1 Introduction

1.1 Course Introduction

> Computer Architecture is designing the organization and hardware to meet goals and functional requirements. (by *Hennesy* & *Patterson*, *CAAQA 5th edition*)

Computer Architecture

- ISA (Instruction Set Architecture)
- Micro-architecture
- Hardware implementation

This COURSE mainly focus on ISA and Micro-architecture.

Why study this COURSE

- Technology is always changing
- Requirements are always changing.
- Understand computer performances.

1.2 Technology and Trends

> **Moore's Law**: The number of transistors on a microchip doubles every two years, though the cost of computers is halved. (by *Gordan Moore*, 1965)
>
> **Dennard Scaling**: As transistors get smaller, their power density stays constant. In other words, power usage of transistors stays in proportion with area or both voltage and current scale (downward) with length. (by *Robert H. Dennard*, 1974)
>
> - The *Dennard Scaling* has expired in 2005.
> - The *Moore's Law* still has effect currently.

The History of Computer Architecture: 4 revolutions.

- **Revolution I**: the invention of micro-processors.
- **Revolution II**: extract implicit instruction-level parallelism. (*RISC*)
- **Revolution III**: support explicit data & thread level parallelism. (*multi-core*)
- **Revolution IV**: focus on heterogeneous processing (*Graphics Processing Units, GPU* or *Special-purpose logic*).

Current Trend:

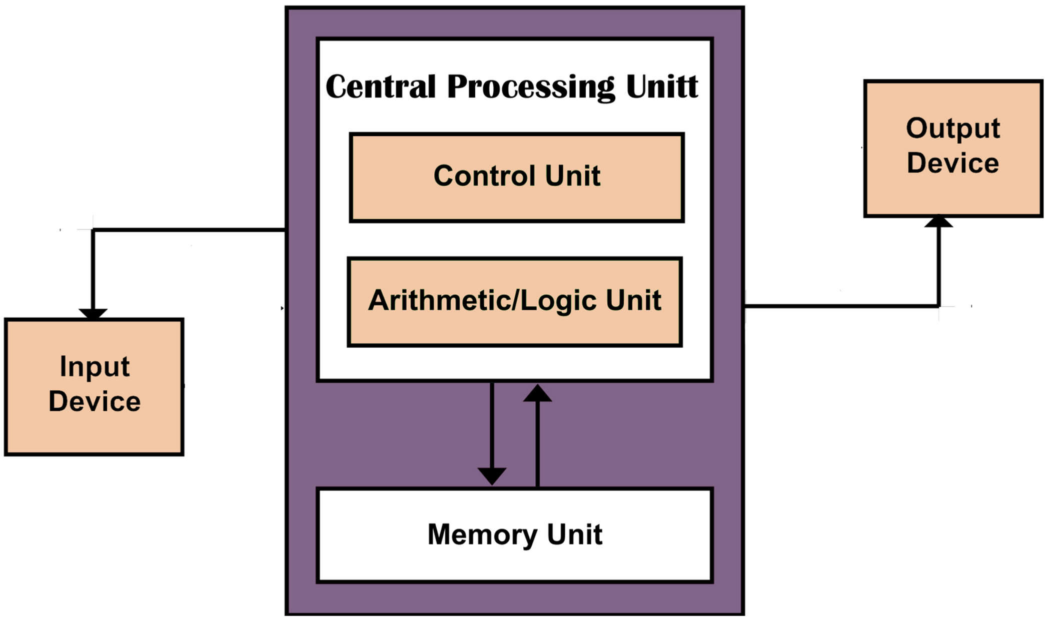
- Complex processor designs;
- Parallelism at chip level;
- Power-conscious designs;
- Specialization: domain-specific processors;
- Open source hardware: RISC-V;
- Security.

1.3 The Basic Architecture of Computer

ENIAC (1946, *UPenn*) The first computer. (*no memory*)

EDVAC (1951, *UPenn*) The first computer with memory, based on *Von Neumann Architecture*.

Von Neumann Architecture



Von Neumann Architecture

The introduction of the Von Neumann Architecture can refer to notes of *EI209: Computer Organization*.

- **Stored Program**: The programs (instructions) are in the *memory*, and the computer executes sequentially.
- **Take memory as center**: This can free *CPU* from input/output work, enable *CPU* to do other work when outer devices communicate with *memory*.

1.4 The Execution of a Program

Languages in Computer

- High-level Language (*such as C, C++, etc.*)
- Assembly Language (by *compiler*)
- Machine Language (in binary) (by *assembler*)

Memory capacity: the capacity of memory. Unit: *bit, byte(B), KB, MB, GB, TB, PB*.

Register

- General Register: $R_0, R_1, \dots, R_{n-1}$, save the intermediate result in a calculation.
- Special Register.
  - **MAR**: Memory Address Register, connected to the *address bus*, is the only way for CPU to communicate with *address bus* (***read-only***). Both an *instruction address* and a *data address* can be stored in *MAR*.
  - **MDR**: Memory Data Register, is connected to the *data bus*. Data can go in both directions, to and from memory (***read and write***).
  - **PC**: Program Counter, store the current instruction's address. When the instruction is executed, *PC* will go forward automatically to the next instruction's address.
  - **IR**: Instruction Register, store the current instruction's code.

Main Memory: *memory*, *MAR* and *MDR*.

Controller: *CU*, *PC* and *IR*.

The Execution of an Instruction

- Get the instruction from *main memory* according to *PC* and put it into *IR*;

- *CU* analyze the instruction in *IR*;
- *CU* control the other part of the computer to finish the instruction (by sending *signal*).

The Execution of a Program execute each instruction in a certain order (usually *sequential*).

> **Instruction Length**: The length of an instruction.
>
> **Storage Length**: The length of *MDR*.
>
> **Word Length of a Computer** The length of data that the controller can handle at one time.

1.5 The Performance of Computer

Performance Metrics

- **Latency** (execution time, response time): time to complete a *given* task, or a *fixed* task.

$$Performance \propto \frac{1}{Latency}$$

- **Throughput** (bandwidth): the rate of completion of tasks; number of tasks per unit time.
    - Exploit parallelism for throughput, not latency. As a result, *improving bandwidth is easier than improving latency*.

Performance Speedup Ratio

$$R = \frac{Performance_X}{Performance_Y}$$

where, R is the performance speedup ratio.

> [*Example*] $Performance_X = 1.5$, $Performance_Y = 1.0$, then $R = 1.5$, we can say:
>
> - X is 1.5 times faster than Y;
> - The speedup ratio of X to Y is 1.5;
> - The performance of X improves 50% comparing to Y.

CPU Performance

$$CPU\ Time = CT \times \sum_{i=1}^{n}(IC_i \times CPI_i) = CT \times IC \times \overline{CPI}$$

where

- $CPU\ Time$: execution time;
- IC: total number of instructions;
- IC_i: number of instructions executed (instruction count) in instruction group i;
- CPI_i: number of average clock cycles per instruction in instruction group i;
- $\overline{CPI}$: average clock cycles per instruction.
- CT : duration of processor clock.

How to Improve CPU Performance (each part):

- **IC**: Instruction Count.
    - Compiler optimizations (*constant folding*)
    - ISA (*more complex instructions*)
- **CPI**: Cycle Per Instruction.
    - Microarchitecture (*pipelining*, *Out-of-order execution*, *branch prediction*)
    - Compiler (*instruction scheduling*)
    - ISA (*simpler instructions*)

- **CT**: Clock Time.
  - Technology (smaller transistors)
  - ISA (Simple instructions that can be easily decoded)
  - Microarchitecture (simple architecture)

> **Misunderstandings**
>
> - Only focus on **CT**.
> - Only focus on **MIPS** (Million Instructions Per Second) or **FLOPS** (Floating Point Operation Per Second)
> - **CPI** depends on **CT**.
>   - In fact, **CT** and **CPI** are irrelevant.

1.6 Principles of Architecture Design

Principles of Computer Architecture

- Take advantage of parallelism.
  - System level: multi-processors, multi-disks, multi-channels.
  - Processor level: operate on multiple instructions at once (*pipelining*, *superscalar issue*)
  - Circuit level: operate on multiple bits at once (*carry-lookahead ALU*)
- Focus on the common case. (*RISC design principle*)
  - Common case first.
  - *Amdahl's law*.
- Principle of locality. (*Catches*)
  - Spatial and Temporal Locality.
  - 90% of the program executing in 10% of the code.

Amdahl's Law

$$S = \frac{1}{(1-P)+P/s}$$

where,

- P : proportion of running time affected by optimization.
- S : total speedup ratio.
- s : the speedup ratio in a part, $s = n$ in parallelization conditions.

Amdahl's Law requires extremely parallel code to take advantage of large multi-processors.

Amdahl's Law is not suitable for the quantity-extensible problem.

> **Little's Law** is a theorem that determines the average number of items in a stationary queuing system based on the average waiting time of an item within a system and the average number of items arriving at the system per unit of time.
>
> $$L = \lambda W$$
>
> where,
>
> - L: items in the system;
> - λ: average arrival rate; the average number of items arriving at the system per unit of time;
> - W: average wait time.
>
> *Assumptions*: system is in steady rate, i. e., average arrival rate = average departure rate.

> Works on any queuing system and even systems of system.
>
> **Application** to get high λ (throughput), we need either:
>
> - low W (latency per request);
> - high L (service requests in parallel).