

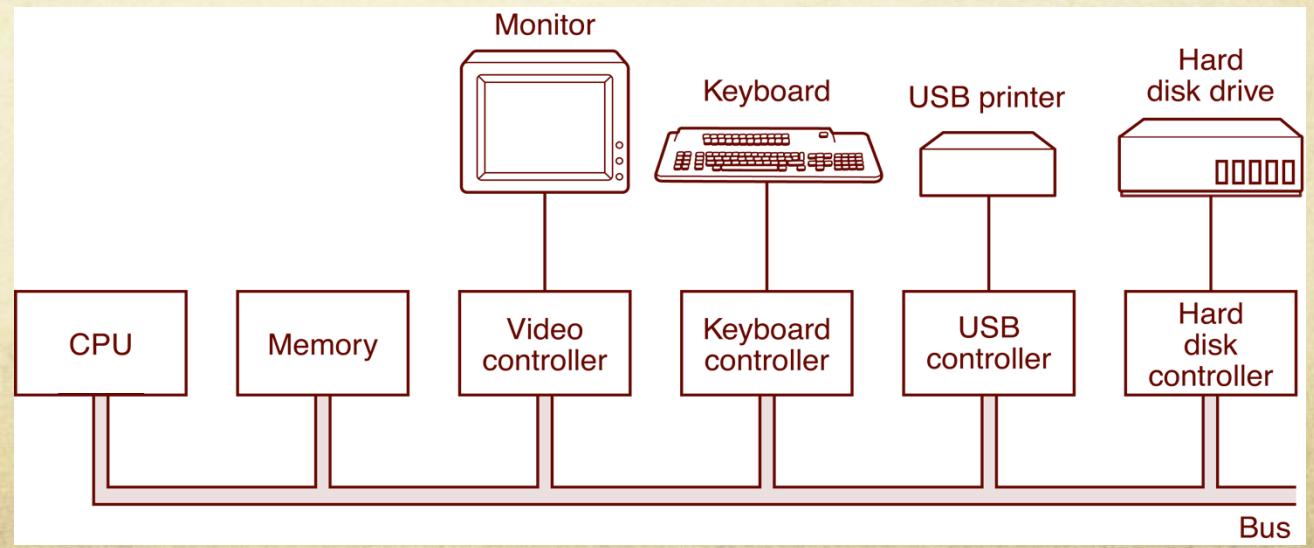


Computer Organization

A brief overview

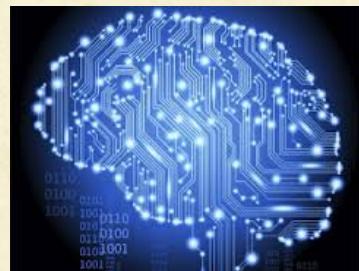
Main components of a PC

- ◆ Processor
- ◆ Memory and storage (e.g., hard disk)
- ◆ Other I/O devices
- ◆ Buses



Processor

- ◆ Central Processing Unit (**CPU**)
- ◆ *Brain* (i.e., *computation center*)



- ◆ Capable of executing specific set of instructions
 - ◆ Instructions have pre-determined codes
- ◆ Instructions may have input/output data

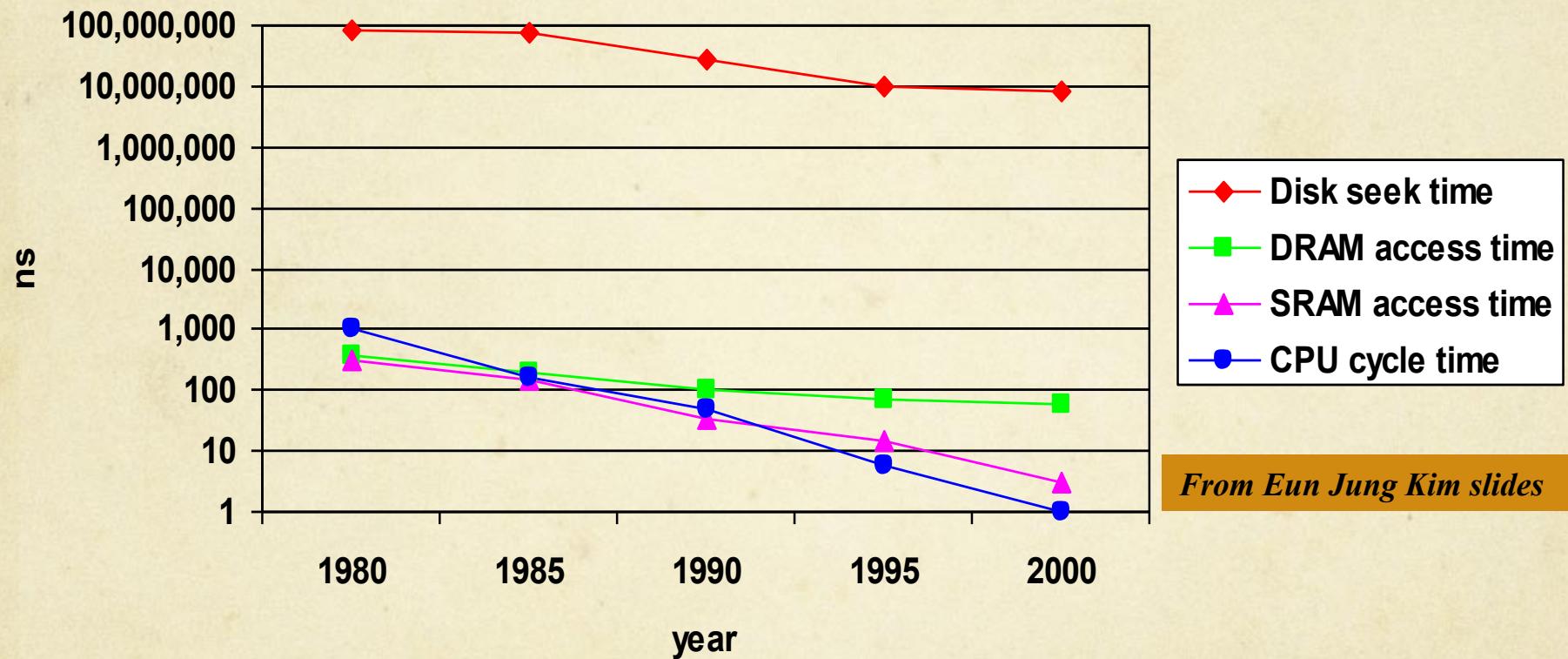
Memory & Storage

- ◆ Used to *store* instructions & data
- ◆ Permanent (*non-volatile*) → Retains data permanently
 - ◆ Read Only Memory (*ROM*)
 - ◆ Name misleading – some ROMs can be modified
 - ◆ Read/Write Memory (e.g., *disks*)
- ◆ Temporary (*volatile*) → Loses data when powered off
 - ◆ Random Access Memory (*RAM*)
 - ◆ Static RAM (*SRAM*)
 - ◆ Retains value indefinitely as long as power is on
 - ◆ Dynamic RAM (*DRAM*)
 - ◆ Value must be refreshed every 10-100 ms

Comparison

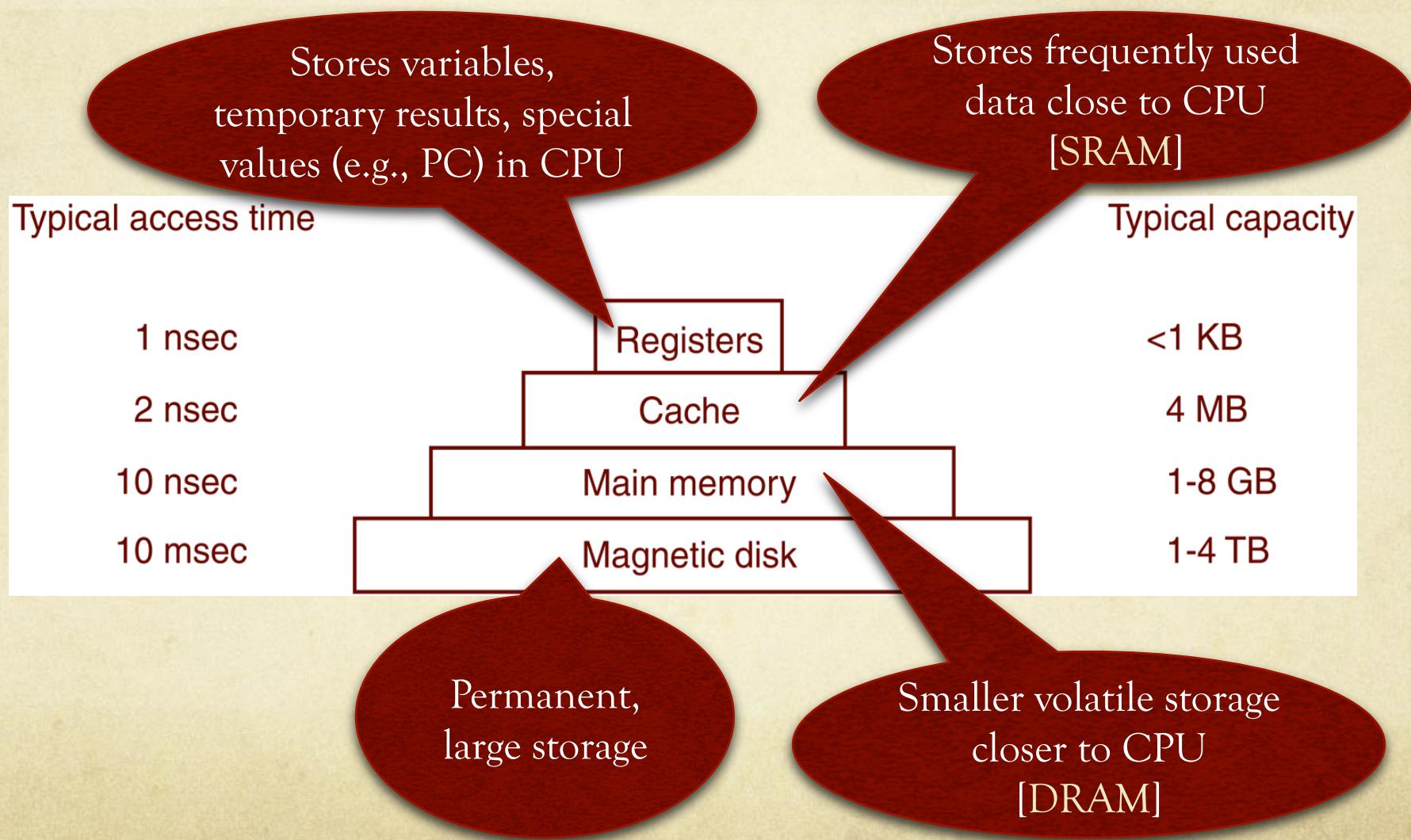
- ◆ Disk
 - ◆ Definitely need this for *permanent* storage
 - ◆ *Cheap* → *large* storage possible
 - ◆ *Slow* → bottleneck because CPUs are fast!
- ◆ DRAM
 - ◆ *More expensive* → *relatively small* storage possible
 - ◆ *Faster*, but still not fast enough!
- ◆ SRAM
 - ◆ *Fast...*
 - ◆ ...but *expensive* → *very small* storage possible

Memory performance



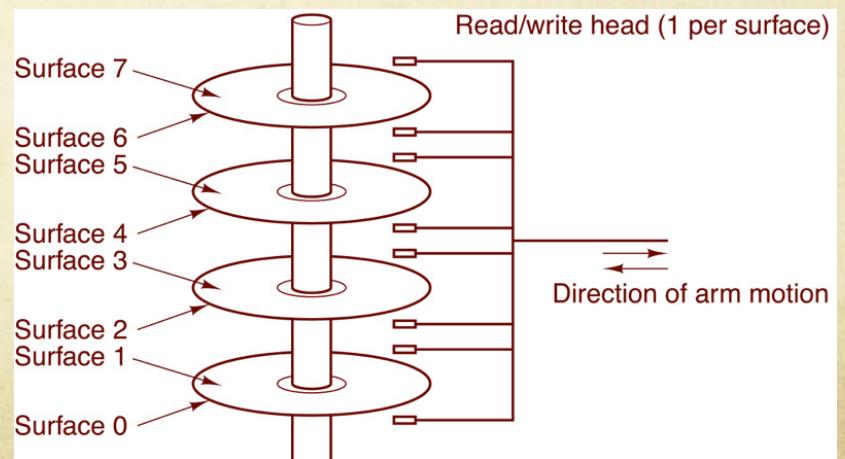
*Ideally, we want fast, cheap & large memory!
So, what should we choose?*

Best of all worlds...



Magnetic disk

- ◆ Very cheap, very large, non-volatile storage
- ◆ Mechanical device → random access of data is *very slow*
- ◆ Set of rotating metal platters
- ◆ Mechanical arms with read/write heads to access data
- ◆ Organized into *tracks* & *sectors*



Main memory

- ◆ Smaller volatile memory closer to CPU
- ◆ Collection of *cells* (1-bit each) organized into sets called *supercells*
- ◆ Each supercell has unique *address* used to *read/write* data
- ◆ Logically organized and handled in *lines/blocks* of supercells

| | |
|------|----------|
| 0x00 | 01101000 |
| 0x01 | 10101001 |
| 0x02 | 10100000 |
| 0x03 | 00000000 |
| 0x04 | 10001010 |
| 0x05 | 01010101 |
| 0x06 | 10010101 |
| 0x08 | 01100110 |
| 0x08 | 10010110 |
| 0x09 | 01101001 |

...

Cache

- ◆ Small, fast memory for frequently used data
 - ◆ Organized into lines/blocks of data similar to main memory
 - ◆ Works on principle of *locality of reference* (temporal/spatial)
- ◆ When CPU requests for data
 - ◆ If line containing data is already in cache → cache *hit*
 - ◆ If line containing data not in cache → cache *miss*
 - *Fetch line containing data from next level in memory hierarchy*
 - *(Typically) put line into current level of cache*
- ◆ System may have multiple *levels* of cache
 - ◆ Cache levels are typically *inclusive*

Registers

- ◆ Very small, very fast memory in CPU itself
- ◆ Special registers
 - ◆ Program counter (*PC*)
 - ◆ Stack pointer (*SP*)
 - ◆ Stores address of top of stack
 - ◆ *Stack* → area of memory containing data for procedures (functions) that have started, but not completed
 - ◆ Program status word (*PSW*)
 - ◆ Stores bits relevant to current state of system (more later)
 - ◆ ...

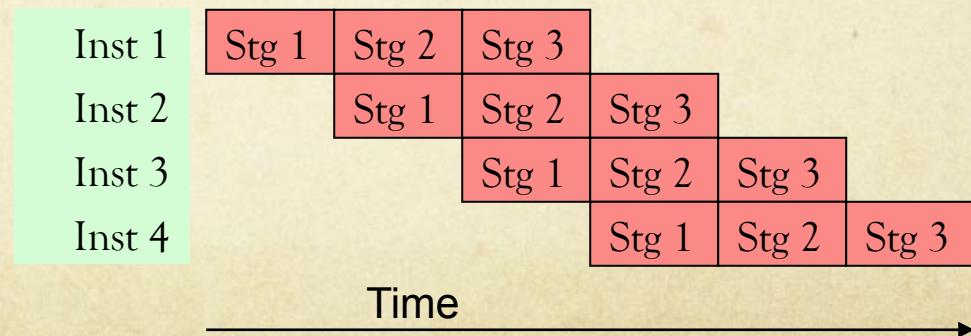
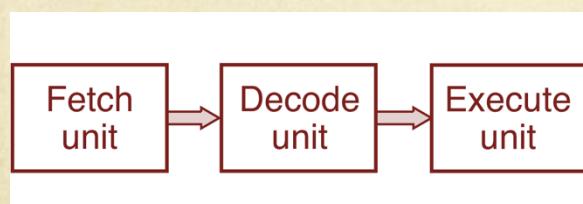
Instruction execution

- ◆ General sequence of operations
 - ◆ Retrieve next instruction from memory (*fetch*)
 - ◆ Program Counter (**PC**) has address of instruction to fetch
 - ◆ Figure out what type of instruction it is (*decode*)
 - ◆ Each type of instruction has a specific format
 - ◆ Fetch *data* – if needed – from memory
 - ◆ *Execute* instruction
 - ◆ *Store* results
 - ◆ Update PC to store address of *next* instruction



Perceived performance is important

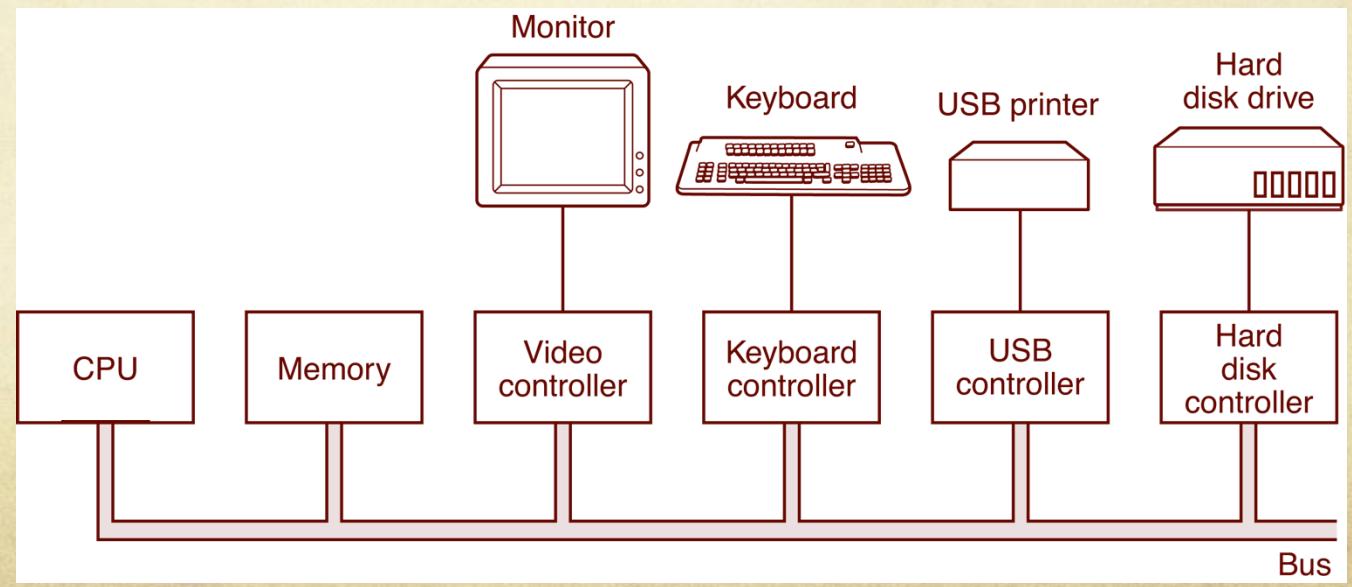
- ◆ Process one instruction fully, then next → simple, but slow
- ◆ Split instruction execution sequence into *stages*
- ◆ Multiple instructions can make progress concurrently
- ◆ This idea is known as *pipelining* → similar to assembly line



Very simple 3-stage pipeline

Input / Output (I/O) devices

- ◆ Monitor, keyboard, (disk), printer, etc...
- ◆ Physical device operation is very complex
 - ◆ Each device has *controller* h/w that directly interacts with it
 - ◆ Controller provides simpler *interface* to device



CPU – I/O device interaction

- ◆ CPU & I/O devices work asynchronously
- ◆ Need a way for CPU to know when device
 - ◆ Has input data to give to CPU
 - ◆ Is ready to receive commands or output data from CPU
 - ◆ Has completed command issued by CPU

Modes of I/O operation

- ◆ One option...



<https://www.youtube.com/watch?v=4vUBsTJYK28>

Modes of I/O operation

- ◆ One option...
 - ◆ CPU *stops* current activity
 - ◆ CPU *asks* if device is ready, e.g.
 - ◆ Device has new input
 - ◆ Device is done with previous command/output
 - ◆ If yes, CPU services device
 - ◆ CPU moves to next device
 - ◆ Once done, CPU *resumes* previously stopped activity
 - ◆ CPU repeats above steps *periodically*
- ◆ *This concept is called polling*

Modes of I/O operation

- ◆ Another approach
 - ◆ CPU sends commands/data to device
 - ◆ CPU goes back to other activity
 - ◆ Device signals completion to CPU using *interrupt*

Interrupt

- ◆ *External event that causes change in flow of current execution*
- ◆ Sequence of activities...
 - ◆ CPU executing instructions
 - ◆ *Interrupt* received
 - ◆ CPU
 - ◆ Finishes current instruction
 - ◆ Recognizes interrupt
 - ◆ Saves current state
 - ◆ Services interrupt
 - ◆ Resumes normal activity

Interrupt

- ◆ Handling code in special function called *interrupt handler*
- ◆ Interrupt handler *services* interrupt
- ◆ *Different* handlers for different types of interrupts
- ◆ Handler may trigger another function for *extended service*

Modes of I/O operation

- ◆ Direct Memory Access (**DMA**)
 - ◆ CPU initiates data transfer
 - ◆ Transfer coordinated by special DMA hardware
 - ◆ *Interrupt* generated upon completion