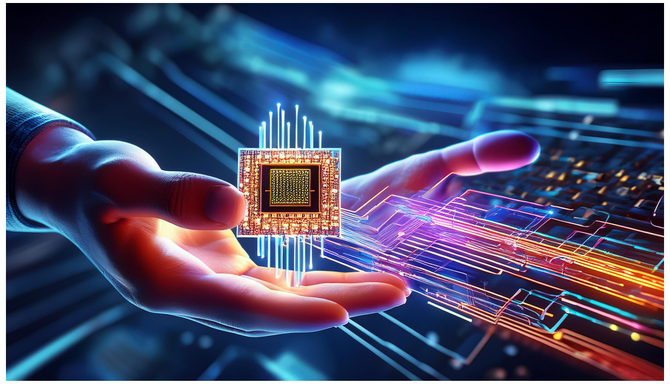
01.10.24 17:09

# CSCI 250
# Introduction to Computer Organisation
# Lecture 2: Computer Memory III



Jetic Gū

2024 Fall Semester (S3)

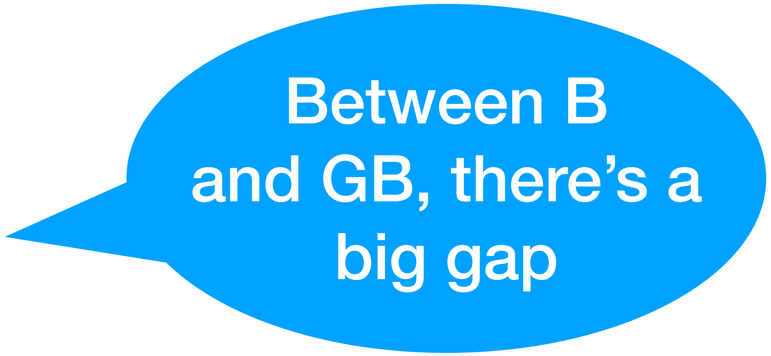
# Overview

- Focus: Course Introduction
- Architecture: Logical Circuits
- Textbook: v4: 13.1
- Core Ideas:
    1. Memory Hierarchy
    2. Cache
    3. Multi-level Cache

# Memory Hierarchy

# Multiple Levels of Storage

- Storage Devices: Hard Drive/SSD
  - Slow, non-volatile, can be TBs
- Main Memory: Random Access Memory
  - Faster, volatile, can be GBs
- CPU: Registers
  - Very fast, volatile, can be Bs



# Memory is a significant Bottleneck

- When you run a computer programme for computation, how much time is spent in accessing information, how much time is actually spent on computation?

P1
Memory Hierarchy

# Consider Sorting: Bubble Sort

- Time complexity: $O(n^2)$
  - Comparison: $O(1)$, 1 CPU cycle
  - Value assignment: memory access
    3-5 CPU cycles+
  - Value retrieval: memory access
    3-5 CPU cycles+
  - memory vs comparison ratio:
    7:1 +

```
def sort(a):
  flag = true                    memory access
  while flag:                    memory access
      flag = false               memory access
      for i in range(len(a)-1):  memory access
          if a[i] > a[i+1]:      memory access + comparison
              swap(a[i], a[i+1]) memory access
              flag = true        memory access
```

- Keep sorting until no adjacent elements are out of order

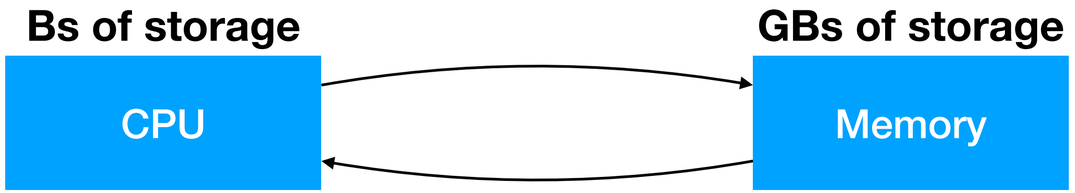
Technical

# Consider: Matrix Multiplication

- Time complexity: $O(nmk)$
  - Multiplication: 2-4 CPU cycles
  - A single memory access
  - Memory access required per multiplication: 5-7+
  - How much time spent on memory access: more than 80%, realistically more than 90% easily!

- $(n \times m) \times (m \times k)$
- $c_{n,k} = \Sigma_{i=1}^{m}(a_{n,i} \times b_{i,k})$
- Keep sorting until no adjacent elements are out of order

# In a lot of applications, Memory is the biggest bottleneck




- Solution: if memory is too slow, registers have too few slots, can we have something in between? Yes, that's called **Cache**

- Cache:
  **faster** than memory, **slower** than register,
  **smaller** than memory, **bigger** than register

Concept

# Cache

# Cache

- Cache is like proxy, information is still stored in main memory
    - e.g. main memory has 8GB, we have 1 level of cache capable of 1KB
    - Since memory access can often be localised, we can retrieve 1KB from the main memory, every read/write operation is performed on Cache only first
    - Then:
      1) periodically sync between Cache and Main memory
      2) sync only when Cache runs out of space

Concept

# Simple Simulation

| | Memory Access | |
|---|---|---|
| 1 | FFFF0000h | Cache |
| 2 | FFFF0010h | Cache |
| 3 | FFFF23A0h | Main |
| 4 | FF2A14BCh | Main |
| 5 | FFF701A0h | Main |
| 6 | FFFF0110h | Main |
| 7 | FF2A1943h | Main |
| 8 | FFFF01A0h | Main |
| 9 | FF4D2042h | Main |
| 10 | FF3700E7h | Main |

**L1 Cache**

| |
|---|
| 1 |

**Block FFFF00h**

**Without Cache: 10 x 10 = 100 cycles**

**Cache has 1 block, can store 16x16 values**

**One 16x16 block: 80 cycles**

- Assume Level 1 Cache only, which uses SRAM so speed is comparable to registers (say 2 cycles); Assume main memory requires 10 cycles

# Simple Simulation

| | Memory Access | |
|---|---|---|
| 1 | FFFF0000h | Cache |
| 2 | FFFF0010h | Cache |
| 3 | FFFF23A0h | Main |
| 4 | FF2A14BCh | Main |
| 5 | FFF701A0h | Main |
| 6 | FFFF0110h | Cache |
| 7 | FF2A1943h | Main |
| 8 | FFFF01A0h | Cache |
| 9 | FF4D2042h | Main |
| 10 | FF3700E7h | Main |

| | L1 Cache |
|---|---|
| 1 | Block FFFF00h |
| 2 | Block FFFF01h |

**Without Cache: 10 x 10 = 100 cycles**

**Cache has 2 blocks, can store two 16x16 values**

**One 16x16 block: 84 cycles**

**Two 16x16 blocks: 68 cycles**

- Assume Level 1 Cache only, which uses SRAM so speed is comparable to registers (say 2 cycles); Assume main memory requires 10 cycles

P2
Cache

# Simple Simulation

| | Memory Access | |
|---|---|---|
| 1 | FFFF0000h | Cache |
| 2 | FFFF0010h | Cache |
| 3 | FFFF23A0h | Cache |
| 4 | FF2A14BCh | Main |
| 5 | FFF701A0h | Main |
| 6 | FFFF0110h | Cache |
| 7 | FF2A1943h | Main |
| 8 | FFFF01A0h | Cache |
| 9 | FF4D2042h | Main |
| 10 | FF3700E7h | Main |

| L1 Cache | |
|---|---|
| 1 | Block FFFFh |

**Without Cache: 10 x 10 = 100 cycles**

**Cache has 1 block, can one two $16^4$ values**

**One 16x16 block: 84 cycles**

**Two 16x16 blocks: 68 cycles**

**One $16^4$ blocks: 60 cycles**

- Assume Level 1 Cache only, which uses SRAM so speed is comparable to registers (say 2 cycles); Assume main memory requires 10 cycles

# Cache Management

- Factors to consider:
  - Cache is almost always managed by **hardware**, OS and programmes usually cannot manipulate cache directly
  - **Size of cache** (vs cost)
  - **Partitioning of Cache**: e.g. how much for a block, division between instructions and data, etc.
  - **Multiple levels of Cache**: Modern CPU in addition to above, has also multiple levels of cache: L1 is the fastest, sometimes inside the CPU chip, then L2 L3 could be in the CPU or on the motherboard

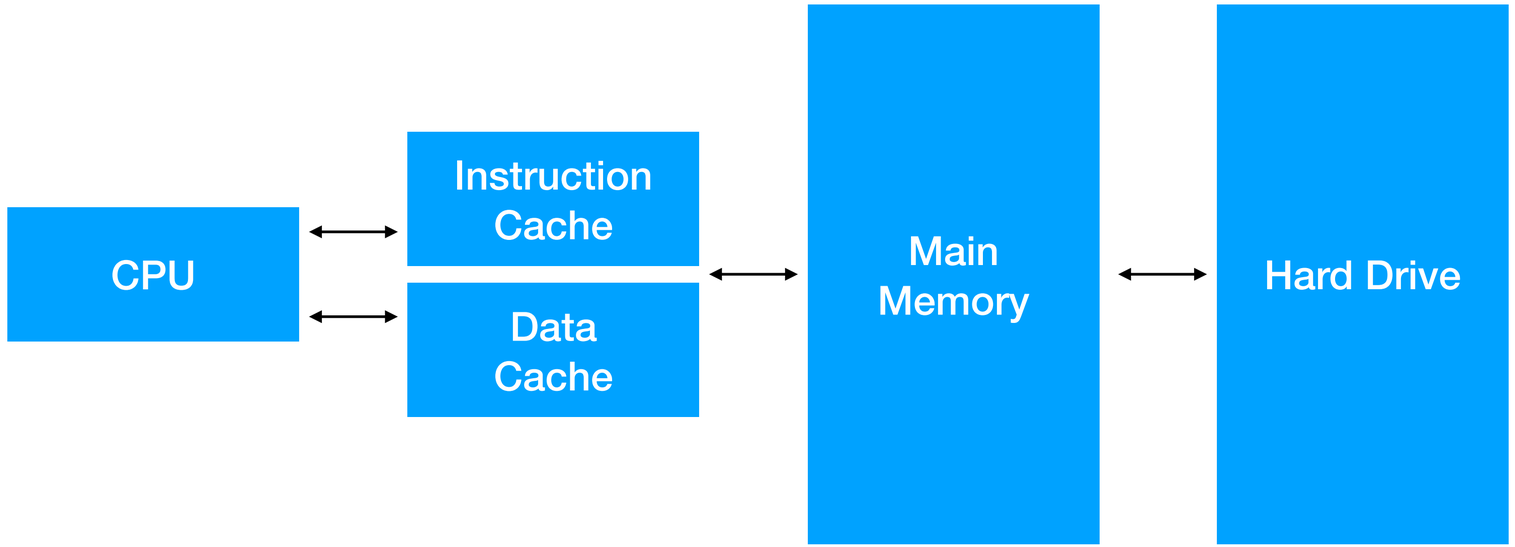
Concept

# Specialised Caches

- Instruction cache vs data cache
  - **Instruction cache** contains computer programme instructions
    This part is often **read-only**, and always put inside **Cache** for the fastest possible access
  - **Data cache** contains computer programme data, data structures, variables, dynamically allocated memory, etc.
    This part is RW, and can be shifted in and outside of **Cache** dynamically
  - Why does it matter? Shifting anything between memory and cache takes time, we want to focus on **data cache optimisation**

Concept

# Specialised Caches




- C/C++: **constant variables** vs **variables**
  `const` variables are stored in instruction cache, data cache can be moved in and out of the main memory more often, so can be a little slower at times[1]

1. Depends on hardware application

Concept

# Multi-Level Cache

P3
Multi-Level Cache

# L1-L3 Cache

- Modern Computers have multiple levels of cache, very commonly 3 levels
  - L1: Located on the CPU die
  - L2: On CPU or motherboard
  - L3: More commonly on the motherboard

Technical

# L1-L3 Cache

- Multi-Level cache parameters
  - Cache latency: L1 has the fastest latency, L2 next, L3 slowest
  - **Hit** rate vs **Miss** rate
    - Hit: the memory address I am trying to access is inside the cache
    - Miss: the memory address I am trying to access is not in the cache
    - Multi-level: e.g.: L1 miss, L2 miss, L3 hit, retrieve from L3, etc.

# Cache Entries: #1

- Address comes in:
  - L1 hit: CPU proceeds to read from L1 cache;
  - carry on;

# Cache Entries: #2

- Address comes in:
  - L1 miss: proceed to L2;
  - L2 hit:
    entry is copied to L1, if necessary replacing the most ancient entry in L1;
  - CPU proceeds to read from L1;
  - carry on;

# Cache Entries: #3

- Address comes in:
  - L1 miss: proceed to L2;
  - L2 miss: proceed to L3;
  - L3 hit:
    entry is copied to L2, if necessary replacing most ancient entry in L2;
    entry is then copied to L1 from L2, if necessary replacing most ancient entry in L1;
  - CPU proceeds to read from L1;
  - carry on;

# Cache Entries: #4

- Address comes in:
  - L1 miss: proceed to L2;
  - L2 miss: proceed to L3;
  - L3 miss: proceed to main memory;
  - Entry is copied to L3, if necessary replacing most ancient entry in L3;
    entry is copied to L2 from L3, if necessary replacing most ancient entry in L2;
    entry is copied to L1 from L2, if necessary replacing most ancient entry in L1;
  - CPU proceeds to read from L1;
  - carry on;

# Lab 2 Part 3

P4
Lab

# Lab 2 Part 3

- Cache simulator
  - Simulate cache operations
  - Parameters: L1 speed/size, L2 speed/size, L3 speed/size
    - List of memory addresses to access, output hit and miss rates, and estimation of total time needed to actually retrieve data
  - Files: `mycache.py`, `mycache_test.py`

Technical

P4
Lab

# Lab 2 Part 3

- Simple algorithm:
  - A stack is maintained inside the **Cache**
  - If the target memory address is inside the **Cache**, it is accessed, you log the latency
  - If the target memory address is not inside the **Cache**, remove the earliest memory block from **Cache**, replace it with the target memory block, the you log the latency
  - See P3 of LS7 for detailed algorithm

Technical