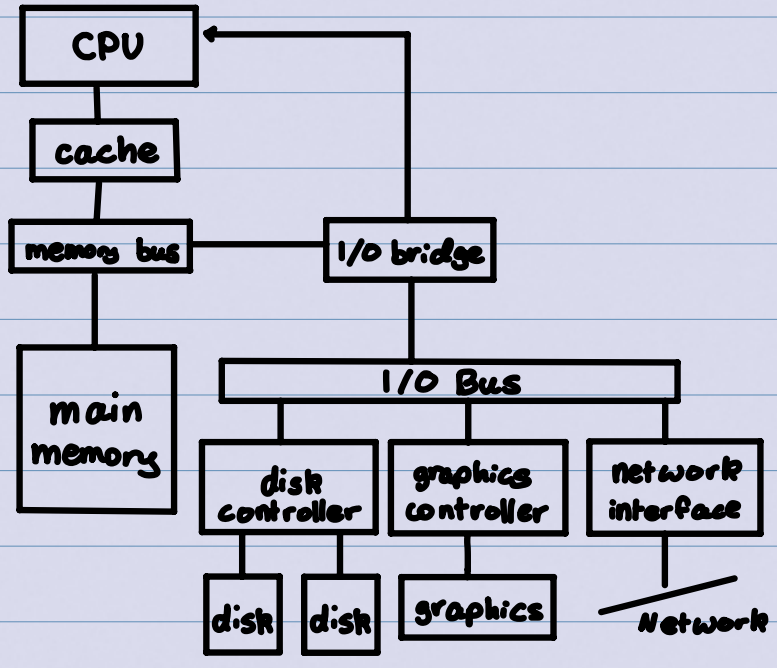
Lecture 1 - 4th Sept 2024

Instruction Set Architecture: how **hardware executes software**

- **What is a computer?**
  - ↳ just a machine that does whatever the software tells it to do.
  - ↳ computer > digital circuits > logic gates > transistors
  - ↳ software is just a series of instructions.
  - ↳ The five classic components of a computer:
    - CPU: Central Arithmetic (ALU) and Central Control
    - Memory System
    - Input and Output

↳ But, here's a more realistic view:



Architecture vs Microarchitecture

↳ **Processor architecture:**

- Functional appearance to software (in this case, compilers!) (ISA)
  - Exactly what instructions does it have?
  - Number of memory/storage locations it has
  - **Interface!**

↳ **Processor microarchitecture:**

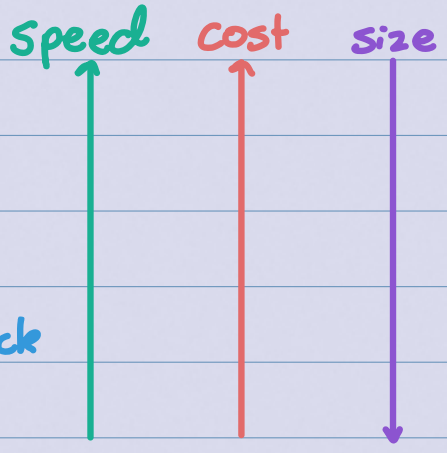
- **Logical structure that implements the architecture**
  - Number of functional units, interconnection, control
  - Size of the caches
  - Not visible to the software
  - **Implementation!**

Memory Hierarchy:

- CPU
- CPU Cache

↗ processor

- Physical Memory → RAM
- Solid State Memory → SSD/USB stick
- Virtual Memory → hard drive

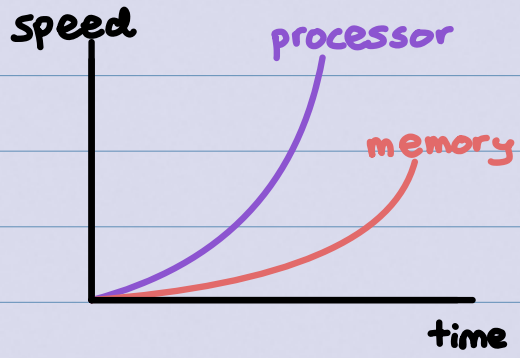


- **Moore's Law:** every **18-24 months,**
  - **2x transistors** on same chip area
  - **2x processor speed**
  - **2x memory capacity**
  - **½ energy/power consumption.**

↳ technically not a law, but it's a **self-fulfilling prophecy.**

Memory Wall: every 2 years,

- 2x Instructions / second
- 2x memory capacity
- 1.1x memory latency



↳ growing disparity between processor and memory performance!
∵ significant effort in reducing / hiding memory latency.

Latency: time from start to finish (aka response time).

Throughput: number of tasks completed per time unit (aka bandwidth)

↳ throughput can exploit parallelism, but latency cannot!

- Improving the latency of a component always improves the overall system throughput.

$$\text{Speedup} = \frac{\text{Performance}(x)}{\text{Performance}(y)}.$$

"x is "speedup" times faster than y"

↳ can be broken down into $\frac{\text{throughput}(x)}{\text{throughput}(y)}$ or $\frac{\text{latency}(y)}{\text{latency}(x)}$.

don't forget - latency will likely be < 1, so need to invert fraction!

Iron Law of Performance:

$$\text{CPU Time} : \frac{\text{instructions}}{\text{program}} \cdot \frac{\text{cycles}}{\text{instruction}} \cdot \frac{\text{seconds}}{\text{cycle}}$$

aka CPI

IPC is inverse - instructions per cycle

Amdahl's Law: $\text{Speedup} = \dfrac{1}{(1 - \text{Frac}_{EHN}) + \dfrac{\text{Frac}_{EHN}}{\text{Speedup}_{EHN}}}$

Example:

- Enhancement 1: speedup of 20 on 10% of time

  ↳ $\dfrac{1}{(1-0.1) + \frac{0.1}{20}} = \dfrac{1}{0.9 + 0.005} = 1.105.$

- Enhancement 2: speedup of 1.6 on 80% of time.

  ↳ $\dfrac{1}{(1-0.8) + \frac{0.8}{1.6}} = \dfrac{1}{0.2 + 0.5} = \dfrac{1}{0.7} = 1.43.$

∴, the second enhancement is better → even though the first enhancement has a speedup of 20, it's only helpful 10% of the time.

↳ "make the common case fast" !!

Power Consumption: transistors consume / dissipate power in two ways:

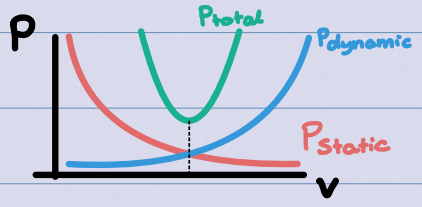
- **Dynamic Power:** Consumed by activities in a circuit (switching transistors)

  ↳ $P = \frac{1}{2} C \cdot V^2 \cdot f \cdot a$

  where: C = capacitance (~ chip area)
  V = power supply voltage
  f = clock frequency
  a = activity factor

- **Static Power**: consumed when powered on but idle (leaking current)
  - ↳ as voltage decreases, leakage increases.

- we strive for the "sweet spot" between static and dynamic power:



- **Power Wall**: supply voltage (generally) decreasing over time.
  - ↳ emphasis on power starting ~2000.
    - Since ~2000, we've "hit a wall" and aren't making as much progress.

- **What is an ISA?**
  - ↳ Functional and precise specification of a computer.

- An ISA is a "contract" between the software and the hardware.
  - Specifies what hardware promises to do when it sees certain instructions, but not how it does it.

CISC vs RISC:

- Early trend was to add more and more instructions to new CPUs to do elaborate operations
  - ↳ CISC (Complex Instruction Set Computing)

- Now, we keep the instruction set small and simple and let the software do the complicated operations by composing simpler ones.
- This makes it easier to build fast hardware.
  ↳ RISC (Reduced Instruction Set Computing)

- Instruction Set:
  - Arithmetic and logic: add, subtract, multiply, divide, AND, OR, XOR, ...
  - Data movement: move, load, store
  - Control flow: branch and jump
  - System (privileged): used to manage processor state, handle exceptions, etc.
  - Each instruction has a specific format and encoding

Registers
- General-Purpose Registers: can be used for various purposes as determined by the programmer or compiler.
- Special-Purpose Registers: have specific roles, such as:
  ↳ Program Counter (PC): holds the address of the next instruction to be executed.
  ↳ Stack Pointer (SP): points to the top of the stack in memory
  ↳ Status Register: holds flags that indicate the results of operations (eg, zero flag).

Memory Model:

this is virtual memory, not physical!

- **Address Space:** defines the **range of memory locations** that can be addressed
  - ↳ **Eg:** a 32-bit address space can address $2^{32} = 4\,GB$.
- **Byte Ordering:** specifies whether multi-byte values are stored with the most **significant byte first** (**big-endian**) or **last** (**little-endian**)
- **Alignment Requirements:** some ISAs require **data to be aligned** in memory **in specific ways.**
  - ↳ **Eg:** 32-bit values may need to be stored at addresses that are multiples of four.

Addressing Modes:

- **Immediate:** the **operand** is included **directly** in the instruction itself.
  - ↳ often used for **small constants** and **quick arithmetic operations.**
- **Register:** the **operand** is stored in a CPU register
  - ↳ often used for **frequently-accessed variables** and **intermediate results.**
- **Direct:** the **instruction contains the memory address** where the operand is located.
  - ↳ often used for accessing **static variables** or **fixed memory locations.**
- **Indirect:** the **instruction specifies a register that contains the memory address** of the operand
  - ↳ often used for accessing **pointer-based structures.**