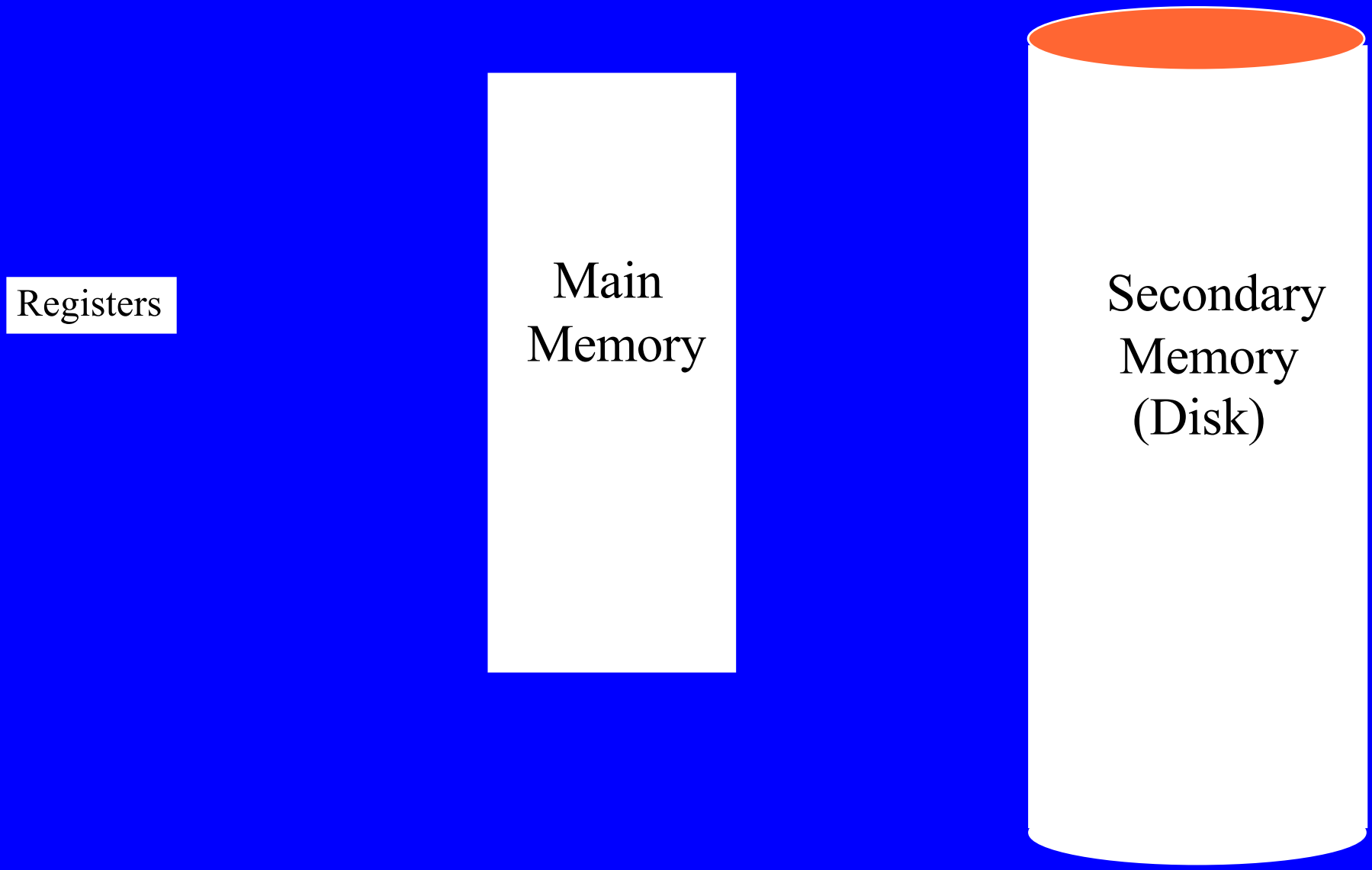
Memory Hierarchy



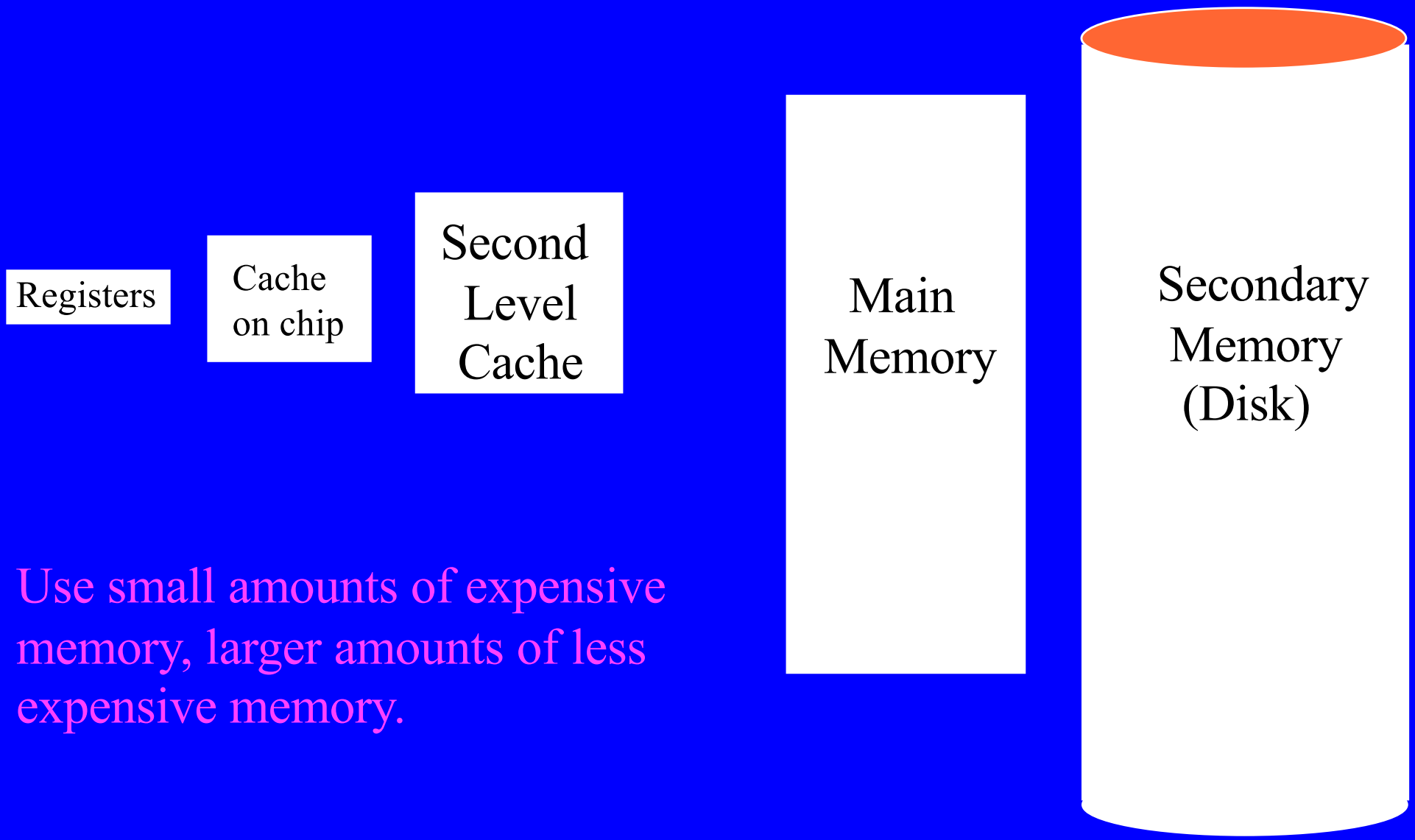
Memory

- Main Memory (DRAM) much slower than processors
- Gap has widened over the past fifteen years, continues to widen
- Memory comes in different technologies: fastest is also the most expensive

Slowest
Least expensive
Mag. Disk: 10-20 million ns
$0.10-$0.20/MByte

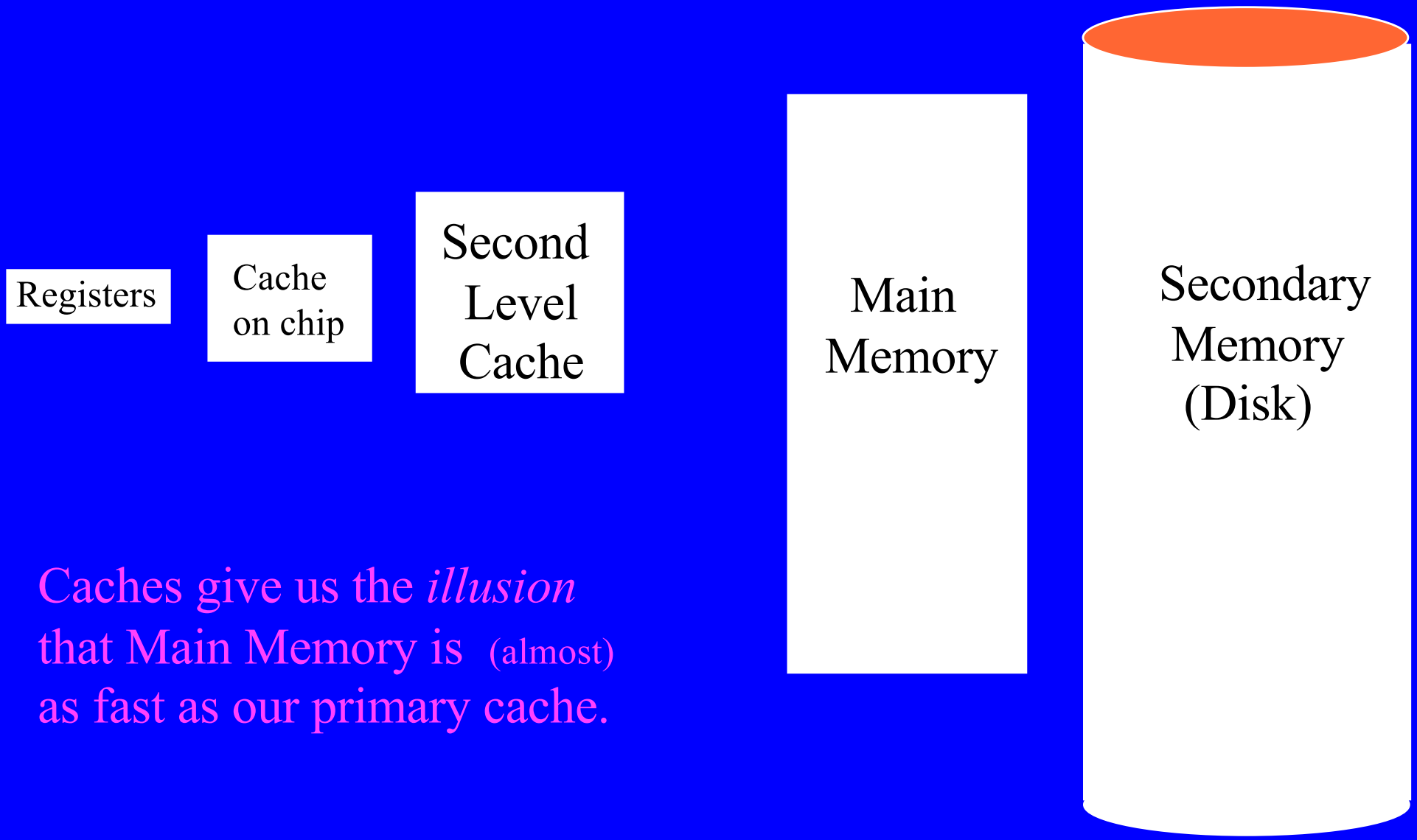
Fastest
Most expensive
SRAM: 5-10 ns
$100-$200/MByte

Memory Hierarchy



Use small amounts of expensive memory, larger amounts of less expensive memory.

Memory Hierarchy



Caches give us the *illusion* that Main Memory is (almost) as fast as our primary cache.

Using Memory

- **Temporal locality** - likely to reuse data soon

  Use same variable repeatedly in time (temporal = time)

  When might we see this?

- **Spatial locality** - likely to access data close by

  Use data close to the current data in space (spatial = space)

  When might we see this?

Using Memory

- **Temporal locality** - likely to reuse data soon (loops)
- **Spatial locality** - likely to access data close by (sequential nature of program instructions, data access)
- Use small amounts of expensive memory, larger amounts of less expensive memory
- Memory hierarchy used to hide memory access latency

Using the Memory Hierarchy

- Check if data is in fast memory (**hit**); if not (**miss**), fetch a block from slower memory
  - "Block" may be a single word or several words
- Fetch time is **miss penalty**
- **Hit ratio** is fraction of memory accesses that are hits; **miss rate** = 1 - hit ratio
- If locality is high, hit rate gets closer to 1
- Effect: memory is almost as fast as fastest level, as big as biggest (slowest) memory

Measuring Performance

- Access Time (**Taccess**) is:

  Taccess = Thit + (missRate * missPenalty)

  where **Thit** is the access time for a hit

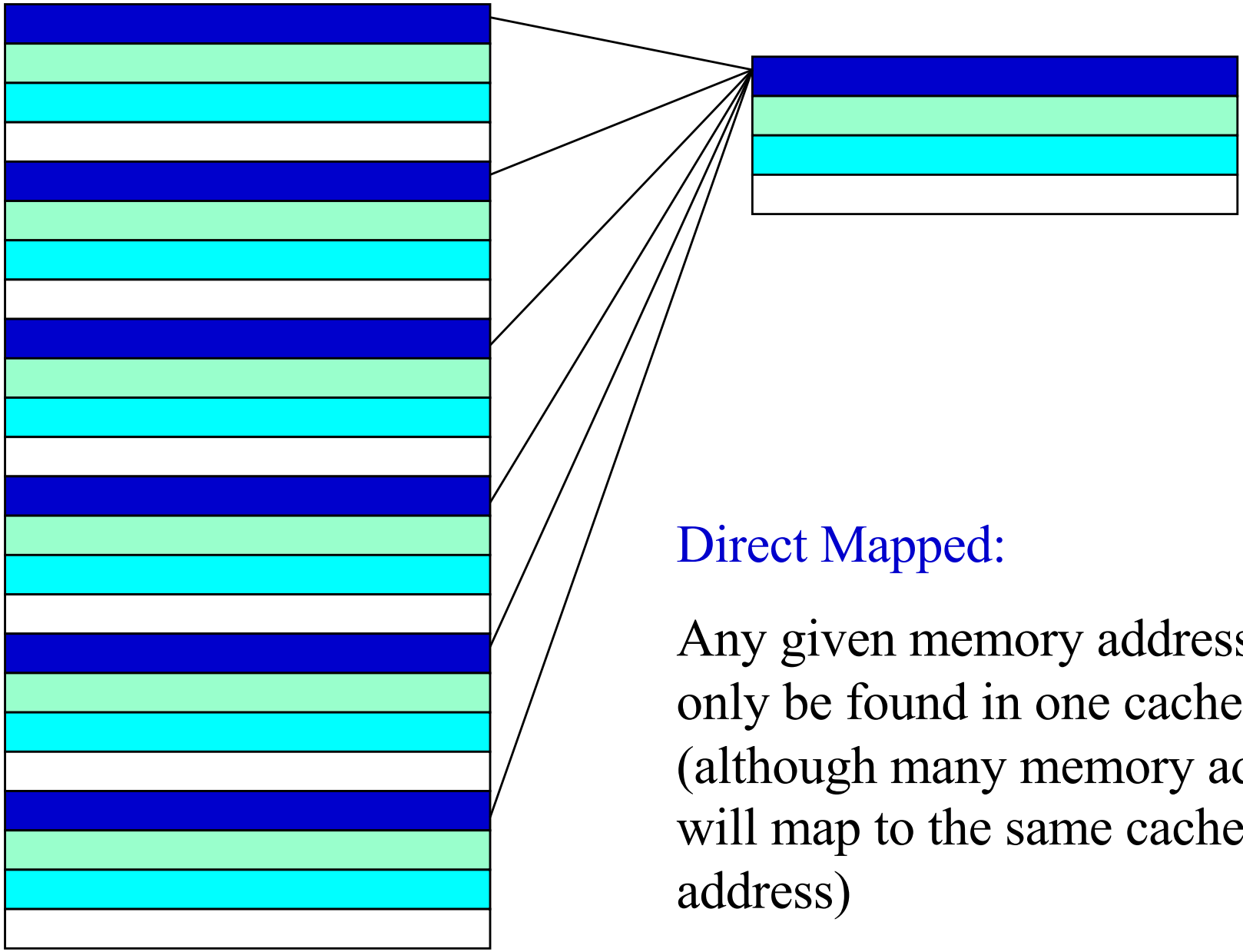
Memory Level Issues

- What level has the data I want?

  (hit or miss at highest level?)
- Where does true image of data/code reside?
  - Caches are working copies, true image is in main memory
  - Virtual memory, true image may be in main memory or in secondary memory -- really a point of view

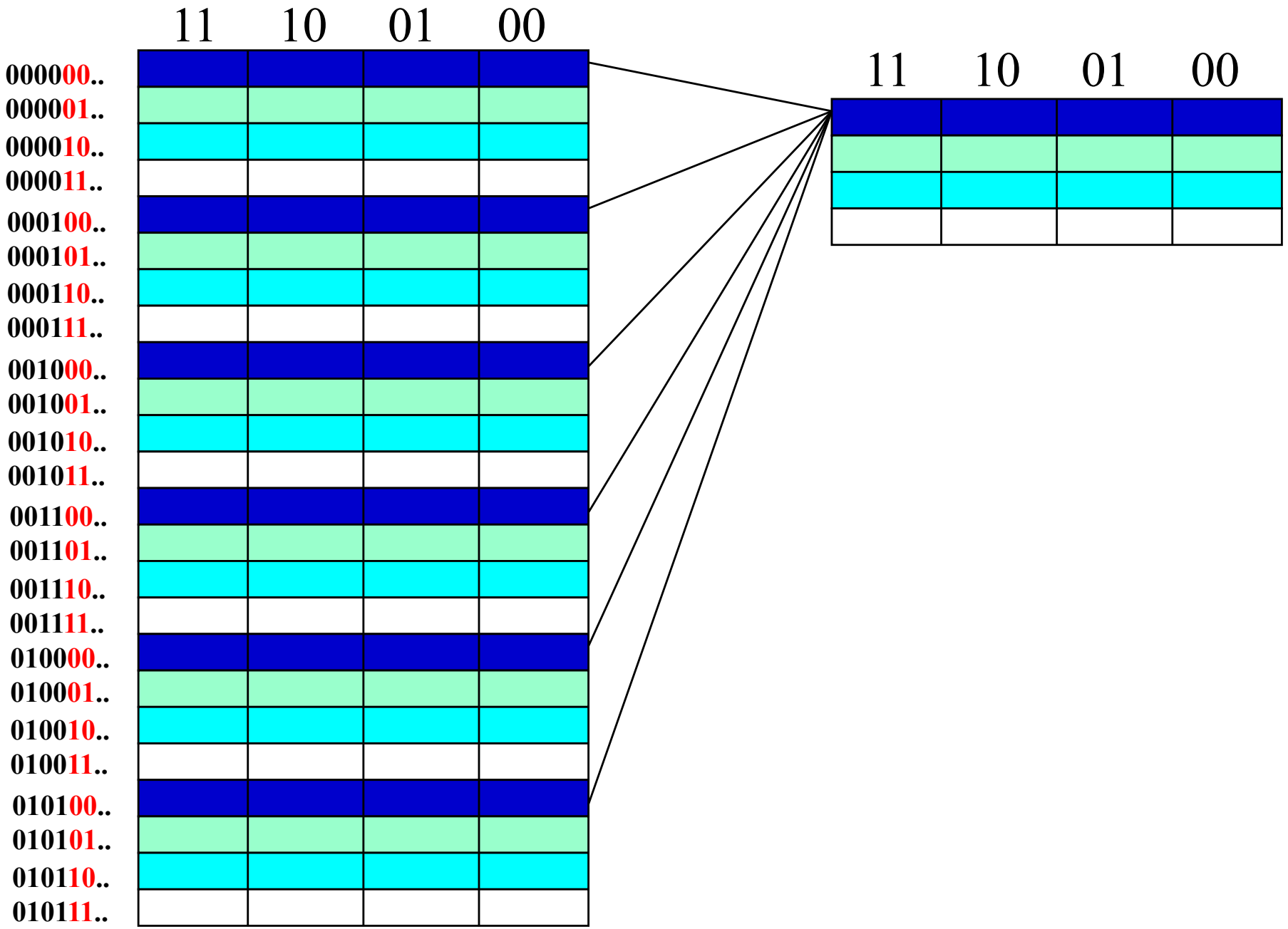
Cache Issues

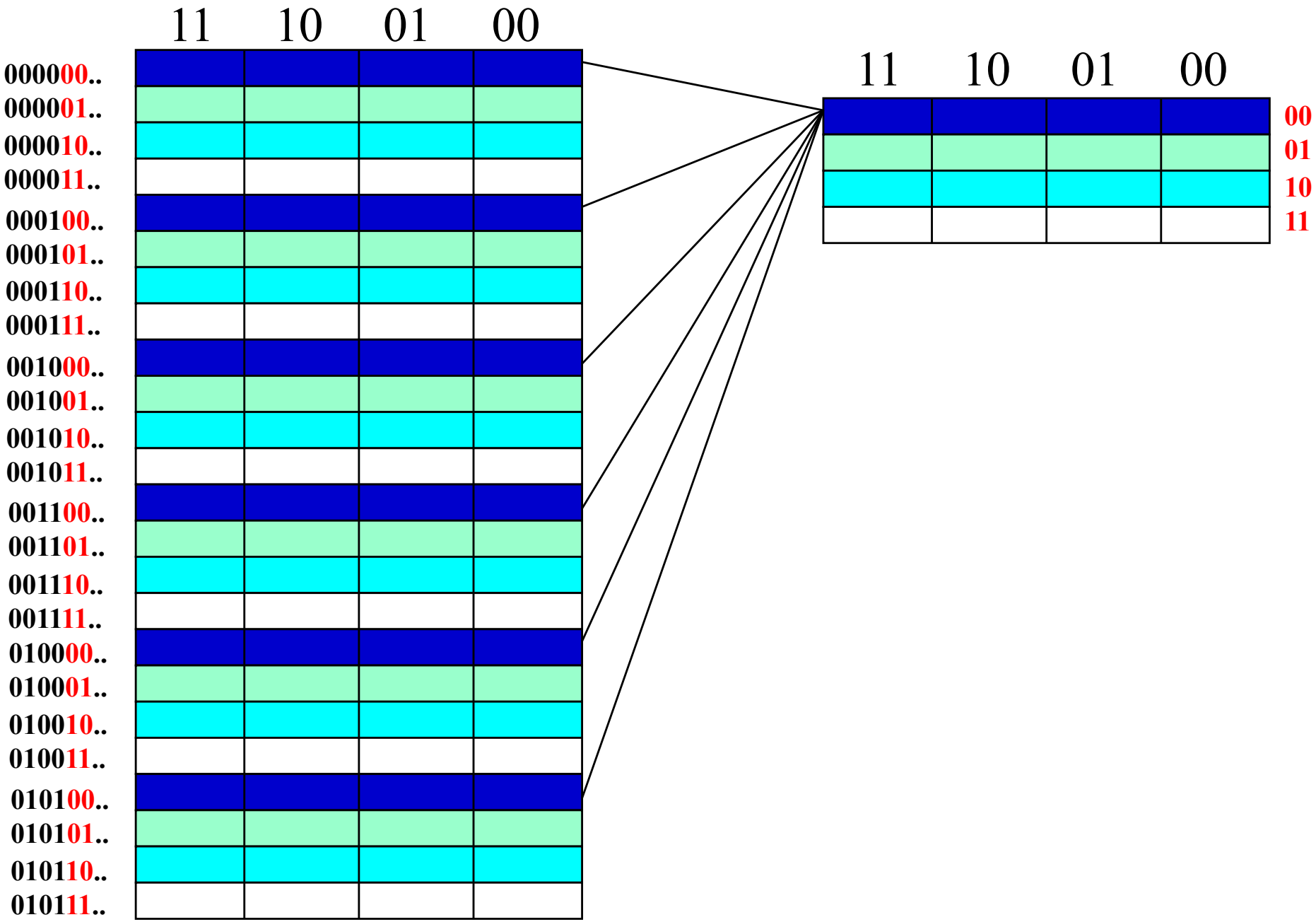
- Cache is a working copy of memory image
- Cache exploits temporal proximity
  - recent data/instruction likely to be used again
- How is cache organized and addressed?
- When cache is written to, how is memory image updated?
- When cache is full and new items need to be put in the cache -- what is removed and replaced?



Direct Mapped:

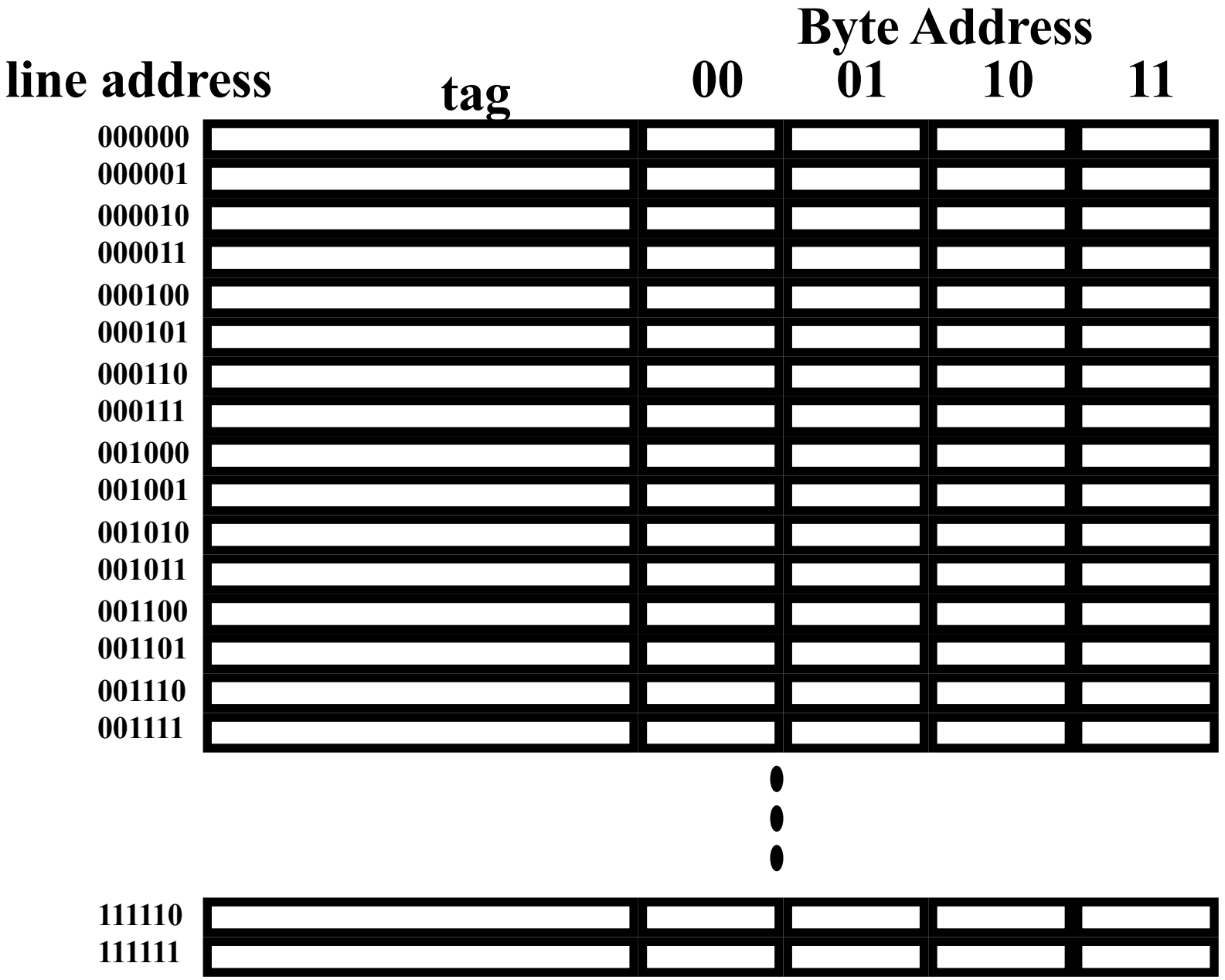
Any given memory address can only be found in one cache address (although many memory addresses will map to the same cache address)


11
10
01
00
000000..
000001..
000010..
000011..
000100..
000101..
000110..
000111..
001000..
001001..
001010..
001011..
001100..
001101..
001110..
001111..
010000..
010001..
010010..
010011..
010100..
010101..
010110..
010111..
11
10
01
00


11
10
01
00
000000..
000001..
000010..
000011..
000100..
000101..
000110..
000111..
001000..
001001..
001010..
001011..
001100..
001101..
001110..
001111..
010000..
010001..
010010..
010011..
010100..
010101..
010110..
010111..
11
10
01
00
00
01
10
11

Cache Example (Direct Mapped)

- 256 Byte Cache (64 4-byte words)
- Each Cache “line” or “block” holds one word (4 bytes)
- Byte in cache is addressed by lowest two bits of address
- Cache line is addressed by next six bits in address
- Each Cache line has a “tag” matching the high 24 bits of the memory address


Byte Address
line address
tag
00
01
10
11
000000
000001
000010
000011
000100
000101
000110
000111
001000
001001
001010
001011
001100
001101
001110
001111
111110
111111

Address 1100 0000 1010 0000 0000 0110 0010 1000

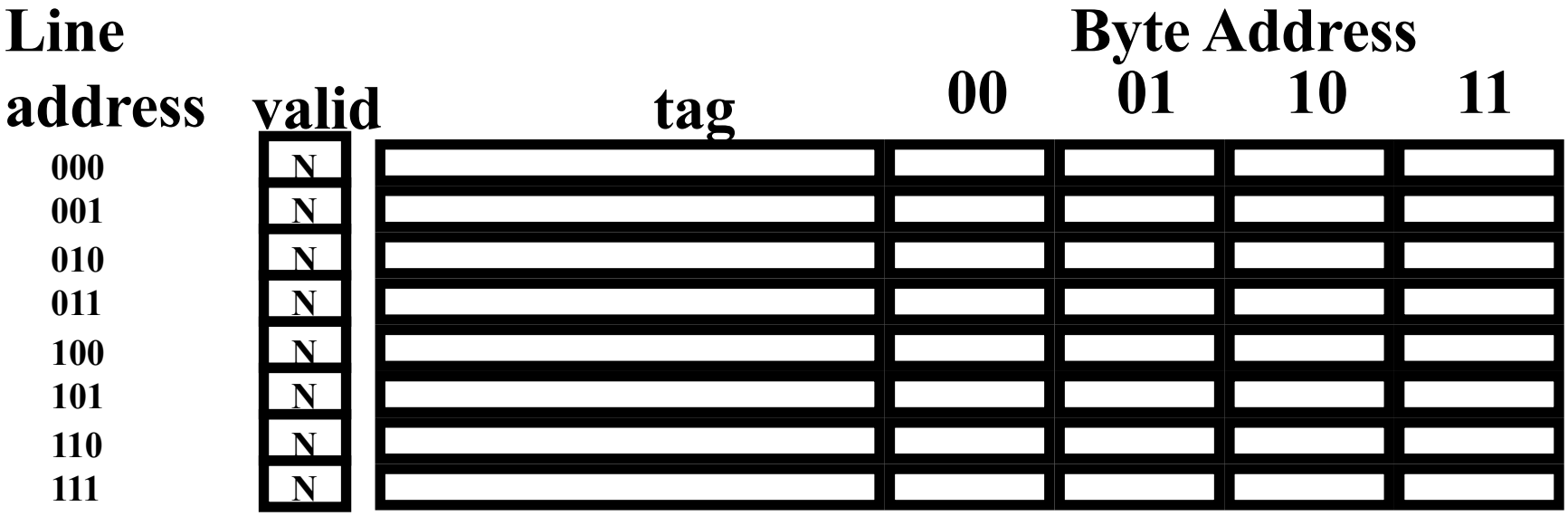
Address 1100 0000 1010 0000 0000 0110 0010 1000

Cache Access

1. Find Cache line address (bits 2 - 7)
2. Compare tag to high 24 bits
   - if matched, cache hit
     - » find Byte address, read or write item
   - if not matched, cache miss, go to memory
     - » for a read: retrieve item and write to cache, then use
     - » for a write: write to memory (or to cache, see below)
3. Direct mapped cache -- every address can only go to one cache line!
4. What happens when cache is written to?

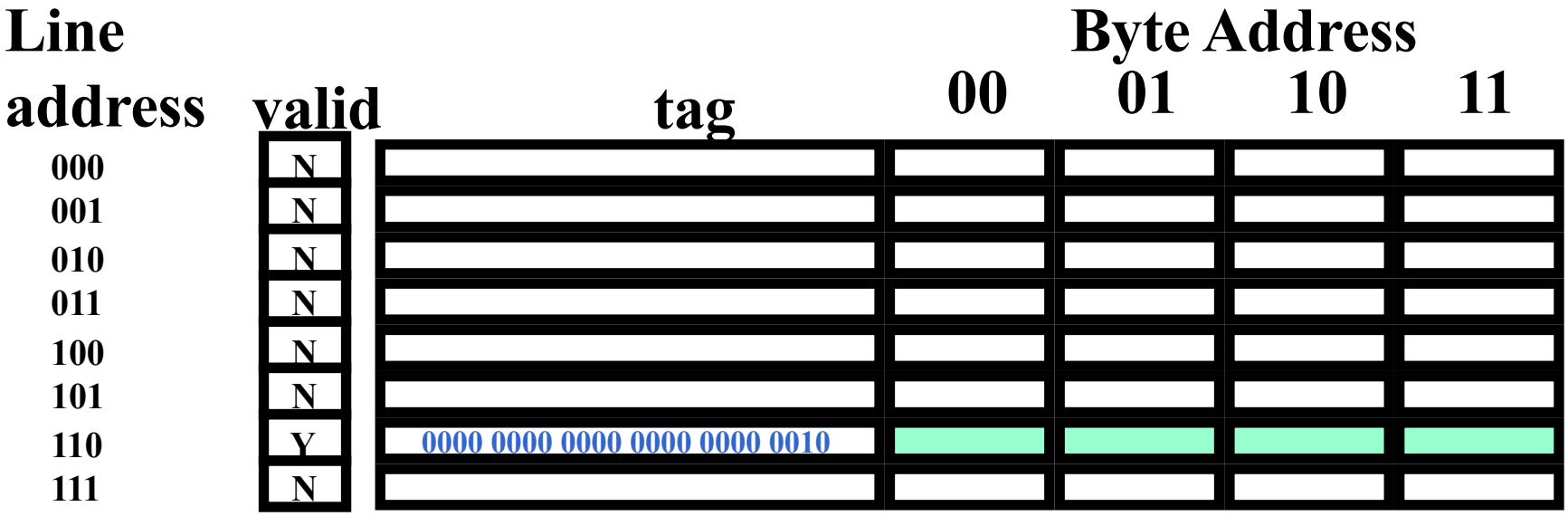
Write Policy

- Write Through
  - Write to memory and to cache
  - Time to write to memory could delay instruction
    - » write buffer can hide this latency
- Write Back (also called Copy Back)
  - Write only to cache
    - » mark cache line as “dirty”, using an additional bit
  - When cache line is replaced, if dirty, then write back to memory



Desired Address

22
26
22
26
16
3
16
18



Desired Address

22 (22 mod 8 = 6, 110); MISS! -- fetch
26
22
26
16
3
16
18

Desired Address

22

26 (26 mod 8 = 2, 010); MISS! -- fetch

22

26

16

3

16

18

Desired Address

22

26

22 (22 mod 8 = 6, 110); HIT!

26

16

3

16

18

Desired Address

22
26
22
26 (26 mod 8 = 2, 010); HIT!
16
3
16
18

Desired Address

22

26

22

26

16 (16 mod 8 = 0, 000); MISS! -- fetch

3

16

18

Desired Address

22

26

22

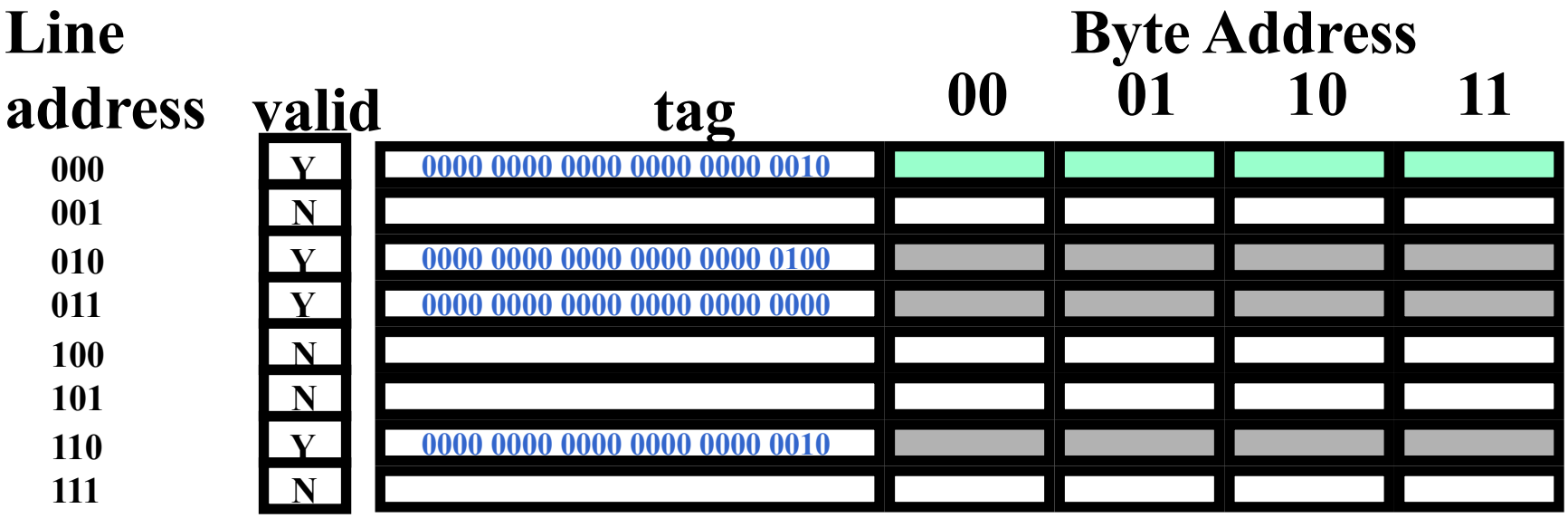
26

16

3 (3 mod 8 = 3, 011); MISS! -- fetch

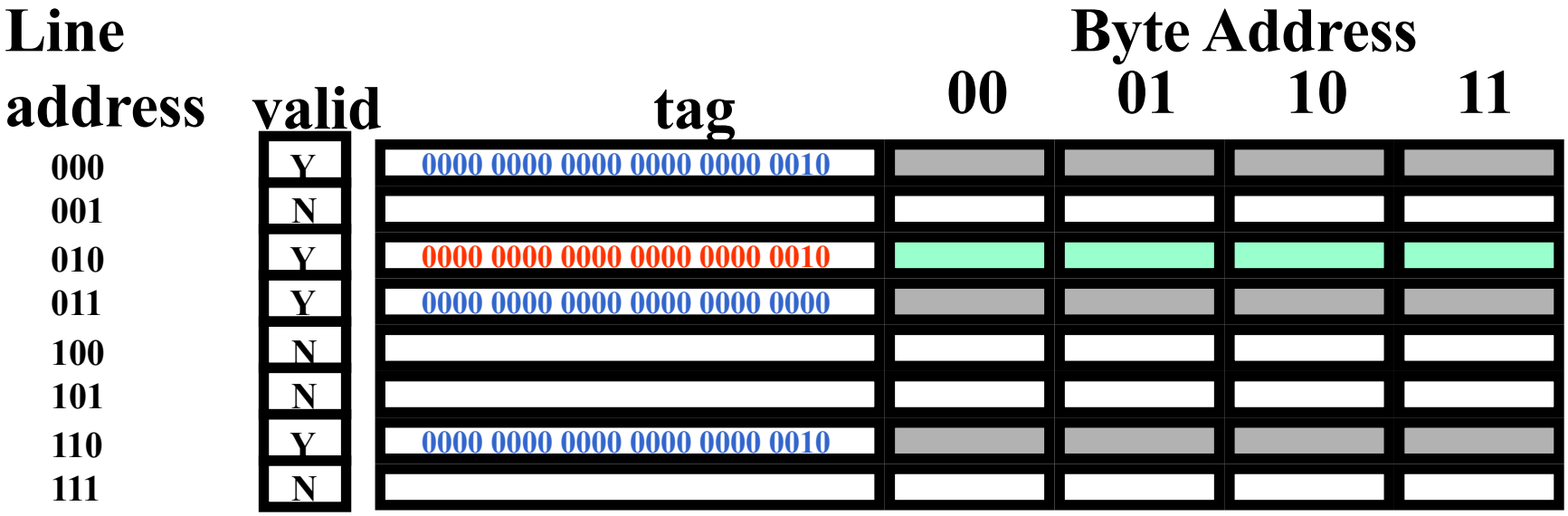
16

18



Desired Address

22
26
22
26
16
3
16 (16 mod 8 = 0, 000); HIT!
18



Desired Address

22

26

22

26

16

3

16

18 (18 mod 8 = 2, 010); MISS! -- fetch (wrong tag)

Accelerating Memory Access

- Bus Bandwidth limited
  - Wider bus, or burst mode
- Memory width limited
  - Wider memory access
- Memory Address limited
  - Burst mode access
    - » one address to retrieve several successive words from memory

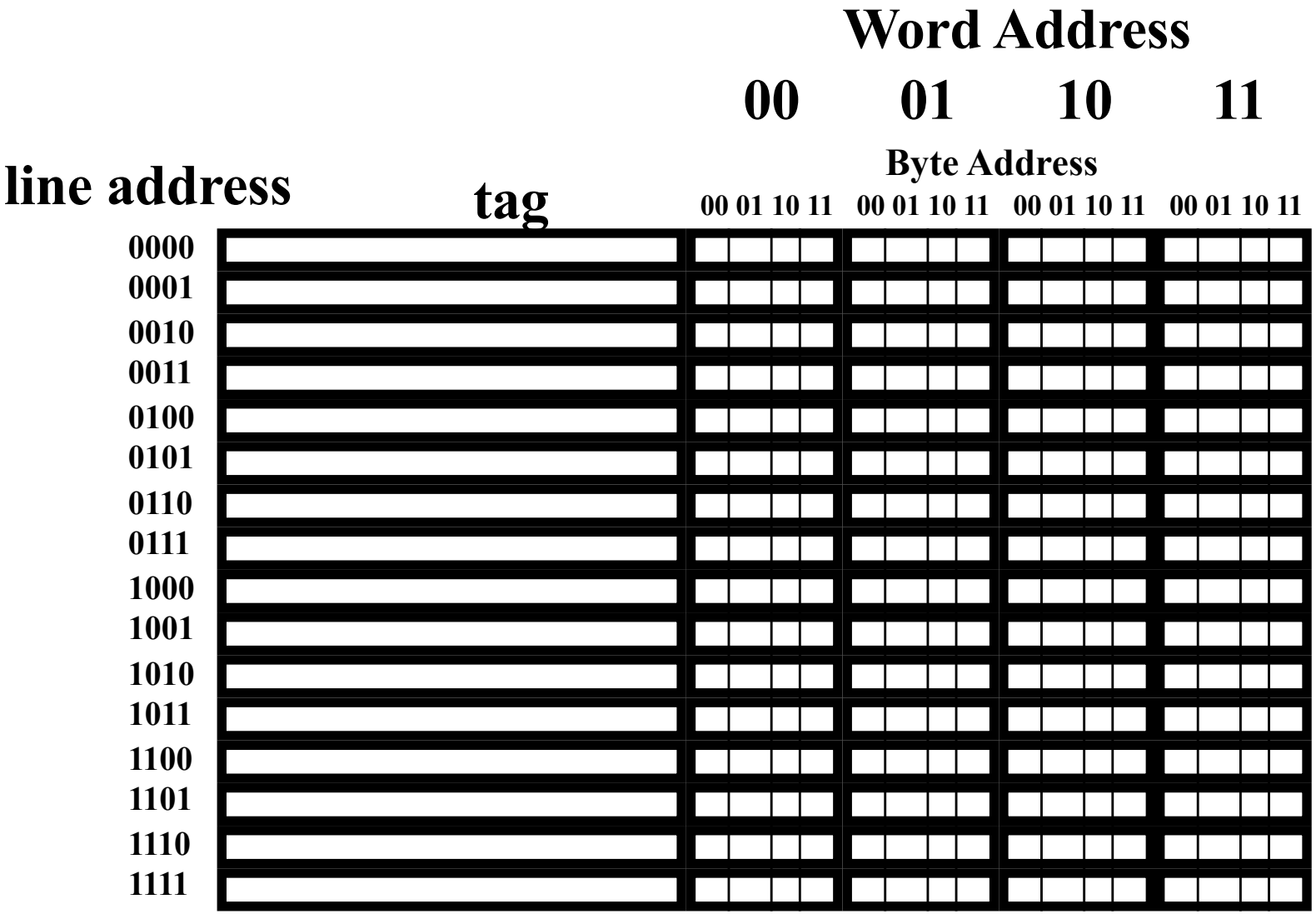
Accelerating Memory Access

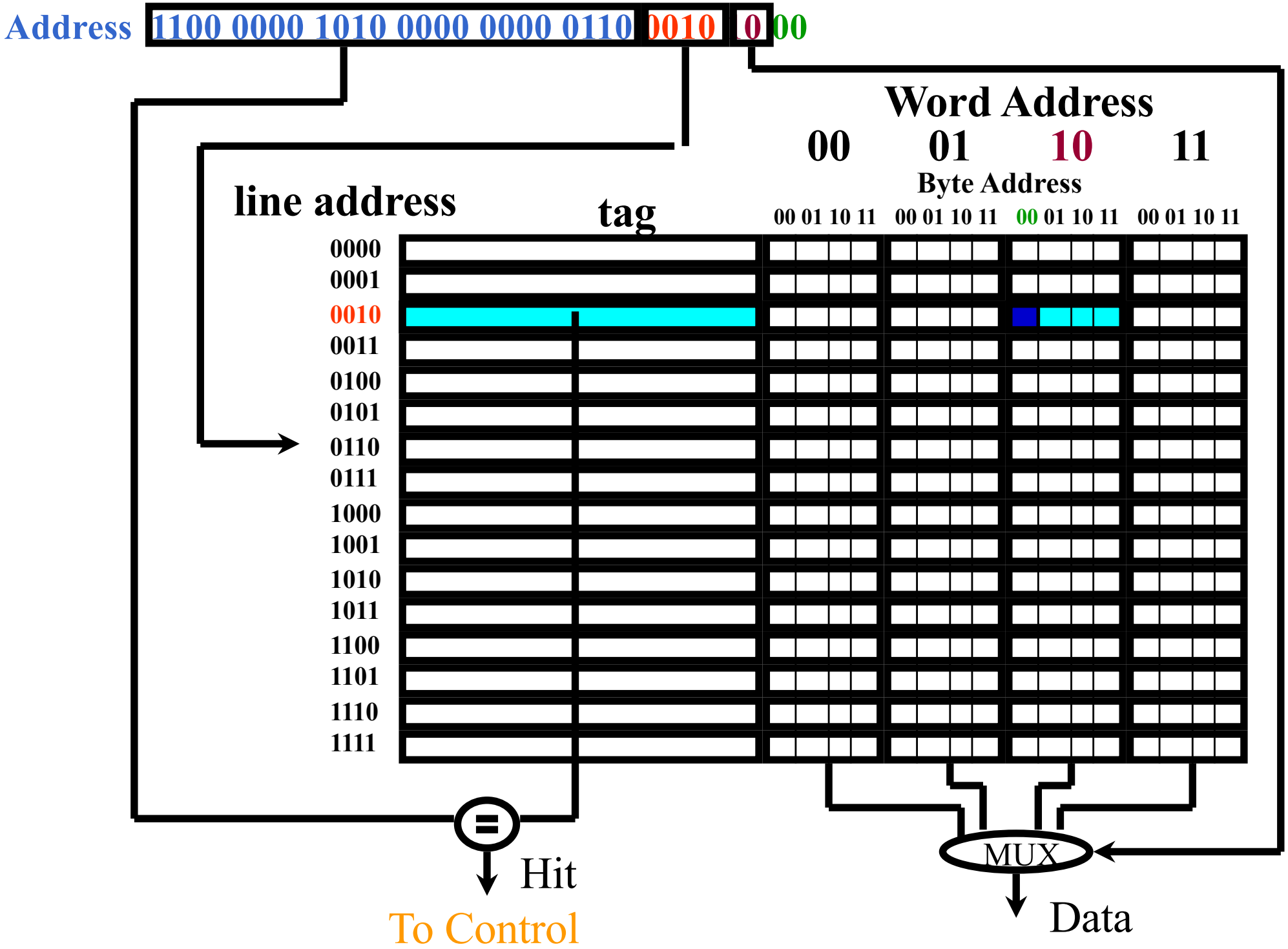
- How can Cache take advantage of faster memory access?
- Store more than one word at a time on each "line" in the cache
  - Any cache miss brings the whole line containing the item into the cache
- Takes advantage of spatial locality
  - next item needed is likely to be at the next address

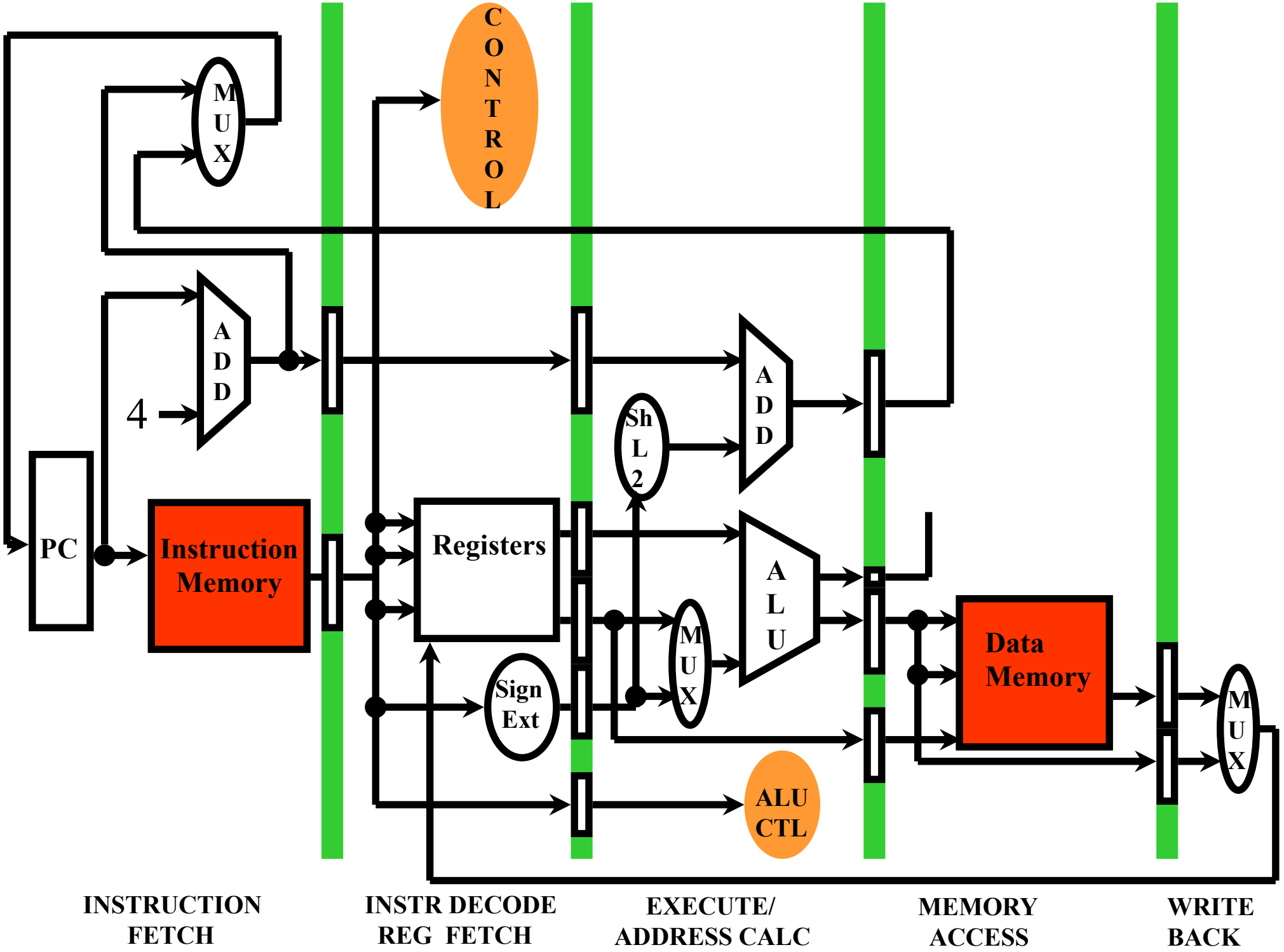
Cache with multi-wordline

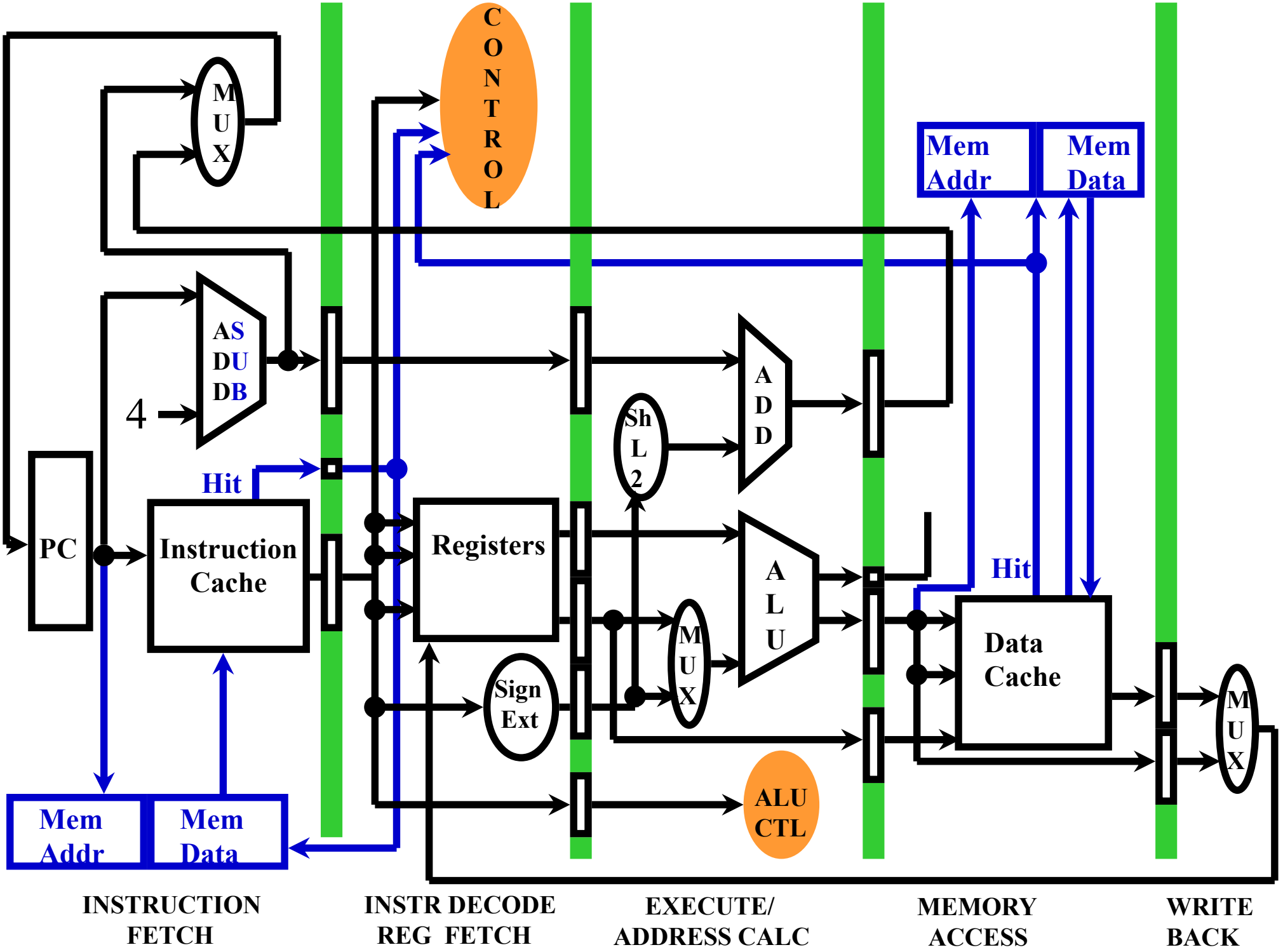
- 256 Byte cache -- 64 4-byte words
- Each block (line) contains four words (16 bytes)
  - 2 bits to address byte in word
  - 2 bits to address word in line
- Cache contains sixteen four-word blocks
  - 4 bits to address cache block (line)
- Each cache line has tag field for upper 24 bits of address

Address 1100 0000 1010 0000 0000 0110 0010 1000




Address
1100 0000 1010 0000 0000 0110 0010 00
Word Address
00 01 10 11
Byte Address
00 01 10 11 00 01 10 11 00 01 10 11 00 01 10 11
line address
tag
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
=
Hit
To Control
MUX
Data


CONTROL
MUX
ADD
4
PC
Instruction Memory
Registers
Sign Ext
Sh L 2
ADD
MUX
ALU
Data Memory
MUX
ALU CTL
INSTRUCTION FETCH
INSTR DECODE REG FETCH
EXECUTE/ ADDRESS CALC
MEMORY ACCESS
WRITE BACK


M
U
X
CONTROL
Mem
Addr
Mem
Data
A
S
D
U
D
B
4
A
D
D
Sh
L
2
Hit
PC
Instruction
Cache
Registers
A
L
U
M
U
X
Hit
Data
Cache
Sign
Ext
M
U
X
Mem
Addr
Mem
Data
ALU
CTL
INSTRUCTION
FETCH
INSTR DECODE
REG FETCH
EXECUTE/
ADDRESS CALC
MEMORY
ACCESS
WRITE
BACK

Instruction Cache Hit / Miss

- Hit or Miss:
  - Instruction is fetched from Cache and placed in Pipeline buffer register
  - PC is latched into Memory Address Register
- Hit:
  - Control sees hit, execution continues
  - Mem Addr unused

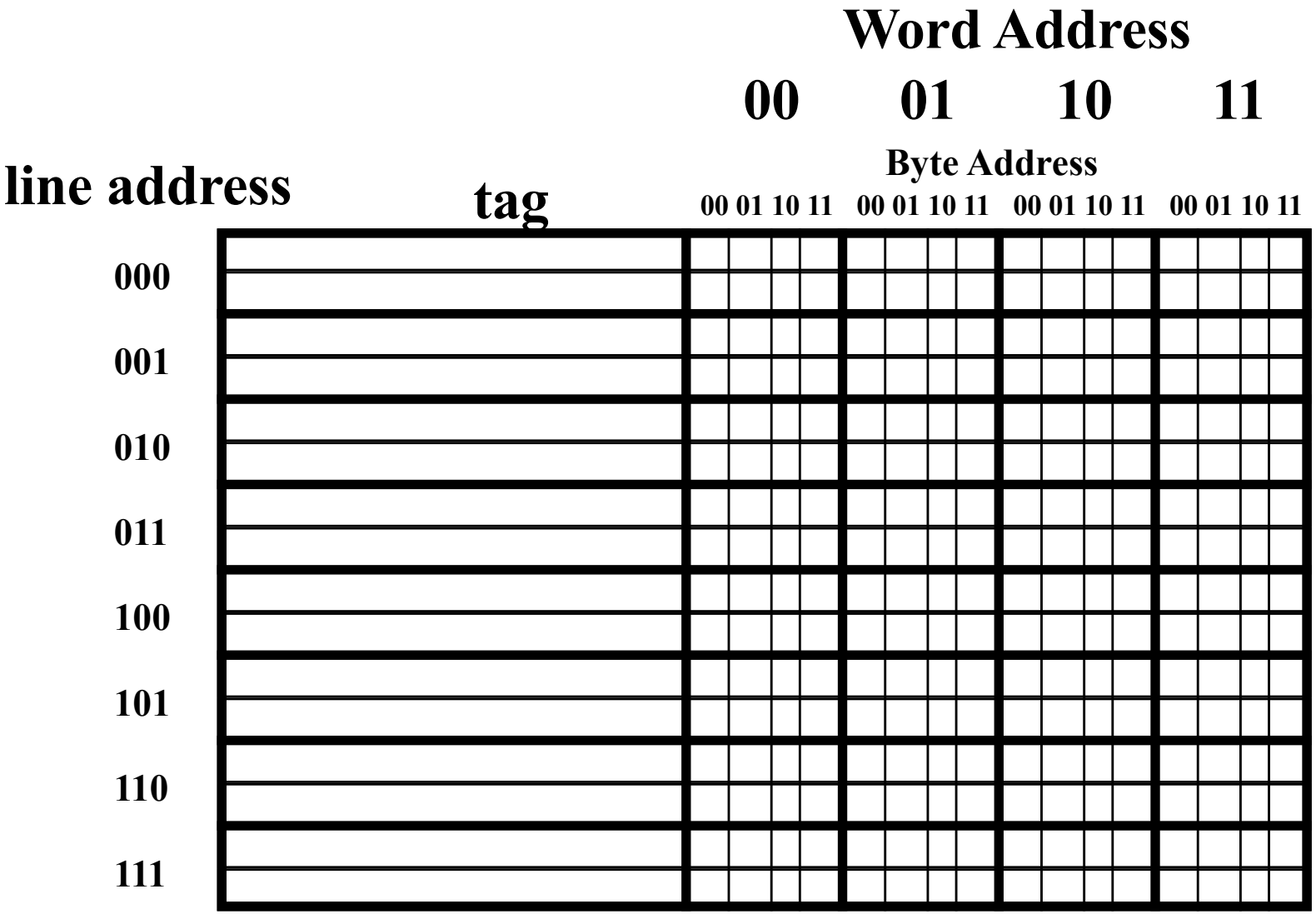
Instruction Cache Hit / Miss

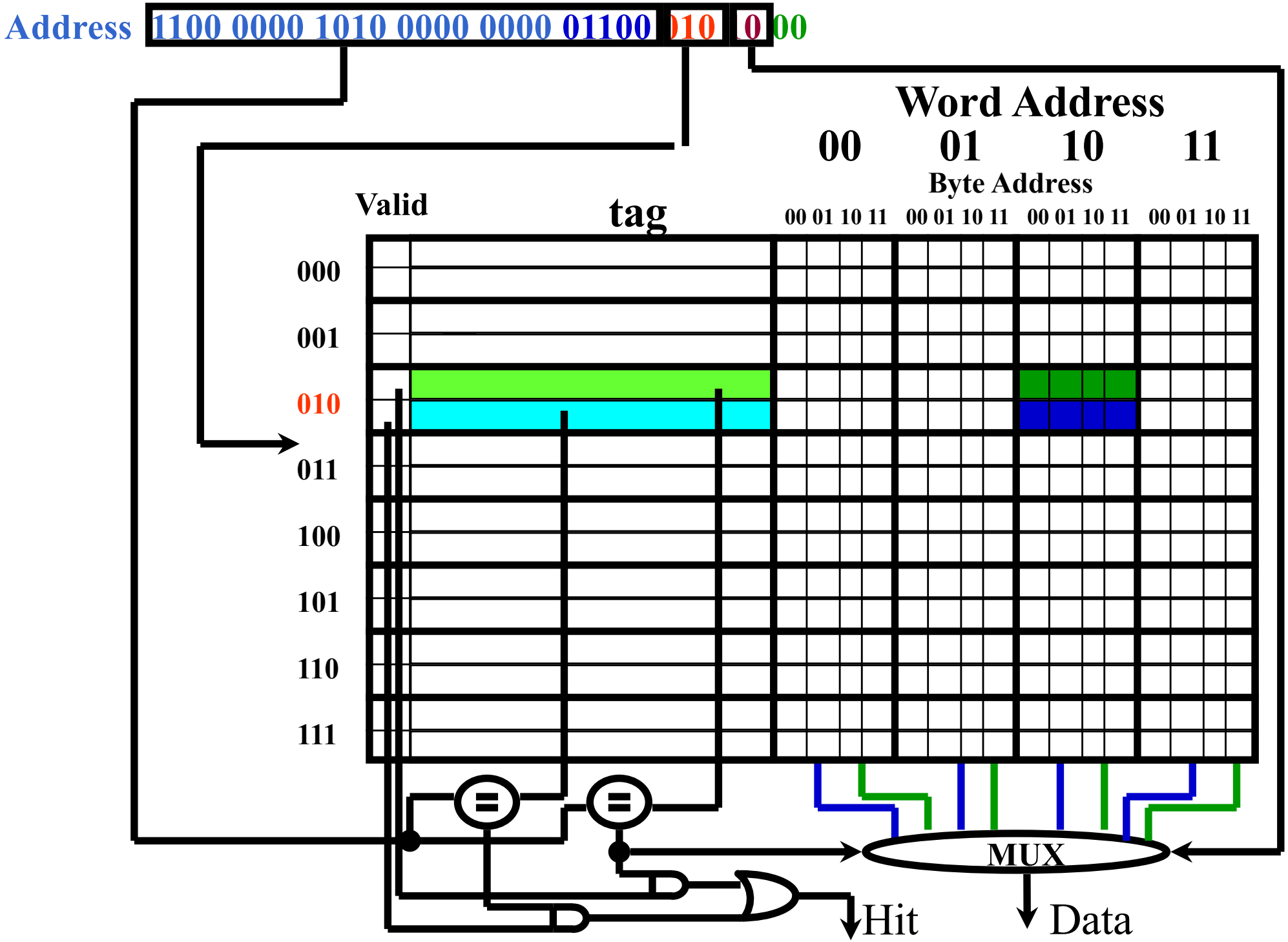
- Miss
  - Control sees miss, execution stalls
  - PC reset to PC - 4
  - Values fetched from registers are unused
  - Memory Read cycle started, using Mem Addr
- Memory Read completes
  - Value stored in cache, new tag written
  - Instruction execution restarts, cache hit

Set Associative Cache

- Direct Mapped Cache
  - Misses caused by collisions -- two address with same cache line
- Set Associative
  - Two or more (power of 2) lines for each address
  - More than one item with same cache line address can be in cache
  - Check means tags for all lines in set must be checked, one which matches yields hit, if none match, a miss

Two-way set associative cache




Address
1100 0000 1010 0000 0000 01100
010
00
Word Address
00
01
10
11
Byte Address
00 01 10 11
00 01 10 11
00 01 10 11
00 01 10 11
Valid
tag
000
001
010
011
100
101
110
111
MUX
Hit
Data

Cache Summary - types

- Direct Mapped
  - Each line in cache takes one address
  - Line size may accommodate several words
- Set Associative
  - Sets of lines serve the same address
  - Needs replacement policy for which line to purge when set is full
  - More flexible, but more complex

Cache Summary

- Cache Hit
  - Item is found in the cache
  - CPU continues at full speed
  - Need to verify valid and tag match
- Cache Miss
  - Item must be retrieved from memory
  - Whole Cache line is retrieved
  - CPU stalls for memory access

Cache Summary

- Write Policies
  - Write Through (always write to memory)
  - Write Back (uses “dirty” bit)
- Associative Cache Replacement Policy
  - LRU (Least Recently Used)
  - Random