EE 357 Unit 14

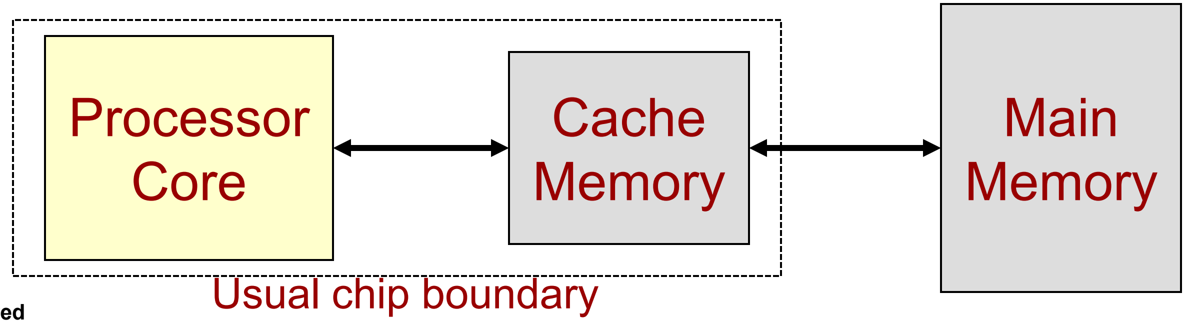
Cache Definitions

Cache Address Mapping

Cache Performance

What is Cache Memory?

- Cache memory is a small, fast memory used to hold **copies** of data that the processor will likely need to access in the near future
- Cache sits between the processor and main memory (MM) and is usually built onto same chip as processor
- Read and write requests will hopefully be satisfied by the fast cache rather than the slow main memory

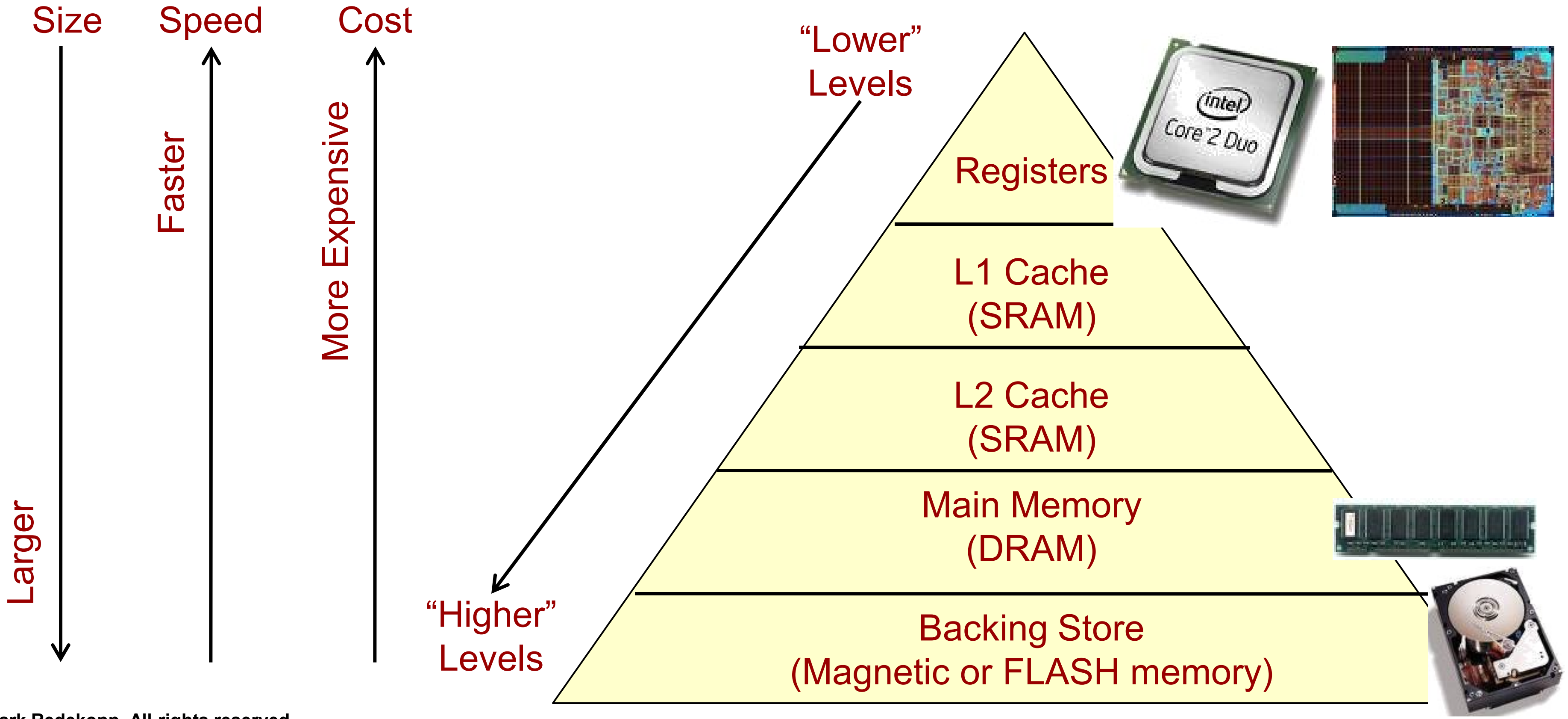


Motivation for Cache Memory

- Large memories are inherently slow
  - We need a large memory to hold code and data from multiple applications
- Small memory is inherently faster
  - Important Fact: Processor is only accessing a small fraction of code and data in any short time period
- Use both!
  - Large memory as a global store and cache as a smaller "working-set" store

Memory Hierarchy

- Memory hierarchy provides ability to access data quickly from lower levels while still providing large memory size



Size
Larger

Speed
Faster

Cost
More Expensive

"Lower" Levels

"Higher" Levels

Registers

L1 Cache
(SRAM)

L2 Cache
(SRAM)

Main Memory
(DRAM)

Backing Store
(Magnetic or FLASH memory)

Principle of Locality

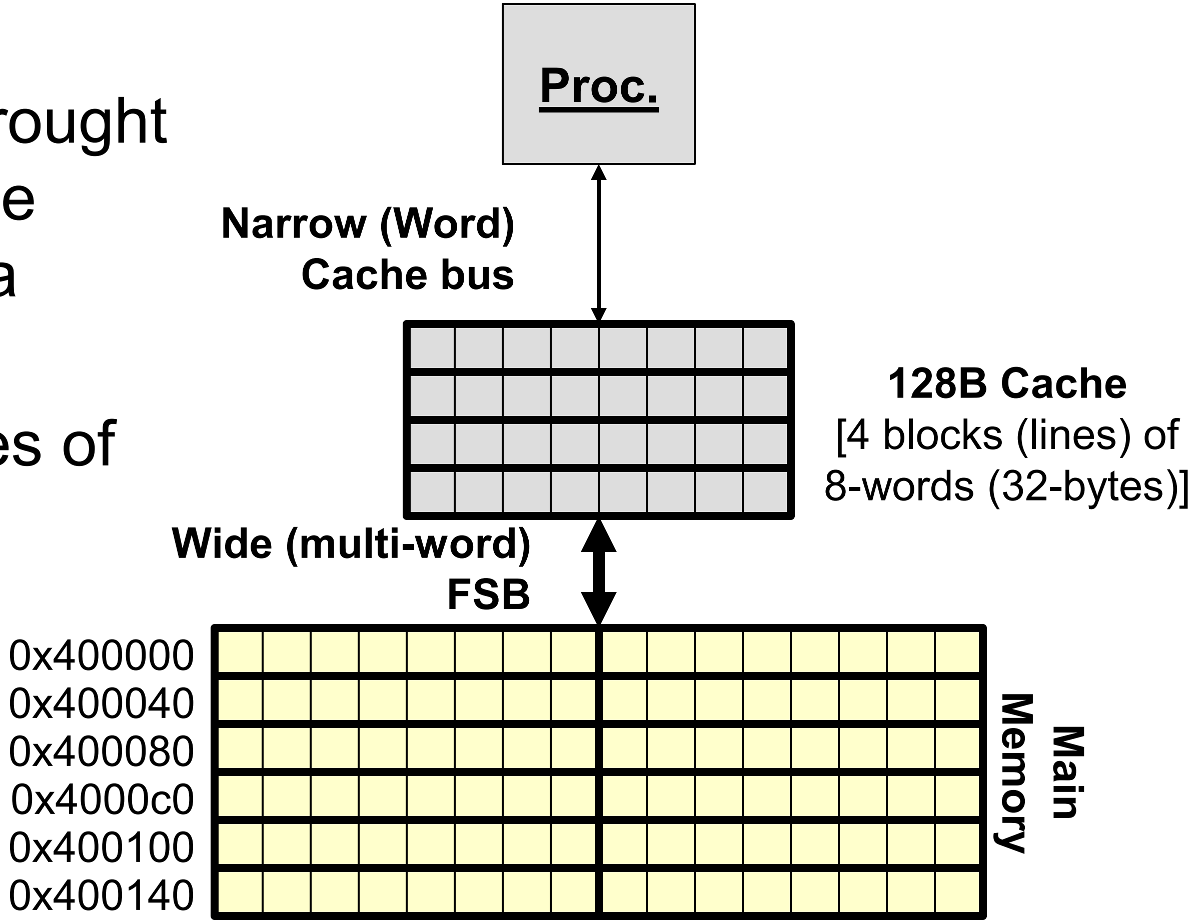
- 2 dimensions of this principle: space & time
- Spatial Locality – Future accesses will likely cluster near current accesses
  - Instructions and data arrays are sequential (they are all one after the next)
- Temporal Locality – Recent accesses will likely be accessed again soon
  - Same code and data are repeatedly accessed (loops, subroutines, etc.)
  - 90/10 rule: Analysis shows that usually 10% of the written instructions account for 90% of the executed instructions

Cache and Locality

- Caches take advantage of locality
- Spatial Locality
  - Caches do not store individual words but blocks of words (a.k.a. “cache line”)
  - Caches always bring in a group of sequential words because if we access one, we are likely to access the next
  - Bringing in blocks of sequential words takes advantage of memory architecture (i.e. FPM, SDRAM, etc.)
- Temporal Locality
  - Leave data in the cache because it will likely be accessed again

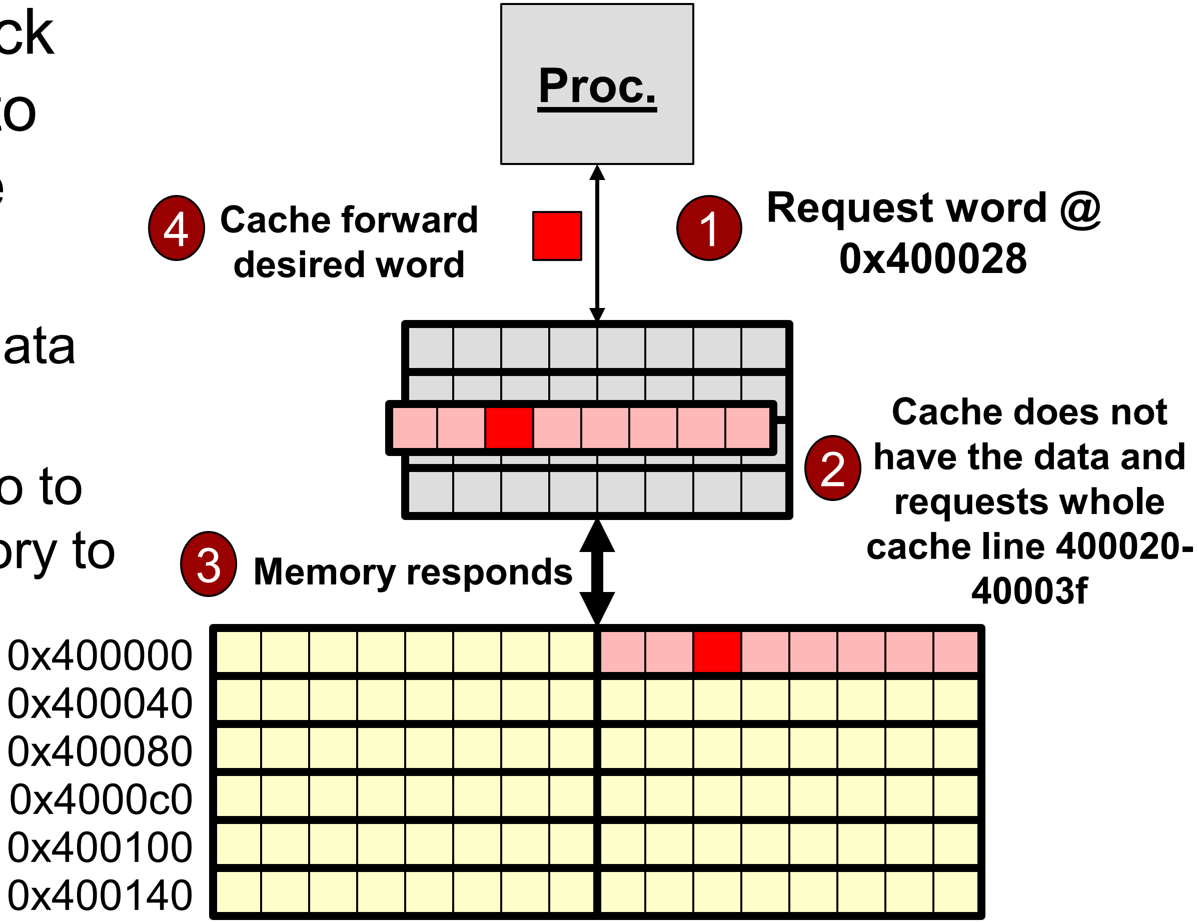
Cache Blocks/Lines

- Cache is broken into “blocks” or “lines”
  - Any time data is brought in, it will bring in the entire block of data
  - Blocks start on addresses multiples of their size
  - Usually block size matches burst size from main memory



Cache Blocks/Lines

- Whenever the processor generates a read or a write, it will first check the cache memory to see if it contains the desired data
  - If so, it can get the data quickly from cache
  - Otherwise, it must go to the slow main memory to get the data


Proc.
1
Request word @ 0x400028
2
Cache does not have the data and requests whole cache line 400020-40003f
3
Memory responds
4
Cache forward desired word
0x400000
0x400040
0x400080
0x4000c0
0x400100
0x400140

Cache Definitions

- **Cache Hit**: if requested data present in cache
  - Read Hit: Desired read data resides in cache
  - Write Hit: Desired write data resides in cache
- **Cache Miss**: if requested data not present in cache
  - Read Miss: Desired read data does not reside in cache
  - Write Miss: Desired write data does not reside in cache

Read-Miss Policies

- When desired read data is not in the cache, the entire block must be read in to the cache
  - Different policies for when the desired read data is returned to the processor
- **Early Restart (Load-Through)**: As a block is being fetched the desired word will be forwarded to processor as soon as it is loaded into the cache.
  - **Critical word first**: Variant of early-restart where the desired word in the block is brought in from memory first, then the rest of the block (i.e. if the 2nd word in an 4-word block is desired, the memory will supply word 2 then 3, 0, 1…using modulo addressing)
- **No Early- Restart (No-Load-Through)**: Entire block is fetched before returning the desired word to the processor

Write Miss Policies

- When desired write data is not in cache, what should be done?
- **Fetch-on-miss (Write-Allocate)**: Bring block from MM to cache and then perform write w/ given write-hit policy (chances are you will read this data soon so bring in the block)
- **Write Around (No Write-Allocate)**: Just update the word in MM and do not bring the block into cache (write around the cache going straight to MM)

Write (Hit) Policies

- If desired write data is already in cache or once it is brought in after a miss, what should be done?
- **Write-Through**: Update the desired word in MM as well as the cached version.
  - This way the MM and cache versions are always "**consistent**" (in sync.)
- **Write-Back**: Update the cached version of the word and NOT the MM copy. Requires that the block be written back to main memory when it is evicted from cache.

- ***NOTE***: For our class we will assume that *Write-Through and No-Write Allocate* are always used together and that *Write-Back and Write-Allocate* are used together.

Cache Memory

- Example code snippet:

(recall $t0,$t1 are aliases for registers $8 and $9)

```
         .text
         addi     $t0,$0,20
LOOP:    addi     $t0,$t0,-1
         bne      $t0,$zero,LOOP
         lui      $t1,0x1001
         ori      $t1,$t1,0x0004
         sw       $t0,0($t1)
```

MM	
addi	0x400000
addi	0x400004
bne	0x400008
lui	0x40000c
ori	0x400010
sw	0x400014
?	0x400018
?	0x40001c

Assumptions:

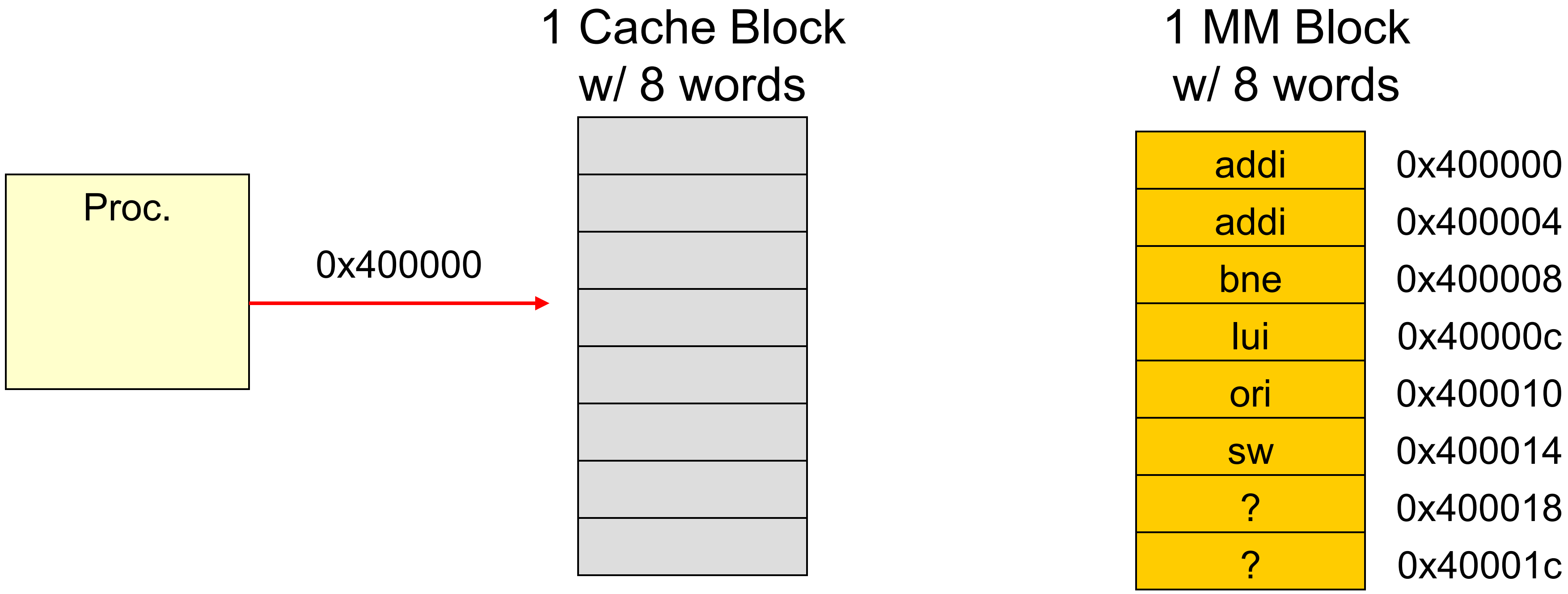
1.) Cache access requires 10ns

2.) MM access requires 100ns

3.) Processor time is negligible

Early-Restart Cache

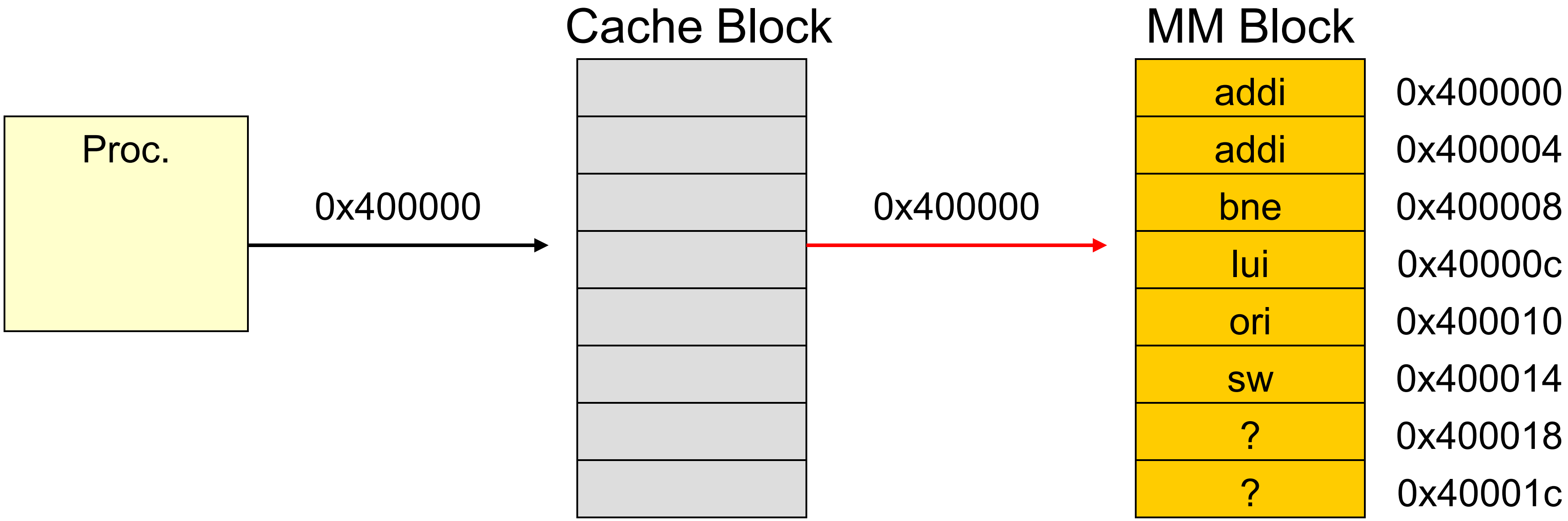
- Early-Restart – cache returns the requested data to the processor as soon as it gets it



Time: 0 ns

Early-Restart Cache

- Early-Restart – cache returns the requested data to the processor as soon as it gets it

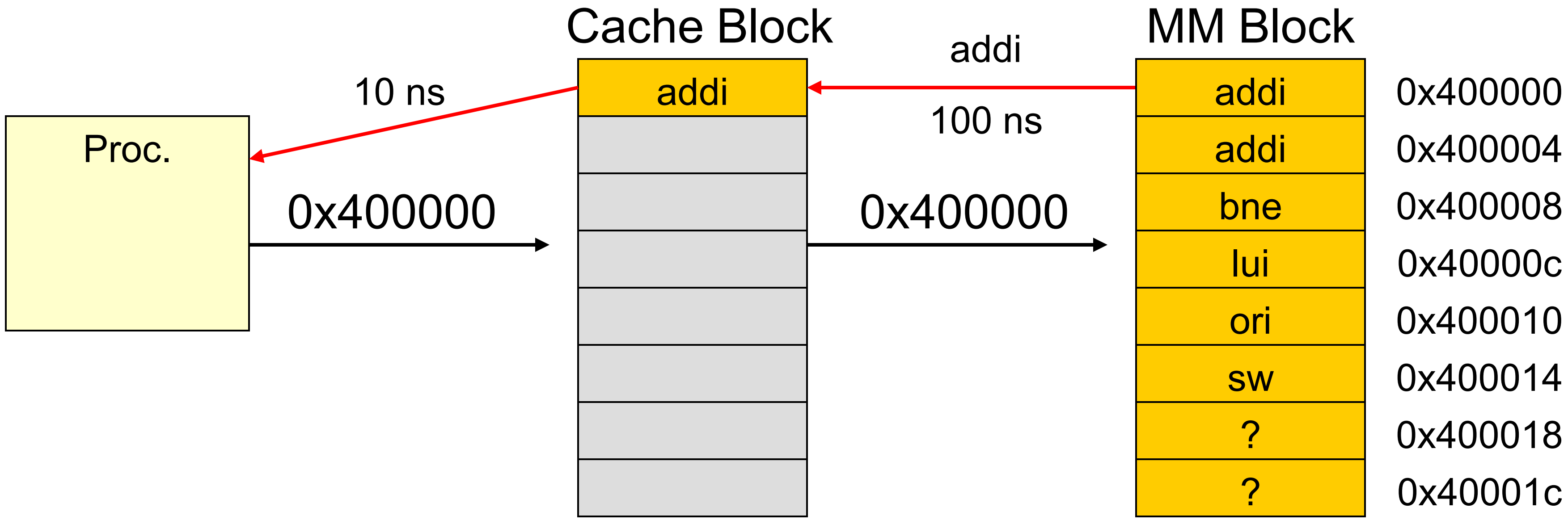


Time: 0 ns

Cache
Miss

Early-Restart Cache

- Early-Restart – cache returns the requested data to the processor as soon as it gets it

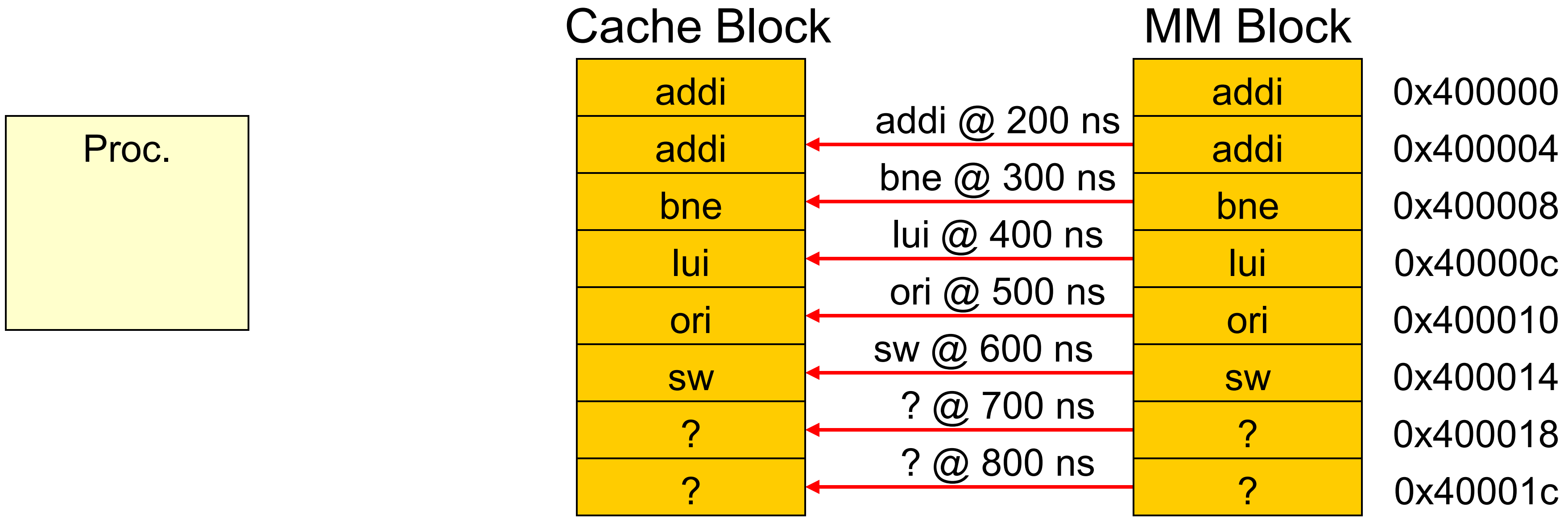


Time: 110 ns

Processor can continue

Early-Restart Cache

- Early-Restart – cache returns the requested data to the processor as soon as it gets it



Proc.

Cache Block

addi
addi
bne
lui
ori
sw
?
?

addi @ 200 ns
bne @ 300 ns
lui @ 400 ns
ori @ 500 ns
sw @ 600 ns
? @ 700 ns
? @ 800 ns

MM Block

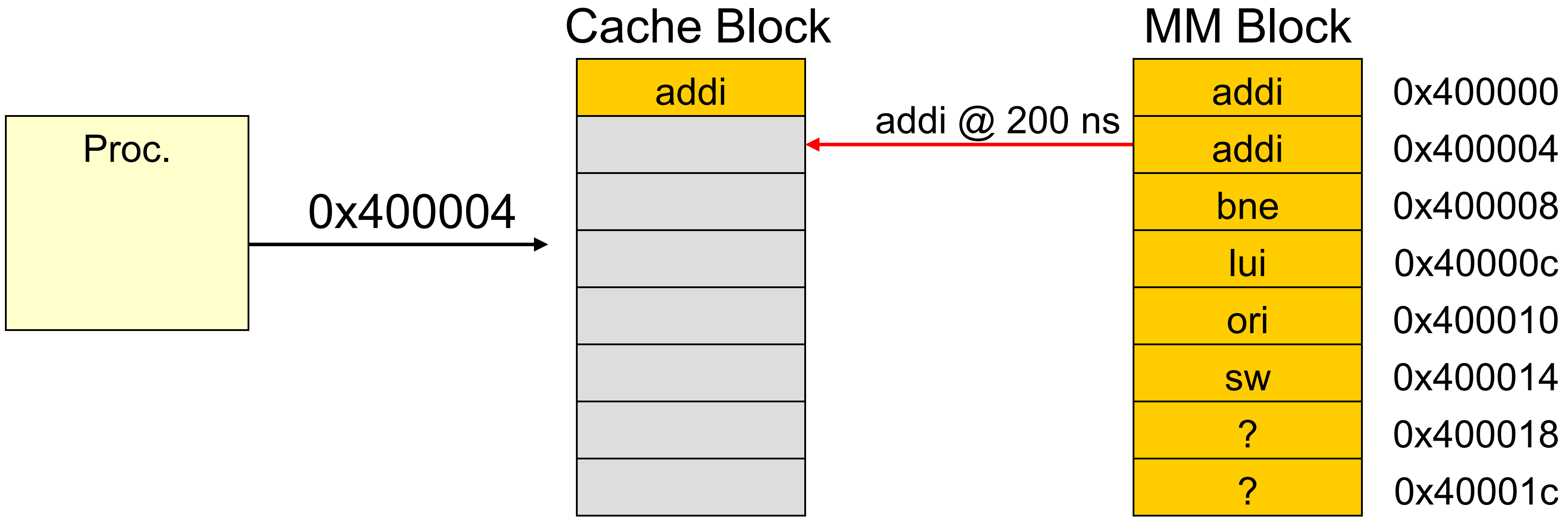
addi	0x400000
addi	0x400004
bne	0x400008
lui	0x40000c
ori	0x400010
sw	0x400014
?	0x400018
?	0x40001c

Time: 800 ns

Each successive word in the block is loaded after 100 ns (in reality this should be less due to sequential burst ability of memory)

Early-Restart Cache

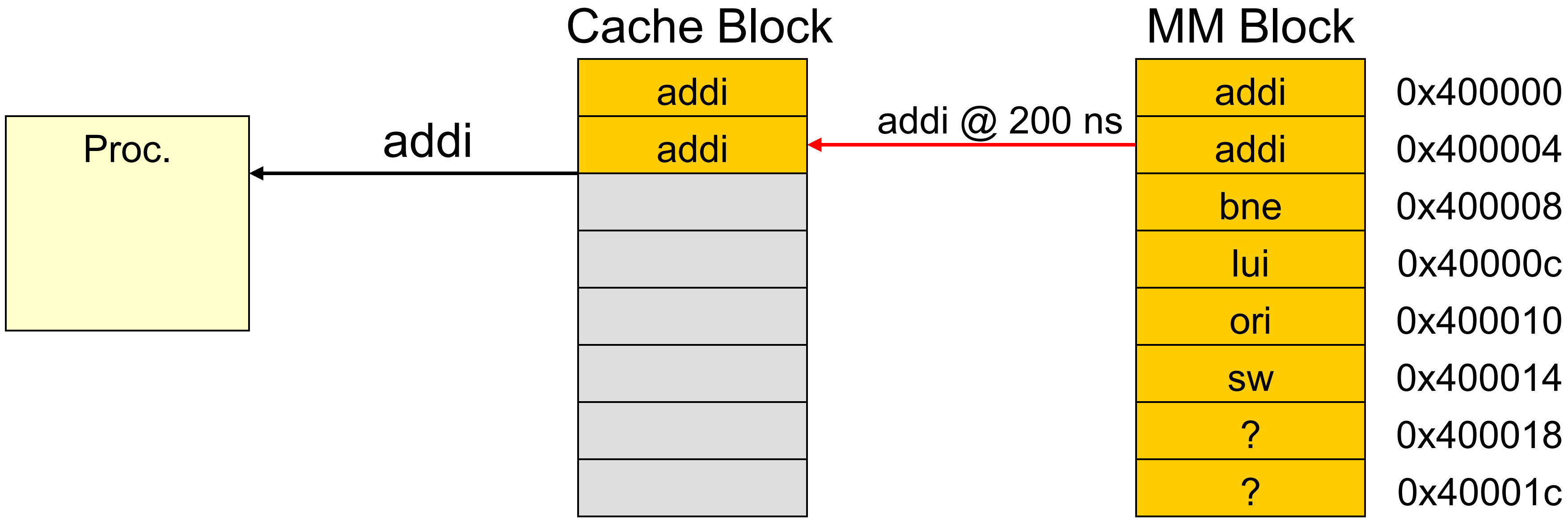
- After 1st word is returned processor could request 2nd word...cache is already fetching



Time: 110 ns

Early-Restart Cache

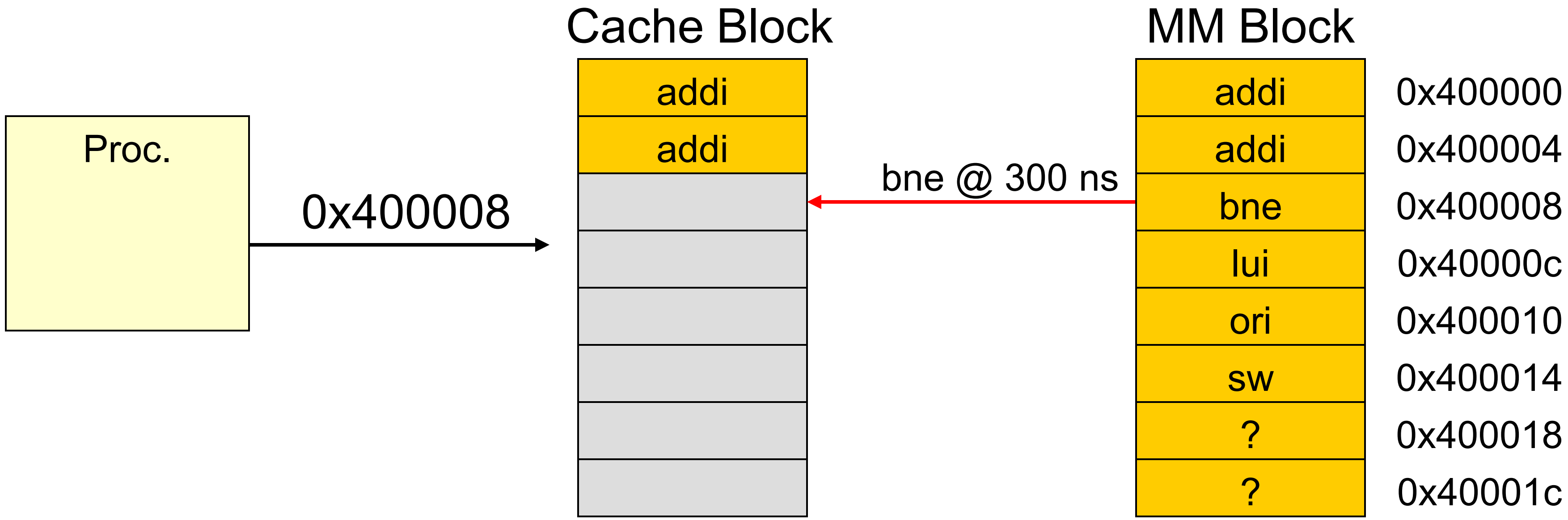
- Word is returned as soon as cache receives it



Time: 210 ns

Early-Restart Cache

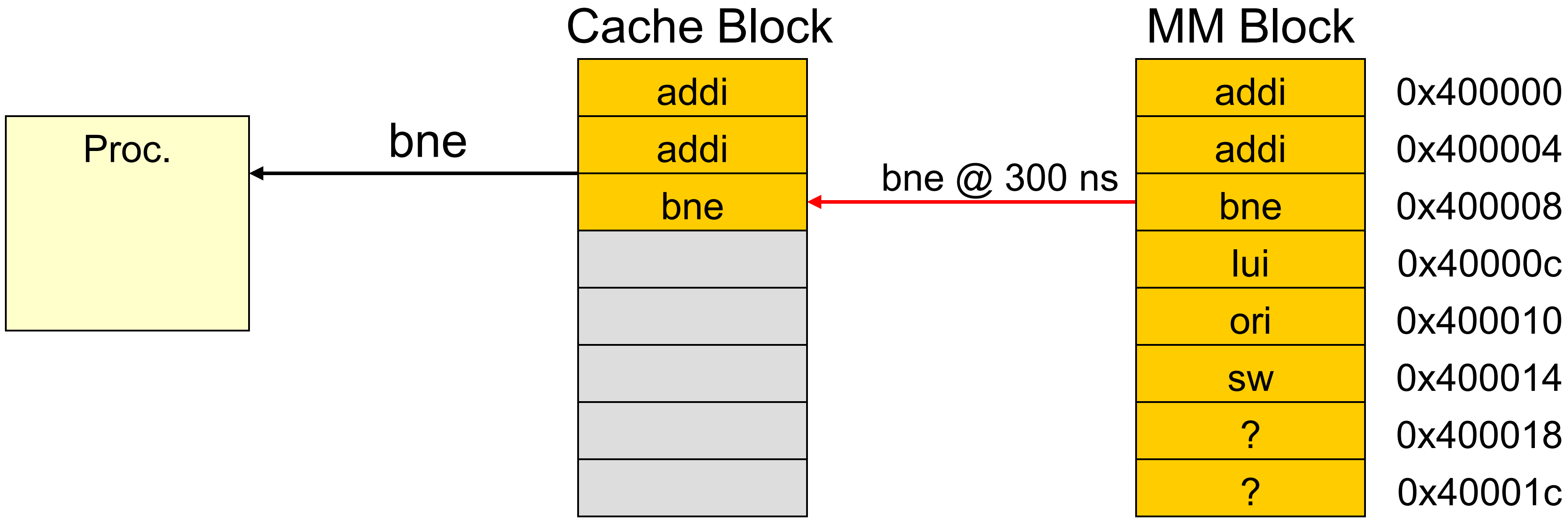
- Processor immediately requests next word



Time: 210 ns

Early-Restart Cache

- Returned to processor as soon as cache receives it

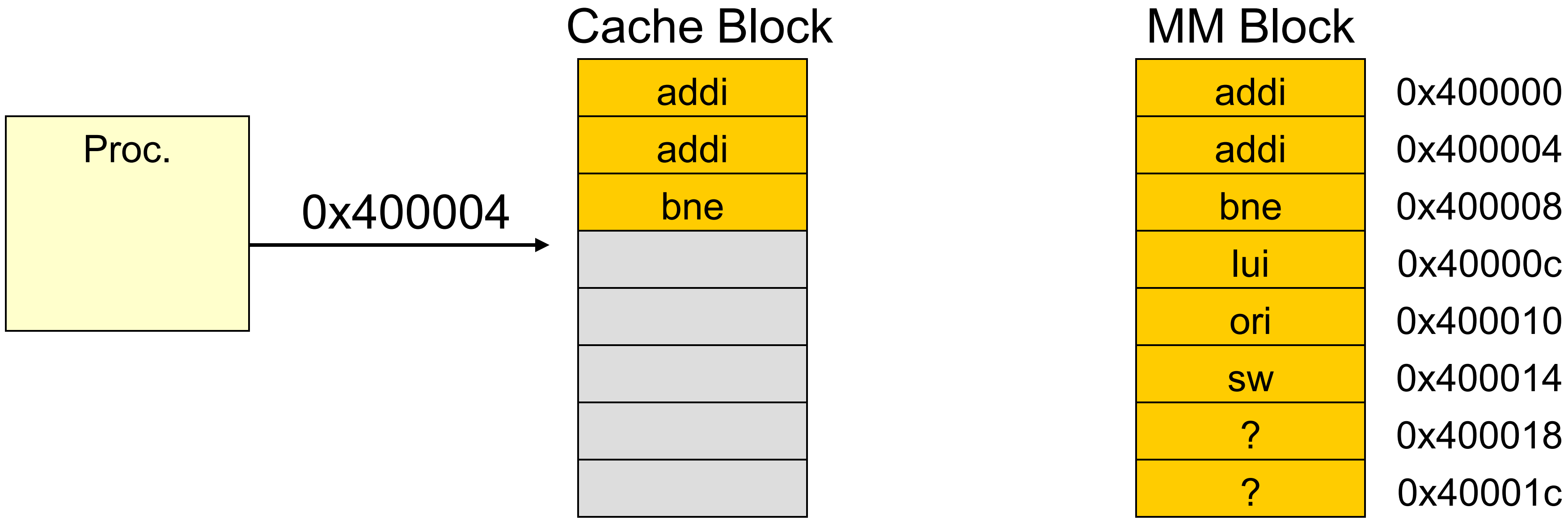


Time: 310 ns

Processor can continue

Early-Restart Cache

- Now due to loop, processor accesses word already in cache, it can get it in only the cache hit time (10 ns)

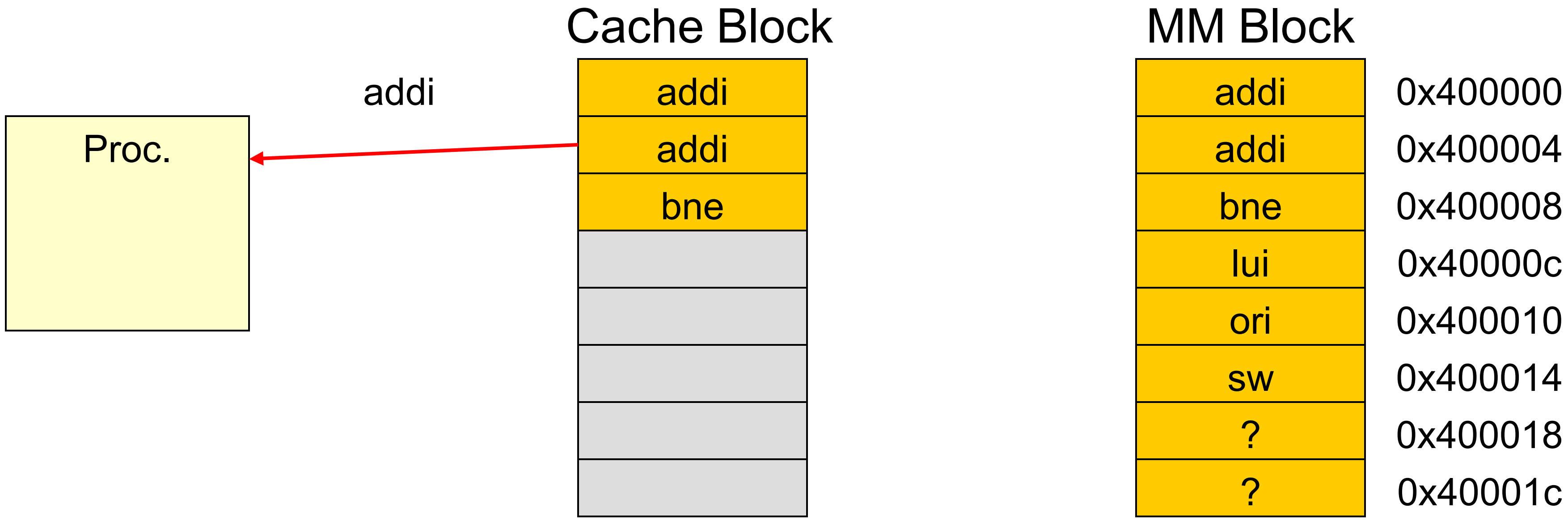


Time: 310 ns

Processor waits

Early-Restart Cache

- If processor accesses word already in cache, it can get it in only the cache hit time (10 ns)

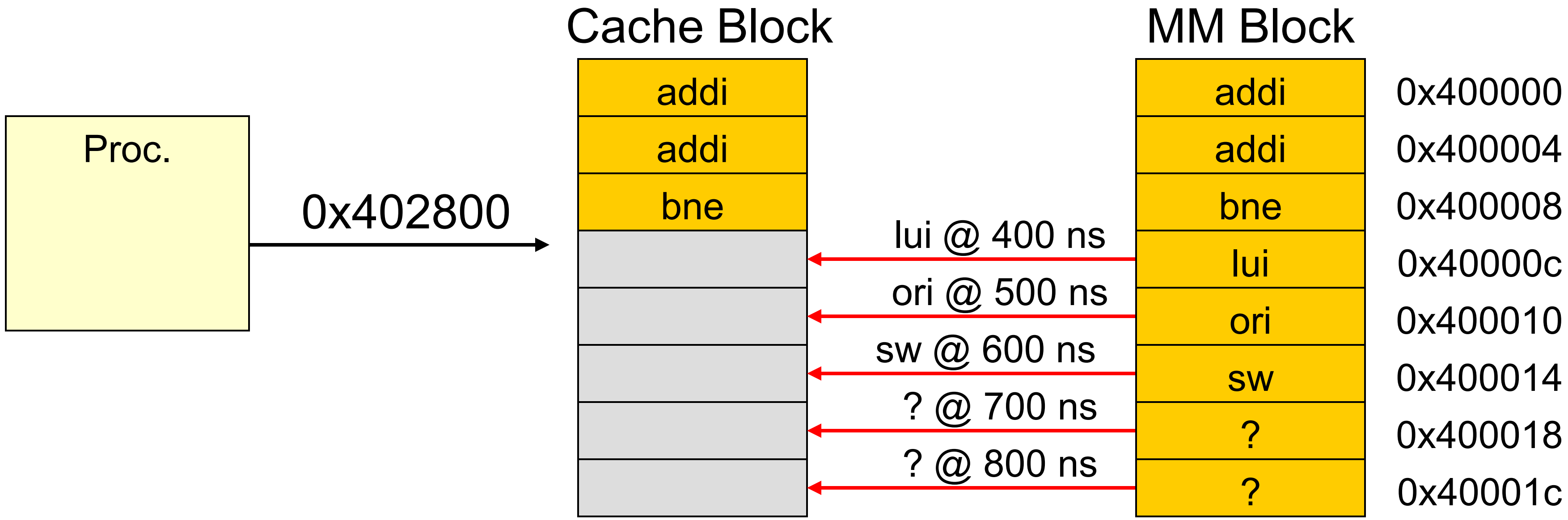


Time: 320 ns

Processor can continue

Early-Restart Cache

- If processor requests a word in another block it must wait for the remainder of the current block



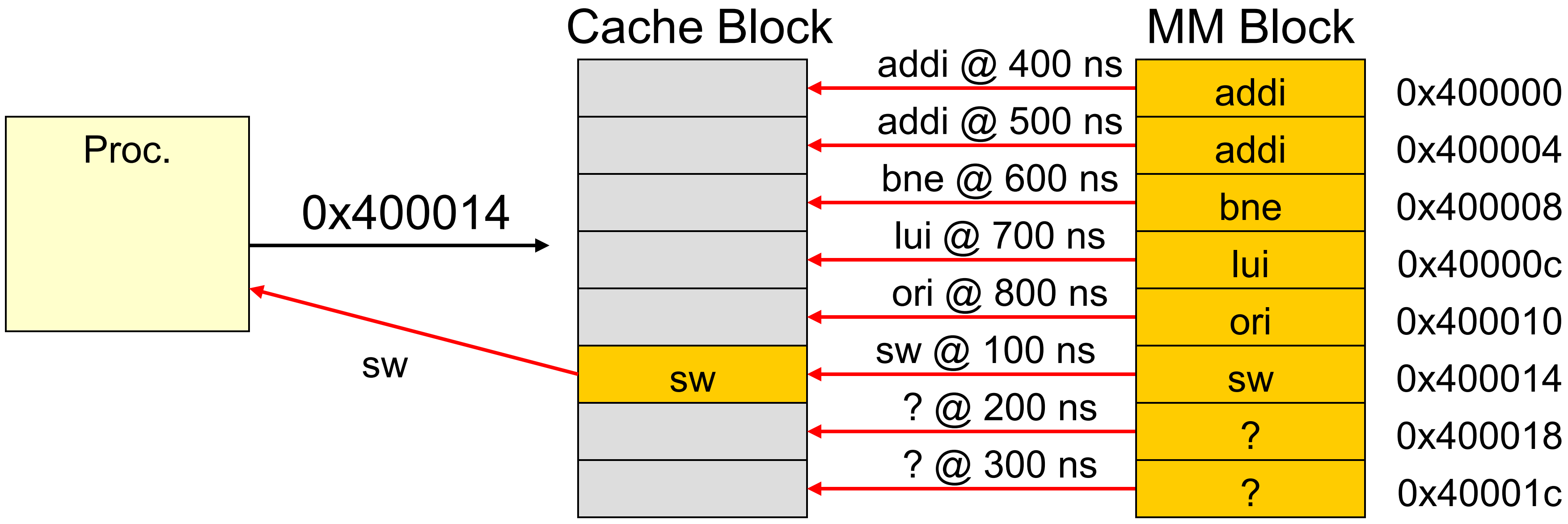
Cache Block		MM Block	
addi		addi	0x400000
addi		addi	0x400004
bne		bne	0x400008
	lui @ 400 ns	lui	0x40000c
	ori @ 500 ns	ori	0x400010
	sw @ 600 ns	sw	0x400014
	? @ 700 ns	?	0x400018
	? @ 800 ns	?	0x40001c

Time: 320 ns

Processor waits until 800 ns for cache to finish current block and keeps waiting until desired word is brought in from next block

Critical Word First Example

- If processor requests a word in the middle of the block, MM will return that word first and then the subsequent words, wrapping around the block address
(i.e. addr n mod block_size)



Time: 110 ns

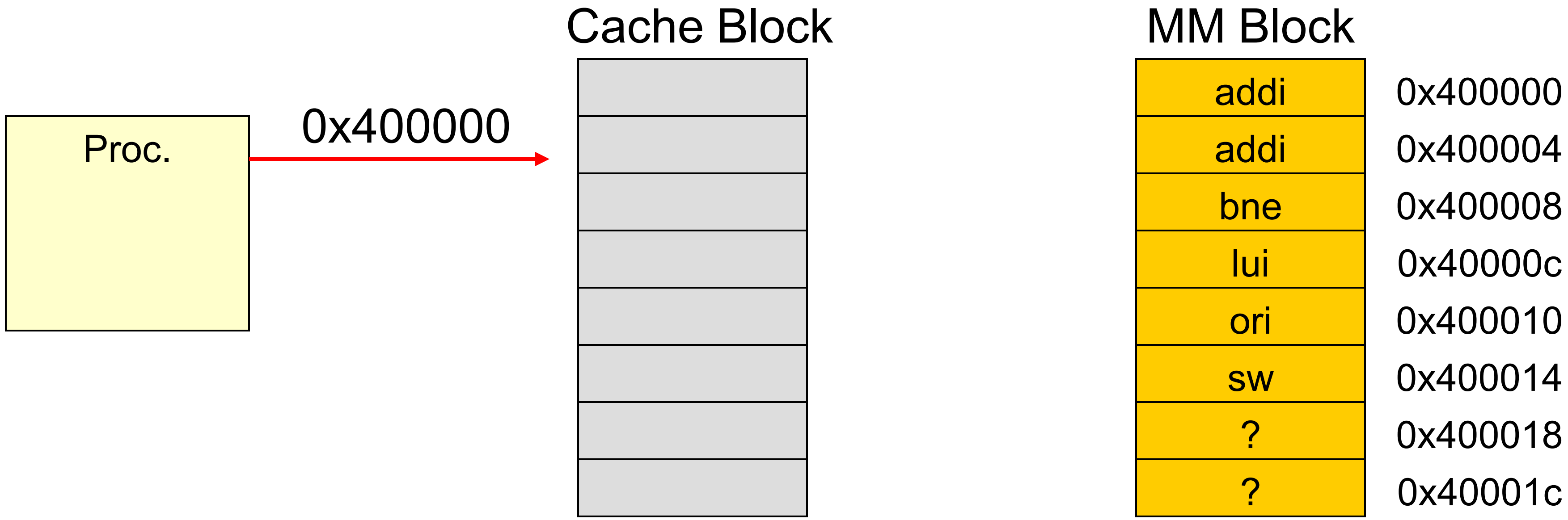
Processor waits

Early-Restart Summary

- As soon as the cache gets the requested word it will return it to the processor and the processor can continue while the cache brings in the rest of the block
- If the processor requests a new word that the cache does not have yet, it must wait until the cache gets that word (the cache may be busy bringing in the rest of the block)
- Critical word first starts with the desired word and brings in the rest of the block after that

No Early-Restart Cache

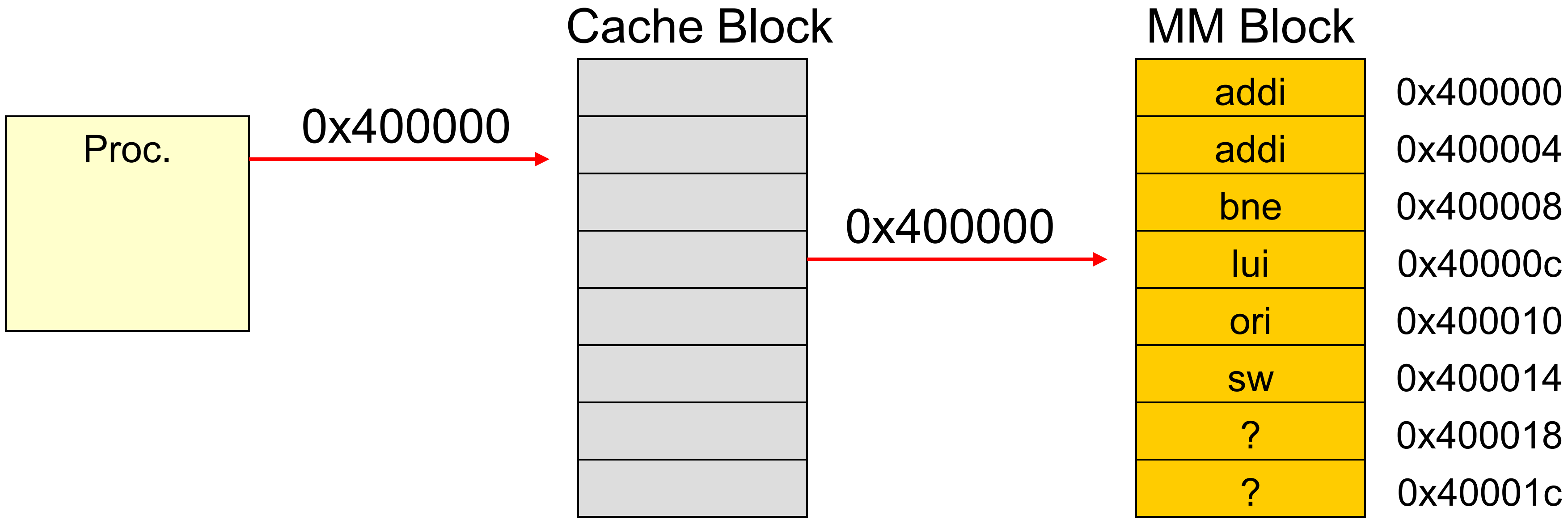
- No Early-Restart – cache fetches entire block before returning desired word to processor



Time: 0 ns

No Early-Restart Cache

- No Early-Restart – cache fetches entire block before returning desired word to processor

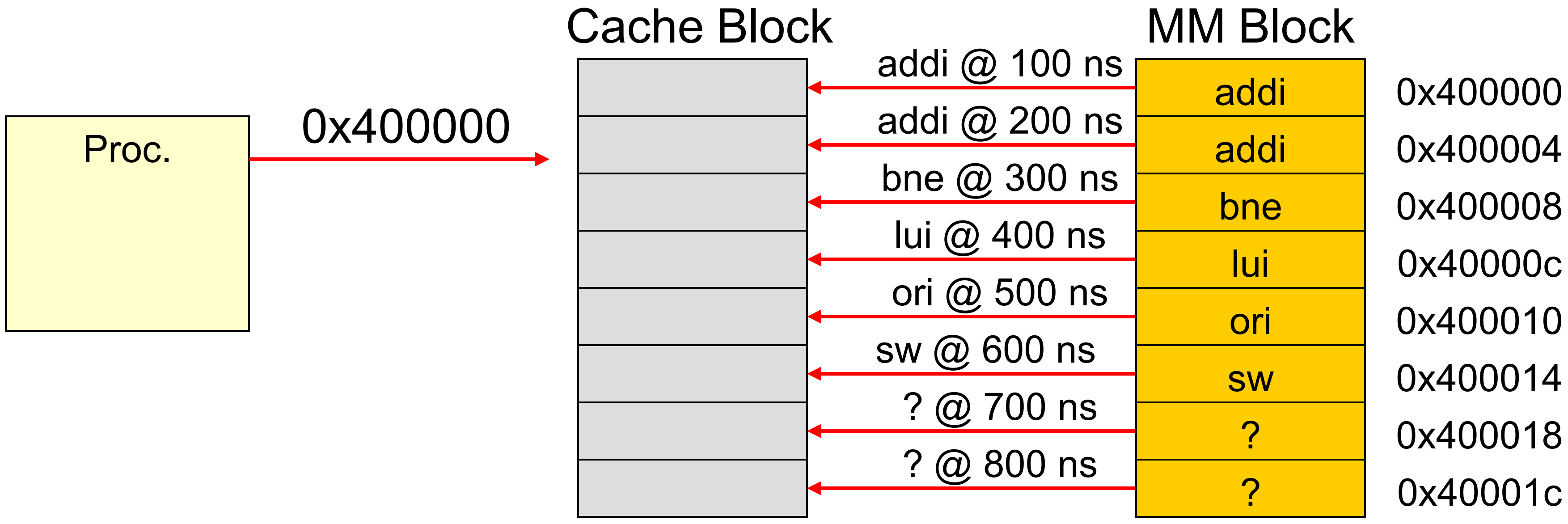


Time: 0 ns

Cache Miss

No Early-Restart Cache

- No Early-Restart – cache fetches entire block before returning desired word to processor

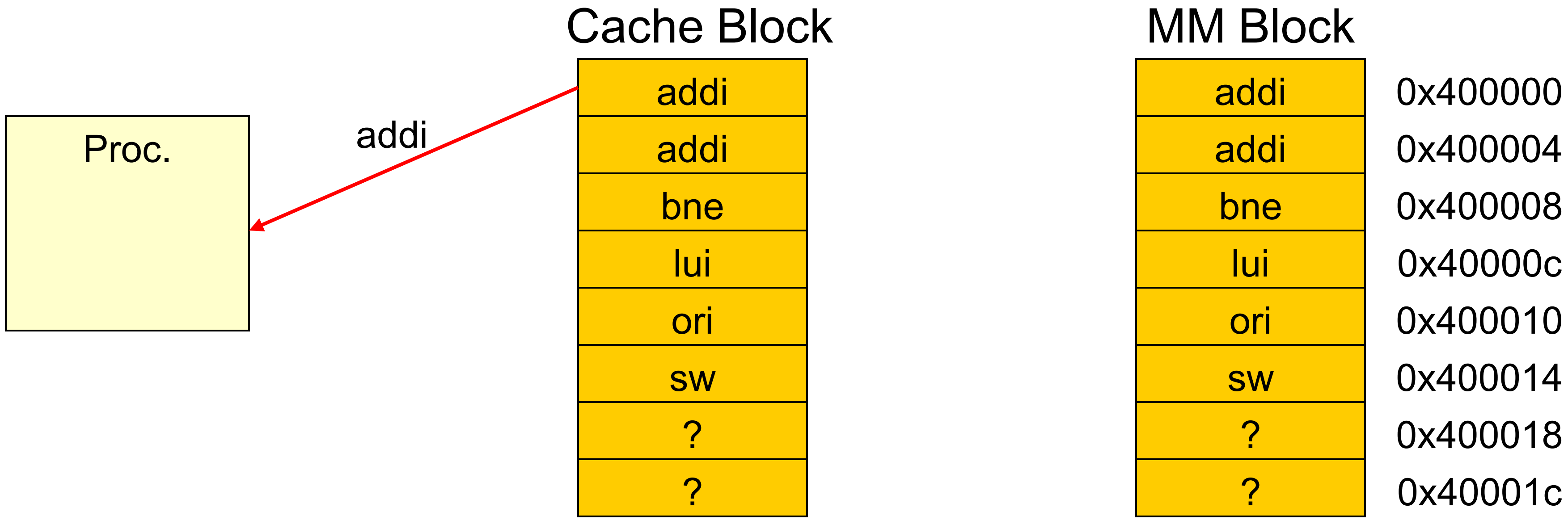


Time: 0 ns

Entire block is brought in while processor waits

No Early-Restart Cache

- No Early-Restart – cache fetches entire block before returning desired word to processor

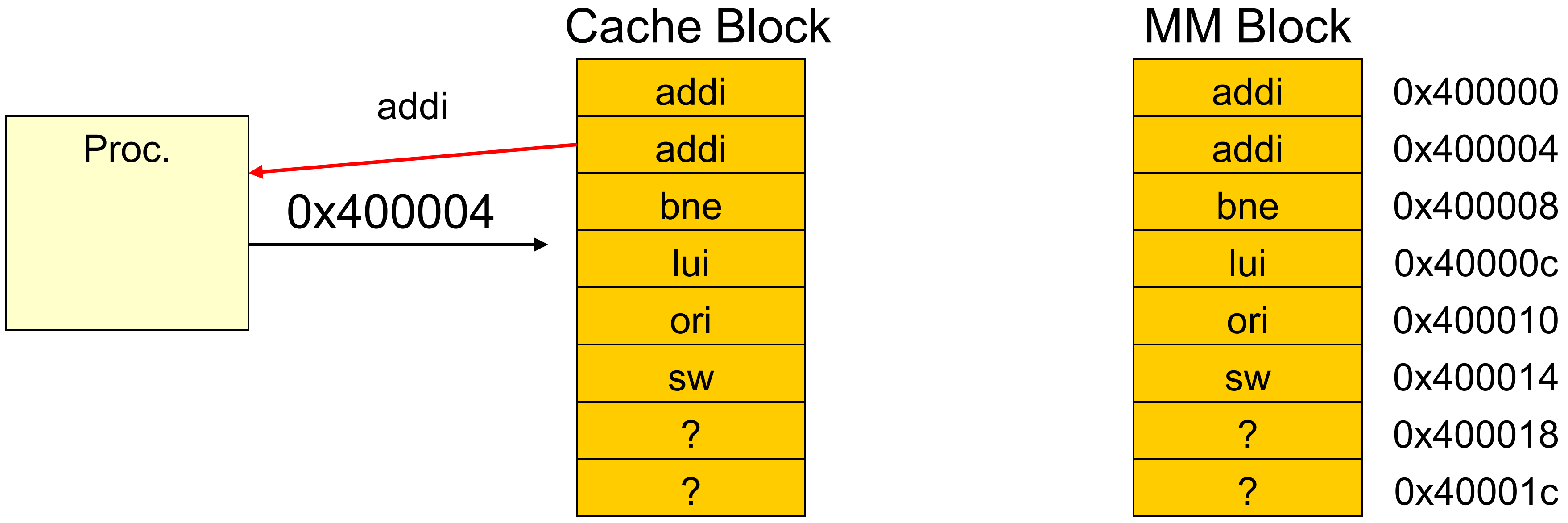


addi is returned after entire block is in cache

Time: 810 ns

No Early-Restart Cache

- Processor can request next word…already in the cache



Cache Block	MM Block	
addi	addi	0x400000
addi	addi	0x400004
bne	bne	0x400008
lui	lui	0x40000c
ori	ori	0x400010
sw	sw	0x400014
?	?	0x400018
?	?	0x40001c

addi returned after 10 ns

Time: 820 ns

No Early-Restart Summary

- The cache will always bring in the entire block before returning the requested word
- Then each subsequent access to that block can be serviced in the cache access time

Handling Writes

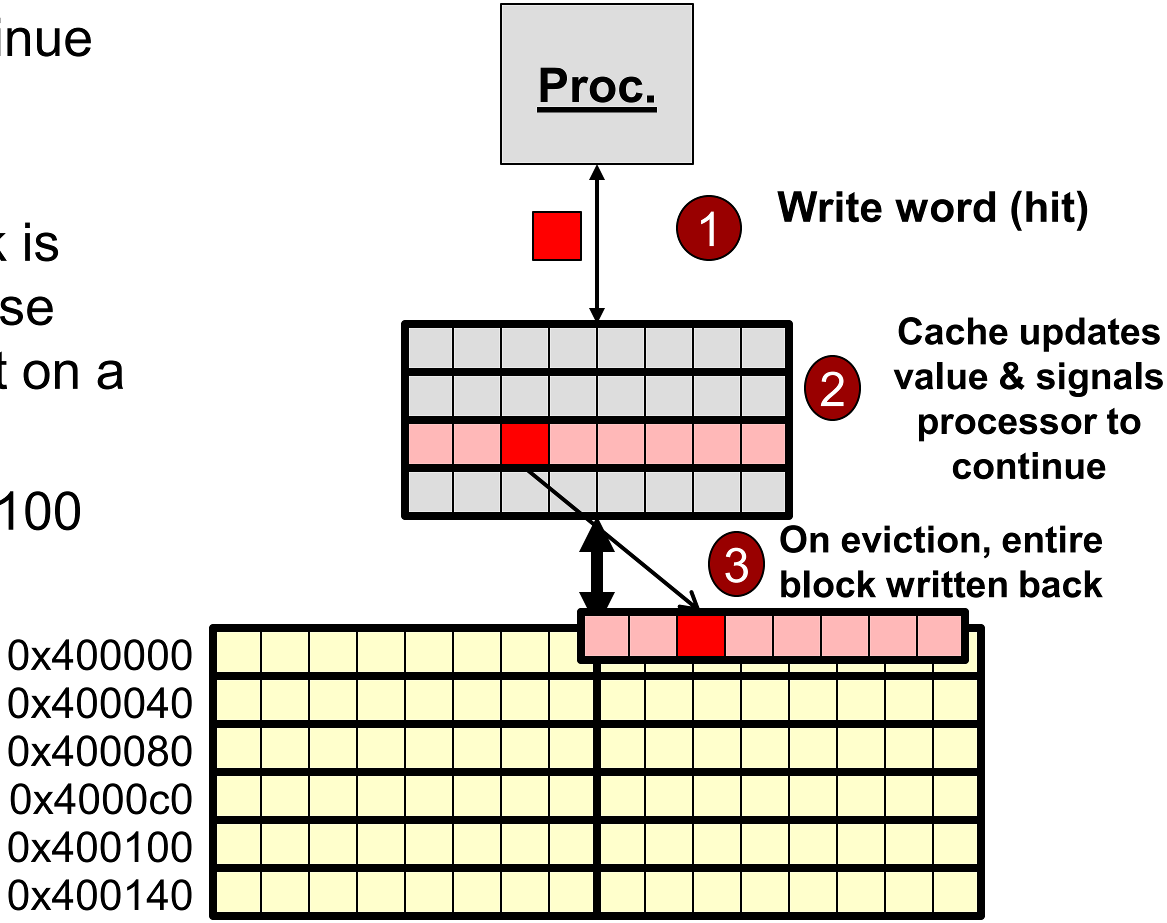
- Write-Back cache (using Write-Allocate)
- Write-Through cache (using No-Write-Allocate)

Write-Back Cache

- Word is only written in the cache (not MM)
- On replacement, entire block must be written back to MM

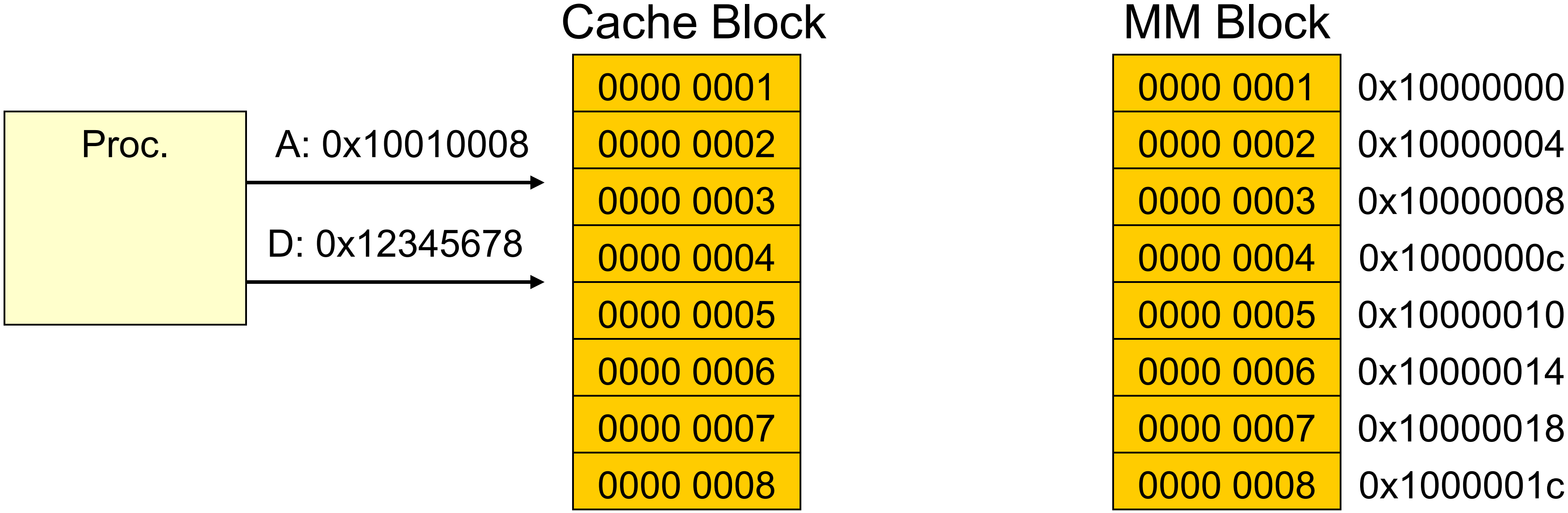
Write Back Cache

- On write-hit
  - Update only cached copy
  - Processor can continue quickly (e.g. 10 ns)
  - Later when block is evicted, entire block is written back (because bookkeeping is kept on a per block basis)
    - Ex: 8 words @ 100 ns per word for writing mem. = 800 ns


Proc.
1
Write word (hit)
2
Cache updates value & signals processor to continue
3
On eviction, entire block written back
0x400000
0x400040
0x400080
0x4000c0
0x400100
0x400140

Write-Back Cache

- Processor writes data to address 0x10010008 which is in cache



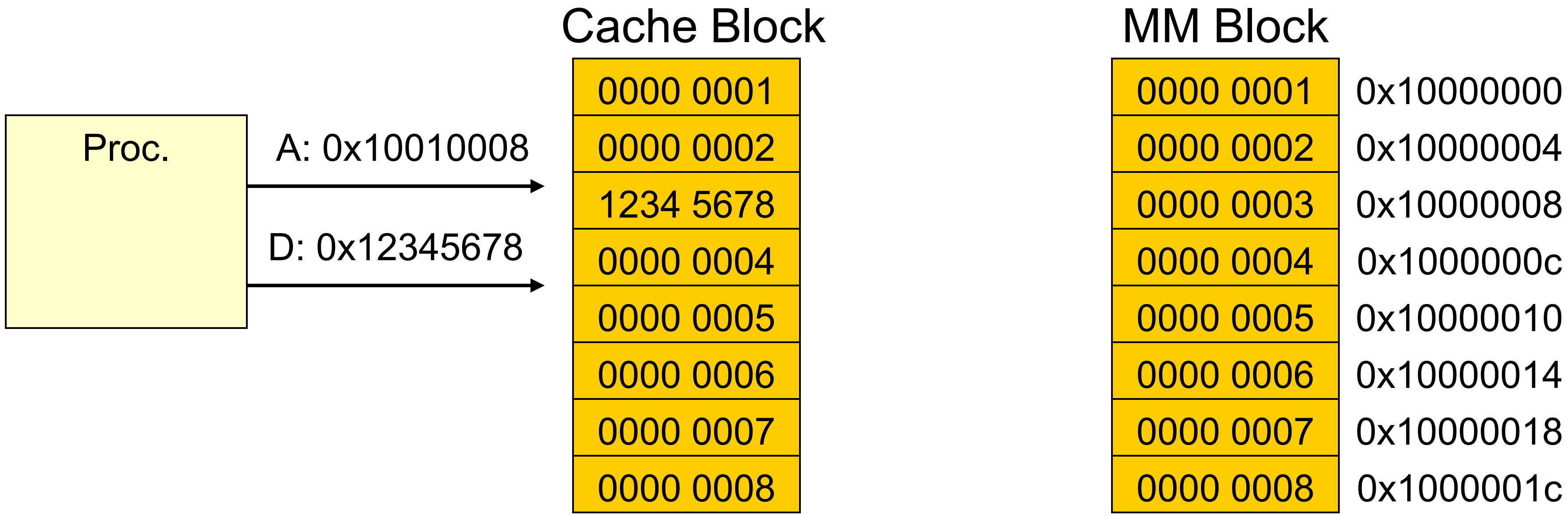
Cache Block
0000 0001
0000 0002
0000 0003
0000 0004
0000 0005
0000 0006
0000 0007
0000 0008

MM Block	
0000 0001	0x10000000
0000 0002	0x10000004
0000 0003	0x10000008
0000 0004	0x1000000c
0000 0005	0x10000010
0000 0006	0x10000014
0000 0007	0x10000018
0000 0008	0x1000001c

Time: 0 ns

Write-Back Cache

- Processor writes data to address 0x10010008 which is in cache



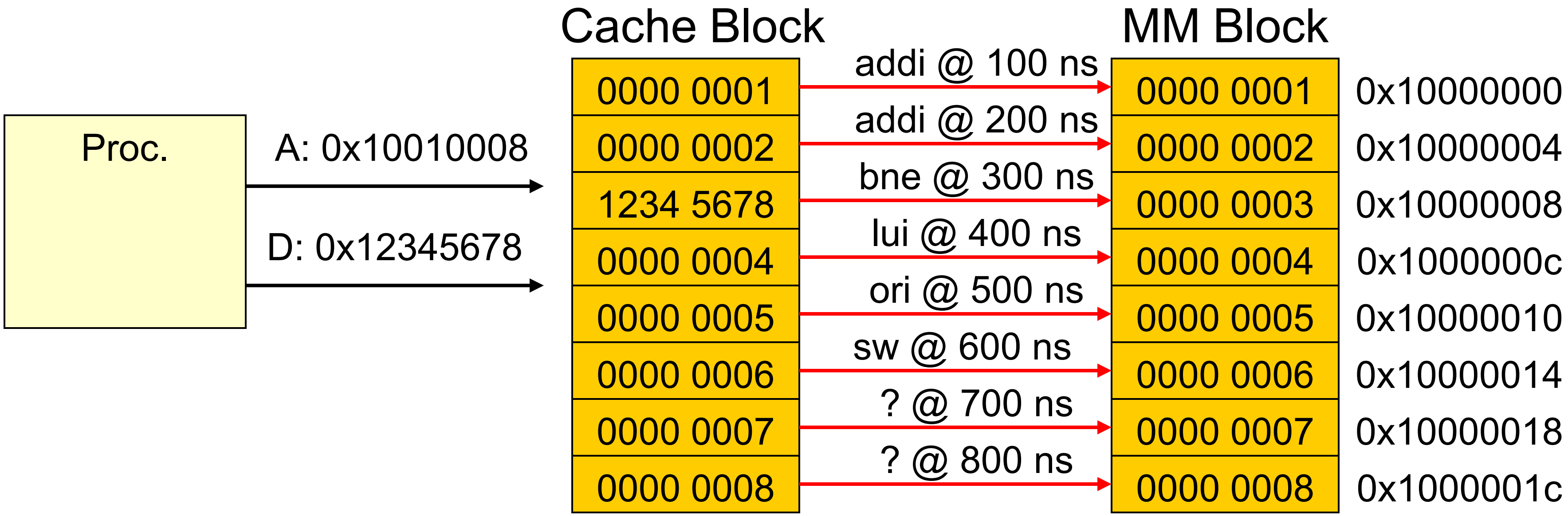
Cache Block	MM Block	
0000 0001	0000 0001	0x10000000
0000 0002	0000 0002	0x10000004
1234 5678	0000 0003	0x10000008
0000 0004	0000 0004	0x1000000c
0000 0005	0000 0005	0x10000010
0000 0006	0000 0006	0x10000014
0000 0007	0000 0007	0x10000018
0000 0008	0000 0008	0x1000001c

Time: 10 ns

Processor can continue after 10 ns it takes to write the cache only

Write-Back Cache

- At some point the entire cache line (block) needs to be written back to MM



Proc.

A: 0x10010008

D: 0x12345678

Cache Block		MM Block	
0000 0001	addi @ 100 ns	0000 0001	0x10000000
0000 0002	addi @ 200 ns	0000 0002	0x10000004
1234 5678	bne @ 300 ns	0000 0003	0x10000008
0000 0004	lui @ 400 ns	0000 0004	0x1000000c
0000 0005	ori @ 500 ns	0000 0005	0x10000010
0000 0006	sw @ 600 ns	0000 0006	0x10000014
0000 0007	? @ 700 ns	0000 0007	0x10000018
0000 0008	? @ 800 ns	0000 0008	0x1000001c

Time: 10 ns

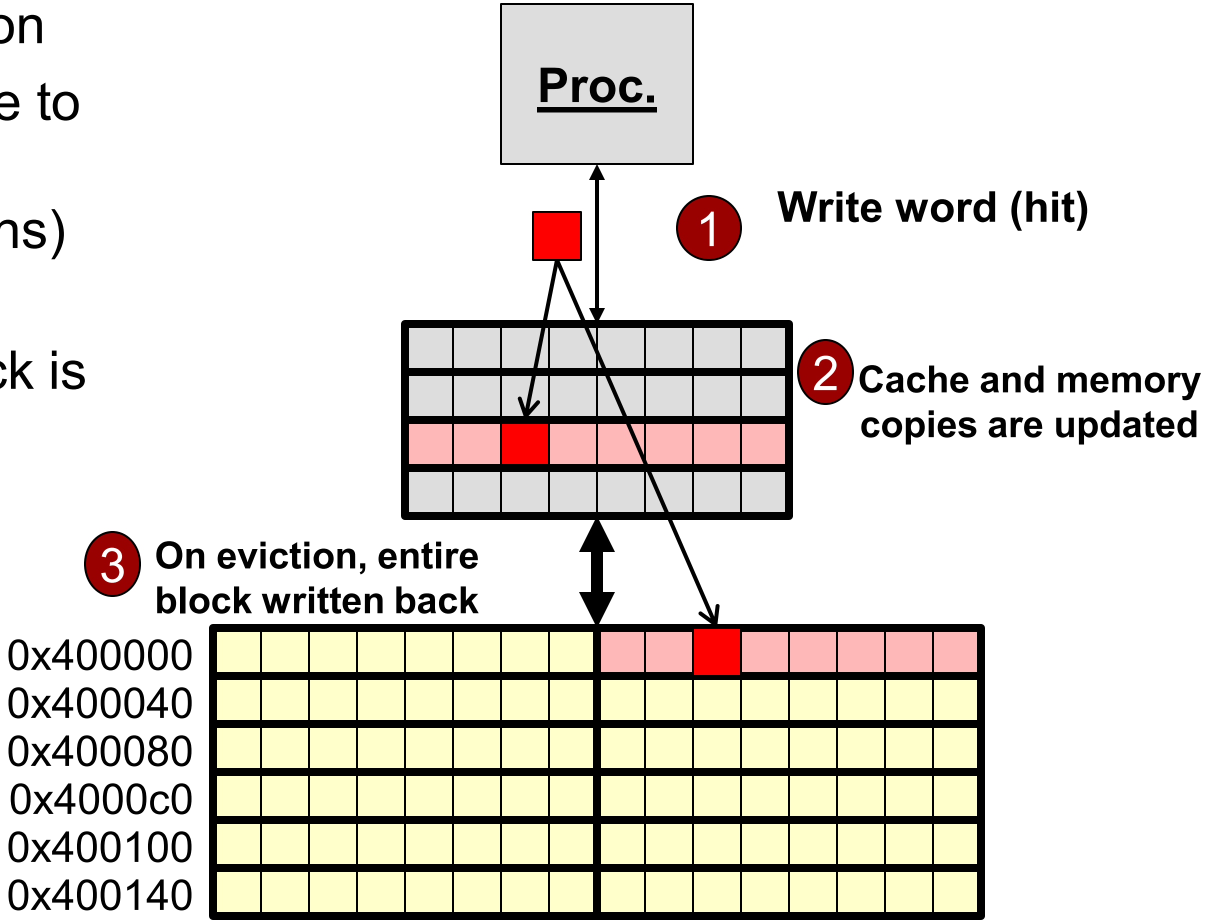
It will take 800 ns at some point to write the cache line back

Write-Through Cache

- Word is written in both cache and MM
- On replacement, block does not have to be written back since MM already has updated values

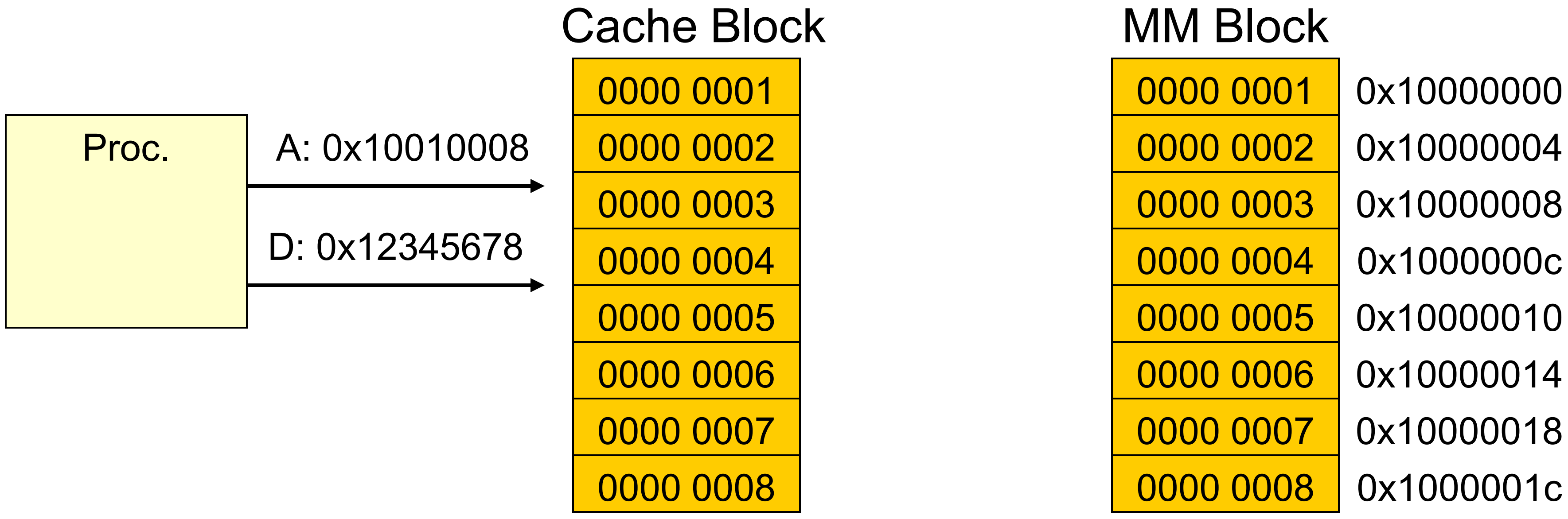
Write Through Cache

- On write-hit
  - Update both cached and main memory version
  - Processor may have to wait for memory to complete (e.g. 100 ns)
  - Later when block is evicted, no writeback is needed


Proc.
1
Write word (hit)
2
Cache and memory copies are updated
3
On eviction, entire block written back
0x400000
0x400040
0x400080
0x4000c0
0x400100
0x400140

Write-Through Cache

- Processor writes data to address 0x10010008 which is in cache



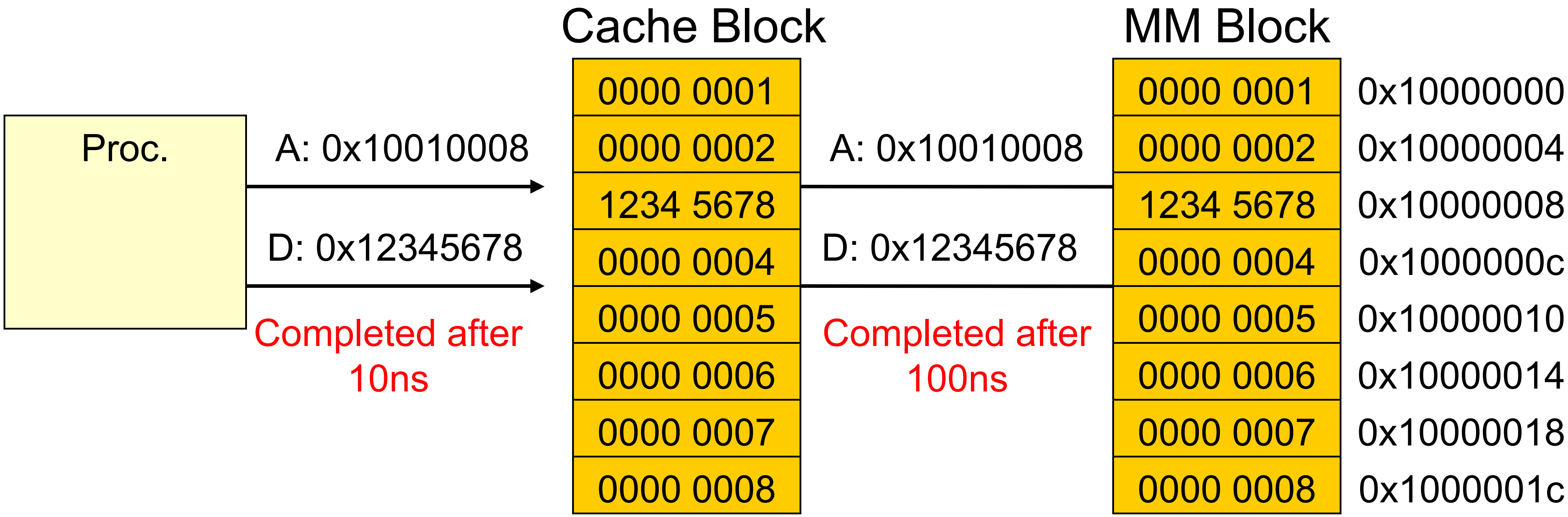
Cache Block
0000 0001
0000 0002
0000 0003
0000 0004
0000 0005
0000 0006
0000 0007
0000 0008

MM Block	
0000 0001	0x10000000
0000 0002	0x10000004
0000 0003	0x10000008
0000 0004	0x1000000c
0000 0005	0x10000010
0000 0006	0x10000014
0000 0007	0x10000018
0000 0008	0x1000001c

Time: 0 ns

Write-Through Cache

- Processor writes data to address 0x10010008 which is in cache and to MM



Proc.

A: 0x10010008

D: 0x12345678

Completed after 10ns

A: 0x10010008

D: 0x12345678

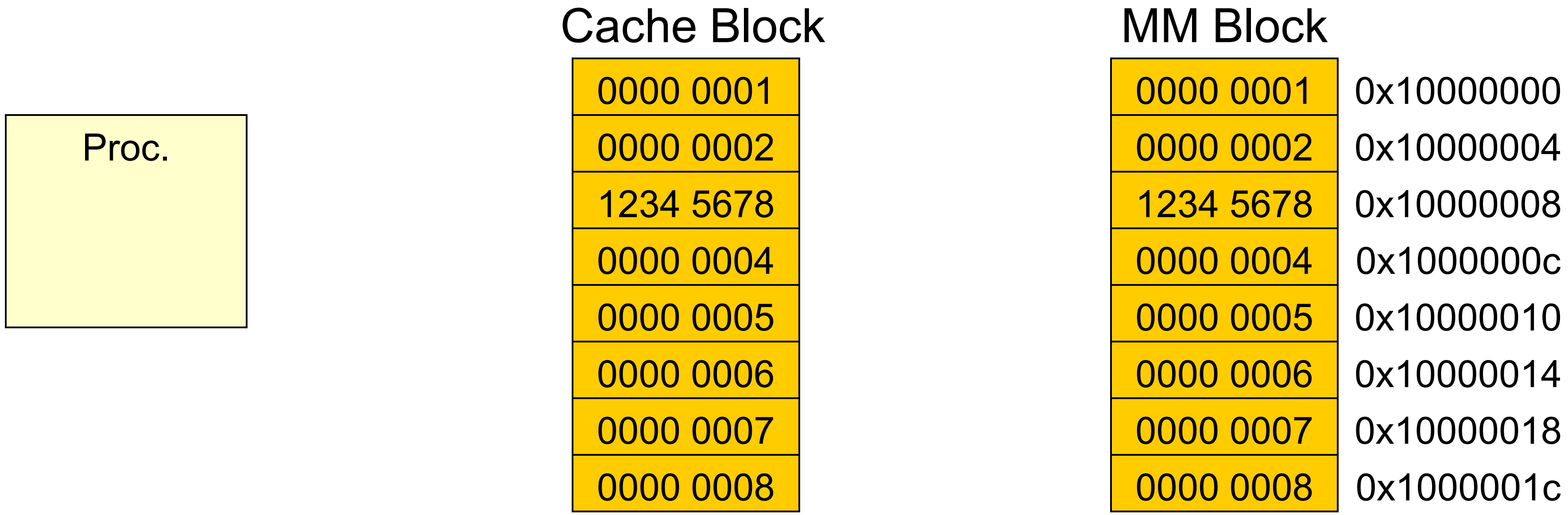
Completed after 100ns

Cache Block	MM Block	
0000 0001	0000 0001	0x10000000
0000 0002	0000 0002	0x10000004
1234 5678	1234 5678	0x10000008
0000 0004	0000 0004	0x1000000c
0000 0005	0000 0005	0x10000010
0000 0006	0000 0006	0x10000014
0000 0007	0000 0007	0x10000018
0000 0008	0000 0008	0x1000001c

Time: 100 ns

Write-Through Cache

- When block is replaced in cache it need not be written back to MM



Cache Block	MM Block	
0000 0001	0000 0001	0x10000000
0000 0002	0000 0002	0x10000004
1234 5678	1234 5678	0x10000008
0000 0004	0000 0004	0x1000000c
0000 0005	0000 0005	0x10000010
0000 0006	0000 0006	0x10000014
0000 0007	0000 0007	0x10000018
0000 0008	0000 0008	0x1000001c

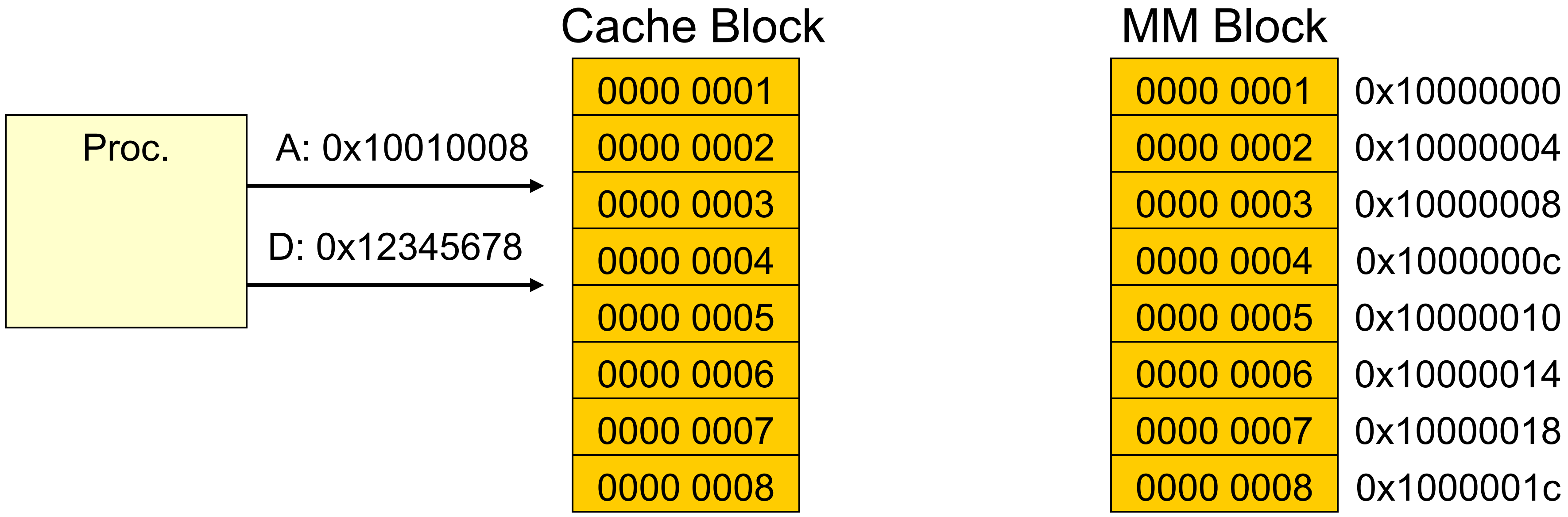
Time: 100 ns

Example

- Assume we execute a loop 40 times, and each iteration writes to address 0x10010008

Write-Back Cache

- Iteration 1: Processor writes data to address 0x10010008 which is in cache



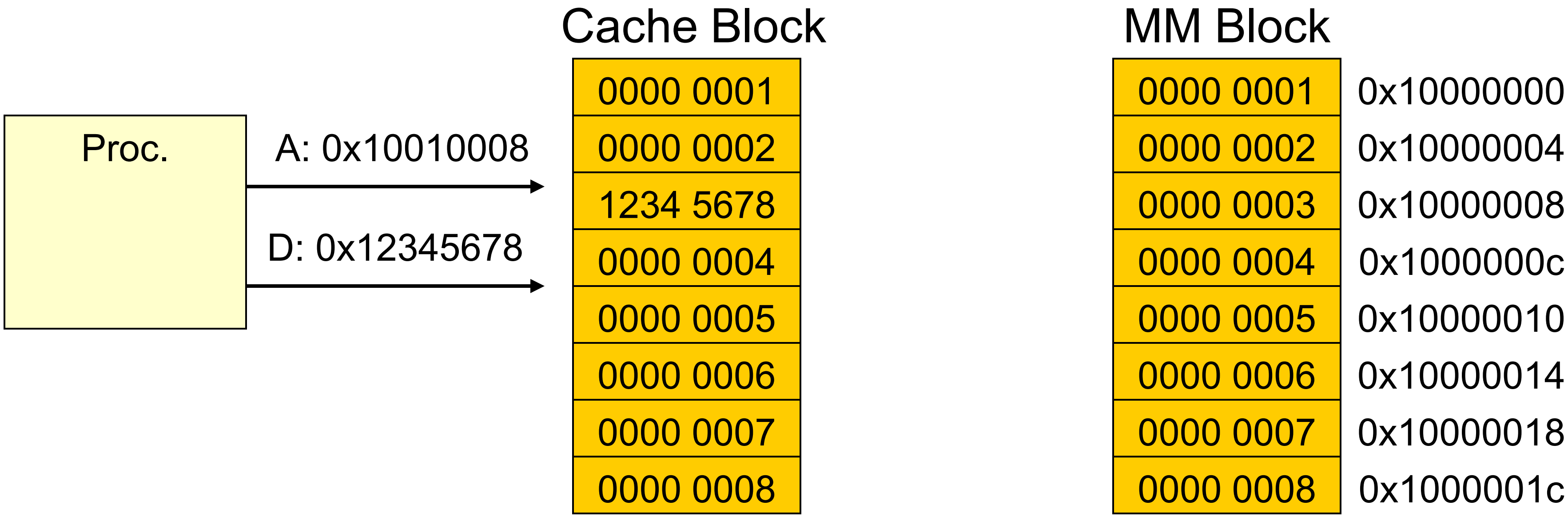
Cache Block
0000 0001
0000 0002
0000 0003
0000 0004
0000 0005
0000 0006
0000 0007
0000 0008

MM Block	
0000 0001	0x10000000
0000 0002	0x10000004
0000 0003	0x10000008
0000 0004	0x1000000c
0000 0005	0x10000010
0000 0006	0x10000014
0000 0007	0x10000018
0000 0008	0x1000001c

Time: 0 ns

Write-Back Cache

- Iteration 1 completes after 10 ns.



Proc.

A: 0x10010008

D: 0x12345678

Cache Block

Cache Block
0000 0001
0000 0002
1234 5678
0000 0004
0000 0005
0000 0006
0000 0007
0000 0008

MM Block

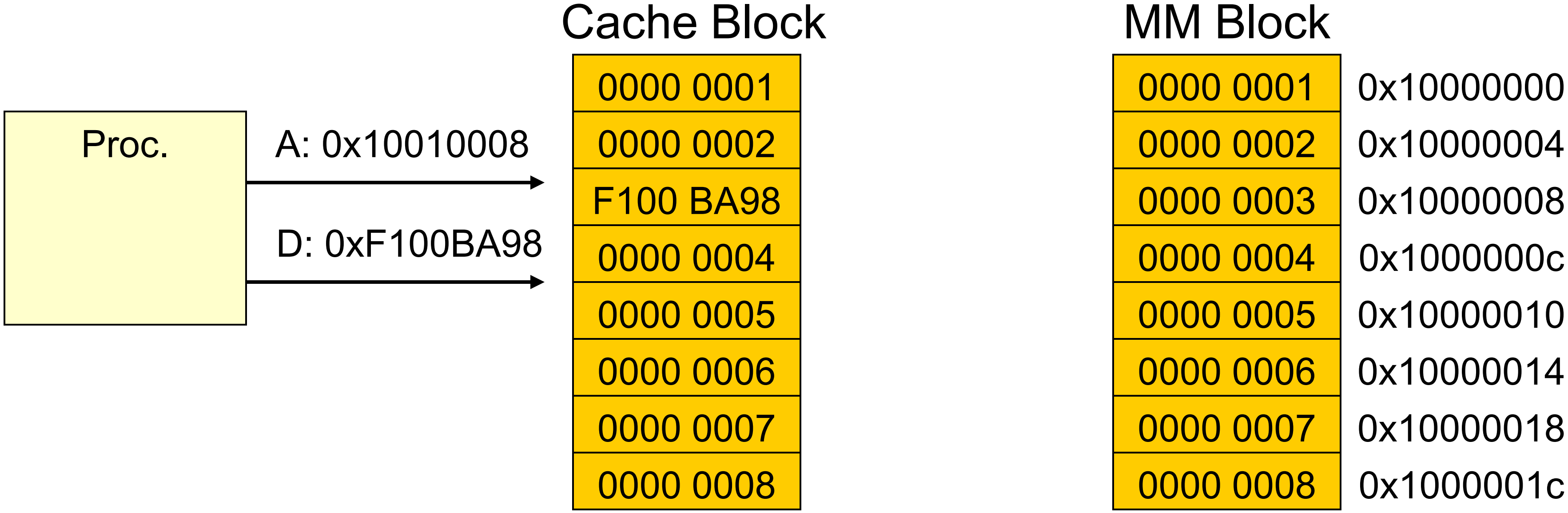
MM Block	
0000 0001	0x10000000
0000 0002	0x10000004
0000 0003	0x10000008
0000 0004	0x1000000c
0000 0005	0x10000010
0000 0006	0x10000014
0000 0007	0x10000018
0000 0008	0x1000001c

Time: 10 ns

Processor can continue after 10 ns it takes to write the cache only

Write-Back Cache

- Iteration 2 completes after 20 ns.



Cache Block
0000 0001
0000 0002
F100 BA98
0000 0004
0000 0005
0000 0006
0000 0007
0000 0008

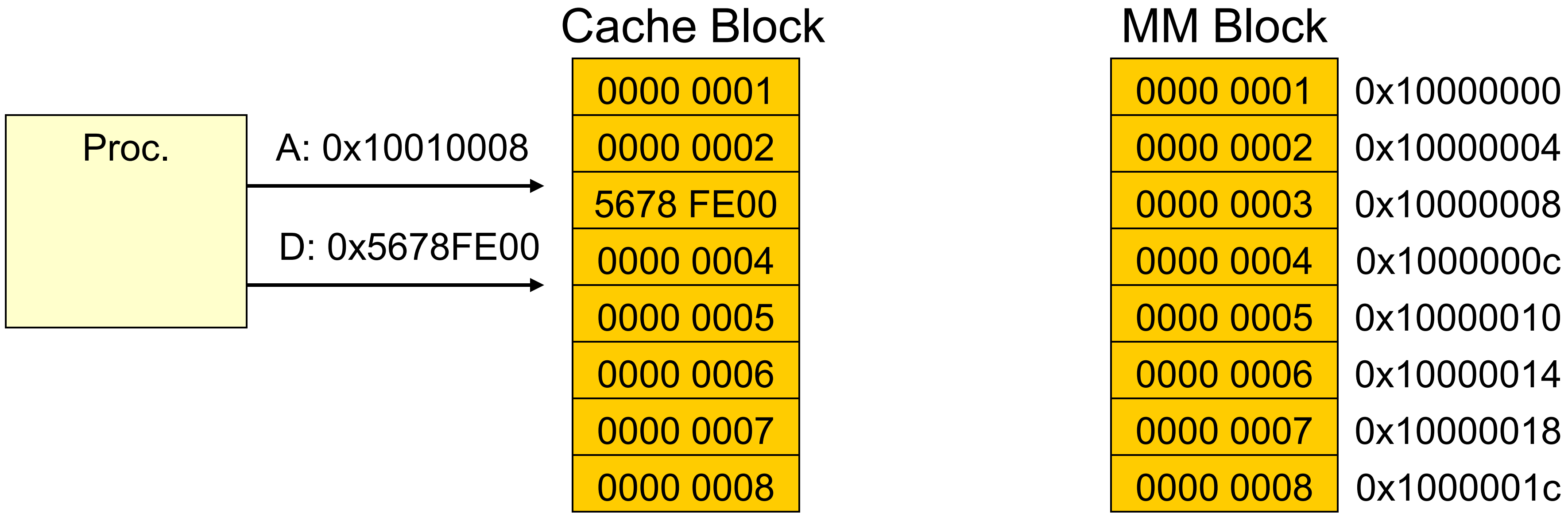
MM Block	
0000 0001	0x10000000
0000 0002	0x10000004
0000 0003	0x10000008
0000 0004	0x1000000c
0000 0005	0x10000010
0000 0006	0x10000014
0000 0007	0x10000018
0000 0008	0x1000001c

Time: 20 ns

Processor can continue after 10 ns it takes to write the cache only

Write-Back Cache

- Iteration 40: Processor writes data to address 0x10010008 which is in cache



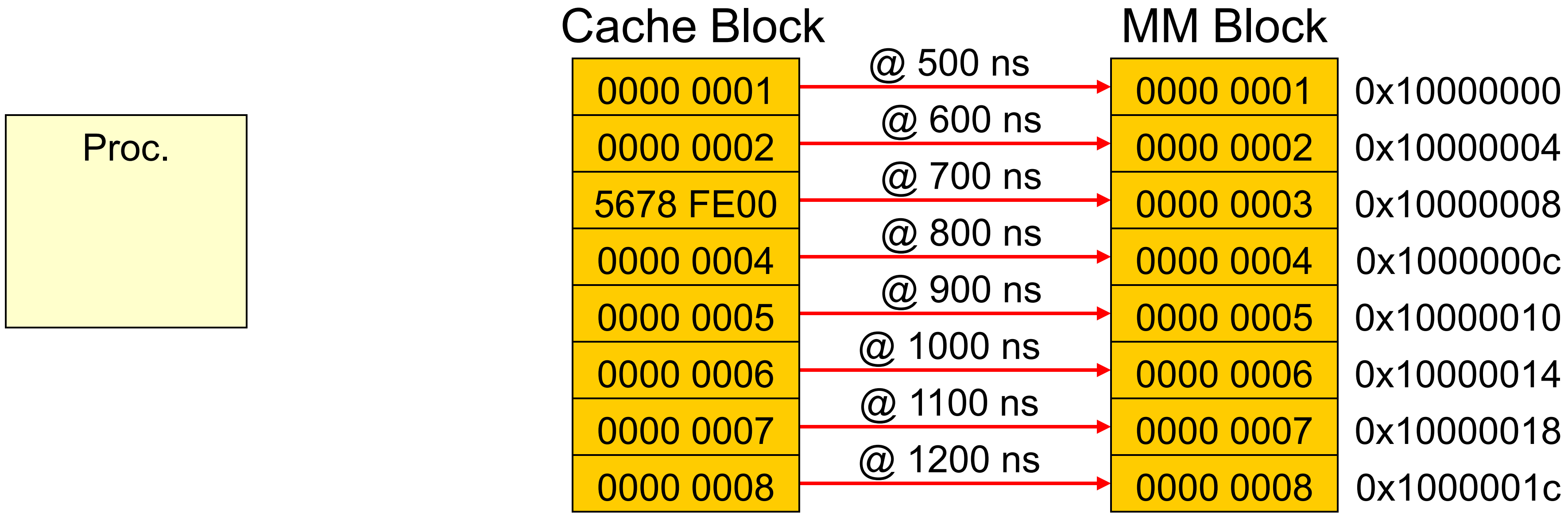
Cache Block
0000 0001
0000 0002
5678 FE00
0000 0004
0000 0005
0000 0006
0000 0007
0000 0008

MM Block	
0000 0001	0x10000000
0000 0002	0x10000004
0000 0003	0x10000008
0000 0004	0x1000000c
0000 0005	0x10000010
0000 0006	0x10000014
0000 0007	0x10000018
0000 0008	0x1000001c

Time: 400 ns

Write-Back Cache

- On replacement, the entire block must be written back (8 words * 100 ns)

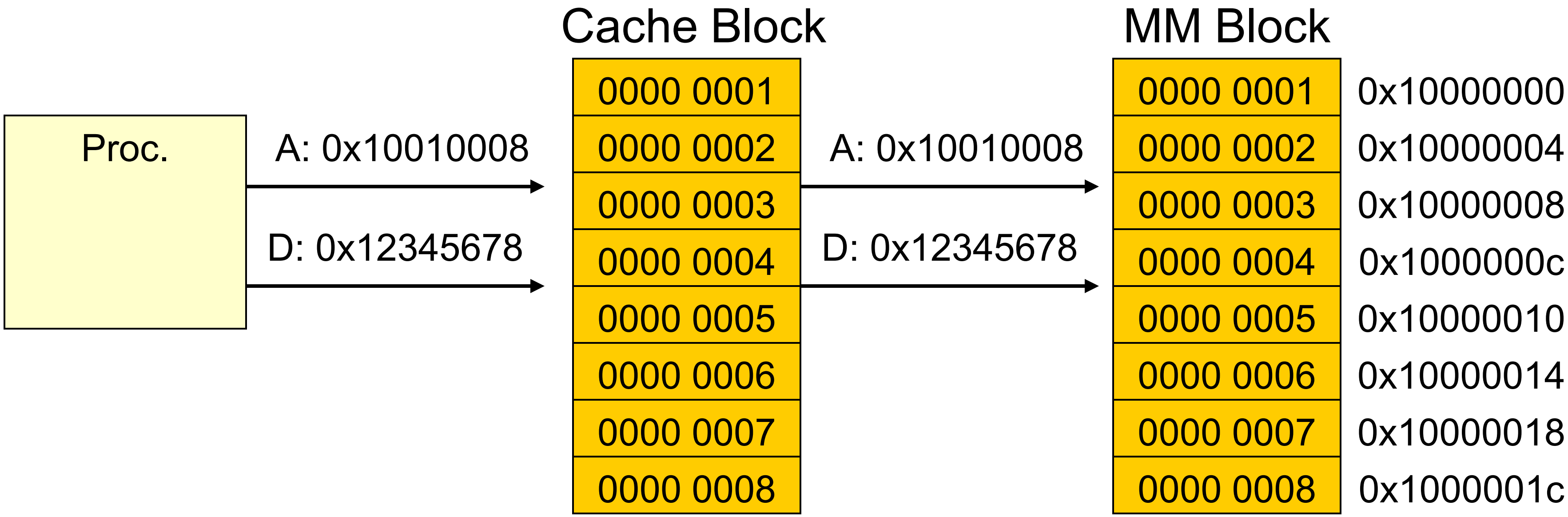


Cache Block		MM Block	
0000 0001	@ 500 ns	0000 0001	0x10000000
0000 0002	@ 600 ns	0000 0002	0x10000004
5678 FE00	@ 700 ns	0000 0003	0x10000008
0000 0004	@ 800 ns	0000 0004	0x1000000c
0000 0005	@ 900 ns	0000 0005	0x10000010
0000 0006	@ 1000 ns	0000 0006	0x10000014
0000 0007	@ 1100 ns	0000 0007	0x10000018
0000 0008	@ 1200 ns	0000 0008	0x1000001c

Time: 1200 ns

Write-Through Cache

- Iteration 1: Processor writes data to address 0x10010008 which is in cache



Proc.

A: 0x10010008

D: 0x12345678

Cache Block

Cache Block
0000 0001
0000 0002
0000 0003
0000 0004
0000 0005
0000 0006
0000 0007
0000 0008

A: 0x10010008

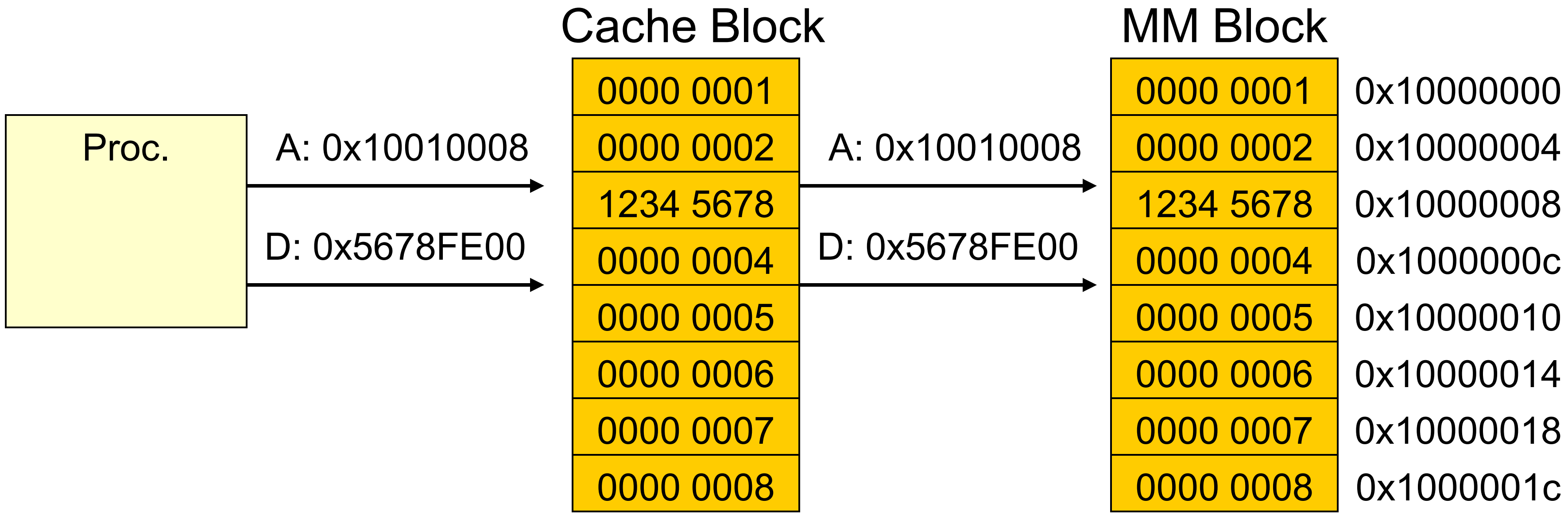
D: 0x12345678

MM Block	
0000 0001	0x10000000
0000 0002	0x10000004
0000 0003	0x10000008
0000 0004	0x1000000c
0000 0005	0x10000010
0000 0006	0x10000014
0000 0007	0x10000018
0000 0008	0x1000001c

Time: 100 ns

Write-Through Cache

- Iteration 40: Processor writes data "ABCD" to address 8000 which is in cache



Proc.

A: 0x10010008

D: 0x5678FE00

A: 0x10010008

D: 0x5678FE00

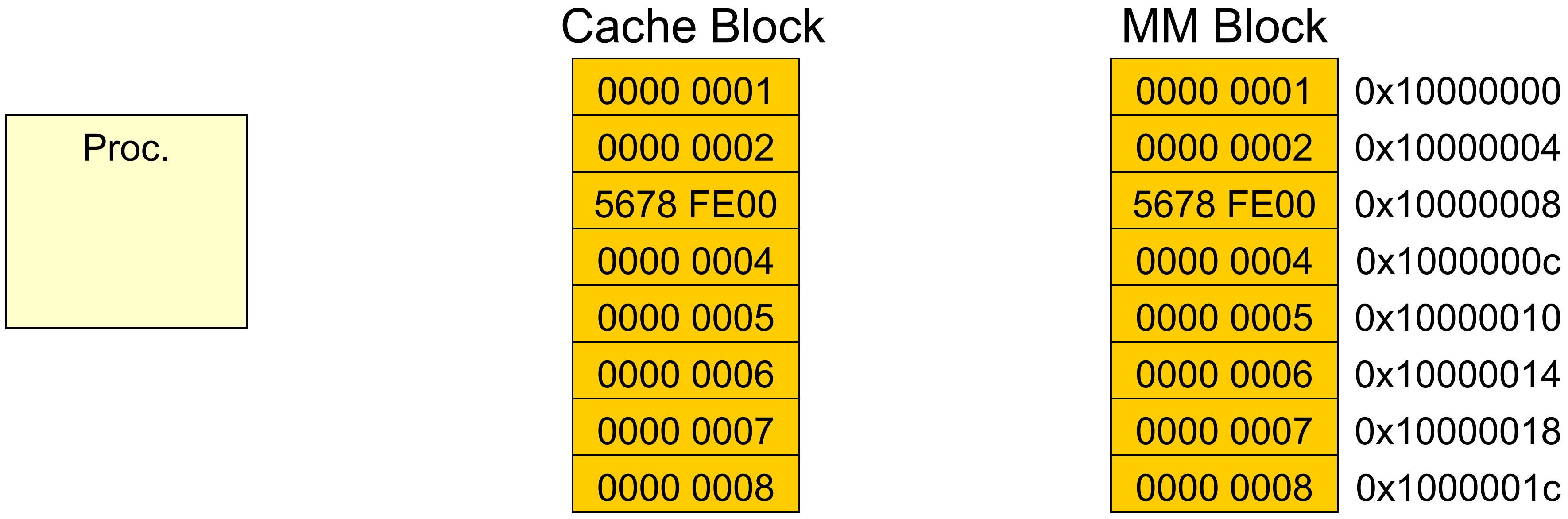
Cache Block
0000 0001
0000 0002
1234 5678
0000 0004
0000 0005
0000 0006
0000 0007
0000 0008

MM Block	
0000 0001	0x10000000
0000 0002	0x10000004
1234 5678	0x10000008
0000 0004	0x1000000c
0000 0005	0x10000010
0000 0006	0x10000014
0000 0007	0x10000018
0000 0008	0x1000001c

Time: 4000 ns

Write-Through Cache

- On replacement, the block in cache need not be written back…they are the same



Proc.

Cache Block
0000 0001
0000 0002
5678 FE00
0000 0004
0000 0005
0000 0006
0000 0007
0000 0008

MM Block	
0000 0001	0x10000000
0000 0002	0x10000004
5678 FE00	0x10000008
0000 0004	0x1000000c
0000 0005	0x10000010
0000 0006	0x10000014
0000 0007	0x10000018
0000 0008	0x1000001c

Time: 4000 ns

Analysis

- Write-back
  - Best results when repeated accesses to a block outweigh cost of writeback (i.e. writes in loops)
- Write-through
  - Best results when few, isolated accesses (no need for writeback)

Replacement Policies

- On a read- or write-miss, a new block must be brought in
- This requires evicting a current block residing in the cache
- Replacement policies
  - FIFO: First-in first-out (oldest block replaced)
  - LRU: Least recently used (usually best but hard to implement)
  - Random: Actually performs surprisingly well

Blocking vs. Non-blocking Cache

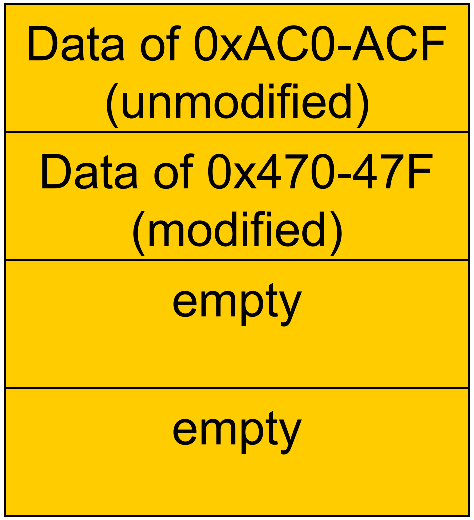
- A cache has an interface between itself and the processor and itself and main memory. Can the cache satisfy requests from the processor at the same time it is bringing in a block from MM?
- **Blocking Cache**: A blocking cache cannot satisfy other requests from the processor while it is fetching or writing blocks from/to main memory.
- **Non-blocking Cache**: A non-blocking cache can satisfy processor requests at the same time it accesses main memory.
  - **Example: In an out-of-order, superscalar processor an instruction might cause a miss and stall but the processor will continue executing the following instructions.**

Mapping Schemes

CACHE IMPLEMENTATION

Cache Implementation

- Assume a cache of 4 blocks of 4 words (16-bytes) each
- What other bookkeeping and identification info is needed?
  - Has the block been modified
  - Is the block empty or full
  - Address range of the data



Data of 0xAC0-ACF (unmodified)

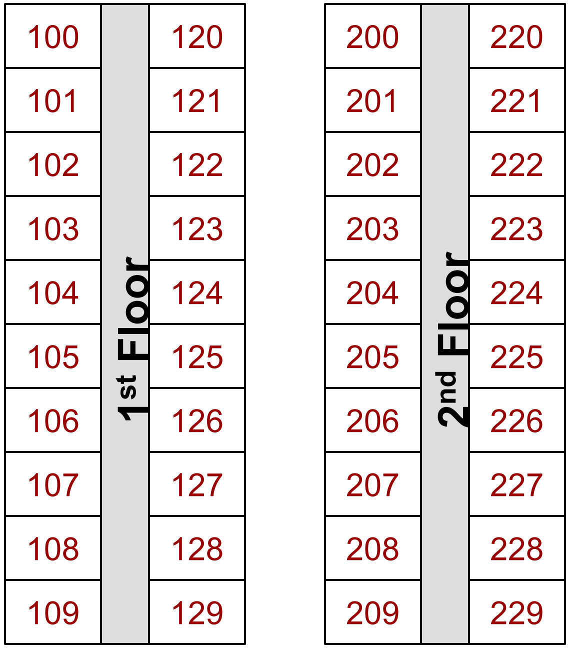
Data of 0x470-47F (modified)

empty

empty

Identifying Blocks via Address Range

- Possible methods
  - Store start and end address (requires multiple comparisons)
  - Ensure block ranges sit on binary boundaries (upper address bits identify the block with a single value)
    - Analogy: Hotel room layout/addressing



	1st Floor	
100		120
101		121
102		122
103		123
104		124
105		125
106		126
107		127
108		128
109		129

	2nd Floor	
200		220
201		221
202		222
203		223
204		224
205		225
206		226
207		227
208		228
209		229

Analogy: Hotel Rooms

1st Digit = Floor
2nd Digit = Aisle
3rd Digit = Room w/in aisle

To refer to the range of rooms on the second floor, left aisle we would just say rooms **20x**

4 word (16-byte) blocks:

Addr. Range	Binary		
000-00f	0000	0000	0000 -1111
010-01f	0000	0001	0000 -1111

8 word (32-byte) blocks:

Addr. Range	Binary		
000-01f	0000	000	00000 -11111
020-03f	0000	001	00000 -11111

Cache Implementation

- Assume 12-bit addresses and 4-word blocks
- Block addresses will range from xx0-xxF
  - Address can be broken down as follows
  - A[11:4] = identifies block range (i.e. xx0-xxF)
  - A[3:2] = selects the 1 of 4 words within the block
  - A[1:0] = unused (always access 32-bit word)

A[11:4]	A[3:2]	A[1:0]
Tag	Word	00

Addr. = 0x124

Word 1 w/in block 120-12F

0001 0010	01	00

Addr. = 0xACC

Word 3 w/in block AC0-ACF

1010 1100	11	00

Implementation Terminology

- What bookkeeping values must be stored with the cache in addition to the block data?
- **Tag** – Portion of the block's address range used to identify the MM block residing in the cache from other MM blocks.
- **Valid bit** – Indicates the block is occupied with valid data (i.e. not empty or invalid)
- **Dirty bit** – Indicates the cache and MM copies are "inconsistent" (i.e. a write has been done to the cached copy but not the main memory copy)
  - Used for write-back caches

Cache Implementation

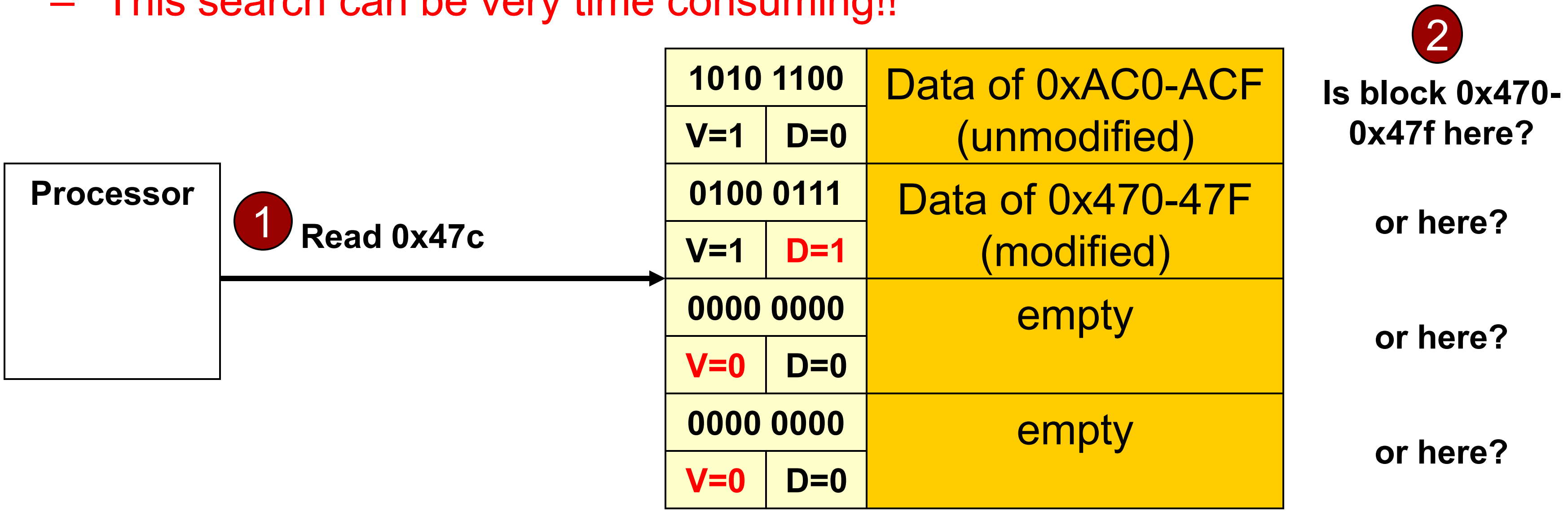
- To identify which MM block resides in each cache block, the tags need to be stored along with the Dirty and Valid bits

Tag

1010 1100		Data of 0xAC0-ACF (unmodified)
V=1	D=0	
0100 0111		Data of 0x470-47F (modified)
V=1	D=1	
0000 0000		empty
V=0	D=0	
0000 0000		empty
V=0	D=0	

Content-Addressable Memory

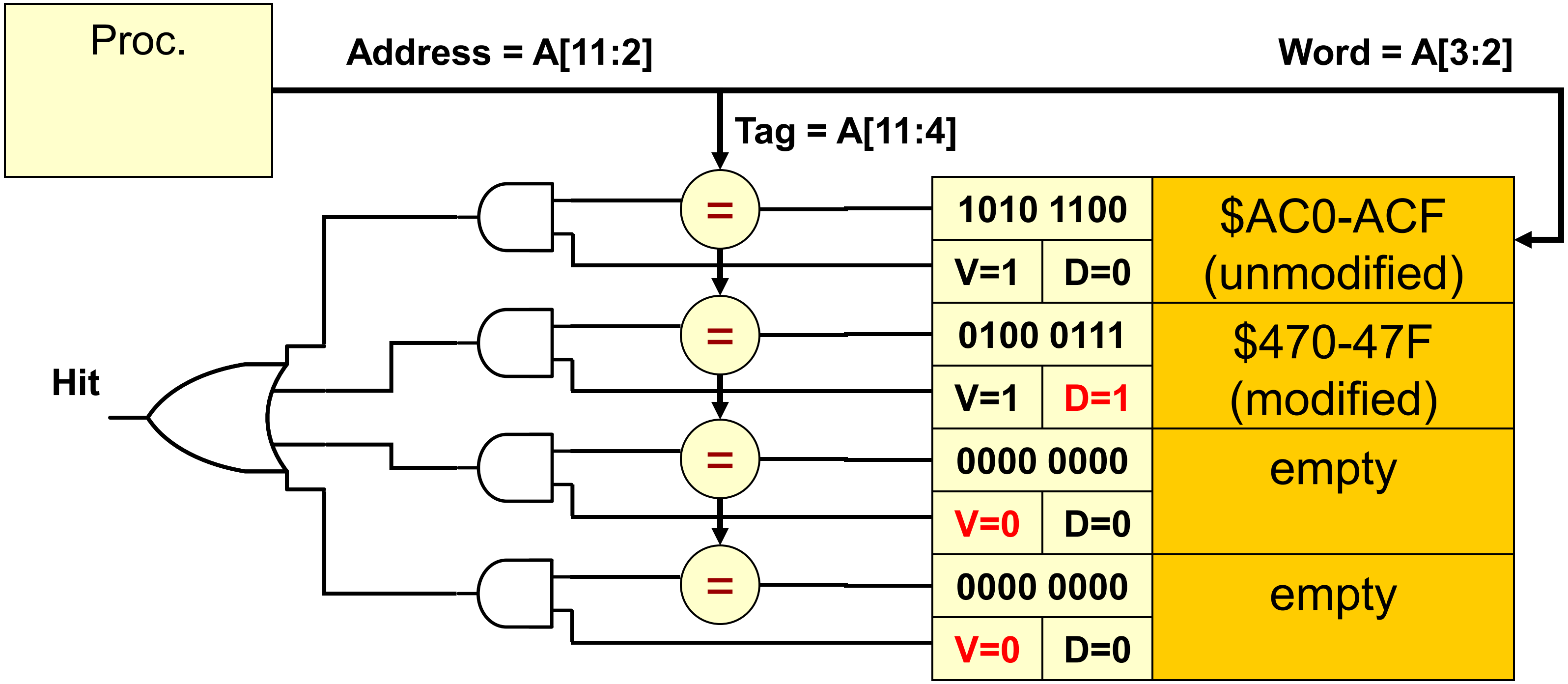
- Cache memory is one form of what is known as “content-addressable” memory
  - This means data can be in any location in memory and does not have one particular address
  - Additional information is saved with the data and is used to “address”/find the desired data (this is the “tag” in this case) via a search on each access
  - This search can be very time consuming!!



1010 1100		Data of 0xAC0-ACF (unmodified)
V=1	D=0	
0100 0111		Data of 0x470-47F (modified)
V=1	D=1	
0000 0000		empty
V=0	D=0	
0000 0000		empty
V=0	D=0	

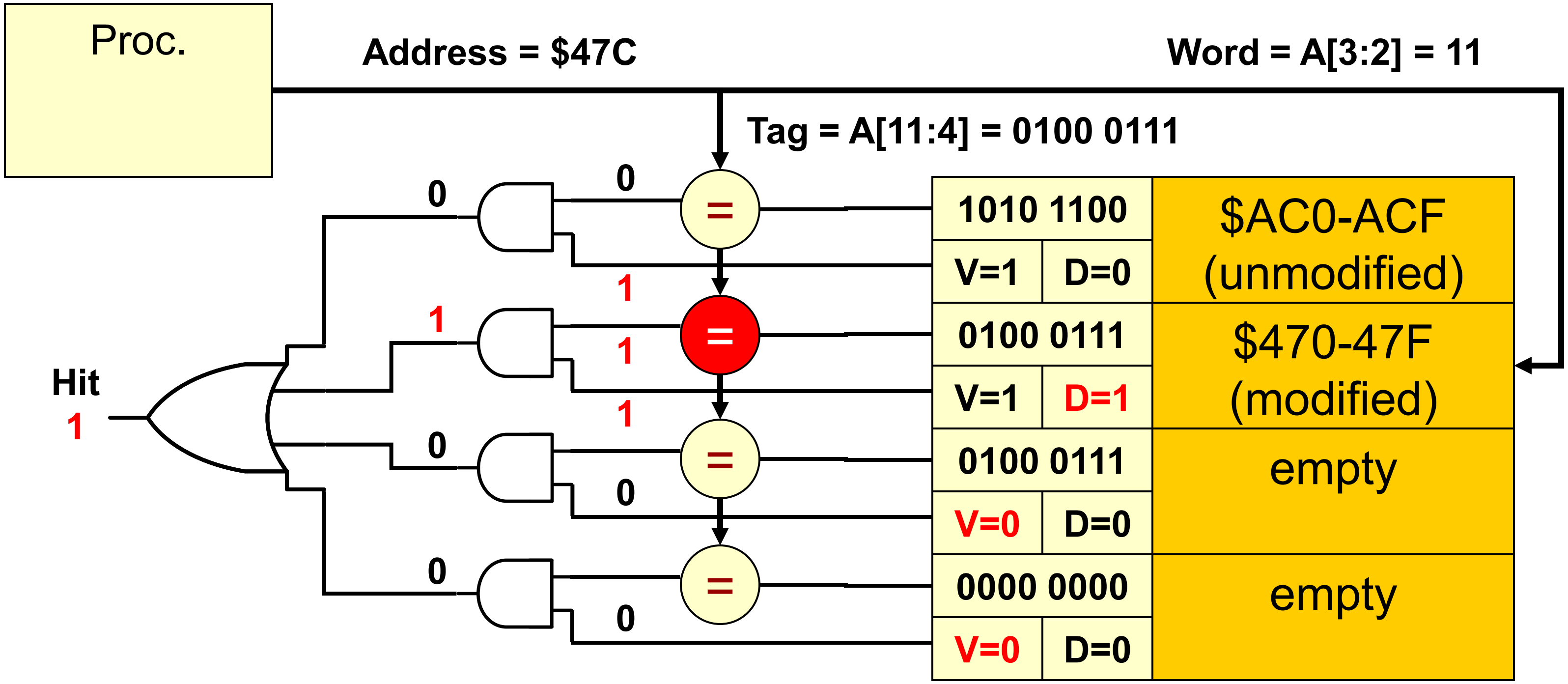
Tag Comparison

- When caches have many blocks (> 16 or 32) it can be expensive (hardware-wise) to check all tags



Tag Comparison Example

- Tag portion of desired address is check against all the tags and qualified with the valid bits to determine a hit



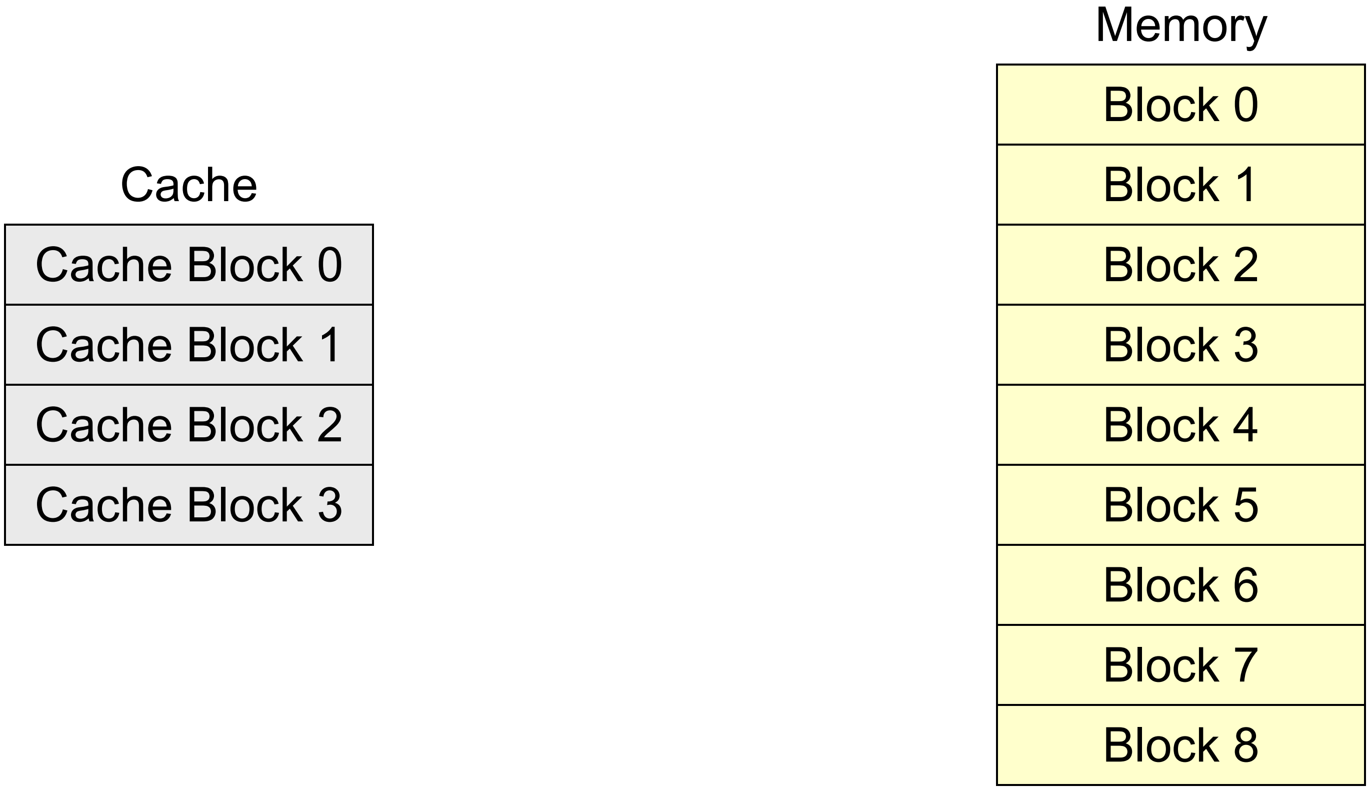
1010 1100	V=1	D=0	$AC0-ACF (unmodified)
0100 0111	V=1	D=1	$470-47F (modified)
0100 0111	V=0	D=0	empty
0000 0000	V=0	D=0	empty

Mapping Techniques

- Determines where blocks can be placed in the cache
- By reducing number of possible MM blocks that map to a cache block, hit logic (searches) can be done faster
- 3 Primary Methods
  - Direct Mapping
  - Fully Associative Mapping
  - Set-Associative Mapping

Mapping techniques

- Example cache w/ only 4 blocks
- MM has many blocks



Cache

Cache Block 0

Cache Block 1

Cache Block 2

Cache Block 3

Memory

Block 0

Block 1

Block 2

Block 3

Block 4

Block 5

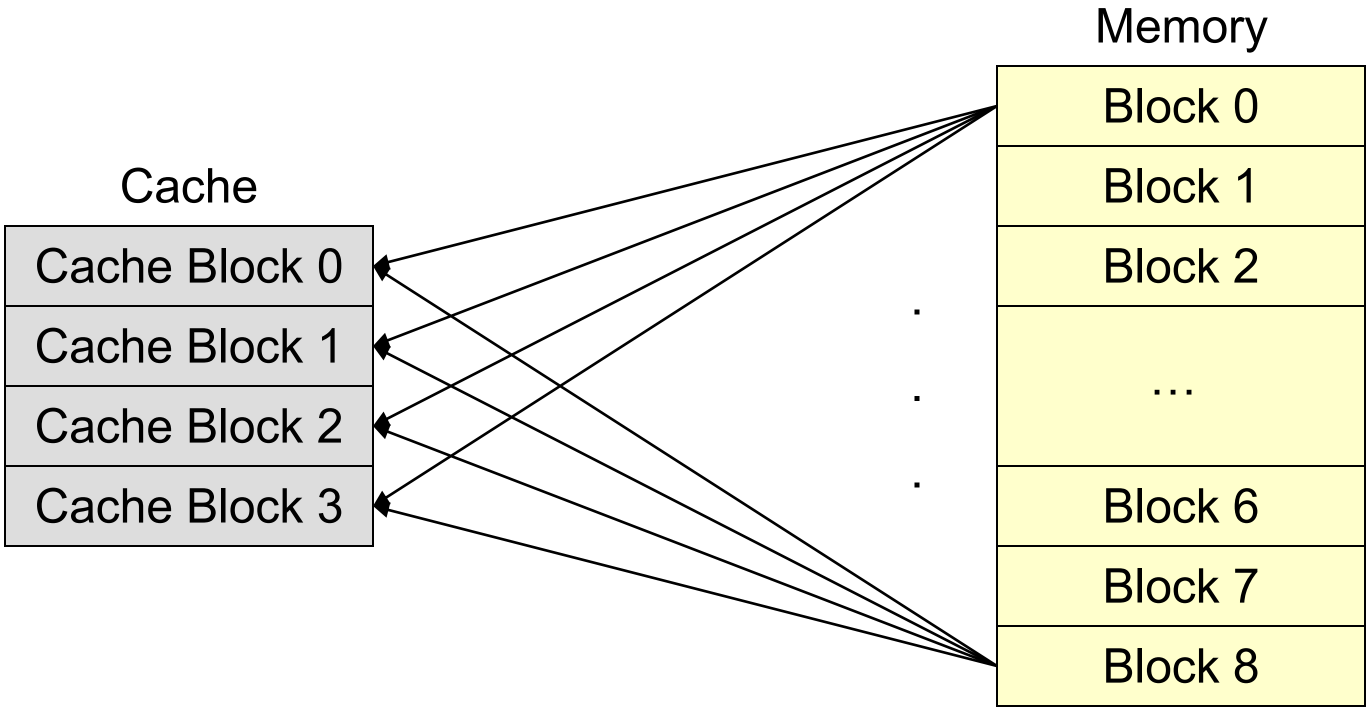
Block 6

Block 7

Block 8

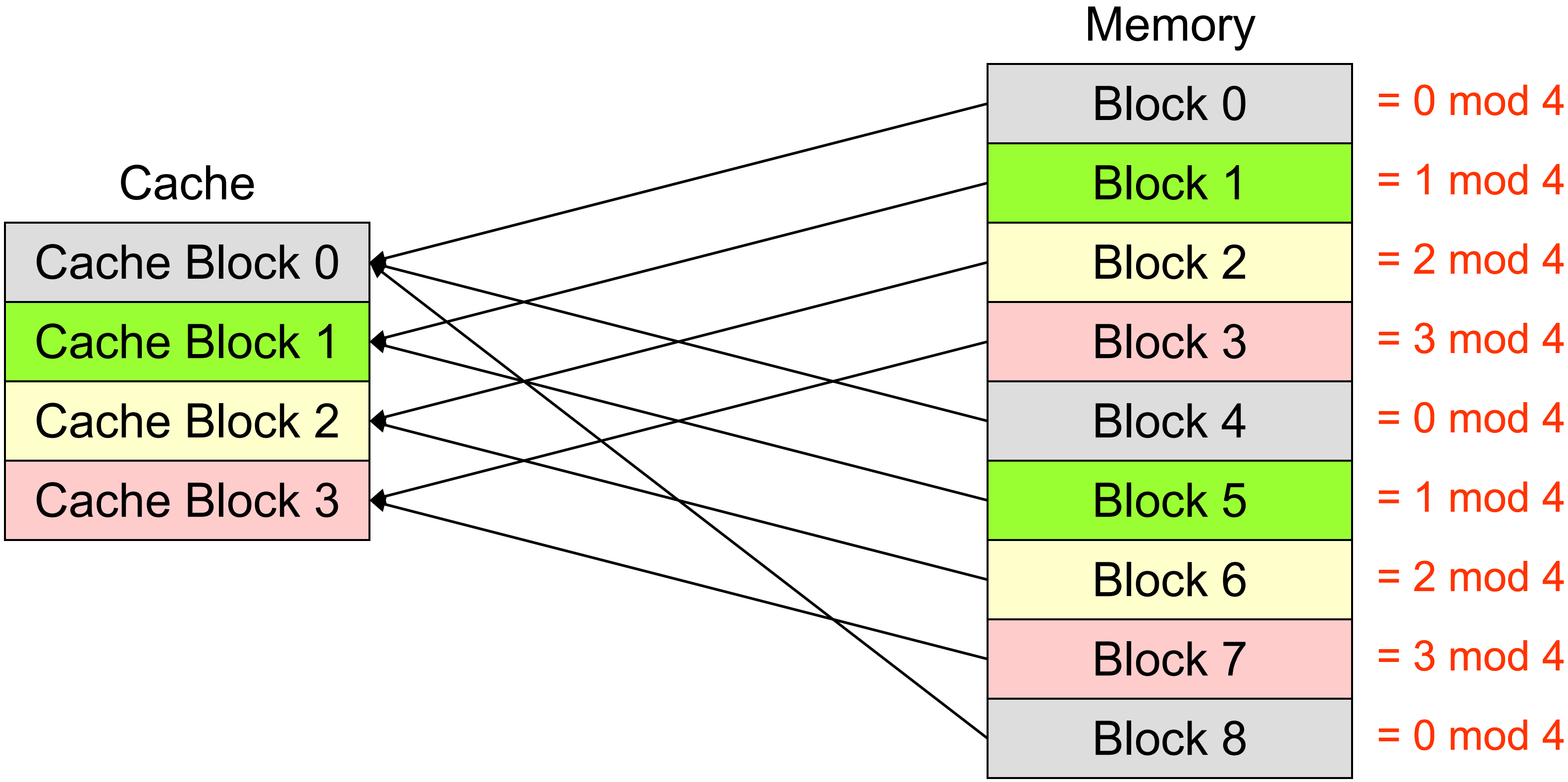
Fully Associative Mapping

- Any block from memory can be put in any cache block (i.e. no restriction)



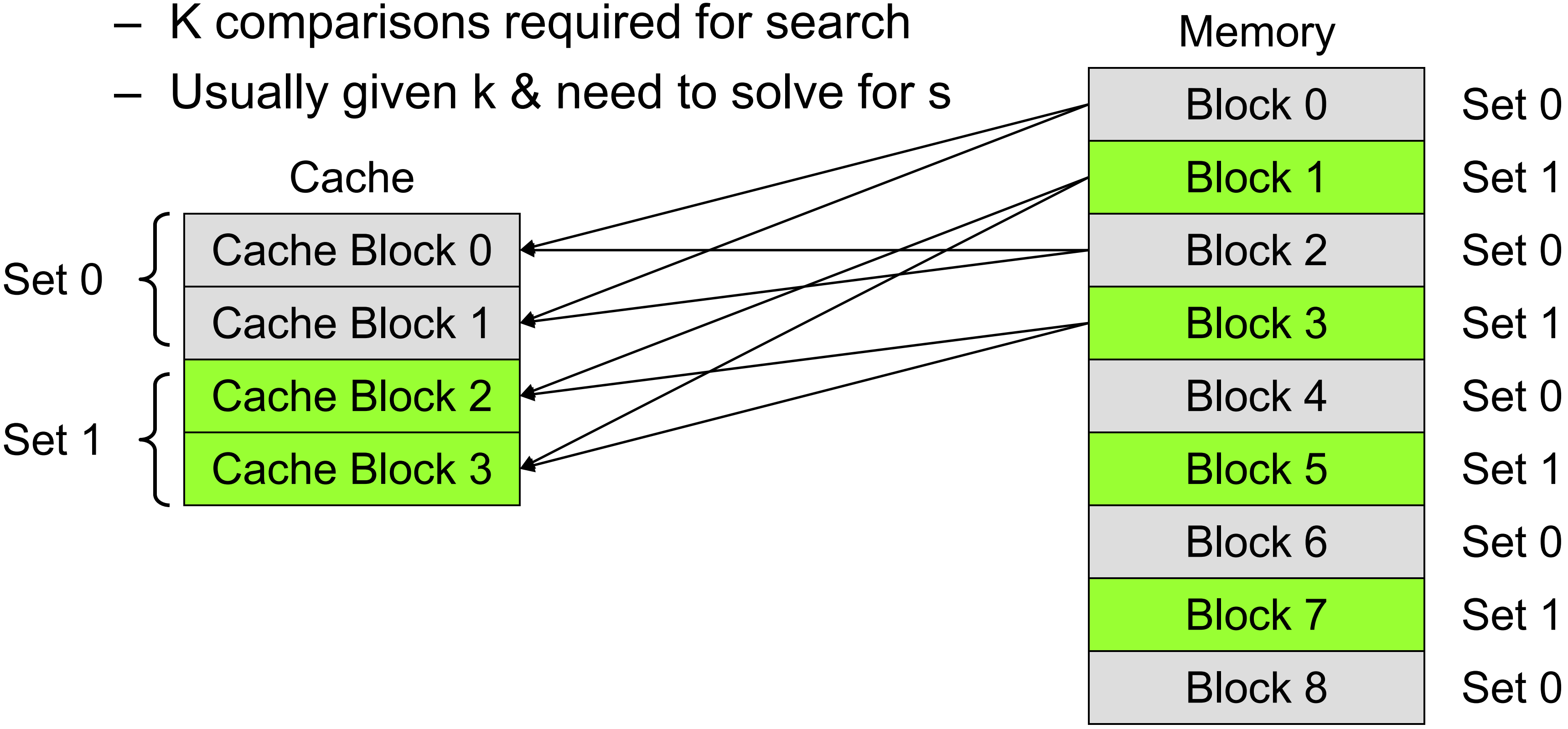
Direct Mapping

- Each block from memory can only be put in one location
- Given n cache blocks,
  MM block i maps to cache block i mod n



K-way Set-Associative Mapping

- Given, S sets, block i of MM maps to set i mod s
- Within the set, block can be put anywhere
- Let k = number of cache blocks in a set = n/s
  - K comparisons required for search
  - Usually given k & need to solve for s



Mapping Implementation

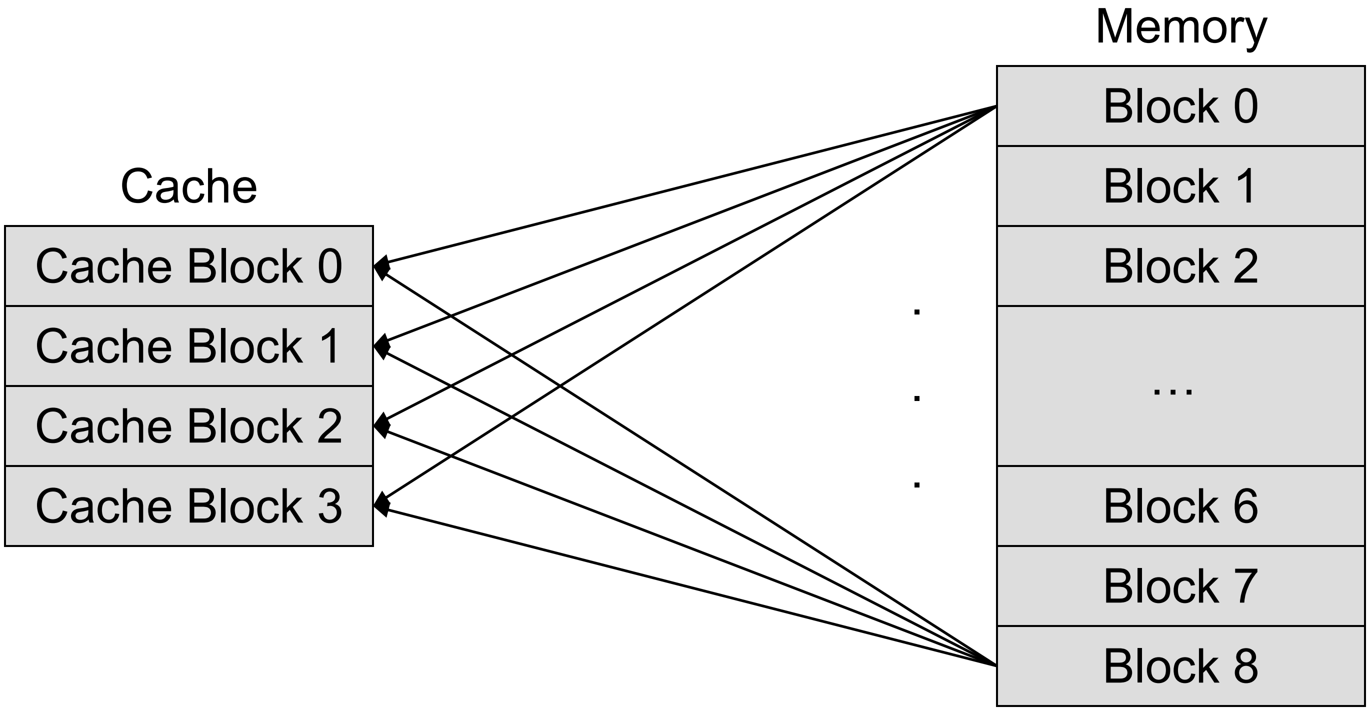
- Q: How to implement mapping schemes
  - Direct Mapping
  - Fully Associative Mapping
  - Set-Associative Mapping
- A: By using the addresses themselves

Assumptions:

- Only access words
- 12-bit MM addresses (4096 Bytes)
- 4-word blocks

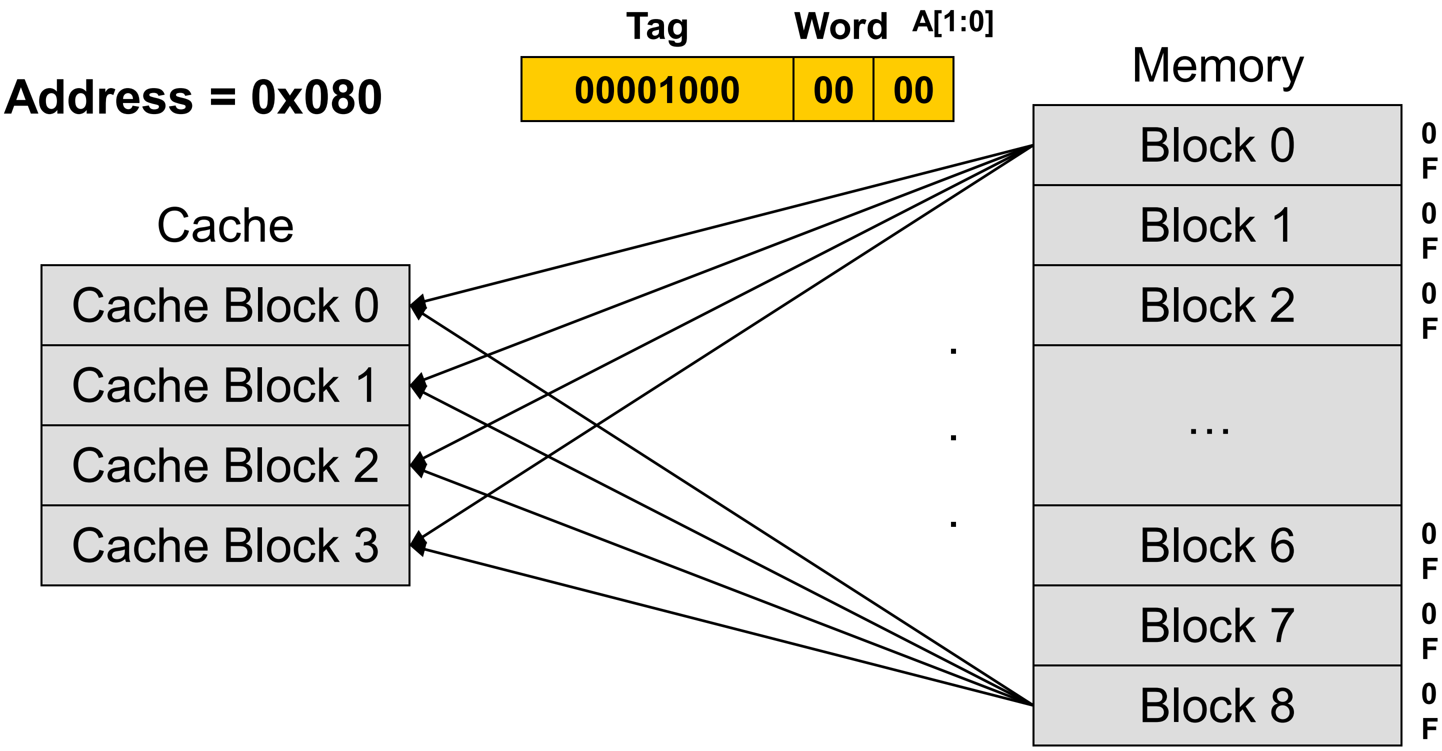
Fully Associative Mapping

- Any block from memory can be put in any cache block (i.e. no mapping)



Fully Associative Implementation

- 12-bit address:
  - 4 = 2^2 words per block => 2 LSB's above A[1:0] used to determine the desired word w/in the block
  - Tag = Block # = Upper bits used to identify the block in the cache

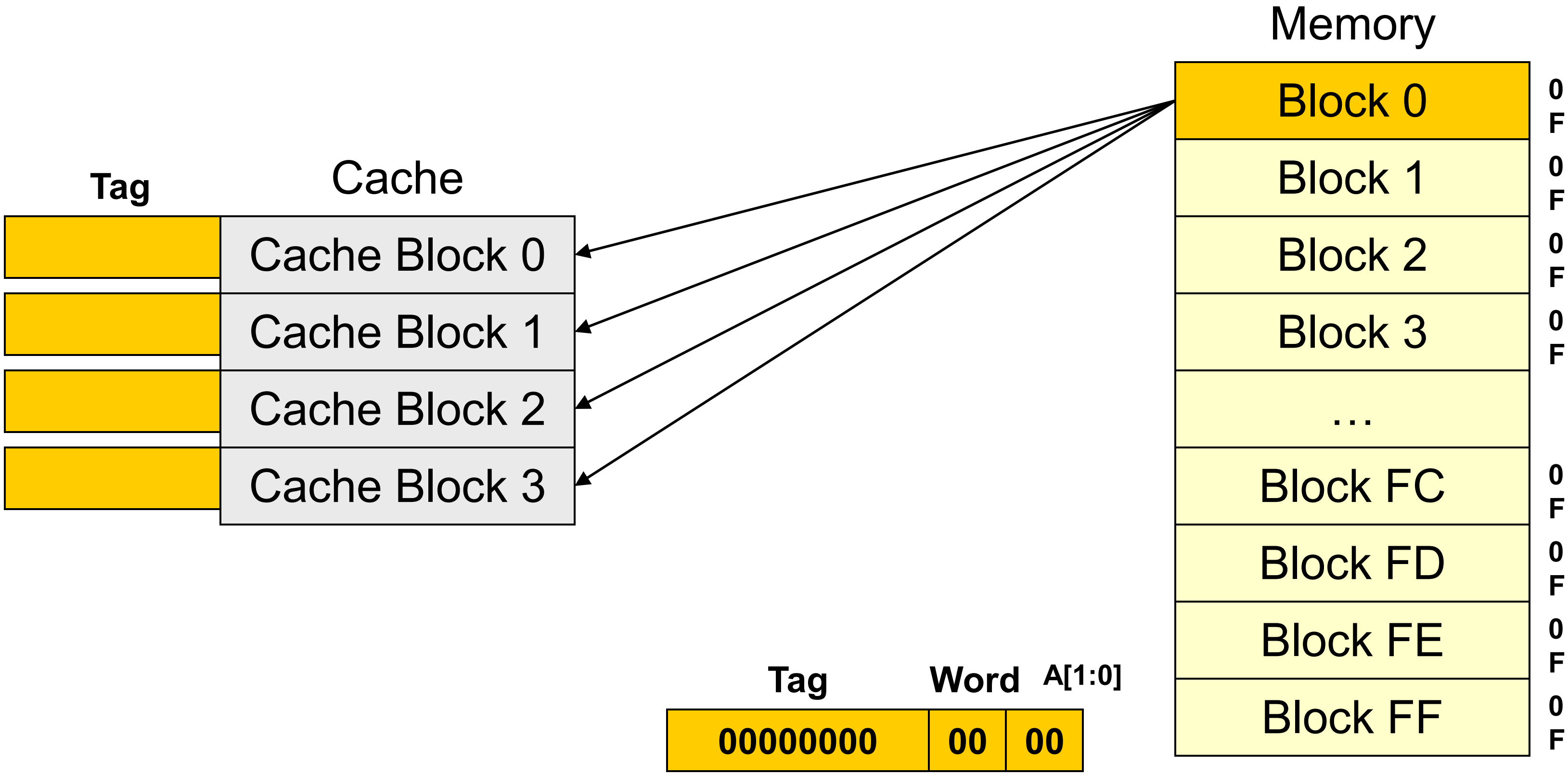


Fully Associative Address Scheme

- A[1:0] unused (word access only)
- Word bits = $\log_2$B bits (B=Block Size)
- Tag = Remaining bits

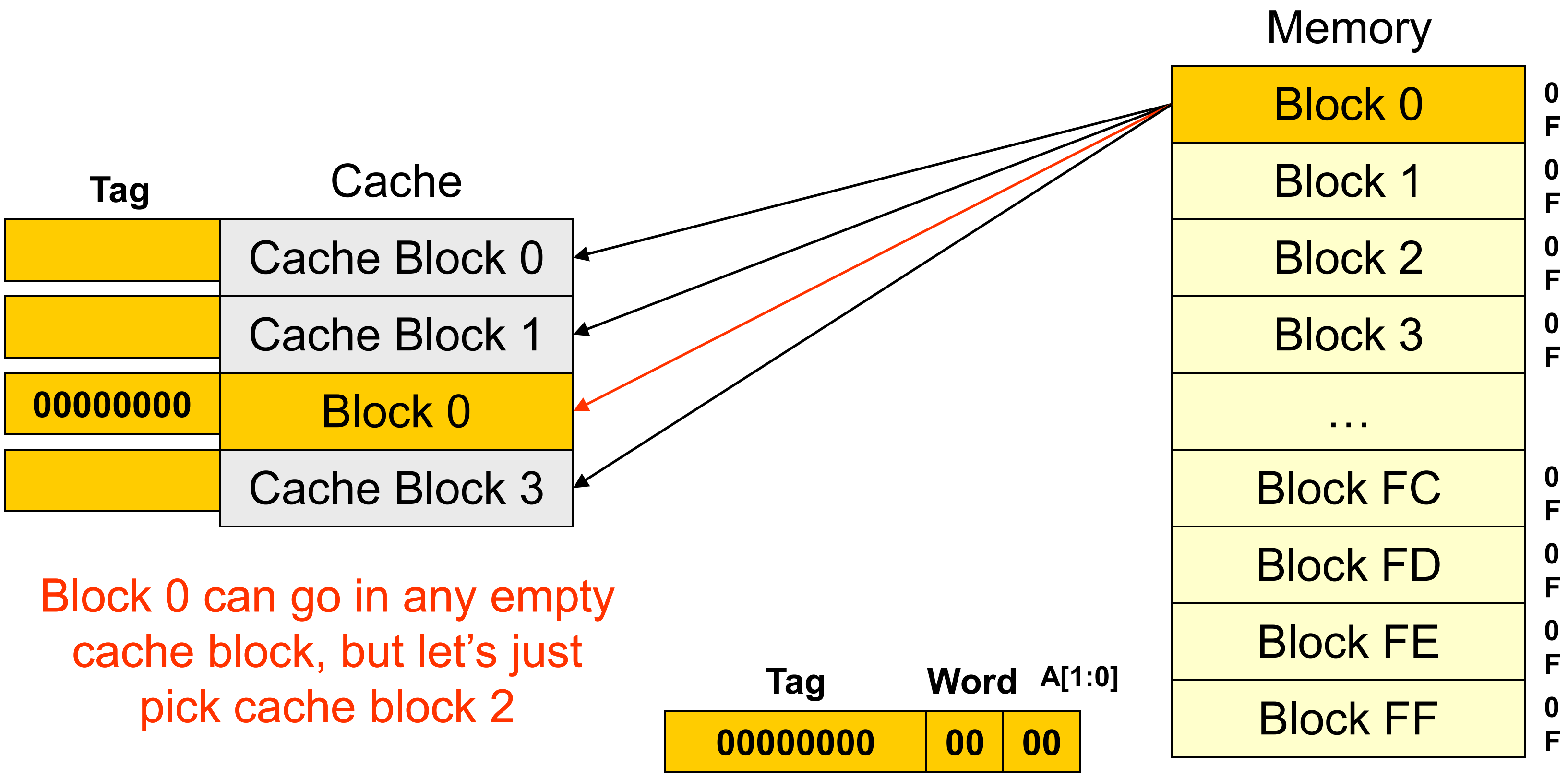
Fully Associative Mapping

- Any block from memory can be put in any cache block (i.e. no mapping scheme)
- Completely flexible



Fully Associative Mapping

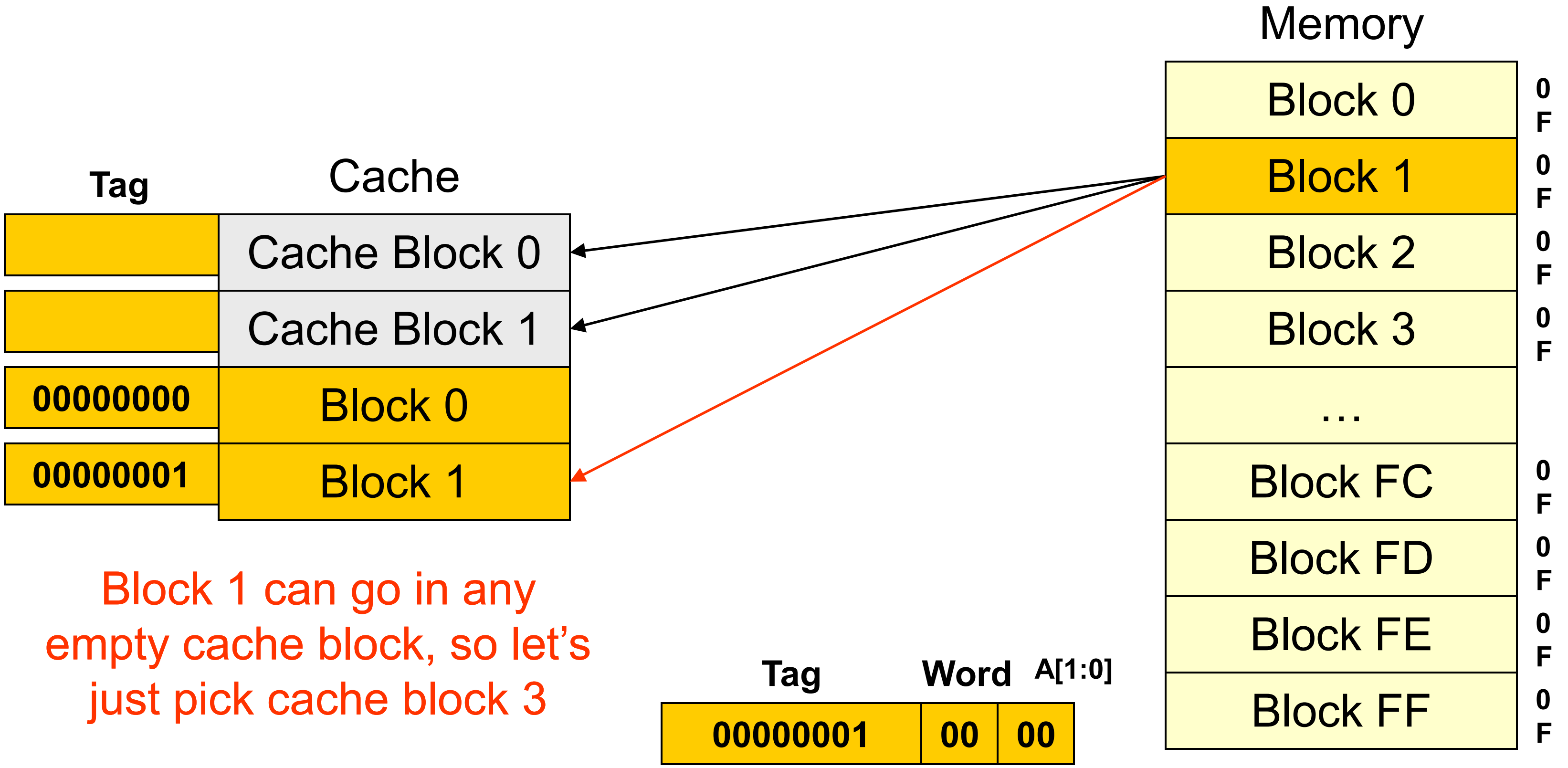
- Any block from memory can be put in any cache block (i.e. no mapping)



Block 0 can go in any empty cache block, but let's just pick cache block 2

Fully Associative Mapping

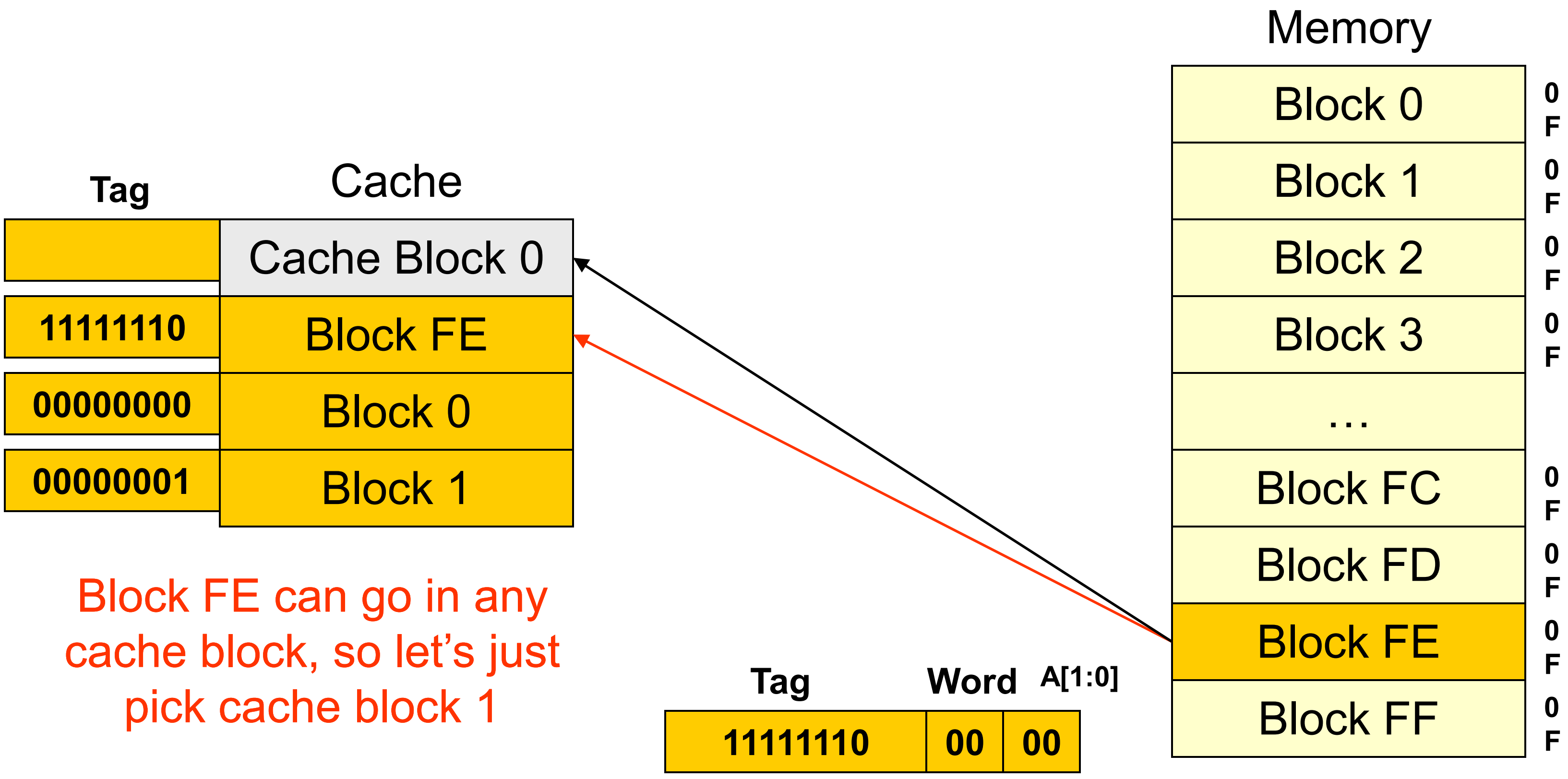
- Any block from memory can be put in any cache block (i.e. no mapping)



Block 1 can go in any empty cache block, so let's just pick cache block 3

Fully Associative Mapping

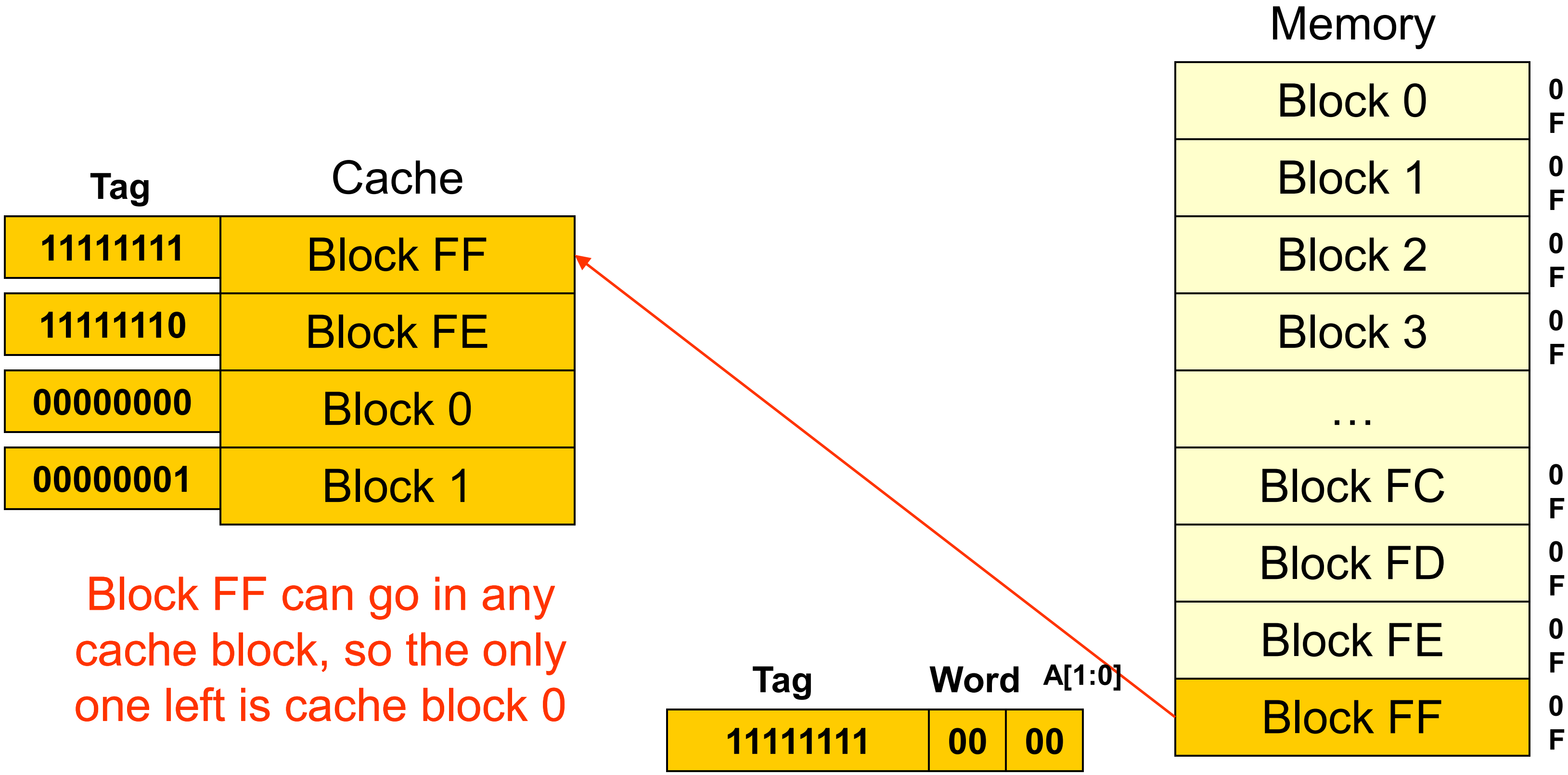
- Any block from memory can be put in any cache block (i.e. no mapping)



Block FE can go in any cache block, so let's just pick cache block 1

Fully Associative Mapping

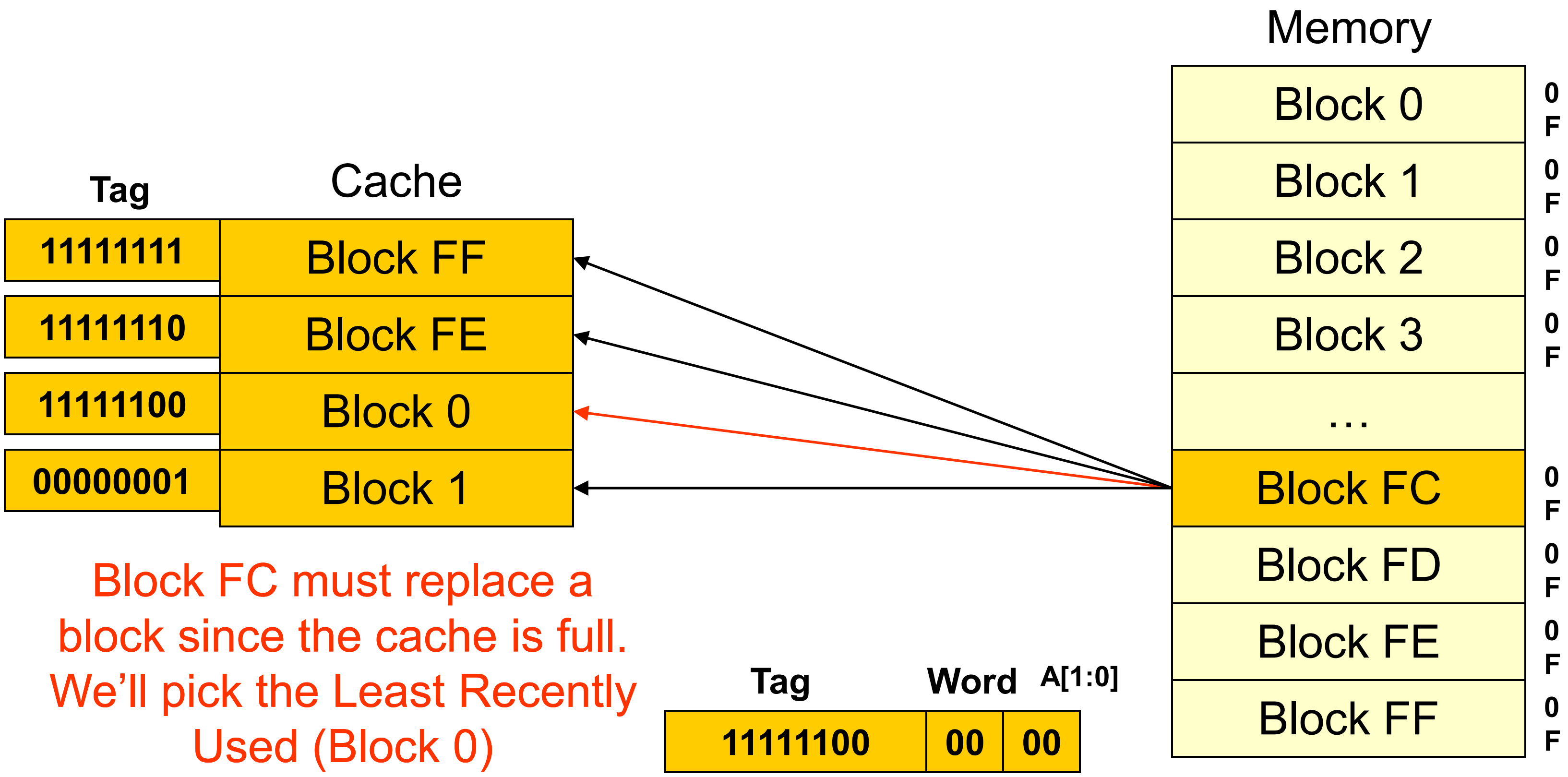
- Any block from memory can be put in any cache block (i.e. no mapping)



Block FF can go in any cache block, so the only one left is cache block 0

Fully Associative Mapping

- Any block from memory can be put in any cache block (i.e. no mapping)



Memory

Block 0	0 F
Block 1	0 F
Block 2	0 F
Block 3	0 F
…	
Block FC	0 F
Block FD	0 F
Block FE	0 F
Block FF	0 F

Cache

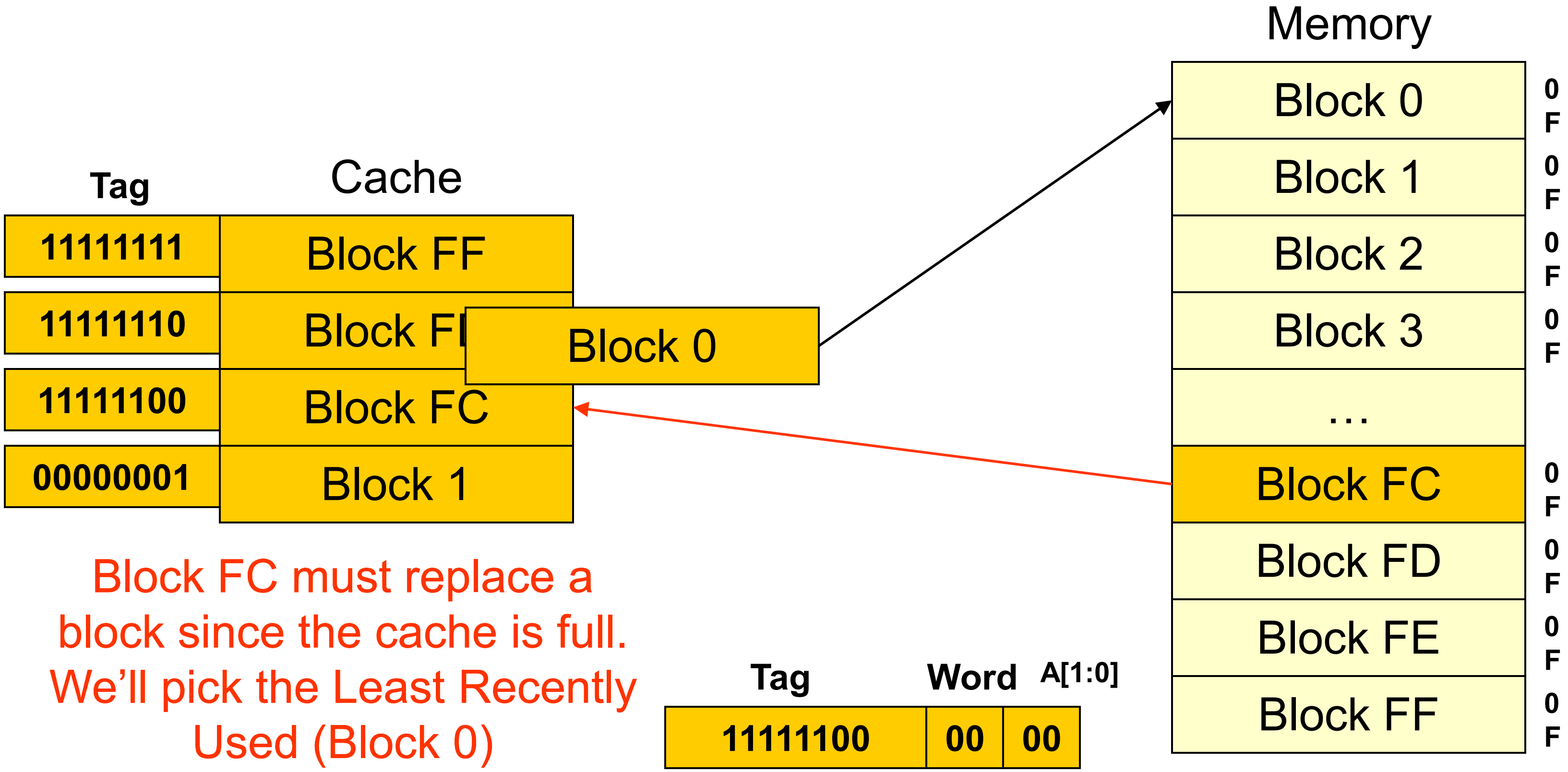
Tag	
11111111	Block FF
11111110	Block FE
11111100	Block 0
00000001	Block 1

Block FC must replace a block since the cache is full. We'll pick the Least Recently Used (Block 0)

Tag	Word	A[1:0]
11111100	00	00

Fully Associative Mapping

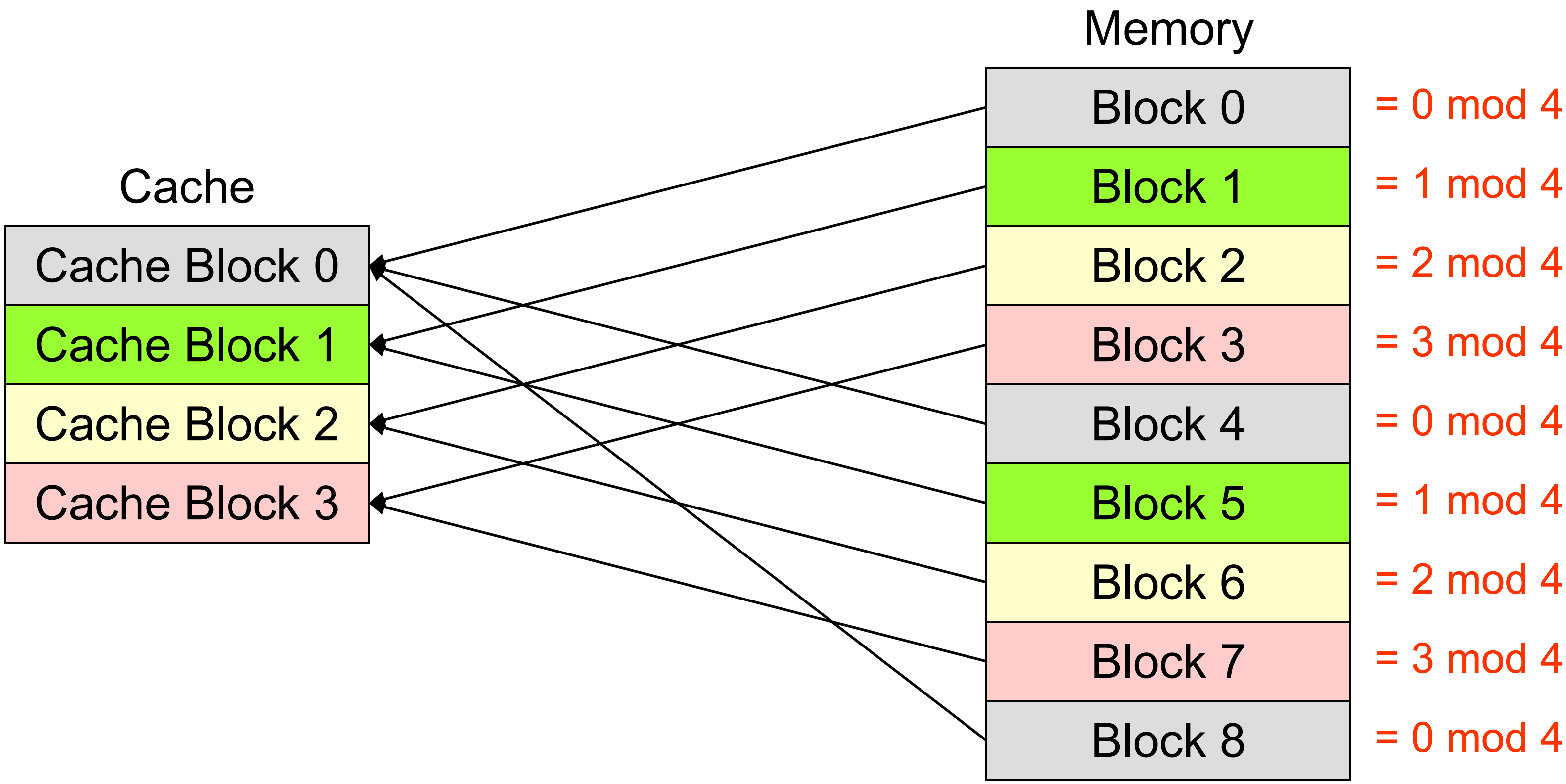
- Any block from memory can be put in any cache block (i.e. no mapping)



Block FC must replace a block since the cache is full. We'll pick the Least Recently Used (Block 0)

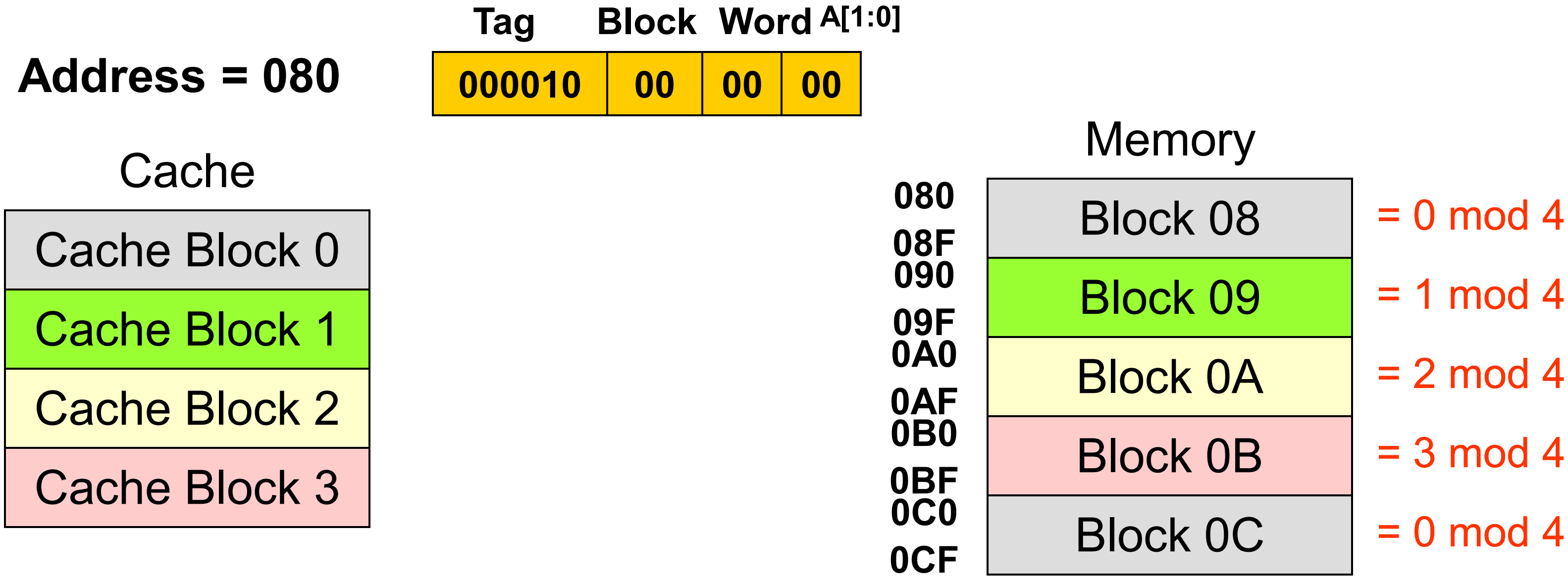
Direct Mapping

- Each block from memory can only be put in one location
- MM block i maps to cache block i mod n



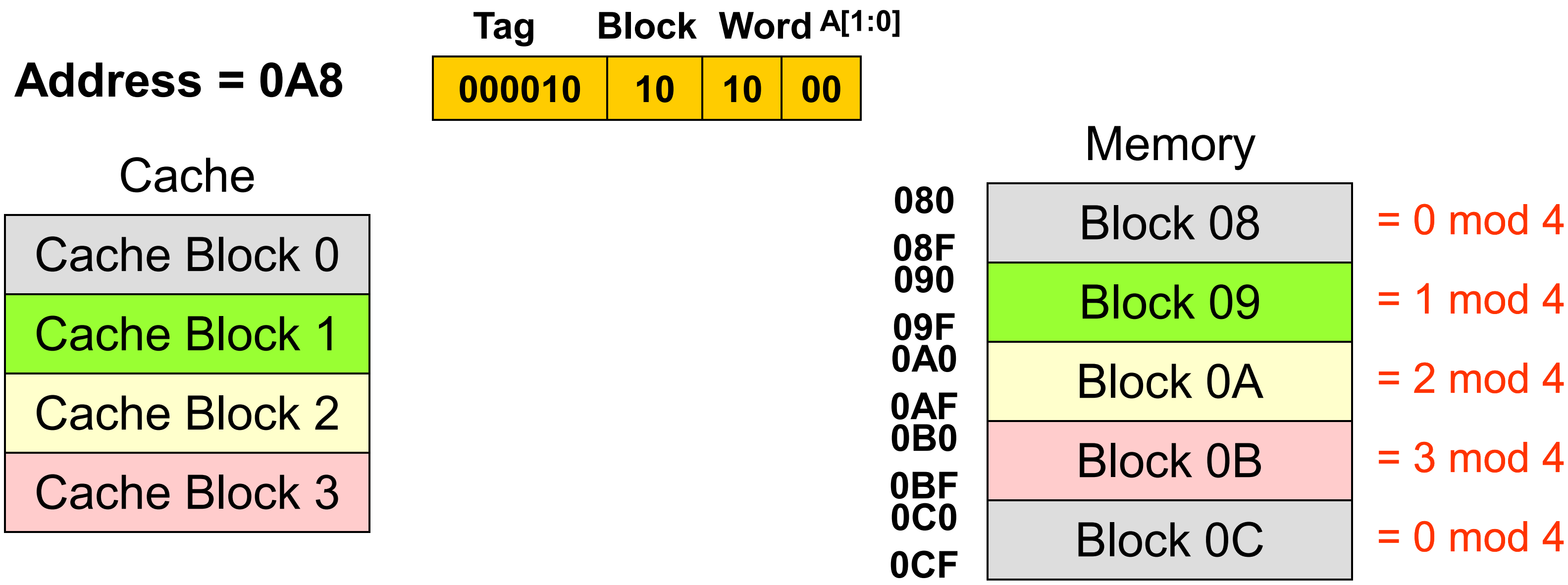
Direct Mapping Implementation

- 12-bit address:
  - 4 = 2^2 words per block => 2 bits of address used to determine the desired word w/in the block
  - 4 = 2^2 possible blocks => 2 bits to determine block…next 2 bits of address
  - Tag = Upper 6 bits used to identify the block in the cache (identifies between block (0,4,8,0xC,0x10, etc.)



Direct Mapping Implementation

- 12-bit address:
  - 4 = 2^2 words per block => 2 bits of address used to determine the desired word w/in the block
  - 4 = 2^2 possible blocks => 2 bits to determine block…next 2 bits of address
  - Tag = Upper 6 bits used to identify the block in the cache (identifies between block (0,4,8,0xC,0x10, etc.)



Address = 0A8

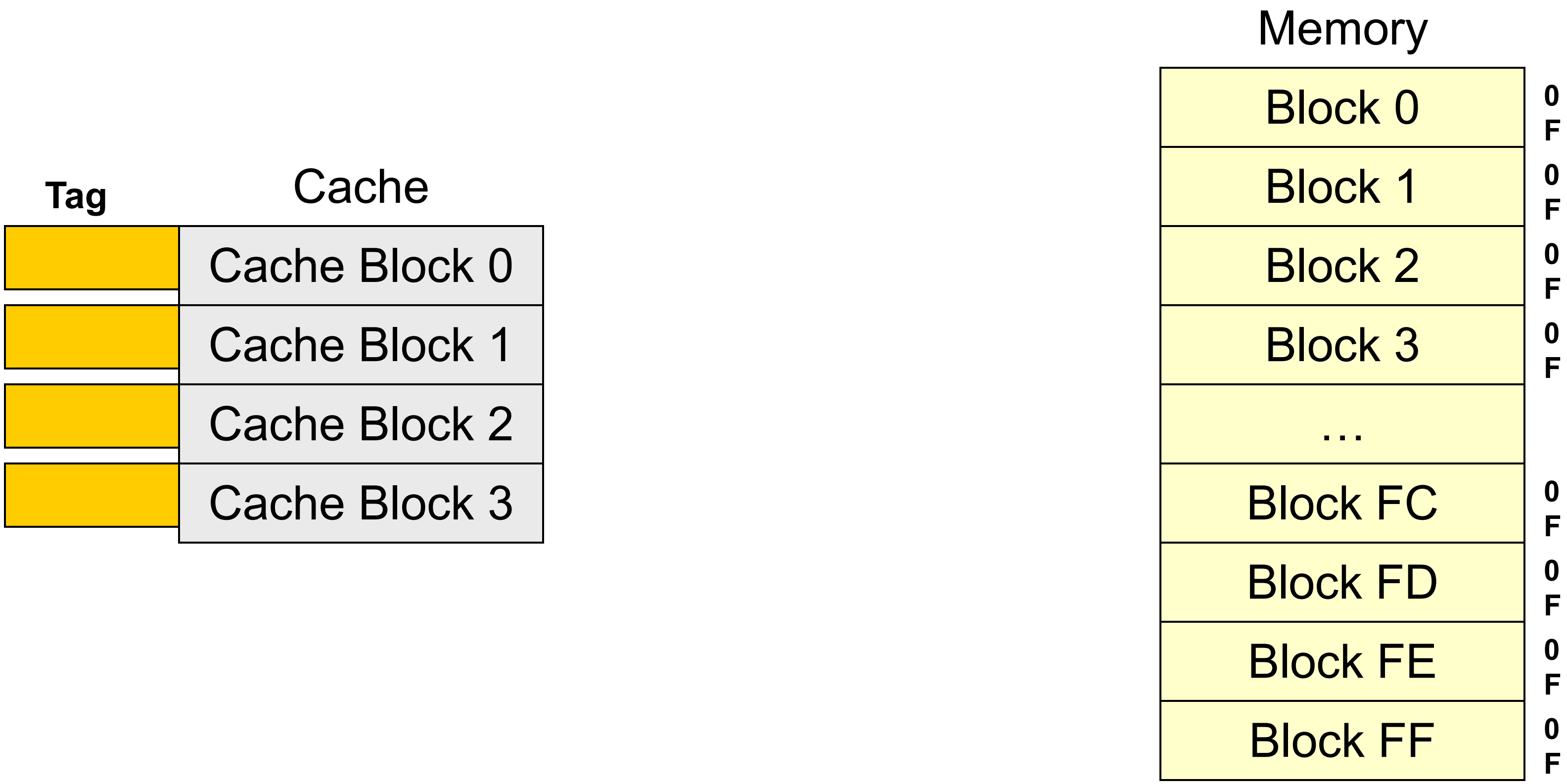
Tag	Block	Word	A[1:0]
000010	10	10	00

Direct Mapping Address Scheme

- A[1:0] unused
- Word bits = $\log_2 B$ bits (B=Block Size)
- Block bits = $\log_2 N$ bits (N=Cache Blocks)
- Tag = Remaining bits

Direct Mapping

- Each block from memory can only be put in one location
- MM block i maps to cache block i mod n



Memory

Block 0

Block 1

Block 2

Block 3

…

Block FC

Block FD

Block FE

Block FF

Tag

Cache

Cache Block 0

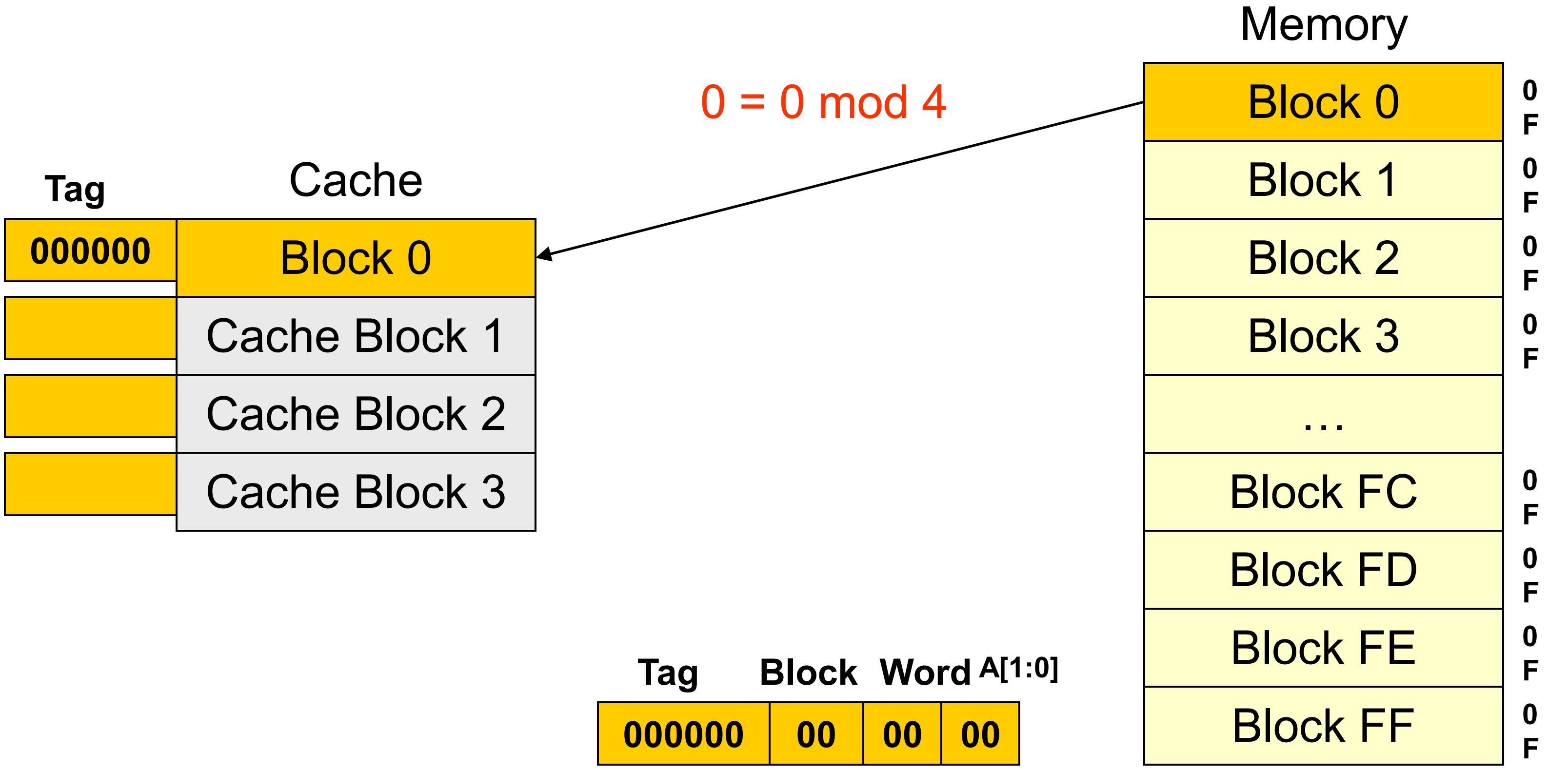
Cache Block 1

Cache Block 2

Cache Block 3

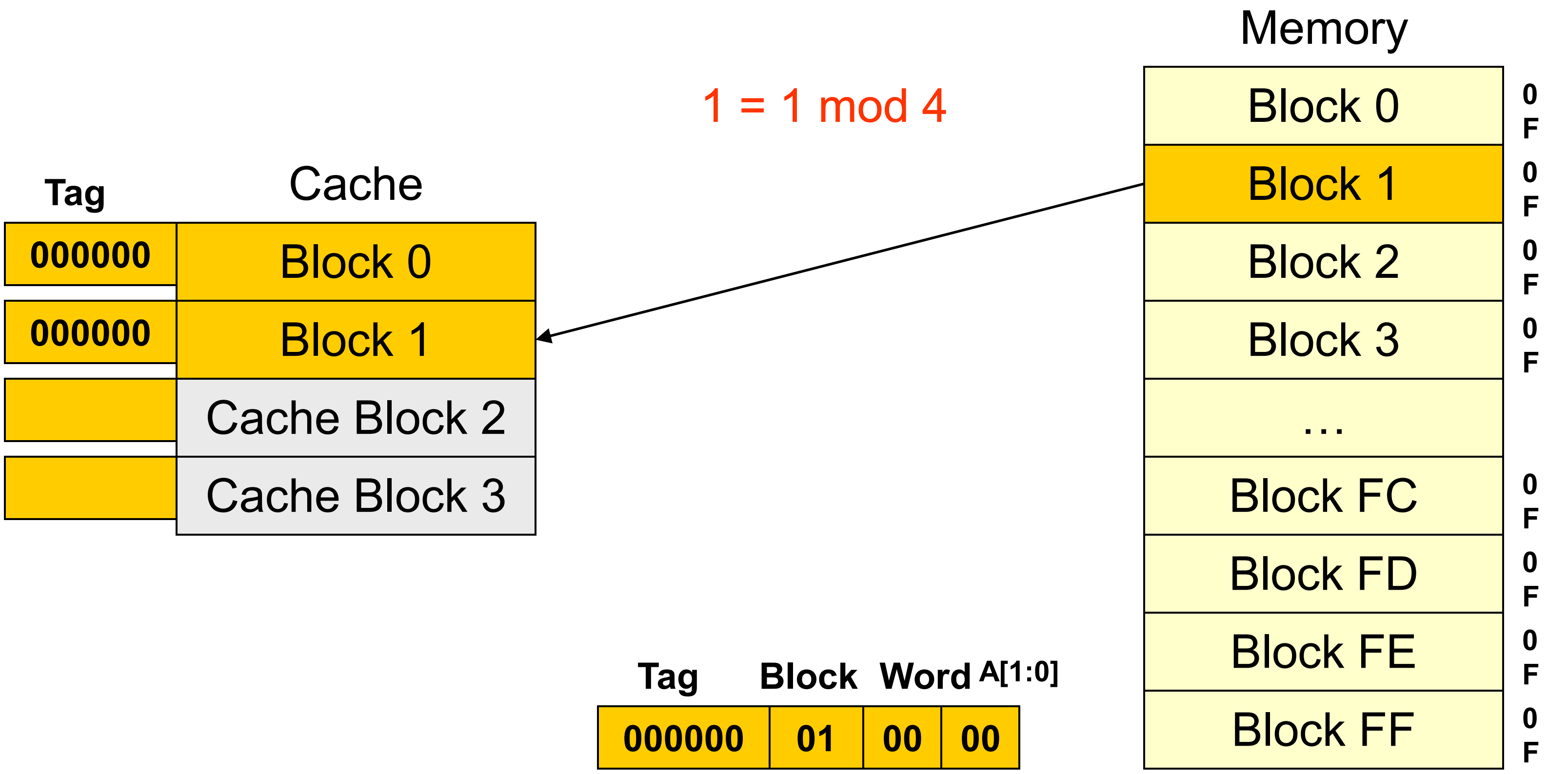
Direct Mapping

- Each block from memory can only be put in one location
- Block i mod n maps to cache block i



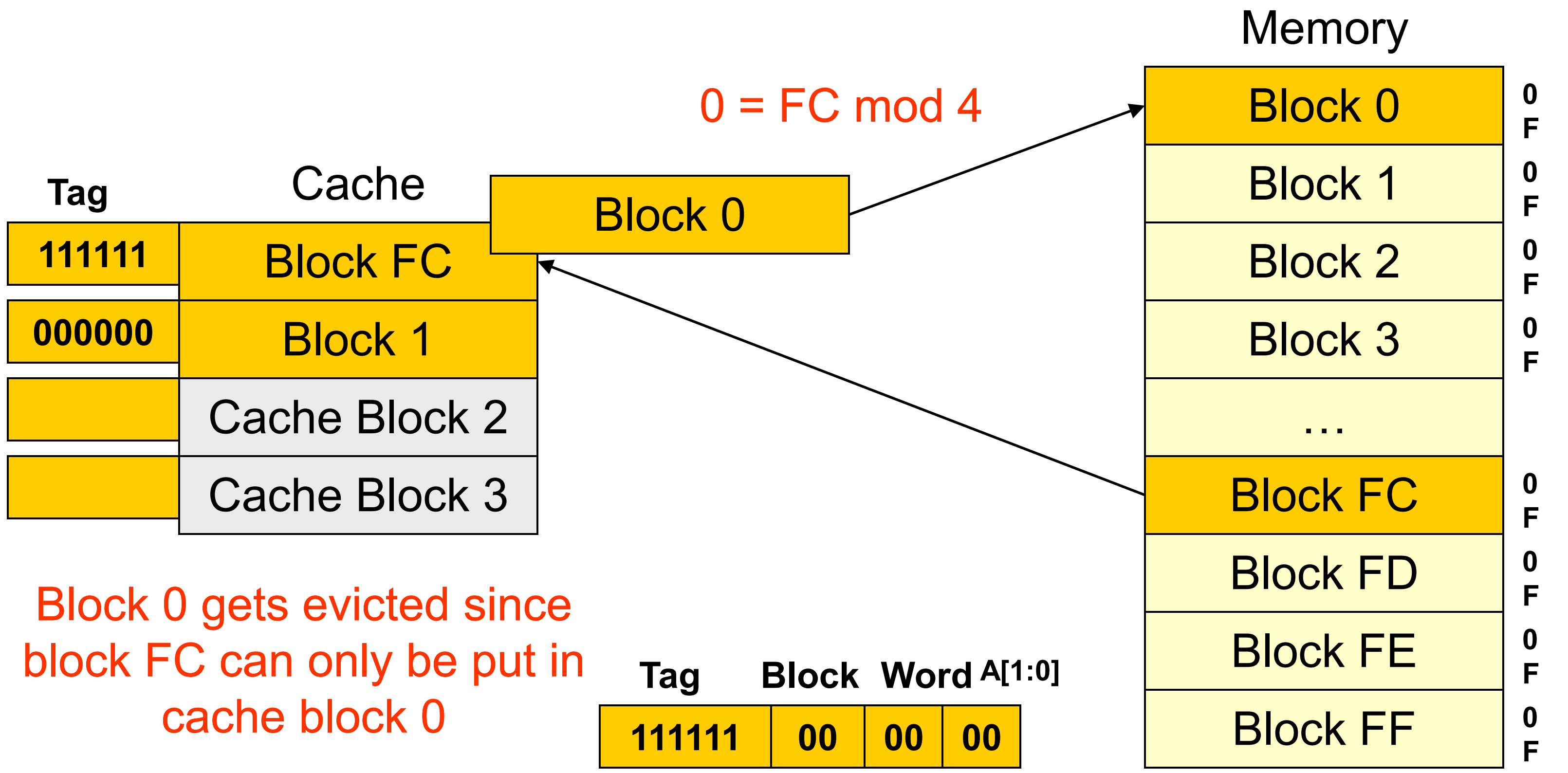
Direct Mapping

- Each block from memory can only be put in one location
- Block i mod n maps to cache block i



Direct Mapping

- Each block from memory can only be put in one location
- Block i mod n maps to cache block i



Memory

0 = FC mod 4

Tag

Cache

Tag	Cache
111111	Block FC
000000	Block 1
	Cache Block 2
	Cache Block 3

Block 0

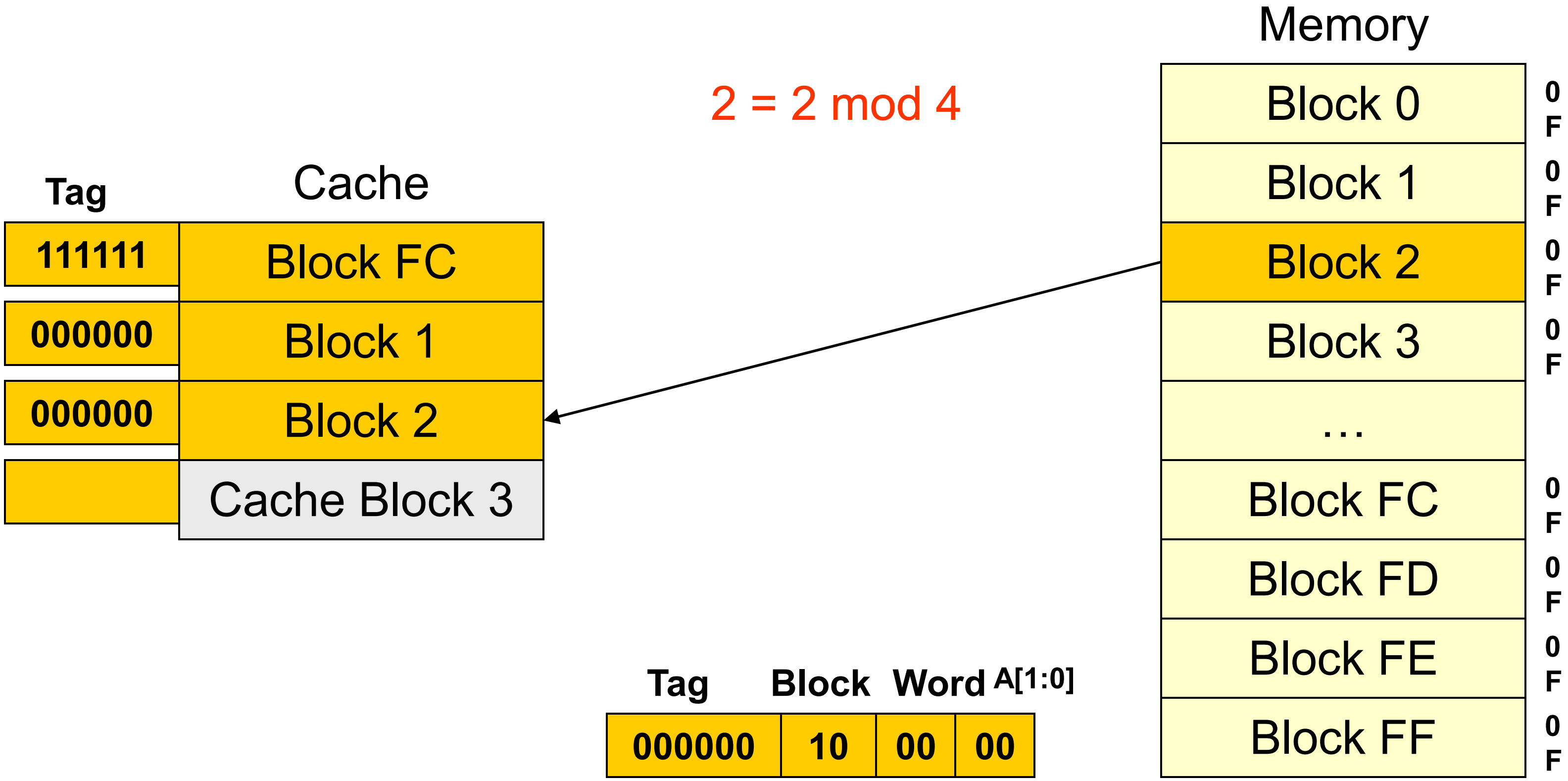
Memory	
Block 0	0 F
Block 1	0 F
Block 2	0 F
Block 3	0 F
…	
Block FC	0 F
Block FD	0 F
Block FE	0 F
Block FF	0 F

Block 0 gets evicted since block FC can only be put in cache block 0

Tag	Block	Word	A[1:0]
111111	00	00	00

Direct Mapping

- Each block from memory can only be put in one location
- Block i mod n maps to cache block i



2 = 2 mod 4

Cache

Tag	Cache
111111	Block FC
000000	Block 1
000000	Block 2
	Cache Block 3

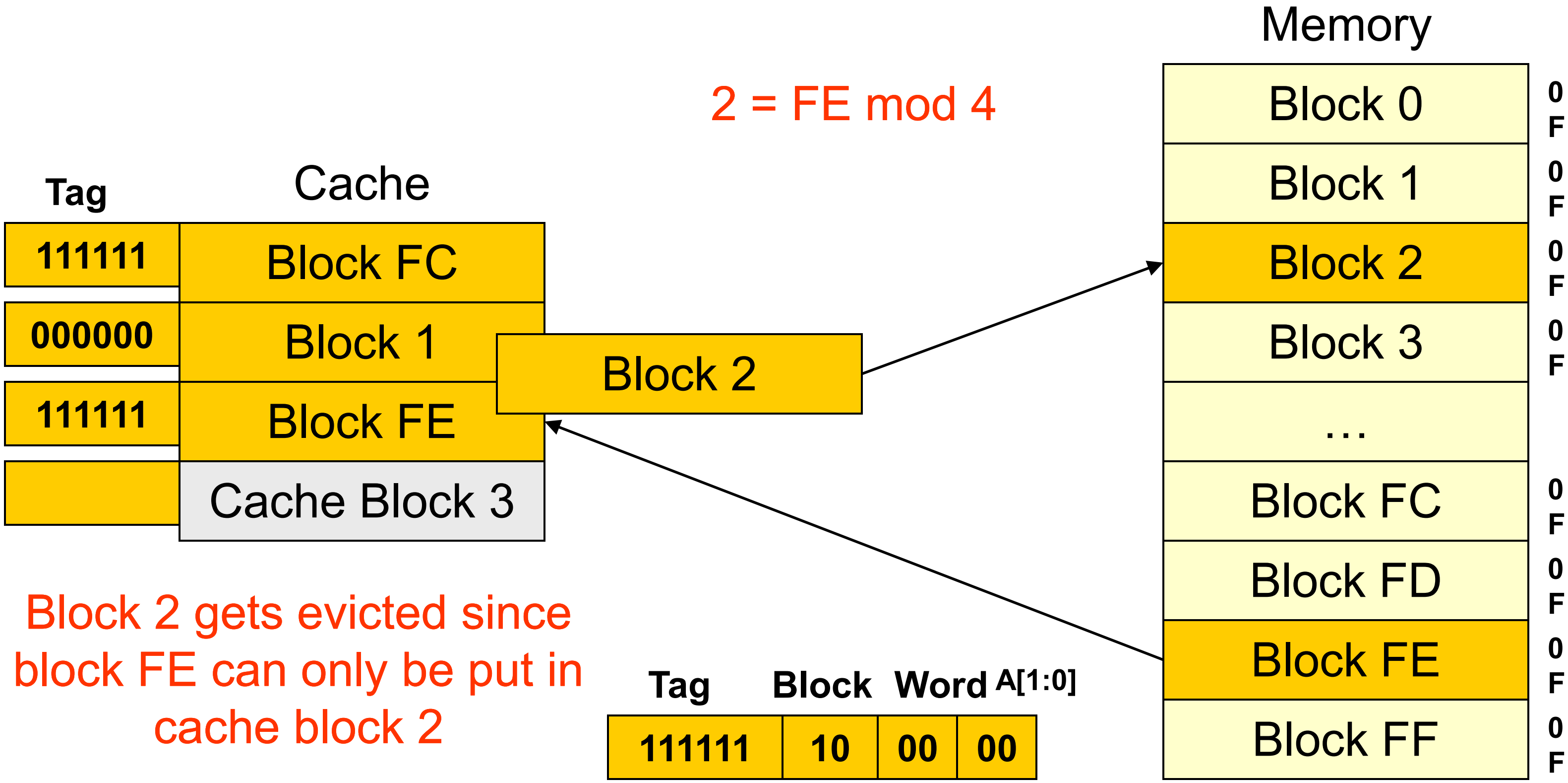
Memory

Memory	
Block 0	0 F
Block 1	0 F
Block 2	0 F
Block 3	0 F
…	
Block FC	0 F
Block FD	0 F
Block FE	0 F
Block FF	0 F

Tag	Block	Word	A[1:0]
000000	10	00	00

Direct Mapping

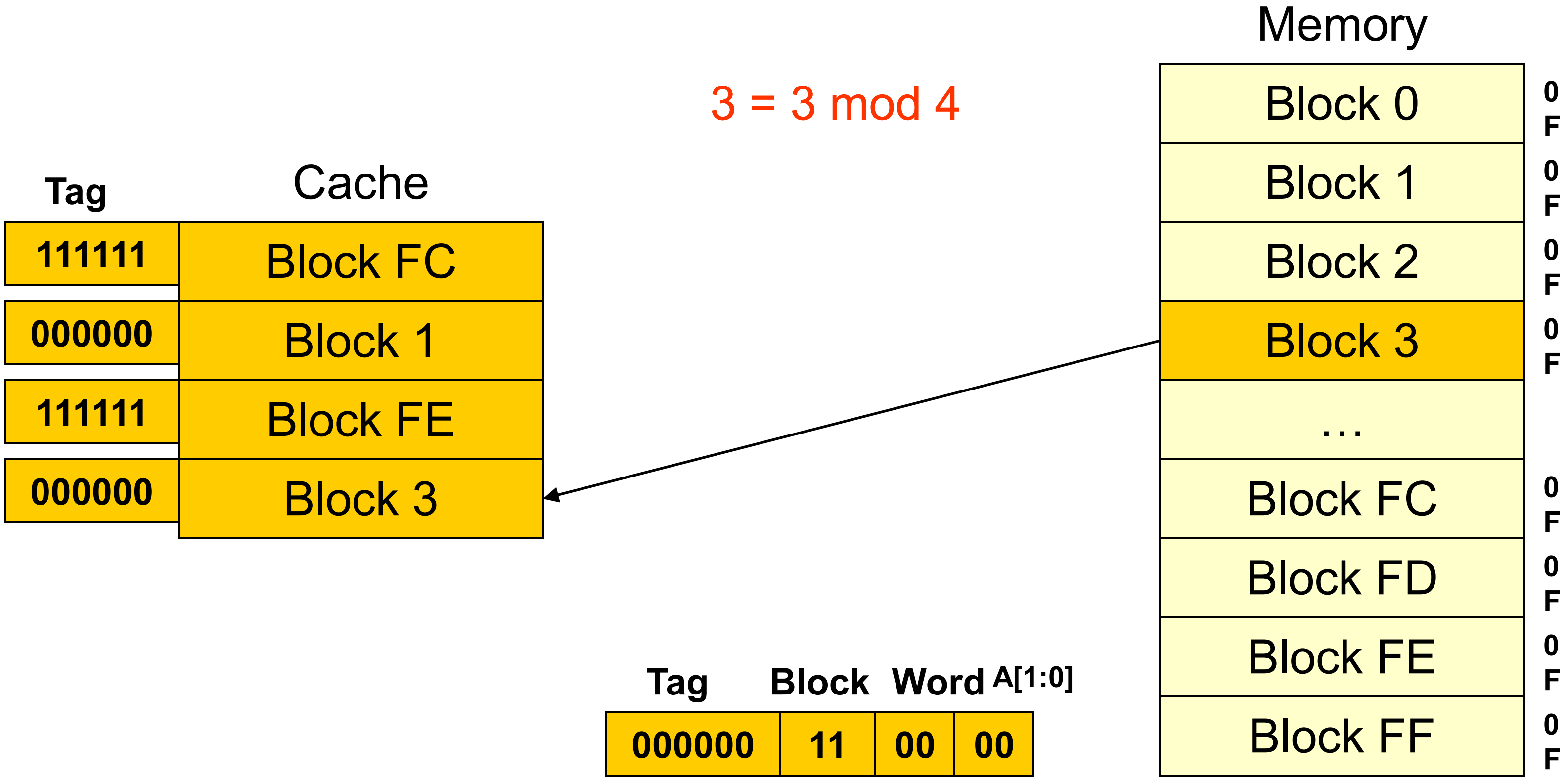
- Each block from memory can only be put in one location
- Block i mod n maps to cache block i



Block 2 gets evicted since block FE can only be put in cache block 2

Direct Mapping

- Each block from memory can only be put in one location
- Block i mod n maps to cache block i



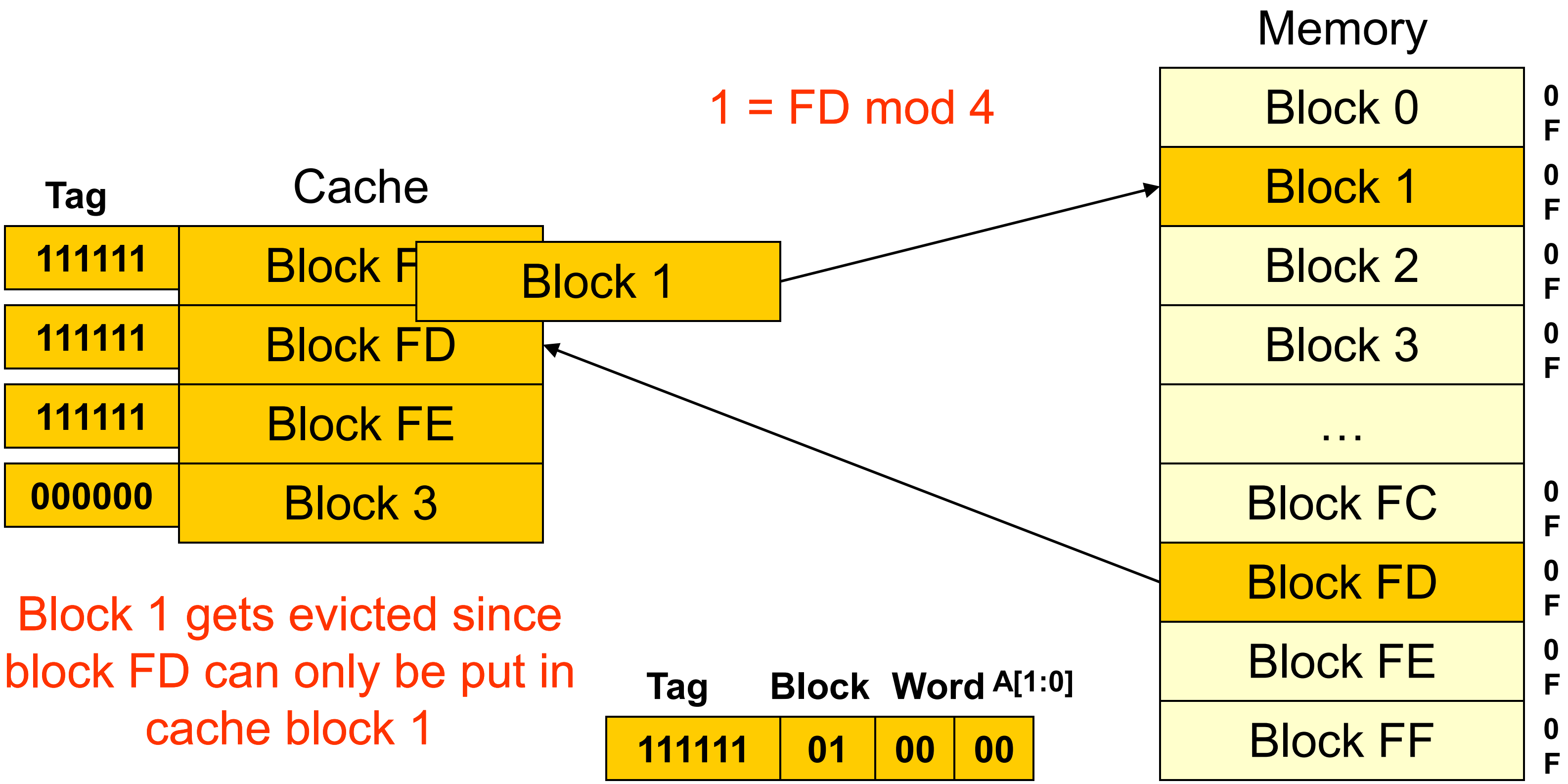
Tag	Cache
111111	Block FC
000000	Block 1
111111	Block FE
000000	Block 3

Memory	
Block 0	0 F
Block 1	0 F
Block 2	0 F
Block 3	0 F
…	
Block FC	0 F
Block FD	0 F
Block FE	0 F
Block FF	0 F

Tag	Block	Word	A[1:0]
000000	11	00	00

Direct Mapping

- Each block from memory can only be put in one location
- Block i mod n maps to cache block i



1 = FD mod 4

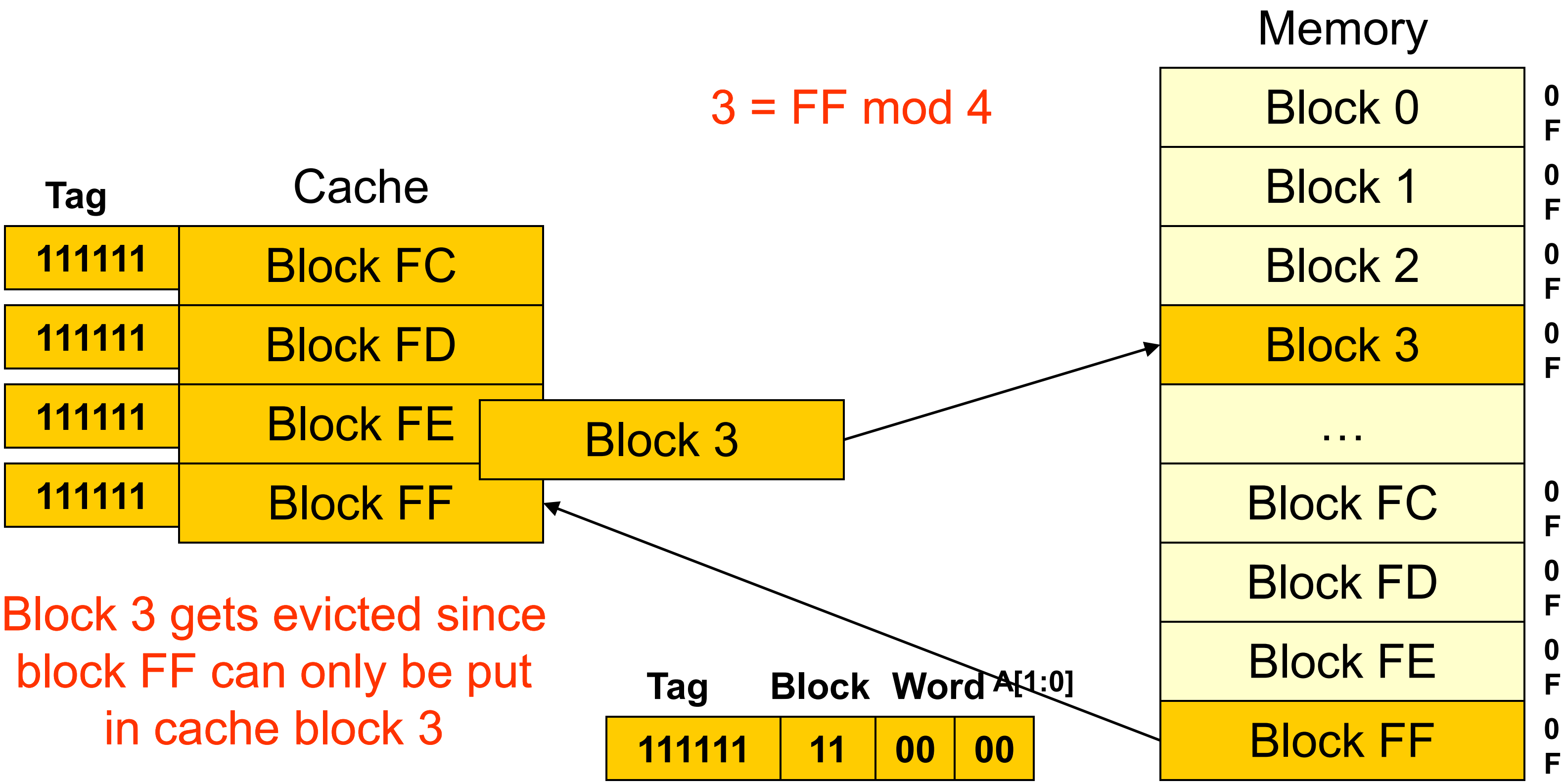
Tag	Cache
111111	Block FF
111111	Block FD
111111	Block FE
000000	Block 3

Block 1 gets evicted since block FD can only be put in cache block 1

Tag	Block	Word	A[1:0]
111111	01	00	00

Direct Mapping

- Each block from memory can only be put in one location
- Block i mod n maps to cache block i

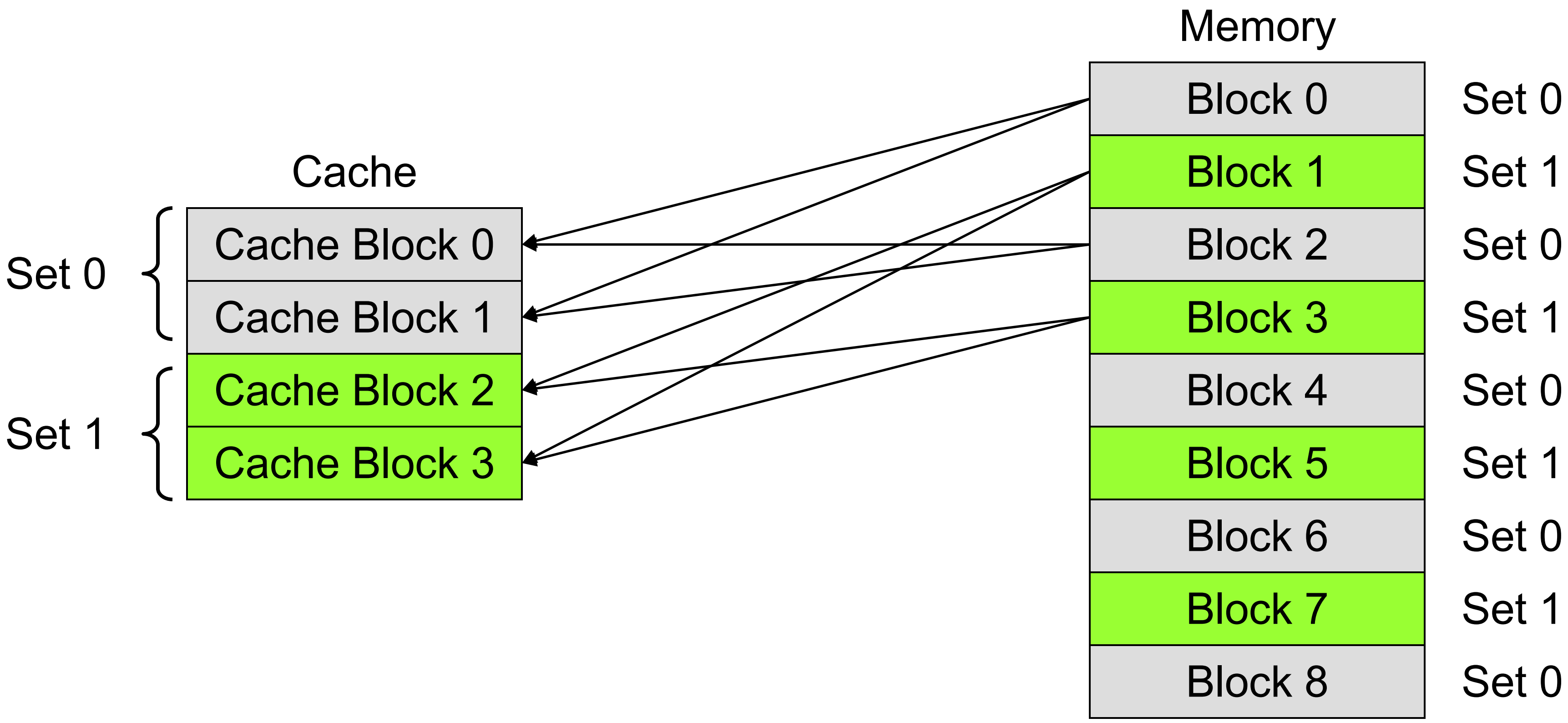


3 = FF mod 4

Block 3 gets evicted since block FF can only be put in cache block 3

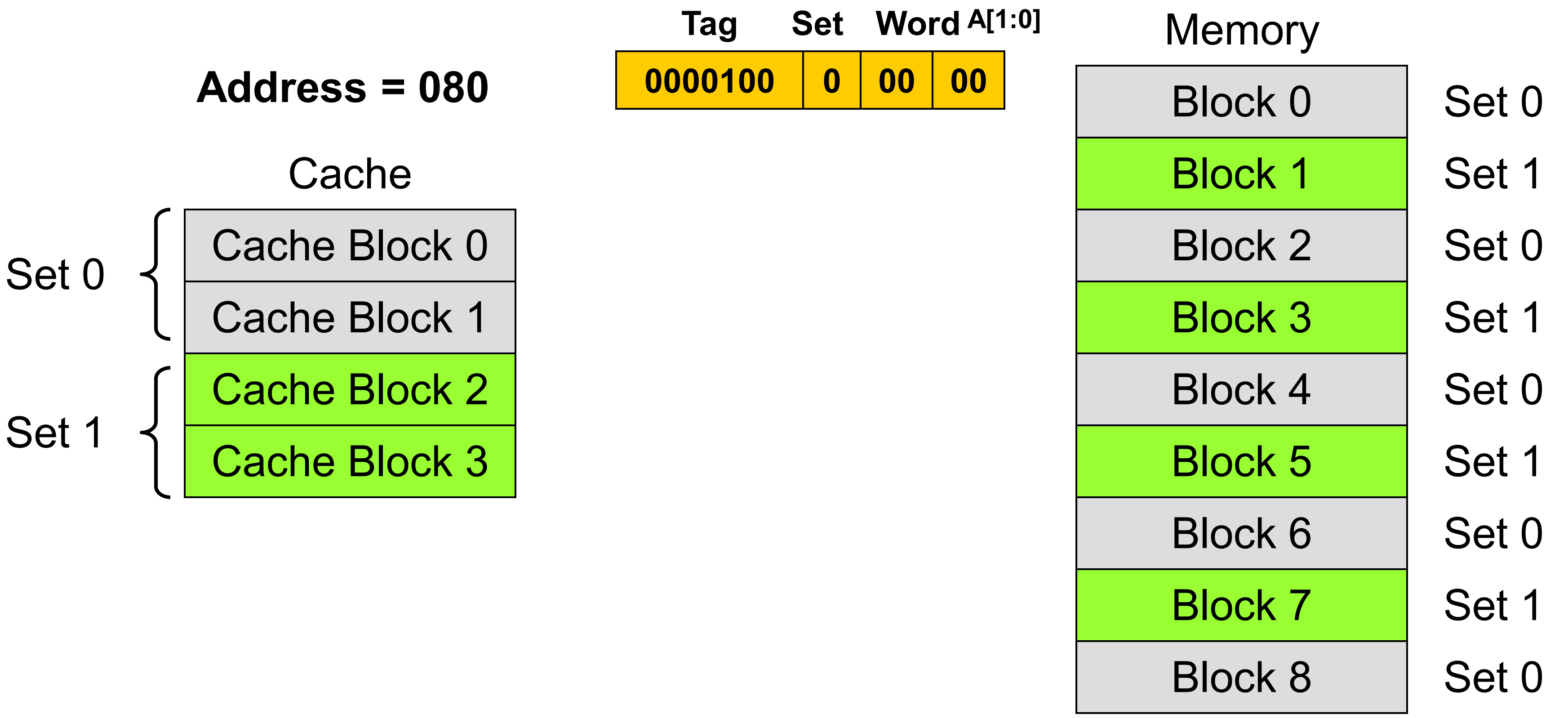
Set-Associative Mapping

- Blocks from set i can map into any cache block from set i



Set-Associative Mapping

- 12-bit address:
  - 4 = 2^2 words per block => 2 bits of address used to determine the desired word w/in the block
  - 2 = 2^1 sets => next 1 bit used to determine which set
  - Tag = Upper 7 bits used to identify the block in the cache



Address = 080

Tag	Set	Word	A[1:0]
0000100	0	00	00

Cache

Set 0	Cache Block 0 Cache Block 1
Set 1	Cache Block 2 Cache Block 3

Memory

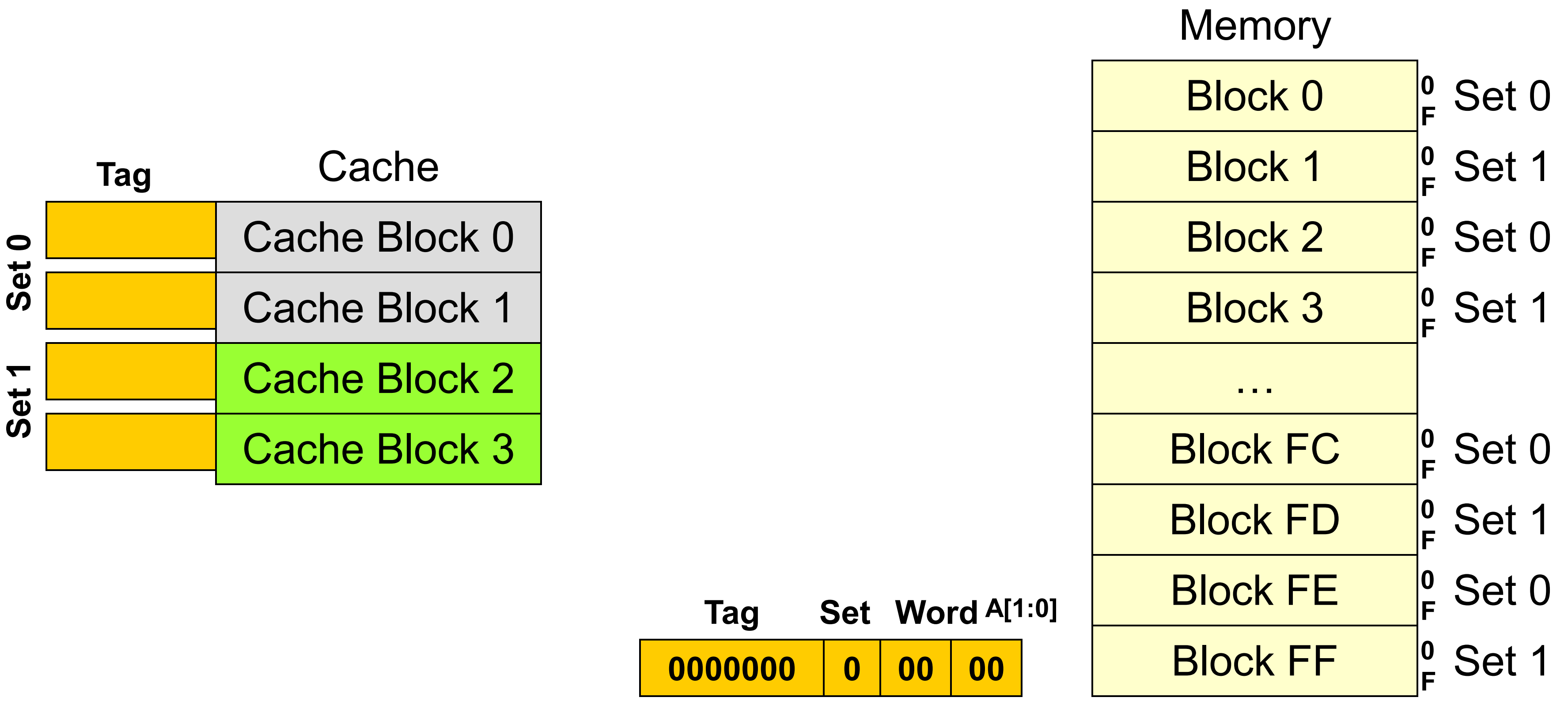
Block 0	Set 0
Block 1	Set 1
Block 2	Set 0
Block 3	Set 1
Block 4	Set 0
Block 5	Set 1
Block 6	Set 0
Block 7	Set 1
Block 8	Set 0

K-way Set Associative Address Scheme

- A[1:0] unused (word access only)
- Word bits = $\log_2 B$ bits (B=Block Size)
- Set bits = $\log_2 S$ bits (S = N/K = # of Sets)
- Tag = Remaining bits

Set-Associative Mapping

- Blocks from set i can map into any cache block from set i



Tag

Cache

Set 0

Cache Block 0

Cache Block 1

Set 1

Cache Block 2

Cache Block 3

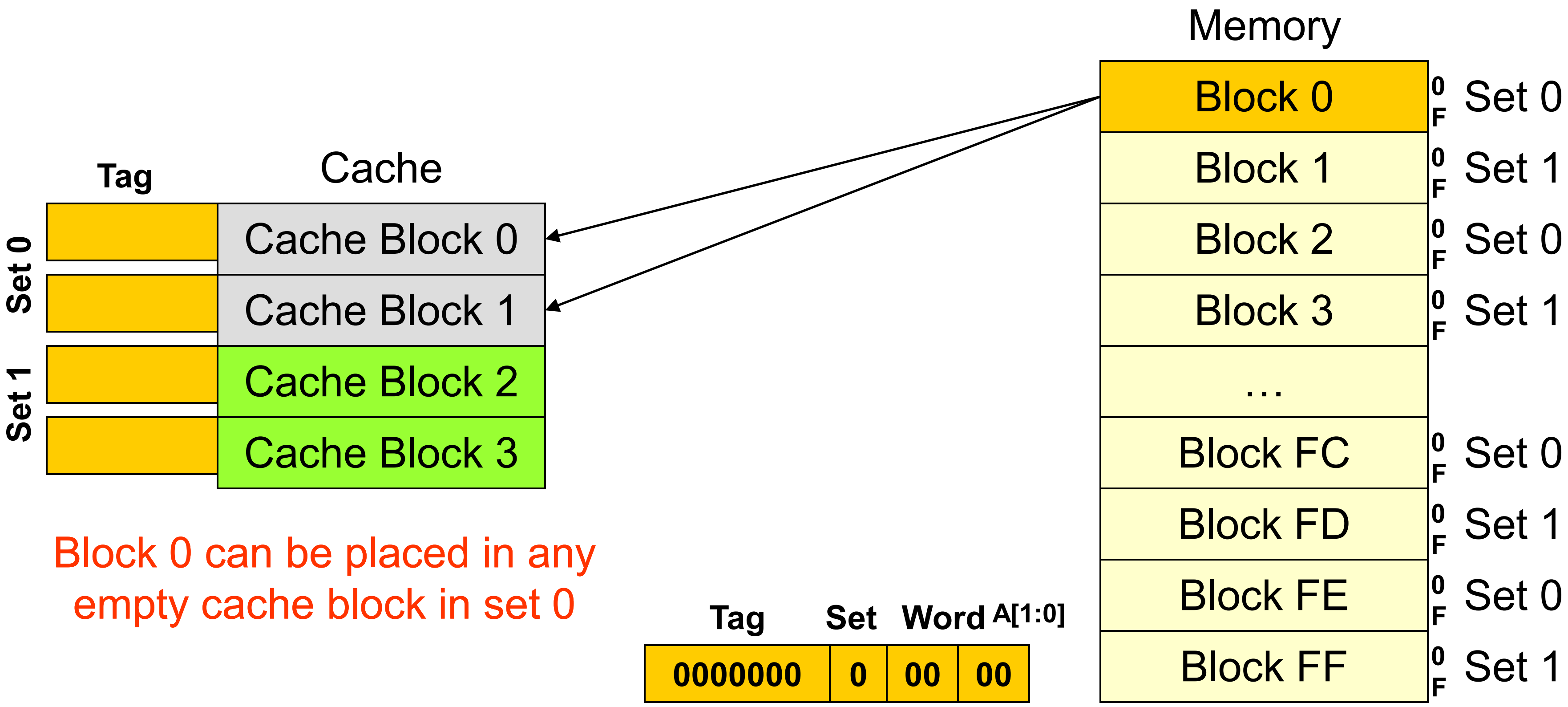
Memory

Block 0	0 F	Set 0
Block 1	0 F	Set 1
Block 2	0 F	Set 0
Block 3	0 F	Set 1
…		
Block FC	0 F	Set 0
Block FD	0 F	Set 1
Block FE	0 F	Set 0
Block FF	0 F	Set 1

Tag	Set	Word	A[1:0]
0000000	0	00	00

Set-Associative Mapping

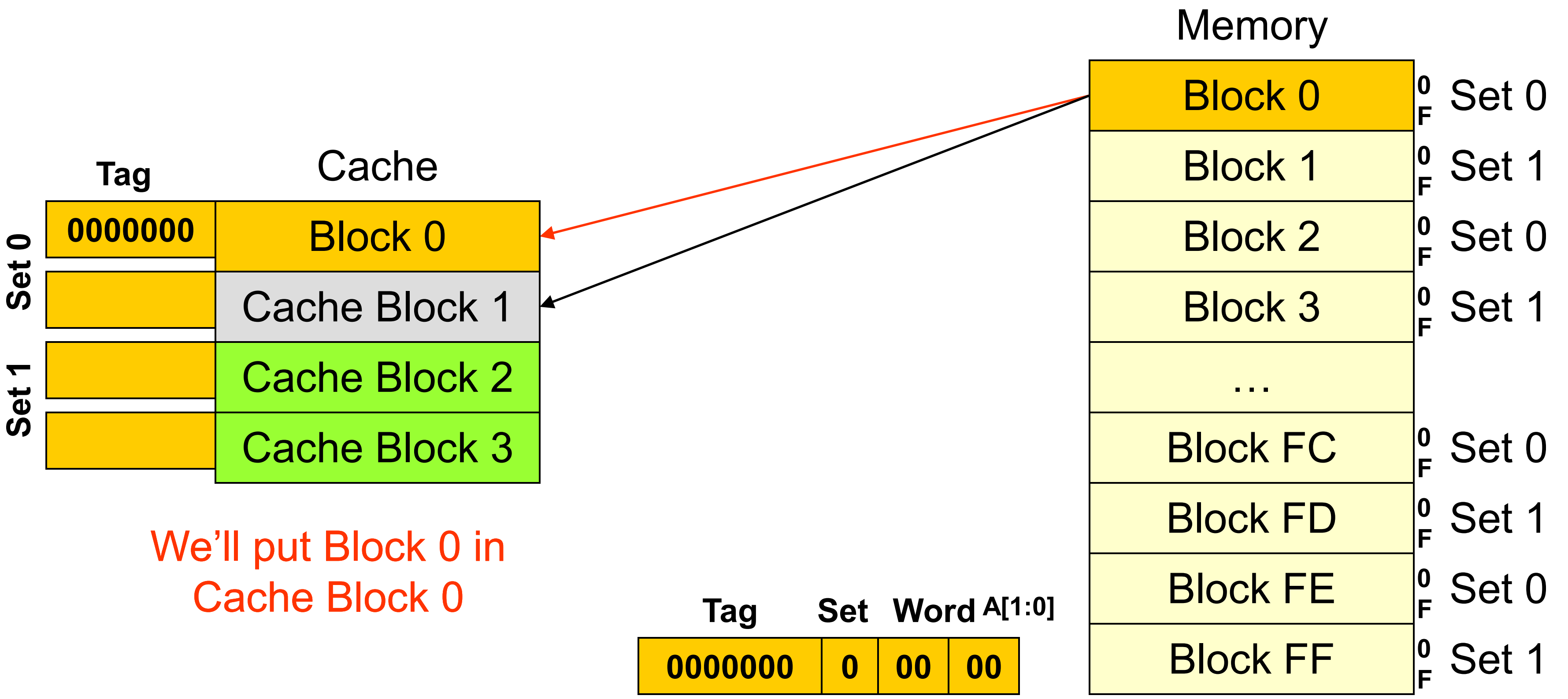
- Blocks from set i can map into any cache block from set i



Block 0 can be placed in any empty cache block in set 0

Set-Associative Mapping

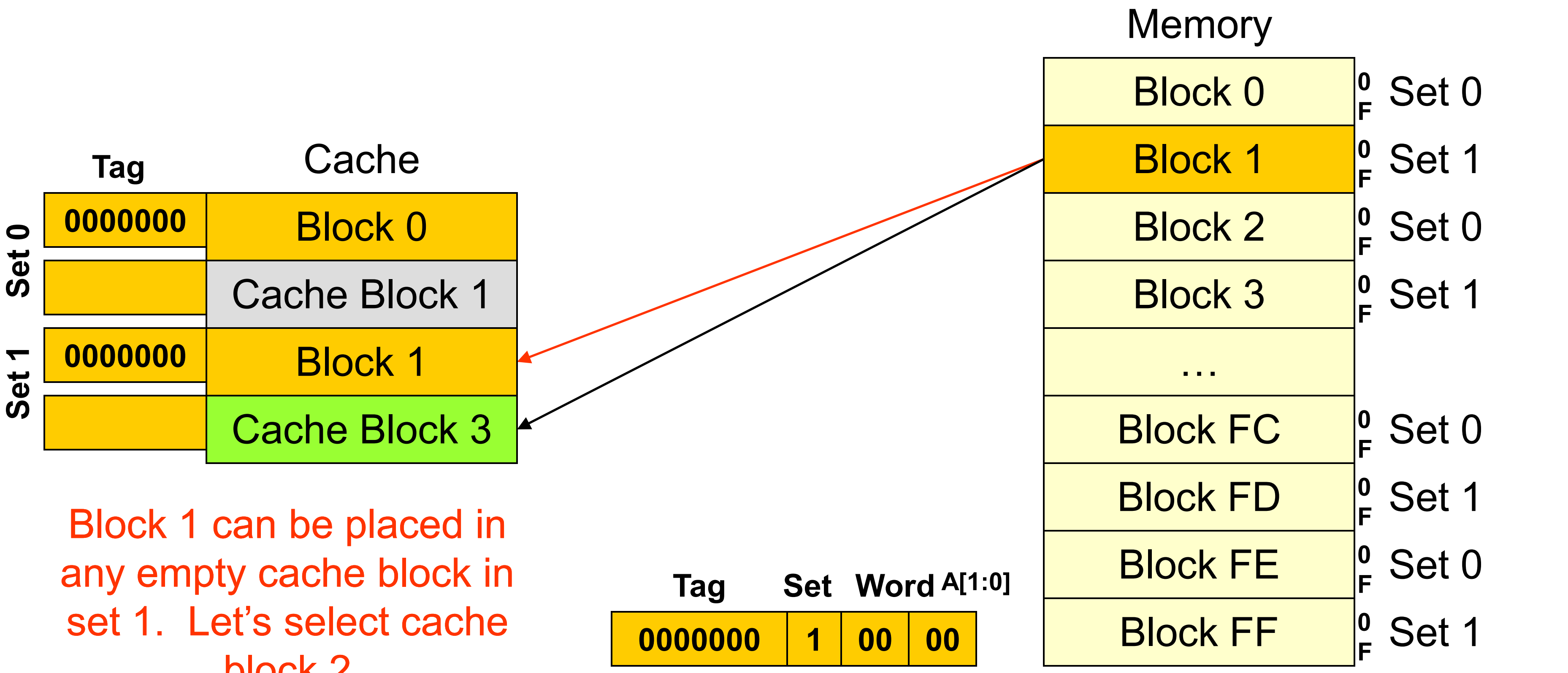
- Blocks from set i can map into any cache block from set i



We'll put Block 0 in Cache Block 0

Set-Associative Mapping

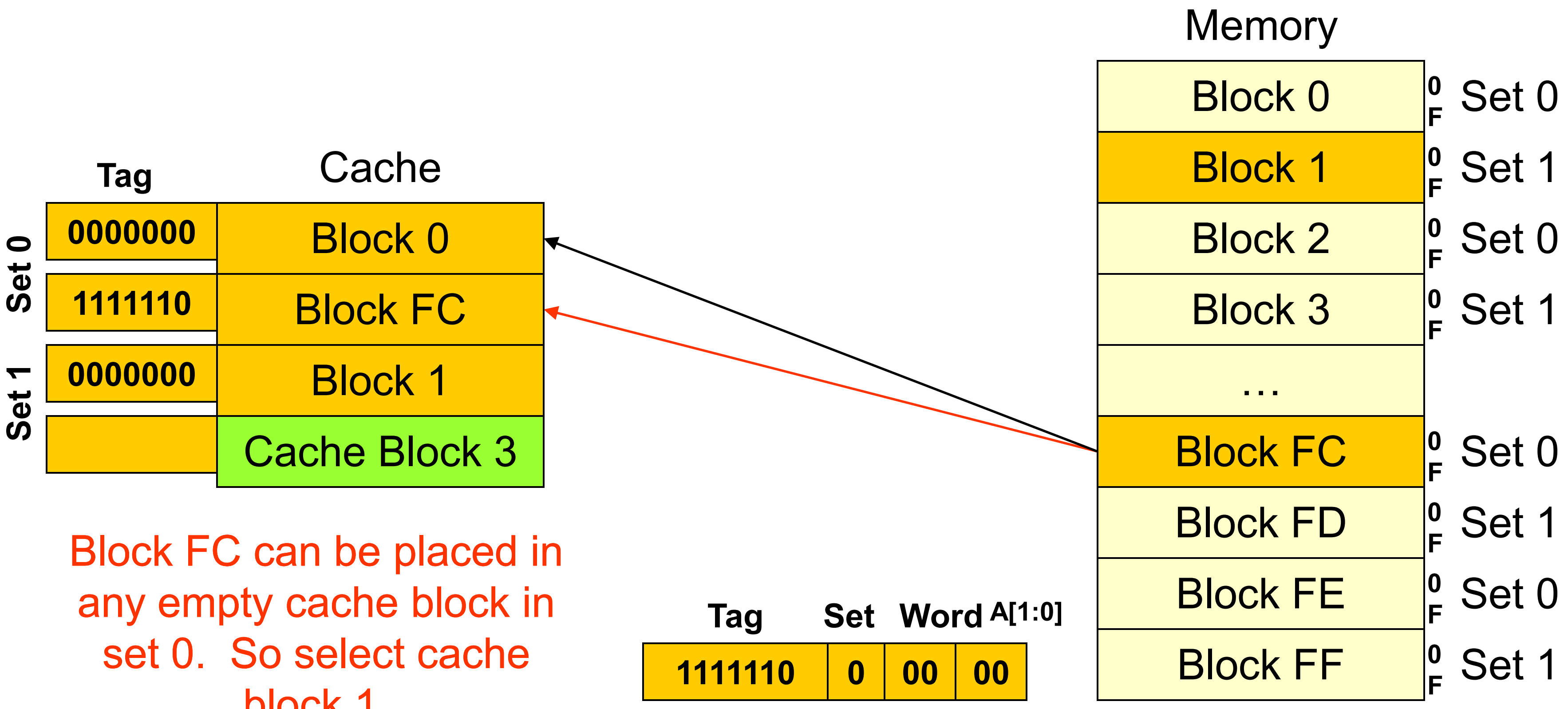
- Blocks from set i can map into any cache block from set i



Block 1 can be placed in any empty cache block in set 1. Let's select cache block 2

Set-Associative Mapping

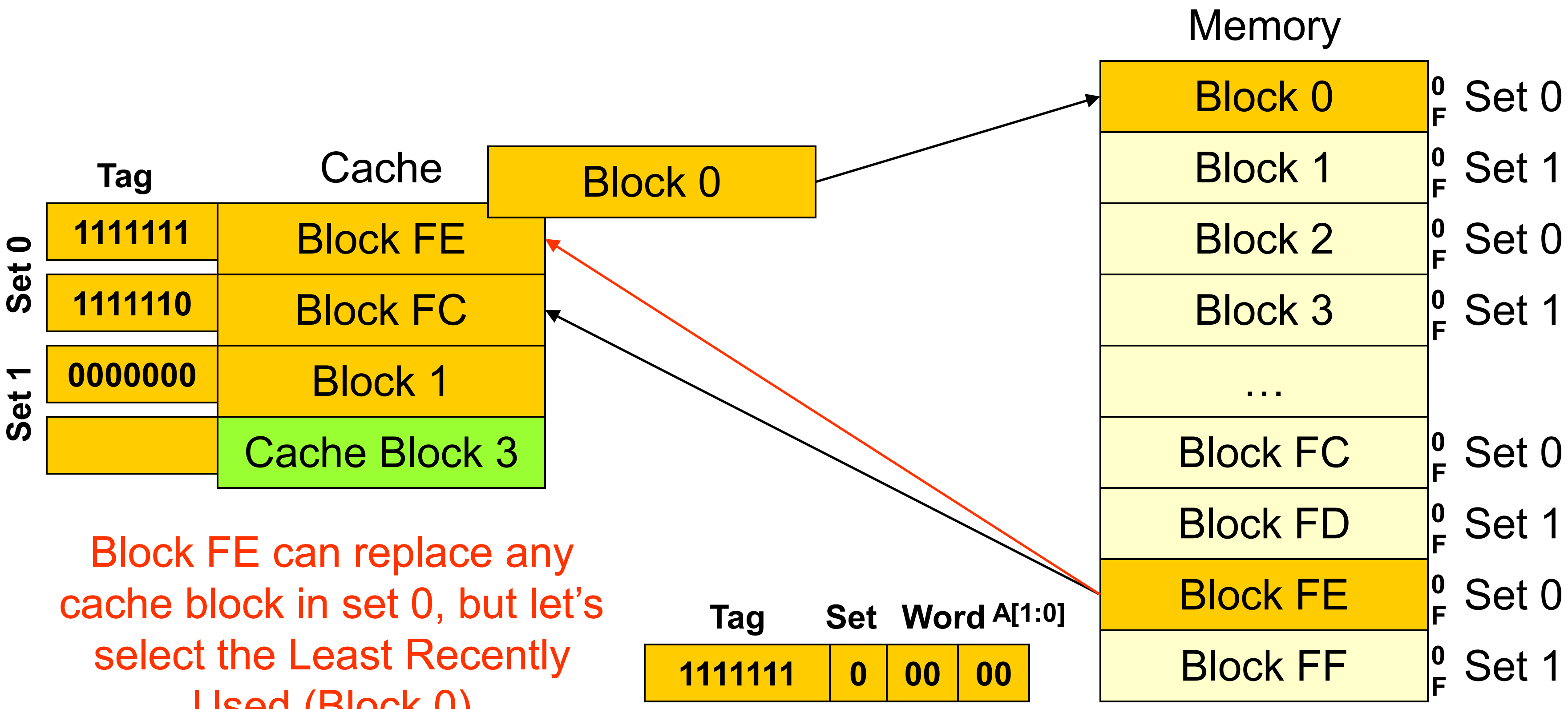
- Blocks from set i can map into any cache block from set i



Block FC can be placed in any empty cache block in set 0. So select cache block 1.

Set-Associative Mapping

- Blocks from set i can map into any cache block from set i



Block FE can replace any cache block in set 0, but let's select the Least Recently Used (Block 0)

Address Mapping Examples

- 16-bit addresses, 1 KB cache, 8 words/block
- Find address mapping for:
  - Fully Associative
  - Direct Mapping
  - 4-way Set Associative
  - 8-way Set Associative

Address Mapping Examples

- First find parameters:
  - B = Block size
  - N = Cache blocks
  - S = Sets for 4-way and 8-way
- B is given as 8 words/block = 32 bytes/block
- N depends on cache size and block size
  - N = (1 KB ÷ 4 bytes/word) ÷ 8 words/block
    = $(2^{10} \div 2^{2}) \div 2^{3} = 2^{5}$ = 32 blocks in the cache
- S for 4-way & 8-way
  - S = N/k = 32/4 = 8 sets
  - S = N/k = 32/8 = 4 sets

Fully Associative

- A1-A0 = unused (only word accesses)
- $\log_2 8$ = 3 word bits (A4-A2)
- Tag = 11 Upper bits (A15-A5)

Parameters:

B = 8
N = 32
S4-way = 8
S8-way = 4

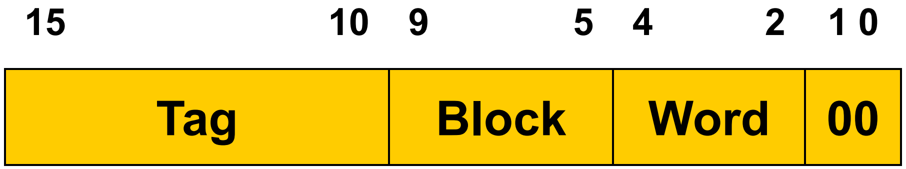
15 5	4 2	1 0
Tag	**Word**	**00**

Direct Mapping

- A1-A0 = unused (only word accesses)
- $\log_2 8$ = 3 word bits (A4-A2)
- $\log_2 32$ = 5 block bits (A9-A5)
- Tag = 6 Upper bits (A15-A10)

Parameters:

B = 8
N = 32
S4-way = 8
S8-way = 4



15 10 9 5 4 2 1 0

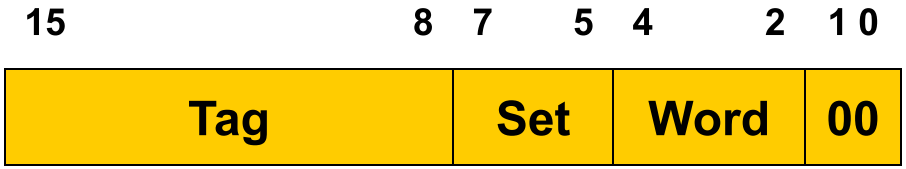
Tag | Block | Word | 00

4-Way Set Assoc. Mapping

- A1-A0 = unused (only word accesses)
- $\log_2 8$ = 3 word bits (A4-A2)
- $\log_2 8$ = 3 set bits (A7-A5)
- Tag = 8 Upper bits (A15-A8)

Parameters:

B = 8
N = 32
S4-way = 8
S8-way = 4

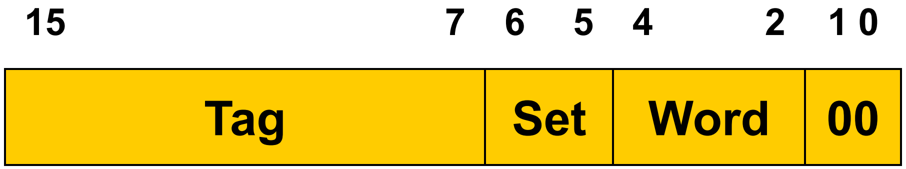


8-Way Set Assoc. Mapping

- A1-A0 = unused (only word accesses)
- $\log_2 8$ = 3 word bits (A4-A2)
- $\log_2 4$ = 2 set bits (A6-A5)
- Tag = 9 Upper bits (A15-A7)

Parameters:

B = 8
N = 32
S4-way = 8
S8-way = 4



15 … 7	6 5	4 … 2	1 0
Tag	Set	Word	00

Cache Operation Example

- Address Trace
  - R: 0x00a0
  - W: 0x00f4
  - R: 0x00b0
  - W: 0x2a2c
- Operations
  - Hit
  - Fetch block XX
  - Evict block XX (w/ or w/o WB)
  - Final WB of block XX)

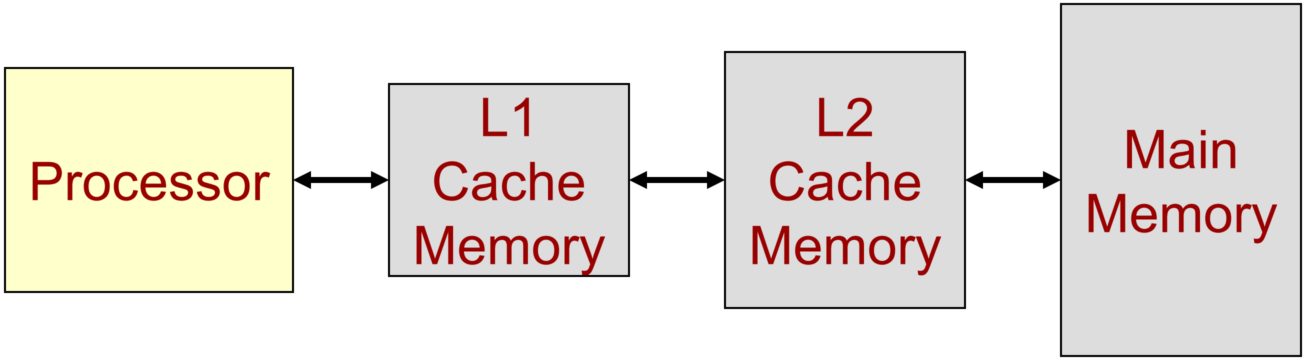
- Perform address breakdown and apply address trace
- 2-Way Set-Assoc, N=4, B=8 words

Address	Tag	Set	Word	Unused
0x00a0	0000 0000 10	1	000	00
0x00f4	0000 0000 11	1	101	00
0x00b0	0000 0000 10	1	100	00
0x2a2c	0010 1010 00	1	011	00

Processor Access	Cache Operation
R: 0x00a0	Fetch Block 00a0-00bf
W: 0x00f4	Fetch Block 00e0-00ff
R: 0x00b0	Hit
W: 0x2a2c	Evict 00e0-00ff w/ WB Fetch Block 2a20-2a3f
Done!	Final WB of 2a20-2a3f

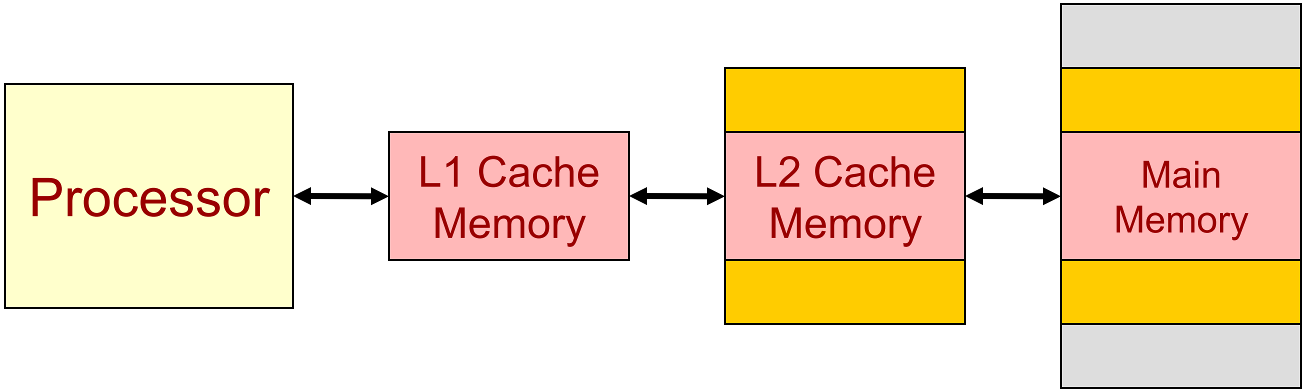
Levels of Cache

- If one cache is good, 2 or more might be better
- L1 cache is closest to processor with increasing numbers progressing outwards
- L1 is smallest and fastest cache with higher levels being slower but bigger to hold more data
- On a read or write, check L1 cache. On a miss, check L2. If it hits in L2 bring the block into L1. If L2 misses then get the data from memory



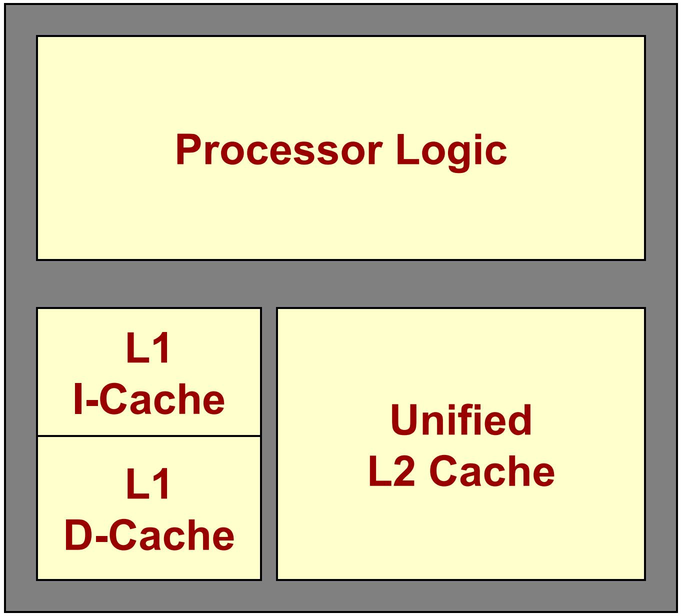
Principle of Inclusion

- When a block is brought in from memory or an upper level, it is brought into all lower levels, not just L1
- This implies that lower levels always contains a subset of higher levels
  - L1 contains most recently used data
  - L2 contains that data + data used earlier
  - MM contains all data



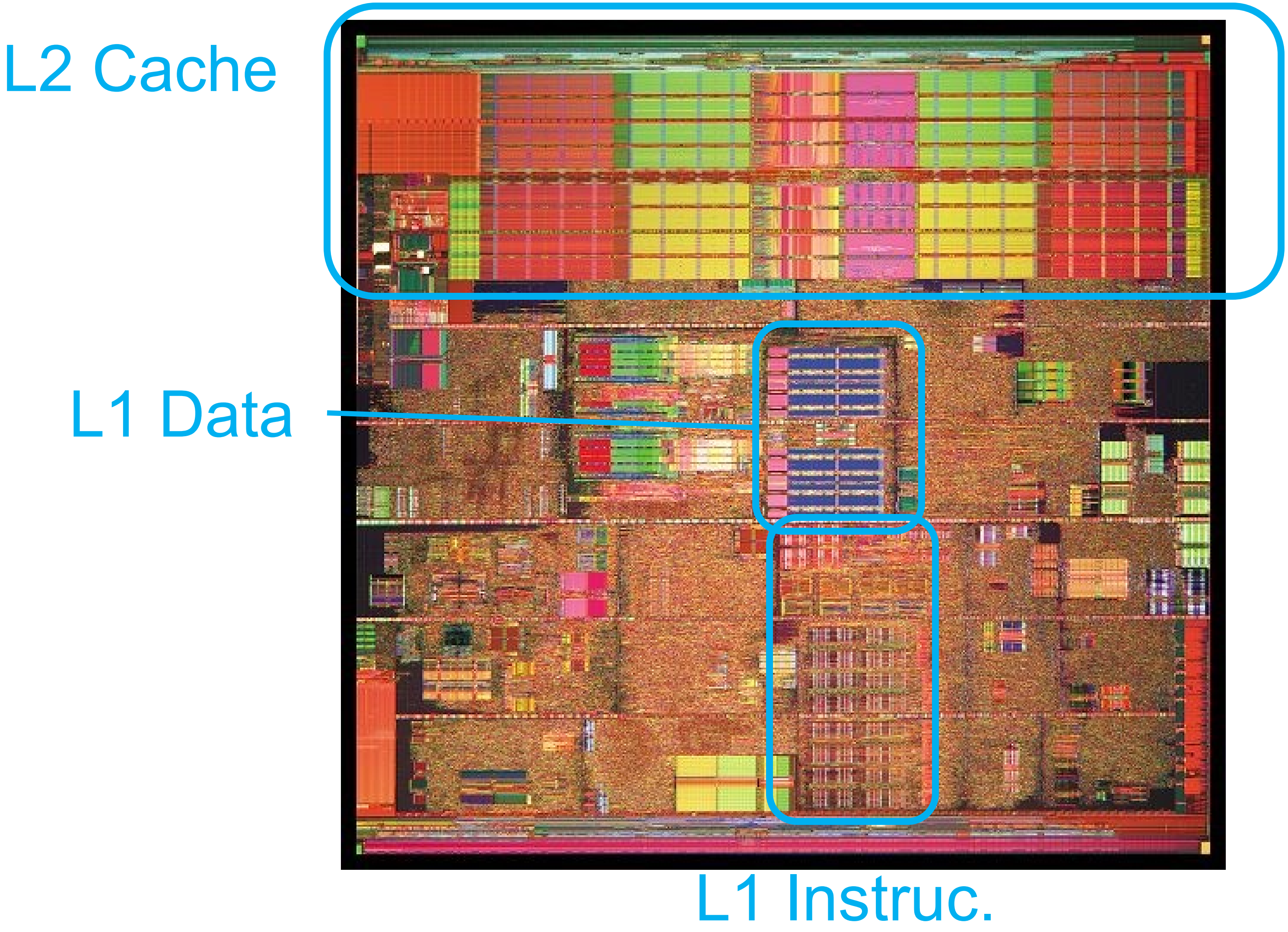
Pentium 4 Cache

- L1 Cache – Broken into separate caches to hold instructions or data
  - I-Cache (Instruction)
    - 12K Entries
  - D-Cache (Data)
    - 16 KB
    - 4-Way Set Assoc.
- L2 Cache
  - Both I & D together
  - 1-2 MB
  - 8-Way Set Associative

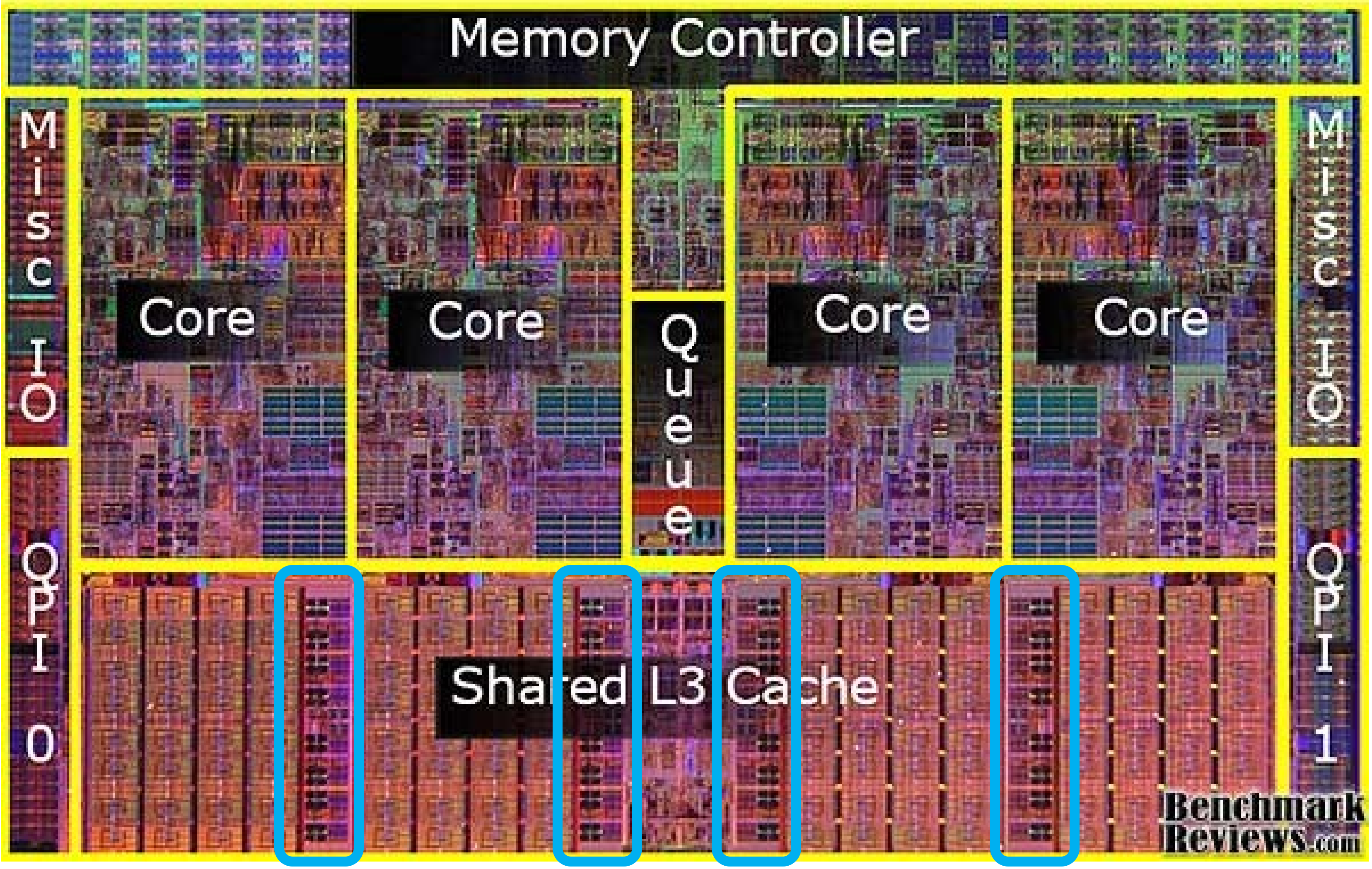


Pentium 4 Cache

Pentium 4



Intel Nehalem Quad Core



Average Access Time

- Define parameters
  - H_i = Hit Rate of Cache Level L_i
    (Note that $1-H_i$ = Miss rate)
  - T_i = Access time of level i
  - R_i = Burst rate per word of level i (after startup access time)
  - B = Block Size
- We will find T_{AVE} = average access time
- Assume No-Load-Through cache
  - Whether hit or miss, you must spend at least T1 time to get the value from the L1 cache

T_{ave} without L2 cache

- 2 possible cases:
  - Either we have a hit and pay only the L1 cache hit time
  - Or we have a miss and read in the whole block to L1 and then read from L1 to the processor
- $T_{ave} = T_1 + \underbrace{(1-H_1)\bullet[T_{MM} + B\bullet R_{MM}]}_{\textbf{(Miss Rate)*(Miss Penalty)}}$
- For T_1=10ns, H_1 = 0.9, B=8, T_{MM}=100ns, R_{MM}=25ns
  - T_{ave} = 10 + [(0.1) • (100+8•25)] = 40 ns

T_{ave} with L2 cache

- 3 possible cases:
  - Either we have a hit and pay the L1 cache hit time
  - Or we miss L1 but hit L2 and read in the block from L2
  - Or we miss L1 and L2 and read in the block from MM
- $T_{ave} = T_1 + \underbrace{(1\text{-}H_1)\bullet H_2\bullet(T_2+B\bullet R_2)}_{\textbf{L1 miss / L2 Hit}} + \underbrace{(1\text{-}H_1)\bullet(1\text{-}H_2)\bullet(T_{MM}+B\bullet R_{MM})}_{\textbf{L1 miss / L2 Miss}}$
- For T_1 = 10ns, H_1 = 0.9, T_2 = 20ns, R_2 = 10ns, H_2 = 0.98, B=8, T_{MM}=100ns, R_{MM}=25 ns
- $T_{ave} = 10 + (0.1)\bullet(.98)\bullet(20+8\bullet10) + (0.1)\bullet(.02)\bullet(100+8\bullet25)$
  $= 10 + 9.8\text{ ns} + 0.6 = 20.4\text{ ns}$

Cache Configurations

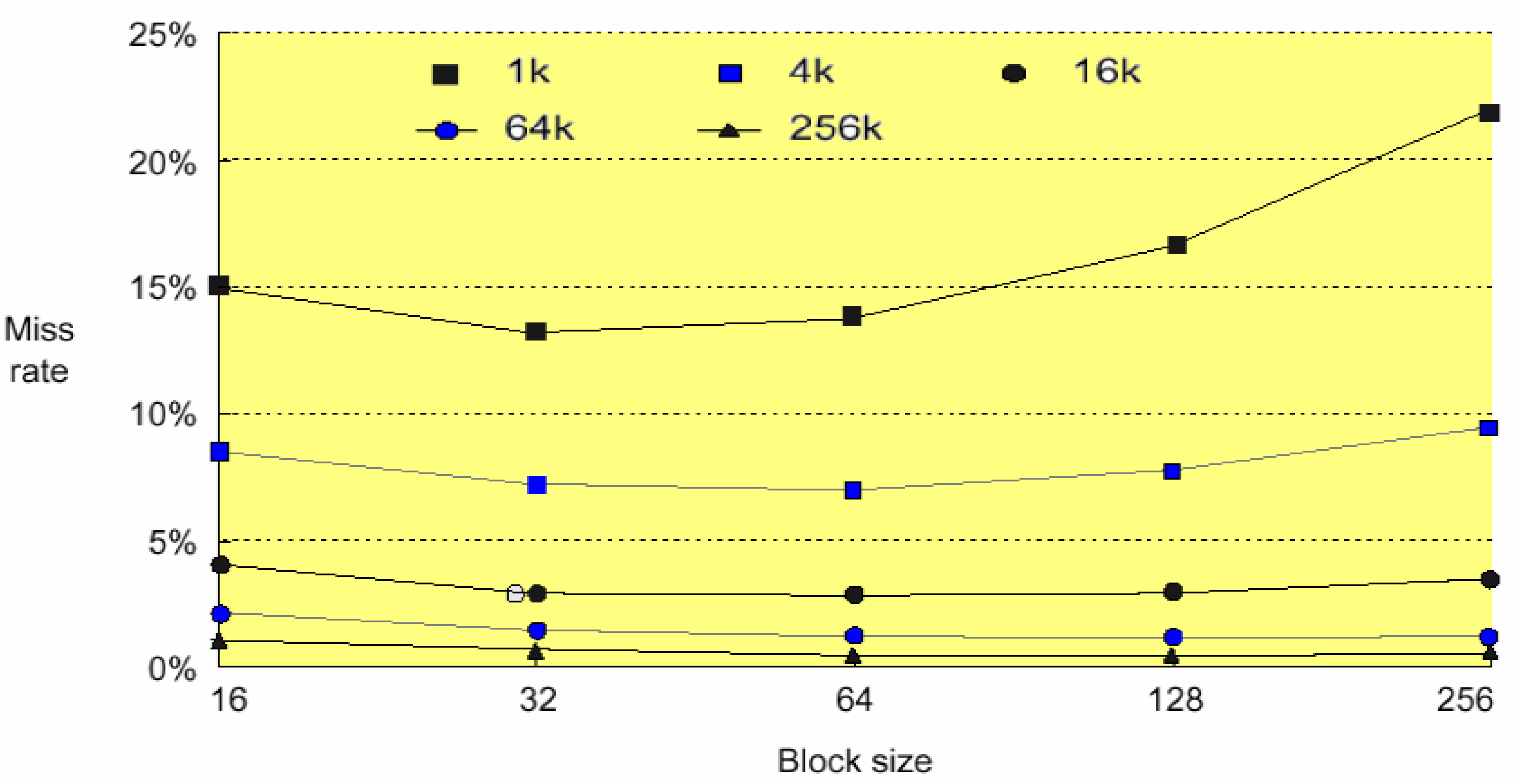
	AMD Opteron	Intel P4	PPC 7447a
Clock rate (2004)	2.0 GHz	3.2 GHz	1.5 – 2 GHz
Instruction Cache	64KB, 2-way SA	96 KB	32 KB, 8-way SA
Latency (clocks)	3	4	1
Data cache	64 KB, 2-way SA	8 KB, 4-way SA	32 KB, 8-way SA
Latency (clocks)	3	2	1
L1 Write Policy	Write-back	Write-through	Programmable
On-chip L2	1 MB, 16-way SA	512 KB, 8-way SA	512 KB, 8-way SA
L2 Latency	6	5	9
Block size (L1/L2)	64	64/128	32/64
L2 Write-Policy	Write-back	Write-back	Programmable

Sources: H&P, “CO&D”, 3rd ed., Freescale.com,

Miss Rate

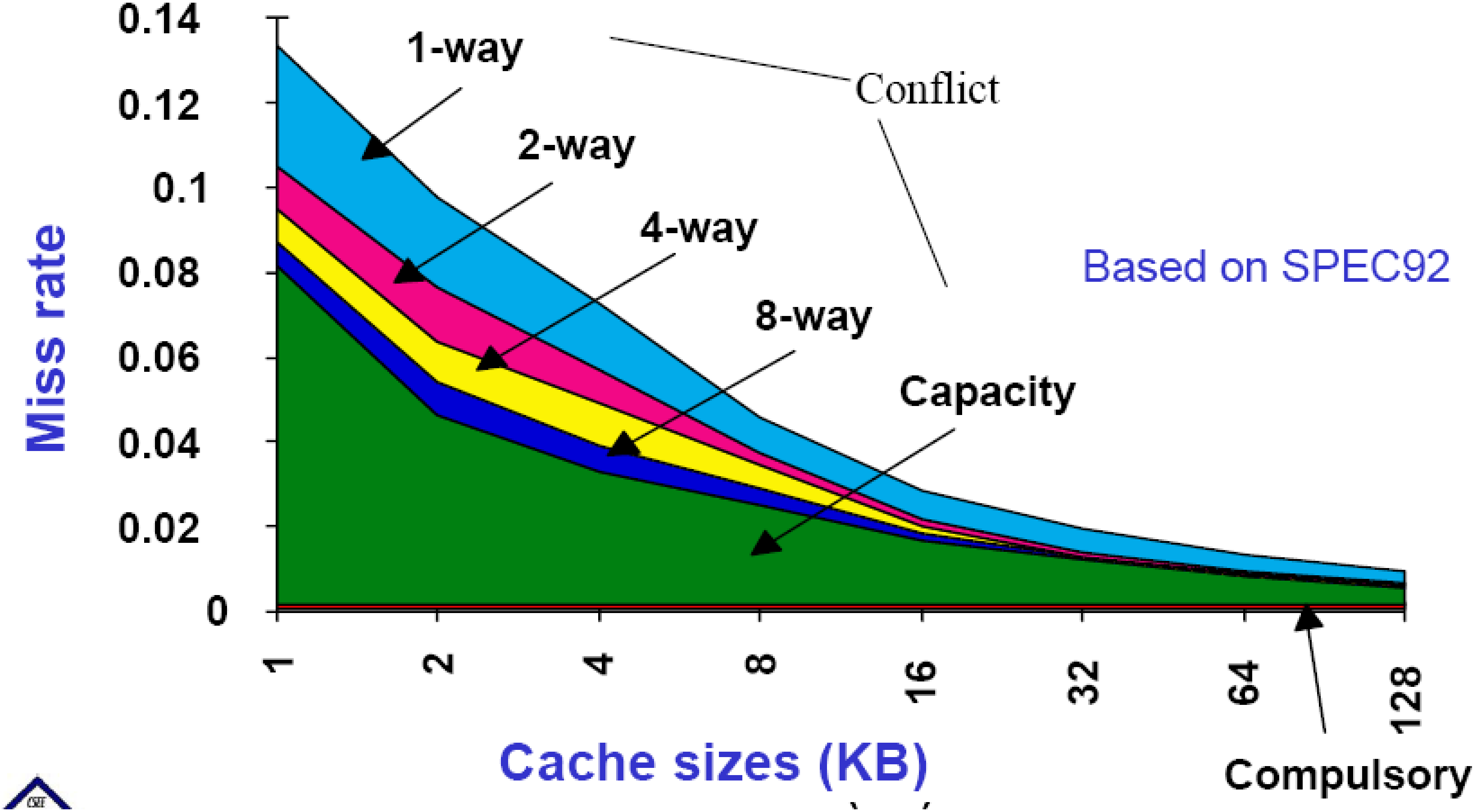
- Reducing Miss Rate means lower T_{AVE}
- To analyze miss rate categorize them based on why they occur
  - Compulsory Misses
    - First access to a block will always result in a miss
  - Capacity Misses
    - Misses because the cache is too small
  - Conflict Misses
    - Misses due to mapping scheme (replacement of direct or set associative)

Miss Rate & Block Size



Graph used courtesy "Computer Architecture: AQA, 3rd ed.", Hennessey and Patterson

Miss Rate & Associativity



Graph used courtesy "Computer Architecture: AQA, 3rd ed.", Hennessey and Patterson

Prefetching

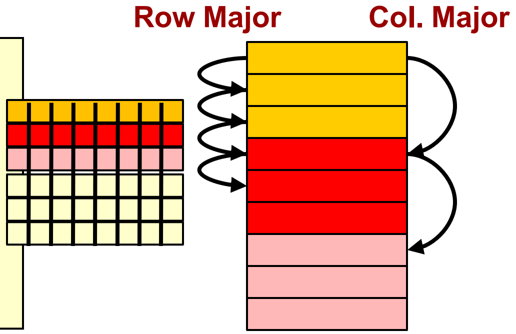
- Hardware Prefetching
  - On miss of block i, fetch block i and i+1
  - Advanced “speculation” / Runahead execution
    - While processor is waiting for a cache block, try to continue executing “future” instructions looking only for memory references and not storing results of instructions
- Software Prefetching
  - Special “Prefetch” Instructions
  - Compiler inserts these instructions to give hints ahead of time as to the upcoming access pattern

Cache-Conscious Programming

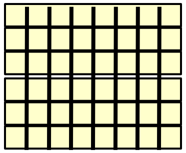
- Order of array indexing
  - Row major vs. column major ordering
- Blocking (keeps working set small)
- Pointer-chasing
  - Linked lists, graphs, tree data structures that use pointers do not exhibit good spatial locality
- General Principles
  - Keep working set reasonably small (temporal locality)
  - Use small strides (spatial locality)
  - Static structures usually better than dynamic ones

```
for(i=0; i<SIZE; i++) {
  for(j=0; j<SIZE; j++) {
    // Row-major
    A[i][j] = A[i][j]*2;
    // Column-major
    A[j][i] = A[j][i]*2;
} }
```

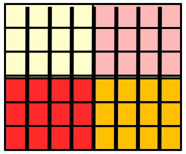
Example of row vs. column major ordering



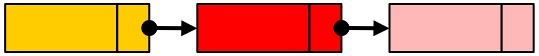
Memory Layout of matrix A



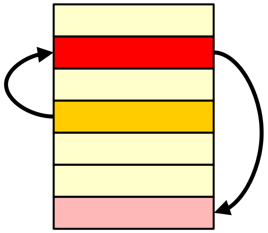
Original Matrix



Blocked Matrix



Linked Lists

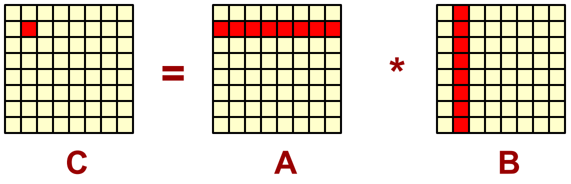


Memory Layout of Linked List

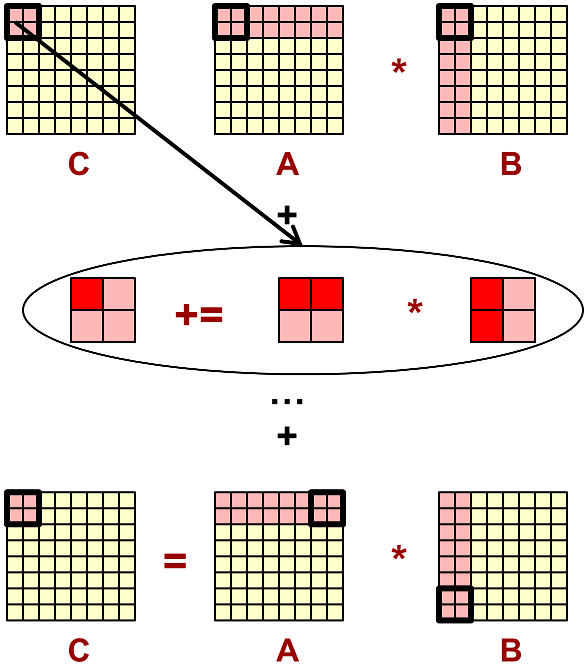
Blocked Matrix Multiply

- Traditional working set
  - 1 row of C, 1 row of A, NxN matrix B
- Break NxN matrix into smaller BxB matrices
  - Perform matrix multiply on blocks
  - Sum results of block multiplies to produce overall multiply result
- Blocked multiply working set
  - Three BxB matrices

```
for(i = 0; i < N; i+=B) {
 for(j = 0; j < N; j+=B) {
  for(k = 0; k < N; k+=B) {
   for(ii = i; ii < i+B; ii++) {
    for(jj = j; jj < j+B; jj++) {
     for(kk = k; kk < k+B; kk++) {
       Cb[ii][jj] += Ab[ii][kk] * Bb[kk][jj];
} } } } } }
```



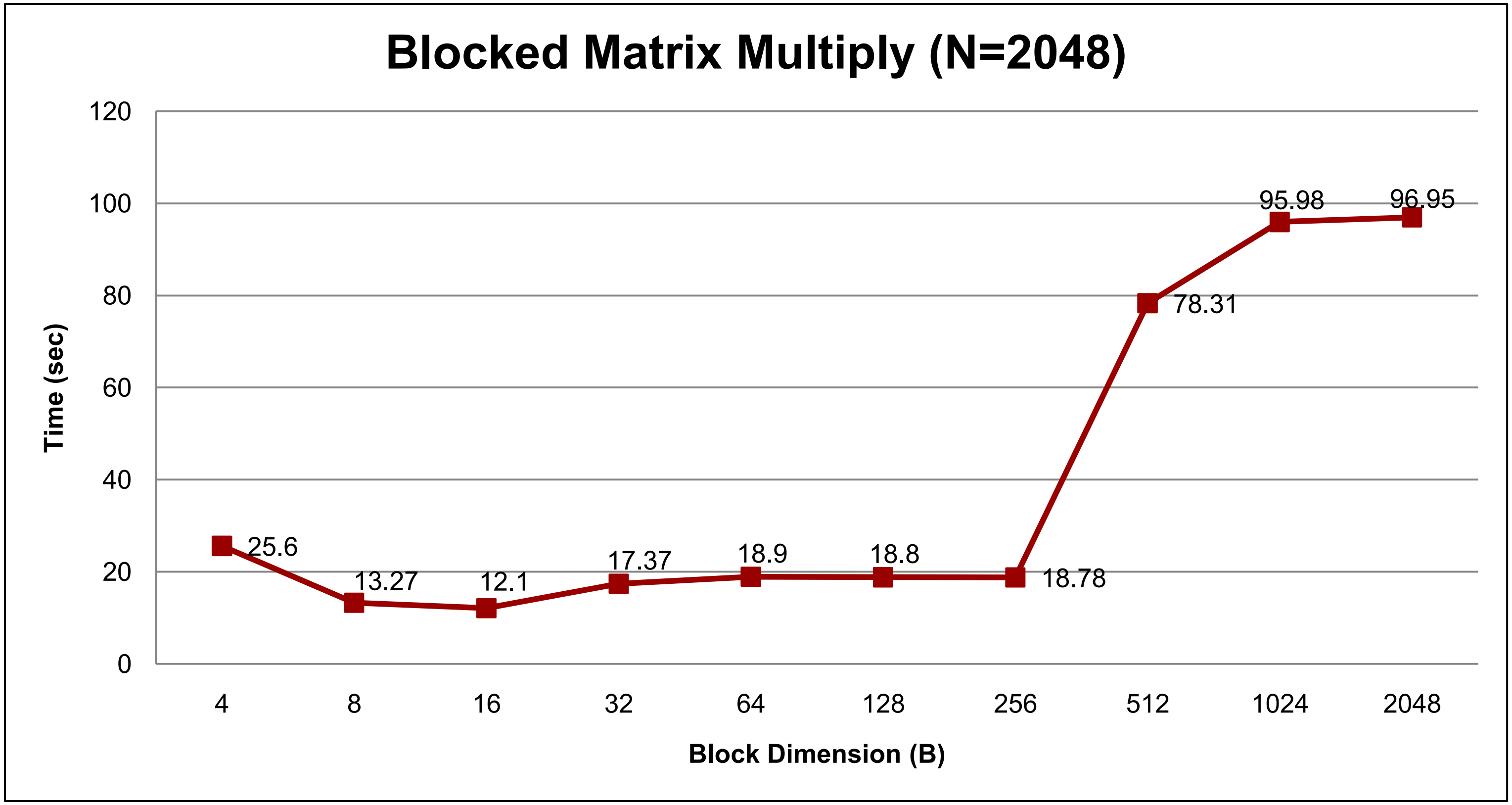
Traditional Multiply



Blocked Multiply

Blocked Multiply Results

- Intel Nehalem processor
  - L1D = 32 KB, L2 = 256KB, L3 = 8 MB


Blocked Matrix Multiply (N=2048)
Time (sec)
Block Dimension (B)