Arquitectura de Computadores

**Capítulo 3.
Organización y Estructura de la Memoria: Cachés
y Memoria Virtual**

Escuela Politécnica Superior

Universidad Autónoma de Madrid

Profesores:
G130 y G131: Iván González Martínez
G136: Francisco J. Gómez Arribas

Memory Technology

- ## Static RAM (SRAM)
  - 0.5ns – 2.5ns, $2000 – $5000 per GB

- ## Dynamic RAM (DRAM)
  - 50ns – 70ns, $20 – $75 per GB

- ## Magnetic disk
  - 5ms – 20ms, $0.20 – $2 per GB

- ## Ideal memory
  - Access time of SRAM
 - Capacity and cost/GB of disk

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

Taking Advantage of Locality

- Memory hierarchy

- Store everything on disk

- Copy recently accessed (and nearby) items from disk to smaller DRAM memory

  - Main memory

- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory

  - Cache memory attached to CPU

Memory Hierarchy Levels

Processor

Data is transferred
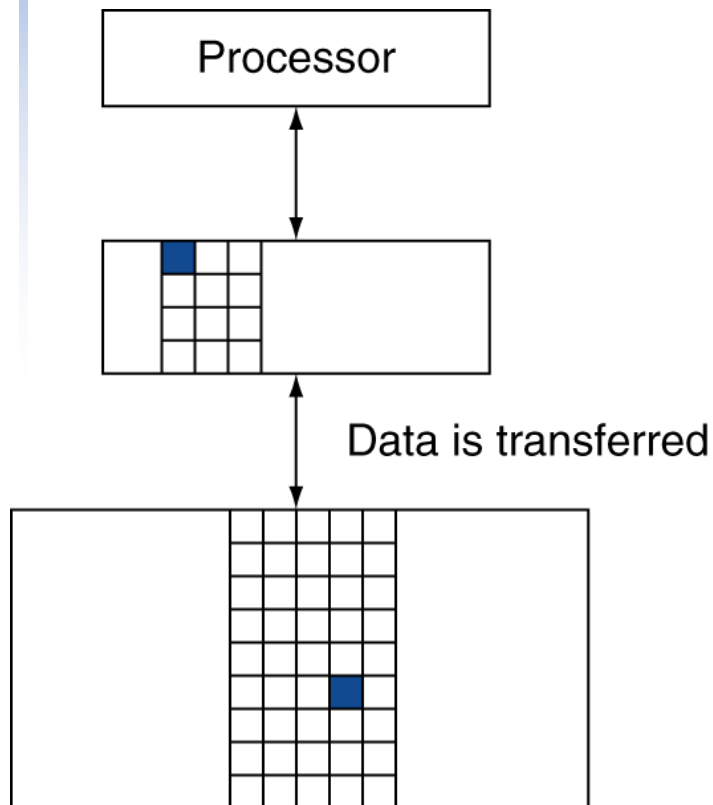
- Block (aka line): unit of copying
  - May be multiple words
- If accessed data is present in upper level
  - Hit: access satisfied by upper level
    - Hit ratio: hits/accesses
- If accessed data is absent
  - Miss: block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses = 1 – hit ratio
  - Then accessed data supplied from upper level

Cache Memory

- ## Cache memory

  - ### The level of the memory hierarchy closest to the CPU

- ## Given accesses X_1, …, X_{n-1}, X_n

X_4
X_1
X_{n-2}
X_{n-1}
X_2
X_3

a. Before the reference to X_n
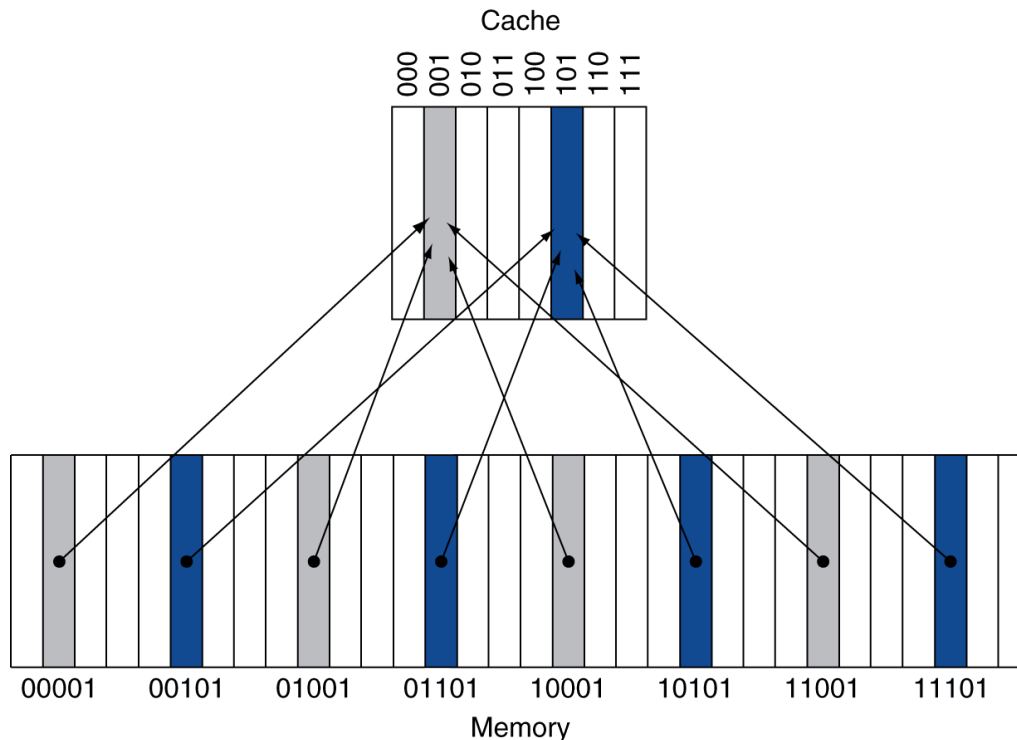
X_4
X_1
X_{n-2}
X_{n-1}
X_2
X_n
X_3

b. After the reference to X_n

- ## How do we know if the data is present?
- ## Where do we look?

Direct Mapped Cache

- Location determined by address
- Direct mapped: only one choice
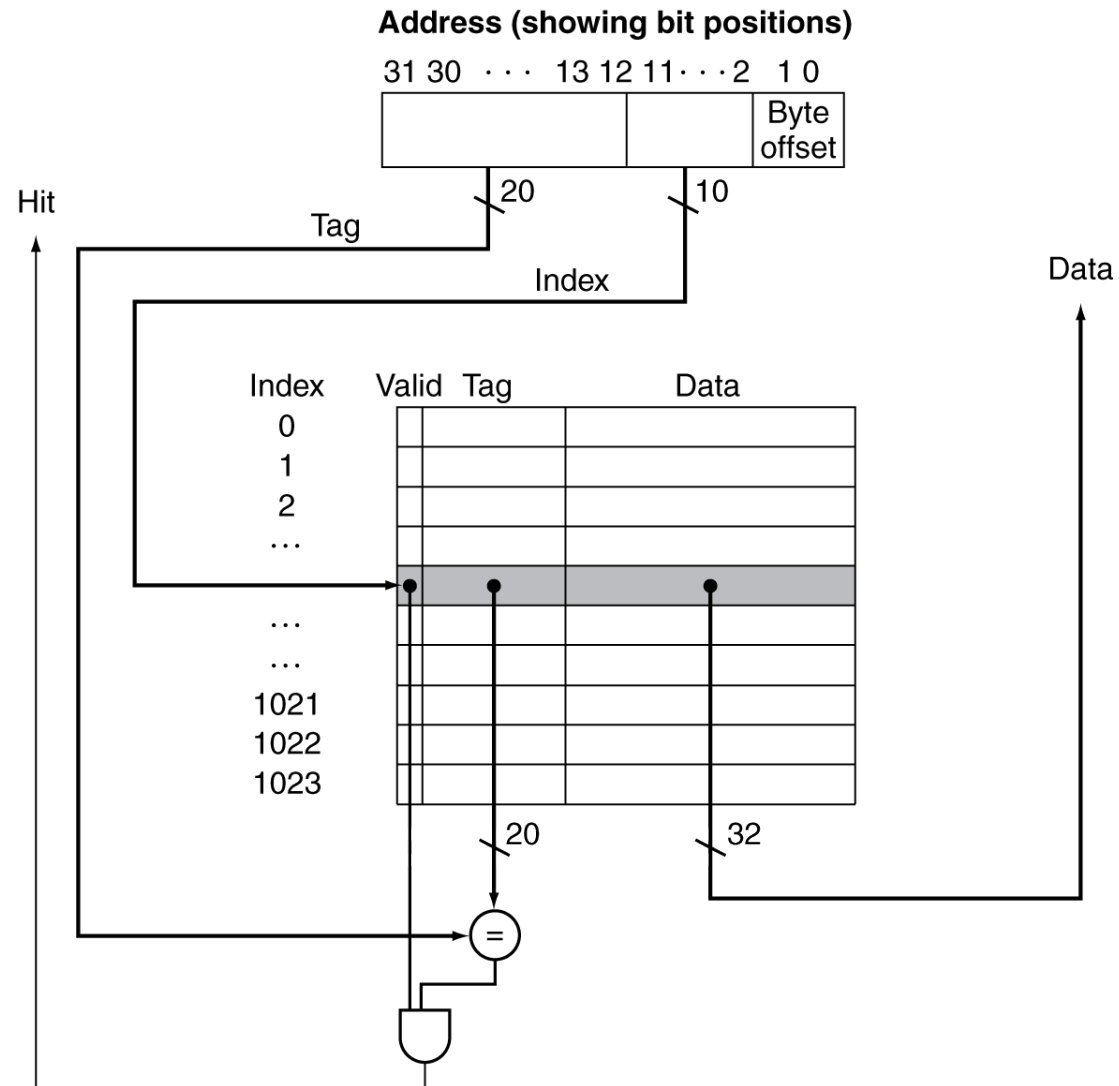   - (Block address) modulo (#Blocks in cache)

- #Blocks is a power of 2
- Use low-order address bits

Tags and Valid Bits

- How do we know which particular block is stored in a cache location?
  - Store block address as well as the data
  - Actually, only need the high-order bits
  - Called the tag
- What if there is no data in a location?
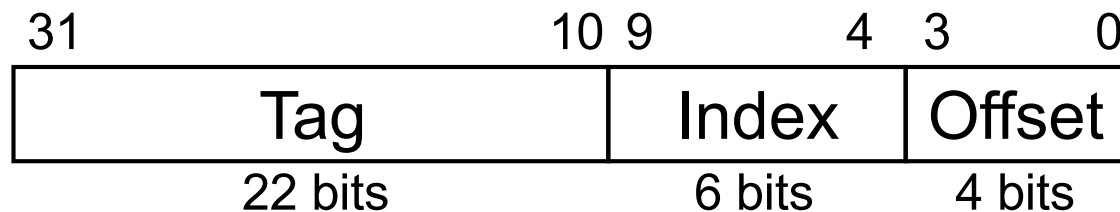 - Valid bit: 1 = present, 0 = not present
  - Initially 0

Address Subdivision

Example: Large Block Size

- 64 blocks, 16 bytes/block
  - To what block number does address 1200 map?
- Block address = $\lfloor 1200/16 \rfloor$ = 75
- Block number = 75 modulo 64 = 11

31 10	9 4	3 0
Tag	Index	Offset
22 bits	6 bits	4 bits

Example: Cache 4 kbytes and 16 bytes/block

20b   **8b**   **4b**

FA067	04	C

ADDRESS (32 b)
FA06704C

DIRECTORY   BLOCKS

00

C

04   FA067 | 2F | 3D | 00 | A0 | .... | FF | A0 | 25 | 7F |

2F3D0...0257F

H

CPU

FF

A0003D2F
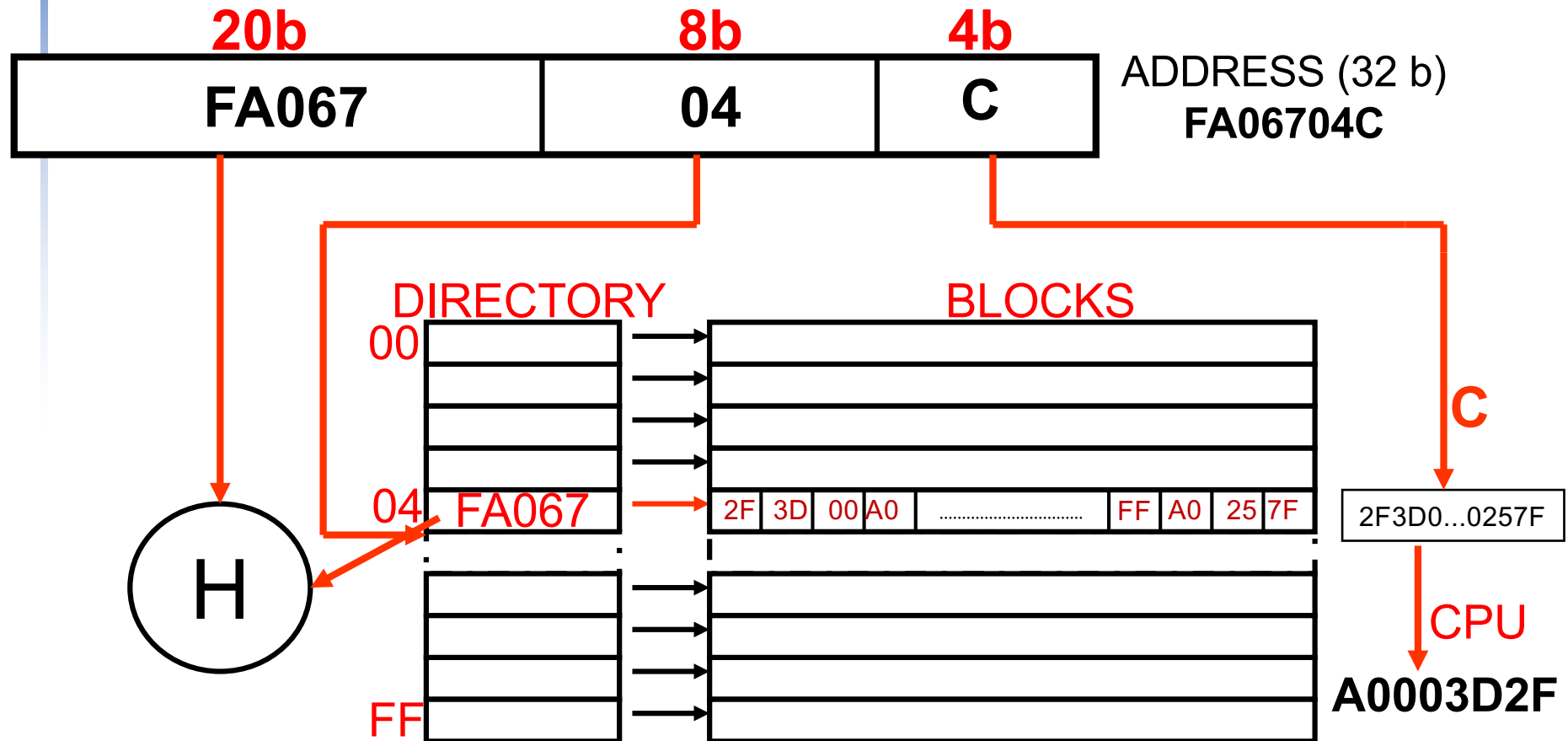
Associative Caches

- Fully associative
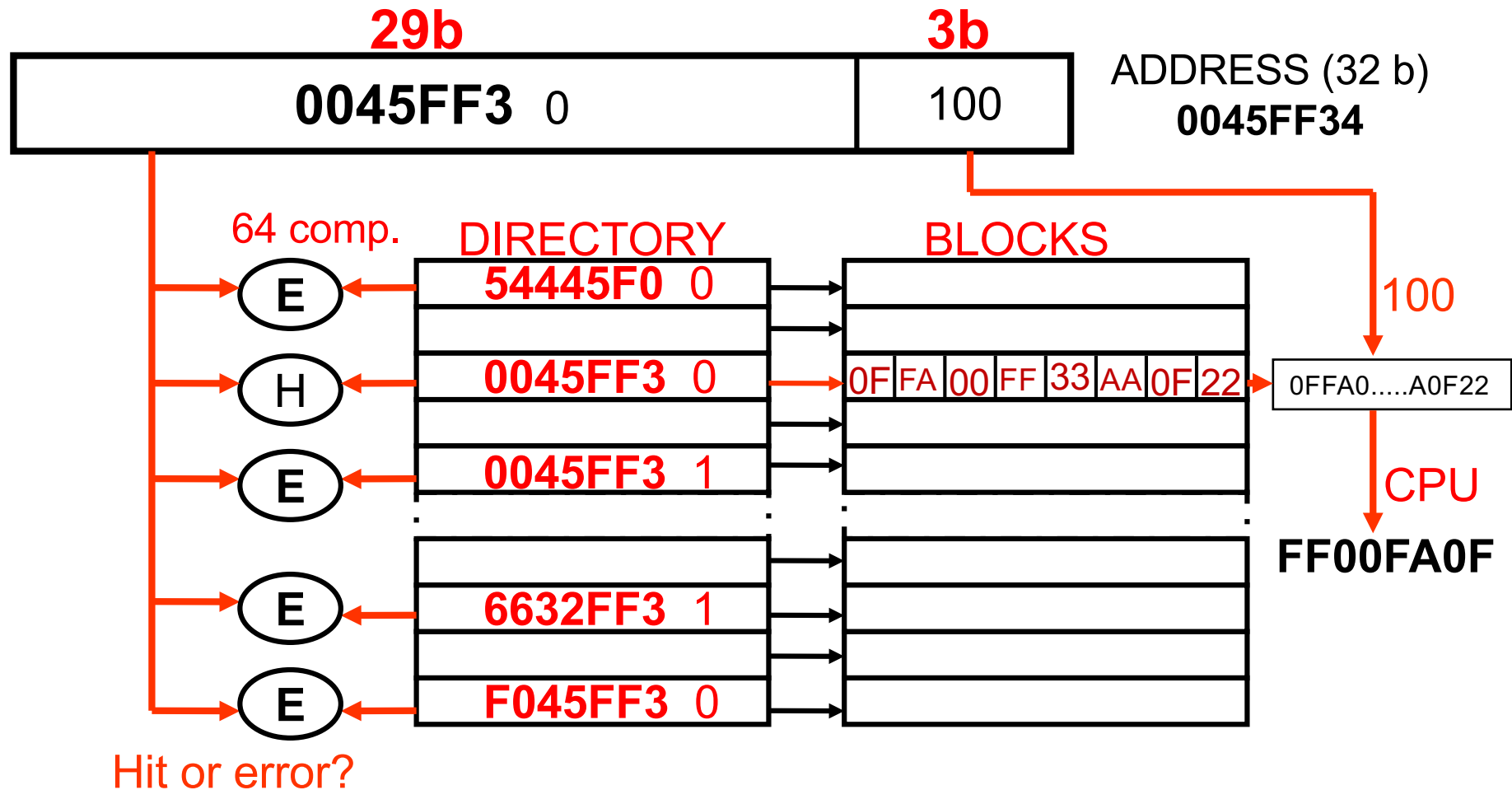 - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
  - Comparator per entry (expensive)
- *n*-way set associative
  - Each set contains *n* entries
  - Block number determines which set
    - (Block address) modulo (#Sets in cache)
  - Search all entries in a given set at once
  - *n* comparators (less expensive)

Fully associative (FA)

ADDRESS

| TAG BITS | BYTE IN BLOCK |

DIRECTORY   BLOCKS

=?   =?   =?   =?   =?

hit or error?

As many comparators as blocks

If hit, it indicates which is the block searched and gives the word searched (all of a part of the block) to CPU
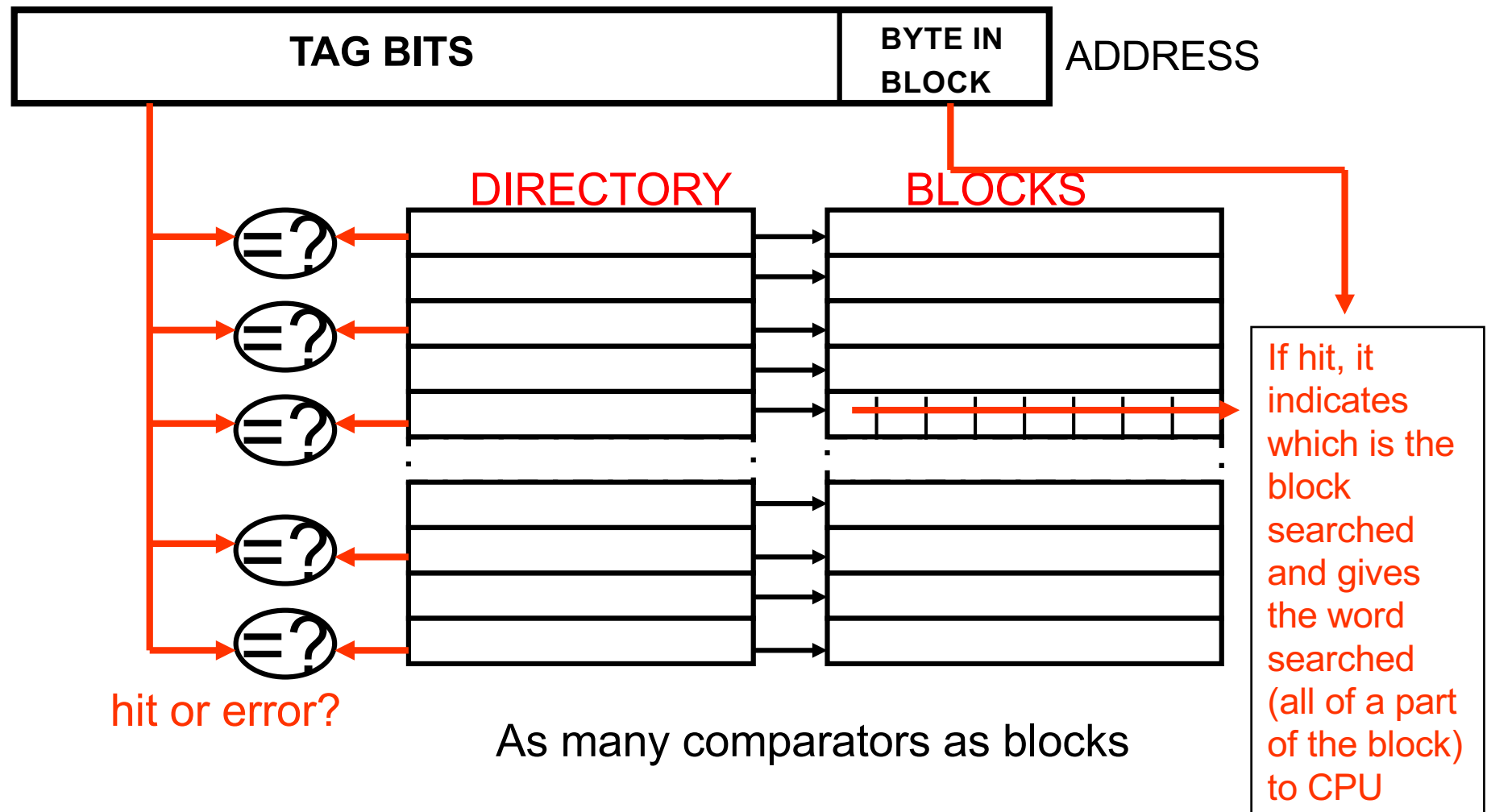
Example: Cache FA 512 bytes and 8 bytes/block

29b   **3b**

0045FF3 0	100

ADDRESS (32 b)
0045FF34

64 comp.

DIRECTORY   BLOCKS

E ← **54445F0 0**

H ← **0045FF3 0** → | 0F | FA | 00 | FF | 33 | AA | 0F | 22 | → | 0FFA0.....A0F22 |

100

E ← **0045FF3 1**

E ← **6632FF3 1**

E ← **F045FF3 0**

CPU

FF00FA0F

Hit or error?

14

N-way set associative (N-A)

TAG BITS	INDEX BITS	BYTE IN BLOCK

ADDRESS

If hit, take the word of the block with hit and give it to CPU

WAY-1
DIRECTORY   BLOCKS

BL-1

WAY-n
DIRECTORY   BLOCKS

BL-N

=?

Hit or error?

=?

As many comparators as ways
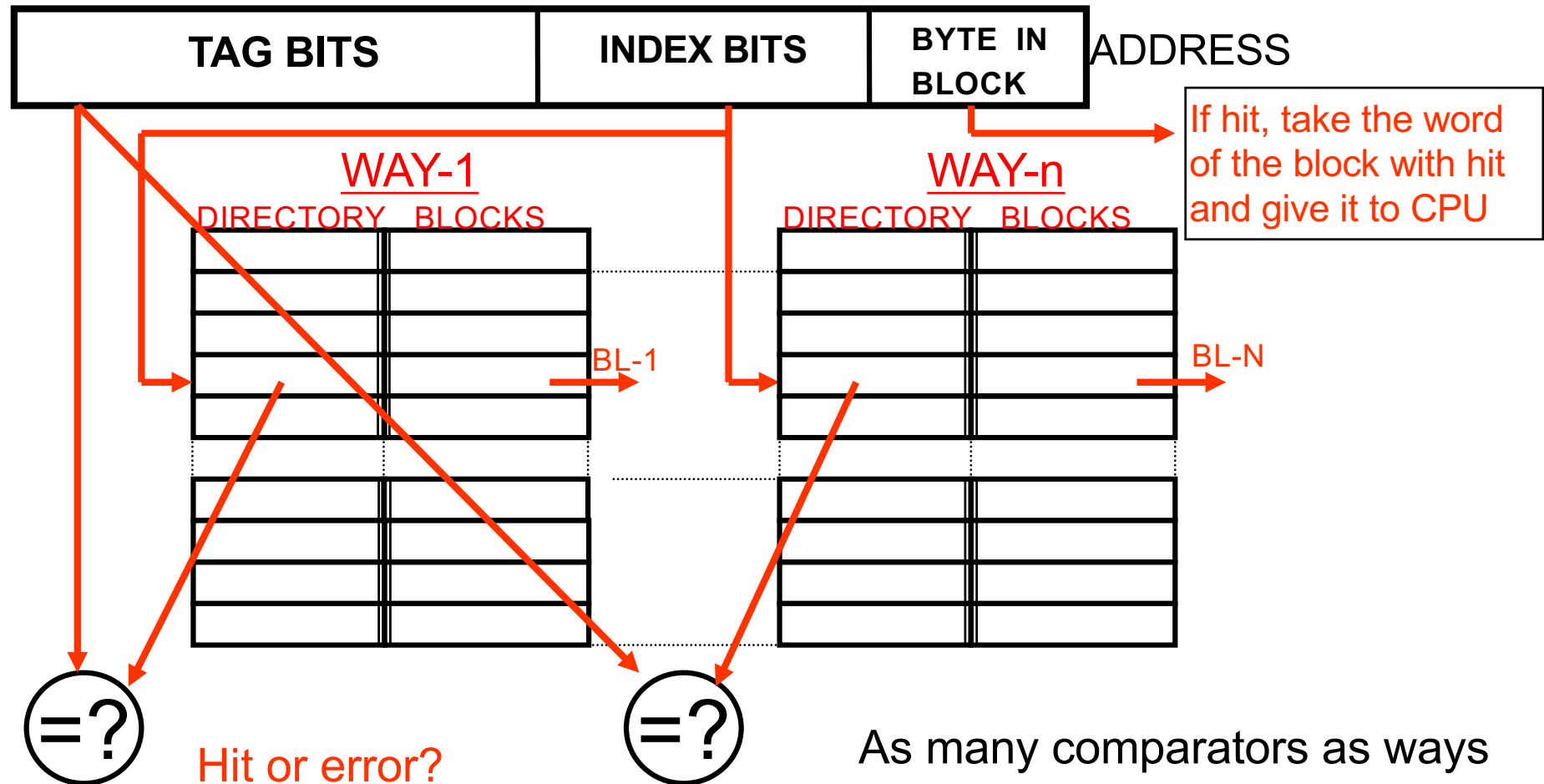
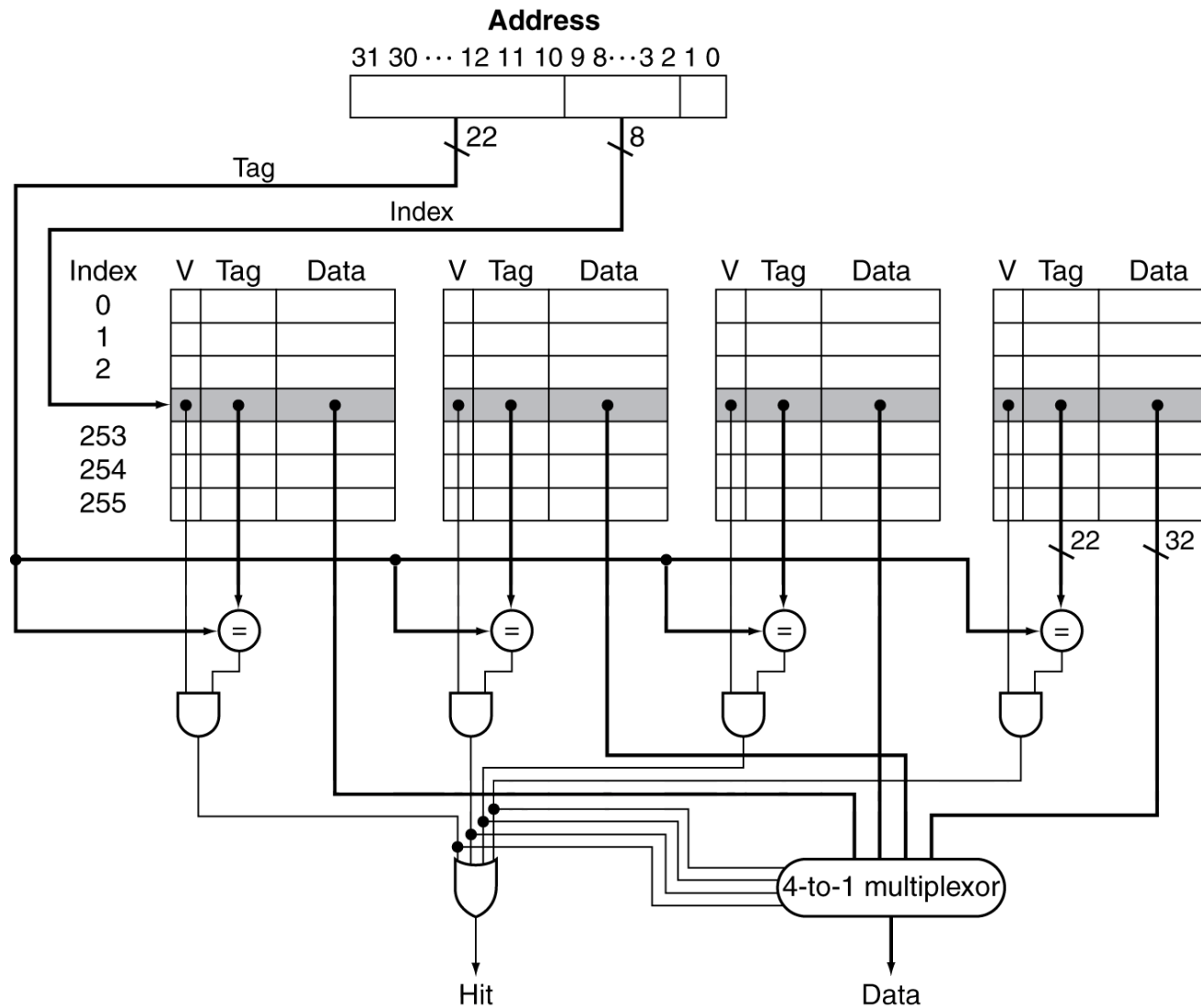# Example: Cache N-A4W 4kbytes and 64 bytes/block
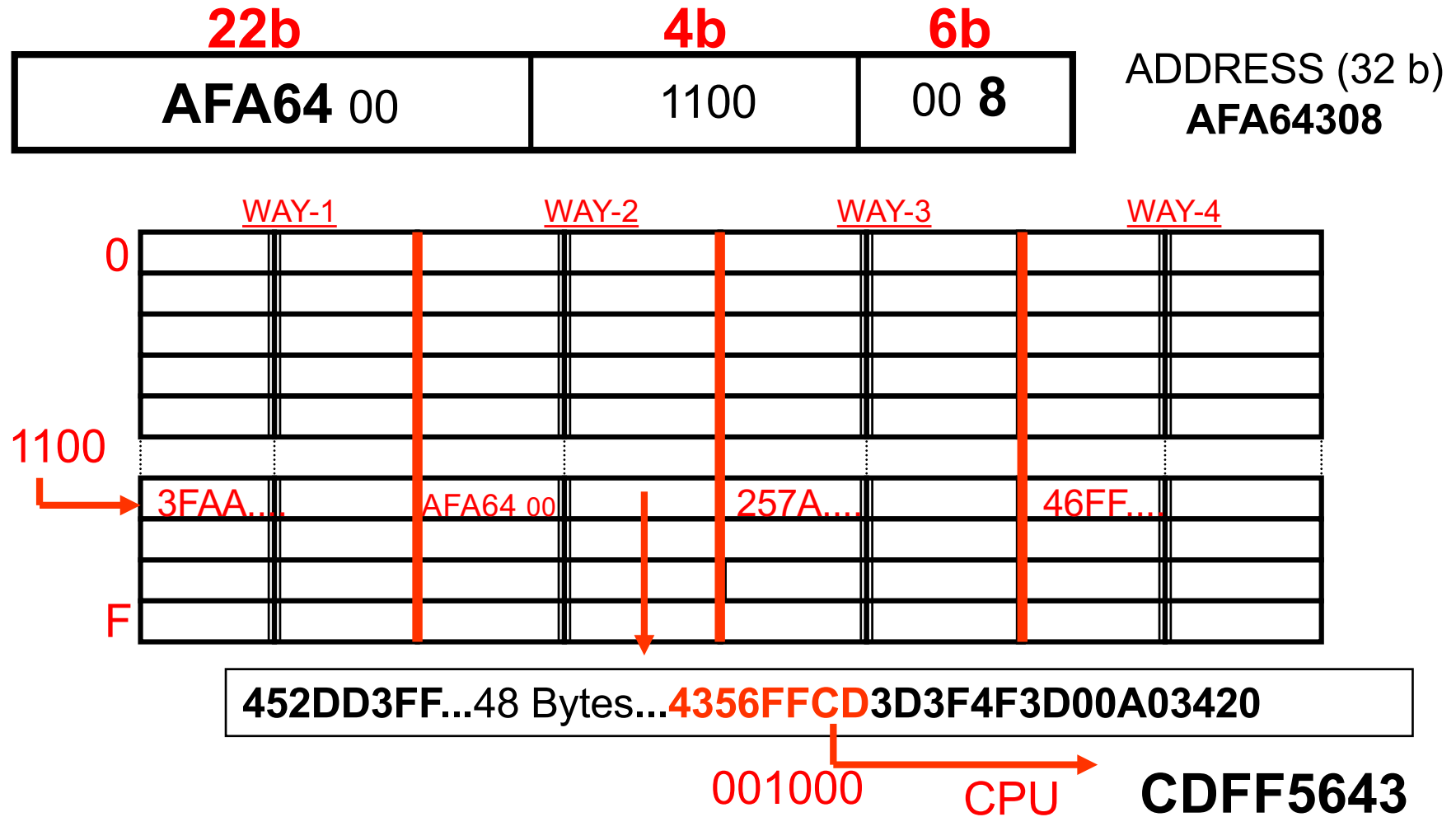
How Much Associativity

- 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

Block Size Considerations

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

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

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 $\times$ 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
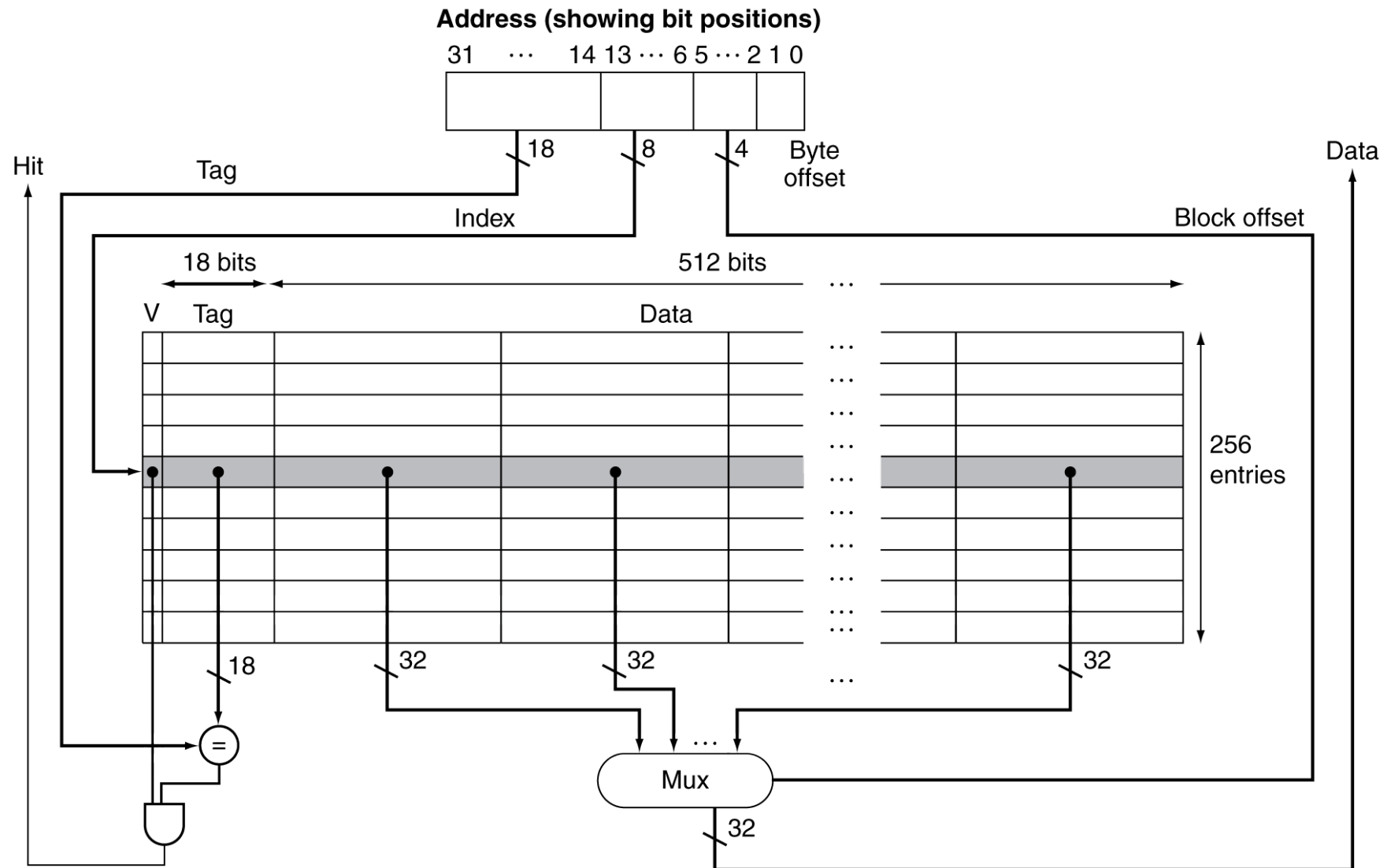
Main Memory Supporting Caches

- ## Use DRAMs for main memory
  - Fixed width (e.g., 1 word)
  - Connected by fixed-width clocked bus
    - Bus clock is typically slower than CPU clock

- ## Example cache block read
  - 1 bus cycle for address transfer
  - 15 bus cycles per DRAM access
  - 1 bus cycle per data transfer

- ## For 4-word block, 1-word-wide DRAM
  - Miss penalty = $1 + 4 \times 15 + 4 \times 1 = 65$ bus cycles
  - Bandwidth = 16 bytes / 65 cycles = 0.25 B/cycle

Increasing Memory Bandwidth
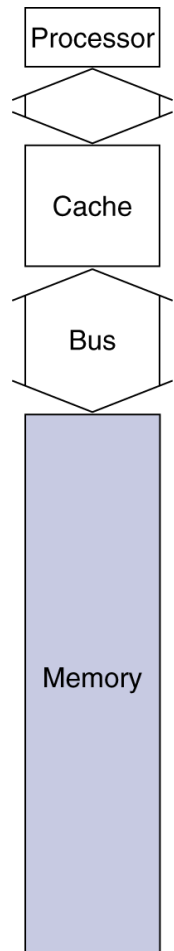
a. One-word-wide memory organization
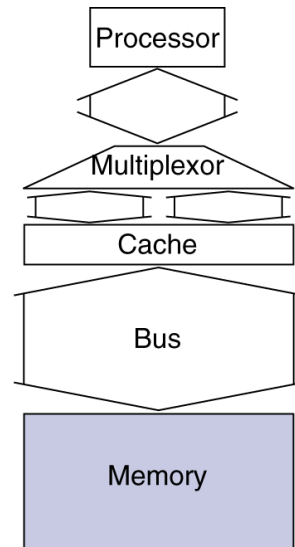
b. Wider memory organization
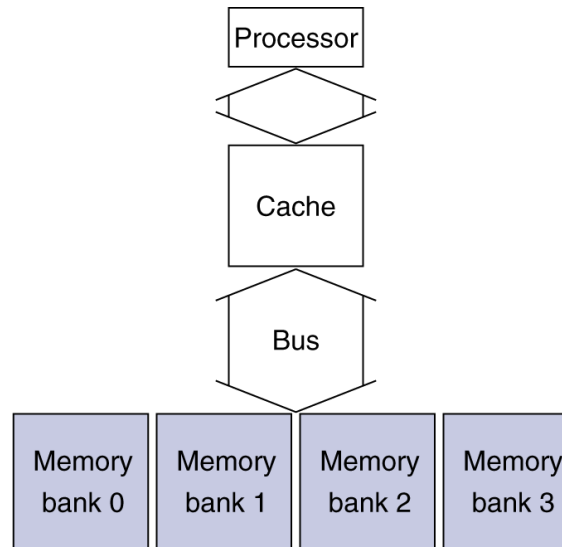
c. Interleaved memory organization

- **4-word wide memory**
  - Miss penalty = 1 + 15 + 1 = 17 bus cycles
  - Bandwidth = 16 bytes / 17 cycles = 0.94 B/cycle
- **4-bank interleaved memory**
  - Miss penalty = 1 + 15 + 4 × 1 = 20 bus cycles
  - Bandwidth = 16 bytes / 20 cycles = 0.8 B/cycle

Advanced DRAM Organization

- Bits in a DRAM are organized as a rectangular array
  - DRAM accesses an entire row
  - Burst mode: supply successive words from a row with reduced latency
- Double data rate (DDR) DRAM
  - Transfer on rising and falling clock edges
- Quad data rate (QDR) DRAM
  - Separate DDR inputs and outputs

DRAM Generations

Year	Capacity	$/GB
1980	64Kbit	$1500000
1983	256Kbit	$500000
1985	1Mbit	$200000
1989	4Mbit	$50000
1992	16Mbit	$15000
1996	64Mbit	$10000
1998	128Mbit	$4000
2000	256Mbit	$1000
2004	512Mbit	$250
2007	1Gbit	$50
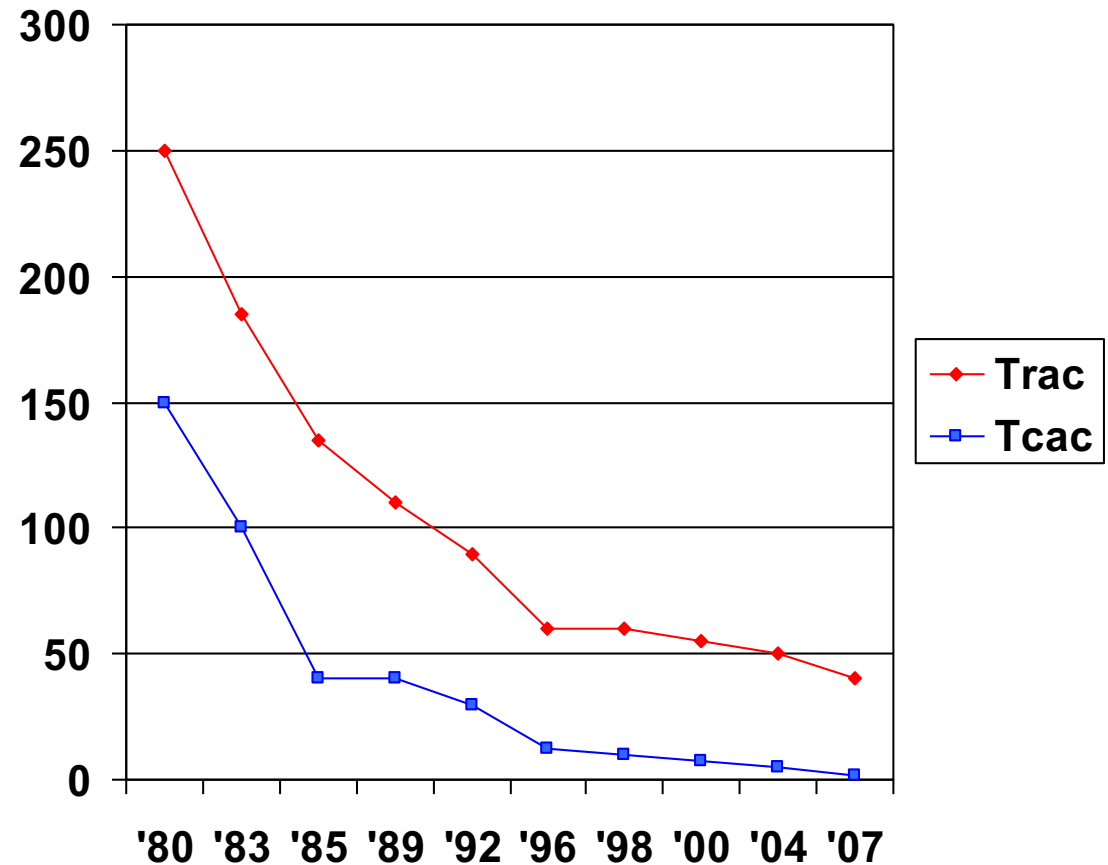
Measuring Cache Performance

- **Components of CPU time**
  - Program execution cycles
    - Includes cache hit time
  - Memory stall cycles
    - Mainly from cache misses
- **With simplifying assumptions:**

Memory stall cycles

$$= \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Cache Performance Example

- Given
    - I-cache miss rate = 2%
    - D-cache miss rate = 4%
    - Miss penalty = 100 cycles
    - Base CPI (ideal cache) = 2
    - Load & stores are 36% of instructions
- Miss cycles per instruction
    - I-cache: $0.02 \times 100 = 2$
    - D-cache: $0.36 \times 0.04 \times 100 = 1.44$
- Actual CPI = 2 + 2 + 1.44 = 5.44
    - Ideal CPU is 5.44/2 =2.72 times faster

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-cache miss rate = 5%
  - AMAT = 1 + 0.05 × 20 = 2ns
    - 2 cycles per instruction

Performance Summary

- 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

Replacement Policy

- Direct mapped: no choice
- Set associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
- 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

Replacement Policy

Example: Cache N-A4W: counter 2bits (LRU)
Which is the LRU block?

BLOCK REFERENCED	C_{B0}	C_{B1}	C_{B2}	C_{B3}	STATE	LRU
Initial state	0	0	0	0	Empty blocks	B0,B1,B2,B3
Error cache access	0	1	1	1	B0 full	B1,B2, B3
Error cache access	1	0	2	2	B0,B1 full	B2,B3
Hit in B0	0	1	2	2	B0,B1 full	B2,B3
Error cache access	1	2	0	3	B0,B1,B2 full	B3
Error cache access	2	3	1	0	All blocks full	B1
Hit in B1	3	0	2	1	All blocks full	B0
Error cache access	0	1	3	2	All blocks full	B2
Error cache access	1	2	0	3	All blocks full	B3

Multilevel Caches

- Primary cache attached to CPU
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache

Multilevel Cache Example

- ## Given

  - CPU base CPI = 1, clock rate = 4GHz

  - Miss rate/instruction = 2%

  - Main memory access time = 100ns

- ## With just primary cache

  - Miss penalty = 100ns/0.25ns = 400 cycles

  - Effective CPI = $1 + 0.02 \times 400 = 9$

Example (cont.)

- Now add L-2 cache

  - Access time = 5ns

  - Global miss rate to main memory = 0.5%

- Primary miss with L-2 hit

  - Penalty = 5ns/0.25ns = 20 cycles

- Primary miss with L-2 miss

  - Extra penalty = 400 cycles

- CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4

- Performance ratio = 9/3.4 = 2.6

Multilevel Cache Considerations

- Primary cache
  - Focus on minimal hit time

- L-2 cache
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact

- Results
  - L-1 cache usually smaller than a single cache
  - L-1 block size smaller than L-2 block size

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 analyse
  - Use system simulation

Interactions with Software

- Misses depend on memory access patterns
  - Algorithm behavior
  - Compiler optimization for memory access
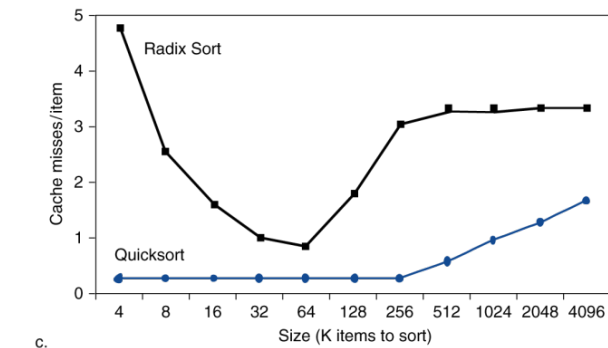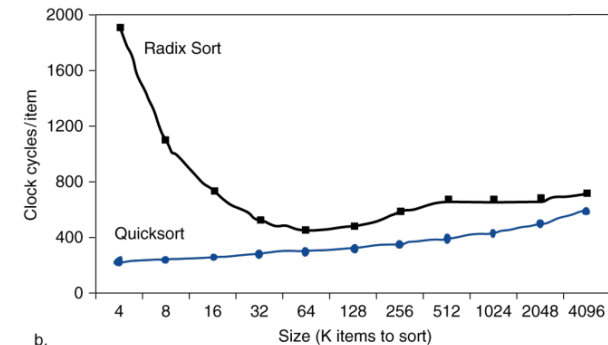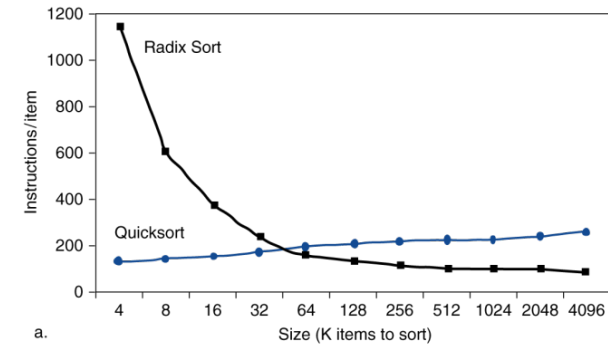
Virtual Memory

- Use main memory as a "cache" for secondary (disk) storage
  - Managed jointly by CPU hardware (MMU) 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

- Fixed-size pages (e.g., 4K)
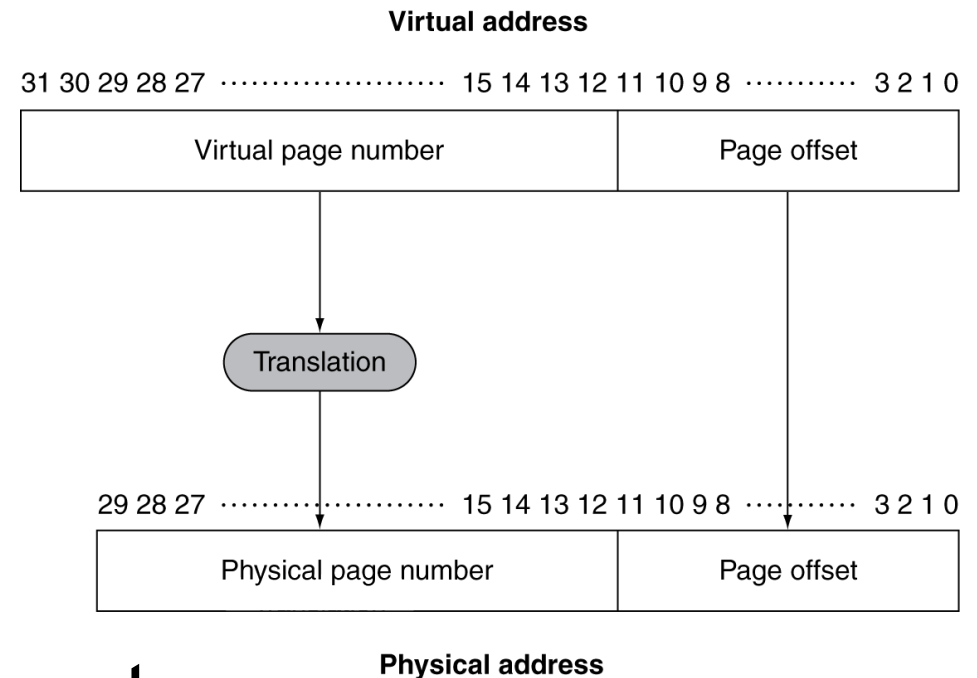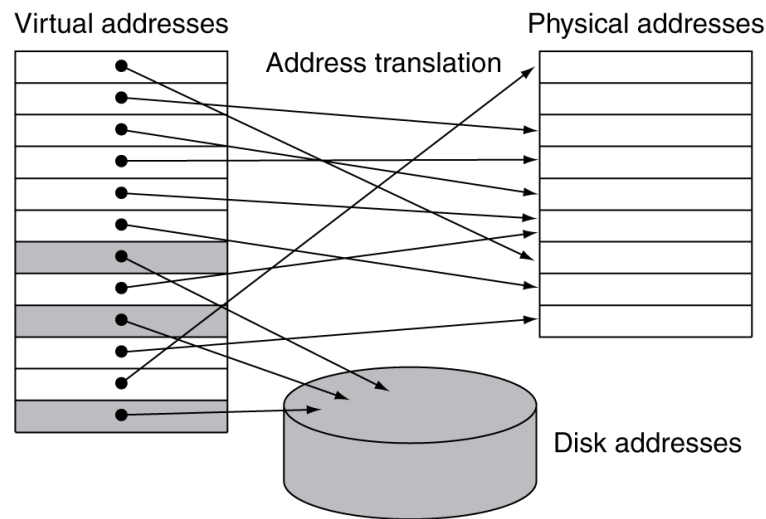
- Variable-size segments
- Segments with fixed-size pages

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

Page Tables

- Stores placement information
  - Array of page table entries (PTE), 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
  - Plus other status bits (referenced, dirty, …)
- If page is not present
  - PTE (Page Translation Entry) can refer to location in swap space on disk

Mapping Pages to Storage
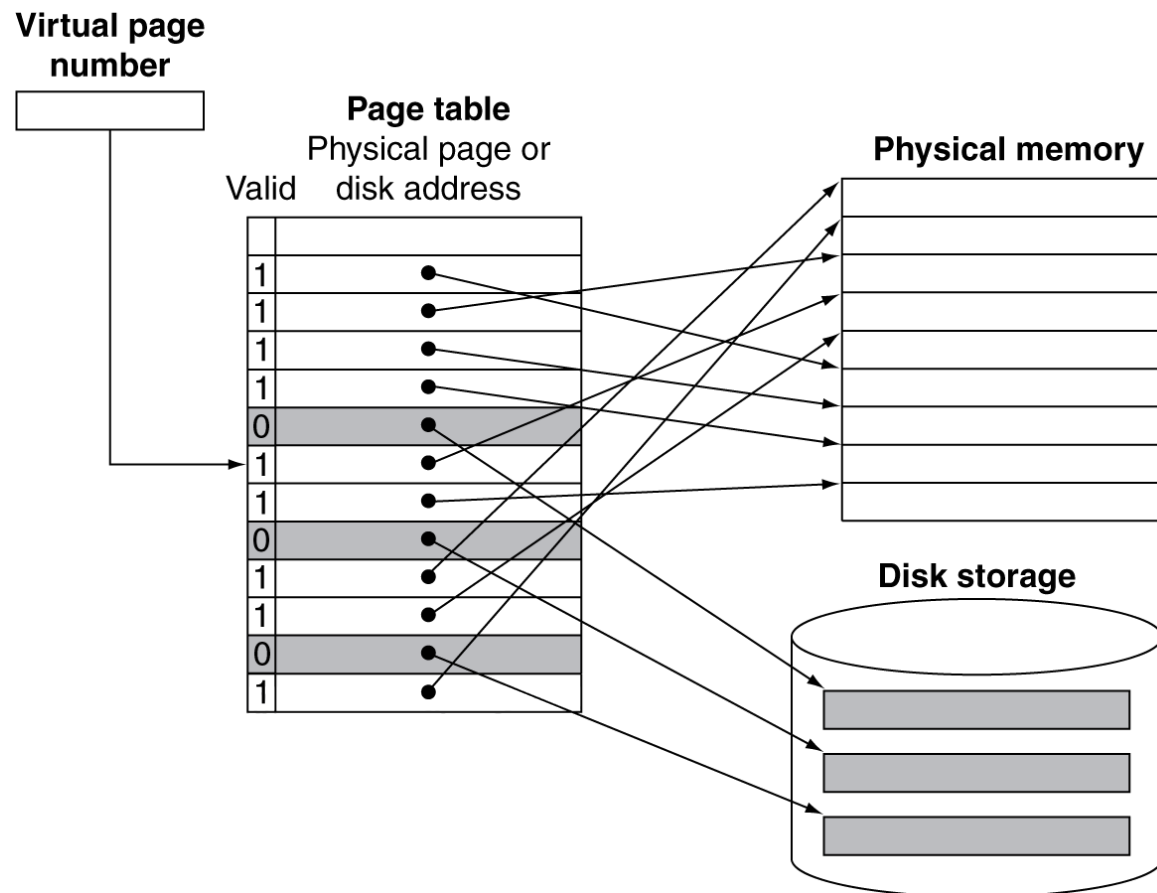
Translation Using a Page Table
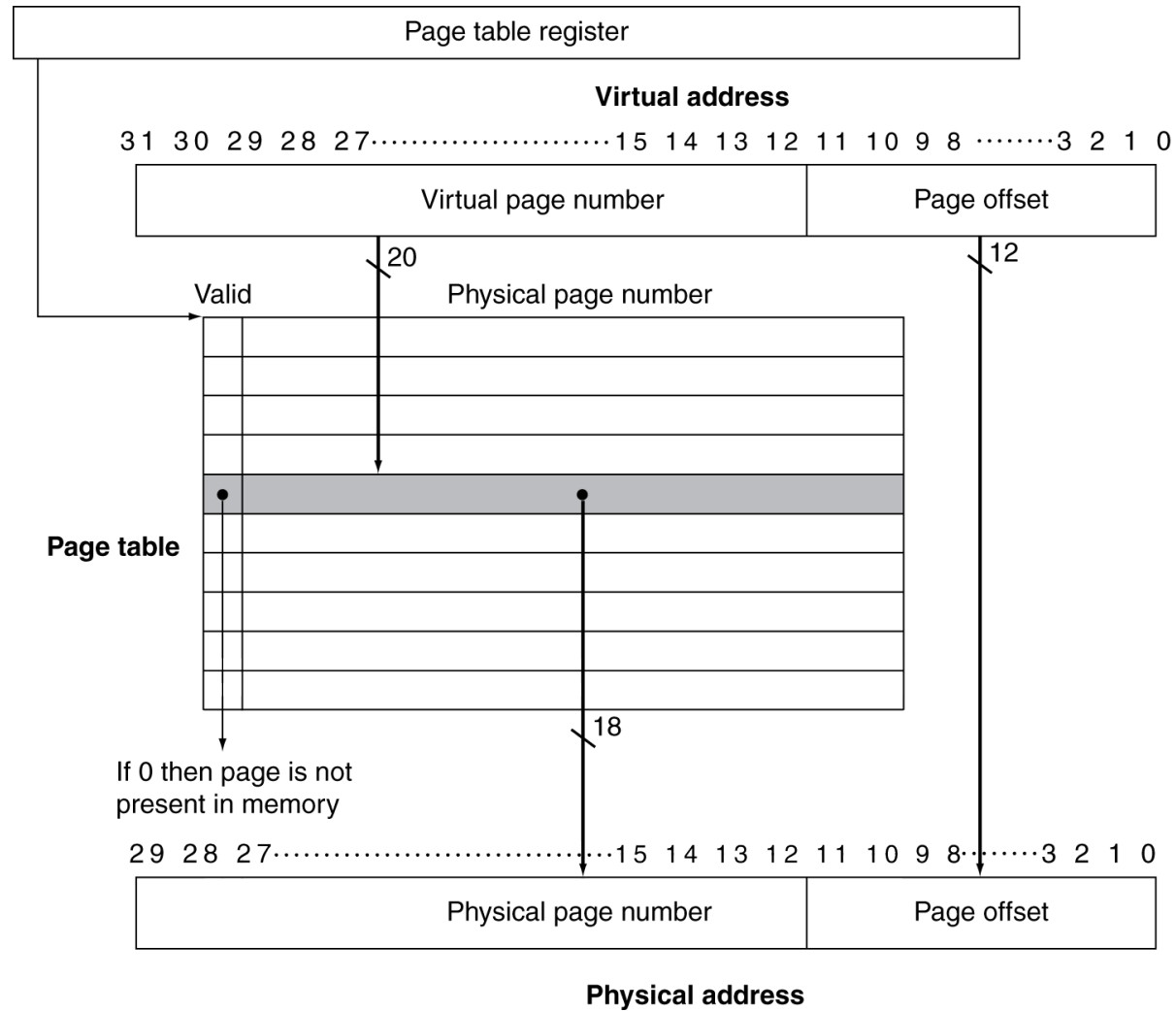
Translation Using a Page Table

Example: Virtual memory: 4 GB (2^{32}), real: 16 MB (2^{24}). Page size: 4 kB (2^{12})

CPU ⟶ {FFAACD00} V.A.

20 b	12 b
FFAAC	**D00**

We need SRAM of 2^{20}x12 bits!

Impossible full associative (2^{20} comparators)

```
00000
        312

FFFFF
```

312	D00

{312D00} ⟶ MEMORY
Real addr.
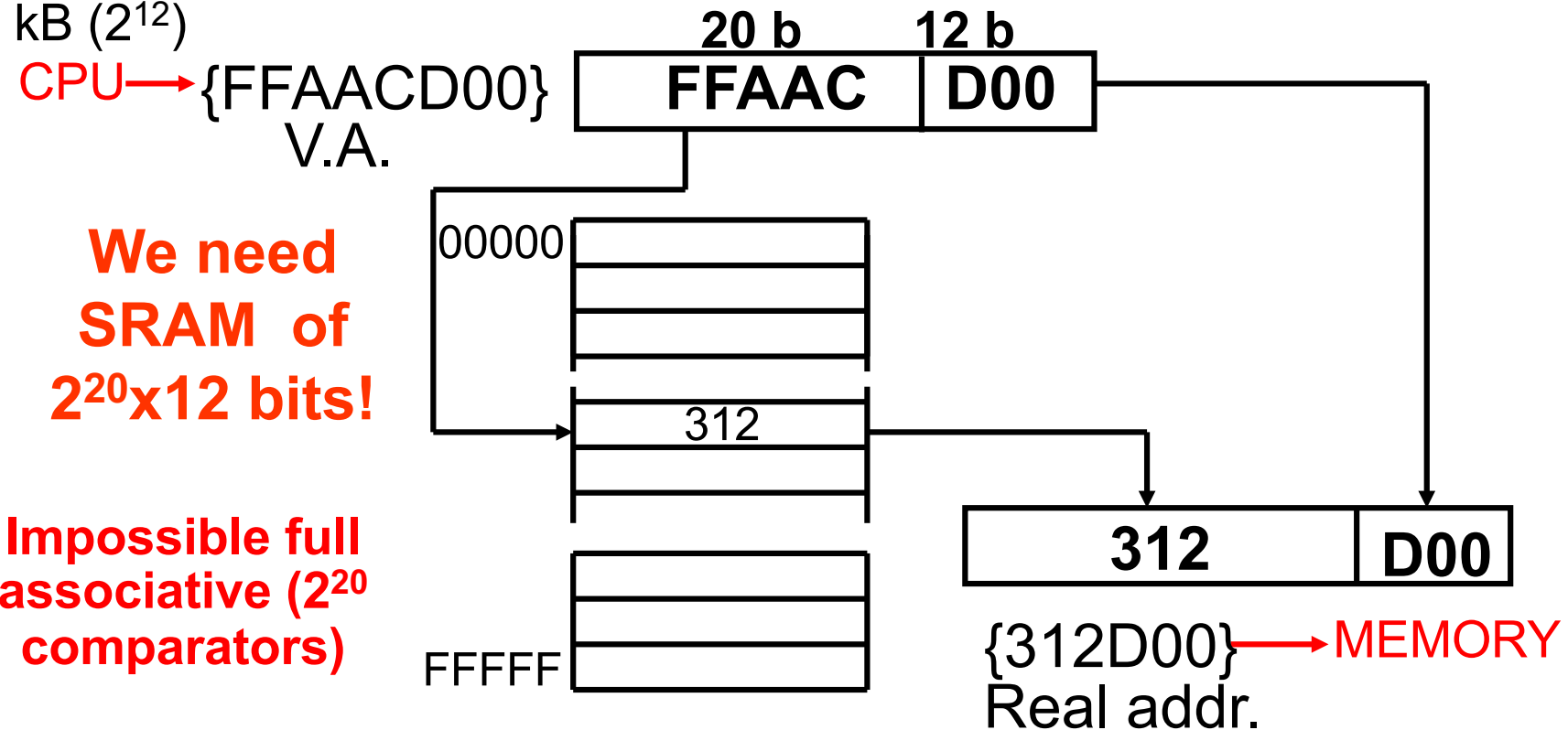
To reduce the page table size, it is built according to the process requirements.

Translation Using a Page Table

❑Multi-level Page Table.

    ❑**To reduce the size of the page table (not all the sub-tables reside in memory)**

CPU

V.A. | Level 1 | Level 2 | .... | Level n | OFFSET |

Register pointer

Base N2

Pag. T. Level-1 or Directory

Base N3

Pag. T. Level-2

FRAME

Pag. T. Level-n

FRAME | OFFSET

Real addr.

MEMORY

Level 1 table is the directory and is like a cache.
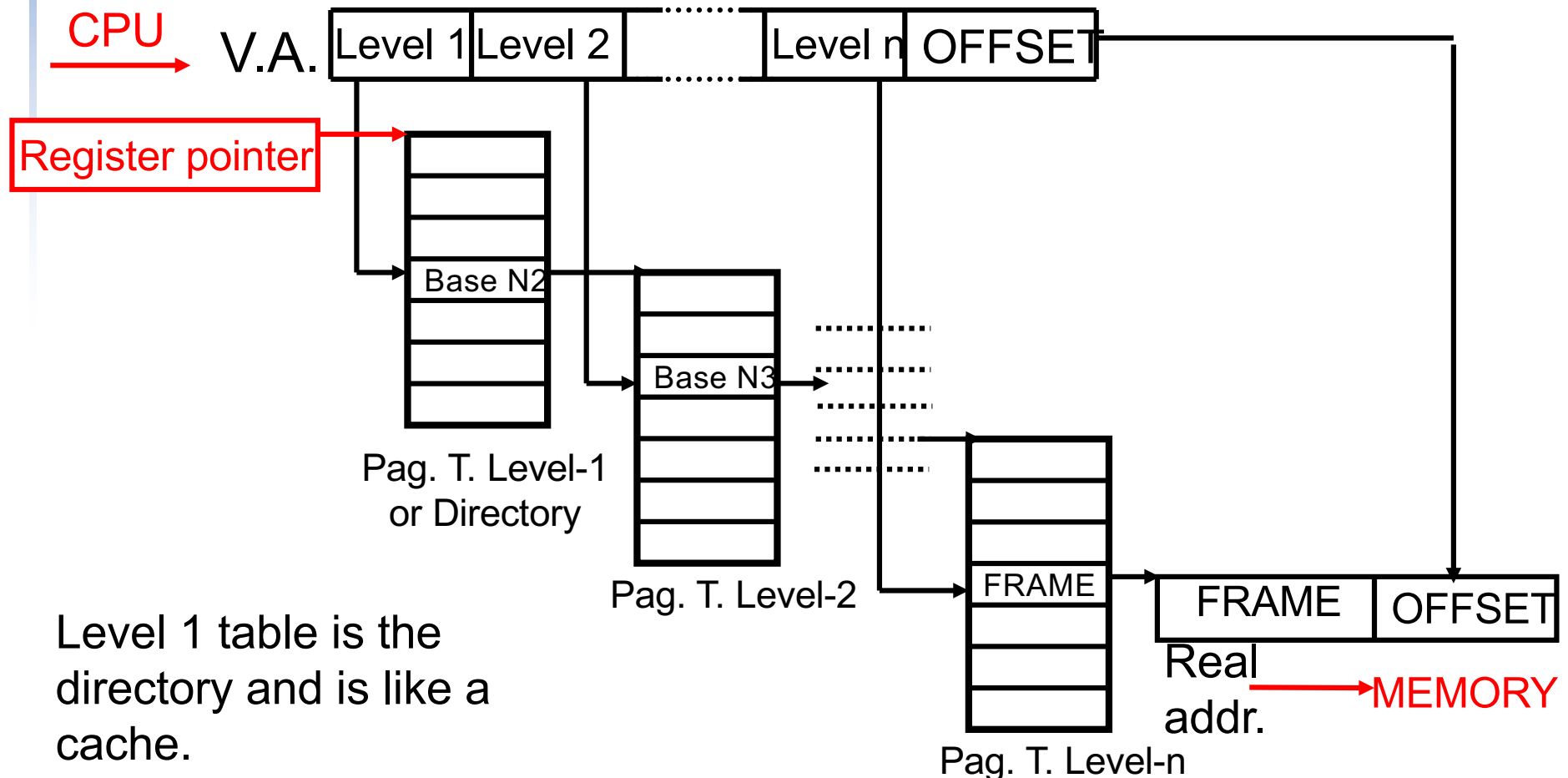
Translation Using a Page Table

Page Table with 3 levels

Example: Virtual Memory: 4 GB (2^{32}), Real: 16 MB (2^{24}). Page size: 1 kB (2^{10})

CPU ⟶ {FFAACD00} V.A.

4b	9b	9b	10b
F	FA1	010C11	0100

00₀

0

FF₁
Table N2

00₀

FF₁
Table N3

0

F
Directory

00₀

FA1

FF₁
Table N2

010C11

00₀

312₁₁

FF₁
Table N3

| 312₁₁ | 0100 |

{312D00} Real addr. ⟶ MEM

Descriptors of 2 Bytes, each table (except the first) fits 1 page
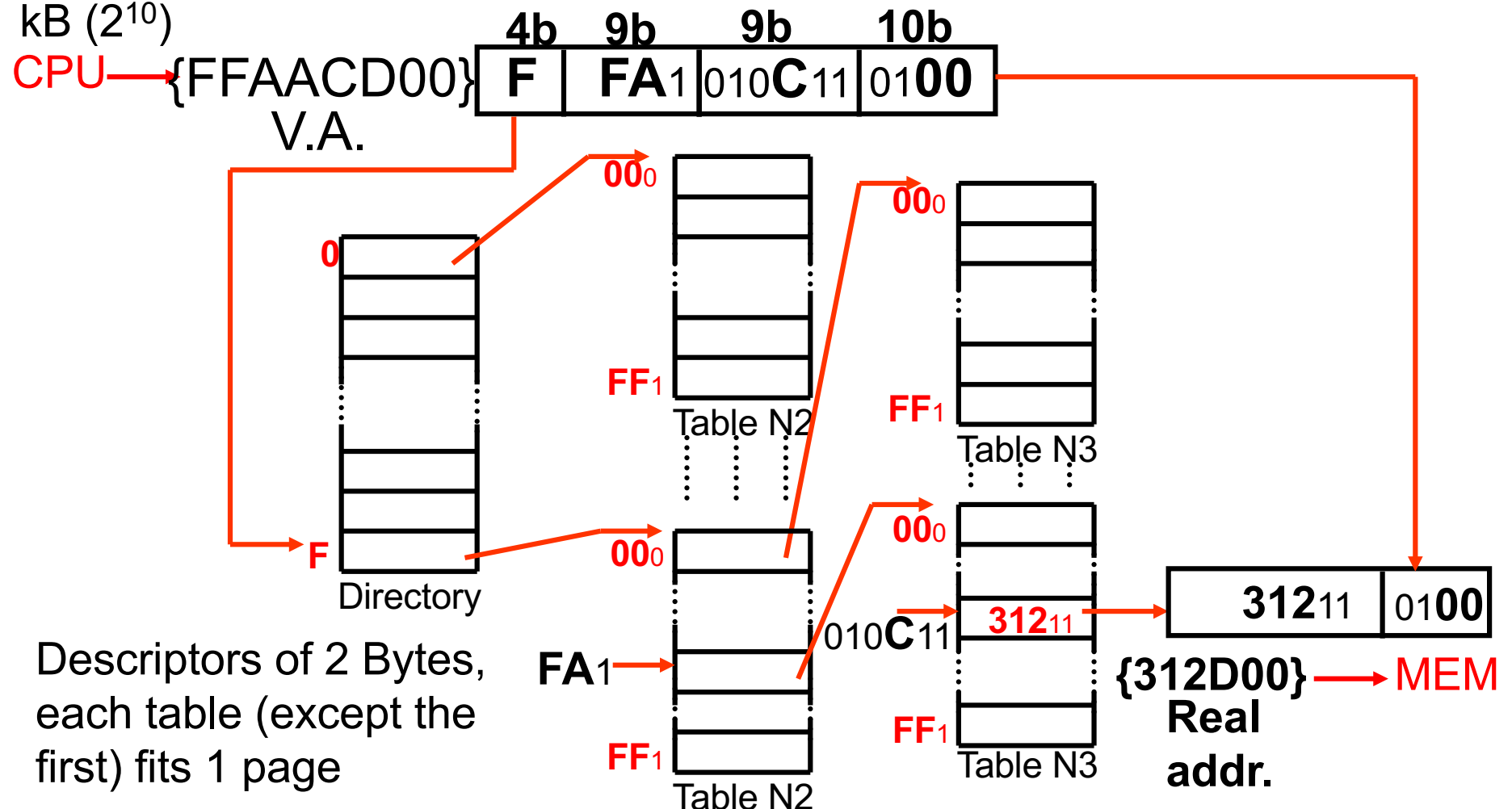
Fast Translation Using a TLB

- Address translation would appear to require extra memory references
    - One to access the PTE (Page Translation Entry)
    - 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

Fast Translation Using a TLB

TLB

Virtual page number

	Valid	Dirty	Ref	Tag	Physical page address
	1	0	1		●
	1	1	1		●
	1	1	1		●
	1	0	1		●
	0	0	0		
	1	0	1		●

Physical memory

Page table

Physical page or disk address

Valid	Dirty	Ref	
1	0	1	●
1	0	0	●
1	0	0	●
1	0	1	●
0	0	0	●
1	0	1	●
1	0	1	●
0	0	0	●
1	1	1	●
1	1	1	●
0	0	0	●
1	1	1	●

Disk storage
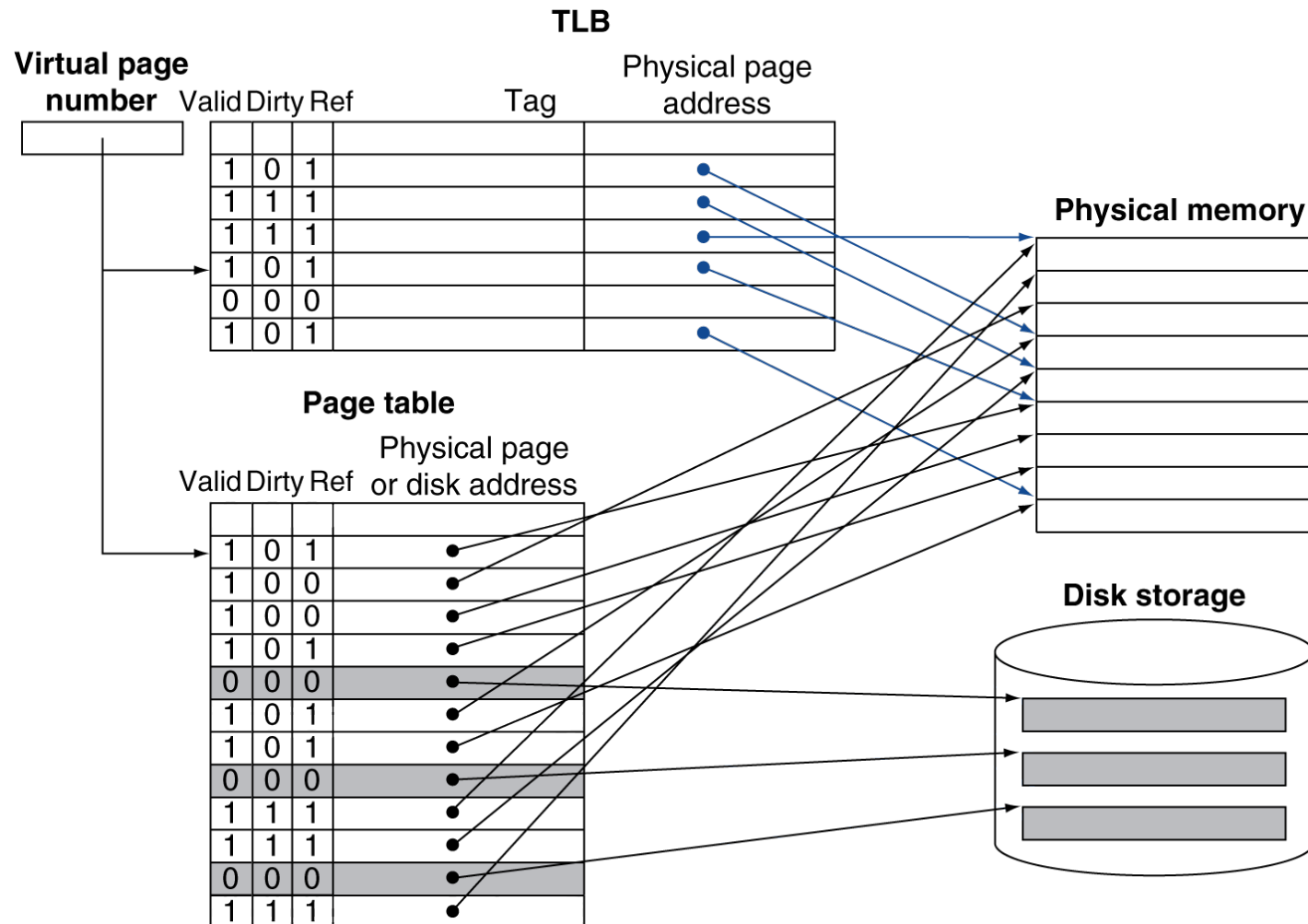
Fast Translation Using a TLB

TLB N-way Associative, 2 ways, 16 entries/way

Ejemplo: Virtual Memory: 4 GB (2^{32}), Real: 16 MB (2^{24}). Page size: 4 kB (2^{12})

CPU → {FFAACD00}

16b	**4b**	**12b**
FFAA	C	D00

	PAGE	WAY-1	FRAME	PAGE	WAY-2	FRAME
0						
	AAFF		2F0	FFAA		312
F						

312	D00

{312D00} → MEMORY
Real addr.

E

H

53
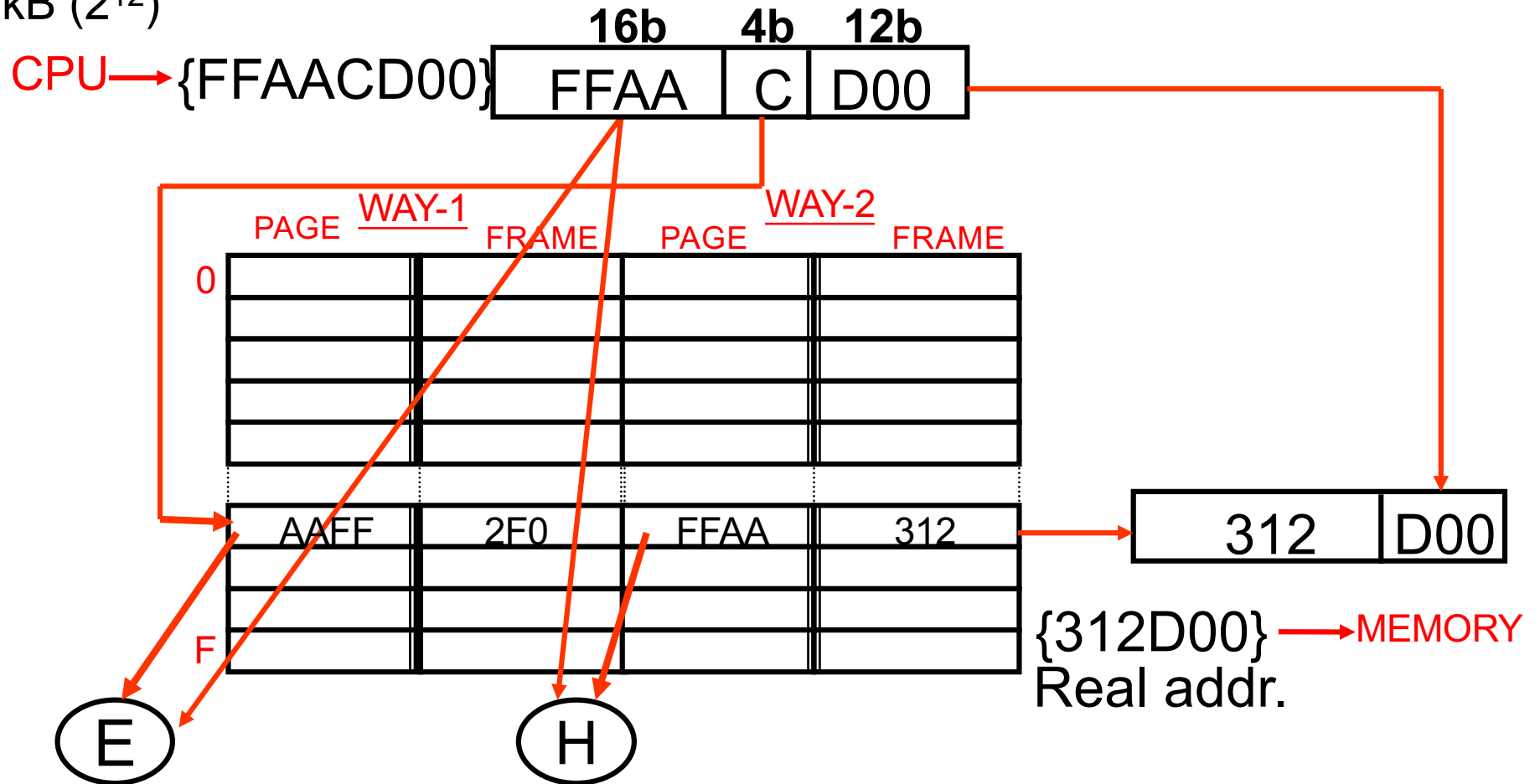
TLB Misses

- If page is in memory
  - Load the PTE from memory and retry
  - Could be handled in hardware
    - Can get complex for more complicated page table structures
  - Or in software
    - Raise a special exception, with optimized handler
- If page is not in memory (page fault)
  - OS handles fetching the page and updating the page table
  - Then restart the faulting instruction

TLB Miss Handler

- TLB miss indicates
    - Page present, but PTE not in TLB
    - Page not present
- Must recognize TLB miss before destination register overwritten
    - Raise exception
- Handler copies PTE from memory to TLB
    - Then restarts instruction
    - If page not present, page fault will occur

Page Fault Handler

- Use faulting virtual address to find PTE

- Locate page on disk

- Choose page to replace
  - If dirty, write to disk first

- Read page into memory and update page table

- Make process runnable again
  - Restart from faulting instruction

Replacement and Writes

- To reduce page fault rate, 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

Page table information

The information stored in a TLB or page table entry is called descriptor and contains:

- Page frame: It gives the real address (Real address = FRAME & OFFSET)
- Bits to control:
  - Present bit: '1' indicates the page referenced resides in main memory
  - Use bit: '1' to indicate that some element of the page has been referenced. It is used to decide which page is replaced.
  - Dirty bit:'1' to indicate that some data in the page has been modified (written).
  - Protection bits: supervisor, only-readable, non-cacheable, used by the OS.
  - Replacement bits: to apply the replacement algorithms (LRU, etc).

Translation using a Segment Table

The address is divided into segment and offset

CPU → V. A. | SEGMENT | OFFSET

POINTER TO A SEGMENT TABLE

DESCRIPTOR

Check that the offset is less than the size of the segment (which may vary)

>?,+

Final real address is a sum and not a concatenation as in page table.

Real address → MEMORY
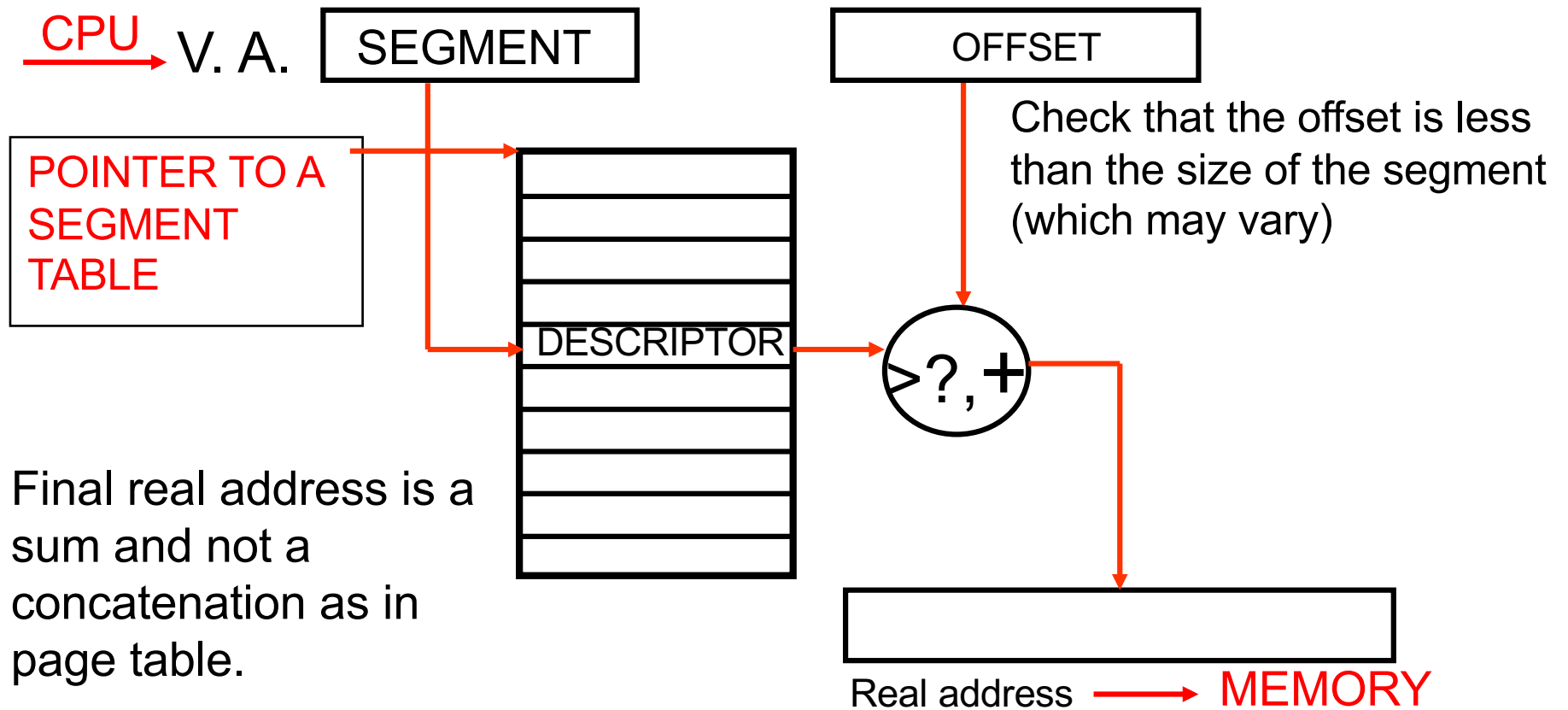
Translation using a Segment Table

What does the segment descriptor contain?

- **Segment start address:** It is added to the offset to compute the real address

- **Segment size:** It must be greater than the offset

- **Bits to control**
  - Present bit in main memory
  - Protection bit: against write operations (code segment)
  - Exclusion bit: to restringe the access (system security)

- **Bits for replacement algorithms: LRU**

Translation using Segment+Page Tables
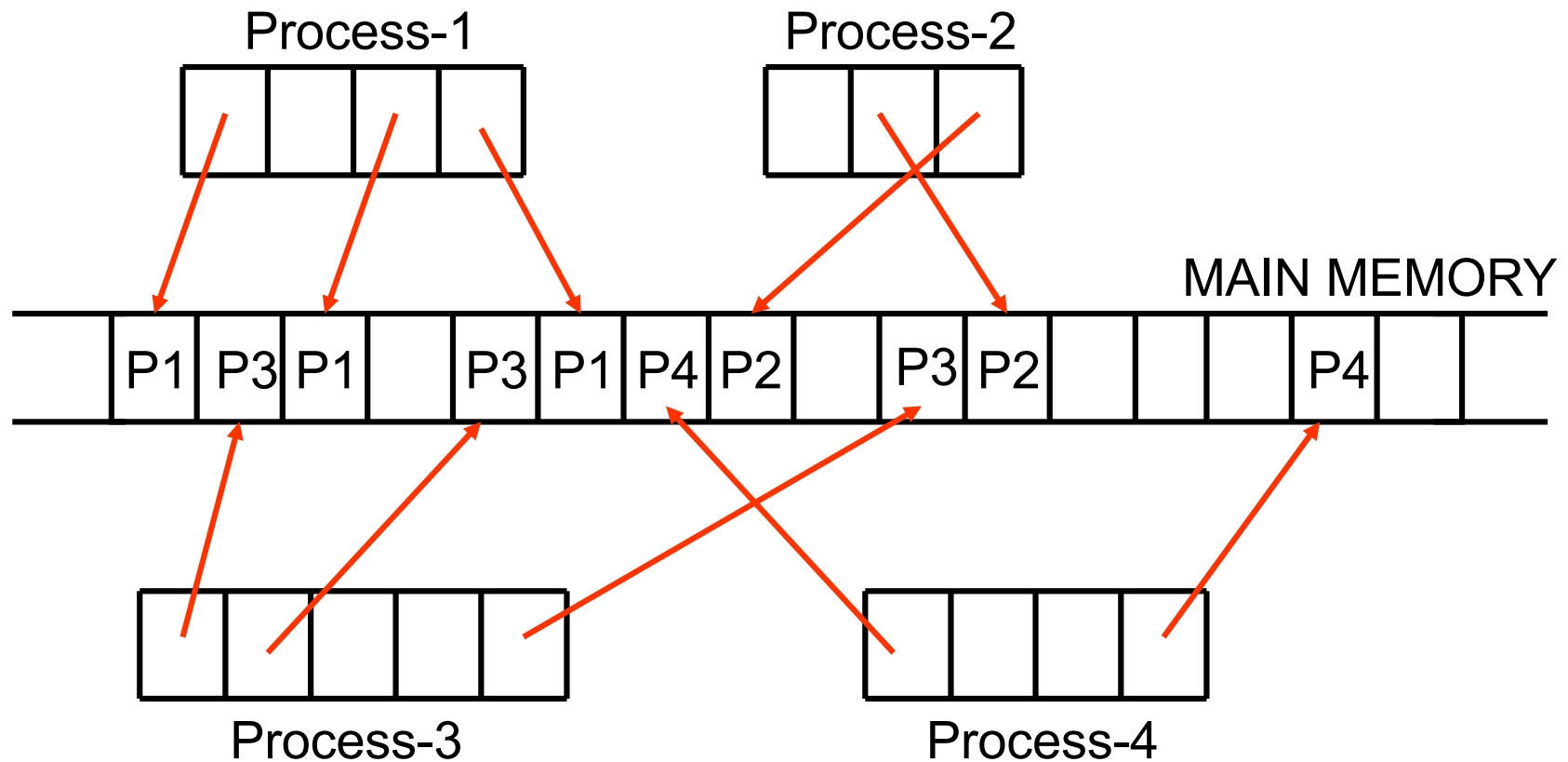
Translation using Segment+Page Tables

How is the translation between virtual address and real address?

OFFSET

CPU

V.A.

SEGMENT

PAGE   OFFSET

POINTER

BASE TAB   SIZE   >?

FRAME

FRAME   OFFSET

Real address ⟶ MEMORY
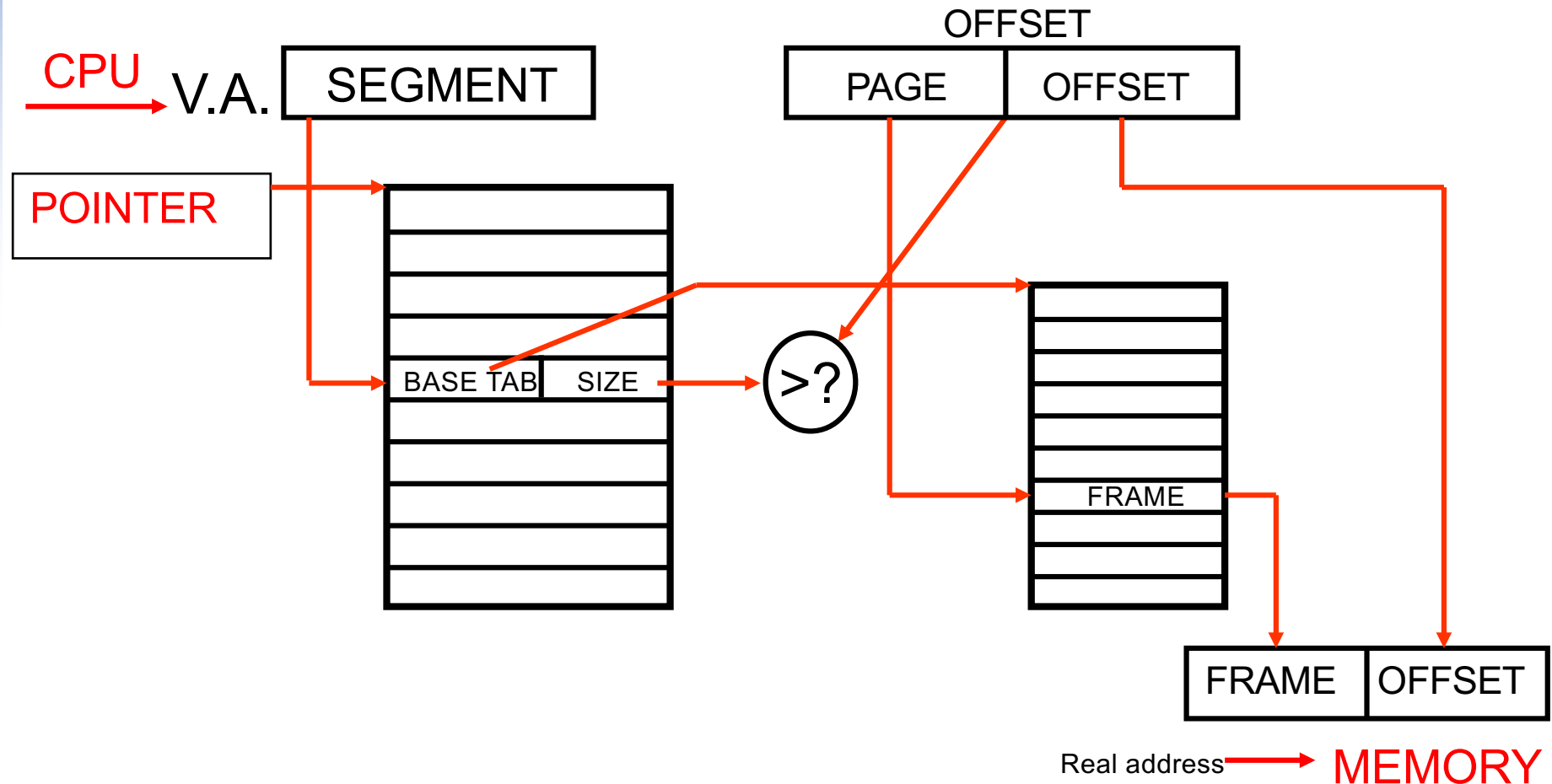
Translation using Segment+Page Tables

TLB with Segment+Page Tables

OFFSET

CPU → V.A. | SEGMENT | PAGE | OFFSET |

TLB columns: SEG-PAG | FRAME

FRAME | OFFSET
Real address → MEMORY

TLB
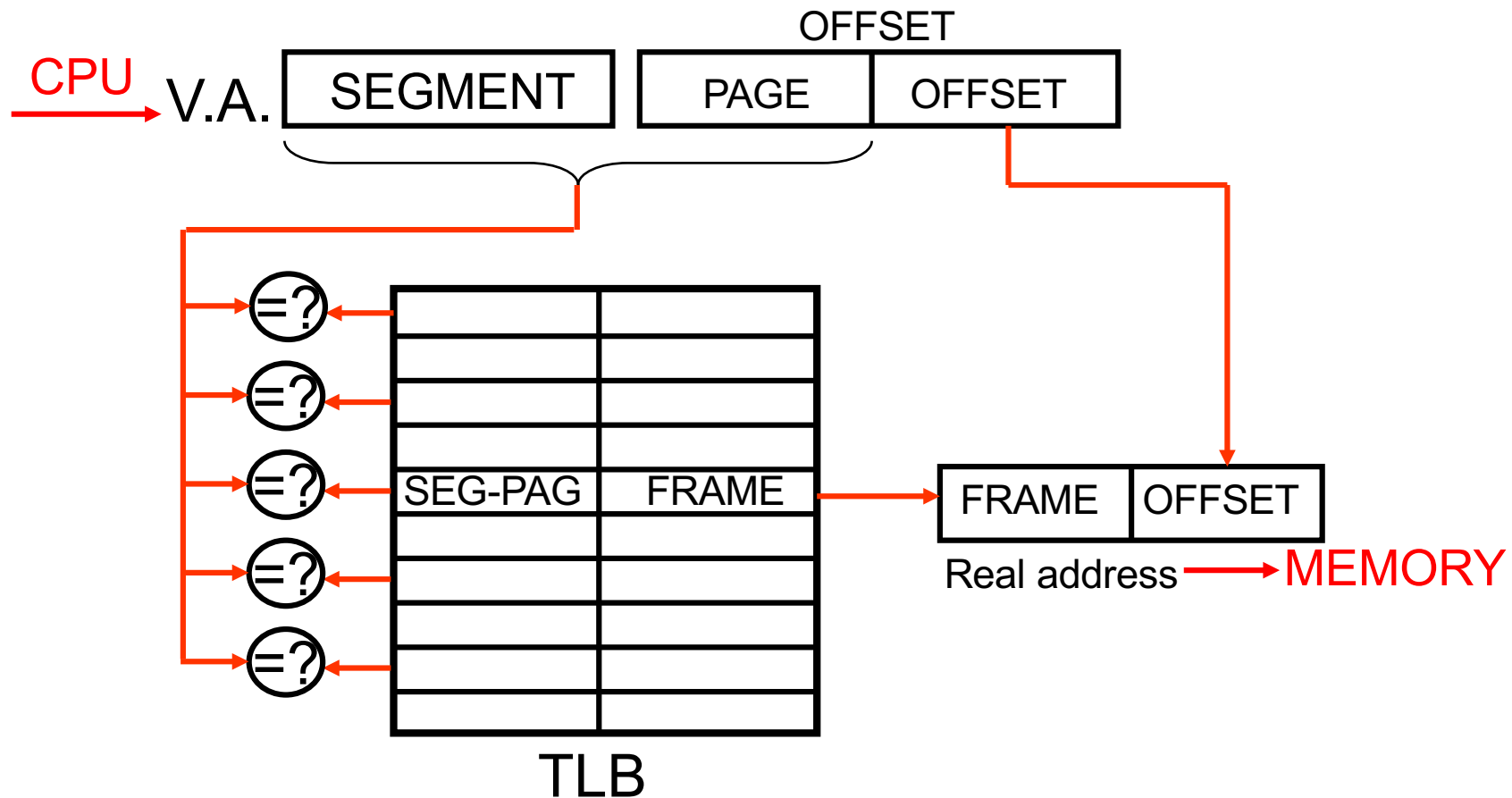
TLB and Cache Interaction
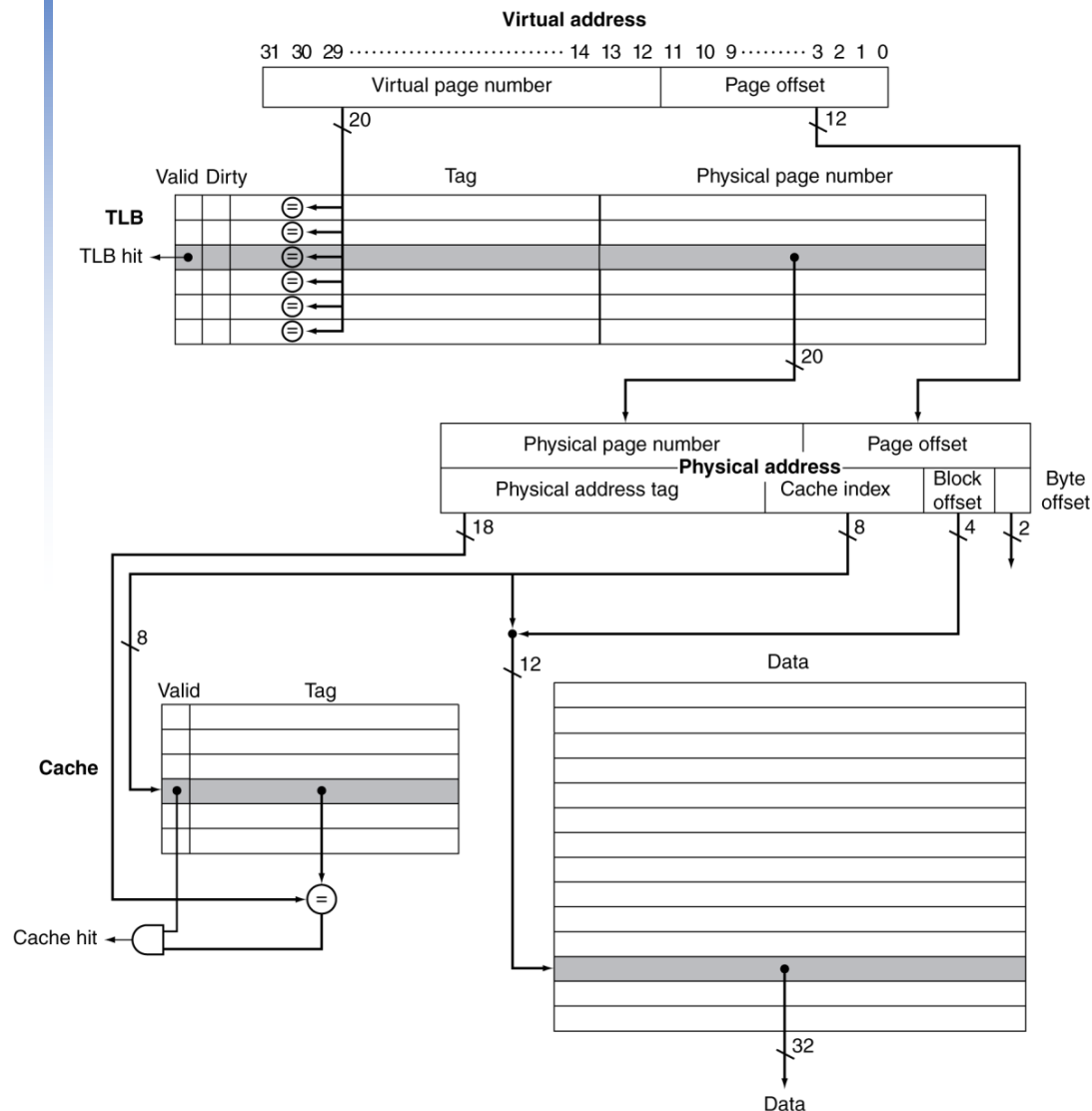
- **If cache tag uses physical address**
  - Need to translate before cache lookup

- **Alternative: use virtual address tag**
  - Complications due to aliasing
    - Different virtual addresses for shared physical address

64

TLB and Cache Interaction

- **Virtual Cache** (from virtual address)
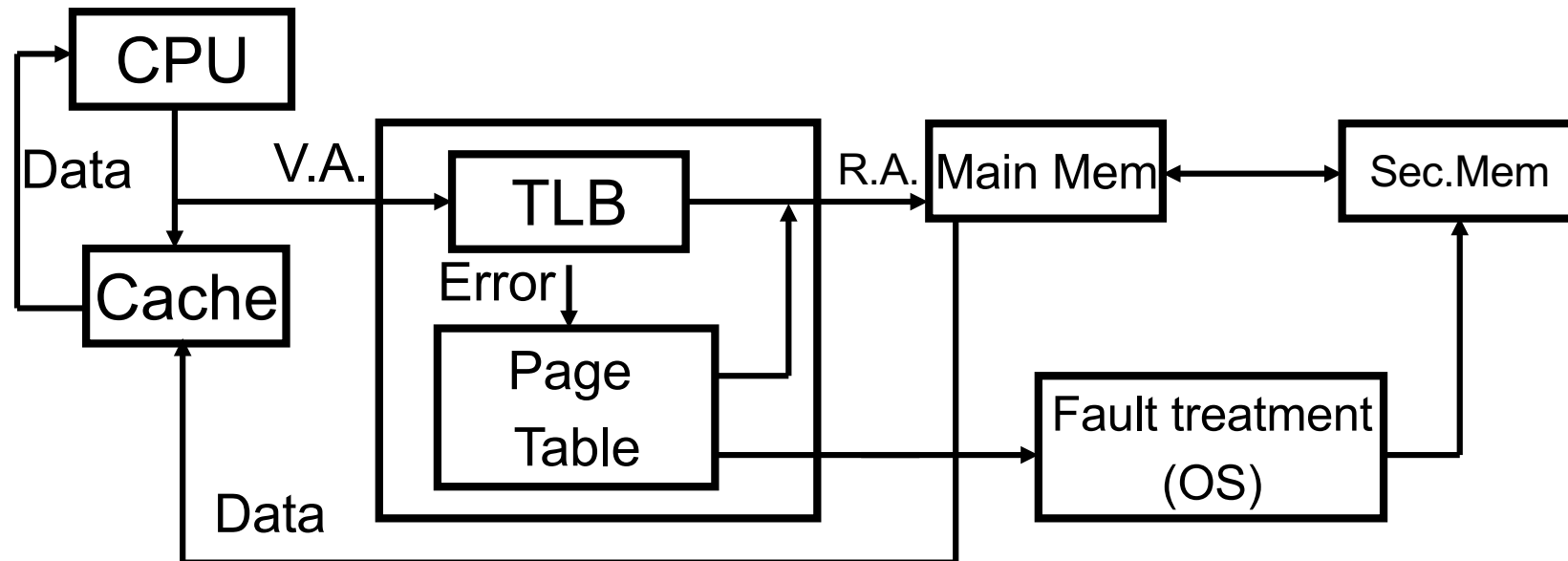
❑Same time access to cache and TLB

❑ Memory access time: hit cache, t_c, error cache, $t_{TLB}+t_B+t_c$

❑*aliasing*: two virtual addresses to the same real address -> 2 entries in virtual cache for the same data

❑ Cache problem with different processes: there can be virtual addresses duplicated. To avoid this, a process identifier is added to the virtual address

65

TLB and Cache Interaction

- **Real Cache** (from real address)

```
                              Data
┌──────────┐ ◄────────────────────────────────────┐
│   CPU    │                                       │
└──────────┘                                       │
     │         V.A.      ┌──────────┐  R.A.  ┌──────────┐   ┌────────────┐   ┌────────────┐
     └────────────────►  │   TLB    │ ─────► │  Cache   │◄─►│  Main Mem  │◄─►│  Sec. Mem  │
                         └──────────┘        └──────────┘   └────────────┘   └────────────┘
              Error │                                                              ▲
                    ▼                                                              │
                ┌──────────┐                    ┌──────────────┐                  │
                │  Page    │                    │ Fault treatment                 │
                │  Table   │ ─────────────────► │    (OS)       │ ────────────────┘
                └──────────┘                    └──────────────┘
```
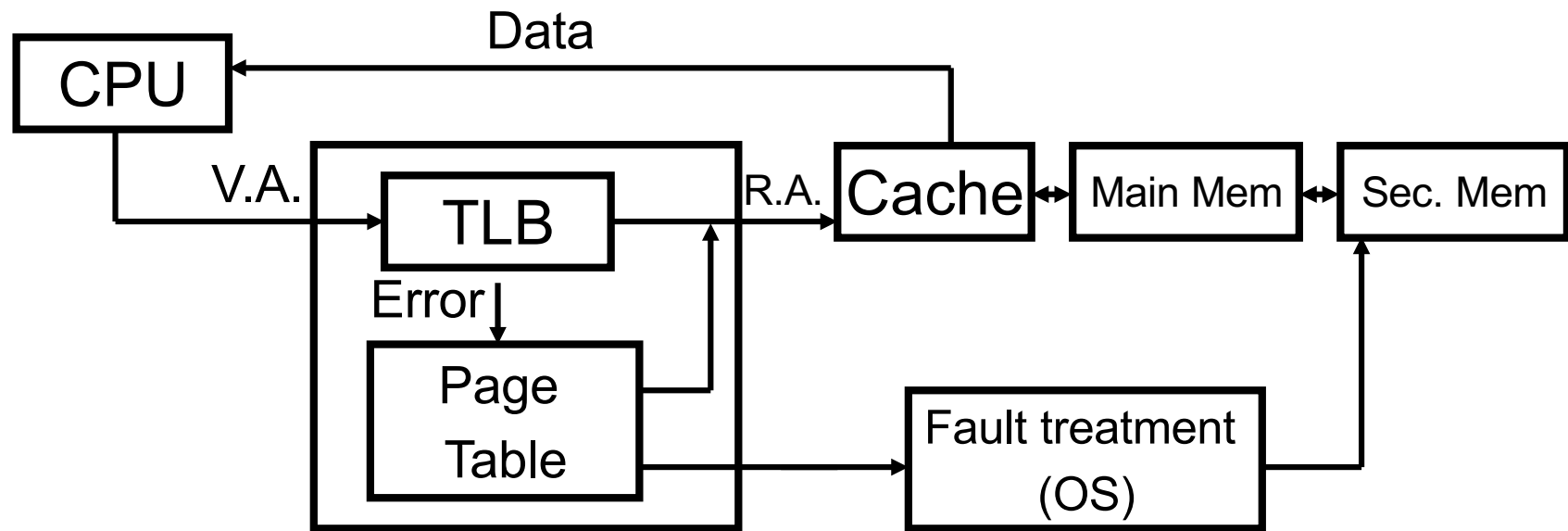
❏ Minimum memory access time: TLB time + cache time

❏ Solved having several address spaces

❏ To speed-up, page offset can contain the index and the byte in block of the cache

TLB and Cache Interaction
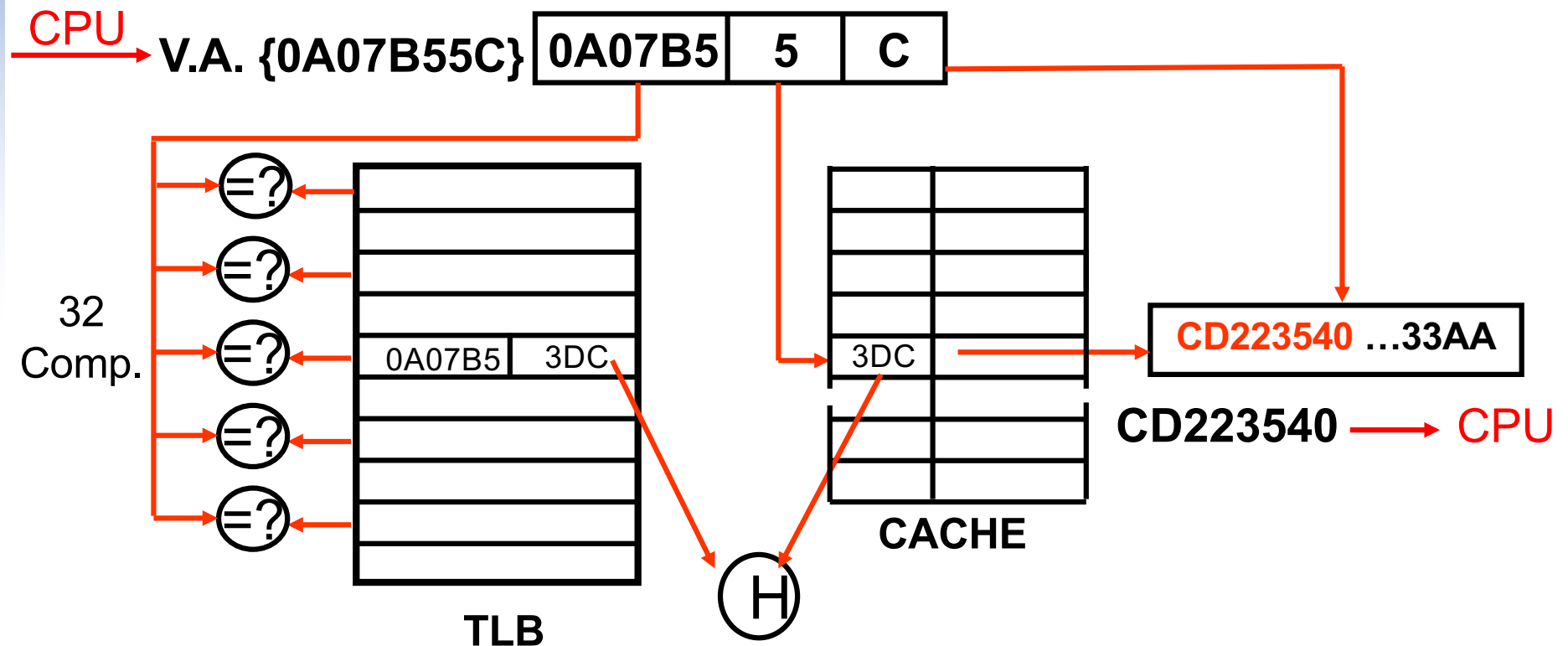
- Real cache with parallel access to the TLB frame and cache tag. Next, compare between frame and tag.
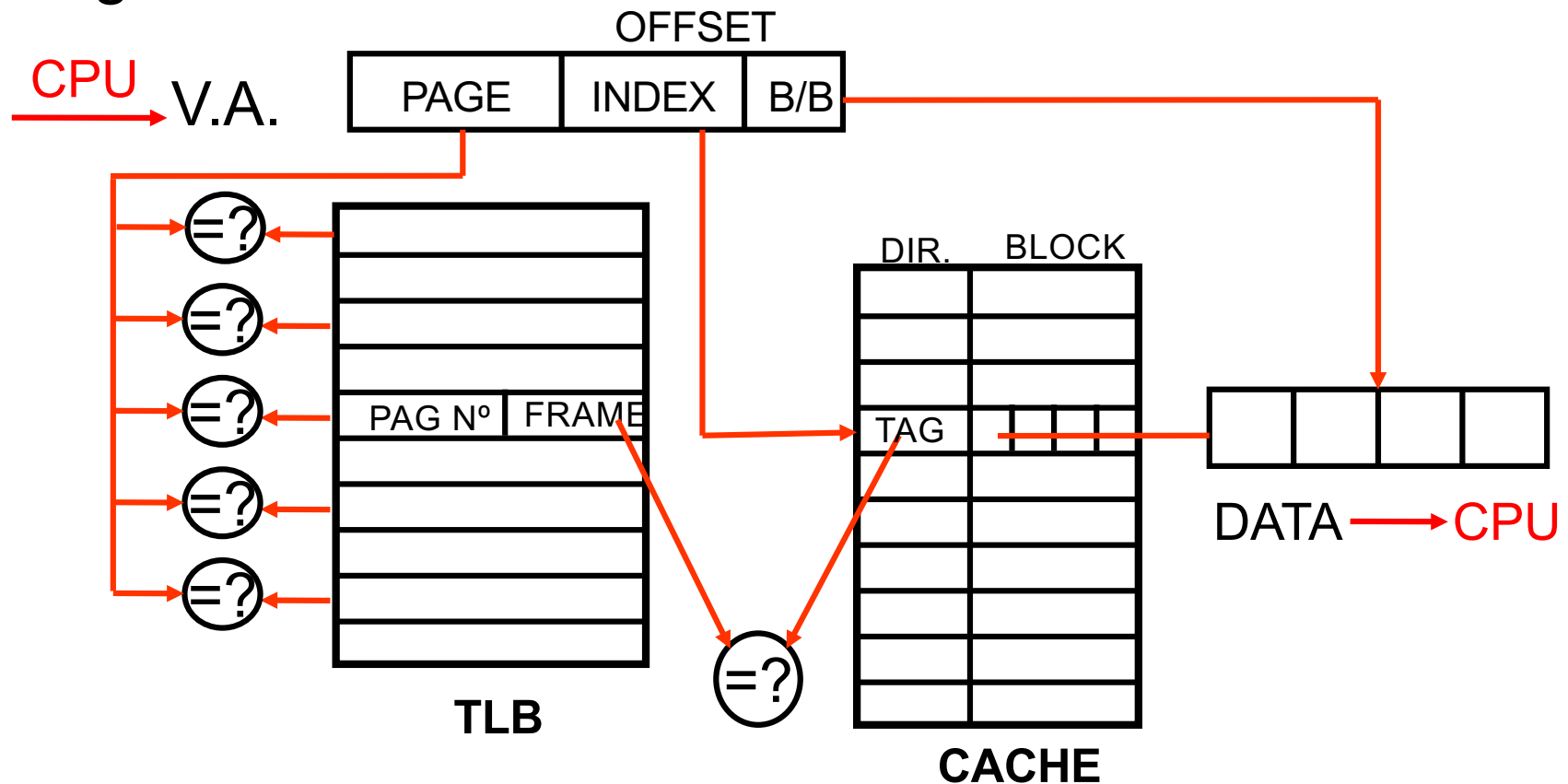
TLB and Cache Interaction

Example.- V.A.: 32b; R.A.: 20b; Pag. table: 256 Bytes

TLB FA 32 entries; Cache DM 256 Bytes, 16 B/B.

"Offset must include the real cache index so that we can access TLB and cache in parallel"

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)

The Memory Hierarchy

- Common principles apply at all levels of the memory hierarchy
  - Based on notions of caching
- At each level in the hierarchy
  - Block placement
  - Finding a block
  - Replacement on a miss
  - Write policy

Block Placement

- Determined by associativity
    - Direct mapped (1-way associative)
        - One choice for placement
    - n-way set associative
        - n choices within a set
    - Fully associative
        - Any location
- Higher associativity reduces miss rate
    - Increases complexity, cost, and access time

Finding a Block

Associativity	Location method	Tag comparisons
Direct mapped	Index	1
n-way set associative	Set index, then search entries within the set	n
Fully associative	Search all entries	#entries
	Full lookup table	0

- **Hardware caches**
  - Reduce comparisons to reduce cost
- **Virtual memory**
  - Full table lookup makes full associativity feasible
  - Benefit in reduced miss rate

Replacement

- Choice of entry to replace on a miss
    - Least recently used (LRU)
        - Complex and costly hardware for high associativity
    - Random
        - Close to LRU, easier to implement
- Virtual memory
    - LRU approximation with hardware support

Write Policy

- ## Write-through
  - Update both upper and lower levels
  - Simplifies replacement, but may require write buffer

- ## Write-back
  - Update upper level only
  - Update lower level when block is replaced
  - Need to keep more state

- ## Virtual memory
  - Only write-back is feasible, given disk write latency

Sources of Misses

- Compulsory misses (aka cold start misses)
  - First access to a block
- Capacity misses
  - Due to finite cache size
  - A replaced block is later accessed again
- Conflict misses (aka collision misses)
  - In a non-fully associative cache
  - Due to competition for entries in a set
  - Would not occur in a fully associative cache of the same total size

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.

Virtual Machines

- Host computer emulates guest operating system and machine resources
  - Improved isolation of multiple guests
  - Avoids security and reliability problems
  - Aids sharing of resources
- Virtualization has some performance impact
  - Feasible with modern high-performance comptuers
- Examples
  - IBM VM/370 (1970s technology!)
  - VMWare
  - Microsoft Virtual PC

Virtual Machine Monitor

- Maps virtual resources to physical resources
  - Memory, I/O devices, CPUs
- Guest code runs on native machine in user mode
  - Traps to VMM on privileged instructions and access to protected resources
- Guest OS may be different from host OS
- VMM handles real I/O devices
  - Emulates generic virtual I/O devices for guest

Example: Timer Virtualization

- In native machine, on timer interrupt
  - OS suspends current process, handles interrupt, selects and resumes next process

- With Virtual Machine Monitor
  - VMM suspends current VM, handles interrupt, selects and resumes next VM

- If a VM requires timer interrupts
  - VMM emulates a virtual timer
  - Emulates interrupt for VM when physical timer interrupt occurs
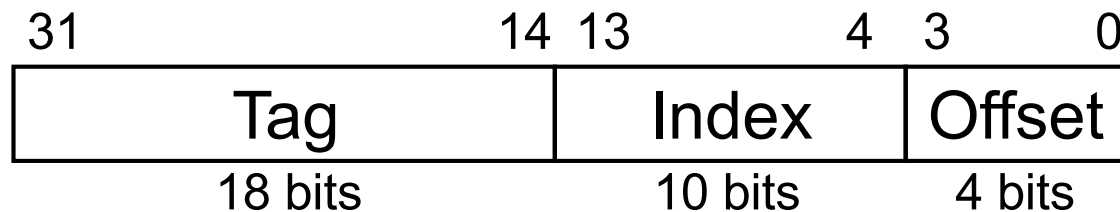
Instruction Set Support

- **User and System modes**

- **Privileged instructions only available in system mode**
  - Trap to system if executed in user mode

- **All physical resources only accessible using privileged instructions**
  - Including page tables, interrupt controls, I/O registers

- **Renaissance of virtualization support**
  - Current ISAs (e.g., x86) adapting

Cache Control

- Example cache characteristics
  - Direct-mapped, write-back, write allocate
  - Block size: 4 words (16 bytes)
  - Cache size: 16 KB (1024 blocks)
  - 32-bit byte addresses
  - Valid bit and dirty bit per block
  - Blocking cache
    - CPU waits until access is complete

31 14	13 4	3 0
Tag	Index	Offset
18 bits	10 bits	4 bits

Interface Signals

CPU

Read/Write →
Valid →
Address 32 →
Write Data 32 →
Read Data 32 ←
Ready ←

Cache

Read/Write →
Valid →
Address 32 →
Write Data 128 →
Read Data 128 ←
Ready ←

Memory

Multiple cycles per access
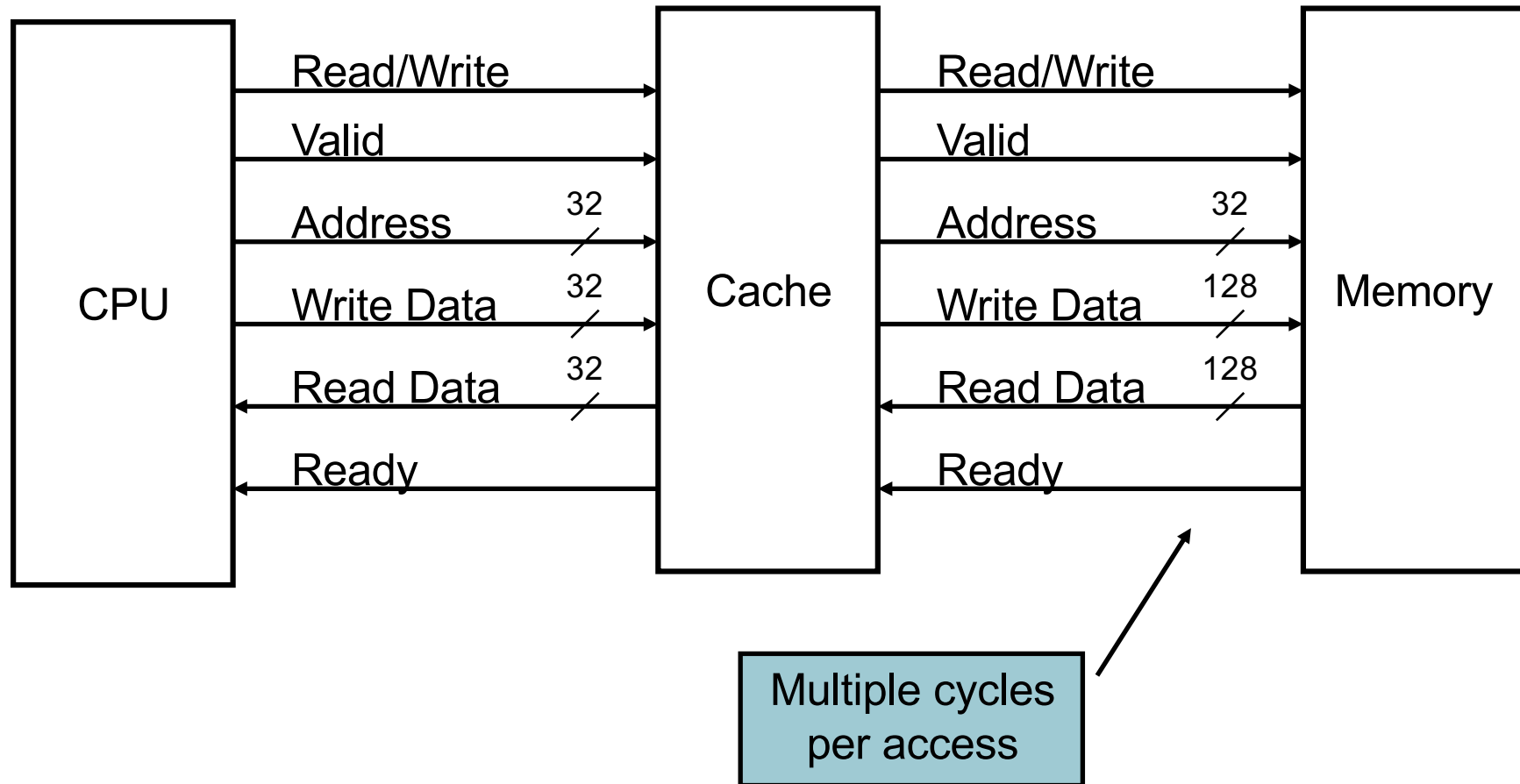
Finite State Machines

- Use an FSM to sequence control steps

- Set of states, transition on each clock edge

  - State values are binary encoded

  - Current state stored in a register

  - Next state
    $= f_n$ (current state,
               current inputs)

- Control output signals
  $= f_o$ (current state)

Combinational control logic

Datapath control outputs

Outputs

Inputs

Inputs from cache datapath

State register

Next state

Cache Controller FSM

Idle

Cache Hit
Mark Cache Ready

Valid CPU request

Compare Tag
If Valid && Hit ,
Set Valid, SetTag,
if Write Set Dirty

Could partition into separate states to reduce clock cycle time

Cache
Miss
and
Old Block
is Clean

Cache
Miss
and
Old Block
is Dirty

Memory Ready

Allocate
Read new block
from Memory

Memory
not
Ready

Memory Ready

Write-Back
Write Old
Block to
Memory

Memory
not
Ready

Cache Coherence Problem

- 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

85

Coherence Defined

- Informally: Reads return most recently written value

- Formally:

  - P writes X; P reads X (no intervening writes)
    $\Rightarrow$ read returns written value

  - P_1 writes X; P_2 reads X (sufficiently later)
    $\Rightarrow$ read returns written value

    - c.f. CPU B reading X after step 3 in example

  - P_1 writes X, P_2 writes X
    $\Rightarrow$ all processors see writes in the same order

    - End up with the same final value for X

Cache Coherence Protocols

- Operations performed by caches in multiprocessors to ensure coherence
  - Migration of data to local caches
    - Reduces bandwidth for shared memory
  - Replication of read-shared data
    - Reduces contention for access
- Snooping protocols
  - Each cache monitors bus reads/writes
- Directory-based protocols
  - Caches and memory record sharing status of blocks in a directory

Invalidating Snooping Protocols

- Cache gets exclusive access to a block when it is to be written
  - Broadcasts an invalidate message on the bus
  - Subsequent read in another cache misses
    - Owning cache supplies updated value

| CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidate for X	1		0
CPU B read X	Cache miss for X	1	1	1

Memory Consistency

- When are writes seen by other processors
  - "Seen" means a read returns the written value
  - Can't be instantaneously
- Assumptions
  - A write completes only when all processors have seen it
  - A processor does not reorder writes with other accesses
- Consequence
  - P writes X then writes Y
    $\Rightarrow$ all processors that see new Y also see new X
  - Processors can reorder reads, but not writes

Multilevel On-Chip Caches

Intel Nehalem 4-core processor
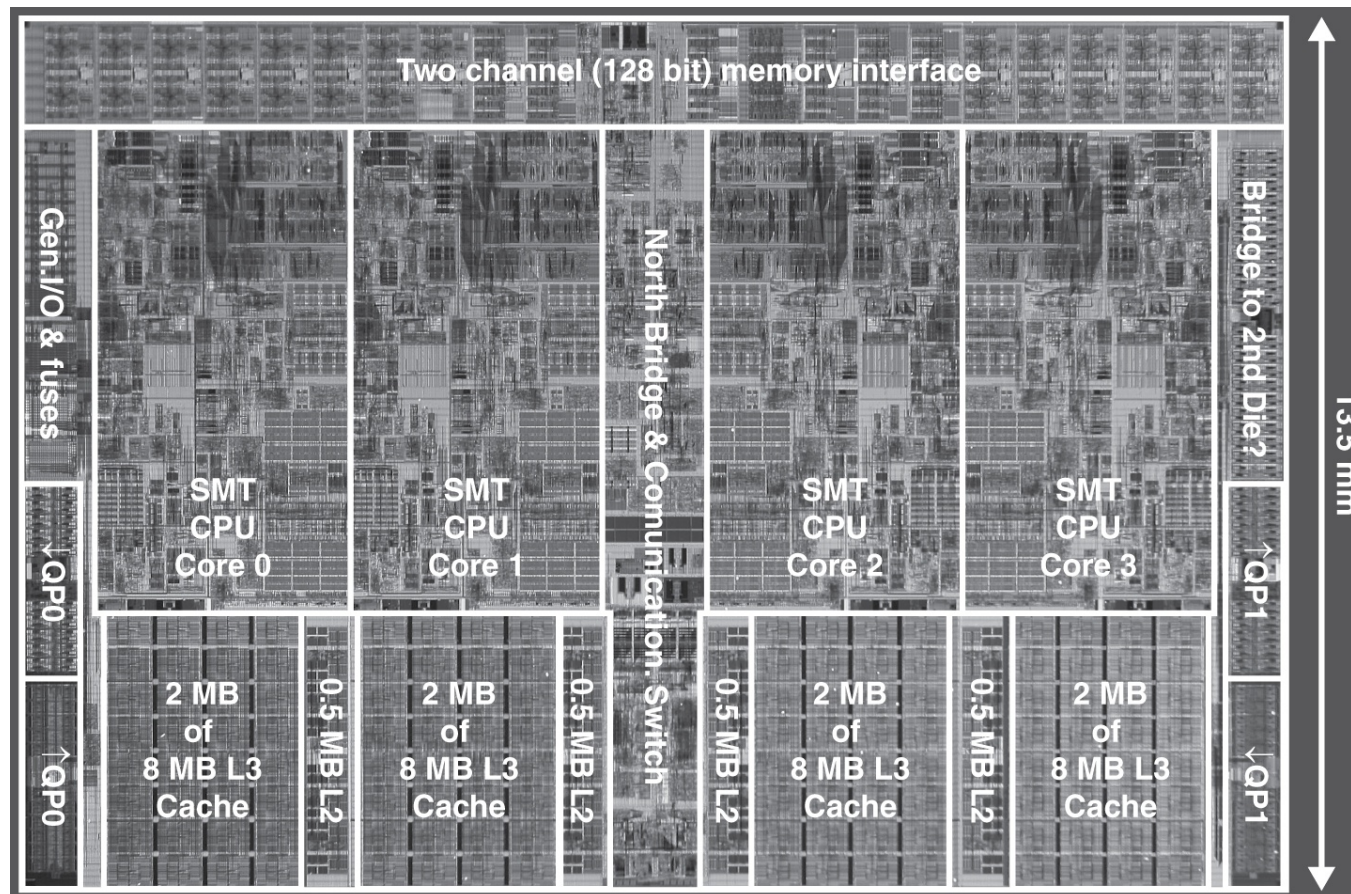
Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB L2 cache

2-Level TLB Organization

	Intel Nehalem	AMD Opteron X4
Virtual addr	48 bits	48 bits
Physical addr	44 bits	48 bits
Page size	4KB, 2/4MB	4KB, 2/4MB
L1 TLB (per core)	L1 I-TLB: 128 entries for small pages, 7 per thread (2×) for large pages L1 D-TLB: 64 entries for small pages, 32 for large pages Both 4-way, LRU replacement	L1 I-TLB: 48 entries L1 D-TLB: 48 entries Both fully associative, LRU replacement
L2 TLB (per core)	Single L2 TLB: 512 entries 4-way, LRU replacement	L2 I-TLB: 512 entries L2 D-TLB: 512 entries Both 4-way, round-robin LRU
TLB misses	Handled in hardware	Handled in hardware

3-Level Cache Organization

	Intel Nehalem	AMD Opteron X4
L1 caches (per core)	L1 I-cache: 32KB, 64-byte blocks, 4-way, approx LRU replacement, hit time n/a L1 D-cache: 32KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a	L1 I-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, hit time 3 cycles L1 D-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, write-back/allocate, hit time 9 cycles
L2 unified cache (per core)	256KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a	512KB, 64-byte blocks, 16-way, approx LRU replacement, write-back/allocate, hit time n/a
L3 unified cache (shared)	8MB, 64-byte blocks, 16-way, replacement n/a, write-back/allocate, hit time n/a	2MB, 64-byte blocks, 32-way, replace block shared by fewest cores, write-back/allocate, hit time 32 cycles

n/a: data not available

Miss Penalty Reduction

- ## Return requested word first
  - ### Then back-fill rest of block

- ## Non-blocking miss processing
  - ### Hit under miss: allow hits to proceed
  - ### Miss under miss: allow multiple outstanding misses

- ## Hardware prefetch: instructions and data

- ## Opteron X4: bank interleaved L1 D-cache
  - ### Two concurrent accesses per cycle

Pitfalls

- Byte vs. word addressing

  - Example: 32-byte direct-mapped cache, 4-byte blocks

    - Byte 36 maps to block 1
    - Word 36 maps to block 4

- Ignoring memory system effects when writing or generating code

  - Example: iterating over rows vs. columns of arrays

  - Large strides result in poor locality

Pitfalls

- In multiprocessor with shared L2 or L3 cache
  - Less associativity than cores results in conflict misses
  - More cores $\Rightarrow$ need to increase associativity

- Using AMAT to evaluate performance of out-of-order processors
  - Ignores effect of non-blocked accesses
  - Instead, evaluate performance by simulation

Pitfalls

- Extending address range using segments
  - E.g., Intel 80286
  - But a segment is not always big enough
  - Makes address arithmetic complicated

- Implementing a VMM on an ISA not designed for virtualization
  - E.g., non-privileged instructions accessing hardware resources
  - Either extend ISA, or require guest OS not to use problematic instructions

Concluding Remarks

- Fast memories are small, large memories are slow
  - We really want fast, 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