

# 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 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)

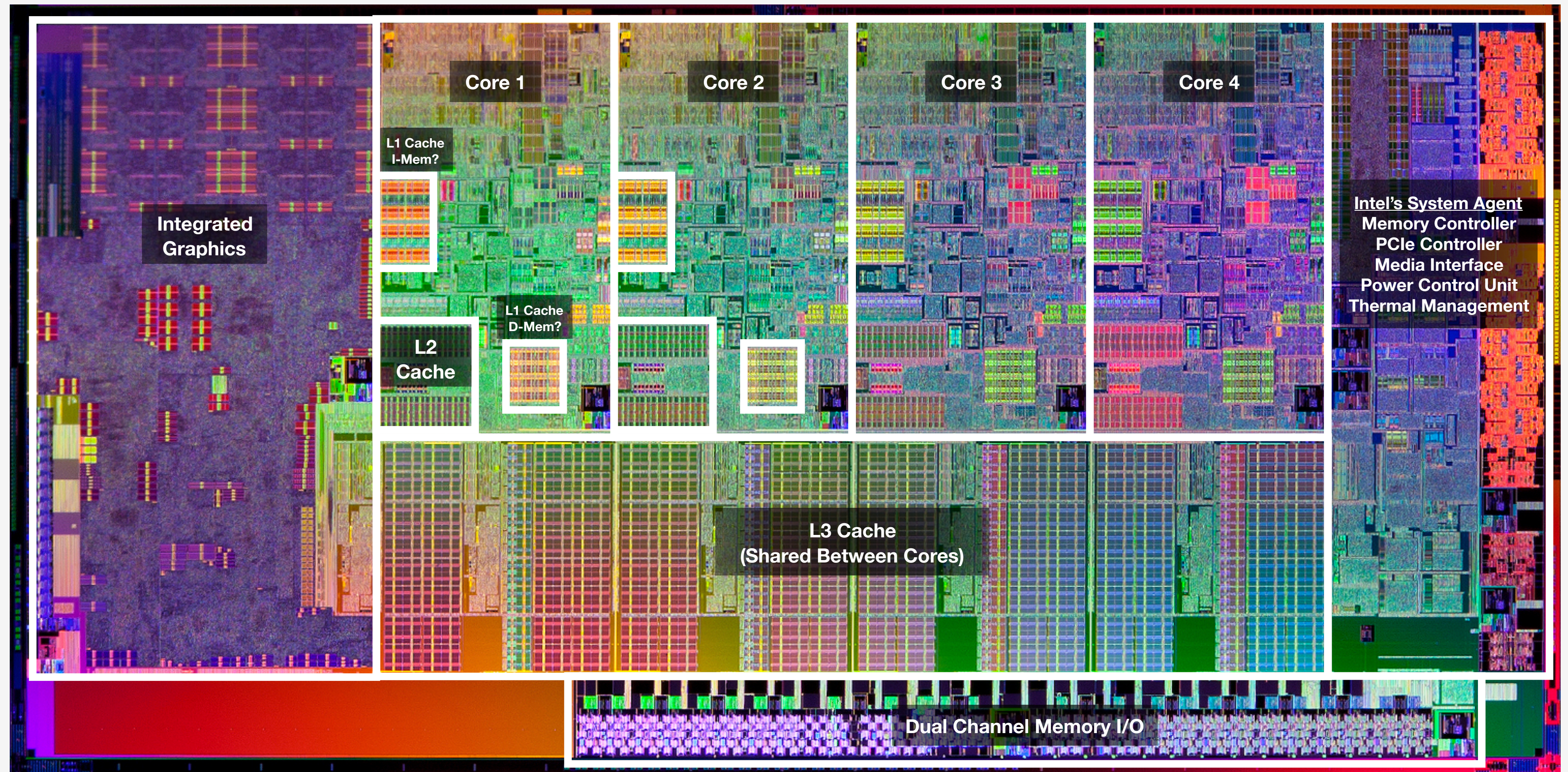
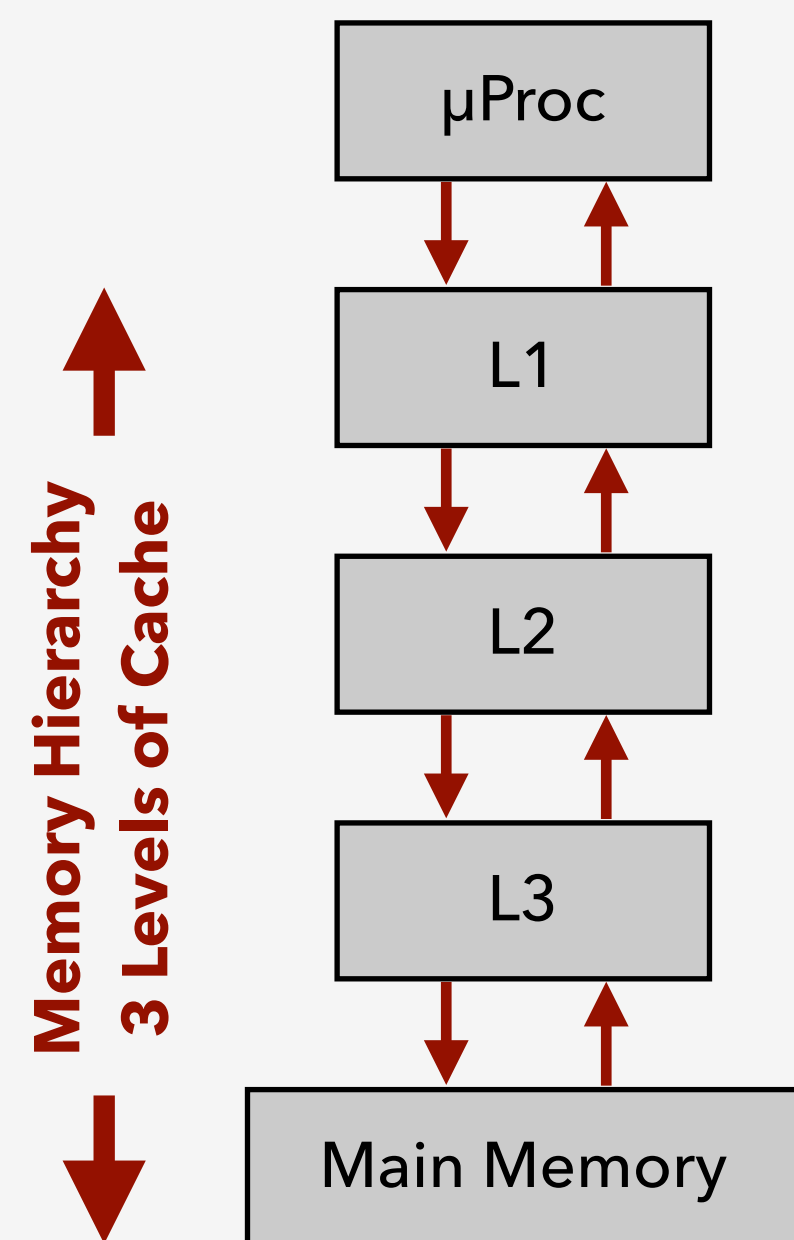
---

- 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)





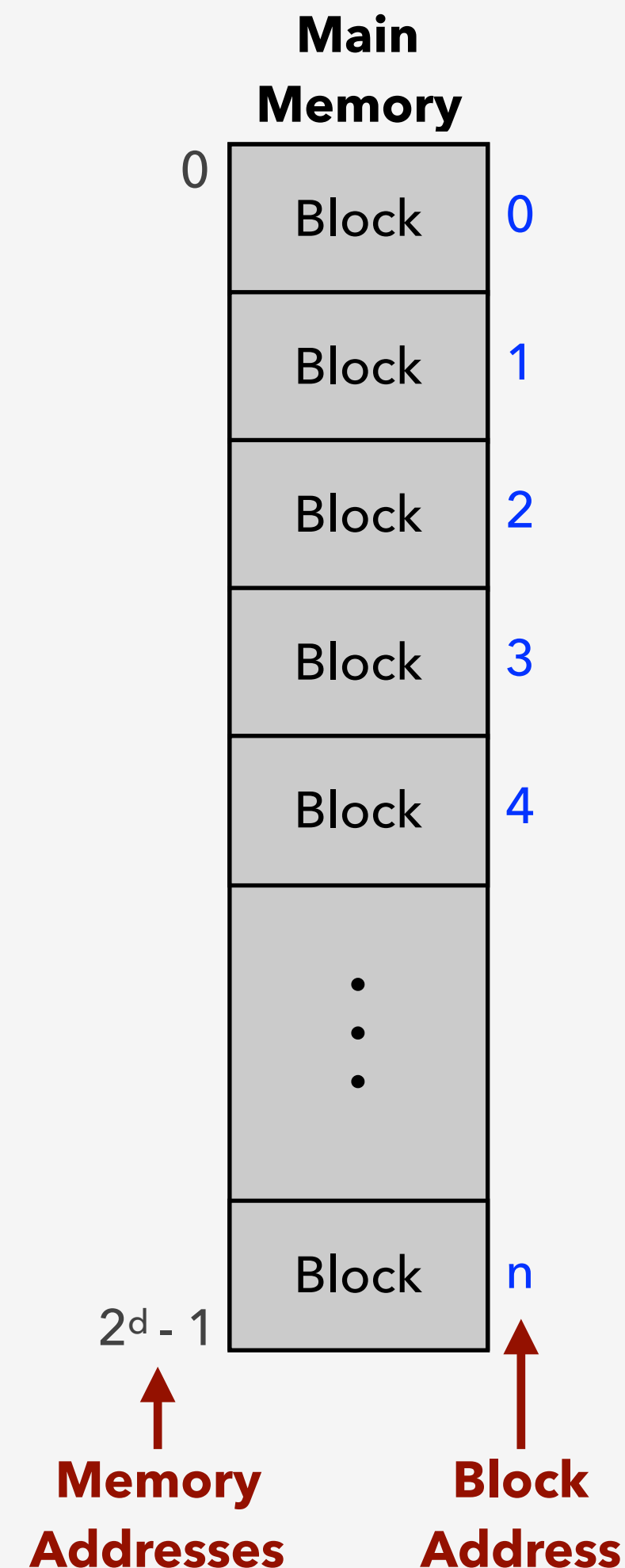
# Cache in the Memory Hierarchy

---

- 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 handled
- 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?

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

Example:

$$128 \text{ Blocks} = \frac{1024 \text{ Words}}{8 \text{ Words/Block}}$$

# Finding the Right Block in Main Memory

- Suppose memory address is generated – the 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

## Calculating a Block Address

Block Address	=	$\frac{\text{Memory Address}}{\text{Block Size}}$
---------------	---	---

## Example

Block #3	=	$\frac{\text{MemWord } 31_{\text{ten}}}{8 \text{ Words/Block}}$
----------	---	---

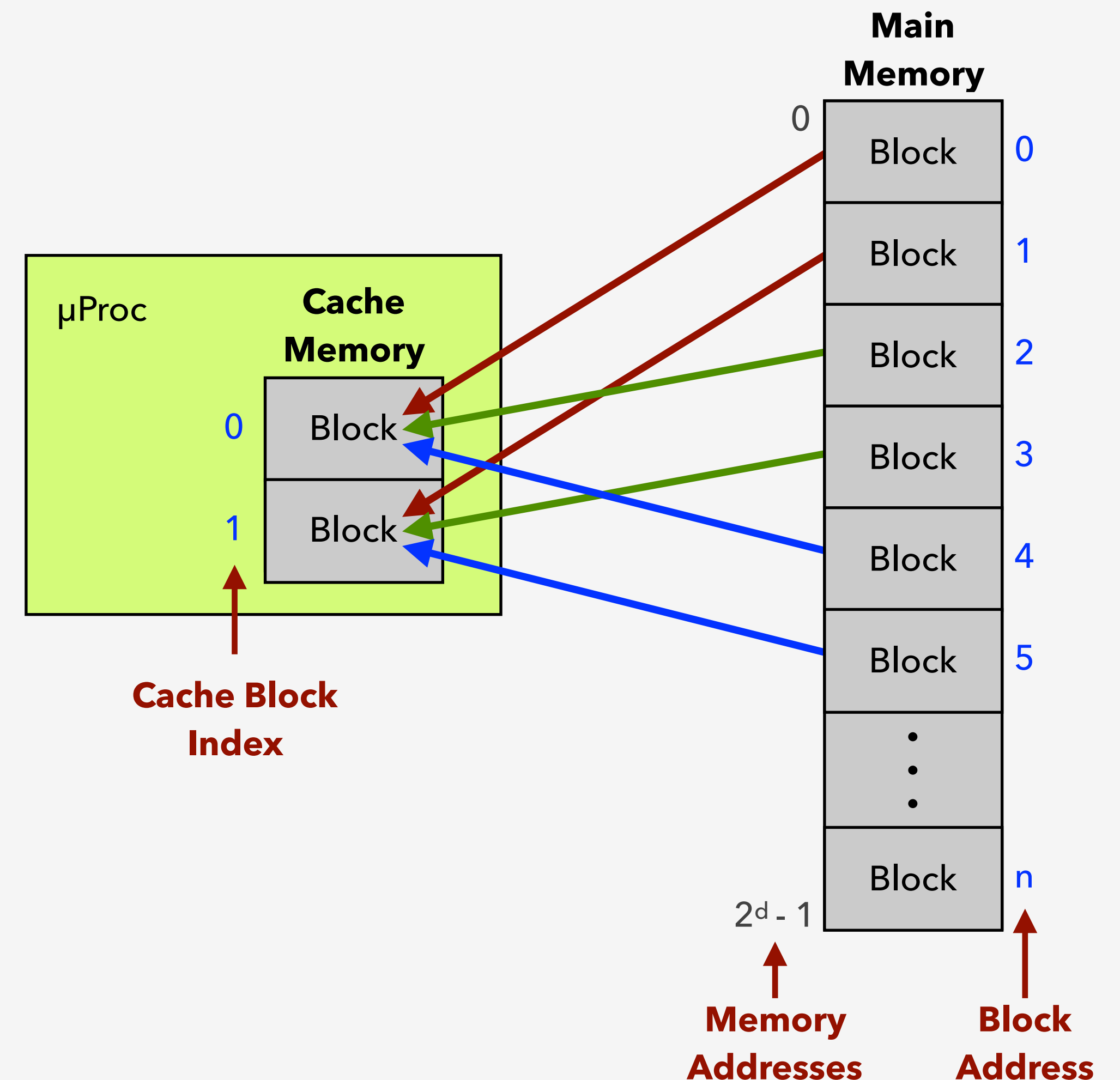
## Example using binary and shifting

MemWord  $31_{\text{ten}} = 11111_{\text{two}}$       8 Words/Block =  $2^3$

To divide by 8, shift-right by 3  
 $\text{SRL}(11111_{\text{two}}) = 11_{\text{two}} = \text{Block } \#3_{\text{ten}}$

# 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 – it is 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)

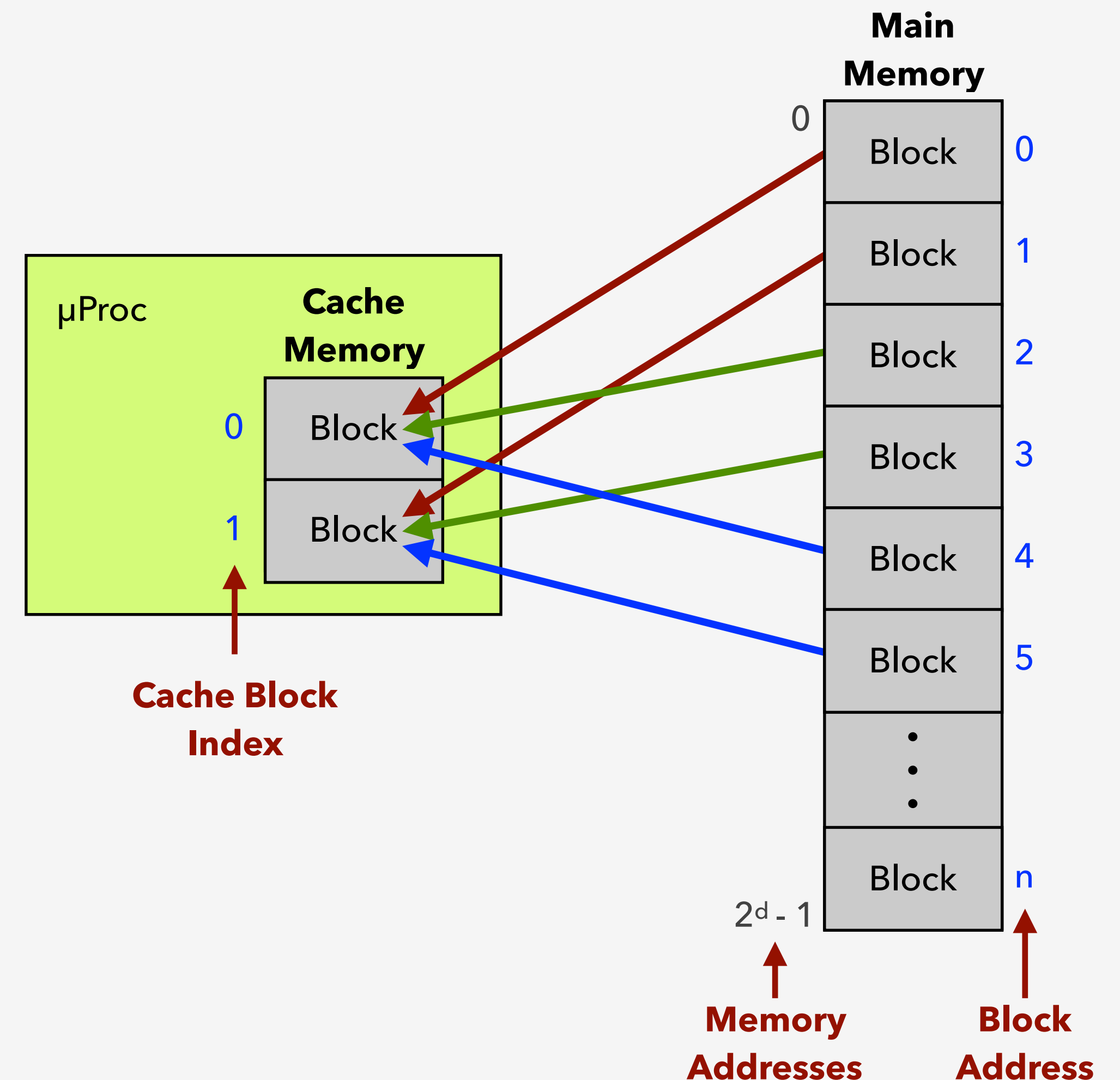




# Direct-Mapped Cache

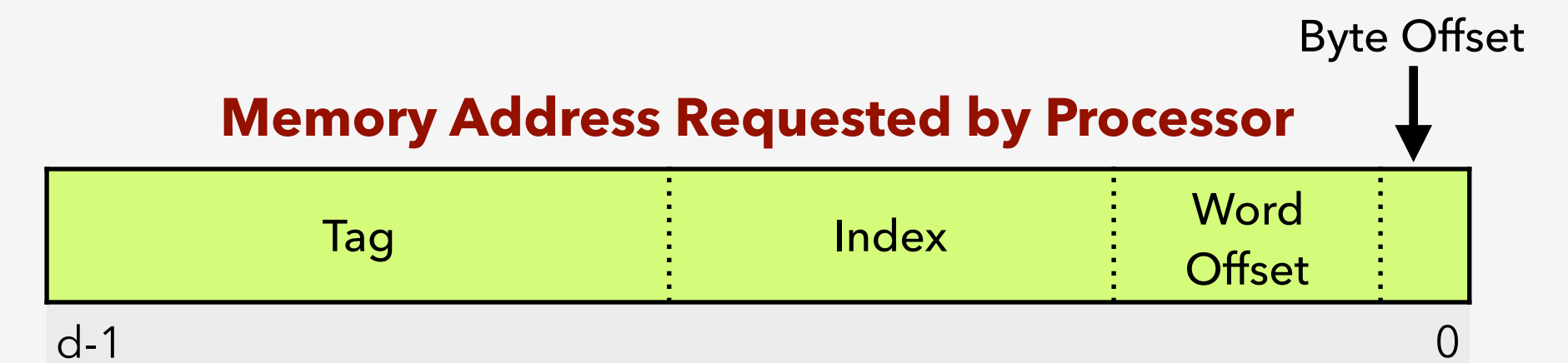
- **Direct-mapped cache** – a cache structure in which each memory location is mapped to exactly one location in the cache
  - Number of blocks is a power of 2
  - Block size is a power of 2
  - Location in cache determined by block address
    - To calculate cache block index, (Block address) modulo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

$$\text{Cache Block Index} = 5 \text{ modulo } 2 = 1$$



# Address Subdivision

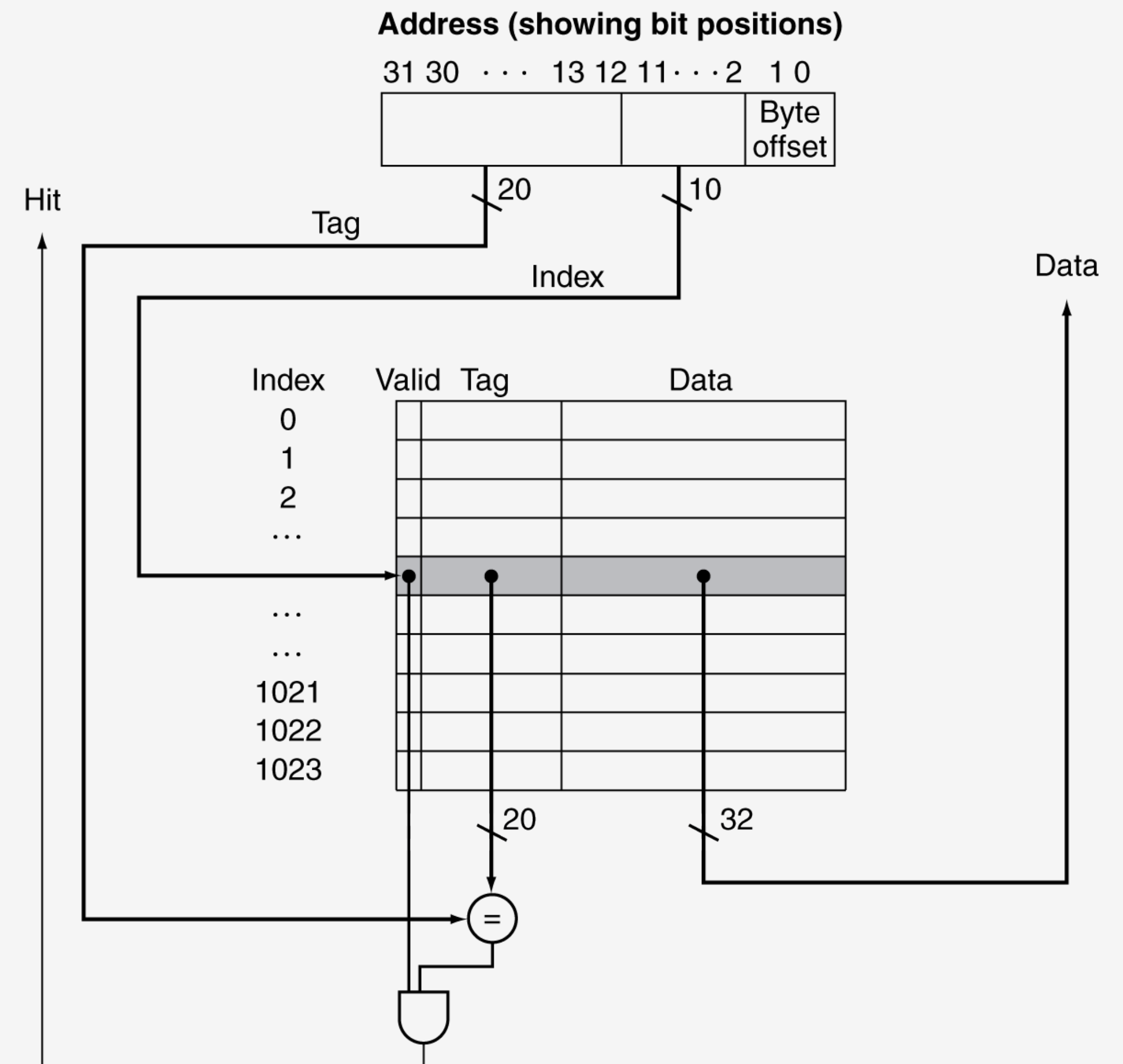
- To utilize cache, a memory byte address is subdivided in multiple fields:
  - Byte offset
    - Specifies a byte within a word
    - Least significant two bits, typically ignored when caching
  - Word offset (if block size > 1 word)
    - Specifies a word within a block
  - Cache block index
    - 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)



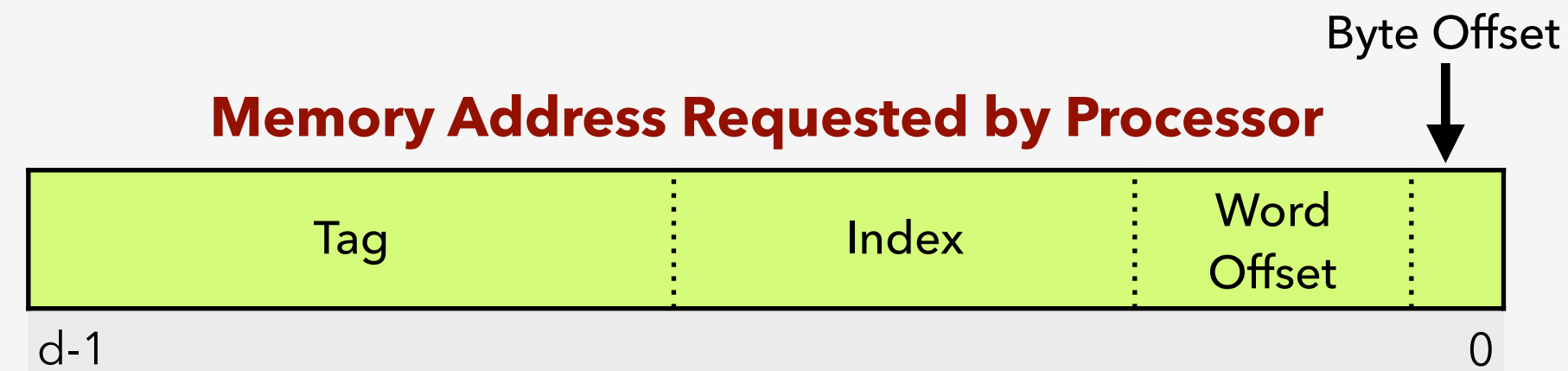


# Cache Structure

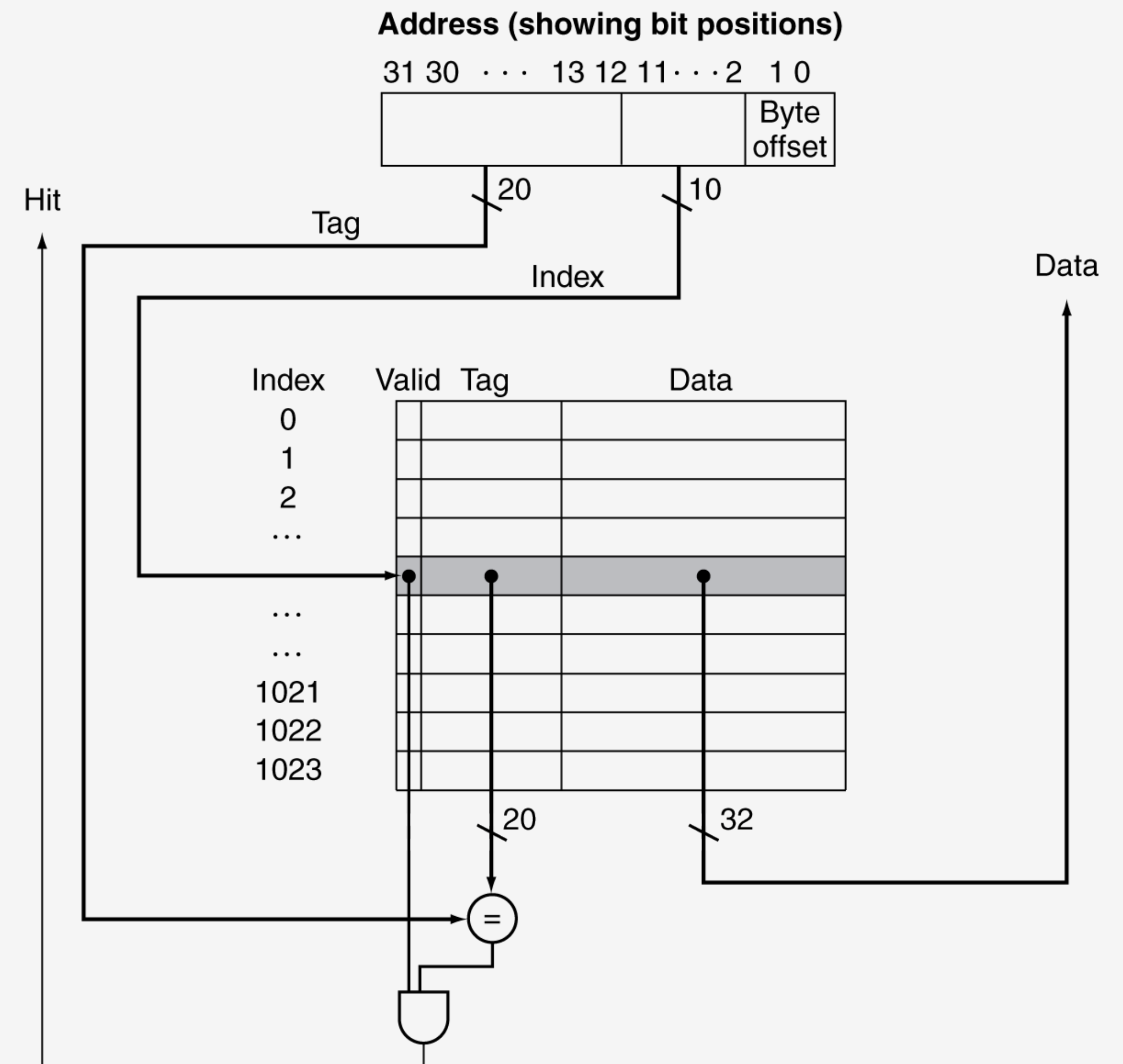
- Each cache entry contains:
  - One or more **Data** words
    - Determined by cache designer
    - Example shows only a single 32-bit word
  - A **Tag** field
    - Differentiates blocks that cache to the same block index in cache memory
  - A **Valid** bit
    - Initialized to 0
    - Set to 1 when cache entry is set
      - Cache blocks are initially empty and invalid



# Address Subdivision/Cache Structure Example



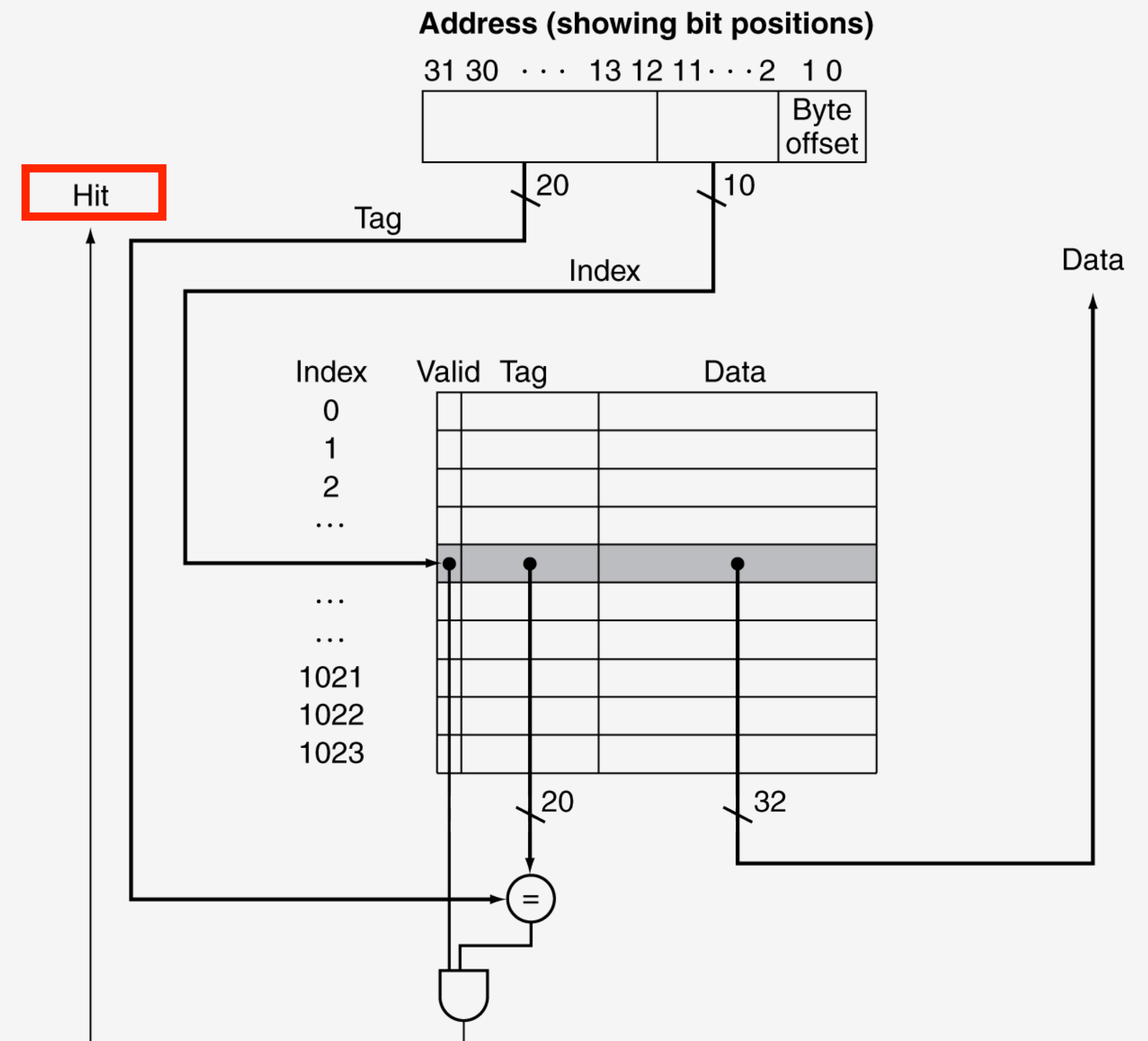
- Byte Offset => bits 1:0 **(2 bits)**
  - Used to address bytes within a word
- Word Offset => Not used in this example since only 1 32-bit word per cache entry
- Cache Index => Bits 11:2 **(10 bits)**
  - Location in cache where this memory address should be stored
- Tag Bits => Bits 31:12 **(20 bits)**
  - Remaining bits used to differentiate blocks





# 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
  - If cache miss was ***Instruction Cache***
    - Restart instruction fetch
  - If cache miss was ***Data Cache***
    - 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 (no byte offset bits)**
    - Total memory space of  $2^8$  words of memory ... cannot fit them all in cache
- First, compute various field sizes

$$\# \text{ Word Offset Bits} = \log_2(\# \text{ Words/Block})$$

$$\# \text{ Word Offset Bits} = \log_2(1 \text{ Word/Block}) = \mathbf{0 \text{ bits}}$$

$$\# \text{ Cache Index Bits} = \log_2(\# \text{ Blocks})$$

$$\# \text{ Cache Index Bits} = \log_2(8 \text{ Blocks}) = \mathbf{3 \text{ bits}}$$

$$\# \text{ Tag Bits} = \# \text{ Addr Bits} - \# \text{ Index Bits} - \# \text{ Offset Bits}$$

$$\# \text{ Tag Bits} = 8 \text{ Addr Bits} - 3 \text{ Index Bits} - 0 \text{ Offset Bits} = \mathbf{5 \text{ bits}}$$



# Cache Example – Cache Structure

# Word Offset Bits = 0 bits  
# Cache 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**

000<sub>two</sub> (0<sub>ten</sub>)

001<sub>two</sub> (1<sub>ten</sub>)

010<sub>two</sub> (2<sub>ten</sub>)

011<sub>two</sub> (3<sub>ten</sub>)

100<sub>two</sub> (4<sub>ten</sub>)

101<sub>two</sub> (5<sub>ten</sub>)

110<sub>two</sub> (6<sub>ten</sub>)

111<sub>two</sub> (7<sub>ten</sub>)

V	Tag	Data
0		
0		
0		
0		
0		
0		
0		
0		

# Cache Example – First Memory Access

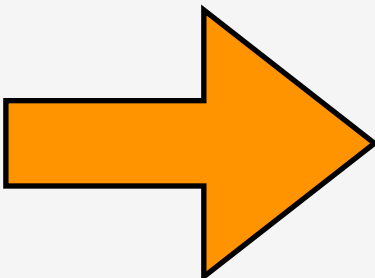
- Access memory address  $22_{\text{ten}}$  (this is a word address)

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

# Word Offset Bits = 0 bits  
# Cache Index Bits = 3 bits  
# Tag Bits = 5 bits

CACHE MEMORY (before access)

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



CACHE MEMORY (after access)

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

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

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  $26_{\text{ten}}$

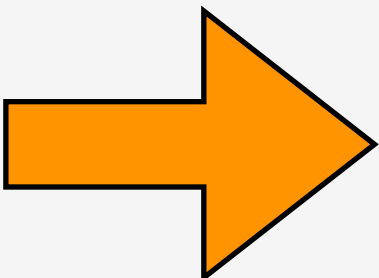
Word Addr	Word Addr (base-2)	Tag	Cache Block Index #	Hit / Miss
$26_{\text{ten}}$	<b>00011010</b> <sub>two</sub>	<b>00011</b> <sub>two</sub>	<b>010</b> <sub>two</sub> ( $2_{\text{ten}}$ )	Miss

# Word Offset Bits = 0 bits  
# Cache Index Bits = 3 bits  
# Tag Bits = 5 bits

CACHE MEMORY (before access)

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

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



CACHE MEMORY (after access)

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

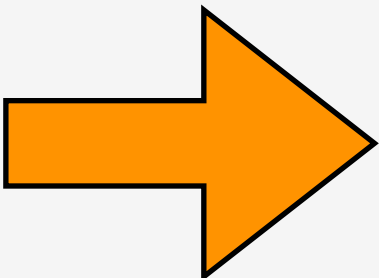
- Access memory address  $22_{\text{ten}}$  (again)

Word Addr	Word Addr (base-2)	Tag	Cache Block Index #	Hit / Miss
$22_{\text{ten}}$	<b>00010</b> <b>110</b> <sub>two</sub>	<b>00010</b> <sub>two</sub>	<b>110</b> <sub>two</sub> ( $6_{\text{ten}}$ )	Hit

# Word Offset Bits = 0 bits  
# Cache Index Bits = 3 bits  
# Tag Bits = 5 bits

CACHE MEMORY (before access)

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



CACHE MEMORY (after access)

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

Block index number  $6_{\text{ten}}$  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**.

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



# Cache Example – Fourth Memory Access

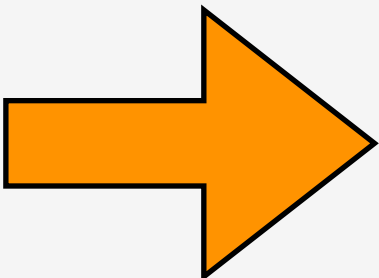
- Access memory address  $26_{\text{ten}}$  (again)

Word Addr	Word Addr (base-2)	Tag	Cache Block Index #	Hit / Miss
$26_{\text{ten}}$	<b>00011</b> <b>010</b> <sub>two</sub>	<b>00011</b> <sub>two</sub>	<b>010</b> <sub>two</sub> ( $2_{\text{ten}}$ )	Hit

# Word Offset Bits = 0 bits  
# Cache Index Bits = 3 bits  
# Tag Bits = 5 bits

CACHE MEMORY (before access)

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



CACHE MEMORY (after access)

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

Block index number  $2_{\text{ten}}$  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**.

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

# Cache Example – Fifth Memory Access

- Access memory address  $16_{\text{ten}}$

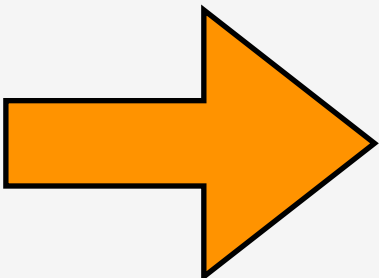
Word Addr	Word Addr (base-2)	Tag	Cache Block Index #	Hit / Miss
$16_{\text{ten}}$	<b>00010000</b> <sub>two</sub>	<b>00010</b> <sub>two</sub>	<b>000</b> <sub>two</sub> ( $0_{\text{ten}}$ )	Miss

# Word Offset Bits = 0 bits  
# Cache Index Bits = 3 bits  
# Tag Bits = 5 bits

CACHE MEMORY (before access)

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

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



CACHE MEMORY (after access)

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

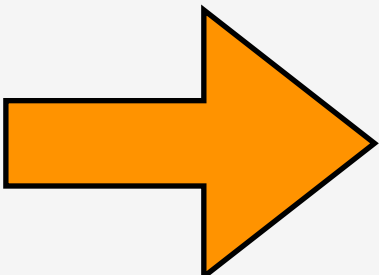
- Access memory address  $18_{\text{ten}}$

Word Addr	Word Addr (base-2)	Tag	Cache Block Index #	Hit / Miss
$18_{\text{ten}}$	<b>00010010</b> <sub>two</sub>	<b>00010</b> <sub>two</sub>	<b>010</b> <sub>two</sub> ( $2_{\text{ten}}$ )	Miss

# Word Offset Bits = 0 bits  
# Cache Index Bits = 3 bits  
# Tag Bits = 5 bits

CACHE MEMORY (before access)

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



CACHE MEMORY (after access)

Block Index	V	Tag	Data
$000_{\text{two}}$ ( $0_{\text{ten}}$ )	1	$00010_{\text{two}}$	mem[16]
$001_{\text{two}}$ ( $1_{\text{ten}}$ )	0		
$010_{\text{two}}$ ( $2_{\text{ten}}$ )	1	<b><math>00010_{\text{two}}</math></b>	<b>mem[18]</b>
$011_{\text{two}}$ ( $3_{\text{ten}}$ )	0		
$100_{\text{two}}$ ( $4_{\text{ten}}$ )	0		
$101_{\text{two}}$ ( $5_{\text{ten}}$ )	0		
$110_{\text{two}}$ ( $6_{\text{ten}}$ )	1	$00010_{\text{two}}$	mem[22]
$111_{\text{two}}$ ( $7_{\text{ten}}$ )	0		

Block index number  $2_{\text{ten}}$  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

$$\# \text{ Word Offset Bits} = \log_2(\# \text{ Words/Block})$$

$$\# \text{ Word Offset Bits} = \log_2(4 \text{ Words/Block}) = \mathbf{2 \text{ bits}}$$

$$\# \text{ Cache Index Bits} = \log_2(\# \text{ Blocks})$$

$$\# \text{ Cache Index Bits} = \log_2(4 \text{ Blocks}) = \mathbf{2 \text{ bits}}$$

$$\# \text{ Tag Bits} = \# \text{ Addr Bits} - \# \text{ Index Bits} - \# \text{ Offset Bits} - \text{Byte Offset Bits}$$

$$\# \text{ Tag Bits} = 8 \text{ Addr Bits} - 2 \text{ Index Bits} - 2 \text{ Offset Bits} - 2 \text{ Byte Offset Bits} = \mathbf{2 \text{ bits}}$$

# 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

# Word Offset Bits = 2 bits  
# Cache Index Bits = 2 bits  
# Tag Bits = 2 bits

Block Index	V	Tag	Data (4 words)			
			Offset 3	Offset 2	Offset 1	Offset 0
00 <sub>two</sub> (0 <sub>ten</sub> )	0					
01 <sub>two</sub> (1 <sub>ten</sub> )	0					
10 <sub>two</sub> (2 <sub>ten</sub> )	0					
11 <sub>two</sub> (3 <sub>ten</sub> )	0					

# Cache Example #2 – First Memory Access

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

# Word Offset Bits = 2 bits  
# Cache Index Bits = 2 bits  
# Tag Bits = 2 bits

Byte Addr	Byte Addr (base-2)	Tag	Cache Block Index #	Offset	Hit / Miss
$12_{\text{ten}}$	$00001100_{\text{two}}$	$00_{\text{two}}$	$00_{\text{two}}$ ( $0_{\text{ten}}$ )	$11_{\text{two}}$ ( $3_{\text{ten}}$ )	Miss

CACHE MEMORY  
(before access)

Block Index

$00_{\text{two}}$  ( $0_{\text{ten}}$ )  
 $01_{\text{two}}$  ( $1_{\text{ten}}$ )  
 $10_{\text{two}}$  ( $2_{\text{ten}}$ )  
 $11_{\text{two}}$  ( $3_{\text{ten}}$ )

V	Tag	Data (4 words)			
		Offset 3	Offset 2	Offset 1	Offset 0
0					
0					
0					
0					

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

CACHE MEMORY  
(after access)

Block Index

$00_{\text{two}}$  ( $0_{\text{ten}}$ )  
 $01_{\text{two}}$  ( $1_{\text{ten}}$ )  
 $10_{\text{two}}$  ( $2_{\text{ten}}$ )  
 $11_{\text{two}}$  ( $3_{\text{ten}}$ )

V	Tag	Data (4 words)			
		Offset 3	Offset 2	Offset 1	Offset 0
1	$00_{\text{two}}$	mem[12]	mem[8]	mem[4]	mem[0]
0					
0					
0					

On cache miss, find the block in which address  $12_{\text{ten}}$  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  $104_{ten}$  (this is a byte address, i.e. word address  $26_{ten}$ )

# Word Offset Bits = 2 bits  
# Cache Index Bits = 2 bits  
# Tag Bits = 2 bits

Byte Addr	Byte Addr (base-2)	Tag	Cache Block Index #	Offset	Hit / Miss
$104_{ten}$	$01101000_{two}$	$01_{two}$	$10_{two} (2_{ten})$	$10_{two} (2_{ten})$	Miss

CACHE MEMORY  
(before access)

Block Index
$00_{two} (0_{ten})$
$01_{two} (1_{ten})$
$10_{two} (2_{ten})$
$11_{two} (3_{ten})$

V	Tag	Data (4 words)			
		Offset 3	Offset 2	Offset 1	Offset 0
1	$00_{two}$	mem[12]	mem[8]	mem[4]	mem[0]
0					
0					
0					

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

CACHE MEMORY  
(after access)

Block Index
$00_{two} (0_{ten})$
$01_{two} (1_{ten})$
$10_{two} (2_{ten})$
$11_{two} (3_{ten})$

V	Tag	Data (4 words)			
		Offset 3	Offset 2	Offset 1	Offset 0
1	$00_{two}$	mem[12]	mem[8]	mem[4]	mem[0]
0					
1	$01_{two}$	mem[108]	mem[104]	mem[100]	mem[96]
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  $96_{\text{ten}}$  (this is a byte address, i.e. word address  $24_{\text{ten}}$ )

Byte Addr	Byte Addr (base-2)	Tag	Cache Block Index #	Offset	Hit / Miss
$96_{\text{ten}}$	$01100000_{\text{two}}$	$01_{\text{two}}$	$10_{\text{two}}$ ( $2_{\text{ten}}$ )	$00_{\text{two}}$ ( $0_{\text{ten}}$ )	Hit

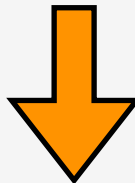
# Word Offset Bits = 2 bits  
# Cache Index Bits = 2 bits  
# Tag Bits = 2 bits

CACHE MEMORY  
(before access)

Block Index
$00_{\text{two}}$ ( $0_{\text{ten}}$ )
$01_{\text{two}}$ ( $1_{\text{ten}}$ )
$10_{\text{two}}$ ( $2_{\text{ten}}$ )
$11_{\text{two}}$ ( $3_{\text{ten}}$ )



V	Tag	Data (4 words)			
		Offset 3	Offset 2	Offset 1	Offset 0
1	$00_{\text{two}}$	mem[12]	mem[8]	mem[4]	mem[0]
0					
1	$01_{\text{two}}$	mem[108]	mem[104]	mem[100]	mem[96]
0					



CACHE MEMORY  
(after access)

Block Index
$00_{\text{two}}$ ( $0_{\text{ten}}$ )
$01_{\text{two}}$ ( $1_{\text{ten}}$ )
$10_{\text{two}}$ ( $2_{\text{ten}}$ )
$11_{\text{two}}$ ( $3_{\text{ten}}$ )

V	Tag	Data (4 words)			
		Offset 3	Offset 2	Offset 1	Offset 0
1	$00_{\text{two}}$	mem[12]	mem[8]	mem[4]	mem[0]
0					
1	$01_{\text{two}}$	mem[108]	mem[104]	mem[100]	mem[96]
0					

Block index number  $2_{\text{ten}}$  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**.

On cache miss, find the block in which address  $104_{\text{ten}}$  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 – Fourth Memory Access

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

Byte Addr	Byte Addr (base-2)	Tag	Cache Block Index #	Offset	Hit / Miss
$172_{\text{ten}}$	<b>10</b> <b>10</b> <b>11</b> <b>00</b> <sub>two</sub>	<b>10</b> <sub>two</sub>	<b>10</b> <sub>two</sub> ( $2_{\text{ten}}$ )	<b>11</b> <sub>two</sub> ( $3_{\text{ten}}$ )	Miss

# Word Offset Bits = 2 bits  
# Cache Index Bits = 2 bits  
# Tag Bits = 2 bits

CACHE MEMORY  
(before access)

Block Index
$00_{\text{two}}$ ( $0_{\text{ten}}$ )
$01_{\text{two}}$ ( $1_{\text{ten}}$ )
 $10_{\text{two}}$ ( $2_{\text{ten}}$ )
$11_{\text{two}}$ ( $3_{\text{ten}}$ )

V	Tag	Data (4 words)			
		Offset 3	Offset 2	Offset 1	Offset 0
1	$00_{\text{two}}$	mem[12]	mem[8]	mem[4]	mem[0]
0					
1	$01_{\text{two}}$	mem[108]	mem[104]	mem[100]	mem[96]
0					

Block index number  $2_{\text{ten}}$  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**.

CACHE MEMORY  
(after access)

Block Index
$00_{\text{two}}$ ( $0_{\text{ten}}$ )
$01_{\text{two}}$ ( $1_{\text{ten}}$ )
$10_{\text{two}}$ ( $2_{\text{ten}}$ )
$11_{\text{two}}$ ( $3_{\text{ten}}$ )

V	Tag	Data (4 words)			
		Offset 3	Offset 2	Offset 1	Offset 0
1	$00_{\text{two}}$	mem[12]	mem[8]	mem[4]	mem[0]
0					
1	<b><math>10_{\text{two}}</math></b>	<b>mem[172]</b>	<b>mem[168]</b>	<b>mem[164]</b>	<b>mem[160]</b>
0					

On cache miss, find the block in which address  $172_{\text{ten}}$  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

$$\text{Block Address} = \lfloor \text{Memory Byte Address} / \text{Bytes per Block} \rfloor$$

$$\text{Block Address} = \lfloor 104_{\text{ten}} / 16 \text{ Bytes per Block} \rfloor = \lfloor 6.5 \rfloor = \mathbf{6}$$

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

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

$$\text{Block Index Number} = \text{Block Address modulo } \# \text{ Cache Blocks}$$

$$\text{Block Index Number} = 6 \text{ modulo } 4 = \mathbf{2}$$

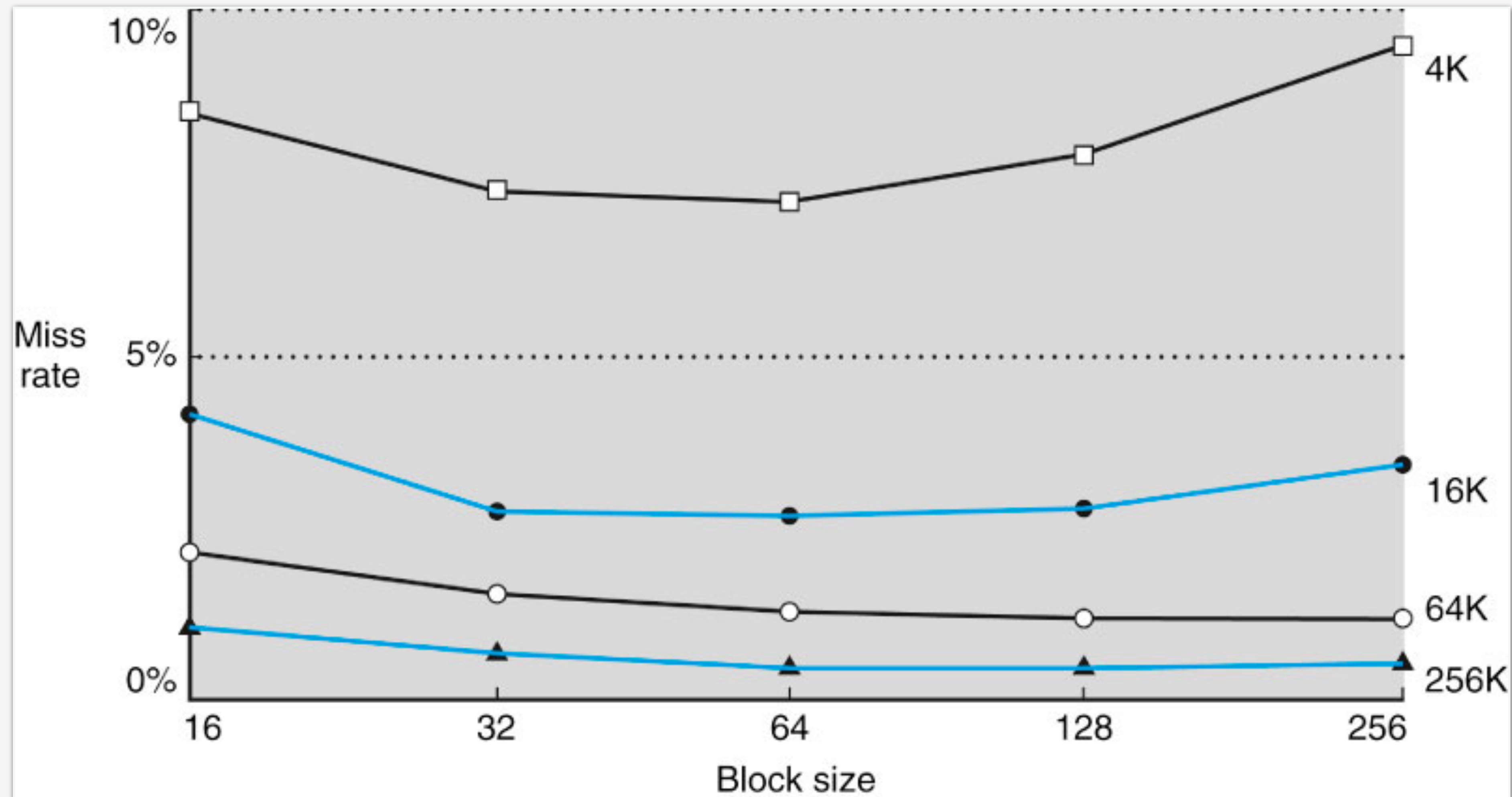
**JUST grab low-order bits**

# 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