CSCI-2500: Computer Organization

Memory Hierarchy (Chapter 5)

Memory Technologies: Speed vs. Cost (1997)

Technology	Access Time	Cost: $/Mbyte
SRAM	5-25ns	$100-$250
DRAM	60-120ns	$5-$10
Mag. disk	10-20 million ns	$0.1-$0.2

Access Time: the length of time it takes to get a value from memory, given an address.

Memory Technologies: Speed vs. Cost (2004)

Technology	Access Time	Cost: $/Gbyte
SRAM	0.5-5ns	$4000-$10K (25x)
DRAM	50-70ns	$100-$200 (50x)
Mag. disk	5-20 million ns	$0.50-$2.00 (12x)

Observe: access time not changing much over the last 7 years, but unit cost per capacity has changed dramatically

Performance and Memory

- SRAM is fast, but too expensive (we want large memories!).
- Using only SRAM (enough of it) would mean that the memory ends up costing more than everything else combined!

Caching

- The idea is to use a small amount of fast memory *near* the processor (in a cache).
- The cache hold frequently needed memory locations.
  - when an instruction references a memory location, we want that value to be in the cache!

Principles of Locality

Temporal: if a memory location is referenced, it is likely that it will be referenced again in the near future.

time

Spatial: if a memory location is referenced, it is likely that nearby items will be referenced in the near future.

space

Programs and Locality

Programs tend to exhibit a great deal of *locality* in memory accesses.

- array, structure/record access
- subroutines (instructions are near each other)
- local variables (counters, pointers, etc) are often referenced many times.

Memory Hierarchy

The general idea is to build a hierarchy:

- at the top is a small, fast memory that is *close* to the processor.
- in the middle are larger, slower memories.
- At the bottom is massive memory with very slow access time.

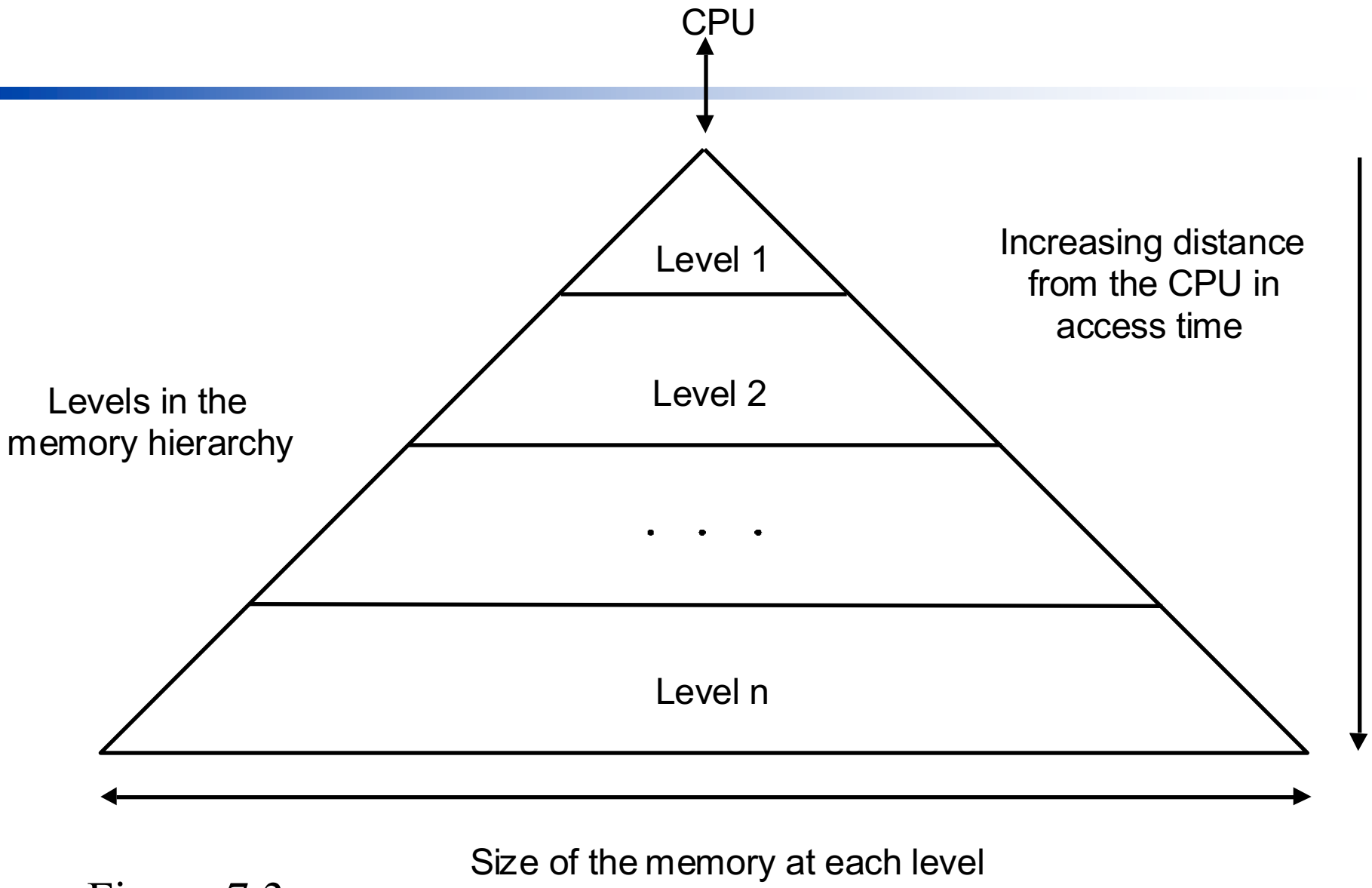

CPU
Level 1
Level 2
Level n
Increasing distance from the CPU in access time
Levels in the memory hierarchy
Size of the memory at each level

Figure 7.3

Cache and Main Memory

- For now we will focus on a 2 level hierarchy:
  - cache (small, fast memory directly connected to the processor).
  - main memory (large, slow memory at level 2 in the hierarchy).

Memory Hierarchy and Data Transfer

Transfer of data is done between adjacent levels in the hierarchy only!

All access by the processor is to the topmost level.

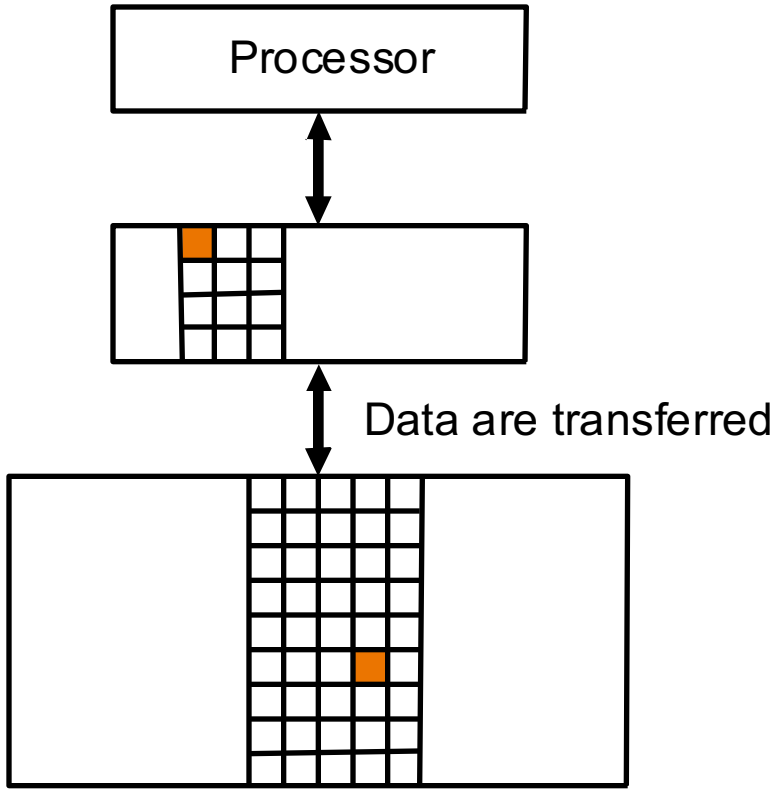


Figure 7.2

Terminology

- *hit*: when the memory location accessed by the processor is in the cache (upper level).
- *miss*: when the memory location accessed by the process is not in the cache.
- *block:* the minimum unit of information transferred between the cache and the main memory. Typically measured in bytes or words.

Terminology (cont.)

- *hit rate*: the ratio of *hits* to total memory accesses.
- *miss rate*: 1 – *hit rate*
- *hit time*: the time to access an element that is in the cache:
  - time to find out if it's in the cache.
  - time to transfer from cache to processor.

Terminology (cont.)

- *miss penalty*: the time to replace a block in the cache with a block from main memory and to deliver deliver the element to the processor.

- *hit time* is small compared to *miss penalty* (otherwise we wouldn't bother with a memory hierarchy!)

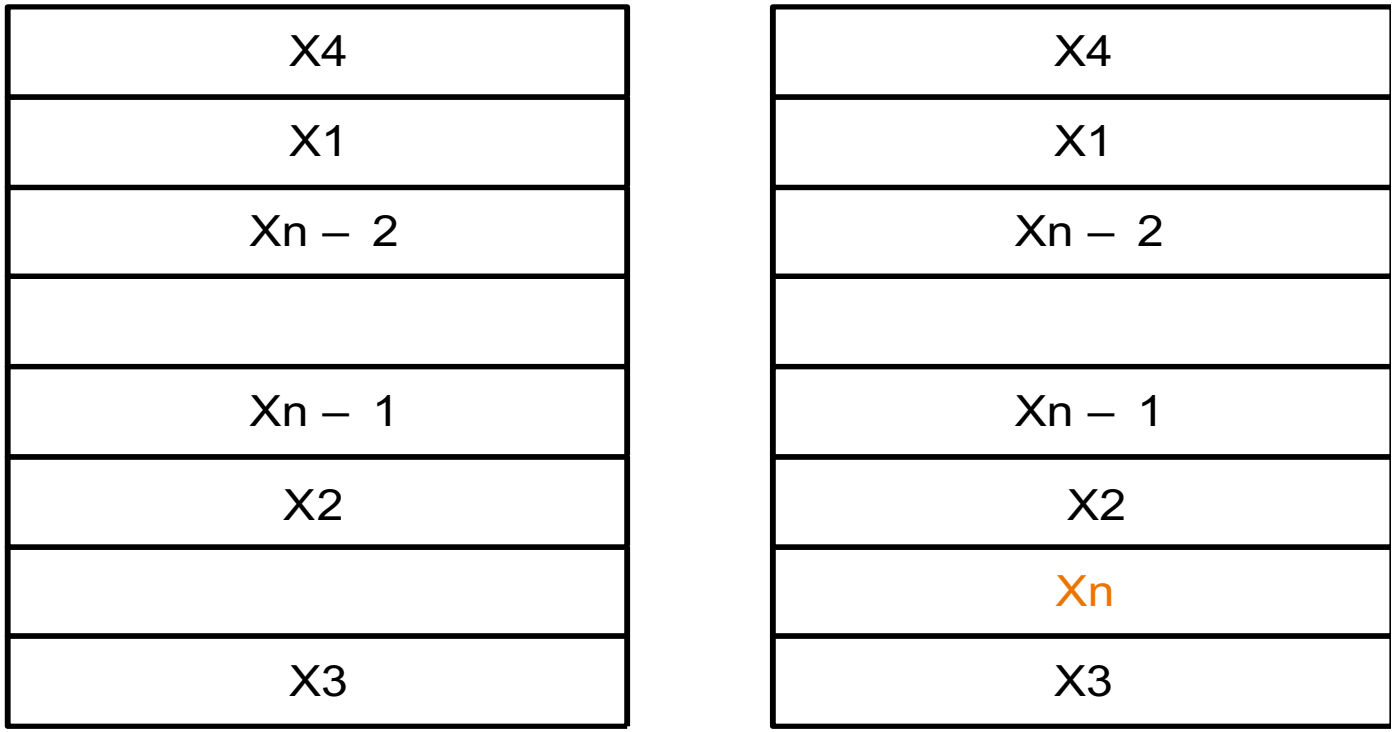
Simple Cache Model

- Assume that the processor accesses memory one word at a time.
- A *block* consists of one word.
- When a word is referenced and is not in the cache, it is put in the cache (copied from main memory).

Cache Usage

- At some point in time the cache holds memory items $X_1, X_2, \ldots X_{n-1}$
- The processor next accesses memory item X_n which is not in the cache.

Cache before and after



a. Before the reference to Xn

b. After the reference to Xn

Issues

- How do we know if an item is in the cache?
- If it is in the cache, how do we know *where* it is?

Direct-Mapped Cache

- Each memory location is *mapped* to a single location in the cache.
  - there in only one place it can be!

- Remember that the cache is smaller than memory, so many memory locations will be *mapped* to the same location in the cache.

Mapping Function

- The simplest mapping is based on the LS bits of the address.
- For example, all memory locations whose address ends in 000 will be mapped to the same location in the cache.
- The requires a cache size of 2^n locations (a power of 2).

A Direct Mapped Cache

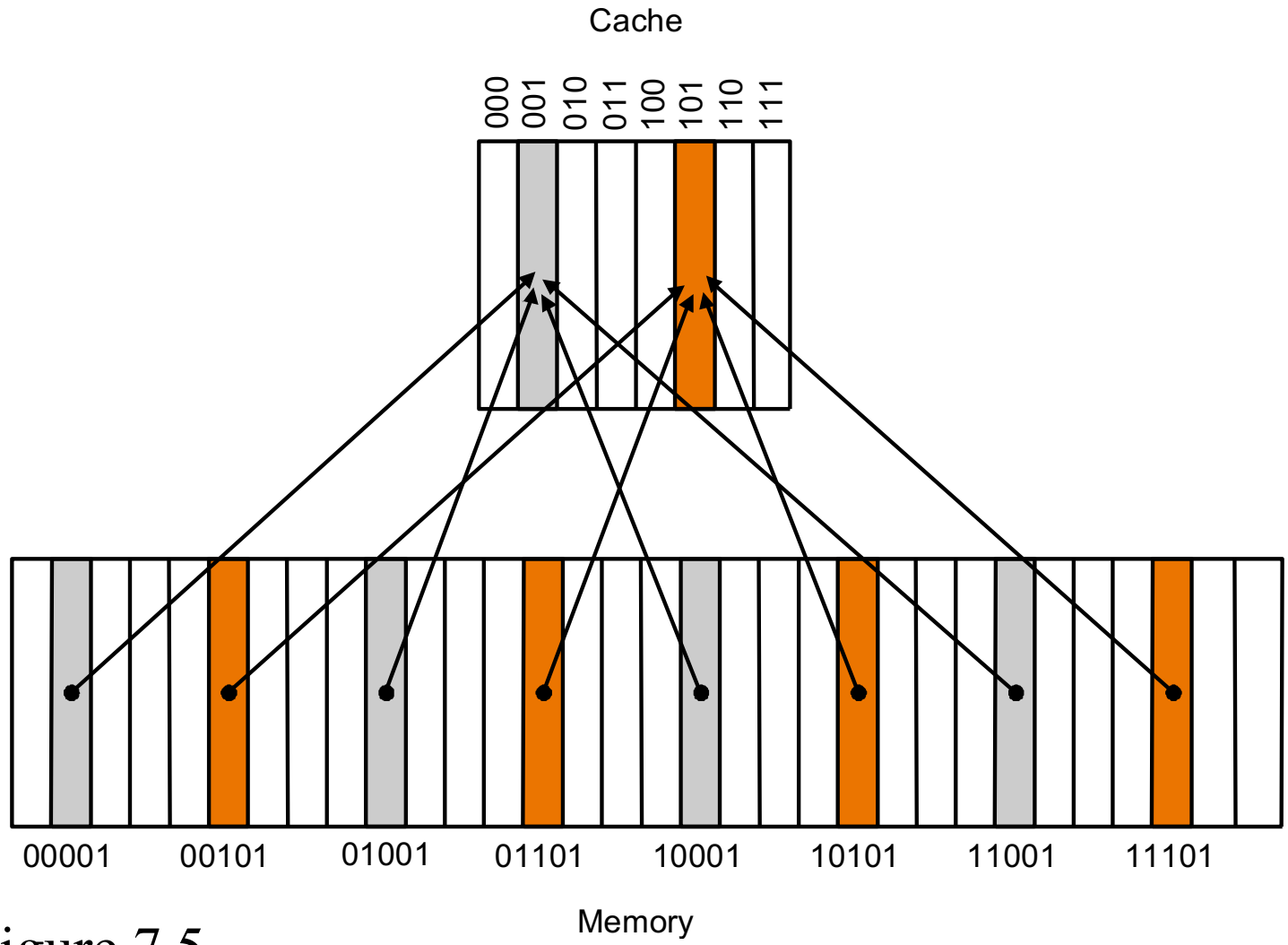


Figure 7.5

Who's in *slot* 000?

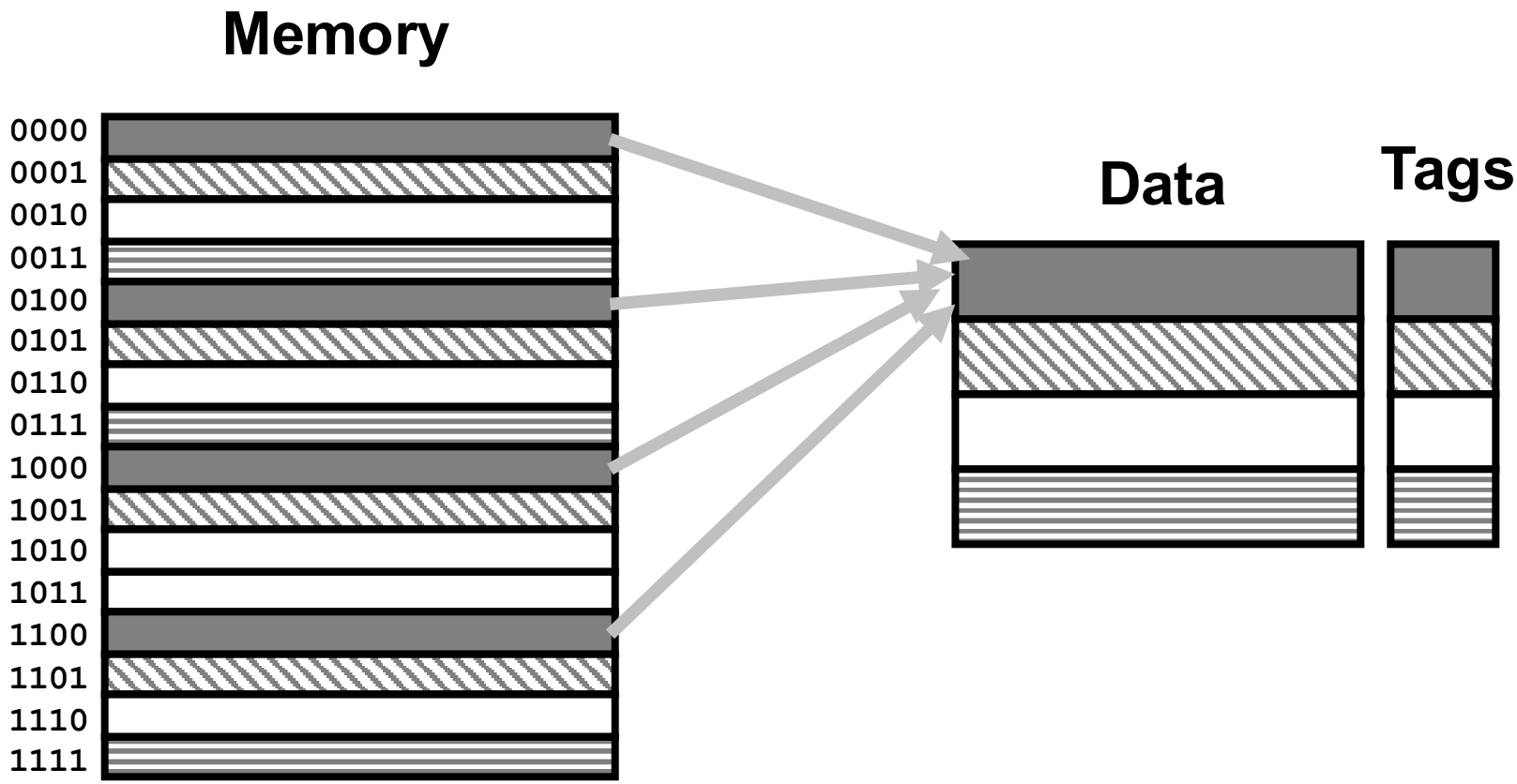
- We still need a way to find out which of the many possible memory elements is currently in a cache *slot*.
  - *slot*: a location in the cache that can hold a block.
- We need to store the address of the item currently using cache slot 000.

Tags

- We don't need to store the entire memory location address, just those bits that are not used to determine the slot number (the *mapping*).
- We call these bits the *tag*.

- The *tag* associated with a cache slot tells who is currently using the slot.

16 word memory, 4 word cache



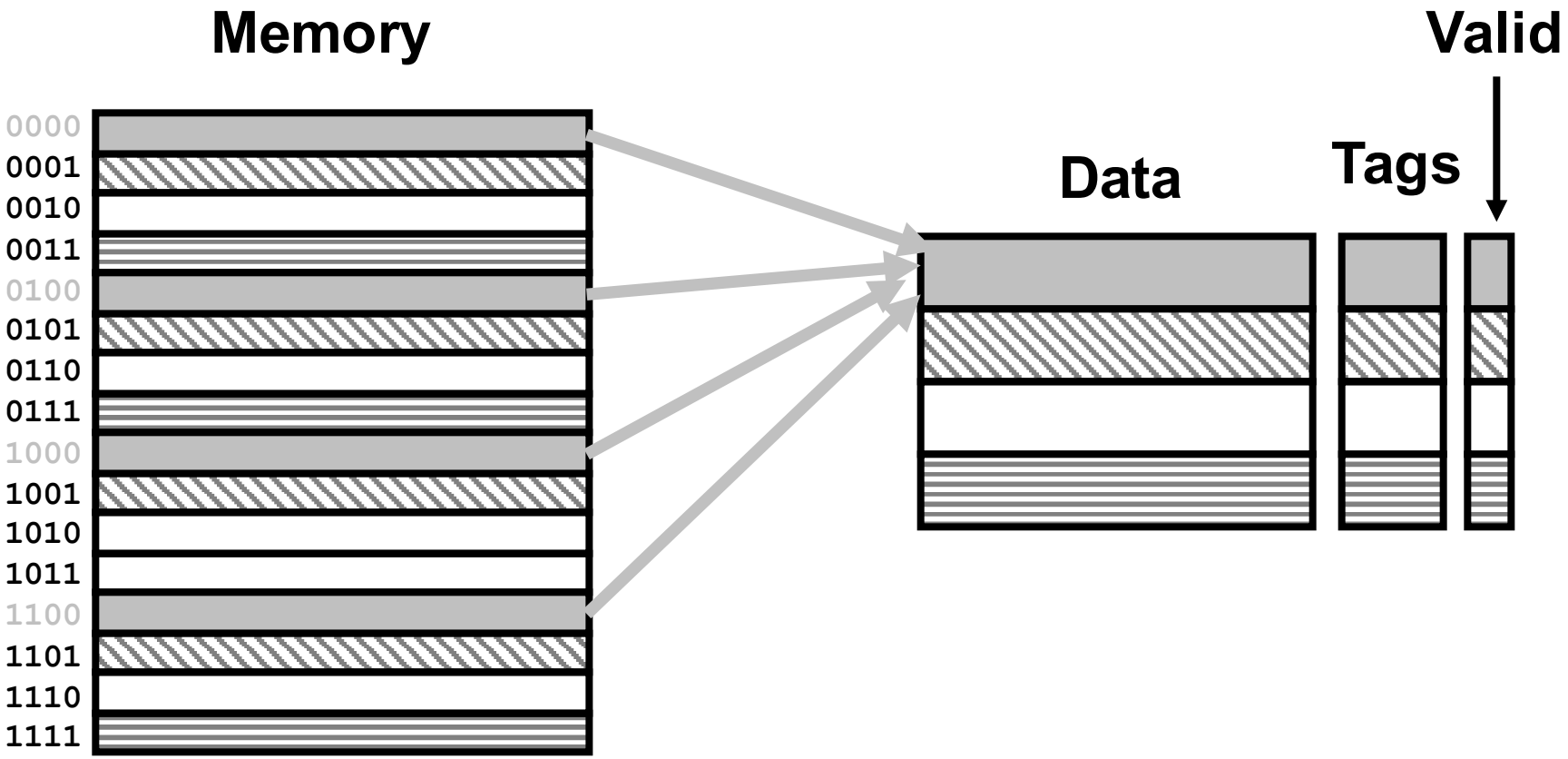
Initialization Problem

- Initially the cache is empty.
  - all the bits in the cache (including the tags) will have random values.
- After some number of accesses, some of the tags are *real* and some are still just random junk.
- How do we know which cache slots are *junk* and which really mean something?

Valid Bits

- Include one more bit with each cache slot that indicates whether the tag is valid or not.
- Provide hardware to initialize these bits to 0 (one bit per cache slot).
- When checking a cache slot for a specific memory location, ignore the tag if the valid bit is 0.
- Change a slot's valid bit to a 1 when putting something in the slot (from main memory).

Revised Cache



Simple Simulation

- We can simulate the operation of our simple direct-mapped cache by listing a sequence of memory locations that are referenced.

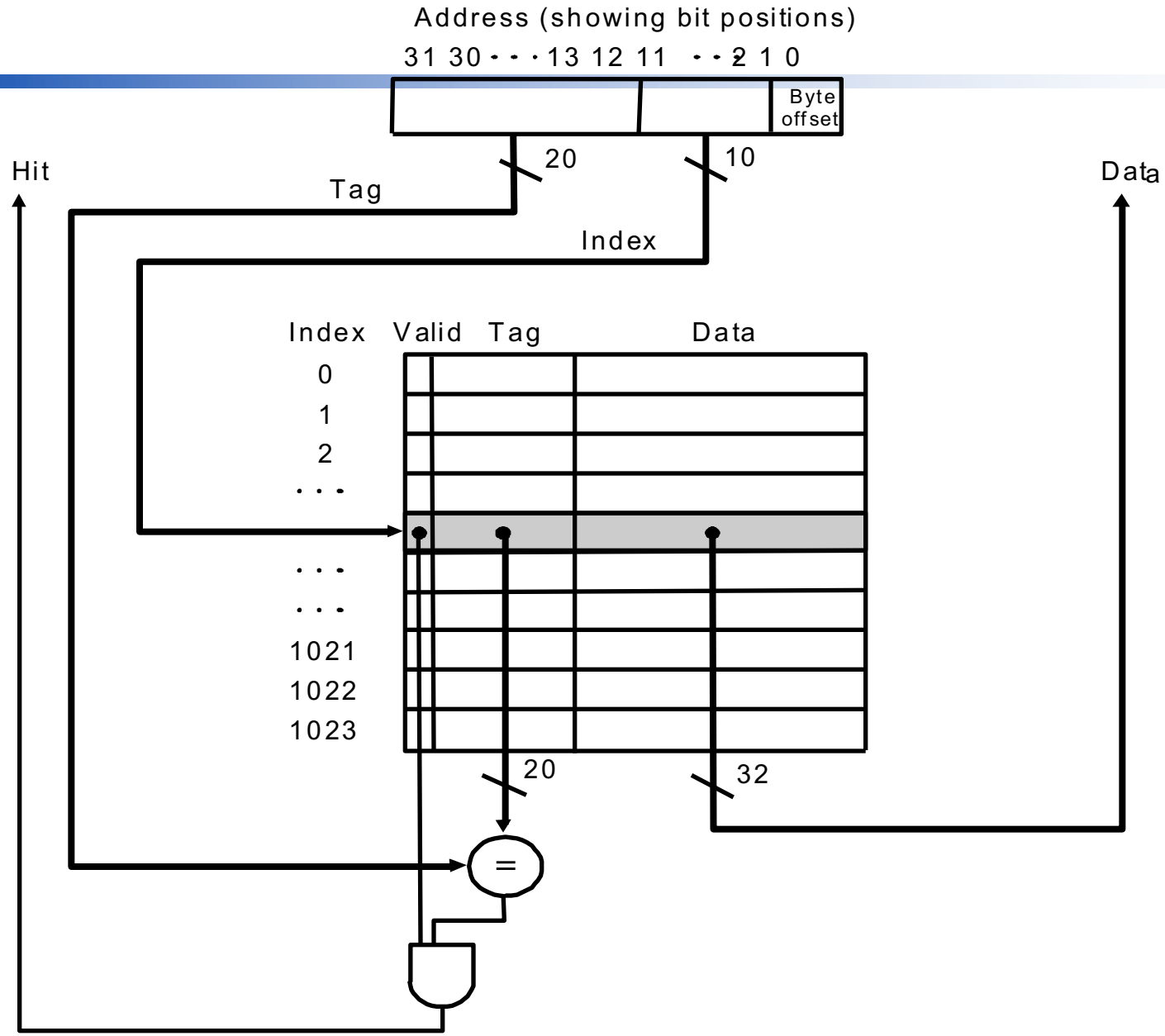
- Assume the cache is initialized with all the valid bits set to 0 (to indicate all the slots are empty).

Memory Access Sequence

Address	Binary Address	Slot	hit or miss
3	0011	11 (3)	*miss*
8	1000	00 (0)	*miss*
3	0011	11 (3)	*hit*
2	0010	10 (2)	*miss*
4	0100	00 (0)	*miss*
8	1000	00 (0)	*miss*

Hardware

- We need to have hardware that can perform all the operations:
  - find the right slot given an address (perform the *mapping*).
  - check the valid bit.
  - compare the tag to part of the address


Address (showing bit positions)
31 30 · · · 13 12 11 · · · 2 1 0
Byte offset
20
10
Hit
Tag
Index
Data
Index
Valid
Tag
Data
0
1
2
· · ·
· · ·
· · ·
1021
1022
1023
20
32
=

Possible Test Question

Given the following:

- 32 bit addresses (2^{32} byte memory, 2^{30} words)
- 64 KB cache (16 K words). Each slots holds 1 word.
- Direct Mapped Cache.

- How many bits are needed for each tag?
- How many memory locations are mapped to the same cache slot?
- How many total bits in the cache (data + tag + valid).

Possible Test Answer

- Memory has 2^{30} words
- Cache has 16K = 2^{14} slots (words).
- Each cache slot can hold any one of 2^{30} / 2^{14} = 2^{16} memory locations, so the tag must be 16 bits.
- 2^{16} is 64K memory locations that map to the same cache slot.
- Add one for the valid bit for each cache line.
- Total memory in bits = 2^{14} x (32+16+1) = 49 x 16K = 784 Kbits (98 Kbytes!)

Handling a Cache Miss

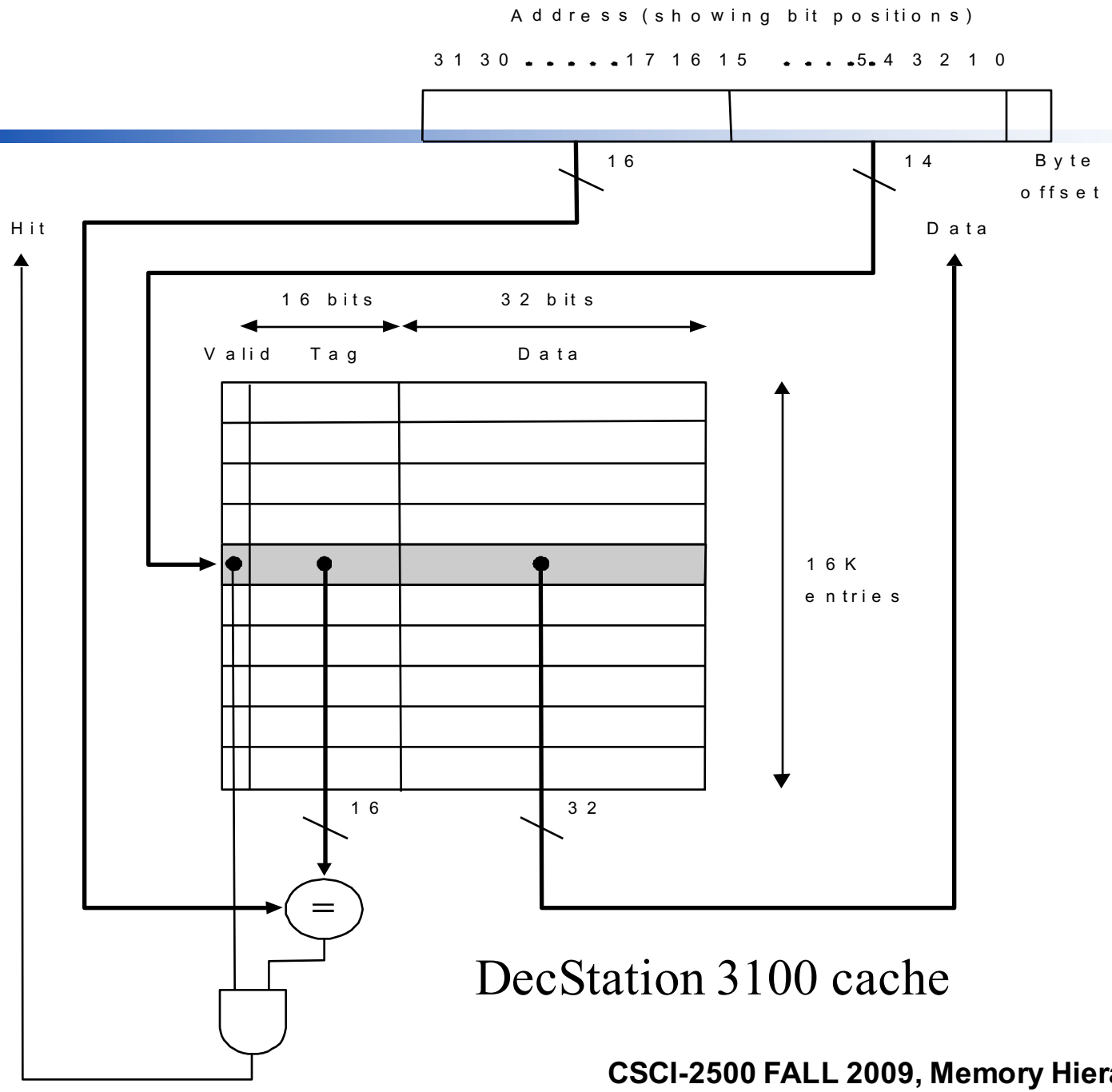
- A miss means the processor must wait until the memory requested is in the cache.
  - a separate controller handles transferring data between the cache and memory.
- In general the processor continuously tries the fetch until it works (until it's a hit).
  - continuously means "once per cycle".
  - in the meantime the pipeline is stalled!

Data vs. Instruction Cache

- Obviously nothing other than a stall can happen if we get a miss when fetching the next instruction!
- It is possible to execute other instructions while waiting for data (need to detect data hazards), this is called *stall on use*.
  - the pipeline stalls only when there are no instructions that can execute without the data.

DecStation 3100 Cache

- Simple Cache implementation
  - 64 KB cache (16K words).
  - 16 bit tags
  - Direct Mapped
  - Two caches, one for instructions and the other for data.



DecStation 3100 cache

Handling Writes

- What happens when a store instruction is executed?
  - what if it's a hit?
  - what if it's a miss?
- DecStation 3100 does the following:
  - don't bother checking the cache, just write the new value in to the cache!
  - Also write the word to main memory (called *write-through*).

Write-Through

- Always updating main memory on each store instruction can slow things down!
  - the memory is tied up for a while.
- It is possible to set up a *write buffer* that holds a number of pending writes.
- If we also update the cache, it is not likely that we need to worry about getting a memory value from the buffer (but it's possible!)

Write-back

- Another scheme for handling writes:
  - only update the cache.
  - when the memory location is booted out of the cache (someone else is being put in to the same slot), write the value to memory.

Cache Performance

For the simple DecStation 3100 cache:

	Miss Rate		
Program	Instruction	Data	Combined
gcc	6.1%	2.1%	5.4%
spice	1.2%	1.3%	1.2%

Spatial Locality?

- So far we've only dealt with temporal locality (it we access an item, it is likely we will access it again soon).
- What about space (the final frontier)?
  - In general we make a *block* hold more than a single word.
  - Whenever we move *data* to the cache, we also move it's neighbors.

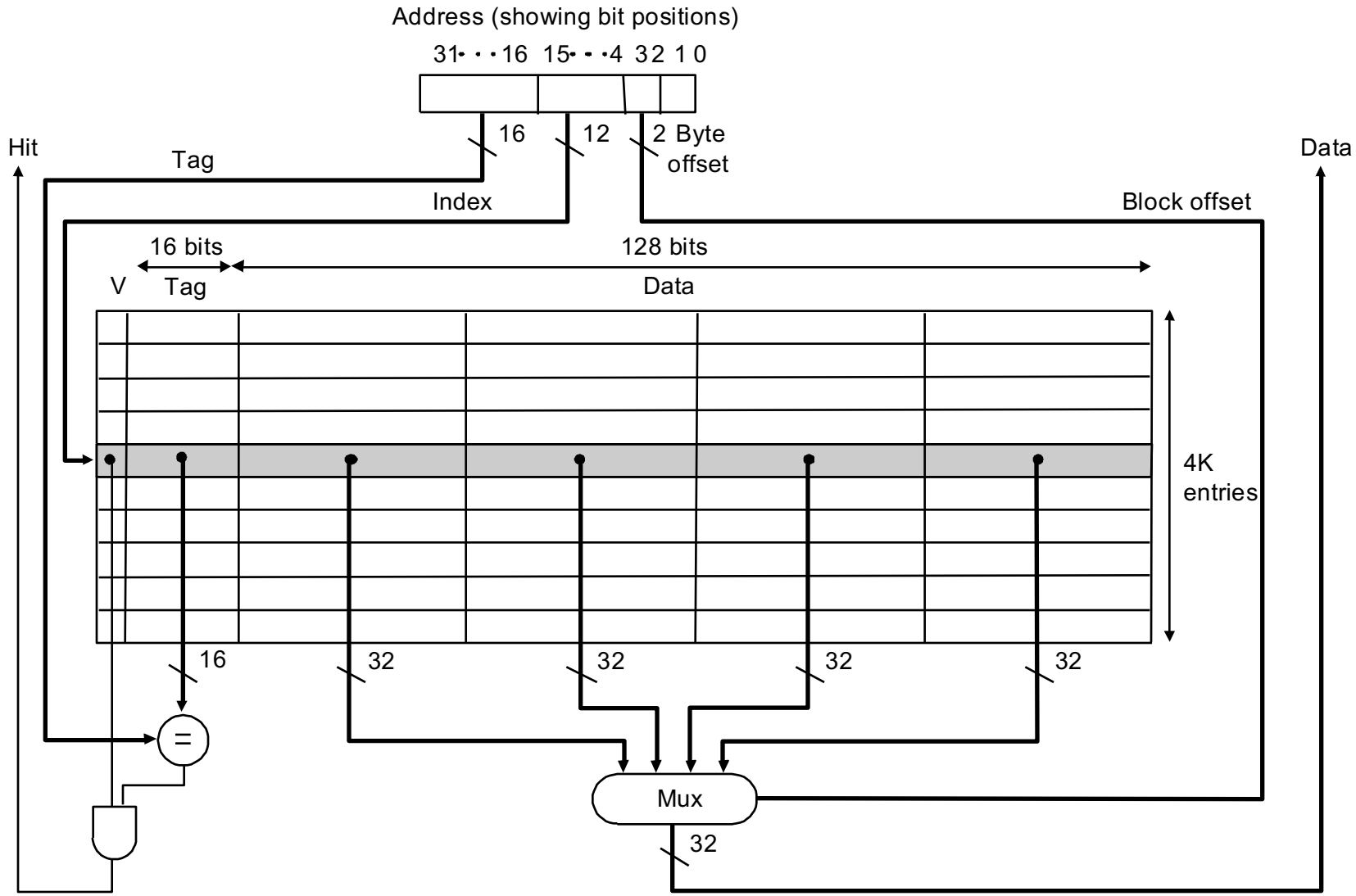
Blocks and Slots

- Each cache slot holds one block.
- Given a fixed cache size (number of bytes) as the block size increases, the number of slots must decrease.
- Reducing the number of slots in the cache increases the number of memory locations that compete for the same slot.

Example multi-word block cache

- 4 words/block
  - we now use a *block address* to determine the slot mapping.
  - the block address in this case is the address/4.
  - on a hit we need to extract a single word (need a multiplexor controlled by the LS 2 address bits).
- 64KB data
  - 16 Bytes/block
  - 4K slots.

Example multi-word block cache



Performance and Block Size

DecStation 3100 cache with block sizes 1 and 4 (words).

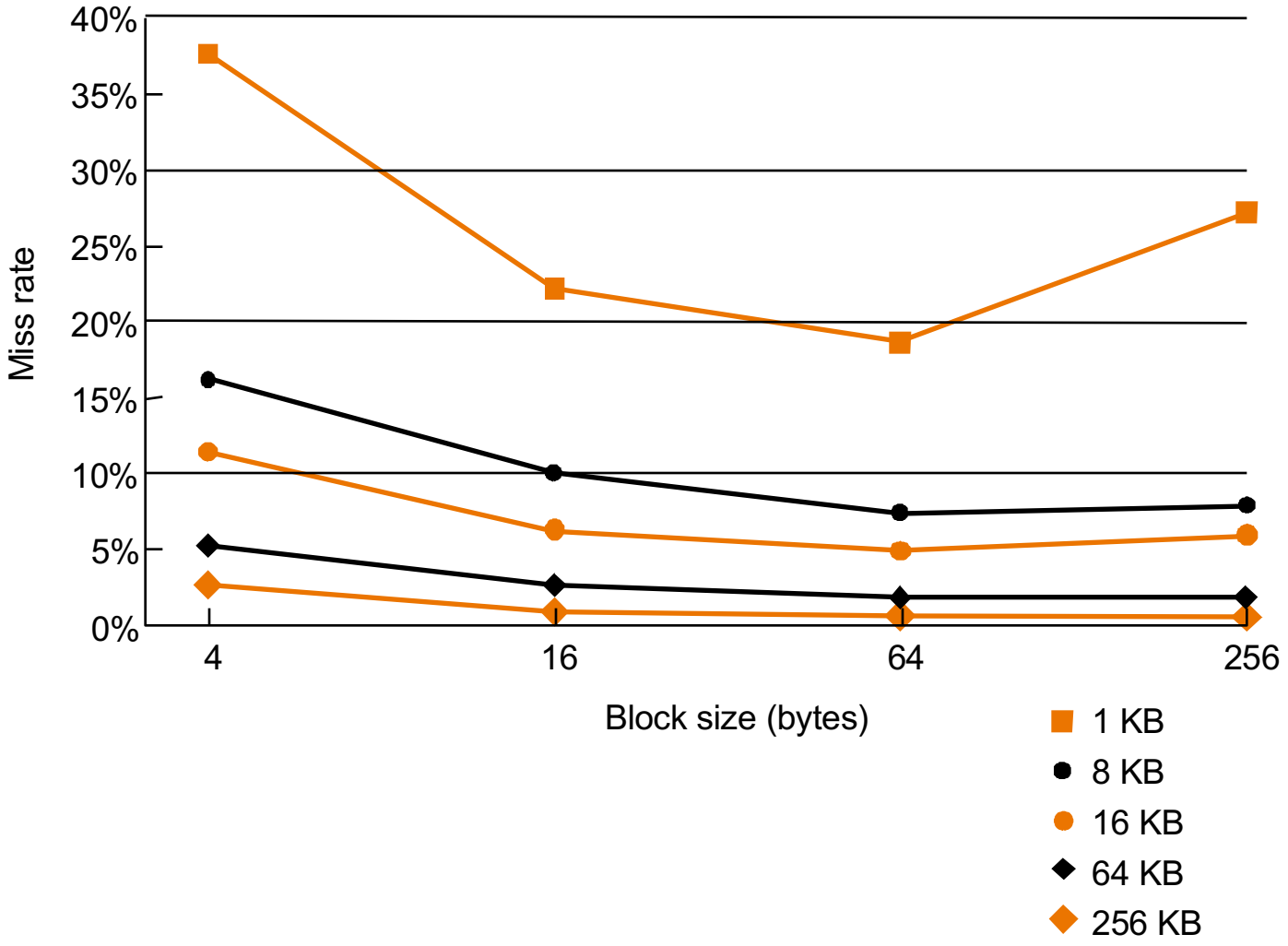
Program	Block Size	Miss Rate Instruction	Data	Combined
gcc	1	6.1%	2.1%	5.4%
gcc	4	2.0%	1.7%	1.9%
spice	1	1.2%	1.3%	1.2%
spice	4	0.3%	0.6%	0.4%

Is bigger always better?

- Eventually increasing the block size will mean that the competition for cache slots is too high
  - miss rate will increase.

- Consider the extreme case: the entire cache is a single block!

Miss rate vs. Block Size



Block Size and Miss Time

- As the block size increases, we need to worry about what happens to the miss time.
- The larger a block is, the longer it takes to transfer from main memory to cache.

- It is possible to design memory systems with transfer of an entire block at a time, but only for relatively small block sizes (4 words).

Example Timings

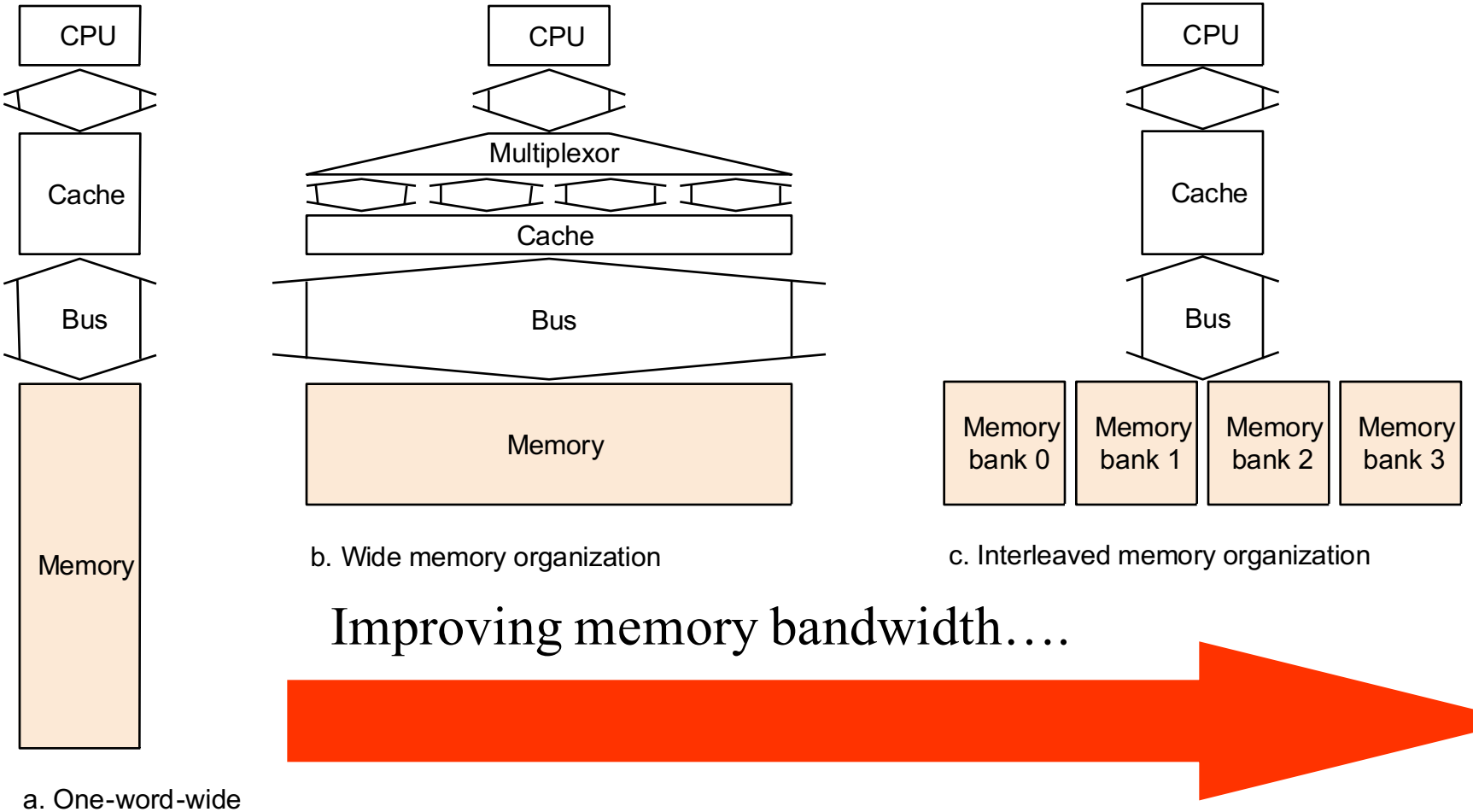
Hypothetical access times:

- 1 cycle to send the address
- 15 cycles to initiate each access
- 1 cycle to transfer each word.

- Miss penalty for 4-word wide memory is:

$$1 + 4x15 + 4x1 = 65 \text{ cycles.}$$

Memory Organization Options



Improving Cache Performance

- Cache performance is based on two factors:
  - miss rate
    - depends on both the hardware and on the program being measured (miss rate can vary).
  - miss penalty
    - the penalty is dictated by the hardware (the organization of memory and memory access times).

Cache and CPU Performance

The total number of cycles it takes for a program is the sum of:

- number of *normal* instruction execution cycles.
- number of cycles stalled waiting for memory.

$$Memory-stall\ cycles = \frac{Memory\ Accesses}{Pr\,ogram} \times Miss\ rate \times Miss\ penalty$$

Cache Calculations

How much faster would this program run with a perfect cache?:

CPI (without memory stalls): 2

Miss Rate: 5%

Miss Penalty: 40 cycles

% of instructions that are load/store: 30%

Speedup Calc

$Time_{perfect}$=IC * 2 (cpi) * cycle time

= IC * 2.0

$Time_{cache}$=IC*(0.3*(2+0.05*40) + 0.7*2)

= IC * 2.6

Speedup: 2.6/2 = 1.3 times faster with a perfect cache.

Clock Rate and Cache Performance

- If we double the clock rate of the processor, we don't change:
  - cache miss rate
  - miss penalty (memory is not likely to change!).
- The cache will not improve, so the speedup is not close to double!

Reducing Miss Rate

- Obviously a larger cache will reduce the miss rate!
- We can also reduce miss rate by reducing the *competition* for cache slots.
  - allow a block to be placed in one of many possible cache slots.

An extreme example of how to mess up a direct mapped cache.

- Assume that every 64th memory element maps to the same cache slot.

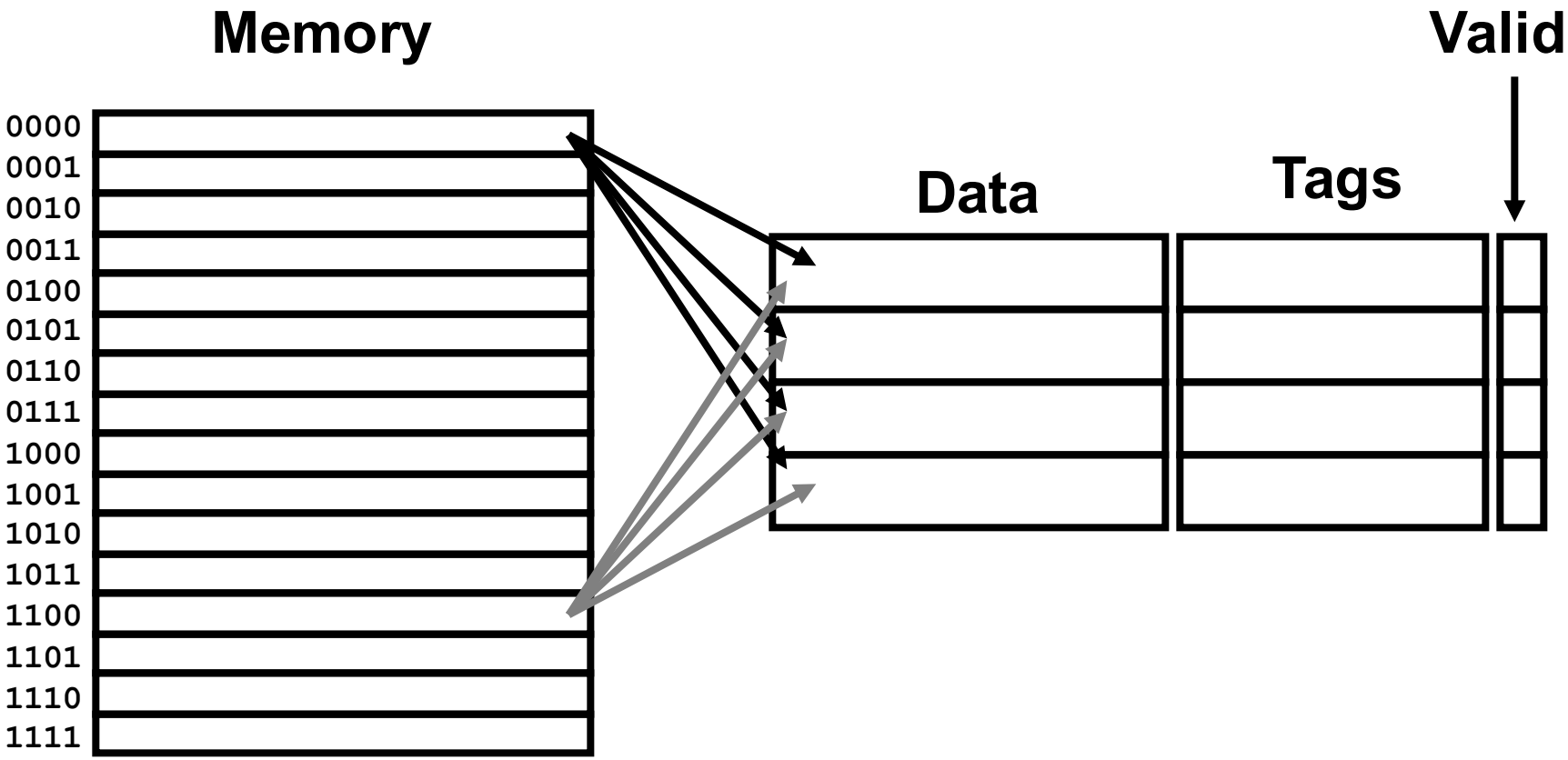
```
for (i=0;i<10000;i++) {
  a[i] = a[i] + a[i+64] + a[i+128];
  a[i+64] = a[i+64] + a[i+128];
}
```

`a[i]`, `a[i+64]` and `a[i+128]` use the same cache slot!

Fully Associative Cache

- Instead of direct mapped, we allow any memory block to be placed in *any* cache slot.
- It's harder to check for a hit (hit time will increase).
- Requires lots more hardware (a comparator for each cache slot).
- Each tag will be a complete block address.

Fully Associative Cache



Tradeoffs

- Fully Associate is much more flexible, so the miss rate will be lower.
- Direct Mapped requires less hardware (cheaper).
  - will also be faster! i.e. better hit time!
- Tradeoff of miss rate vs. hit time.

Middle Ground

- We can also provide more flexibility without going to a fully associative *placement policy*.
- For each memory location, provide a small number of cache slots that can hold the memory element.
- This is much more flexible than direct-mapped, but requires less hardware than fully associative.

Set Associative

- A fixed number of locations where each block can be placed.
- *n-way set associative* means there are *n* places (slots) where each block can be placed.
- Chop up the cache in to a number of *sets* each set is of size *n*.

Block Placement Options
(memory block address 12)

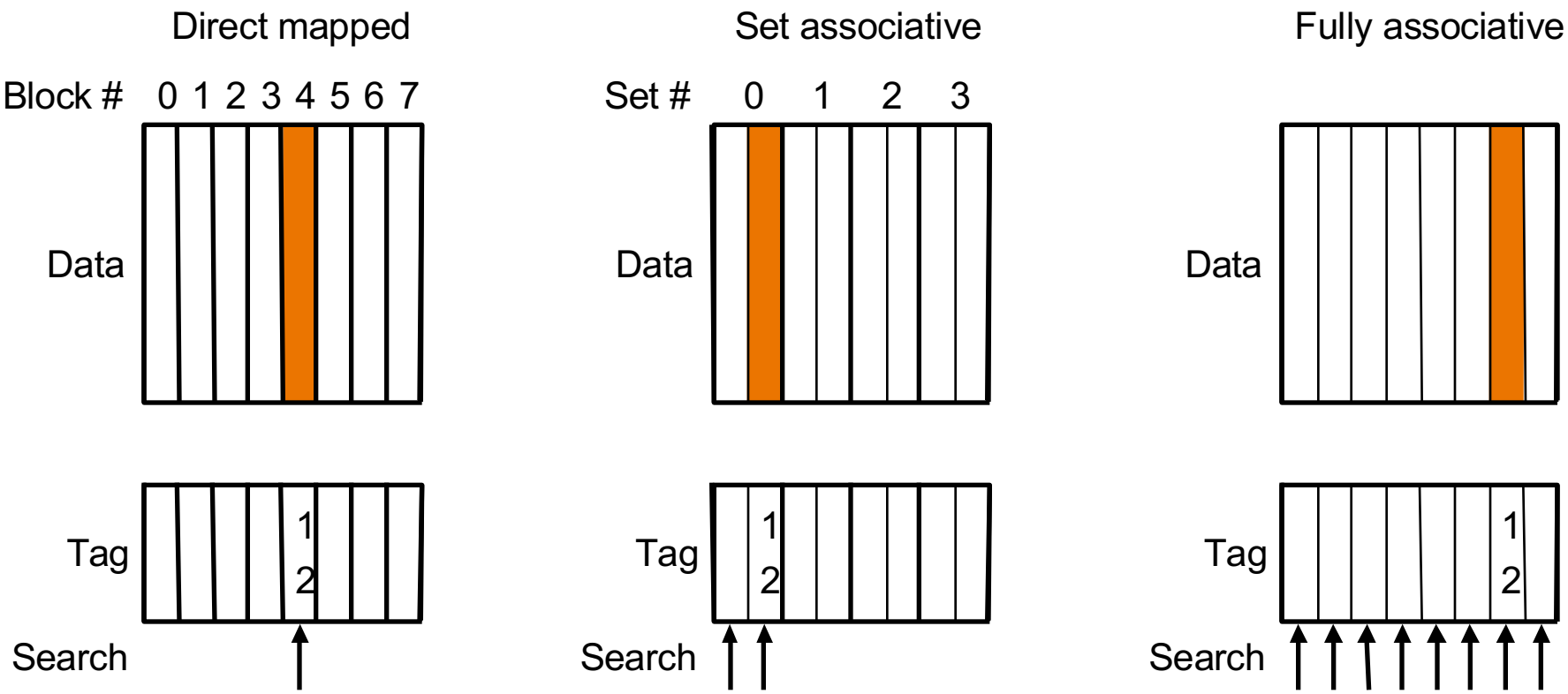
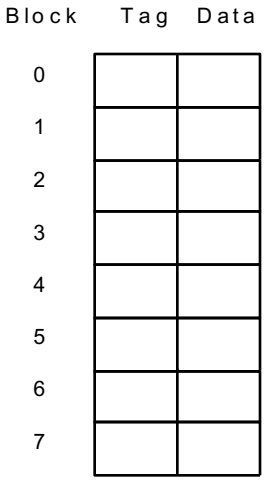


Figure 7.15

Possible 8-block Cache designs

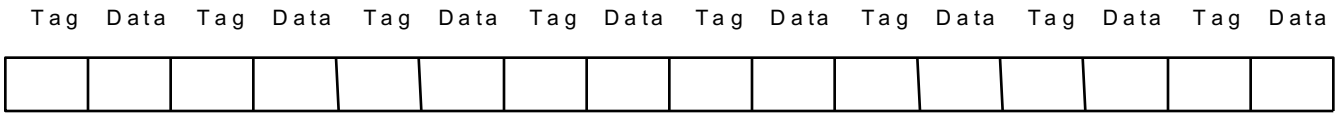
One-way set associative
(direct mapped)



Two-way set associative

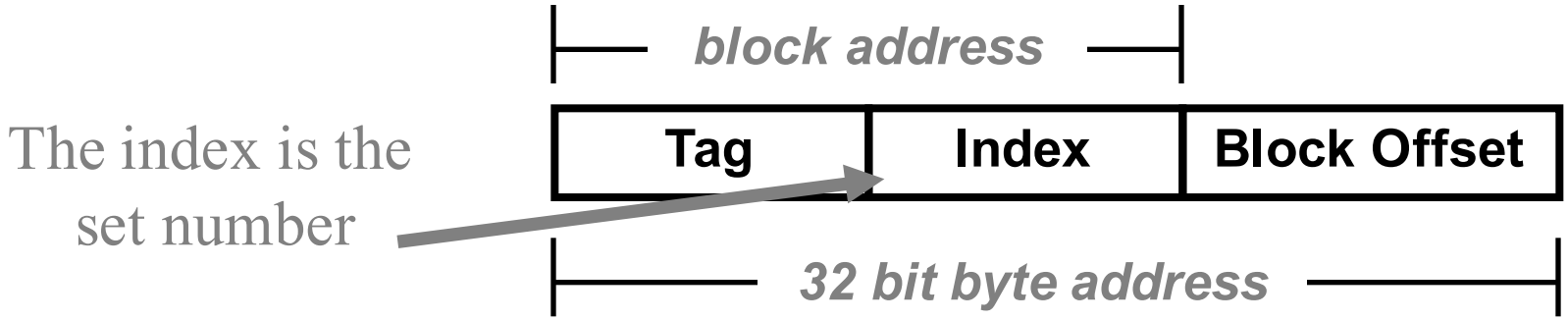
Four-way set associative

Eight-way set associative (fully associative)



Block Addresses & Set Associative Caching

- The LS bits of block address is used to determine which set the block can be placed in.
- The rest of the bits must be used for the tag.



Possible Test Question

- Block Size: 4 words
- Cache size (data only): 64 K Bytes
- 8-way set associative (each set has 8 slots).
- 32 bit address space (bytes).

- How many sets are there in the cache?
- How many memory blocks compete for placement in each set?

Answer

Cache size:

64 K Bytes is 2^{16} bytes

2^{16} bytes is 2^{14} words

2^{14} words is 2^{11} sets of 8 blocks each

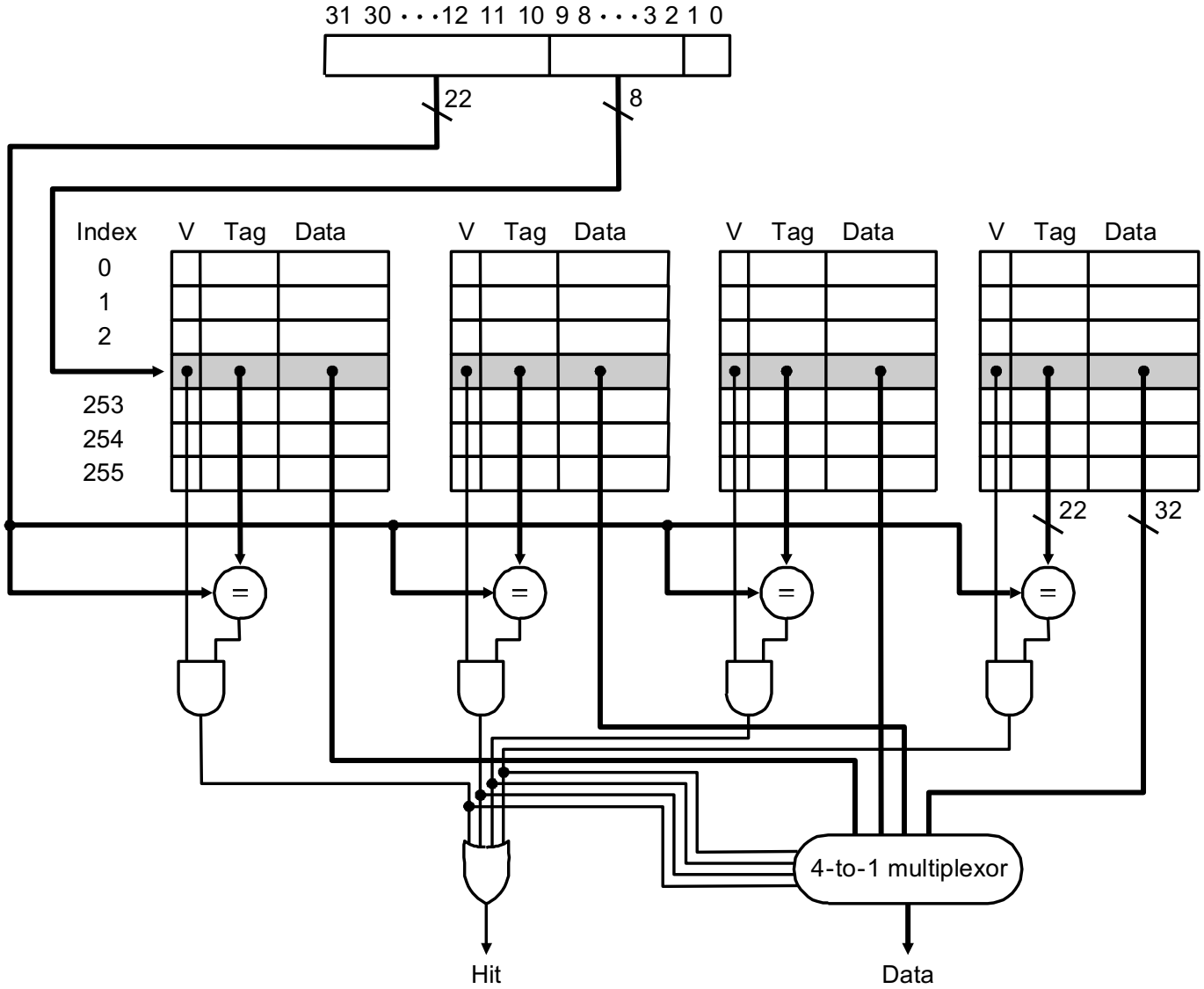
Memory Size:

2^{32} bytes = 2^{30} words = 2^{28} blocks

blocks per set:

$2^{28}/2^{11} = 2^{17}$ blocks per set

4-way Set Associative Cache



4-way set associative and the extreme example.

```
for (i=0;i<10000;i++) {
  a[i] = a[i] + a[i+64] + a[i+128];
  a[i+64] = a[i+64] + a[i+128];
}
```

`a[i]`, `a[i+64]` and `a[i+128]` belong to the same set – that's OK, we can hold all 3 in the cache at the same time.

Performance Comparison

Program	Associativity	Miss Rate		
		Instruction	Data	Combined
gcc	1 (direct)	2.0%	1.7%	1.9%
gcc	2	1.6%	1.4%	1.5%
gcc	4	1.6%	1.4%	1.5%
spice	1 (direct)	0.3%	0.6%	0.4%
spice	2	0.3%	0.6%	0.4%
spice	4	0.3%	0.6%	0.4%

DecStation 3100 cache with block size 4 words.

A note about set associativity

- Direct mapped is really just 1-way set associative (1 block per set).
- Fully associative is *n*-way set associative, where *n* is the number of blocks in the cache.

Question

Cache size 4K blocks.

block size 4 words (16 bytes)

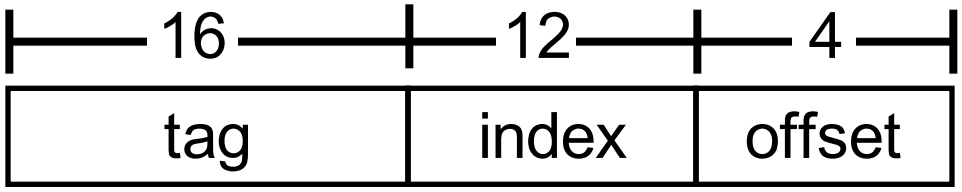
32 bit address

- How many bits for storing the tags (for the entire cache), if the cache is:
  - direct mapped
  - 2-way set associative
  - 4-way set associative
  - fully associative

Answer

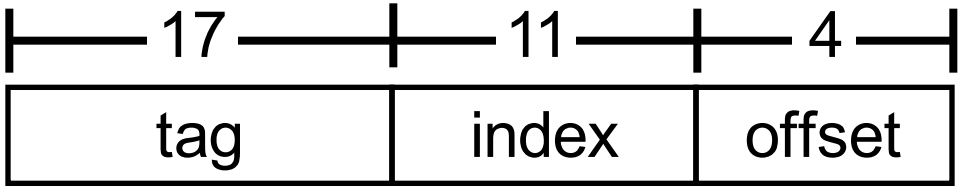
Direct Mapped:

16 * 4K = 64K bits



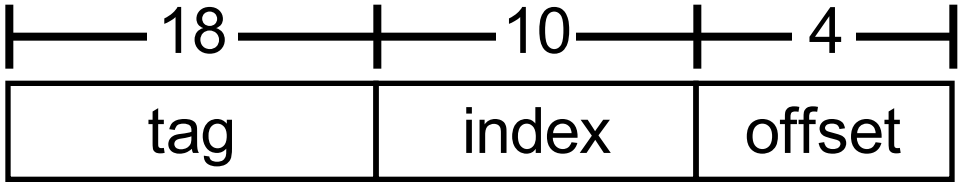
2-way:

17 * 4K = 68K bits



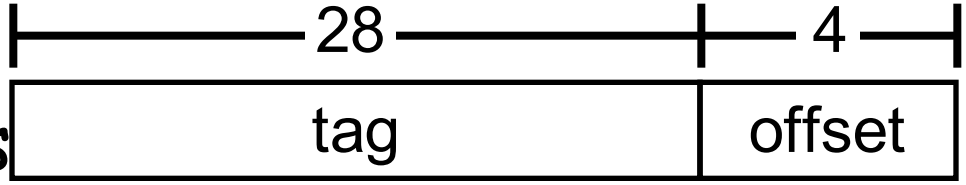
4-way:

18 * 4K = 72K bits



Fully Associative:

28 * 4K = 112K bits



Block Replacement Policy

- With a direct mapped cache there is no choice which memory element gets removed from the cache when a new element is moved to the cache.
- With a set associative cache, eventually we will need to remove an element from a *set*.

Replacement Policy: LRU

LRU: Least recently used.

- keep track of how *old* each block is (the blocks in the cache).
- When we need to put a new element in the cache, use the slot occupied by the oldest block.
- Every time a block in the cache is accessed (a hit), set the age to 0.
- Increase the age of all blocks in a set whenever a block in the set is accessed.

LRU in hardware

- We must implement this strategy in hardware!
- 2-way is easy, we need only 1 bit to keep track of which element in the set is older.
- 4-way is tougher (but possible).
- 8-way requires too much hardware (typically LRU is only approximated).

Multilevel Caches

- Most modern processors include an *on-chip cache* (the cache is part of the processor chip).
- The size of the on-chip cache is restricted by the size of the chip!
- Often, a secondary cache is used between the on-chip cache and the main memory.

Adding a secondary cache

- Typically use SRAM (fast, expensive). Miss penalty is much lower than for main memory.
- Using a fast secondary cache can change the design of the primary cache:
  - make the on-chip cache hit time as small as possible!

Performance Analysis

- Processor with CPI of 1 if all memory access handled by the on-chip cache.
- Clock rate 5 GHz (.2 ns period)
- Main memory access time 100ns
- Miss rate for primary cache is 2%

- How much faster if we add a secondary cache with 5ns access time that reduces the miss rate (to main memory) to 0.5%.

Analysis without secondary cache

Without the secondary cache the CPI will be based on:

- the CPI without memory stall (for all except misses)
- the CPI with a memory stall (just for cache misses).

- Without a stall the CPI is 1, and this happens 98% of the time.
- With a stall the CPI is 1 + miss penalty which is 100/.2 = 500 cycles. This happens 2% of the time.

CPI Calculation (no secondary cache)

Total CPI = Base CPI + Memory-Stall cycles per instruction

CPI = 1.0 + (2% * 500) = 11.0

With secondary cache

With secondary cache the CPI will be based on:

- the CPI without memory stall (for all except misses)
- the CPI with a stall for accessing the secondary cache (for cache misses that are resolved in the secondary cache).
- the CPI with a stall for accessing secondary cache and main memory (for accesses to main memory).

The stall for accessing secondary cache is 5/.2 = 25 cycles.

CPI Calculation (with secondary cache)

Total CPI = 1 + Primary stalls per instruction +
Secondary stalls per instruction

= 1 + (2% * 25) + (.5% * 500)
= 1 + 0.5 + 2.5
= 4.0

Processor w/ 2ndary Cache is 11/4 = 2.8x faster!

Virtual Memory

Disk caching

- Use main memory as a *cache* for magnetic disk.
- We can do this for a number of reasons:
  - speed up disk access
  - pretend we have more main memory than we really have.
  - support multiple programs easily (each can pretend it has all the memory).

Our focus

- We will focus on using the disk as a storage area for chunks of main memory that are not being used.

- The basic concepts are similar to providing a cache for main memory, although we now view part of the hard disk as *being the memory*.

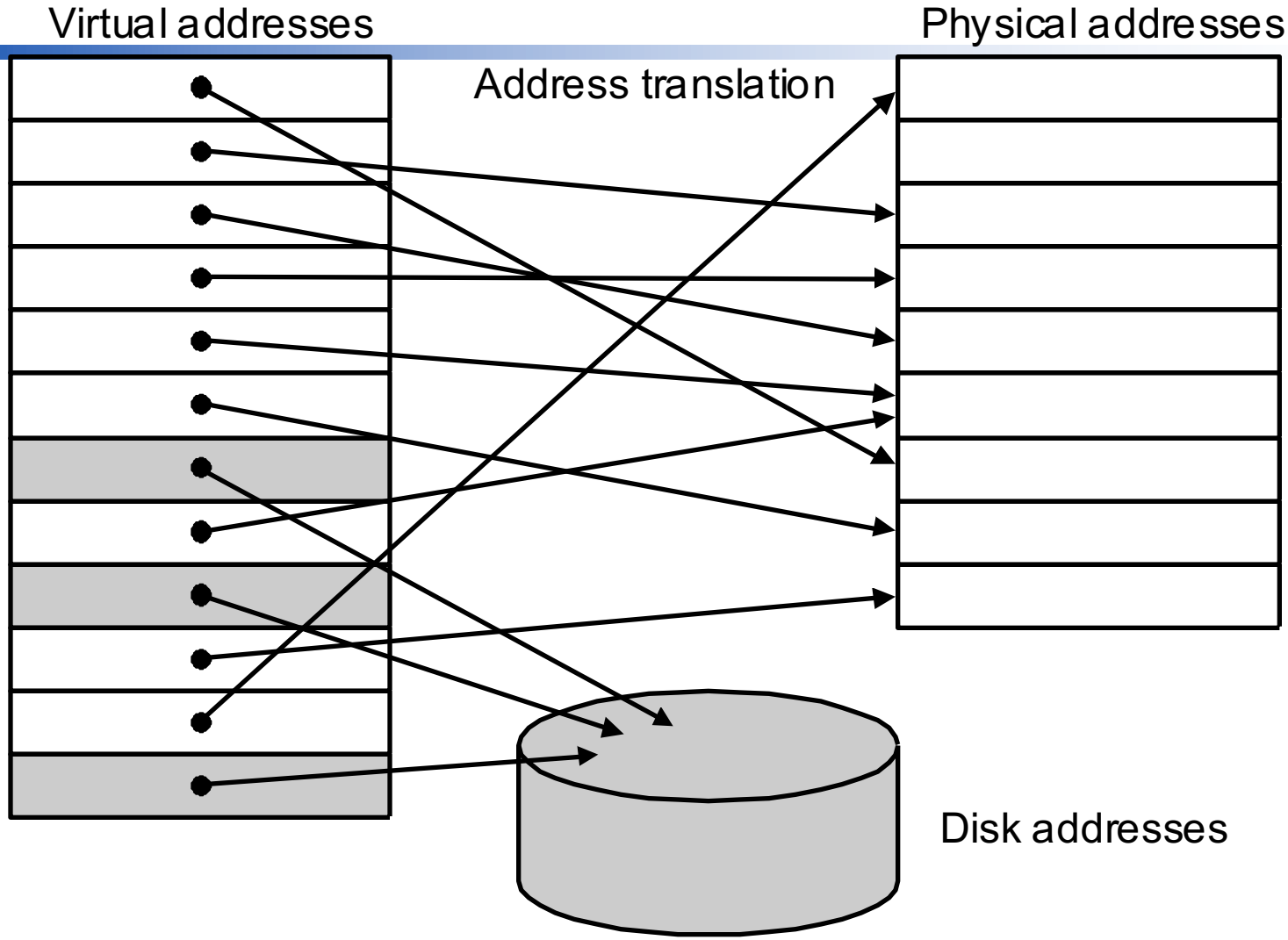
Virtual memory

Consider a machine with a 32 bit address space:

- it probably doesn't have 2^{32} = 4 GB of main memory!
- How do we write programs without knowing how much memory is really available ahead of time?
- Why not *pretend* we always have 4GB, and if we use more than we really have, store some blocks on the hard disk.
  - this must happen automatically to be useful.
  - Note: 64-bit architectures typically have something like a 48 bit address or 262144 GB address space which is ~256 TB

Motivation

- Pretend we have 4GB, we really have only 512MB.
- At any time, the processor needs only a small portion of the 4GB memory.
  - only a few programs are active
  - an active program might not need all the memory that has been reserved by the program.
- We just keep the stuff needed in the main memory, and store the rest on disk.


Virtual addresses
Physical addresses
Address translation
Disk addresses

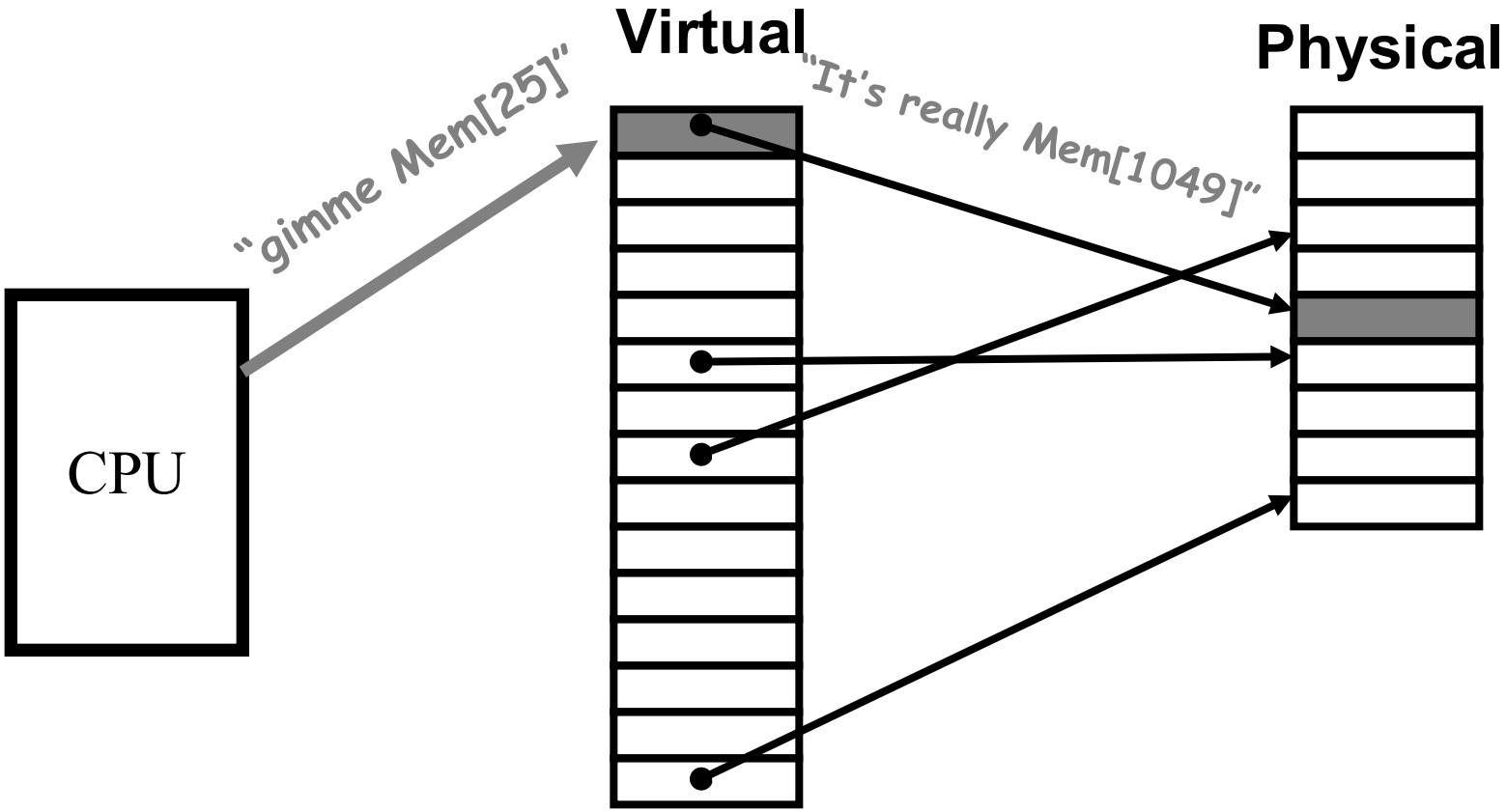
A Program's view of memory

- We can write programs that address the *virtual memory*.
- There is hardware that translates these virtual addresses to physical addresses.
- The operating system is responsible for managing the movement of memory between disk and main memory, and for keeping the address translation table accurate.

Terminology

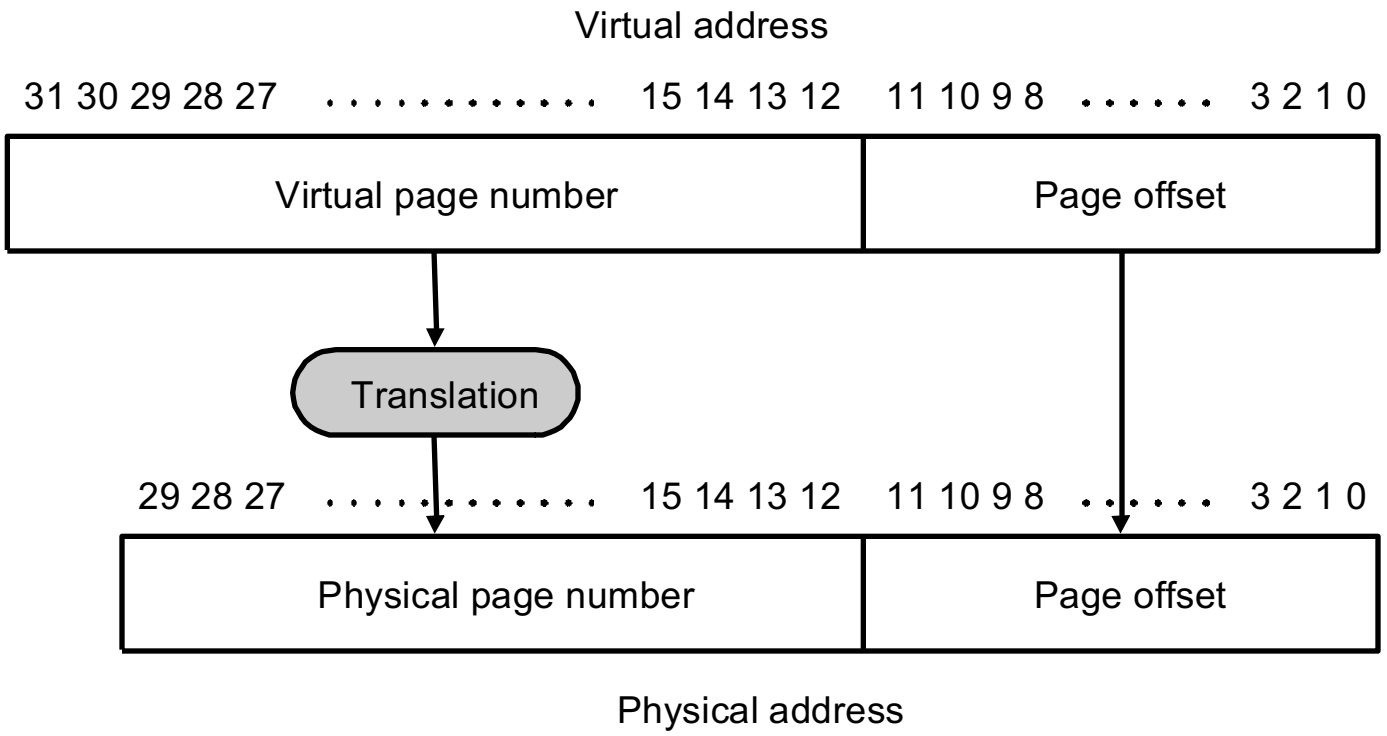
- *page*: The unit of memory transferred between disk and the main memory.
- *page fault*: when a program accesses a virtual memory location that is not currently in the main memory.
- *address translation*: the process of finding the physical address that corresponds to a virtual address.

Virtual Memory & Address Translation



Translation and Pages

- Only the page number need be translated.
- The offset within the page stays constant.



CPU & address translation

- The CPU doesn't need to worry about address translation – this is handled by the memory system (e.g., MMU)
- As far as the CPU is concerned, it *is* using physical addresses.

Advantages of VM

- A program can be written (linked) to use whatever addresses it wants to! It doesn't matter where it is physically loaded!
- When a program is loaded, it doesn't need to be placed in continuous memory locations
  - any group of physical memory *pages* will do fine.

Design Issue

- A Page Fault is a disaster!
  - disk is very, very, very slow compared to memory – millions of cycles!
- Minimization of faults is the primary design consideration for virtual memory systems.
- This "page" is important! It's your "fault" if you miss this point ☺

Minimizing faults

- Pages should be big enough to make a transfer from disk worthwhile. 4KB-64KB are typical sizes.
  - Some systems have 1 to 256 MB page sizes
- Fully associative placement is the most flexible (will reduce the rate of faults).
  - software handles the placement of pages.

What about rights writes?

- Write through is not practical for a virtual memory system (writes to disk are way to slow).

- Write back is always used.
  - write the entire page to disk only when kicked out of the main memory and placed on disk.

The *dirty* bit

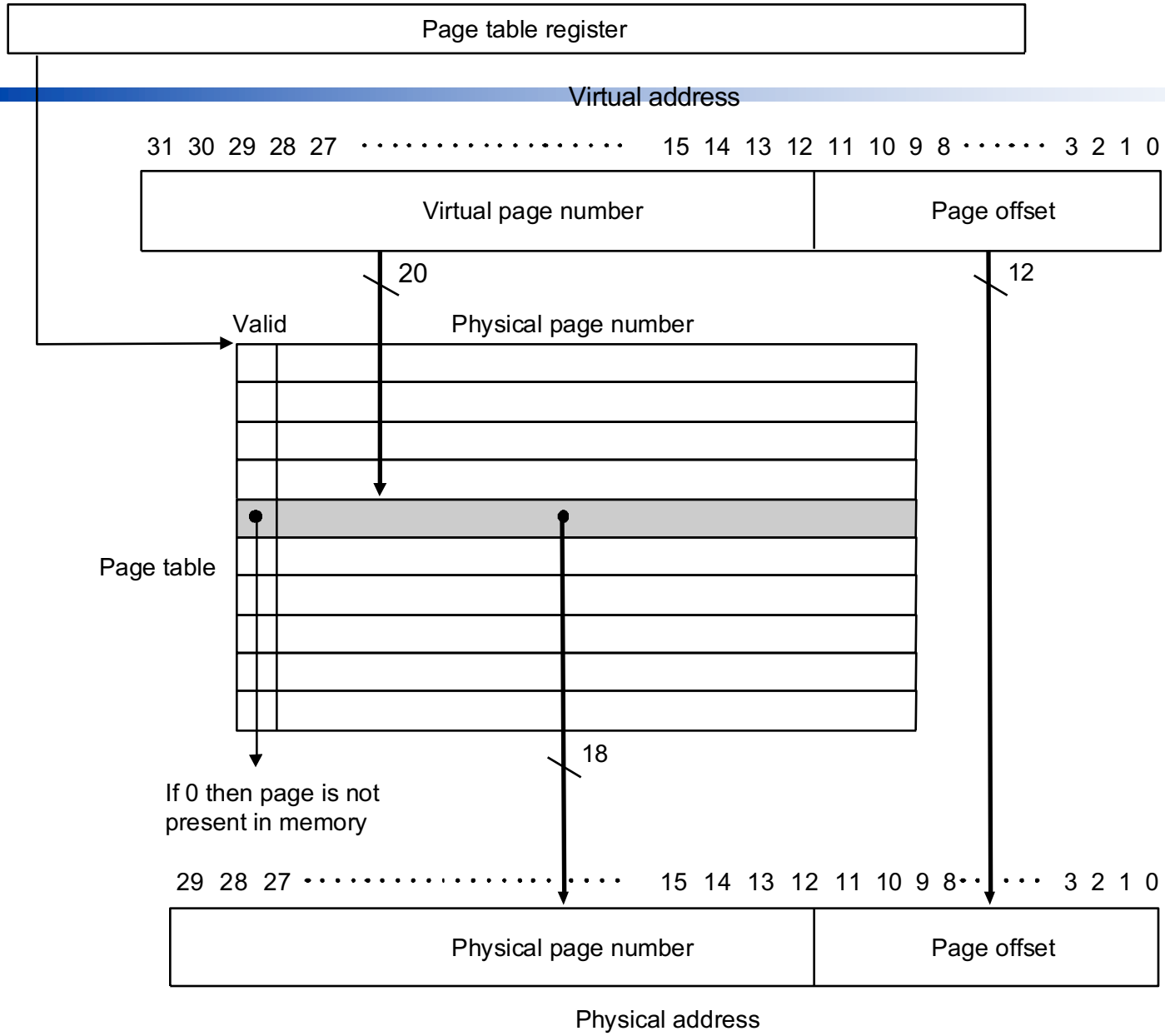
- It would be wasteful to *always* write an entire page to disk if nothing in the page has changed.
- A flag is used to keep track of which pages have been changed in main memory (if not change happens, no need to write the page to disk).
- The flag is called the *dirty bit*.

Address Translation

- Address translation must be fast (it happens to every memory access).
- We need a fully associative placement policy.
- We can't afford to go looking at every virtual page to find the right one
  - we don't use the *tag bits* approach

Page Table

- We need a large table that holds the physical address for each virtual page.
- Want virtual page 1234? Look at row 1234 in the table.
  - the page table is a big array indexed by virtual page number.
- The table will be huge! 2^{32}/page size.


Page table register
Virtual address
31 30 29 28 27 15 14 13 12 11 10 9 8 3 2 1 0
Virtual page number
Page offset
20
12
Valid
Physical page number
Page table
18
If 0 then page is not present in memory
29 28 27 15 14 13 12 11 10 9 8 3 2 1 0
Physical page number
Page offset
Physical address

Processes and Page Tables

- Each process has it's own page table!
  - each program can pretend it is loaded and running at the same address.
- One page table is huge, now we need to worry about lots of page tables.
- We can't include dedicated hardware that holds all these page tables.

Page Tables memory needs

- Assume 32 bit virtual address space.
- Assume 16K Byte page size.
  - each page table needs $2^{32}/2^{14} = 2^{18}$ elements.
- We would like to support 256 different processes.
- We need $2^8 * 2^{18} = 2^{26}$ page table elements, assume each is 1 word wide.
- Total needed is 256 MBytes!
- A solution – "Page" the page table.

Page Table Elements

- Each element in the page table needs to include:
  - a valid bit.
  - if the page is in memory, the physical address.
  - if the page is on disk, some indication of where on the disk

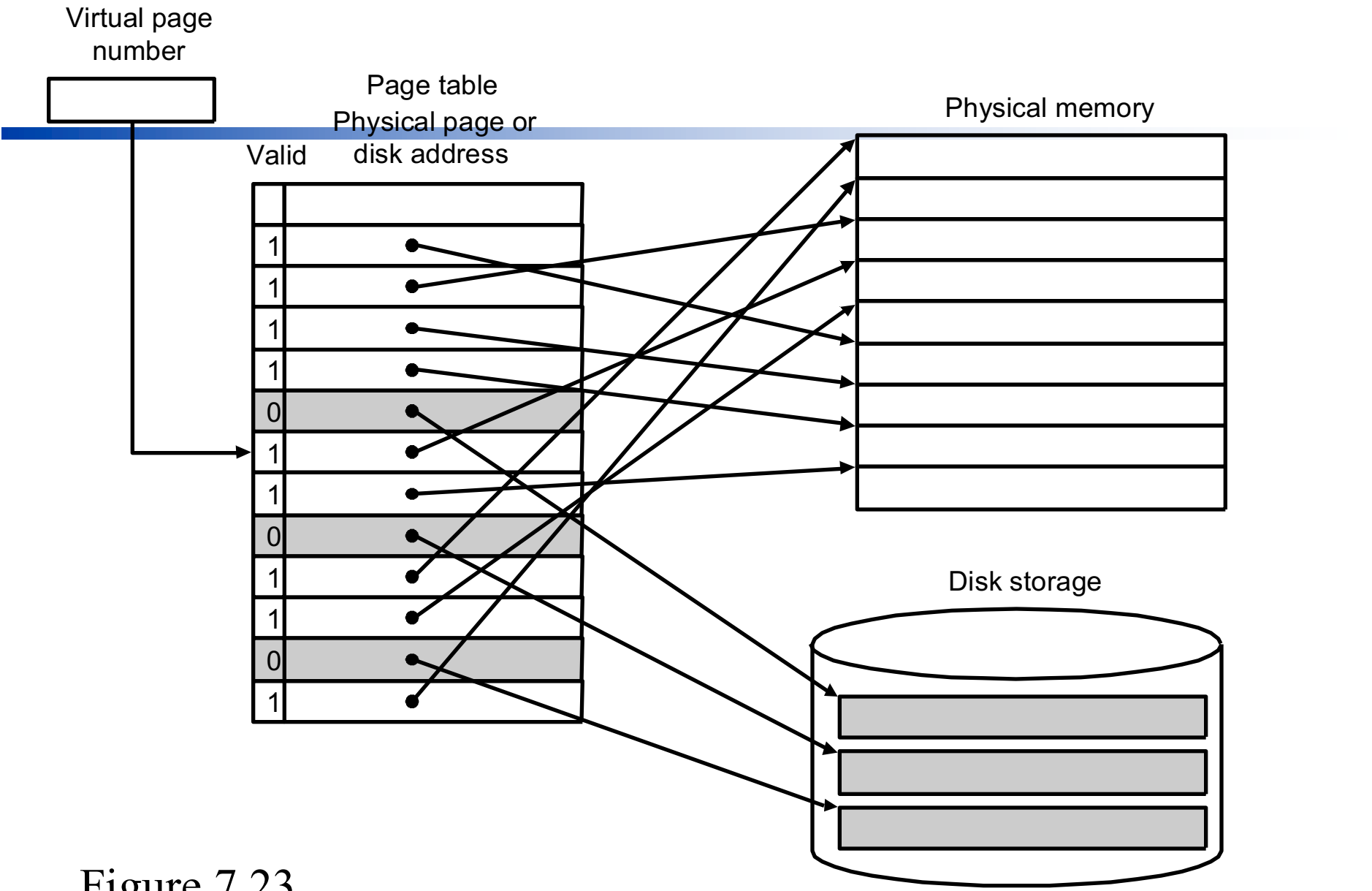

Virtual page number
Page table
Physical page or disk address
Valid
Physical memory
1
1
1
1
0
1
1
0
1
1
0
1
Disk storage

Figure 7.23

I need to go buy more memory!

- Page tables are stored in main memory.
- Most programs are small, so we don't need to actually create the entire page table for each process.
  - just enough to cover the actual pages that have been reserved for use by the program.
  - this number will be quite small (a few thousand pages is enough for a large program).

Speed of address translation

- Page tables are in memory.
- We need to access an element of the page table every time a translation is needed.
- A translation is needed on every memory access!
- Every memory access really requires 2 memory accesses!
  - This is very bad .. Especially for your uber-faster, superscalar pipelined processor!

Making address translation fast

- We can create a dedicated cache that holds the most recently used page table entries.
  - the same page table entry is used for all memory locations in the page. Spatial Locality.
- This cache is called a *Translation Lookaside Buffer* (TLB).

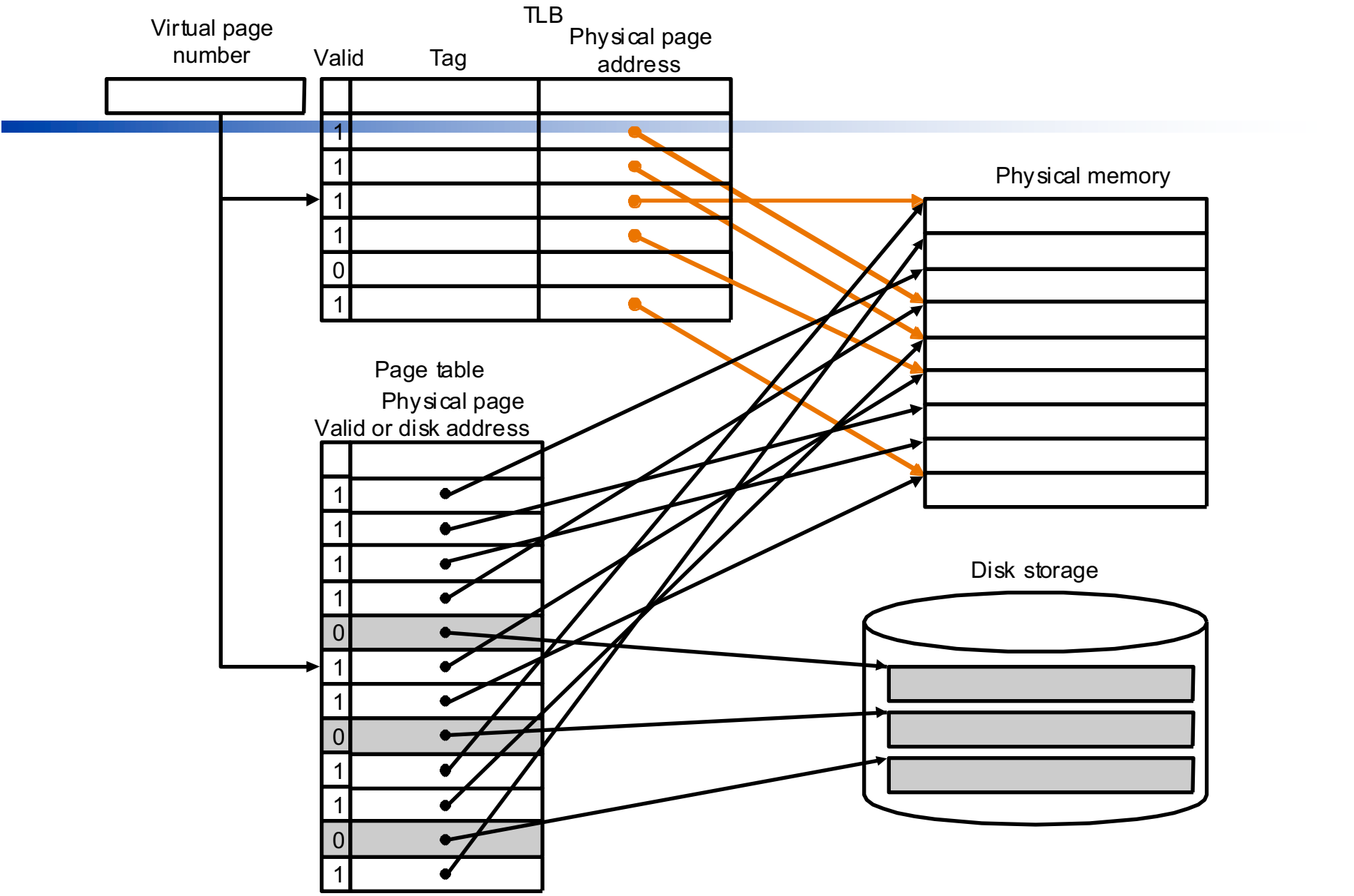
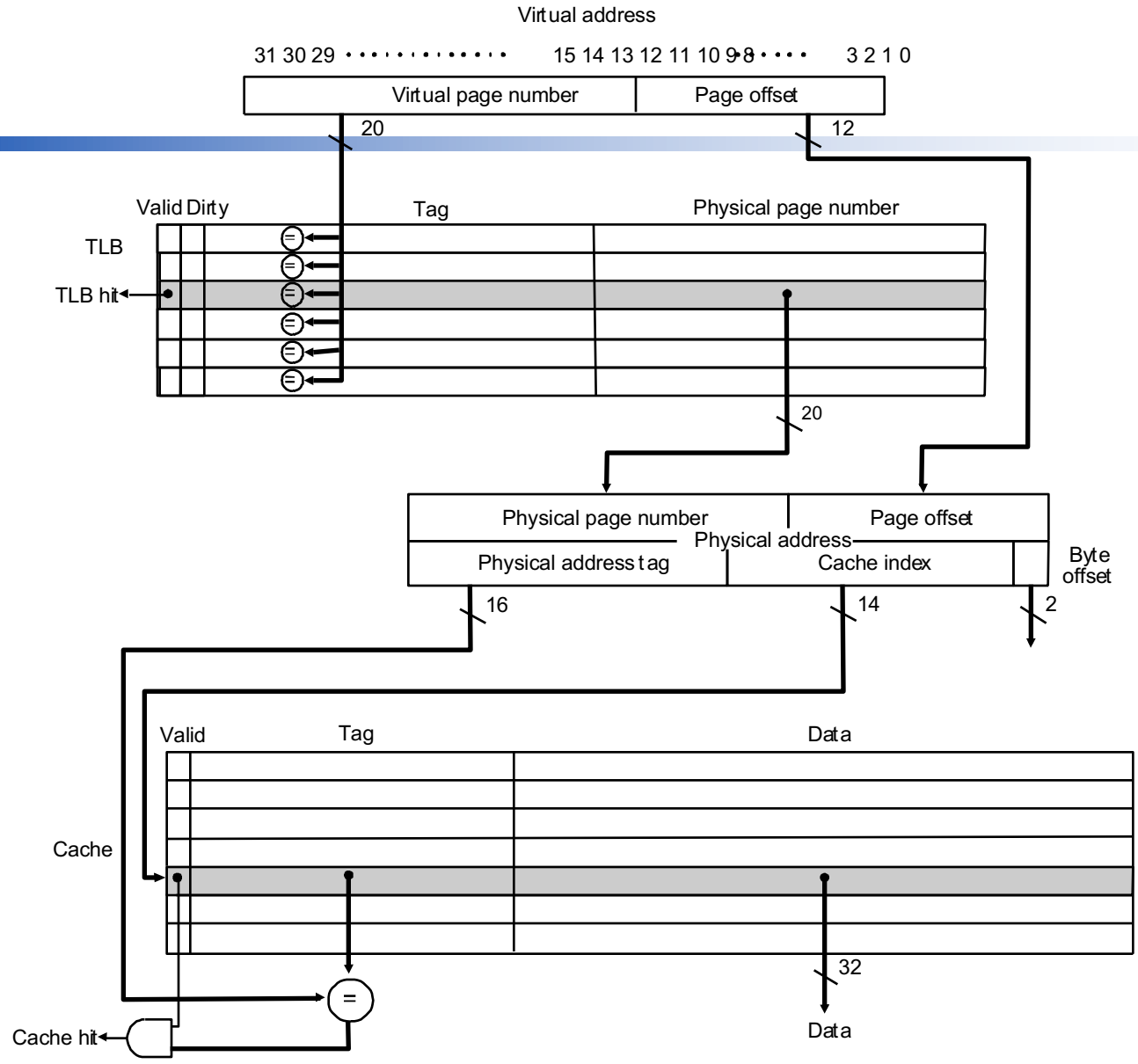

Virtual page number
TLB
Valid
Tag
Physical page address
1
1
1
1
0
1
Physical memory
Page table
Physical page
Valid or disk address
1
1
1
1
0
1
1
0
1
1
0
1
Disk storage

Figure 7.24

DecStation 3100 TLB

- 32 bit address space
- 4KB Page size
  - virtual page address is 20 bits.
- TLB has 64 slots
  - each has 20 bit tag, 20 bit physical page address, a valid bit and a dirty bit.
  - fully associative.


Virtual address
31 30 29 • • • • • • • • • • • • • • • 15 14 13 12 11 10 9 8 • • • • 3 2 1 0
Virtual page number
Page offset
20
12
Valid Dirty
Tag
Physical page number
TLB
TLB hit
20
Physical page number
Page offset
Physical address
Physical address tag
Cache index
Byte offset
16
14
2
Valid
Tag
Data
Cache
32
Data
Cache hit

Cache + Virtual Memory

- The Decstation 3100 does address translation *before* the cache.
- The cache operates on physical memory addresses.
- It is also possible to cache virtual memory, although there are some problems.
  - if programs can share pages, a single word from physical memory could end up in the cache twice! (the same physical location could have 2 different virtual addresses).

Protection

- Virtual memory allows multiple processes to share the same physical memory.
- What if my process tries to write to your process's memory?
  - we don't want this to be possible!
  - we don't even want it to be able to read!
- We can provide protection via the page tables

Independent Page Tables

- Each process has it's own page table.
- All page tables are created by the operating system – your program can't change it's own page table.

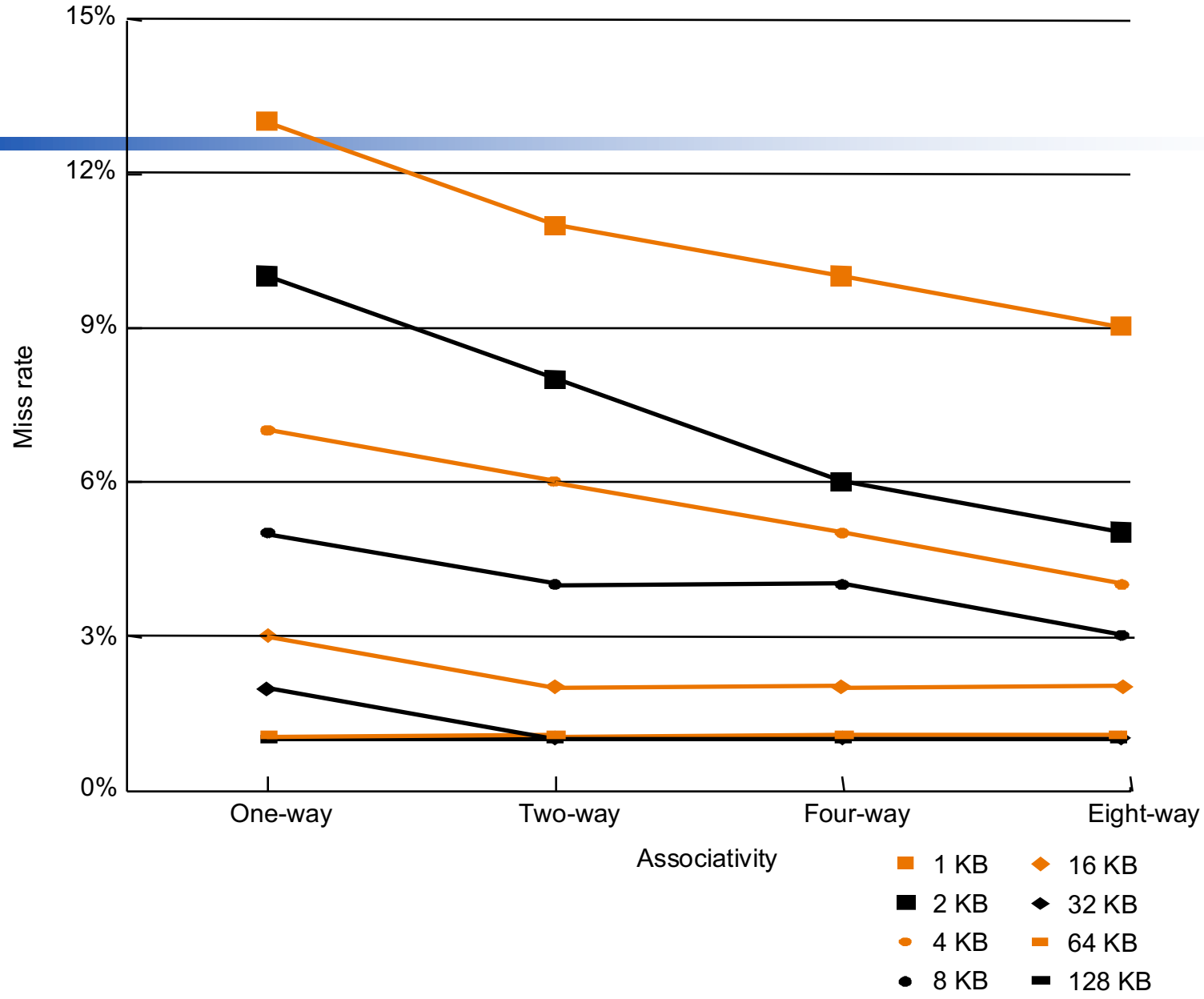
- Supporting virtual memory requires a combination of hardware and software.

Common Issues

- There are a number of issues that are common to both cache and virtual memory system design:
  - block placement policy.
  - how is a block found?
  - block replacement policy.
  - write policy.

Block Placement Options

- Direct-Mapped
  - cheap, easy to implement, relatively high miss rate.
- Set Associative
  - middle ground
- Fully Associative
  - expensive (lots of hardware or software), minimizes miss rate.


15%
12%
9%
6%
3%
0%
Miss rate
One-way
Two-way
Four-way
Eight-way
Associativity
1 KB
2 KB
4 KB
8 KB
16 KB
32 KB
64 KB
128 KB

How is a block found?

This depends on placement policy.

- Direct Mapped: uses an index.
- Set Associative: index selects a set, and we need to look at all set elements.
- Fully Associative: need to look at all elements.

Replacement Policies

- Direct-Mapped: not an issue.
- Set and fully associative
  - LRU (*least recently used*) hard to implement in hardware for large sets, often approximated.
  - random easy to implement, does nearly as well as LRU approximations.
- LRU is always used (or approximated) for virtual memory.

Write Policies

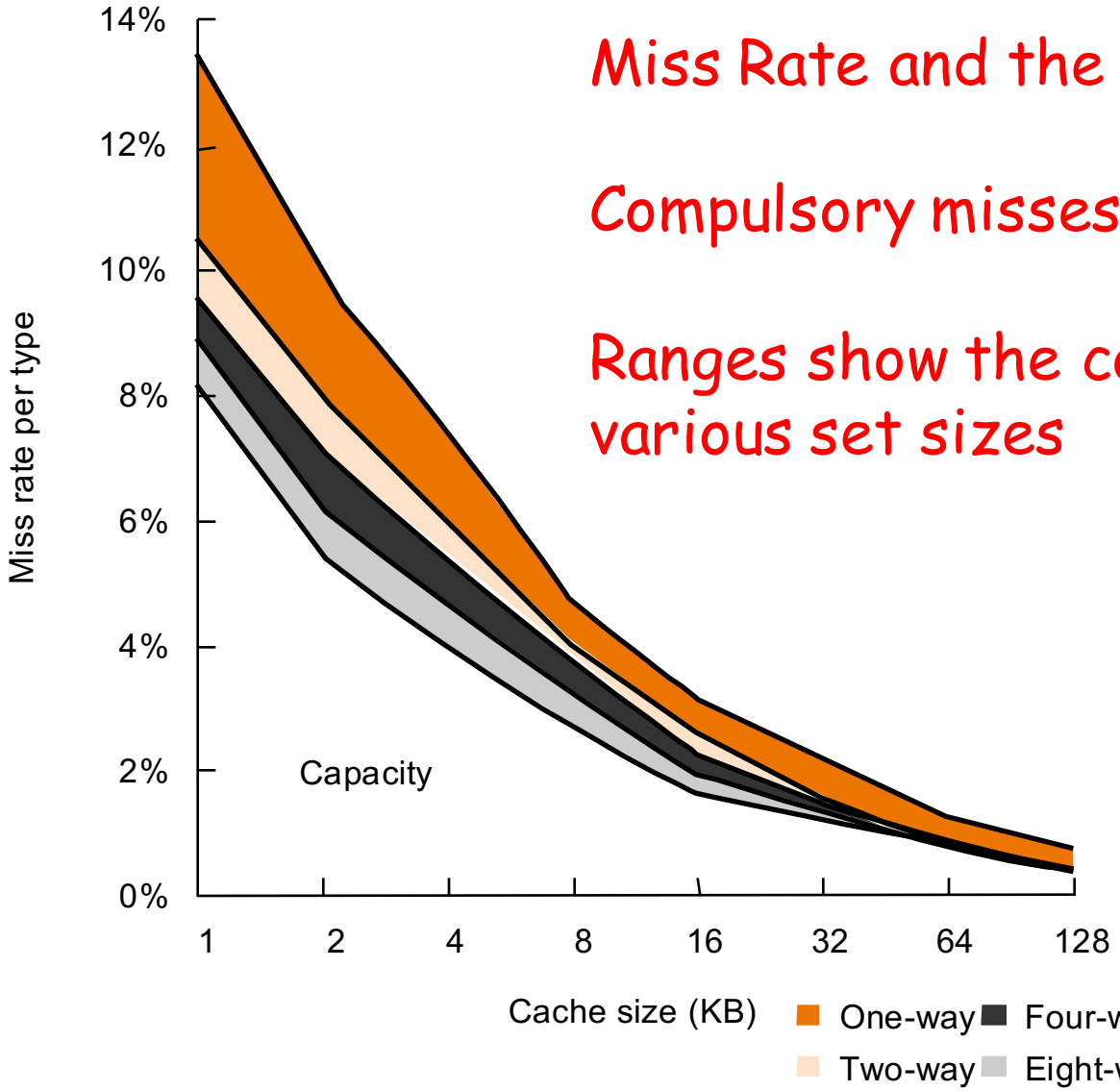
- Write-Through: update the cache and lower level memory.

- Write-Back: update the cache only. When block/page is booted from the cache - write to lower-level memory if any changes.

Where do misses come from?

- Compulsory misses: the first access is always a miss. Can't avoid these.
- Capacity misses: cache can't hold all the blocks needed.
- Conflict misses: multiple blocks compete for the same cache slot(s) and collide.

Where do misses come from?



Miss Rate and the cause of misses.

Compulsory misses are baseline of 0.2%

Ranges show the conflict misses for various set sizes

Cache friendly code
(a great name for a band!)

- There are sometimes things you can do to your program to take advantage of the cache.
  - usually it's not necessary to know much about the specific architecture of the cache on which a program is run.
- The patterns of array element access is one good example.

Matrix Multiplication

```
for (i=0; i!=500; i++)
  for (j=0;j!=500; j++)
    for (k=0;k!=500; k++)
      x[i][j] = x[i][j] + y[i][k]*z[k][j];
```

almost twice as fast on SGI Mips R4000

```
for (k=0; k!=500; k++)
  for (j=0;j!=500; j++)
    for (i=0;i!=500; i++)
      x[i][j] = x[i][j] + y[i][k]*z[k][j];
```