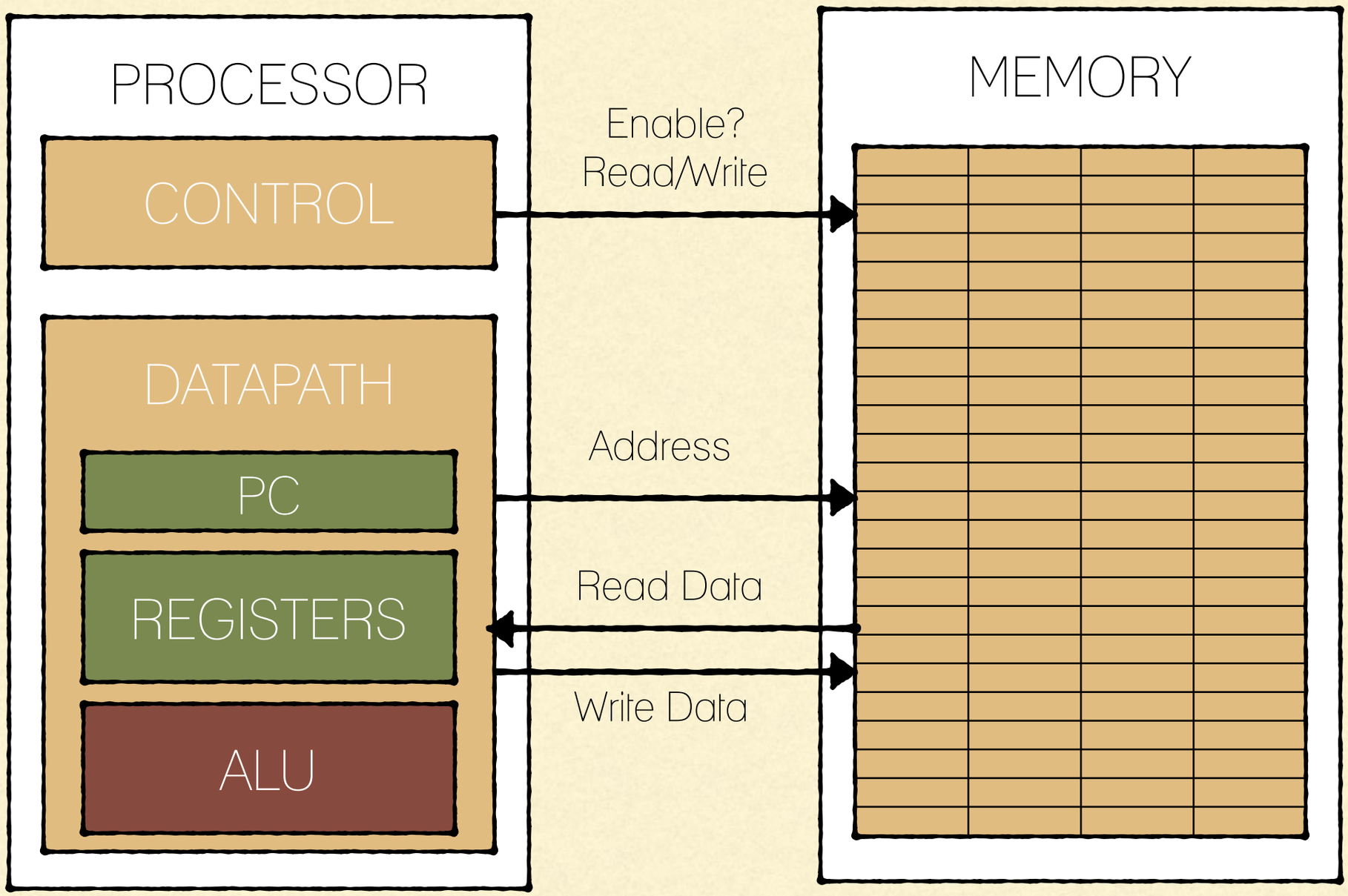
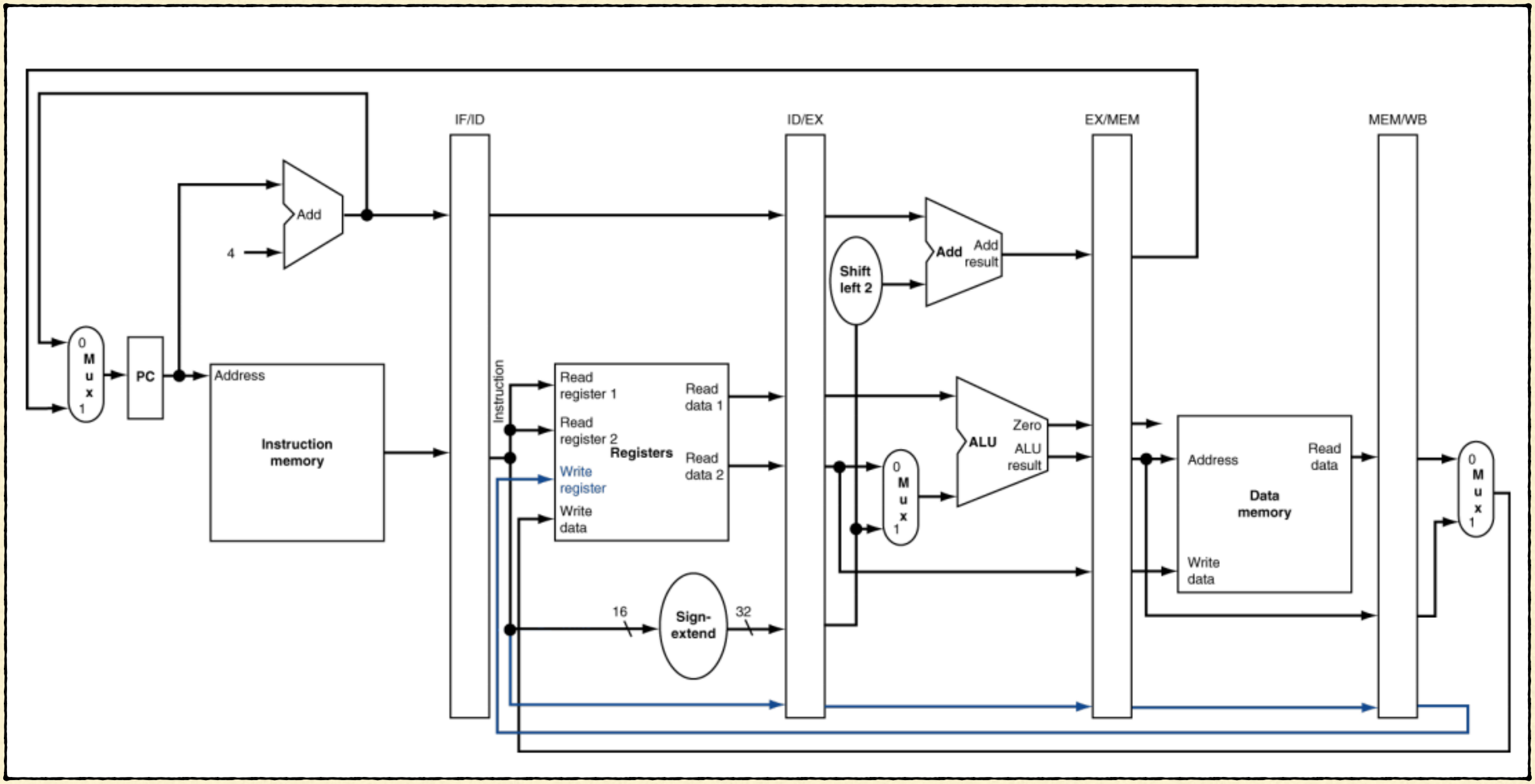
# CACHE

Ayman Hajja, PhD

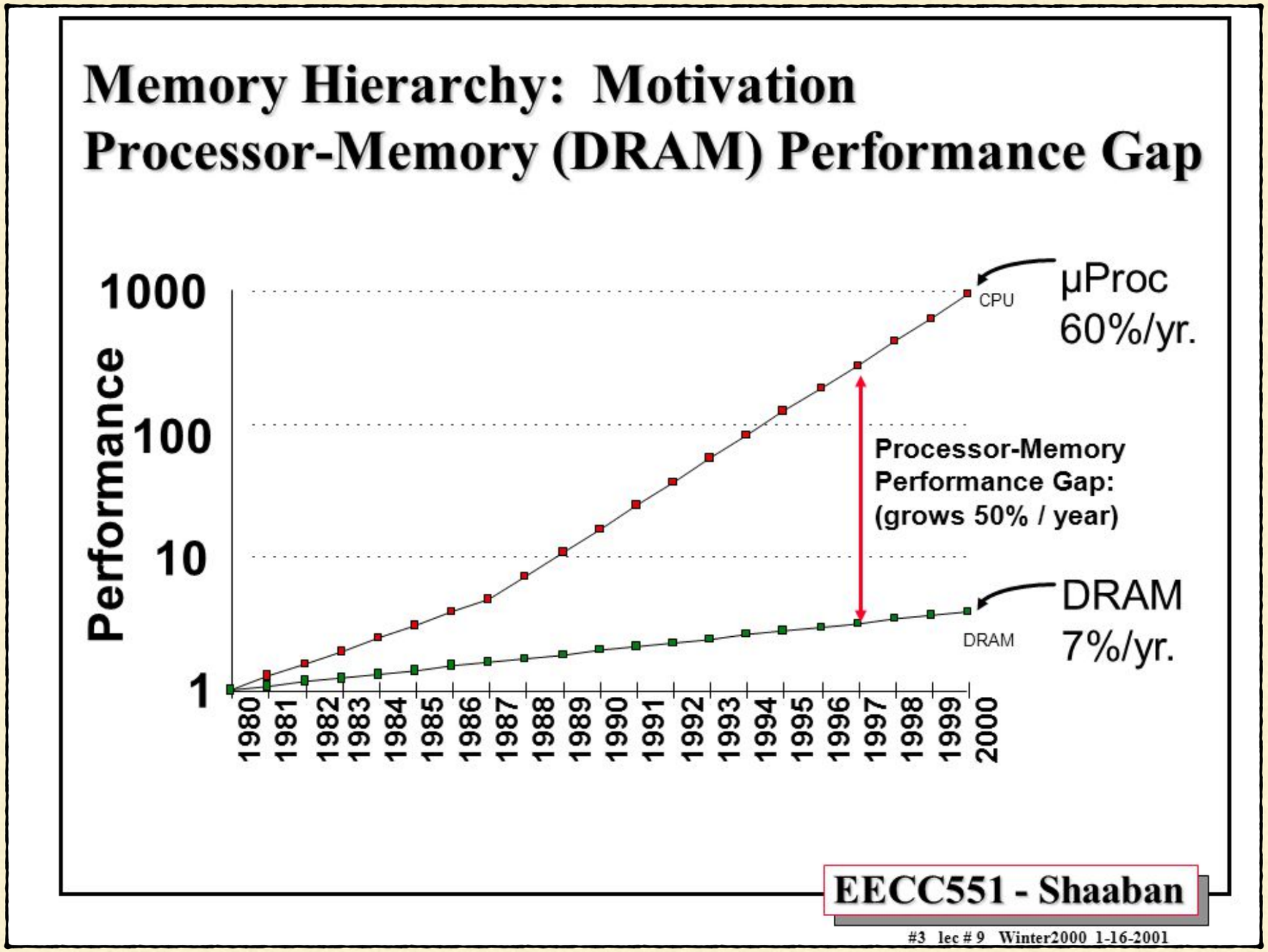
# COMPONENTS OF A COMPUTER



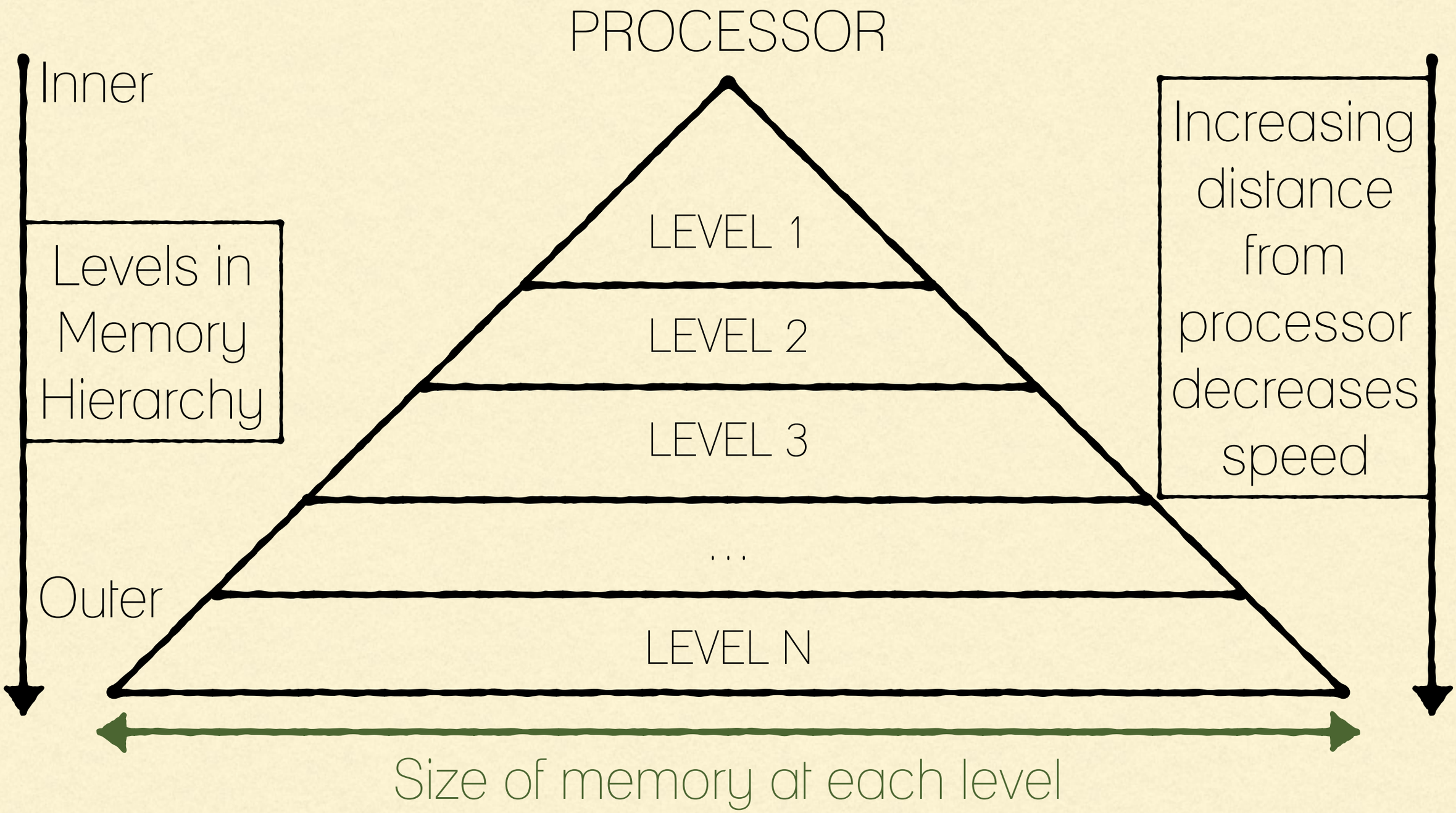
# PIPELINED CPU



# PROCESSOR-DRAM GAP



# MEMORY HIERARCHY



# LIBRARY ANALOGY

- Writing a report about Susan Sontag
- Go to library to get reserved book and place on desk in library
- If need more, check them out and keep on desk but don't return earlier books since might need them
- You hope that this collection of ~10 books on desk enough to write report, despite 10 being only 0.000001% in library

# PRINCIPLE OF LOCALITY (LIBRARY EXAMPLE)

- Temporal Locality (locality in time)
    - Go back to same book on desktop multiple times
    - If a memory location is referenced, then it will tend to be referenced again soon
- Spatial Locality (locality in space)
    - When go to bookshelf, pick up multiple books about Susan Sontag since library stores related books together
    - If a memory location is referenced, the location with nearby addresses will tend to be referenced soon

# PRINCIPLE OF LOCALITY

- Principle of Locality: Programs access small portion of address space at any period of time
- What program structures lead to temporal and spatial locality in instruction access?

# PRINCIPLE OF LOCALITY

- Principle of Locality: Programs access small portion of address space at any period of time
- What program structures lead to temporal and spatial locality in instruction access?
  - Temporal: loops
  - Spatial: sequential fetch

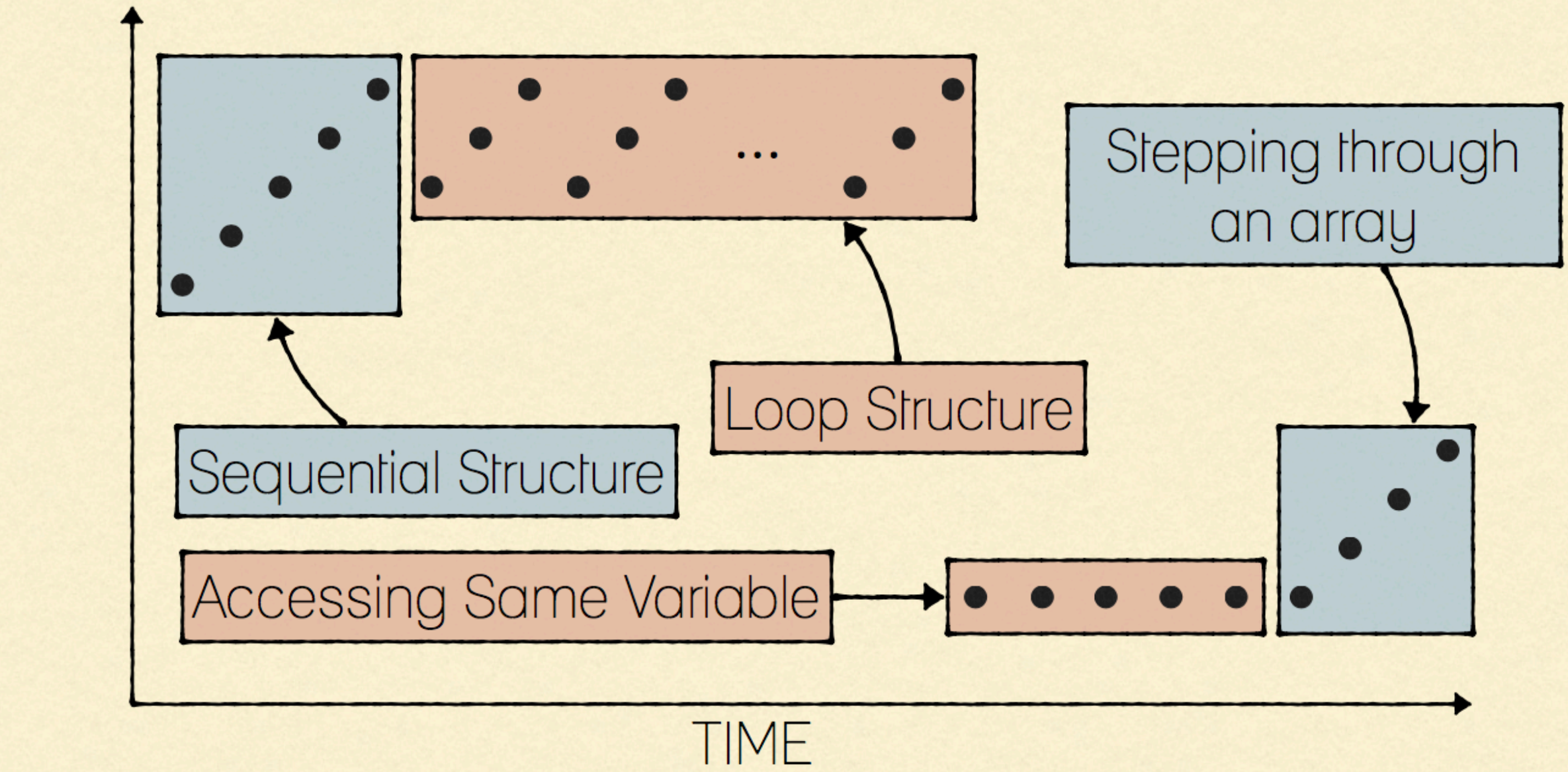
# PRINCIPLE OF LOCALITY

- Principle of Locality: Programs access small portion of address space at any period of time
- What program structures lead to temporal and spatial locality in instruction access?
  - Temporal: loops
  - Spatial: sequential fetch
- How about in data accesses?

# PRINCIPLE OF LOCALITY

- Principle of Locality: Programs access small portion of address space at any period of time
- What program structures lead to temporal and spatial locality in instruction access?
  - Temporal: loops
  - Spatial: sequential fetch
- How about in data accesses?
  - Temporal: using same variables
  - Spatial: stepping through an array

# PRINCIPLE OF LOCALITY



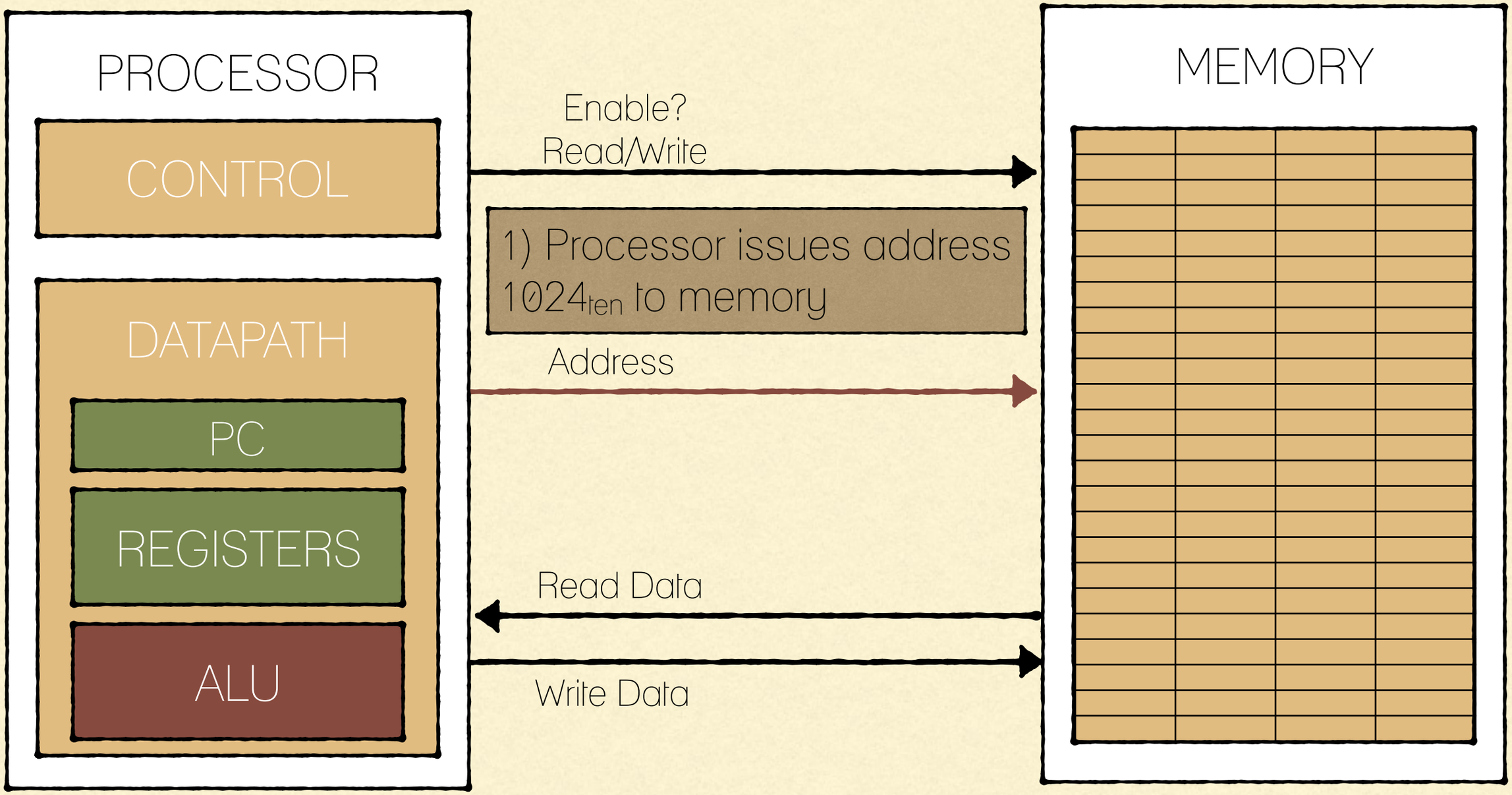
# CACHE PHILOSOPHY

- Programmer-invisible hardware mechanism to give illusion of speed of fastest memory with size of largest memory
  - Works fine even if programmer has no idea what a cache is
  - However, performance-oriented programmers today sometimes “reverse engineer” cache design to design data structures to mach cache

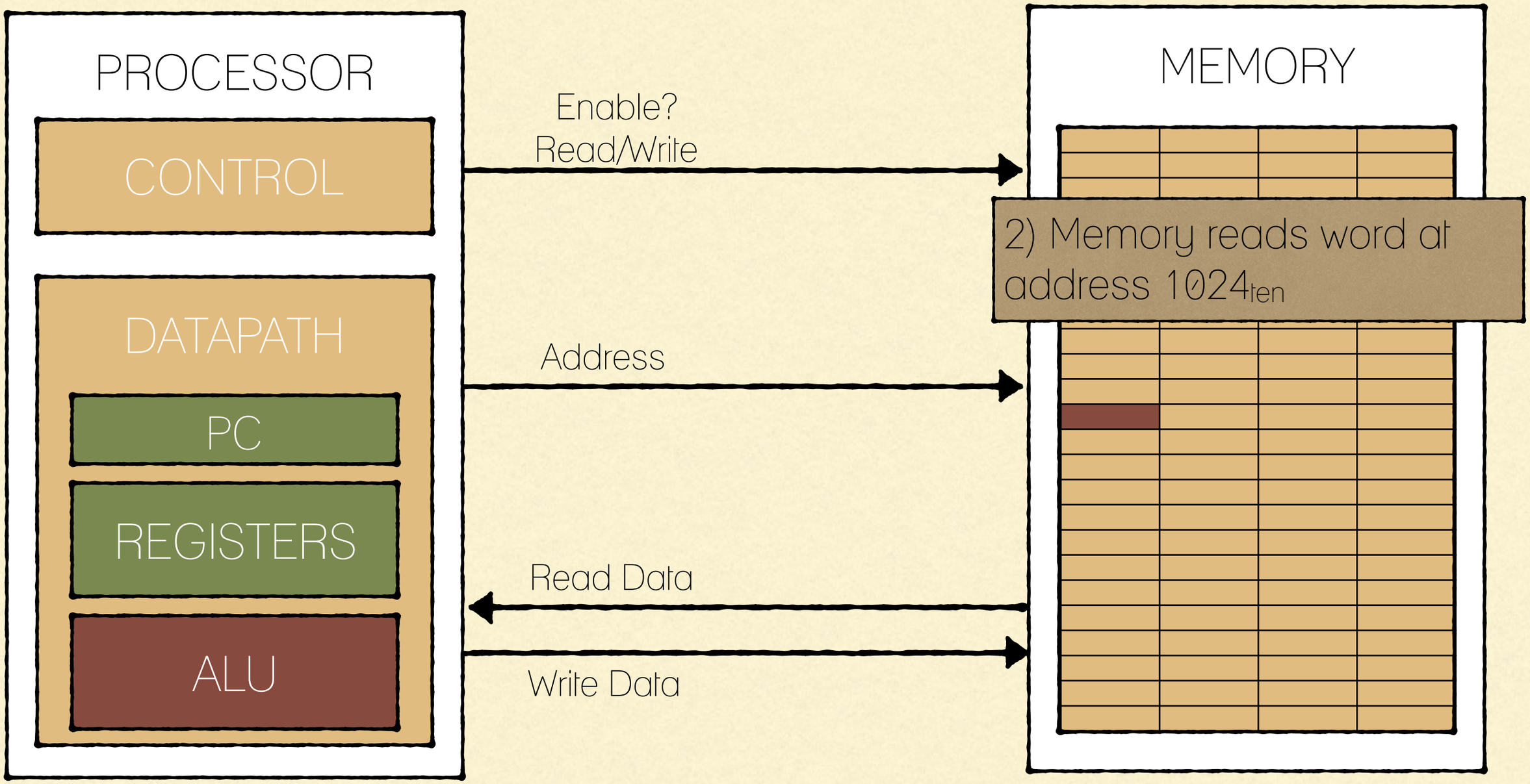
# MEMORY ACCESS WITHOUT CACHE

- Load word instruction: lw $t0, 0($t1)
- $t1 contains $1024_{ten}$, Memory[1024] = 99
- Processor issues address $1024_{ten}$ to Memory
- Memory reads word at address $1024_{ten}$(99)
- Memory sends 99 to Processor
- Processor loads 99 into register $t0

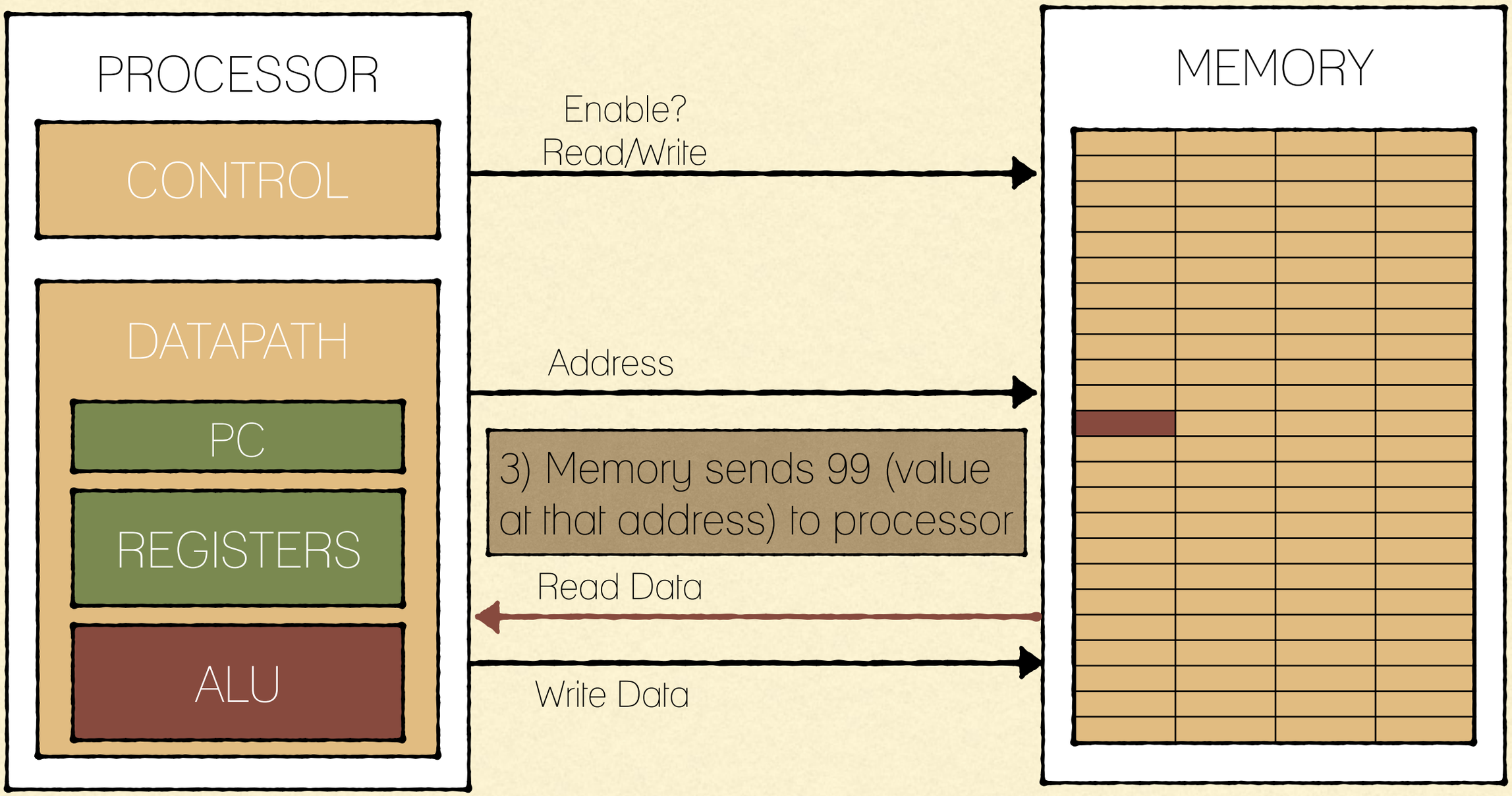
# COMPONENTS OF A COMPUTER



# COMPONENTS OF A COMPUTER



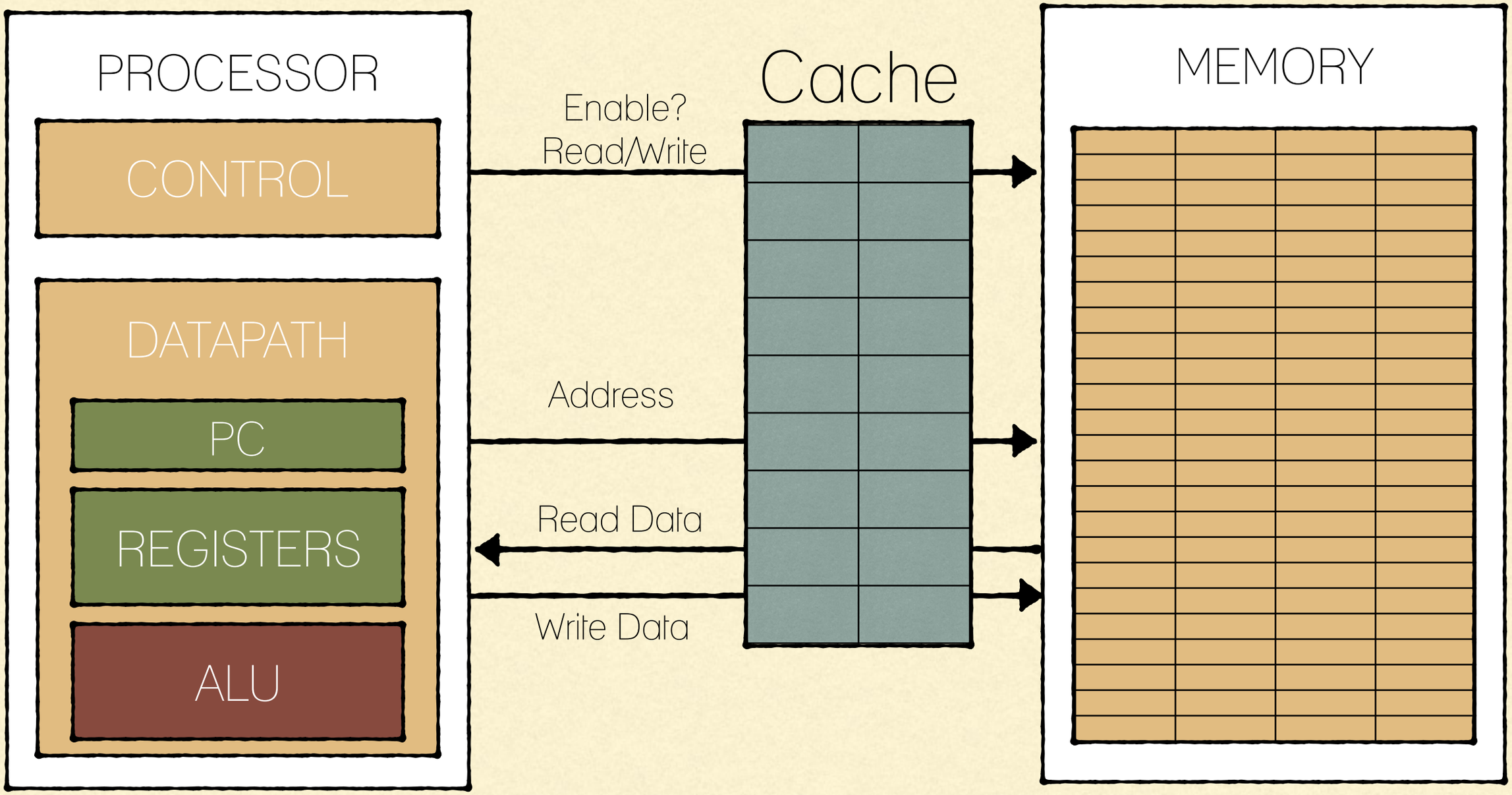
# COMPONENTS OF A COMPUTER



# MEMORY ACCESS WITH CACHE

- Load word instruction: lw $t0, 0($t1)
- $t1 contains $1024_{ten}$, Memory[1024] = 99
- With cache:
    1. Processor issues address $1024_{ten}$ to Cache
    2. Cache checks to see if it has copy of data at address $1024_{ten}$
        - If finds a match (Hit): cache reads 99, sends to processor
        - No match (Miss): cache sends address 1024 to Memory
            - Memory read 99 at address 1024
            - Memory sends 99 to Cache
            - Cache replaces word with new 99
            - Cache sends 99 to processor
    3. Processor loads 99 into register $t0

# COMPONENTS OF A COMPUTER



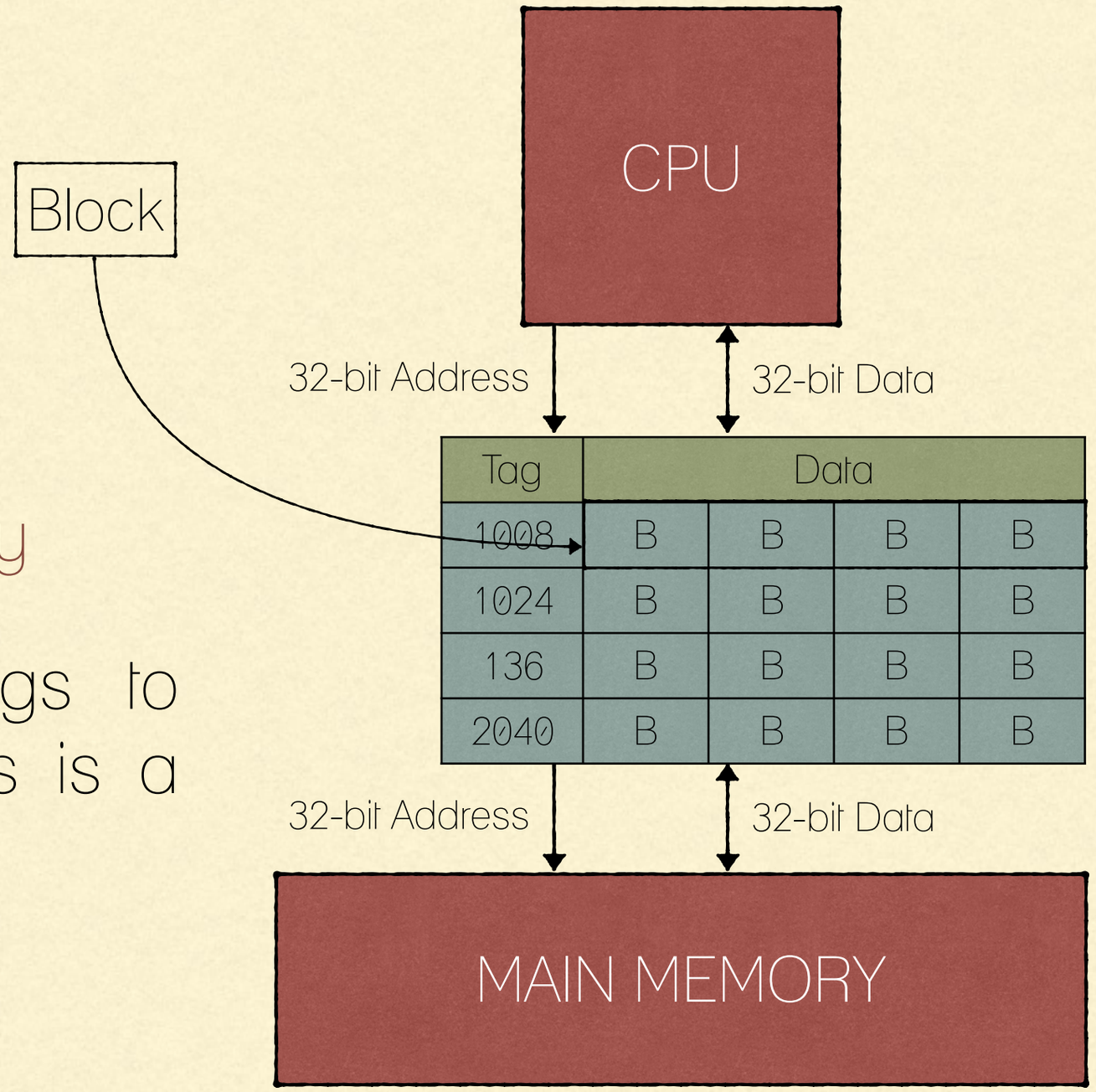
# CACHE "TAGS"

- Need way to tell if have copy of location in memory so that can decide on hit or miss
- On cache miss, put memory address of block in "tag address" of cache block
  - 1024 placed in tag next to data from memory (99)

| Tag | Data |
|---|---|
| 252 | 12 |
| 1024 | 99 |
| 131 | 7 |
| 2041 | 20 |

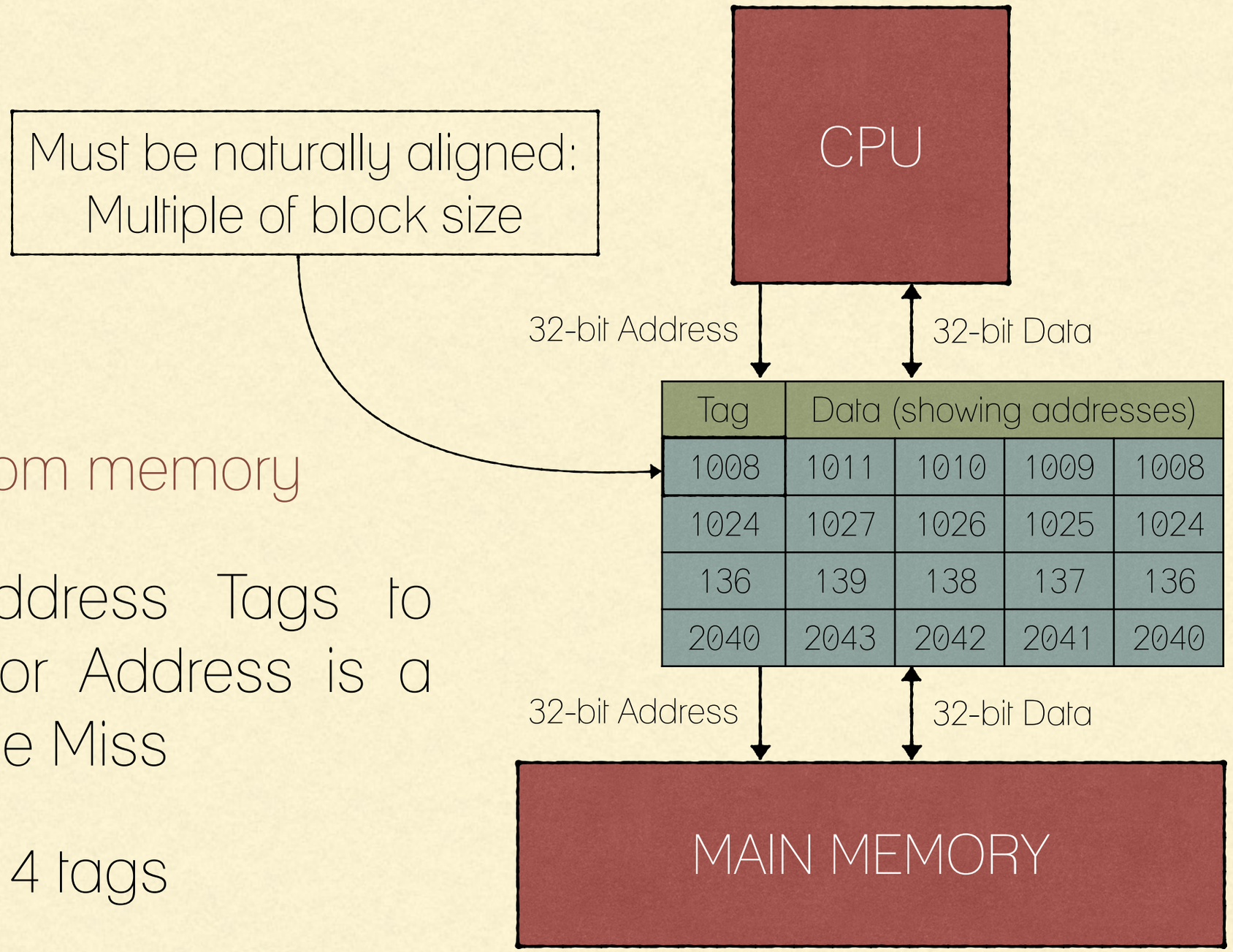
# ANATOMY OF A 16 BYTE CACHE, 4 BYTE BLOCK

- Operations:
  - Cache Hit
  - Cache Miss
    - Refill cache from memory
- Cache needs address Tags to decide if Processor Address is a Cache Hit or Cache Miss
  - Compares all 4 tags



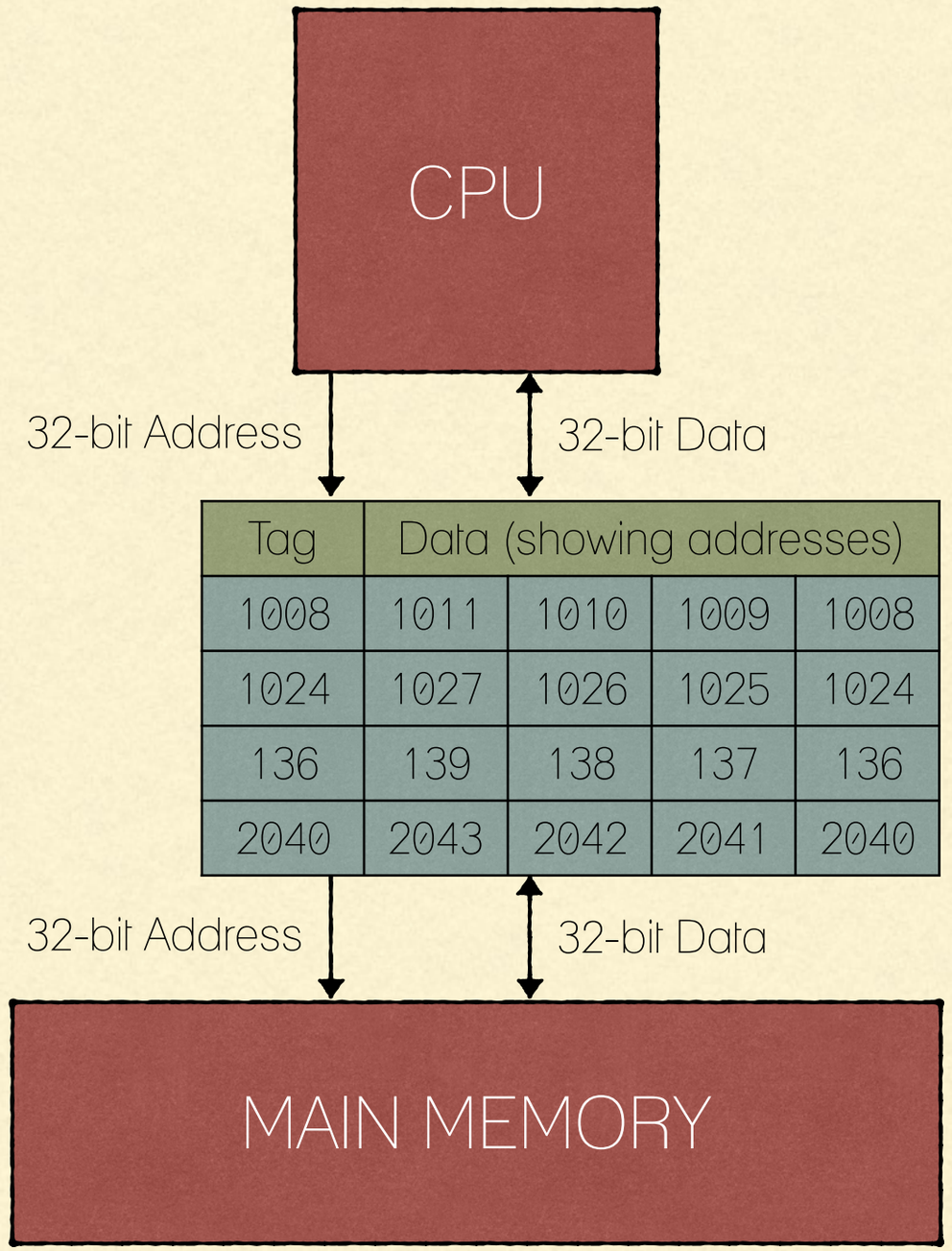
# ANATOMY OF A 16 BYTE CACHE, 4 BYTE BLOCK

- Operations:
  - Cache Hit
  - Cache Miss
    - Refill cache from memory
- Cache needs address Tags to decide if Processor Address is a Cache Hit or Cache Miss
  - Compares all 4 tags



# ANATOMY OF A 16 BYTE CACHE, 4 BYTE BLOCK

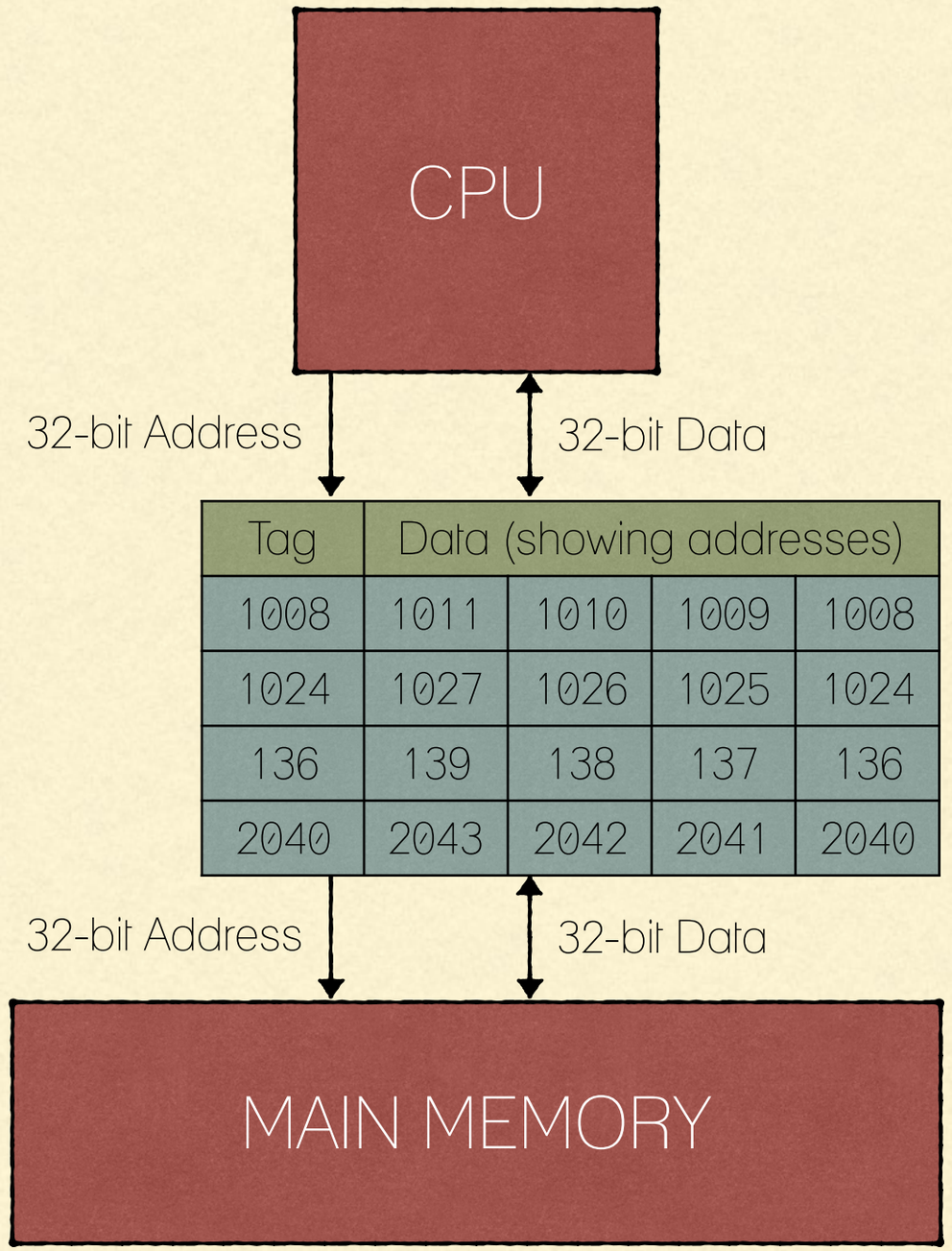
- Say we'd like to load half a word (address 1026 for example). How can we do that?



# ANATOMY OF A 16 BYTE CACHE, 4 BYTE BLOCK

- Say we'd like to load half a word (address 1026 for example). How can we do that?

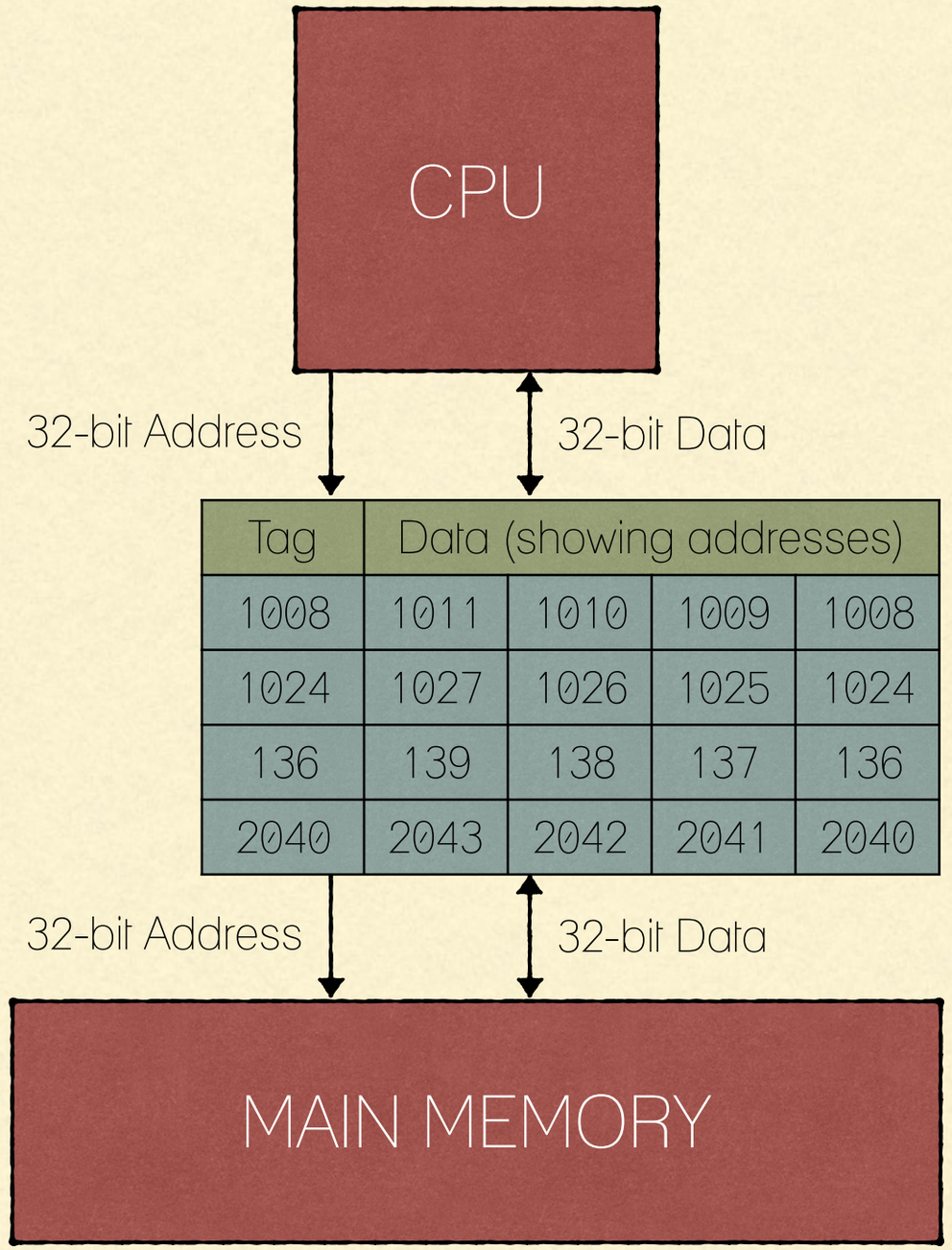
| Decimal | Binary |
|---|---|
| 1024 | 00000000000000000000010000000000 |
| 1025 | 00000000000000000000010000000001 |
| 1026 | 00000000000000000000010000000010 |
| 1027 | 00000000000000000000010000000011 |



# ANATOMY OF A 16 BYTE CACHE, 4 BYTE BLOCK

- Only compare 30 most significant bits to tag, then use 2 least significant bits to determine exact byte

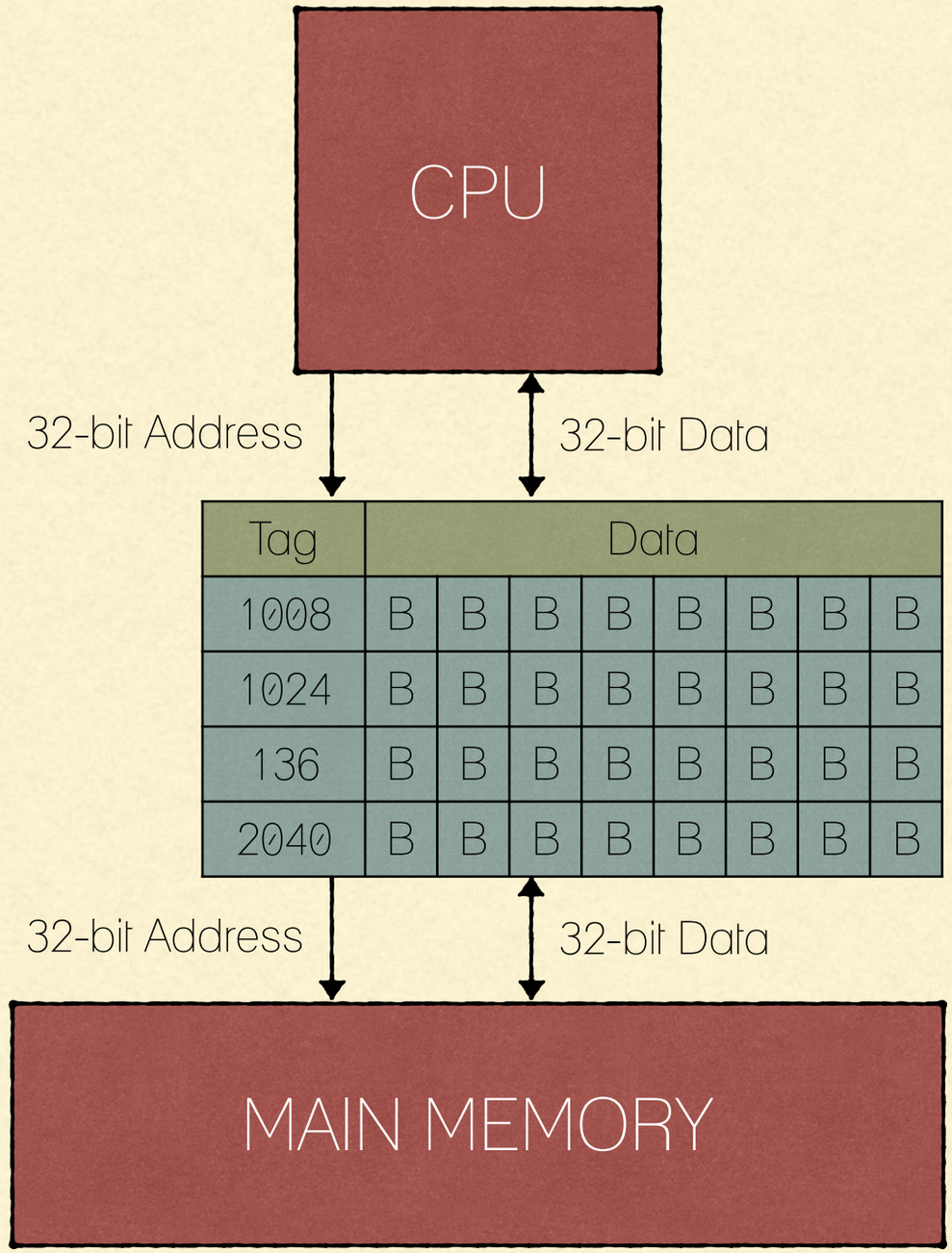
| Decimal | Binary |
|---|---|
| 1024 | 00000000000000000000010000000000 |
| 1025 | 00000000000000000000010000000001 |
| 1026 | 00000000000000000000010000000010 |
| 1027 | 00000000000000000000010000000011 |



# ANATOMY OF A 32 BYTE CACHE, 8 BYTE BLOCK

- Size of block can be more than 4 Bytes, in this case it's 8-byte long

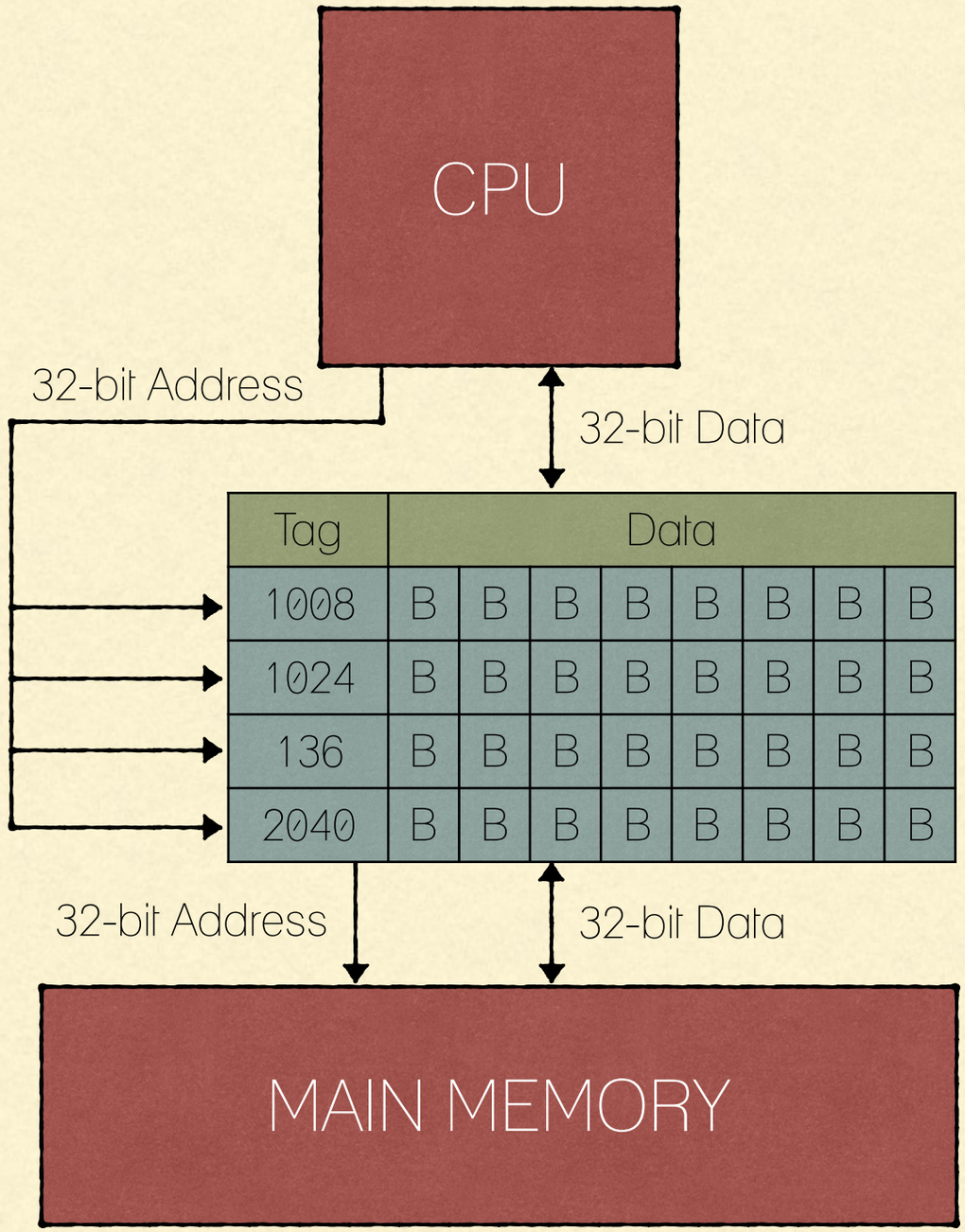
| Decimal | Binary |
|---|---|
| 1024 | 00000000000000000000010000000000 |
| 1025 | 00000000000000000000010000000001 |
| 1026 | 00000000000000000000010000000010 |
| 1027 | 00000000000000000000010000000011 |
| 1028 | 00000000000000000000010000000100 |
| 1029 | 00000000000000000000010000000101 |
| 1030 | 00000000000000000000010000000110 |
| 1031 | 00000000000000000000010000000111 |



# ANATOMY OF A 32 BYTE CACHE, 8 BYTE BLOCK

- Blocks must be aligned in multiple of 8s

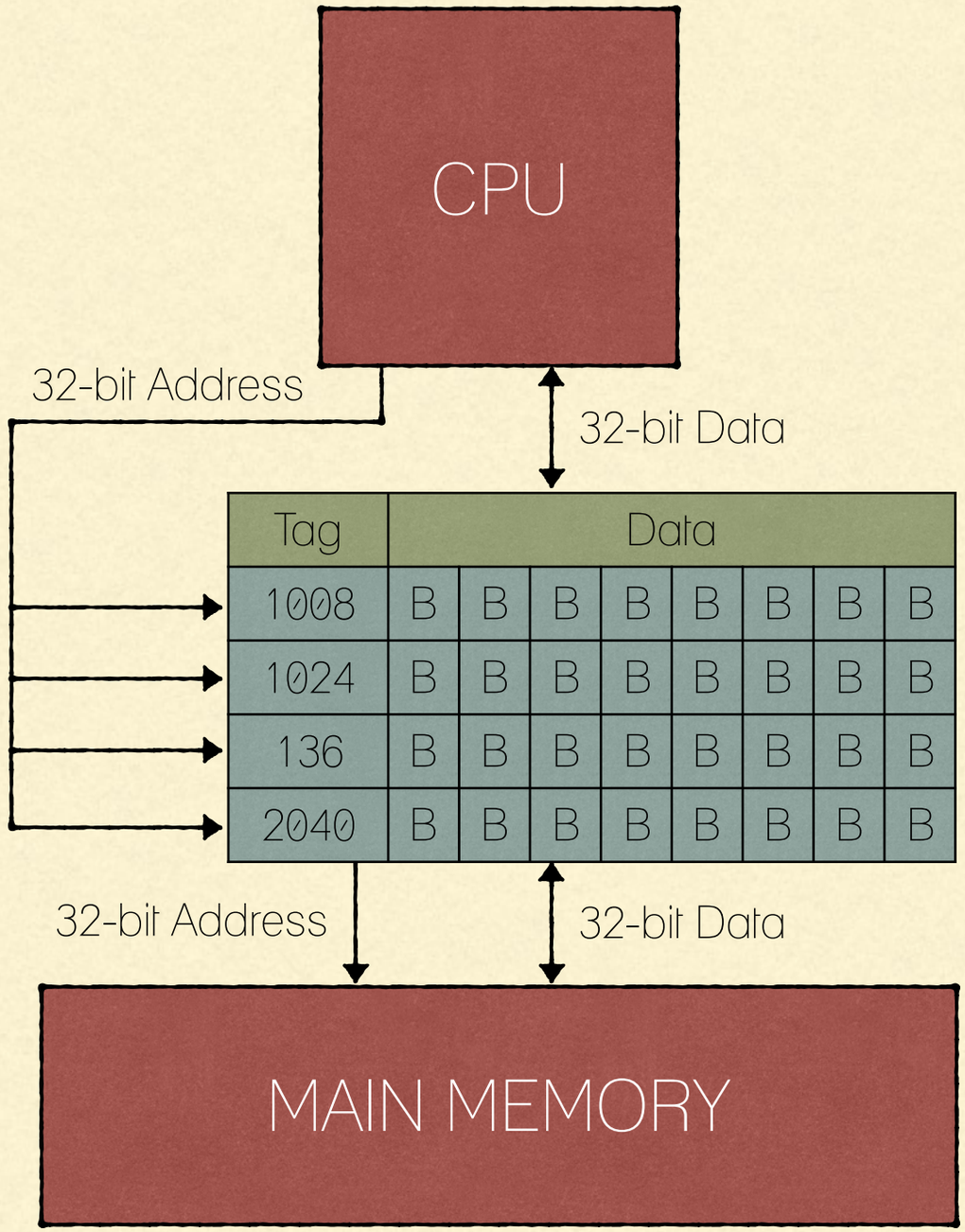
| Decimal | Binary |
|---|---|
| 1024 | 00000000000000000000010000000000 |
| 1025 | 00000000000000000000010000000001 |
| 1026 | 00000000000000000000010000000010 |
| 1027 | 00000000000000000000010000000011 |
| 1028 | 00000000000000000000010000000100 |
| 1029 | 00000000000000000000010000000101 |
| 1030 | 00000000000000000000010000000110 |
| 1031 | 00000000000000000000010000000111 |



# ANATOMY OF A 32 BYTE CACHE, 8 BYTE BLOCK

- In this case, we'll use the 29 MSB's to compare the tags, and the 3 LSB's to identify exact bit (see next slide)

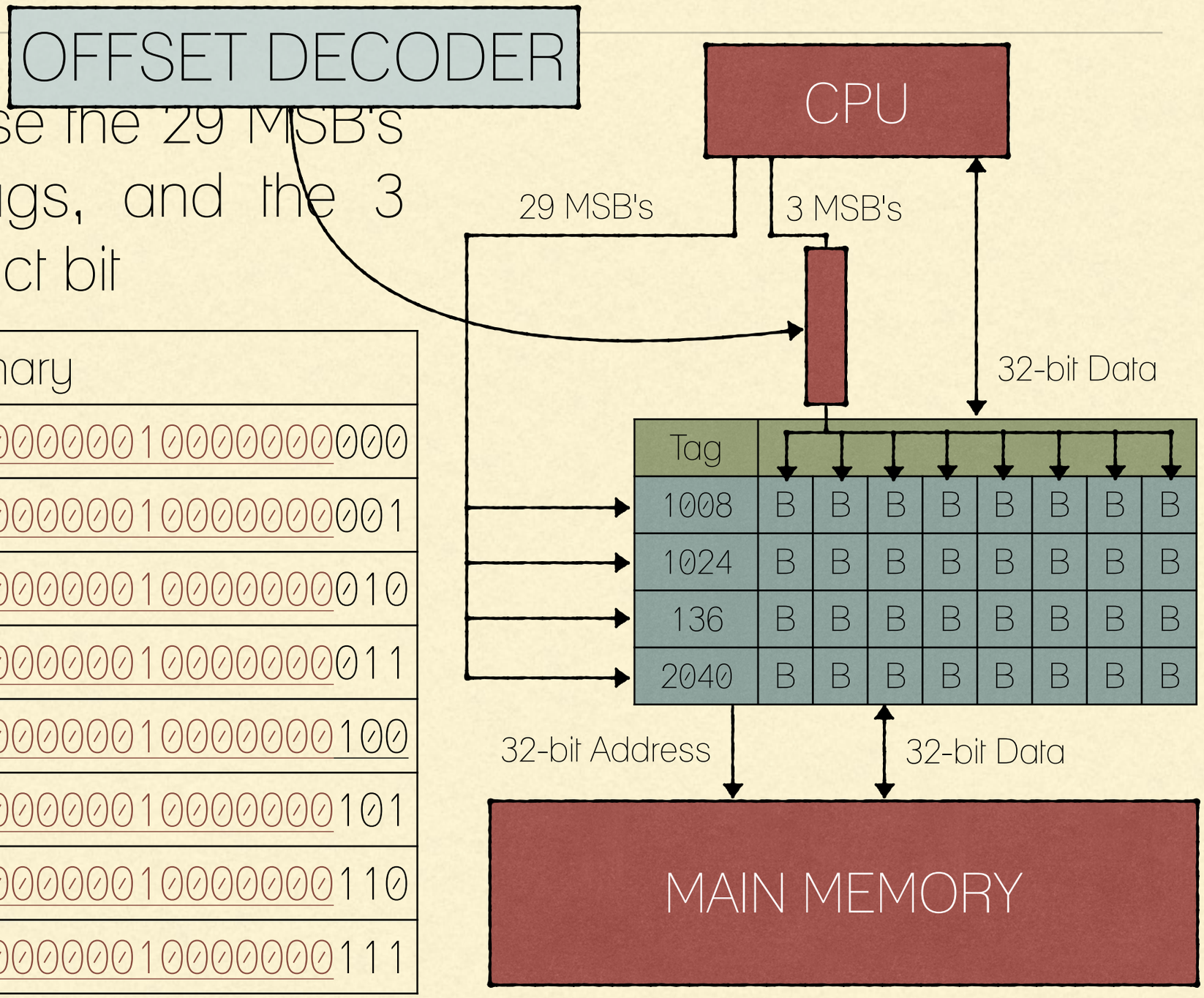
| Decimal | Binary |
|---|---|
| 1024 | 00000000000000000000001000000000 |
| 1025 | 00000000000000000000001000000001 |
| 1026 | 00000000000000000000001000000010 |
| 1027 | 00000000000000000000001000000011 |
| 1028 | 00000000000000000000001000000100 |
| 1029 | 00000000000000000000001000000101 |
| 1030 | 00000000000000000000001000000110 |
| 1031 | 00000000000000000000001000000111 |



# ANATOMY OF A 32 BYTE CACHE, 8 BYTE BLOCK

- In this case, we'll use the 29 MSB's to compare the tags, and the 3 LSB's to identify exact bit

| Decimal | Binary |
|---|---|
| 1024 | 00000000000000000000000010000000000 |
| 1025 | 00000000000000000000000010000000001 |
| 1026 | 00000000000000000000000010000000010 |
| 1027 | 00000000000000000000000010000000011 |
| 1028 | 00000000000000000000000010000000100 |
| 1029 | 00000000000000000000000010000000101 |
| 1030 | 00000000000000000000000010000000110 |
| 1031 | 00000000000000000000000010000000111 |



# PROCESSOR ADDRESS FIELDS USED BY CACHE CONTROLLER

- This approach is called “Fully Associative”
- Block offset: Byte address within block
- Tag: Remaining portion of processor address

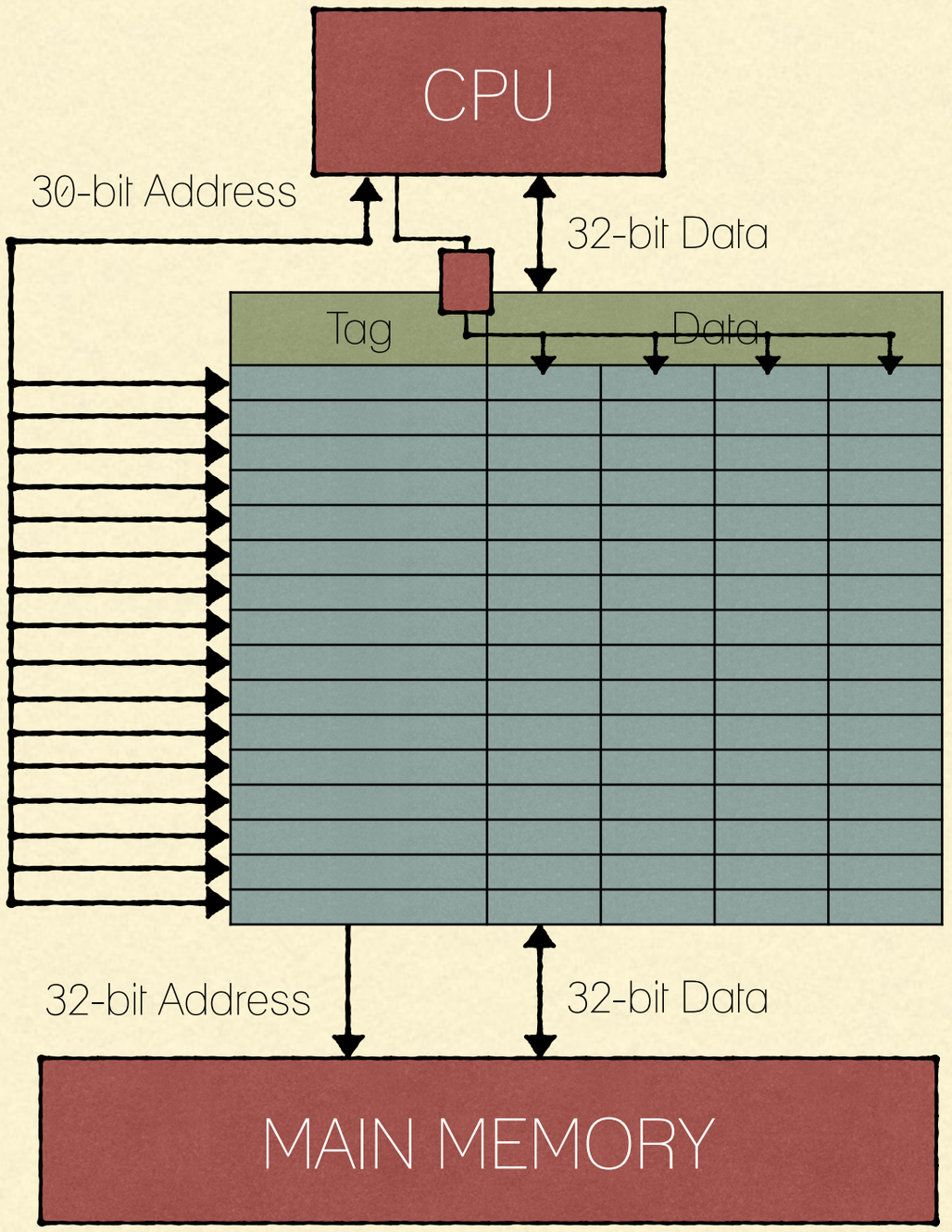

- Size of Tag = Address size - $\log_2$(number of bytes per block)

# ANATOMY OF A 64 BYTE CACHE, 4 BYTE BLOCK

- In this example, block size is 4 bytes (one word)
- The size of this cache is 16 memory blocks (or words)

| Tag (30 BITS) | Block offset (2 BITS) |
|---|---|

- Need to compare <u>every single tag</u> to the processor address
- Comparators are expensive!

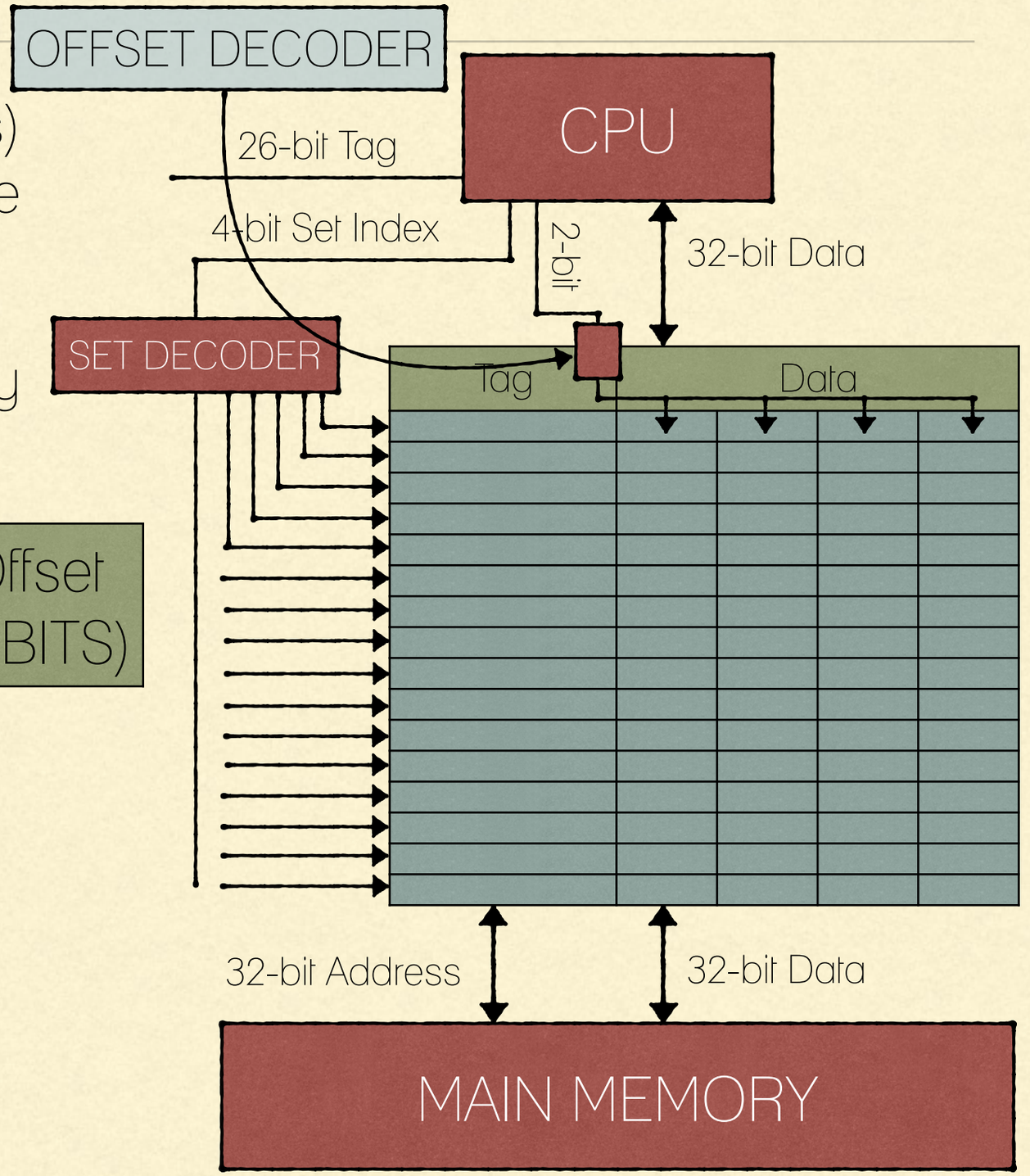


# ANATOMY OF A 64 BYTE CACHE, 4 BYTE BLOCK

- Idea: we allocate $\log_2$(num of blocks) bits (out of the 30) to direct us to the exact block in our cache.
- The size of this cache is 16 memory blocks, we need 4 bits ($\log_2(16) = 4$)

| Tag (26 BITS) | Set Index (4 BITS) | Offset (2 BITS) |
|---|---|---|

- How's that going to help?



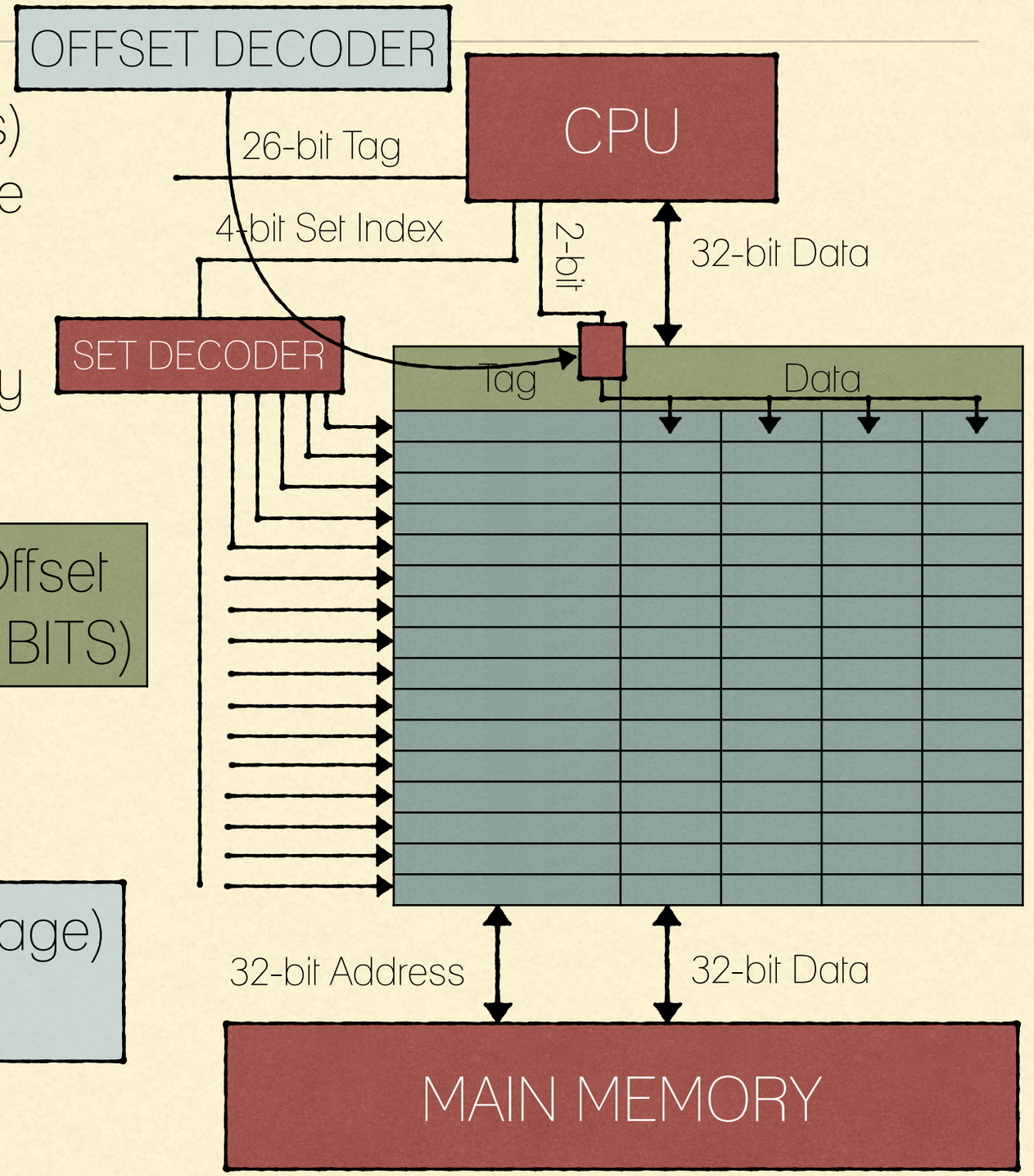
# ANATOMY OF A 64 BYTE CACHE, 4 BYTE BLOCK

- Idea: we allocate $\log_2$(num of blocks) bits (out of the 30) to direct us to the exact block in our cache.
- The size of this cache is 16 memory blocks, we need 4 bits ($\log_2(16) = 4$)

| Tag (26 BITS) | Set Index (4 BITS) | Offset (2 BITS) |
|---|---|---|

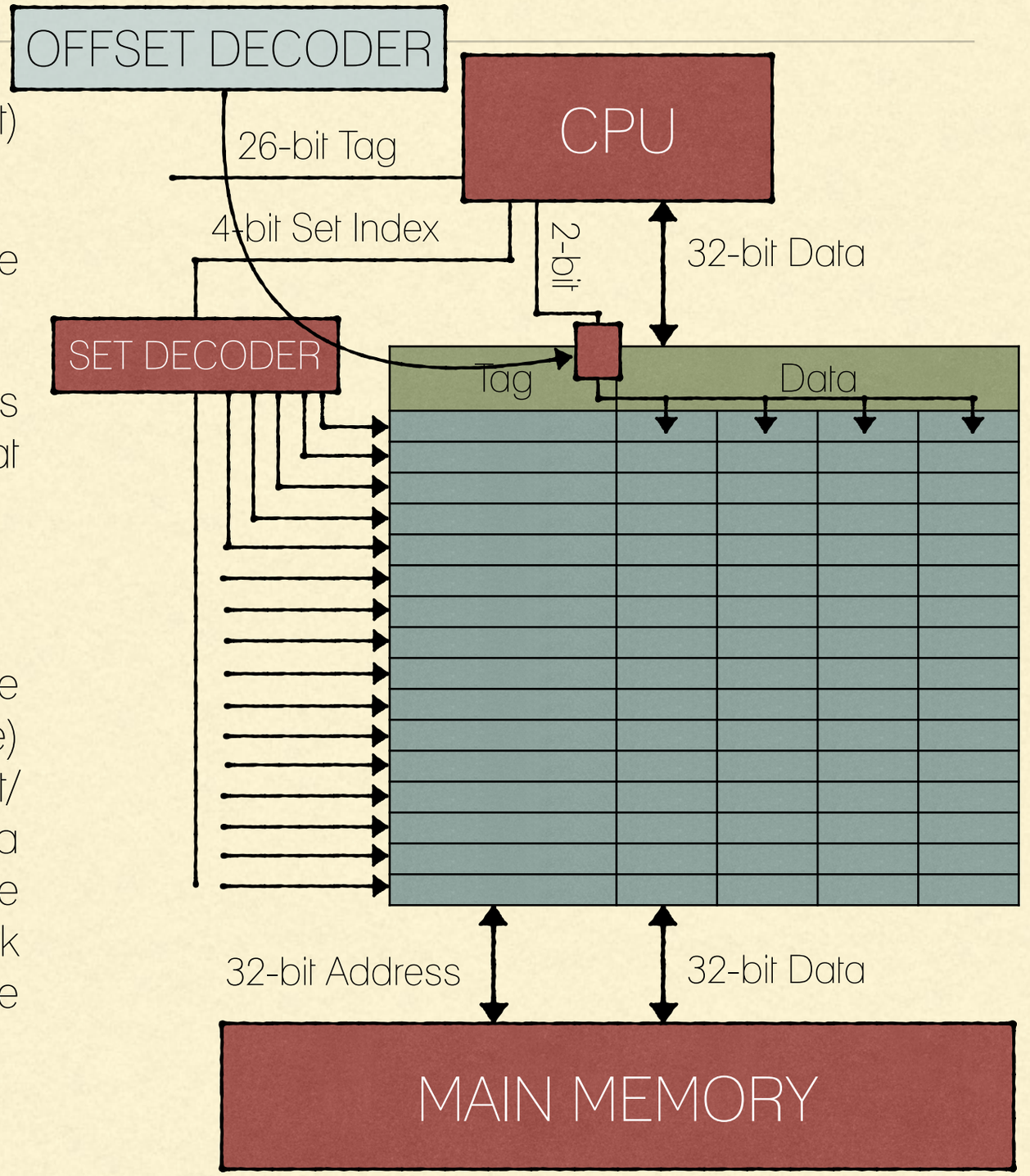
- How's that going to help?

1. Only one comparator (main advantage)
2. Smaller Tag



# ANATOMY OF A 64 BYTE CACHE, 4 BYTE BLOCK

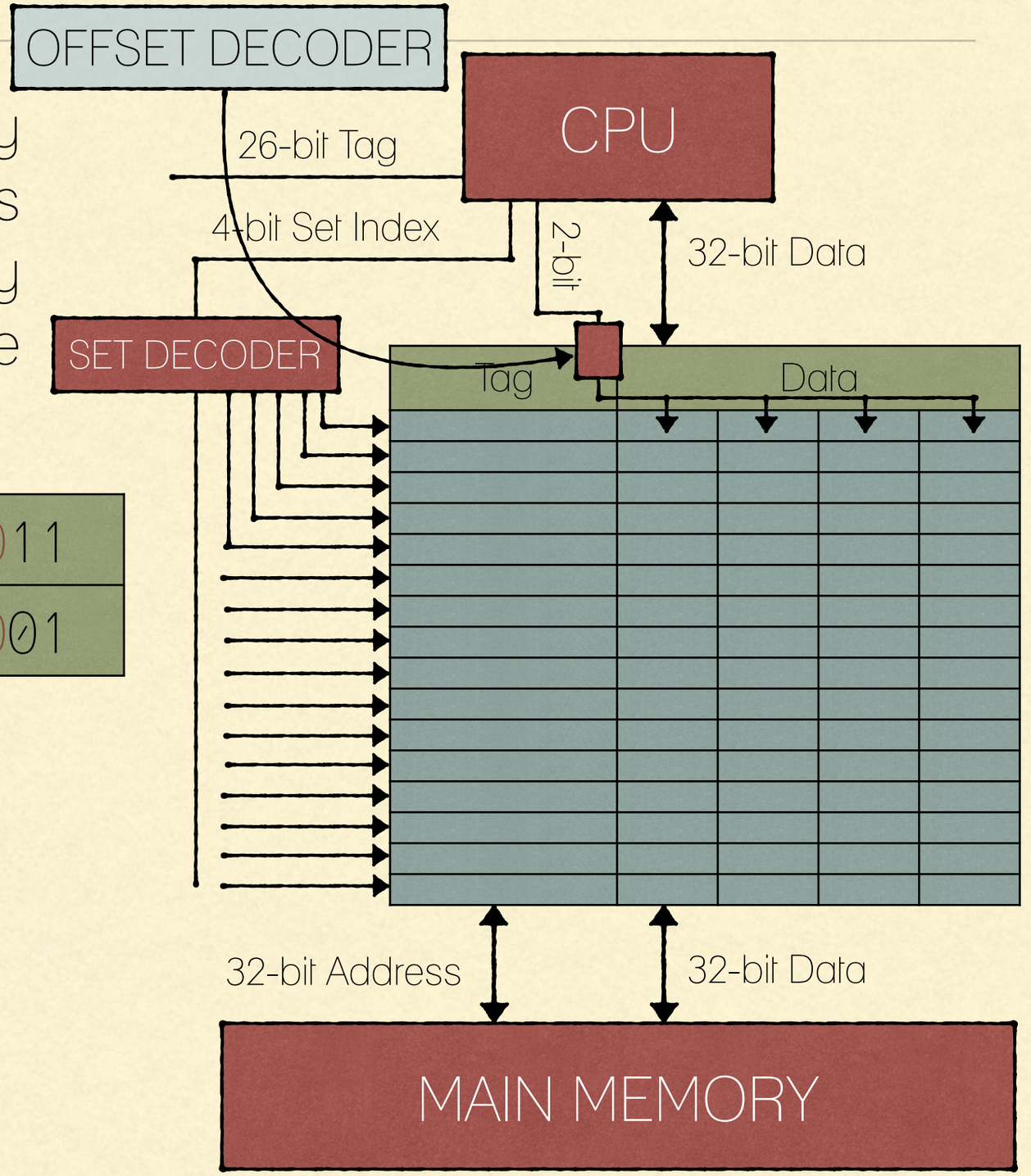
- Say the CPU requests the instruction (or data-point) at/for some address:
  - The set index will be decoded and only one row/block will activate
  - We then compare the tag of the address requested by the CPU, with the tag stored at that row/block.
    - If they're equal that means it's a "match"
    - If they're not equal, that means the Cache needs to go to the RAM (or 2nd level cache) and request the instruction (or data-point) at/for that address and sends it to the CPU (a copy of that instruction/data-point will be stored in the cache in that same row/block that corresponds to the set index of the address)



# ANATOMY OF A 64 BYTE CACHE, 4 BYTE BLOCK

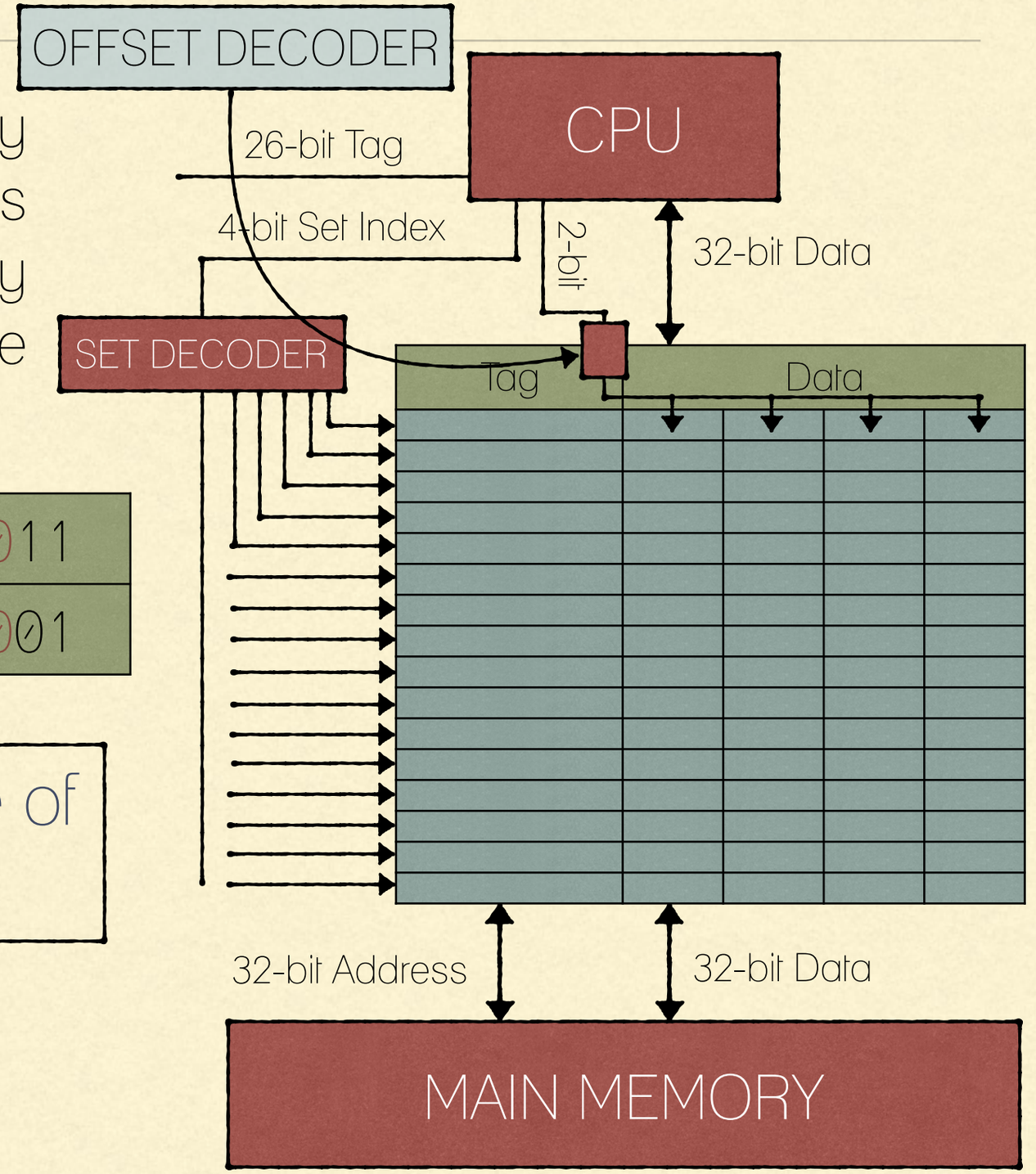
- Since the set index will activate only one row/block for a given value, this means that the following two binary addresses cannot be stored in the cache at the same time

| |
|---|
| 10010010010001001000100010001000010010011 |
| 11110010111000001000111010001000010010001 |



# ANATOMY OF A 64 BYTE CACHE, 4 BYTE BLOCK

- Since the set index will activate only one row/block for a given value, this means that the following two binary addresses cannot be stored in the cache at the same time




| |
|---|
| 10010010010001001000100010001000010**0**1**0**11 |
| 11110010111000001000111010**0**01**0**01 |

This is one main disadvantage of this “Direct-Mapped” Design

# ANOTHER EXAMPLE: ANATOMY OF A 64 BYTE MAIN MEMORY, 4 BYTE BLOCK, 16 BYTE CACHE

| Tag (2 BITS) | Set Index (2 BITS) | Block offset (2 BITS) |
|---|---|---|

| Cache Index | Tag | DATA |
|---|---|---|
| 00 | 00 | 4 Bytes |
| 01 | 00 | 4 Bytes |
| 10 | 00 | 4 Bytes |
| 11 | 00 | 4 Bytes |

Main Memory

| | |
|---|---|
| 4 Bytes | 0000xx |
| 4 Bytes | 0001xx |
| 4 Bytes | 0010xx |
| 4 Bytes | 0011xx |
| 4 Bytes | 0100xx |
| 4 Bytes | 0101xx |
| 4 Bytes | 0110xx |
| 4 Bytes | 0111xx |
| 4 Bytes | 1000xx |
| 4 Bytes | 1001xx |
| 4 Bytes | 1010xx |
| 4 Bytes | 1011xx |
| 4 Bytes | 1100xx |
| 4 Bytes | 1101xx |
| 4 Bytes | 1110xx |
| 4 Bytes | 1111xx |

# ANOTHER EXAMPLE: ANATOMY OF A 64 BYTE MAIN MEMORY, 4 BYTE BLOCK, 16 BYTE CACHE

| Tag (2 BITS) | Set Index (2 BITS) | Block offset (2 BITS) |
|---|---|---|

CPU requests this address: 00 10 00

| Cache Index | Tag | DATA |
|---|---|---|
| 00 | 00 | 4 Bytes |
| 01 | 00 | 4 Bytes |
| 10 | 00 | 4 Bytes |
| 11 | 00 | 4 Bytes |

Main Memory

| | |
|---|---|
| 4 Bytes | 0000xx |
| 4 Bytes | 0001xx |
| 4 Bytes | 0010xx |
| 4 Bytes | 0011xx |
| 4 Bytes | 0100xx |
| 4 Bytes | 0101xx |
| 4 Bytes | 0110xx |
| 4 Bytes | 0111xx |
| 4 Bytes | 1000xx |
| 4 Bytes | 1001xx |
| 4 Bytes | 1010xx |
| 4 Bytes | 1011xx |
| 4 Bytes | 1100xx |
| 4 Bytes | 1101xx |
| 4 Bytes | 1110xx |
| 4 Bytes | 1111xx |

# ANOTHER EXAMPLE: ANATOMY OF A 64 BYTE MAIN MEMORY, 4 BYTE BLOCK, 16 BYTE CACHE

| Tag (2 BITS) | Set Index (2 BITS) | Block offset (2 BITS) |
|---|---|---|

CPU requests this address: 00 10 00

| Cache Index | Tag | DATA |
|---|---|---|
| 00 | 00 | 4 Bytes |
| 01 | 00 | 4 Bytes |
| 10 | 00 | 4 Bytes |
| 11 | 00 | 4 Bytes |

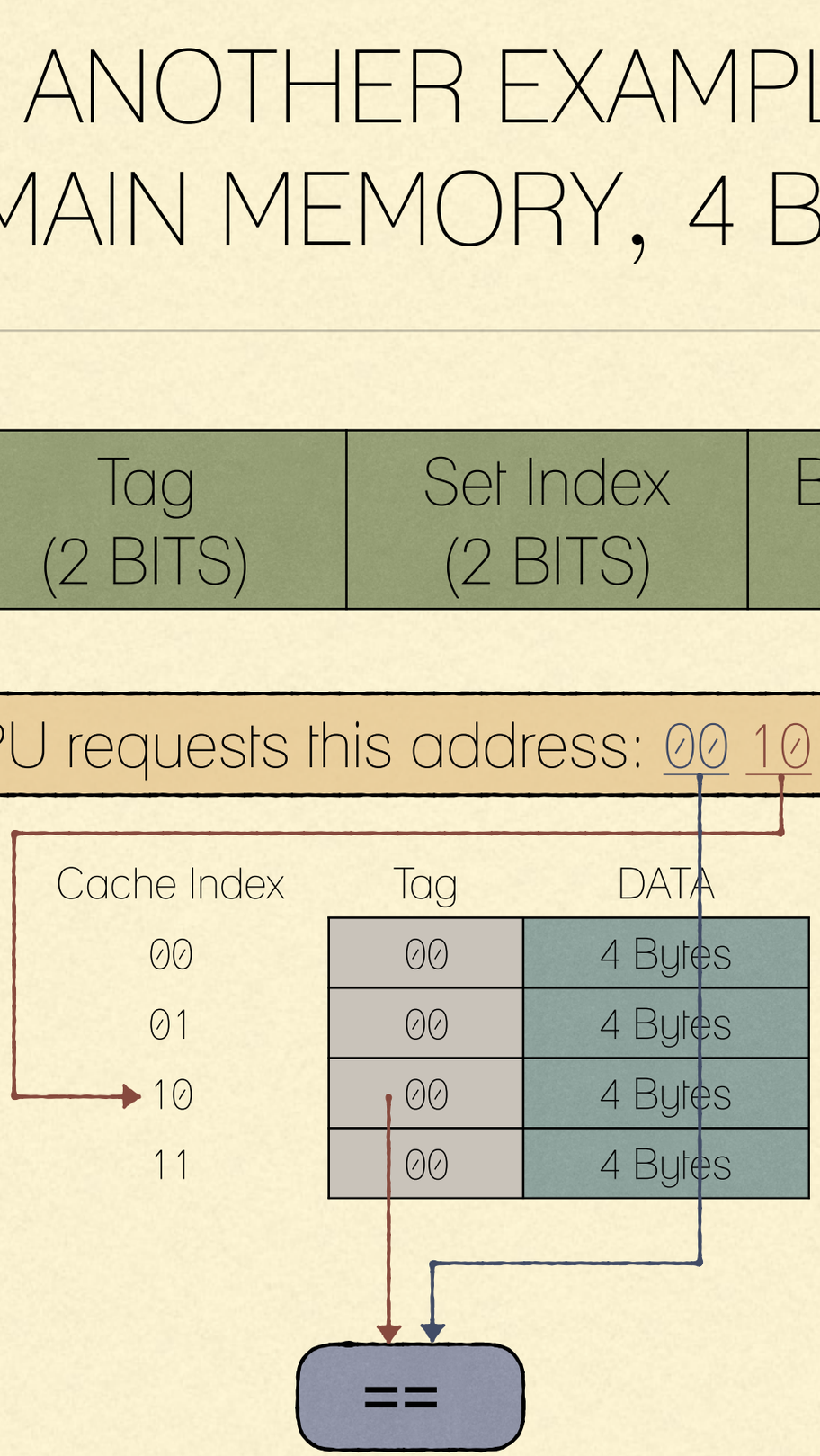
Main Memory

| | |
|---|---|
| 4 Bytes | 0000xx |
| 4 Bytes | 0001xx |
| 4 Bytes | 0010xx |
| 4 Bytes | 0011xx |
| 4 Bytes | 0100xx |
| 4 Bytes | 0101xx |
| 4 Bytes | 0110xx |
| 4 Bytes | 0111xx |
| 4 Bytes | 1000xx |
| 4 Bytes | 1001xx |
| 4 Bytes | 1010xx |
| 4 Bytes | 1011xx |
| 4 Bytes | 1100xx |
| 4 Bytes | 1101xx |
| 4 Bytes | 1110xx |
| 4 Bytes | 1111xx |

# ANOTHER EXAMPLE: ANATOMY OF A 64 BYTE MAIN MEMORY, 4 BYTE BLOCK, 16 BYTE CACHE

| Tag (2 BITS) | Set Index (2 BITS) | Block offset (2 BITS) |
|---|---|---|

CPU requests this address: 00 10 00



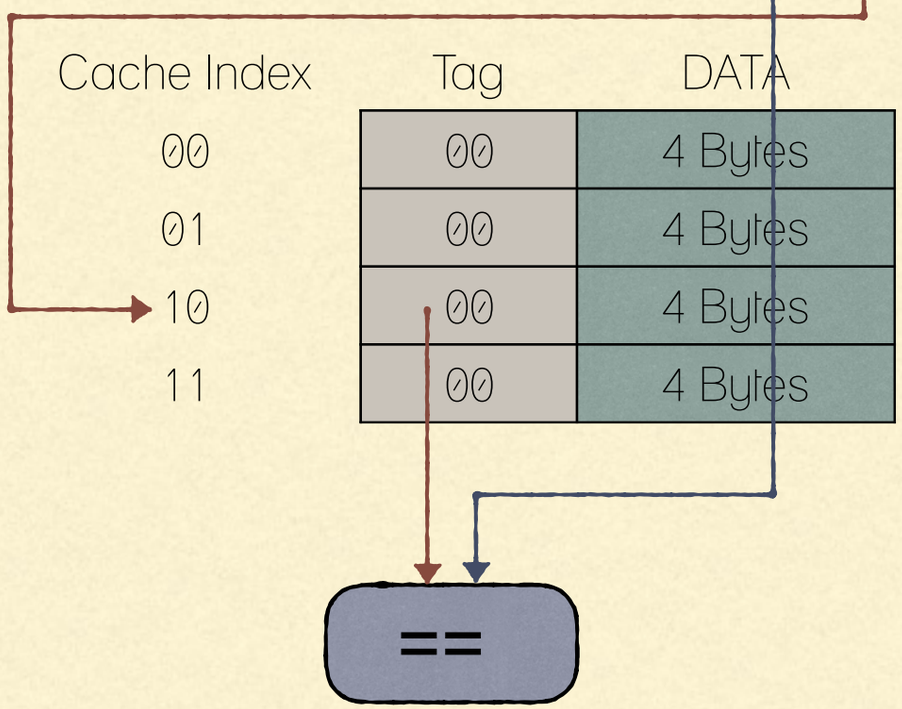

Main Memory

| | |
|---|---|
| 4 Bytes | 0000xx |
| 4 Bytes | 0001xx |
| 4 Bytes | 0010xx |
| 4 Bytes | 0011xx |
| 4 Bytes | 0100xx |
| 4 Bytes | 0101xx |
| 4 Bytes | 0110xx |
| 4 Bytes | 0111xx |
| 4 Bytes | 1000xx |
| 4 Bytes | 1001xx |
| 4 Bytes | 1010xx |
| 4 Bytes | 1011xx |
| 4 Bytes | 1100xx |
| 4 Bytes | 1101xx |
| 4 Bytes | 1110xx |
| 4 Bytes | 1111xx |

# ANOTHER EXAMPLE: ANATOMY OF A 64 BYTE MAIN MEMORY, 4 BYTE BLOCK, 16 BYTE CACHE

Problem: Tag bits will always have a value, what if Tag value was 00 when the computer starts, which matches the tag value of the address of instruction/data-point we're fetching in this example? This is a problem because comparing 00 to the tag of the address will yield in a "match", even though the content of the cache is "garbage" (since we haven't loaded anything from the RAM to the cache

CPU requests this address: 00 10 00




| | |
|---|---|
| 4 Bytes | 0011xx |
| 4 Bytes | 0100xx |
| 4 Bytes | 0101xx |
| 4 Bytes | 0110xx |
| 4 Bytes | 0111xx |
| 4 Bytes | 1000xx |
| 4 Bytes | 1001xx |
| 4 Bytes | 1010xx |
| 4 Bytes | 1011xx |
| 4 Bytes | 1100xx |
| 4 Bytes | 1101xx |
| 4 Bytes | 1110xx |
| 4 Bytes | 1111xx |

# ANOTHER EXAMPLE: ANATOMY OF A 64 BYTE MAIN MEMORY, 4 BYTE BLOCK, 16 BYTE CACHE

(2

To solve this problem, we introduce an additional bit that will be initialized to 0 when the computer starts. When we fill the cache with a block from the main memory (or 2nd level cache), the value of the 'valid bit' at that block changes to 1.

CPU requests this address: 00 10 00

| Cache Index | Valid Bit | Tag | DATA |
|---|---|---|---|
| 00 | 0 | 00 | 4 Bytes |
| 01 | 0 | 00 | 4 Bytes |
| 10 | 0 | 00 | 4 Bytes |
| 11 | 0 | 00 | 4 Bytes |

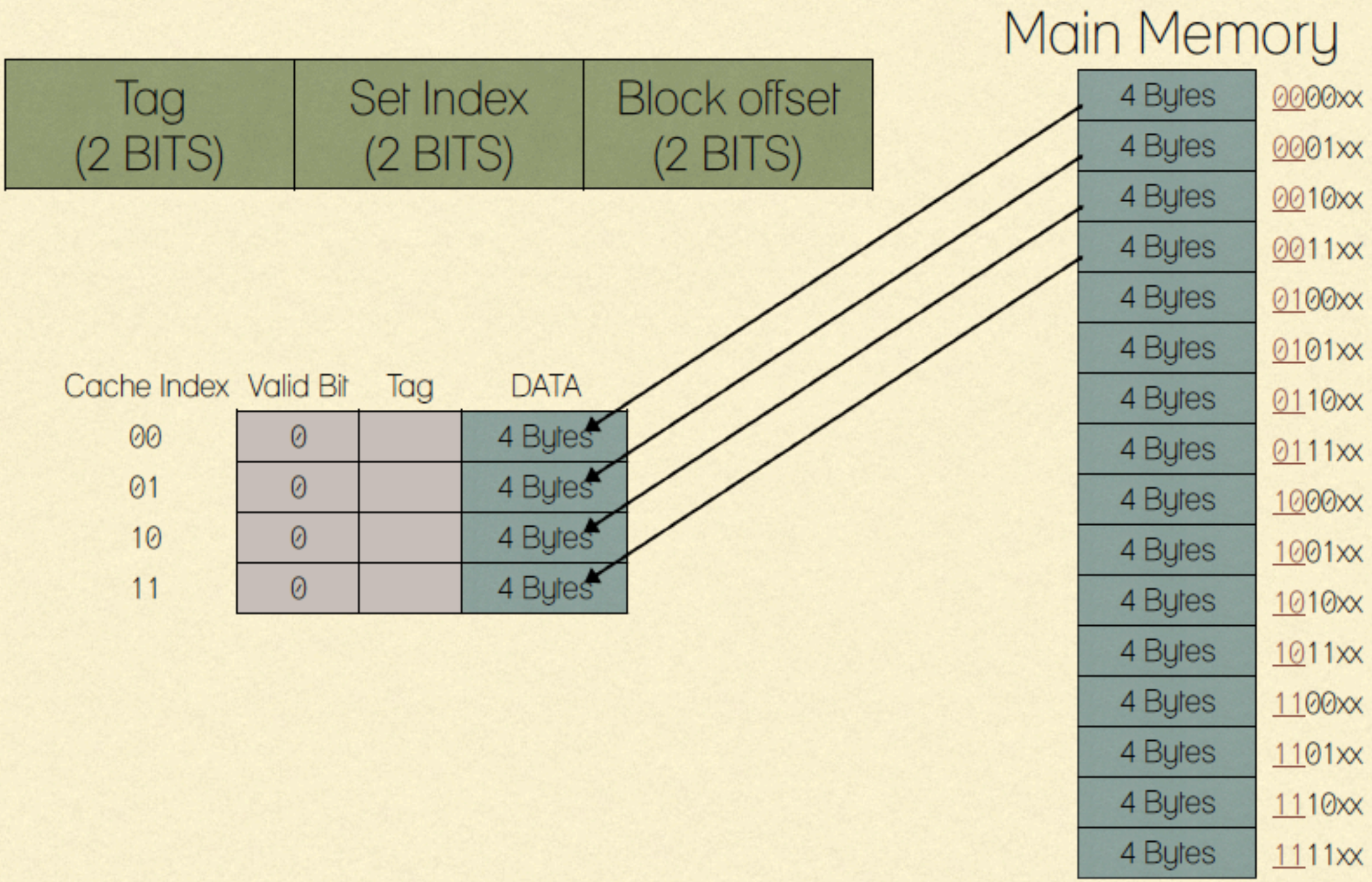
==

When output value is 0, it means we have to go to memory (or next level cache); when output value is 1, that mens it's a hit/match

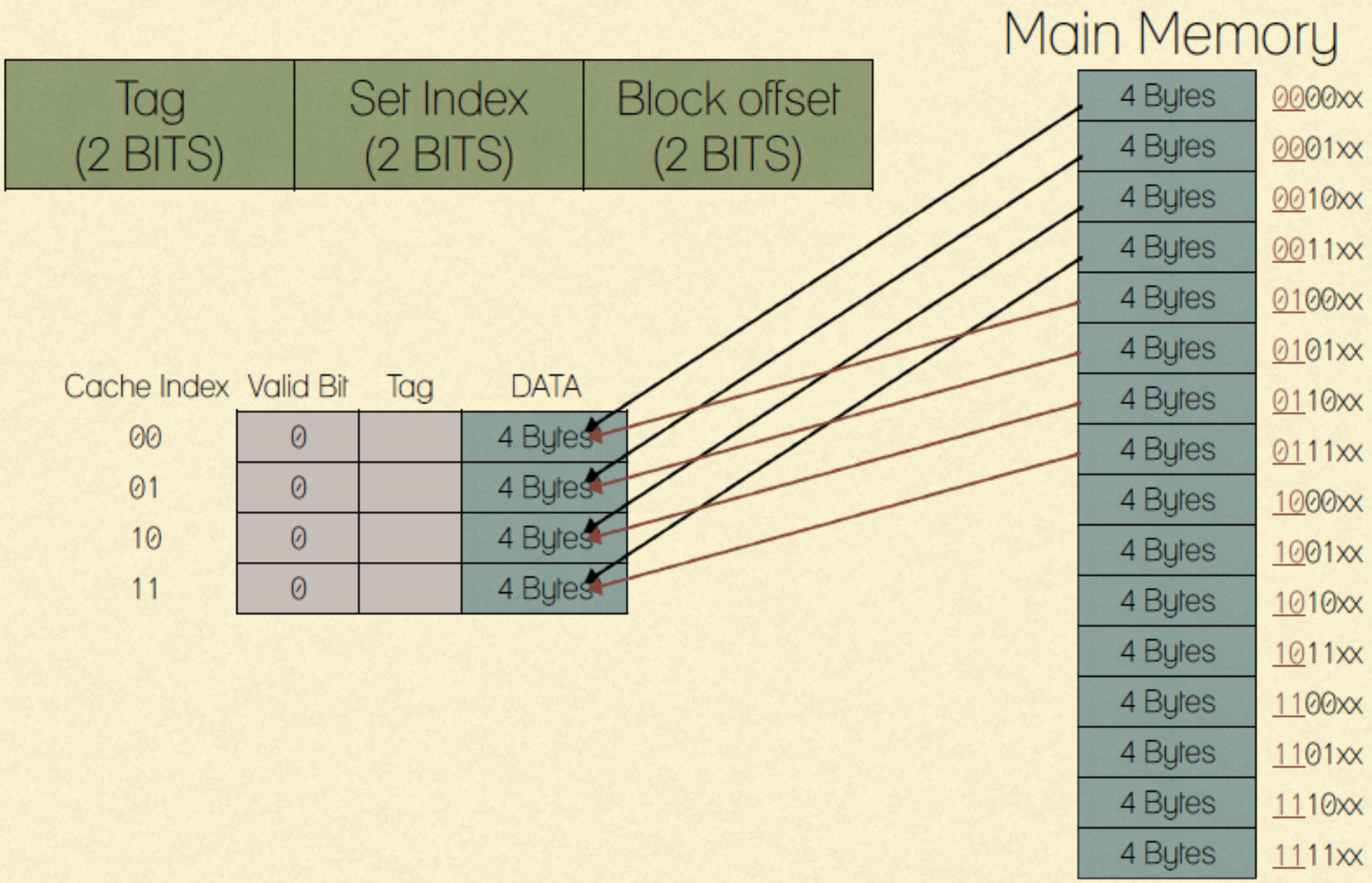
Main Memory

| | |
|---|---|
| 4 Bytes | 0011xx |
| 4 Bytes | 0100xx |
| 4 Bytes | 0101xx |
| 4 Bytes | 0110xx |
| 4 Bytes | 0111xx |
| 4 Bytes | 1000xx |
| 4 Bytes | 1001xx |
| 4 Bytes | 1010xx |
| 4 Bytes | 1011xx |
| 4 Bytes | 1100xx |
| 4 Bytes | 1101xx |
| 4 Bytes | 1110xx |
| 4 Bytes | 1111xx |

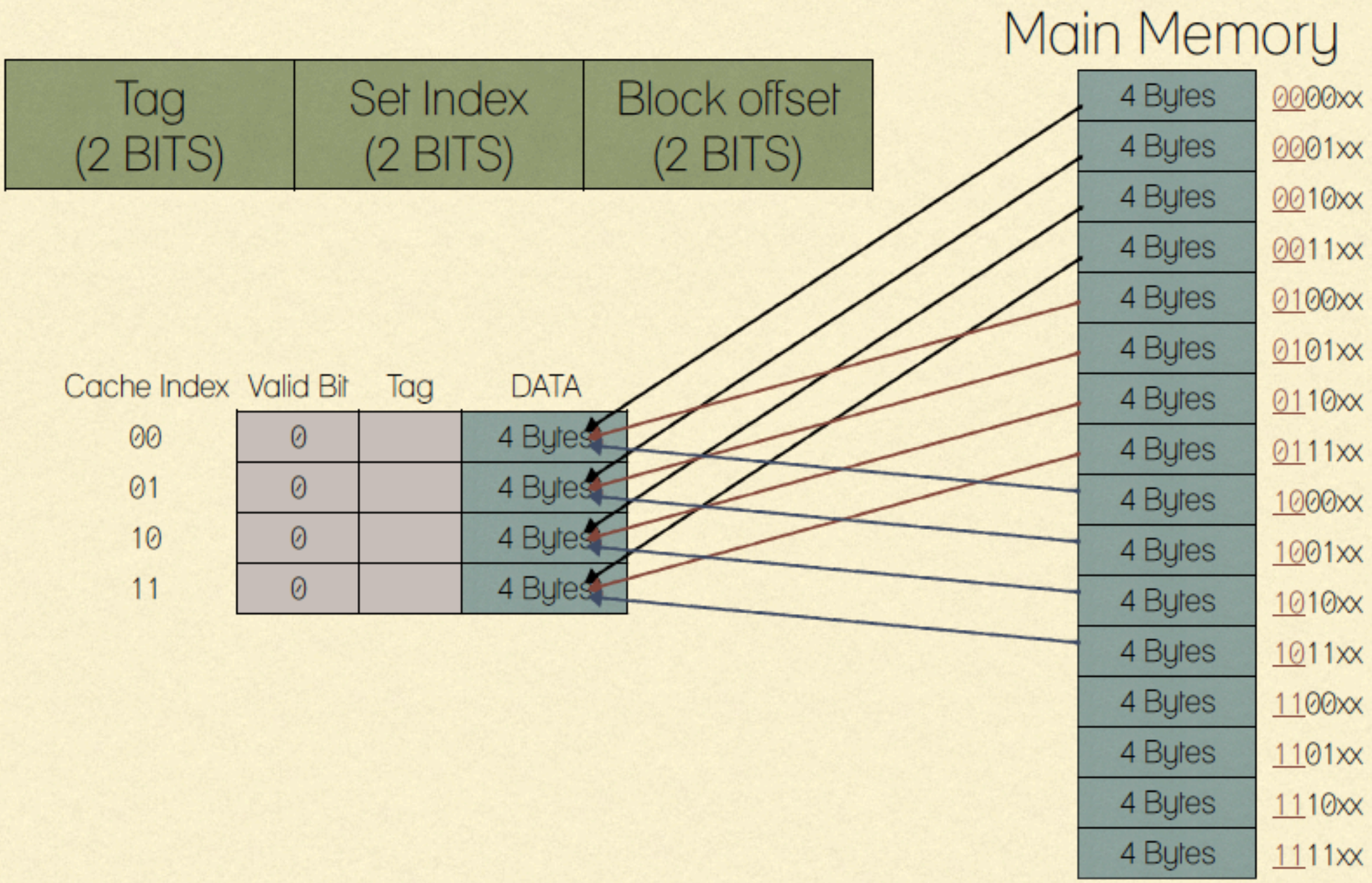
# ANOTHER EXAMPLE: ANATOMY OF A 64 BYTE MAIN MEMORY, 4 BYTE BLOCK, 16 BYTE CACHE



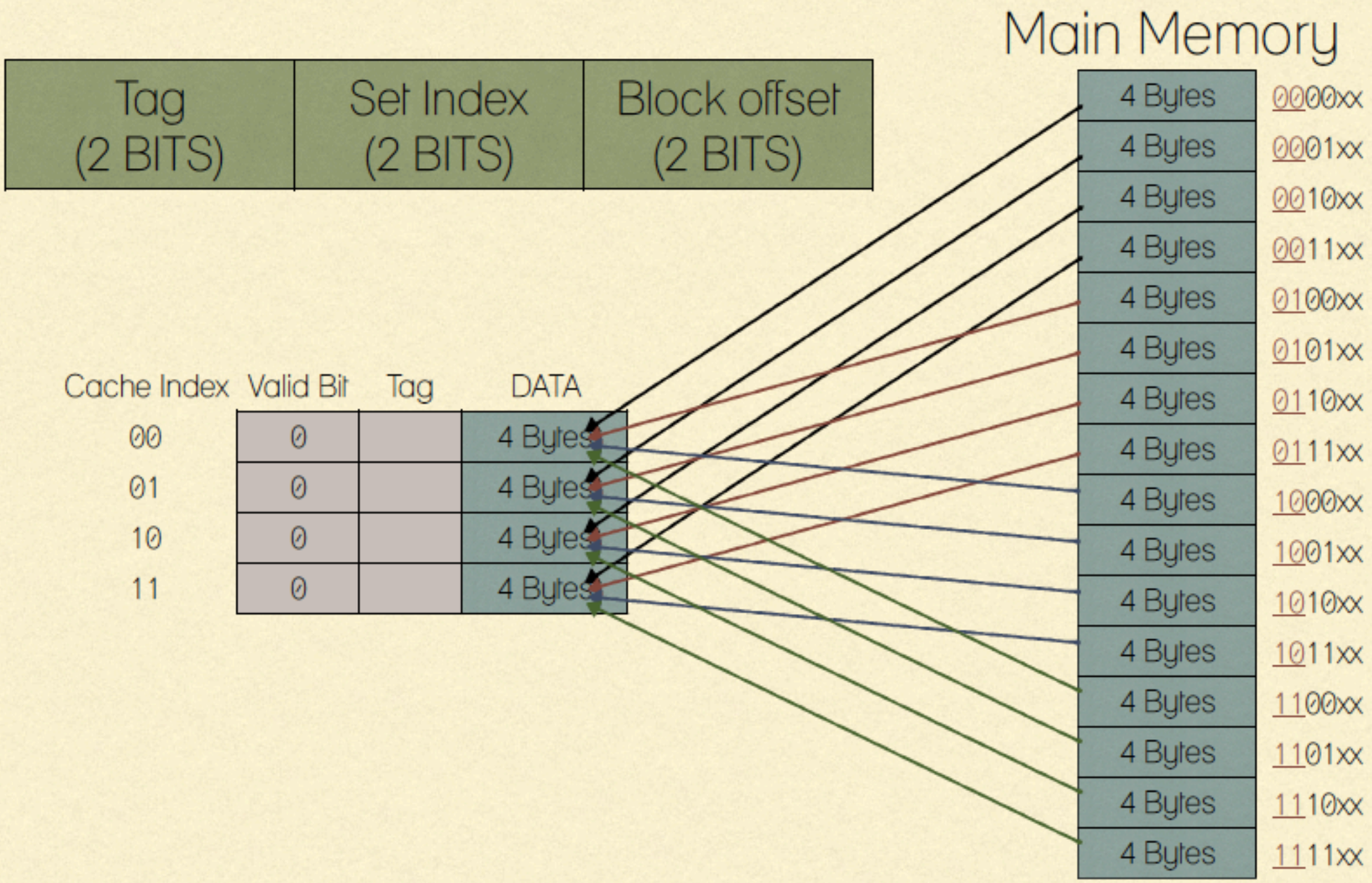
# ANOTHER EXAMPLE: ANATOMY OF A 64 BYTE MAIN MEMORY, 4 BYTE BLOCK, 16 BYTE CACHE



# ANOTHER EXAMPLE: ANATOMY OF A 64 BYTE MAIN MEMORY, 4 BYTE BLOCK, 16 BYTE CACHE



# ANOTHER EXAMPLE: ANATOMY OF A 64 BYTE MAIN MEMORY, 4 BYTE BLOCK, 16 BYTE CACHE



# QUESTION: DOES THE SET INDEX FIELD NEED TO BE NEXT TO THE BLOCK OFFSET?

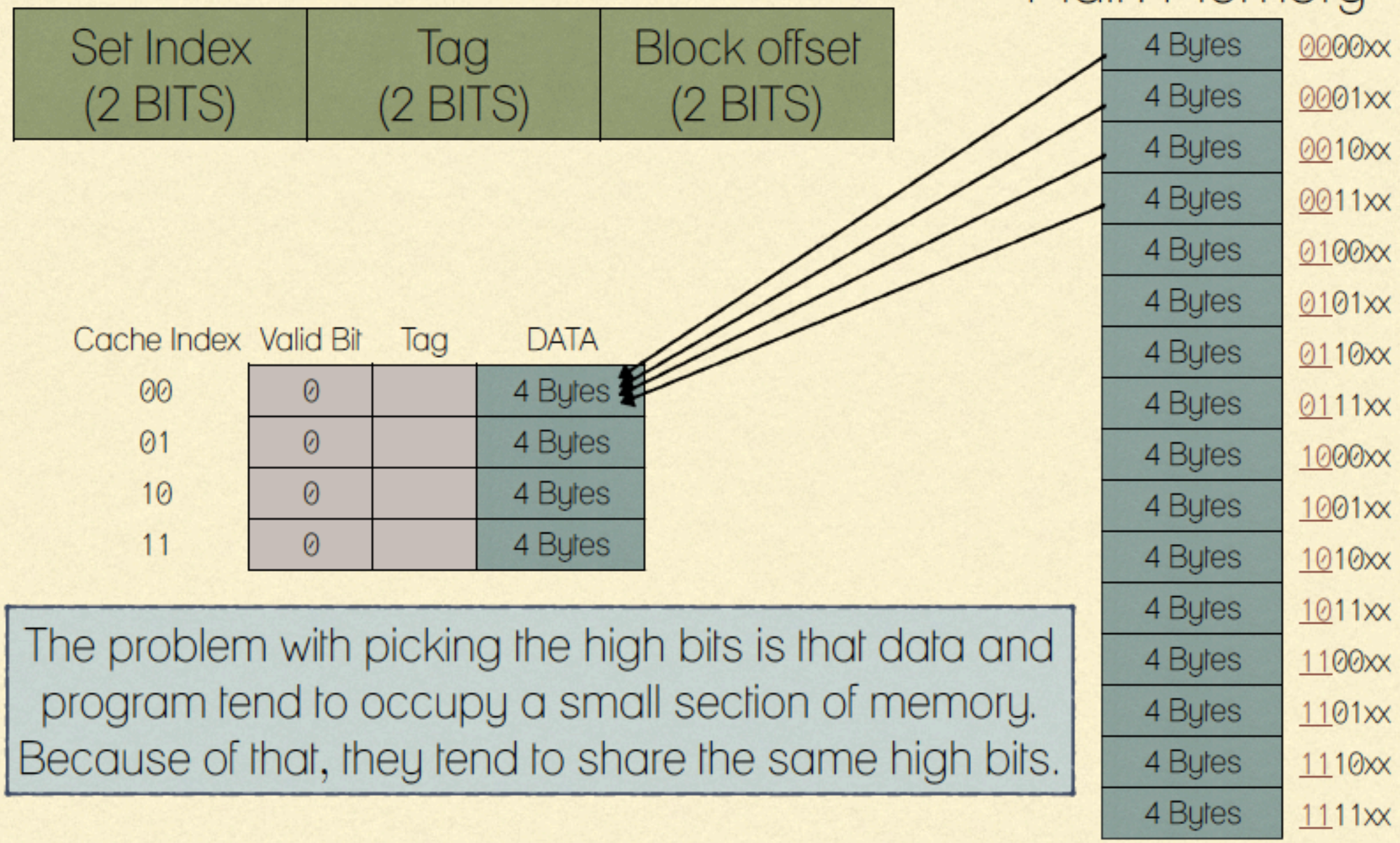
| Set Index (2 BITS) | Tag (2 BITS) | Block offset (2 BITS) |
|---|---|---|

| Cache Index | Valid Bit | Tag | DATA |
|---|---|---|---|
| 00 | 0 | | 4 Bytes |
| 01 | 0 | | 4 Bytes |
| 10 | 0 | | 4 Bytes |
| 11 | 0 | | 4 Bytes |

Main Memory

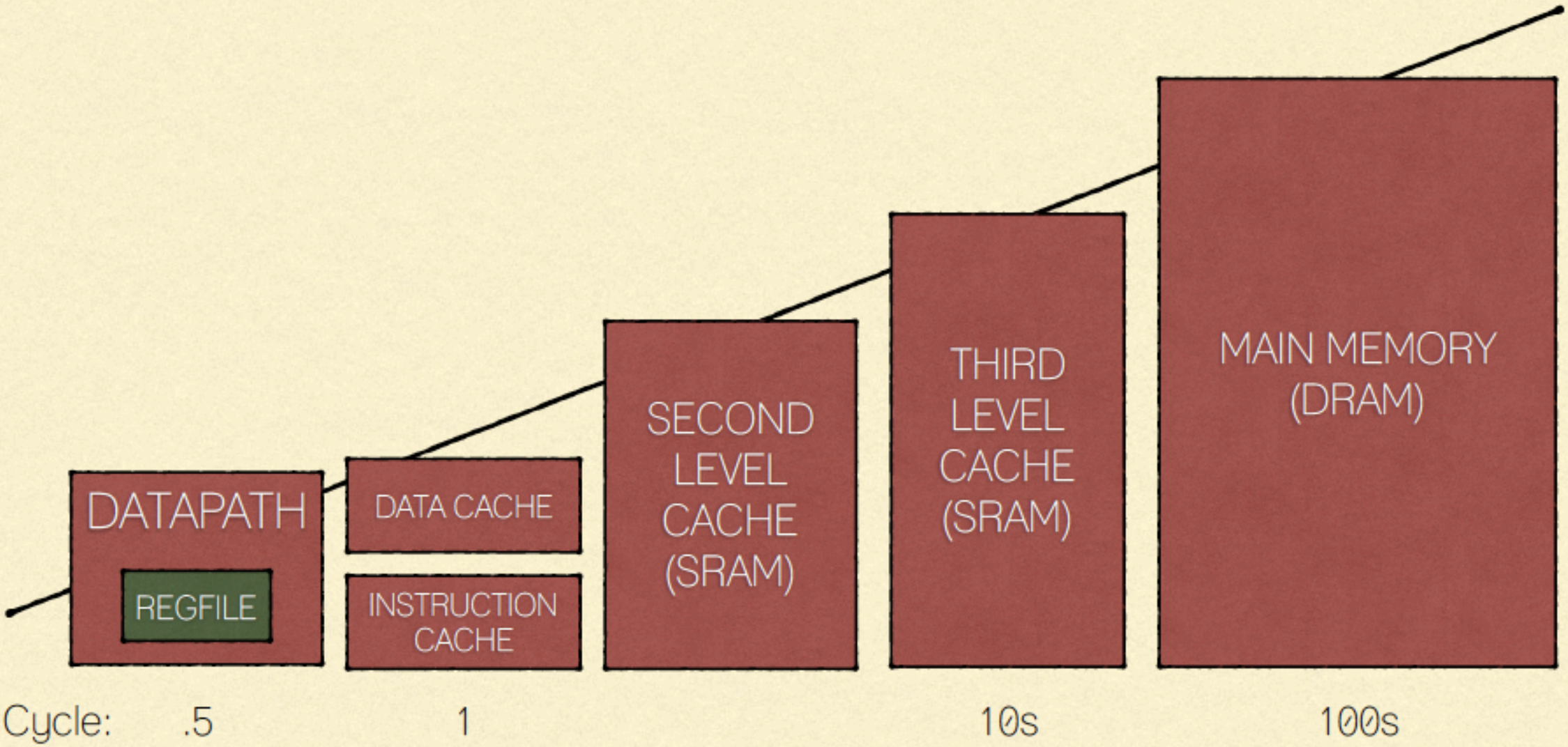
| | |
|---|---|
| 4 Bytes | 0000xx |
| 4 Bytes | 0001xx |
| 4 Bytes | 0010xx |
| 4 Bytes | 0011xx |
| 4 Bytes | 0100xx |
| 4 Bytes | 0100xx |
| 4 Bytes | 0101xx |
| 4 Bytes | 0110xx |
| 4 Bytes | 0111xx |
| 4 Bytes | 1000xx |
| 4 Bytes | 1001xx |
| 4 Bytes | 1010xx |
| 4 Bytes | 1011xx |
| 4 Bytes | 1100xx |
| 4 Bytes | 1101xx |
| 4 Bytes | 1110xx |
| 4 Bytes | 1111xx |

# DOES THE SET INDEX FIELD NEED TO BE NEXT TO THE BLOCK OFFSET?




The problem with picking the high bits is that data and program tend to occupy a small section of memory. Because of that, they tend to share the same high bits.

# TYPICAL MEMORY HIERARCHY



# CACHE TERMS

- Hit rate: faction of accesses that hit in the case
- Miss rate: 1 - Hit rate
- Miss penalty: time to replace a block from the lower level in memory hierarchy to cache (given in clock cycles)
- Hit time: time to access cache memory (including tag comparison)
- Abbreviation: $

# AVERAGE MEMORY ACCESS TIME (AMAT)

- Average Memory Access Time (AMAT) is the average time to access memory considering both hits and misses in the cache:
- AMAT = Time for a hit + Miss rate * Miss penalty

# QUESTION

- AMAT = Time for a hit + Miss rate * Miss penalty
- Given a 200 psec clock, a miss penalty of 50 clock cycles, a miss rate of .02 misses per instruction and a cache hit time of 1 clock cycle, what is AMAT?

A. Less than or equal to 200 psec

B. 400 psec

C. 600 psec

D. Greater than or equal to 800 psec

# QUESTION

- AMAT = Time for a hit + Miss rate * Miss penalty
- Given a 200 psec clock, a miss penalty of 50 clock cycles, a miss rate of .02 misses per instruction and a cache hit time of 1 clock cycle, what is AMAT?

A. Less than or equal to 200 psec

B. 400 psec (correct answer)

C. 600 psec

D. Greater than or equal to 800 psec

# HANDLING STORES WITH WRITE-THROUGH

- Store instructions write to memory, changing values
- Need to make sure cache and memory have same values on writes: 2 policies
    1. Write-through Policy: write to cache and write through the cache to memory:
        - Every write eventually gets to memory
        - Too slow, so include write buffer (to stop CPU from stalling if memory cannot keep up)

# HANDLING STORES WITH WRITE-BACK

1. Write-through Policy: write to cache and write through the cache to memory:
    - Every write eventually gets to memory
    - Too slow, so include "write buffer" (to stop CPU from stalling if memory cannot keep up)
2. Write-back Policy: write only to cache and then write cache block to memory when evict block from cache
    - Writes collected in cache, only single write to memory per block
    - Include a bit to see if wrote to block or not, and then only write back if bit is set (called "dirty" bit — writing makes it "dirty")

# WRITE-BACK EXAMPLE

- Load word at address 010100. What's going to happen?

| Cache Index | Dirty Bit | Valid Bit | Tag | DATA |
|---|---|---|---|---|
| 00 | 0 | 0 | 00 | 4 Bytes |
| 01 | 0 | 0 | 01 | 4 Bytes |
| 10 | 0 | 0 | 10 | 4 Bytes |
| 11 | 0 | 0 | 00 | 4 Bytes |

# WRITE-BACK EXAMPLE

- Load word at address 010100. What's going to happen?

| Cache Index | Dirty Bit | Valid Bit | Tag | DATA |
|---|---|---|---|---|
| 00 | 0 | 0 | 00 | 4 Bytes |
| 01 | 0 | 0 | 01 | 4 Bytes |
| 10 | 0 | 0 | 10 | 4 Bytes |
| 11 | 0 | 0 | 00 | 4 Bytes |

# WRITE-BACK EXAMPLE

- Load word at address 010100. What's going to happen?

1. Go to main memory and load word (since Valid Bit is 0)
2. Update DATA and change Valid Bit to 1

| Cache Index | Dirty Bit | Valid Bit | Tag | DATA |
|---|---|---|---|---|
| 00 | 0 | 0 | 00 | 4 Bytes |
| 01 | 0 | 1 | 01 | 4 Bytes |
| 10 | 0 | 0 | 10 | 4 Bytes |
| 11 | 0 | 0 | 00 | 4 Bytes |

# WRITE-BACK EXAMPLE

- Load word at address 010100. What's going to happen?

    1. Go to Main Memory and load word (since Valid Bit is 0)
    2. Update DATA and change Valid Bit to 1
    3. Dirty Bit changes to (or stays at) 0

| Cache Index | Dirty Bit | Valid Bit | Tag | DATA |
|---|---|---|---|---|
| 00 | 0 | 0 | 00 | 4 Bytes |
| 01 | 0 | 1 | 01 | 4 Bytes |
| 10 | 0 | 0 | 10 | 4 Bytes |
| 11 | 0 | 0 | 00 | 4 Bytes |

# WRITE-BACK EXAMPLE

- Store word at address 010100. What's going to happen?

| Cache Index | Dirty Bit | Valid Bit | Tag | DATA |
|---|---|---|---|---|
| 00 | 0 | 0 | 00 | 4 Bytes |
| 01 | 0 | 1 | 01 | 4 Bytes |
| 10 | 0 | 0 | 10 | 4 Bytes |
| 11 | 0 | 0 | 00 | 4 Bytes |

# WRITE-BACK EXAMPLE

- Store word at address 010100. What's going to happen?
    1. Update DATA in cache (don't go to main memory)
    2. What's going to happen to Dirty Bit?

| Cache Index | Dirty Bit | Valid Bit | Tag | DATA |
|---|---|---|---|---|
| 00 | 0 | 0 | 00 | 4 Bytes |
| 01 | 0 | 1 | 01 | 4 Bytes |
| 10 | 0 | 0 | 10 | 4 Bytes |
| 11 | 0 | 0 | 00 | 4 Bytes |

# WRITE-BACK EXAMPLE

- Store word at address 010100. What's going to happen?
    1. Update DATA in cache (don't go to main memory)
    2. What's going to happen to Dirty Bit? Dirty Bit becomes 1

| Cache Index | Dirty Bit | Valid Bit | Tag | DATA |
|---|---|---|---|---|
| 00 | 0 | 0 | 00 | 4 Bytes |
| 01 | 1 | 1 | 01 | 4 Bytes |
| 10 | 0 | 0 | 10 | 4 Bytes |
| 11 | 0 | 0 | 00 | 4 Bytes |

# WRITE-BACK EXAMPLE

- Load word at address 110100. What's going to happen?

| Cache Index | Dirty Bit | Valid Bit | Tag | DATA |
|---|---|---|---|---|
| 00 | 0 | 0 | 00 | 4 Bytes |
| 01 | 1 | 1 | 01 | 4 Bytes |
| 10 | 0 | 0 | 10 | 4 Bytes |
| 11 | 0 | 0 | 00 | 4 Bytes |

# WRITE-BACK EXAMPLE

- Load word at address 110100. What's going to happen?

    1. Since Tag doesn't match, need to update block

| Cache Index | Dirty Bit | Valid Bit | Tag | DATA |
|---|---|---|---|---|
| 00 | 0 | 0 | 00 | 4 Bytes |
| 01 | 1 | 1 | 01 | 4 Bytes |
| 10 | 0 | 0 | 10 | 4 Bytes |
| 11 | 0 | 0 | 00 | 4 Bytes |

# WRITE-BACK EXAMPLE

- Load word at address 110100. What's going to happen?
    1. Since Tag doesn't match, need to update block
    2. Since Dirty Bit is 1, need to update main memory before replacing block

| Cache Index | Dirty Bit | Valid Bit | Tag | DATA |
|---|---|---|---|---|
| 00 | 0 | 0 | 00 | 4 Bytes |
| 01 | 1 | 1 | 01 | 4 Bytes |
| 10 | 0 | 0 | 10 | 4 Bytes |
| 11 | 0 | 0 | 00 | 4 Bytes |

# WRITE-BACK EXAMPLE

- Load word at address 110100. What's going to happen?

    1. Since Tag doesn't match, need to update block
    2. Since Dirty Bit is 1, need to update main memory before replacing block
    3. After main memory updated, new data replaces old data in block (and we change Tag value to new tag) — we also change "Dirty Bit" to 0

| Cache Index | Dirty Bit | Valid Bit | Tag | DATA |
|---|---|---|---|---|
| 00 | 0 | 0 | 00 | 4 Bytes |
| 01 | 0 | 1 | 11 | 4 Bytes |
| 10 | 0 | 0 | 10 | 4 Bytes |
| 11 | 0 | 0 | 00 | 4 Bytes |