Chapter 3: Large and Fast – Exploiting the Memory Hierarchy

JOSÉ Mª GONZÁLEZ LINARES

DEPT. DE ARQUITECTURA DE COMPUTADORES

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA

Index

The Memory Hierarchy Goal

Large memories are slow and fast memories are small

But we need infinitely large and fast memories

How do we create a memory that gives the illusion of being large, fast and cheap most of the time?

- Taking advantage of principle of locality
- Using a hierarchical organization
- With parallelism

Principle of Locality

Programs access a small proportion of their address space at any time

Temporal locality

Items accessed recently are likely to be accessed again soon

E.g., instructions in a loop, induction variables

Spatial locality

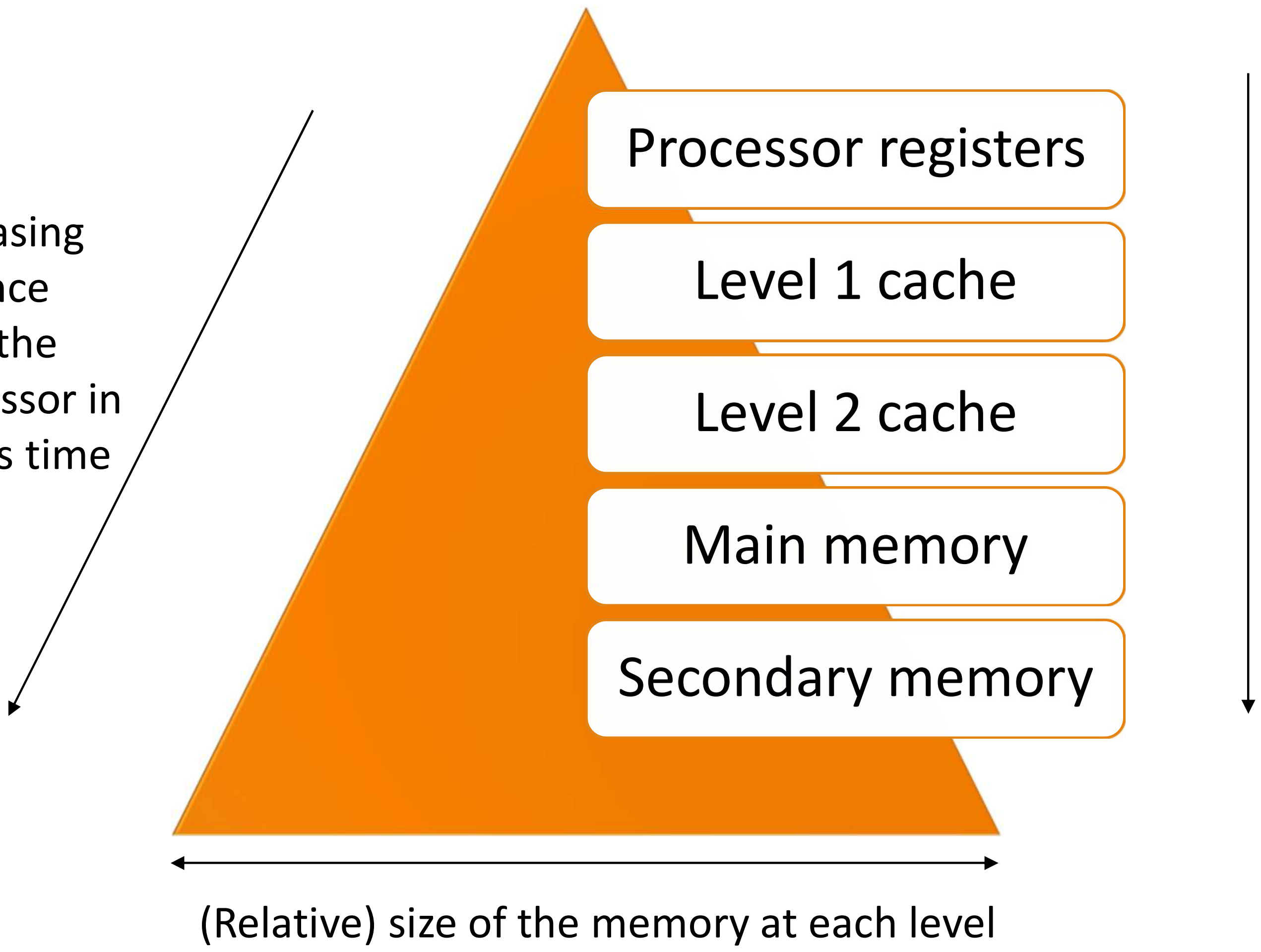
Items near those accessed recently are likely to be accessed soon

E.g., sequential instruction access, array data

A hierarchical organization

Cache memory attached to CPU	Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
Main memory	Copy recently accessed (and nearby) items from disk to smaller DRAM memory
Disk	Store everything

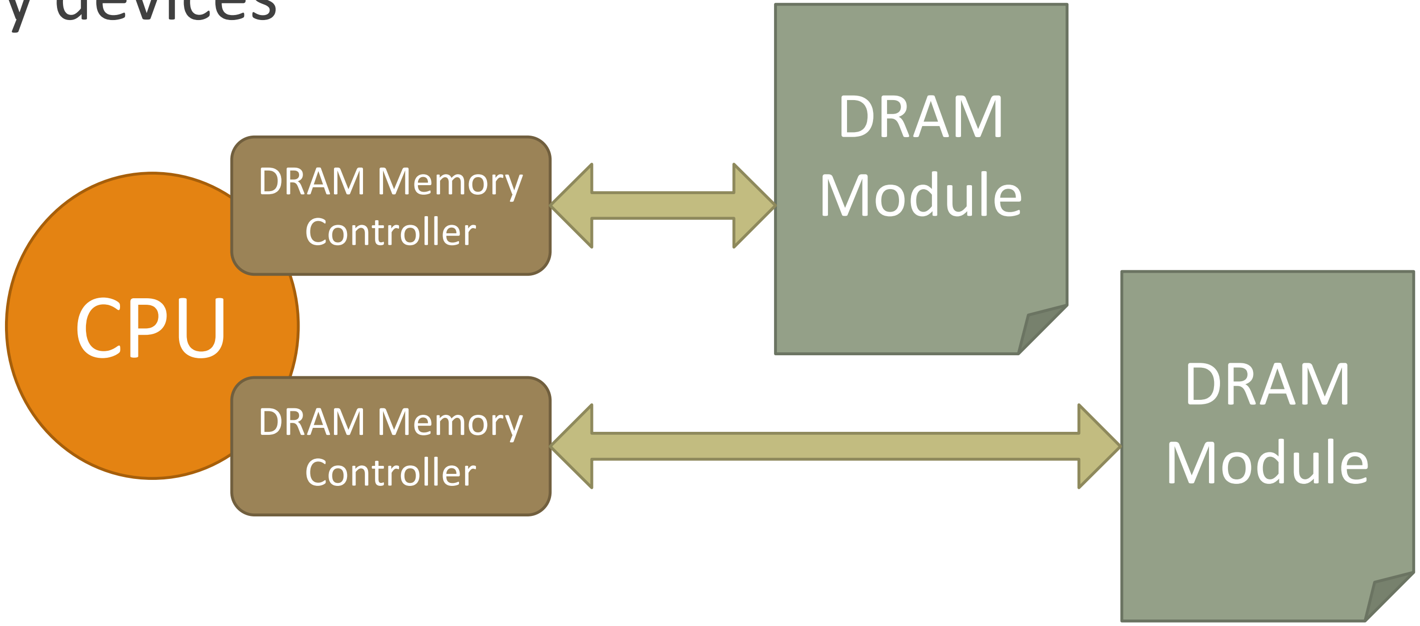
The memory hierarchy pyramid


Increasing distance from the processor in access time
Processor registers
Level 1 cache
Level 2 cache
Main memory
Secondary memory
(Relative) size of the memory at each level

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

Parallelism

Add several channels of communication between the processor and the memory devices



Memory Technology

Memory Technology

Static RAM (SRAM)

- 0.5ns – 2.5ns
- Used in caches
- Static: content will last “forever”, as long as power is left on

Dynamic RAM (DRAM)

- 30ns – 70ns
- Used in main memory
- Dynamic: needs to be “refreshed” regularly (~every 8 ms), consumes 1% to 2% of the active cycles of the DRAM

Magnetic disk and solid state devices

- 5ms – 20ms
- Used in virtual memory

DRAM Technology

Data stored as a charge in a capacitor

- Single transistor used to access the charge

Must periodically be refreshed

- Read contents and write back, performed on a DRAM “row”

Bits in a DRAM are organized as a rectangular array

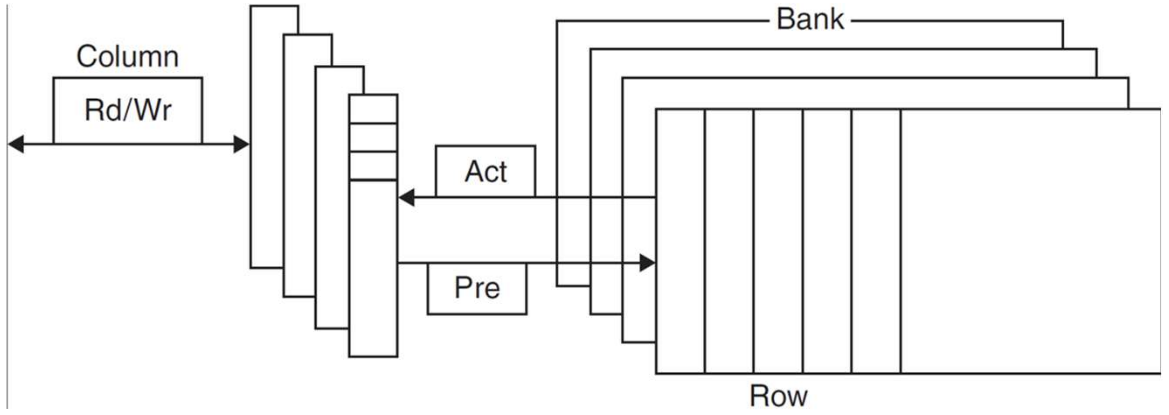
Burst mode: supply successive words from a row with reduced latency (improves bandwidth)

Double data rate (DDR) DRAM

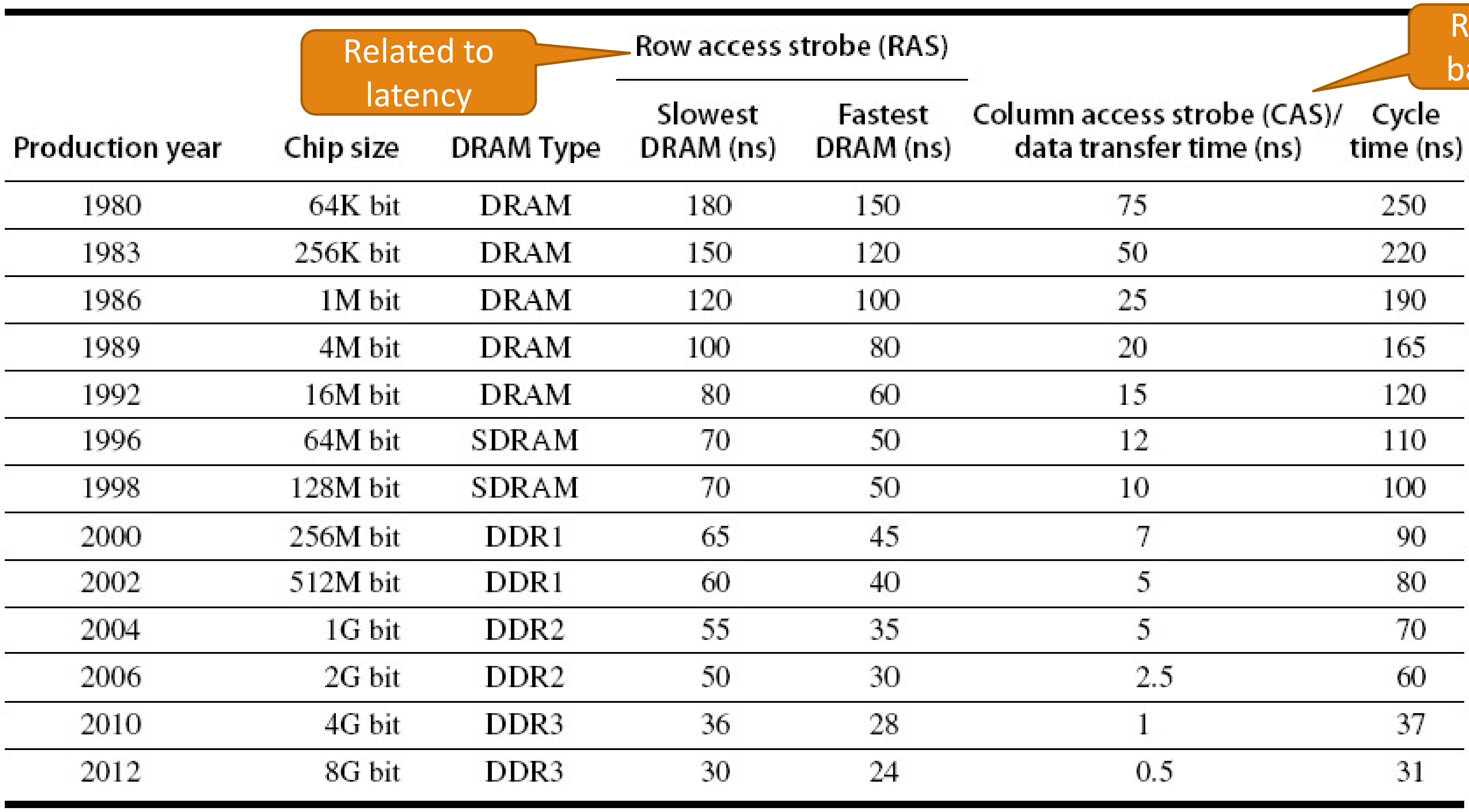
- Transfer on rising and falling clock edges

Quad data rate (QDR) DRAM

- Separate DDR inputs and outputs



Times of fast and slow DRAMs

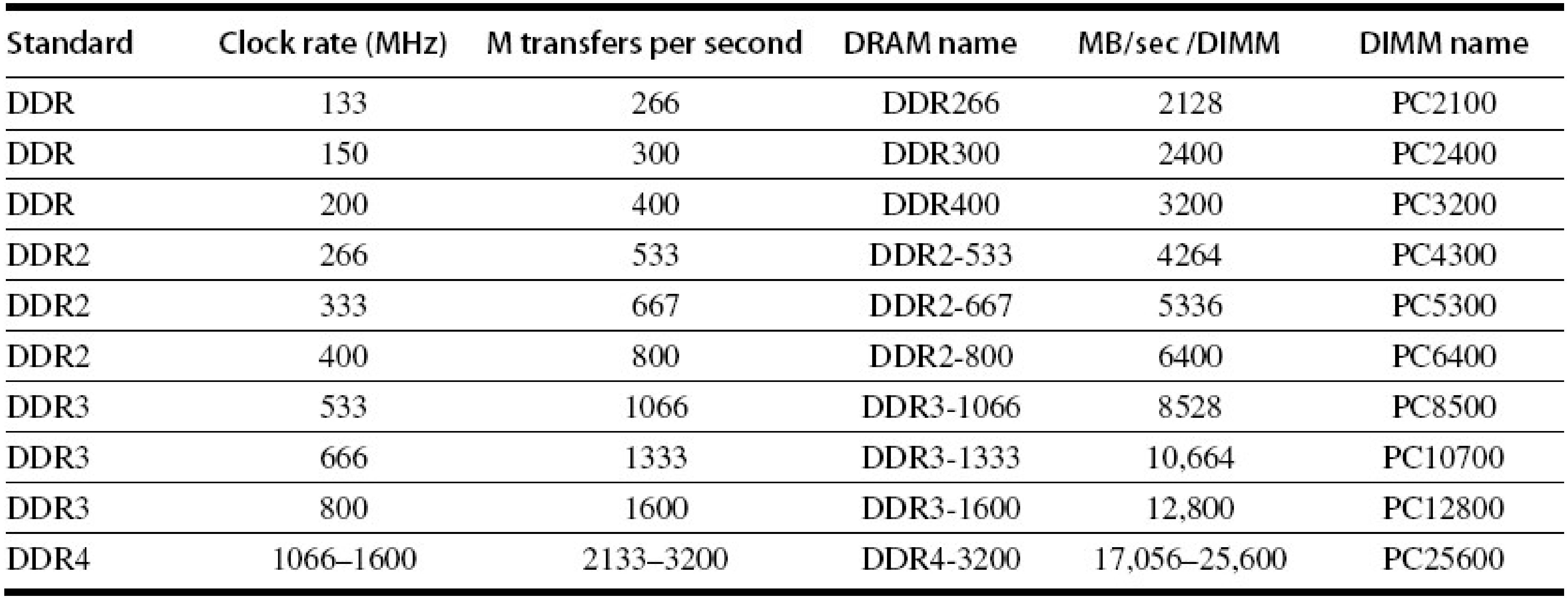


Related to latency

Related to bandwidth

			Row access strobe (RAS)			
Production year	Chip size	DRAM Type	Slowest DRAM (ns)	Fastest DRAM (ns)	Column access strobe (CAS)/ data transfer time (ns)	Cycle time (ns)
1980	64K bit	DRAM	180	150	75	250
1983	256K bit	DRAM	150	120	50	220
1986	1M bit	DRAM	120	100	25	190
1989	4M bit	DRAM	100	80	20	165
1992	16M bit	DRAM	80	60	15	120
1996	64M bit	SDRAM	70	50	12	110
1998	128M bit	SDRAM	70	50	10	100
2000	256M bit	DDR1	65	45	7	90
2002	512M bit	DDR1	60	40	5	80
2004	1G bit	DDR2	55	35	5	70
2006	2G bit	DDR2	50	30	2.5	60
2010	4G bit	DDR3	36	28	1	37
2012	8G bit	DDR3	30	24	0.5	31

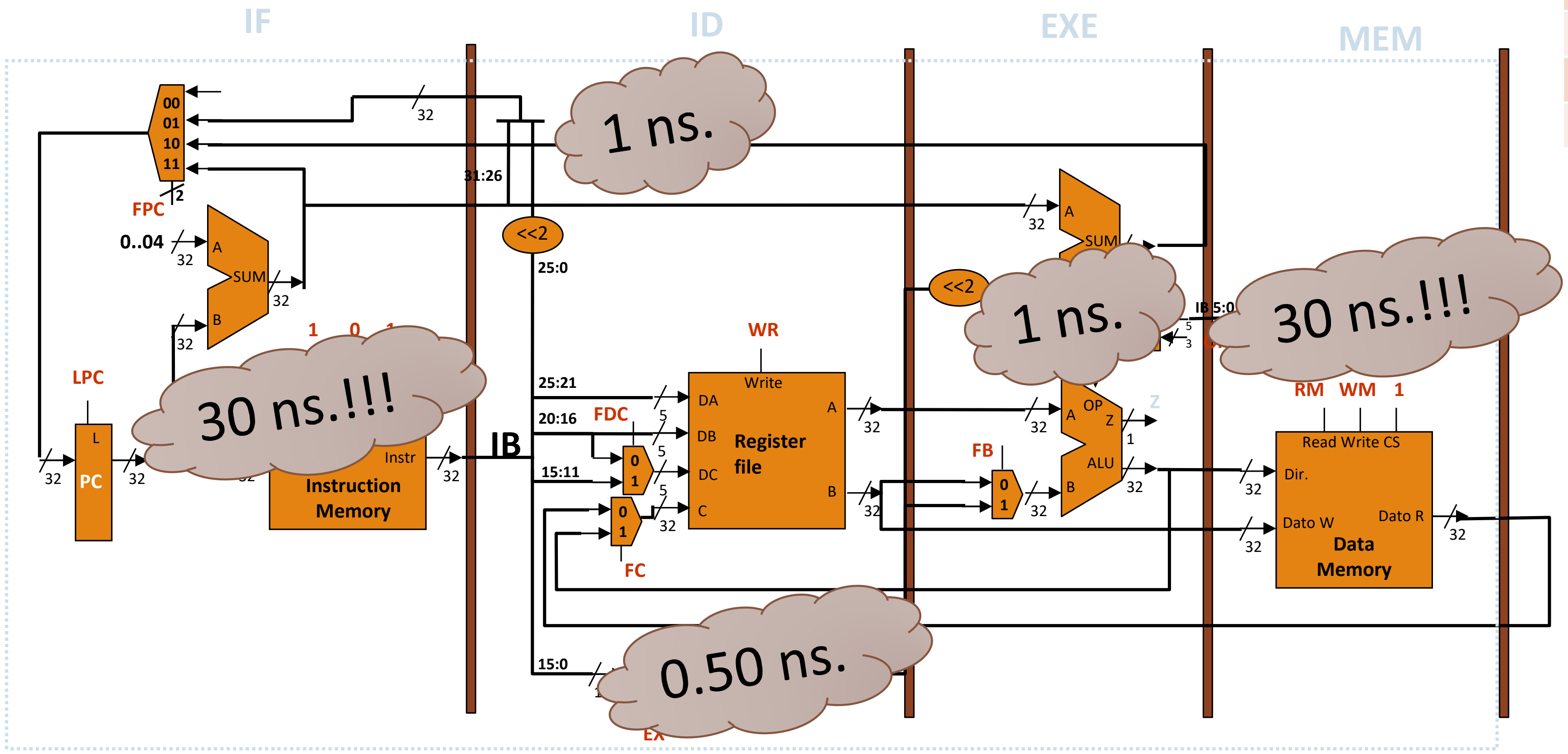
Clock rates and bandwidth



Standard	Clock rate (MHz)	M transfers per second	DRAM name	MB/sec /DIMM	DIMM name
DDR	133	266	DDR266	2128	PC2100
DDR	150	300	DDR300	2400	PC2400
DDR	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	8528	PC8500
DDR3	666	1333	DDR3-1333	10,664	PC10700
DDR3	800	1600	DDR3-1600	12,800	PC12800
DDR4	1066–1600	2133–3200	DDR4-3200	17,056–25,600	PC25600

The Basics of Cache

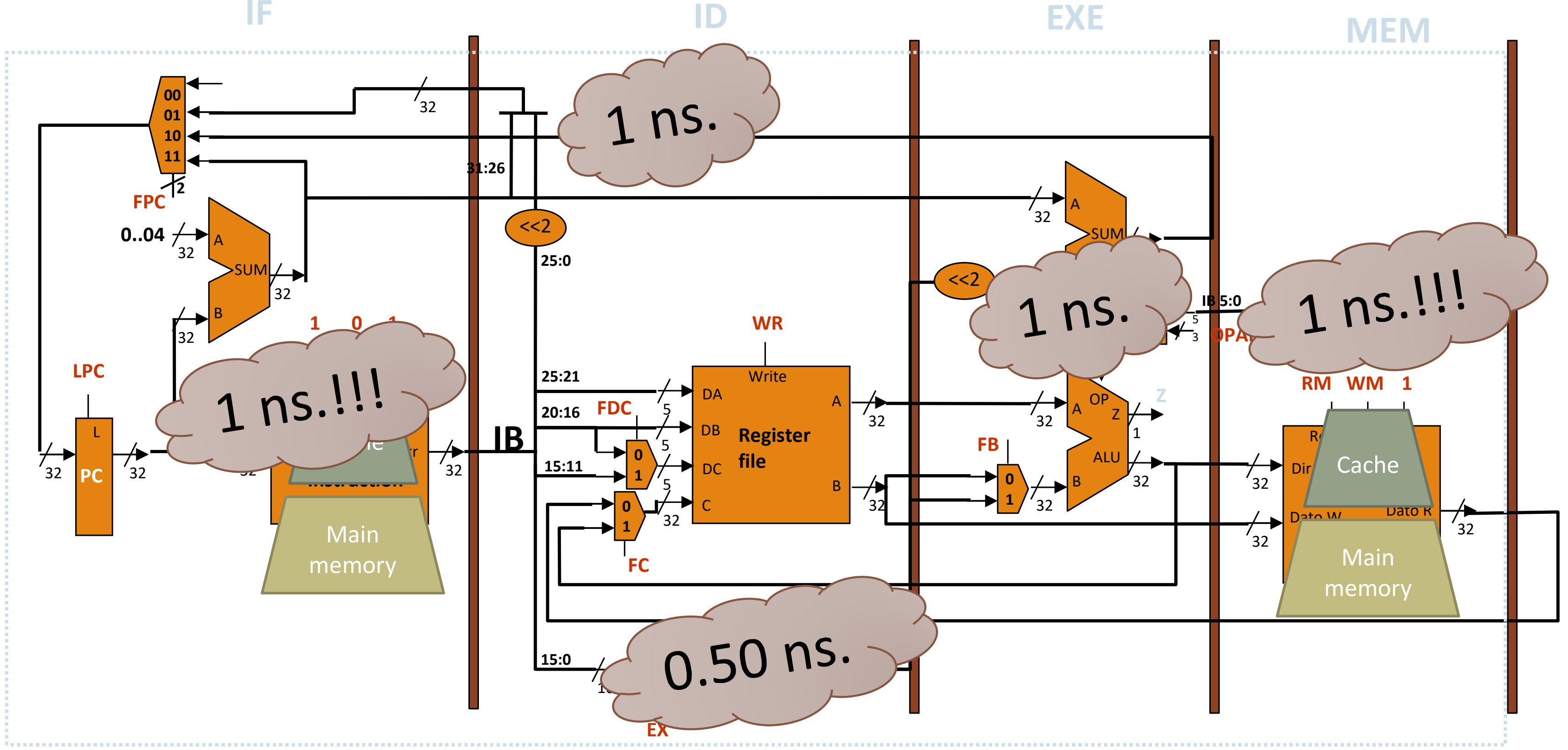
Example MIPS: no hierarchy



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

Clock cycle: 30 ns

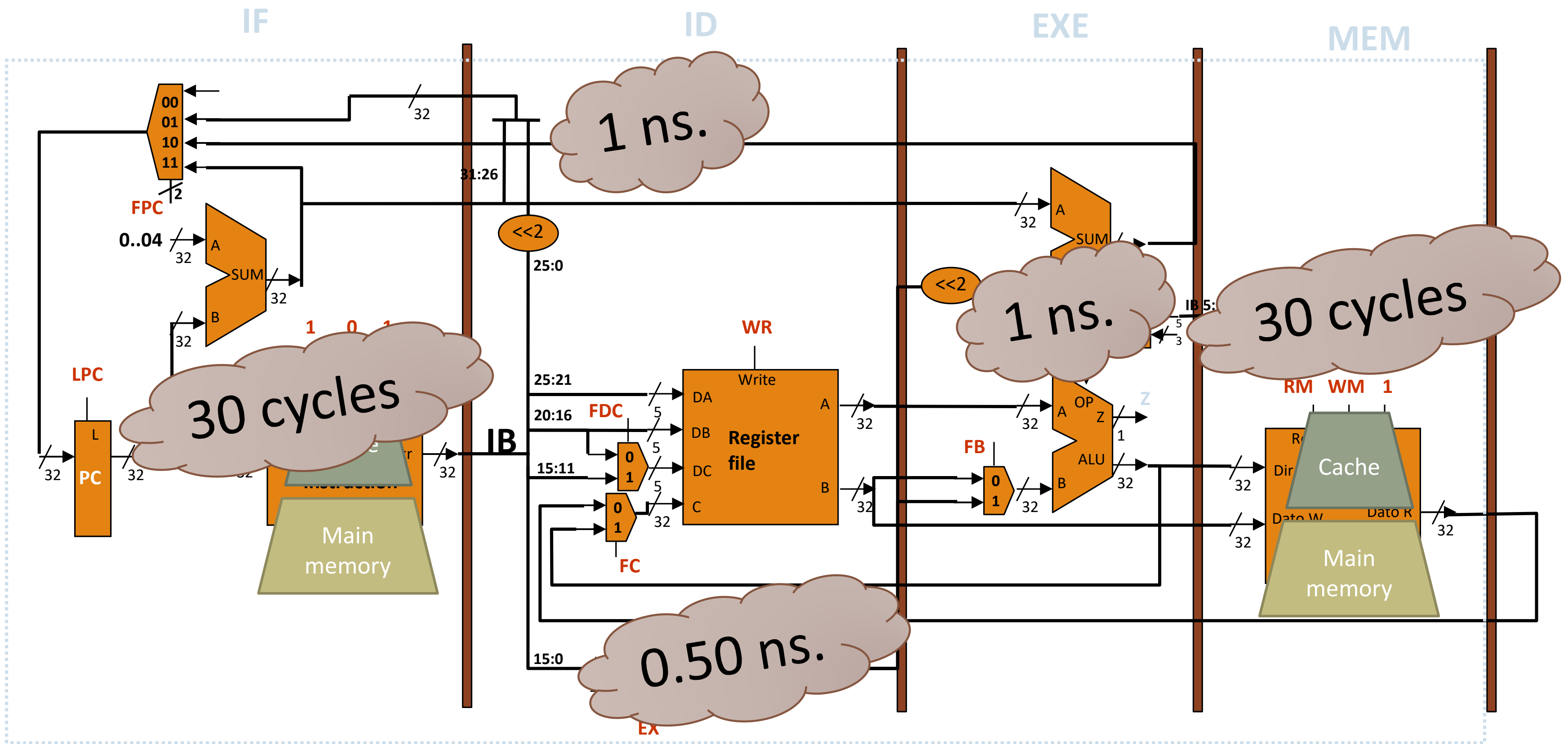
Example MIPS: memory hierarchy – hit on cache



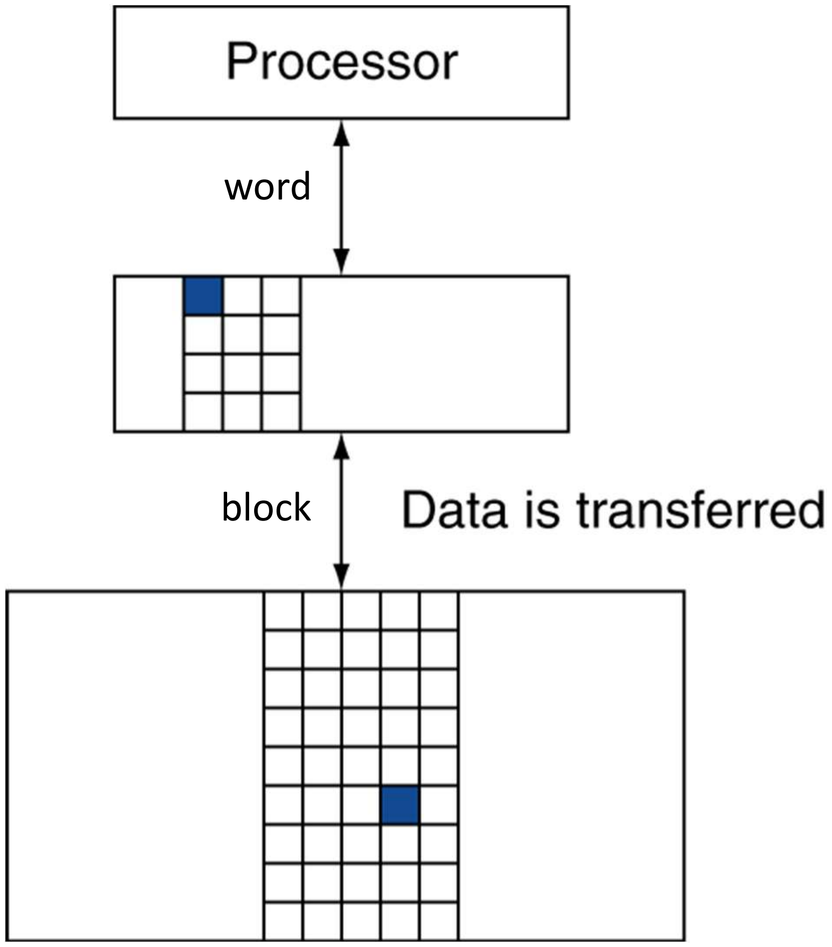
Element	T active	T inactive
Inst. M.	**1 ns**	0 ns
File Reg.	1 ns	0 ns
ALU	1 ns	0 ns
Data M.	**1 ns**	0 ns

Clock cycle: 1 ns

Example MIPS: memory hierarchy – miss on cache



Memory Hierarchy Levels



Block (aka line): unit of copying

- May be multiple words

If accessed data is present in upper level

- **Hit**: access satisfied by upper level
- Hit ratio: hits/accesses

If accessed data is absent

- **Miss**: block copied from lower level
- Time taken: miss penalty
- Miss ratio: misses/accesses = 1 – hit ratio
- Then accessed data supplied from upper level

The Memory Hierarchy: Terminology

Block (or line): the minimum unit of information that is present (or not) in a cache

Hit Rate (R_{hit}): the fraction of memory accesses found in a level of the memory hierarchy

- $R_{hit} = {}^{Number\ of\ hits}/_{Number\ of\ memory\ references}$
- 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 usually consider Hit time= 1 cc**

Miss Rate (R_{miss}): the fraction of memory accesses *not* found in a level of the memory hierarchy $\Rightarrow$ $R_{miss}=1 - R_{hit}$

- $R_{miss} = {}^{Number\ of\ miss}/_{Number\ of\ memory\ references}$
- 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})

Average Access Time

Previous values give hints about performance

A more insightful measure: 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, instructions cache 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
  - **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

$$CPU\ time = IC \times CPI \times CC = IC \times \underbrace{\left(CPI_{base} + \frac{Memory{-}st \quad cycles}{IC}\right)}_{\text{CPI}_{\text{effective}}} \times CC$$

$$Memory - stall\ cycles = \#Ref\,(I\$) \times R_{miss}(I\$) \times P_{miss}(I\$) + \#Ref\,(D\$) \times R_{miss}(D\$) \times P_{miss}(D\$)$$

$$\frac{Memory - stall\ cycles}{IC} = 1 \times R_{miss}(I\$) \times P_{miss}(I\$) + \frac{\#Ref\,(D\$) \times R_{miss}(D\$) \times P_{miss}(D\$)}{IC}$$

* *CPI$_{base}$ includes the stall due to data & control hazards (perfect cache –no misses)*
* $\#Ref\,(I\$) = IC$ *(Number of references to I$)*
* $\#Ref\,(D\$)$ =*Number of load and store instructions*

Cache Performance Example

Given

- I\$ miss rate $R_{miss}(I\$) = 2\%$
- D\$ miss rate $R_{miss}(D\$) = 4\%$
- Miss penalty (I\$ & D\$) $P_{miss}(I\$)$, $P_{miss}(D\$) = 80$ cycles
- Base CPI (ideal cache) $CPI_{base} = 1.5$
- Load & stores are 36% of instructions *(#Ref(D\$) /IC = 0.36)*

compute the effective CPI:

Miss cycles per instruction

- I\$: $0.02 \times 80 = 1.6$
- D\$: $0.36 \times 0.04 \times 80 = 1.152$

Effective CPI = 1.5 + 1.6 + 1.152 = 4.252

Performance Summary

$$AMAT = HitTime + MissRate \times MissPenalty$$

$$CPU\ time = IC \times CPI \times CC = IC \times \left(CPI_{base} + \frac{Memory\text{−}stall\ cycles}{IC}\right) \times CC$$

- When CPU performance increased
  - Miss penalty becomes more significant
- Decreasing base CPI
  - Greater proportion of time spent on memory stalls
- Increasing clock rate
  - Memory stalls account for more CPU cycles
- Can’t neglect cache behavior when evaluating system performance

Cache mapping and organization

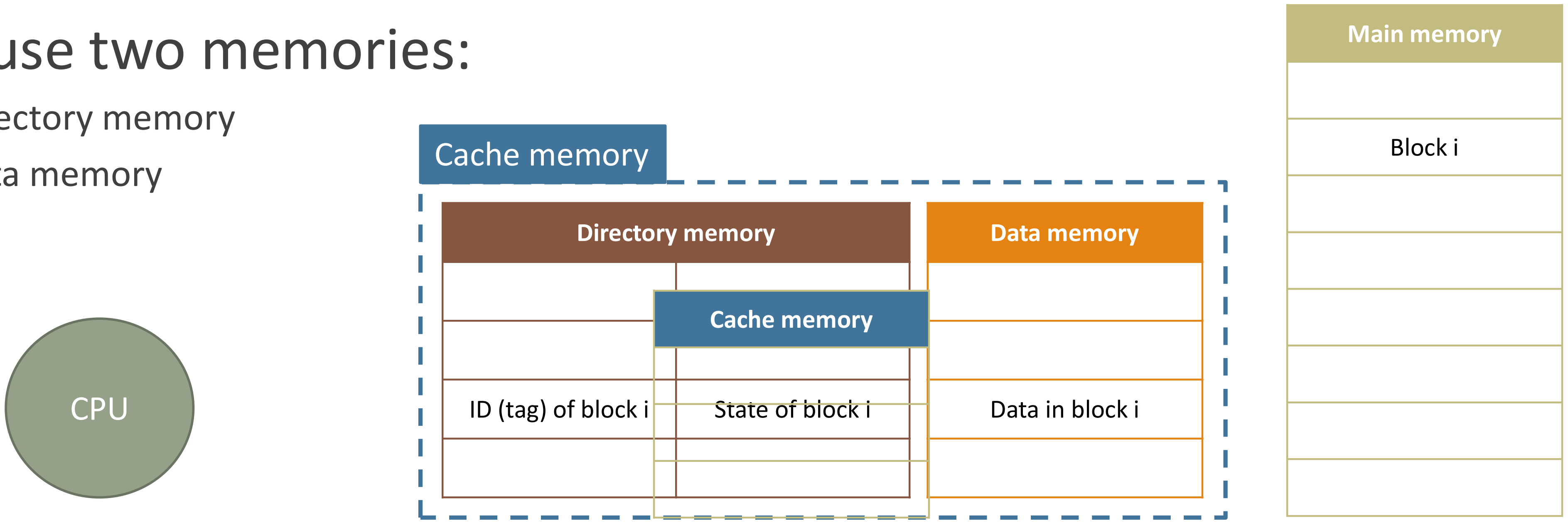
Cache mapping

Each cache line must be able to host multiple memory blocks at different times

How do we know if block *i* is present? **Where** do we look for it?

We use two memories:

- A directory memory
- A data memory



A word about memory addresses

A memory address is a reference to a memory location that stores the smallest unit of storage.

Most computers are byte-addressable.

A word is typically stored using several consecutive bytes

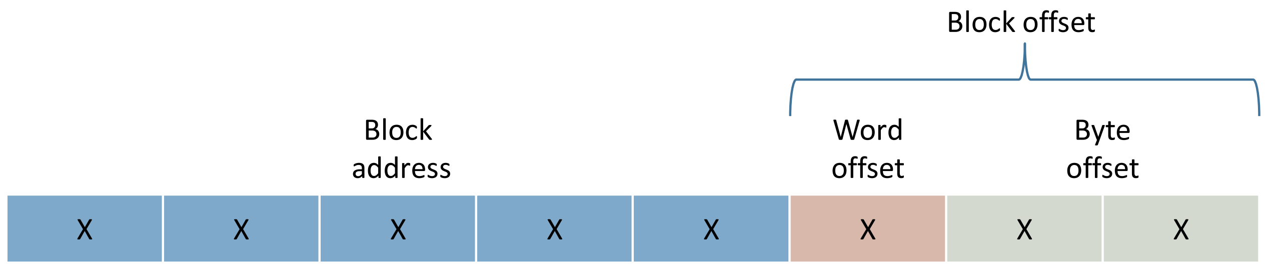
Endianness is the order of these bytes

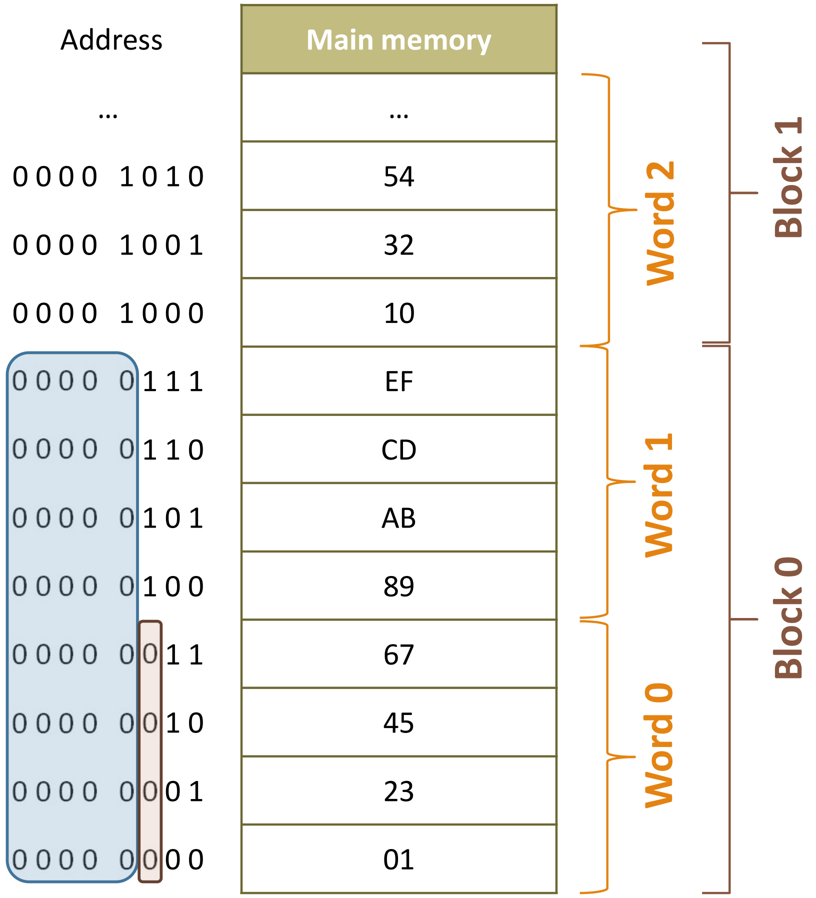
- Big-endian: most significant byte is stored at the smallest memory address
- Little-endian: least-significant byte is stored at the smallest memory address

Another word about memory addresses

A memory address is a binary sequence

Example: A byte-addressable memory of size 256 bytes, words of 32 bits and blocks of 2 words

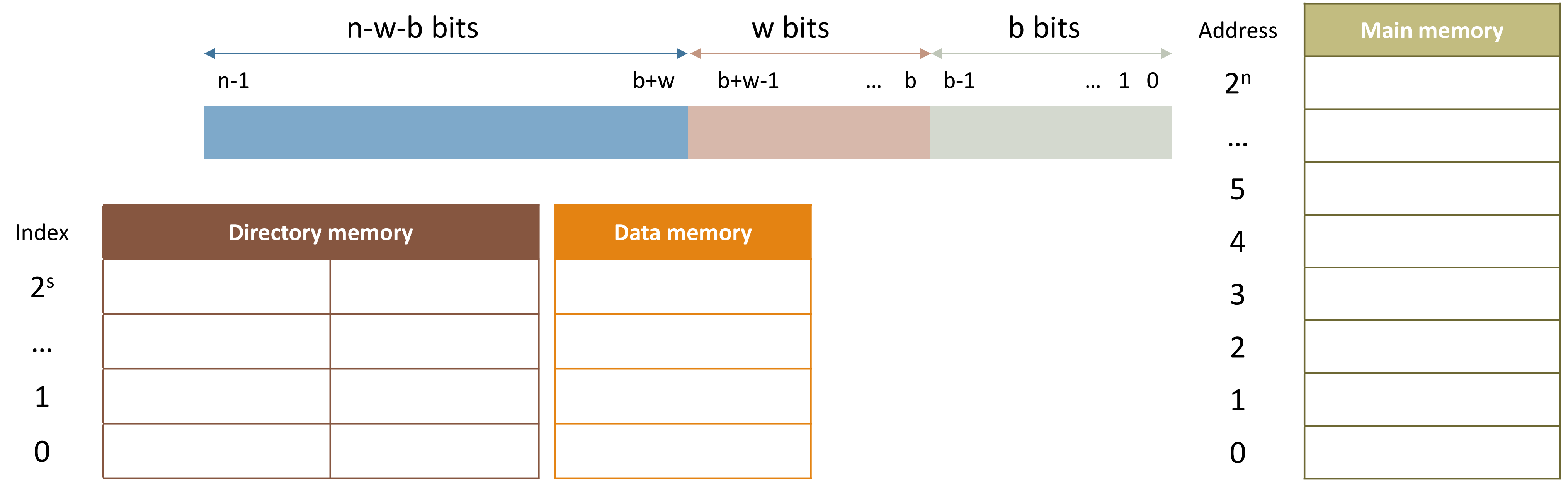




A final word about memory (and cache) addresses

Let be a main memory of size $N=2^n$, made of words with 2^b bytes, grouped into blocks of 2^w words, and a cache of size $S=2^s$ lines

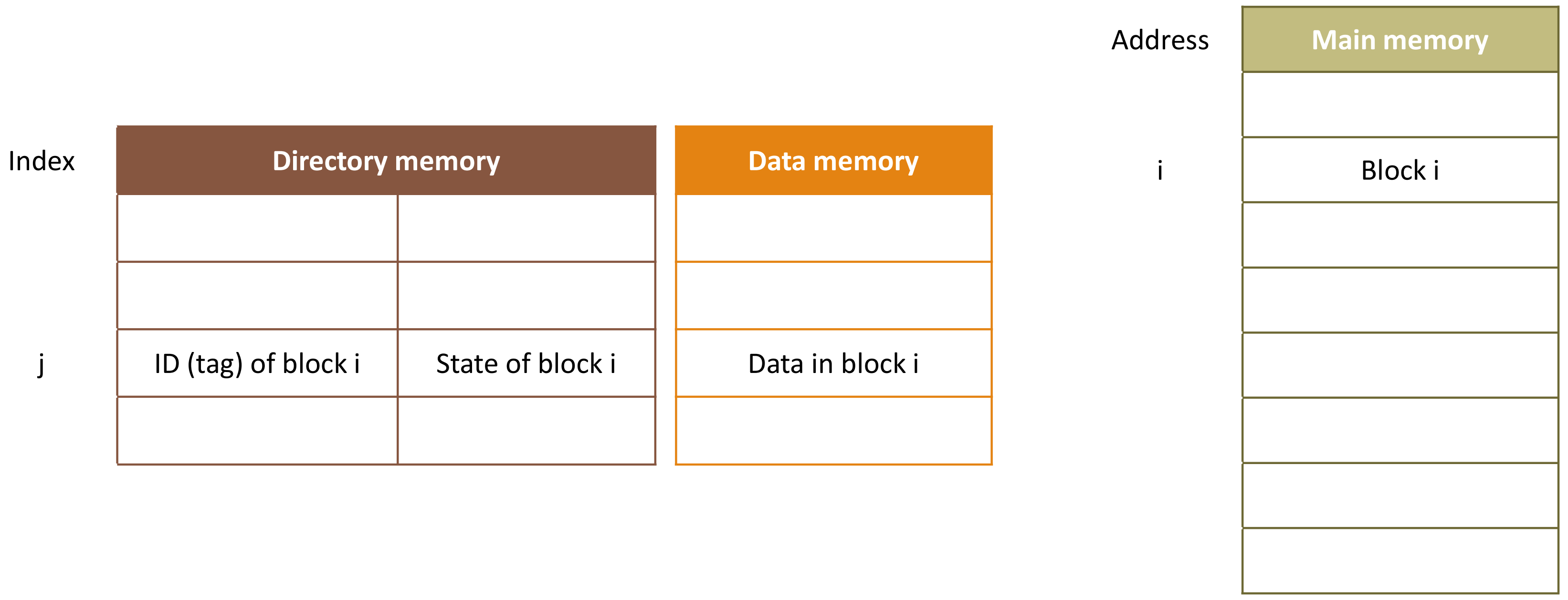
Line size is equal to block size



Cache mapping

Block address is used to map a memory block to a cache line

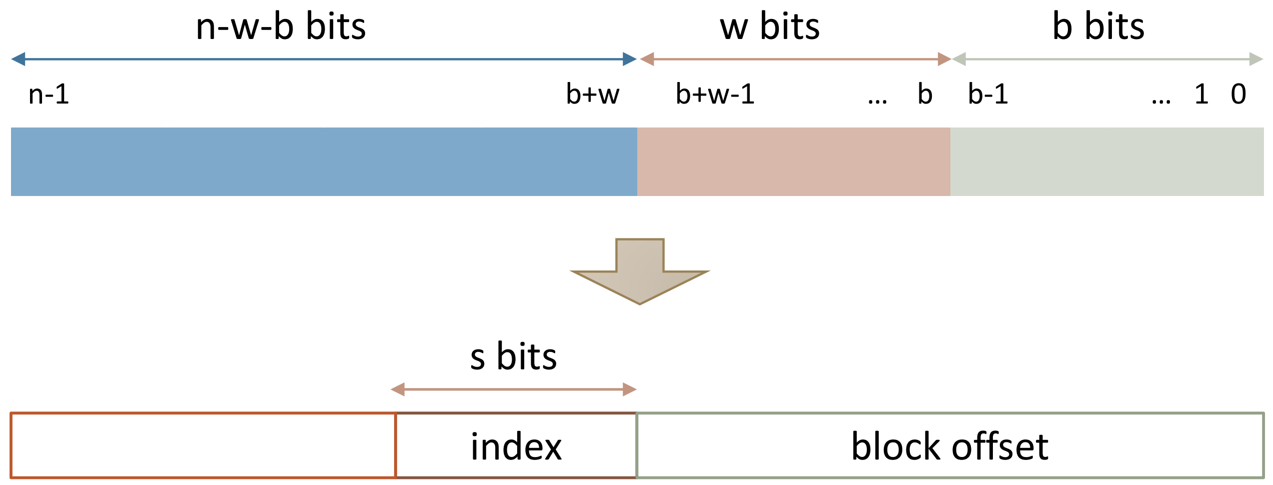
```
j, tag = mapping(i)
```



Direct mapped cache

A memory block is always mapped to the same cache line

Simplest mapping: take the *s* least significant bits of the block address



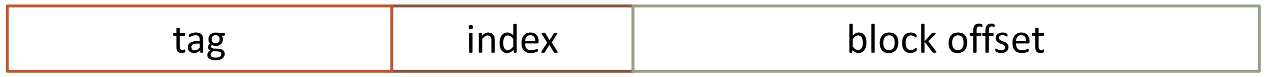
Equivalent to the modulo operation with 2^s

Tag and valid bits

Cache line with index j can store block address j, or $j + S$, or $j + 2S$ …

How do we know which particular block is stored in a cache location?

- Store block address in directory memory
- Actually only need the high-order bits, called the tag



What happens the first time, when there was no data

- Store a valid bit in directory memory
- 1 is present, 0 is not present

Direct mapped cache example

Let be a byte-addressable main memory of size N=256, with words of 4 bytes and blocks of 1 word

Cache size is S=8

Current program requests the next memory references:

88, 104, 88, 72, 104

Size of

○ Memory address:	8
○ Byte offset:	2
○ Word offset:	0
○ Block offset:	2
○ Block address:	6

Size of

○ Cache index:	3
○ Tag:	3

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Direct mapped cache example

Let be a byte-addressable main memory of size N=256, with words of 4 bytes and blocks of 1 word

Cache size is S=8

Current program requests the next memory references:

88, 104, 88, 72, 104

Reference	Binary
88	0101 1000

Block address:	22 (010110)
Cache line:	6 (110)
Tag:	010
Hit or miss:	Miss

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Direct mapped cache example

Let be a byte-addressable main memory of size N=256, with words of 4 bytes and blocks of 1 word

Cache size is S=8

Current program requests the next memory references:

88, 104, 88, 72, 104

Number of misses: 1

Number of hits: 0

Reference	Binary
88	0101 1100

Block address:	22 (010110)
Cache line:	6 (110)
Tag:	010
Hit or miss:	Miss

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	**Y**	**010**	**Mem[0101 1100]**
111	N		

Direct mapped cache example

Let be a byte-addressable main memory of size N=256, with words of 4 bytes and blocks of 1 word

Cache size is S=8

Current program requests the next memory references:

88, 104, 88, 72, 104

Number of misses: 1

Number of hits: 0

Reference	Binary
104	0110 1000

Block address:	26 (011010)
Cache line:	2 (010)
Tag:	011
Hit or miss:	Miss

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	010	Mem[0101 1100]
111	N		

Direct mapped cache example

Let be a byte-addressable main memory of size N=256, with words of 4 bytes and blocks of 1 word

Cache size is S=8

Current program requests the next memory references:

88, 104, 88, 72, 104

Number of misses: 2

Number of hits: 0

Reference	Binary
104	0110 1000

Block address:	26 (011010)
Cache line:	2 (010)
Tag:	011
Hit or miss:	Miss

Index	V	Tag	Data
000	N		
001	N		
010	**Y**	**011**	**Mem[0110 1000]**
011	N		
100	N		
101	N		
110	Y	010	Mem[0101 1100]
111	N		

Direct mapped cache example

Let be a byte-addressable main memory of size N=256, with words of 4 bytes and blocks of 1 word

Cache size is S=8

Current program requests the next memory references:

88, 104, 88, 72, 104

Number of misses: 2

Number of hits: 0

Reference	Binary
88	0101 1100

Block address:	22 (010110)
Cache line:	6 (110)
Tag:	010
Hit or miss:	**Hit**

Index	V	Tag	Data
000	N		
001	N		
010	Y	011	Mem[0110 1000]
011	N		
100	N		
101	N		
110	Y	010	Mem[0101 1100]
111	N		

Direct mapped cache example

Let be a byte-addressable main memory of size N=256, with words of 4 bytes and blocks of 1 word

Cache size is S=8

Current program requests the next memory references:

88, 104, 88, 72, 104

Number of misses: 2

Number of hits: 1

Reference	Binary
88	0101 1100

Block address:	22 (010110)
Cache line:	6 (110)
Tag:	010
Hit or miss:	**Hit**

Index	V	Tag	Data
000	N		
001	N		
010	Y	011	Mem[0110 1000]
011	N		
100	N		
101	N		
110	Y	010	Mem[0101 1100]
111	N		

Direct mapped cache example

Let be a byte-addressable main memory of size N=256, with words of 4 bytes and blocks of 1 word

Cache size is S=8

Current program requests the next memory references:

88, 104, 88, 72, 104

Number of misses: 2

Number of hits: 1

Reference	Binary
72	0100 1000

Block address:	18 (010010)
Cache line:	2 (010)
Tag:	010
Hit or miss:	**Miss**

Index	V	Tag	Data
000	N		
001	N		
010	Y	011	Mem[0110 1000]
011	N		
100	N		
101	N		
110	Y	010	Mem[0101 1100]
111	N		

Direct mapped cache example

Let be a byte-addressable main memory of size N=256, with words of 4 bytes and blocks of 1 word

Cache size is S=8

Current program requests the next memory references:

88, 104, 88, 72, 104

Number of misses: 3

Number of hits: 1

Reference	Binary
72	0100 1000

Block address:	18 (010010)
Cache line:	2 (010)
Tag:	010
Hit or miss:	**Miss**

Index	V	Tag	Data
000	N		
001	N		
010	**Y**	**010**	**Mem[0100 1000]**
011	N		
100	N		
101	N		
110	Y	010	Mem[0101 1100]
111	N		

Direct mapped cache example

Let be a byte-addressable main memory of size N=256, with words of 4 bytes and blocks of 1 word

Cache size is S=8

Current program requests the next memory references:

88, 104, 88, 72, 104

Number of misses: 3

Number of hits: 1

Reference	Binary
104	0110 1000

Block address:	26 (011010)
Cache line:	2 (010)
Tag:	011
Hit or miss:	**Miss**

Index	V	Tag	Data
000	N		
001	N		
010	Y	010	Mem[0100 1000]
011	N		
100	N		
101	N		
110	Y	010	Mem[0101 1100]
111	N		

Direct mapped cache example

Let be a byte-addressable main memory of size N=256, with words of 4 bytes and blocks of 1 word

Cache size is S=8

Current program requests the next memory references:

88, 104, 88, 72, 104

Number of misses: 4

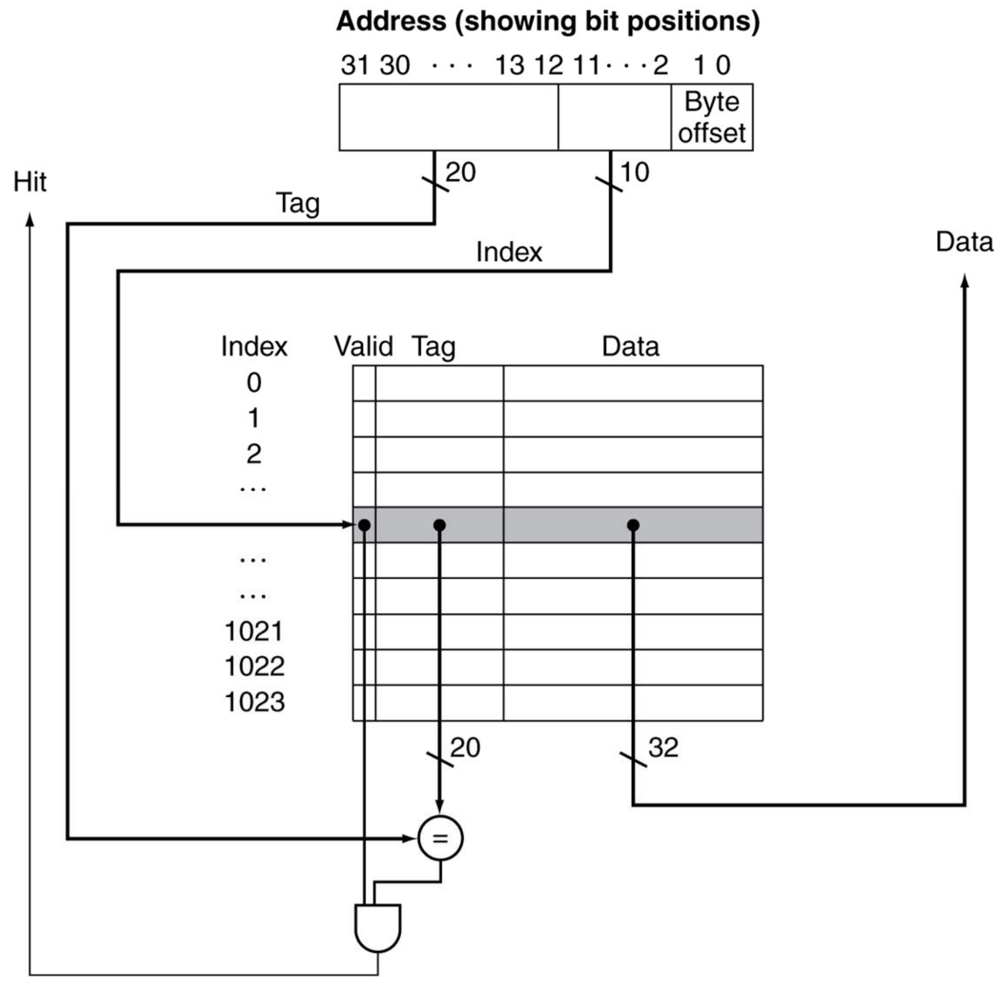
Number of hits: 1

Reference	Binary
104	0110 1000

Block address:	26 (011010)
Cache line:	2 (010)
Hit or miss:	**Miss**
Tag:	011

Index	V	Tag	Data
000	N		
001	N		
010	**Y**	**011**	**Mem[0110 1000]**
011	N		
100	N		
101	N		
110	Y	010	Mem[0101 1100]
111	N		

Direct mapped cache organization



Block Size Considerations

- Larger blocks should reduce miss rate
  - Due to spatial locality
- But in a fixed-sized cache
  - Larger blocks ⇒ fewer of them
    - More competition ⇒ increased miss rate
  - Larger blocks ⇒ pollution
- Larger miss penalty
  - Can override benefit of reduced miss rate
  - Early restart and critical-word-first can help

Large block size example

Let be a byte-addressable main memory of size N=256, with words of 4 bytes and blocks of 4 words

Cache size is S=2

Current program requests the next memory references:

88, 104, 88, 72, 104

Size of

Memory address:	8
Byte offset:	2
Word offset:	2
Block offset:	4
Block address:	4

Size of

Cache index:	1
Tag:	3

Index	V	Tag	Data
0	N		
1	N		

Reference	Binary	Block address	Cache index	Tag	Hit/miss
88	0101 1000	5 (0101)	1	010	miss

Large block size example

Let be a byte-addressable main memory of size N=256, with words of 4 bytes and blocks of 4 words

Cache size is S=2

Current program requests the next memory references:

88, 104, 88, 72, 104

Size of

- Memory address: 8
- Byte offset: 2
- Word offset: 2
- Block offset: 4
- Block address: 4

Size of

- Cache index: 1
- Tag: 3

Index	V	Tag	Data			
0	N					
1	**Y**	**010**	**M[92]**	**M[88]**	**M[84]**	**M[80]**

Reference	Binary	Block address	Cache index	Tag	Hit/miss
88	0101 1000	5 (0101)	1	010	miss
104	0110 1000	6 (0110)	0	011	miss

Large block size example

Let be a byte-addressable main memory of size N=256, with words of 4 bytes and blocks of 4 words

Cache size is S=2

Current program requests the next memory references:

88, 104, 88, 72, 104

Size of

Memory address:	8
Byte offset:	2
Word offset:	2
Block offset:	4
Block address:	4

Size of

Cache index:	1
Tag:	3

Index	V	Tag	Data			
0	**Y**	**011**	**M[108]**	**M[104]**	**M[100]**	**M[96]**
1	Y	010	M[92]	M[88]	M[84]	M[80]

Reference	Binary	Block address	Cache index	Tag	Hit/miss
88	0101 1000	5 (0101)	1	010	miss
104	0110 1000	6 (0110)	0	011	miss

Large block size example

Let be a byte-addressable main memory of size N=256, with words of 4 bytes and blocks of 4 words

Cache size is S=2

Current program requests the next memory references:

88, 104, 88, 72, 104

Size of
- Memory address: 8
- Byte offset: 2
- Word offset: 2
- Block offset: 4
- Block address: 4

Size of
- Cache index: 1
- Tag: 3

Index	V	Tag	Data			
0	Y	011	M[108]	M[104]	M[100]	M[96]
1	Y	010	M[92]	M[88]	M[84]	M[80]

Reference	Binary	Block address	Cache index	Tag	Hit/miss
88	0101 1000	5 (0101)	1	010	miss
104	0110 1000	6 (0110)	0	011	miss
88	0101 1000	5 (0101)	1	010	hit
72	0100 1000	4 (0100)	0	010	miss

Large block size example

Let be a byte-addressable main memory of size N=256, with words of 4 bytes and blocks of 4 words

Cache size is S=2

Current program requests the next memory references:

88, 104, 88, 72, 104

Size of

Memory address:	8
Byte offset:	2
Word offset:	2
Block offset:	4
Block address:	4

Size of

Cache index:	1
Tag:	3

Index	V	Tag	Data			
0	**Y**	**010**	**M[76]**	**M[72]**	**M[68]**	**M[64]**
1	Y	010	M[92]	M[88]	M[84]	M[80]

Reference	Binary	Block address	Cache index	Tag	Hit/miss
88	0101 1000	5 (0101)	1	010	miss
104	0110 1000	6 (0110)	0	011	miss
88	0101 1000	5 (0101)	1	010	hit
72	0100 1000	4 (0100)	0	010	miss
104	0110 1000	6 (0110)	0	011	miss

Cache Misses

On cache hit, CPU proceeds normally

On cache miss

- Stall the CPU pipeline
- Fetch block from next level of hierarchy
- Instruction cache miss → Restart instruction fetch
- Data cache miss → Complete data access

Handling writes: Write-Through

- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write-through: also update memory
- But makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = 1 + 0.1×100 = 11
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full

Write-Back

- Alternative: On data-write hit, just update the block in cache
    - Keep track of whether each block is dirty
- When a dirty block is replaced
    - Write it back to memory
    - Can use a write buffer to allow replacing block to be read first

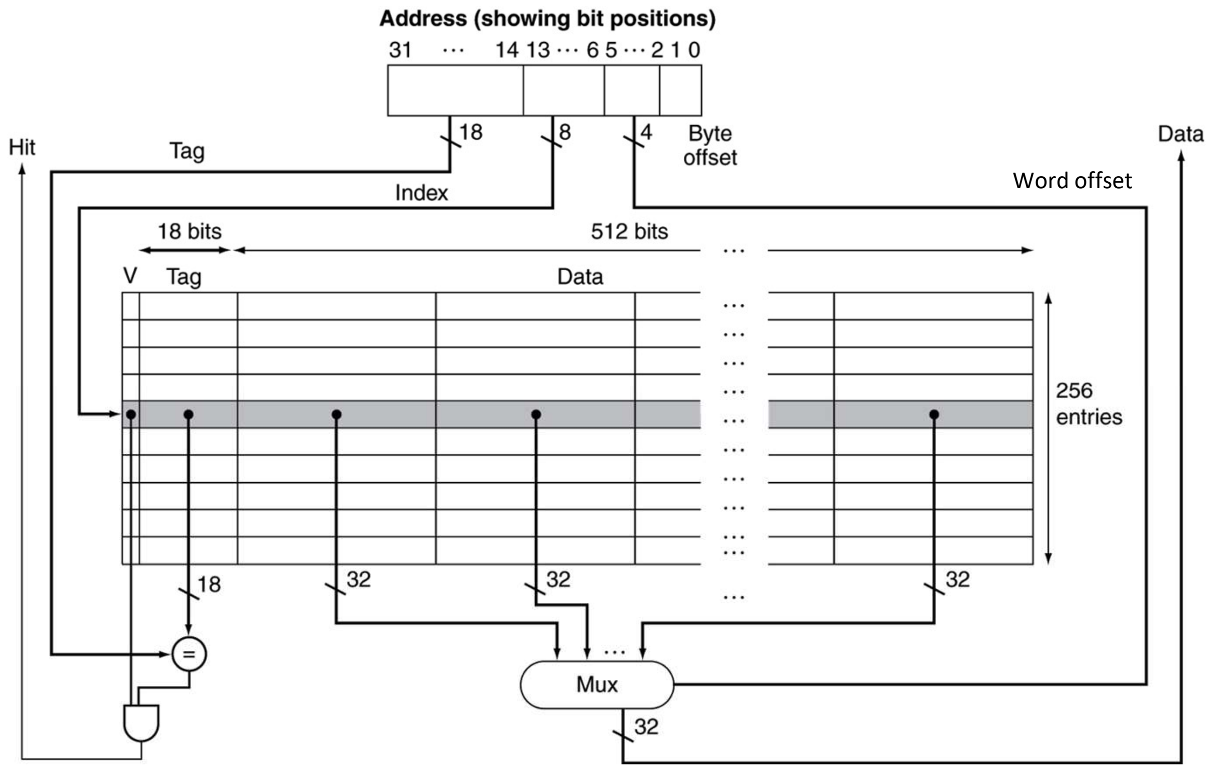
Write Allocation

- What should happen on a write miss?
- Alternatives for write-through
  - Allocate on miss: fetch the block
  - Write around: don’t fetch the block
    - Since programs often write a whole block before reading it (e.g., initialization)
- For write-back
  - Usually fetch the block

Example: Intrinsity FastMATH

- Embedded MIPS processor
  - 12-stage pipeline
  - Instruction and data access on each cycle
- Split cache: separate I-cache and D-cache
  - Each 16KB: 256 blocks × 16 words/block
  - D-cache: write-through or write-back
- SPEC2000 miss rates
  - I-cache: 0.4%
  - D-cache: 11.4%
  - Weighted average: 3.2%

Example: Intrinsity FastMATH



Associative Caches

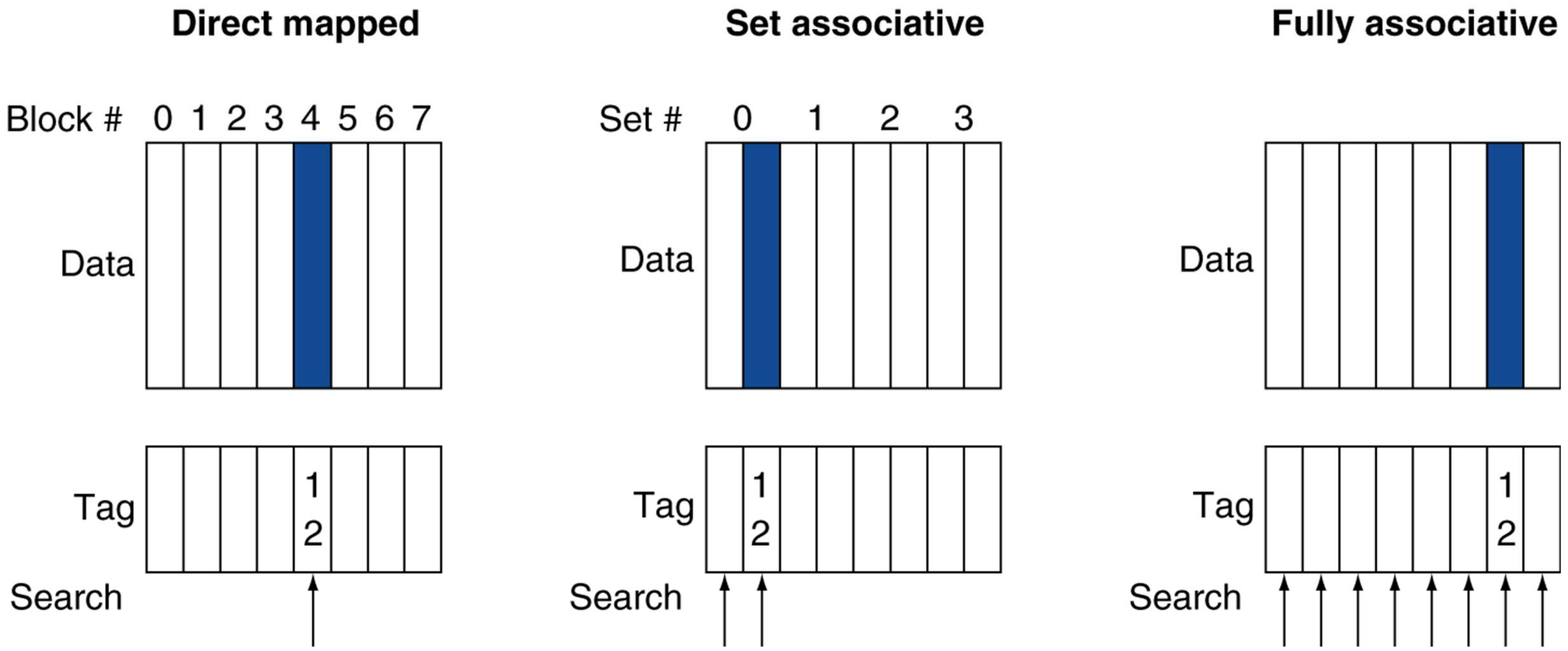
Fully associative

- A block can be mapped to any cache line
- Requires all entries to be searched at once
- Comparator per entry (expensive)

n-way set associative

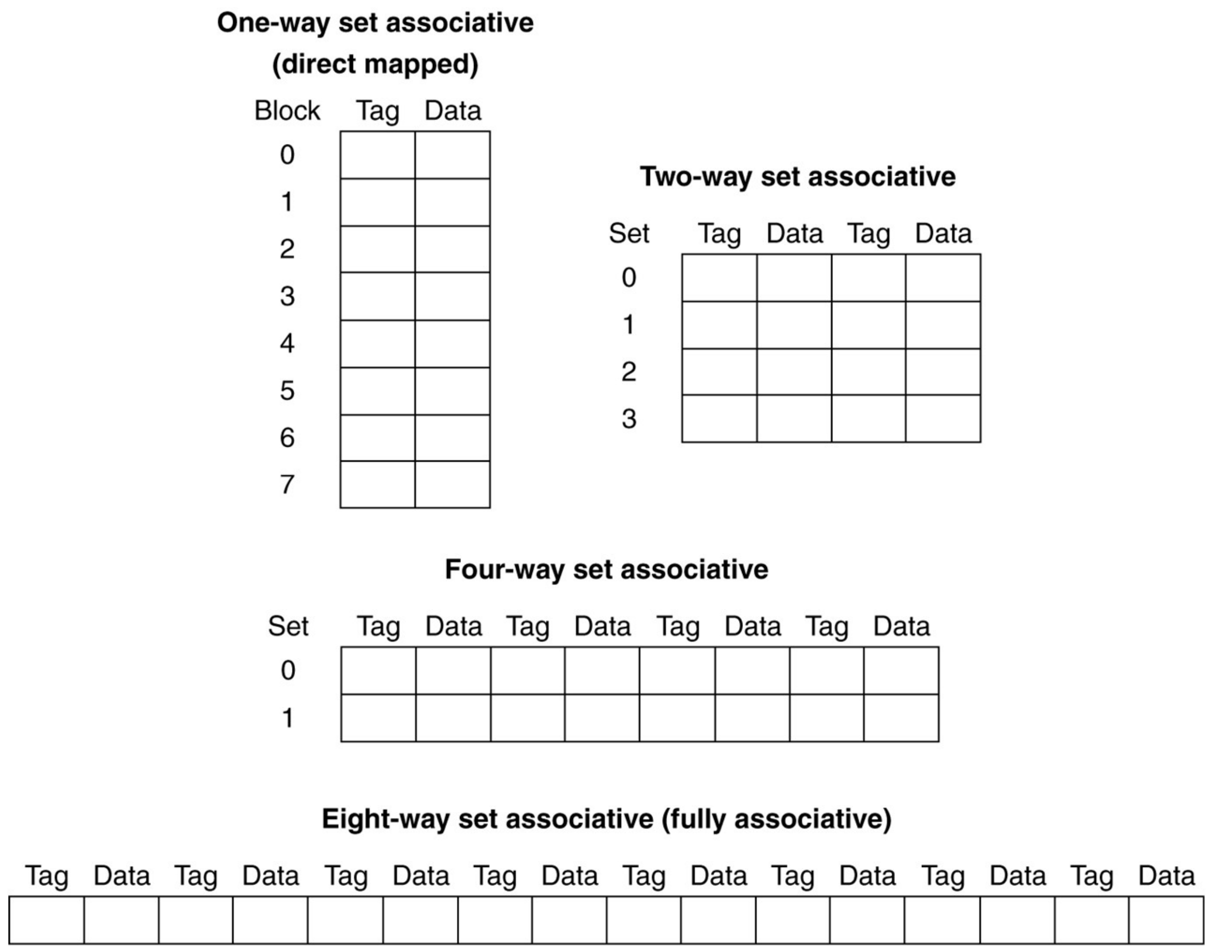
- Each set contains *n* entries (associativity)
- Block address determines which set
  - (Block address) modulo (#Sets in cache)
- Search all entries in each set at once
- *n* comparators (less expensive)

Associative Cache Example



Spectrum of Associativity

For a cache with 8 entries



One-way set associative (direct mapped)
Block
Tag
Data
0
1
2
3
4
5
6
7
Two-way set associative
Set
Tag Data Tag Data
0
1
2
3
Four-way set associative
Set
Tag Data Tag Data Tag Data Tag Data
0
1
Eight-way set associative (fully associative)
Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data

Associativity Example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Sequence of block addresses: 0, 8, 0, 6, 8
- Direct mapped

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	

Associativity Example

2-way set associative

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	**Mem[0]**			
8	0	miss	Mem[0]	**Mem[8]**		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	Mem[0]	**Mem[6]**		
8	0	miss	**Mem[8]**	Mem[6]		

Fully associative

Block address		Hit/miss	Cache content after access			
0		miss	**Mem[0]**			
8		miss	Mem[0]	**Mem[8]**		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	**Mem[6]**	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

How Much Associativity

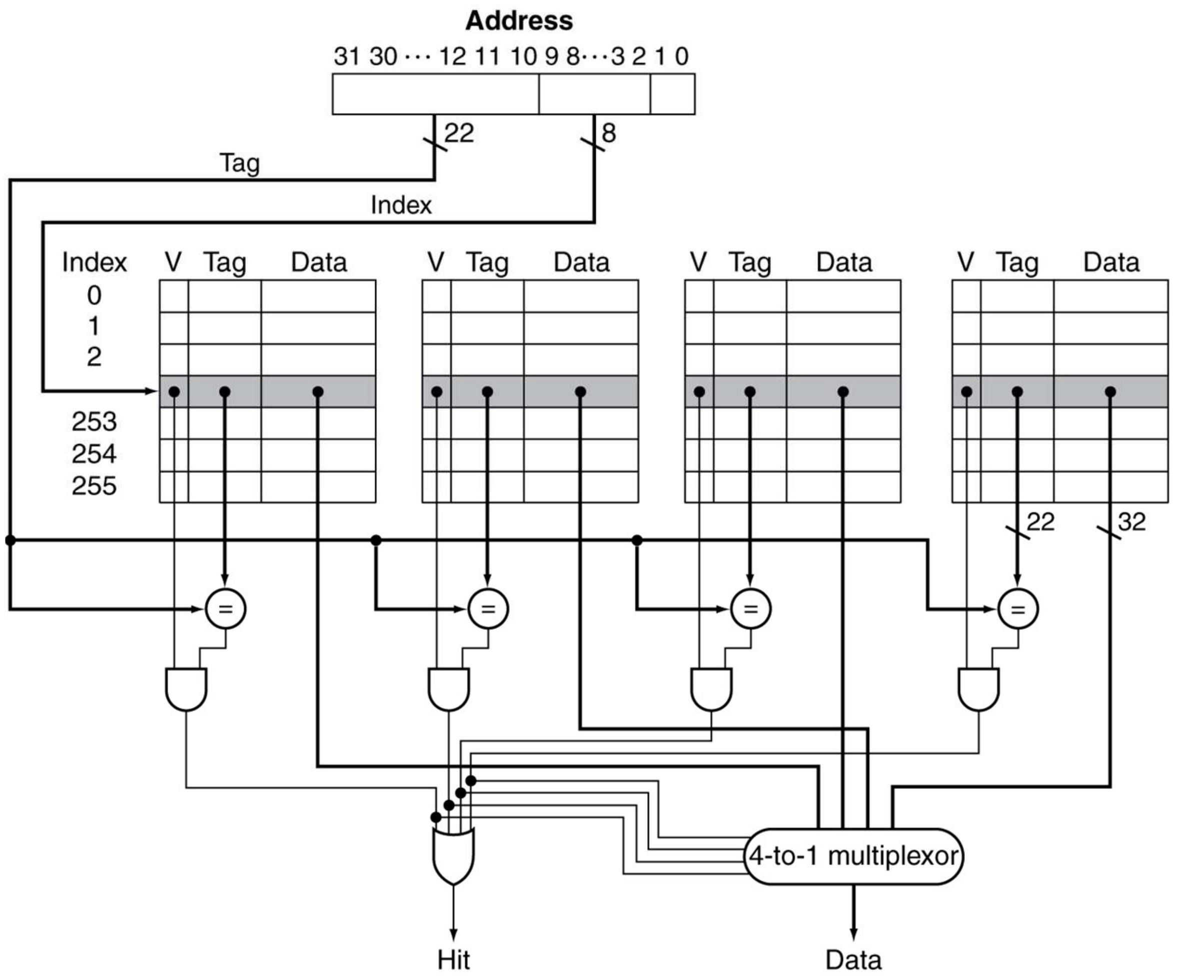
Increased associativity decreases miss rate

- But with diminishing returns

Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000

- 1-way: 10.3%
- 2-way: 8.6%
- 4-way: 8.3%
- 8-way: 8.1%

Set Associative Cache Organization



What to replace on a miss?

Direct mapped

- No choice

Set associative

- Prefer non-valid entry, if there is one
- Otherwise, choose among entries in the set

Replacement policy

First-in, First-out (FIFO)

- Choose the one that entered first

Least-recently used (LRU)

- Choose the one unused for the longest time
  - Simple for 2-way, manageable for 4-way, too hard beyond that

Random

- Gives approximately the same performance as LRU for high associativity

DRAM Performance Improvement

Row buffer

- Allows several words to be read and refreshed in parallel
- Improves latency

Synchronous DRAM

- Allows for consecutive accesses in bursts without needing to send each address
- Improves bandwidth

DRAM banking

- Allows simultaneous access to multiple DRAMs
- Improves bandwidth

Increasing Memory Bandwidth

Let be a cache with block size = 4 words and DRAM modules with

- 1 bus cycle for address transfer
- 15 bus cycles per DRAM access
- 1 bus cycle per data transfer

a) One word-wide:

$MissPenalty = 4 \times (1 + 15 + 1) = 68\ cycles$

$Bandwidth = {}^{16\ bytes}/_{68\ cycles} = 0.23\ {}^{bytes}/_{cycle}$

b) A wider bus increases bandwidth at a high cost

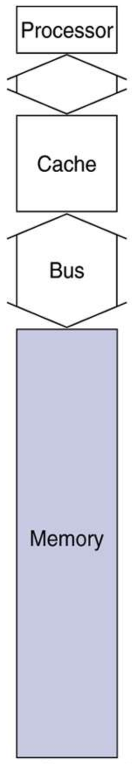
$MissPenalty = 1 + 15 + 1 = 17\ cycles$

$Bandwidth = {}^{16\ bytes}/_{17\ cycles} = 0.94\ {}^{bytes}/_{cycle}$

c) Interleaving increases bandwidth at a lower cost

$MissPenalty = 1 + 15 + 4 \times 1 = 20\ cycles$

$Bandwidth = {}^{16\ bytes}/_{20\ cycles} = 0.8\ {}^{bytes}/_{cycle}$



a. One-word-wide memory organization

Multilevel Caches

Primary cache (L1) attached to CPU

- Small, but fast
- Split caches, one for instructions and another for data

Level-2 cache services misses from primary cache

- Larger, slower, but still faster than main memory
- Main memory services L2 cache misses

High-end systems include L3 cache

- Shared between cores
- Sometimes configured as victim cache

Multilevel Caches Performance

$$AMAT = HitTime_{L1} + MissRate_{L1} \times MissPenalty_{L1}$$

- $MissPenalty_{L1} = HitTime_{L2} + MissRate_{L2} \times MissPenalty_{L2}$

$$\begin{aligned} AMAT = \quad & HitTime_{L1} + \\ & MissRate_{L1} \times HitTime_{L2} + \\ & MissRate_{L1} \times MissRate_{L2} \times MissPenalty_{L2} \end{aligned}$$

Local miss rates: $MissRate_{L1}$ for L1, $MissRate_{L2}$ for L2

Global miss rate: $MissRate_{L1}$ for L1, $MissRate_{L1} \times MissRate_{L2}$ for L2

Multilevel Caches Performance

$$CPI_{effective} = CPI_{base} + \frac{L2\text{−}stall\ cycles}{IC} + \frac{Memory\text{−}stall\ cycles}{IC}$$

$$\begin{aligned}\frac{L2\text{−}stall\ cycles}{IC} &= \frac{\#misses_{L1} \times MissPenalty_{L1}}{IC} \\ &= \frac{\#references_{L1}}{IC} \times \frac{\#misses_{L1}}{\#references_{L1}} \times MissPenalty_{L1}\end{aligned}$$

$$\begin{aligned}\frac{Memory\text{−}stall\ cycles}{IC} &= \frac{\#misses_{L2} \times MissPenalty_{L2}}{IC} \\ &= \frac{\#references_{L2}}{IC} \times \frac{\#misses_{L2}}{\#references_{L2}} \times MissPenalty_{L2}\end{aligned}$$

Multilevel Cache Example

Given

- CPU base CPI = 1
- Clock rate = 4GHz (clock cycle=0.25 ns)
- References per instruction: ${}^{\#\text{references}}/_{\text{IC}} = 1.6$
- Main memory access time = 100ns
- $MissPenalty = {}^{100ns}/_{0.25ns} = 400\ cycles$

$$\boldsymbol{AMAT = 100ns}$$
$$\boldsymbol{CPI_{effective} = 1 + 1.6 \times 400 = 641}$$

With just a primary cache

- HitTime = 1 cycle
- MissRate = 1.25%
- Miss rate per instruction in L1: ${}^{\#\text{misses}}/_{\text{IC}} = 1.6 \times 0.0125 = 0.02$

$$\boldsymbol{AMAT = 0.25 + 0.0125 \times 100 = 1.5\ ns}$$
$$\boldsymbol{CPI_{effective} = 1 + 0.02 \times 400 = 9}$$

Add a L2 cache

- Access time = 5ns
- MissRate = 40%
- $HitTime_{L2} = {}^{5ns}/_{0.25ns} = 20\ cycles$
- $GlobalMissRate = 1.25\% \times 40\% = 0.0125 \times 0.4 = 0.005 = 0.5\%$
- Miss rate per instruction in L2: ${}^{\#\text{misses}}/_{\text{IC}} = 1.6 \times 0.005 = 0.008$

$$\boldsymbol{AMAT = 0.25 + 0.0125 \times 5 + 0.005 \times 100 = 0.8125\ ns}$$
$$\boldsymbol{CPI_{effective} = 1 + 0.02 \times 20 + 0.008 \times 400 = 4.6}$$

Multilevel Cache Considerations

L1 cache

- Focus on minimal hit time

L2 cache

- Focus on low miss rate to avoid main memory access
- Hit time has less overall impact

The big picture

- L1 cache usually smaller than a single cache
- L1 block size smaller than L2 block size
- L2 uses higher associativity

The Three Cs Model

Compulsory misses (cold start)

- First access to a block
- Some of them are unavoidable even with an infinity size cache

Capacity misses

- Working set bigger than cache size (a replaced block is later accessed again)
- Unavoidable with a fully associative cache, only remedy is to increase cache size

Conflict misses (collision)

- Blocks compete for entries in the same set
- Avoidable by increasing associativity

Coherence misses

- Not in the original model, appear with multiprocessors

Cache Design Trade-offs

Design change	Effect on miss rate	Negative performance effect
Increase cache size	Decrease capacity misses	May increase access time
Increase associativity	Decrease conflict misses	May increase access time
Increase block size	Decrease compulsory misses	Increases miss penalty. For very large block size, may increase miss rate due to pollution.

Interactions with Advanced CPUs

Out-of-order CPUs can execute instructions during cache miss

- Pending store stays in load/store unit
- Dependent instructions wait in reservation stations
- Independent instructions continue

Effect of miss depends on program data flow

- Much harder to analyze
- Use system simulation

Virtual Memory

Virtual Memory

Use main memory as a “cache” for secondary (disk) storage

- Managed jointly by CPU hardware and the operating system (OS)

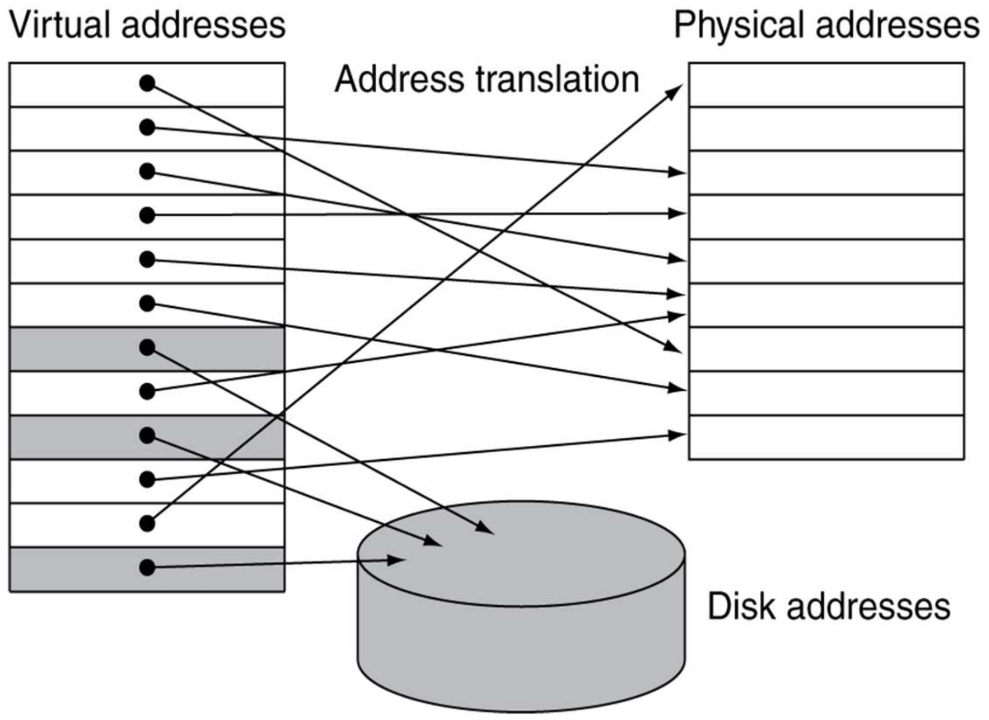
Programs share main memory

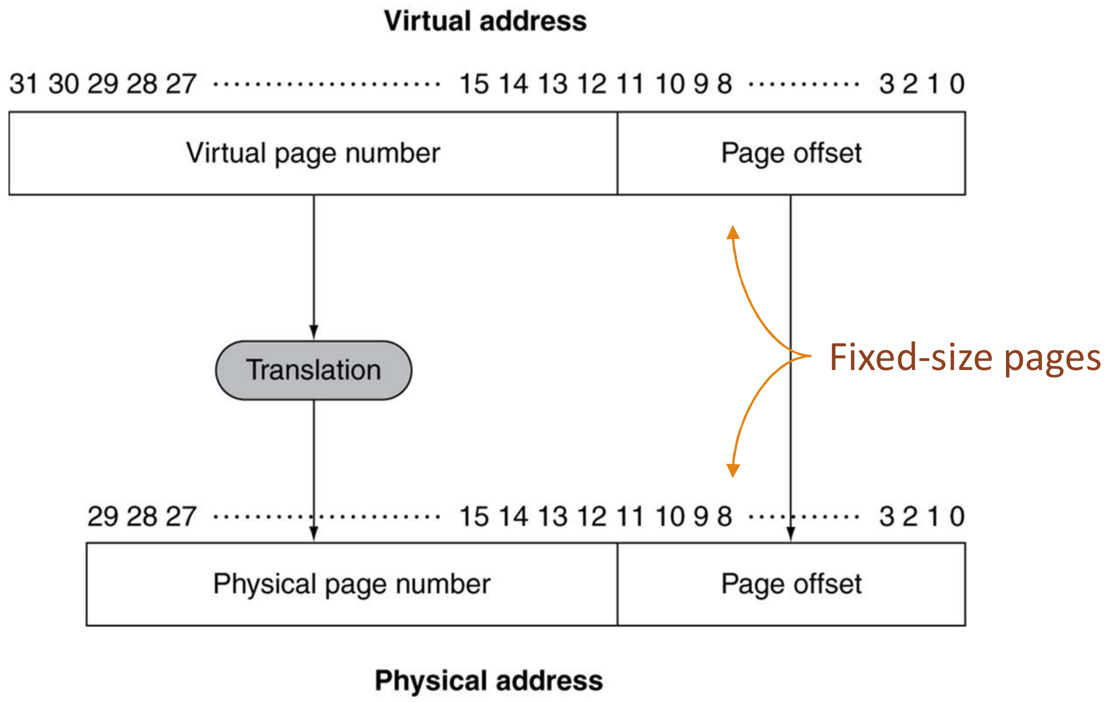
- Each gets a private virtual address space holding its frequently used code and data
- Protected from other programs

CPU and OS translate virtual addresses to physical addresses

- VM “block” is called a page
- VM translation “miss” is called a page fault

Address Translation





Translation using a Page Table

Stores placement information

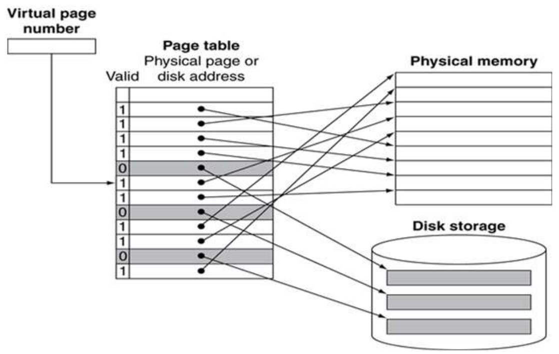
- Array of page table entries, indexed by virtual page number
- Page table register in CPU points to page table in physical memory

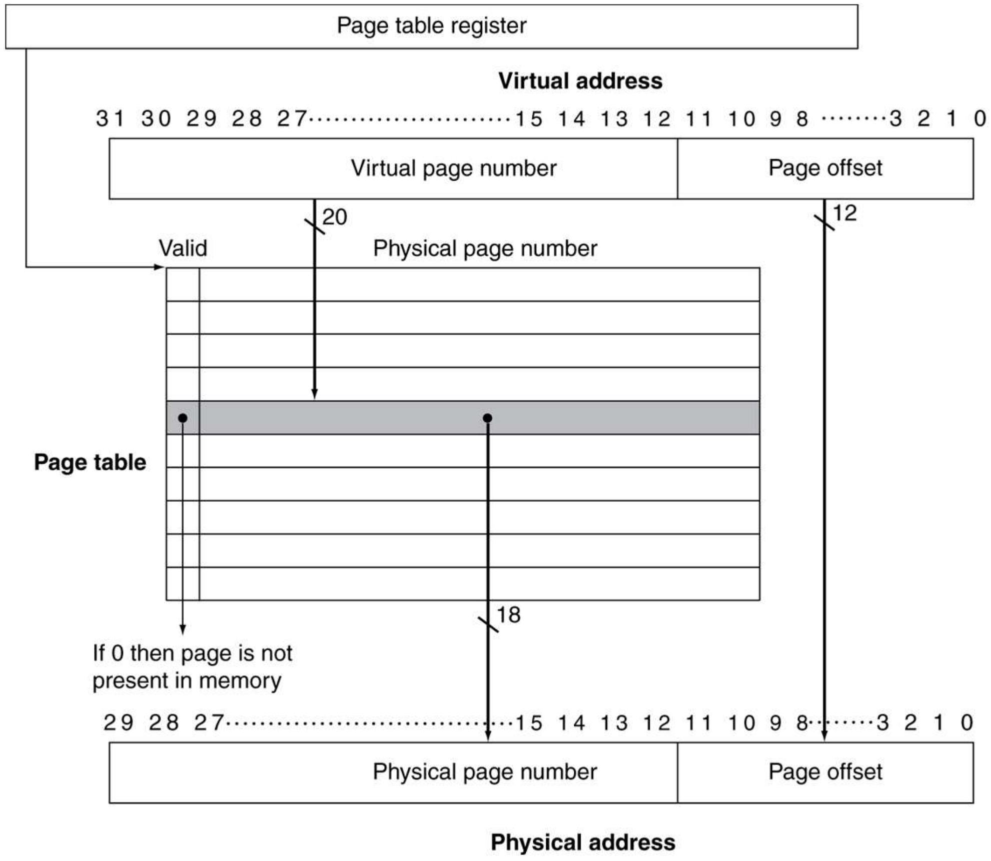
If page is present in memory

- PTE stores the physical page number
- Besides other status bits (referenced, dirty, …)

If page is not present

- PTE can refer to location in swap space on disk





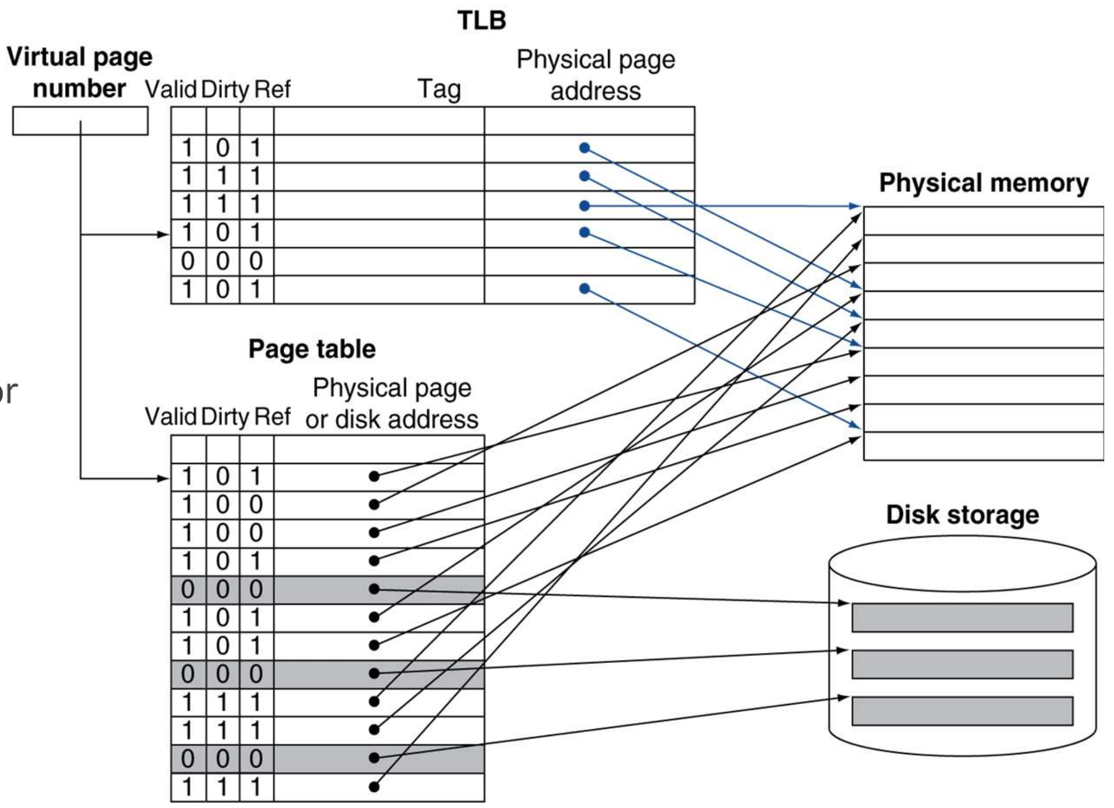
Fast Translation using a TLB

Address translation would appear to require extra memory references

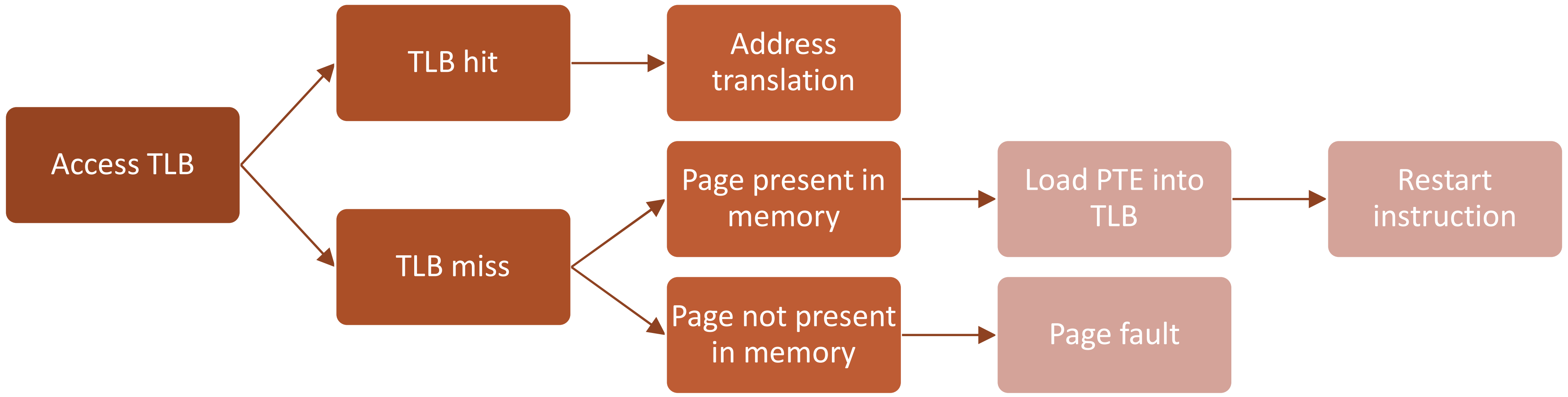
- One to access the PTE
- Then the actual memory access

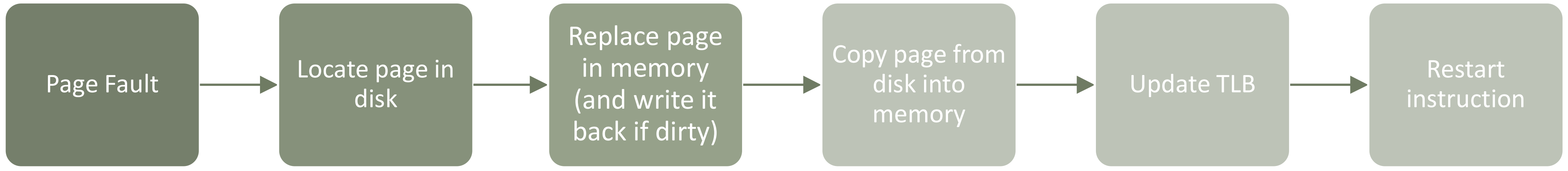
But access to page tables has good locality

- So, use a fast cache of PTEs within the CPU
- Called a Translation Look-aside Buffer (TLB)
- Typical: 16–512 PTEs, 0.5–1 cycle for hit, 10–100 cycles for miss, 0.01%–1% miss rate
- Misses could be handled by hardware or software



Translation steps




Page Fault
Locate page in disk
Replace page in memory (and write it back if dirty)
Copy page from disk into memory
Update TLB
Restart instruction

Page Fault Penalty

On page fault, the page must be fetched from disk

- Takes millions of clock cycles
- Handled by OS code

Try to minimize page fault rate

- Fully associative placement
- Smart replacement algorithms, prefer least-recently used (LRU) replacement
  - Reference bit (aka use bit) in PTE set to 1 on access to page
  - Periodically cleared to 0 by OS
  - A page with reference bit = 0 has not been used recently

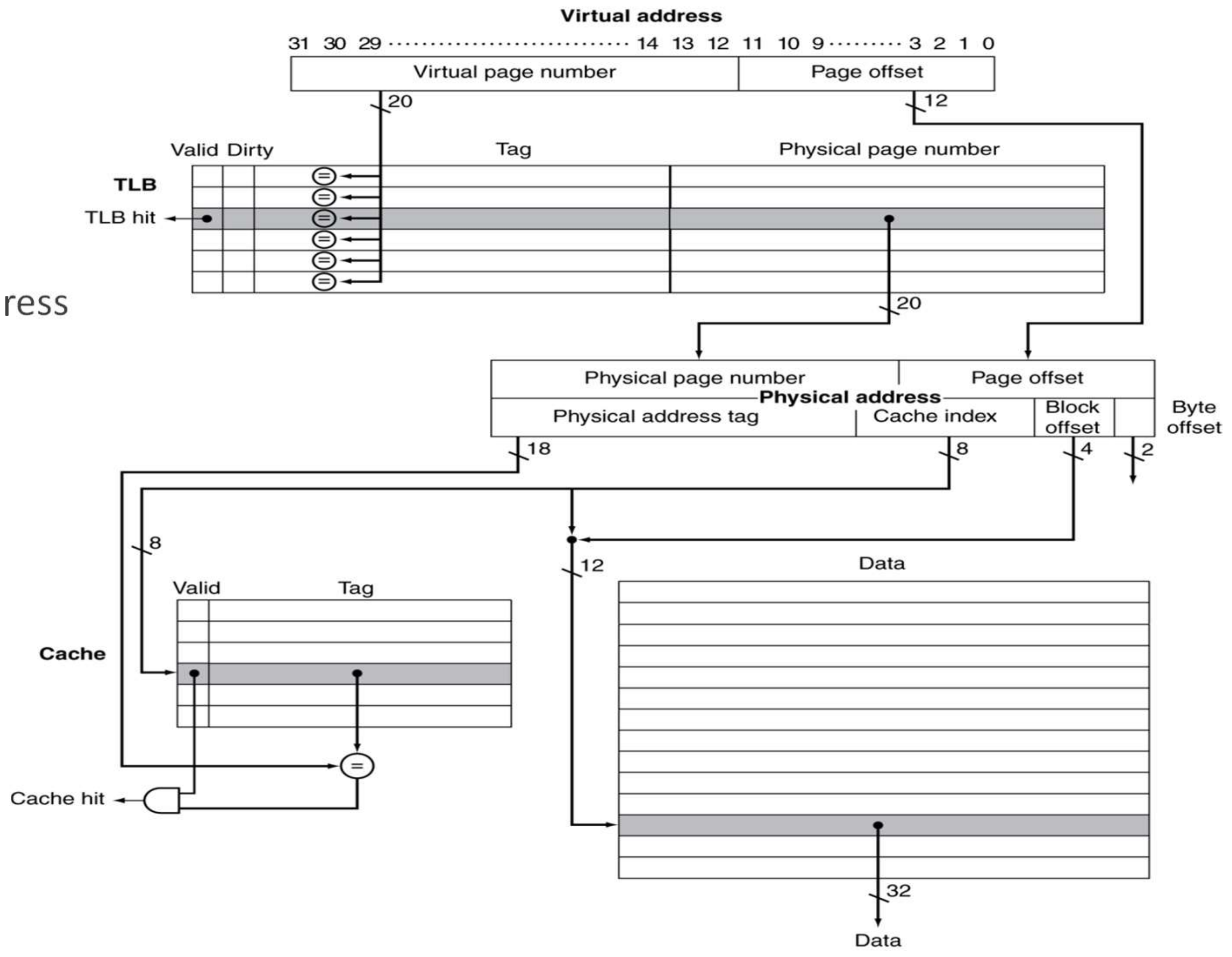
Disk writes take millions of cycles

- Block at once, not individual locations
- Write through is impractical, use write-back
- Dirty bit in PTE set when page is written

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



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)

Concluding remarks

Fast memories are small, large memories are slow

We really want fast AND large memories

Caching gives this illusion

Principle of locality

Programs use a small part of their memory space frequently

Memory hierarchy

L1 cache ↔ L2 cache ↔ … ↔ DRAM memory ↔ disk

Memory system design is critical for multiprocessors