Introduction to computer architecture

April 8th

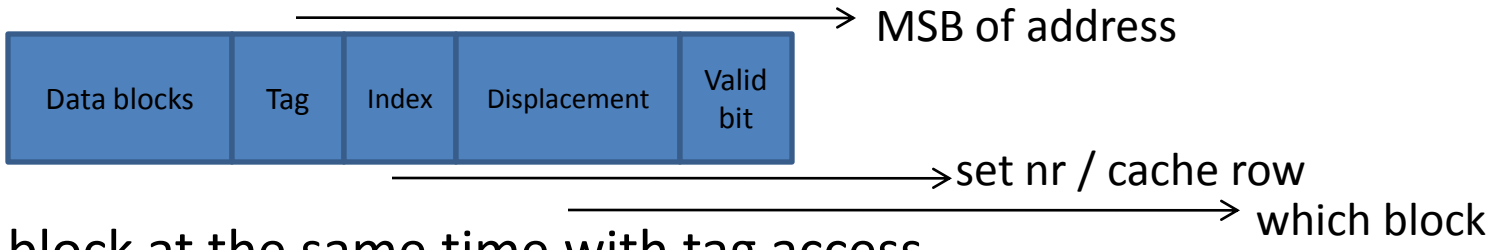
Write strategy

- Reads dominate cache accesses
  - Writes abt 7% of memory traffic, 21% data cache traffic

- Goal:

Make common case fast – optimize for reads

But don't neglect writes (Amdahl's Law)!



- Read block at the same time with tag access
  - If miss ignore value read
  - Not in embedded processors!
- But writes *cannot safely begin* before we know it's a hit!!

Write strategy

- Write-through: all writes go both to cache and to main memory
  - Advantages:
    - Simple data coherence: multicore, multiprocessors, I/O DMA, etc
    - Easy to implement
  - Disadvantages:
    - Lots of traffic
    - Slows down writes

- Write-back: write only to block in cache (and set update bit for cache slot). Modified cache blocks written back to memory only when replaced.
  - Read miss may result in write to main memory!
  - Multiple writes within 1 block require single write to main memory
  - Advantages:
    - Fast
    - Less power
    - Less use of memory bandwidth
  - Disadvantages:
    - Other caches get out of synch
    - I/O must access main memory through cache

Write miss

- On write miss:
  - Write allocate – similar to read miss, i.e. first allocate block in cache
  - No-write allocate – write directly to main memory, don’t modify cache
- Usually
  - write-back caches use write allocate
  - write-through caches use no-write

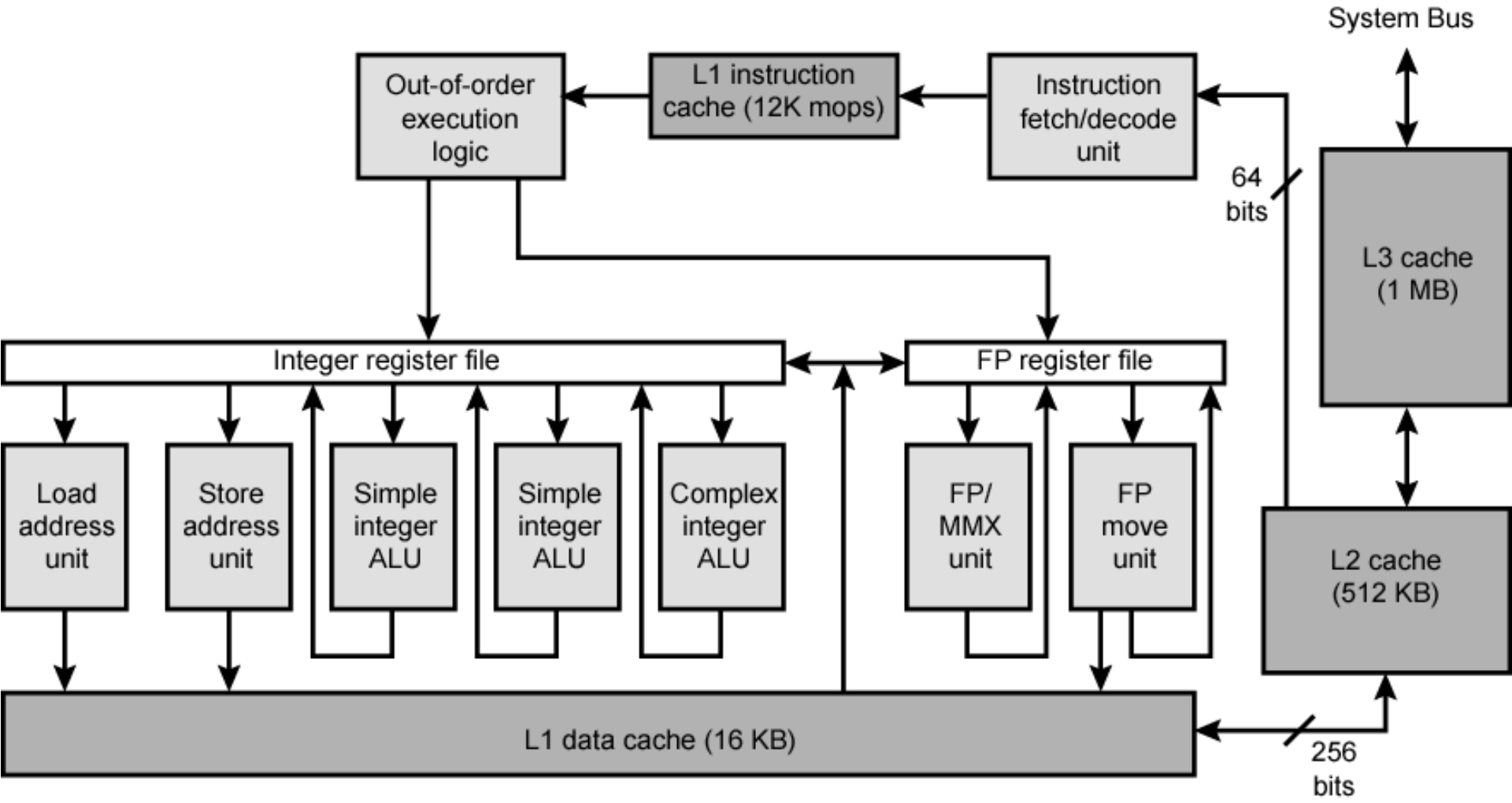
Example: Pentium 4 Cache

- 80386 – no on chip cache
- 80486 – 8k using 16 byte lines and four way set associative organization
- Pentium (all versions) – two on chip L1 caches
  - Data & instructions
- Pentium III – L3 cache added off chip
- Pentium 4
  - L1 caches
    - 8k bytes
    - 64 byte lines
    - four way set associative
  - L2 cache
    - Feeding both L1 caches
    - 256k
    - 128 byte lines
    - 8 way set associative
  - L3 cache on chip

Intel Cache Evolution

Problem	Solution	Processor on which feature first appears
External memory slower than the system bus.	Add external cache using faster memory technology.	386
Increased processor speed results in external bus becoming a bottleneck for cache access.	Move external cache on-chip, operating at the same speed as the processor.	486
Internal cache is rather small, due to limited space on chip	Add external L2 cache using faster technology than main memory	486
Contention occurs when both the Instruction Prefetcher and the Execution Unit simultaneously require access to the cache. In that case, the Prefetcher is stalled while the Execution Unit's data access takes place.	Create separate data and instruction caches.	Pentium
Increased processor speed results in external bus becoming a bottleneck for L2 cache access.	Create separate back-side bus that runs at higher speed than the main (front-side) external bus. The BSB is dedicated to the L2 cache.	Pentium Pro
	Move L2 cache on to the processor chip.	Pentium II
Some applications deal with massive databases and must have rapid access to large amounts of data. The on-chip caches are too small.	Add external L3 cache.	Pentium III
	Move L3 cache on-chip.	Pentium 4

Pentium 4 Block Diagram



Pentium 4 Core Processor

- Fetch/Decode Unit
  - Fetches instructions from L2 cache
  - Decode into micro-ops
  - Store micro-ops in L1 cache
- Out of order execution logic
  - Schedules micro-ops
  - Based on data dependence and resources
  - May speculatively execute
- Execution units
  - Execute micro-ops
  - Data from L1 cache
  - Results in registers
- Memory subsystem
  - L2 cache and systems bus

Pentium 4 Design Reasoning

- Decodes instructions into RISC like micro-ops before L1 cache
- Micro-ops fixed length
  - Superscalar pipelining and scheduling
- Pentium instructions long & complex
- Performance improved by separating decoding from scheduling & pipelining
- Data cache is write back
  - Can be configured to write through
- L1 cache controlled by 2 bits in register
  - CD = cache disable
  - NW = not write through
  - 2 instructions to invalidate (flush) cache and write back then invalidate
- L2 and L3 8-way set-associative
  - Line size 128 bytes

Improving cache performance

- Reduce miss penalty
- Reduce miss rate
- Reduce miss penalty or rate via parallelism
- Reduce time to hit in the cache

Improving cache performance

- Reduce miss penalty
  - E.g. multilevel caches, etc
- Reduce miss rate
- Reduce miss penalty or rate via parallelism
- Reduce time to hit in the cache

Improving cache performance

- Reduce miss penalty
  - E.g. multilevel caches, etc
- Reduce miss rate
  - E.g. larger block size, larger cache size, higher associativity, way prediction, compiler optimizations – reorder accesses, etc
- Reduce miss penalty or rate via parallelism
- Reduce time to hit in the cache

Improving cache performance

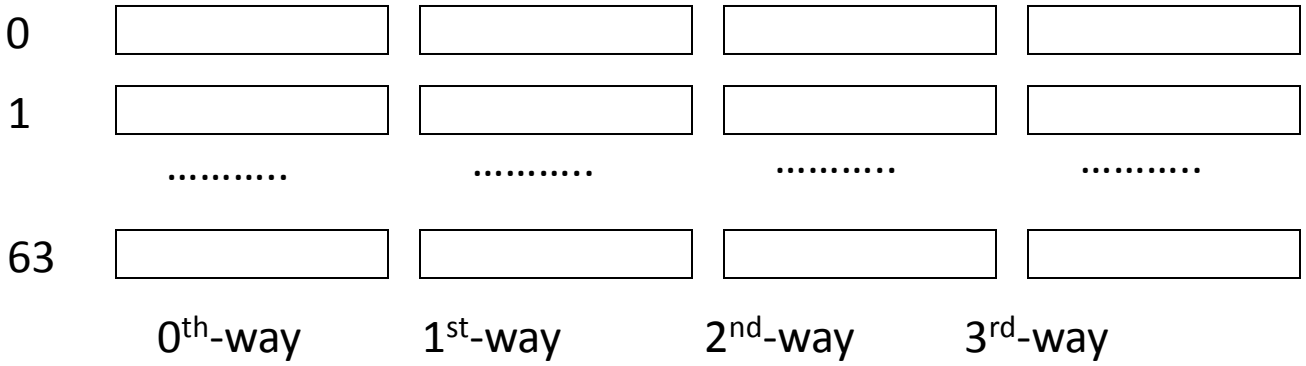
- Reduce miss penalty
  - E.g. multilevel caches, etc
- Reduce miss rate
  - E.g. larger block size, larger cache size, higher associativity, way prediction, compiler optimizations – reorder accesses, etc
- Reduce miss penalty or rate via parallelism
- Reduce time to hit in the cache
  - Avoid address translation, pipelined cache access, small/simple caches, etc

Problem 1

- Given a computer with:
  - DRAM main memory:
    - Size 512 MB
    - T_{mm} = 200ns (including time to handle cache miss)
  - Cache:
    - Size 16 KB
    - 64B line size
    - 4-way set associative
    - Replacement policy: FIFO
    - T_c = 10ns
    - Hit rate h = 95%
- Sketch the cache structure specifying the number of sets and the size of each set
- Compute the average memory access time

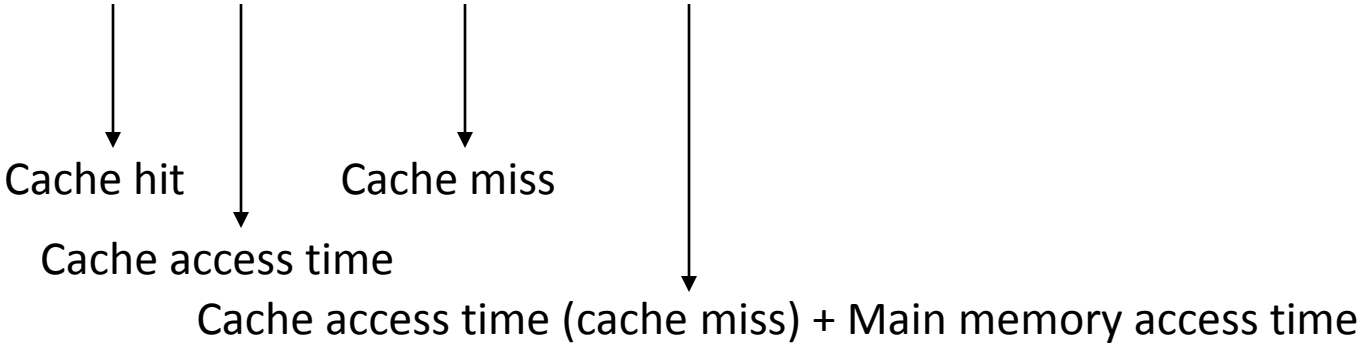
Solution – cache structure

- 16 KB = 2^{14} B
- 64 B = 2^{6} B line size
- 4-way associative memory -> 4 lines -> 2^{6} X 2^{2} = 2^{8} = 256 B per set
- Nr of sets: 2^{14} / 2^{8} = 2^{6} sets



Solution – access time

- $T_{avg} = h \times T_c + (1-h) \times T_{mm}$



- $T_{avg} = 0.95 \times 10 + 0.05 \times 200 = 19.5$ ns

Problem 2

- Given 32-bit computer with MIPS ISA and the following code fragment at address 0x00000000

```
        li    $t0, 1000
        li    $t1, 0
        li    $t2, 0
loop:   addi  $t1, $t1, 1
        addi  $t2, $t2, 4
        bneq  $t1, $t0, loop
```

Problem 2 (cont'd)

- The computer has cache with following characteristics:
  - Size 32 KB
  - 16 B line size
  - 4-way set associative
- Compute:
  - Nr of cache misses for given code fragment
  - Cache hit rate for given code fragment
- Assumptions:
  - No interrupts
  - Cache initially empty

Solution

```
        li   $t0, 1000
        li   $t1, 0
        li   $t2, 0
loop:   addi  $t1, $t1, 1
        addi  $t2, $t2, 4
        bneq  $t1, $t0, loop
```

- Each instruction accesses the memory *once*
- Loop executed 1000 times
- Total memory accesses:

  3 + 3 x 1000 = 3003

Solution

```
        li   $t0, 1000
        li   $t1, 0
        li   $t2, 0
loop:   addi $t1, $t1, 1
        addi $t2, $t2, 4
        bneq $t1, $t0, loop
```

- 16 B line size
- 32-bit instructions -> 4 B per instruction -> 4 instructions per cache line:
  - First 4 instructions stored in first line (at address 0x00000000)
  - Next 2 instructions stored in second line
- Cache initially empty:
  - 1 cache miss for first 4 instructions
  - 1 cache miss for next 2 instructions

2 cache misses

  - No cache miss afterwards! (all instructions allready in cache)
    - 999 x 3 cache hits from last 999 loop iterations
    - 4 cache hits for:

```
li   $t1, 0
li   $t2, 0
loop:   addi $t1, $t1, 1
bneq $t1, $t0, loop
```

3001 hits, 3003 accesses -> hit rate = 3001/3003

Problem 3

- Cache
  - Size 32 KB (data + instructions)
  - 64 byte cache line, byte-addressable
  - Fully associative
  - t_c = 30ns
- Main memory
  - 1 MB
  - byte-addressable
  - t_m = 80ns

- What does the structure of the cache memory look like?

Cache structure

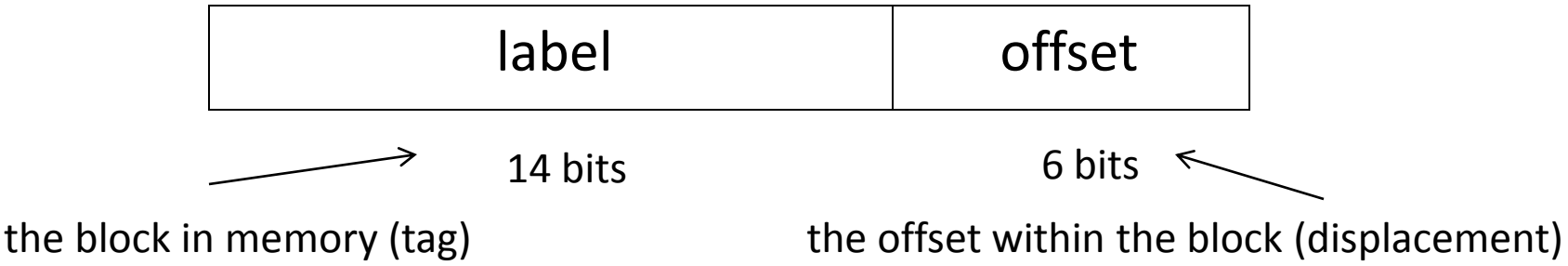
- Fully associative -> main memory blocks can be stored *anywhere* in the cache -> nr of bits for label given by

  main_memory_size / size_cache_block = 1 MB / 64 B = 2^{20} / 2^{6} = 2^{14} -> 14 bits

- Nr cache lines: 32 KB / 64 B = 2^{9} lines
- Nr bits per cache line: 64 B = 2^{9} bits

What is the address format?

- Cache line 64 B, byte-addressable
  -> 6 bits for offset
- 14 bits for label
- What about set?
  - Not necessary - any set will do in fully associative caches



- Given the following address sequence:

0x7B042

0x7D042

0x7D074

- Which ones result in a cache miss and why?
- What is the access time to the above sequence?

Cache miss/hit

- 0x7B042 -> 0111 1011 0000 0100 0010 ->
  01111011000001 for label
  000010 for offset

  Cache miss (empty cache)
- 0x7D042 -> 0111 1101 0000 0100 0010 ->
  01111101000001 for label
  000010 for offset

  Address different, cache miss
- 0x7D074 -> 0111 1101 0000 0111 0100 ->
  01111101000001 for label
  110100 for offset

  Block (line) already in cache!

Access time

- Cache hit: t_c = 30ns
- Cache miss: t_m + t_c = 110ns

- Access time: 2 misses, 1 hit

  2 x 110 + 30 = 250ns

Number of cache misses/hits

- for (i = 0; i < 128; i++) {
  a[i] = 3 * i;
  b[i] = 5 * i;
  }
- Where a, b are vectors of integers stored in memory at consecutive addresses:
  - 0x0F020 (a)
  - 0x08F20 (b)
- The index and constants are NOT stored in cache
- What is the number of cache misses / hits?

- Size of a: 128 * 4 = 512 B
- Size of b: 128 * 4 = 512 B
- Together: 1 KB
- Cache size: 32 KB -> a and b fit together in the cache
- Cache is fully associative -> blocks can be stored anywhere in the cache!

- a[0] cache miss -> bring 64 byte block into cache, i.e. 16 words / integers
  - Next 15 accesses to elements of a are hits
  - Same for b

- Number of misses (128 / 16) * 2
- Number of hits (128 * 15 / 16) * 2

Problem 4

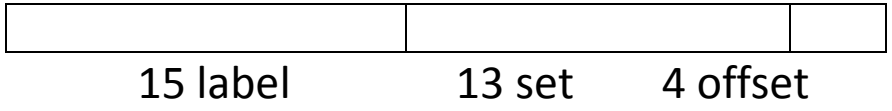
- 32 bit computer, byte-addressable
- Instruction and data caches
  - 16 byte line size
  - n-way set associative
  - LRU replacement algorithm
- for (i = 0; i < 2^18; i++) {
  a[i] = b[i] + c[i] + d[i];
  }

  where a, b, c, d vectors of 2^18 integers stored in memory at consecutive addresses. a starts at 0x00500000.
- Cache initially empty

- Data cache – 256 KB
  - n = 2 -> 2-way
  - n = 4 -> 4-way

- What are the hit rates?
- What happens if the cache size doubles?

- Vector size: 2^18 * 4 = 2^20 bytes
  - a: from 0x00500000 to 0x005FFFFF
  - b: from 0x00600000 to 0x006FFFFF
  - c: from 0x00700000 to 0x007FFFFF
  - d: from 0x00800000 to 0x008FFFFF
- 2-way associative
  - Set size: 16 bytes * 2 = 2^5 bytes
  - Nr of sets: 256 / 32 = 2^13 -> 13 bits for set
  - 16 byte cache line -> 4 bits for offset
  - 15 bits for label



Cache misses – 2-way associative

b[0]	0x00600000	set 0, line 0	miss
c[0]	0x00700000	set 0, line 1	miss
d[0]	0x00800000	set 0, line 0	miss (replace b)
a[0]	0x00500000	set 0, line 1	miss (replace c)
b[1]	0x00600004	set 0, line 0	miss (replace d)
c[1]	0x00700004	set 0, line 1	miss (replace a)
d[1]	0x00800004	set 0, line 0	miss (replace b)
a[1]	0x00500004	set 0, line 1	miss (replace c)
….			
b[4]	0x00600010	set 1, line 0	miss
c[4]	0x00700010	set 1, line 1	miss
d[4]	0x00800010	set 1, line 0	miss (replace b)
a[4]	0x00500010	set 1, line 1	miss (replace c)
….			

Hit rate = 0

4-way associative

- 16 * 4 = 2^6 bytes per set
- 256 / 64 = 2^12 -> 12 bits for set
- 4 bits for offset
- 16 bits for label

b[0]	0x00600000	set 0, line 0	miss
c[0]	0x00700000	set 0, line 1	miss
d[0]	0x00800000	set 0, line 2	miss
a[0]	0x00500000	set 0, line 3	miss
b[1]	0x00600004	set 0, line 0	hit
c[1]	0x00700004	set 0, line 1	hit
d[1]	0x00800004	set 0, line 2	hit
a[1]	0x00500004	set 0, line 3	hit
….			
b[4]	0x00600010	set 1, line 0	miss
c[4]	0x00700010	set 1, line 1	miss
d[4]	0x00800010	set 1, line 2	miss
a[4]	0x00500010	set 1, line 3	miss
….			

12 hits every 16 accesses -> 75%

Double size cache?

- No difference