

# CACHE Memory



What?

When?

Why?

Where?

How?



# Motivation

- The potential power of high-performance microprocessors depends on the speed of the memory
- Introduce wait states into the memory cycle (the READY pin) when memory is too slow
- SRAM: expensive, consume more power
- DRAM: cheaper, slow
- One solution: use small amount of SRAM (cache), along with a large amount of DRAM to achieve near zero wait states



# What is Cache memory?

- A cache is a **high-speed** memory system placed between the microprocessor and the DRAM
- Cache memory devices are usually SRAM with faster access time
- The size of cache is usually between 32K and 1M



# When did Intel start using Cache?

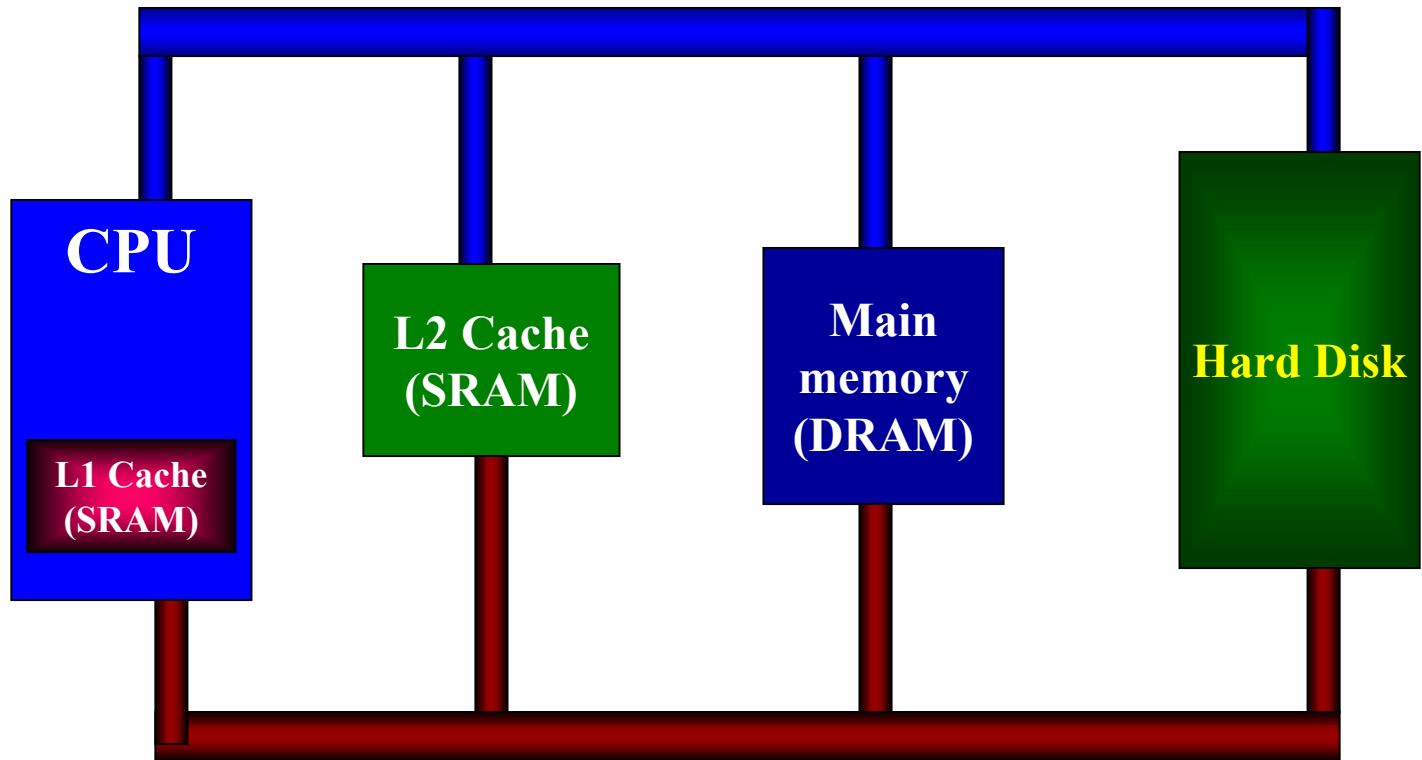
- The cache memory idea was first introduced in the IBM 360/85 computer, then PDP-11/70
- (old edition) P.237, “Intel 80486 integrated the 80387 math coprocessor in addition to 8K bytes of cache memory into a single chip”.
- (new edition) P. 691, “the 486 has 8K bytes of on-chip cache to store both code and data”
- Off-chip cache: level two cache or secondary cache



# Why use cache memory?

- Performance
- The principle of **locality of reference**:
  - programs tend to reuse data and instructions that have used recently
  - rule of thumb: a program spends 90% of its execution time in only 10% of the code

# Where is the cache memory?





# How does it work?

- On the surface,...
- As the microprocessor processes data, it looks **first** in the cache memory and if it finds the data there, it does not have to do the more time-consuming reading of data from larger memory
- Question: how can the microprocessor achieve this capability?



# A More Detailed look,...

- The **cache controller** must know what's in the cache (address)
- When CPU request info, the address requested by the CPU is compared with the address of the data kept in cache
- hit/miss: if the requested data is available in cache, then it's a \_\_\_\_\_. Otherwise, it's a \_\_\_\_\_.



# Cache Organization

- Fully associative
- Direct mapped
- Set associative
- Assuming 8-bit data bus and 16-bit address bus in the following discussion



# Fully associative – how does it work?

- Fig. 22-5 is a 128 bytes of cache. However, it uses 384 bytes of RAM. Why?
- Tag cache: hold addresses of the data ( $128 \times 16$ )
- Data cache: hold data ( $128 \times 8$ )
- When CPU request data, the 16 bits address is known, and is compared with all 128 addresses kept by the tag
- If it's a **hit**, the data is read from the data cache. Otherwise, the data is brought from main memory and a copy of it is given to the cache (both tag and data cache)



# Fully Associative—performance issues

- The hit ratio concern:

The more data, the higher hit ratio

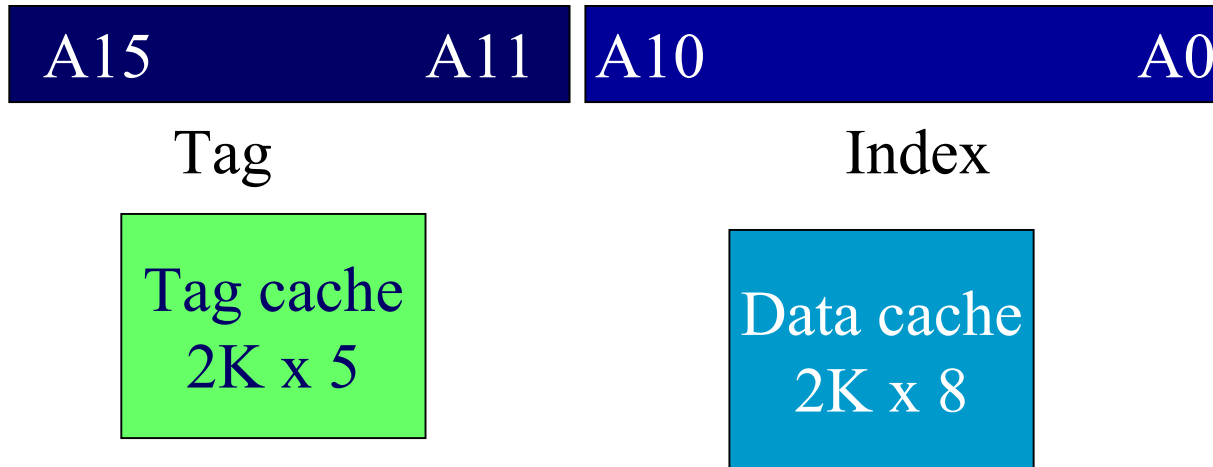
- The timing concern:

The more data, the more time needed for searching

- What type of searching is this?

# Direct-mapped Cache- how does it work?

- Fig.22-6 is a 2K-byte cache
- Divide the address into 2 parts: the index and the tag
- Index: the lower bits of the address (A0-A10), directly mapped into SRAM
- Tag: the higher bits of the address (A11-A15)





# Direct-mapped cache - concerns

- A0-A10 is directly mapped
- need to compare A11-A15 (only once)
- the size of the Tag Cache is smaller, and the number of comparison is reduced to 1
- So, what's the problem? F7A9 and 27A9 can not co-exist in the cache

# Set Associative

- Fig.22-7 is a 2-way set associative cache
- Fig 22-8 is a 4-way set associative cache
- In direct mapped cache, one tag for each index.
- In set associative, more than one tag for each index (2 tags for 2-way set associative, 4 tags for 4-way associative)
- The higher the set, the higher the hit rate, but the size of the SRAM is increased and the number of tag comparisons is also increased.

# Updating Main Memory

- Purpose: to prevent **data inconsistency between cache and main memory**
- 2 methods:
  - **Write-through**: data is written to cache and main memory at the same time
  - **Write-back** (copy back): data is written back to main memory by the cache controller **only if** cache's copy is about to be altered, through the use of the "dirty bit"

# Write-Back

- Dirty bit in the cache: '1' means the cache data is new data which exists only in cache and not in main memory
- Dirty bit is cleared when the cache data is written to main memory
- Can DMA and CPU work concurrently given that cache mechanism is provided?



# Cache Coherency

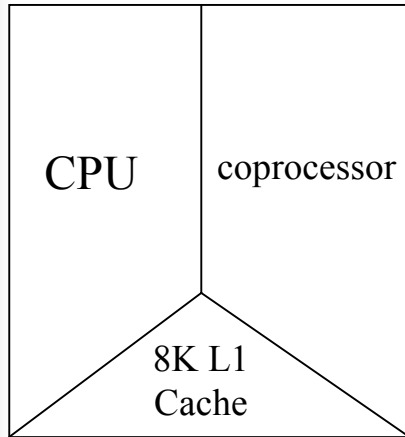
- Question: when main memory is accessed by more than one processor (DMA or multiprocessors), how can you be sure that the cache contains the most recent data?
- How can data inconsistency happen?



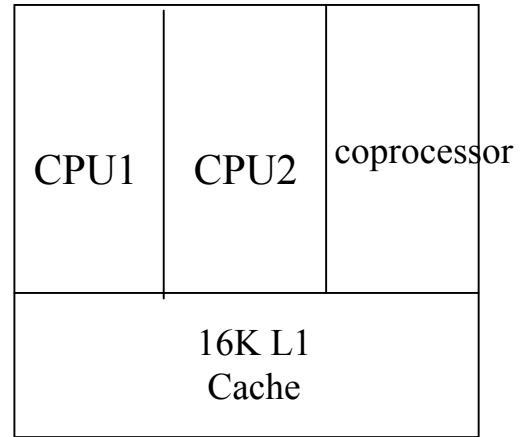
# Cache Replacement Policy

- Purpose: To make room before bringing data in from main memory
- Issues:
  - Which data to be swapped out?
  - How many data to be swapped?
- The “Which” concern:
  - LRU (least recently used) algorithm: cache controller keeps records of utilization, the least used one will be swapped out (similar to virtual memory and main memory)
  - Other policies

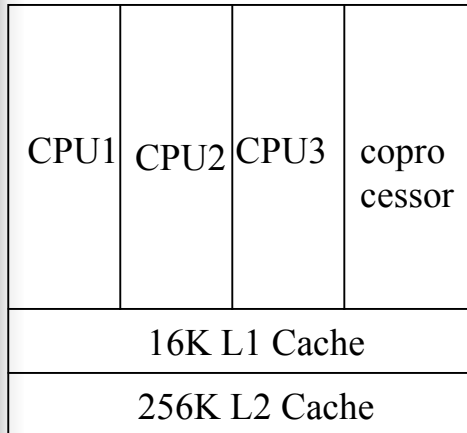
# 80486DX



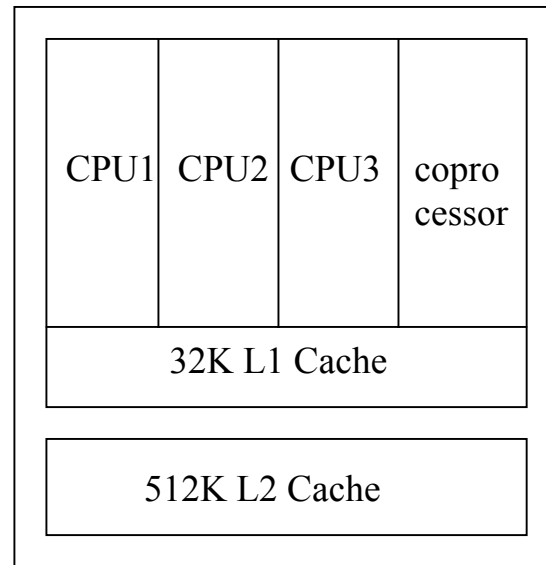
# Pentium



# Pentium Pro



# Pentium II





# Two-level Cache ( L1 and L2)

- Reference: “Computer Architecture, A Quantitative Approach” by John L. Hennessy and David A. Petterson.
- Motivation: CPU are getting faster and main memories are getting larger, but slower relative to the faster CPUs. Should I make the cache **faster** to keep pace with the speeds of CPUs, or make the cache **larger** to overcome the gap?
- One answer is: **Both**
- How? Adding another level of cache



# Multi-level cache

- The first level cache can be small enough to match the clock cycle time of the CPU
- While the second level cache can be large enough to capture many accesses that would go to main memory



# IA-32 caches

- The IA-32 processors implement 4 types of caches: the trace cache, level 1 (L1), level 2 (L2), and level 3(L3)