19MCE304- DESIGN OF EMBEDDED SYSTEMS

Co-processors and memory system mechanisms are essential components of computer architecture that enhance processing capabilities and improve the efficiency of memory operations. Here is an overview of these concepts:

Co-Processors

Definition

- **Co-Processor**: A specialized processor designed to assist the main CPU (central processing unit) by handling specific types of tasks, such as mathematical calculations, graphics rendering, or cryptographic operations.

Types of Co-Processors

1. **Math Co-Processor (Floating Point Unit - FPU)**
    - **Purpose**: Performs complex mathematical computations, particularly floating-point arithmetic.
    - **Example**: The x87 FPU in x86 architecture.
2. **Graphics Co-Processor (Graphics Processing Unit - GPU)**
    - **Purpose**: Handles rendering of images, videos, and animations.
    - **Example**: NVIDIA GeForce, AMD Radeon GPUs.
3. **Cryptographic Co-Processor**
    - **Purpose**: Performs encryption and decryption operations to enhance security.
    - **Example**: AES (Advanced Encryption Standard) engines in modern CPUs.
4. **I/O Co-Processor**
    - **Purpose**: Manages input/output operations to offload the main CPU.
    - **Example**: DMA (Direct Memory Access) controllers.
5. **Signal Processing Co-Processor (DSP)**

- **Purpose**: Specialized for handling signal processing tasks such as audio and video processing.
- **Example**: Texas Instruments DSPs.

Benefits

- **Performance Improvement**: Offloads specialized tasks from the CPU, allowing it to focus on general-purpose processing.
- **Efficiency**: Executes specific tasks more efficiently than the general-purpose CPU.
- **Parallel Processing**: Enables parallel processing, improving overall system throughput.

MEMORY SYSTEM MECHANISMS:

Modern microprocessors do more than just read and write a monolithic memory. Architectural features improve both the speed and capacity of memory systems.

Microprocessor clock rates are increasing at a faster rate than memory speeds, such that memories are falling further and further behind microprocessors every day. As a result, computer architects resort to ***caches*** to increase the average performance of the memory system.

Although memory capacity is increasing steadily, program sizes are increasing as well, and designers may not be willing to pay for all the memory demanded by an application. ***Modern microprocessor units (MMUs)*** perform address translations that provide a larger virtual memory space in a small physical memory. In this section, we review both caches and MMUs.

1. Caches:

Caches are widely used to speed up memory system performance. Many microprocessor architectures include caches as part of their definition.

The cache speeds up average memory access time when properly used. It increases the variability of memory access times accesses in the cache will be fast, while access to locations not cached will be slow. This variability in performance makes it especially important to understand how caches work so that we can better understand how to predict cache performance and factor variabilities into system design.

A cache is a small, fast memory that holds copies of some of the contents of main memory. Because the cache is fast, it provides higher-speed access for the CPU; but since it is small, not all requests can be satisfied by the cache, forcing the system to wait for the slower main memory. Caching makes sense when the CPU is using only a relatively small set of memory locations at any one time; the set of active locations is often called the *working set*.

Figure 1.19 shows how the cache support reads in the memory system. A *cache controller* mediates between the CPU and the memory system comprised of the main memory.

The cache controller sends a memory request to the cache and main memory. If the requested location is in the cache, the cache controller forwards the location's contents to the CPU and aborts the main memory request; this condition is known as a *cache hit*.

If the location is not in the cache, the controller waits for the value from main memory and forwards it to the CPU; this situation is known as a cache miss.

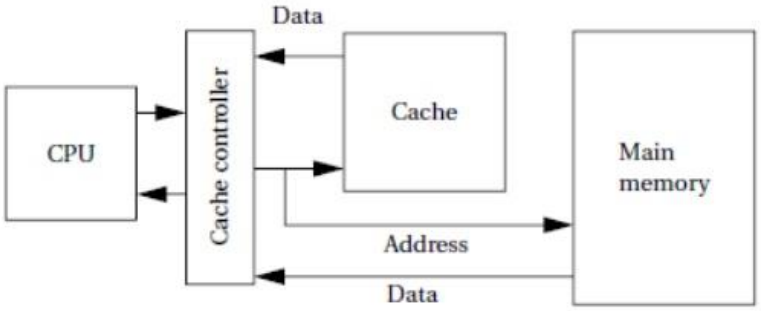

Fig 1.19: The cache in the memory system.

We can classify cache misses into several types depending on the situation that generated them:

A compulsory miss (also known as a cold miss) occurs the first time a location is used,

A capacity miss is caused by a too-large working set, and A conflict miss happens when two locations map to the same location in the cache.

Even before we consider ways to implement caches, we can write some basic formulas for memory system performance. Let h be the hit rate, the probability that a given memory location is in the cache. It follows that 1_h is the miss rate, or the probability that the location is not in the cache. Then we can compute the average memory access time as

$$t_{av} = ht_{cache} + (1 - h)t_{main}. \qquad (1.1)$$

where t_{cache} is the access time of the cache and t_{main} is the main memory access time. The memory access times are basic parameters available from the memory manufacturer.

The hit rate depends on the program being executed and the cache organization, and is typically measured using simulators. The best-case memory access time (ignoring cache controller overhead) is t_{cache}, while the worst-case access time is t_{main}. Given that t_{main} is typically 50–60 ns for DRAM, while t_{cache} is at most a few nanoseconds, the spread between worst-case and best-case memory delays is substantial.

where t_{cache} is the access time of the cache and t_{main} is the main memory access time. The memory access times are basic parameters available from the memory manufacturer. The hit rate

depends on the program being executed and the cache organization, and is typically measured using simulators.

The best-case memory access time (ignoring cache controller overhead) is t_{cache}, while the worst-case access time is t_{main}. Given that t_{main} is typically 50–60 ns for DRAM,while t_{cache} is at most a few nanoseconds, the spread between worst-case and best-case memory delays is substantial.

Modern CPUs may use multiple levels of cache as shown in Figure 1.20. The ***first-level cache*** (commonly known as ***L1 cache***) is closest to the CPU, the ***second-level cache (L2 cache)*** feeds the first-level cache, and so on.

The second-level cache is much larger but is also slower. If $h1$ is the first-level hit rate and $h2$ is the rate at which access hit the second-level cache but not the first-level cache, then the average access time for a two-level cache system is

$$t_{av} = h_1 t_{L1} + h_2 t_{L2} + (1 - h_1 - h_2) t_{main}. \qquad (1.2)$$

As the program's working set changes, we expect locations to be removed from the cache to make way for new locations. When set-associative caches are used, we have to think about what happens when we throw out a value from the cache to make room for a new value.

We do not have this problem in direct-mapped caches because every location maps onto a unique block, but in a set-associative cache we must decide which set will have its block thrown out to make way for the new block.

One possible replacement policy is least recently used (LRU), that is, throw out the block that has been used farthest in the past. We can add relatively small amounts of hardware to the cache to keep track of the time since the last access for each block. Another policy is random replacement, which requires even less hardware to implement.

The simplest way to implement a cache is a ***direct-mapped cache***, as shown in Figure 1.20. The cache consists of cache ***blocks***, each of which includes a tag to show which memory location is represented by this block, a data field holding the contents of that memory, and a valid tag to show whether the contents of this cache block are valid. An address is divided into three sections.

The index is used to select which cache block to check. The tag is compared against the tag value in the block selected by the index. If the address tag matches the tag value in the block, that block includes the desired memory location.

If the length of the data field is longer than the minimum addressable unit, then the lowest bits of the address are used as an offset to select the required value from the data field. Given the structure of the cache, there is only one block that must be checked to see whether a location is in the cache—the index uniquely determines that block. If the access is a hit, the data value is read from the cache.

Writes are slightly more complicated than reads because we have to update main memory as well as

the cache. There are several methods by which we can do this. The simplest scheme is known as ***write-through***—every write changes both the cache and the corresponding main memory location (usually through a write buffer).

This scheme ensures that the cache and main memory are consistent, but may generate some additional main memory traffic. We can reduce the number of times we write to main memory by using a ***write-back*** policy: If we write only when we remove a location from the cache, we eliminate the writes when a location is written several times before it is removed from the cache.

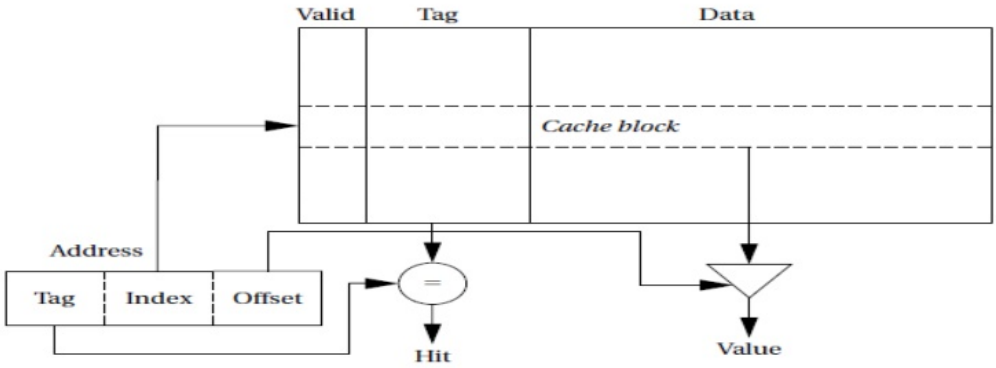

Fig 1.20: A direct-mapped cache.

The direct-mapped cache is both fast and relatively low cost, but it does have limits in its caching power due to its simple scheme for mapping the cache onto main memory. Consider a direct-mapped cache with four blocks, in which locations 0, 1, 2, and 3 all map to different blocks. But locations 4, 8, 12…all map to the same block as location 0; locations 1, 5, 9, 13…all map to a single block; and so on. If two popular locations in a program happen to map onto the same block, we will not gain the full benefits of the cache. As seen in Section 5.6, this can create program performance problems.

The limitations of the direct-mapped cache can be reduced by going to the ***set-associative*** cache structure shown in Figure 1.21.A set-associative cache is characterized by the number of ***banks*** or ***ways*** it uses, giving an *n*-way set-associative cache.

A set is formed by all the blocks (one for each bank) that share the same index. Each set is implemented with a direct-mapped cache. A cache request is broadcast to all banks simultaneously. If any of the sets has the location, the cache reports a hit.

Although memory locations map onto blocks using the same function, there are *n* separate blocks for each set of locations. Therefore, we can simultaneously cache several locations that happen to map onto the same cache block. The set associative cache structure incurs a little extra overhead and is slightly slower than a direct-mapped cache, but the higher hit rates that it can provide often compensate.

The set-associative cache generally provides higher hit rates than the direct mapped cache because conflicts between a small number of locations can be resolved within the cache. The

set-associative cache is somewhat slower, so the CPU designer has to be careful that it doesn't slow down the CPU's cycle time too much. A more important problem with set-associative caches for embedded program.

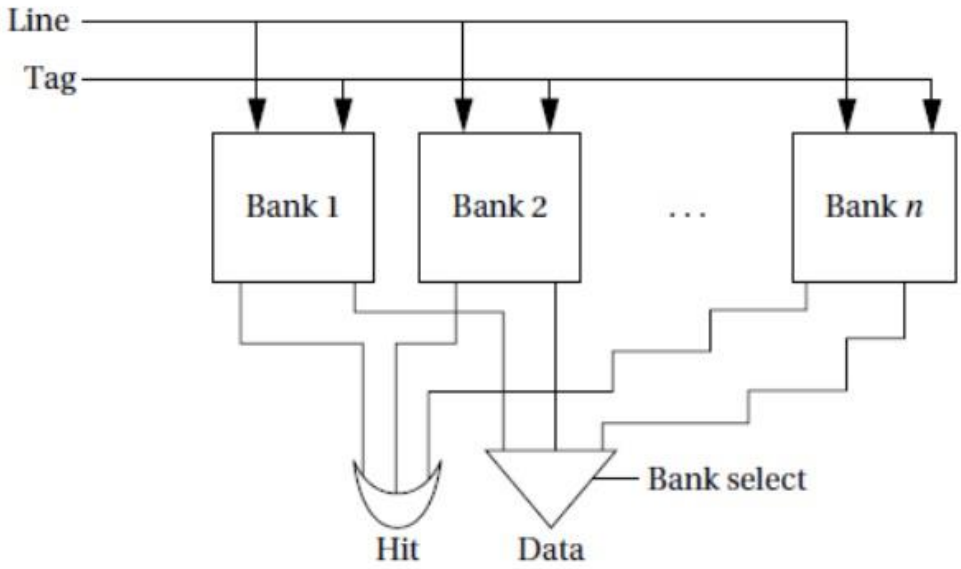

Fig 1.21: A set-associative cache.

Design is predictability. Because the time penalty for a cache miss is so severe, we often want to make sure that critical segments of our programs have good behavior in the cache. It is relatively easy to determine when two memory locations will conflict in a direct-mapped cache.

Conflicts in a set-associative cache are more subtle, and so the behavior of a set-associative cache is more difficult to analyze for both humans and programs.