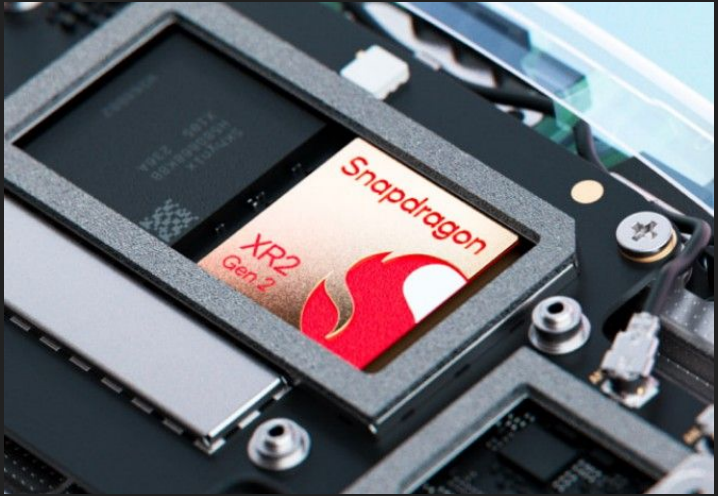
03 - Hardware and The OS

CEG 4350/5350 Operating Systems Internals and Design
Max Gilson

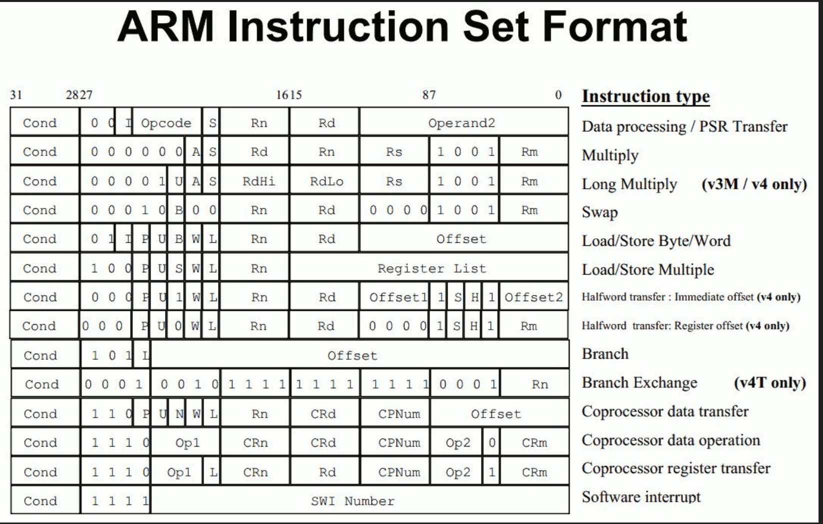
The CPU

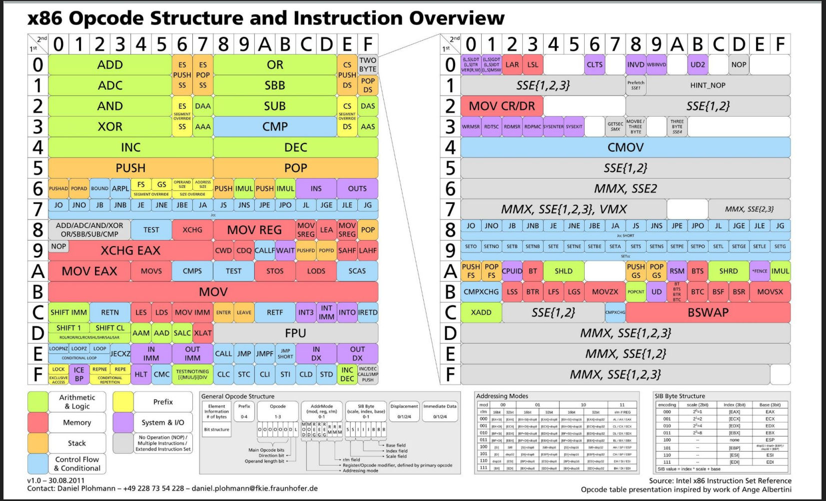
- The Central Processing Unit (CPU) executes the instructions we provide to it
- Without the CPU, the computer cannot run programs or do anything useful
- The CPU has a specific instruction set architecture (ISA) that defines what instructions it runs and how it runs them



The ISA

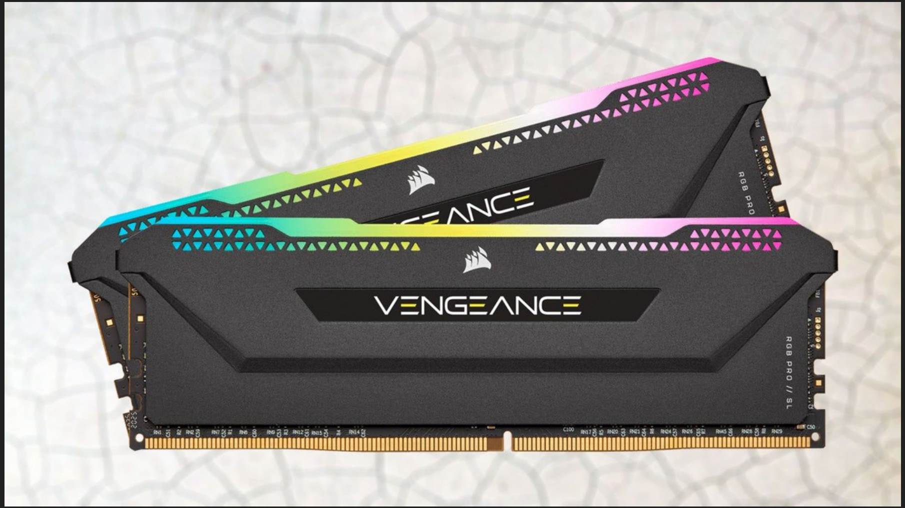
- The ISA is a “blueprint” for the tools the CPU has at its disposal
- The ISA defines what registers, instructions, and systems are available for the CPU to use
- x86, ARM, RISC-V, MIPS are examples
    - If you write code for one ISA it cannot run on another ISA without recompiling or rewriting code
    - Assembly code written for x86 will never run on ARM without a complete rewrite from scratch
    - A C program can be compiled for multiple ISAs





The RAM

- The Random Access Memory (RAM) stores our programs and data
- Without RAM, the CPU wouldn't have instructions to execute
- Any program you want to run has to loaded into RAM



The Storage Device

- The storage device holds our files and programs
  - Hard Disk Drive (HDD)
  - Solid State Drive (SSD)
  - Floppy Disk
- Our storage device is an I/O device
- When you want to execute a program or read a file, it is copied from storage and put into RAM
- If you hear the word “disk” we are talking about these

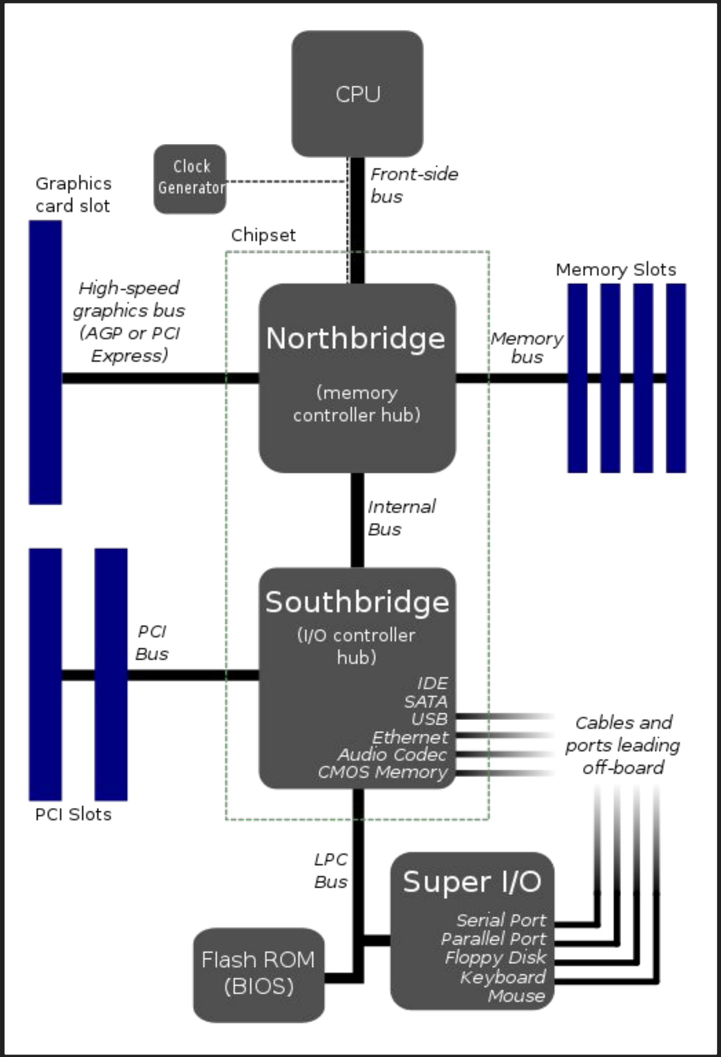
SSD

HDD

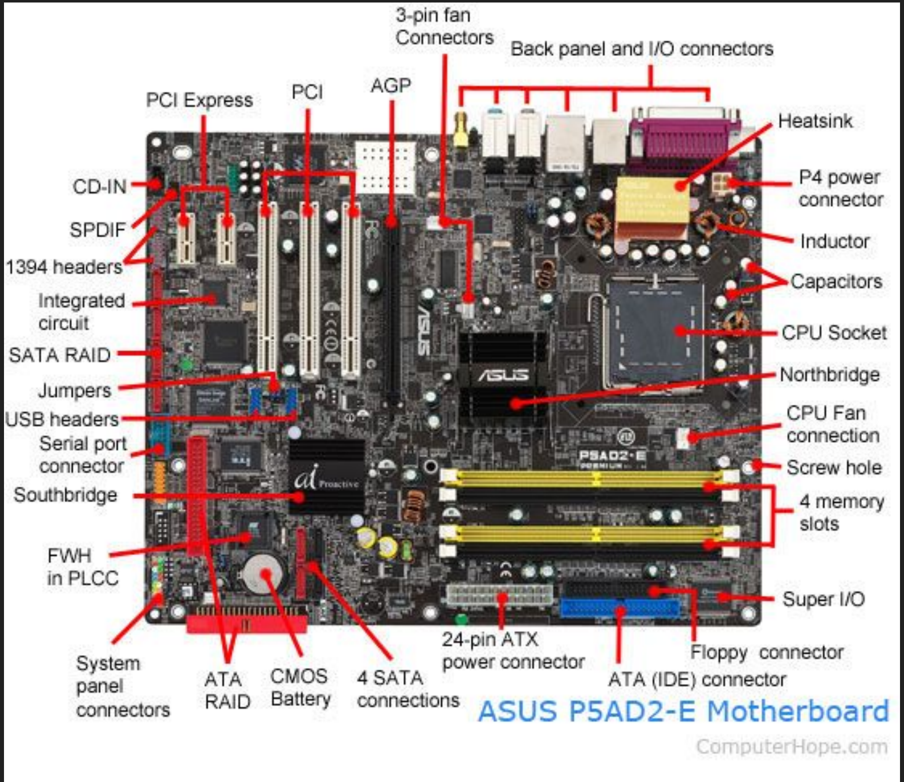
Floppy Disk

The Motherboard

- The motherboard is a physical component of a computer that connects CPU, RAM, and I/O devices together
- The CPU, RAM, and I/O all plug into the motherboard as individual modules
- There are other components on the motherboard that assist in the transfer of data between these components
    - Northbridge interconnects CPU, RAM, and Southbridge
        - Fast stuff
    - Southbridge interconnects I/O
        - Slower stuff
    - Busses are the paths for data to travel

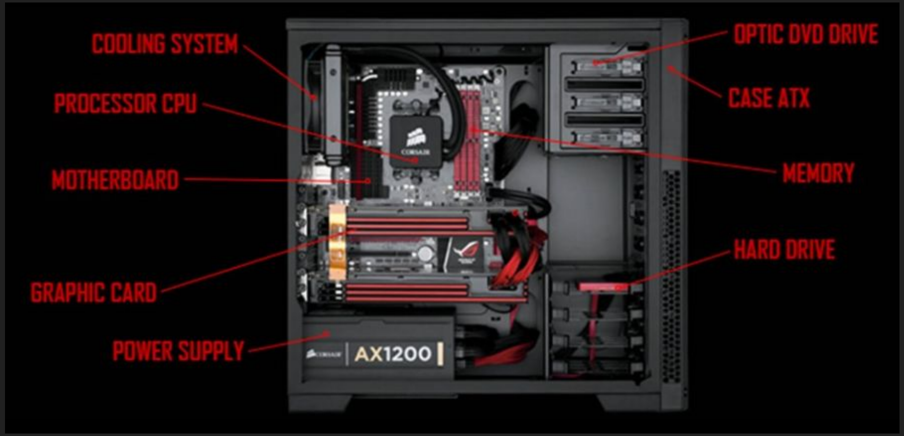


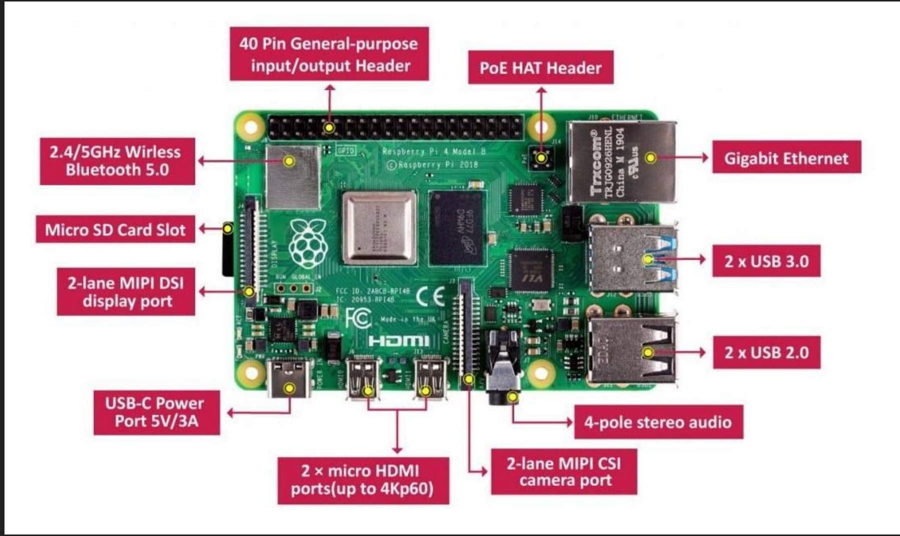
The Motherboard (cont.)



Some Extras

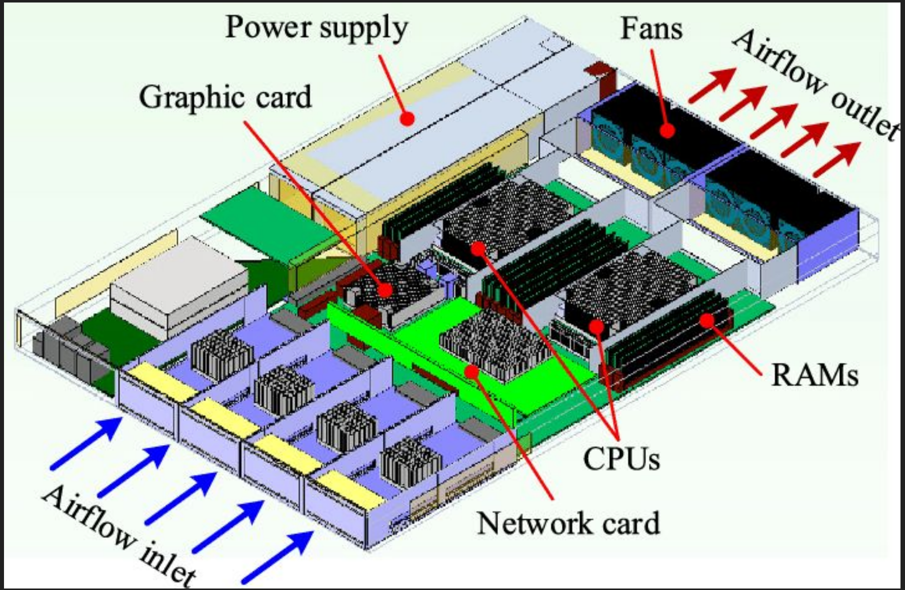
- The CPU, RAM, and Storage are our main components
- We need something to power our computer
  - Power Supply Unit (PSU)
  - Battery (embedded system)
- We *might* need some other things
  - Mouse, Keyboard, Graphics Card
  - Case
  - CPU cooler and fans
  - Network connectivity





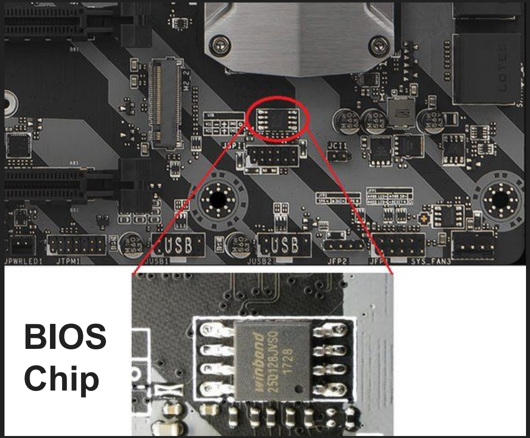
There are Many Ways To Design a Computer

- Computers are ubiquitous and complex
- Applications for computers vary but general purpose computers can do a lot
  - Desktops/laptops
- Before trying to solve a problem, ensure you’re working with the correct hardware
  - Locally running a LLM AI on a Raspberry Pi is technically possible but produces lackluster results



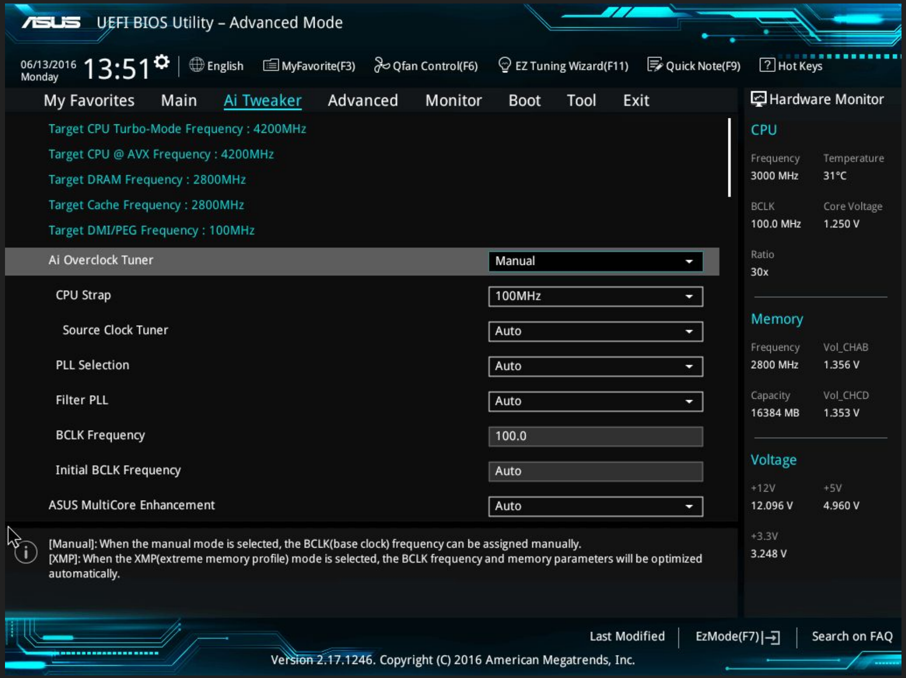
The Boot Process

1. The power button is pressed
   a. The PSU begins sending power to the system
2. The Basic Input and Output System (BIOS) prepares the hardware and loads the bootloader from the Master Boot Record (MBR) into memory
   a. The BIOS is a program stored on a chip on the motherboard usually 16MB max size
   b. The MBR is the first 512 bytes on the storage device
   c. Power-On Self Test (POST) initializes RAM, search for storage, USB devices, performs quick tests (i.e. keyboard check), initializes video card
   d. If a bootable disk is found, start its bootloader



Unified Extensible Firmware Interface

- More modern machines use Unified Extensible Firmware Interface (UEFI) instead of BIOS
- UEFI does everything BIOS does and more
  - Provides nice GUI interface
  - Mouse support
  - Secure boot
  - Faster boot times
  - More options for configuring boot
    - Up to 128 physical partitions (BIOS has 4)



The Boot Process (cont.)

3. The bootloader stored in the MBR sets up hardware for the OS and loads the kernel into memory
   a. The bootloader is a very small (512 bytes maximum) program
   b. The bootloader must run in “Real Mode” and will switch to “Protected Mode” on x86 ISA to execute the kernel
   c. The MBR also specifies which partitions are available on the storage
   d. Sometimes, there is little bootloader code in the MBR and it simply jumps the computer’s execution to the active partition’s code
   e. The small bootloader in the MBR is the First-Stage Bootloader, the code in the main partition is the Second-Stage Bootloader
   f. Multiple partitions can be used to boot different OSs from the same storage drive

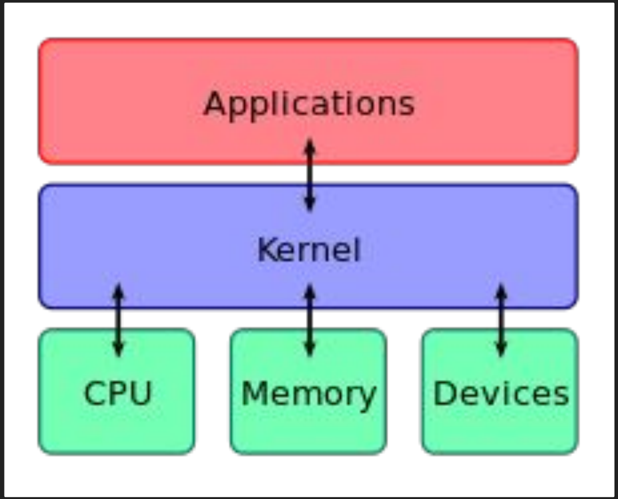
Structure of a classical generic MBR

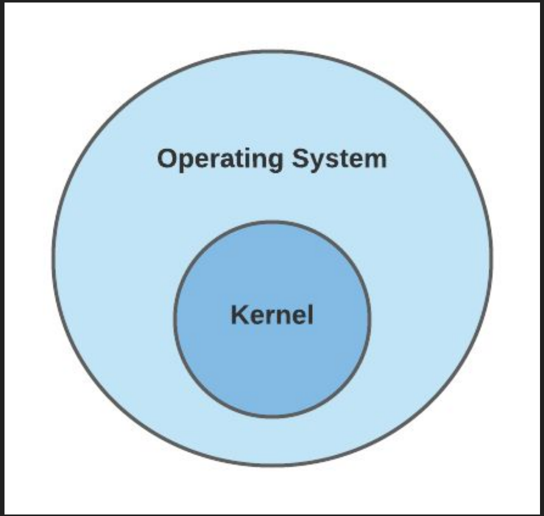
Address		Description		Size (bytes)
Hex	Dec			
+000$_{hex}$	+0	Bootstrap code area		446
+1BE$_{hex}$	+446	Partition entry №1	*Partition table* (for primary partitions)	16
+1CE$_{hex}$	+462	Partition entry №2		16
+1DE$_{hex}$	+478	Partition entry №3		16
+1EE$_{hex}$	+494	Partition entry №4		16
+1FE$_{hex}$	+510	55$_{hex}$	*Boot signature*[a]	2
+1FF$_{hex}$	+511	AA$_{hex}$		
			Total size: 446 + 4×16 + 2	512

Element (offset)	Size	Description
0	byte	Boot indicator bit flag: 0 = no, 0x80 = bootable (or "active")
1	byte	Starting head
2	6 bits	Starting sector (Bits 6-7 are the upper two bits for the Starting Cylinder field.)
3	10 bits	Starting Cylinder
4	byte	System ID
5	byte	Ending Head
6	6 bits	Ending Sector (Bits 6-7 are the upper two bits for the ending cylinder field)
7	10 bits	Ending Cylinder
8	uint32_t	Relative Sector (to start of partition -- also equals the partition's starting LBA value)
12	uint32_t	Total Sectors in partition

The Boot Process (cont.)

4. The kernel is now running and begins to initialize and execute the core functions of the operating system
    a. The kernel connects and manages resources for the computer's operating system
    b. Task management, memory management, and I/O interfaces
    c. The kernel sits between the hardware and the user's programs
5. The operating system is now running and takes inputs from the user and user's programs to do complex and useful tasks
    a. Running Netflix, playing video games, crypto mining, etc.





The Shutdown Process

- When the computer is instructed to shutdown, the OS may do some things first
  - It is not a good idea for the OS to “pull the plug”
- If there is unsaved file data in RAM, it needs to be saved to storage (or it is lost)
- If there are processes that are running, the OS should request them to terminate
  - This takes a long time because the processes may be in virtual memory (waiting in slow storage) and have to finish up before being forced to terminate
- Once processes have stopped and the file systems are unloaded, the kernel sends a signal to the BIOS that turns the PSU off

The Kernel

- The kernel is the lowest level of the operating system
  - There are many components and often involves low level programming (assembly)
  - Low level programming might be necessary to interface the hardware directly
  - Kernels are usually a combination of both C and assembly
  - Even though C is a high level programming language, it gets us close enough to the hardware most of the time

Execution Modes

- On x86 based processors we have two main CPU modes:
    - Real Mode
        - 16 bit instructions, 16 bit registers, 16 bit addresses
        - CAN call BIOS interrupts
        - No safety nets or protections
        - CPU boots into this mode
        - Kept for legacy purposes
    - Protected Mode
        - 32 bit instructions, 32 bit registers, 32 bit addresses
        - CANNOT call BIOS interrupts
        - Extra protections, multitasking, virtual memory
        - The CPU must be forced into this mode from running in real mode
        - All major OSs run in protected mode
- Protected mode is enabled by setting bit 0 to “1” in register CR0
    - It’s not that simple though
    - Switching back to real mode is not easy
- There are more than these two modes but these are what we will focus on for now

Hardware Abstraction Layer

- The OS must implement a Hardware Abstraction Layer (HAL)
- The HAL is a software component that acts as an interface between the hardware and the operating system
- At its most basic, we must have a HAL that allows us to:
  - Write to display
  - Write to storage
  - Read from keyboard
  - Read from storage

VGA Text Mode

- Using VGA text mode, writing to the display is the simplest hardware interface to implement
  - VGA text mode makes things easy, otherwise we need more advanced techniques, i.e. writing drivers for graphics card 😨
  - VGA text mode is limited, we cannot display nice graphics
- To set VGA text mode the computer must be in “Real Mode”
  - Once you are still in real mode, set AH = x00 and AL = x##
  - AL should be set to your desired BIOS Video Mode found here: https://www.minuszerodegrees.net/video/bios_video_modes.htm
  - Once these values are set call the INT 0x10 interrupt
- More info on INT 0x10 interrupt (including how to disable the blinking cursor) can be found here:
  - https://en.wikipedia.org/wiki/INT_10H

Write to Display

- You can write to the display by writing to memory locations starting at 0xB8000
- Each character on the display is comprised of two bytes, first the ASCII character and second the color
  - For example, if you want to display the character “a” in monochrome green at the first character of the display, you must set memory to:
    - 0xB8000 <- 0x61
    - 0xB8001 <- 0x2A

Write to Display (cont.)

- In the previous example:
  - 0xB8000 <- 0x61
  - 0xB8001 <- 0x2A
- 0x61 is an ASCII character ‘a’
  - Available characters: https://www.asciitable.com/
- 0x2A is the color code for green text on a light green background
  - Available colors: https://wiki.osdev.org/Printing_To_Screen
    - Hint: look for “Color Number” in the “Color Table”
  - The most significant 3 bits specifies the background color
  - The least significant 4 bits specifies the text color
  - 0x2A = 0<u>010</u> <u>1010</u>
- For light grey text on black background use 0x07
  - 0x07 = 0<u>000</u> <u>0111</u>

Write to Display (cont.)

- In C it is very easy to write to video memory:
  - char *vidmem = (char *) 0xB8000;
    - Creates a pointer pointing to address 0xB8000
  - vidmem[0] = 0x61;
    - Sets the first location the pointer points to
    - 0xB8000 to 0x61
  - vidmem[1] = 0x2A;
    - Sets the second location the pointer points to
    - 0xB8001 to 0x2A

Reading From Keyboard

- Reading from the keyboard is not as easy as reading from a memory location
- To read from the keyboard port 0x60 and 0x64 must be read from
  - Port 0x60 is NOT a memory address
  - Port 0x60 is the Keyboard Data I/O port
  - Port 0x60 is the keyboard data port (provides a scancode of what was typed)
  - Port 0x64 is the keyboard status port (indicates if the keyboard is ready to send data)
    - The least significant bit (bit 0) will indicate keyboard readiness (1 for ready, 0 for waiting)
- There are many different types of keyboards
  - US English, UK (British) English, Chinese, Spanish, etc.
  - To ensure support for these keyboards, scancodes are used (not ASCII)
  - There is no ASCII equivalent for special keys like SHIFT or INSERT so scancodes must be used to register all the keys
- Keyboard scancodes can be found here: https://wiki.osdev.org/PS/2_Keyboard

Reading From Keyboard (cont.)

- Reading from the port must be done in assembly:

  ```
  in al, 0x60              ; put byte from port 80 into al
  ```

  - The above code reads port 0x60 and puts the result into the AL register
  - If using inline assembly, the `inb` (input byte) and `inw` (input word) can be used for reading from ports
- The scancode must be converted to an ASCII character, which can be implemented with an array of ASCII characters mapped to scancode indices
- This assembly code can be called from a C program, or written in a C program using inline assembly, to create your own barebones scanf function
  - This is the preferred method to avoid having to swap between C and assembly in different .c and .asm files

Accessing Ports using Inline Assembly in C

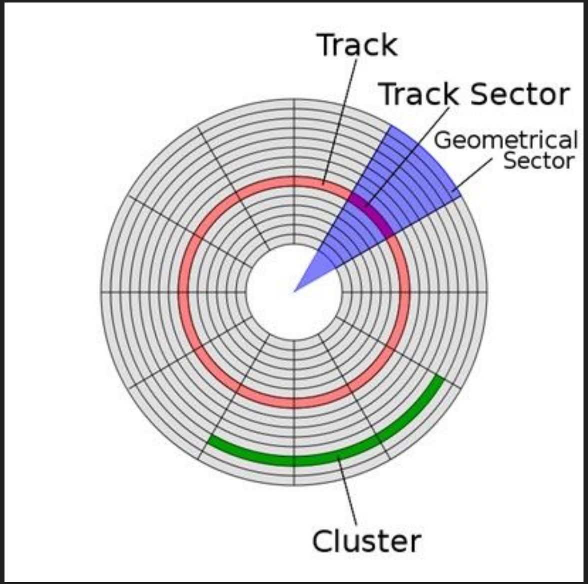
```c
typedef unsigned char  uint8;
typedef unsigned short uint16;

void outb(uint16 port, uint8 value)
{
    asm volatile ("outb %1, %0" : : "dN" (port), "a" (value));
}

void outw(uint16 port, uint16 value)
{
    asm volatile ("outw %1, %0" : : "dN" (port), "a" (value));
}

uint8 inb(uint16 port)
{
   uint8 ret;
   asm volatile("inb %1, %0" : "=a" (ret) : "dN" (port));
   return ret;
}

uint16 inw(uint16 port)
{
   uint16 ret;
   asm volatile ("inw %1, %0" : "=a" (ret) : "dN" (port));
   return ret;
}
```

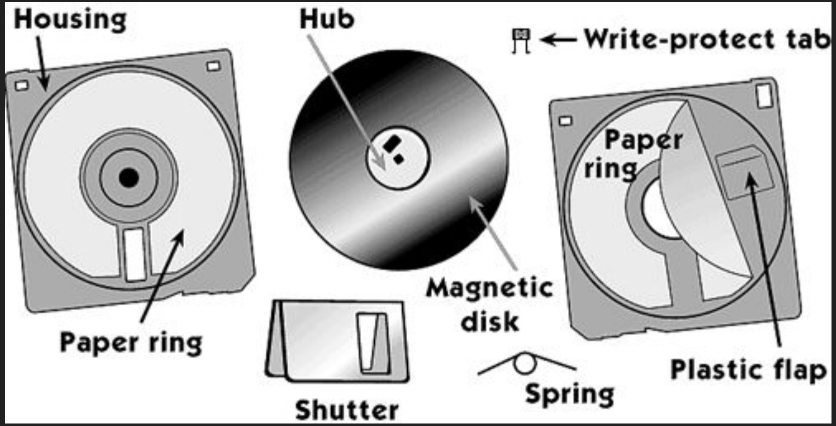
Reading from and Writing to Storage

- We will only focus on floppy disks for our storage
- Floppy disks are the easiest to interface with and are still supported to this day
  - Even though first introduced in 1967
- HDD or SSD drives are a more difficult storage to interface with
- Even though floppies are easier than HDD/SSD, it is still very complex
- For those interested in just how complicated this can get:
  - https://wiki.osdev.org/Floppy_Disk_Controller

Anatomy of a Floppy Disk

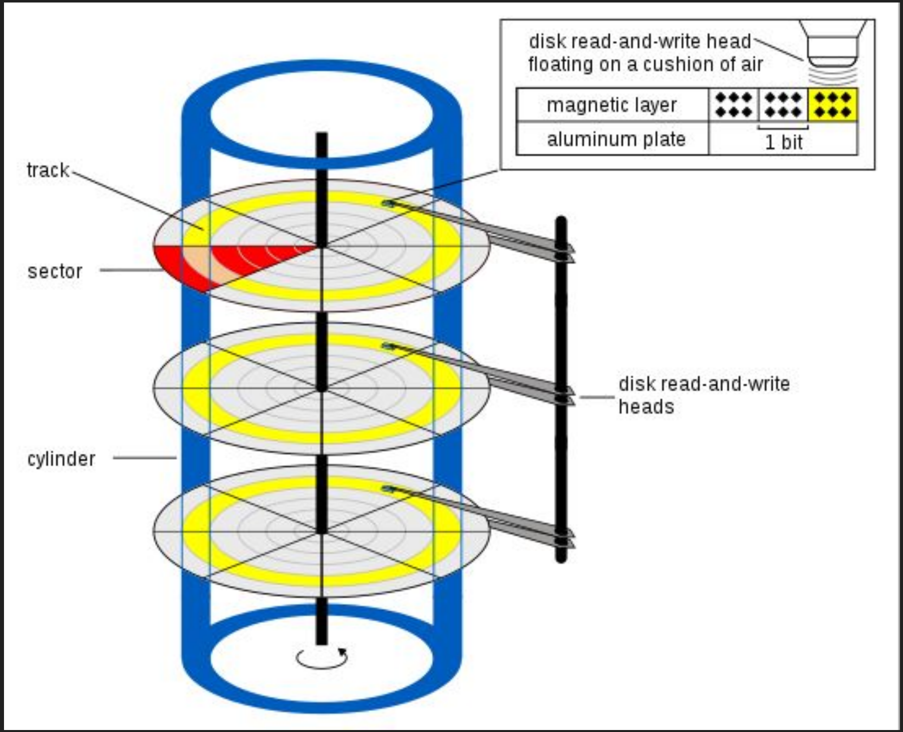
- 512 bytes per sector
- 18 sectors per track
- 80 tracks per side
- 2 heads
- Total 1,474,560 bytes per disk
- 3.5 inch 1.44 MB disks are the most common





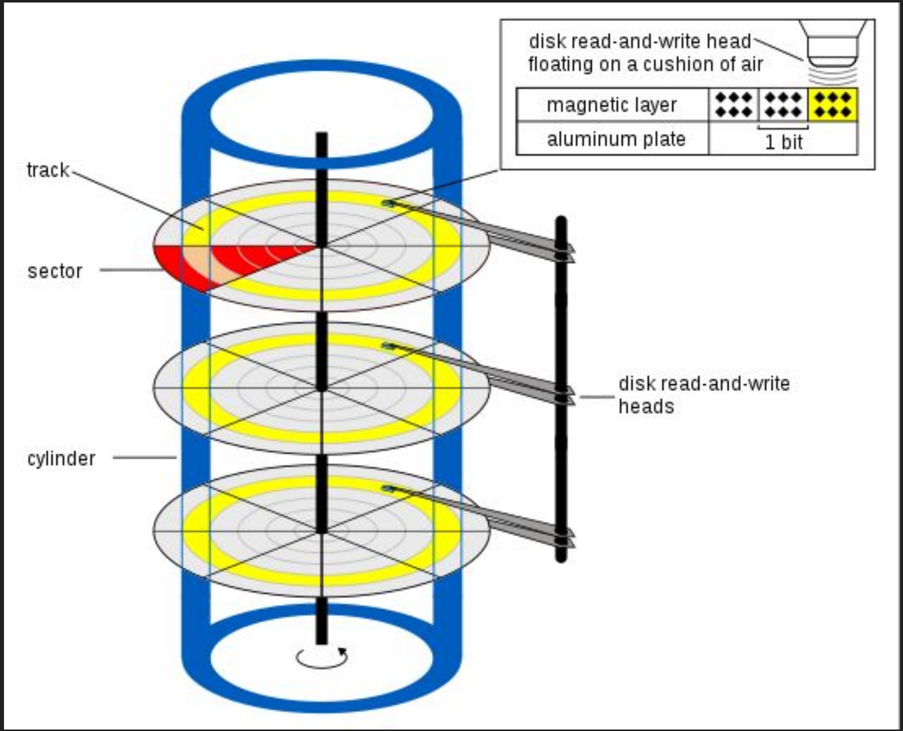
Cylinder-Head-Sector Addressing

- The cylinder-head-sector (CHS) addressing scheme is a way of accessing a specific memory location in storage
- This is an old school way of addressing and is required by INT 13,2 BIOS interrupt



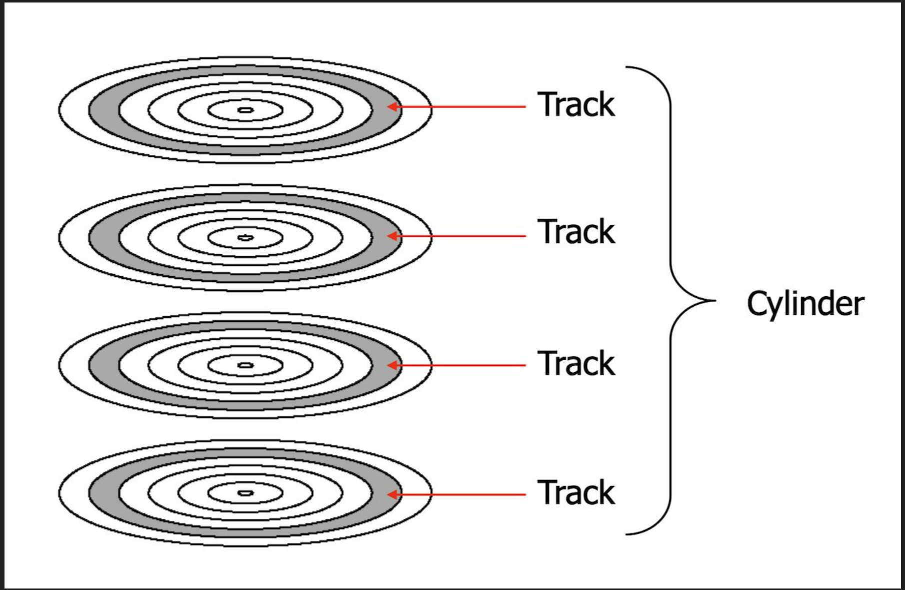
Cylinder-Head-Sector Addressing (cont.)

- To read from cylinder 0, head 0, sector 2:
  - CHS = 0,0,2
- Note: Sectors start at 1, there is NO sector 0!
  - Cylinder and head start from 0



Cylinder-Head-Sector Addressing (cont.)

- The cylinder is the aggregate of all tracks on all platters (disks)
- Cylinder count is not much of a concern today since most OSs use logical block addressing
  - Often the OS will report inaccurate numbers for the physical CHS that exists



Logical Block Addressing

- It is more intuitive to use logical block addressing (LBA)
  - We don't care where our data is physically on the drive, we just want to access it specifically
- LBA provides a linear address space for dealing with storage
- Instead of specifying the cylinder, head, and sector, the next sector is +1 value
- To calculate the CHS value for a given LBA value use the following equations
  - Hint: We know a floppy drive has 18 sectors per track and 2 heads

$$C = (\text{LBA} \;/\; \text{sectors per track}) \;/\; \text{number of heads}$$
$$H = (\text{LBA} \;/\; \text{sectors per track}) \;\%\; \text{number of heads}$$
$$S = (\text{LBA} \;\%\; \text{sectors per track}) \; + 1$$

LBA and CHS equivalence with 16 heads per cylinder

LBA value	CHS tuple
0	0, 0, 1
1	0, 0, 2
2	0, 0, 3
62	0, 0, 63
63	0, 1, 1
945	0, 15, 1
1007	0, 15, 63
1008	1, 0, 1
1070	1, 0, 63
1071	1, 1, 1
1133	1, 1, 63
1134	1, 2, 1
2015	1, 15, 63
2016	2, 0, 1
16,127	15, 15, 63
16,128	16, 0, 1
32,255	31, 15, 63
32,256	32, 0, 1
16,450,559	16319, 15, 63
16,514,063	16382, 15, 63