EC 19-20
Computer Architecture
Fall 2019

Chapter 3: Exploiting the Memory Hierarchy

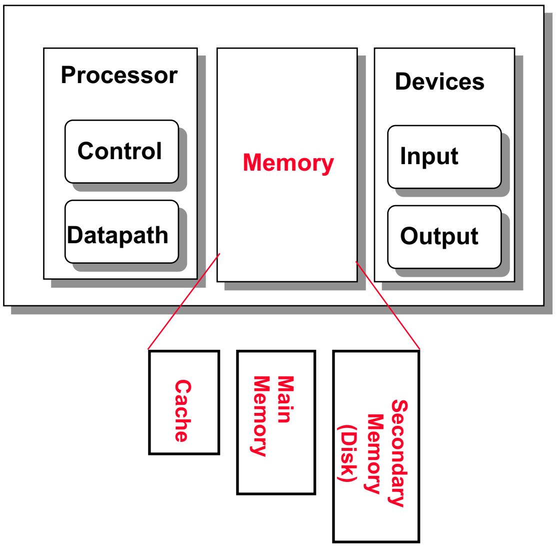
Mª Angeles González Navarro

www.ac.uma.es/~angeles

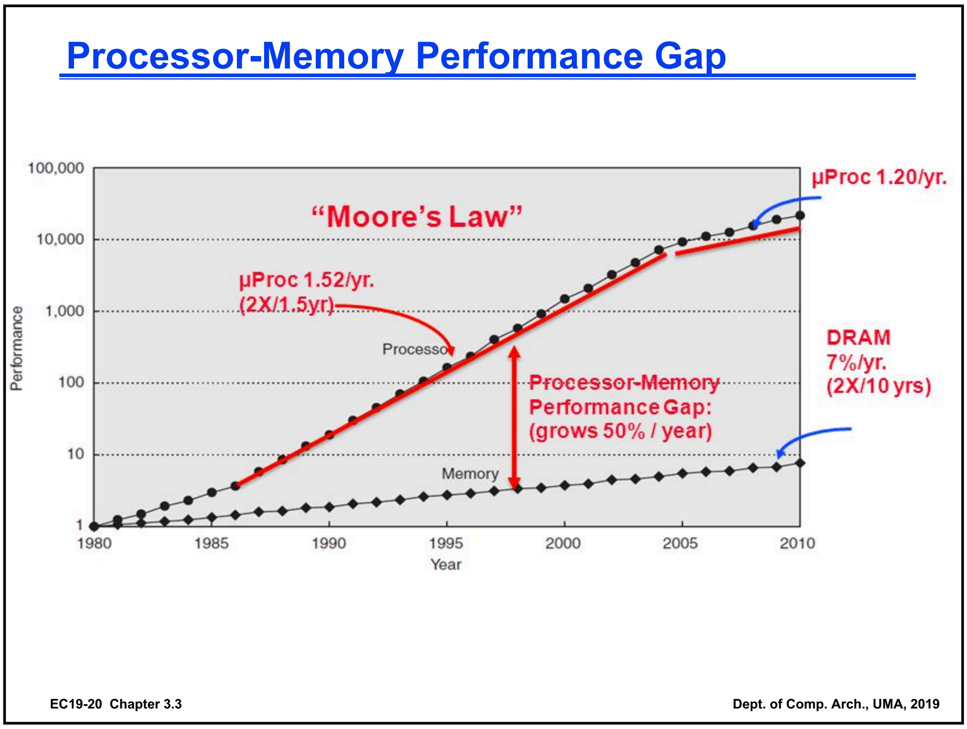
angeles@ac.uma.es

[Adapted from Mary Jane Irwin's slides (PSU) based on *Computer Organization and Design*, ARM Ed. Patterson & Hennessy, © 2017, Elsevier]

Review: Major Components of a Computer



Processor-Memory Performance Gap



"Moore's Law"

µProc 1.52/yr. (2X/1.5yr)

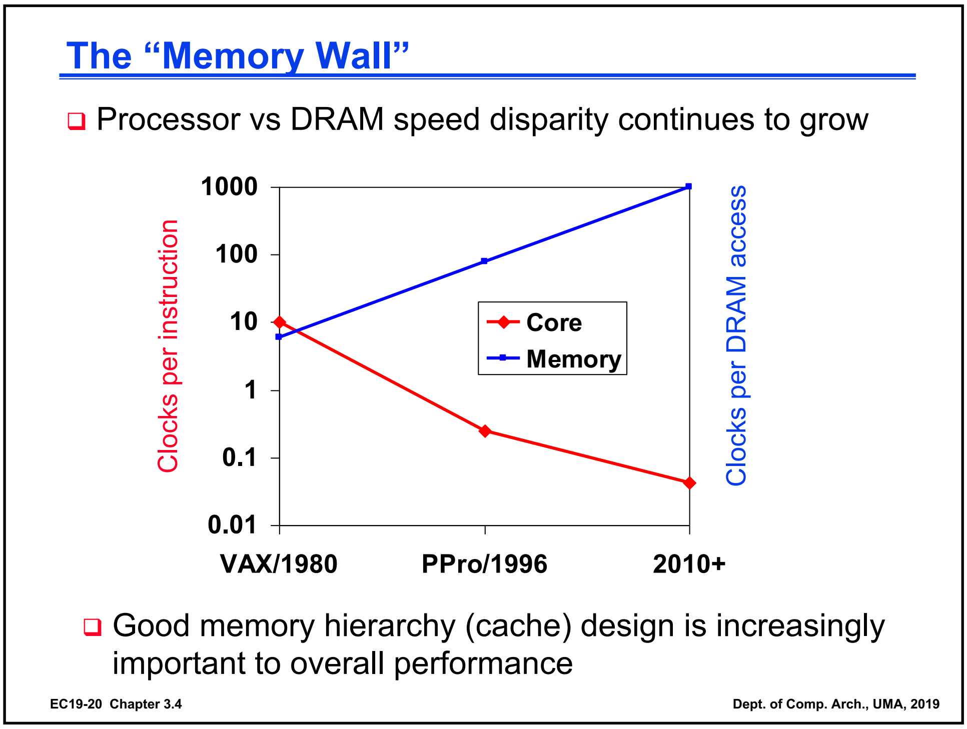
µProc 1.20/yr.

Processor-Memory Performance Gap: (grows 50% / year)

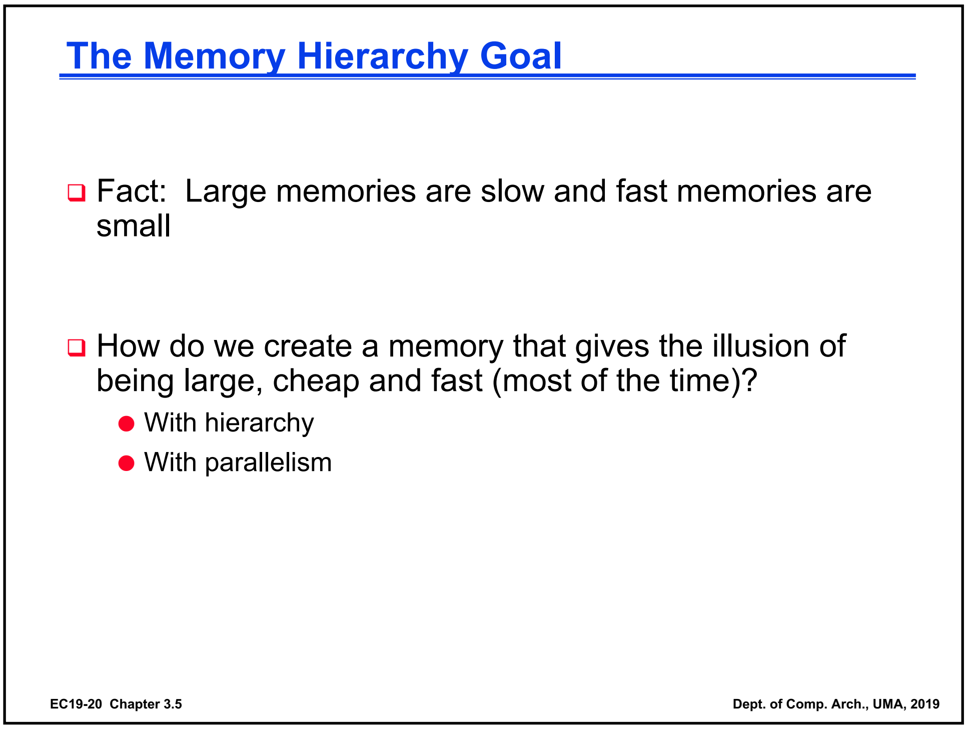
DRAM 7%/yr. (2X/10 yrs)

The "Memory Wall"

- ❑ Processor vs DRAM speed disparity continues to grow

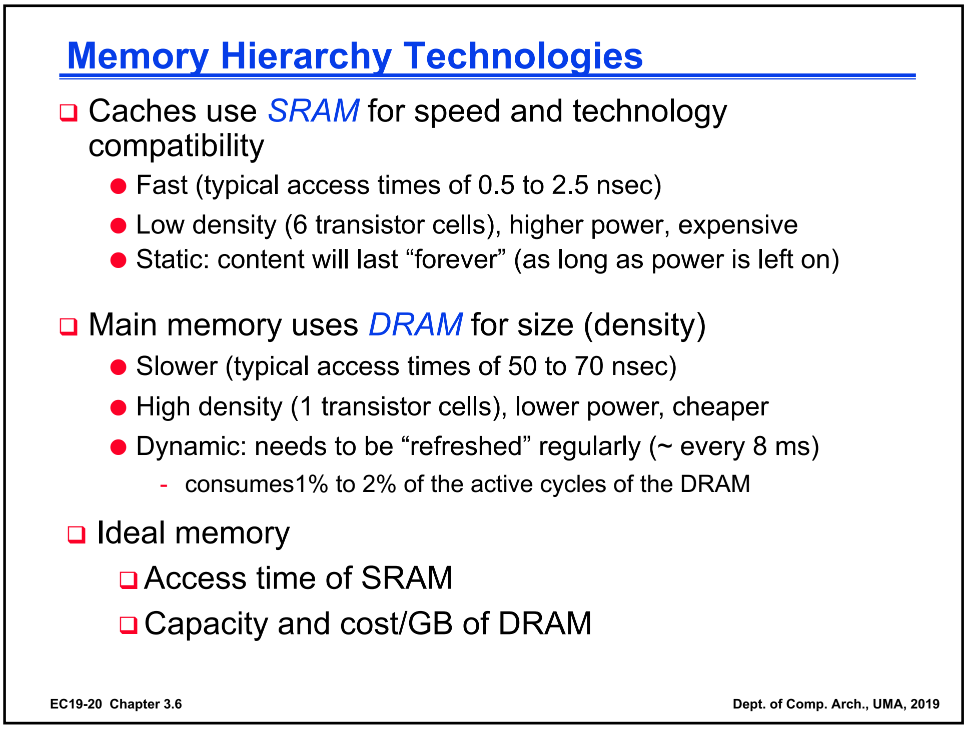


- ❑ Good memory hierarchy (cache) design is increasingly important to overall performance



The Memory Hierarchy Goal

- Fact: Large memories are slow and fast memories are small
- How do we create a memory that gives the illusion of being large, cheap and fast (most of the time)?
  - With hierarchy
  - With parallelism

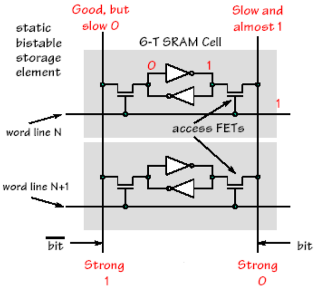


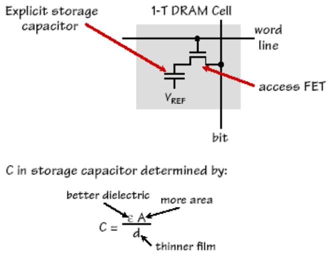
Memory Hierarchy Technologies

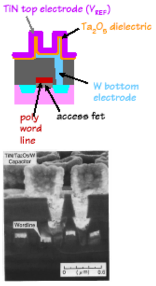
- Caches use *SRAM* for speed and technology compatibility
  - Fast (typical access times of 0.5 to 2.5 nsec)
  - Low density (6 transistor cells), higher power, expensive
  - Static: content will last “forever” (as long as power is left on)
- Main memory uses *DRAM* for size (density)
  - Slower (typical access times of 50 to 70 nsec)
  - High density (1 transistor cells), lower power, cheaper
  - Dynamic: needs to be “refreshed” regularly (~ every 8 ms)
    - consumes1% to 2% of the active cycles of the DRAM
- Ideal memory
  - Access time of SRAM
  - Capacity and cost/GB of DRAM

Memory Hierarchy Technologies cont.

- ❑ SRAM cell
  - FF structure
  - Not refreshing
  - 6T per cell (low density, higher power)
  - Expensive (higher cost per bit)
  - Faster (0.5 to 2.5 nsec)
- ❑ DRAM cell
  - Explicit storage capacitor
  - It needs refreshing (destructive reading)
  - 1T per cell (high density, lower power)
  - Cheaper (lower cost per bit)
  - Slower (50 to 70 nsec)

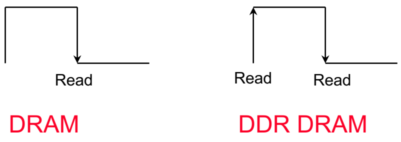



Explicit storage capacitor
1-T DRAM Cell
word line
access FET
V_{REF}
bit
C in storage capacitor determined by:
better dielectric
more area
thinner film
$C = \frac{\varepsilon A}{d}$



Advanced DRAM Organization

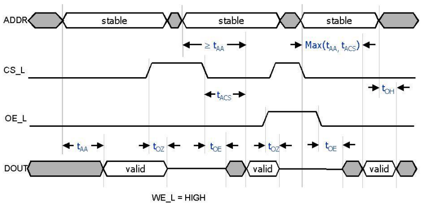
- ❑ Bits in a DRAM are organized as a rectangular array
  - Addresses divided into 2 halves (row and column)
    - *RAS* or *Row Access Strobe* triggering the row decoder
    - *CAS* or *Column Access Strobe* triggering the column selector
  - DRAM accesses an entire row
  - Burst mode: supply successive words from a row with reduced latency
- ❑ Double data rate (DDR) DRAM (DDR4)
  - Transfer on rising and falling clock edges



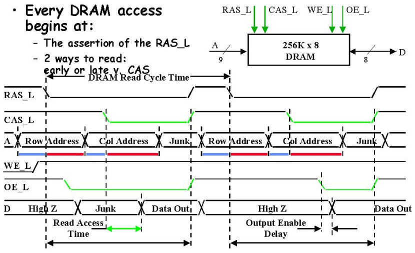
Advanced DRAM Organization cont.

- DRAM read timing slower that SRAM read timing:
  - Dynamic: needs to be “refreshed” regularly (~ every 8 ms)
    - consumes1% to 2% of the active cycles of the DRAM
  - Addresses divided into 2 halves (row and column)
    - *RAS* or *Row Access Strobe* triggering the row decoder
    - *CAS* or *Column Access Strobe* triggering the

SRAM Read Timing (typical)



DRAM Read Timing


- Every DRAM access begins at:
  - The assertion of the RAS_L
  - 2 ways to read: early or late v. CAS

Early Read Cycle: OE_L asserted before CAS_L

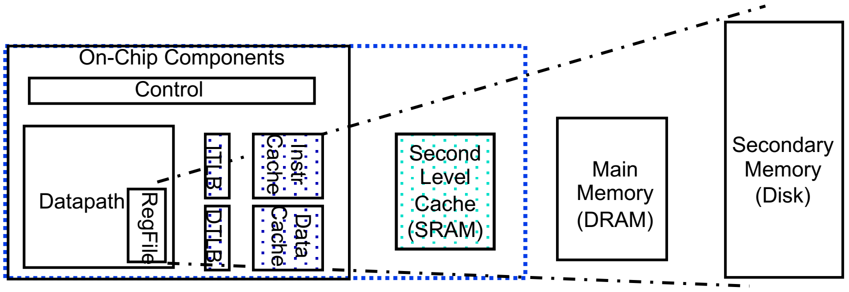
Late Read Cycle: OE_L asserted after CAS_L

Memory Hierarchy

- We focus in MEMORY HIERARCHY
- There are several levels in the hierarchy:
  - Each lower level contains a copy of some data of the higher level
  - Lower level: fast and small
  - Higher level: slow and large

A Typical Memory Hierarchy

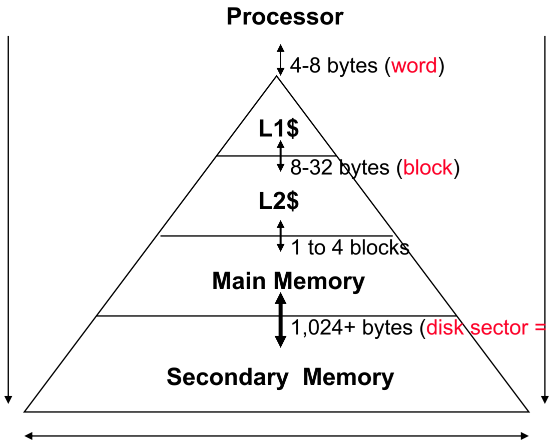
- Take advantage of the principle of locality to present the user with as much memory as is available in the *cheapest* technology at the speed offered by the *fastest* technology



Speed (cycles):	½'s	1's	10's	100's	10,000's
Size (bytes):	100's	10K's	M's	G's	T's
Cost:	highest				lowest

Characteristics of the Memory Hierarchy

Increasing distance from the processor in access time



Inclusive– what is in L1$ is a subset of what is in L2$ is a subset of what is in MM that is a subset of is in SM

(Relative) size of the memory at each level

The Memory Hierarchy: Why Does it Work?

❑ Temporal Locality (locality in time)

- ● If a memory location is referenced, then it will tend to be referenced again soon
- ⇒ Keep most recently accessed data items closer to the processor
- ⇒ *Programs have many loops*

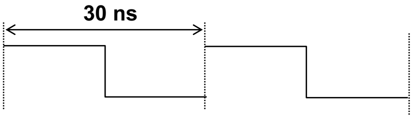
❑ Spatial Locality (locality in space)

- ● If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon
- ⇒ Move blocks consisting of contiguous words closer to the processor
- ⇒ *Programs have consecutive instructions*

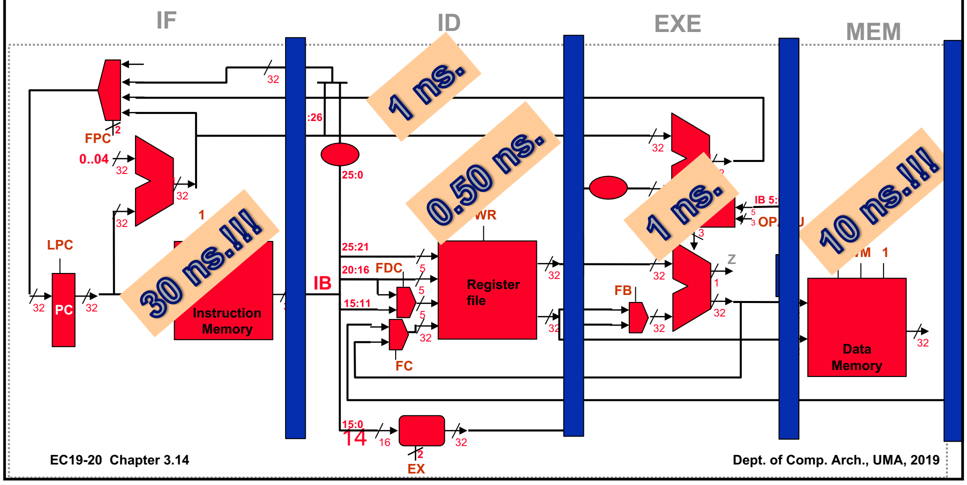
Actual processors: 96.8% of memory accesses find the data in the cache

Example MIPS: no hierarchy

Clock cycle (cc): **30 ns**

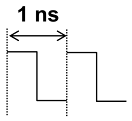


Element	T active	T inactive
Inst. M.	30 ns	0 ns
File Reg.	1 ns	29 ns
ALU	1 ns	29 ns
Data M.	10 ns	20 ns

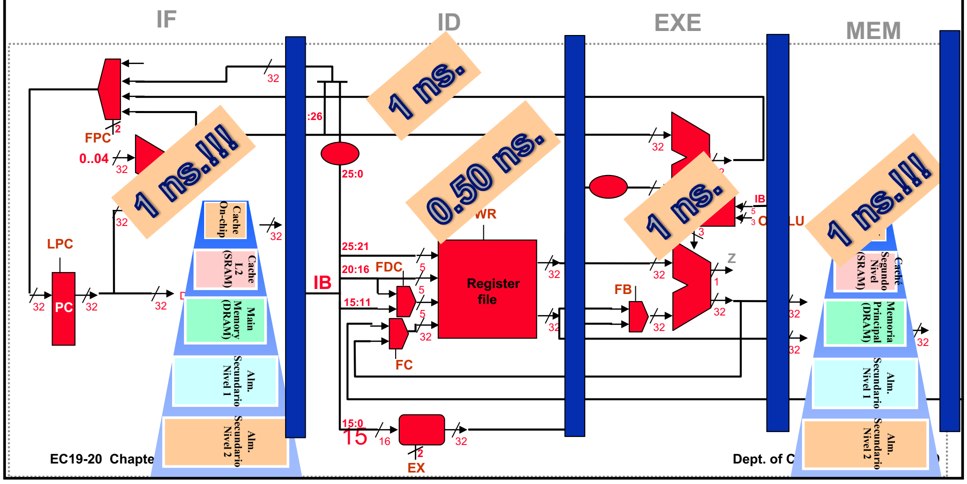


Example MIPS: memory hierarchy

Clock cycle (cc): **1 ns**



Element	T active	T inactive
Inst. M.	***1 ns***	0 ns
File Reg.	1 ns	0 ns
ALU	1 ns	0 ns
Data M.	***1 ns***	0 ns

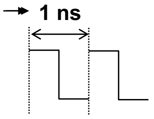


EC19-20 Chapte

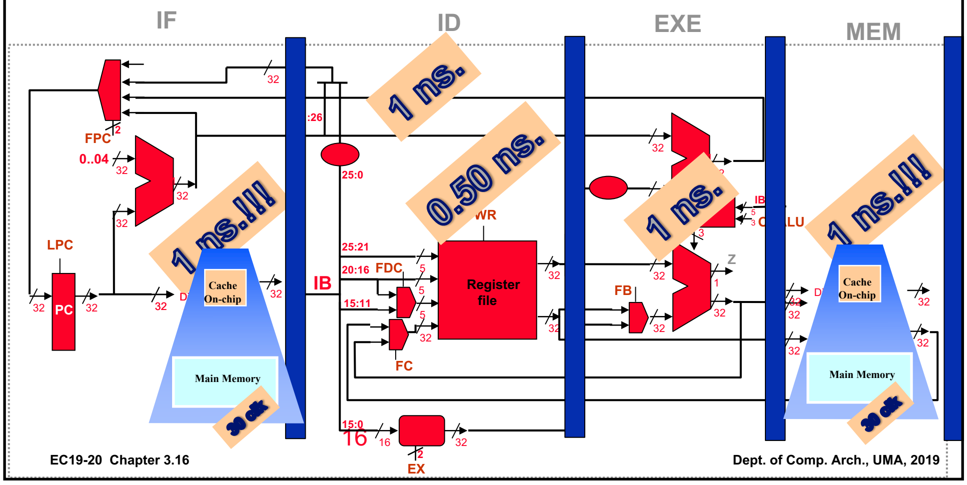
Dept. of C

MIPS Cache – Main Memory

Clock cycle: **1 ns**

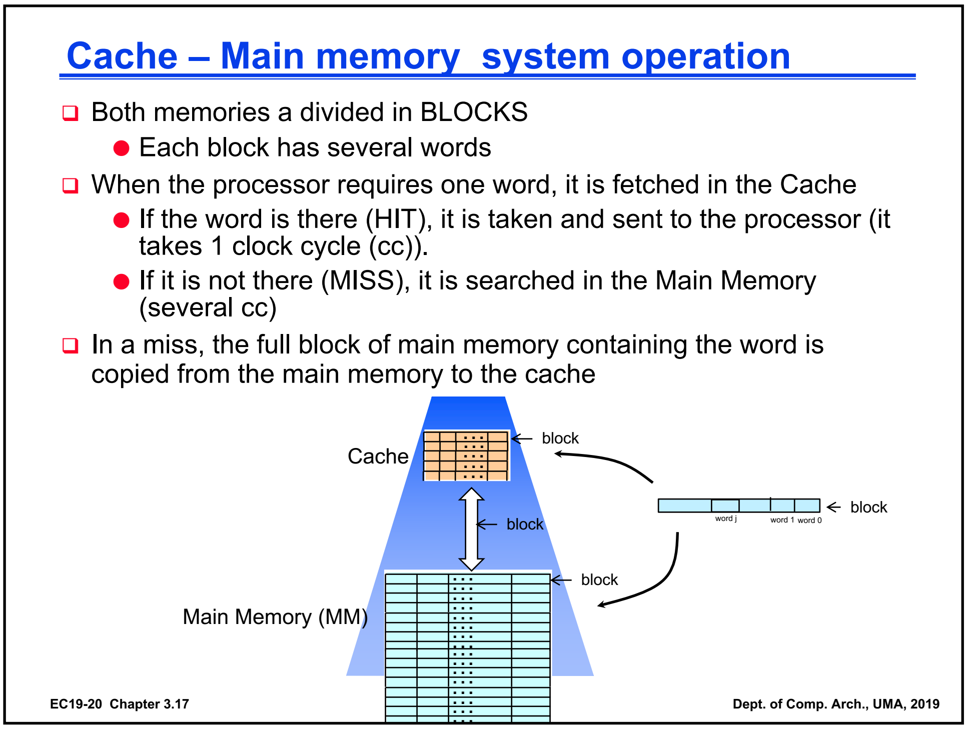


Element	T active	T inactive
Inst. M.	***1 ns***	0 ns
File Reg.	1 ns	0 ns
ALU	1 ns	0 ns
Data M.	***1 ns***	0 ns



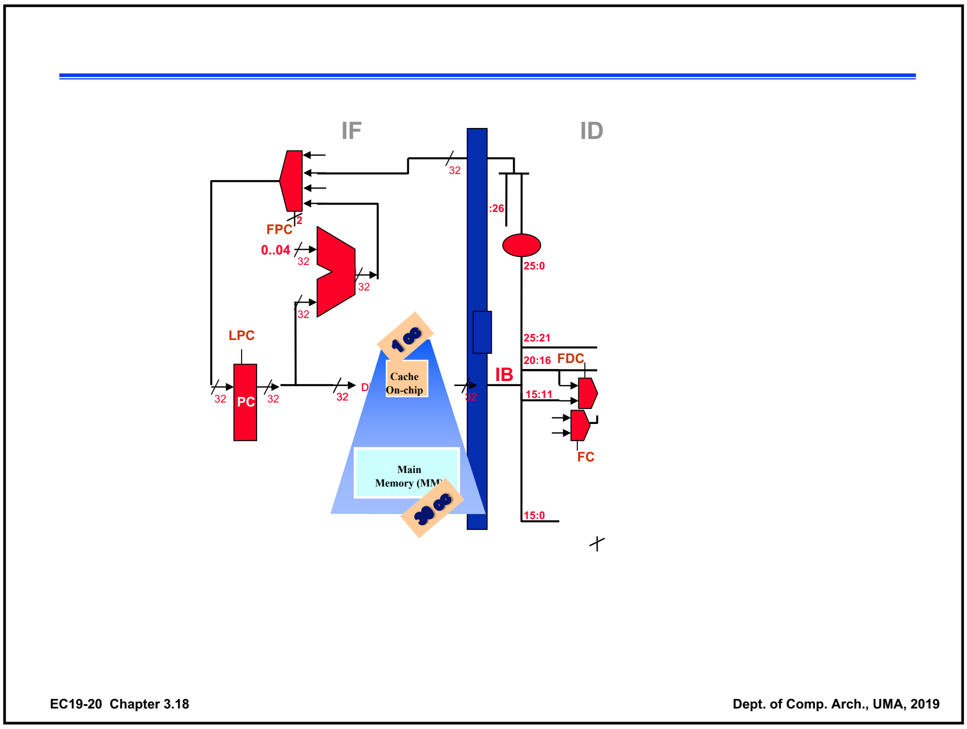
EC19-20 Chapter 3.16

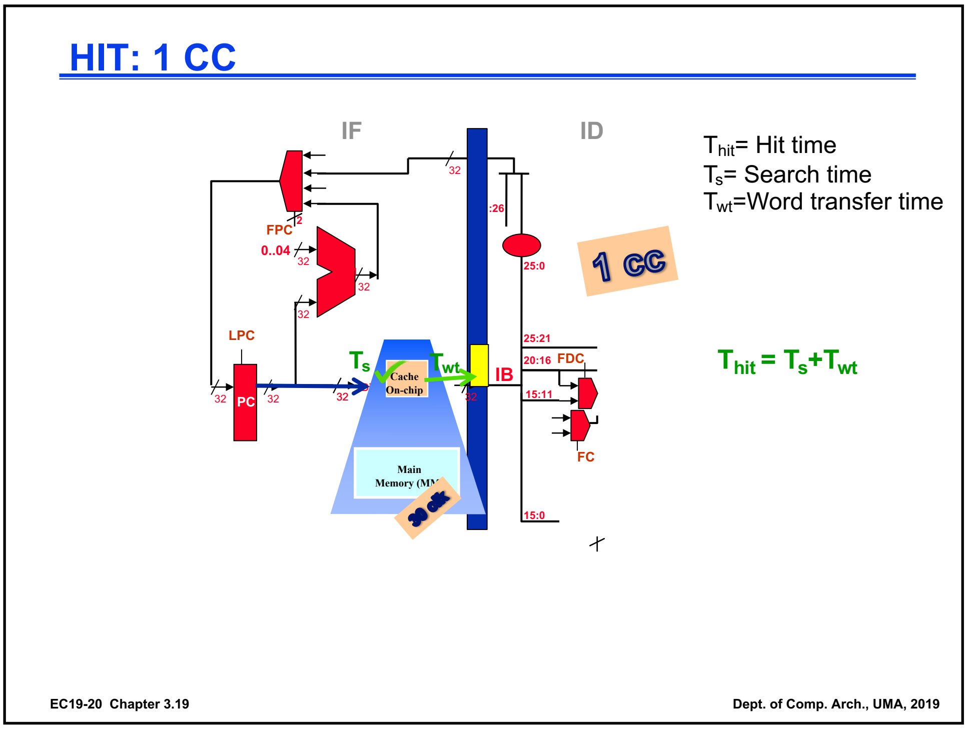
Dept. of Comp. Arch., UMA, 2019



Cache – Main memory system operation

- ❑ Both memories a divided in BLOCKS
  - ● Each block has several words
- ❑ When the processor requires one word, it is fetched in the Cache
  - ● If the word is there (HIT), it is taken and sent to the processor (it takes 1 clock cycle (cc)).
  - ● If it is not there (MISS), it is searched in the Main Memory (several cc)
- ❑ In a miss, the full block of main memory containing the word is copied from the main memory to the cache



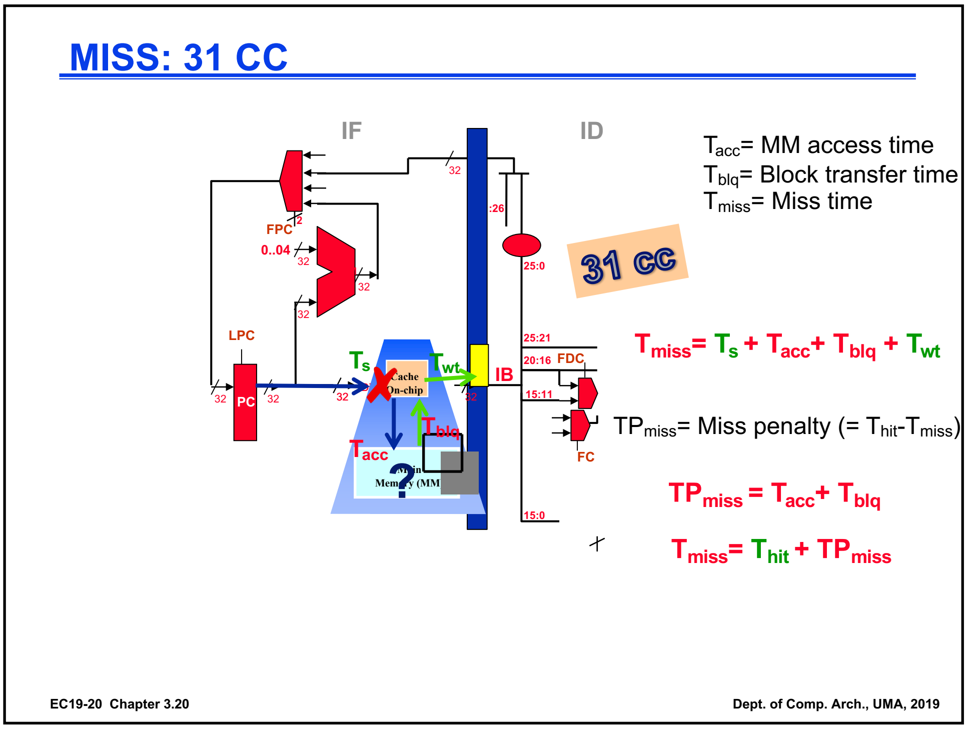


HIT: 1 CC

T_{hit}= Hit time
T_s= Search time
T_{wt}=Word transfer time

1 CC

$T_{hit} = T_s + T_{wt}$



MISS: 31 CC

T_{acc}= MM access time
T_{blq}= Block transfer time
T_{miss}= Miss time

31 CC

$T_{miss} = T_s + T_{acc} + T_{blq} + T_{wt}$

TP_{miss}= Miss penalty (= T_{hit}-T_{miss})

$TP_{miss} = T_{acc} + T_{blq}$

$T_{miss} = T_{hit} + TP_{miss}$

The Memory Hierarchy: Terminology

- Block (or line): the minimum unit of information that is present (or not) in a cache
- Hit Rate (P_{hit}): the fraction of memory accesses found in a level of the memory hierarchy
  - $P_{hit} = {}^{Number\ of\ hits}/_{Number\ of\ memory\ references}$
  - Hit Time (T_{hit}): Time to access that level which consists of

    Time to search the block (hit/miss) (T_s) + Time to transfer the word (T_{wt})

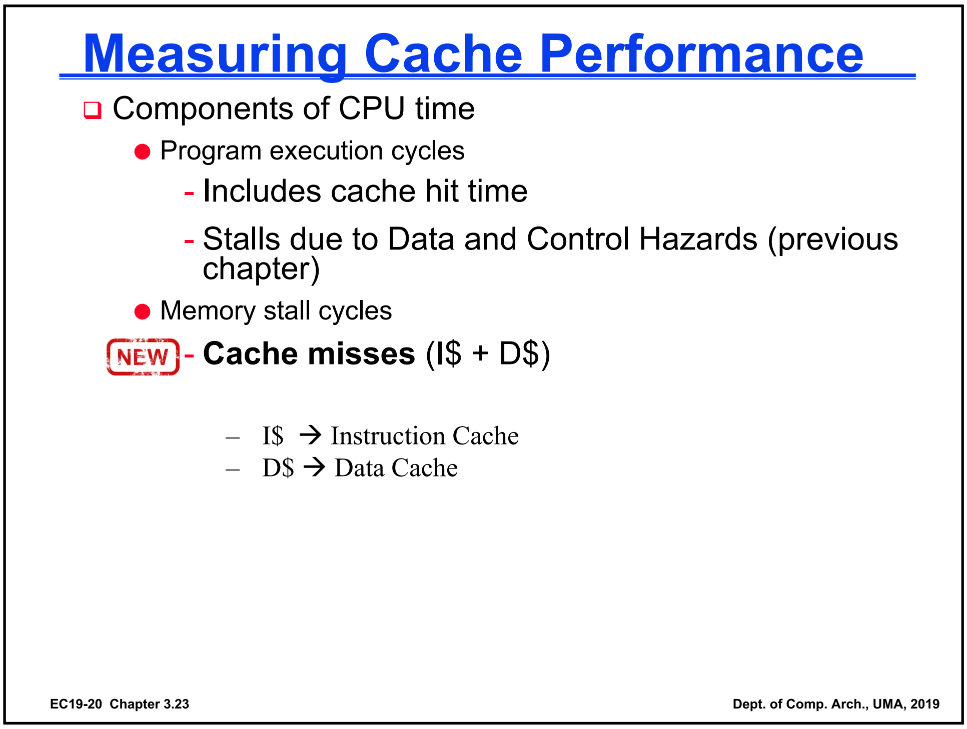
    → **By default, we always consider Hit time= 1 cc**
- Miss Rate (P_{miss}): the fraction of memory accesses *not* found in a level of the memory hierarchy $\Rightarrow$ $P_{miss}=1 – P_{hit}$
  - $P_{miss} = {}^{Number\ of\ misses}/_{Number\ of\ memory\ references}$
  - Miss Penalty: Time to replace a block in that level with the corresponding block from a lower level which consists of

    Time to access the block in the lower level (T_{acc}) + Time to transmit that block to the level that experienced the miss (T_{blk})

Hit Time << Miss Penalty

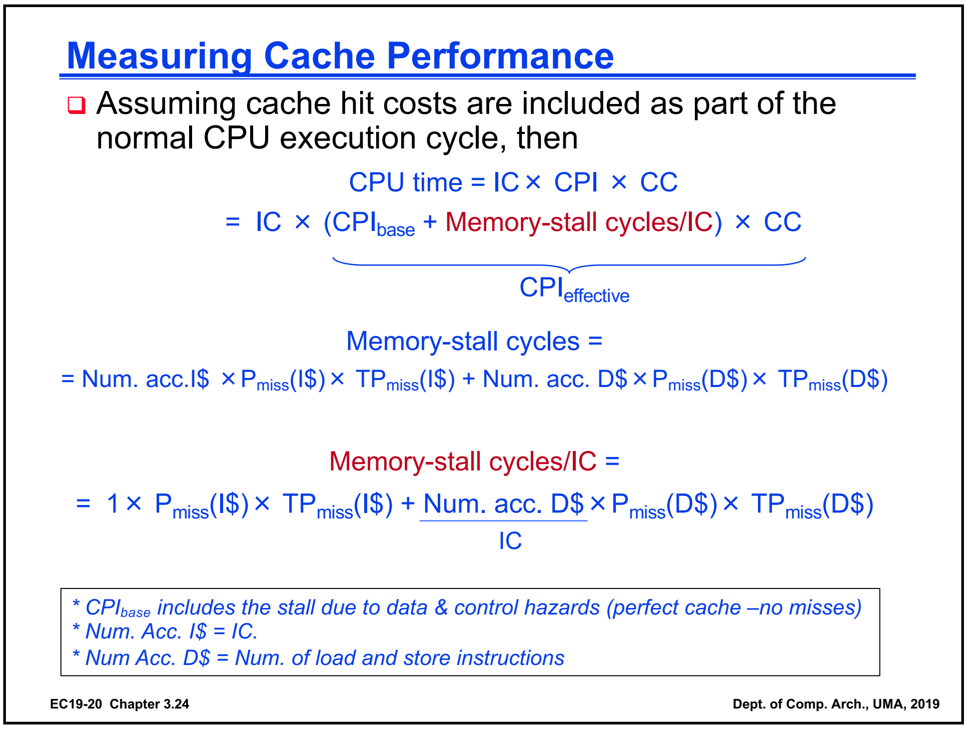
Average Access Time

- Hit time is also important for performance
- Average memory access time (AMAT)
  - AMAT = Hit time + Miss rate × Miss penalty
- Example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I$ache miss rate = 5%
  - AMAT = 1 + 0.05 × 20 = 2ns
    - 2 cycles per instruction



Measuring Cache Performance

- Components of CPU time
  - Program execution cycles
    - Includes cache hit time
    - Stalls due to Data and Control Hazards (previous chapter)
  - Memory stall cycles
    - NEW - **Cache misses** (I\$ + D\$)
      - I\$ → Instruction Cache
      - D\$ → Data Cache



Measuring Cache Performance

- Assuming cache hit costs are included as part of the normal CPU execution cycle, then

$$\text{CPU time} = \text{IC} \times \text{CPI} \times \text{CC}$$

$$= \text{IC} \times \underbrace{(\text{CPI}_{\text{base}} + \text{Memory-stall cycles/IC})}_{\text{CPI}_{\text{effective}}} \times \text{CC}$$

$$\text{Memory-stall cycles} =$$

$$= \text{Num. acc.I\$} \times P_{\text{miss}}(\text{I\$}) \times TP_{\text{miss}}(\text{I\$}) + \text{Num. acc. D\$} \times P_{\text{miss}}(\text{D\$}) \times TP_{\text{miss}}(\text{D\$})$$

$$\text{Memory-stall cycles/IC} =$$

$$= 1 \times P_{\text{miss}}(\text{I\$}) \times TP_{\text{miss}}(\text{I\$}) + \frac{\text{Num. acc. D\$}}{\text{IC}} \times P_{\text{miss}}(\text{D\$}) \times TP_{\text{miss}}(\text{D\$})$$

** CPI_{base} includes the stall due to data & control hazards (perfect cache –no misses)*
** Num. Acc. I\$ = IC.*
** Num Acc. D\$ = Num. of load and store instructions*

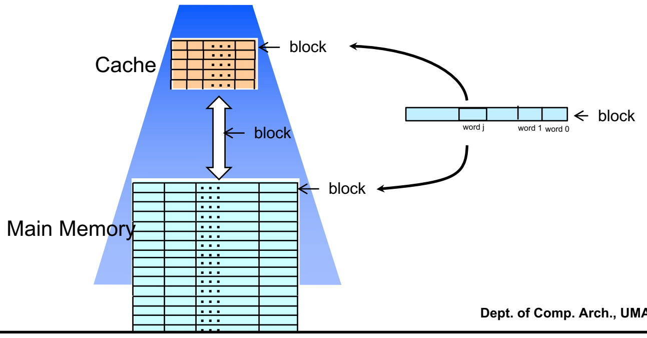
Cache Performance Example

- Given
  - I$ miss rate P_{miss}(I$)= 2%
  - D$ miss rate P_{miss}(D$)= 4%
  - Miss penalty (I$ & D$) TP_{miss}(I$), TP_{miss}(D$) = 80 cycles
  - Base CPI (ideal cache) CPI_{base}= 1.5
  - Load & stores are 36% of instructions *(Num.acc.D$ / num. inst = 0.36)*

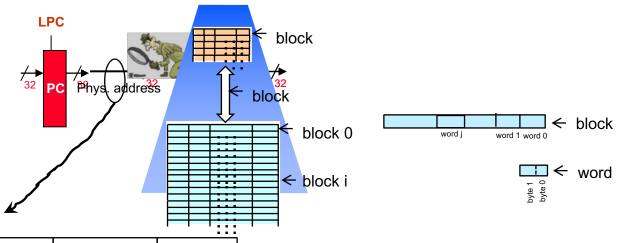
- Miss cycles per instruction
  - I$: 0.02 × 80 = 1.6
  - D$: 0.36 × 0.04 × 80 = 1.152
- Effective CPI = 1.5 + 1.6 + 1.152 = 4.252

Cache – Main memory system operation

- Both memories a divided in BLOCKS
  - Each block has several words
- When the processor requires one word, it is fetched in the Cache
  - If the word is there (HIT), it is taken and sent to the processor (1 cc).
  - If it is not there (MISS), it is searched in the Main Memory (several cc)
- In a miss, the full block of main memory containing the word is copied from the main memory to the cache



Address Subdivision



Block number | Block offset | Byte offset ← Physical address (MM address)

Byte offset → Num. of byte inside a word

Block offset → Num. of word inside a block

Block number → Num.of block of MM

Address Subdivision

Whole address → Size: depends on the Num. of bytes of the MM

Block number | Block offset | Byte offset ← Physical address (MM address)

Byte offset → Size: depends on the Num. of bytes per word

Block offset → Size: dependsNum. of words per block

Block number → Size: depends on the Num.of blocks of MM

Example: MM of 2^{32} bytes, Words of 4 bytes (2 bits, bit 1,0), blocks of 8 words (3 bits, bits 4,3,2)

Physical address: 718h ➡ 718h = 00111 0001 1000

38h 6 0

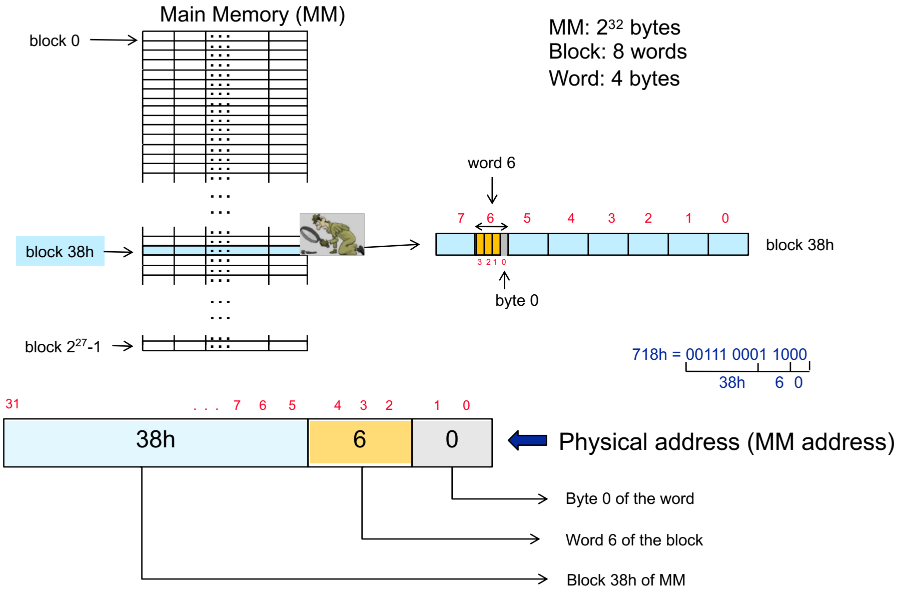
31 . . . 7 6 5 4 3 2 1 0

38h | 6 | 0 ← Physical address (MM address)

0 → Byte 0 of the word

6 → Word 6 of the block

38h → Block 38h of MM



Main Memory (MM)

block 0

block 38h

block 2^{27}-1

MM: 2^{32} bytes
Block: 8 words
Word: 4 bytes

word 6

7 6 5 4 3 2 1 0

3 2 1 0

block 38h

byte 0

718h = 00111 0001 1000

38h 6 0

31 . . . 7 6 5 4 3 2 1 0

38h | 6 | 0

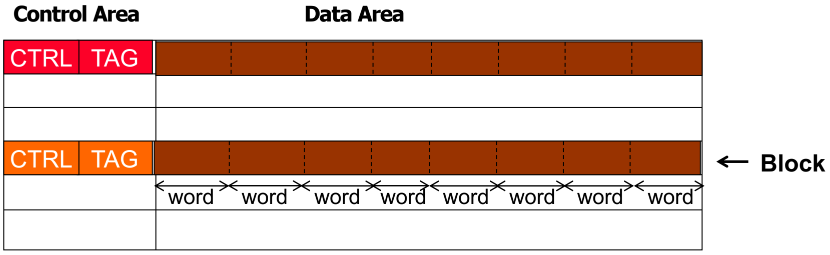
Physical address (MM address)

Byte 0 of the word

Word 6 of the block

Block 38h of MM

- **Internal cache architecture:**
  - Two parts:
    - **<u>Data area (storage):</u>** Contains copies of some blocks of MM
    - **<u>Control area (directory):</u>** auxiliar information about the current blocks in the cache (i.e., blocks of MM which are currently copied in the CM)
      - Tags (TAG) and control bits (CTRL).



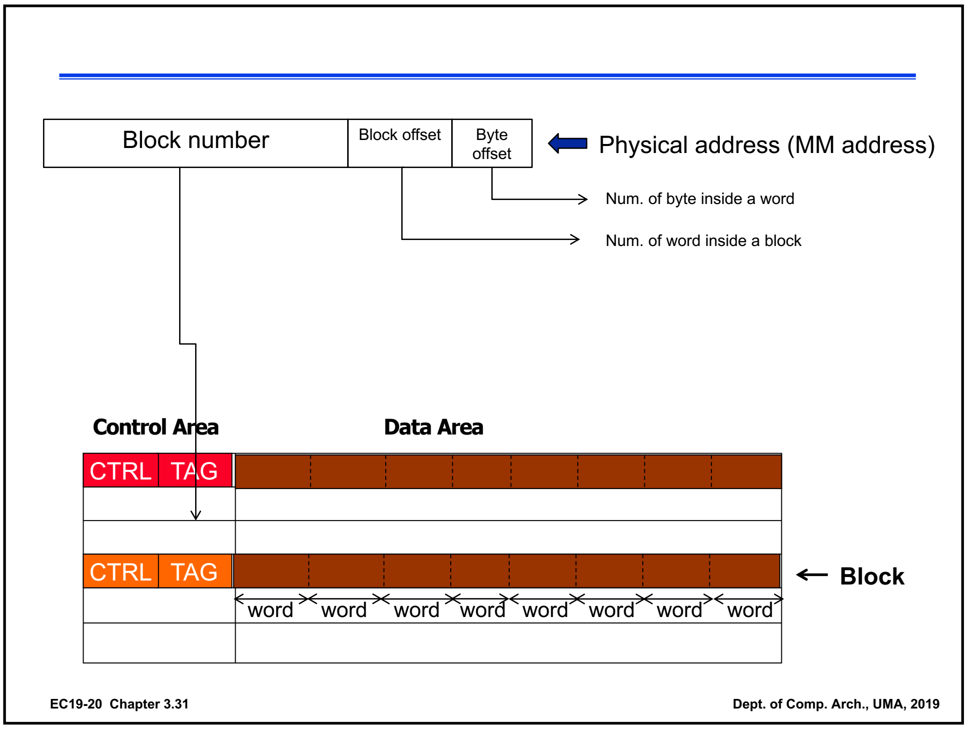
Control Area **Data Area**

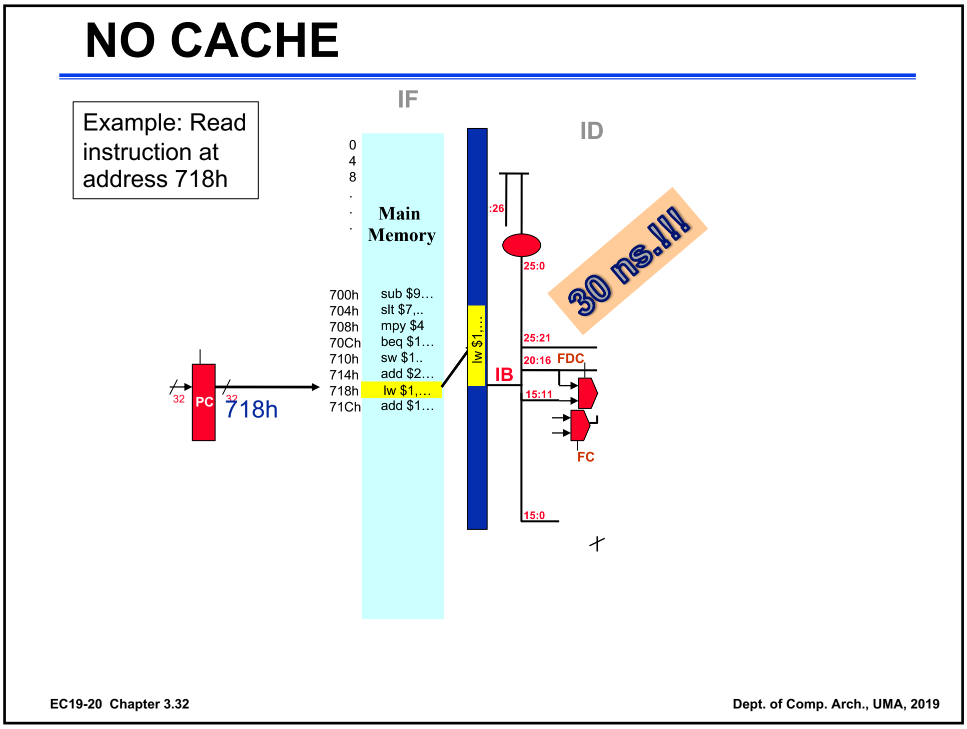
CTRL TAG

CTRL TAG

word word word word word word word word

Block





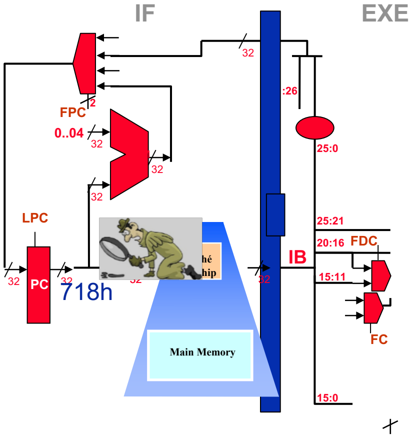
NO CACHE

Example: Read instruction at address 718h

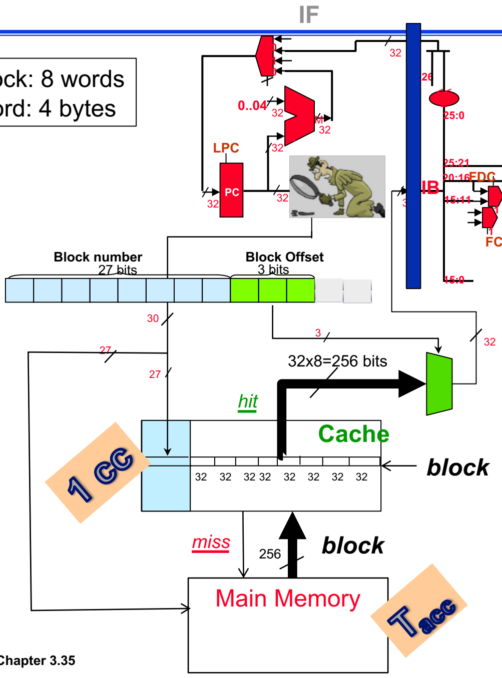
700h	sub $9…
704h	slt $7,..
708h	mpy $4
70Ch	beq $1…
710h	sw $1..
714h	add $2…
718h	lw $1,…
71Ch	add $1…

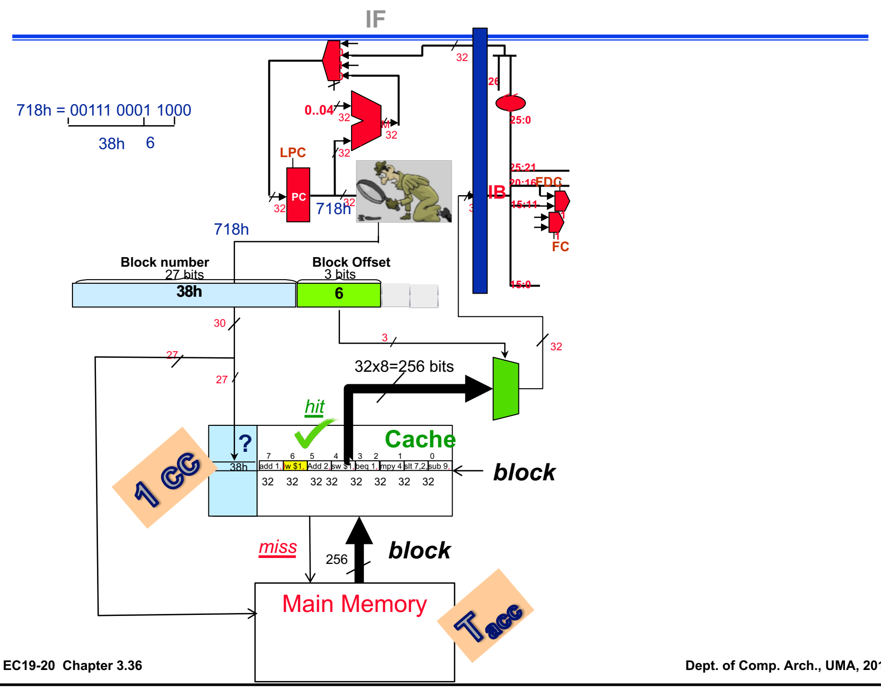
CACHE

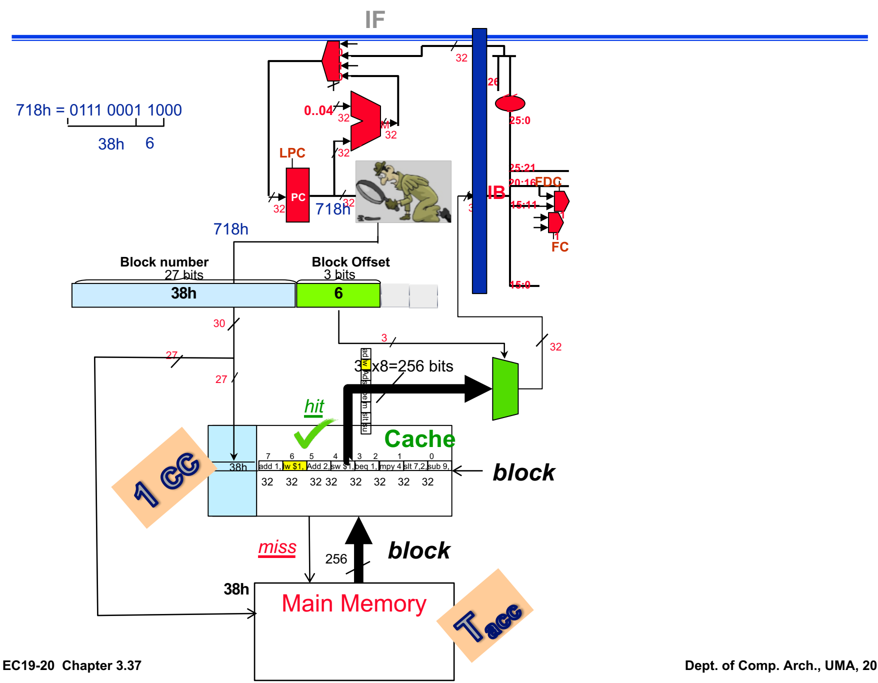
Example: Read instruction at address 718h

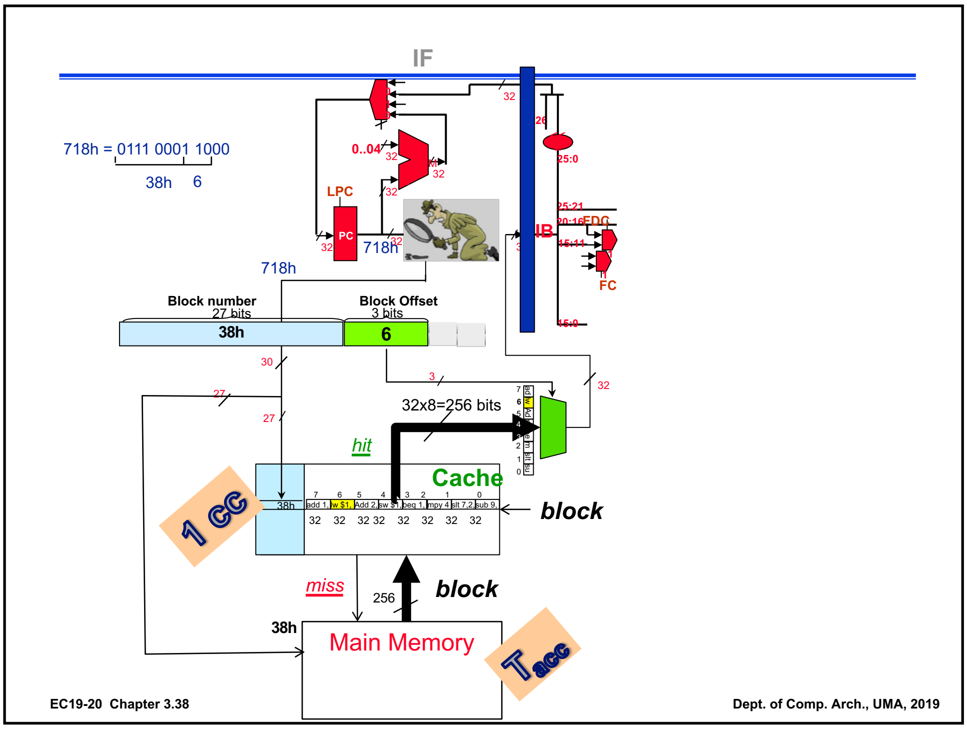


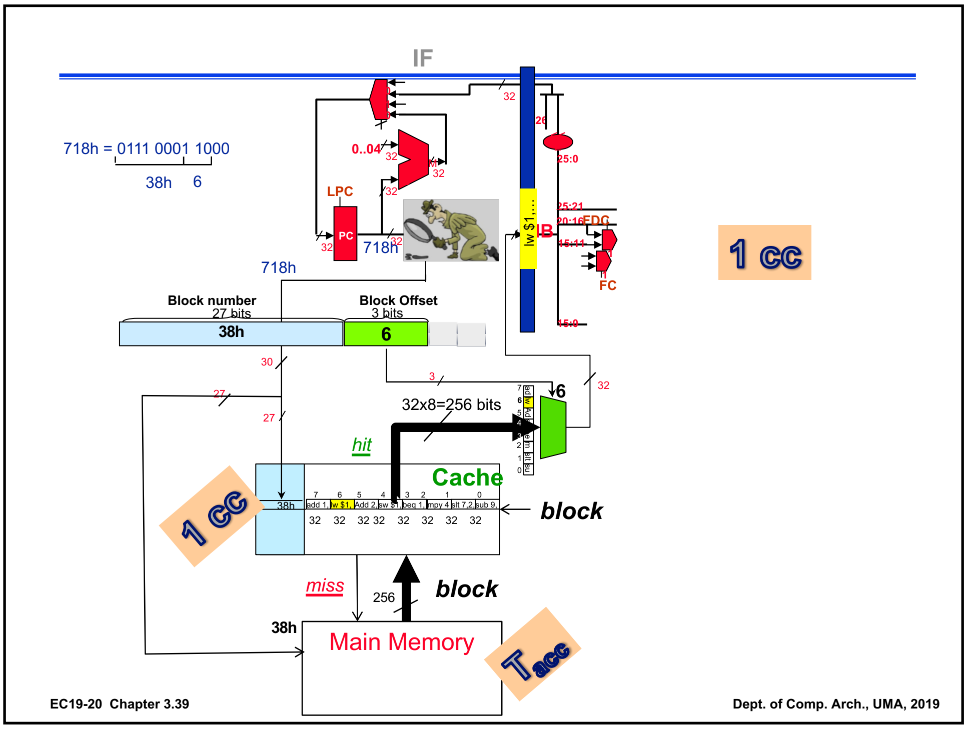
Block: 8 words
Word: 4 bytes

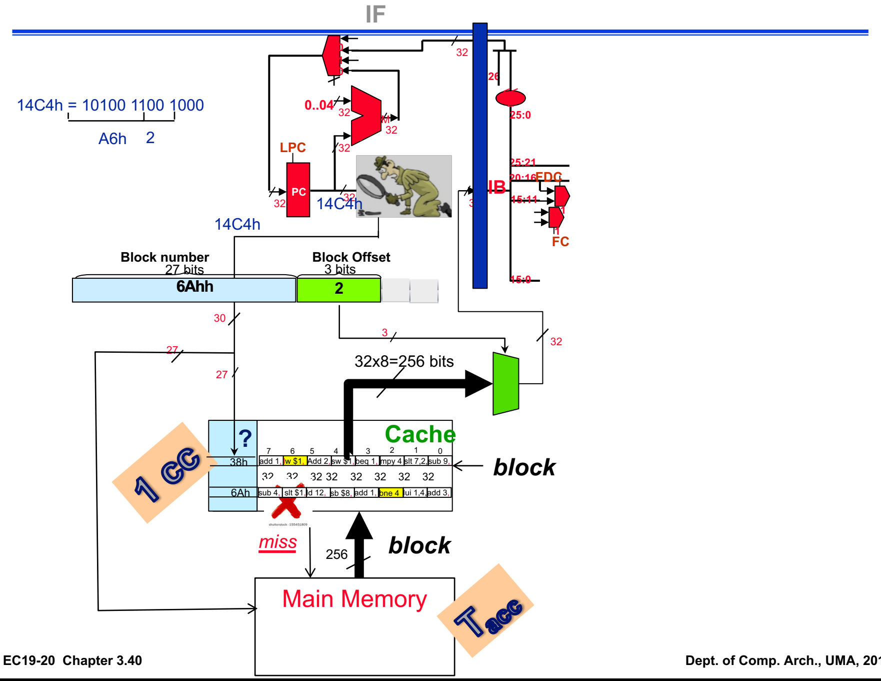


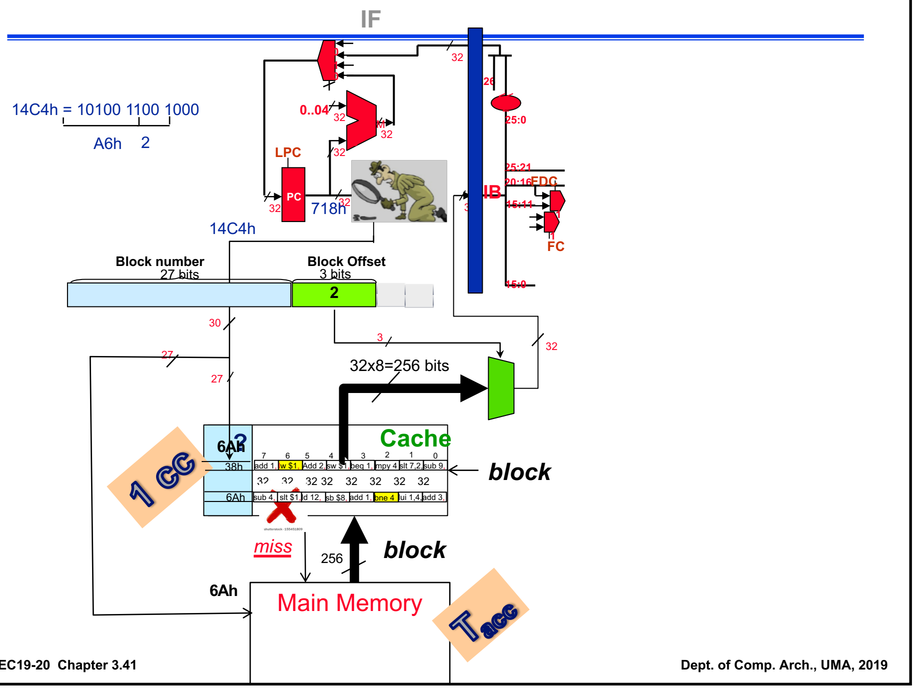

IF
718h = 00111 0001 1000
38h
6
0..04
32
LPC
PC
718h
718h
IB
EDG
FC
25:0
25:21
20:16
15:11
15:0
26
Block number
27 bits
38h
Block Offset
3 bits
6
30
27
27
3
32
32x8=256 bits
hit
?
38h
Cache
7 6 5 4 3 2 1 0
add 1, lw $1, Add 2, sw $1, beq 1, mpy 4 slt 7,2 sub 9,
32 32 32 32 32 32 32 32
block
1 CC
miss
256
block
Main Memory
Tacc

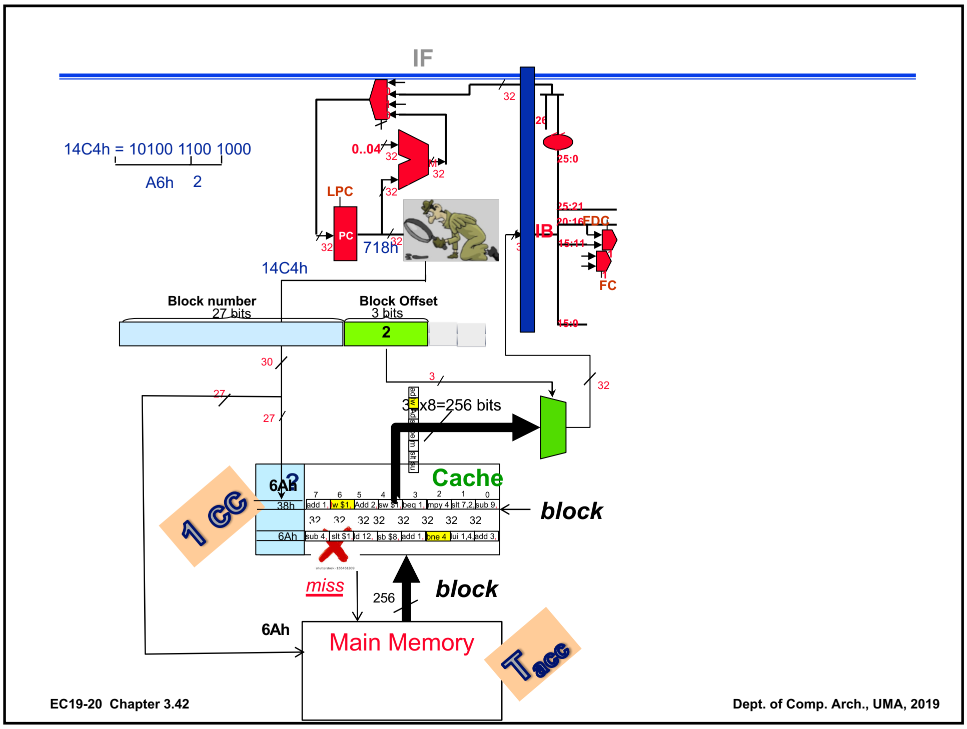

IF
718h = 0111 0001 1000
38h
6
0..04
32
LPC
PC
718h
718h
IB
EDG
FC
25:0
25:21
20:16
15:11
15:0
26
Block number
27 bits
38h
Block Offset
3 bits
6
30
27
27
3
32
32x8=256 bits
hit
38h
Cache
7 6 5 4 3 2 1 0
add 1, lw $1, Add 2, sw $1, beq 1, mpy 4 slt 7,2 sub 9,
32 32 32 32 32 32 32 32
block
1 CC
miss
256
block
38h
Main Memory
Tacc


IF
718h = 0111 0001 1000
38h 6
0..04
LPC
PC
718h
IB
EDG
FC
25:0
25:21
20:16
15:11
15:0
Block number
27 bits
38h
Block Offset
3 bits
6
30
27
3
32
32x8=256 bits
hit
Cache
38h
add 1, lw $1, Add 2, sw $1, beq 1, mpy 4 slt 7,2 sub 9,
block
miss
256
block
38h
Main Memory
1 CC
Tacc


IF
718h = 0111 0001 1000
38h 6
0..04
LPC
PC
718h
lw $1, ...
IB
EDG
FC
25:0
25:21
20:16
15:11
15:0
Block number
27 bits
38h
Block Offset
3 bits
6
30
27
3
6
32
32x8=256 bits
hit
Cache
38h
add 1, lw $1, Add 2, sw $1, beq 1, mpy 4 slt 7,2 sub 9,
block
miss
256
block
38h
Main Memory
1 CC
1 CC
Tacc

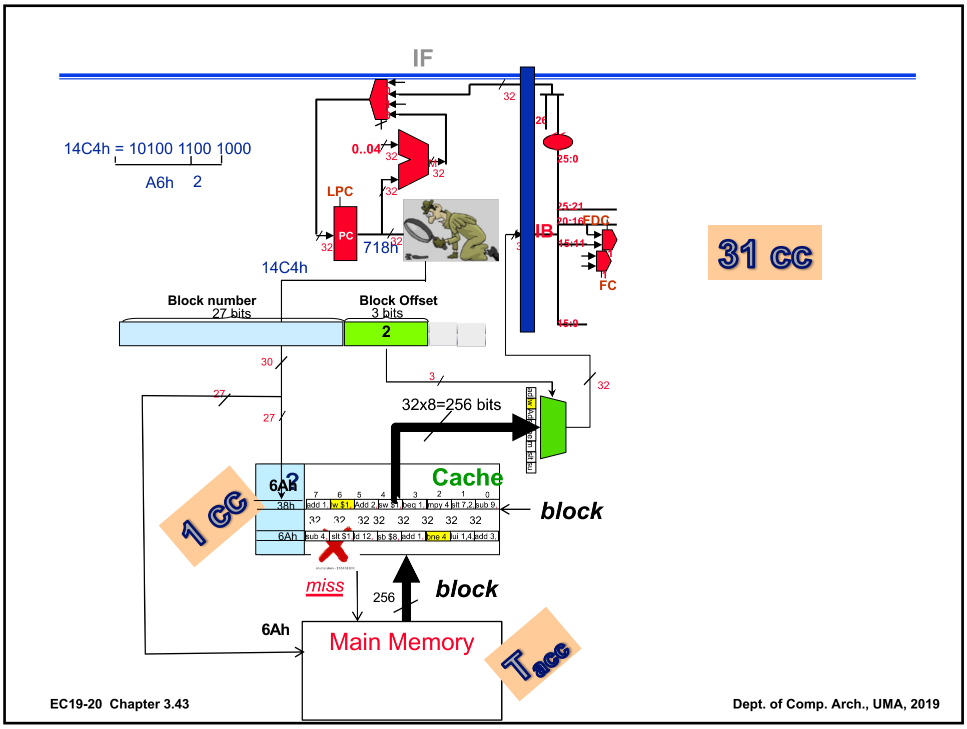

IF
14C4h = 10100 1100 1000
A6h 2
0..04
32
LPC
PC
14C4h
14C4h
26
25:0
25:21
20:16
FDC
15:11
FC
IB
15:0
32
Block number
27 bits
6Ahh
Block Offset
3 bits
2
30
3
32
27
27
32x8=256 bits
Cache
?
7 6 5 4 3 2 1 0
38h
add 1, lw $1, Add 2, sw $1, beq 1, mpy 4, slt 7,2, sub 9,
32 32 32 32 32 32 32 32
6Ah
sub 4, slt $1, ld 12, sb $8, add 1, bne 4, lui 1,4, add 3,
block
1 CC
miss
256
block
Main Memory
Tacc


IF
14C4h = 10100 1100 1000
A6h 2
0..04
32
LPC
PC
718h
14C4h
26
25:0
25:21
20:16
FDC
15:11
FC
IB
15:0
32
Block number
27 bits
Block Offset
3 bits
2
30
3
32
27
27
32x8=256 bits
Cache
6Ah
?
7 6 5 4 3 2 1 0
38h
add 1, lw $1, Add 2, sw $1, beq 1, mpy 4, slt 7,2, sub 9,
32 32 32 32 32 32 32 32
6Ah
sub 4, slt $1, ld 12, sb $8, add 1, bne 4, lui 1,4, add 3,
block
1 CC
miss
256
block
6Ah
Main Memory
Tacc



EC19-20 Chapter 3.42

Dept. of Comp. Arch., UMA, 2019



EC19-20 Chapter 3.43

Dept. of Comp. Arch., UMA, 2019

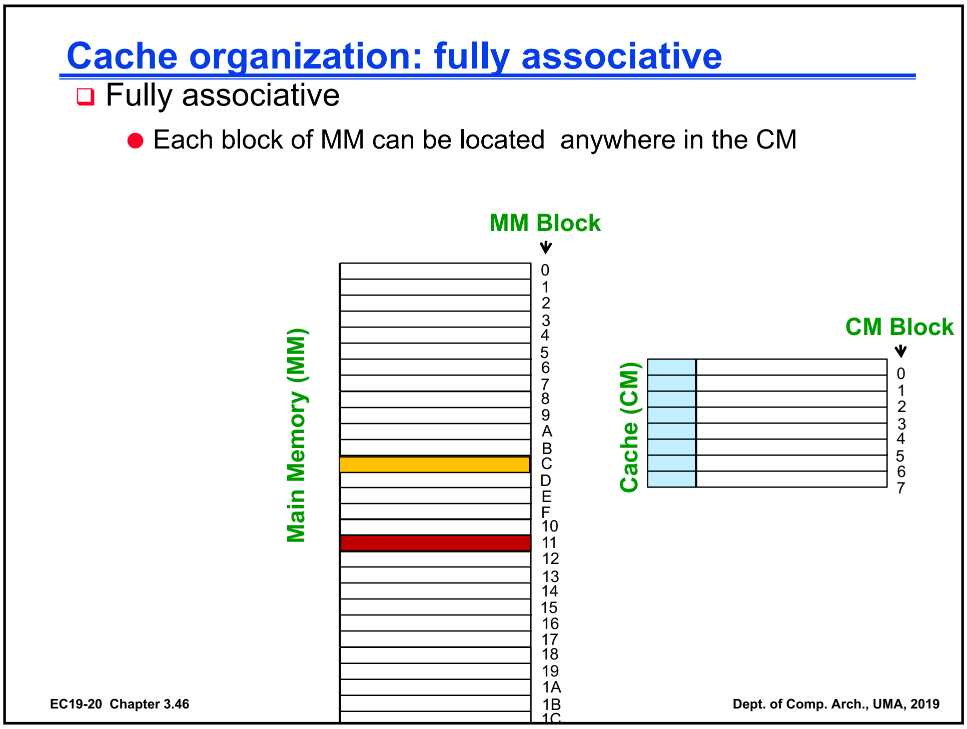
How is the Hierarchy Managed?

- registers ↔ memory
  - by compiler (programmer?)
- cache ↔ main memory
  - by the cache controller hardware
- main memory ↔ disks
  - by the operating system (virtual memory)
  - virtual to physical address mapping assisted by the hardware (TLB)
  - by the programmer (files)

Cache organization

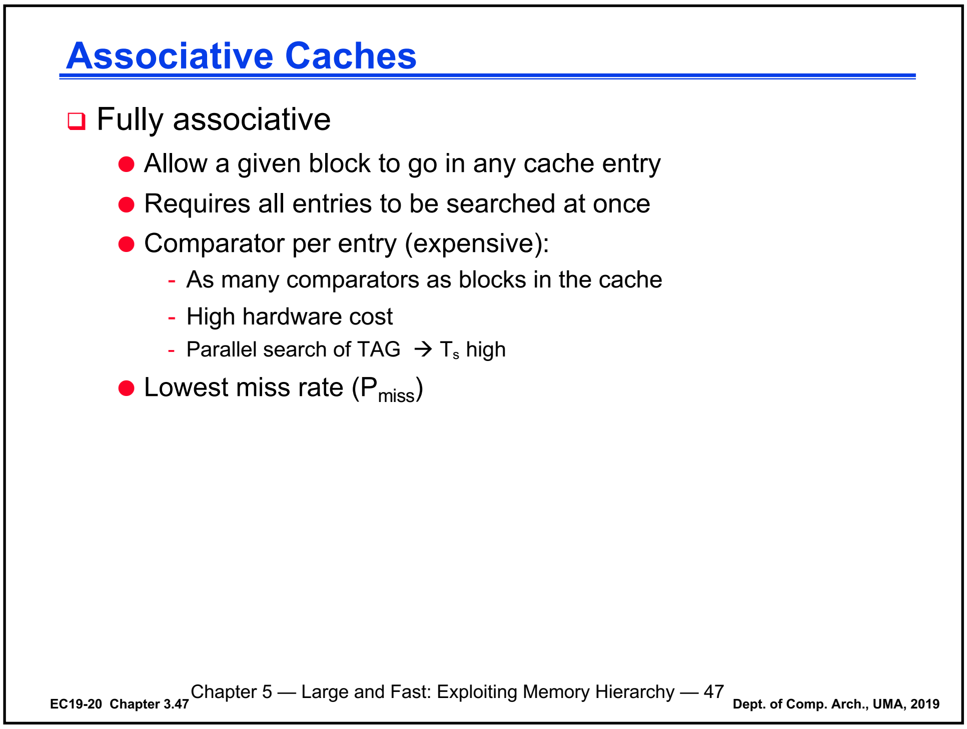
Depending on the way that a block of main memory is located in the cache memory we have:

- Fully associative
- Direct mapped
- Set associative



Cache organization: fully associative

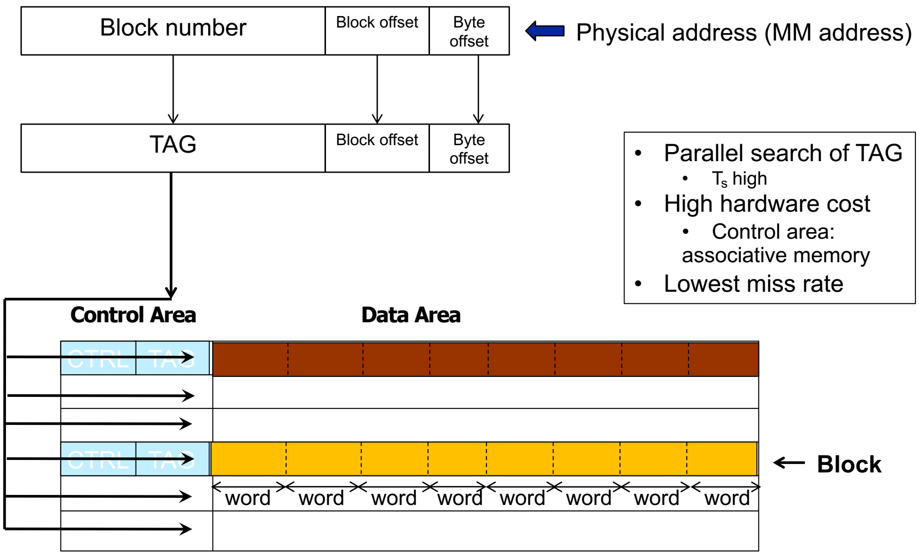
- ❑ Fully associative
  - Each block of MM can be located anywhere in the CM



Associative Caches

- ❑ Fully associative
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
  - Comparator per entry (expensive):
    - As many comparators as blocks in the cache
    - High hardware cost
    - Parallel search of TAG → T_s high
  - Lowest miss rate (P_{miss})

Fully associative: Address Subdivision

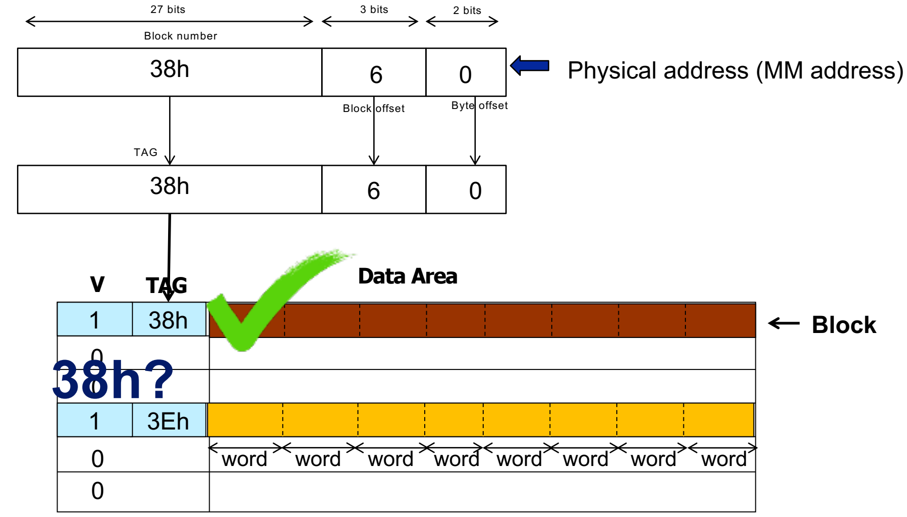


- Parallel search of TAG
  - T_s high
- High hardware cost
  - Control area: associative memory
- Lowest miss rate

Fully associative: Address Subdivision

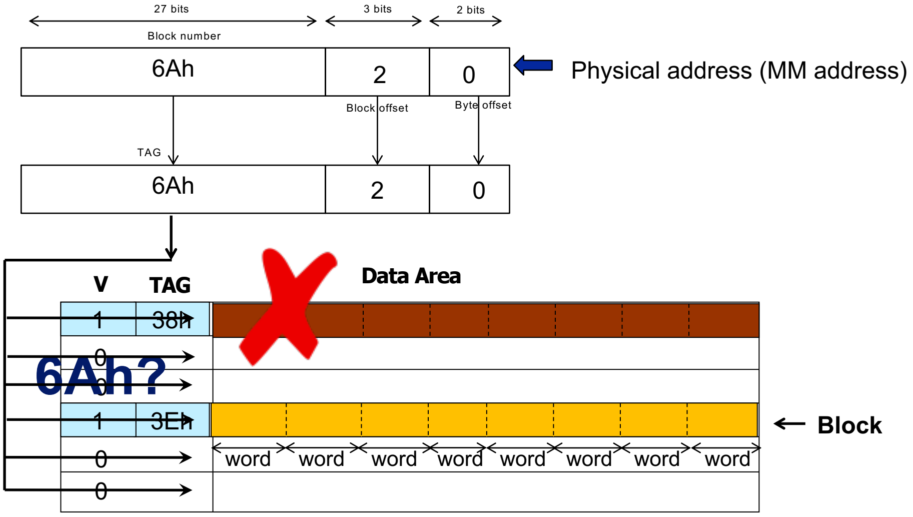
Ej. MM: 2^{32}bytes, block:8 words, word: 4 bytes
Physical addres: 718h → Hit

718h = 00111 0001 1000
38h 6 0



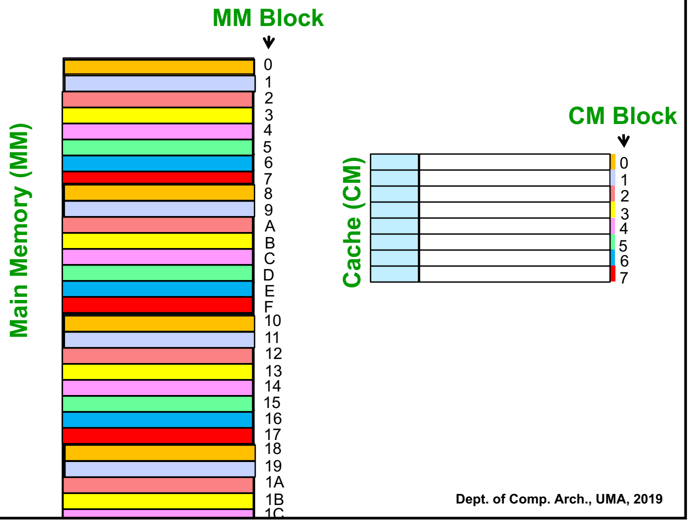
Fully associative: Address Subdivision

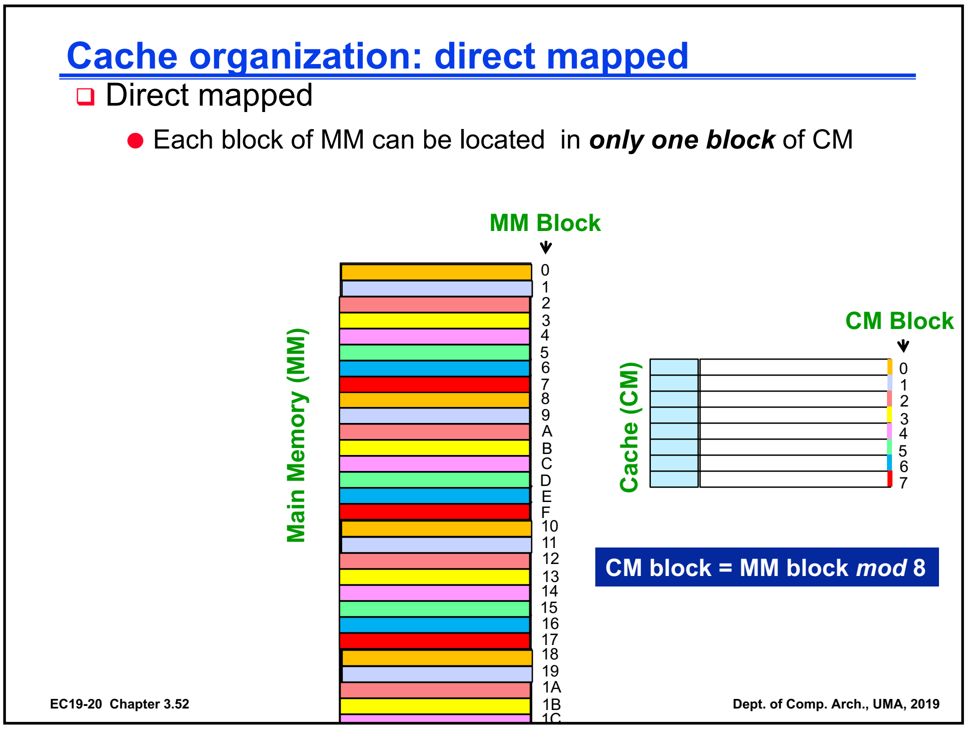
Ej. MM: 2^{32}bytes, block:8 words, word: 4 bytes
Physical addres: 14C4h → Miss



Cache organization: direct mapped

- Direct mapped
  - Each block of MM can be located in only one block of CM

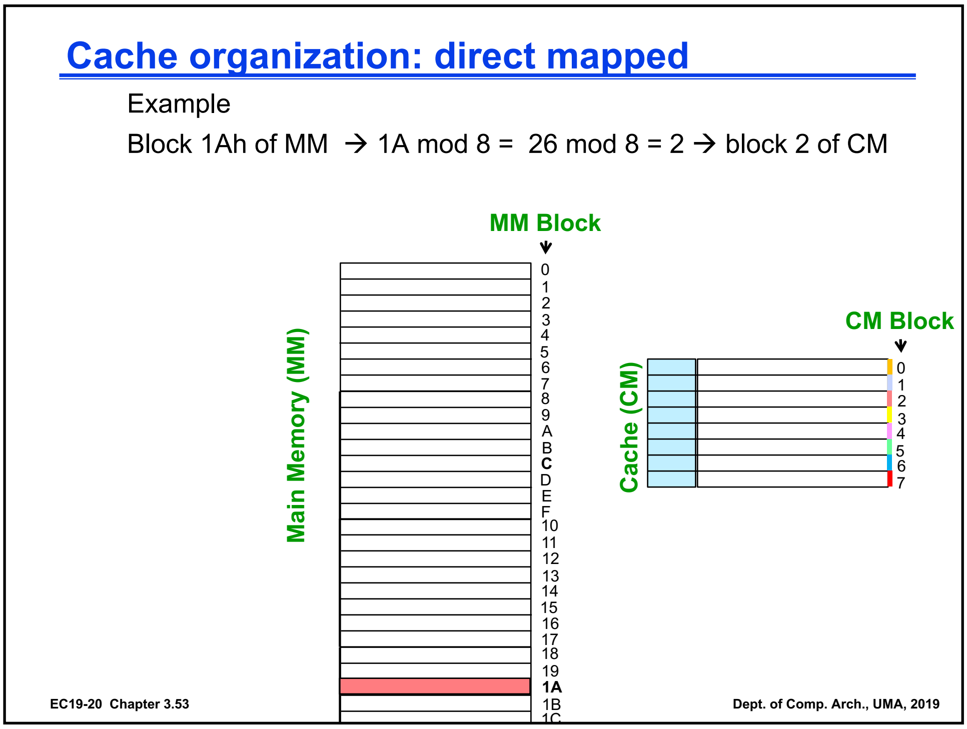




Cache organization: direct mapped

- Direct mapped
  - Each block of MM can be located in ***only one block*** of CM

CM block = MM block *mod* 8



Cache organization: direct mapped

Example

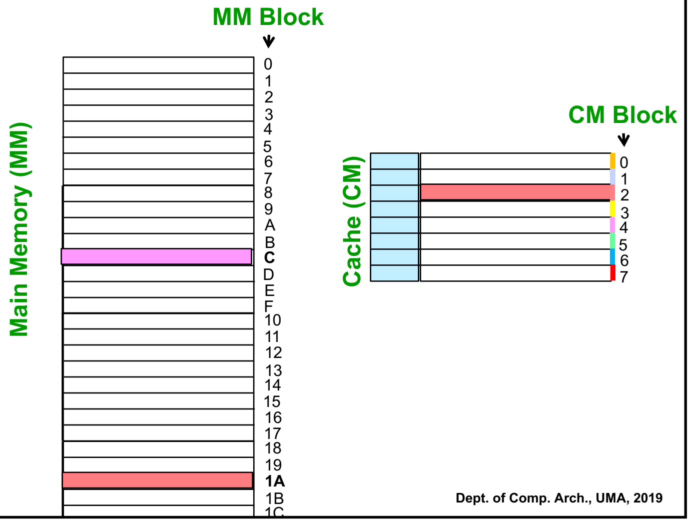
Block 1Ah of MM → 1A mod 8 = 26 mod 8 = 2 → block 2 of CM

Cache organization: direct mapped

Example

Block 1Ah of MM → 1A mod 8 = 26 mod 8 = 2 → block 2 of CM

Block Ch of MM → C mod 8 = 12 mod 8 = 4 → block 4 of CM

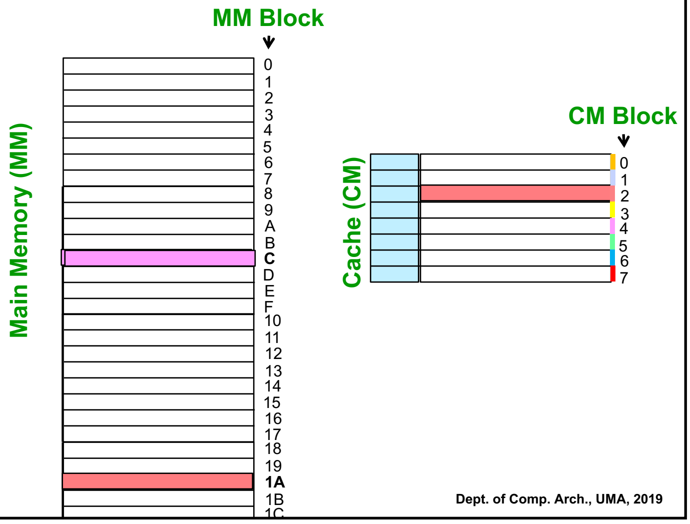


Cache organization: direct mapped

Example

Block 1Ah of MM → 1A mod 8 = 26 mod 8 = 2 → block 2 of CM

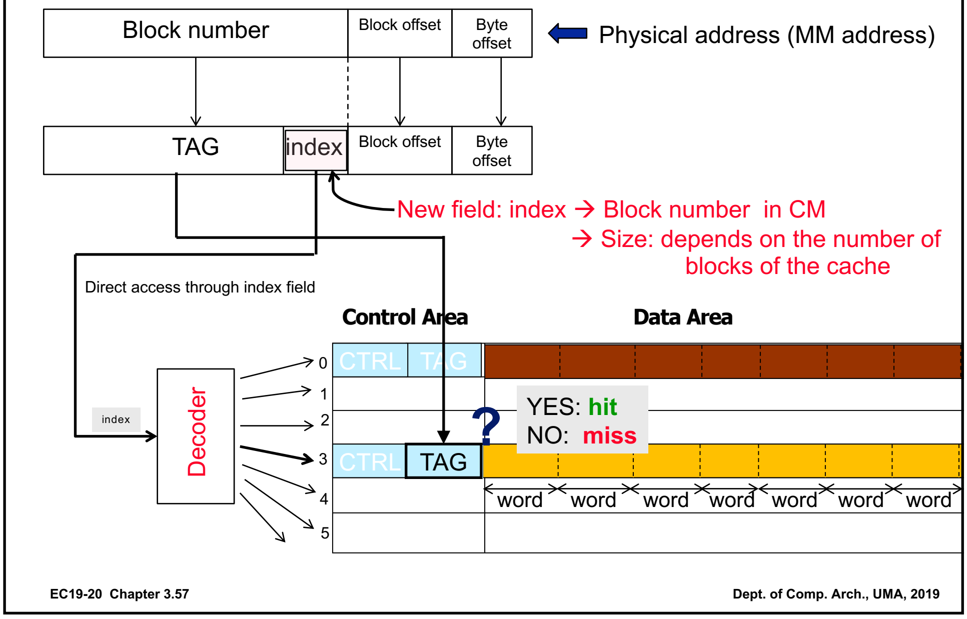
Block Ch of MM → C mod 8 = 12 mod 8 = 4 → block 4 of CM



Direct mapped Caches

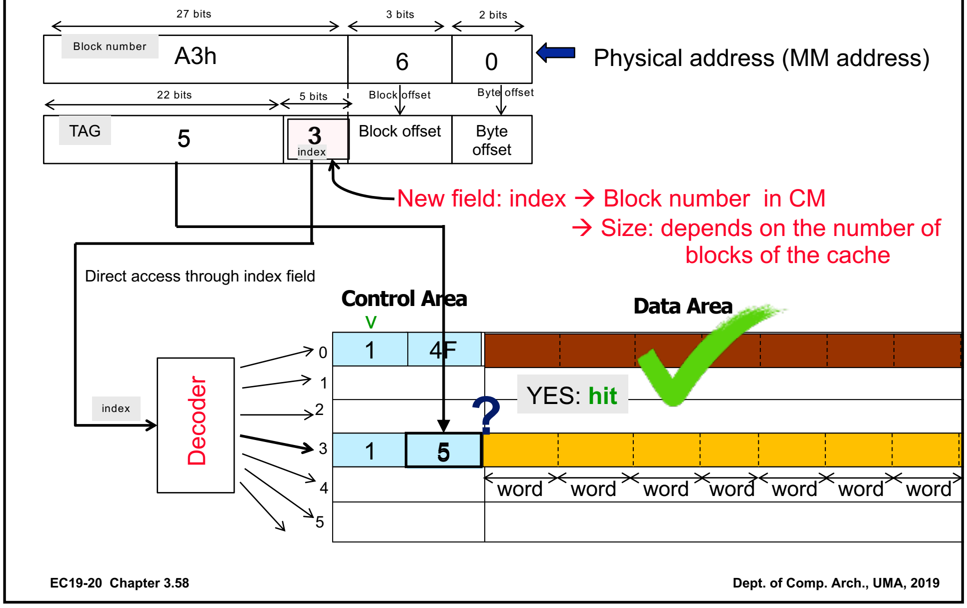
- Direct mapped
  - Only a choice for each block
  - Only one entry is searched
  - Only one comparator (cheap):
    - Low hardware cost
  - Higest miss rate (P_{miss})

Direct mapped: Address Subdivision



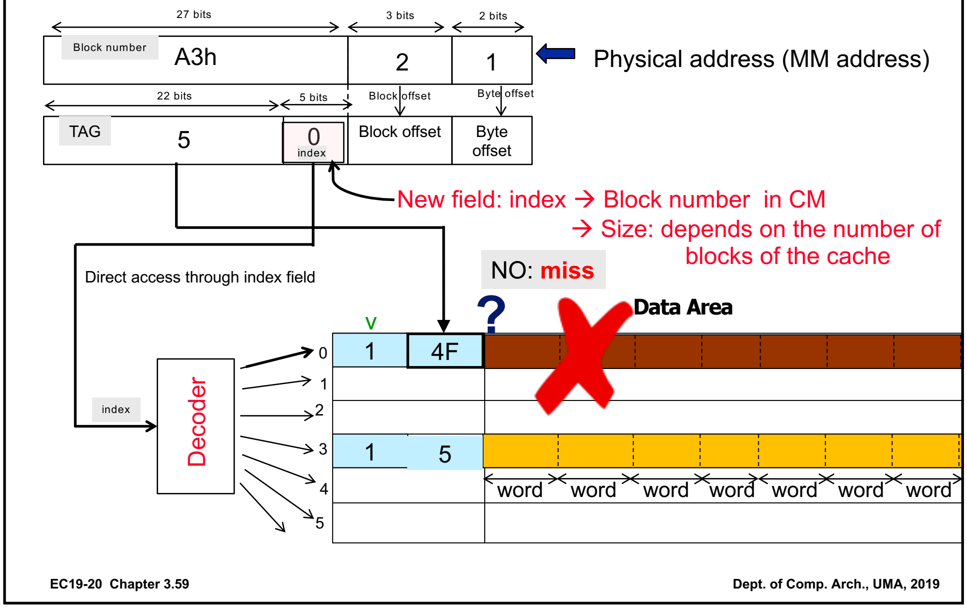
New field: index → Block number in CM
→ Size: depends on the number of blocks of the cache

Example: MM: 2^{32}bytes, Cache: 32 blocks (2^5), block:8 words, word: 4 bytes
Physical addres: 1478h → **Hit**



New field: index → Block number in CM
→ Size: depends on the number of blocks of the cache

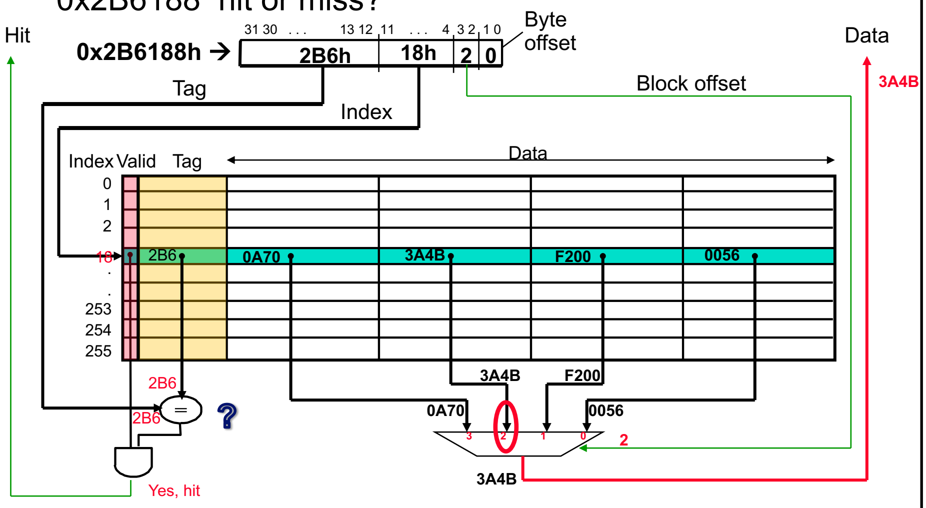
Example: MM: 2^{32}bytes, Cache: 32 blocks, block:8 words, word: 4 bytes
Physical addres: 1409h → **Miss**



New field: index → Block number in CM
→ Size: depends on the number of blocks of the cache

Another example Direct Mapped Cache

- CM-1Kwords, Block-4 words, Word-4 bytes. Address 0x2B6188 hit or miss?

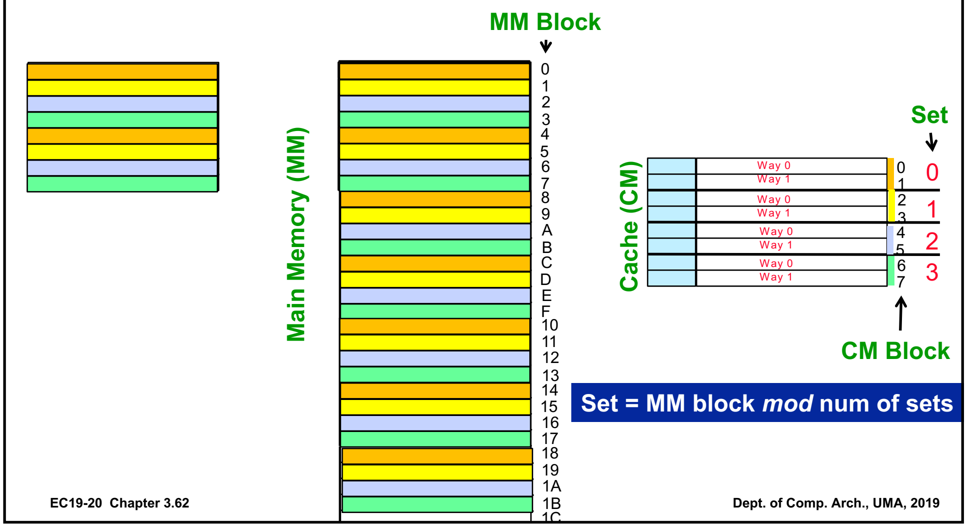


Cache organization: Set associative

- The CM is divided in sets
- Each set has several blocks
- Each MM block can be located in only one set of CM
  - Inside a set, the block can be allocated anywhere
  - Thus, set associative is
    - ✓ Direct mapped between MM blocks and MC sets
    - ✓ Fully associative inside a set
- The block inside a set is also call ***way***
  - Examples
    - 2-way set associative → 2 blocks per set
    - 3-way set associative → 3 blocks per set
- Extreme cases:
  - 1-way set associative → direct mapped
  - 2^k-way set associtive → fully associative (2^k blocks in CM)

Cache organization: Set associative

- Direct mapped
  - Each block of MM can be located in only one SET of CM
  - Example with 2-way set associative, 4 sets in CM (8 blocks in CM)



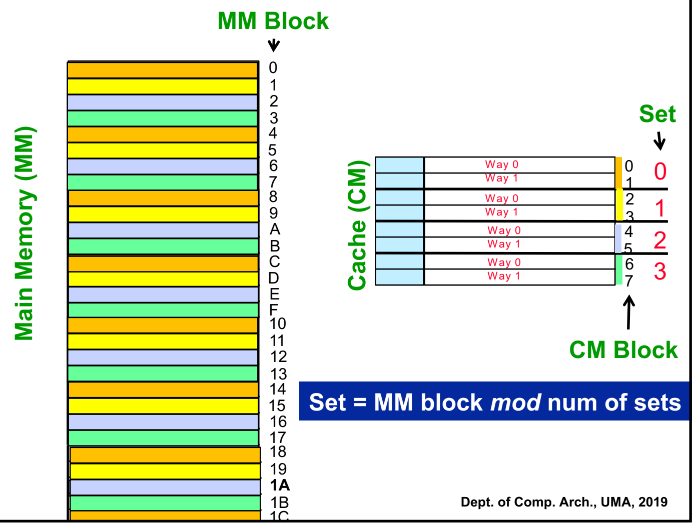
Set = MM block *mod* num of sets

Cache organization: Set associative

Example

Block 1Ah of MM → 1A mod 4 = 26 mod 4 = 2 → SET 2 of CM

Block Ch of MM → C mod 4 = 12 mod 4 = 0 → SET 0 of CM

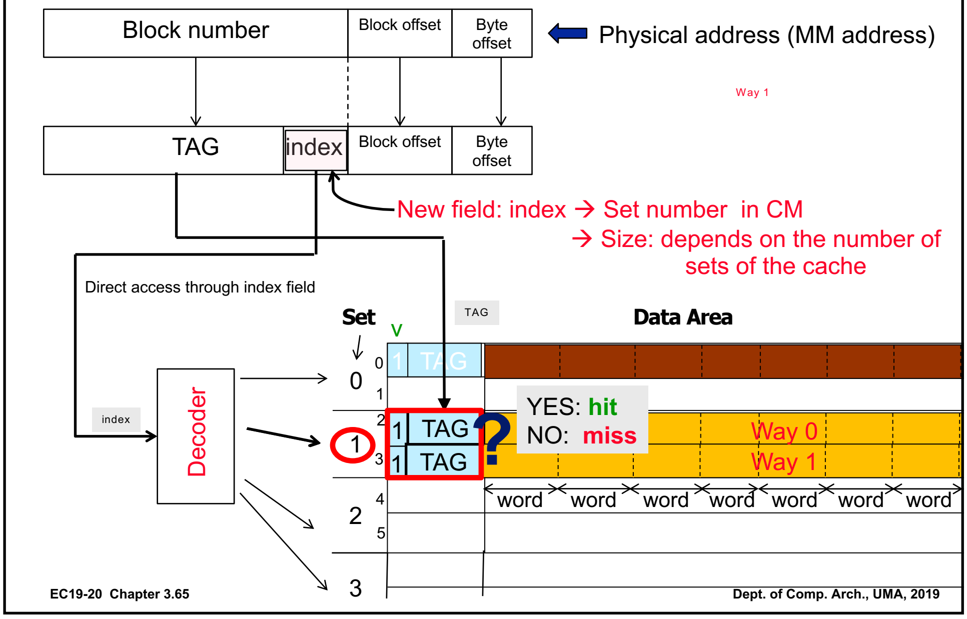


Set = MM block *mod* num of sets

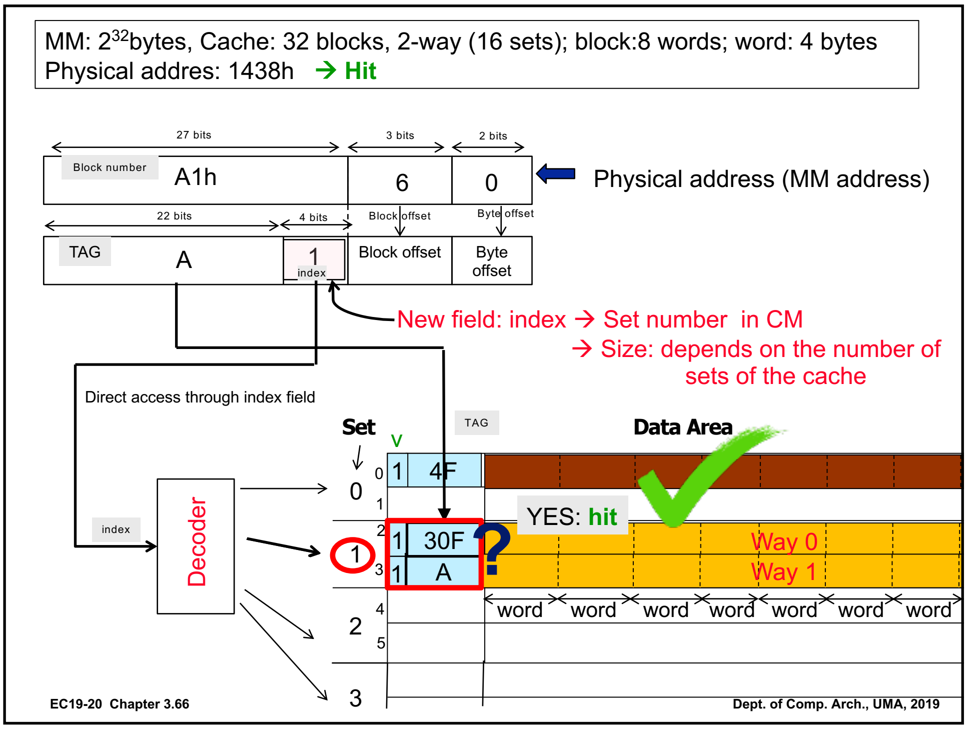
Cache organization: Set associative

- Set associative
  - As many comparators as ways:
    - Lower hardware cost than full associative, higher hardware cost than direct mapped (trade-off solution)
  - Medium miss rate (P_{miss})
  - From a practical point of view, a 8-way set associative has a P_{miss} similar to a fully associative

Set associative: Address Subdivision



New field: index → Set number in CM
→ Size: depends on the number of sets of the cache



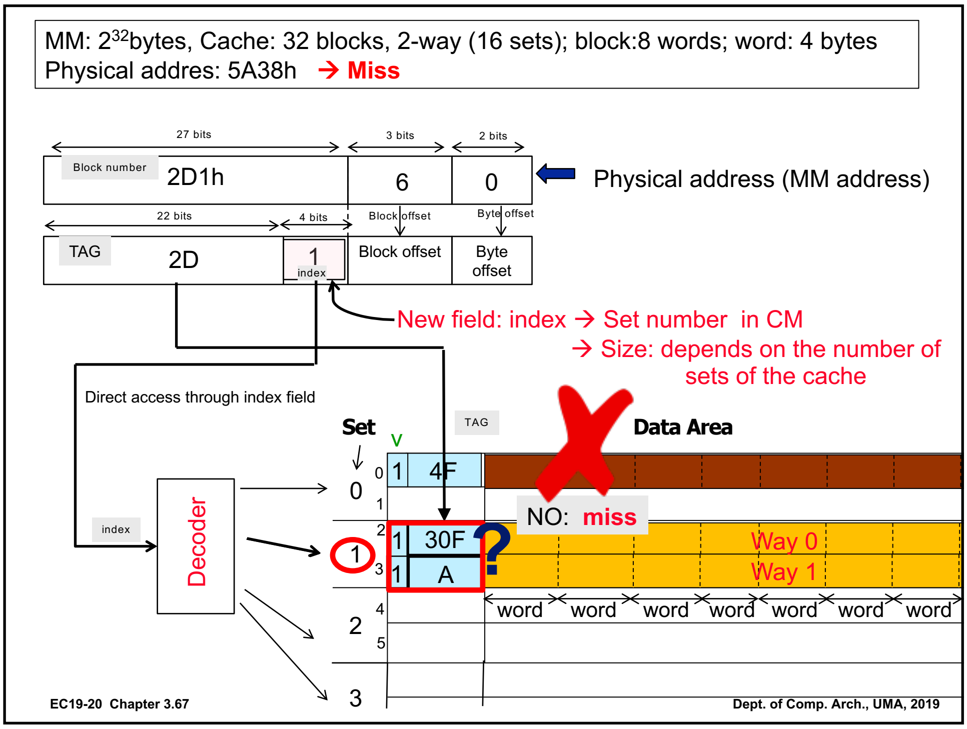
MM: 2^{32}bytes, Cache: 32 blocks, 2-way (16 sets); block:8 words; word: 4 bytes
Physical addres: 1438h → **Hit**

Physical address (MM address)

New field: index → Set number in CM
→ Size: depends on the number of sets of the cache

Direct access through index field

YES: **hit**



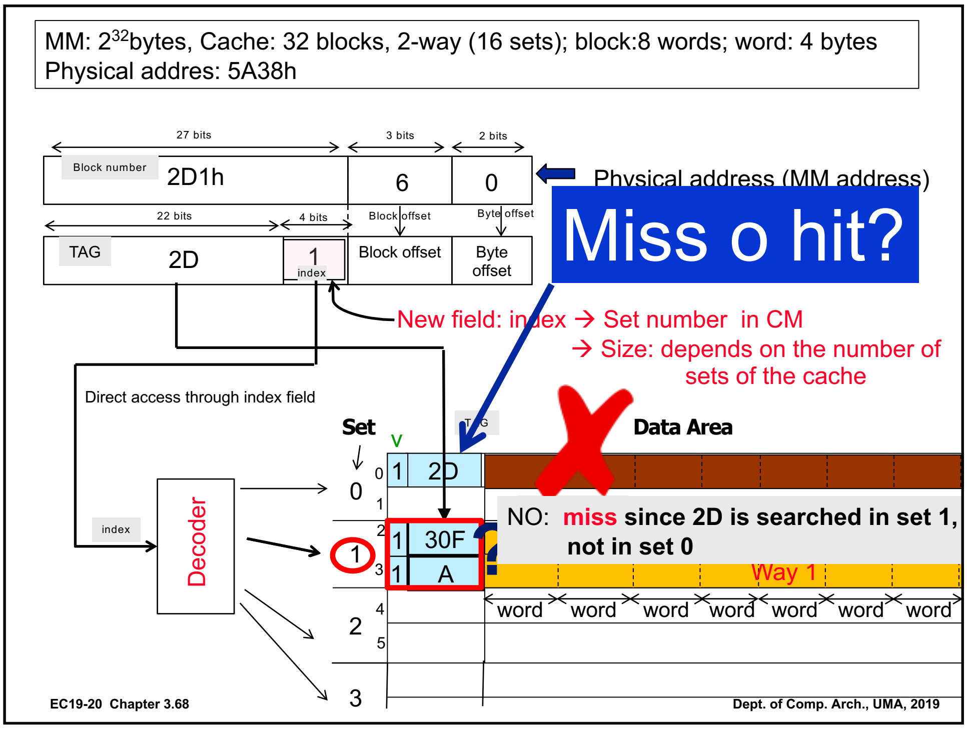
MM: 2^{32}bytes, Cache: 32 blocks, 2-way (16 sets); block:8 words; word: 4 bytes
Physical addres: 5A38h → **Miss**

Physical address (MM address)

New field: index → Set number in CM
→ Size: depends on the number of sets of the cache

Direct access through index field

NO: **miss**



MM: 2^{32}bytes, Cache: 32 blocks, 2-way (16 sets); block:8 words; word: 4 bytes
Physical addres: 5A38h

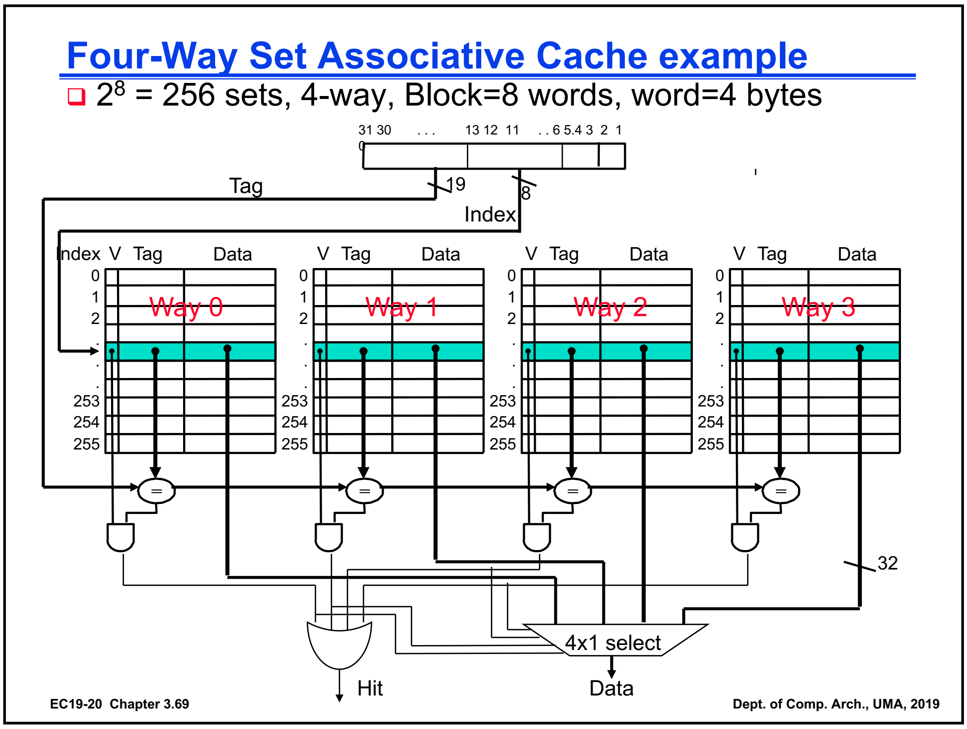
Miss o hit?

Physical address (MM address)

New field: index → Set number in CM
→ Size: depends on the number of sets of the cache

Direct access through index field

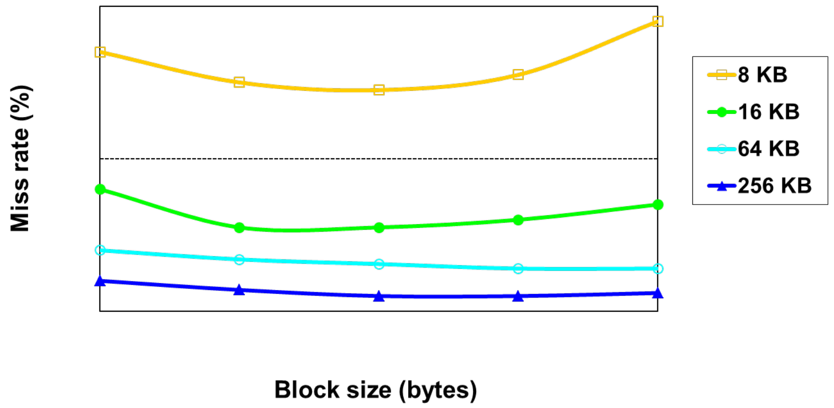
NO: miss since 2D is searched in set 1, not in set 0



Four-Way Set Associative Cache example

- 2^8 = 256 sets, 4-way, Block=8 words, word=4 bytes

Miss Rate vs Block Size vs Cache Size



- Miss rate goes up if the block size becomes a significant fraction of the cache size because the number of blocks that can be held in the same size cache is smaller (increasing capacity misses)

Cache Field Sizes

- The number of bits in a cache includes both the storage for data and for the tags
  - 32-bit byte address
  - For a direct mapped cache with 2^n blocks, *n* bits are used for the index
  - MIPS: For a block size of 2^w words (2^{w+2} bytes), *w* bits are used to address the word within the block and 2 bits are used to address the byte within the word
- What is the size of the tag field?
- The total number of bits in a direct-mapped cache is then

$$2^N \text{ x (block size + tag field size + valid field size)}$$

- How many total bits are required for a direct mapped cache with 16KB of data and 4-word blocks assuming a 32-bit address?

Reducing Cache Miss Rates #1

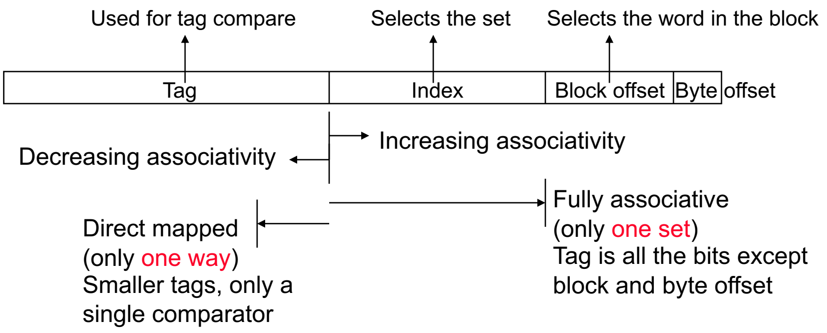
1. Allow more flexible block placement

- In a direct mapped cache a memory block maps to exactly one cache block
- At the other extreme, could allow a memory block to be mapped to *any* cache block – fully associative cache
- A compromise is to divide the cache into sets each of which consists of n “ways” (n-way set associative). A memory block maps to a unique set (specified by the index field) and can be placed in any way of that set (so there are n choices)

(block address) modulo (# sets in the cache)

Range of Set Associative Caches

- For a fixed size cache, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number or ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit

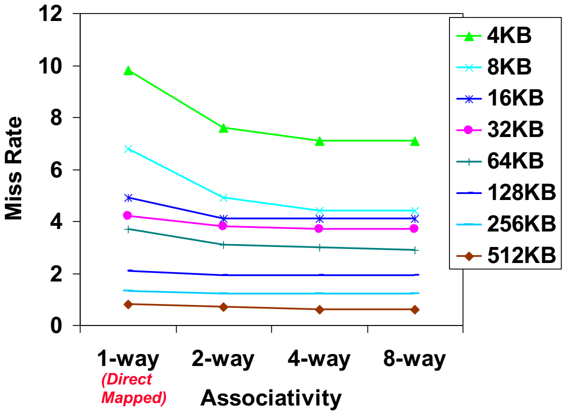


Sources of Cache Misses

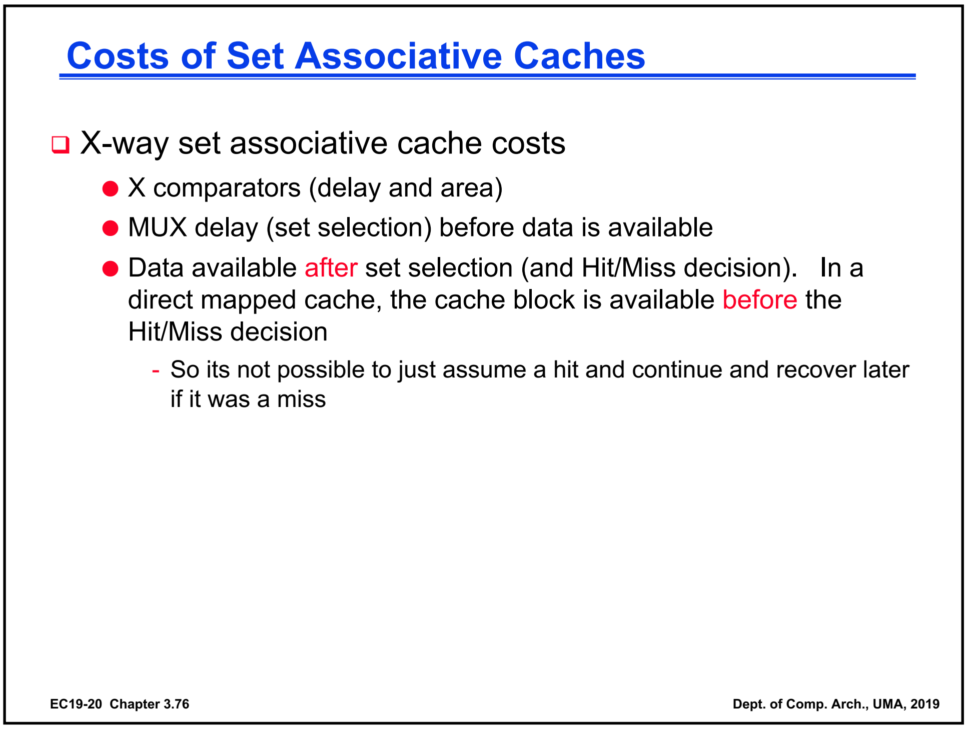
- ❑ Compulsory (cold start or process migration, first reference):
  - ● First access to a block, “cold” fact of life, not a whole lot you can do about it. If you are going to run “millions” of instruction, compulsory misses are insignificant
  - ● Solution: increase block size (increases miss penalty; very large blocks could increase miss rate)
- ❑ Capacity:
  - ● Cache cannot contain all blocks accessed by the program
  - ● Solution: increase cache size (may increase access time)
- ❑ Conflict (collision):
  - ● Multiple memory locations mapped to the same cache location
  - ● Solution 1: increase cache size
  - ● Solution 2: increase associativity (stay tuned) (may increase access time)

Benefits of Set Associative Caches

- ❑ The choice of direct mapped or set associative depends on the cost of a miss versus the cost of implementation

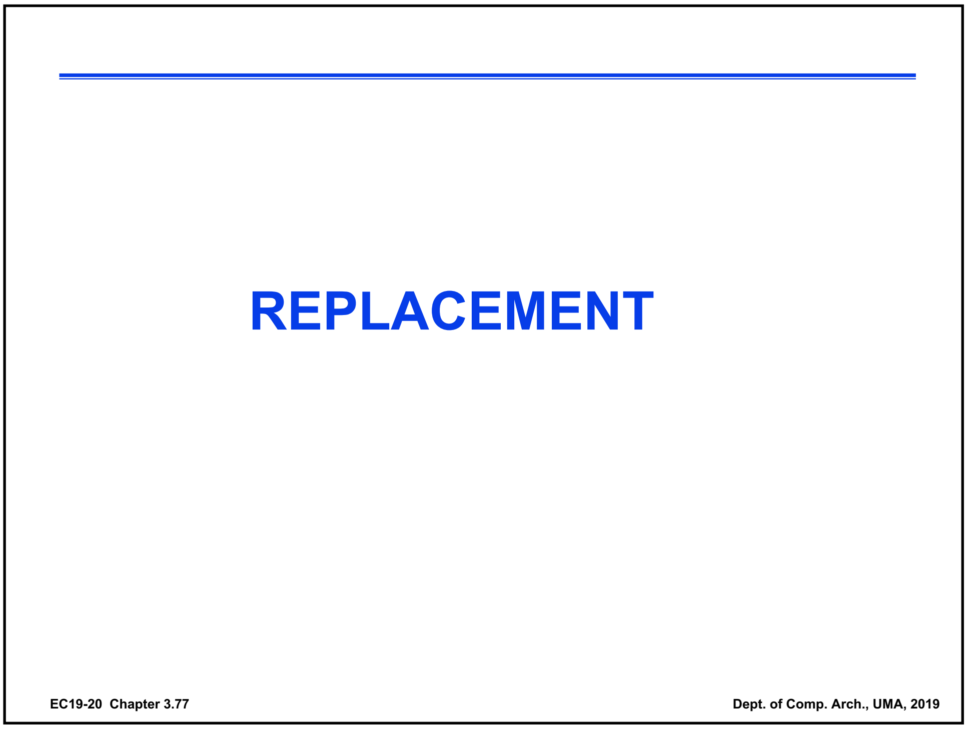


- ❑ Largest gains are in going from direct mapped to 2-way (20%+ reduction in miss rate)

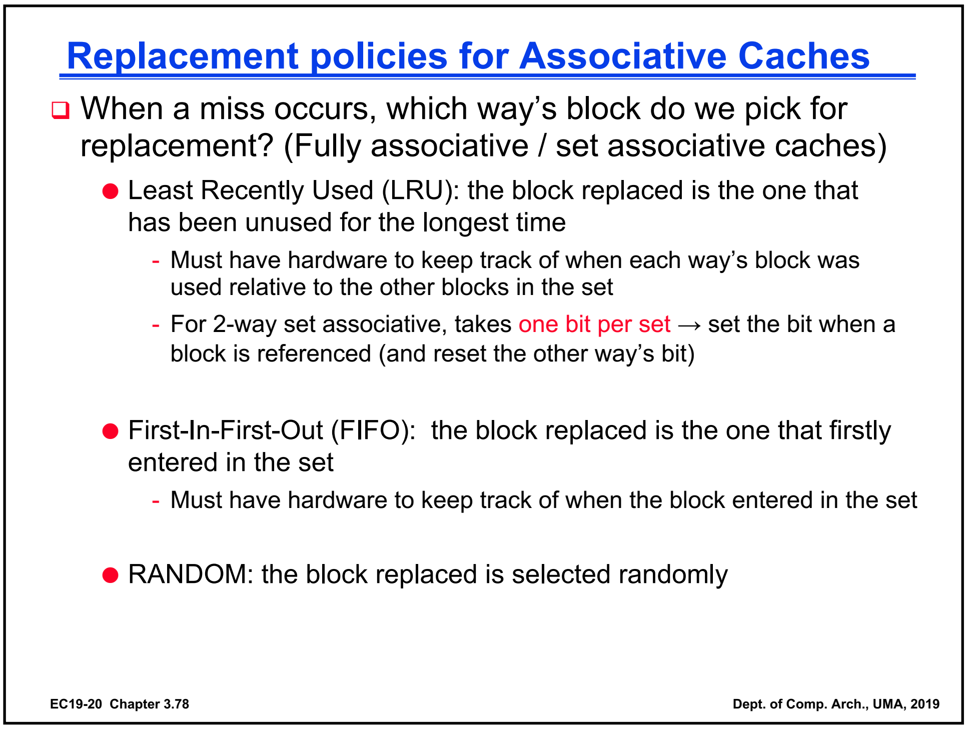


Costs of Set Associative Caches

- ❑ X-way set associative cache costs
  - ● X comparators (delay and area)
  - ● MUX delay (set selection) before data is available
  - ● Data available after set selection (and Hit/Miss decision). In a direct mapped cache, the cache block is available before the Hit/Miss decision
    - So its not possible to just assume a hit and continue and recover later if it was a miss

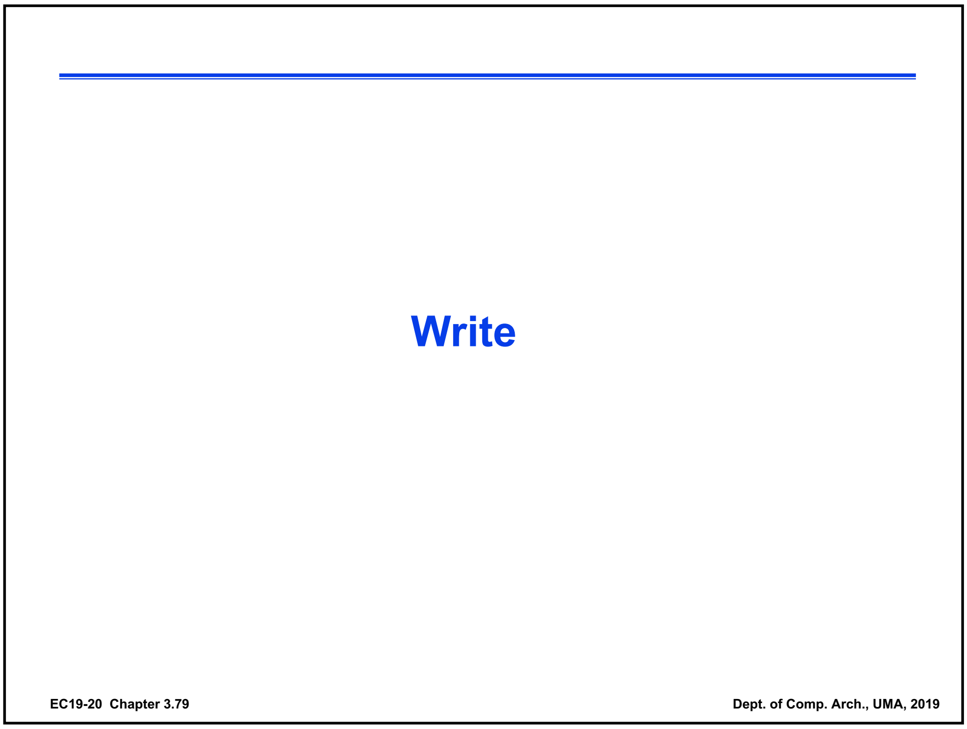


REPLACEMENT

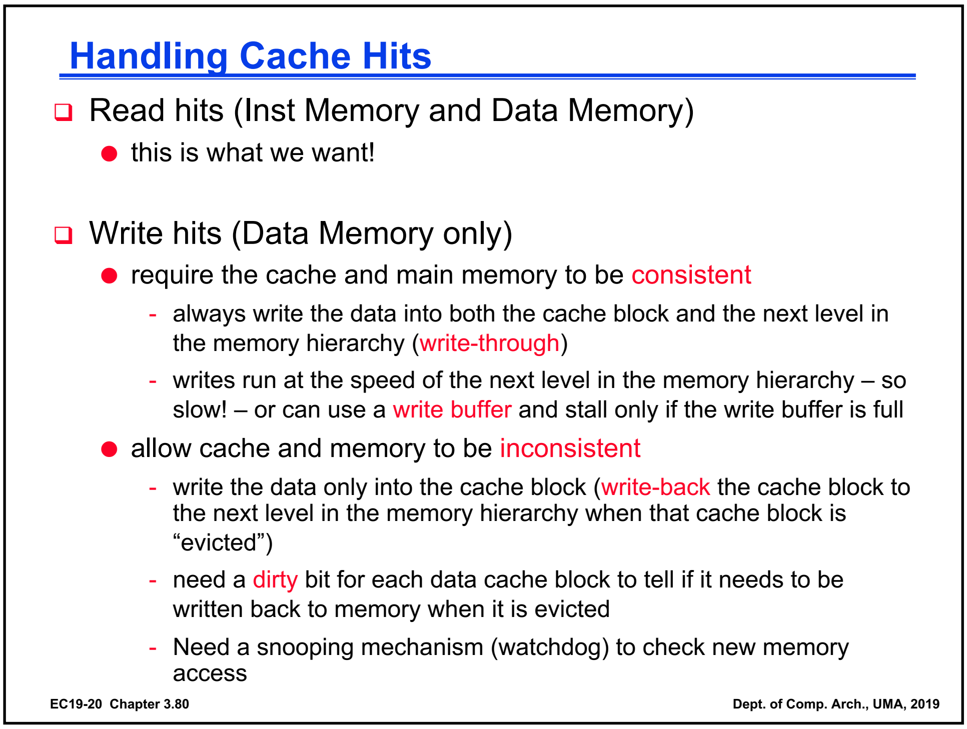


Replacement policies for Associative Caches

- When a miss occurs, which way's block do we pick for replacement? (Fully associative / set associative caches)
  - Least Recently Used (LRU): the block replaced is the one that has been unused for the longest time
    - Must have hardware to keep track of when each way's block was used relative to the other blocks in the set
    - For 2-way set associative, takes one bit per set → set the bit when a block is referenced (and reset the other way's bit)
  - First-In-First-Out (FIFO): the block replaced is the one that firstly entered in the set
    - Must have hardware to keep track of when the block entered in the set
  - RANDOM: the block replaced is selected randomly

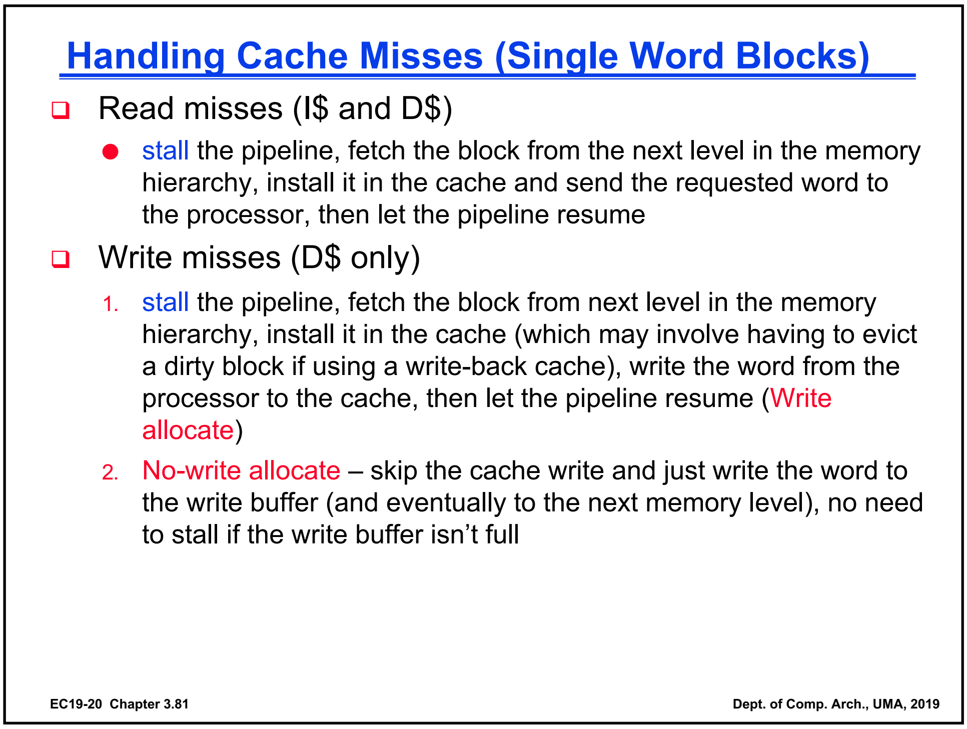


Write



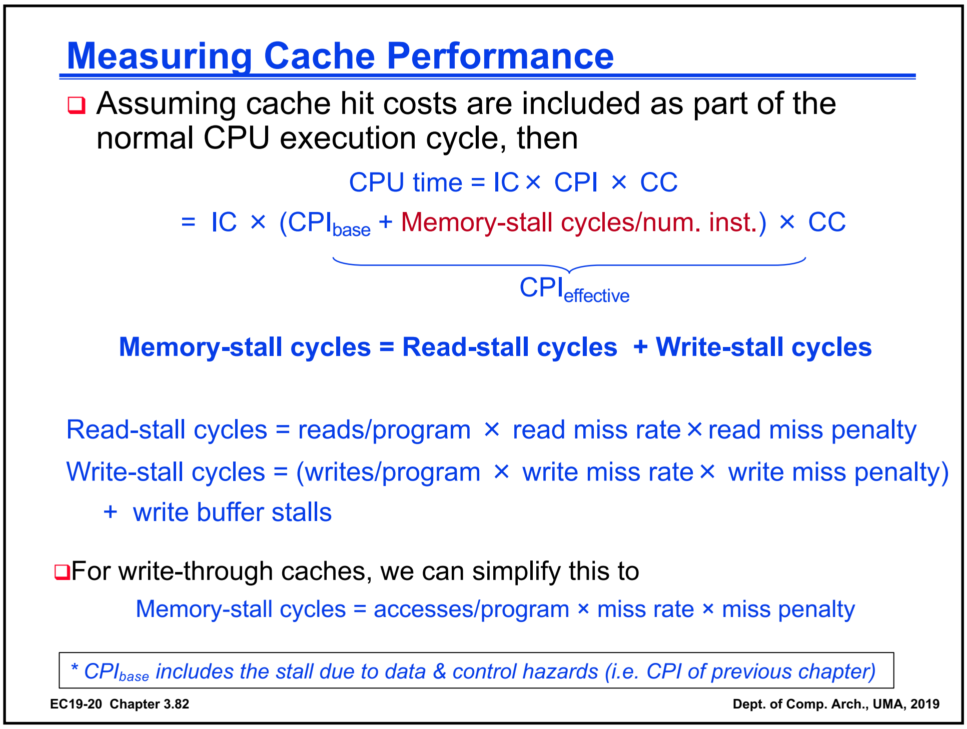
Handling Cache Hits

- ❑ Read hits (Inst Memory and Data Memory)
  - this is what we want!
- ❑ Write hits (Data Memory only)
  - require the cache and main memory to be consistent
    - always write the data into both the cache block and the next level in the memory hierarchy (write-through)
    - writes run at the speed of the next level in the memory hierarchy – so slow! – or can use a write buffer and stall only if the write buffer is full
  - allow cache and memory to be inconsistent
    - write the data only into the cache block (write-back the cache block to the next level in the memory hierarchy when that cache block is “evicted”)
    - need a dirty bit for each data cache block to tell if it needs to be written back to memory when it is evicted
    - Need a snooping mechanism (watchdog) to check new memory access



Handling Cache Misses (Single Word Blocks)

- ❑ Read misses (I$ and D$)
  - stall the pipeline, fetch the block from the next level in the memory hierarchy, install it in the cache and send the requested word to the processor, then let the pipeline resume
- ❑ Write misses (D$ only)
  1. stall the pipeline, fetch the block from next level in the memory hierarchy, install it in the cache (which may involve having to evict a dirty block if using a write-back cache), write the word from the processor to the cache, then let the pipeline resume (Write allocate)
  2. No-write allocate – skip the cache write and just write the word to the write buffer (and eventually to the next memory level), no need to stall if the write buffer isn’t full



Measuring Cache Performance

- ❑ Assuming cache hit costs are included as part of the normal CPU execution cycle, then

$$\text{CPU time} = \text{IC} \times \text{CPI} \times \text{CC}$$

$$= \text{IC} \times \underbrace{(\text{CPI}_{base} + \text{Memory-stall cycles/num. inst.})}_{\text{CPI}_{effective}} \times \text{CC}$$

Memory-stall cycles = Read-stall cycles + Write-stall cycles

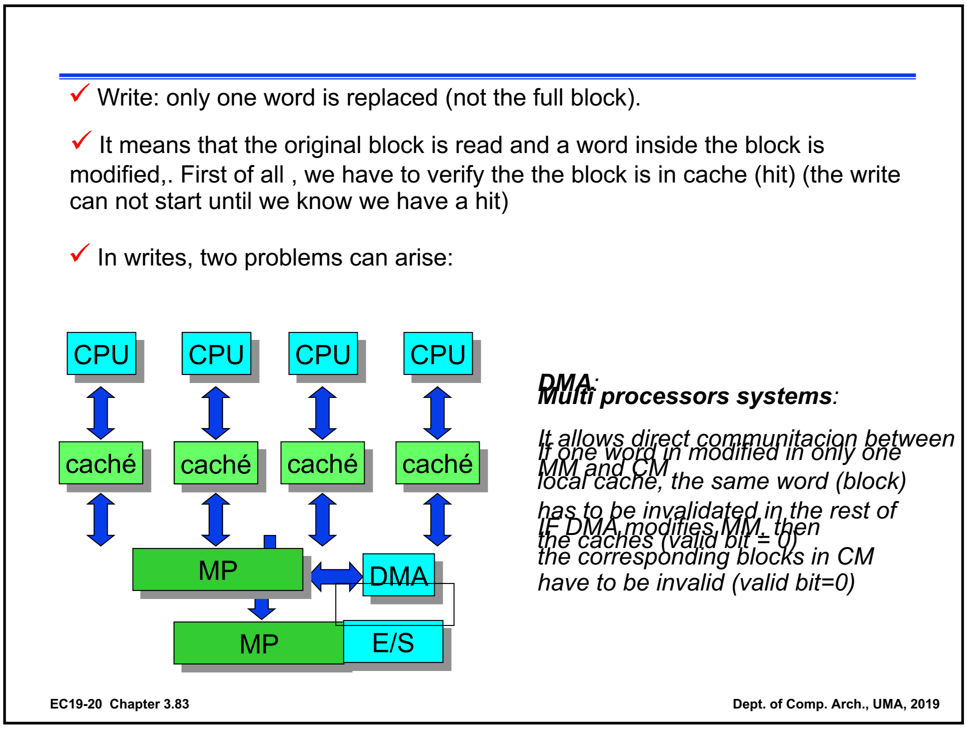
Read-stall cycles = reads/program × read miss rate × read miss penalty

Write-stall cycles = (writes/program × write miss rate × write miss penalty)
+ write buffer stalls

- ❑For write-through caches, we can simplify this to

Memory-stall cycles = accesses/program × miss rate × miss penalty

* *CPI_{base} includes the stall due to data & control hazards (i.e. CPI of previous chapter)*



- ✓ Write: only one word is replaced (not the full block).
- ✓ It means that the original block is read and a word inside the block is modified,. First of all , we have to verify the the block is in cache (hit) (the write can not start until we know we have a hit)
- ✓ In writes, two problems can arise:

DMA:

It allows direct communitacion between MM and CM

IF DMA modifies MM, then the corresponding blocks in CM have to be invalid (valid bit=0)

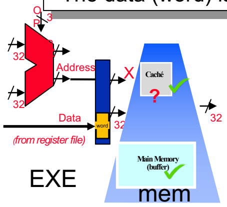
Multi processors systems:

If one word in modified in only one local cache, the same word (block) has to be invalidated in the rest of the caches (valid bit = 0)

Write through

Data memory address **X** is generated in the EXE stage as well as the data (from Reg. File) to be stored in DC

The data (word) is written in the block of MM (via memory buffer)



Search in Caché

write hit

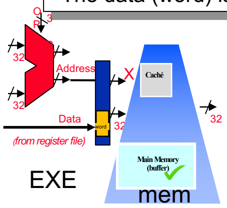
If HIT, the data is written in both the block of MM (through a ***Memory buffer****) and the corresponding block of CM in parallel*

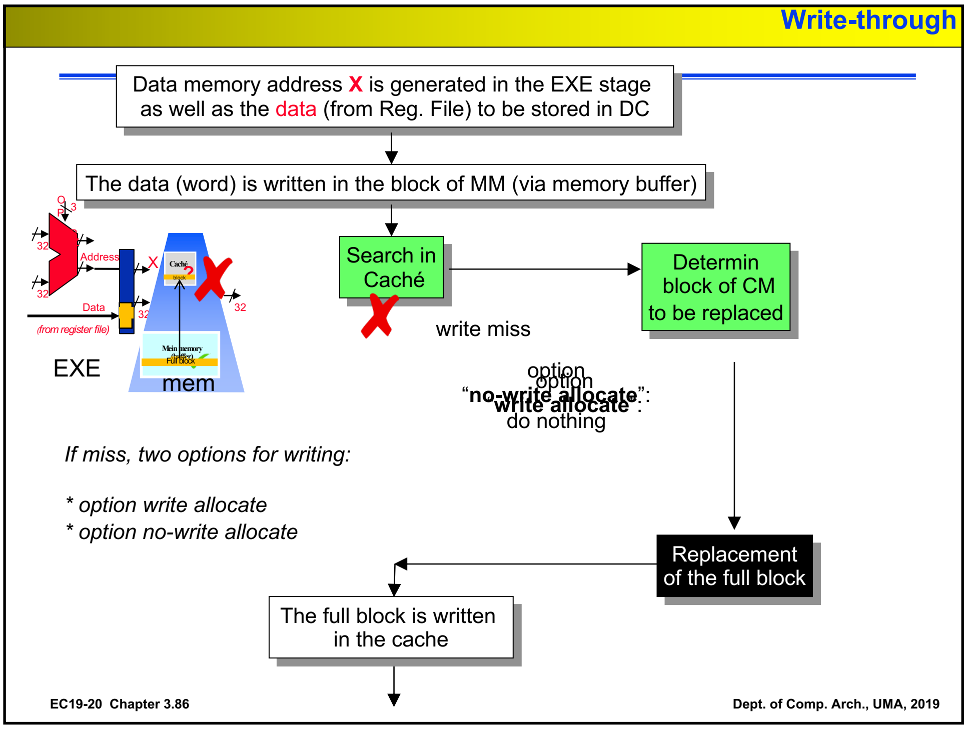
The word is written in the block of the cache

Write through

Data memory address **X** is generated in the EXE stage as well as the data (from Reg. File) to be stored in DC

The data (word) is written in the block of MM (via memory buffer)





Write-through

Data memory address X is generated in the EXE stage as well as the data (from Reg. File) to be stored in DC

The data (word) is written in the block of MM (via memory buffer)

Search in Caché

write miss

Determin block of CM to be replaced

option “no-write allocate”: do nothing

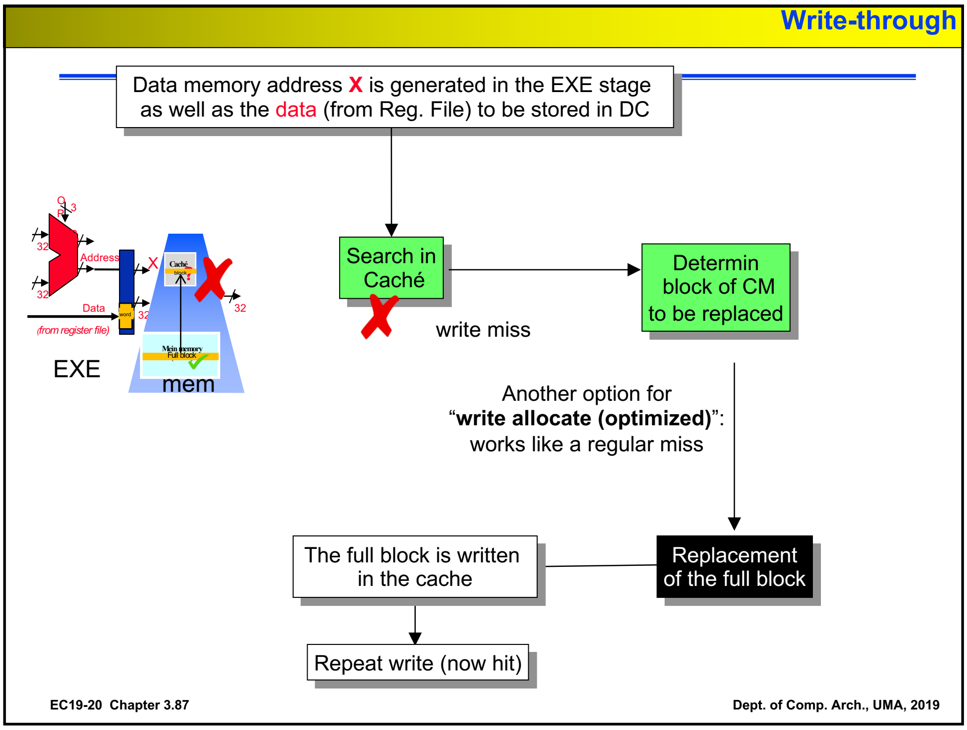
option “write allocate”:

If miss, two options for writing:

** option write allocate*
** option no-write allocate*

Replacement of the full block

The full block is written in the cache



Write-through

Data memory address X is generated in the EXE stage as well as the data (from Reg. File) to be stored in DC

Search in Caché

write miss

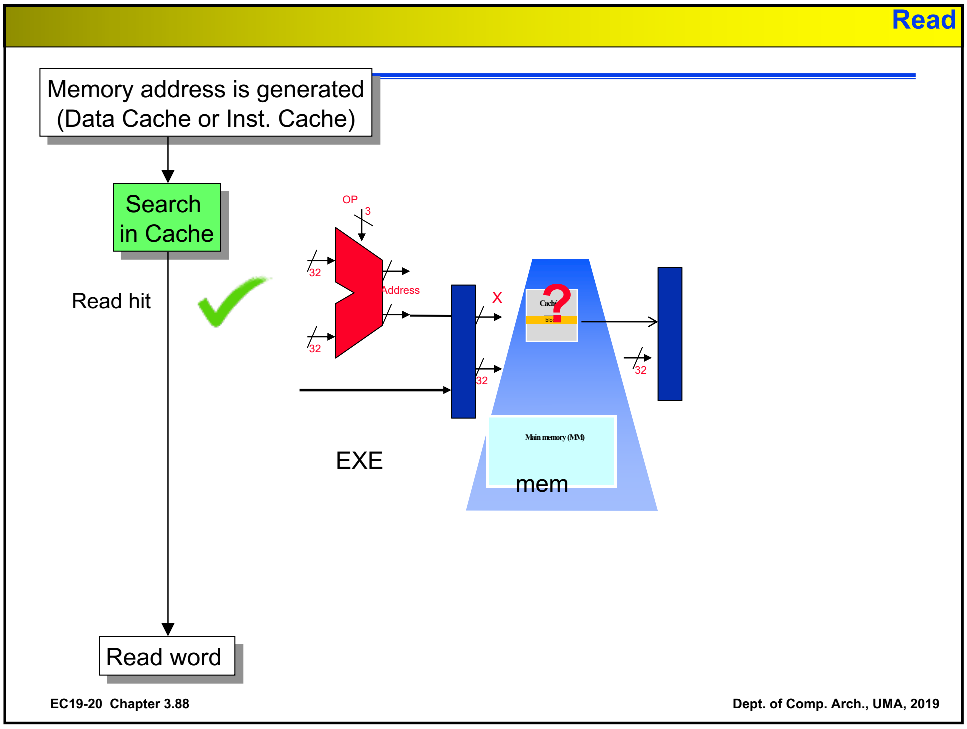
Determin block of CM to be replaced

Another option for “**write allocate (optimized)**”: works like a regular miss

Replacement of the full block

The full block is written in the cache

Repeat write (now hit)



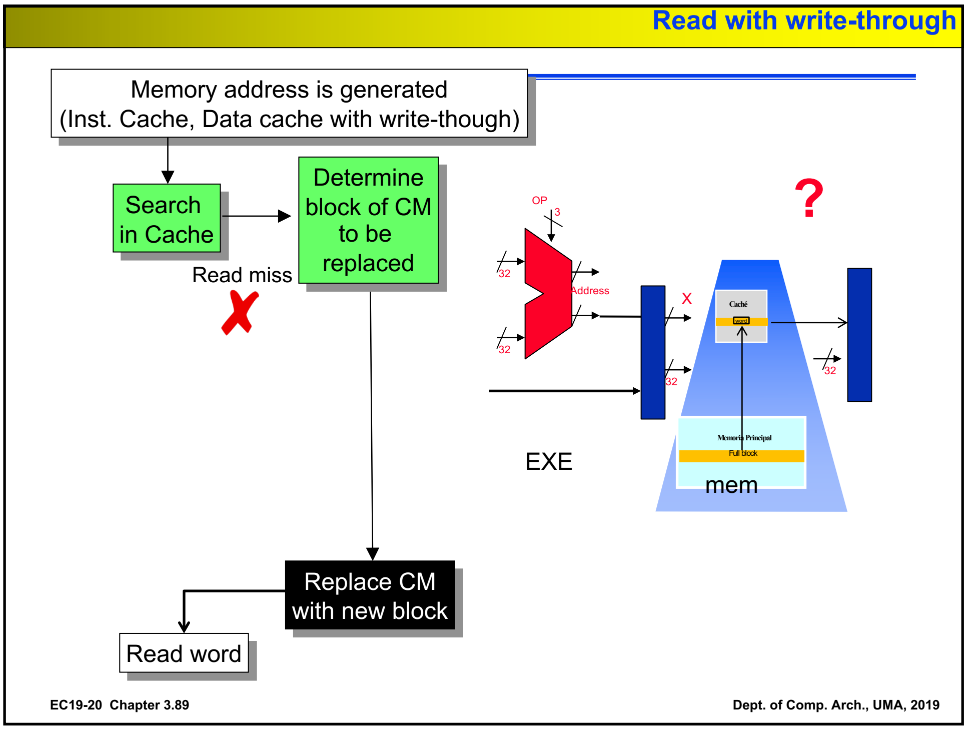
Read

Memory address is generated
(Data Cache or Inst. Cache)

Search in Cache

Read hit

Read word



Read with write-through

Memory address is generated
(Inst. Cache, Data cache with write-though)

Search in Cache

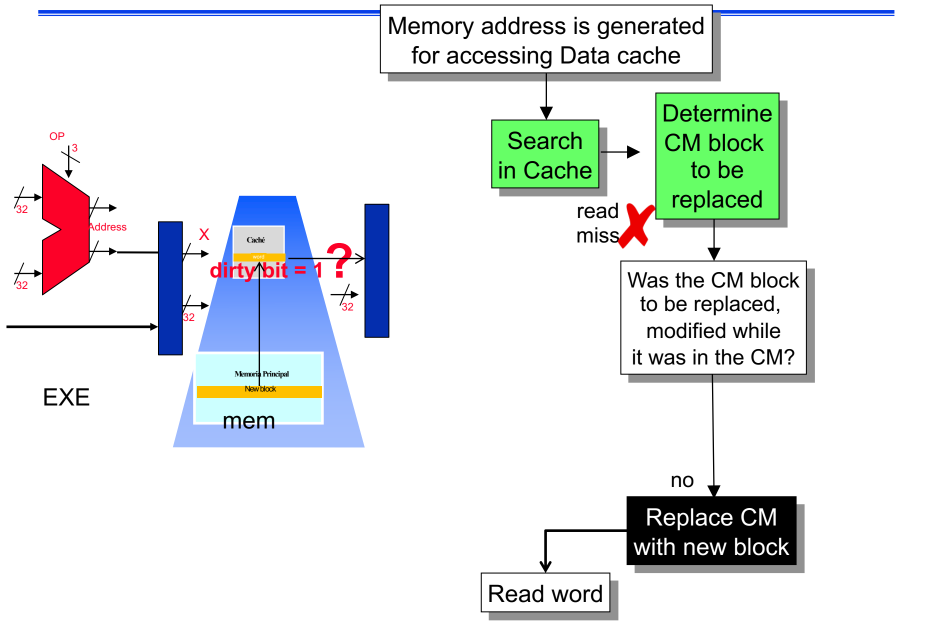
Read miss

Determine block of CM to be replaced

Replace CM with new block

Read word

Read with write-back



Memory address is generated for accessing Data cache

Search in Cache

read miss

Determine CM block to be replaced

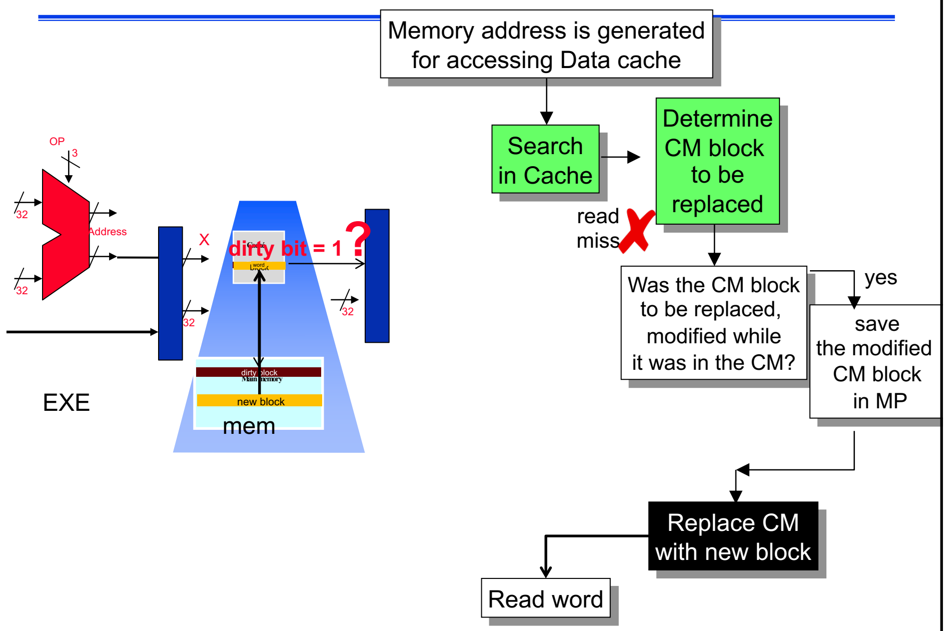
Was the CM block to be replaced, modified while it was in the CM?

no

Replace CM with new block

Read word

Read with write-back



Memory address is generated for accessing Data cache

Search in Cache

read miss

Determine CM block to be replaced

Was the CM block to be replaced, modified while it was in the CM?

yes

save the modified CM block in MP

Replace CM with new block

Read word

Multi-level cache

Multi-level cache

- With advancing technology have more than enough room on the die for bigger L1 caches *or* for a second level of caches – normally a unified L2 cache (i.e., it holds both instructions and data) and in some cases even a unified L3 cache
- For our example, $\text{CPI}_{\text{ideal}}$ of 2, 100 cycle miss penalty (to main memory) and a 25 cycle miss penalty (to L2$), 36% load/stores, a 2% (4%) L1 I$ (D$) miss rate, add a 0.5% L2$ miss rate

$$\text{CPI}_{\text{effective}} = 2 + .02 \times 25 + .005 \times 100 + .36 \times .04 \times 25 + .36 \times .005 \times 100 = 3.54$$

(compared to $\text{CPI}_{\text{effective}} = 2 + .02 \times 100 + 0.36 \times 0.04 \times 100 = 5.44$ with no UL2$)

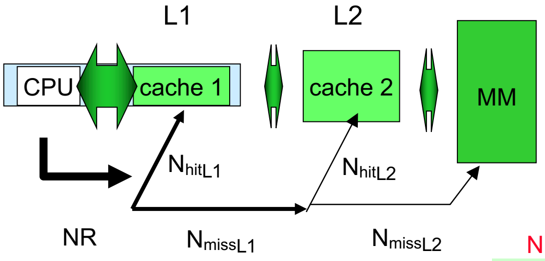
Multilevel Cache Design Considerations

- Design considerations for L1 and L2 caches are very different
  - Primary cache should focus on minimizing hit time in support of a shorter clock cycle
    - Smaller with smaller block sizes
  - Secondary cache(s) should focus on reducing miss rate to reduce the penalty of long main memory access times
    - Larger with larger block sizes
    - Higher levels of associativity
- The miss penalty of the L1 cache is significantly reduced by the presence of an L2 cache – so it can be smaller (i.e., faster) but have a higher miss rate
- For the L2 cache, hit time is less important than miss rate
  - The L2$ hit time determines L1$'s miss penalty
  - L2$ local miss rate >> than the global miss rate

Multi-level cache

✓ Two level cache L_1 y L_2

NR = Number of References
N_{hit} = Number of Hits
N_{miss} = Number of Misses



$NR = N_{hitL1} + N_{missL1} =$
$= N_{hitL1} + (N_{hitL2} + N_{hitL2})$

Number of references of L2 is the number of misses of L1

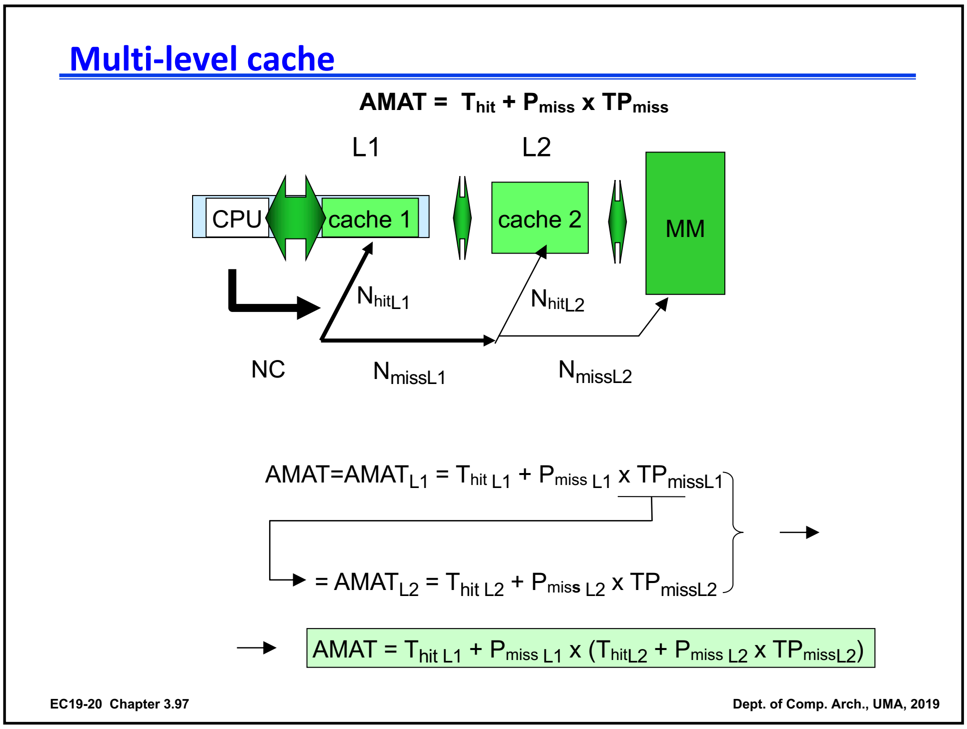
L1: Upper level (closest to CPU) (For sepatate caches L1I y L1D).

L2. Lower level (Secondary cache). (Normally it is unified)

$P_{missL1} = \frac{N_{missL1}}{NR}$ → Local miss rate L1

$P_{missL2} = \frac{N_{missL2}}{N_{missL1}}$ → Local miss rate L2

$P_{miss} = \frac{N_{missL2}}{NR} = \frac{N_{missL1}}{NR} \times \frac{N_{missL2}}{N_{missL1}} = P_{missL1} \times P_{missL2}$ = **global** miss rate



Multi-level cache

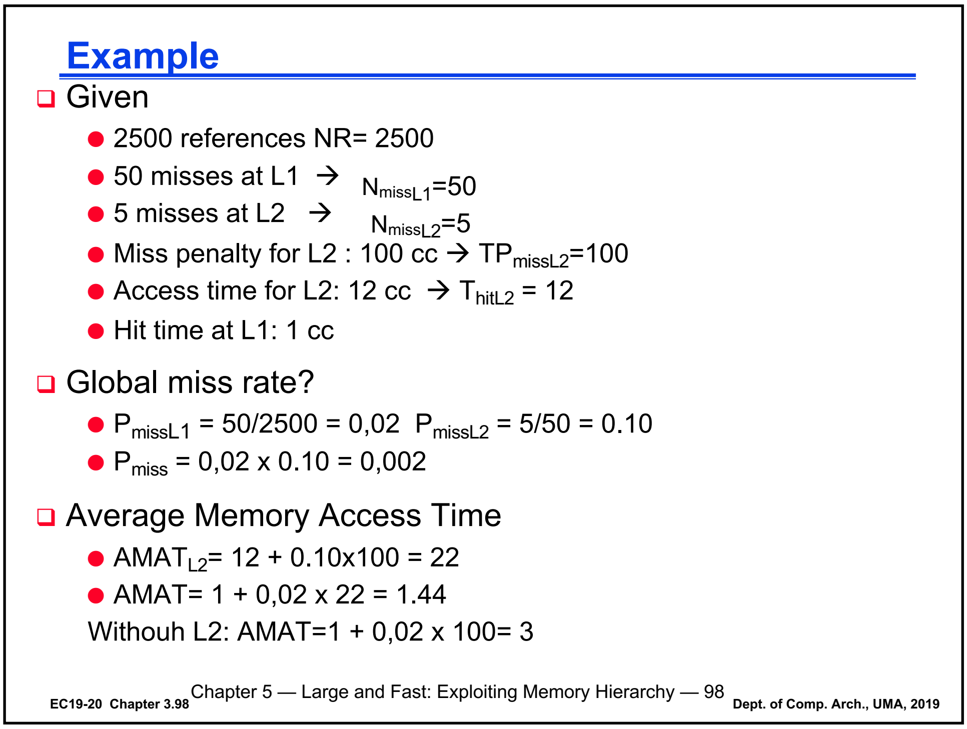
$\mathbf{AMAT} = \mathbf{T_{hit} + P_{miss} \ x\ TP_{miss}}$

$\mathrm{AMAT}=\mathrm{AMAT}_{L1} = T_{hit\ L1} + P_{miss\ L1}\ x\ \underline{TP_{missL1}}$

$= \mathrm{AMAT}_{L2} = T_{hit\ L2} + P_{miss\ L2}\ x\ TP_{missL2}$

→

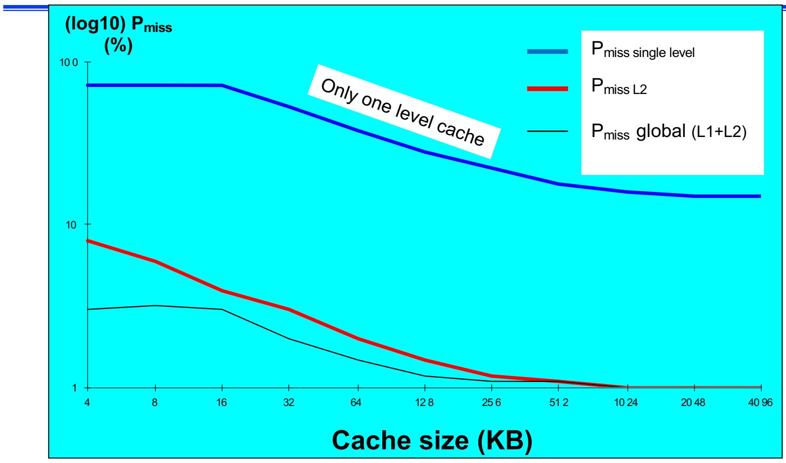
→ $\boxed{\mathrm{AMAT} = T_{hit\ L1} + P_{miss\ L1}\ x\ (T_{hitL2} + P_{miss\ L2}\ x\ TP_{missL2})}$



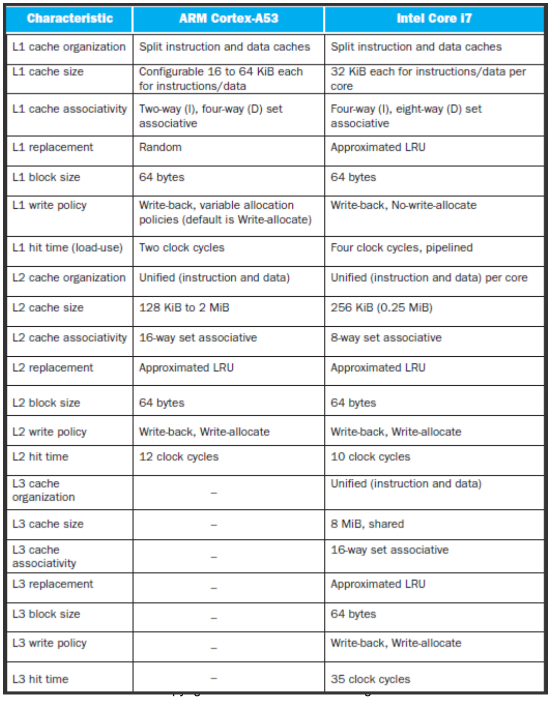
Example

- Given
  - 2500 references NR= 2500
  - 50 misses at L1 → N_{missL1}=50
  - 5 misses at L2 → N_{missL2}=5
  - Miss penalty for L2 : 100 cc → TP_{missL2}=100
  - Access time for L2: 12 cc → T_{hitL2} = 12
  - Hit time at L1: 1 cc
- Global miss rate?
  - P_{missL1} = 50/2500 = 0,02 P_{missL2} = 5/50 = 0.10
  - P_{miss} = 0,02 x 0.10 = 0,002
- Average Memory Access Time
  - $AMAT_{L2}$= 12 + 0.10x100 = 22
  - AMAT= 1 + 0,02 x 22 = 1.44

  Withouh L2: AMAT=1 + 0,02 x 100= 3



Current Processors

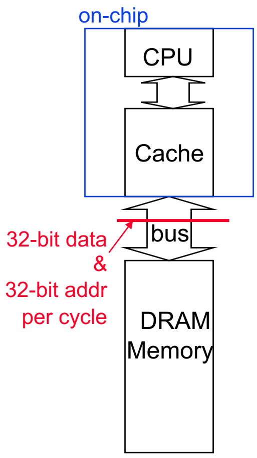


Characteristic	ARM Cortex-A53	Intel Core i7
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	Configurable 16 to 64 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	Two-way (I), four-way (D) set associative	Four-way (I), eight-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, variable allocation policies (default is Write-allocate)	Write-back, No-write-allocate
L1 hit time (load-use)	Two clock cycles	Four clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 2 MiB	256 KiB (0.25 MiB)
L2 cache associativity	16-way set associative	8-way set associative
L2 replacement	Approximated LRU	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	12 clock cycles	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, shared
L3 cache associativity	–	16-way set associative
L3 replacement	–	Approximated LRU
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	35 clock cycles

Main Memory

Memory Systems that Support Caches

- The off-chip interconnect and memory architecture can affect overall system performance in dramatic ways

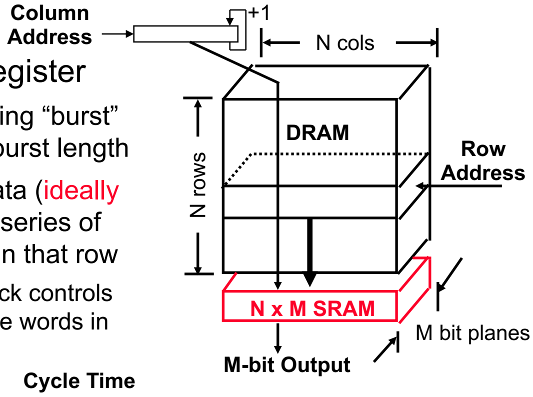


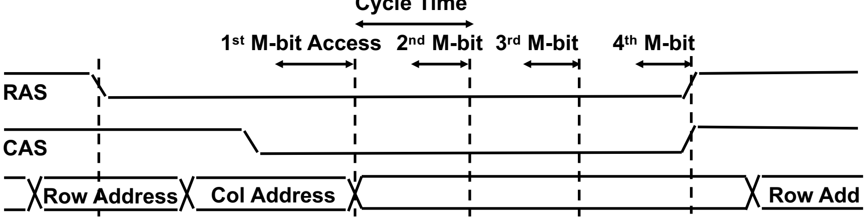
One word wide organization (one word wide bus and one word wide memory)

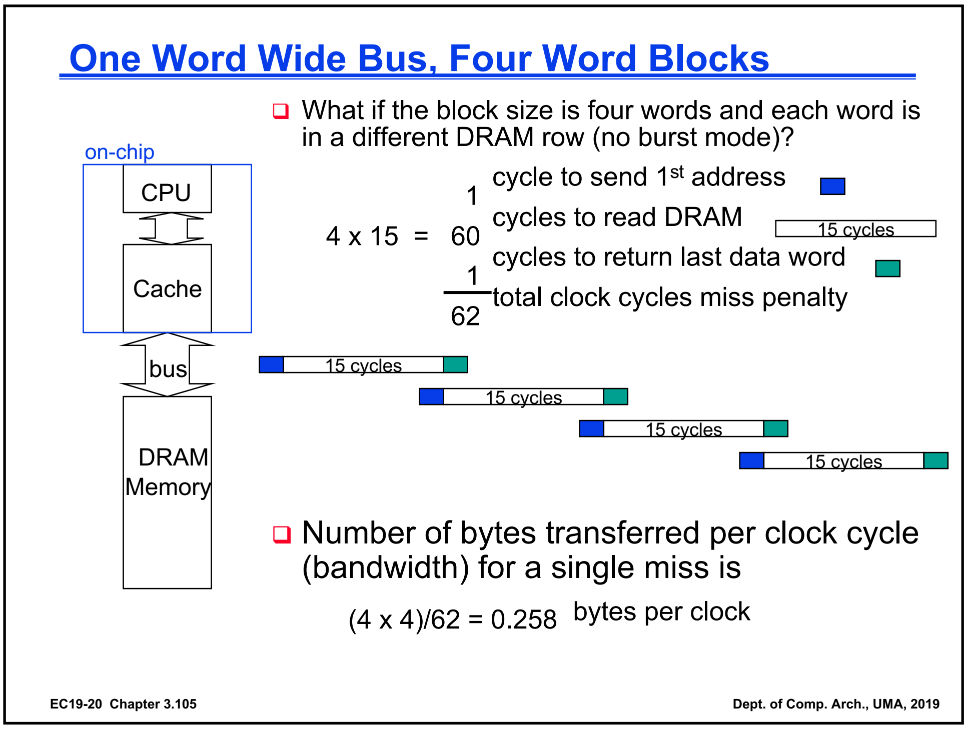
- Assume
  1. 1 memory bus clock cycle to send the addr
  2. 15 memory bus clock cycles to get the 1st word in the block from DRAM (row cycle time), 5 memory bus clock cycles for 2nd, 3rd, 4th words (column access time)
  3. 1 memory bus clock cycle to return a word of data
- Memory-Bus to Cache bandwidth
  - number of bytes accessed from memory and transferred to cache/CPU per memory bus clock cycle

Review: (DDR) SDRAM Operation (Burst mode)

- After a row is read into the SRAM register
  - Input CAS as the starting “burst” address along with a burst length
  - Transfers a burst of data (ideally a cache block) from a series of sequential addr’s within that row
    - The memory bus clock controls transfer of successive words in the burst







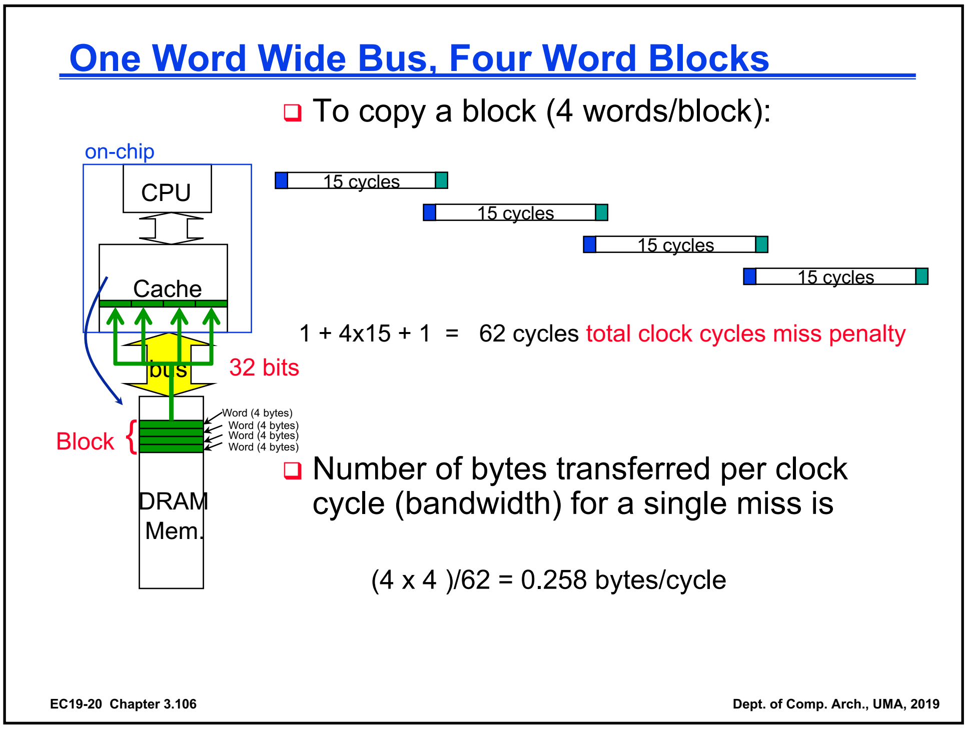
One Word Wide Bus, Four Word Blocks

- What if the block size is four words and each word is in a different DRAM row (no burst mode)?

	1	cycle to send 1st address
4 x 15 =	60	cycles to read DRAM
	1	cycles to return last data word
	62	total clock cycles miss penalty

- Number of bytes transferred per clock cycle (bandwidth) for a single miss is

(4 x 4)/62 = 0.258 bytes per clock



One Word Wide Bus, Four Word Blocks

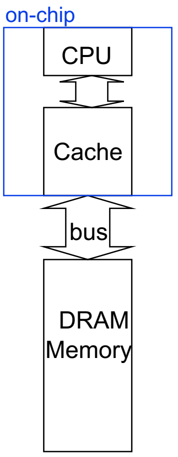
- To copy a block (4 words/block):

1 + 4x15 + 1 = 62 cycles total clock cycles miss penalty

- Number of bytes transferred per clock cycle (bandwidth) for a single miss is

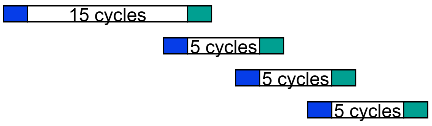
(4 x 4)/62 = 0.258 bytes/cycle

One Word Wide Bus, Four Word Blocks



- What if the block size is four words and all words are in the same DRAM row (burst mode)?

1	cycle to send 1st address
15 + 3*5 = 30	cycles to read DRAM
1	cycles to return last data word
32	total clock cycles miss penalty

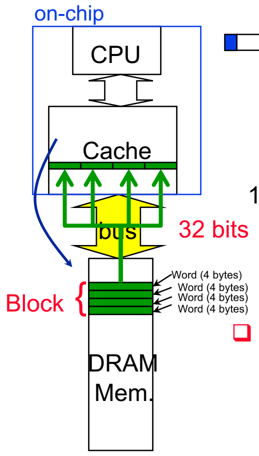


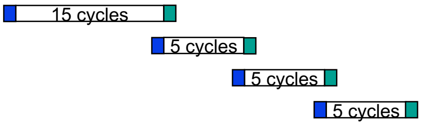
- Number of bytes transferred per clock cycle (bandwidth) for a single miss is

(4 x 4)/32 = 0.5 bytes per clock

One Word Wide Bus, Four Word Blocks

- To copy a block (4 words/block):

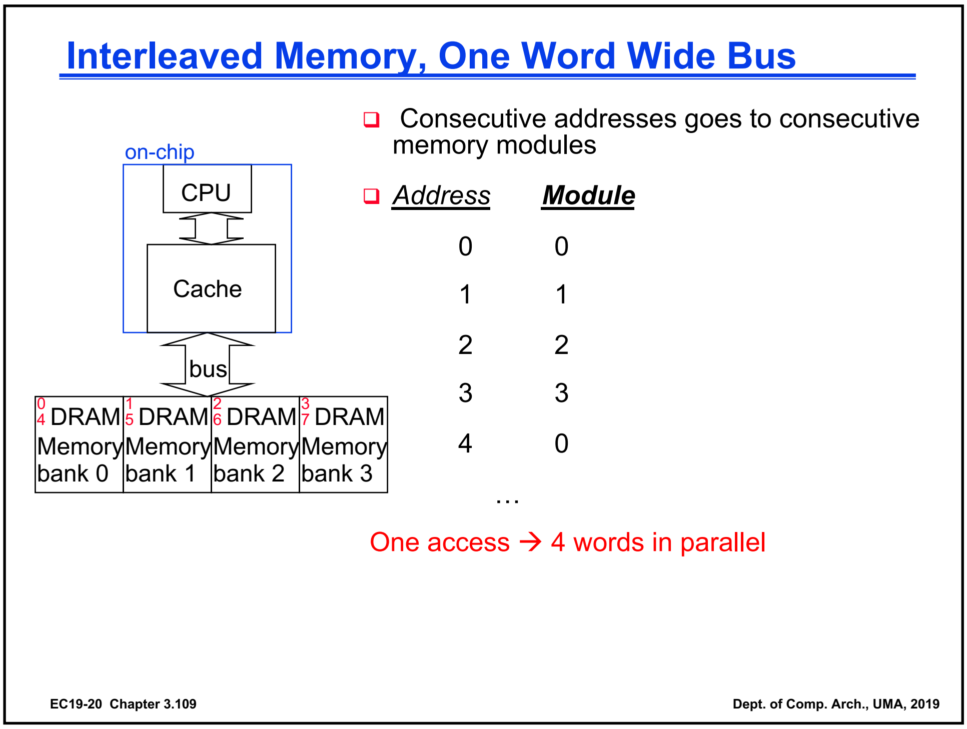




1 + 15 + 5x3 + 1 = 32 cycles total clock cycles miss penalty

- Number of bytes transferred per clock cycle (bandwidth) for a single miss is

(4 x 4)/32 = 0.5 bytes/cycle

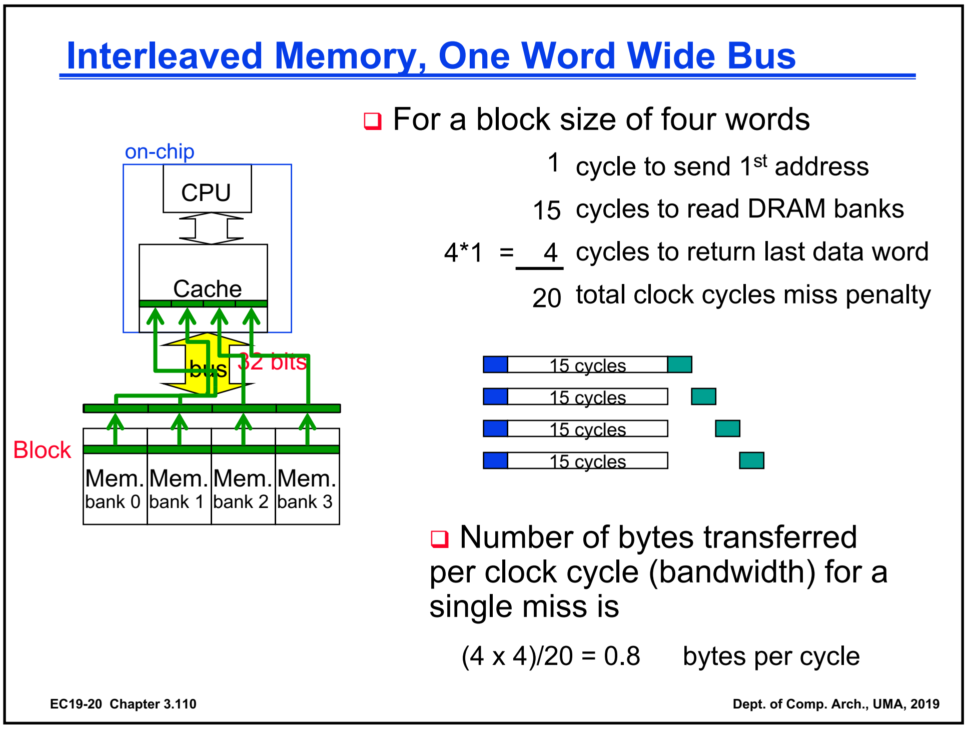


Interleaved Memory, One Word Wide Bus

- Consecutive addresses goes to consecutive memory modules
- *Address* ***Module***

Address	***Module***
0	0
1	1
2	2
3	3
4	0
…	

One access → 4 words in parallel



Interleaved Memory, One Word Wide Bus

- For a block size of four words

1 cycle to send 1^{st} address

15 cycles to read DRAM banks

4*1 = $\underline{4}$ cycles to return last data word

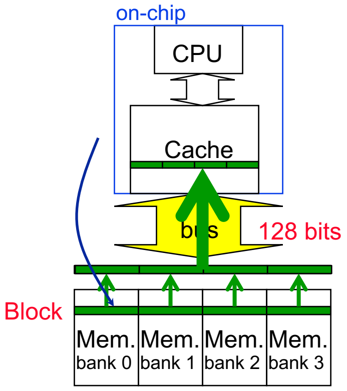
20 total clock cycles miss penalty

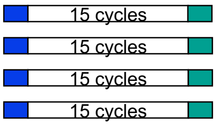
- Number of bytes transferred per clock cycle (bandwidth) for a single miss is

(4 x 4)/20 = 0.8 bytes per cycle

Interleaved Memory, One Word Wide Bus

- ❑ An improvement: wider bus (128 bits)





1+ 15 + 1 = 17 cycles total clock cycles miss penalty

- ❑ Number of bytes transferred per clock cycle (bandwidth) for a single miss is

(4 x 4)/17 = 0.94 bytes per cycle

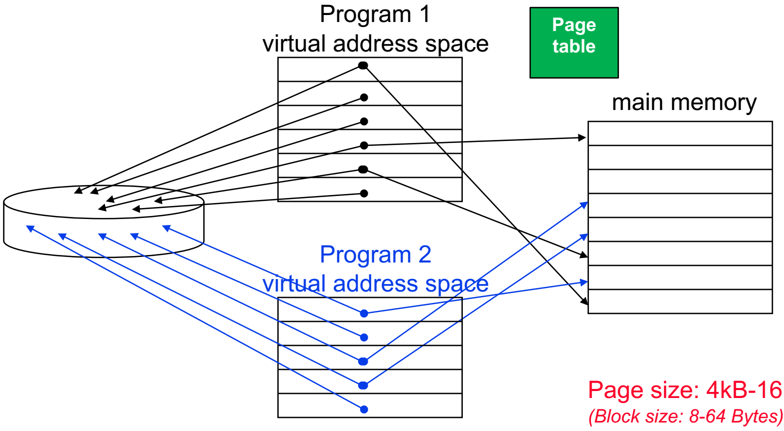
Virtual memory

Virtual Memory

- ❑ Use main memory as a “cache” for secondary memory
  - Allows efficient and safe sharing of memory among multiple programs
  - Provides the ability to easily run programs larger than the size of physical memory
  - Simplifies loading a program for execution by providing for code relocation (i.e., the code can be loaded anywhere in main memory)
- ❑ What makes it work? – again the Principle of Locality
  - A program is likely to access a relatively small portion of its address space during any period of time
- ❑ Each program is compiled into its own address space – a “virtual” address space
  - During run-time each virtual address must be translated to a physical address (an address in main memory)

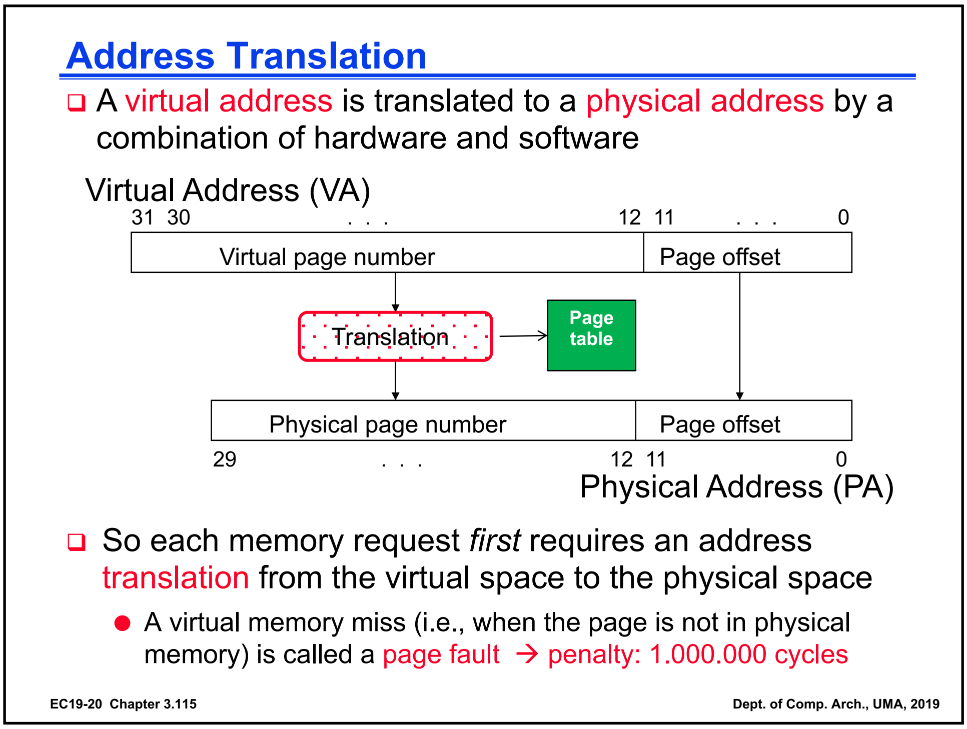
Two Programs Sharing Physical Memory

- ❑ A program’s address space is divided into pages (all one fixed size) or segments (variable sizes)
  - The starting location of each page (either in main memory or in secondary memory) is contained in the program’s page table



Page size: 4kB-16kB

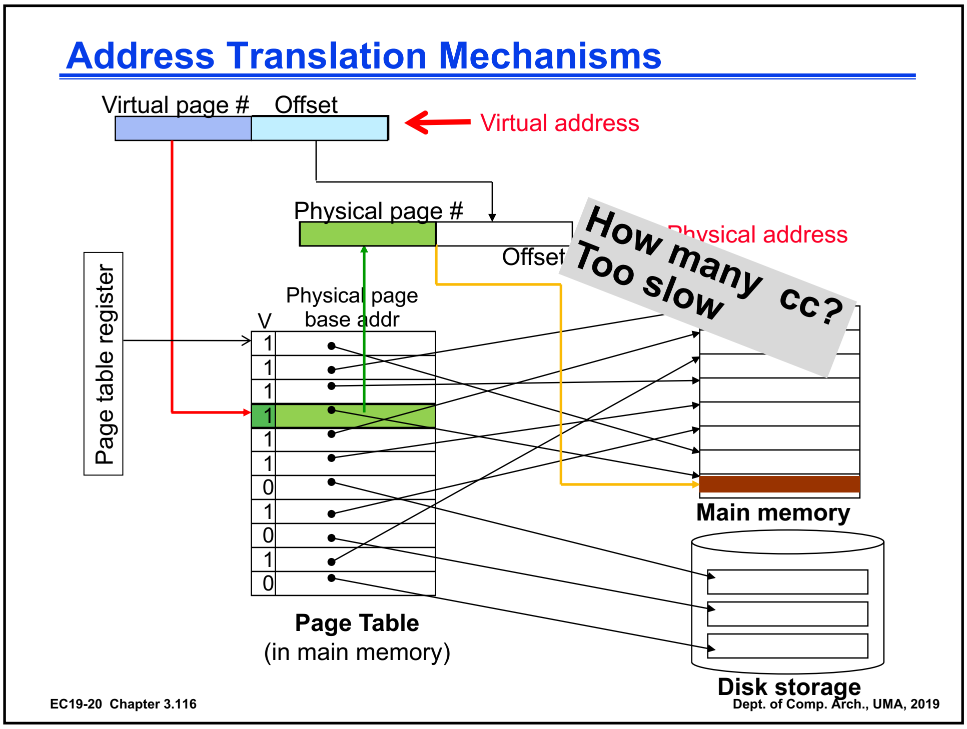
(Block size: 8-64 Bytes)



Address Translation

- A virtual address is translated to a physical address by a combination of hardware and software

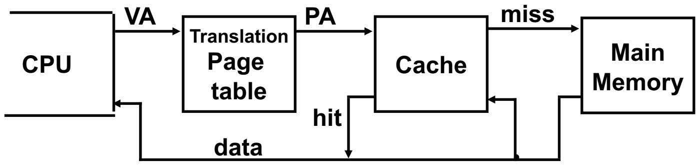
- So each memory request *first* requires an address translation from the virtual space to the physical space
  - A virtual memory miss (i.e., when the page is not in physical memory) is called a page fault → penalty: 1.000.000 cycles



Address Translation Mechanisms

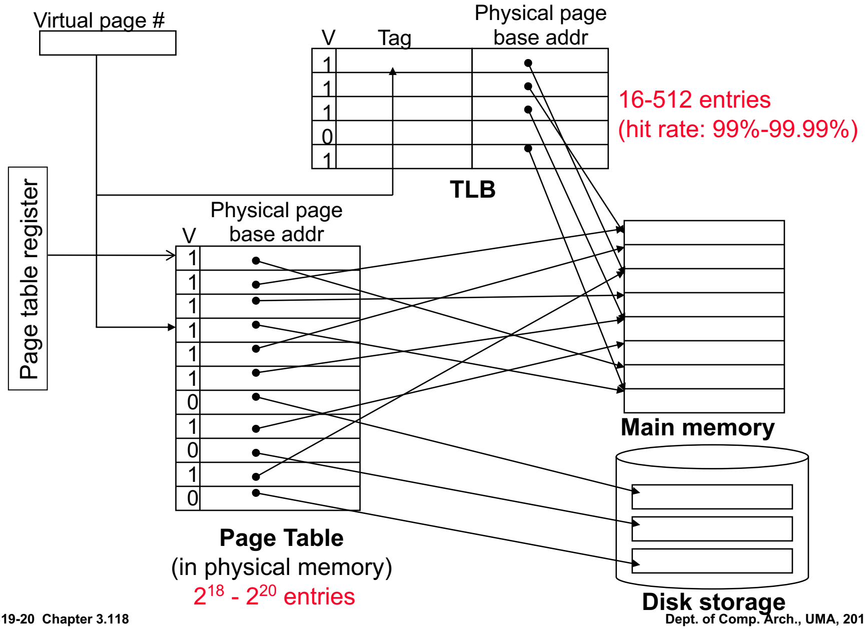
Virtual Addressing with a Cache

- Thus it takes an *extra* memory access to translate a VA to a PA



- This makes memory (cache) accesses very expensive (if every access was really *two* accesses)
- The hardware fix is to use a ***Translation Lookaside Buffer*** **(TLB)** – a small cache that keeps track of recently used address mappings to avoid having to do a page table lookup

Making Address Translation Fast



16-512 entries
(hit rate: 99%-99.99%)

Page Table
(in physical memory)
2^{18} - 2^{20} entries

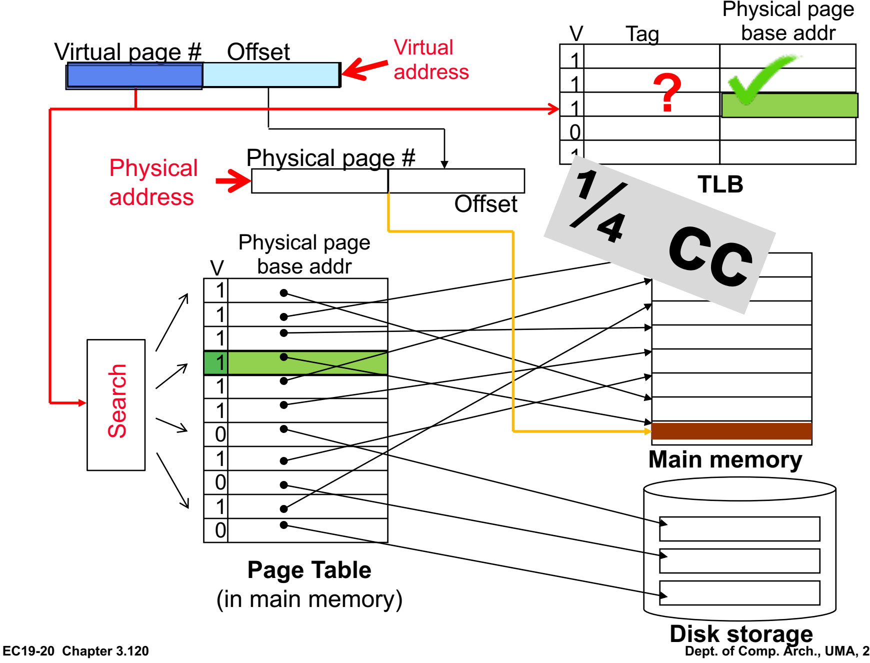
Translation Lookaside Buffers (TLBs)

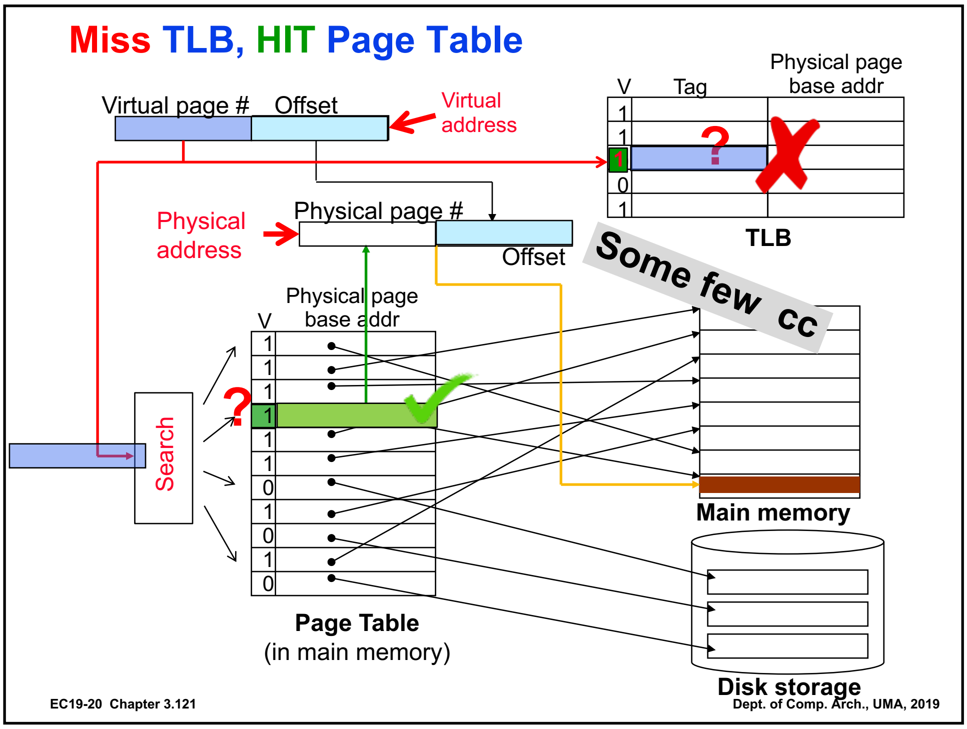
- Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped

V	TAG Virt. Page #	Physical Page #	Dirty	Ref	Access

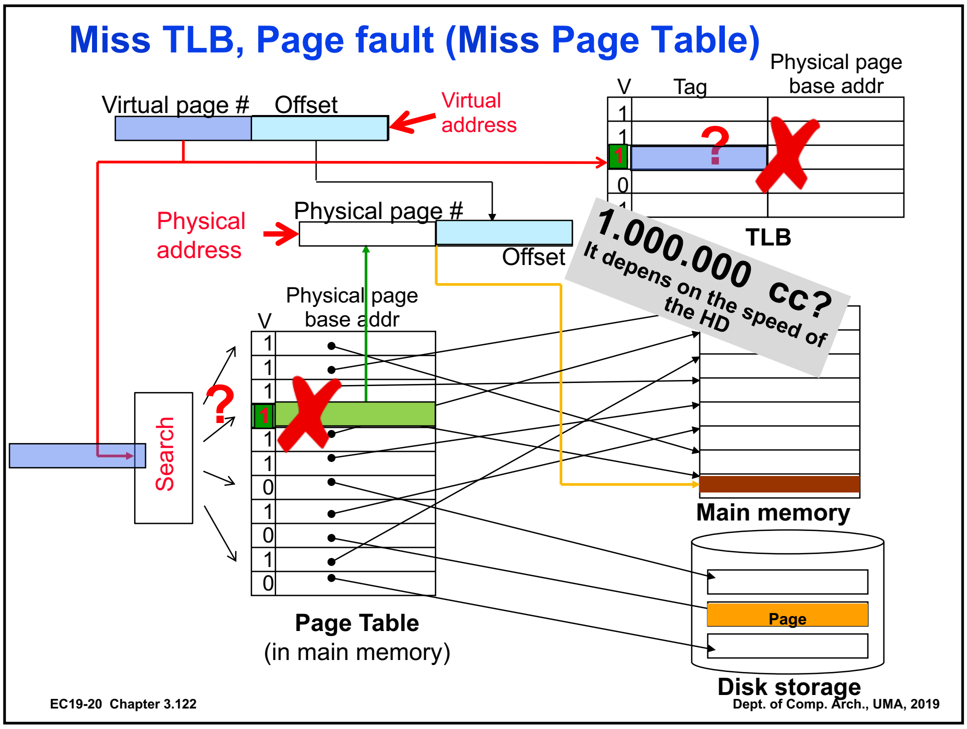
- TLB access time is typically smaller than cache access time (because TLBs are much smaller than caches)
  - TLBs are typically not more than 512 entries even on high end machines

Address Translation Mechanisms HIT TLB





Miss TLB, HIT Page Table

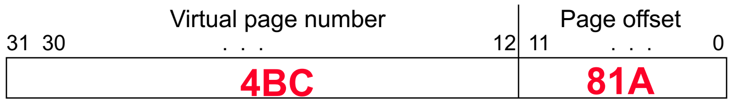


Miss TLB, Page fault (Miss Page Table)

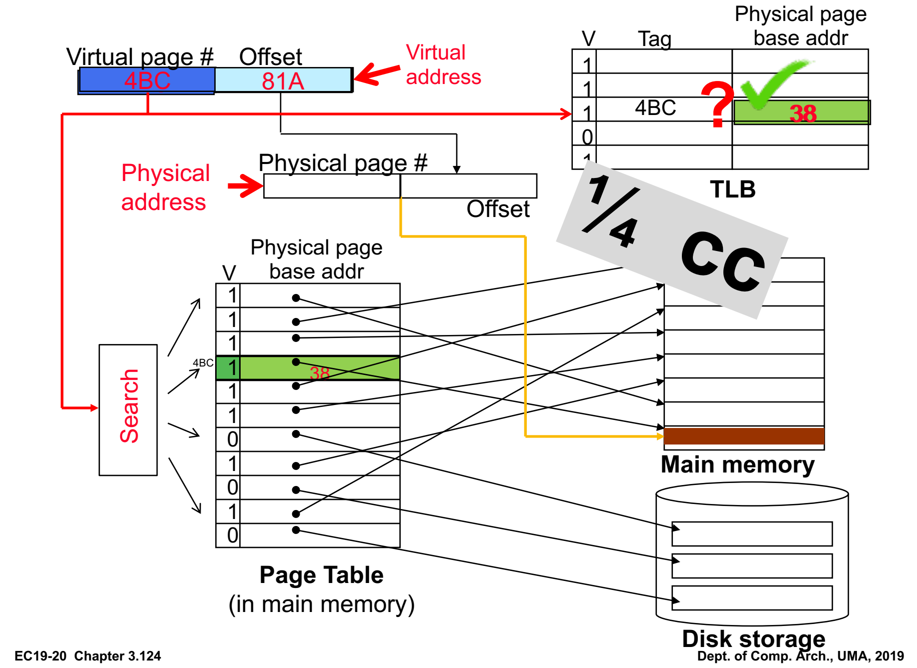
Example

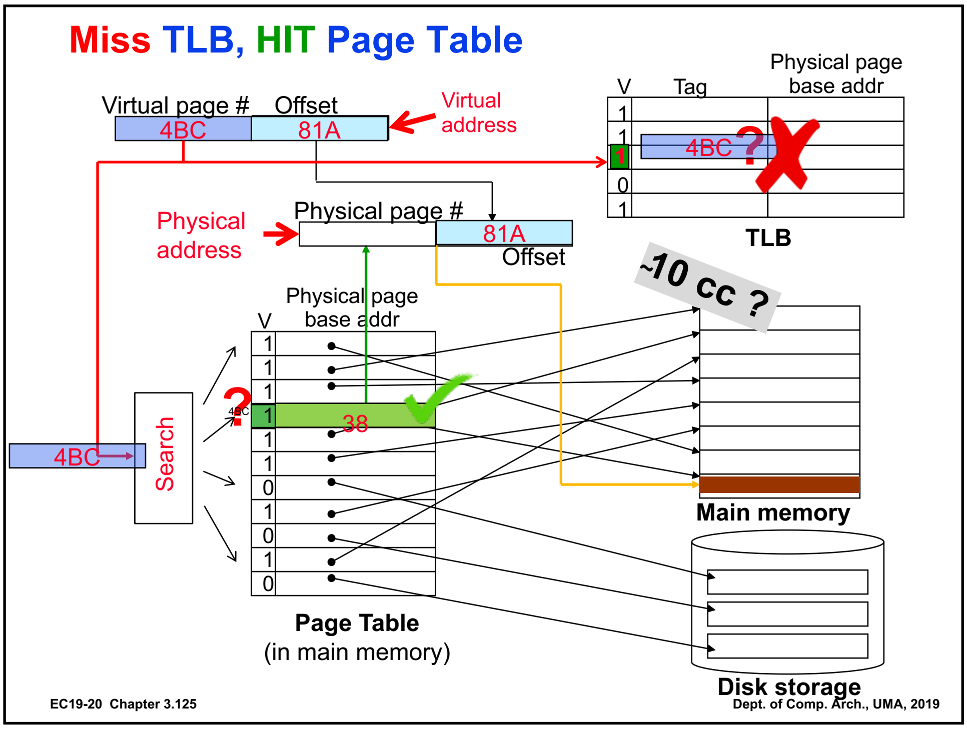
Page size: 4KB = 2^{12} bytes (12 bits page offset)

Virtual Address   4BC81Ah



Ex. HIT: Page: 4KB   Virt. Add. 4BC81Ah





Miss TLB, HIT Page Table

Virtual page # Offset

4BC 81A

Virtual address

V Tag Physical page base addr

1
1
1 4BC ?
0
1

TLB

Physical address

Physical page #

81A

Offset

~10 cc ?

Physical page base addr

V

1
1
1
1 38
1
1
0
1
0
1
0

4BC

Search

?

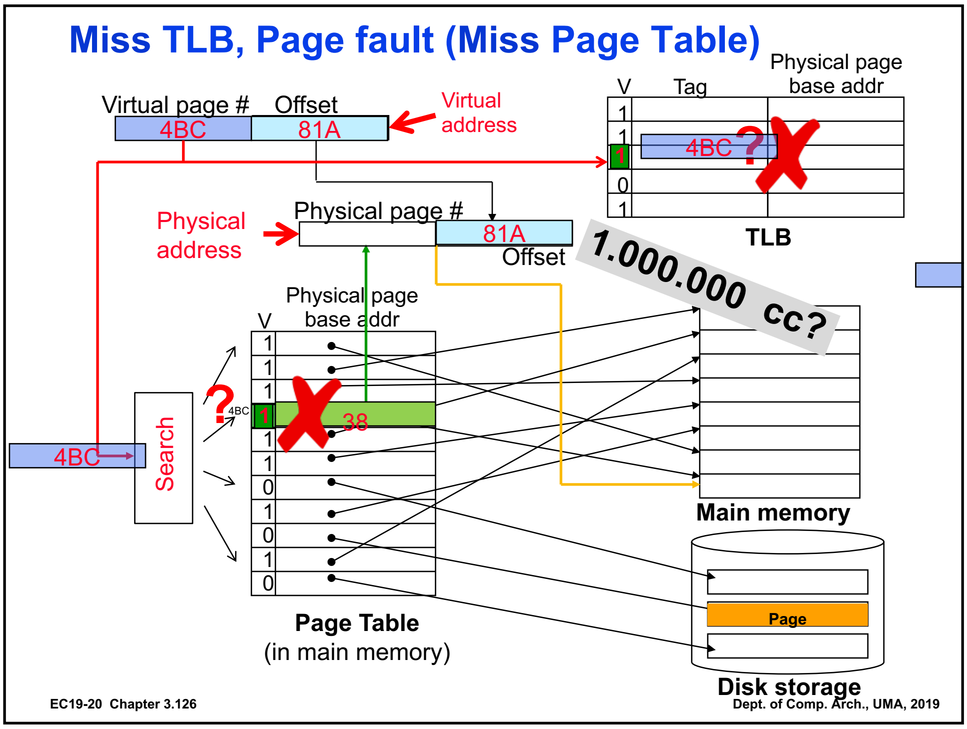
4BC

Page Table

(in main memory)

Main memory

Disk storage



Miss TLB, Page fault (Miss Page Table)

Virtual page # Offset

4BC 81A

Virtual address

V Tag Physical page base addr

1
1
1 4BC ?
0
1

TLB

Physical address

Physical page #

81A

Offset

1.000.000 cc?

Physical page base addr

V

1
1
1
1 38
1
1
0
1
0
1
0

4BC

Search

?

4BC

Page Table

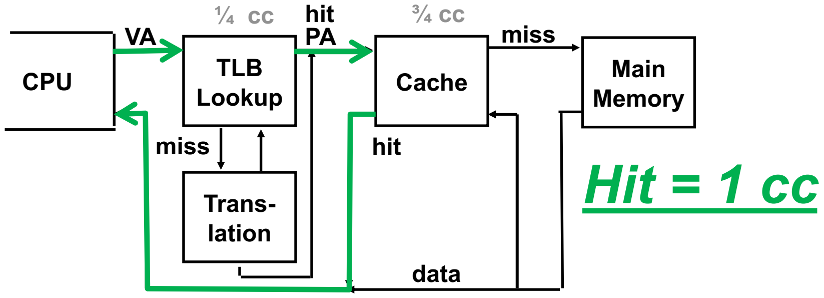
(in main memory)

Main memory

Page

Disk storage

A TLB in the Memory Hierarchy



- ❑ A TLB miss – is it a page fault or merely a TLB miss?
  - ● If the page is loaded into main memory, then the TLB miss can be handled (in hardware or software) by loading the translation information from the page table into the TLB
    - Takes 10’s of cycles to find and load the translation info into the TLB
  - ● If the page is not in main memory, then it’s a true page fault
    - Takes 1,000,000’s of cycles to service a page fault
- ❑ TLB misses are much more frequent than true page faults

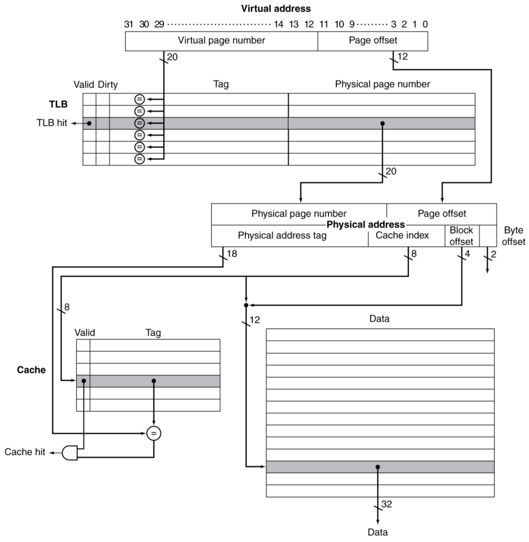
TLB Event Combinations

TLB	Page Table	Cache	Possible? Under what circumstances?
Hit	Hit	Hit	
Hit	Hit	Miss	
Miss	Hit	Hit	
Miss	Hit	Miss	
Miss	Miss	Miss	
Hit	Miss	Miss/ Hit	
Miss	Miss	Hit	

TLB Event Combinations

TLB	Page Table	Cache	Possible? Under what circumstances?
Hit	Hit	Hit	Yes – what we want!
Hit	Hit	Miss	Yes – although the page table is not checked if the TLB hits
Miss	Hit	Hit	Yes – TLB miss, PA in page table
Miss	Hit	Miss	Yes – TLB miss, PA in page table, but data not in cache
Miss	Miss	Miss	Yes – page fault
Hit	Miss	Miss/ Hit	Impossible – TLB translation not possible if page is not present in memory
Miss	Miss	Hit	Impossible – data not allowed in cache if page is not in memory

TLB and Cache Interaction

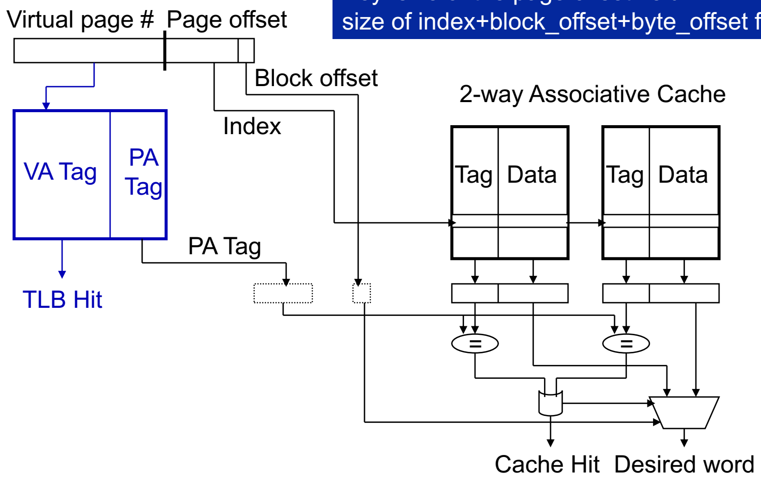


- ❑ If cache tag uses physical address
  - ● Need to translate before cache lookup
- ❑ Alternative: use virtual address tag
  - ● Complications due to aliasing
    - Different virtual addresses for shared physical address

Reducing Translation Time: Overlap TLB-Cache

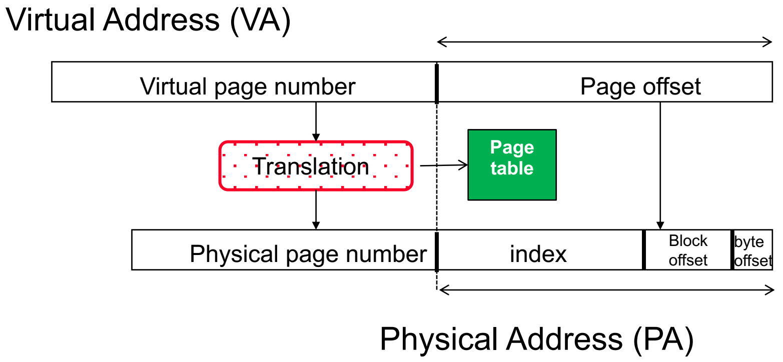
- Can overlap the cache access with the TLB access?
  - Works when the high order bits of the VA are used to access the TLB while the low order bits are used as index into cache

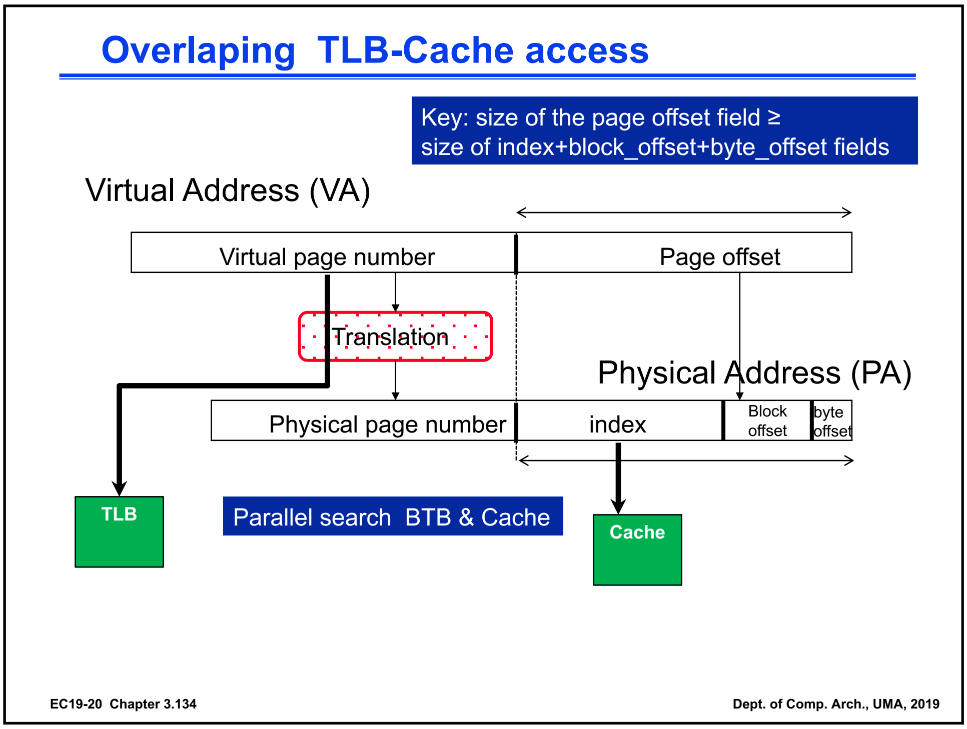
Key: size of the page offset field ≥ size of index+block_offset+byte_offset fields



Reducing Translation Time: Overlap TLB-Cache

Key: size of the page offset field ≥ size of index+block_offset+byte_offset fields





Overlaping TLB-Cache access

Key: size of the page offset field ≥ size of index+block_offset+byte_offset fields

Virtual Address (VA)

Virtual page number | Page offset

Translation

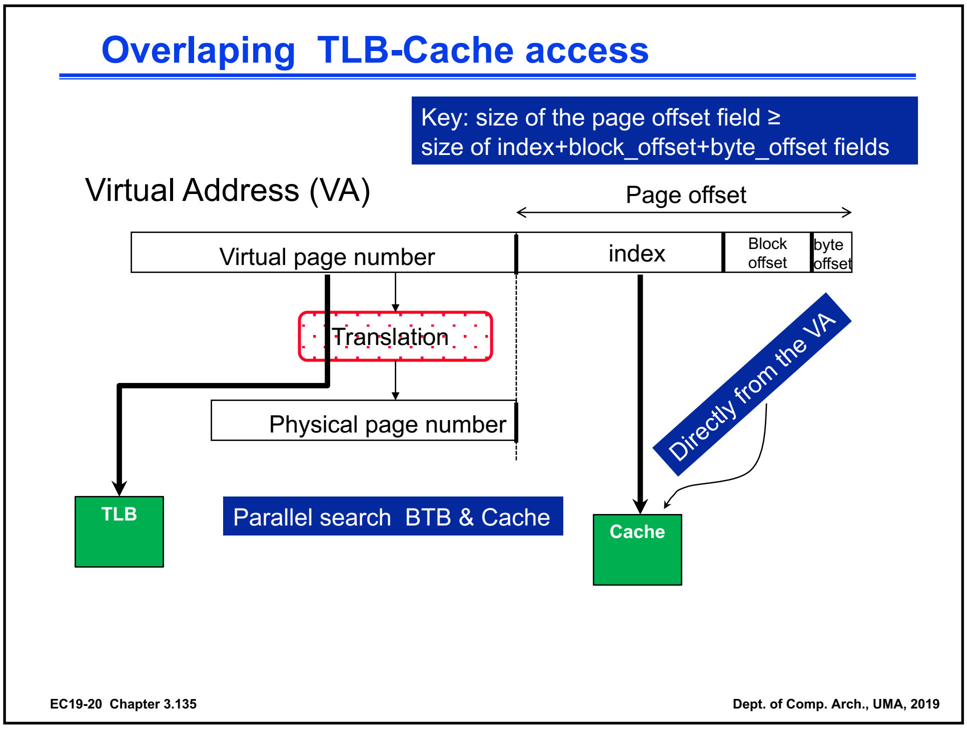
Physical Address (PA)

Physical page number | index | Block offset | byte offset

TLB

Parallel search BTB & Cache

Cache



Overlaping TLB-Cache access

Key: size of the page offset field ≥ size of index+block_offset+byte_offset fields

Virtual Address (VA)

Page offset

Virtual page number | index | Block offset | byte offset

Translation

Physical page number

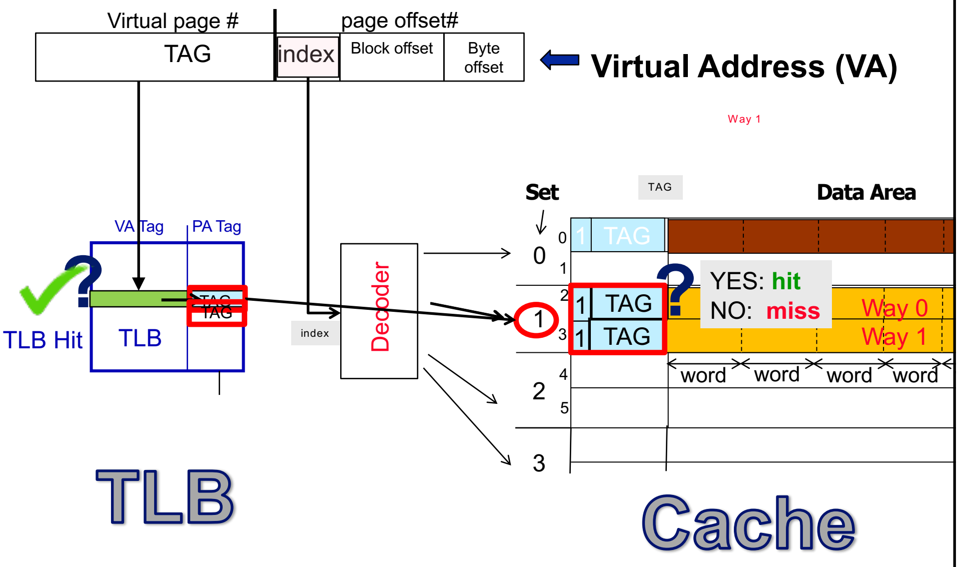
Directly from the VA

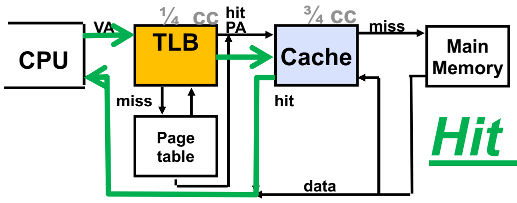
TLB

Parallel search BTB & Cache

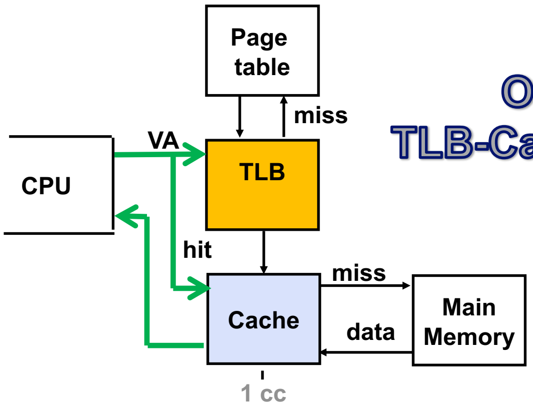
Cache

Overlaping TLB-Cache access



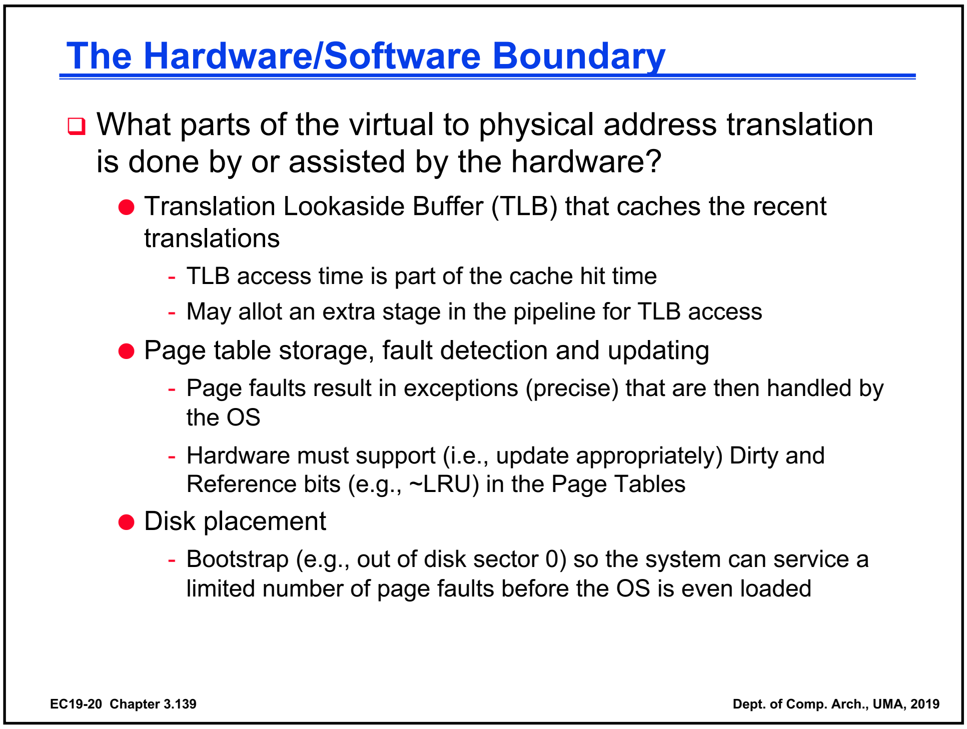


Hit = 1 cc



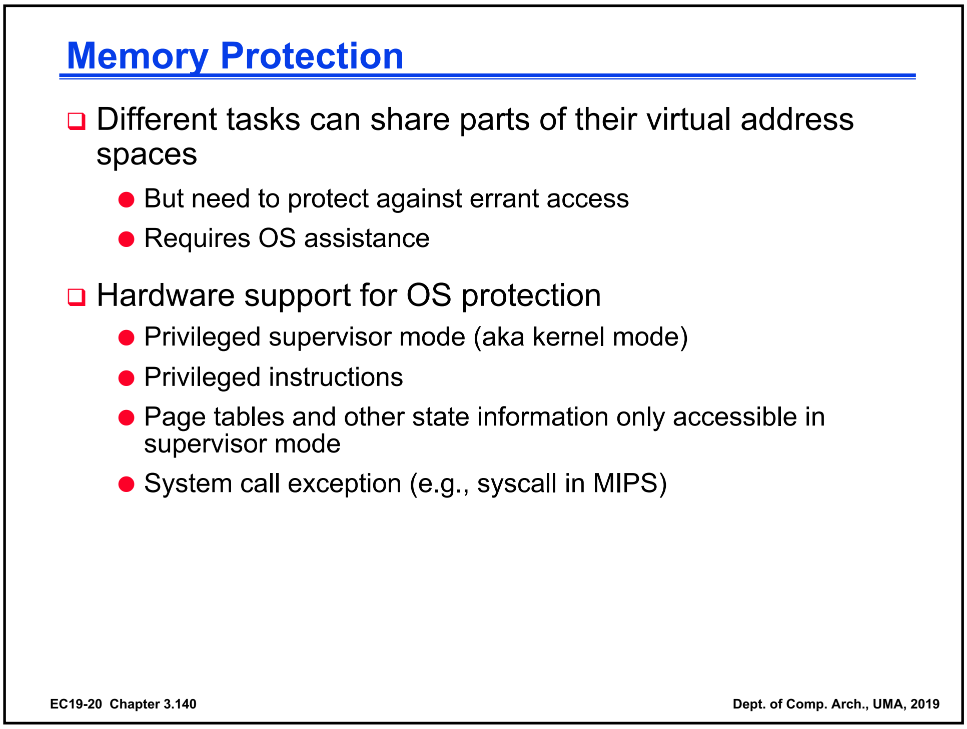
Overlap TLB-Cache access

Hit = 1 cc



The Hardware/Software Boundary

- ❑ What parts of the virtual to physical address translation is done by or assisted by the hardware?
  - ● Translation Lookaside Buffer (TLB) that caches the recent translations
    - TLB access time is part of the cache hit time
    - May allot an extra stage in the pipeline for TLB access
  - ● Page table storage, fault detection and updating
    - Page faults result in exceptions (precise) that are then handled by the OS
    - Hardware must support (i.e., update appropriately) Dirty and Reference bits (e.g., ~LRU) in the Page Tables
  - ● Disk placement
    - Bootstrap (e.g., out of disk sector 0) so the system can service a limited number of page faults before the OS is even loaded



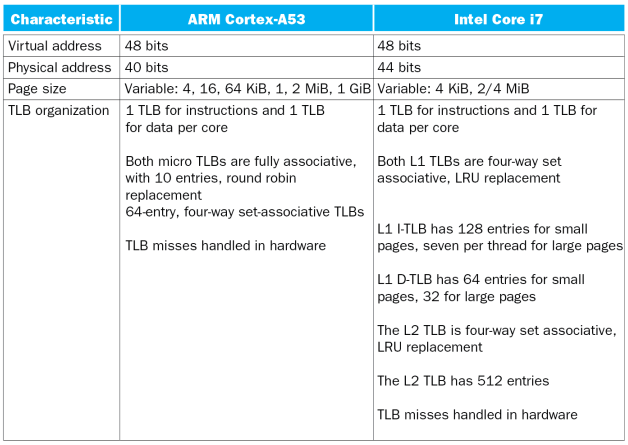
Memory Protection

- ❑ Different tasks can share parts of their virtual address spaces
  - ● But need to protect against errant access
  - ● Requires OS assistance
- ❑ Hardware support for OS protection
  - ● Privileged supervisor mode (aka kernel mode)
  - ● Privileged instructions
  - ● Page tables and other state information only accessible in supervisor mode
  - ● System call exception (e.g., syscall in MIPS)

Some Virtual Memory Design Parameters

	Paged VM	TLBs
Total size	16,000 to 250,000 words	16 to 512 entries
Total size (KB)	250,000 to 1,000,000,000	0.25 to 16
Block size (B)	4000 to 64,000	4 to 8
Hit time		0.5 to 1 clock cycle
Miss penalty (clocks)	10,000,000 to 100,000,000	10 to 100
Miss rates	0.00001% to 0.0001%	0.01% to 1%

2-Level TLB Organization



Characteristic	ARM Cortex-A53	Intel Core i7
Virtual address	48 bits	48 bits
Physical address	40 bits	44 bits
Page size	Variable: 4, 16, 64 KiB, 1, 2 MiB, 1 GiB	Variable: 4 KiB, 2/4 MiB
TLB organization	1 TLB for instructions and 1 TLB for data per core Both micro TLBs are fully associative, with 10 entries, round robin replacement 64-entry, four-way set-associative TLBs TLB misses handled in hardware	1 TLB for instructions and 1 TLB for data per core Both L1 TLBs are four-way set associative, LRU replacement L1 I-TLB has 128 entries for small pages, seven per thread for large pages L1 D-TLB has 64 entries for small pages, 32 for large pages The L2 TLB is four-way set associative, LRU replacement The L2 TLB has 512 entries TLB misses handled in hardware

Summary: virtual memory

- The Principle of Locality:
  - Program likely to access a relatively small portion of the address space at any instant of time.
    - Temporal Locality: Locality in Time
    - Spatial Locality: Locality in Space
- Caches, TLBs, Virtual Memory all understood by examining how they deal with the four questions
  1. Where can entry be placed?
  2. How is entry found?
  3. What entry is replaced on miss?
  4. How are writes handled?
- Page tables map virtual address to physical address
  - TLBs are important for fast translation

Other considerations

Cache Coherence Problem

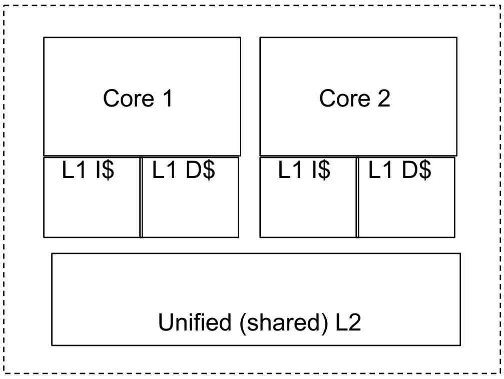
§5.10 Parallelism and Memory Hierarchies: Cache Coherence

- Suppose two CPU cores share a physical address space
  - Write-through caches

Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1

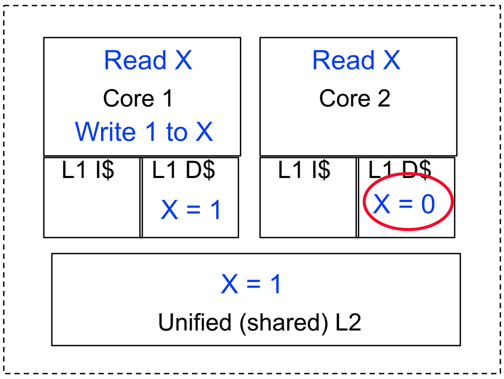
Cache Coherence in Multicores

- In future multicore processors its likely that the cores will *share* a common physical address space, causing a cache coherence problem



Cache Coherence in Multicores

- In multicore processors its likely that the cores will *share* a common physical address space, causing a cache coherence problem



Read X

Core 1

Write 1 to X

Read X

Core 2

L1 I$

L1 D$

X = 1

L1 I$

L1 D$

X = 0

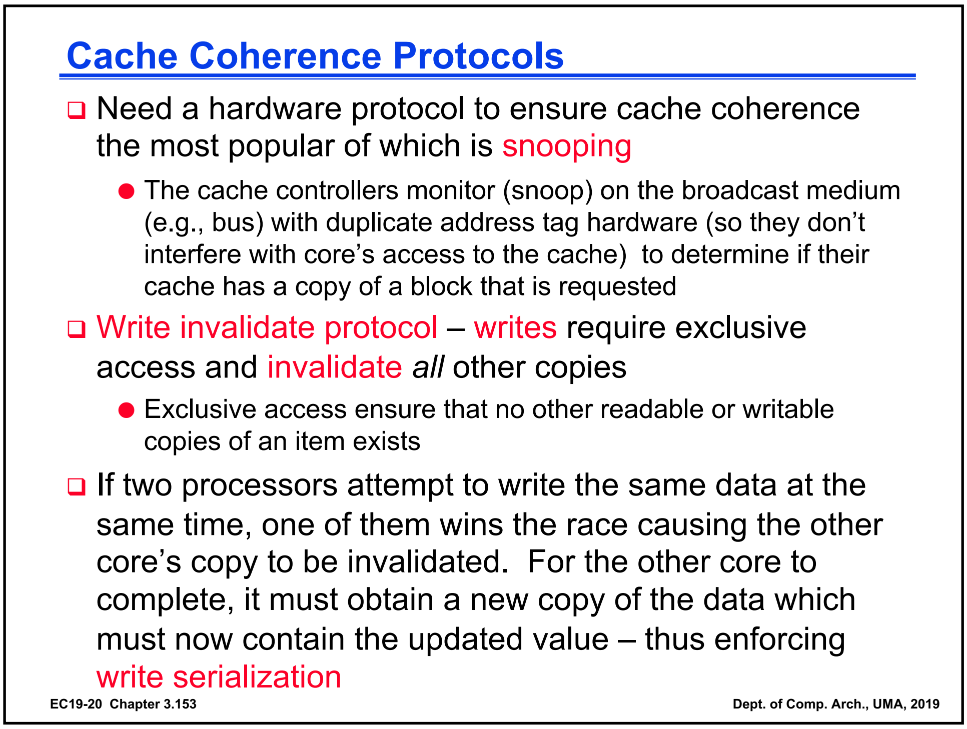
X = 1

Unified (shared) L2

A Coherent Memory System

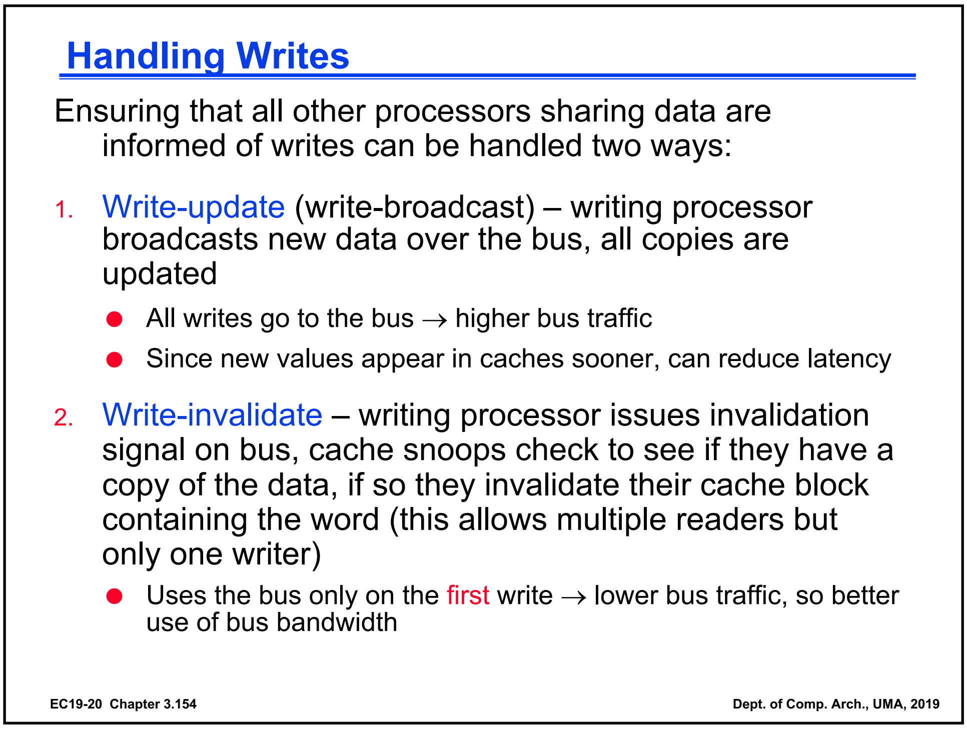
- Any read of a data item should return the most recently written value of the data item
  - Coherence – defines what values can be returned by a read
    - Writes to the same location are serialized (two writes to the same location must be seen in the same order by all cores)
  - Consistency – determines when a written value will be returned by a read

- To enforce coherence, caches must provide
  - Replication of shared data items in multiple cores’ caches
    - Replication reduces both latency and contention for a read shared data item
  - Migration of shared data items to a core’s local cache
    - Migration reduced the latency of the access the data and the bandwidth demand on the shared memory (L2 in our example)



Cache Coherence Protocols

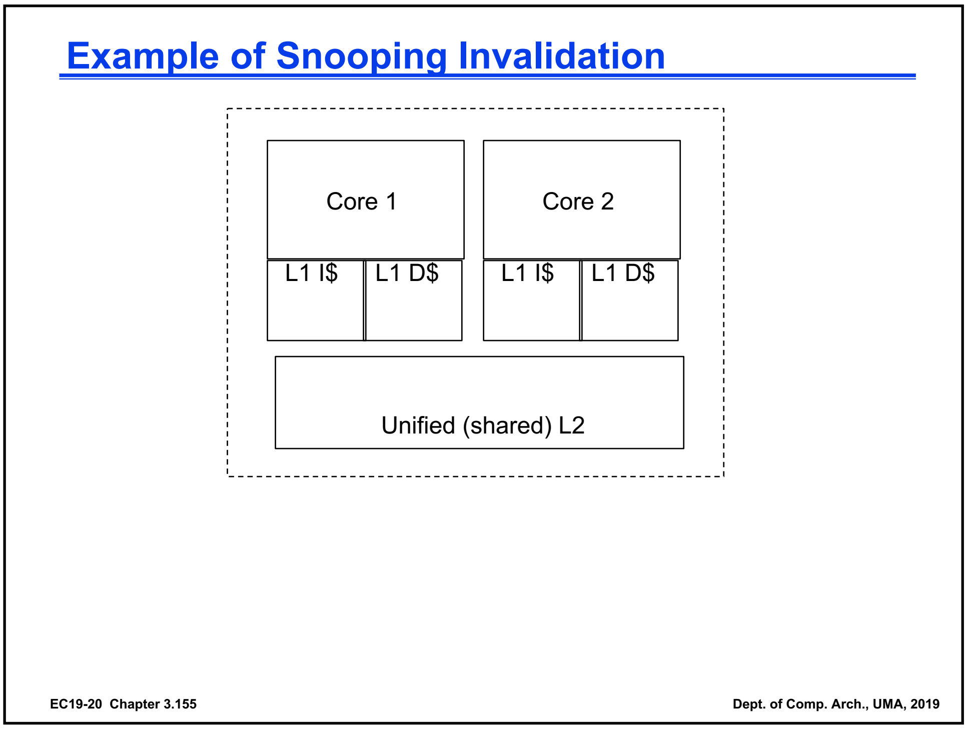
- Need a hardware protocol to ensure cache coherence the most popular of which is snooping
  - The cache controllers monitor (snoop) on the broadcast medium (e.g., bus) with duplicate address tag hardware (so they don't interfere with core's access to the cache) to determine if their cache has a copy of a block that is requested
- Write invalidate protocol – writes require exclusive access and invalidate *all* other copies
  - Exclusive access ensure that no other readable or writable copies of an item exists
- If two processors attempt to write the same data at the same time, one of them wins the race causing the other core's copy to be invalidated. For the other core to complete, it must obtain a new copy of the data which must now contain the updated value – thus enforcing write serialization



Handling Writes

Ensuring that all other processors sharing data are informed of writes can be handled two ways:

1. Write-update (write-broadcast) – writing processor broadcasts new data over the bus, all copies are updated
   - All writes go to the bus → higher bus traffic
   - Since new values appear in caches sooner, can reduce latency
2. Write-invalidate – writing processor issues invalidation signal on bus, cache snoops check to see if they have a copy of the data, if so they invalidate their cache block containing the word (this allows multiple readers but only one writer)
   - Uses the bus only on the first write → lower bus traffic, so better use of bus bandwidth



Example of Snooping Invalidation

Core 1

Core 2

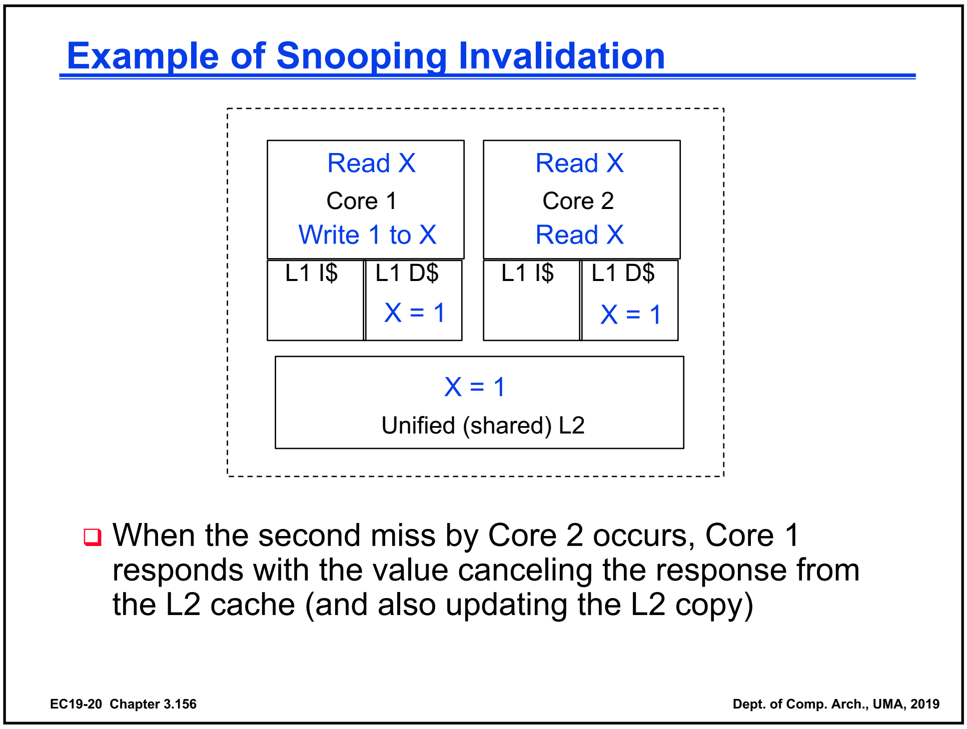
L1 I$

L1 D$

L1 I$

L1 D$

Unified (shared) L2



Example of Snooping Invalidation

Read X

Core 1

Write 1 to X

Read X

Core 2

Read X

L1 I$

L1 D$

X = 1

L1 I$

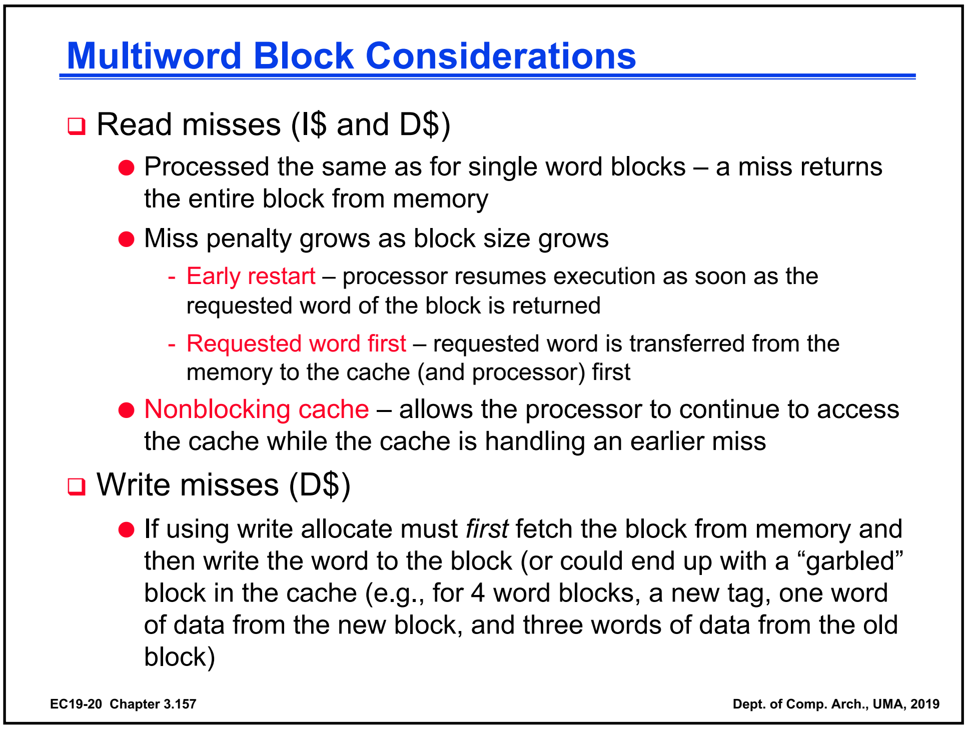
L1 D$

X = 1

X = 1

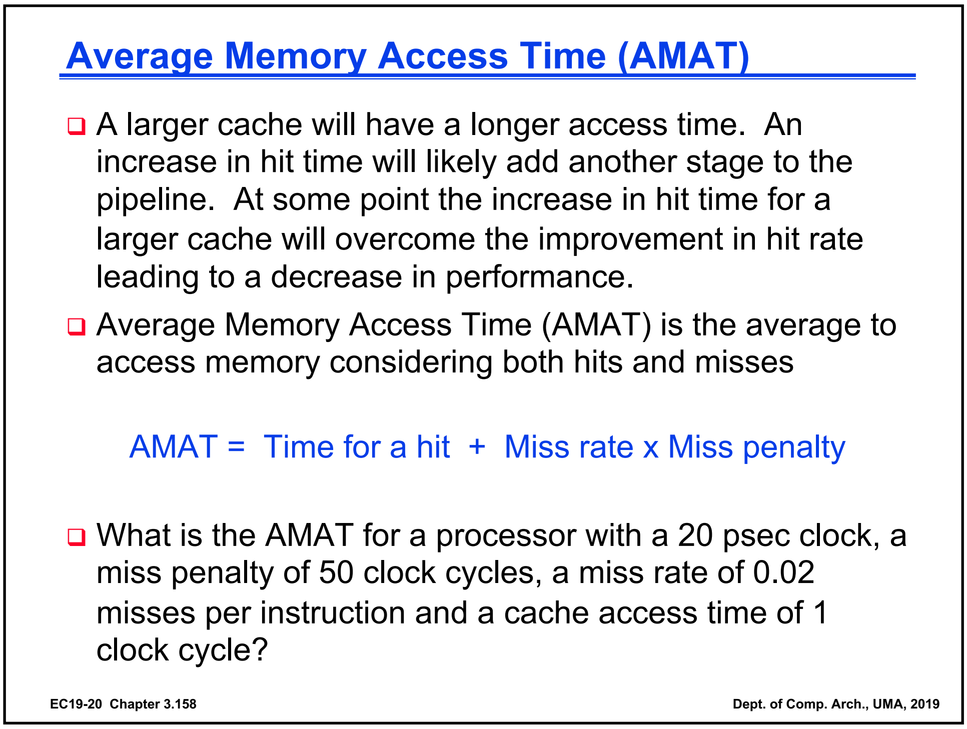
Unified (shared) L2

- When the second miss by Core 2 occurs, Core 1 responds with the value canceling the response from the L2 cache (and also updating the L2 copy)



Multiword Block Considerations

- Read misses (I$ and D$)
  - Processed the same as for single word blocks – a miss returns the entire block from memory
  - Miss penalty grows as block size grows
    - Early restart – processor resumes execution as soon as the requested word of the block is returned
    - Requested word first – requested word is transferred from the memory to the cache (and processor) first
  - Nonblocking cache – allows the processor to continue to access the cache while the cache is handling an earlier miss
- Write misses (D$)
  - If using write allocate must *first* fetch the block from memory and then write the word to the block (or could end up with a "garbled" block in the cache (e.g., for 4 word blocks, a new tag, one word of data from the new block, and three words of data from the old block)

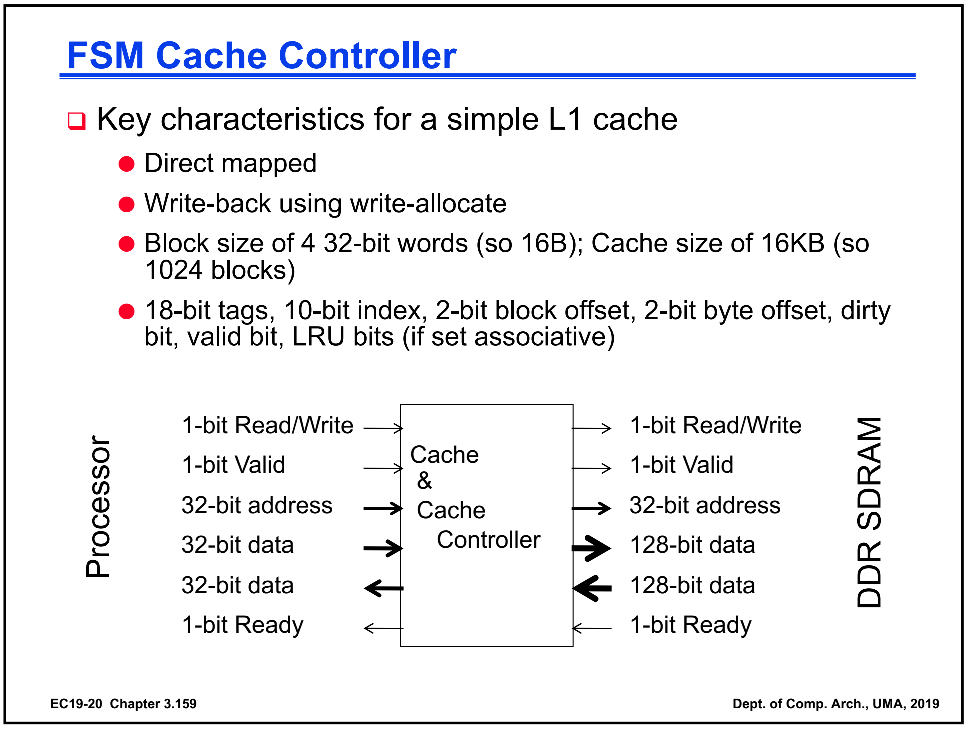


Average Memory Access Time (AMAT)

- A larger cache will have a longer access time. An increase in hit time will likely add another stage to the pipeline. At some point the increase in hit time for a larger cache will overcome the improvement in hit rate leading to a decrease in performance.
- Average Memory Access Time (AMAT) is the average to access memory considering both hits and misses

AMAT = Time for a hit + Miss rate x Miss penalty

- What is the AMAT for a processor with a 20 psec clock, a miss penalty of 50 clock cycles, a miss rate of 0.02 misses per instruction and a cache access time of 1 clock cycle?



FSM Cache Controller

- Key characteristics for a simple L1 cache
  - Direct mapped
  - Write-back using write-allocate
  - Block size of 4 32-bit words (so 16B); Cache size of 16KB (so 1024 blocks)
  - 18-bit tags, 10-bit index, 2-bit block offset, 2-bit byte offset, dirty bit, valid bit, LRU bits (if set associative)

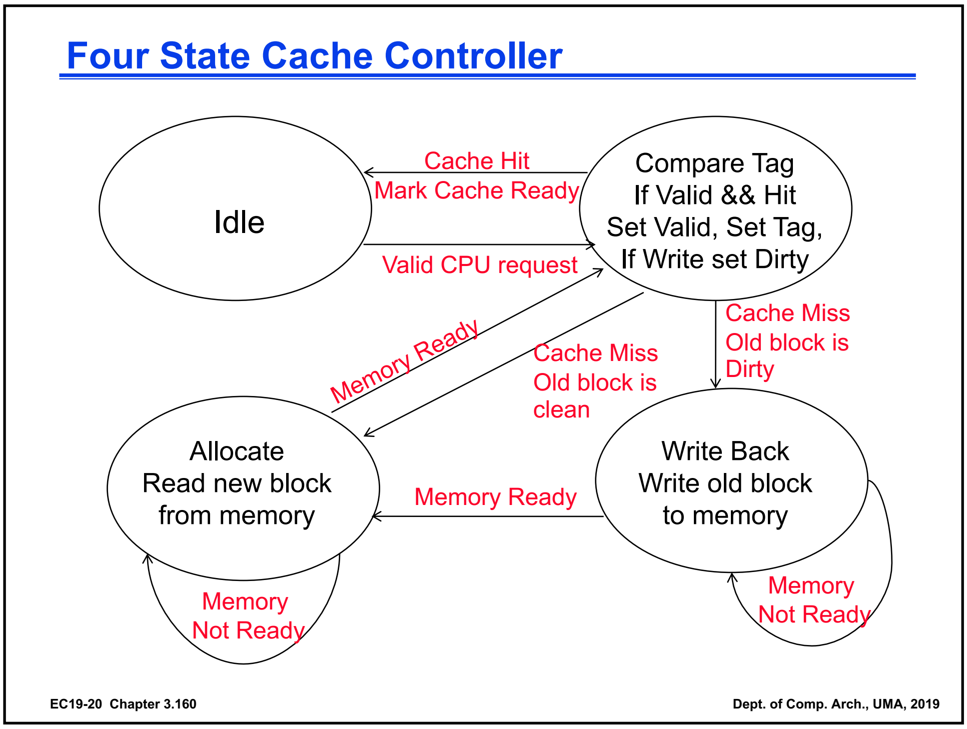
Processor

1-bit Read/Write →
1-bit Valid →
32-bit address →
32-bit data →
32-bit data ←
1-bit Ready ←

Cache & Cache Controller

→ 1-bit Read/Write
→ 1-bit Valid
→ 32-bit address
→ 128-bit data
← 128-bit data
← 1-bit Ready

DDR SDRAM



Four State Cache Controller

Idle

Compare Tag
If Valid && Hit
Set Valid, Set Tag,
If Write set Dirty

Allocate
Read new block
from memory

Write Back
Write old block
to memory

Cache Hit
Mark Cache Ready

Valid CPU request

Memory Ready

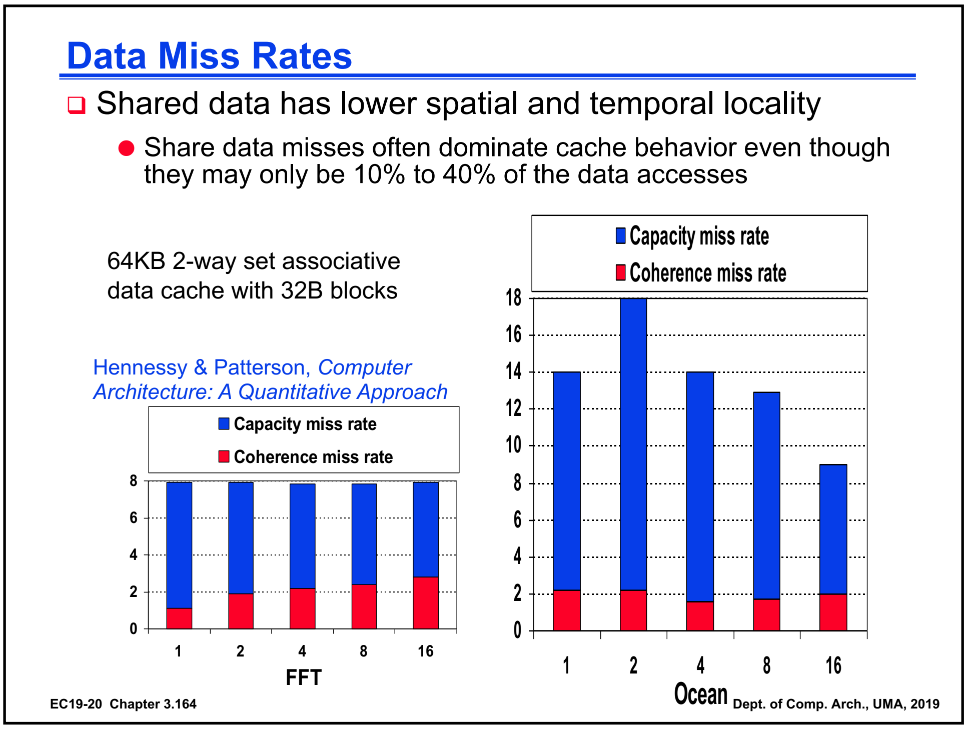
Cache Miss
Old block is
clean

Cache Miss
Old block is
Dirty

Memory Ready

Memory
Not Ready

Memory
Not Ready

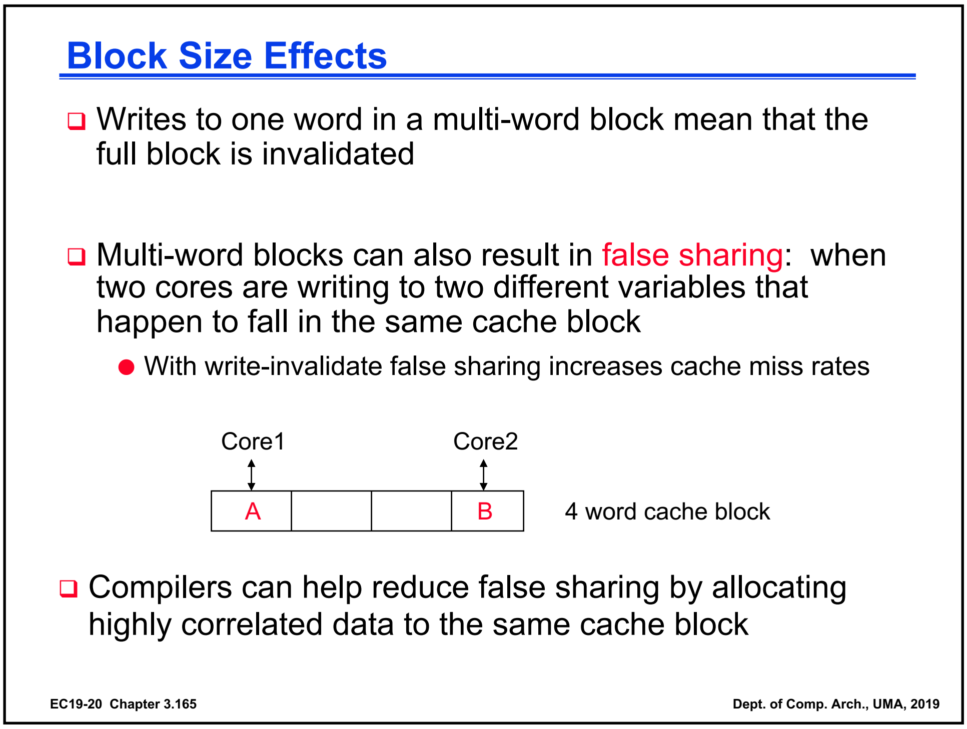


Data Miss Rates

- Shared data has lower spatial and temporal locality
  - Share data misses often dominate cache behavior even though they may only be 10% to 40% of the data accesses

64KB 2-way set associative data cache with 32B blocks

Hennessy & Patterson, *Computer Architecture: A Quantitative Approach*



Block Size Effects

- Writes to one word in a multi-word block mean that the full block is invalidated
- Multi-word blocks can also result in false sharing: when two cores are writing to two different variables that happen to fall in the same cache block
  - With write-invalidate false sharing increases cache miss rates

Core1 Core2

A B 4 word cache block

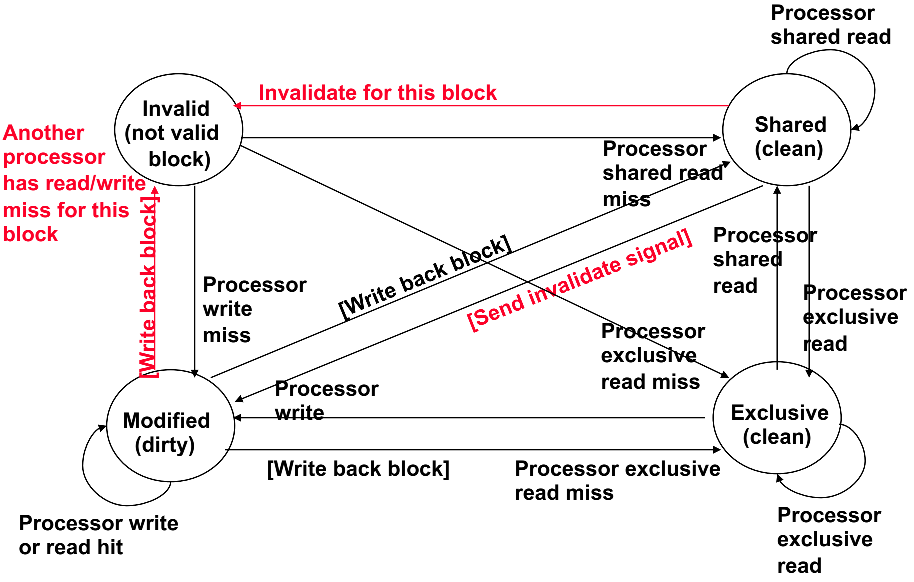
- Compilers can help reduce false sharing by allocating highly correlated data to the same cache block

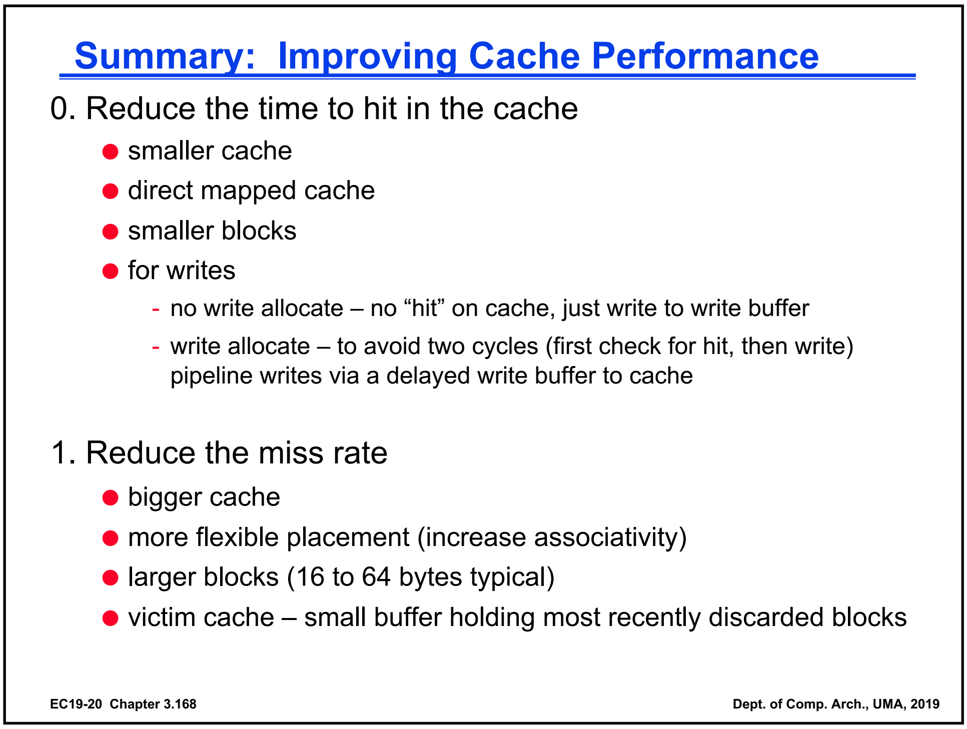
Other Coherence Protocols

- ❑ There are many variations on cache coherence protocols

- ❑ Another write-invalidate protocol used in the Pentium 4 (and many other processors) is MESI with four states:
  - Modified – same
  - Exclusive – only one copy of the shared data is allowed to be cached; memory has an up-to-date copy
    - Since there is only one copy of the block, write hits don't need to send invalidate signal
  - Shared – multiple copies of the shared data may be cached (i.e., data permitted to be cached with more than one processor); memory has an up-to-date copy
  - Invalid – same

MESI Cache Coherency Protocol





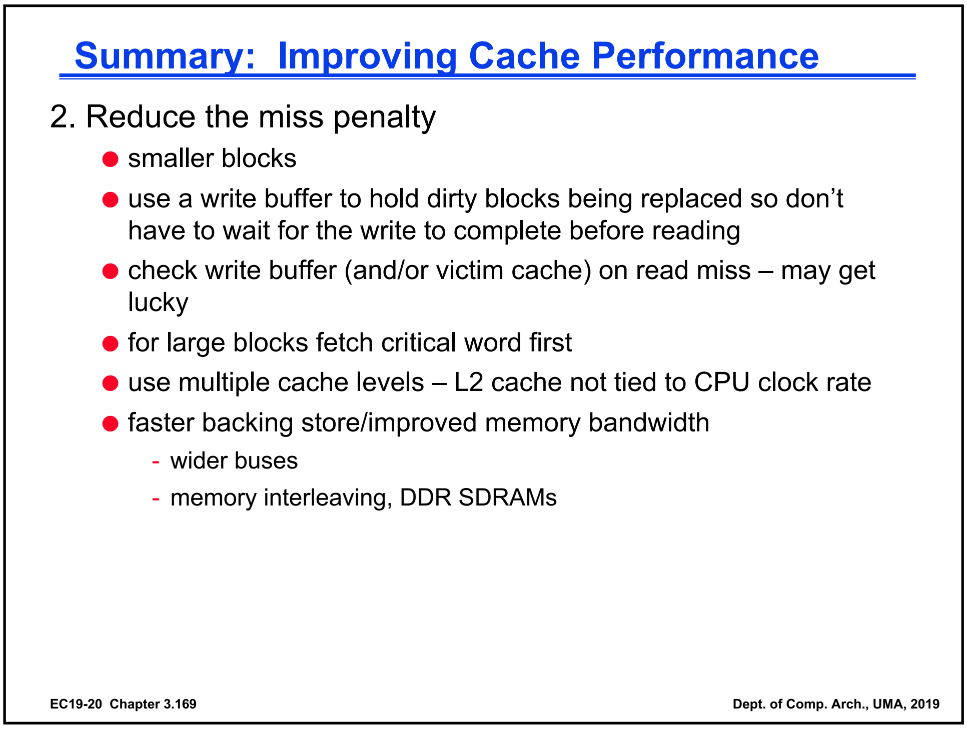
Summary: Improving Cache Performance

0. Reduce the time to hit in the cache

- smaller cache
- direct mapped cache
- smaller blocks
- for writes
  - no write allocate – no "hit" on cache, just write to write buffer
  - write allocate – to avoid two cycles (first check for hit, then write) pipeline writes via a delayed write buffer to cache

1. Reduce the miss rate

- bigger cache
- more flexible placement (increase associativity)
- larger blocks (16 to 64 bytes typical)
- victim cache – small buffer holding most recently discarded blocks



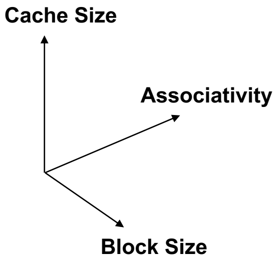
Summary: Improving Cache Performance

2. Reduce the miss penalty

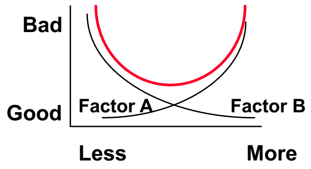
- smaller blocks
- use a write buffer to hold dirty blocks being replaced so don't have to wait for the write to complete before reading
- check write buffer (and/or victim cache) on read miss – may get lucky
- for large blocks fetch critical word first
- use multiple cache levels – L2 cache not tied to CPU clock rate
- faster backing store/improved memory bandwidth
  - wider buses
  - memory interleaving, DDR SDRAMs

Summary: The Cache Design Space

- ❑ Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back
  - write allocation



- ❑ The optimal choice is a compromise
  - depends on access characteristics
    - workload
    - use (I$ache, D$ache, TLB)
  - depends on technology / cost



- ❑ Simplicity often wins

Handling a TLB Miss

- ❑ Consider a TLB miss for a page that is present in memory (i.e., the Valid bit in the page table is set)
  - A TLB miss (or a page fault exception) must be asserted by the end of the same clock cycle that the memory access occurs so that the next clock cycle will begin exception processing

Register	CP0 Reg #	Description
`EPC`	14	Where to restart after exception
`Cause`	13	Cause of exception
`BadVAddr`	8	Address that caused exception
`Index`	0	Location in TLB to be read/written
`Random`	1	Pseudorandom location in TLB
`EntryLo`	2	Physical page address and flags
`EntryHi`	10	Virtual page address
`Context`	4	Page table address & page number

A MIPS Software TLB Miss Handler

- When a TLB miss occurs, the hardware saves the address that caused the miss in `BadVAddr` and transfers control to 8000 0000_{hex}, the location of the TLB miss handler

```
TLBmiss:
 mfc0   $k1, Context    #copy addr of PTE into $k1
 lw     $k1, 0($k1)     #put PTE into $k1
 mtc0   $k1, EntryLo    #put PTE into EntryLo
 tlbwr                  #put EntryLo into TLB
                        #     at Random
 eret                   #return from exception
```

- `tlbwr` copies from `EntryLo` into the TLB entry selected by the control register `Random`

- A TLB miss takes about a dozen clock cycles to handle

Some Virtual Memory Design Parameters

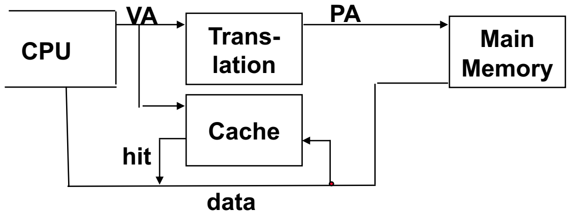
	Paged VM	TLBs
Total size	16,000 to 250,000 words	16 to 512 entries
Total size (KB)	250,000 to 1,000,000,000	0.25 to 16
Block size (B)	4000 to 64,000	4 to 8
Hit time		0.5 to 1 clock cycle
Miss penalty (clocks)	10,000,000 to 100,000,000	10 to 100
Miss rates	0.00001% to 0.0001%	0.01% to 1%

Two Machines’ TLB Parameters

	Intel Nehalem	AMD Barcelona
Address sizes	48 bits (vir); 44 bits (phy)	48 bits (vir); 48 bits (phy)
Page size	4KB	4KB
TLB organization	L1 TLB for instructions and L1 TLB for data per core; both are 4-way set assoc.; LRU L1 ITLB has 128 entries, L2 DTLB has 64 entries L2 TLB (unified) is 4-way set assoc.; LRU L2 TLB has 512 entries TLB misses handled in hardware	L1 TLB for instructions and L1 TLB for data per core; both are fully assoc.; LRU L1 ITLB and DTLB each have 48 entries L2 TLB for instructions and L2 TLB for data per core; each are 4-way set assoc.; round robin LRU Both L2 TLBs have 512 entries TLB misses handled in hardware

Why Not a Virtually Addressed Cache?

- ❑ A virtually addressed cache would only require address translation on cache misses



but

- Two programs which are sharing data will have two different virtual addresses for the same physical address – aliasing – so have two copies of the shared data in the cache and two entries in the TBL which would lead to coherence issues
  - Must update all cache entries with the same physical address or the memory becomes inconsistent