# ECE260: Fundamentals of Computer Engineering

## Basics of Cache Memory

James Moscola
Dept. of Engineering & Computer Science
York College of Pennsylvania

# Cache Memory

- ***Cache memory*** – a hardware (or software) component that stores data so future requests for that data can be served faster [from Wikipedia]
  - Data stored in a cache is typically the result of an earlier access or computation
  - May be a duplicate of data stored elsewhere
  - Exploits the principle of of locality
    - Keep often used data in quickly accessible memory
    - Keep data that may be required soon in quickly accessible memory
- In hardware, cache memory is the level (or levels) of the memory hierarchy closest to the CPU
  - Exists between the processor and the main system memory (DRAM)
  - Cache memory made from fast SRAM
  - Faster, but less capacity than main system memory

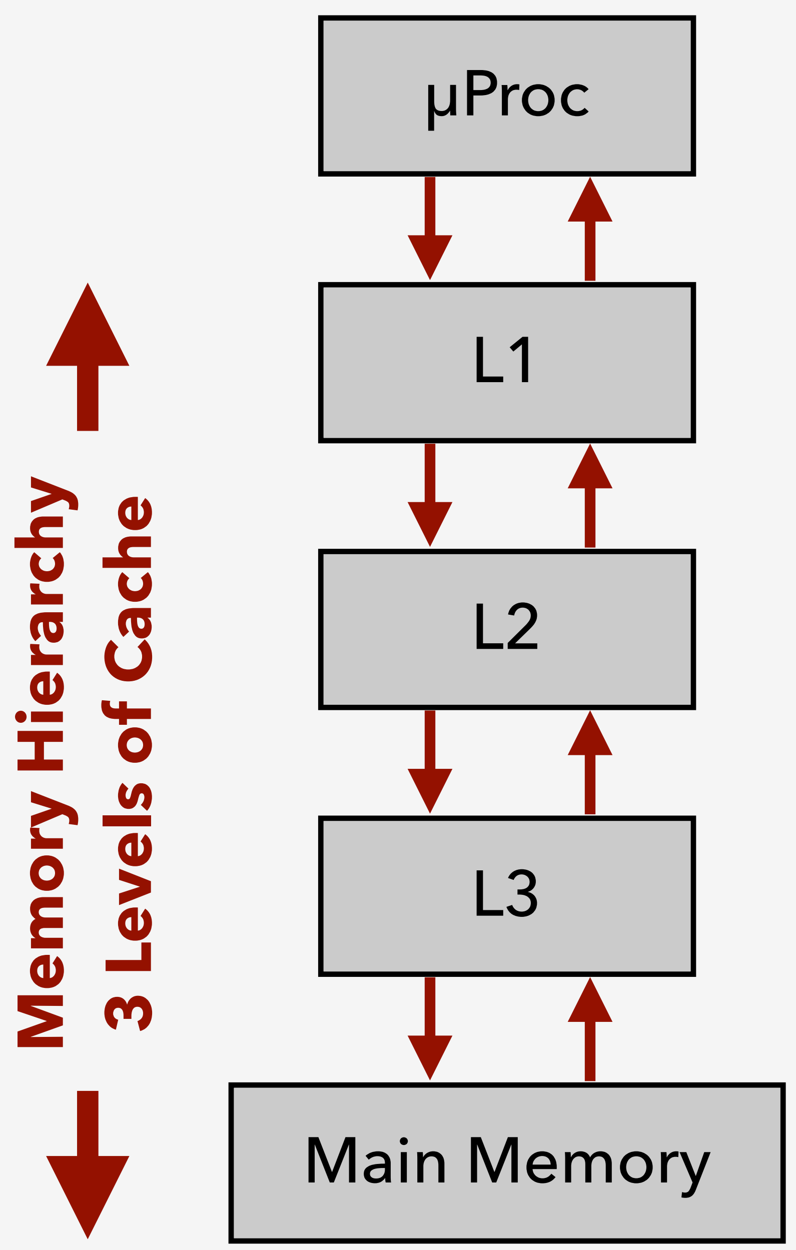
# Cache Analogy (used by author in your textbook)

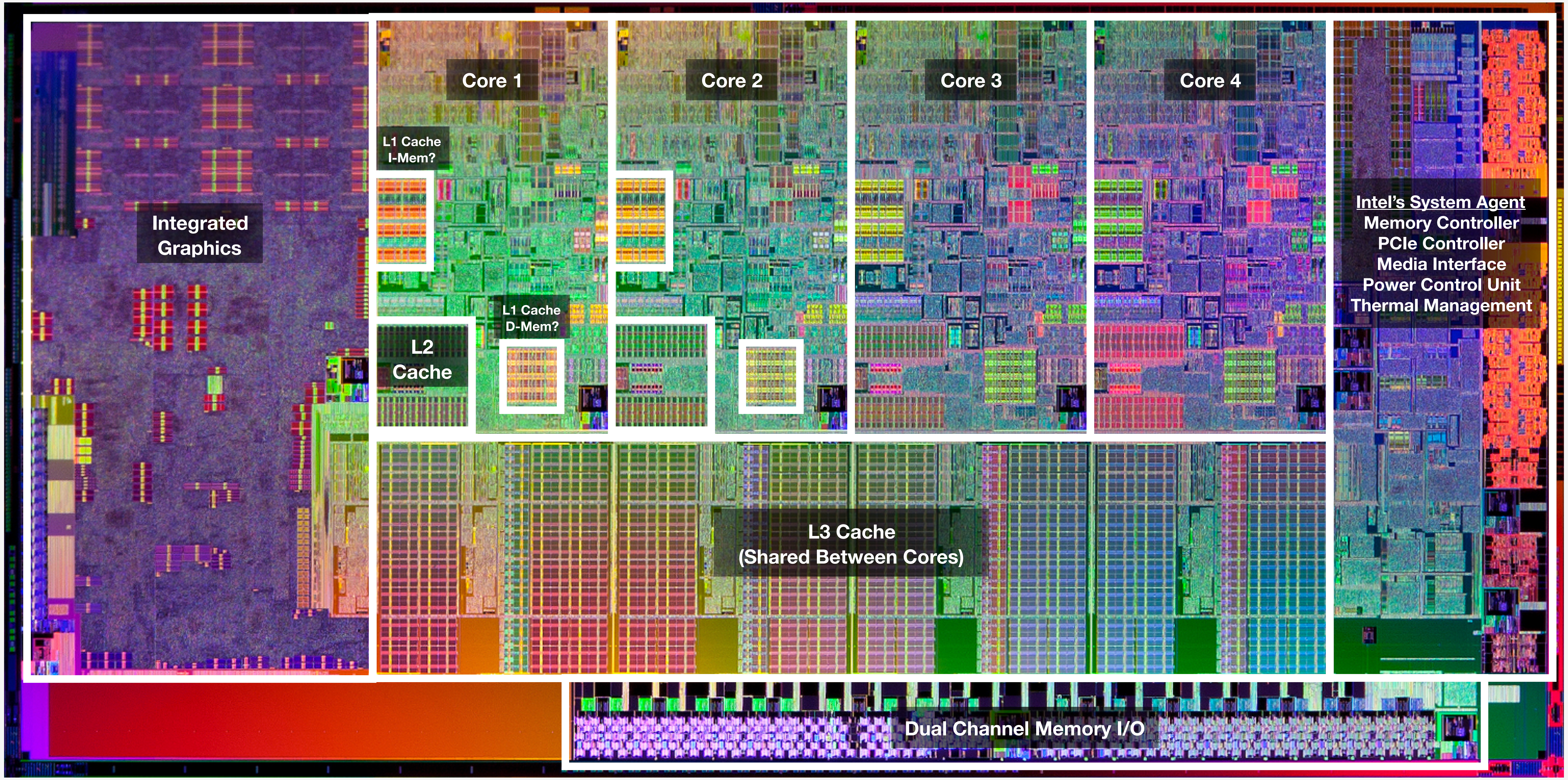
- You are writing a paper for a history course while sitting at a table in the library
  - As you work you realize you need a reference book from the library stacks
  - You stop writing, fetch the book from the stacks, and continue writing your paper
  - You don't immediately return the book, you might need it again soon (***temporal locality***)
    - You might even need multiple chapters from the same book (***spatial locality***)
  - After a short while, you have a few books on your table, and you can work smoothly without needing to fetch more books from the shelves
  - The library table is a ***cache*** for the rest of the library

- Now you switch to doing your biology homework
  - You need to fetch a new reference book for your biology from the shelf
  - If your table is full, you must return a history book to the stacks to make room for the biology book

# Cache Memory on Processor Die (Quad-Core CPU)

Processor die for Intel's Sandy Bridge Processor (circa 2011)



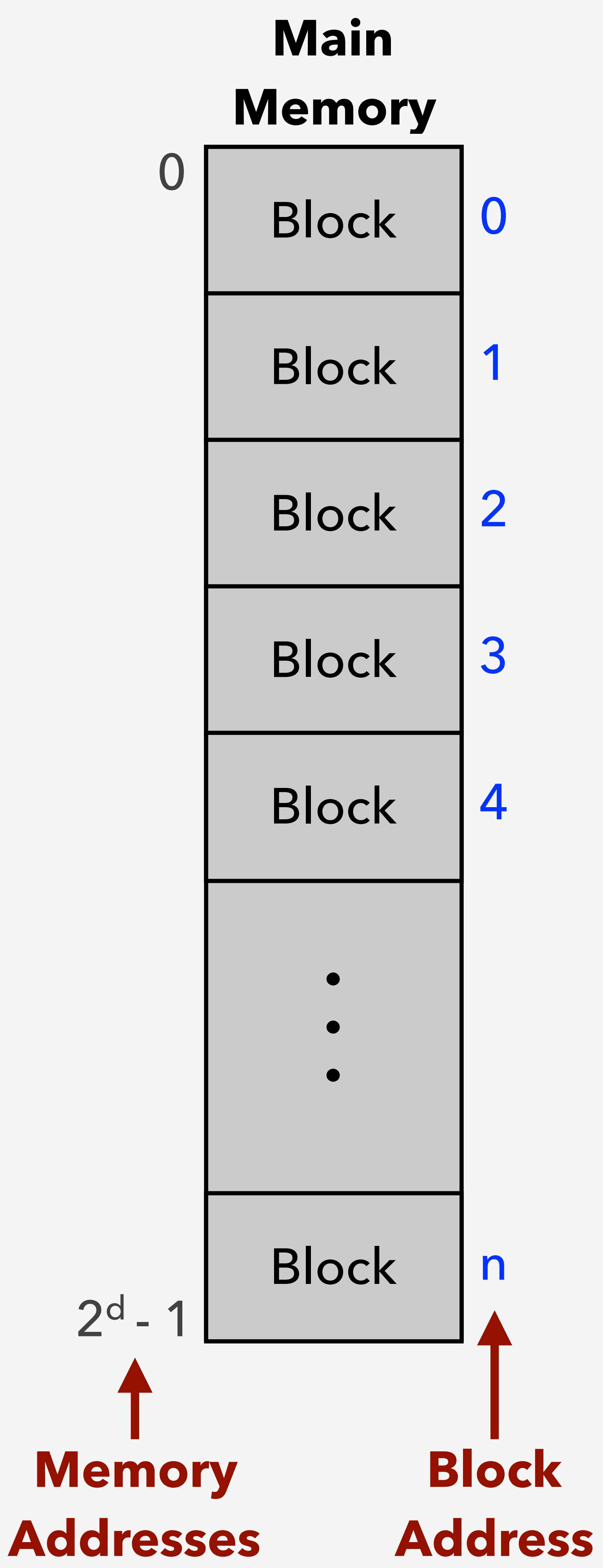



# Cache in the Memory Hierarchy

- Want to avoid reading data from slower main memory
- When processor needs to read data from a memory address
  - First, check cache memory
    - If data is **present** in cache memory (i.e. ***cache hit***), then done
    - If data is **not present** in cache (i.e. ***cache miss***), then retrieve it from main memory and copy it into the cache for use by processor
- If multiple levels of cache exist, check them in order: L1 ➜ L2 ➜ L3 ➜ Main Memory
  - Only check L2 if data not in L1
  - Only check L3 if data not in L2
  - Only check main memory if data not in L3
  - When data is found, copy it to higher levels of memory hierarchy (i.e. copy from L3 to L2 and L1)

# Main Memory in a Computer

- To understand cache, first see how main memory is treated
- Given a processor where memory is addressed with a ***d-bit address***
  - Main memory range 0 to $2^d$-1
- Main memory is divided into ***blocks***
  - Blocks are all equally sized
  - Each block contains one or more data words (i.e. each block consists of one or more memory addresses)
  - Just as each memory address has a number, each block can also be numbered




**How many *blocks* total?**

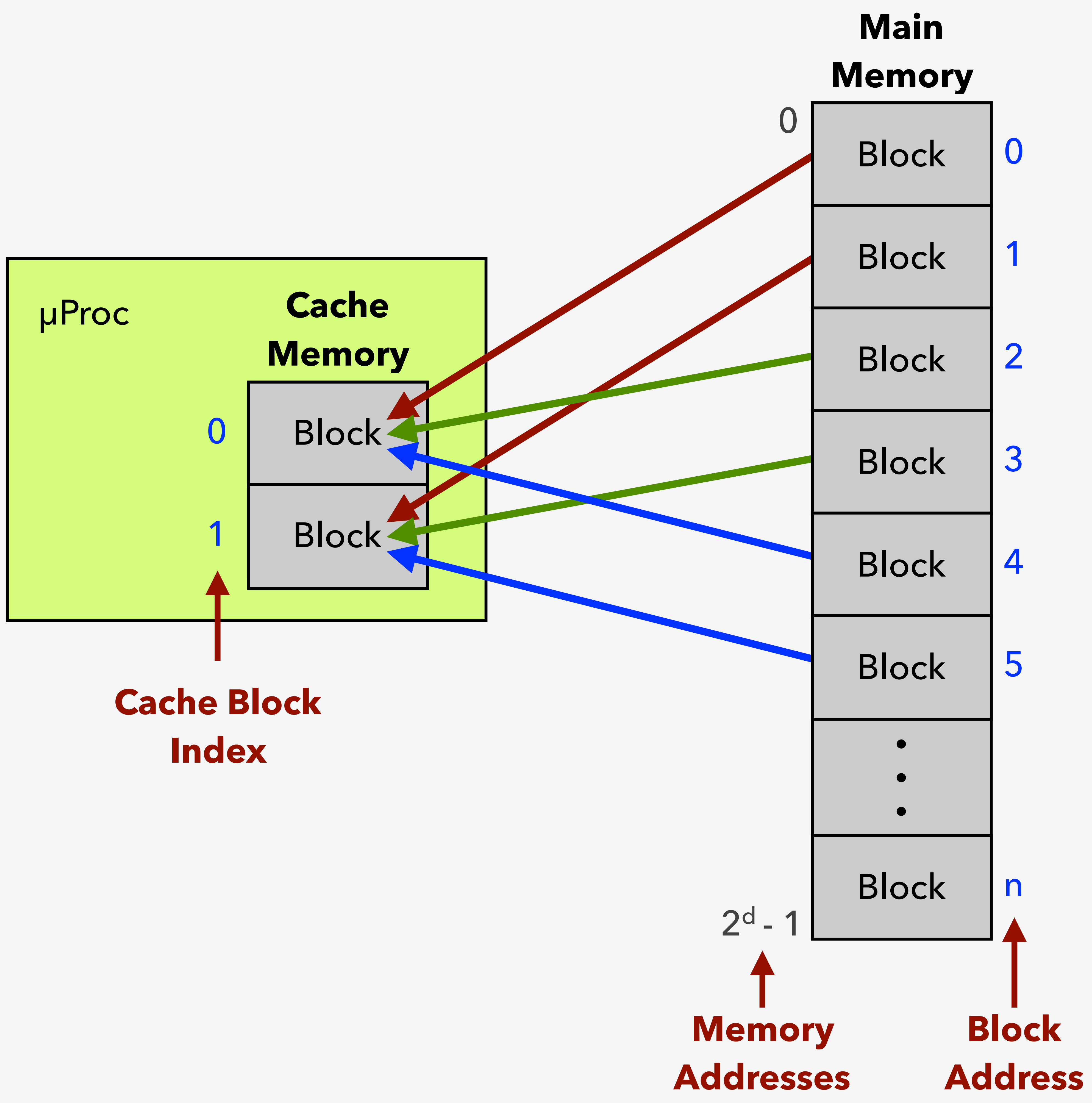
$$\#\,\text{Blocks} = \frac{\text{Total Memory Size}}{\text{Block Size}}$$

# Finding the Right Block in Main Memory

- Suppose memory address is generated – address is between 0 and $2^d$-1
- In which block is a memory address located?
  - To calculate block address, divide memory address by block size – result is the block address
    - With block size a power of 2, simply shift memory address to find block address
    - Assume block size is $2^b$ (and $b < d$)
      - Remove the least significant b bits (this is called the offset)
      - Remaining d minus b bits is the block address

# Cache Memory in a Computer

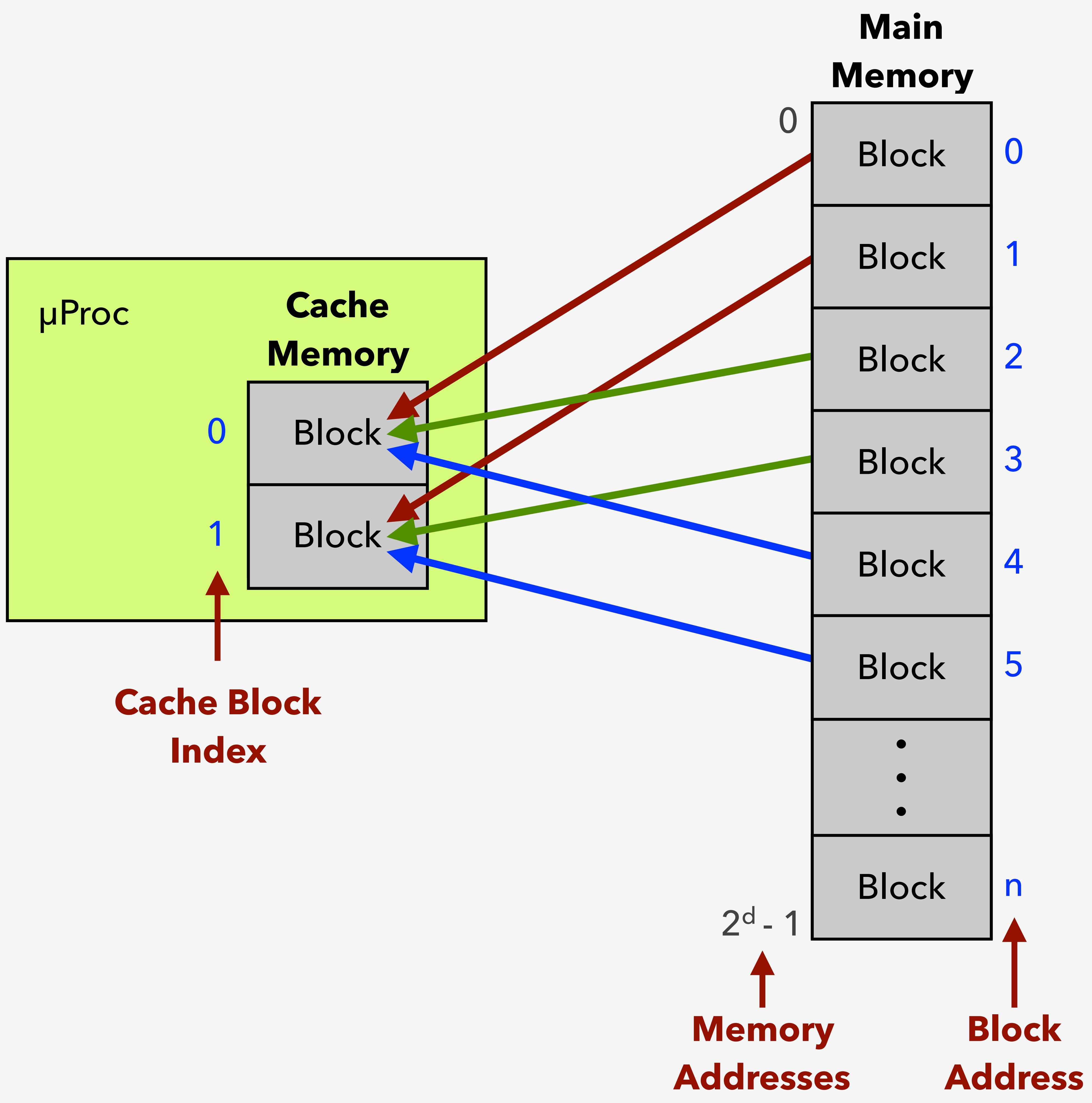
- Cache memory is divided into blocks that are exactly the same size as the the main memory blocks
- Number of blocks in cache is much less than the number of total blocks in main memory
  - Cache contains only a subset of the blocks from main memory – not large enough to store all blocks
- Many different caching techniques
  - We will focus on ***direct-mapped caching***
    - To calculate cache block index, (Block address) modulo (#Cache blocks)



# Direct-Mapped Cache

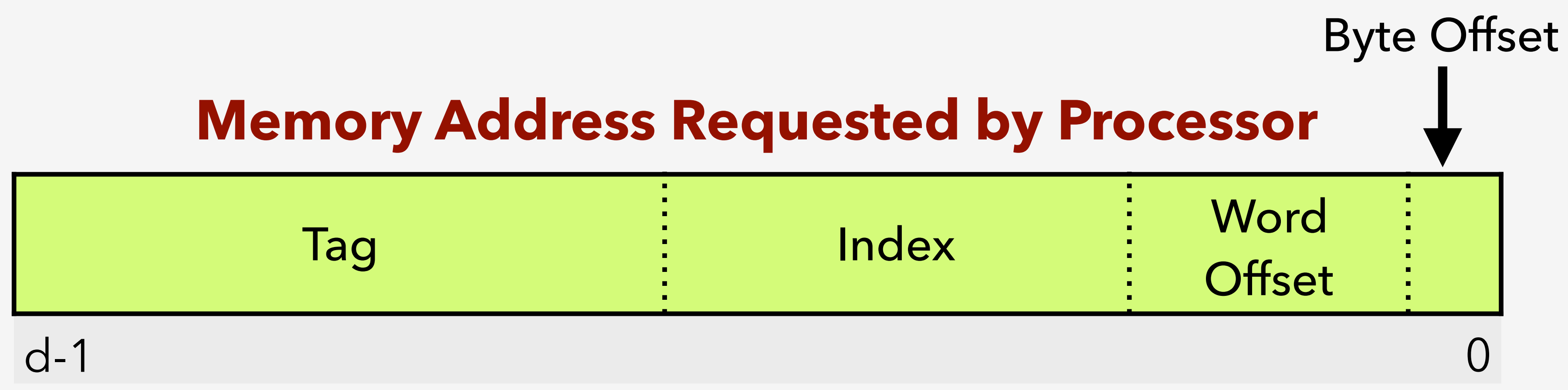
- ***Direct-mapped cache*** – a cache structure in which each memory location is mapped to exactly <u>one</u> location in the cache
  - Number of blocks is a power of 2
  - Block size is a power of 2
  - Location in cache determined by address
    - To calculate cache block index, (Block address) modulo (#Cache blocks)

**Block Index Number** = 5 modulo 2 **= 1**



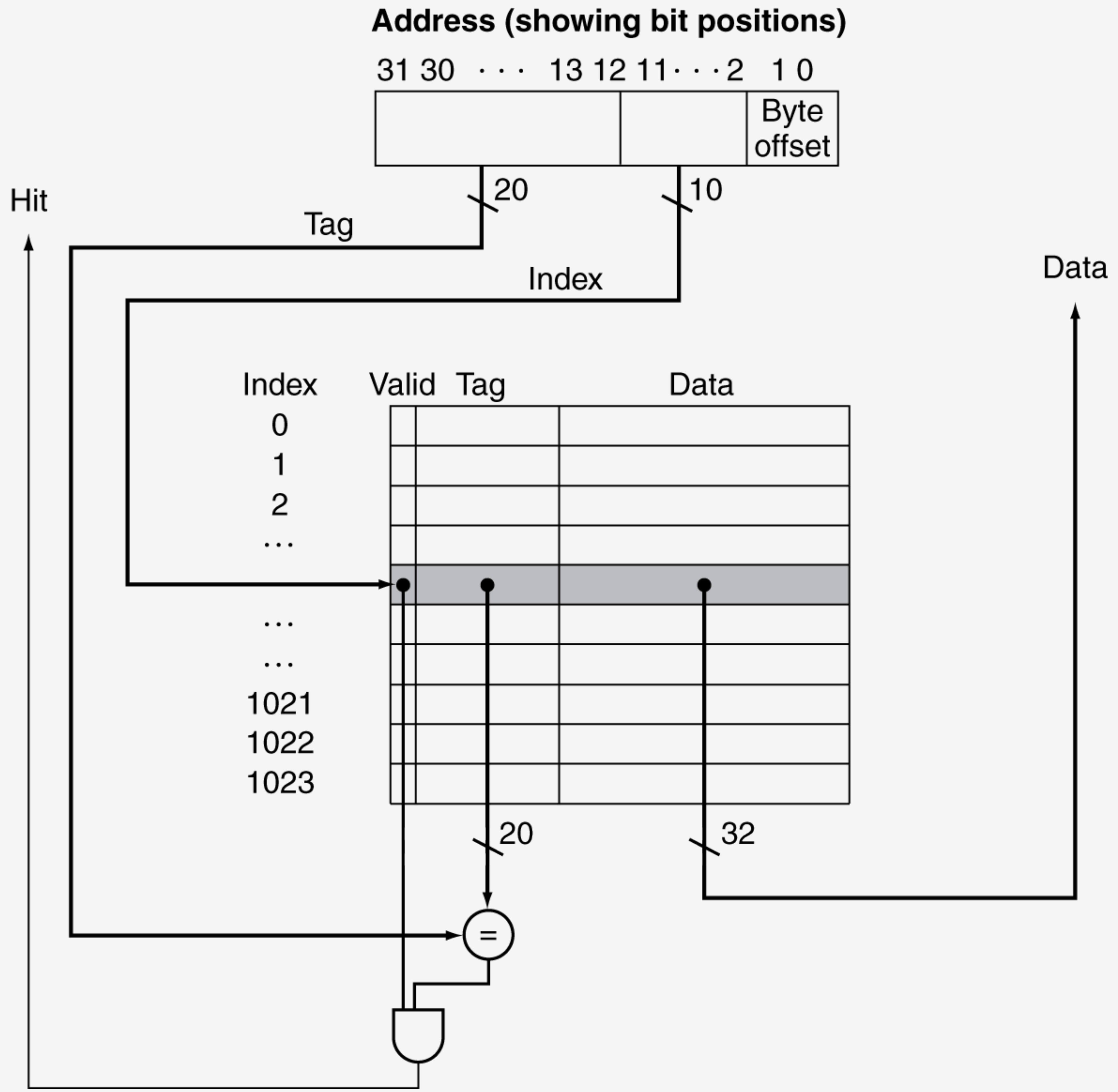
# Address Subdivision

- To utilize cache, a memory byte address is subdivided in multiple fields:
  - Byte offset
    - Specifies a byte within a word
    - Least significant two bits, typically ignored when caching
  - Word offset (if block size > 1 word)
    - Specifies a word within a block
  - Cache block index
    - Specifies which cache block a memory address should be stored
  - Tag
    - Many blocks map to the same index of cache memory
    - Tag bits uniquely identify each block (really just high-order bits of memory address)



# Cache Structure

- Each cache entry contains:
  - One or more data words
    - Determined by cache designer
  - A tag field
    - Differentiates blocks that cache to the same index in cache memory
  - A valid bit
    - Initialized to 0
    - Set to 1 when cache entry is set



# Handling Cache Misses

- On a cache hit, CPU proceeds normally
- On a cache miss, cannot proceed normally because data is not available
  - Stall the CPU pipeline
  - Fetch desired block from next level of memory hierarchy
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access

# Direct-Mapped Cache Example – Parameters

- Example of a sequence of memory references to a ***direct-mapped*** cache using word addressing
  - For this example, the cache has ***8 blocks with only 1 word per block***
  - Cache is initially empty – all valid bits are initialized to '0' to indicate that the cache block is invalid
  - Assume memory addresses are 8-bit word addresses
    - Total memory space of $2^8$ words of memory … cannot fit them all in cache
  - First, compute various field sizes

| | |
|---|---|
| **# Offset Bits** = $\log_2$(# Words/Block) | **# Offset Bits** = $\log_2$(1 Word/Block) **= 0 bits** |
| **# Index Bits** = $\log_2$(# Blocks) | **# Index Bits** = $\log_2$(8 Blocks) **= 3 bits** |
| **# Tag Bits** = # Addr Bits - # Index Bits - # Offset Bits | **# Tag Bits** = 8 Addr Bits - 3 Index Bits - 0 Offset Bits **= 5 bits** |

# Cache Example – Cache Structure

**# Offset Bits = 0 bits**
**# Index Bits = 3 bits**
**# Tag Bits = 5 bits**

**Valid bits are all initially '0' to indicate invalid entry**

**Tag bits are used to differentiate blocks that cache to the same index**

**Block index numbers range from 0 to (#Blocks - 1)**

**The data from the main memory that is being cached**

**Cache is structured as a table that is indexed using block numbers.**

**For this example, block index numbers are shown in both binary and $base_{10}$**

| Block Index | V | Tag | Data |
|---|---|---|---|
| $000_{two}$ $(0_{ten})$ | 0 | | |
| $001_{two}$ $(1_{ten})$ | 0 | | |
| $010_{two}$ $(2_{ten})$ | 0 | | |
| $011_{two}$ $(3_{ten})$ | 0 | | |
| $100_{two}$ $(4_{ten})$ | 0 | | |
| $101_{two}$ $(5_{ten})$ | 0 | | |
| $110_{two}$ $(6_{ten})$ | 0 | | |
| $111_{two}$ $(7_{ten})$ | 0 | | |

# Cache Example – First Memory Access

- Access memory address $22_{ten}$ **(NOTE: this is a word address)**

| Word Addr | Word Addr (base-2) | Tag | Cache Block Index # | Hit / Miss |
|---|---|---|---|---|
| $22_{ten}$ | $00010110_{two}$ | $00010_{two}$ | $110_{two}$ ($6_{ten}$) | Miss |

**# Offset Bits = 0 bits**
**# Index Bits = 3 bits**
**# Tag Bits = 5 bits**

**CACHE MEMORY (before access)**

| Block Index | V | Tag | Data |
|---|---|---|---|
| $000_{two}$ ($0_{ten}$) | 0 | | |
| $001_{two}$ ($1_{ten}$) | 0 | | |
| $010_{two}$ ($2_{ten}$) | 0 | | |
| $011_{two}$ ($3_{ten}$) | 0 | | |
| $100_{two}$ ($4_{ten}$) | 0 | | |
| $101_{two}$ ($5_{ten}$) | 0 | | |
| $110_{two}$ ($6_{ten}$) | 0 | | |
| $111_{two}$ ($7_{ten}$) | 0 | | |

**Block index number $6_{ten}$ is set to invalid, therefore this is a *cache miss*.**

**CACHE MEMORY (after access)**

| Block Index | V | Tag | Data |
|---|---|---|---|
| $000_{two}$ ($0_{ten}$) | 0 | | |
| $001_{two}$ ($1_{ten}$) | 0 | | |
| $010_{two}$ ($2_{ten}$) | 0 | | |
| $011_{two}$ ($3_{ten}$) | 0 | | |
| $100_{two}$ ($4_{ten}$) | 0 | | |
| $101_{two}$ ($5_{ten}$) | 0 | | |
| $110_{two}$ ($6_{ten}$) | 1 | $00010_{two}$ | mem[22] |
| $111_{two}$ ($7_{ten}$) | 0 | | |

**Copy data from main system memory into cache, *set tag and valid bit*. Resume original instruction that needed the data.**

# Cache Example – Second Memory Access

- Access memory address $26_{ten}$

| Word Addr | Word Addr (base-2) | Tag | Cache Block Index # | Hit / Miss |
|---|---|---|---|---|
| $26_{ten}$ | $00011010_{two}$ | $00011_{two}$ | $010_{two}$ ($2_{ten}$) | Miss |

**# Offset Bits = 0 bits**
**# Index Bits = 3 bits**
**# Tag Bits = 5 bits**

**CACHE MEMORY (before access)**

| Block Index | V | Tag | Data |
|---|---|---|---|
| $000_{two}$ ($0_{ten}$) | 0 | | |
| $001_{two}$ ($1_{ten}$) | 0 | | |
| $010_{two}$ ($2_{ten}$) | 0 | | |
| $011_{two}$ ($3_{ten}$) | 0 | | |
| $100_{two}$ ($4_{ten}$) | 0 | | |
| $101_{two}$ ($5_{ten}$) | 0 | | |
| $110_{two}$ ($6_{ten}$) | 1 | $00010_{two}$ | mem[22] |
| $111_{two}$ ($7_{ten}$) | 0 | | |

**Block index number $2_{ten}$ is set to invalid, therefore this is a *cache miss*.**

**CACHE MEMORY (after access)**

| Block Index | V | Tag | Data |
|---|---|---|---|
| $000_{two}$ ($0_{ten}$) | 0 | | |
| $001_{two}$ ($1_{ten}$) | 0 | | |
| $010_{two}$ ($2_{ten}$) | 1 | $00011_{two}$ | mem[26] |
| $011_{two}$ ($3_{ten}$) | 0 | | |
| $100_{two}$ ($4_{ten}$) | 0 | | |
| $101_{two}$ ($5_{ten}$) | 0 | | |
| $110_{two}$ ($6_{ten}$) | 1 | $00010_{two}$ | mem[22] |
| $111_{two}$ ($7_{ten}$) | 0 | | |

**Copy data from main system memory into cache, *set tag and valid bit*. Resume original instruction that needed the data.**

# Cache Example – Third Memory Access

- Access memory address $22_{ten}$ (again)

| Word Addr | Word Addr (base-2) | Tag | Cache Block Index # | Hit / Miss |
|---|---|---|---|---|
| $22_{ten}$ | $00010110_{two}$ | $00010_{two}$ | $110_{two}$ ($6_{ten}$) | Hit |

**# Offset Bits = 0 bits**
**# Index Bits = 3 bits**
**# Tag Bits = 5 bits**

**CACHE MEMORY (before access)**

| Block Index | V | Tag | Data |
|---|---|---|---|
| $000_{two}$ ($0_{ten}$) | 0 | | |
| $001_{two}$ ($1_{ten}$) | 0 | | |
| $010_{two}$ ($2_{ten}$) | 1 | $00011_{two}$ | mem[26] |
| $011_{two}$ ($3_{ten}$) | 0 | | |
| $100_{two}$ ($4_{ten}$) | 0 | | |
| $101_{two}$ ($5_{ten}$) | 0 | | |
| $110_{two}$ ($6_{ten}$) | 1 | $00010_{two}$ | mem[22] |
| $111_{two}$ ($7_{ten}$) | 0 | | |

**Block index number $6_{ten}$ is set to valid AND the tag in the cache memory matches the tag field of the requested memory address, therefore this is a *cache hit*.**

**CACHE MEMORY (after access)**

| Block Index | V | Tag | Data |
|---|---|---|---|
| $000_{two}$ ($0_{ten}$) | 0 | | |
| $001_{two}$ ($1_{ten}$) | 0 | | |
| $010_{two}$ ($2_{ten}$) | 1 | $00011_{two}$ | mem[26] |
| $011_{two}$ ($3_{ten}$) | 0 | | |
| $100_{two}$ ($4_{ten}$) | 0 | | |
| $101_{two}$ ($5_{ten}$) | 0 | | |
| $110_{two}$ ($6_{ten}$) | **1** | **$00010_{two}$** | **mem[22]** |
| $111_{two}$ ($7_{ten}$) | 0 | | |

**No need to access main system memory. Provide requested data to processor.**

# Cache Example – Fourth Memory Access

- Access memory address $26_{ten}$ (again)

| Word Addr | Word Addr (base-2) | Tag | Cache Block Index # | Hit / Miss |
|---|---|---|---|---|
| $26_{ten}$ | $00011010_{two}$ | $00011_{two}$ | $010_{two}$ ($2_{ten}$) | Hit |

**# Offset Bits = 0 bits**
**# Index Bits = 3 bits**
**# Tag Bits = 5 bits**

**CACHE MEMORY (before access)**

| Block Index | V | Tag | Data |
|---|---|---|---|
| $000_{two}$ ($0_{ten}$) | 0 | | |
| $001_{two}$ ($1_{ten}$) | 0 | | |
| $010_{two}$ ($2_{ten}$) | 1 | $00011_{two}$ | mem[26] |
| $011_{two}$ ($3_{ten}$) | 0 | | |
| $100_{two}$ ($4_{ten}$) | 0 | | |
| $101_{two}$ ($5_{ten}$) | 0 | | |
| $110_{two}$ ($6_{ten}$) | 1 | $00010_{two}$ | mem[22] |
| $111_{two}$ ($7_{ten}$) | 0 | | |

**Block index number $2_{ten}$ is set to valid AND the tag in the cache memory matches the tag field of the requested memory address, therefore this is a *cache hit*.**

**CACHE MEMORY (after access)**

| Block Index | V | Tag | Data |
|---|---|---|---|
| $000_{two}$ ($0_{ten}$) | 0 | | |
| $001_{two}$ ($1_{ten}$) | 0 | | |
| $010_{two}$ ($2_{ten}$) | **1** | **$00011_{two}$** | **mem[26]** |
| $011_{two}$ ($3_{ten}$) | 0 | | |
| $100_{two}$ ($4_{ten}$) | 0 | | |
| $101_{two}$ ($5_{ten}$) | 0 | | |
| $110_{two}$ ($6_{ten}$) | 1 | $00010_{two}$ | mem[22] |
| $111_{two}$ ($7_{ten}$) | 0 | | |

**No need to access main system memory. Provide requested data to processor.**

# Cache Example – Fifth Memory Access

- Access memory address $16_{ten}$

| Word Addr | Word Addr (base-2) | Tag | Cache Block Index # | Hit / Miss |
|---|---|---|---|---|
| $16_{ten}$ | $00010000_{two}$ | $00010_{two}$ | $000_{two}$ $(0_{ten})$ | Miss |

**# Offset Bits = 0 bits**
**# Index Bits = 3 bits**
**# Tag Bits = 5 bits**

**CACHE MEMORY (before access)**

| Block Index | V | Tag | Data |
|---|---|---|---|
| $000_{two}$ $(0_{ten})$ | 0 | | |
| $001_{two}$ $(1_{ten})$ | 0 | | |
| $010_{two}$ $(2_{ten})$ | 1 | $00011_{two}$ | mem[26] |
| $011_{two}$ $(3_{ten})$ | 0 | | |
| $100_{two}$ $(4_{ten})$ | 0 | | |
| $101_{two}$ $(5_{ten})$ | 0 | | |
| $110_{two}$ $(6_{ten})$ | 1 | $00010_{two}$ | mem[22] |
| $111_{two}$ $(7_{ten})$ | 0 | | |

**Block index number $0_{ten}$ is set to invalid, therefore this is a *cache miss*.**

**CACHE MEMORY (after access)**

| Block Index | V | Tag | Data |
|---|---|---|---|
| $000_{two}$ $(0_{ten})$ | **1** | **$00010_{two}$** | **mem[16]** |
| $001_{two}$ $(1_{ten})$ | 0 | | |
| $010_{two}$ $(2_{ten})$ | 1 | $00011_{two}$ | mem[26] |
| $011_{two}$ $(3_{ten})$ | 0 | | |
| $100_{two}$ $(4_{ten})$ | 0 | | |
| $101_{two}$ $(5_{ten})$ | 0 | | |
| $110_{two}$ $(6_{ten})$ | 1 | $00010_{two}$ | mem[22] |
| $111_{two}$ $(7_{ten})$ | 0 | | |

**Copy data from main system memory into cache, *set tag and valid bit*. Resume original instruction that needed the data.**

# Cache Example – Sixth Memory Access

- Access memory address $18_{ten}$

| Word Addr | Word Addr (base-2) | Tag | Cache Block Index # | Hit / Miss |
|---|---|---|---|---|
| $18_{ten}$ | **$00010010_{two}$** | **$00010_{two}$** | **$010_{two}$ ($2_{ten}$)** | **Miss** |

**# Offset Bits = 0 bits**
**# Index Bits = 3 bits**
**# Tag Bits = 5 bits**

**CACHE MEMORY (before access)**

| Block Index | V | Tag | Data |
|---|---|---|---|
| $000_{two}$ ($0_{ten}$) | 1 | $00010_{two}$ | mem[16] |
| $001_{two}$ ($1_{ten}$) | 0 | | |
| $010_{two}$ ($2_{ten}$) | 1 | $00011_{two}$ | mem[26] |
| $011_{two}$ ($3_{ten}$) | 0 | | |
| $100_{two}$ ($4_{ten}$) | 0 | | |
| $101_{two}$ ($5_{ten}$) | 0 | | |
| $110_{two}$ ($6_{ten}$) | 1 | $00010_{two}$ | mem[22] |
| $111_{two}$ ($7_{ten}$) | 0 | | |

**CACHE MEMORY (after access)**

| Block Index | V | Tag | Data |
|---|---|---|---|
| $000_{two}$ ($0_{ten}$) | 1 | $00010_{two}$ | mem[16] |
| $001_{two}$ ($1_{ten}$) | 0 | | |
| $010_{two}$ ($2_{ten}$) | 1 | **$00010_{two}$** | **mem[18]** |
| $011_{two}$ ($3_{ten}$) | 0 | | |
| $100_{two}$ ($4_{ten}$) | 0 | | |
| $101_{two}$ ($5_{ten}$) | 0 | | |
| $110_{two}$ ($6_{ten}$) | 1 | $00010_{two}$ | mem[22] |
| $111_{two}$ ($7_{ten}$) | 0 | | |

**Block index number $2_{ten}$ is set to valid. HOWEVER, the tag in the cache DOES NOT MATCH the tag field of the requested memory address, therefore this is a *cache miss*.**

**Copy data from main system memory into cache *OVERWRITING* previous contents, set new tag value. Resume original instruction that needed the data.**

# Direct-Mapped Cache Example #2 – Parameters

- Example of a sequence of memory references to a ***direct-mapped*** cache using byte addressing
  - For this example, the cache has ***4 blocks with 16 bytes per block (i.e. 4 words per block)***
  - Cache is initially empty – all valid bits are initialized to '0' to indicate that the cache block is invalid
  - Assume memory addresses are 8-bit byte addresses
    - Two least significant bits specify byte offset within a word – remove these bits
    - Total memory space of $2^8$ bytes of memory … cannot fit them all in cache
  - First, compute various field sizes

| | |
|---|---|
| **# Offset Bits** = $\log_2$ (# Words/Block) | **# Offset Bits** = $\log_2$ (4 Words/Block) **= 2 bits** |
| **# Index Bits** = $\log_2$ (# Blocks) | **# Index Bits** = $\log_2$ (44 Blocks) **= 2 bits** |
| **# Tag Bits** = # Addr Bits - # Index Bits - # Offset Bits - Byte Offset Bits | **# Tag Bits** = 8 Addr Bits - 2 Index Bits - 2 Offset Bits - 2 Byte Offset Bits **= 2 bits** |

# Cache Example #2 – Cache Structure

- This cache structure stores multiple data words per block
- When one of the data words is accessed, the entire block is brought into the cache from main system memory

**# Offset Bits = 2 bits**
**# Index Bits = 2 bits**
**# Tag Bits = 2 bits**

| **Block Index** | **V** | **Tag** | **Data (4 words)** Offset 3 | Offset 2 | Offset 1 | Offset 0 |
|---|---|---|---|---|---|---|
| $00_{two}$ $(0_{ten})$ | 0 | | | | | |
| $01_{two}$ $(1_{ten})$ | 0 | | | | | |
| $10_{two}$ $(2_{ten})$ | 0 | | | | | |
| $11_{two}$ $(3_{ten})$ | 0 | | | | | |

# Cache Example #2 – First Memory Access

- Access byte address $12_{ten}$ (NOTE: this is a byte address, i.e. word address $3_{ten}$)

| Byte Addr | Byte Addr (base-2) | Tag | Cache Block Index # | Offset | Hit / Miss |
|---|---|---|---|---|---|
| $12_{ten}$ | $000011\cancel{00}_{two}$ | $00_{two}$ | $00_{two}$ ($0_{ten}$) | $11_{two}$ ($3_{ten}$) | Miss |

**# Offset Bits = 2 bits**
**# Index Bits = 2 bits**
**# Tag Bits = 2 bits**

**CACHE MEMORY (before access)**

| Block Index | V | Tag | Data (4 words) Offset 3 | Offset 2 | Offset 1 | Offset 0 |
|---|---|---|---|---|---|---|
| $00_{two}$ ($0_{ten}$) | 0 | | | | | |
| $01_{two}$ ($1_{ten}$) | 0 | | | | | |
| $10_{two}$ ($2_{ten}$) | 0 | | | | | |
| $11_{two}$ ($3_{ten}$) | 0 | | | | | |

**Block index number $0_{ten}$ is set to invalid, therefore this is a *cache miss*.**

**CACHE MEMORY (after access)**

| Block Index | V | Tag | Data (4 words) Offset 3 | Offset 2 | Offset 1 | Offset 0 |
|---|---|---|---|---|---|---|
| $00_{two}$ ($0_{ten}$) | 1 | $00_{two}$ | mem[12] | mem[8] | mem[4] | mem[0] |
| $01_{two}$ ($1_{ten}$) | 0 | | | | | |
| $10_{two}$ ($2_{ten}$) | 0 | | | | | |
| $11_{two}$ ($3_{ten}$) | 0 | | | | | |

**On cache miss, find the block in which address $12_{ten}$ resides, *copy entire block* from main system memory into cache. Also, *set tag and valid bit* and resume original instruction that needed data.**

# Cache Example #2 – Second Memory Access

- Access byte address $104_{ten}$ **(NOTE: this is a byte address, i.e. word address $26_{ten}$)**

| Byte Addr | Byte Addr (base-2) | Tag | Cache Block Index # | Offset | Hit / Miss |
|---|---|---|---|---|---|
| $104_{ten}$ | $011010\cancel{00}_{two}$ | $01_{two}$ | $10_{two}$ $(2_{ten})$ | $10_{two}$ $(2_{ten})$ | **Miss** |

**# Offset Bits = 2 bits**
**# Index Bits = 2 bits**
**# Tag Bits = 2 bits**

**CACHE MEMORY (before access)**

| Block Index | V | Tag | Data (4 words): Offset 3 | Offset 2 | Offset 1 | Offset 0 |
|---|---|---|---|---|---|---|
| $00_{two}$ $(0_{ten})$ | 1 | $00_{two}$ | mem[12] | mem[8] | mem[4] | mem[0] |
| $01_{two}$ $(1_{ten})$ | 0 | | | | | |
| $10_{two}$ $(2_{ten})$ | 0 | | | | | |
| $11_{two}$ $(3_{ten})$ | 0 | | | | | |

**Block index number $2_{ten}$ is set to invalid, therefore this is a *cache miss*.**

**CACHE MEMORY (after access)**

| Block Index | V | Tag | Data (4 words): Offset 3 | Offset 2 | Offset 1 | Offset 0 |
|---|---|---|---|---|---|---|
| $00_{two}$ $(0_{ten})$ | 1 | $00_{two}$ | mem[12] | mem[8] | mem[4] | mem[0] |
| $01_{two}$ $(1_{ten})$ | 0 | | | | | |
| $10_{two}$ $(2_{ten})$ | **1** | **$01_{two}$** | mem[108] | **mem[104]** | mem[100] | mem[96] |
| $11_{two}$ $(3_{ten})$ | 0 | | | | | |

**On cache miss, find the block in which address $104_{ten}$ resides, *copy entire block* from main system memory into cache. Also, *set tag and valid bit* and resume original instruction that needed data.**

# Cache Example #2 – Third Memory Access

- Access byte address $96_{ten}$ (NOTE: this is a byte address, i.e. word address $24_{ten}$)

| Byte Addr | Byte Addr (base-2) | Tag | Cache Block Index # | Offset | Hit / Miss |
|---|---|---|---|---|---|
| $96_{ten}$ | $011000\cancel{00}_{two}$ | $01_{two}$ | $10_{two}$ $(2_{ten})$ | $00_{two}$ $(2_{ten})$ | Hit |

**# Offset Bits = 2 bits**
**# Index Bits = 2 bits**
**# Tag Bits = 2 bits**

**CACHE MEMORY (before access)**

| Block Index | V | Tag | Data (4 words): Offset 3 | Offset 2 | Offset 1 | Offset 0 |
|---|---|---|---|---|---|---|
| $00_{two}$ $(0_{ten})$ | 1 | $00_{two}$ | mem[12] | mem[8] | mem[4] | mem[0] |
| $01_{two}$ $(1_{ten})$ | 0 | | | | | |
| $10_{two}$ $(2_{ten})$ | 1 | $01_{two}$ | mem[108] | mem[104] | mem[100] | mem[96] |
| $11_{two}$ $(3_{ten})$ | 0 | | | | | |

**Block index number $2_{ten}$ is set to valid AND the tag in the cache memory matches the tag field of the requested memory address, therefore this is a *cache hit*.**

**CACHE MEMORY (after access)**

| Block Index | V | Tag | Data (4 words): Offset 3 | Offset 2 | Offset 1 | Offset 0 |
|---|---|---|---|---|---|---|
| $00_{two}$ $(0_{ten})$ | 1 | $00_{two}$ | mem[12] | mem[8] | mem[4] | mem[0] |
| $01_{two}$ $(1_{ten})$ | 0 | | | | | |
| $10_{two}$ $(2_{ten})$ | **1** | **$01_{two}$** | mem[108] | mem[104] | mem[100] | **mem[96]** |
| $11_{two}$ $(3_{ten})$ | 0 | | | | | |

**On cache miss, find the block in which address $104_{ten}$ resides, copy entire block from main system memory into cache. Also, *set tag and valid bit* and resume original instruction that needed data.**

# Cache Example #2 – Third Memory Access

- Access byte address $172_{ten}$ (NOTE: this is a byte address, i.e. word address $43_{ten}$)

**# Offset Bits = 2 bits**
**# Index Bits = 2 bits**
**# Tag Bits = 2 bits**

| Byte Addr | Byte Addr (base-2) | Tag | Cache Block Index # | Offset | Hit / Miss |
|---|---|---|---|---|---|
| $172_{ten}$ | $101011\cancel{00}_{two}$ | $10_{two}$ | $10_{two}$ ($2_{ten}$) | $11_{two}$ ($2_{ten}$) | Miss |

**CACHE MEMORY (before access)**

| Block Index | V | Tag | Data (4 words): Offset 3 | Offset 2 | Offset 1 | Offset 0 |
|---|---|---|---|---|---|---|
| $00_{two}$ ($0_{ten}$) | 1 | $00_{two}$ | mem[12] | mem[8] | mem[4] | mem[0] |
| $01_{two}$ ($1_{ten}$) | 0 | | | | | |
| $10_{two}$ ($2_{ten}$) | 1 | $01_{two}$ | mem[108] | mem[104] | mem[100] | mem[96] |
| $11_{two}$ ($3_{ten}$) | 0 | | | | | |

**Block index number $2_{ten}$ is set to valid. <u>HOWEVER</u>, the tag in the cache <u>DOES NOT MATCH</u> the tag field of the requested memory address, therefore this is a *cache miss*.**

**CACHE MEMORY (after access)**

| Block Index | V | Tag | Data (4 words): Offset 3 | Offset 2 | Offset 1 | Offset 0 |
|---|---|---|---|---|---|---|
| $00_{two}$ ($0_{ten}$) | 1 | $00_{two}$ | mem[12] | mem[8] | mem[4] | mem[0] |
| $01_{two}$ ($1_{ten}$) | 0 | | | | | |
| $10_{two}$ ($2_{ten}$) | 1 | $10_{two}$ | **mem[172]** | mem[168] | mem[164] | mem[160] |
| $11_{two}$ ($3_{ten}$) | 0 | | | | | |

**On cache miss, find the block in which address $172_{ten}$ resides, copy entire block from main system memory into cache *OVERWRITING* previous contents. Set new tag value and resume original instruction that needed the data.**

# Calculating Block Address and Block Index Numbers

- Using cache parameters from previous example: ***4 blocks with 16 bytes per block***
  - ***Block address*** references the address of a block in main system memory

**Block Address** = ⌊Memory Byte Address / Bytes per Block⌋

**Block Address** = ⌊$104_{ten}$ / 16 Bytes per Block⌋ = ⌊6.5⌋ **= 6**

**Example byte address: $104_{ten}$**
**JUST SHIFT or remove low-order bits**

  - ***Block index number*** references the index location into which the block should be stored

**Block Index Number** = Block Address modulo # Cache Blocks

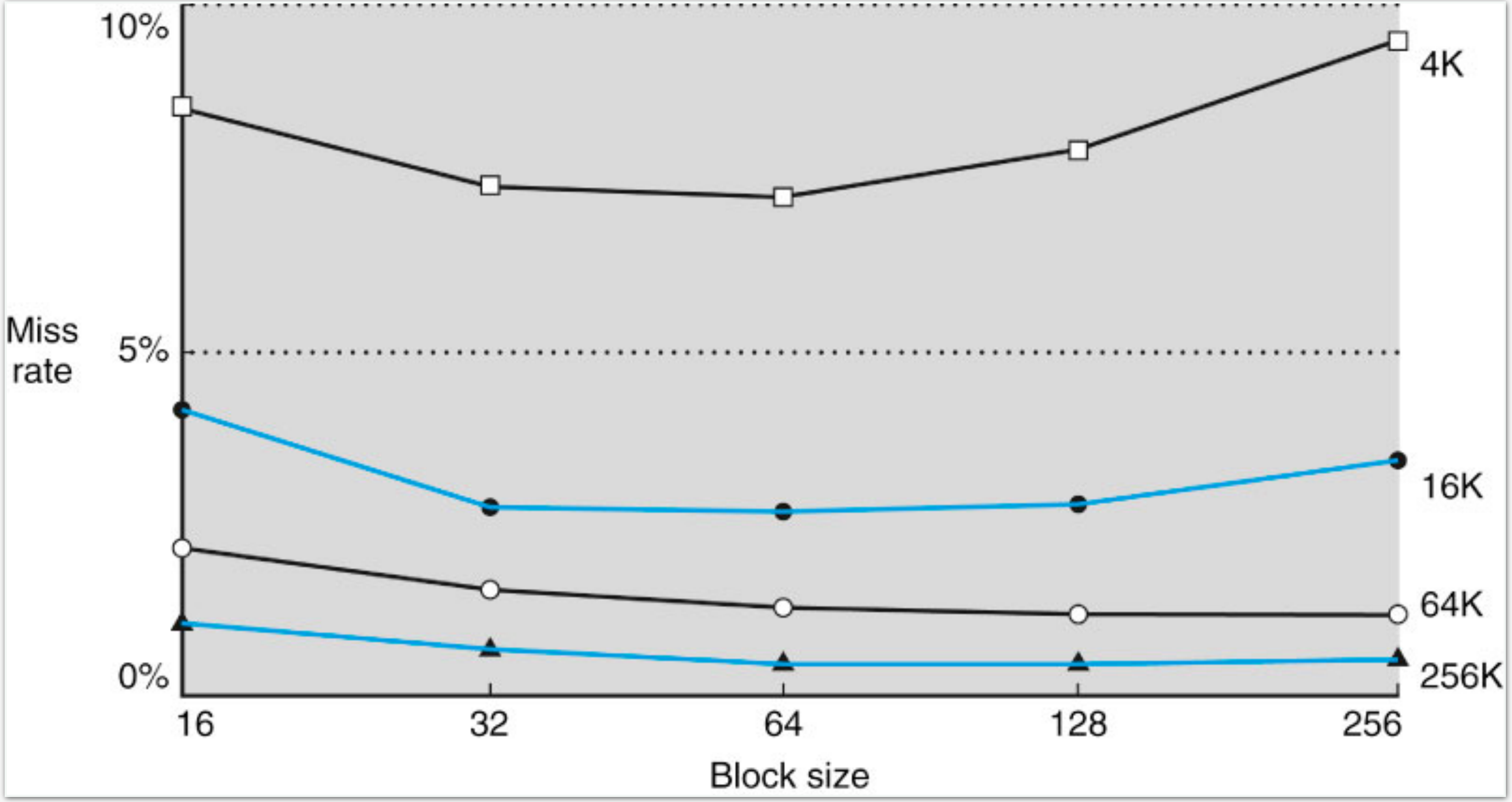
**Block Index Number** = 6 modulo 4 **= 2**

**JUST grab low-order bits**

# Block Size Considerations

- Larger block sizes should reduce miss rate
  - Due to spatial locality
- However, in a fixed-sized cache
  - Larger blocks means there are fewer of them
    - More competition for blocks means an increased miss rate
    - Block may be overwritten in cache before program is done with it
- Larger block size also means larger miss penalty
  - When a miss occurs, must copy more data from main system memory for larger blocks
  - Can override benefit of reduced miss rate

# Block Size Considerations (continued)




Each line represents a different cache size.

Larger caches have smaller miss rate.

Making blocks too large can increase miss rate.

# Handling Writes – Write-Through

- On data-write hit, could just update the block in cache and not the main system memory
  - However, then cache and main system memory would be ***inconsistent***
- In a ***write-through*** scheme, when cache is written also write the data to main system memory
- May negatively impact performance
  - Each write to main system memory takes time
  - Doesn't take advantage of burst write mode
  - Data may be updated repeatedly, no need to write transient data to main memory (e.g. loop variable)
- Various methods exist to help alleviate the performance issues caused by the write-through scheme
  - However, better schemes exist

# Handling Writes – Write-Back

- Alternative to write-through scheme
- On data-write hit, just update the block in cache and not the main system memory
  - Keep track of whether each block is "***dirty***"
    - Dirty blocks are blocks that have been modified but not yet written back to main system memory
- Anytime a dirty block is replaced
  - Write it back to main system memory before overwriting with new block
  - Can use a write buffer to allow replacing block to be read first