M5 Memory Systems

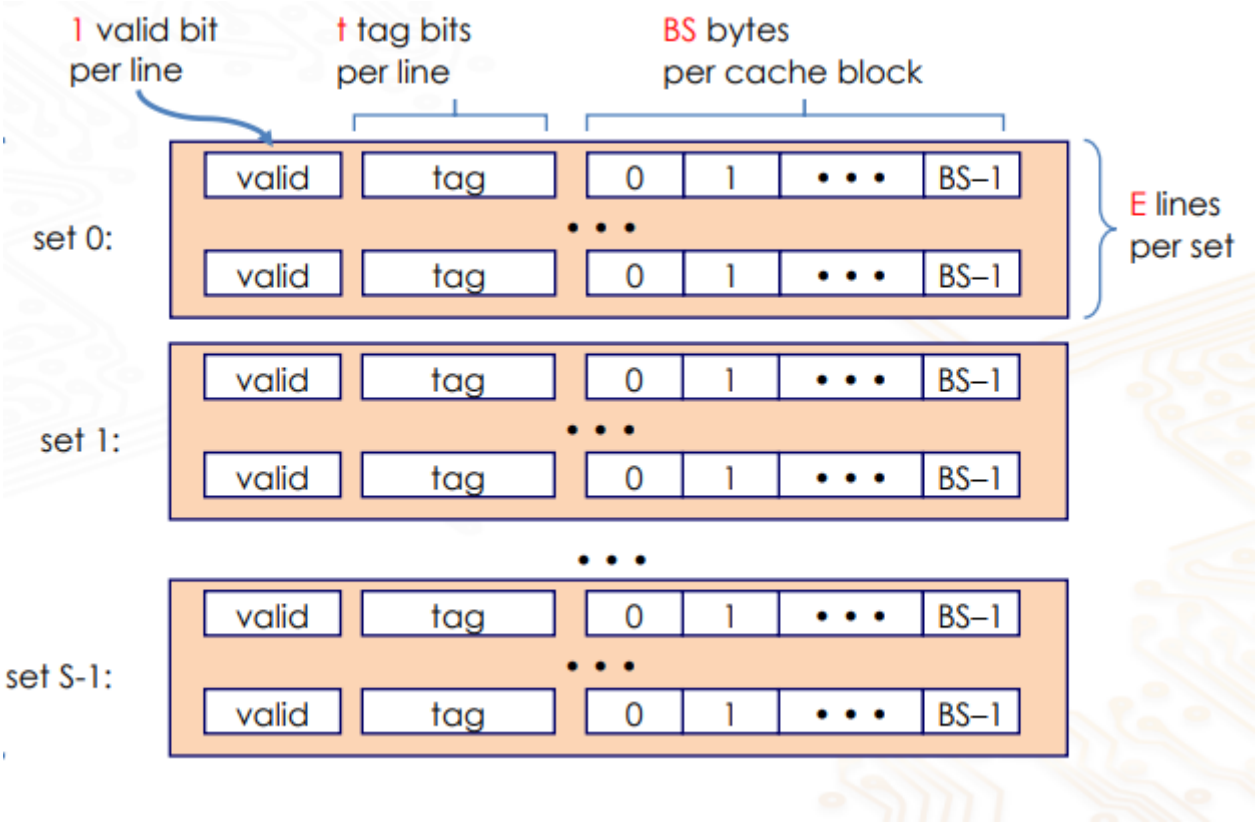
1. Cache Design

1.1 Cache addressing

- Main memory and cache are divided into blocks of the same size, each block maps to a location in cache determined by the index bits in the **main memory address**
  `[tag][index][offset]`
  Cache lines are divided into different fields `[valid][tag][cache block]`
  - valid bit tells status of the data in the cache (i.e. indicates if data is valid)

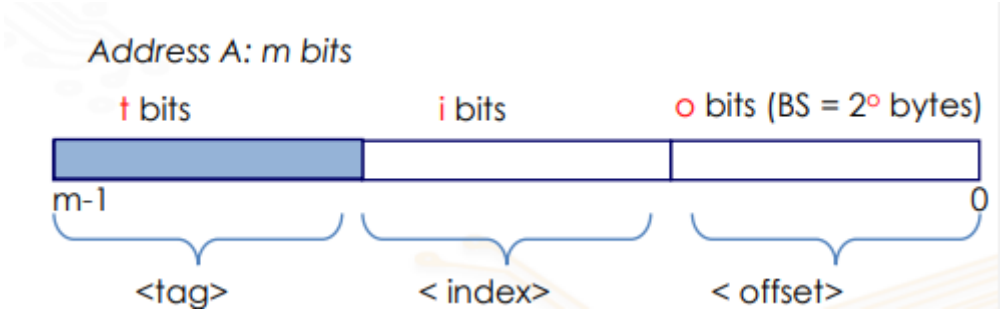
1.2 Organisation schemes

- The cache is an array of sets
  - Each set carries an array of cache lines
    - Each cache line holds a block of data



- Cache size is given by $(1 \text{ bit} + t \text{ bits} + \text{BS bytes}) \times E \times S$
  - where S is the number of sets

- Cache is accessed using m bits



  - `tag` is the same as `tag` in the cache, `index` is the set which the target cache line resides, and `offset` is the location of the target word within the set

Portion	Length	Purpose
Offset	o=$\log_2$(block size)	Select word within block/line
Index	i=$\log_2$(number of sets)	Select the set
Tag	t=(address size) - o - i	ID block/line within set

- Number of bits in `offset` is given by number of bytes in 1 block (if byte addressable)
- Number of bits in `set` is given by size of cache / size of set
- Number of bits in `tag` is the remaining bits
- **Other special stuff**
  - Number of bits in a cache line = "#tag + #valid + #data" (data bits = block size)
  - Cache Size = "#cachelines x #bits in cacheline"
  - Cache Overhead = Cache Size / Data size in cache (excludes tag and valid bits)

Mapping of main memory blocks to cache lines

- Decided by mapping schemes which determines where data is placed when it is originally copied into cache
- Provides a method for the CPU to find previously copied data when searching cache
- **Direct Mapped Cache**
  - Consists of an array of fixed size frames called cache lines / blocks which holds a block (consecutive bytes) of main memory
  - Each memory block is mapped to a specific cache line, set of memory blocks with the same cache-index gets mapped to the same cache line
  - Since number of lines in a set is 1, size of cache will be roughly $\text{BS bytes} \times S$
  - **Accessing**
    - Index bits used to determine the set of interest (seen in the orange table)
    - Set will only contain one cache line, so check valid bit of the set (cache line)
    - Check tag bits of cache line to see if they match

        - Valid bit must be set **and** tag bits must match to generate a cache hit
    - Offset is used to determine starting byte location in cache line
  - **Disadvantages**
    - When two blocks are mapped to the same location, continuous cache misses will be generated if sequence of operation requires data from both blocks alternatively (e.g. ABABABA...)
- **Fully associative cache**
  - Allows main memory block to be placed anywhere in the cache and the only way to find a block is to search the whole cache
  - Cache needs to be built from associative memory to search in parallel
  - Only one set is present (this means main memory needs to be partitioned as `[tag][offset]`) and requires a longer `tag`
  - **Accessing**
    - Searches the whole cache to find matching tag
      - Cache miss if no match
    - If cache hit, then block offset selects starting byte
- **Set associative cache**
  - Number of sets = number of blocks / set associativity (2 if 2-way set associative)
  - Combination of both direct mapped cache and fully associative cache
  - Each set contains more cache lines
  - **Accessing**
    - Index bit determines which set target cache line is mapped to
    - Searches the set for the tag bits
    - If tag bit match, check valid bit
      - **Tag bits checked first instead of valid bit**
    - If cache hit, then block offset selects starting byte

2. Cache Replacement and Write Policies

2.1 Replcacement Policy

- Victimize, evict, replace, cast out
- Policies include FIFO, LRU, NMRU (not most recently used), Pseudo-Random
- Pick victim within set where a = associativity
  - if $a \leq 2$, LRU is cheap and easy

- harder if $a \geq 2$

2.2 Write Policies

- For every data block in the cache, there must be at least one same block in the main memory
  - All copies must be changed when there is a writeback to a cache block

Write-through

- Every write propagates directly through memory hierarchy
  - L1 cache, L2 cache, memory, disk
- Disadvantage
  - High bandwidth requirement
  - Memory becomes slow when size increases
- Better to update L1 and L2 only, and use write-back policy beyond L2

Write-back

- Maintain state of each line in a cache
  - Invalid - not present in cache
  - Clean - present in cache but not modified
  - Dirty - present in cache and modified
- Store state in tag array next to address tag (mark dirty bit)
- Check dirty bit on eviction of cache line
  - If set write back modified line to next level
- Disadvantages
  - More stale (unupdated) copies in lower-level of hierarchy
  - Must always check higher level for dirty copies before accessing a copy in lower level
  - Causes cache coherence in multiprocessors
    - e.g. Core 1 wants to access same data that has been updated by Core 0, but data has not been evicted from Core 0's cache, so copy in main memory is not updated and Core 1 cannot access Core 0's cache
  - I/O devices that use DMA can cause problems
    - Coherent I/O
    - Must check cache for dirty copies before reading main memory

3. Instruction cache vs Data cache

Data cache - Stores data only

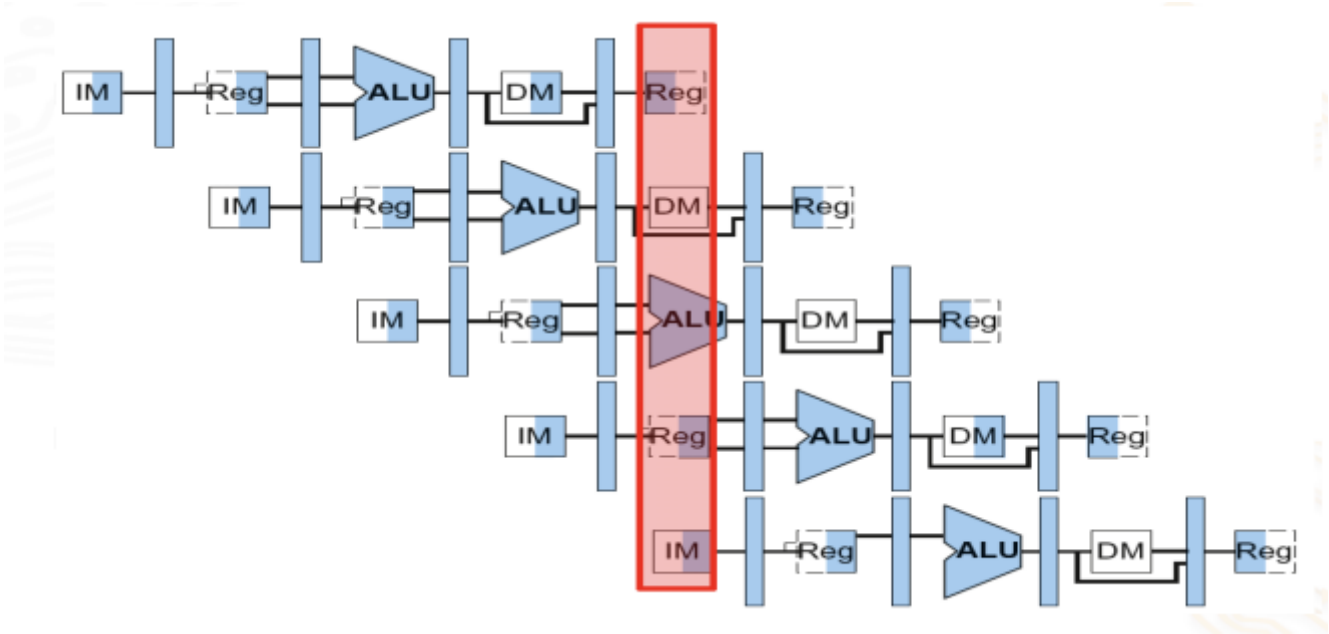
- When data is requested it looks in data cache
- Retrieves information and continues execution if found, otherwise retrieve from main memory and place in data cache

Instruction cache - Stores intructions only

- Instructions occur sequentially in most programs with occasional branching
- Follows principle of locality → high hit rate

When both are unified

- Cache only has one port for data and instruction access which results in conflict



4. Cache Performance

- Cache hit: improves performance
- Cache miss: fetch from next level caches
  - Performance impact due to cache misses: miss penalty; leads to reduction of performance

Miss Penalty

- Detect miss: 1 cycle
- Find cache line to replace: 1 or more cycles
  - have to write back if dirty
- Request cache lines from next levels: several cycles
- Transfer cache line from next levels: several cycles
  - (block size) / (bus width)
- Fill cache line with data, update tag array: 1 or more cycles
- Resume execution
- **Min 6 cycles, to 100 cycles**

4.1 Cache Miss

- Determined by **temporal locality**, **spatial locality**, **cache organisation (block size, associativity, number of sets)**
- Reasons
  - Compulsory miss
    - First reference to a given block of memory
  - Capacity miss
    - Working set exceeds cache capacity
    - Useful blocks displaced
  - Conflict miss
    - Placement restrictions (due to not fully associative) cause useful blocks to be displaced

Rate Effects

- Number of blocks
  - Bigger is better $\rightarrow$ fewer conflicts
- Associativity
  - Higher associativity reduces conflicts however diminishing returns beyond 8-way set-associativity
- Block Size
  - Larger blocks exploit spatial locality
- Tradeoffs
  - Larger blocks reduce compulsory misses but increase miss penalty because more data needs to be moved to the cache
  - Larger blocks increase conflict misses
  - Associativity reduces conflict misses
  - Associativity increases access time (since data can be mapped more randomly)

Performance

- Execution time $= \mathrm{IC} \times \mathrm{CPI} \times \text{Cycle time}$
  - If cache hit costs are included, $\mathrm{CPI} = \mathrm{CPI}_{\text{ideal}} + \text{Memory-stall cycles}$
  - Memory-stall cycles = memory accesses / program $\times$ miss rate $\times$ miss penalty

- A processor with a CPI_{ideal} of 2, a 100 cycle miss penalty,
- 36% load/store instr's, and 2% IM and 4% DM miss rates

Memory-stall cycles = Memory stall cycles in IM + memory stall cycles in DM

Memory stall cycles in IM =2% (miss rate of IM)× 100 (Miss penalty) =2
(All instructions has to be fetched from IM)

Memory stall cycles in DM =
36% (LW/SW) × 4% (miss rate)× 100 (miss penalty) = 1.44

So CPI_{stalls} = 2 + 2+ 1.44 = **5.44** (more than twice the CPI_{ideal} !)

- What if the CPI_{ideal} is reduced to 1?

  For ideal CPI = 1, then CPI_{stall} = 4.44 and the amount of execution time spent on memory stalls would have risen from 3.44/5.44 = 63% to 3.44/4.44 = 77%

Cache levels

- **First level cache (L1 cache)**
- **Second level cache (L2 cache)**
  - usually holds both instructions and data
- **Last level cache (LLC)**

- For our example, CPI_{ideal} of 2,
  - 100 cycle miss penalty (to main memory)
  - 25 cycle miss penalty (to Unified L2 cache)
  - 36% load/stores,
  - 2% miss-rate for L1 I-Mem and 4% miss-rate for L1 D-Mem
  - 0.5% Unified L2 cache miss rate.

Note: every instruction will access the instruction memory for instruction fetch (100%)
only load/store instructions will access the data memory (36%)

$$
\begin{aligned}
CPI &= CPI_{ideal} + CPI_{L1Inst\_miss} + CPI_{L1data\_miss} + CPI_{L2Inst\_miss} + CPI_{L2data\_miss} \\
&= CPI_{ideal} + CPI_{L2\_hit\_inst} + CPI_{L2\_hit\_data} + CPI_{MM\_inst} + CPI_{MM\_data} \\
&= 2 + 2\%*25 + 36\%*4\%*25 + 2\%*0.5\%*100 + 36\%*4\%*0.5\%*100 \\
&= 2+0.5+0.36+0.01+0.0072 \\
&= 2.8772 \text{ cycles/inst. (as compared to 5.44 (no L2cache))}
\end{aligned}
$$

Average Memory Access Time (AMAT)

- Average time required to access memory considering both hits and misses
  - Time for hit + Miss Rate × Miss Penalty

- What is the AMAT for a processor with clock frequency 2 Ghz and with miss penalty of 25ns, a miss rate of 2% misses per instruction and a cache access time of 1 clock cycle?

  Miss penalty in clock cycles= access time/clk period

  = 25ns/(0.5 ns/ cycle) =50 clock cycles.

  Time for hit= cache access time =1 clock cycle

  Time for miss= miss rate * miss penalty=2% * 50=1

  AMAT =t_{avg} = t_{hit} + miss-rate * miss penalty= 1 + 0.02x50 = 2