

Caching

Some slides based on those provided by the authors of
our text

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

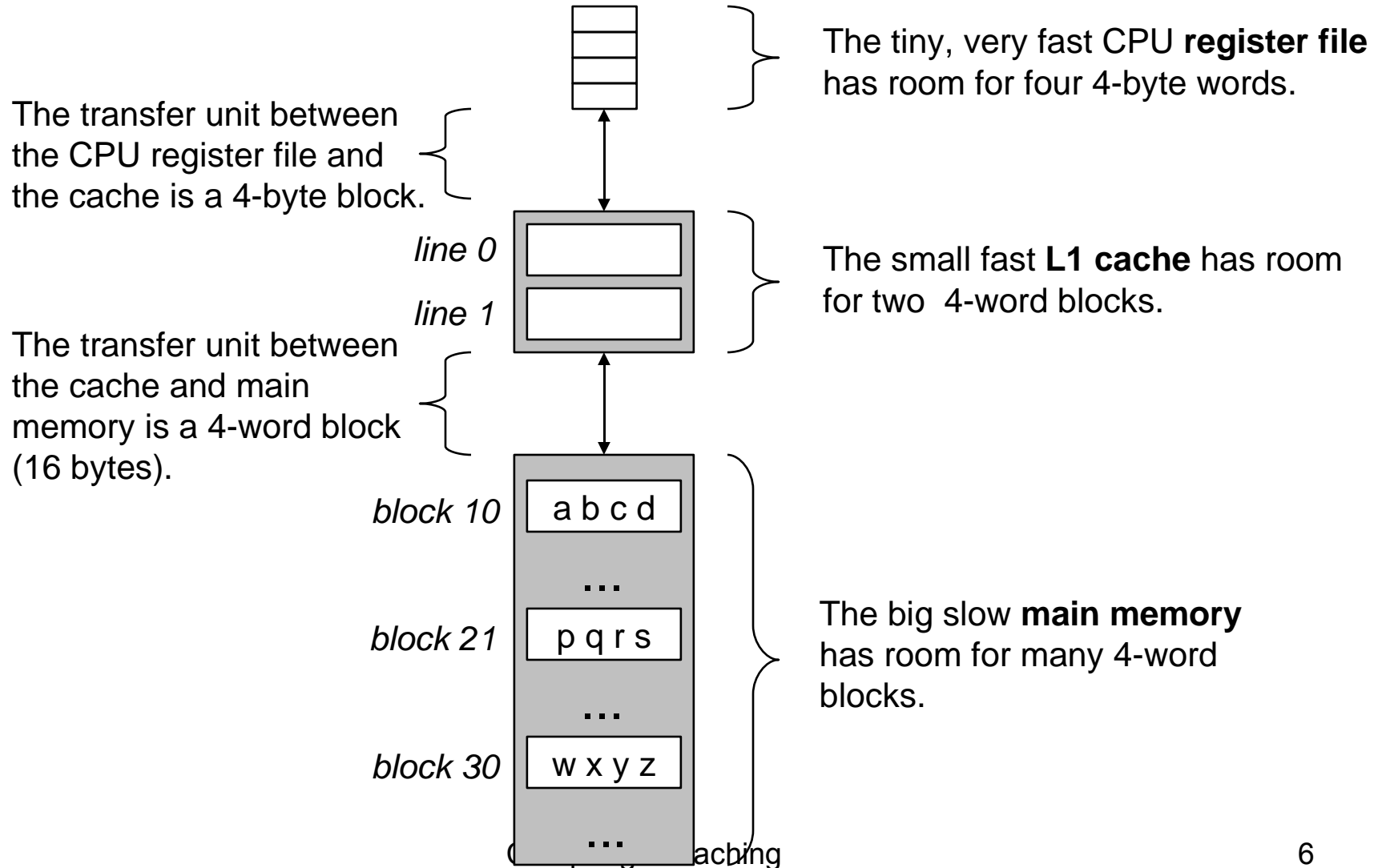
Programs tend to exhibit a great deal of *locality* in memory accesses.

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

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).

Inserting an L1 cache between the CPU and main memory



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*: $1 - \textit{hit rate}$
- *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, \dots, X_{n-1}
- The processor next accesses memory item X_n which is not in the cache.

Cache before and after

X4
X1
X _{n-2}
X _{n-1}
X2
X3

a. Before the reference to X_n

X4
X1
X _{n-2}
X _{n-1}
X2
X _n
X3

b. After the reference to X_n

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 is 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.
- This requires a cache size of 2^n locations (a power of 2).

A Direct Mapped Cache

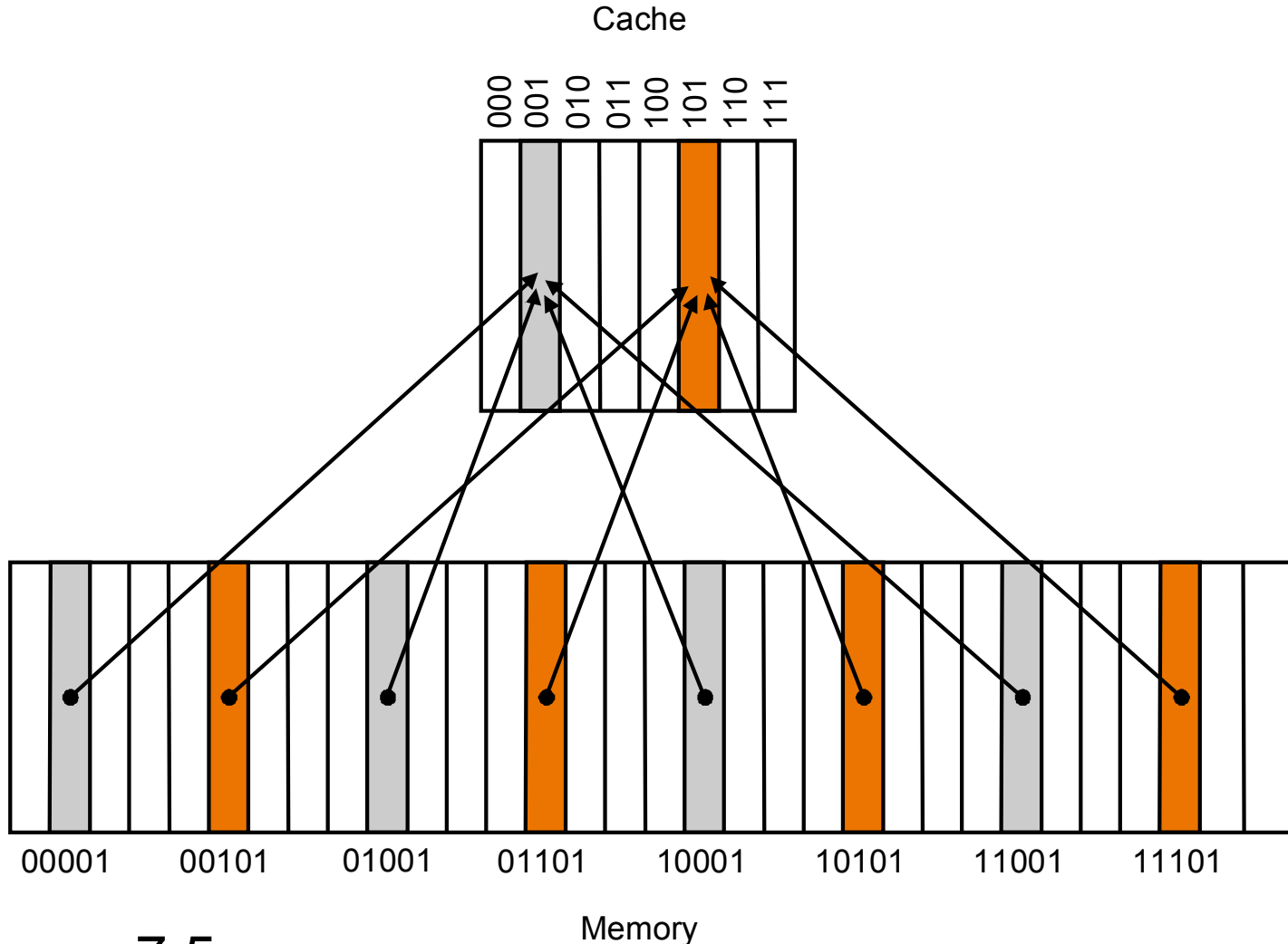


Figure 7.5

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.

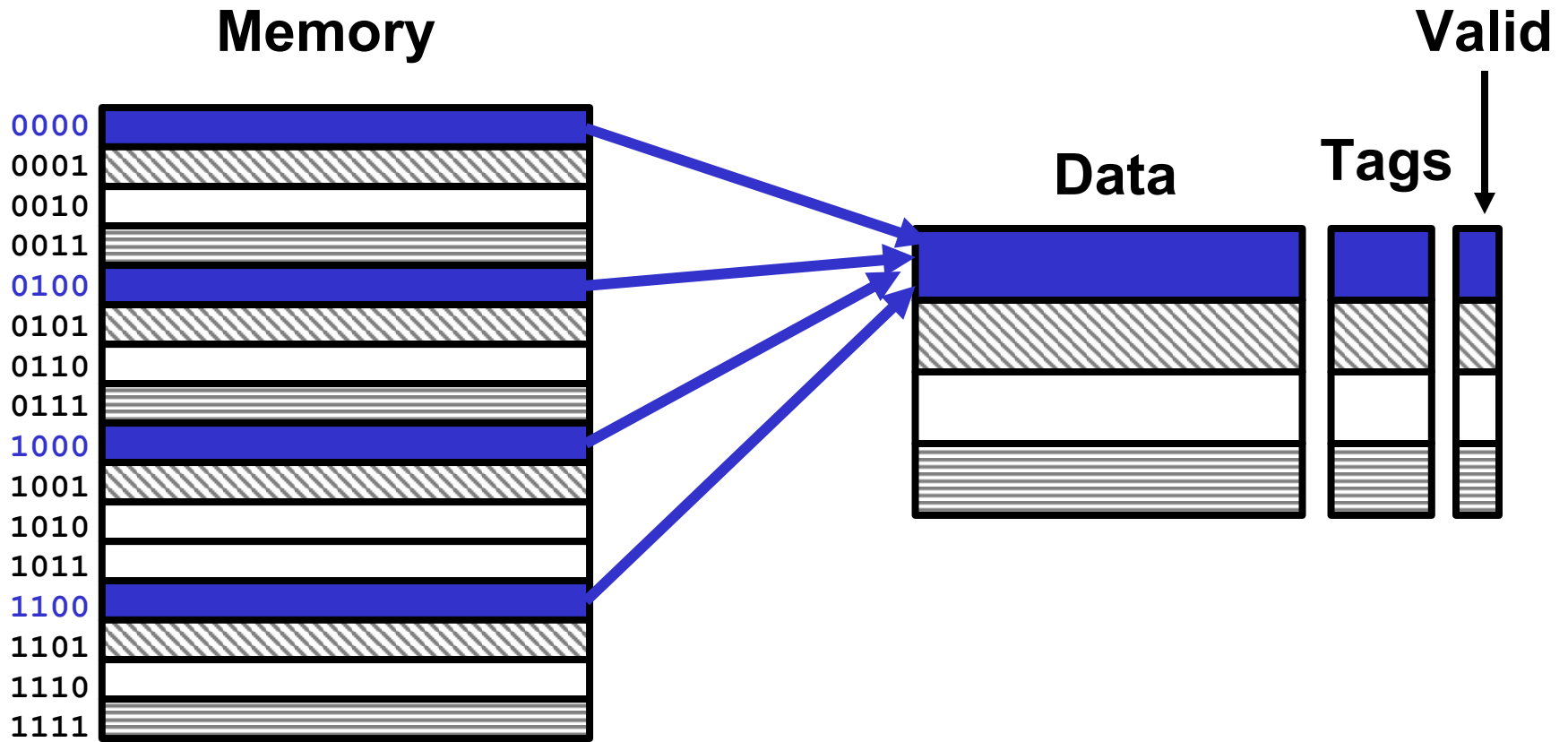
Initialization Problem

- 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)	<i>miss</i>
8	1000	00 (0)	<i>miss</i>
3	0011	11 (3)	<i>hit</i>
2	0010	10 (2)	<i>miss</i>
4	0100	00 (0)	<i>miss</i>
8	1000	00 (0)	<i>miss</i>

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

Slot V Tag Data

00

01

10

11

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

Initialize Cache Valid Bits	Slot	V	Tag	Data
	00	0		
	01	0		
	10	0		
	11	0		

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

First Access	Slot	V	Tag	Data
	00	0		
	01	0		
	10	0		
	11	1	00	Mem [3]

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

2nd Access

Slot

V Tag

Data

00

1

10

Mem [8]

01

0

10

0

11

1

00

Mem [3]

Address	Binary Address	Slot	hit or miss
3	0011	11 (3)	<i>miss</i>
8	1000	00 (0)	<i>miss</i>
→ 3	0011	11 (3)	<i>hit</i>
2	0010	10 (2)	<i>miss</i>
4	0100	00 (0)	<i>miss</i>
8	1000	00 (0)	<i>miss</i>

3rd Access

Slot

V Tag

Data

00

1

10

Mem [8]

01

0

10

0

11

1

00

Mem [3]

Address	Binary Address	Slot	hit or miss
3	0011	11 (3)	<i>miss</i>
8	1000	00 (0)	<i>miss</i>
3	0011	11 (3)	<i>hit</i>
→ 2	0010	10 (2)	<i>miss</i>
4	0100	00 (0)	<i>miss</i>
8	1000	00 (0)	<i>miss</i>

4th Access

Slot

V Tag

Data

00

1

10

Mem [8]

01

0

10

0

00

Mem [2]

11

1

00

Mem [3]

Address	Binary Address	Slot	hit or miss
3	0011	11 (3)	<i>miss</i>
8	1000	00 (0)	<i>miss</i>
3	0011	11 (3)	<i>hit</i>
2	0010	10 (2)	<i>miss</i>
→ 4	0100	00 (0)	<i>miss</i>
8	1000	00 (0)	<i>miss</i>

5th Access

Slot	V	Tag	Data
00	1	01	Mem [4]
01	0		
10	0	00	Mem [2]
11	1	00	Mem [3]

Address	Binary Address	Slot	hit or miss
3	0011	11 (3)	<i>miss</i>
8	1000	00 (0)	<i>miss</i>
3	0011	11 (3)	<i>hit</i>
2	0010	10 (2)	<i>miss</i>
4	0100	00 (0)	<i>miss</i>
→ 8	1000	00 (0)	<i>miss</i>

6th Access

Slot

V Tag

Data

00

1	10	Mem [8]
0		
0	00	Mem [2]
1	00	Mem [3]

01

10

11

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

Some Questions

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).

Some Answers

- Memory has 2^{30} words
- Cache has $16K = 2^{14}$ slots (words).
- Each cache slot can hold any one of $2^{30} \div 2^{14} = 2^{16}$ memory locations, so the tag must be 16 bits.
- 2^{16} is 64K memory locations that map to the same cache slot.
- Total memory in bits = $2^{14} \times (32+16+1)$
= $49 \times 16K = 784$ 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.

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.

Spatial Locality?

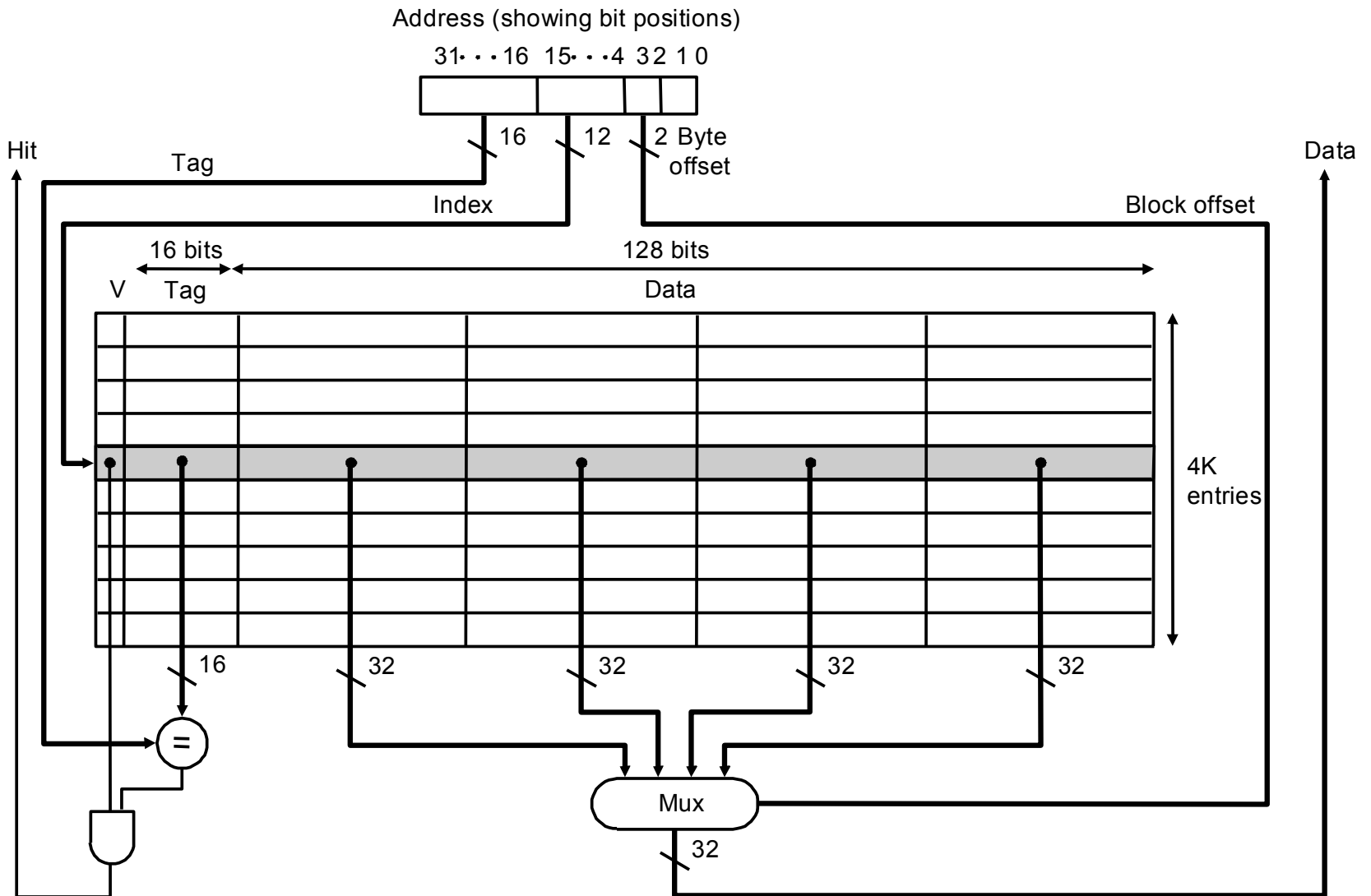
- So far we've only dealt with temporal locality (if we access an item, it is likely we will access it again soon).
- What about space (the final frontier)?
 - In general we make a *block* hold more than a single word.
 - Whenever we move *data* to the cache, we also move its neighbors (*troi* lives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

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.



Performance and Block Size

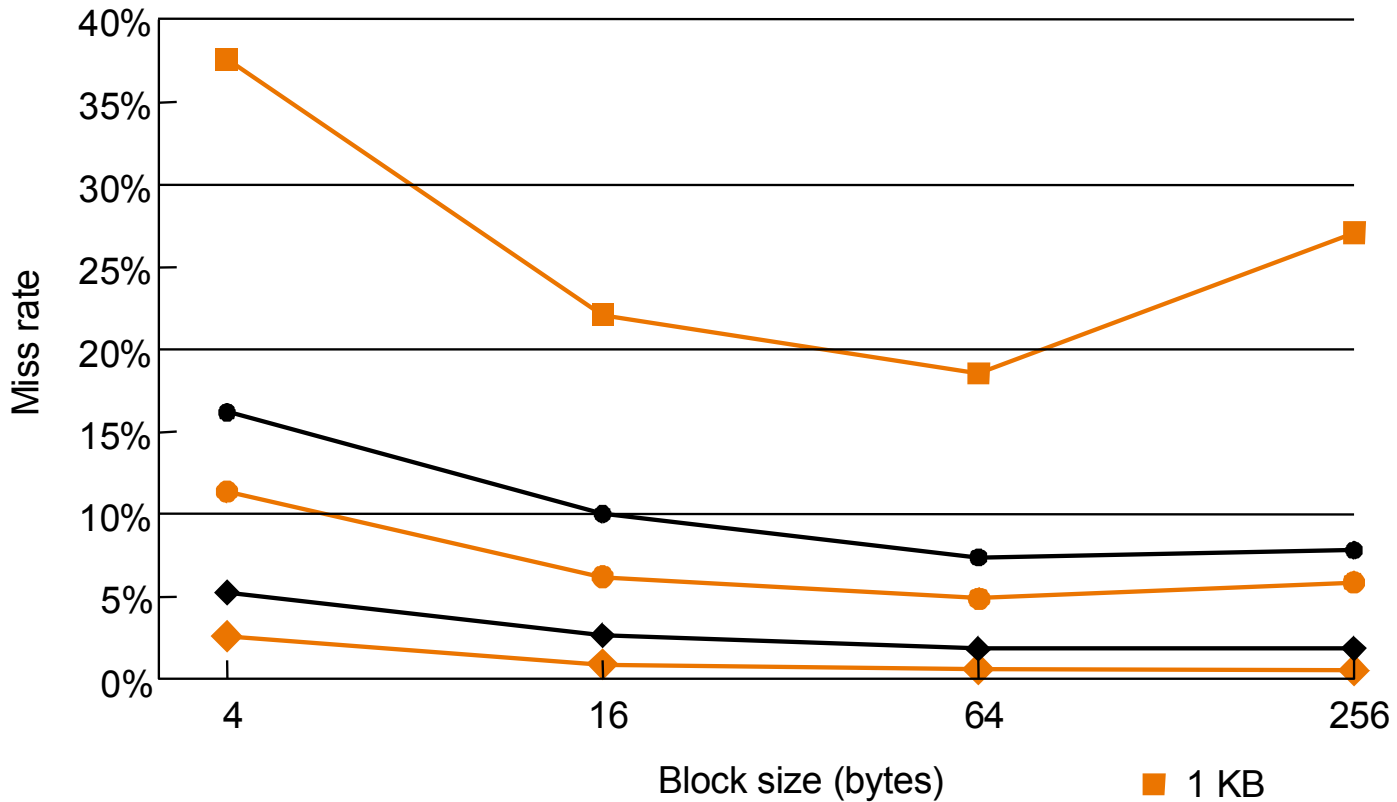
DecStation 3100 cache with block sizes 1 and 4 (words).

Program	Block	Miss Rate		
	Size	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

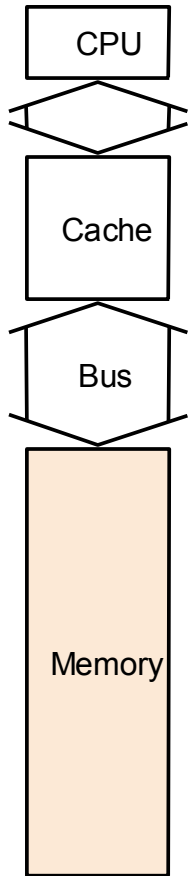
Hypothetical access times:

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

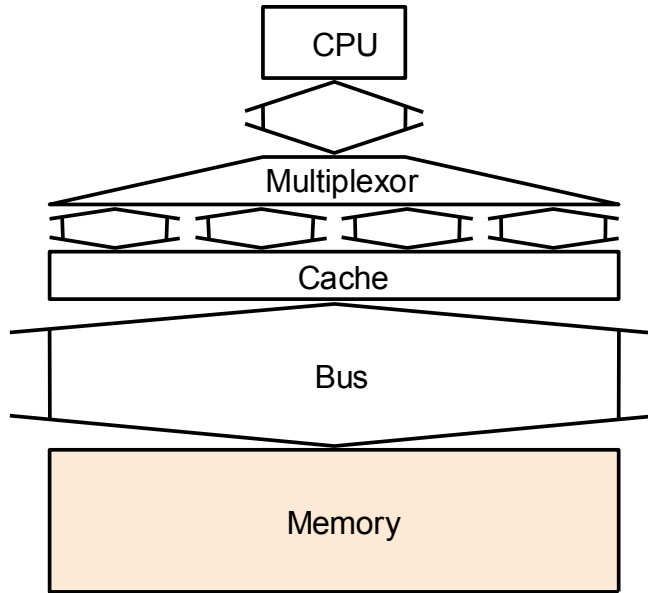
- Miss penalty for 4-word wide memory is:

$$1 + 4 \times 15 + 4 \times 1 = 65 \text{ cycles.}$$

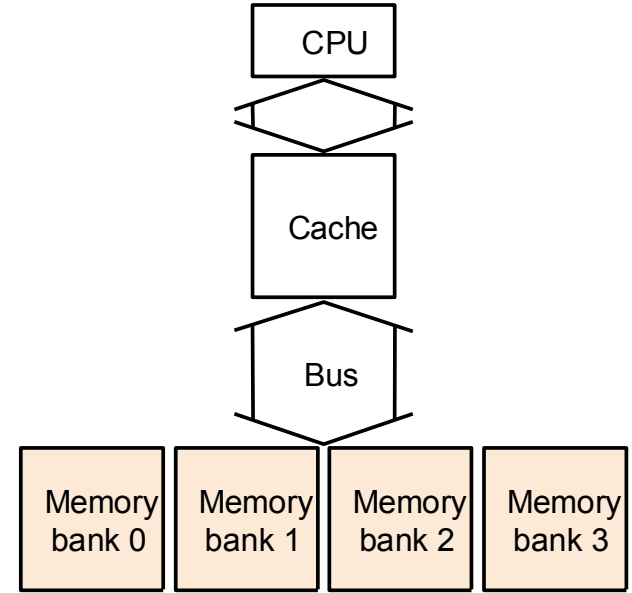
Memory Organization Options



a. One-word-wide memory organization



b. Wide memory organization



c. Interleaved memory organization

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.

$$\text{Memory-stall cycles} = \frac{\text{Memory Accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

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.

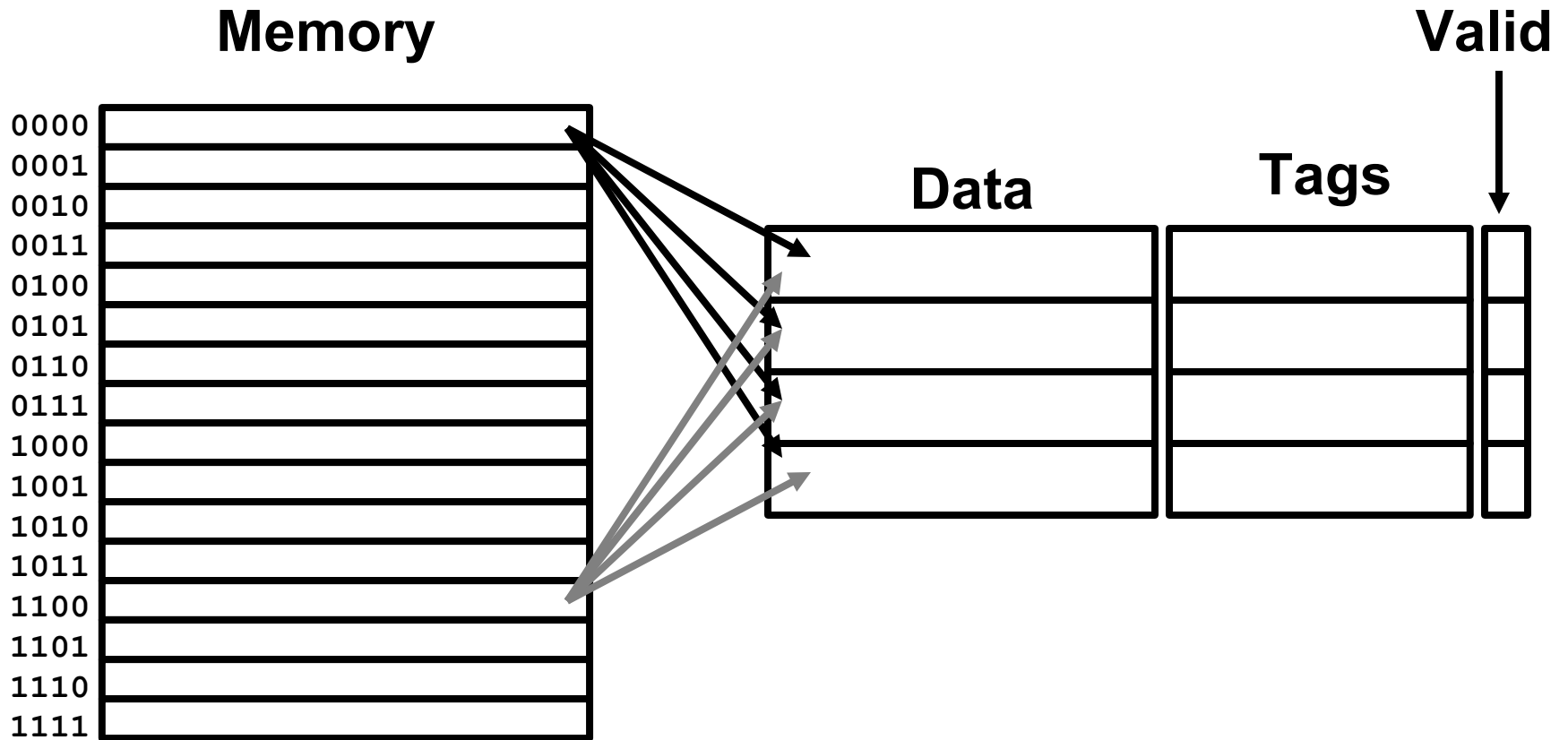
```
for (i=0;i<10000;i++) {  
    a[i] = a[i] + a[i+64] + a[i+128];  
    a[i+64] = a[i+64] + a[i+128];  
}
```

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!
- 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.

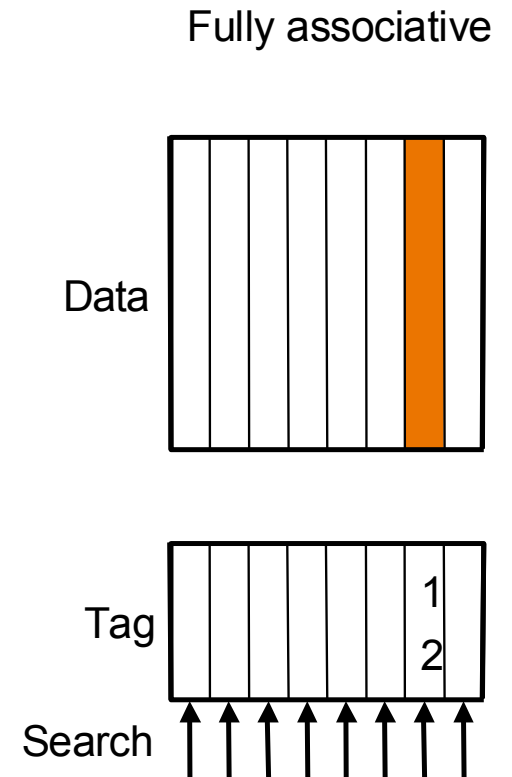
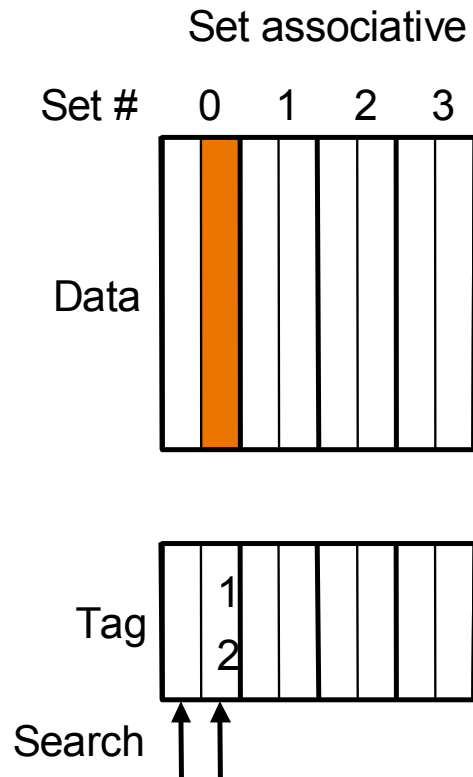
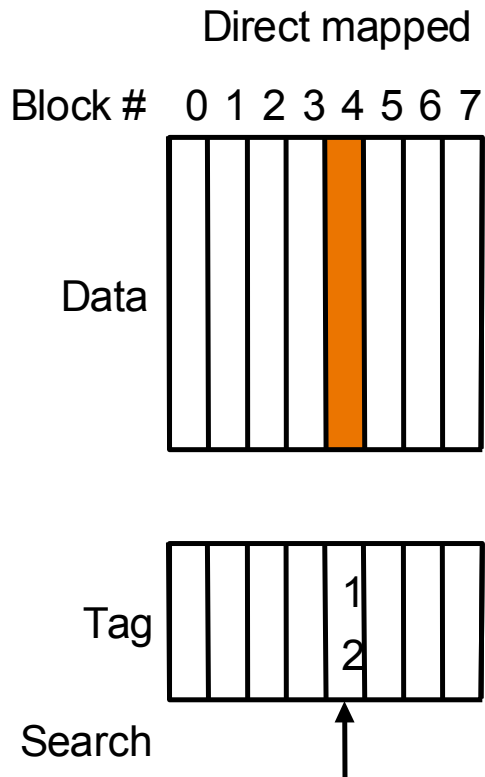
Set Associative

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 .

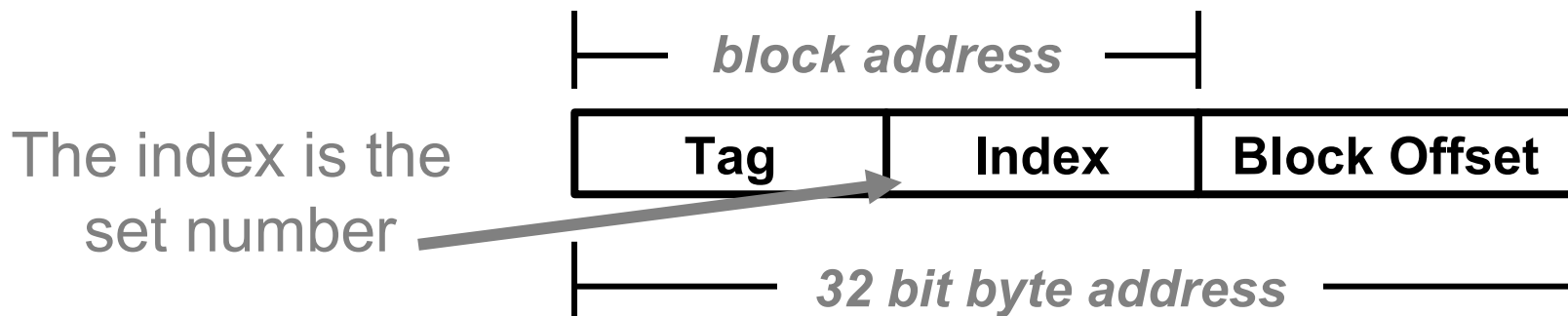
Block Placement Options

(memory block address 12)



Block Addresses and 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.



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.

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

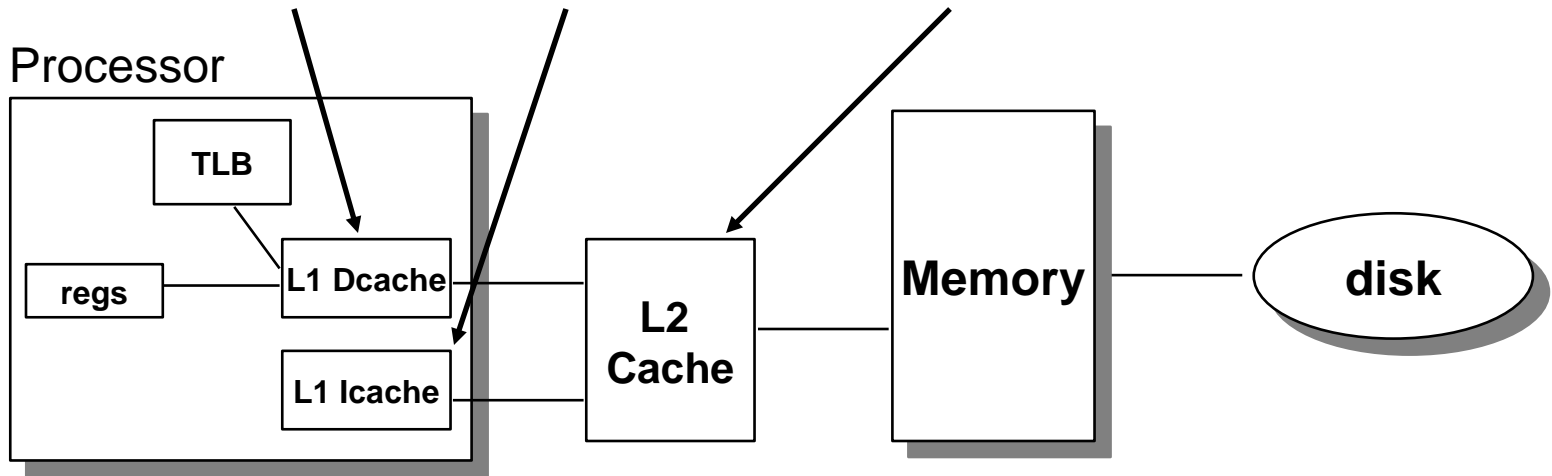
- Most modern processors include an *on-chip cache* (the cache is part of the processor chip).
- The size of the on-chip cache is restricted by the size of the chip!
- Often, a secondary cache is used between the on-chip cache and the main memory.

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!

Multi-level caches

Options: *separate* data and instruction caches, or a *unified* cache



size:	200 B	8-64 KB	1-4MB SRAM	128 MB DRAM	30 GB
speed:	3 ns	3 ns	6 ns	60 ns	8 ms
\$/Mbyte:			\$100/MB	\$1.50/MB	\$0.05/MB
line size:	8 B	32 B	32 B	8 KB	

larger, slower, cheaper



larger line size, higher associativity, more likely to write back

Intel Pentium cache hierarchy

