CSEE 3827: Fundamentals of Computer Systems, Spring 2022

Lecture 13

Prof. Dan Rubenstein (danr@cs.columbia.edu)

Agenda (P&H 5.1-5.2)

- Memory Hierarchies
- Caching and Cache Replacement Policies
  - Main focus: how to map memory “blocks” to cache
    - Direct mapping
    - Fully Associative and n-way Associative
  - Writing to memory and cache consistency
  - Virtual Memory: what is it?
- Where this course fits in with current tech

Example of Multicore Circuit

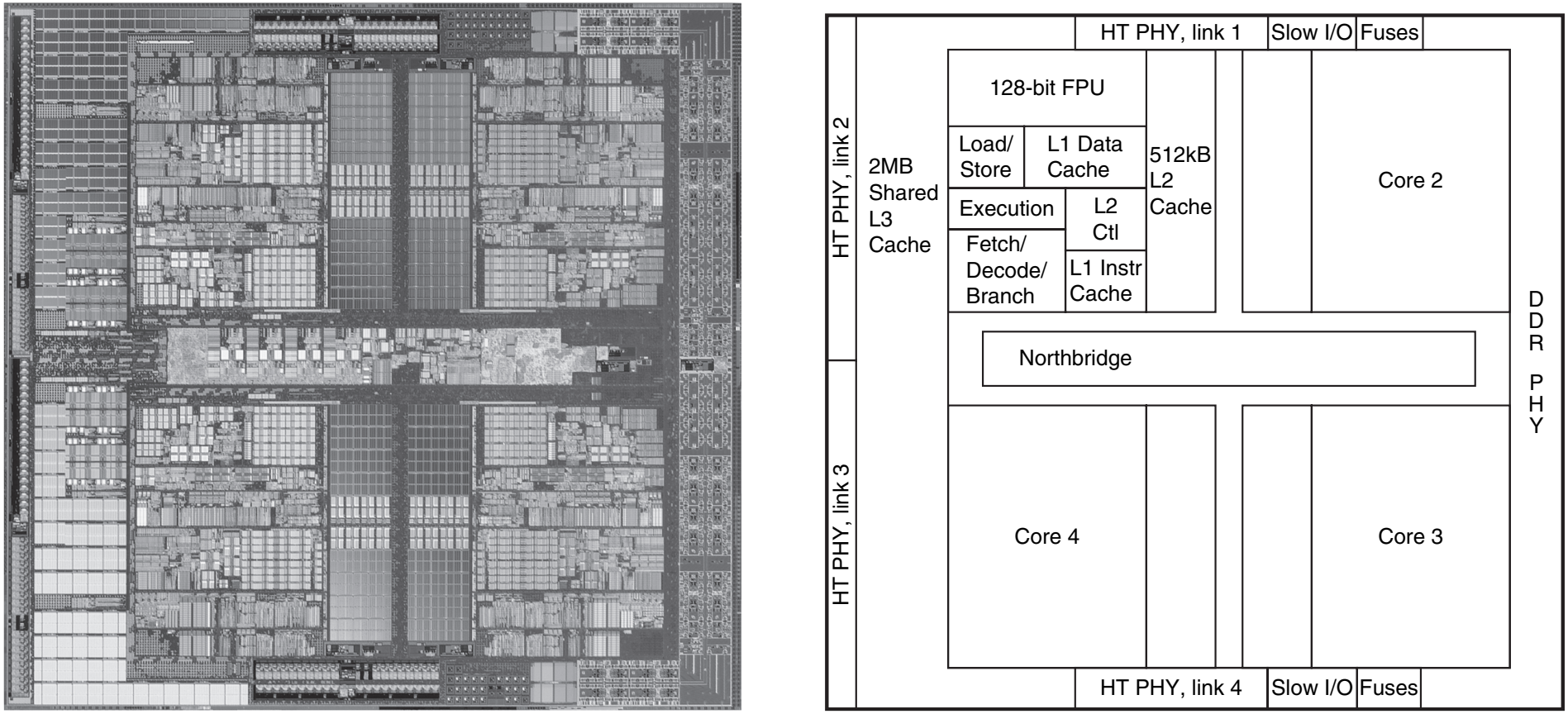


FIGURE 1.9 Inside the AMD Barcelona microprocessor. The left-hand side is a microphotograph of the AMD Barcelona processor chip, and the right-hand side shows the major blocks in the processor. This chip has four processors or "cores". The microprocessor in the laptop in Figure 1.7 has two cores per chip, called an Intel Core 2 Duo.

Memory Hierarchies

Canonical Memory Hierarchy: Cheaper is slower

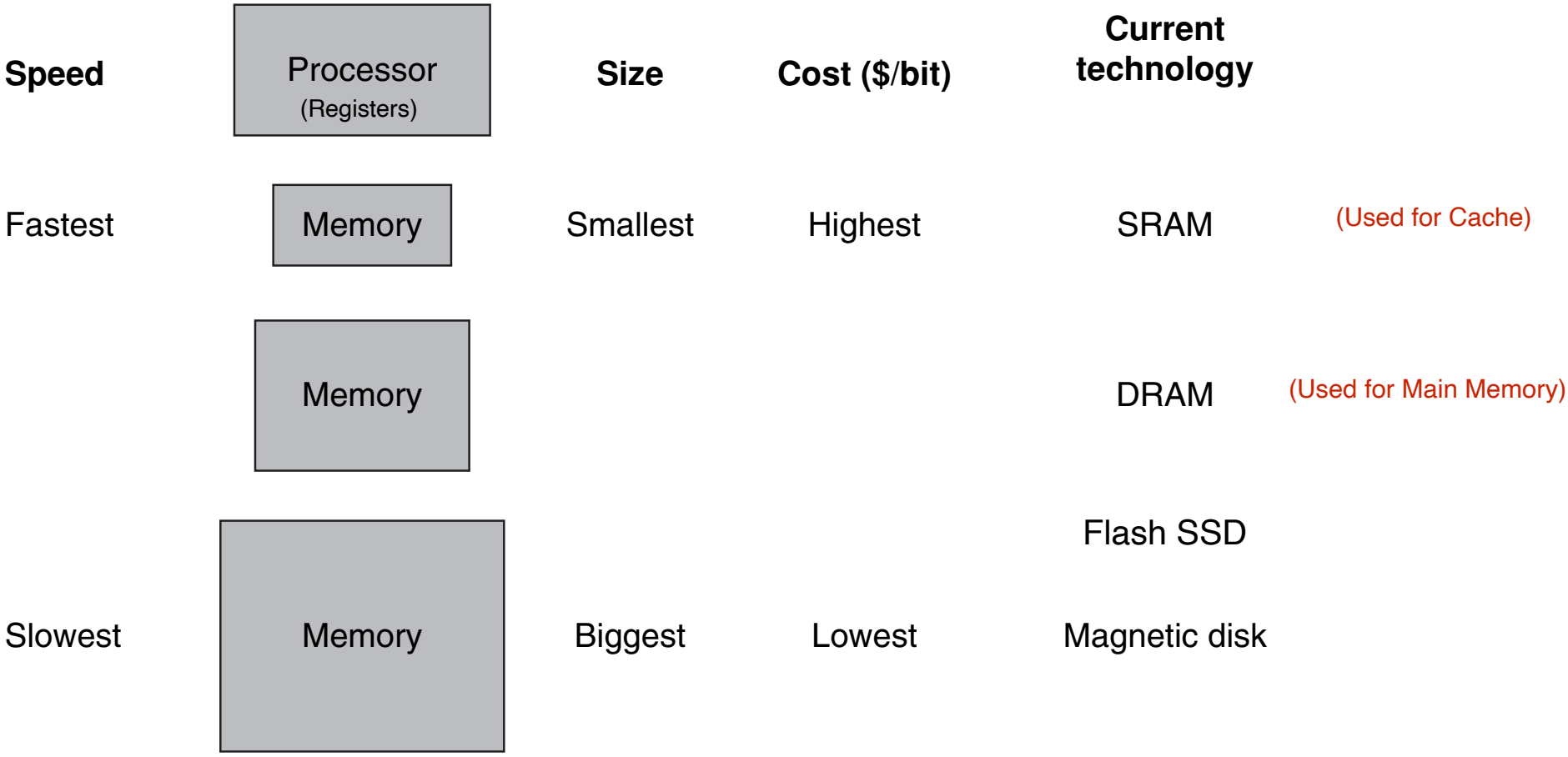


FIGURE 5.1 The basic structure of a memory hierarchy. By implementing the memory system as a hierarchy, the user has the illusion of a memory that is as large as the largest level of the hierarchy, but can be accessed as if it were all built from the fastest memory. Flash memory has replaced disks in many embedded devices, and may lead to a new level in the storage hierarchy for desktop and server computers: see Section 6.4.

Memory Technology: Old prices (2007 / 2008?)

Static RAM (SRAM) → 0.5ns – 2.5ns, $2000 – $5000 per GB (used in cache)

Dynamic RAM (DRAM) → 50ns – 70ns, $20 – $75 per GB (main memory)

Magnetic disk → 5ms – 20ms, $0.20 – $2 per GB

Ideal memory = access time of SRAM + capacity and cost/GB of disk

Memory Technology: Current prices (2022)

Static RAM (SRAM) → 1 ns, $500 per GB (used in cache)

Dynamic RAM (DRAM) → 10 ns, $6 per GB (main memory)

Flash → 50 ns, $1.50 per GB

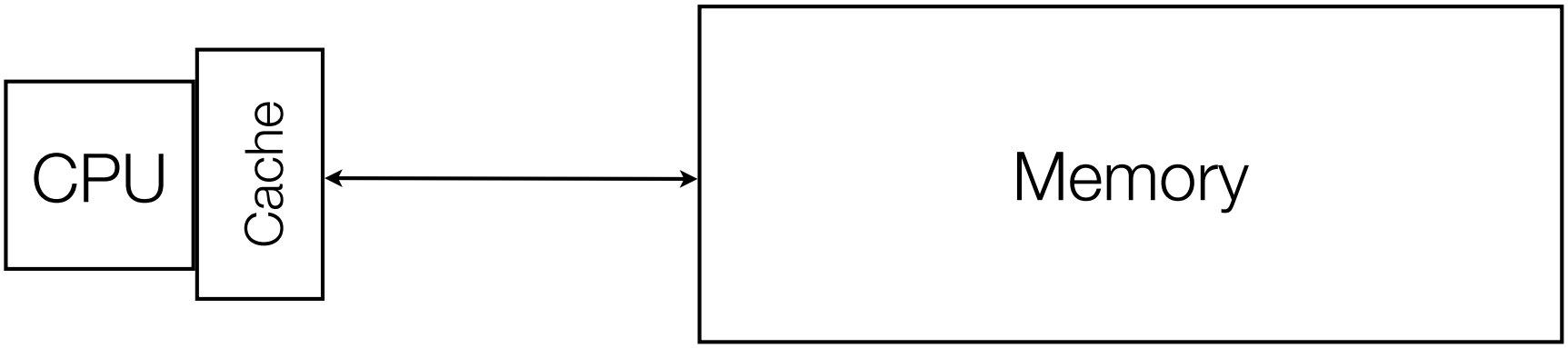
Magnetic disk → 4ms (4000 ns), $0.10 per GB

Ideal memory = access time of SRAM + capacity and cost/GB of disk

Cache

What is a cache?

- Temporary copy of memory
  - CPU can read from and write to cache in much less time than memory

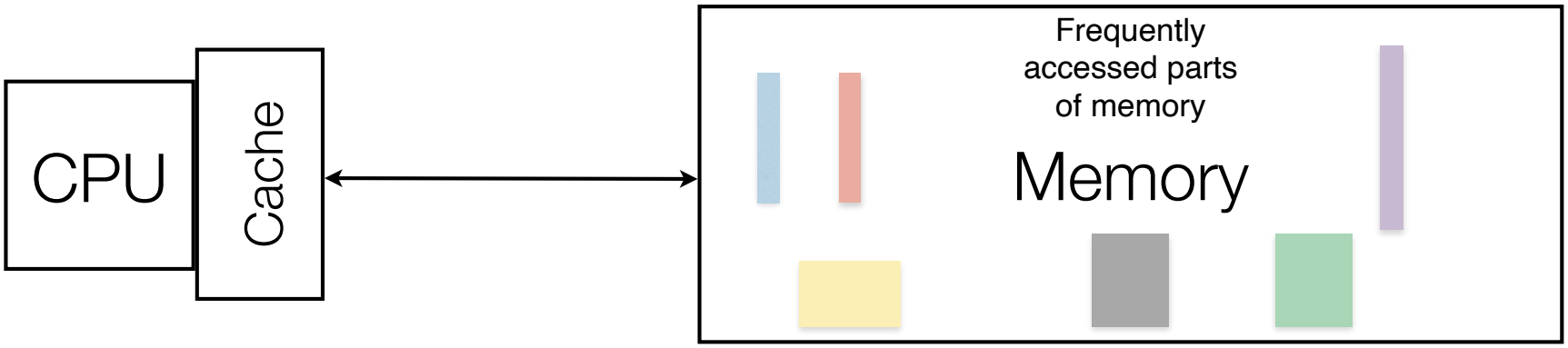


- Cache does not have separate addresses
  - CPU (and assembly code) still “thinks” it is accessing main memory
  - Cache acts as a temporary copy of memory
  - CPU can read from and write to cache in much less time than memory

Cache is faster but much smaller than main mem: can’t store everything, so try to be “smart” about what is stored in the cache to minimize fetches from / writes to main memory

What is a cache?

- Temporary copy of memory
  - CPU can read from and write to cache in much less time than memory

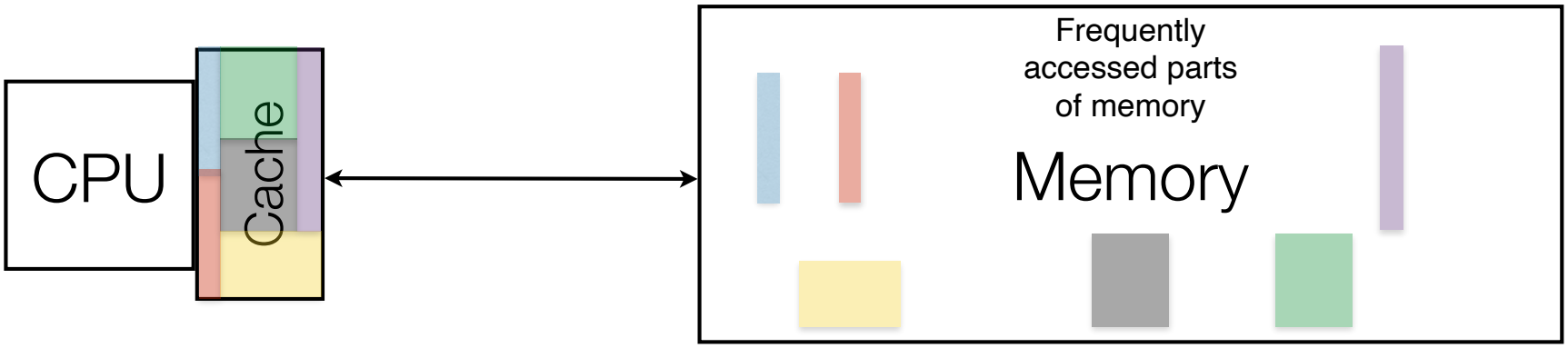


- Cache does not have separate addresses
  - CPU (and assembly code) still “thinks” it is accessing main memory
  - Cache acts as a temporary copy of memory
  - CPU can read from and write to cache in much less time than memory

Cache is faster but much smaller than main mem: can’t store everything, so try to be “smart” about what is stored in the cache to minimize fetches from / writes to main memory

What is a cache?

- Temporary copy of memory
  - CPU can read from and write to cache in much less time than memory

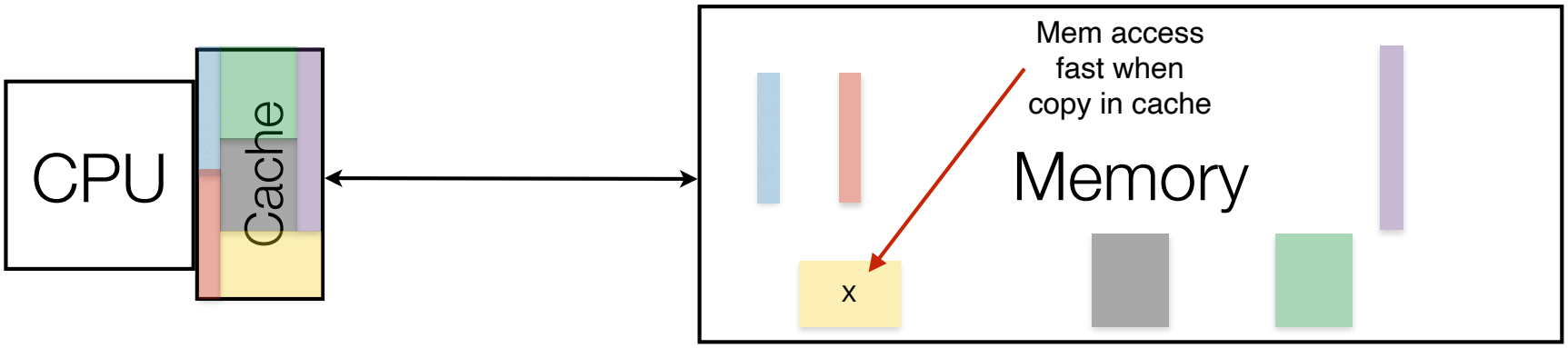


- Cache does not have separate addresses
  - CPU (and assembly code) still “thinks” it is accessing main memory
  - Cache acts as a temporary copy of memory
  - CPU can read from and write to cache in much less time than memory

Cache is faster but much smaller than main mem: can’t store everything, so try to be “smart” about what is stored in the cache to minimize fetches from / writes to main memory

What is a cache?

- Temporary copy of memory
  - CPU can read from and write to cache in much less time than memory

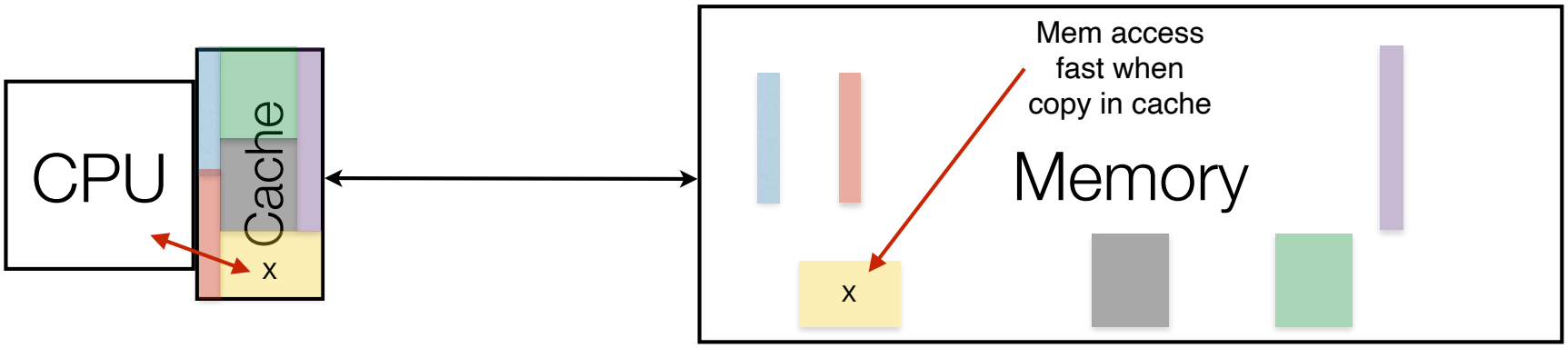


- Cache does not have separate addresses
  - CPU (and assembly code) still “thinks” it is accessing main memory
  - Cache acts as a temporary copy of memory
  - CPU can read from and write to cache in much less time than memory

Cache is faster but much smaller than main mem: can’t store everything, so try to be “smart” about what is stored in the cache to minimize fetches from / writes to main memory

What is a cache?

- Temporary copy of memory
  - CPU can read from and write to cache in much less time than memory

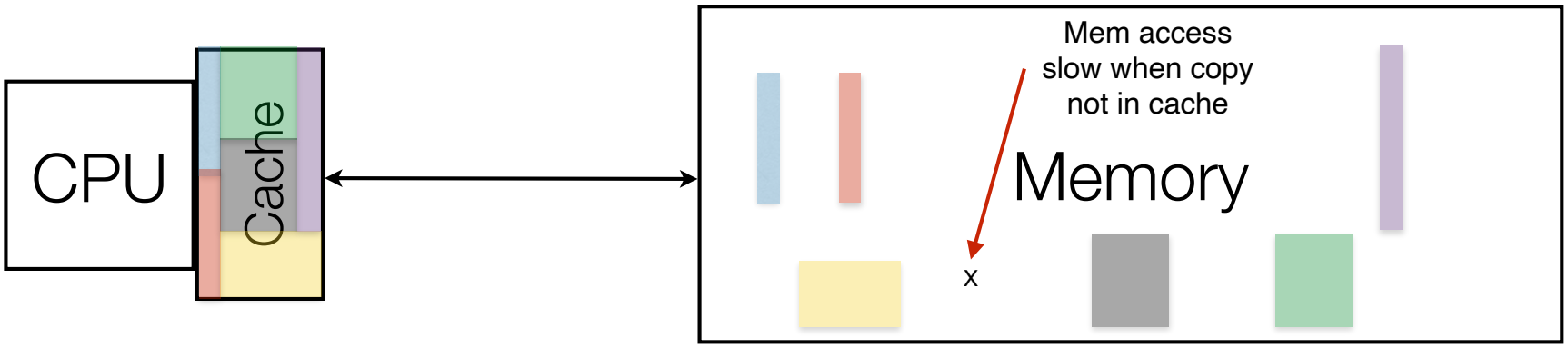


- Cache does not have separate addresses
  - CPU (and assembly code) still “thinks” it is accessing main memory
  - Cache acts as a temporary copy of memory
  - CPU can read from and write to cache in much less time than memory

Cache is faster but much smaller than main mem: can’t store everything, so try to be “smart” about what is stored in the cache to minimize fetches from / writes to main memory

What is a cache?

- Temporary copy of memory
  - CPU can read from and write to cache in much less time than memory

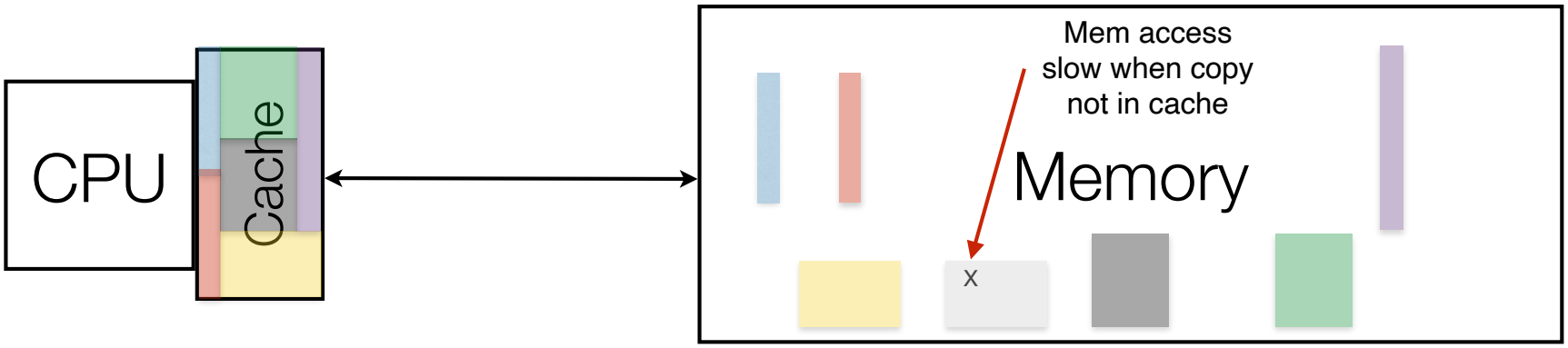


- Cache does not have separate addresses
  - CPU (and assembly code) still “thinks” it is accessing main memory
  - Cache acts as a temporary copy of memory
  - CPU can read from and write to cache in much less time than memory

Cache is faster but much smaller than main mem: can’t store everything, so try to be “smart” about what is stored in the cache to minimize fetches from / writes to main memory

What is a cache?

- Temporary copy of memory
  - CPU can read from and write to cache in much less time than memory

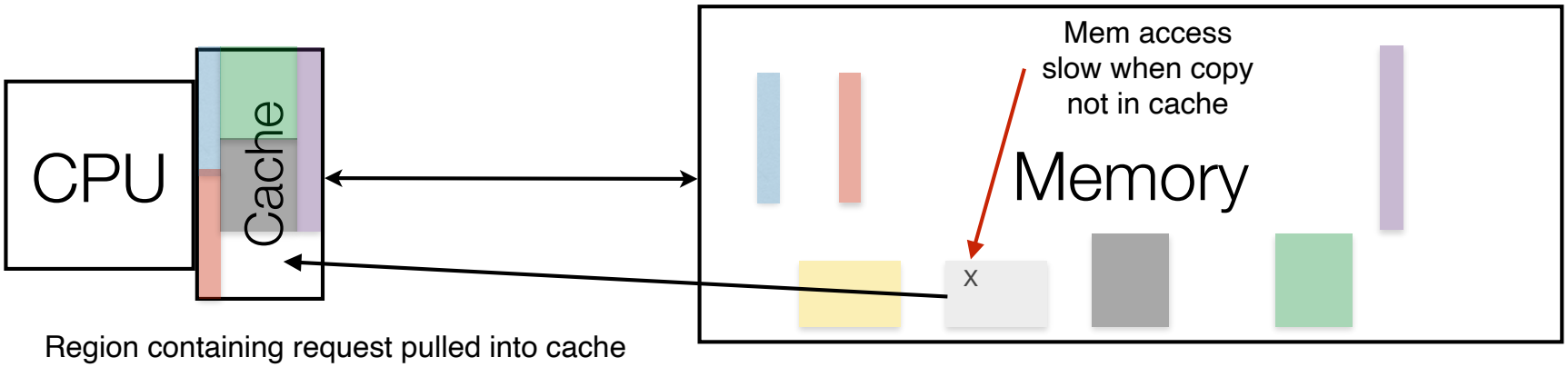


- Cache does not have separate addresses
  - CPU (and assembly code) still “thinks” it is accessing main memory
  - Cache acts as a temporary copy of memory
  - CPU can read from and write to cache in much less time than memory

Cache is faster but much smaller than main mem: can’t store everything, so try to be “smart” about what is stored in the cache to minimize fetches from / writes to main memory

What is a cache?

- Temporary copy of memory
  - CPU can read from and write to cache in much less time than memory

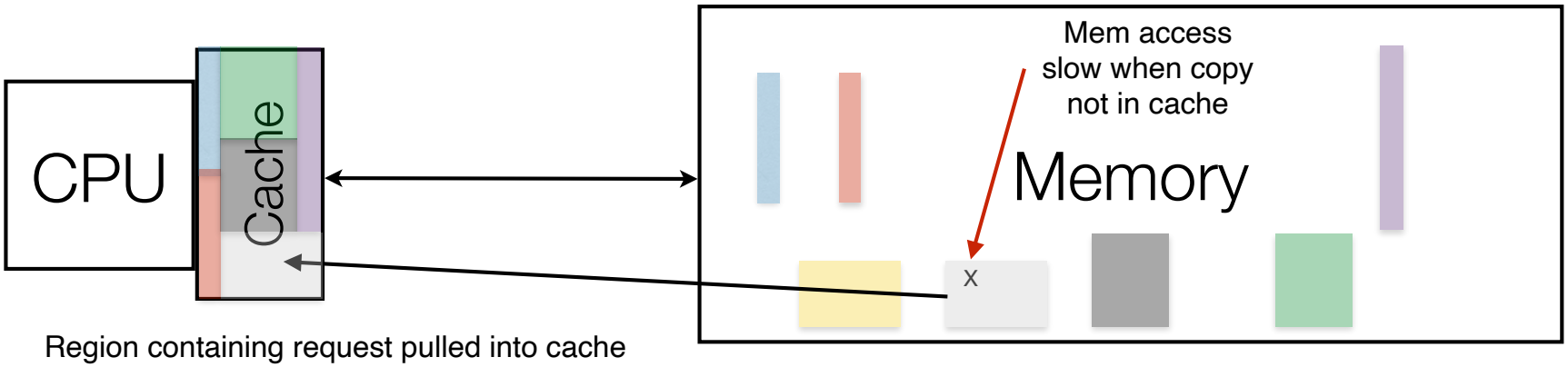


- Cache does not have separate addresses
  - CPU (and assembly code) still “thinks” it is accessing main memory
  - Cache acts as a temporary copy of memory
  - CPU can read from and write to cache in much less time than memory

Cache is faster but much smaller than main mem: can’t store everything, so try to be “smart” about what is stored in the cache to minimize fetches from / writes to main memory

What is a cache?

- Temporary copy of memory
  - CPU can read from and write to cache in much less time than memory

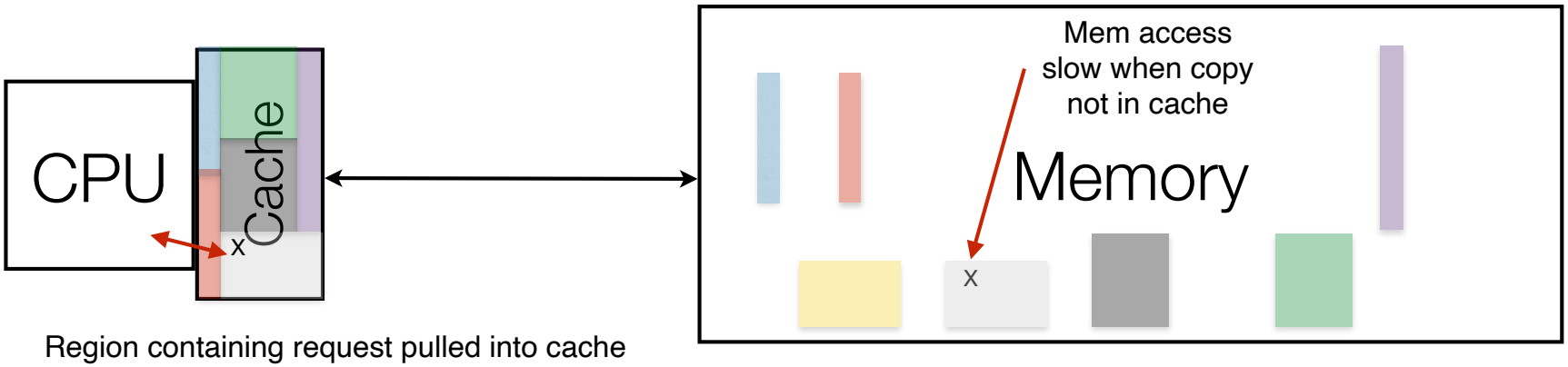


- Cache does not have separate addresses
  - CPU (and assembly code) still “thinks” it is accessing main memory
  - Cache acts as a temporary copy of memory
  - CPU can read from and write to cache in much less time than memory

Cache is faster but much smaller than main mem: can’t store everything, so try to be “smart” about what is stored in the cache to minimize fetches from / writes to main memory

What is a cache?

- Temporary copy of memory
  - CPU can read from and write to cache in much less time than memory



- Cache does not have separate addresses
  - CPU (and assembly code) still “thinks” it is accessing main memory
  - Cache acts as a temporary copy of memory
  - CPU can read from and write to cache in much less time than memory

Cache is faster but much smaller than main mem: can’t store everything, so try to be “smart” about what is stored in the cache to minimize fetches from / writes to main memory

Cache Locality (spatial & temporal)

Principle of Locality

Programs access a small proportion of their address space at any time

Temporal Locality:
Items accessed recently are likely to be accessed again soon
e.g., instructions in a loop, induction variables

Spatial locality:
Items near those accessed recently are likely to be accessed soon
E.g., sequential instruction access, array data

Spatial & Temporal Locality Example

```
        fact:
40000       addi $sp, $sp, -8       # adjust stack for 2 items
40004       sw   $ra, 4($sp)        # save return address
40008       sw   $a0, 0($sp)        # save argument
40012       slti $t0, $a0, 1        # test for n < 1
40016       beq  $t0, $zero, L1
40020       addi $v0, $zero, 1      # if so, result is 1
40024       addi $sp, $sp, 8        #   pop 2 items from stack
40028       jr   $ra                #   and return
40032   L1: addi $a0, $a0, -1       # else decrement n
40036       jal  fact               # recursive call
40040       lw   $a0, 0($sp)        # restore original n
40044       lw   $ra, 4($sp)        #   and return address
40048       addi $sp, $sp, 8        # pop 2 items from stack
40052       mul  $v0, $a0, $v0      # multiply to get result
40056       jr   $ra                # and return
```

- What memory addresses are going to be heavily utilized while fact(15) runs? (i.e., 15 iterations of the code to the left)

Spatial & Temporal Locality Example

```
        fact:
40000       addi $sp, $sp, -8       # adjust stack for 2 items
40004       sw   $ra, 4($sp)        # save return address
40008       sw   $a0, 0($sp)        # save argument
40012       slti $t0, $a0, 1        # test for n < 1
40016       beq  $t0, $zero, L1
40020       addi $v0, $zero, 1      # if so, result is 1
40024       addi $sp, $sp, 8        #   pop 2 items from stack
40028       jr   $ra                #   and return
40032   L1: addi $a0, $a0, -1       # else decrement n
40036       jal  fact               # recursive call
40040       lw   $a0, 0($sp)        # restore original n
40044       lw   $ra, 4($sp)        #   and return address
40048       addi $sp, $sp, 8        # pop 2 items from stack
40052       mul  $v0, $a0, $v0      # multiply to get result
40056       jr   $ra                # and return
```

- What memory addresses are going to be heavily utilized while fact(15) runs? (i.e., 15 iterations of the code to the left)

- addresses 40000-40059 (program code)

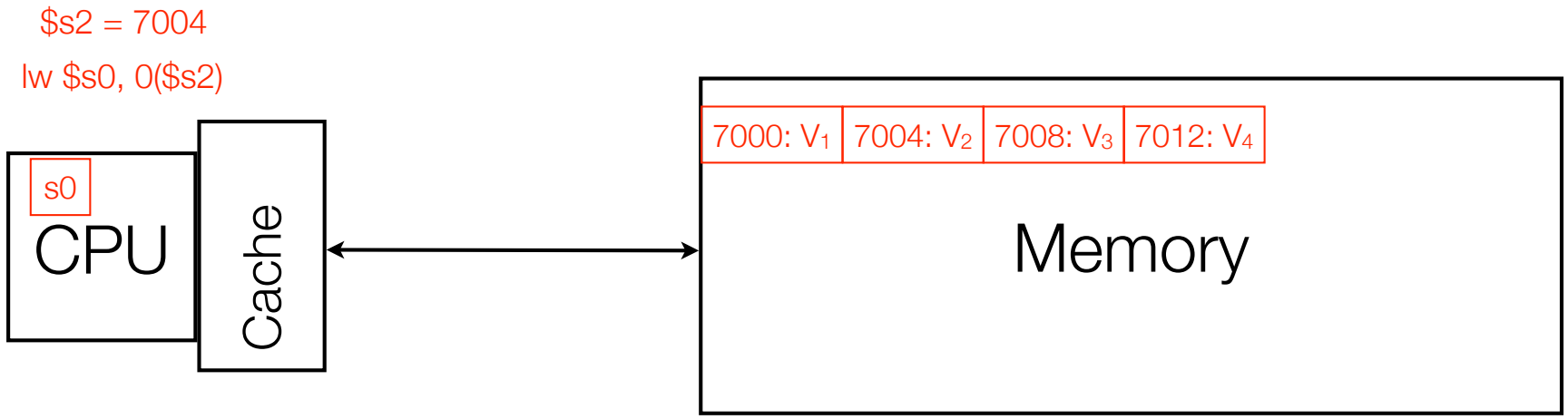
- addresses near current value of $sp (part of stack actively being used)

Blocks (in the cache)

Memory Hierarchy Levels

Block (aka line): fixed-size unit of copying between cache and memory, may be multiple words

e.g., 4 word block

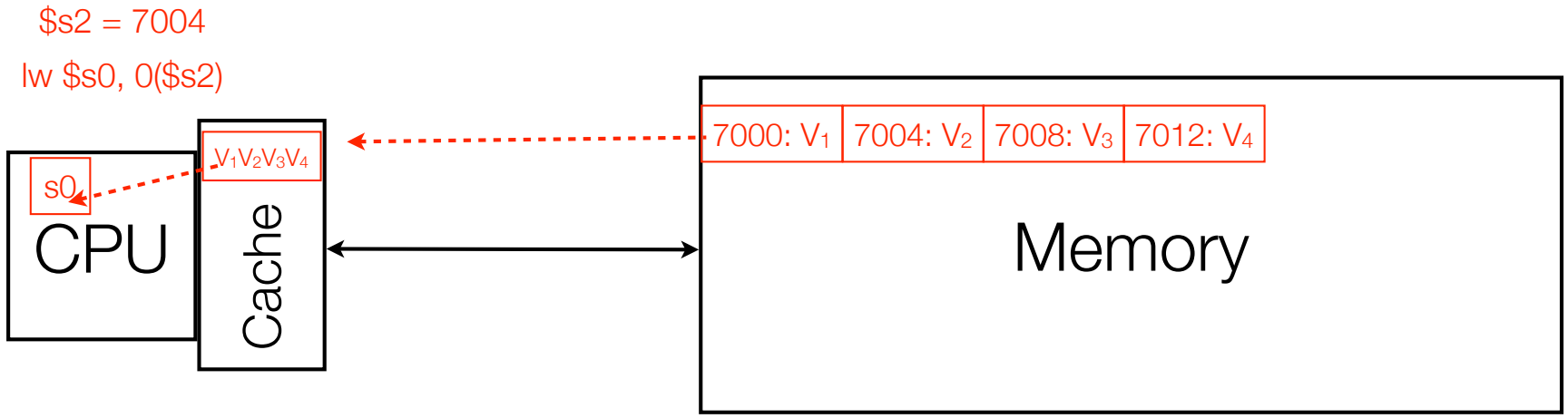


For k-word blocks, a cache miss results in a transfer of k words at once

Memory Hierarchy Levels

Block (aka line): fixed-size unit of copying between cache and memory, may be multiple words

e.g., 4 word block



For k-word blocks, a cache miss results in a transfer of k words at once

Block v. Word v. Byte

- Block is to Word in Caching as Word is to Byte in MIPS
  - In MIPS, (almost all) instructions/operations operate on data a word (4 bytes) at a time
    - Advantage: most operations use the entire word: faster/easier to fetch / move / compute on the 4 bytes simultaneously
  - In Caching, Words are moved between memory and cache a block at a time
    - Advantage: spatial locality: if a particular word from memory is being accessed by the CPU, there is a high likelihood that nearby words in the memory will be accessed soon as well.

Memory Addresses at the byte level

00000000	00000001	00000010	00000011	00000100	00000101	00000110	00000111
00001000	00001001	00001010	00001011	00001100	00001101	00001110	00001111
00010000	00010001	00010010	00010011	00010100	00010101	00010110	00010111
00011000	00011001	00011010	00011011	00011100	00011101	00011110	00011111
00100000	00100001	00100010	00100011	00100100	00100101	00100110	00100111
00101000	00101001	00101010	00101011	00101100	00101101	00101110	00101111
00110000	00110001	00110010	00110011	00110100	00110101	00110110	00110111
00111000	00111001	00111010	00111011	00111100	00111101	00111110	00111111
01000000	01000001	01000010	01000011	01000100	01000101	01000110	01000111
01001000	01001001	01001010	01001011	01001100	01001101	01001110	01001111
01010000	01010001	01010010	01010011	01010100	01010101	01010110	01010111
01011000	01011001	01011010	01011011	01011100	01011101	01011110	01011111
01100000	01100001	01100010	01100011	01100100	01100101	01100110	01100111
01101000	01101001	01101010	01101011	01101100	01101101	01101110	01101111
01110000	01110001	01110010	01110011	01110100	01110101	01110110	01110111
01111000	01111001	01111010	01111011	01111100	01111101	01111110	01111111
10000000	10000001	10000010	10000011	10000100	10000101	10000110	10000111
10001000	10001001	10001010	10001011	10001100	10001101	10001110	10001111
10010000	10010001	10010010	10010011	10010100	10010101	10010110	10010111
10011000	10011001	10011010	10011011	10011100	10011101	10011110	10011111
10100000	10100001	10100010	10100011	10100100	10100101	10100110	10100111
10101000	10101001	10101010	10101011	10101100	10101101	10101110	10101111
10110000	10110001	10110010	10110011	10110100	10110101	10110110	10110111
10111000	10111001	10111010	10111011	10111100	10111101	10111110	10111111
11000000	11000001	11000010	11000011	11000100	11000101	11000110	11000111
11001000	11001001	11001010	11001011	11001100	11001101	11001110	11001111
11010000	11010001	11010010	11010011	11010100	11010101	11010110	11010111
11011000	11011001	11011010	11011011	11011100	11011101	11011110	11011111
11100000	11100001	11100010	11100011	11100100	11100101	11100110	11100111
11101000	11101001	11101010	11101011	11101100	11101101	11101110	11101111
11110000	11110001	11110010	11110011	11110100	11110101	11110110	11110111
11111000	11111001	11111010	11111011	11111100	11111101	11111110	11111111

Byte addresses (8 bit addresses in this example)

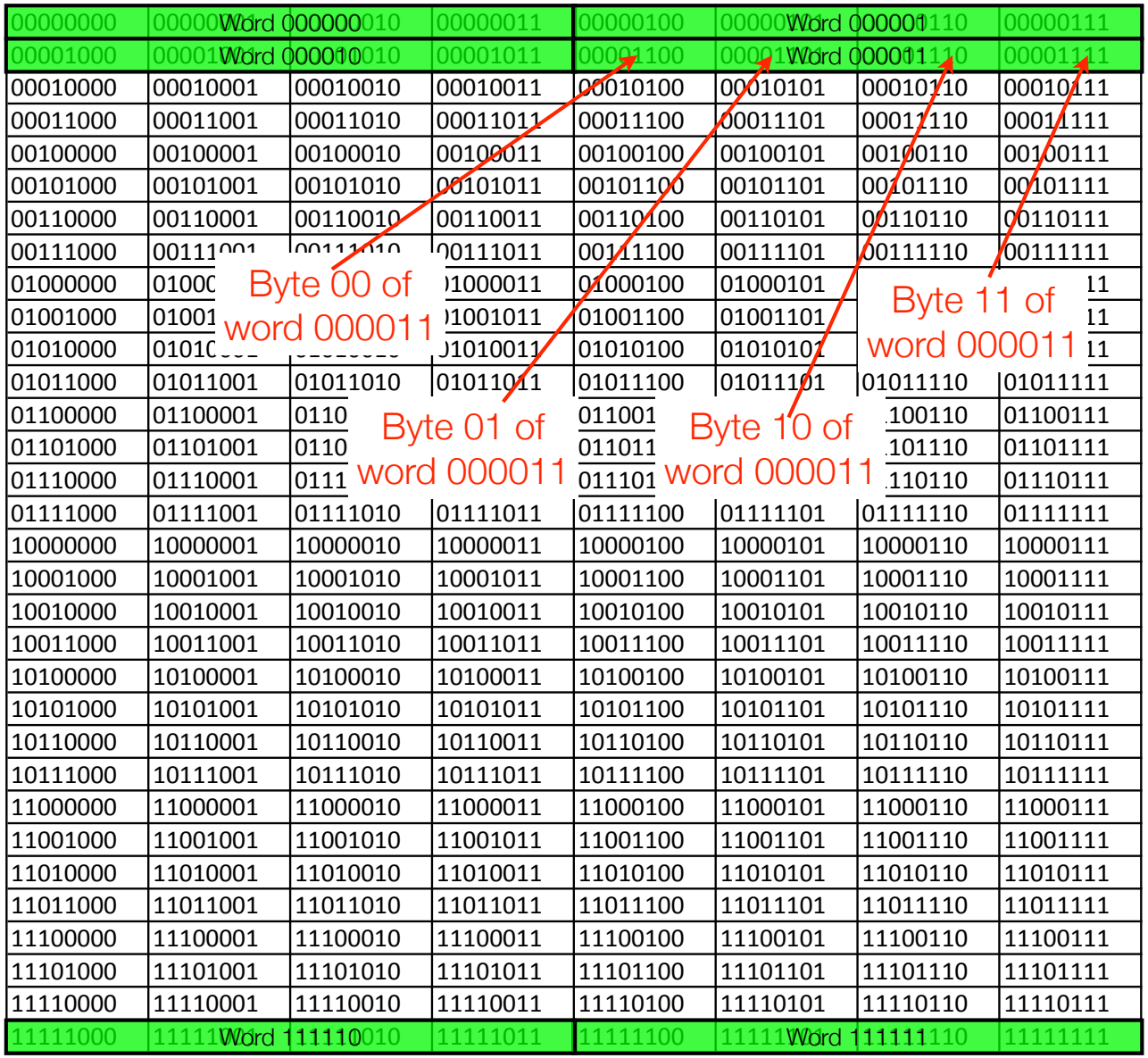
Memory Addresses at the Word level

00000000	00000001	Word 000000 00010	00000011	00000100	00000101	Word 000001 1110	00000111
00001000	00001001	Word 000010 0010	00001011	00001100	00001101	Word 000011 1110	00001111
00010000	00010001	00010010	00010011	00010100	00010101	00010110	00010111
00011000	00011001	00011010	00011011	00011100	00011101	00011110	00011111
00100000	00100001	00100010	00100011	00100100	00100101	00100110	00100111
00101000	00101001	00101010	00101011	00101100	00101101	00101110	00101111
00110000	00110001	00110010	00110011	00110100	00110101	00110110	00110111
00111000	00111001	00111010	00111011	00111100	00111101	00111110	00111111
01000000	01000001	01000010	01000011	01000100	01000101	01000110	01000111
01001000	01001001	01001010	01001011	01001100	01001101	01001110	01001111
01010000	01010001	01010010	01010011	01010100	01010101	01010110	01010111
01011000	01011001	01011010	01011011	01011100	01011101	01011110	01011111
01100000	01100001	01100010	01100011	01100100	01100101	01100110	01100111
01101000	01101001	01101010	01101011	01101100	01101101	01101110	01101111
01110000	01110001	01110010	01110011	01110100	01110101	01110110	01110111
01111000	01111001	01111010	01111011	01111100	01111101	01111110	01111111
10000000	10000001	10000010	10000011	10000100	10000101	10000110	10000111
10001000	10001001	10001010	10001011	10001100	10001101	10001110	10001111
10010000	10010001	10010010	10010011	10010100	10010101	10010110	10010111
10011000	10011001	10011010	10011011	10011100	10011101	10011110	10011111
10100000	10100001	10100010	10100011	10100100	10100101	10100110	10100111
10101000	10101001	10101010	10101011	10101100	10101101	10101110	10101111
10110000	10110001	10110010	10110011	10110100	10110101	10110110	10110111
10111000	10111001	10111010	10111011	10111100	10111101	10111110	10111111
11000000	11000001	11000010	11000011	11000100	11000101	11000110	11000111
11001000	11001001	11001010	11001011	11001100	11001101	11001110	11001111
11010000	11010001	11010010	11010011	11010100	11010101	11010110	11010111
11011000	11011001	11011010	11011011	11011100	11011101	11011110	11011111
11100000	11100001	11100010	11100011	11100100	11100101	11100110	11100111
11101000	11101001	11101010	11101011	11101100	11101101	11101110	11101111
11110000	11110001	11110010	11110011	11110100	11110101	11110110	11110111
11111000	11111001	Word 111110 10	11111011	11111100	11111101	Word 111111 10	11111111

Word "addresses" (6 bits in example):

Word $A_5A_4A_3A_2A_1A_0$ formed from bytes with addresses $A_5A_4A_3A_2A_1A_0XX$

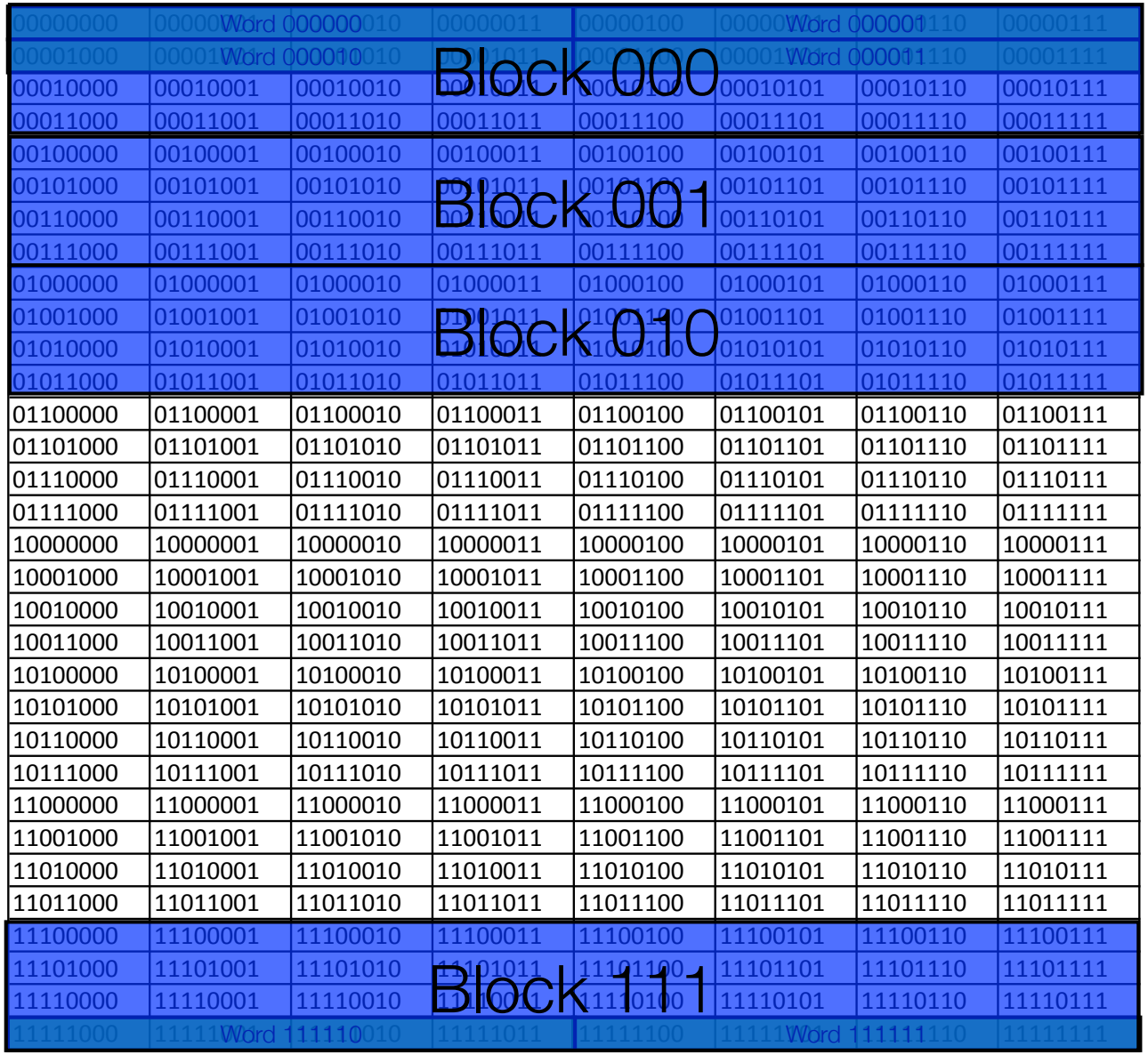
Memory Addresses at the Word level



Word "addresses" (6 bits in example):

Word $A_5A_4A_3A_2A_1A_0$ formed from bytes with addresses $A_5A_4A_3A_2A_1A_0XX$

Memory Addresses at the Block level for n=3

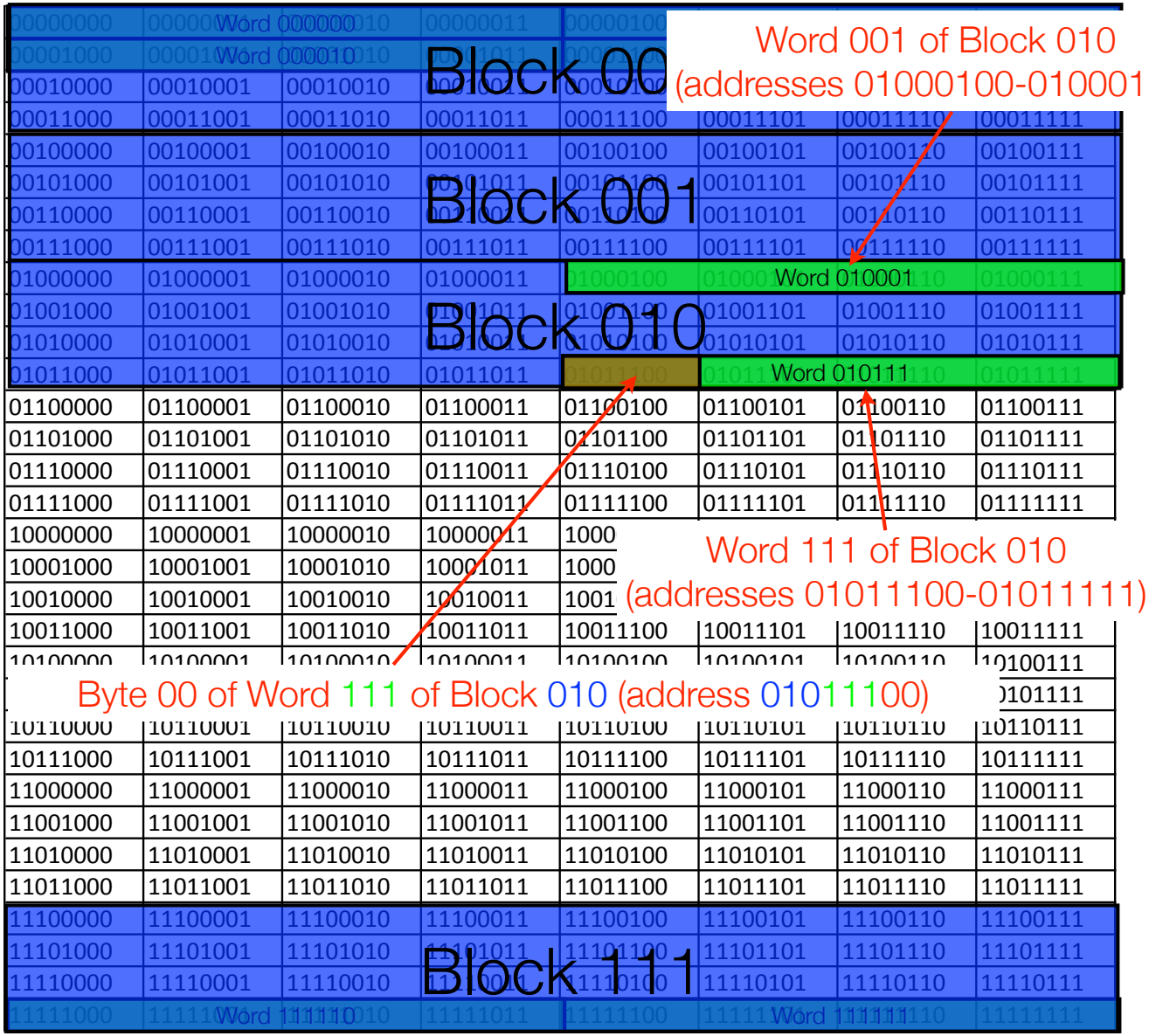


00000000	00000001	00000010	00000011	00000100	00000101	00000110	00000111
00001000	00001001	00001010	00001011	00001100	00001101	00001110	00001111
00010000	00010001	00010010	00010011	00010100	00010101	00010110	00010111
00011000	00011001	00011010	00011011	00011100	00011101	00011110	00011111
00100000	00100001	00100010	00100011	00100100	00100101	00100110	00100111
00101000	00101001	00101010	00101011	00101100	00101101	00101110	00101111
00110000	00110001	00110010	00110011	00110100	00110101	00110110	00110111
00111000	00111001	00111010	00111011	00111100	00111101	00111110	00111111
01000000	01000001	01000010	01000011	01000100	01000101	01000110	01000111
01001000	01001001	01001010	01001011	01001100	01001101	01001110	01001111
01010000	01010001	01010010	01010011	01010100	01010101	01010110	01010111
01011000	01011001	01011010	01011011	01011100	01011101	01011110	01011111
01100000	01100001	01100010	01100011	01100100	01100101	01100110	01100111
01101000	01101001	01101010	01101011	01101100	01101101	01101110	01101111
01110000	01110001	01110010	01110011	01110100	01110101	01110110	01110111
01111000	01111001	01111010	01111011	01111100	01111101	01111110	01111111
10000000	10000001	10000010	10000011	10000100	10000101	10000110	10000111
10001000	10001001	10001010	10001011	10001100	10001101	10001110	10001111
10010000	10010001	10010010	10010011	10010100	10010101	10010110	10010111
10011000	10011001	10011010	10011011	10011100	10011101	10011110	10011111
10100000	10100001	10100010	10100011	10100100	10100101	10100110	10100111
10101000	10101001	10101010	10101011	10101100	10101101	10101110	10101111
10110000	10110001	10110010	10110011	10110100	10110101	10110110	10110111
10111000	10111001	10111010	10111011	10111100	10111101	10111110	10111111
11000000	11000001	11000010	11000011	11000100	11000101	11000110	11000111
11001000	11001001	11001010	11001011	11001100	11001101	11001110	11001111
11010000	11010001	11010010	11010011	11010100	11010101	11010110	11010111
11011000	11011001	11011010	11011011	11011100	11011101	11011110	11011111
11100000	11100001	11100010	11100011	11100100	11100101	11100110	11100111
11101000	11101001	11101010	11101011	11101100	11101101	11101110	11101111
11110000	11110001	11110010	11110011	11110100	11110101	11110110	11110111
11111000	11111001	11111010	11111011	11111100	11111101	11111110	11111111

Block addresses (n=3 bits in example):

All words (and bytes) of memory whose addresses start with the same prefix (3 bit prefix in this example)

Memory Addresses at the Block level for n=3



Block addresses (n=3 bits in example):

Block $B_2B_1B_0$ formed from byte addresses $B_2B_1B_0XXXXX$ (or words with addresses $B_2B_1B_0XXX$)
i.e., addresses where $B_2B_1B_0$ is the prefix

In example, each block holds 2^5=32 bytes of memory (8 words)

Reading from Cache
Hits and Misses
(and Stalls and Fetches)

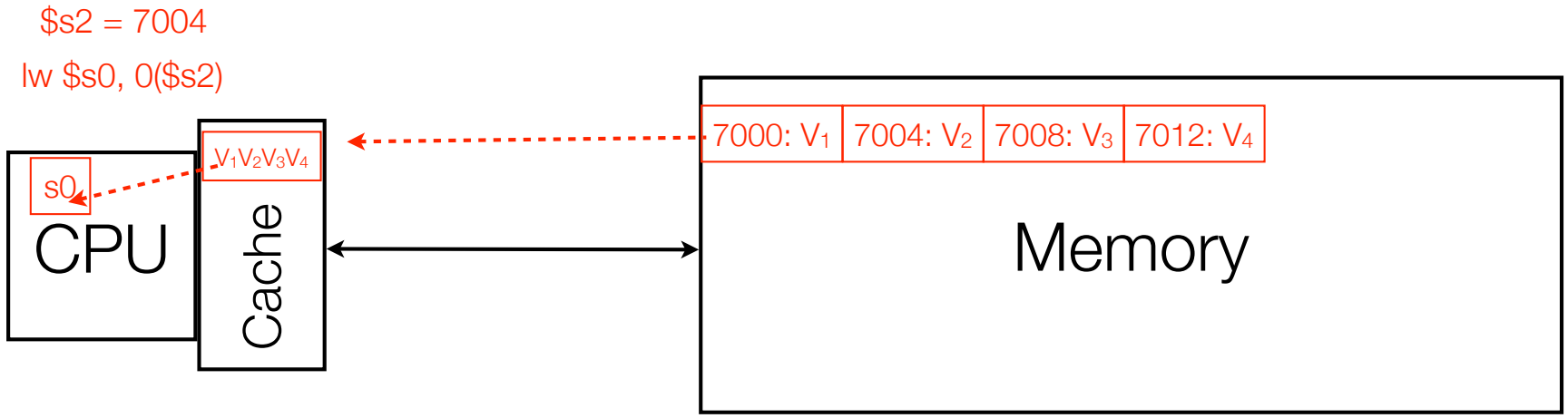
Cache Hits and Misses

→ If requested address's data already present in cache

- **Cache Hit: access satisfied**
- **Hit ratio:** hits/accesses

→ If requested address's data not yet in cache

- **Cache Miss: Entire block** is copied from memory to cache
- **Miss penalty:** time required to copy the block from memory to the cache
- **Miss ratio:** misses/accesses



For k-word blocks, a cache miss results in a transfer of k words at once

Handling Cache Misses

- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy (in our context, main memory)
  - After Instruction cache miss, restart instruction fetch
  - After data cache miss, complete data access

Cache Challenges

- **When** to move data from memory to cache?
    - Small blocksize?
        - Less “wasted” moves
        - Don’t take advantage of parallelism (many words movable at same time) - separate moves are more costly timewise
- **Where** in cache to move fetched memory?
    - Lots of options: least recently used, fixed mapping, will explore pros and cons
- **Overall Goal**: minimize cache misses (maximize cache hits) without (spending too much time) fetching unused data

Cache Mapping Strategy #1: Direct Mapping

Block “address” in memory v. cache

- A block lives in memory at a (set of) “addresses”, but may also temporarily be copied in the cache
- In **direct mapping**, the block has a **fixed location** in the cache to which it can be copied (we call it the block’s cache address).
- In picture below, block address is shown in 4x4 grid (16 blocks of memory)
- Cache only has 4 slots, so multiple memory blocks must map to same cache block. How to map?

Qu: How to map memory block to cache block?

4x4 blocks of memory

Memory (shown in Blocks)

0000 $Data_A$	0001 $Data_B$	0010 $Data_C$	0011 $Data_D$
0100 $Data_E$	0101 $Data_F$	0110 $Data_G$	0111 $Data_H$
1000 $Data_I$	1001 $Data_J$	1010 $Data_K$	1011 $Data_L$
1100 $Data_M$	1101 $Data_N$	1110 $Data_O$	1111 $Data_P$

00	01	10	11

Cache

4x1 blocks of cache

Block “address” in memory v. cache

- A block lives in memory at a (set of) “addresses”, but may also temporarily be copied in the cache
- In **direct mapping**, the block has a **fixed location** in the cache to which it can be copied (we call it the block’s cache address).
- In picture below, block address is shown in 4x4 grid (16 blocks of memory)
- Cache only has 4 slots, so multiple memory blocks must map to same cache block. How to map?

4x4 blocks of memory

Memory (shown in Blocks)

0000 $Data_A$	0001 $Data_B$	0010 $Data_C$	0011 $Data_D$
0100 $Data_E$	0101 $Data_F$	0110 $Data_G$	0111 $Data_H$
1000 $Data_I$	1001 $Data_J$	1010 $Data_K$	1011 $Data_L$
1100 $Data_M$	1101 $Data_N$	1110 $Data_O$	1111 $Data_P$

00	01	10	11

Cache

4x1 blocks of cache

Example Mapping: map memory block $B_3B_2B_1B_0$ to cache block B_1B_0

Block “address” in memory v. cache

- A block lives in memory at a (set of) “addresses”, but may also temporarily be copied in the cache
- In **direct mapping**, the block has a **fixed location** in the cache to which it can be copied (we call it the block’s cache address).
- In picture below, block address is shown in 4x4 grid (16 blocks of memory)
- Cache only has 4 slots, so multiple memory blocks must map to same cache block. How to map?

4x4 blocks of memory

Memory (shown in Blocks)

0000 $Data_A$	0001 $Data_B$	0010 $Data_C$	0011 $Data_D$
0100 $Data_E$	0101 $Data_F$	0110 $Data_G$	0111 $Data_H$
1000 $Data_I$	1001 $Data_J$	1010 $Data_K$	1011 $Data_L$
1100 $Data_M$	1101 $Data_N$	1110 $Data_O$	1111 $Data_P$

00	01	10	11

Cache

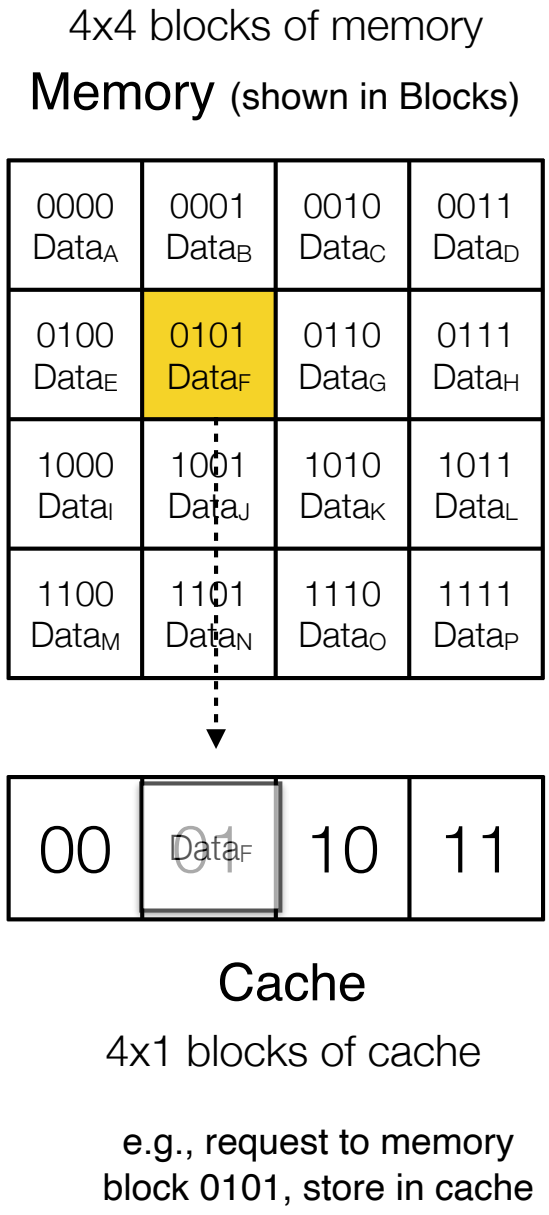
4x1 blocks of cache

e.g., request to memory block 0101, store in cache

Example Mapping: map memory block $B_3B_2B_1B_0$ to cache block B_1B_0

Block “address” in memory v. cache

- A block lives in memory at a (set of) “addresses”, but may also temporarily be copied in the cache
- In **direct mapping**, the block has a **fixed location** in the cache to which it can be copied (we call it the block’s cache address).
- In picture below, block address is shown in 4x4 grid (16 blocks of memory)
- Cache only has 4 slots, so multiple memory blocks must map to same cache block. How to map?



Example Mapping: map memory block $B_3B_2B_1B_0$ to cache block B_1B_0

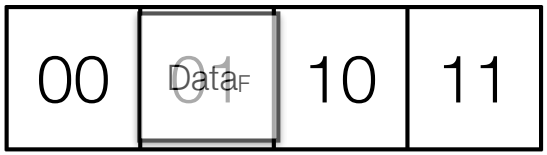
Block “address” in memory v. cache

- A block lives in memory at a (set of) “addresses”, but may also temporarily be copied in the cache
- In **direct mapping**, the block has a **fixed location** in the cache to which it can be copied (we call it the block’s cache address).
- In picture below, block address is shown in 4x4 grid (16 blocks of memory)
- Cache only has 4 slots, so multiple memory blocks must map to same cache block. How to map?

4x4 blocks of memory

Memory (shown in Blocks)

0000 $Data_A$	0001 $Data_B$	0010 $Data_C$	0011 $Data_D$
0100 $Data_E$	0101 $Data_F$	0110 $Data_G$	0111 $Data_H$
1000 $Data_I$	1001 $Data_J$	1010 $Data_K$	1011 $Data_L$
1100 $Data_M$	1101 $Data_N$	1110 $Data_O$	1111 $Data_P$



Cache

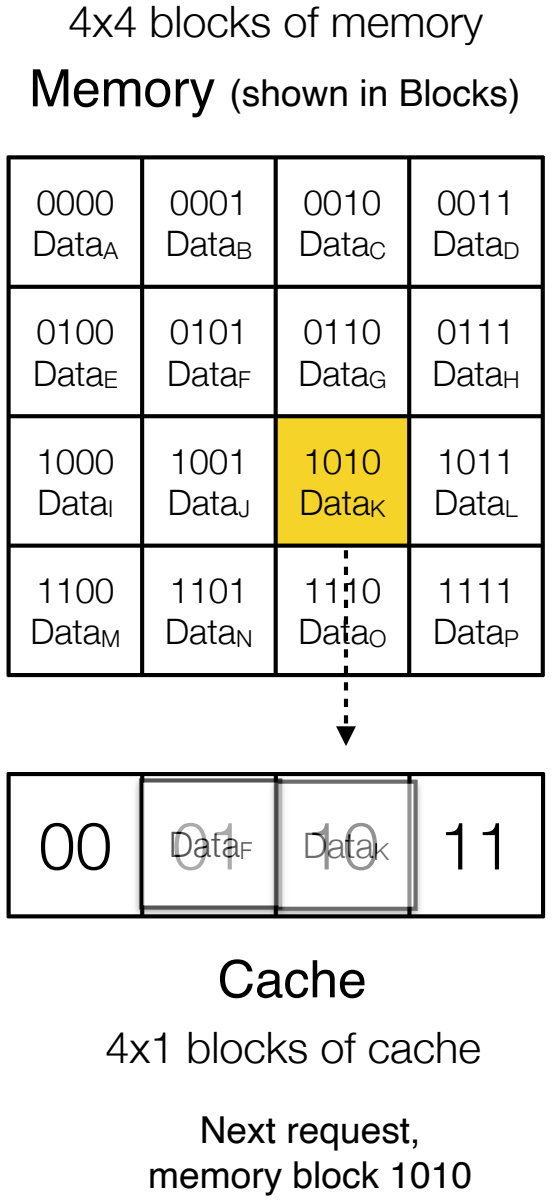
4x1 blocks of cache

Next request, memory block 1010

Example Mapping: map memory block $B_3B_2B_1B_0$ to cache block B_1B_0

Block “address” in memory v. cache

- A block lives in memory at a (set of) “addresses”, but may also temporarily be copied in the cache
- In **direct mapping**, the block has a **fixed location** in the cache to which it can be copied (we call it the block’s cache address).
- In picture below, block address is shown in 4x4 grid (16 blocks of memory)
- Cache only has 4 slots, so multiple memory blocks must map to same cache block. How to map?



Example Mapping: map memory block $B_3B_2B_1B_0$ to cache block B_1B_0

Block “address” in memory v. cache

- A block lives in memory at a (set of) “addresses”, but may also temporarily be copied in the cache
- In **direct mapping**, the block has a **fixed location** in the cache to which it can be copied (we call it the block’s cache address).
- In picture below, block address is shown in 4x4 grid (16 blocks of memory)
- Cache only has 4 slots, so multiple memory blocks must map to same cache block. How to map?

Example Mapping: map memory block $B_3B_2B_1B_0$ to cache block B_1B_0

4x4 blocks of memory

Memory (shown in Blocks)

0000 $Data_A$	0001 $Data_B$	0010 $Data_C$	0011 $Data_D$
0100 $Data_E$	0101 $Data_F$	0110 $Data_G$	0111 $Data_H$
1000 $Data_I$	1001 $Data_J$	1010 $Data_K$	1011 $Data_L$
1100 $Data_M$	1101 $Data_N$	1110 $Data_O$	1111 $Data_P$

00	01 $Data_F$	10 $Data_K$	11

Cache

4x1 blocks of cache

Next request, memory block 0010

Block “address” in memory v. cache

- A block lives in memory at a (set of) “addresses”, but may also temporarily be copied in the cache
- In **direct mapping**, the block has a **fixed location** in the cache to which it can be copied (we call it the block’s cache address).
- In picture below, block address is shown in 4x4 grid (16 blocks of memory)
- Cache only has 4 slots, so multiple memory blocks must map to same cache block. How to map?

4x4 blocks of memory

Memory (shown in Blocks)

0000 $Data_A$	0001 $Data_B$	0010 $Data_C$	0011 $Data_D$
0100 $Data_E$	0101 $Data_F$	0110 $Data_G$	0111 $Data_H$
1000 $Data_I$	1001 $Data_J$	1010 $Data_K$	1011 $Data_L$
1100 $Data_M$	1101 $Data_N$	1110 $Data_O$	1111 $Data_P$

00	01 $Data_F$	10 $Data_K$	11

Cache

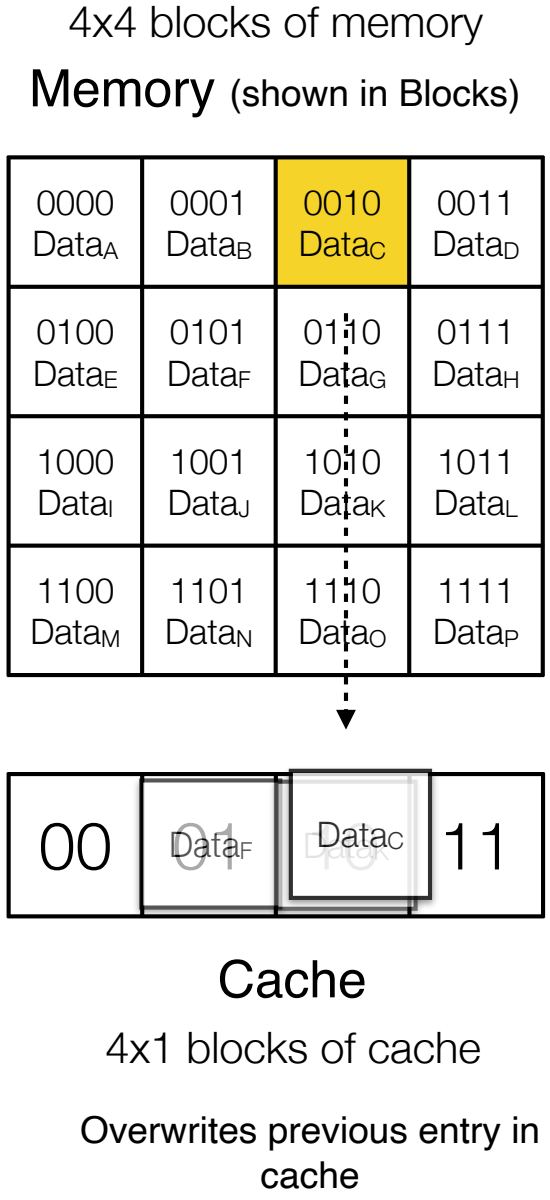
4x1 blocks of cache

Next request, memory block 0010

Example Mapping: map memory block $B_3B_2B_1B_0$ to cache block B_1B_0

Block “address” in memory v. cache

- A block lives in memory at a (set of) “addresses”, but may also temporarily be copied in the cache
- In **direct mapping**, the block has a **fixed location** in the cache to which it can be copied (we call it the block’s cache address).
- In picture below, block address is shown in 4x4 grid (16 blocks of memory)
- Cache only has 4 slots, so multiple memory blocks must map to same cache block. How to map?

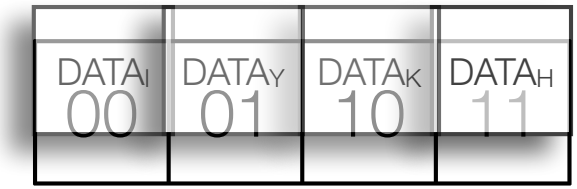


Example Mapping: map memory block $B_3B_2B_1B_0$ to cache block B_1B_0

Tags and Valid Bits

Memory (shown in Blocks)

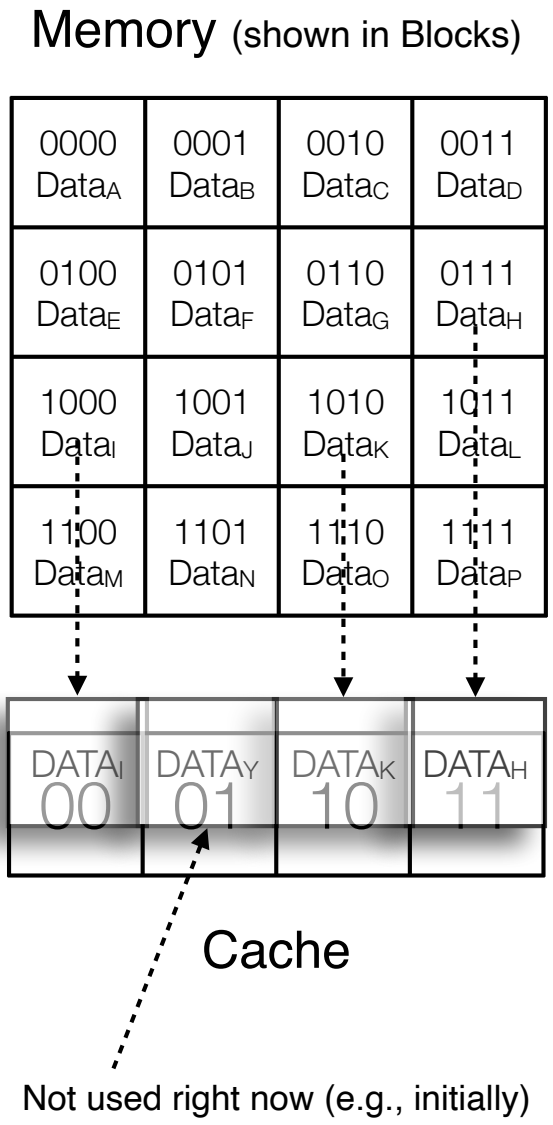
0000	0001	0010	0011
0100	0101	0110	0111
1000	1001	1010	1011
1100	1101	1110	1111



Cache

- Q: Since block data copied to cache, not address info, how do we know which particular mem block is stored in a cache location?
- i.e., which row is the data copied from?

Tags and Valid Bits

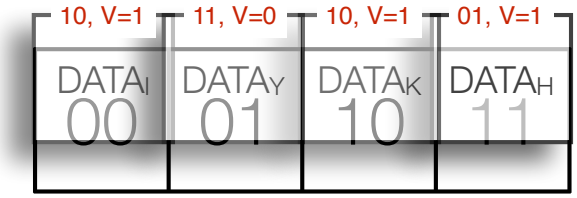


- Q: Since block data copied to cache, not address info, how do we know which particular mem block is stored in a cache location?
- i.e., which row is the data copied from?

Tags and Valid Bits

Memory (shown in Blocks)

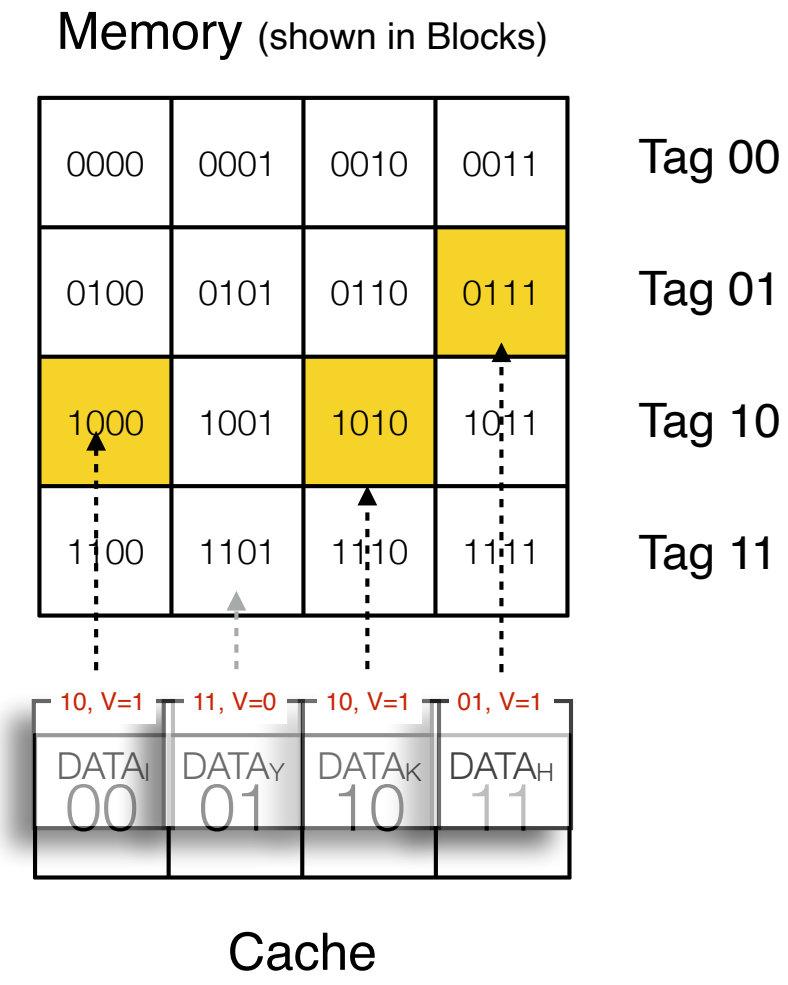
0000	0001	0010	0011
0100	0101	0110	0111
1000	1001	1010	1011
1100	1101	1110	1111



Cache

- Q: Since block data copied to cache, not address info, how do we know which particular mem block is stored in a cache location?
- i.e., which row is the data copied from?
  - Each active block marked with a tag
  - Mem block = tag . cache block
- What if there is no data in a cache location?
  - Valid bit: 1 = present, 0 = not present
  - Initially 0

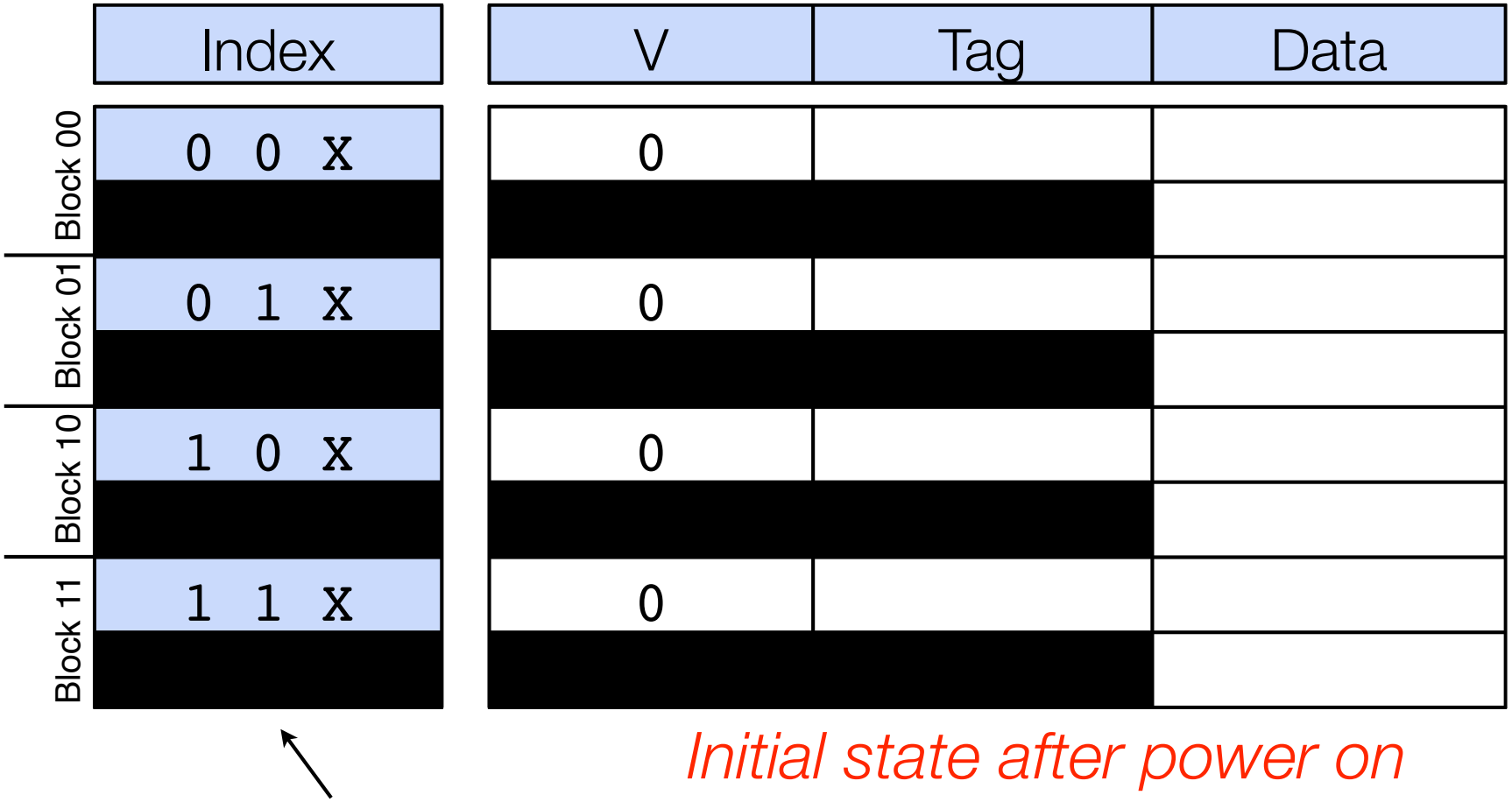
Tags and Valid Bits



- Q: Since block data copied to cache, not address info, how do we know which particular mem block is stored in a cache location?
- i.e., which row is the data copied from?
    - Each active block marked with a tag
    - Mem block = tag . cache block
- What if there is no data in a cache location?
    - Valid bit: 1 = present, 0 = not present
    - Initially 0

Cache Example

4-block cache, 2 words/block, 32B memory

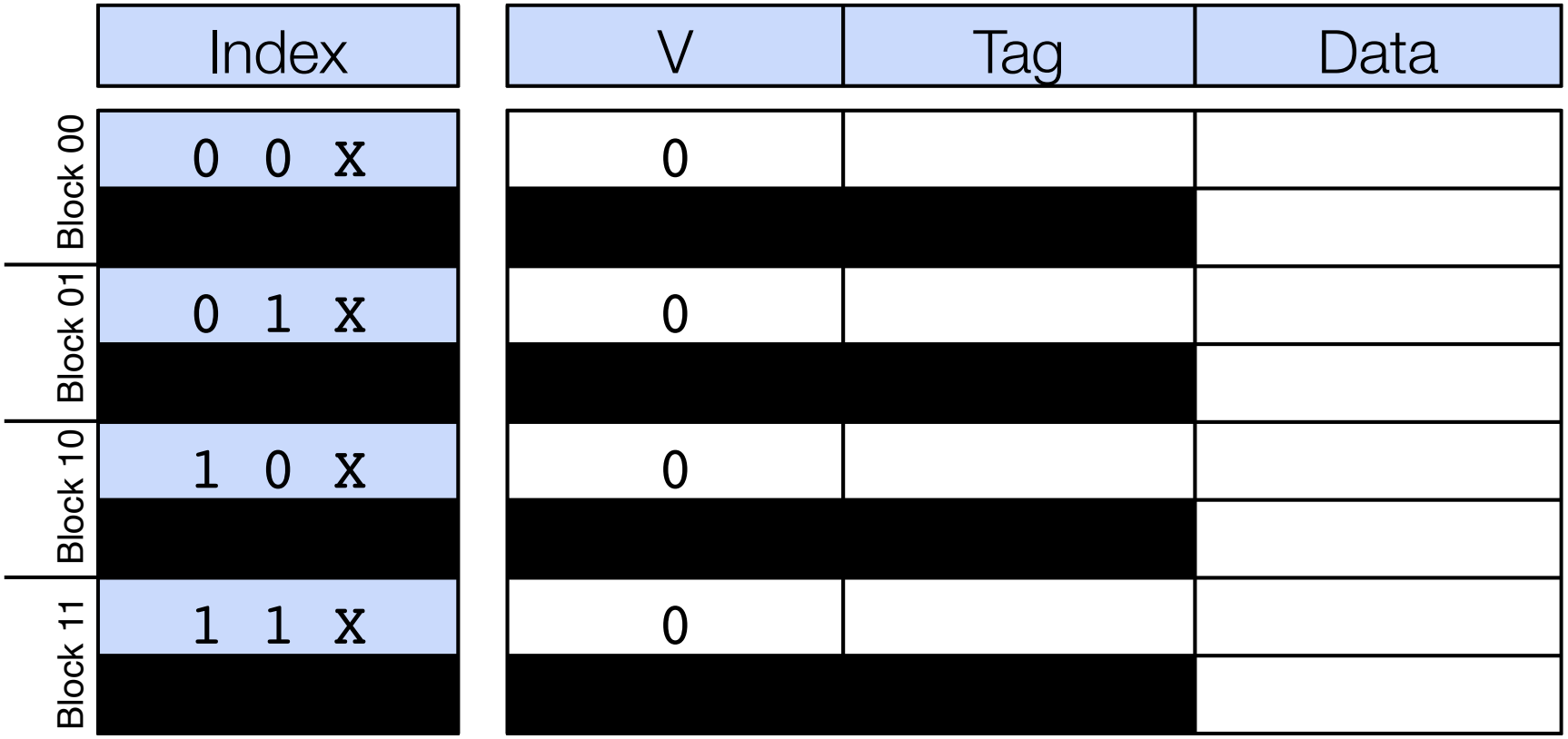


Initial state after power on

Index indicates where addresses ending with specific 3-bit sequence should be stored in cache

Cache Example

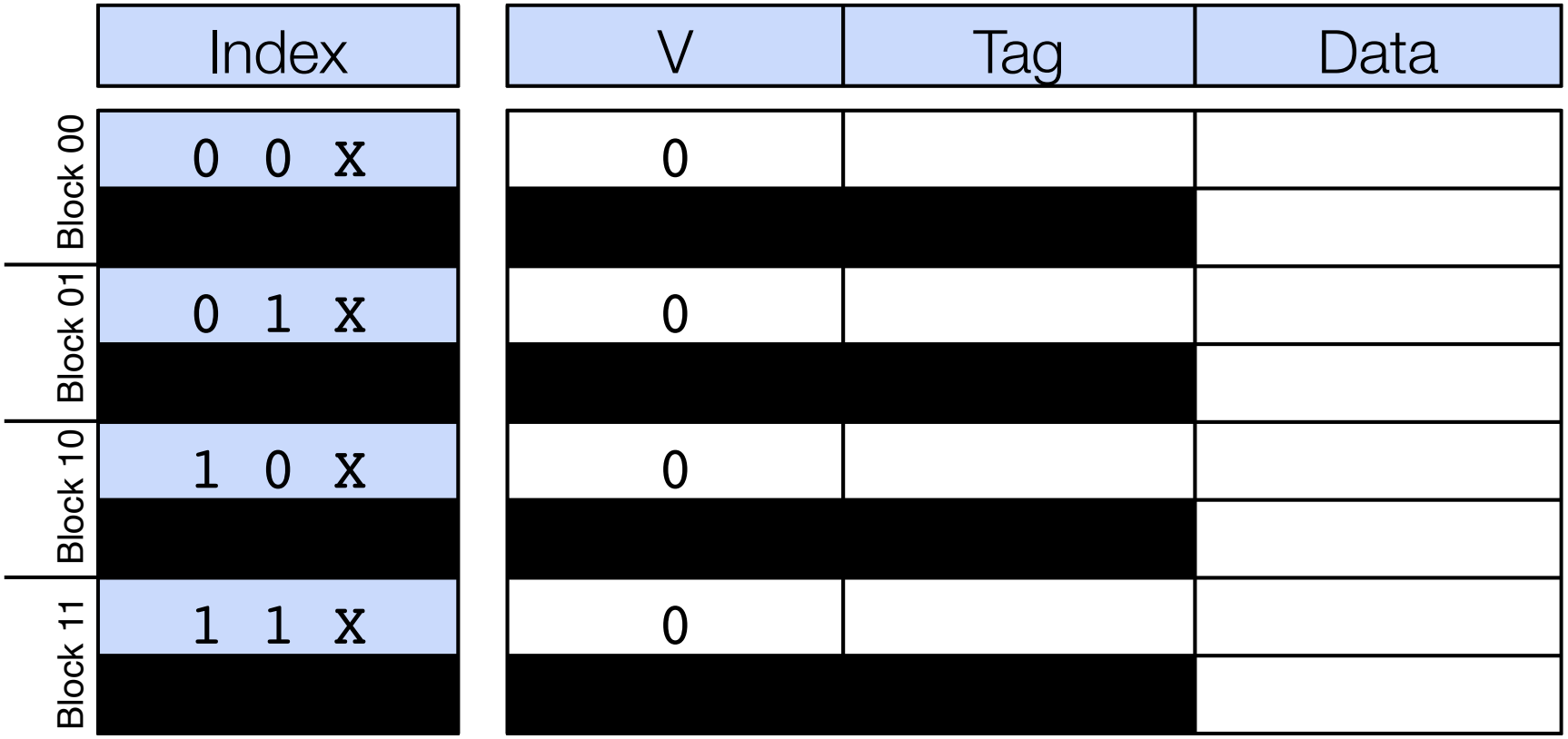
4-block cache, 2 words/block, 32B memory



request address 10110 (miss)

Cache Example

4-block cache, 2 words/block, 32B memory

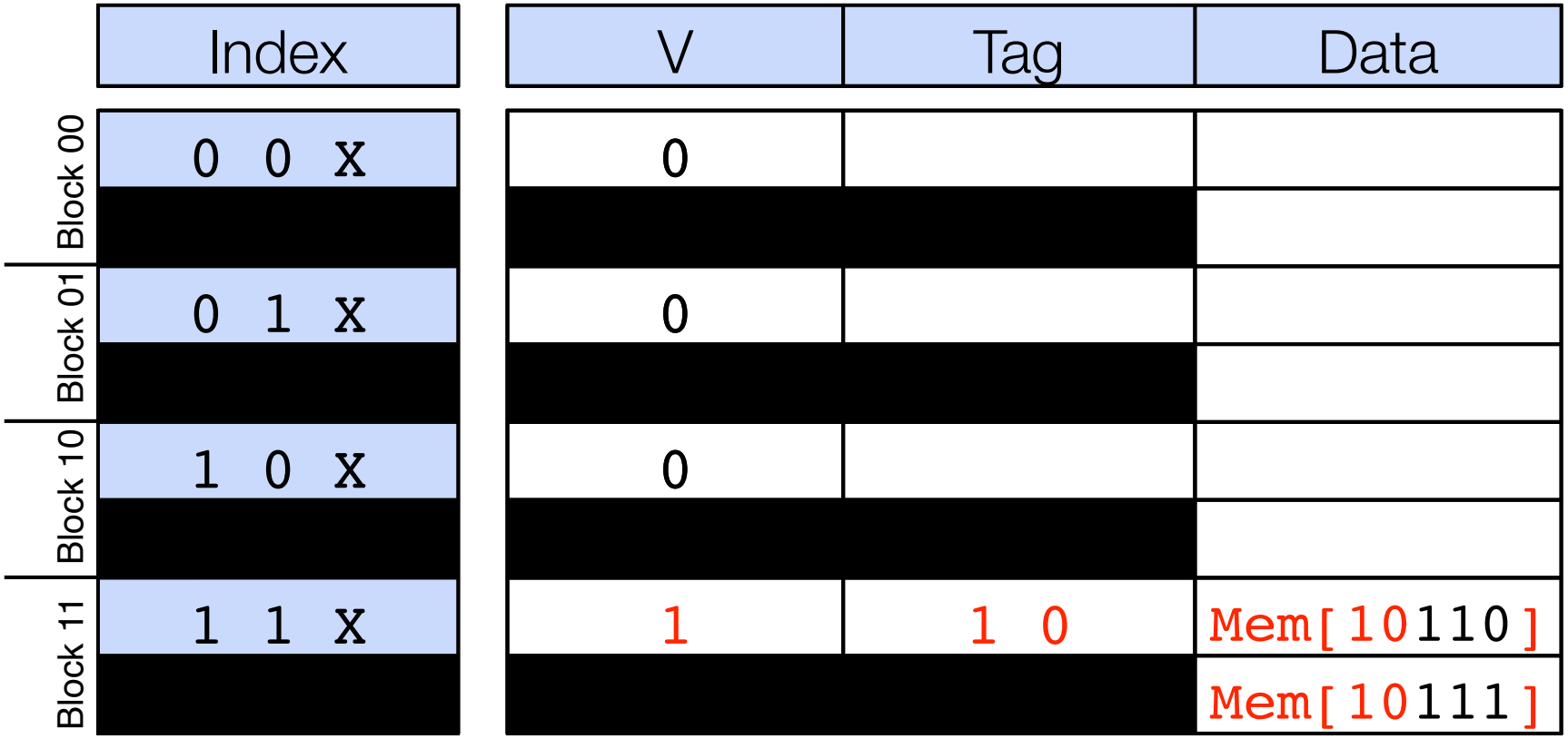


request address 10110 (miss)

Tag: 10 — Cache block: 110

Cache Example

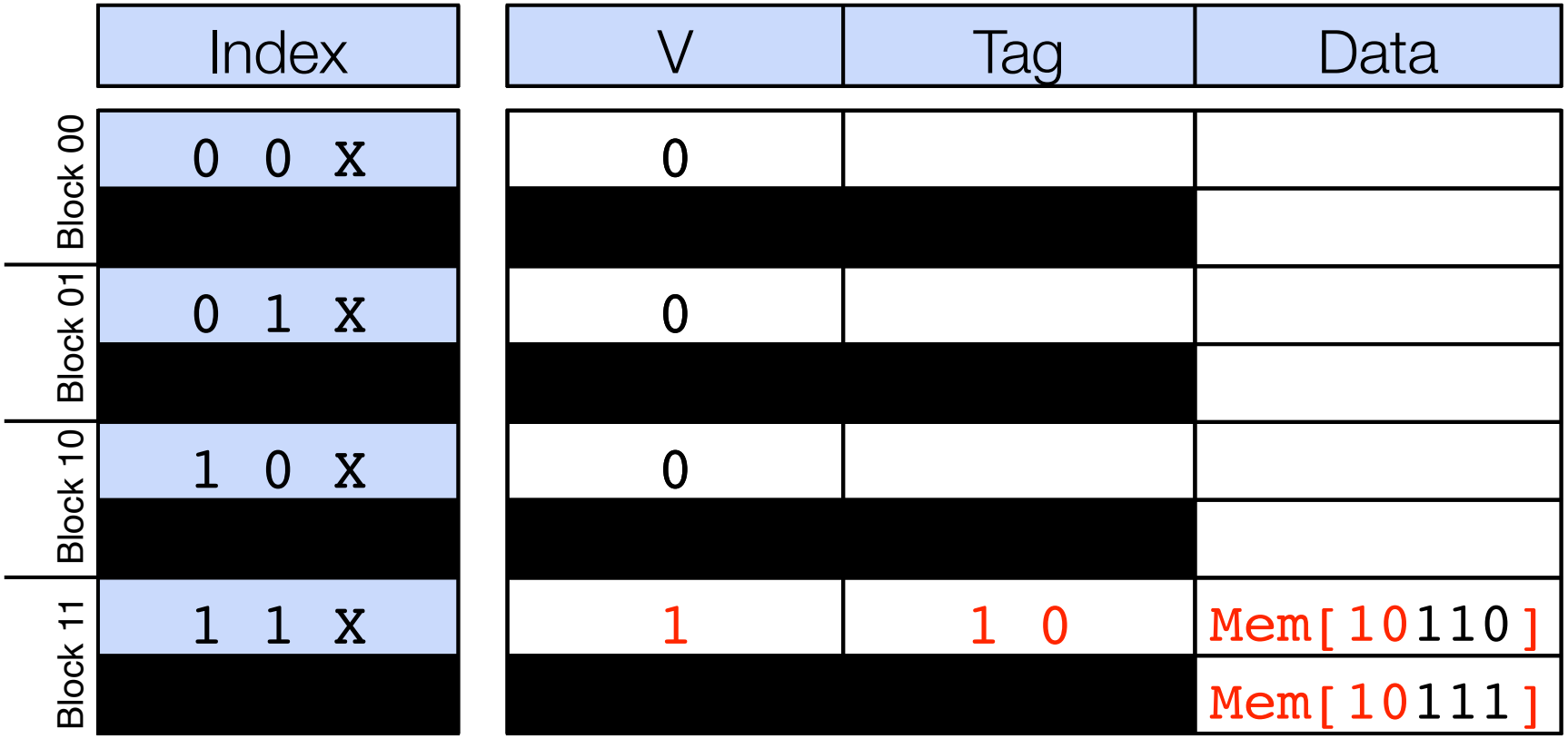
4-block cache, 2 words/block, 32B memory



After handling miss of address 10110

Cache Example

4-block cache, 2 words/block, 32B memory



request address 11011 *(miss)*

Tag | Cache block

Cache Example

4-block cache, 2 words/block, 32B memory

Index	V	Tag	Data
0 0 X	0		
0 1 X	1	1 1	Mem[11010]
			Mem[11011]
1 0 X	0		
1 1 X	1	1 0	Mem[10110]
			Mem[10111]

After handling miss of address 11011

Cache Example

4-block cache, 2 words/block, 32B memory

Index	V	Tag	Data
0 0 X	0		
0 1 X	1	1 1	Mem[11010]
			Mem[11011]
1 0 X	0		
1 1 X	1	1 0	Mem[10110]
			Mem[10111]

request address 10000 (miss)

Tag | Cache block

Cache Example

4-block cache, 2 words/block, 32B memory

Index	V	Tag	Data
0 0 X	1	1 0	Mem[10000]
			Mem[10001]
0 1 X	1	1 1	Mem[11010]
			Mem[11011]
1 0 X	0		
1 1 X	1	1 0	Mem[10110]
			Mem[10111]

After handling miss of address 10000

Cache Example

4-block cache, 2 words/block, 32B memory

Index	V	Tag	Data
0 0 X	1	1 0	Mem[10000]
			Mem[10001]
0 1 X	1	1 1	Mem[11010]
			Mem[11011]
1 0 X	0		
1 1 X	1	1 0	Mem[10110]
			Mem[10111]

request address 00011 (miss)

Tag | Cache block

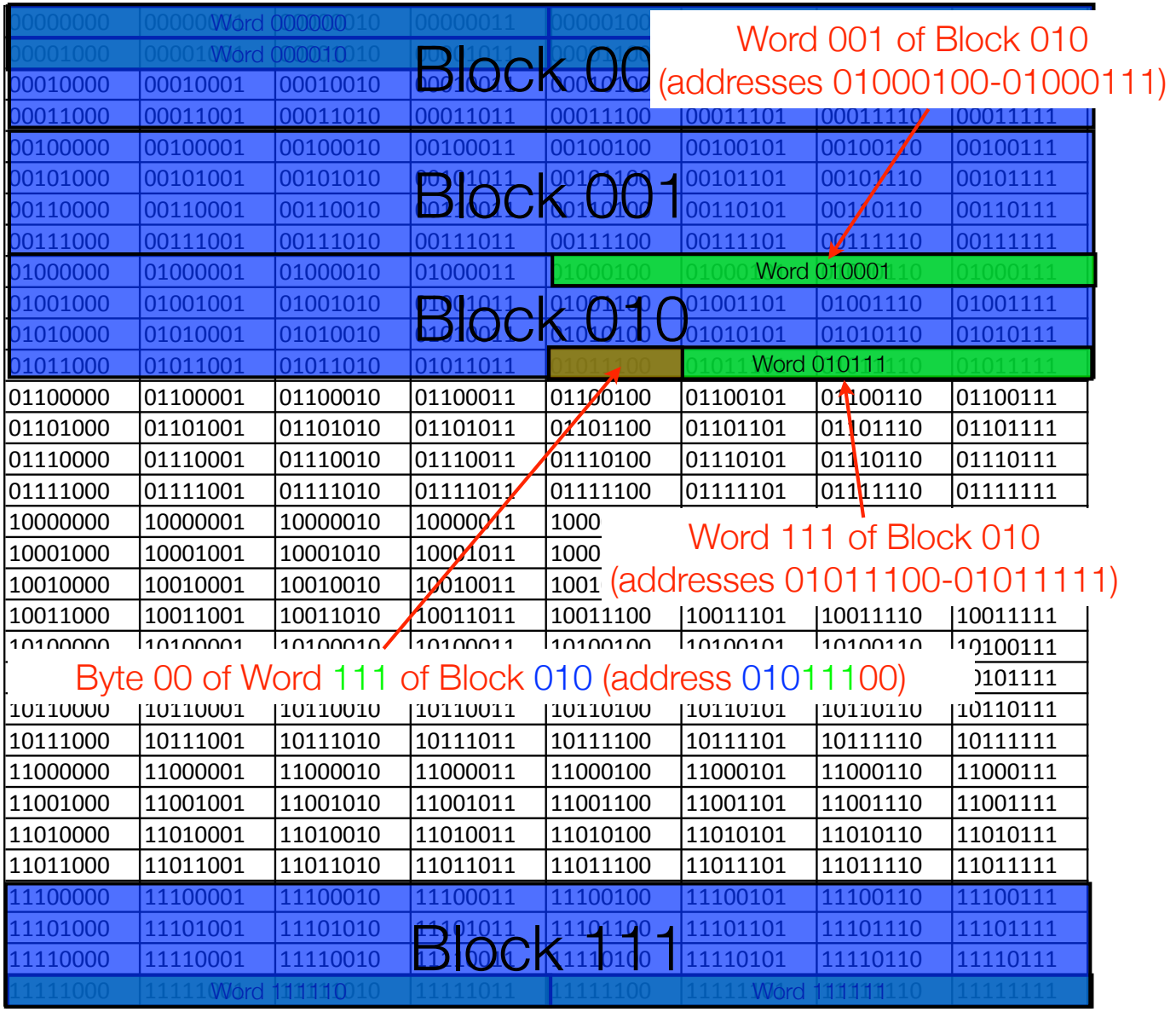
Cache Example

4-block cache, 2 words/block, 32B memory

Index	V	Tag	Data
0 0 X	1	1 0	Mem[10000]
			Mem[10001]
0 1 X	1	0 0	Mem[00010]
			Mem[00011]
1 0 X	0		
1 1 X	1	1 0	Mem[10110]
			Mem[10111]

After handling miss of address 00011
Cache Collision! Overwrite! (Note tag change)

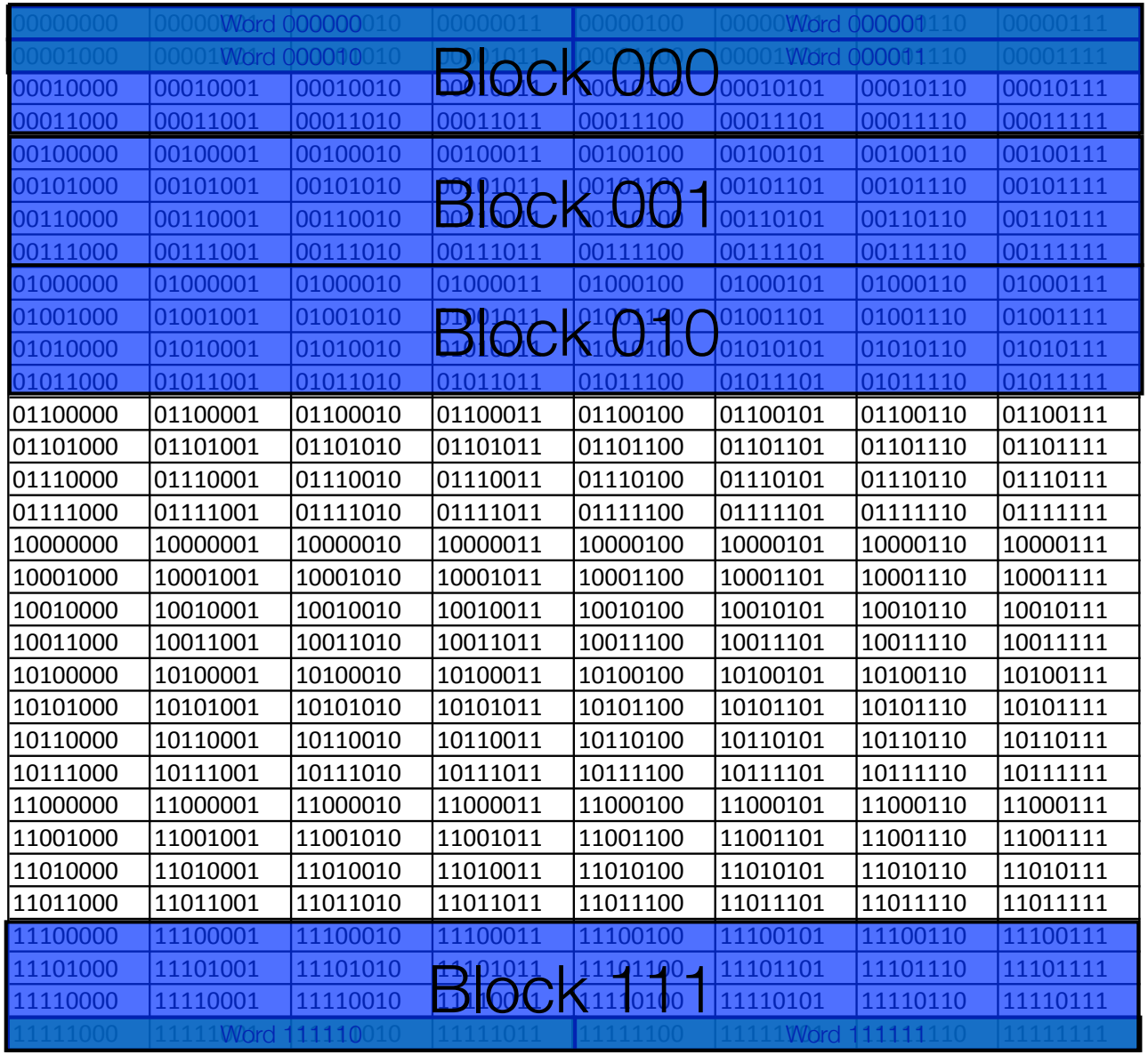
Another view: 2-block cache example



Block addresses (3 = 8-3-2 bits):

Block $B_2B_1B_0$ formed from bytes with addresses $B_2B_1B_0XXXXX$

Another view: 2-block cache example

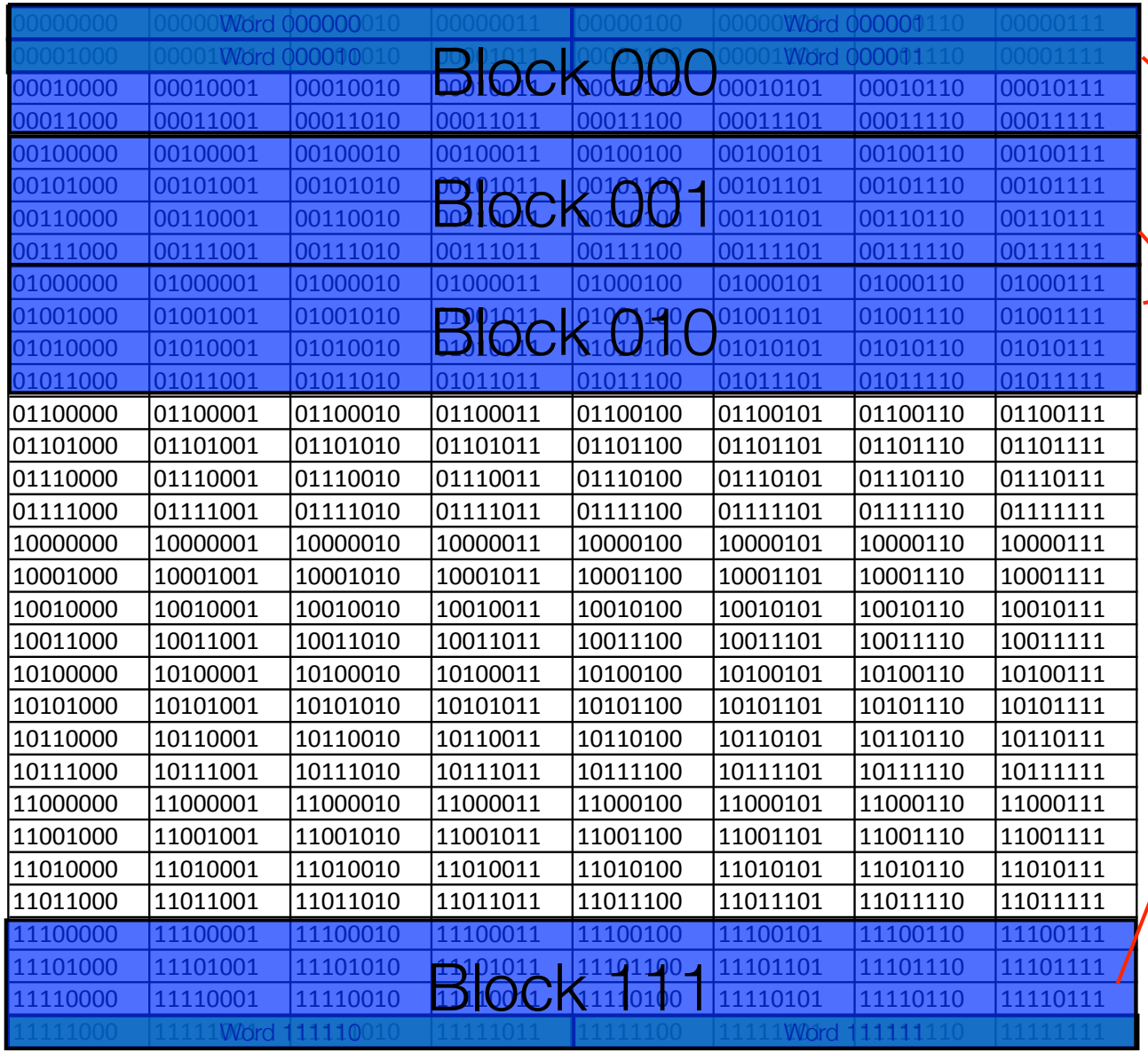


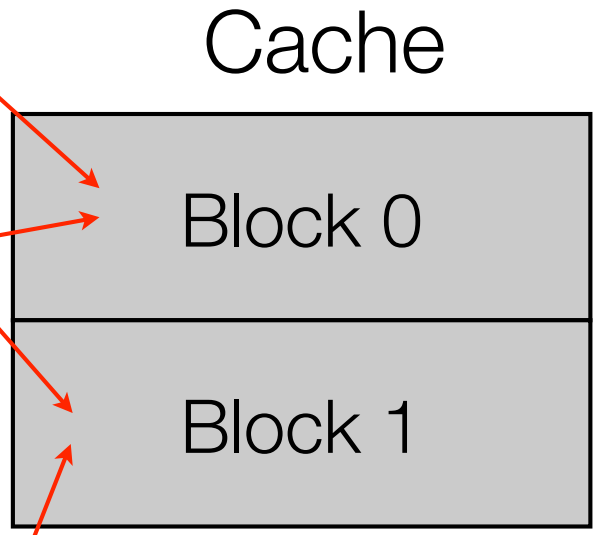
00000000	00000001	00000010	00000011	00000100	00000101	00000110	00000111
00001000	00001001	00001010	00001011	00001100	00001101	00001110	00001111
00010000	00010001	00010010	00010011	00010100	00010101	00010110	00010111
00011000	00011001	00011010	00011011	00011100	00011101	00011110	00011111
00100000	00100001	00100010	00100011	00100100	00100101	00100110	00100111
00101000	00101001	00101010	00101011	00101100	00101101	00101110	00101111
00110000	00110001	00110010	00110011	00110100	00110101	00110110	00110111
00111000	00111001	00111010	00111011	00111100	00111101	00111110	00111111
01000000	01000001	01000010	01000011	01000100	01000101	01000110	01000111
01001000	01001001	01001010	01001011	01001100	01001101	01001110	01001111
01010000	01010001	01010010	01010011	01010100	01010101	01010110	01010111
01011000	01011001	01011010	01011011	01011100	01011101	01011110	01011111
01100000	01100001	01100010	01100011	01100100	01100101	01100110	01100111
01101000	01101001	01101010	01101011	01101100	01101101	01101110	01101111
01110000	01110001	01110010	01110011	01110100	01110101	01110110	01110111
01111000	01111001	01111010	01111011	01111100	01111101	01111110	01111111
10000000	10000001	10000010	10000011	10000100	10000101	10000110	10000111
10001000	10001001	10001010	10001011	10001100	10001101	10001110	10001111
10010000	10010001	10010010	10010011	10010100	10010101	10010110	10010111
10011000	10011001	10011010	10011011	10011100	10011101	10011110	10011111
10100000	10100001	10100010	10100011	10100100	10100101	10100110	10100111
10101000	10101001	10101010	10101011	10101100	10101101	10101110	10101111
10110000	10110001	10110010	10110011	10110100	10110101	10110110	10110111
10111000	10111001	10111010	10111011	10111100	10111101	10111110	10111111
11000000	11000001	11000010	11000011	11000100	11000101	11000110	11000111
11001000	11001001	11001010	11001011	11001100	11001101	11001110	11001111
11010000	11010001	11010010	11010011	11010100	11010101	11010110	11010111
11011000	11011001	11011010	11011011	11011100	11011101	11011110	11011111
11100000	11100001	11100010	11100011	11100100	11100101	11100110	11100111
11101000	11101001	11101010	11101011	11101100	11101101	11101110	11101111
11110000	11110001	11110010	11110011	11110100	11110101	11110110	11110111
11111000	11111001	11111010	11111011	11111100	11111101	11111110	11111111

Block addresses (3 = 8-3-2 bits):

Block $B_2B_1B_0$ formed from bytes with addresses $B_2B_1B_0XXXXX$

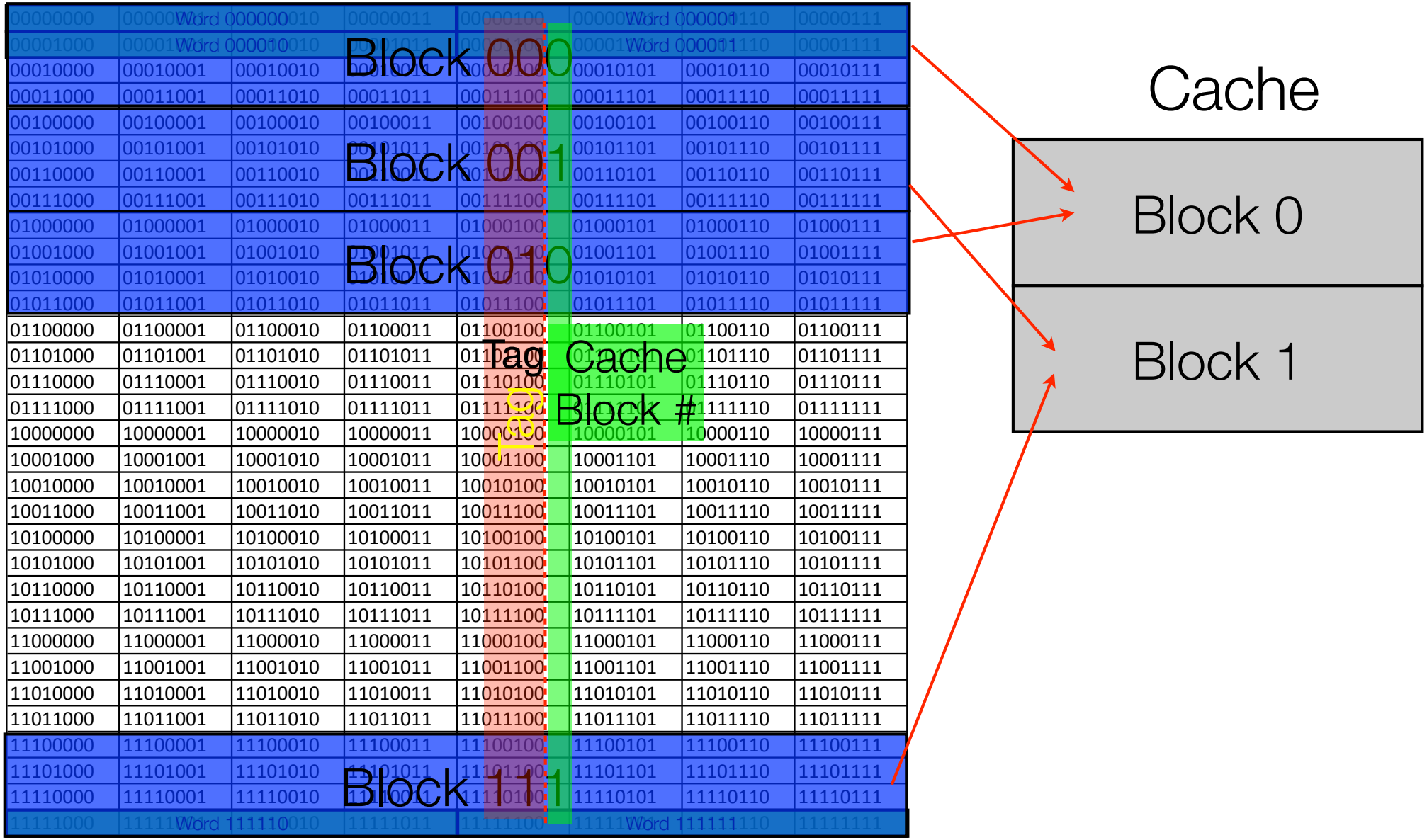
Another view: 2-block cache example



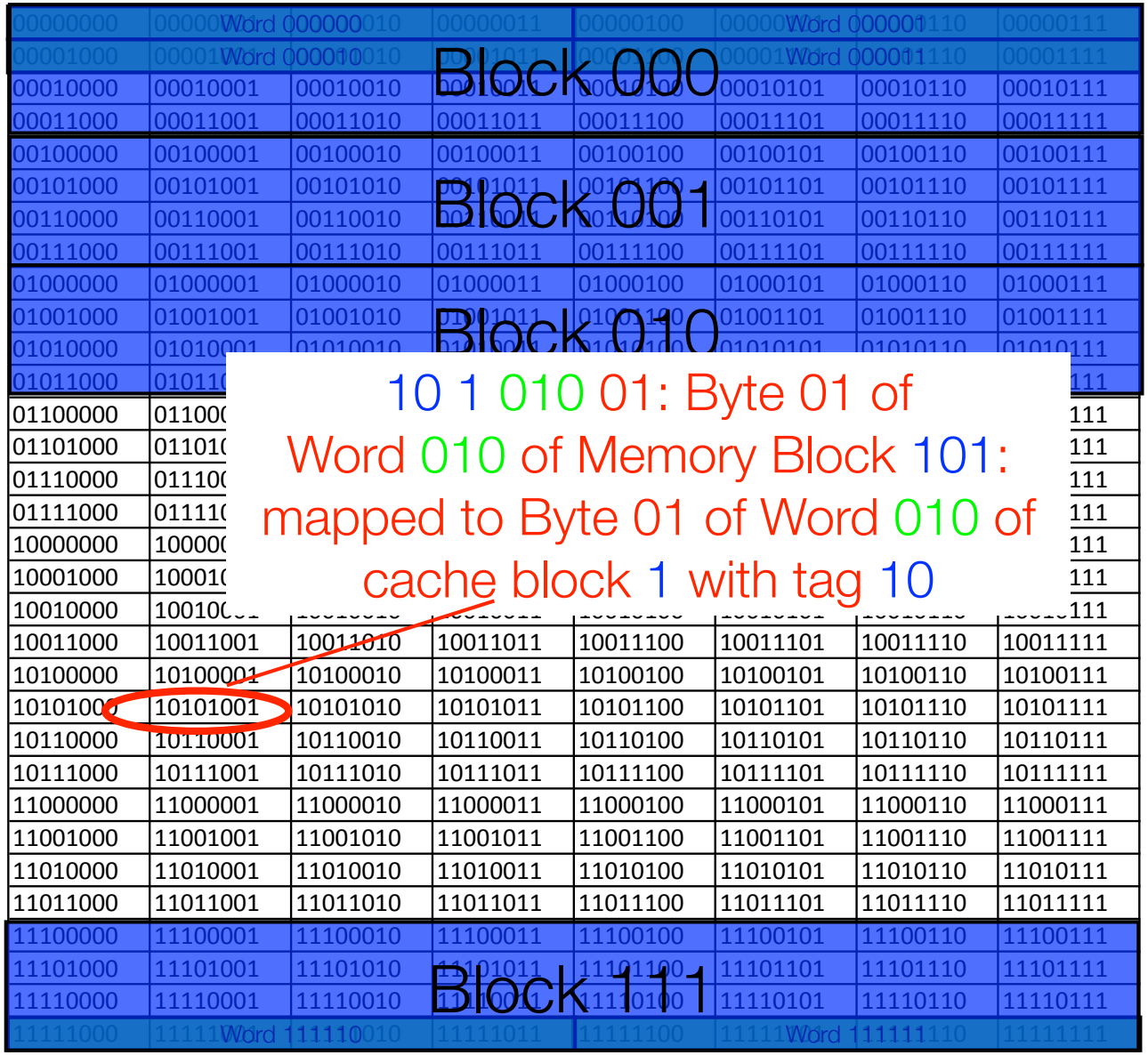


m=1: single bit from memory block address determines block's cache address

Another view: 2-block cache example



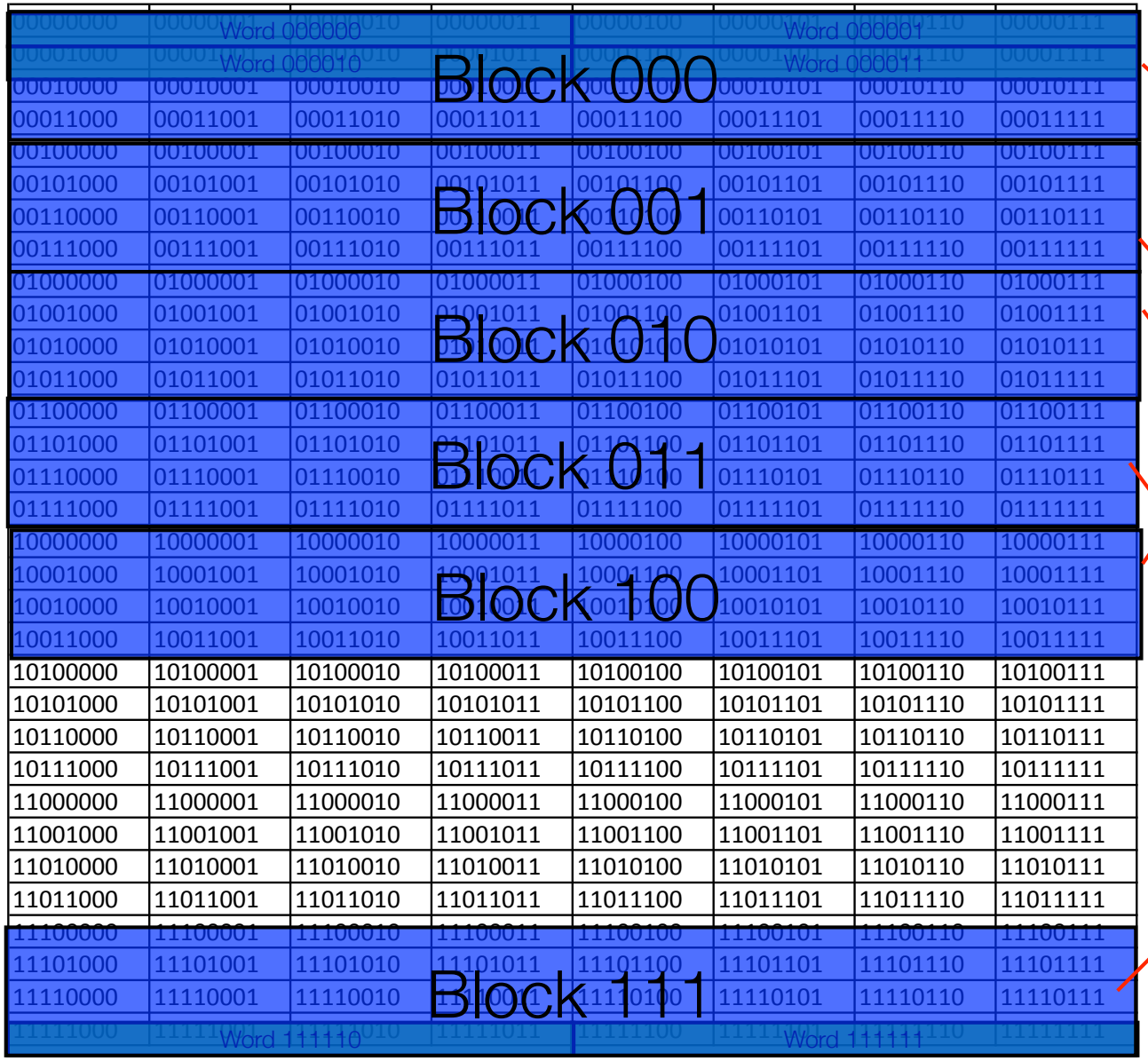
Another view: 2-block cache example



Block addresses (3 = 8-3-2 bits):

Block $B_2B_1B_0$ formed from bytes with addresses $B_2B_1B_0XXXXX$

Another view: 4-block cache example



Cache

Block 00

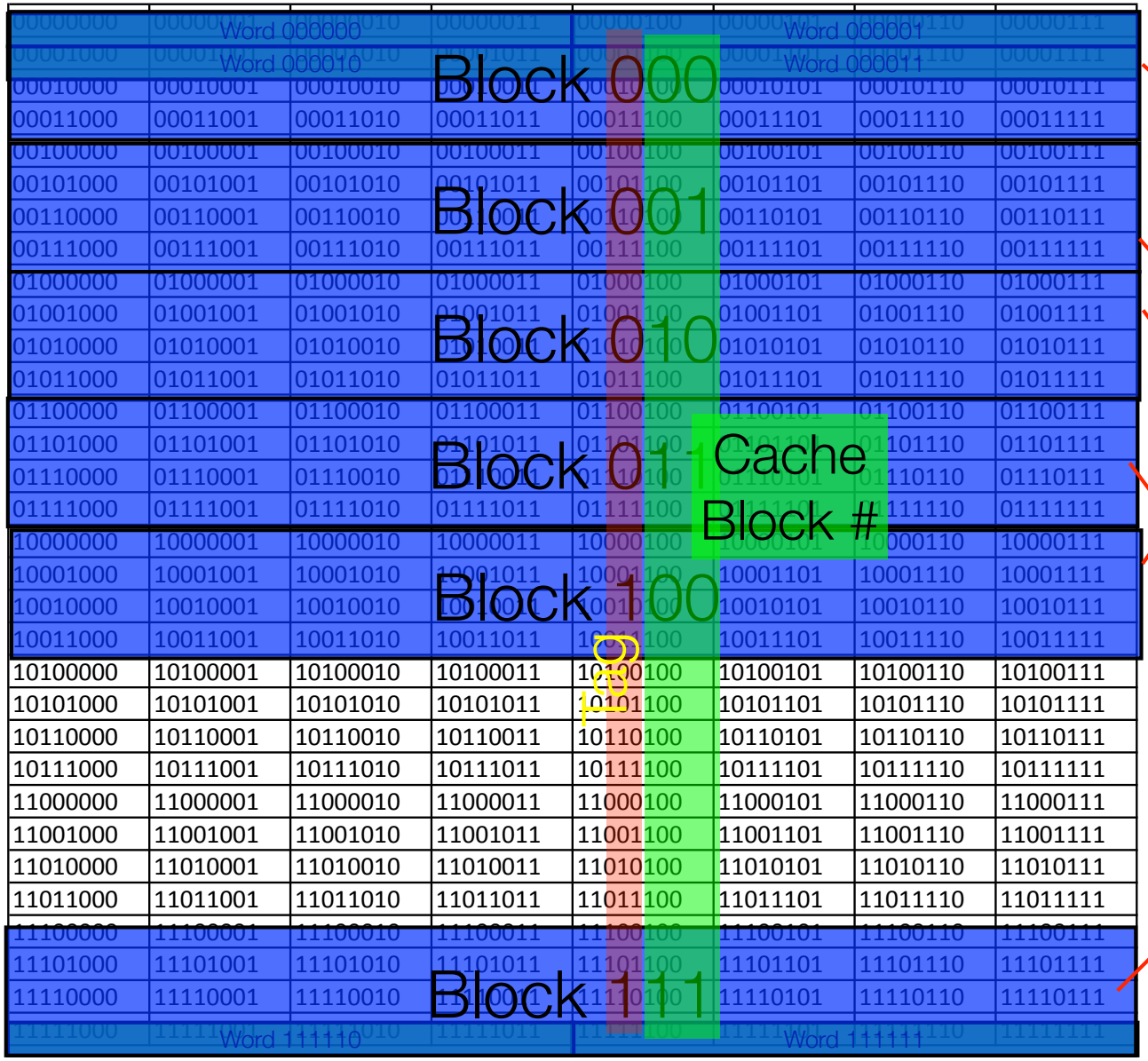
Block 01

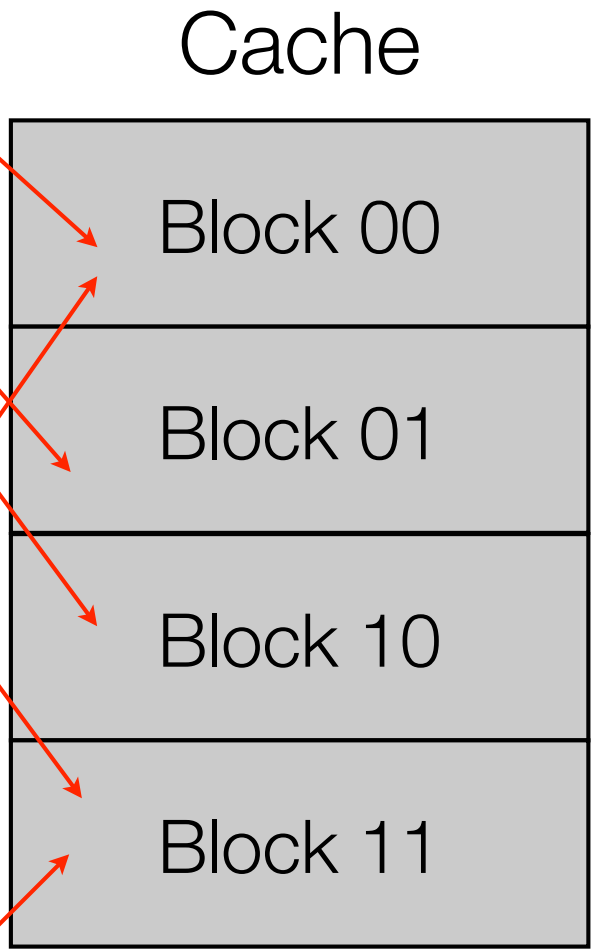
Block 10

Block 11

m=1: single bit from memory block address determines block's cache address

Another view: 4-block cache example



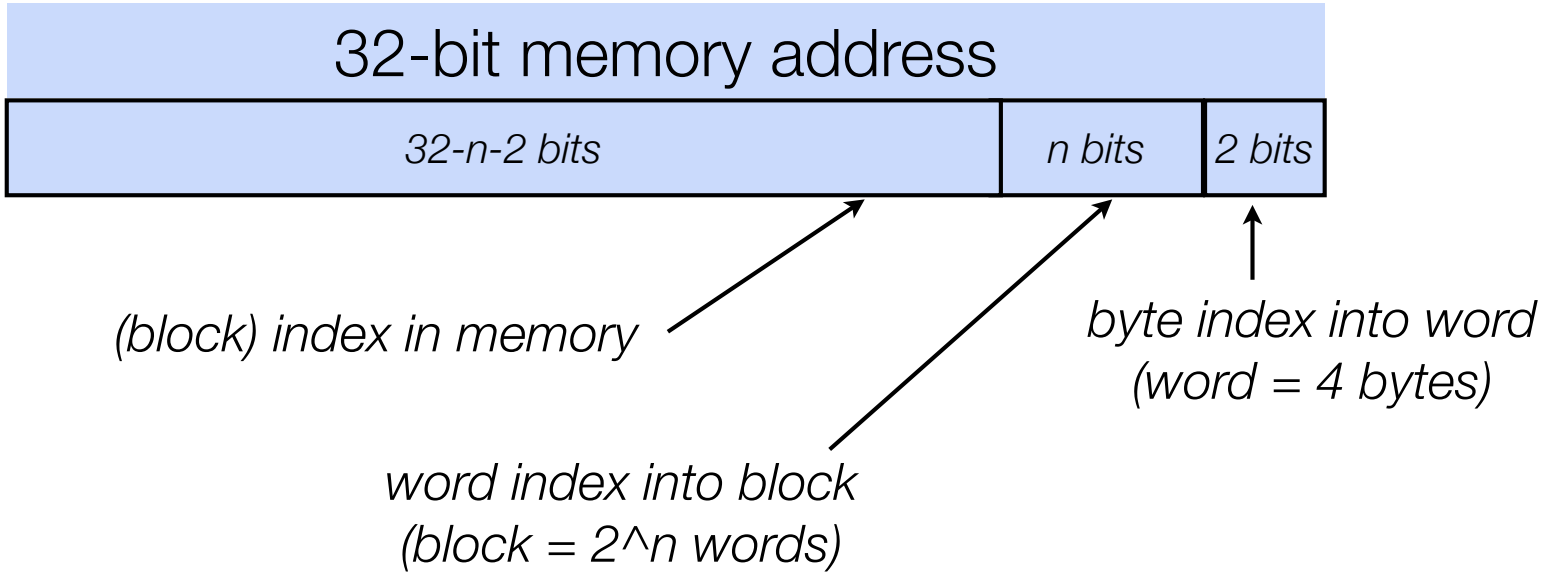


m=1: single bit from memory block address determines block's cache address

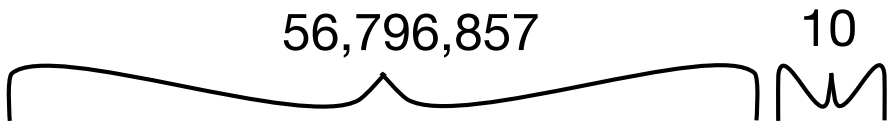
Direct Mapping a MIPS memory address

Multi-word Cache Blocks (Direct Mapping)

Byte address:



- e.g., n=4 (16 words per block), $2^{32-n-2}=2^{26}$ blocks of memory

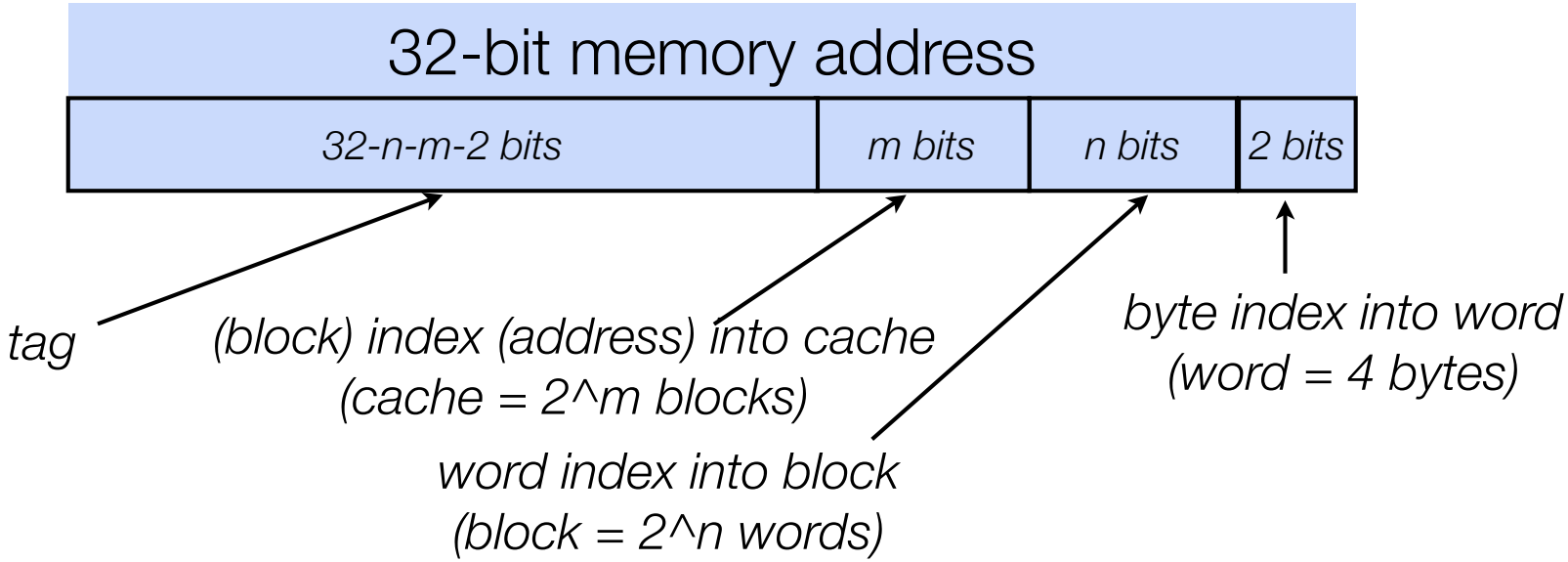


  - 11011000101010011010111001 1010 01:
    - byte #1 of 10th word in 56,796,857th block of memory

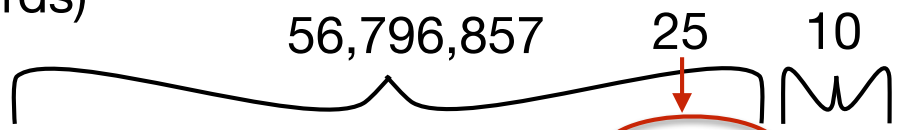
    - (Nothing specific to cache yet!!!)

32-block cache direct mapping example

Block representation in cache in m-to-1 mapping:

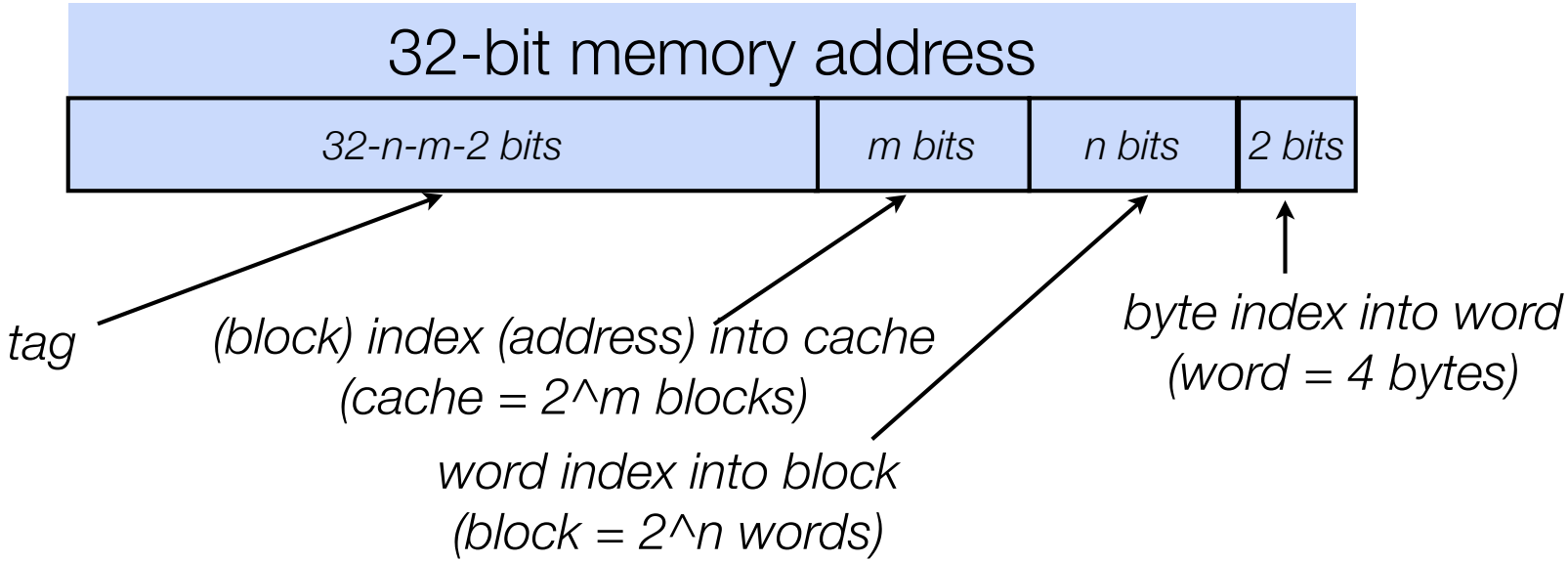


- e.g., m=5, n=4 (16 words per block, 32 blocks in cache: cache stores 32*16 words)



  - 11011000101010011010111001 1010 01:
    - byte #1 of 10th word in 56,796,857th block of memory
    - mapped to byte #1 of 10th word of 25th block of in cache
  - All words whose address is prefixed with 11011000101010011010111001 mapped into the 25th block of the cache

32-block cache direct mapping example



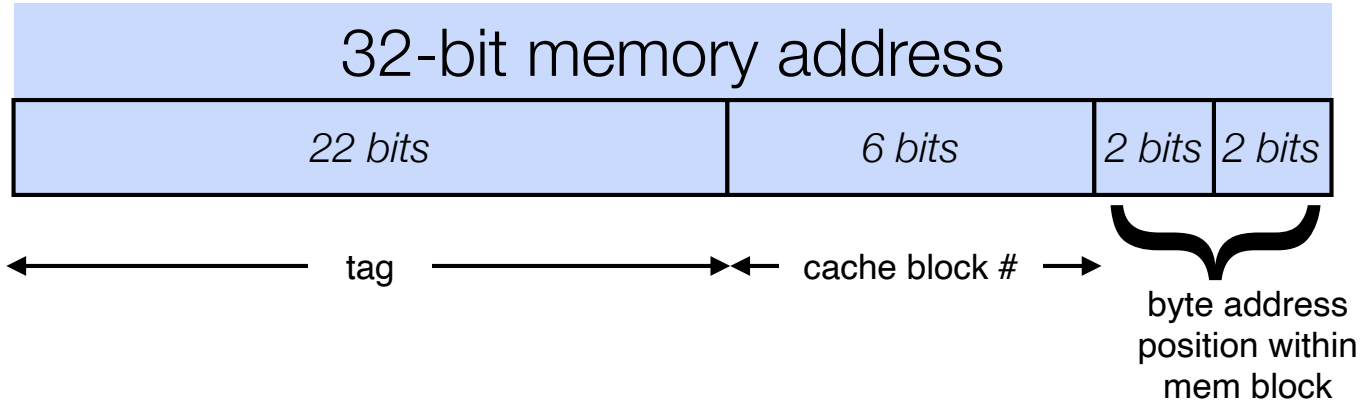
- e.g., m=5, n=4, consider memory addresses:
  - 11011000101010100110101 11001 1010 01: byte #1 of 10th word in 25th block of cache
  - 10010110110110110110110 11001 1110 00: byte #0 of 14th word in 25th block of cache
  - Note: both of these addresses cannot be stored in the direct-mapped cache
    - Different tags → words in different memory blocks
    - Same m-bit cache index → map to same (block) region of cache

A Note on blocks

- Each memory location A maps to a block of memory B
  - e.g., block size of 25 words (100 addresses), memory location 1099112 is in the 10991st block
- Each block of memory B maps to a block of cache B’
  - e.g., block 10991 of memory maps to block 91 in a 100-block (drect-mapped) cache
- To find a particular memory address A in the cache (or see if its there)
  - Step 1: Determine its block in memory (A → B)
  - Step 2: Look for that block in the cache (B → B’) : check the tag
- When talking about caching policies, focus on the 2nd mapping (B → B’) since the first mapping is just a relabelling within memory (byte # to block #)

Example: Other Block Size

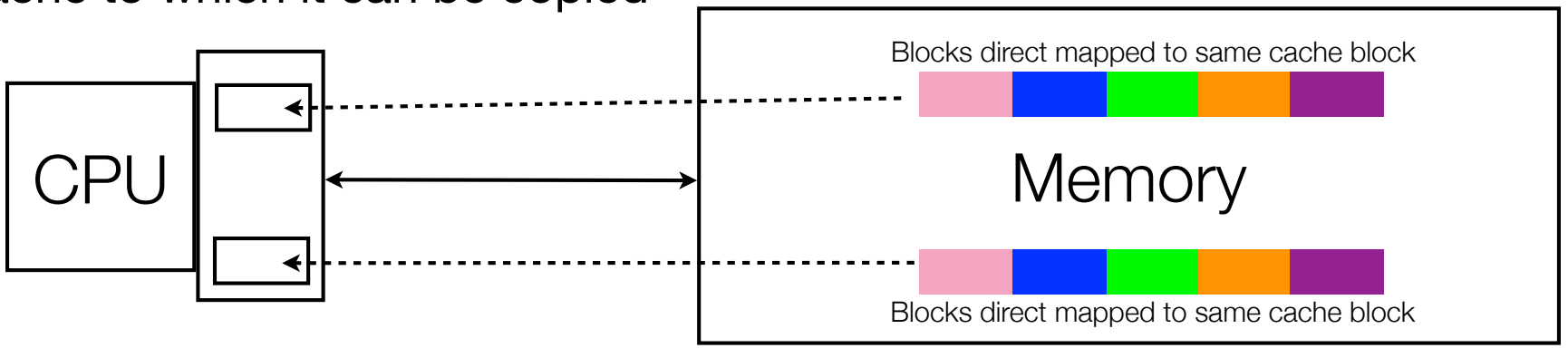
- Suppose cache holds 64 blocks, Cache and mem block size is 16 bytes
  - To what block number does (byte) address 1200 map?
- Block address = floor(1200/16) = 75 (75th block in memory)
- Block number = 75 modulo 64 = 11 (Direct mapping, would map to 11th block in cache)



Cache Mapping Strategy #2: Associative Mapping

Associative Caches

- Recall: **Direct-Mapped** - each block of memory has a single location in the cache to which it can be copied



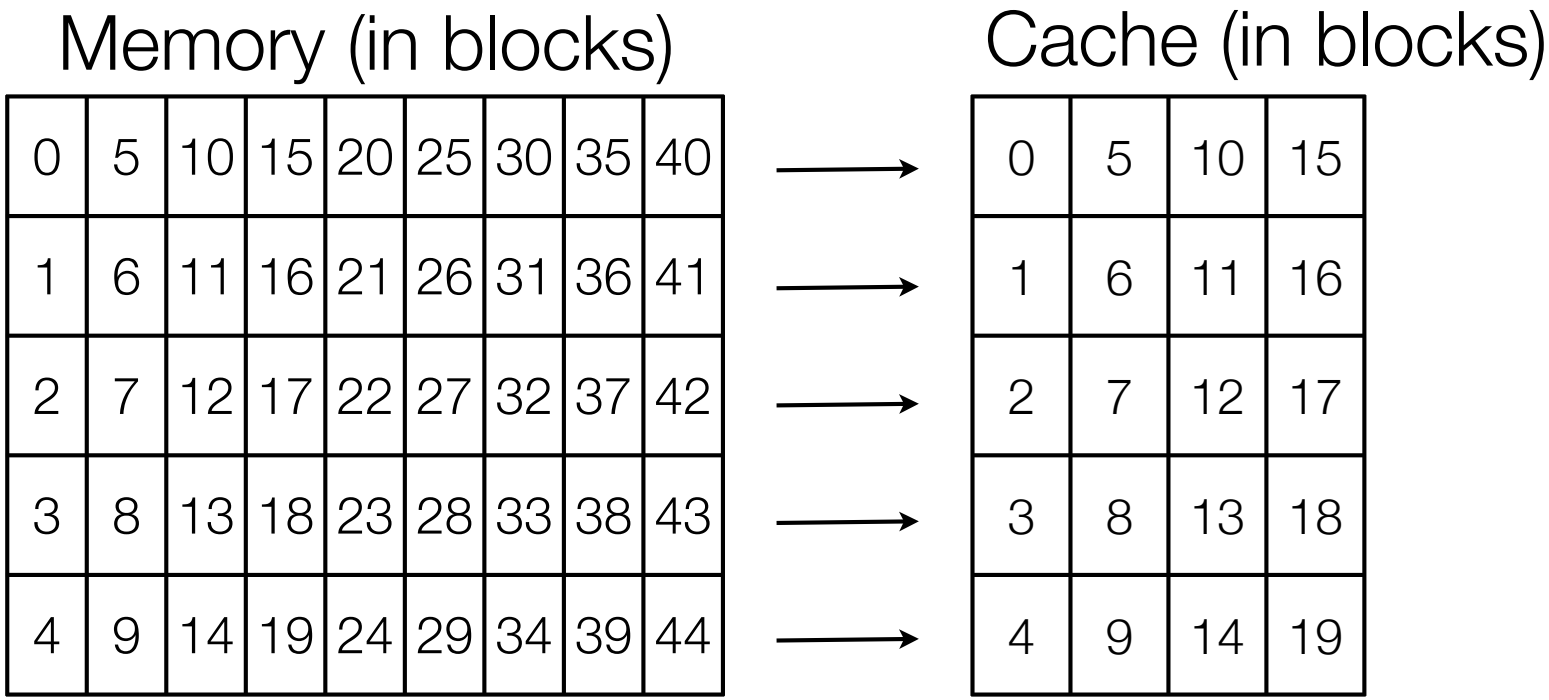
- **Fully Associative**: any block can go anywhere in cache
  - Memory still transferred in units of blocks
  - Where to put a newly retrieved block in cache?
    - Empty block (when one exists), then least recently accessed, or least frequently accessed
- **Fully Associative Pros**: less likely to overwrite recently accessed memory locations
- **Fully Associative Cons**: time needed to determine / locate a block of memory in the cache

Set Associative Caching

- Tradeoff of Fully Associative and Direct Mapped
  - n-way Associative: memory block can be stored in n locations in cache
- Given an M-block memory, an m-block cache that is n-way associative:
  - n=1: Direct mapped
  - n=m: Fully Associative
  - In general, M * n / m blocks “share” cache locations (M=blocks in memory)
- Simple mapping scheme: memory block i can map to blocks i mod (m/n) + km/n for k=0,1,2,..., n-1
- e.g., m=20, n=4
    - each mem block 0, 5, 10, 15, 20, 25, ... maps to any of cache blocks 0, 5, 10, 15
    - each mem block 1, 6, 11, 16, 21, 26, ... maps to any of cache blocks 1, 6, 11, 16
    - ...
    - each mem block 4, 9, 14, 19, 24, 29, ... maps to any of cache blocks 4, 9, 14, 19

Set Associative Caching

- e.g., m=20, n=4: block i can map to blocks i mod 5 + 5k for k=0,1,2,..., 3
  - each mem block 0, 5, 10, 15, 20, 25, ... maps to any of cache blocks 0, 5, 10, 15
  - each mem block 1, 6, 11, 16, 21, 26, ... maps to any of cache blocks 1, 6, 11, 16
  - ...
  - each mem block 4, 9, 14, 19, 24, 29, ... maps to any of cache blocks 4, 9, 14, 19



Block Size Considerations

- Larger blocks should reduce miss rate, due to spatial locality
- But in a fixed-sized cache
  - Larger blocks → fewer of them → more competition → increased miss rate
  - Larger blocks → pollution
- Larger miss penalty, which could override benefit of reduced miss rate

One more example

- Direct Mapped example

- Remember:
  - Map Memory into blocks, then for purposes of caching just think in terms of blocks
  - Just a question of which cache block the memory block maps to (is the harder part)

Mapping Memory Words (or Bytes) to Blocks

- Note: we can do this mapping before even revealing the size of the cache
- Suppose 8 words in a block
- word addr (in MIPS) = floor(byte addr / 4) (drop 2 least sig. bits)
- block # = word addr / 8 (drop 3 least sig. bits) - why divide by 8?

Time →

byte addr	22	173	21	30	268	29	42	38	174	105	0	23	141	28	8	159	145	15	6
word addr																			
mem block #																			

Mapping Memory Words (or Bytes) to Blocks

- Note: we can do this mapping before even revealing the size of the cache
- Suppose 8 words in a block
- word addr (in MIPS) = floor(byte addr / 4) (drop 2 least sig. bits)
- block # = word addr / 8 (drop 3 least sig. bits) - why divide by 8?

Time →

byte addr	22	173	21	30	268	29	42	38	174	105	0	23	141	28	8	159	145	15	6
word addr	5	43	5	7	67	7	10	9	43	26	0	5	35	7	2	39	36	3	1
mem block #																			

Mapping Memory Words (or Bytes) to Blocks

- Note: we can do this mapping before even revealing the size of the cache
- Suppose 8 words in a block
- word addr (in MIPS) = floor(byte addr / 4) (drop 2 least sig. bits)
- block # = word addr / 8 (drop 3 least sig. bits) - why divide by 8?

Time →

byte addr	22	173	21	30	268	29	42	38	174	105	0	23	141	28	8	159	145	15	6
word addr	5	43	5	7	67	7	10	9	43	26	0	5	35	7	2	39	36	3	1
mem block #	0	5	0	0	8	0	1	1	5	3	0	0	4	0	0	4	4	0	0

Still haven't considered cache!!!

Direct Mapped Cache with this example

- Suppose Cache contains 4 blocks
  - still 8-word blocksize, so 32 words in the total in the cache
- Memory block i direct maps to block i mod 4 of cache

Time →

byte addr	22	173	21	30	268	29	42	38	174	105	0	23	141	28	8	159	145	15	6
word addr	5	43	5	7	67	7	10	9	43	26	0	5	35	7	2	39	36	3	1
mem block #	0	5	0	0	8	0	1	1	5	3	0	0	4	0	0	4	4	0	0
cache block #																			
Cache hit?																			

Direct Mapped Cache with this example

- Suppose Cache contains 4 blocks
  - still 8-word blocksize, so 32 words in the total in the cache
- Memory block i direct maps to block i mod 4 of cache

Time →

byte addr	22	173	21	30	268	29	42	38	174	105	0	23	141	28	8	159	145	15	6
word addr	5	43	5	7	67	7	10	9	43	26	0	5	35	7	2	39	36	3	1
mem block #	0	5	0	0	8	0	1	1	5	3	0	0	4	0	0	4	4	0	0
cache block #																			
Cache hit?																			

Direct Mapped Cache with this example

- Suppose Cache contains 4 blocks
  - still 8-word blocksize, so 32 words in the total in the cache
- Memory block i direct maps to block i mod 4 of cache

Time →

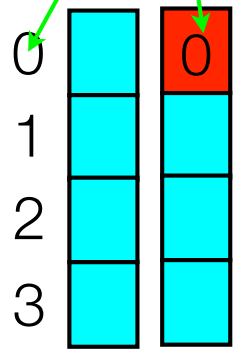
byte addr	22	173	21	30	268	29	42	38	174	105	0	23	141	28	8	159	145	15	6
word addr	5	43	5	7	67	7	10	9	43	26	0	5	35	7	2	39	36	3	1
mem block #	0	5	0	0	8	0	1	1	5	3	0	0	4	0	0	4	4	0	0
cache block #	0	1	0	0	0	0	1	1	1	3	0	0	0	0	0	0	0	0	0
Cache hit?																			

Direct Mapped Cache with this example

- Suppose Cache contains 4 blocks
  - still 8-word blocksize, so 32 words in the total in the cache
- Memory block i direct maps to block i mod 4 of cache

Time →

byte addr	22	173	21	30	268	29	42	38	174	105	0	23	141	28	8	159	145	15	6
word addr	5	43	5	7	67	7	10	9	43	26	0	5	35	7	2	39	36	3	1
mem block #	0	5	0	0	8	0	1	1	5	3	0	0	4	0	0	4	4	0	0
cache block #	0	1	0	0	0	0	1	1	1	3	0	0	0	0	0	0	0	0	0
Cache hit?																			

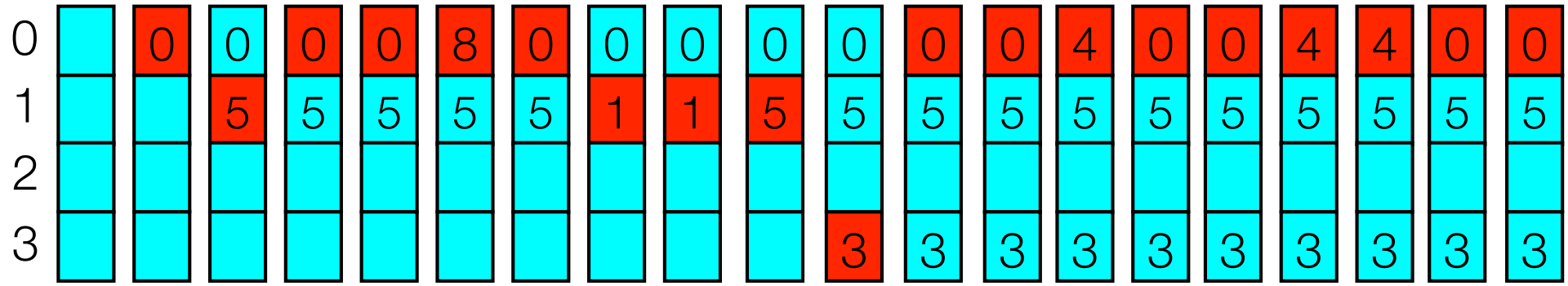


Direct Mapped Cache with this example

- Suppose Cache contains 4 blocks
  - still 8-word blocksize, so 32 words in the total in the cache
- Memory block i direct maps to block i mod 4 of cache

Time →

byte addr	22	173	21	30	268	29	42	38	174	105	0	23	141	28	8	159	145	15	6
word addr	5	43	5	7	67	7	10	9	43	26	0	5	35	7	2	39	36	3	1
mem block #	0	5	0	0	8	0	1	1	5	3	0	0	4	0	0	4	4	0	0
cache block #	0	1	0	0	0	0	1	1	1	3	0	0	0	0	0	0	0	0	0
Cache hit?																			

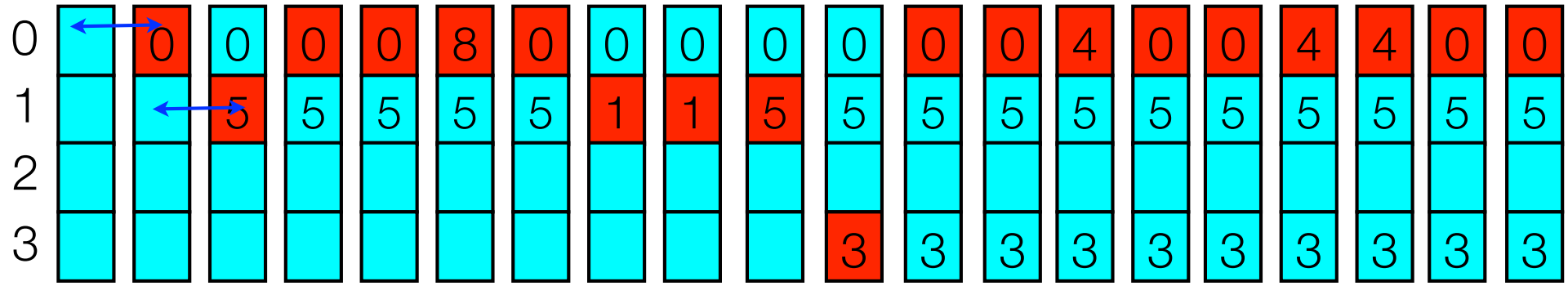


Direct Mapped Cache with this example

- Suppose Cache contains 4 blocks
  - still 8-word blocksize, so 32 words in the total in the cache
- Memory block i direct maps to block i mod 4 of cache

Time →

byte addr	22	173	21	30	268	29	42	38	174	105	0	23	141	28	8	159	145	15	6
word addr	5	43	5	7	67	7	10	9	43	26	0	5	35	7	2	39	36	3	1
mem block #	0	5	0	0	8	0	1	1	5	3	0	0	4	0	0	4	4	0	0
cache block #	0	1	0	0	0	0	1	1	1	3	0	0	0	0	0	0	0	0	0
Cache hit?	N	N																	

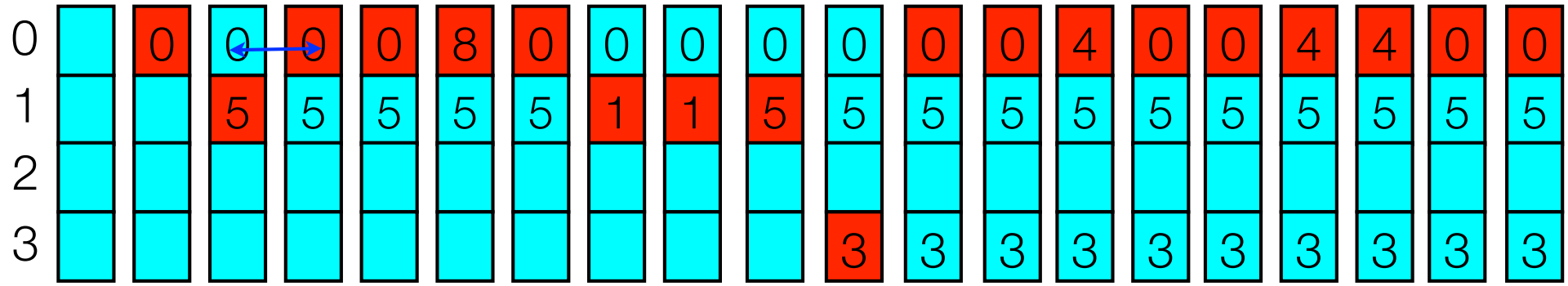


Direct Mapped Cache with this example

- Suppose Cache contains 4 blocks
  - still 8-word blocksize, so 32 words in the total in the cache
- Memory block i direct maps to block i mod 4 of cache

Time →

byte addr	22	173	21	30	268	29	42	38	174	105	0	23	141	28	8	159	145	15	6
word addr	5	43	5	7	67	7	10	9	43	26	0	5	35	7	2	39	36	3	1
mem block #	0	5	0	0	8	0	1	1	5	3	0	0	4	0	0	4	4	0	0
cache block #	0	1	0	0	0	0	1	1	1	3	0	0	0	0	0	0	0	0	0
Cache hit?	N	N	Y																

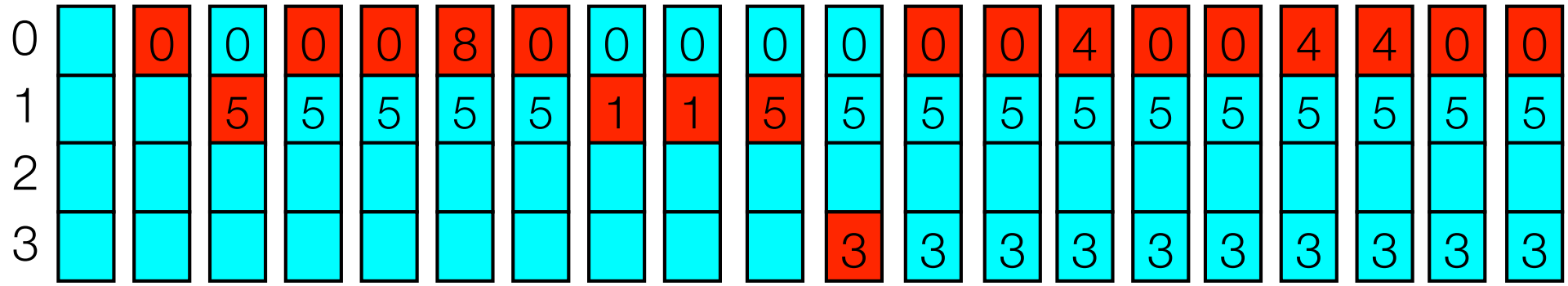


Direct Mapped Cache with this example

- Suppose Cache contains 4 blocks
  - still 8-word blocksize, so 32 words in the total in the cache
- Memory block i direct maps to block i mod 4 of cache

Time →

byte addr	22	173	21	30	268	29	42	38	174	105	0	23	141	28	8	159	145	15	6
word addr	5	43	5	7	67	7	10	9	43	26	0	5	35	7	2	39	36	3	1
mem block #	0	5	0	0	8	0	1	1	5	3	0	0	4	0	0	4	4	0	0
cache block #	0	1	0	0	0	0	1	1	1	3	0	0	0	0	0	0	0	0	0
Cache hit?	N	N	Y	Y	N	N	N	Y	N	N	Y	Y	N	N	Y	N	Y	N	Y



Writing to Memory (Cache consistency)

Where we are...

- Cache Miss: occurs when memory is accessed (read or write) - need to pull data in memory into cache

- Cache inconsistency: data stored in block of cache does not match data in memory

  - Can occur after a sw instruction: “memory” is modified, so value is modified in cache, what about in the main memory?

Handling Writes: Write Through

- On data-write hit (memory block already cached), could just update the block in cache, but then cache and memory would be inconsistent
  - Problem: Can't overwrite block in cache (with data from memory in different memory block that maps to same cache block) until made consistent
- **Write through**: on write, update memory as well as cache
  - Makes writes take longer (e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles, effective CPI = 1 + 0.1×100 = 11)
  - Solution: write buffer which holds data waiting to be written to memory. CPU can now continue immediately, stalling only if write buffer is full.

Handling Writes: Write Back

- An alternative to write through
- **Write Back**: on data-write hit, just update the block in cache
  - Keep track of whether each block is dirty
  - When a dirty block is to be replaced, only then write it back to memory
  - Can use a write buffer to allow replacing block to be read first

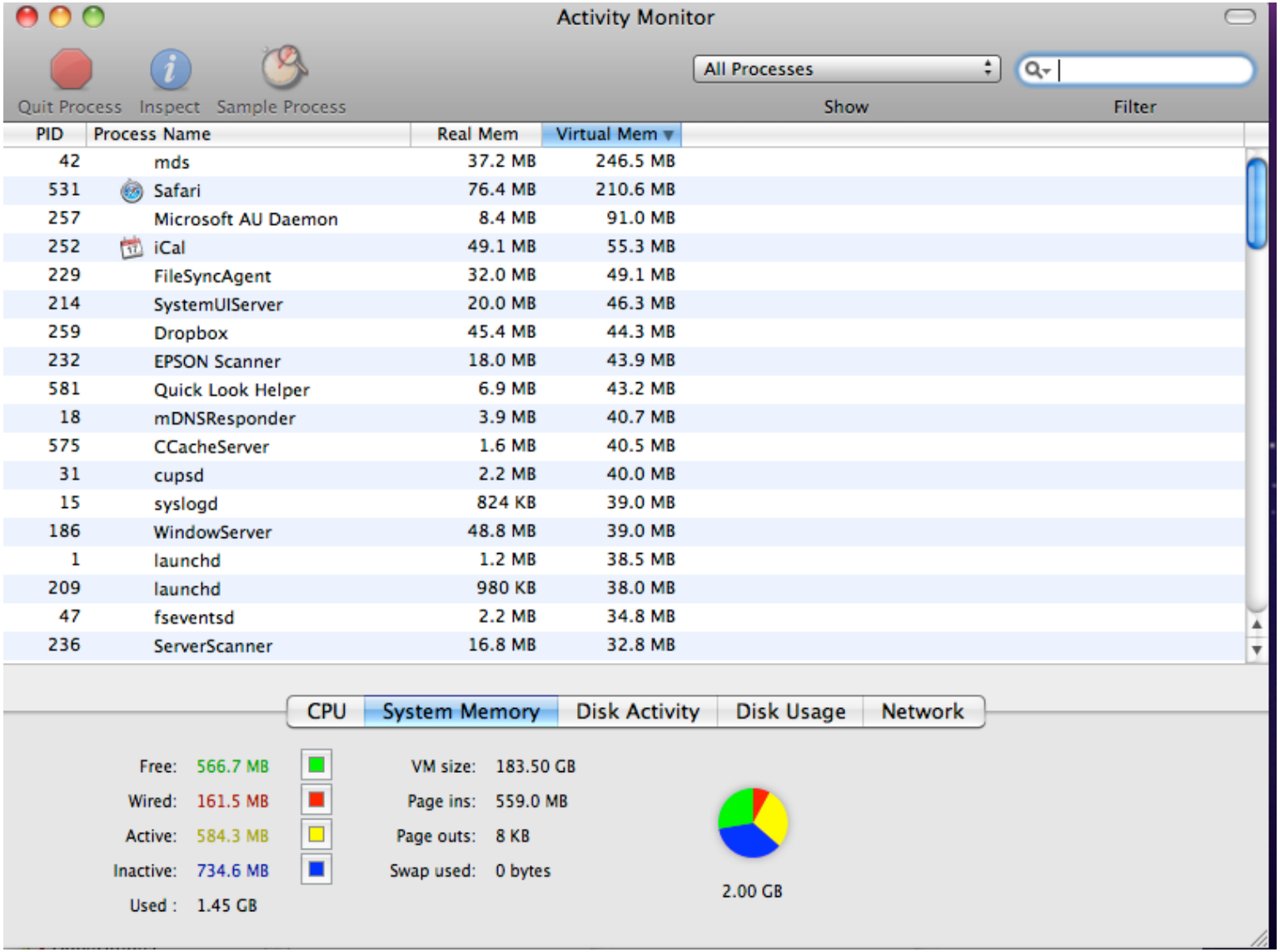
Virtual Memory

Virtual Memory

- The “reverse” of cache
- All programs running don’t “fit” into memory
- Idea: use disk to store parts of memory used by program while they are not being run
- When part of program needs to access memory currently stored on disk, transfer from disk to mem.

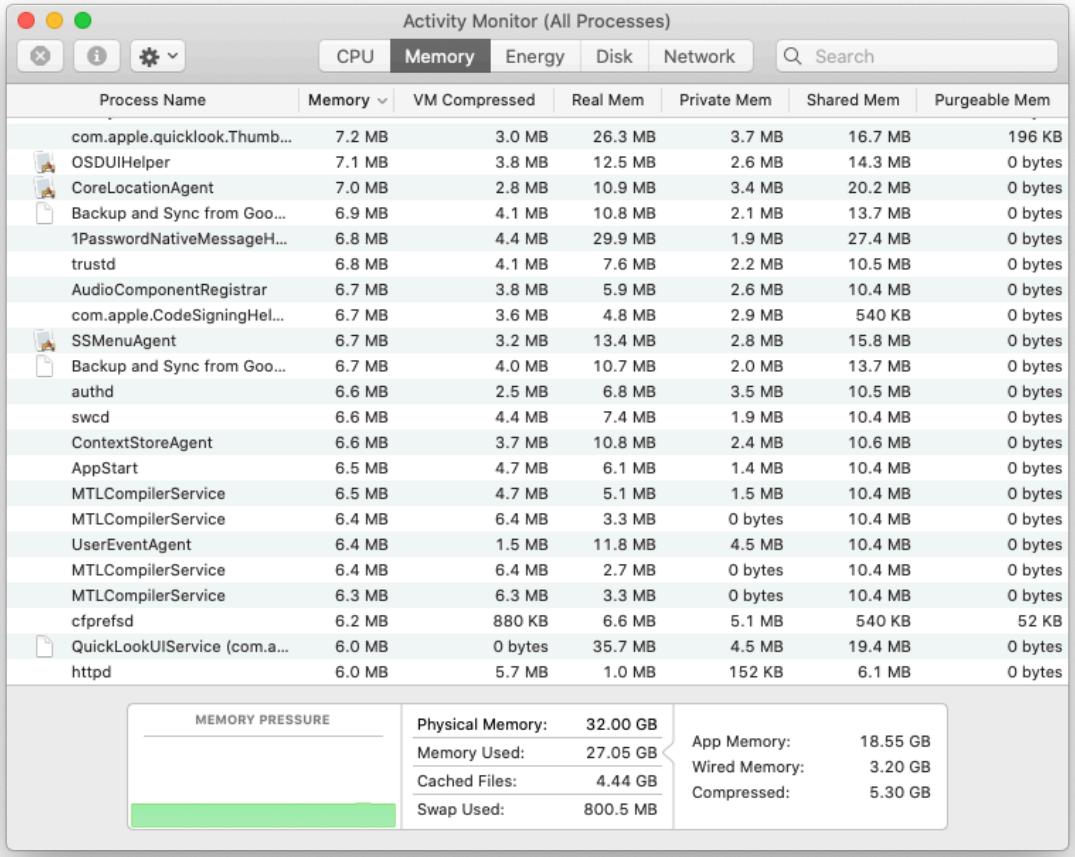
Virtual Mem in practice

- Mac's Old Activity Monitor



Virtual Mem in practice

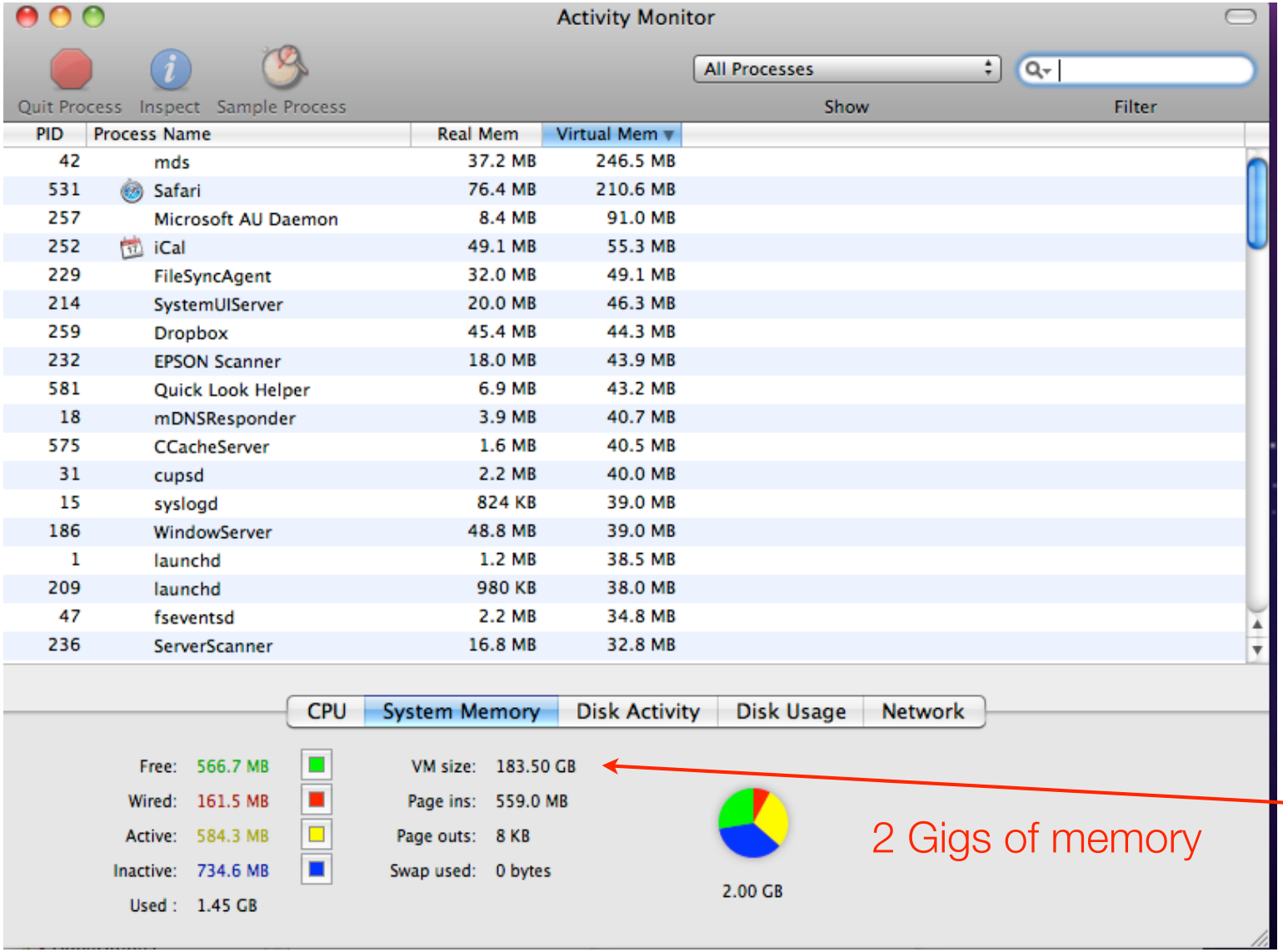
- Mac's current Activity Monitor



- App memory: memory used by apps
- Wired memory: cannot be cached on disk
- Compressed: portion of memory compressed (saves space, increases computation)

Virtual Mem in practice

- Mac's Activity Monitor



2 Gigs of memory

~200 Gigs of Virtual Mem (not all in use)!!!

Computer Architecture, Then and Now and Here

History of Processor Performance

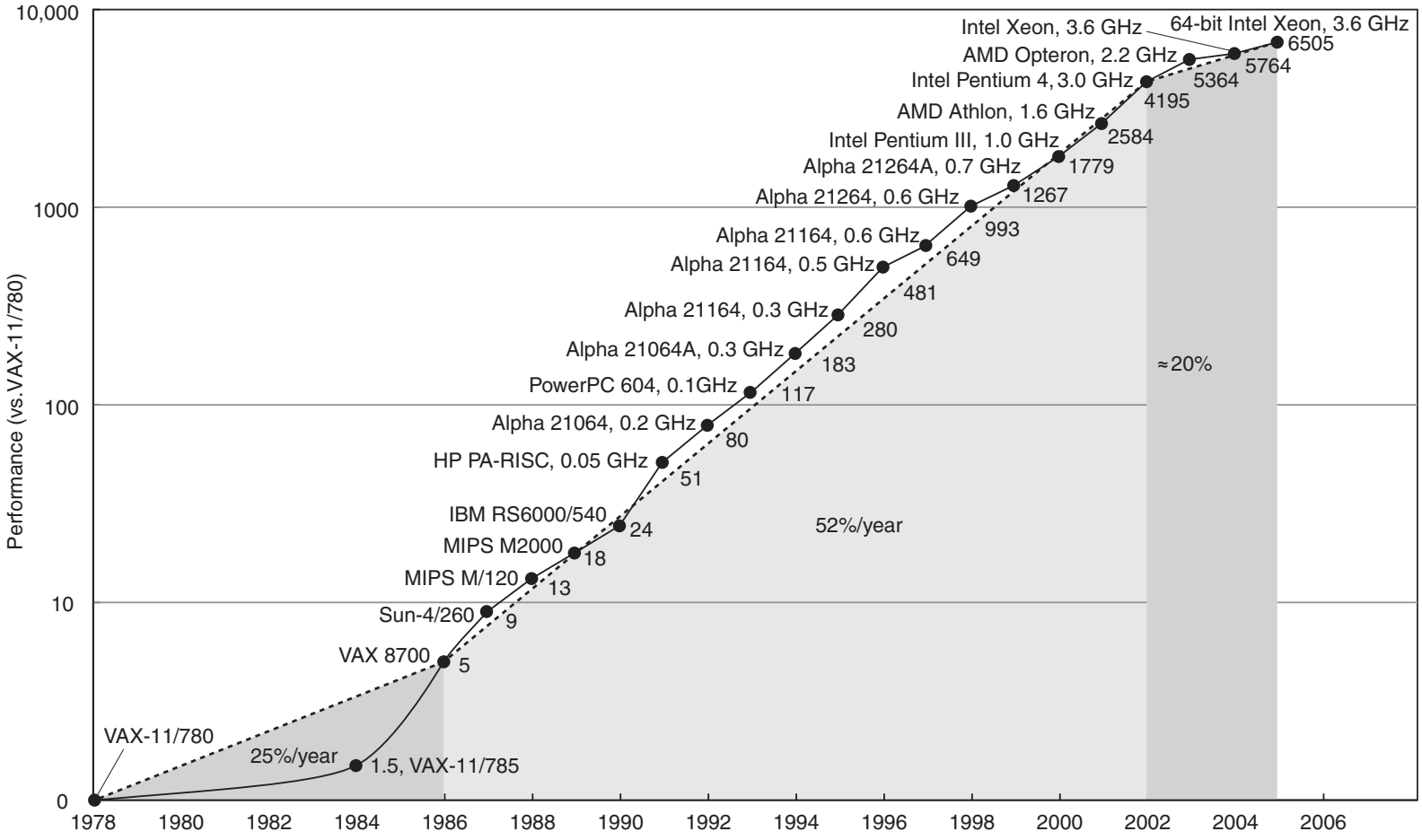


FIGURE 1.16 Growth in processor performance since the mid-1980s. This chart plots performance relative to the VAX 11/780 as measured by the SPECint benchmarks (see Section 1.8). Prior to the mid-1980s, processor performance growth was largely technology-driven and averaged about 25% per year. The increase in growth to about 52% since then is attributable to more advanced architectural and organizational ideas. By 2002, this growth led to a difference in performance of about a factor of seven. Performance for floating-point-oriented calculations has increased even faster. Since 2002, the limits of power, available instruction-level parallelism, and long memory latency have slowed uniprocessor performance recently, to about 20% per year.

History of Processor Performance

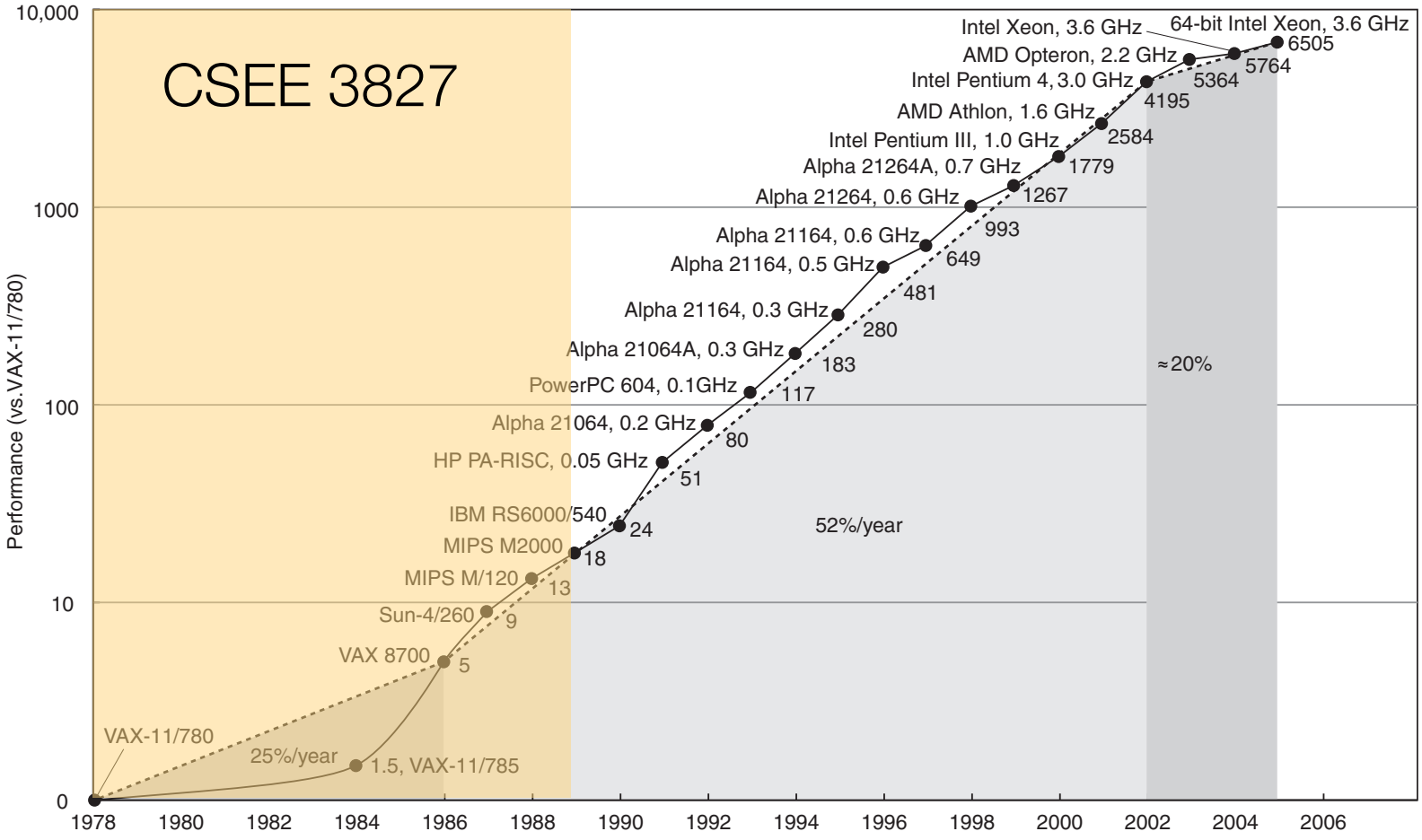


FIGURE 1.16 Growth in processor performance since the mid-1980s. This chart plots performance relative to the VAX 11/780 as measured by the SPECint benchmarks (see Section 1.8). Prior to the mid-1980s, processor performance growth was largely technology-driven and averaged about 25% per year. The increase in growth to about 52% since then is attributable to more advanced architectural and organizational ideas. By 2002, this growth led to a difference in performance of about a factor of seven. Performance for floating-point-oriented calculations has increased even faster. Since 2002, the limits of power, available instruction-level parallelism, and long memory latency have slowed uniprocessor performance recently, to about 20% per year.

History of Processor Performance

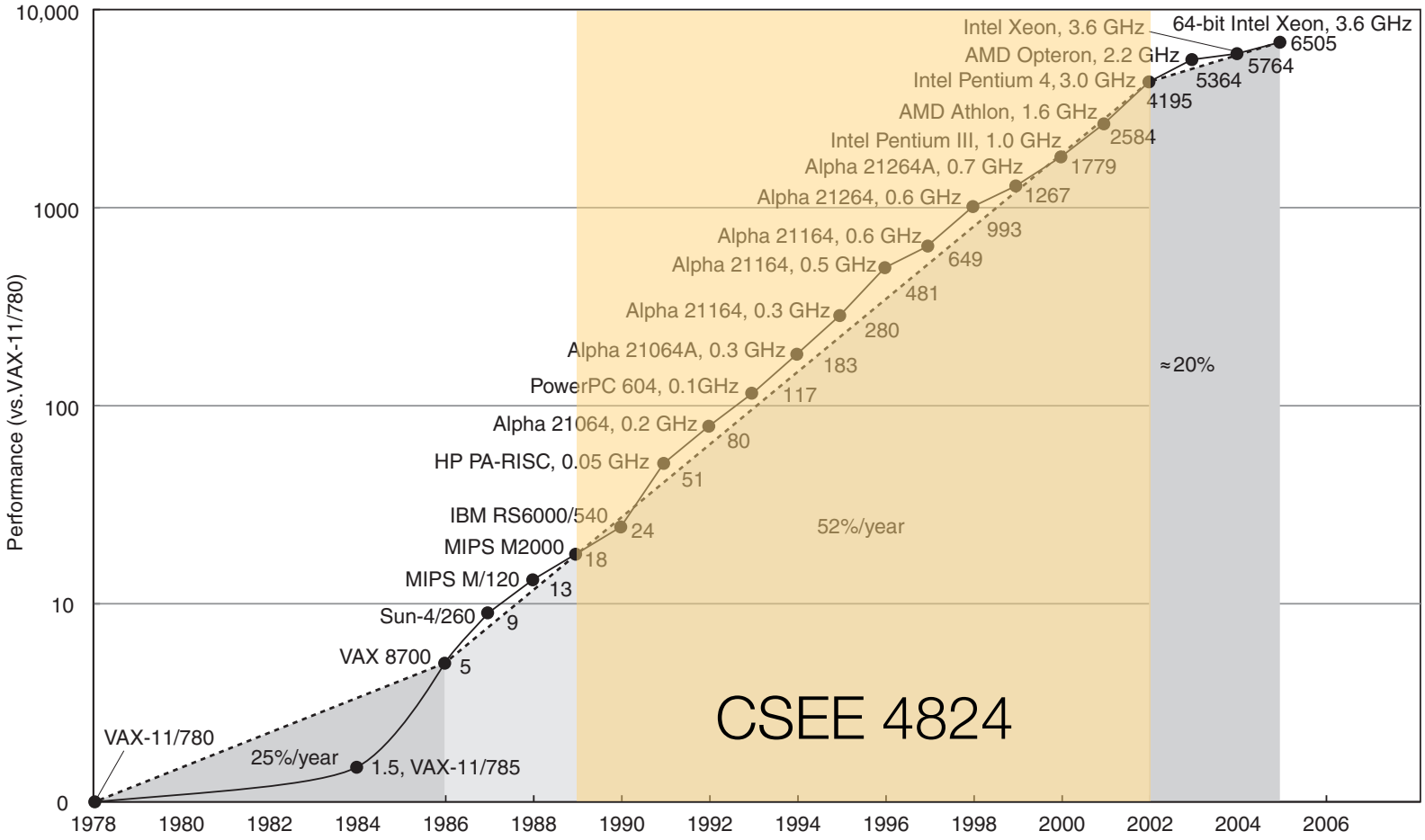


FIGURE 1.16 Growth in processor performance since the mid-1980s. This chart plots performance relative to the VAX 11/780 as measured by the SPECint benchmarks (see Section 1.8). Prior to the mid-1980s, processor performance growth was largely technology-driven and averaged about 25% per year. The increase in growth to about 52% since then is attributable to more advanced architectural and organizational ideas. By 2002, this growth led to a difference in performance of about a factor of seven. Performance for floating-point-oriented calculations has increased even faster. Since 2002, the limits of power, available instruction-level parallelism, and long memory latency have slowed uniprocessor performance recently, to about 20% per year.

Modern Processor Performance

While single threaded performance has leveled, multithreaded performance potential scaling.

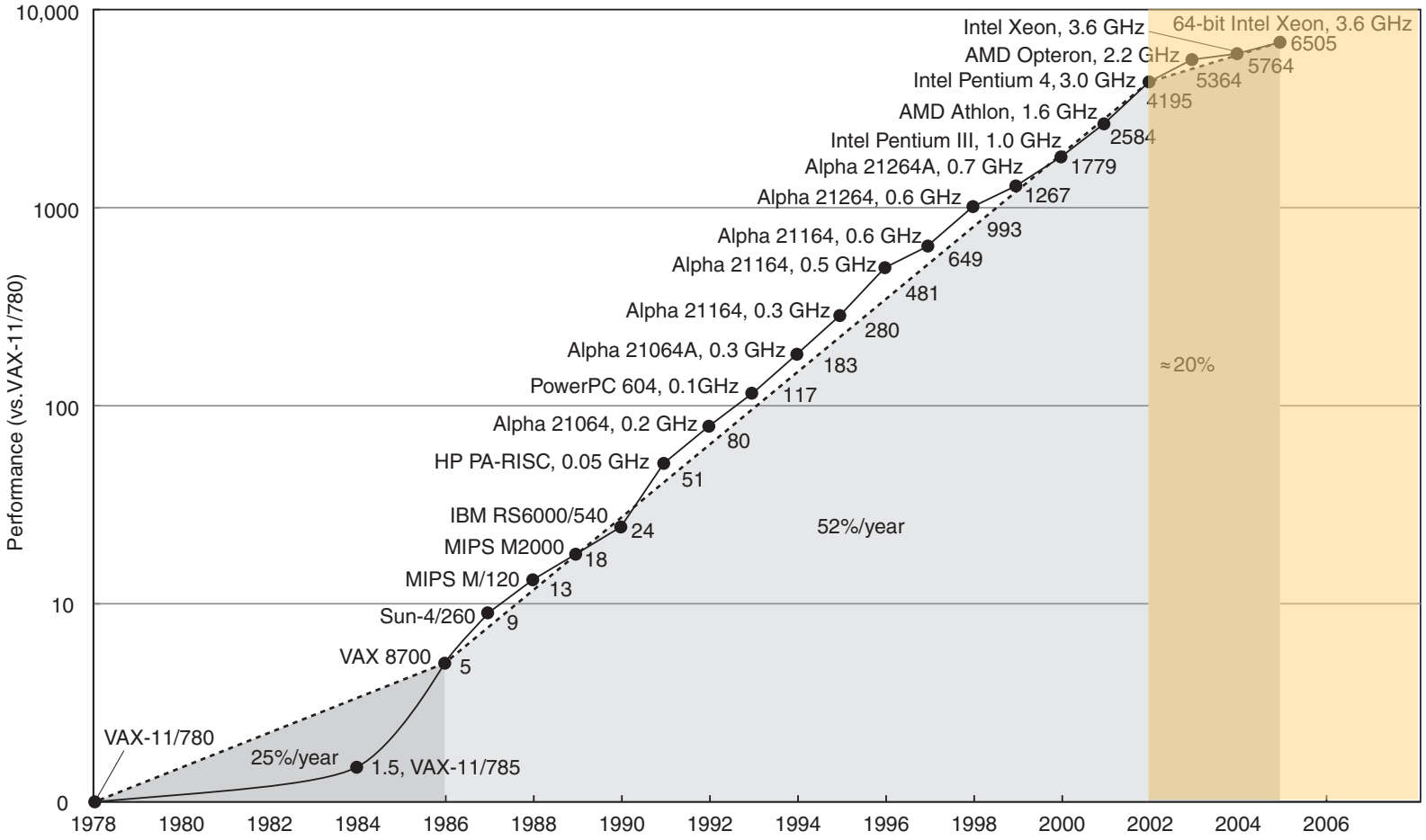
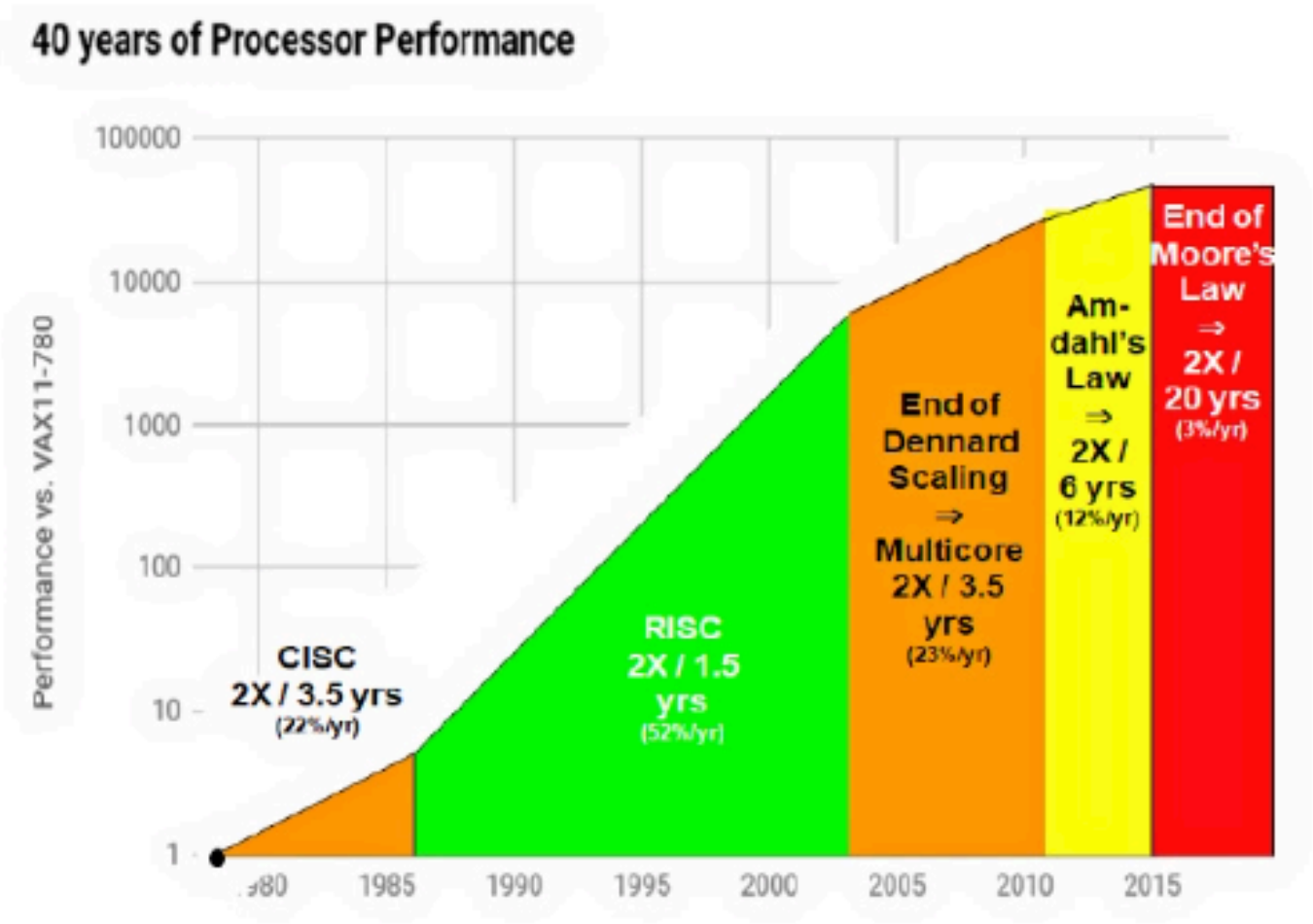


FIGURE 1.16 Growth in processor performance since the mid-1980s. This chart plots performance relative to the VAX 11/780 as measured by the SPECint benchmarks (see Section 1.8). Prior to the mid-1980s, processor performance growth was largely technology-driven and averaged about 25% per year. The increase in growth to about 52% since then is attributable to more advanced architectural and organizational ideas. By 2002, this growth led to a difference in performance of about a factor of seven. Performance for floating-point-oriented calculations has increased even faster. Since 2002, the limits of power, available instruction-level parallelism, and long memory latency have slowed uniprocessor performance recently, to about 20% per year.

Modern Processor Performance

While sing[illegible]ential scaling.



Source: John Hennessy and David Patterson, Computer Architecture: A Quantitative Approach, 6/e. 2018

Modern Processor Performance

While single threaded performance has leveled, multithreaded performance potential scaling.

GHz