Caches – basic idea

- Small, fast memory
- Stores frequently-accessed *blocks* of memory.
- When it fills up, discard some blocks and replace them with others.
- Works well if we reuse data blocks
  - Examples:
    - Incrementing a variable
    - Loops
    - Function calls

Why do caches work

- Locality principles
  - Temporal locality
    - Likely to reference same location several times
      - Variables are reused in program
      - Loops, function calls, etc.
  - Spacial locality
    - Reference is likely to be near another recent reference
    - Matrices, arrays
    - Stack accesses

Cache performance example

- Problem (let's assume single cycle CPU)
  - 500 MHz CPU → cycle time = 2 ns
  - Instructions: arithmetic 50%, load/store 30%, branch 20%.
  - Cache: hit rate: 95%, miss penalty: 60 ns (or 30 cycles), hit time: 2 ns (or 1 cycle)
- MIPS CPI w/o cache for load/store:
  - 0.5 * 1 + 0.2 * 1 + 0.3 * 30 = 9.7
- MIPS CPI with cache for load/store:
  - 0.5 * 1 + 0.2 * 1 + 0.3 * (.95*1 + 0.05*30) = 1.435

Cache types

- Direct-mapped
  - Memory location maps to single specific cache line (block)
- Set-associative
  - Memory location maps to a *set* containing several blocks.
  - Sets can have 2,4,8,etc. blocks. Blocks/set = associativity
  - Why? Resolves conflicts in direct-mapped caches.
- Fully-associative
  - Cache only has one set. All memory locations map to this set.
  - This one set has all the blocks, and a given location could be in any of these blocks
  - No conflict misses, but costly (why?). Only used in very small caches.

Direct-mapped cache example

- 4 KB cache, each block is 32 bytes
- How many blocks?
- How long is the index to select a block?
- How long is the offset (displacement) to select a byte in block?
- How many bits left over if we assume 32-bit address? These bits are tag bits

Direct-mapped cache example

- 4 KB cache, each block is 32 bytes
  - 4 KB = 2^{12}, 32 = 2^5
- How many blocks?
  - 2^{12} bytes / 2^5 bytes in block = 2^7 = 128 blocks
- How long is the index to select a block?
  - $\log_2 128$ = 7 bits
- How long is the offset (displacement) to select a byte in block?
  - 5 bits
- How many bits left over if we assume 32-bit address? These bits are tag bits
  - 32 – 7 – 5 = 20 bits

Example continued

- Address and cache:

Cache size

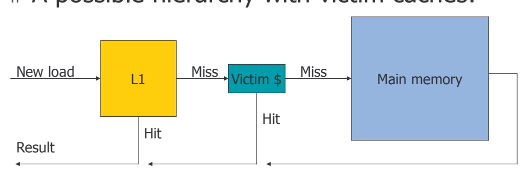
- 4 KB visible size
- Let's look at total space and overhead:
  - Each block contains:
    - 1 valid bit
    - 20-bit tag
    - 32 bytes of data = 256 bits
    - Total block (line) size: 1+20+256 = 277 bits
  - Total cache size in hardware, including overhead storage:
    - 277 bits * 128 blocks = 35456 bits = 4432 bytes = 4.32 Kb
    - Overhead: 0.32 Kb (336 bytes) for valid bits and tags

Cache access examples…

- Consider a direct-mapped cache with 8 blocks and 2-byte block. Total size = 8 * 2 = 16 bytes
- Address: 1 bit for offset/displacement, 3 bits for index, rest for tag
- Consider a stream of reads to these bytes:
  - These are byte addresses:
  - 3, 13, 1, 0, 5, 1, 4, 32, 33, 1
  - Corresponding block addresses ((byteaddr/2)%8):
  - 1, 6, 0, 0, 2, 0, 2, 0 (16%8), 0, 0.
  - Tags: 2 for 32, 33, 0 for all others ((byteaddr/2)/8).
- Let's look at what this looks like. How many misses?
- What if we increase associativity to 2? Will have 4 sets, 2 blocks in each set, still 2 bytes in each block. Total size still 16 bytes. How does behavior change?…
- What if we add a victim cache?

Victim cache

- Reduce conflict misses
  - Especially in direct-mapped caches
- Very small, fully-associative
- A possible hierarchy with victim caches:



Review of Victim Cache Operation

- Hit in L1 – done; nothing else needed
- Miss in L1 for block *b*, hit in victim cache at location *v:*
  - swap contents of *b* and *v*
- Miss in L1, miss in victim cache:
  - load missing item from next level and put in L1
  - put entry replaced in L1 in victim cache
  - if victim cache is full, evict one of its entries