EECS151/251A Discussion 10

Christopher Yarp
Apr. 12, 2019
Plan for Today

• Practice Problem
• Cache Review
• Questions
Practice Problem

• How would you create a memory with 32x8 memory with 2 write ports and 2 read ports given ...
  • 32x8 memory blocks with single read ports and single write ports
  • 32x1 memory blocks with 2 write ports and 1 read port

• Optimize for area

• Assume that the area of the 32x1 memory is > 2/7 the cost of a 32x8 memory
• We will assume that, in the case when the 2 write ports attempt to write to the same address in the same cycle, the result is undefined
• We will assume that, in the case when an address is simultaneously written to and read from, the read value is undefined.
Adding a Second Write Port

Diagram showing the connections between different components, including a 32x1, two 32x8 blocks, and data ports. The circuit includes signals for addresses (AddrR, AddrW0, AddrW1), data (DataW0, DataW1), and write enable (WE0, WE1, WE) for each port.
Adding a Second Read Port

Diagram showing two 32x8 memory chips connected to form a larger memory module with additional read ports.
Same Strategy with Our Dual Write Port Mem

[Diagram of memory configuration with labeled signals and connections]
In general, there is a tradeoff between the speed of memory and the size or cost of it.

- SRAM is fast but expensive and not especially dense
- 1T DRAM is dense, less expensive, but slower
- Spinning hard disks are dense, inexpensive, but very slow

We want fast, high capacity, and low cost memory. But we can't have all of those qualities.

Solution: The memory hierarchy

- Include multiple types of memory in our system. Small quantities of fast memory and large quantities of slower memory.
- Move values between them when needed
Using the Memory Hierarchy

• The memory hierarchy takes advantage of how applications access memory: there tends to some locality of access
  • **Spatial Locality**: Nearby addresses are likely to be accessed soon
    • Ex. Accessing elements sequentially in a vector, accessing variables on the stack, ...
  • **Temporal Locality**: Addresses that have been accessed recently are likely to be accessed again soon
    • Ex. Accessing loop variables, results computed/reused between loop iterations, ...

• When we access a memory address, we move it from our large/slow memory into our small/fast memory.
  • We also move data in nearby addresses along with it.

• We typically call this smaller, faster memory a *cache*. 
Cache Operation

• Most caches transfer blocks of data between memory and adjacent memories in the hierarchy. These blocks are sometimes known as *cache lines*.

• Moving whole cache lines allows us to exploit spatial locality.

• Depending on the type of cache, there are conditions on where cache lines can be stored in the cache

• Common Types:
  • Direct Mapped Cache: Each cache line can only be written into a single location in the cache.
  • Fully Associative Cache: There are no restrictions on where a cache line can be written to in the cache
  • Set Associative: There is a set of locations in the cache that a particular cache line can be written into
Direct Mapped Caches

Example:
- DRAM Size = 4 GiB
- Cache Line Size = 16 Bytes
- Cache Size = 64 Bytes (4 Lines)
Direct Mapped Caches

• Each cache line can only be written to a single location in the cache.
  • We can’t have 2 lines in the cache at the same time if they share the same index.
  • If we load one, we need to evict the other, even if there are open cache entries
  • This is called a conflict miss

• Pros
  • Simple HW implementation
    • Finding if a line is in the cache simply involves calculating the index then checking the tab and valid entries at that index

• Cons:
  • Conflict misses may force cache lines to be evicted even if they will be acceded again soon
Fully Associative Cache

Example:
- DRAM Size = 4 GiB
- Cache Line Size = 16 Bytes
- Cache Size = 64 Bytes (4 Lines)
Fully Associative Cache

• There is no restriction of where a cache line can be stored in the cache.
  • Conflict misses are impossible
• However, if the cache is full and we want to fetch a new value into the cache, a line will need to be evicted from the cache to make room
  • This is called a capacity miss
• The decision on what to replace is called the cache replacement policy
  • One example is evicting the least recently used line (LRU)

• Pros:
  • Avoids conflict misses

• Cons:
  • Requires much more hardware:
    • Finding a line in the cache requires comparing the tag and valid bit of each cache entry
    • Capacity misses require running the cache replacement policy which may require additional state
Set Associative Cache

Example:
- DRAM Size = 4 GiB
- Cache Line Size = 16 Bytes
- Cache Size = 128 Bytes (8 Lines)
- 2 Way Set Associative:
  - 4 Indexes
Set Associative Cache

• A cross between the Direct Mapped and the Fully Associative Cache
• Indexes now refer to sets of locations in the cache
• Cache lines are restricted to being stored in the set that corresponds to their index (similar to direct mapped)
• Within the set, there cache entries are fully associative, allowing flexibility in where a cache line is stored
  • Replacement policies apply to entries within a set
• Set associative caches are defined as being “N way set associative”.
  • N defines the number of entries per set