrupt service routines resides within the operating system's kernel (and run in kernel mode). They need to preserve the state of CPU registers by **saving** them into the main memory.
  - This includes saving the program counter (PC) register.
  - Note that in many CPU platforms some interrupt types may have hardware support, in which the CPU registers are saved automatically. This is often achieved by having multiple banks of CPU registers.
- Prior to the ISR's exit, it then needs to **restore** the CPU registers. The PC needs to be the last register to be restored, why?

Interrupt Timeline



System Boot

- When powering up the system, execution starts at a fixed memory location that marks the start of the boot routine.
  - The “reset vector” has a branch instruction to that memory location
  - A non-volatile memory (e.g. ROM, EEPROM or a flash memory) **MUST** be used to hold initial boot code/routine (aka **bootstrap loader**).
    - **Why?**
  - A program stored in ROM, EEPROM or flash may be referred to as firmware
- The boot routine initializes the necessary aspects of the system, e.g.:
  - Hardware resources required to read from the disk
  - serial or USB keyboard interface, etc.
- Operating system must be made available so that the CPU can start it
  - The **bootstrap loader** locates the kernel from non-volatile storage (e.g. hard disk), loads it into memory, and starts it.
  - Sometimes two-step process where **boot block** at fixed location in ROM is loaded and executed (PBL), which then loads a secondary bootstrap loader (SBL) from disk
- Common second stage boot loaders (SBL):
  - **GRUB**, allows selection of kernel from multiple disks, versions, kernel options. GRUB is popular for x86 CPUs
  - **UBoot** is the popular SBL used with ARM CPUs (common with embedded Linux and Android).
- Kernel loads and system is then starts **running**
  - May **remap the interrupt vector table** to a location within the kernel

Now, we can answer the question from 5 slides ago

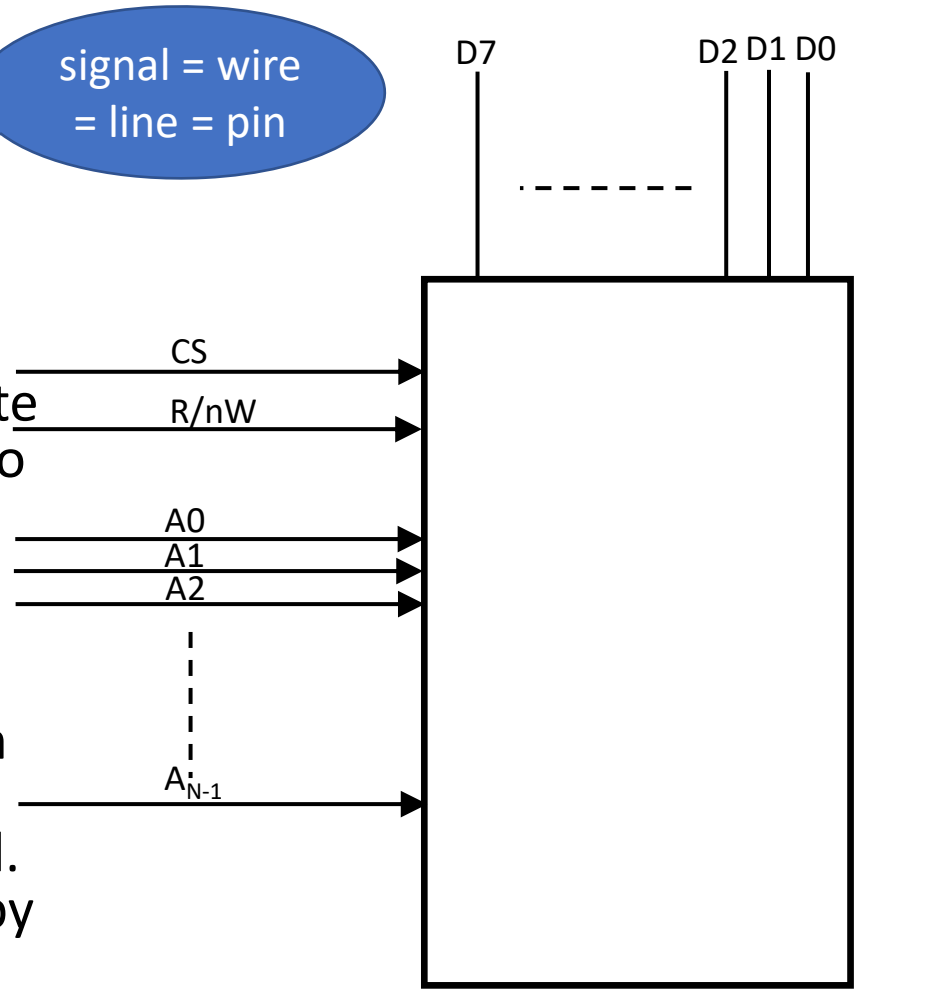
Memories

- The basic unit of computer storage is the **bit**.
  - A bit can contain one of two values, 0 and 1.
  - All other storage in a computer is based on collections of bits.
- A **byte** is 8 bits
  - On most (if not all) computers it is the smallest accessible (i.e. via reads/writes) unit storage. For example, most computers don't have an instruction to move a bit but do have one to move a byte.
- A **word** is the computer's native unit of data. Thus it is architecture-dependent
  - In a 64-bit system, a memory word is 8 bytes. CPU registers are 8 bytes too.
  - In a 32-bit system, a memory word is 4 bytes. CPU registers are 4 bytes.

  - A computer is most efficient when it accesses external memory one word at a time.
    - It can still access smaller unites from the main memory, e.g. a single byte.

Memories (main memory)

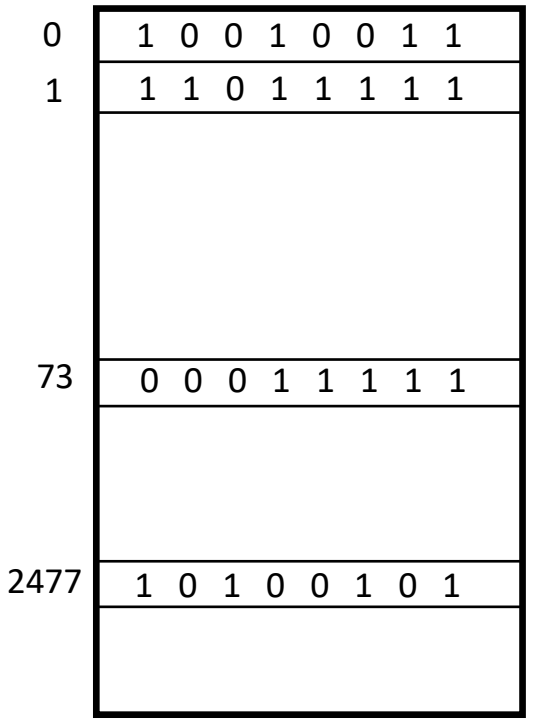
- A memory is digital chip that stores data generally accepts:
  - CS (**chip select**) signal - selects the entire chip. If this signal is not asserted, then all other signals do not matter.
  - **Address** signals (pins) – select a memory byte within the chip, where data will be written to or read from.
    - The address signals combine to form an **address bus**.
  - R/nW (**read, not write**) signal – tell the chip whether data will be written to or read from the chip.
- The **Data bus** (D7 – D0) is an input/output signal.
  - If R/nW is high, then the data bus is driven by the memory chip → Memory produces data
  - If R/nW is low, then the data bus is driven by the CPU → Memory accepts data.



A byte-addressable memory

Memories – cont.

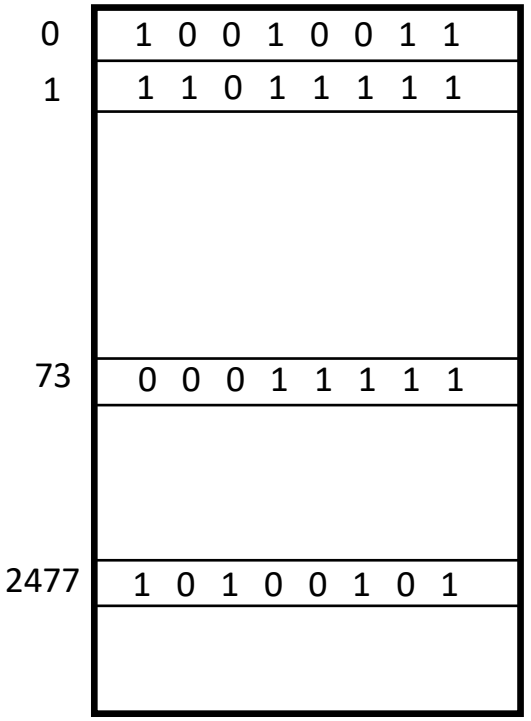
- For N address bits 2^N bytes may be addressed.
  - Ex: 16-bit address bus may address 64K bytes
  - 20-bit address bus may address ? Bytes
- Memories may also be **word-addressable**, e.g. In a 64-bit system, a memory may be addressed as 8 bytes (64 bits).
  - In such case, the lower address bits may not be used. How many?
- Alternatively, it may be byte-addressable (8-bits).



A byte-addressable memory

Memories – cont.

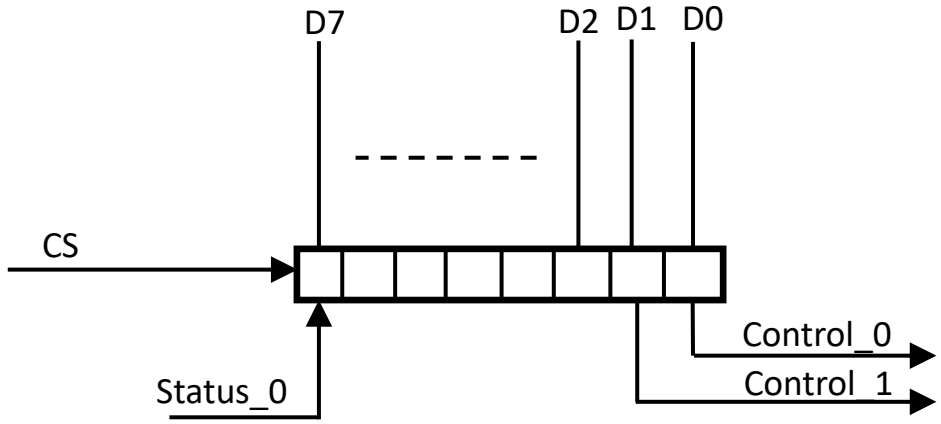
- Volatile vs non-volatile
  - Read-only memory (**ROM**) is a form of non-volatile memory.
  - Electrically erasable programmable read-only (**EEPROM**) and Flash memories are non-volatile and may be re-programmed.
    - Write times take significantly longer than read times.
  - Random Access Memory (**RAM**) is volatile.
  - ROM, EEPROM and flash memories may be randomly accessed, RAM is just the name commonly used to refer to volatile memory.



A byte-addressable memory

Registers

- Similar to memories in that they store information, but with an additional feature:
  - The information **feeds to some external logic function**.

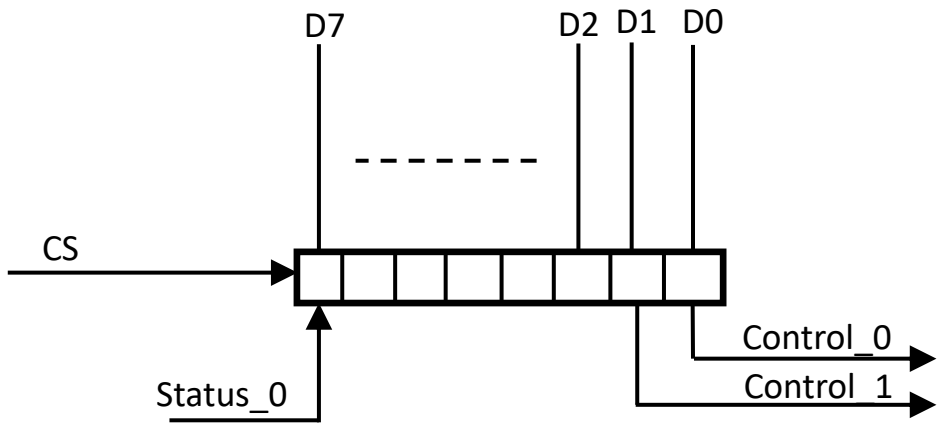


- As a result, a register bit may be used as:
  - An output/control bit that controls some digital function, or
  - An input/status bit that reflects the status of another digital function

- Status bit example: PACKET_XFER_COMPLETE, SENSOR_TRIGGERED, etc.
- Control bit example: PACKET_SEND, MOTOR_EN, MOTOR_ROTATE_RIGHT, etc.

Registers – cont.

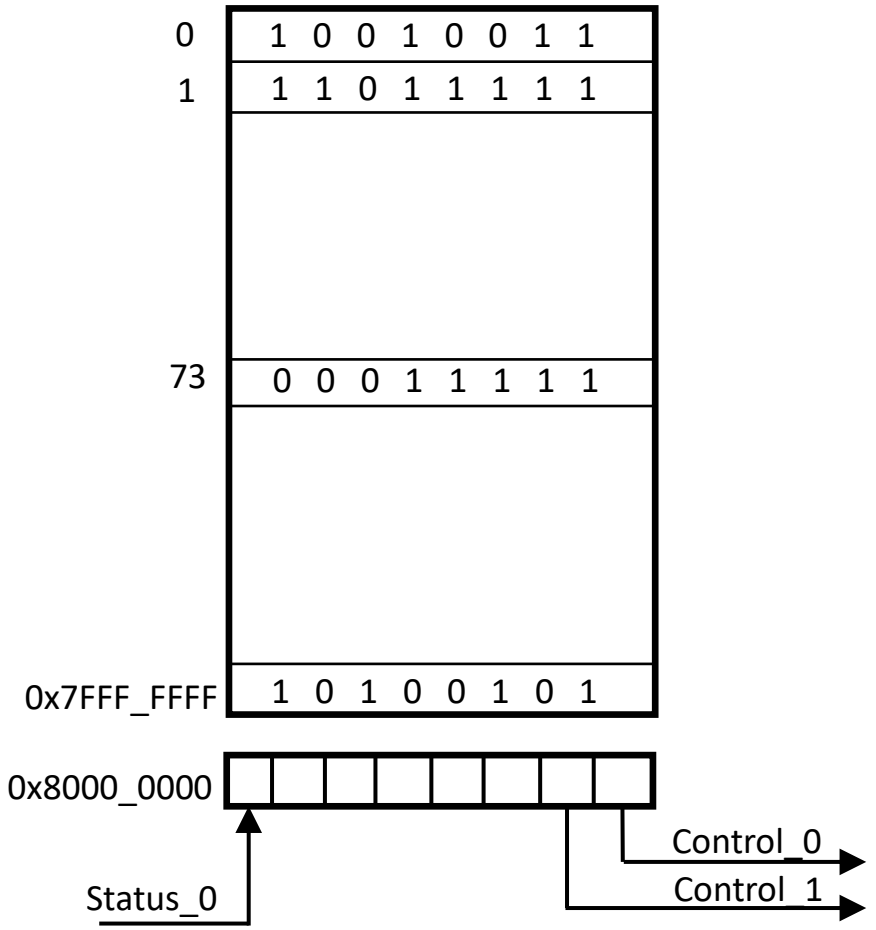
- **A CPU register** is inside the CPU and accessible as an operand in the machine instruction
- **A device register** is most commonly memory-mapped to some address within the CPU's address space (aka memory space);
  - Thus, **each register has a single unique memory address** through which you can access (read/write) that register.



registers are not located inside of any memory though

Registers – Example

- A 32-bit system with a 2GB memory, and a single control register.
- How do we use “polling” to wait for a status bit to be asserted (logic 1 in this case).
  - We need to create a loop that checks the value of “Status_0” bit, which is bit 7 in the diagram.
  - Assume the register is located at memory-mapped address 0x80000000




```
unsigned char status;
volatile unsigned char *addr = 0x80000000;
do {
    status = *addr;
} while ((status & 0x80) == 0); // Stay in the loop till condition is false
```

I/O Operation

- I/O devices and the CPU can operate or execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer (usually memory mapped)
- CPU, and thus software, communicate with devices by:
  - Writing to control registers (to instruct the controller to do an operation)
  - Reading from status registers (to determine if the operation has completed)
  - Moving data from main memory to local buffers and vise versa.
- The CPU waits till the operation is done via one of two methods:
  - **Polling:** The CPU continuously reads the status till it detects operation is complete.
  - **Interrupt-driven-I/O:** CPU moves on and executes other tasks. Device controller informs CPU of completion by asserting an interrupt pin.
- A **Device Driver** (part of the kernel software) exists for each device controller to manage I/O
  - Provides uniform interface between controller and rest of the kernel

Storage Definitions and Notation Review

Computer storage, along with most computer throughput, is generally measured and manipulated in bytes and collections of bytes.
A **kilobyte**, or **KB**, is 1,024 bytes
a **megabyte**, or **MB**, is $1,024^2$ bytes
a **gigabyte**, or **GB**, is $1,024^3$ bytes
a **terabyte**, or **TB**, is $1,024^4$ bytes
a **petabyte**, or **PB**, is $1,024^5$ bytes
an **exabyte**, or **EB**, is $1,024^6$ bytes

Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes. Networking measurements are an exception to this general rule; they are given in bits (because networks move data a bit at a time).

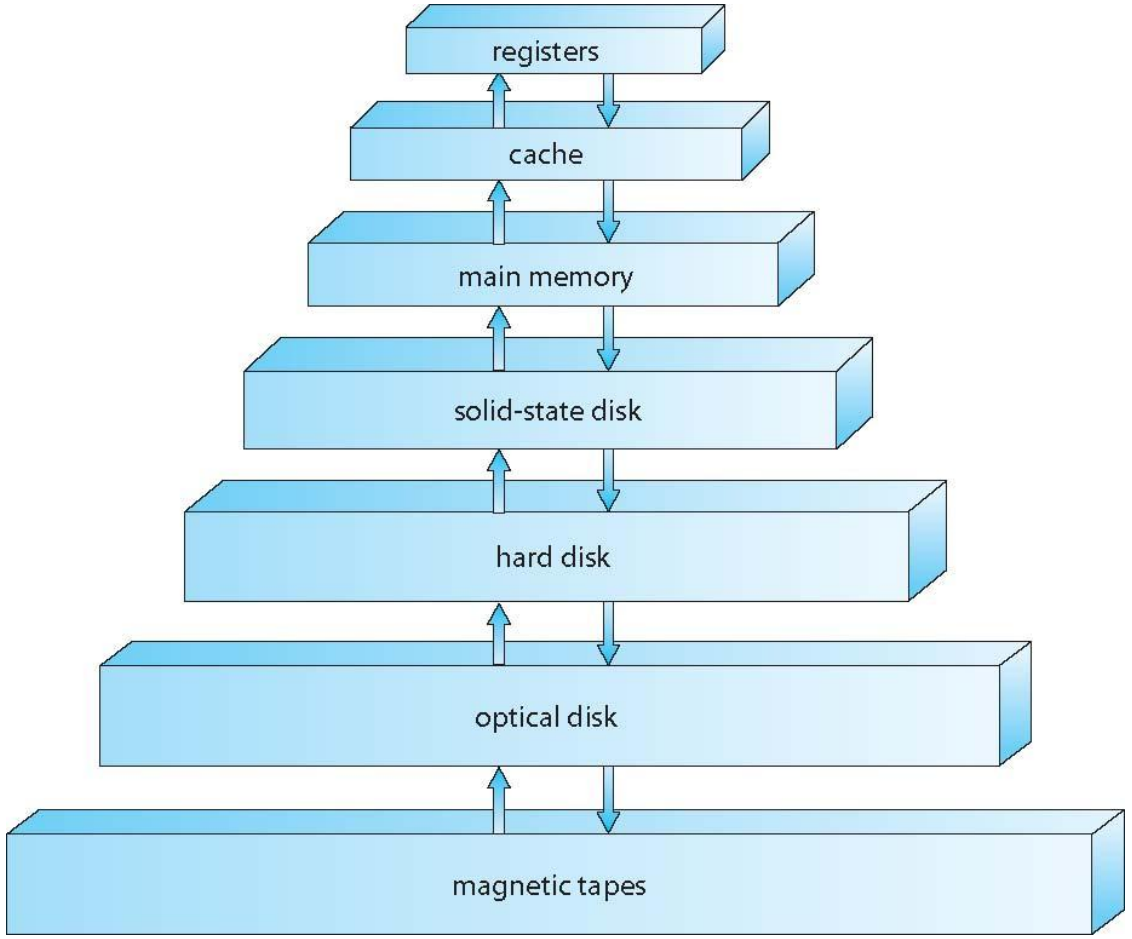
Storage Structure

- Main memory – is the only storage media that the CPU can access **directly**:
  - RAM: random access memory, volatile
  - ROM: read only memory, non-volatile (e.g. ROM, EEPROM or flash)
- Secondary storage – extension of main memory that provides large nonvolatile storage capacity, however, the CPU can only access this memory **indirectly** via a device controller (using its control/status and data interfaces)
  - **Hard disks** – rigid metal or glass platters covered with magnetic recording material.
    - Disk surface is logically divided into tracks, which are subdivided into sectors.
  - **Solid-state disks** – faster than hard disks, also nonvolatile
    - Becoming more popular

Storage Hierarchy

- Storage systems organized in hierarchy
  - Speed
  - Cost

  (usually, the larger the memory, the slower it is)
- **Caching** – As a concept, it means copying information into faster storage system;
  - Main memory can be viewed as a cache for secondary storage
  - CPU's internal cache memory is a cache for the main memory



Caching

- Important principle, performed at **many levels** in a computer (in hardware, operating system, software)
- Information in use is copied from slower to faster storage temporarily
  - Faster storage (cache) checked first to determine if information is there
    - If it is, information used directly from the cache (**cache hit**).
    - If not, data copied to cache and used there (**cache miss**).
- Why is it advantageous to use cache?
- Cache management is an important design problem
  - Cache size (affects speed + cost)
  - Replacement policy (e.g. LIFO, LRU, etc.)

Direct Memory Access Structure

- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
  - Utilized in network interface cards (NIC), Hard drives, etc.

- Only one interrupt is generated per block of data, rather than the one interrupt per word or byte.

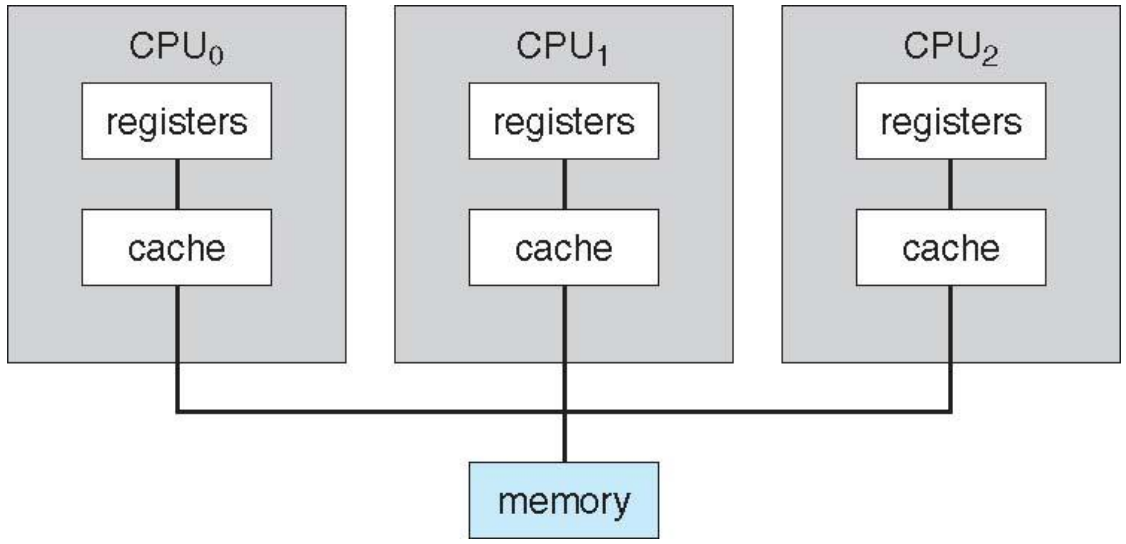
Computer-System Architecture

- Many systems use a **single general-purpose processor**
  - Also often referred to as **application processor**.
  - Most systems have special-purpose processors as well (e.g. a GPU or a DSP), but these do not make the system a multiprocessor system.

- **Multiprocessors** systems growing in use and importance
  - Also known as **parallel systems**, **tightly-coupled systems**
  - Advantages include:
    1. **Increased throughput**
    2. **Economy of scale**
    3. **Increased reliability** – graceful degradation or fault tolerance

Computer-System Architecture – cont.

- **Multiprocessors** - Two types:
  1. **Asymmetric Multiprocessing** – processors are not treated as equal.
     - Processors may be dedicated to specific tasks
     - e.g. boss and worker processors
  2. **Symmetric Multiprocessing (SMP)** – all processors are treated equally
     - Single instance of the OS.
     - Each processor is **capable** of performing any task, such as handling interrupts, running the OS kernel, running applications, etc.



Symmetric multiprocessors

Multi-Core Designs

- A **multicore** system may have multiple cores in a single chip, and is thus a multiprocessor system.
- Systems may be built of multiple chips, each with multiple cores.