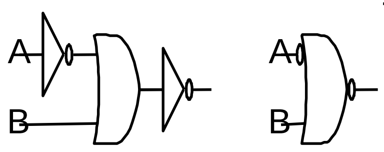
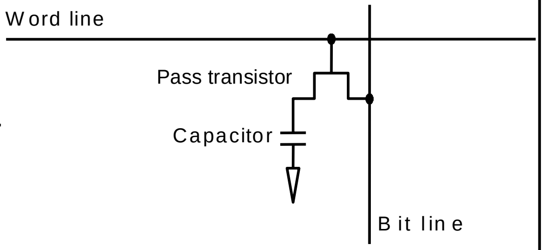
Chapter Seven

Large & Fast: Exploring Memory Hierarchy

Memories: Review

- SRAM (*Static Random Access Memory*):
  - value is stored on a pair of inverting gates
  - very fast but takes up more space than DRAM (4 to 6 transistors)
- DRAM (*Dynamic Random Access Memory*):
  - value is stored as a charge on capacitor (must be refreshed)
  - very small but slower than SRAM (factor of 5 to 10)



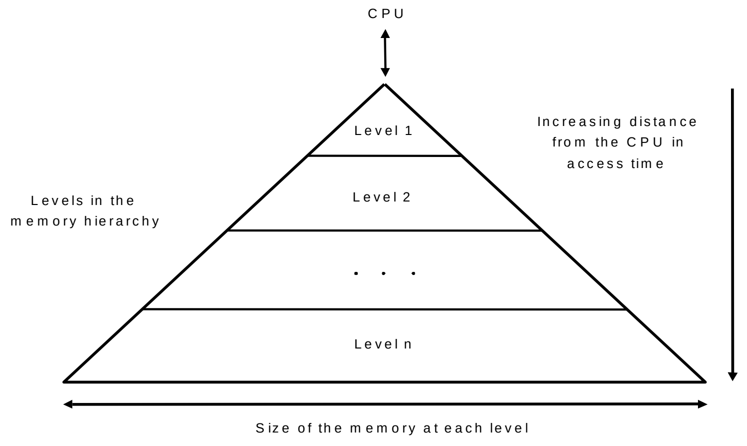


Exploiting Memory Hierarchy

- **Users want large and fast memories!**
  SRAM access times are 2 - 25ns at cost of $100 to $250 per Mbyte.
  DRAM access times are 60-120ns at cost of $5 to $10 per Mbyte.
  Disk access times are 10 to 20 million ns at cost of $.10 to $.20 per Mbyte. Prices according to the year of 1997.
- **Main memory is implemented from DRAM, while levels closer to the CPU (caches) use SRAM.**
- **A memory hierarchy consists of multiple levels of memory with different speeds and sizes.**
- **The fastest memories are more expensive per bit than the slower memories and thus usually smaller.**

Exploiting Memory Hierarchy

- **A memory hierarchy uses smaller and faster memory technologies close to the processor.**
- **Data cannot be present in level i unless it is present in level i + 1.**


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

Principle of Locality

- **A principle that makes having a memory hierarchy a good idea**
- **If an item is referenced:**
  Temporal locality (*locality in time*): it will tend to be referenced again soon
- **Example1: most programs contain loops, so instructions and data are likely to be accessed repeatedly, showing high amount of temporal locality**

  Spatial locality (*locality in space*): nearby items will tend to be referenced soon.
- **Example2: Since instructions are normally accessed sequentially, programs show high spatial locality.**
- **Example3: Access to elements of an array will naturally have high degrees of spatial locality.**

Two levels of Memory Hierarchy

- **Our initial focus: two levels (upper, lower).**
- **The upper level – the one closer to the processor – is smaller and faster than the lower level.**
  - **block: minimum unit of data**
  - **hit: data requested is in the upper level**
  - **miss: data requested is not in the upper level**
- **In case of a miss, the lower level in the hierarchy is then accessed to retrieve the block containing the requested data.**
- **Hit rate (hit ratio) is the fraction of memory accesses found in the upper level; it is often used as a measure of the performance of the memory hierarchy.**
- **Miss rate (1 - hit rate) is the fraction of memory accesses not found in the upper level.**

Two levels of Memory Hierarchy - Performance

- **Hit Time: is the time to access the upper level of the memory hierarchy, which includes the time needed to determine whether the access is a hit or a miss.**
- **Miss penalty: is the time to replace a block in the upper level with the corresponding block from the lower level, plus the time to deliver this block to the processor.**
- **Because the upper level is smaller and built using faster memory parts, the hit time will be much smaller than the time to access the next level in the hierarchy.**
- **Because all programs spend much of their time accessing memory, the memory system is necessarily a major factor in determining performance.**

The Big Picture – Memory Hierarchy

- **Memory hierarchies take advantage of temporal locality by keeping more recently accessed data items closer to the processor.**
- **Memory hierarchies take advantage of spatial locality by moving blocks consisting of multiple contiguous words in memory to upper levels of the hierarchy.**

The Basics of Caches

- **Cache: *a safe place for hiding or storing things.***
- **Cache memory is between the CPU and main memory.**
- **Example of a cache miss:**
  - **The cache just before and just after a reference to a word Xn that is not initially in the cache.**
  - **This reference causes a miss that forces the cache to fetch Xn from memory and insert it into the cache.**

X1
Xn-2
Xn-1
X2

Before reference to Xn

X1
Xn-2
Xn
Xn-1
X2

After reference to Xn

The Basics of Caches

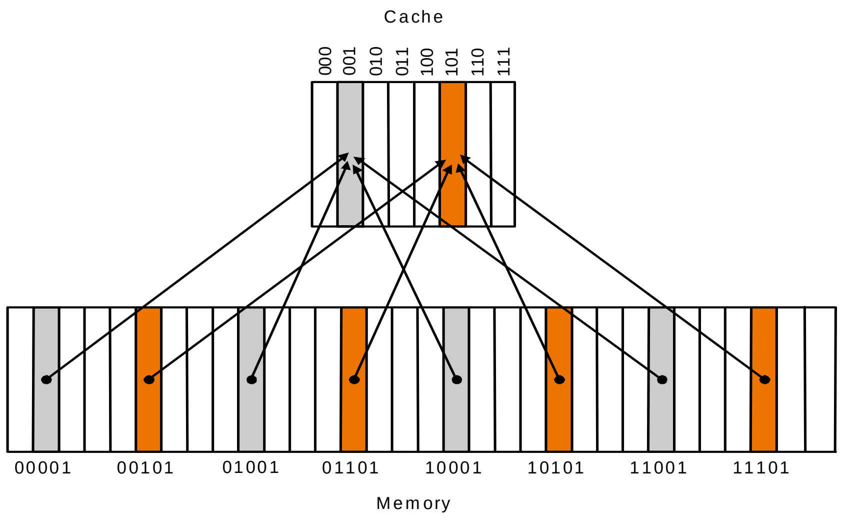
- **Two issues:**
  - **How do we know if a data item is in the cache?**
  - **If it is, how do we find it?**
- **Our first example:**
  - **block size is one word of data.**
  - **"<u>direct mapped</u>" – assign cache location based on address of word in memory, since each memory location is mapped to exactly one location in the cache.**

For each item of data at the lower level (such as main memory), there is exactly one location in the cache where it might be.

e.g., lots of items at the lower level (such as main memory) share locations in the upper level (such as cache memory)

Direct Mapped Cache

- **Mapping: address is modulo the number of blocks in the cache**



Example: Direct Mapped Cache

- **The figure in previous slide shows a direct-mapped cache with 8 entries showing addresses of main memory words between 0 and 31 that map to same cache locations.**
- **Because there are 8 words in cache, an address X maps to cache word X modulo 8. That is, the low-order $\log_2 (8) = 3$ bits are used as the cache index.**
- **In other words, $2^3 = 8$ → So, we need 3 bits to present 8 entries in cache and since the main memory contains 32 entries then we need 5 bits to present these entries (that is $2^5 = 32$). Thus, the lower 3 bits of the 5 bits are used as cache index.**
- **Therefore, addresses 00001, 01001, 10001, and 11001 all maps to entry 001 of the cache, while addresses 00101, 01101, 10101, and 11101 all map to entry 101 of the cache (check out the figure in the previous slide).**

Direct Mapped Cache – Tags

- Because each cache location can contain the contents of a number of different memory locations, how do we know whether the data in the cache corresponds to a requested word?
- Solution: We add a set of tags to cache.
  - The tags contain the upper portion of the address, corresponding to the bits that are not used as an index into the cache, thus we only need to ensure that the upper portion of the supplied address matches the tag.
- For example, in figure in slide 11, we need only have the upper 2 of the 5 address bits in the tag, since the lowest 3 bits of the address select the block.

13

Direct Mapped Cache – Valid bit

- When a processor starts up, the cache will be empty, and the tag fields will be meaningless.
- Even after executing many instructions, some of the cache entries may still be empty.
- Thus we need to know that the tag should be ignored for such entries.
- Solution: We add a valid bit to indicate whether an entry contains a valid address.

14

Accessing a Cache

- For the next few slides we will show how the contents of an 8-word direct-mapped cache as it responds to a series of requests from the processor.
- Since there are 8 blocks in the cache, the low-order 3 bits of an address give the block number.
- Here is the action for each reference:

Decimal address of reference	Binary address of reference	Hit or Miss in Cache	Assigned cache block
22	10110	Miss figure b	22 mod 8 = 6 (110)
26	11010	Miss figure c	26 mod 8 = 2 (010)
22	10110	Hit	22 mod 8 = 6 (110)
26	11010	Hit	26 mod 8 = 2 (010)
16	10000	Miss figure d	16 mod 8 = 0 (000)
3	00011	Miss figure e	3 mod 8 = 3 (011)
16	10000	Hit	16 mod 8 = 0 (000)
18	10010	Miss figure f	18 mod 8 = 2 (010)

Accessing a Cache

a) The initial state of the cache after power-on

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

b) After handling a miss of address 10110

Index	Valid	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Memory 10110
111	N		

Accessing a Cache

c) After handling a miss of address 11010

Index	Valid	Tag	Data
000	N		
001	N		
010	Y	11	Memory 11010
011	N		
100	N		
101	N		
110	Y	10	Memory 10110
111	N		

d) After handling a miss of address 10000

Index	Valid	Tag	Data
000	Y	10	Memory 10000
001	N		
010	Y	11	Memory 11010
011	N		
100	N		
101	N		
110	Y	10	Memory 10110
111	N		

Accessing a Cache

e) After handling a miss of address 00011

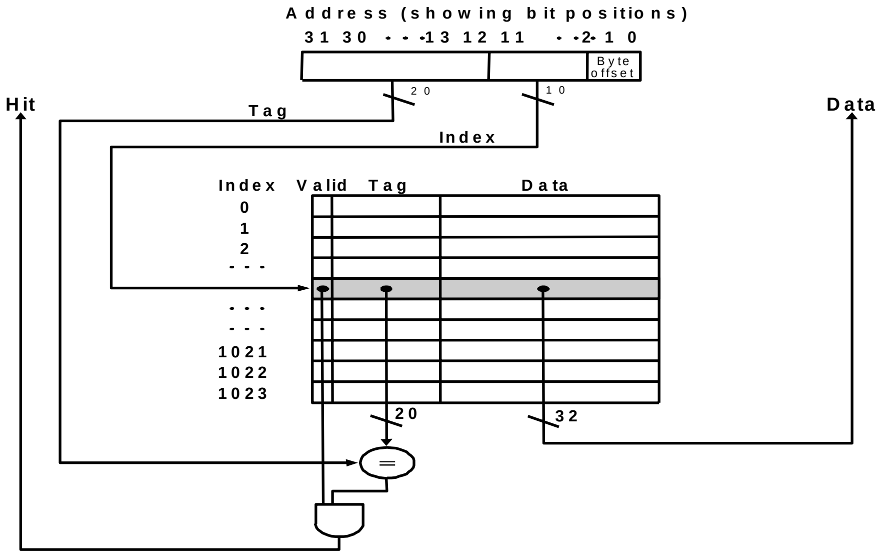
Index	Valid	Tag	Data
000	Y	10	Memory 10000
001	N		
010	Y	<u>11</u>	Memory 11010
011	Y	00	Memory 00011
100	N		
101	N		
110	Y	10	Memory 10110
111	N		

f) After handling a miss of address 10010

Index	Valid	Tag	Data
000	Y	10	Memory 10000
001	N		
010	Y	<u>10</u>	Memory 10010
011	Y	00	Memory 00011
100	N		
101	N		
110	Y	10	Memory 10110
111	N		

Note: We need to replace 26 (11010) in order to fetch 18 (10010)

Direct Mapped Cache – In MIPS



19

Direct Mapped Cache – In MIPS

- **For the cache in the previous slide:**
  - **The lower portion (index) of the address is used to select a cache entry consisting of a data word and a tag.**
  - **The tag from the cache is compared against the upper portion of the address (tag) to determine whether the entry in the cache corresponds to the requested address.**
  - **Because the cache has 2^{10} (or 1024) words, and a block size of 1 word, 10 bits are used to index the cache, leaving 32-10=20 bits to be compared against the tag.**
  - **If the tag and upper 20 bits of the address are equal and the valid bit is on, then the request hits in the cache, and the word is supplied to the processor. Otherwise, a miss occurs.**
  - **The byte offset (2 lower bits) in the address are used to select a byte within a word (a word = 4 bytes(32bits)).**

20

Direct Mapped Cache – In MIPS

- The index of a cache block, together with the tag contents of that block, uniquely specify the memory address of the word contained in the cache block.
- Because the index field is used as an address to access the cache and because an n-bit field has 2^n values, the total number of entries in the cache must be a power of 2.
- In the MIPS architecture, the least significant 2 bits of every address specify a byte within a word and are not used to select the word in the cache.

Direct Mapped Cache – An Example

- Assuming a 32-bit address, a direct-mapped cache of size 2^n words (number of entries) with one-word (4-byte) blocks (data block size) will require a tag field whose size is 32-(n+2) bits, because 2 bits are used for the byte offset and n bits are used for the index.
- The total number of bits in a direct-mapped cache is 2^n x (block size + tag size + valid field size).
- Since the block size is one word (32 bits) and the address size is 32 bits, the number of bits in such a cache is $2^n \times (32 + (32 - n - 2) + 1) = 2^n \times (63 – n)$.

Example: Bits in Cache

- How many total bits are required for a direct-mapped cache with 64KByte of data and one-word blocks, assuming a 32-bit address?
- Answer:
  - We know that 64KB is 64K/4 = 16K words, which is $2^{10} \times 2^4 = 2^{14}$ words, and with a block size of one word, 2^{14} blocks.
  - Each block has 32 bits of data plus a tag, which is 32 – 14 – 2 bits, plus a valid bit.
  - Thus the total cache size is
    $2^{14} \times (32 + (32 - 14 - 2) + 1) = 2^{14} \times 49 = 784 \times 2^{10} = 784$ Kbits
    or 784K/8 = 98KByte for a 64-KByte cache.
  - For this cache, the total number of bits in the cache is over 1.5 times as many needed just for the storage of the data.

Handling Cache Misses

- The control unit must detect a miss and process the miss by fetching the data from memory.
- Thus, the steps to be taken on an instruction cache miss are:

1. Send the original PC value (current PC – 4) to the memory.
2. Instruct main memory to perform a read and wait for the memory to complete its access.
3. Write the cache entry, putting the data from memory in the data portion of the entry, writing the upper bits of the address (from the ALU) into the tag field, and turning the valid bit on.
4. Restart the instruction execution at the first step, which will refetch the instruction, this time finding it in the cache.

Handling Cache Misses – Stall in use

- On a miss, we stall the processor until the memory responds with the data.
- To reduce the number of cycles that a processor is stalled for a cache miss, we can allow a processor to continue executing instructions while the cache miss is handled.
- This strategy does not help for instruction misses because we cannot fetch new instructions to execute.
- In the case of data misses, however, we can allow the machine to continue fetching and executing instructions until the loaded word is required. Here the name for this technique: ***stall in use***.
- While this additional effort may save cycles, it will probably not save very many cycles because the loaded data will likely be needed very shortly and because other instructions will need access to the cache.

Data inconsistent between cache and memory

- Data inconsistent : When we wrote data into only data cache (without changing main memory), then after write into cache, memory would have different value from that in the cache.
- Solution: To keep the main memory and the cache consistent is always write the data into both the memory and the cache, this is called write-through.
- With a write-through scheme, every write causes the data to be written to main memory.
- A problem: These writes will take a long time and could slow down the machine.

Data inconsistent (Continues)

- One solution is to use a write buffer:
  - A write buffer stores the data while it is waiting to be written to memory.
  - After writing data into cache and into write buffer, the processor can continue execution.
  - When a write to main memory completes, the entry in the write buffer is freed.
  - If the write buffer is full when the processor reaches a write, the processor must stall until there is an empty position in the write buffer.

Data inconsistent (Continues)

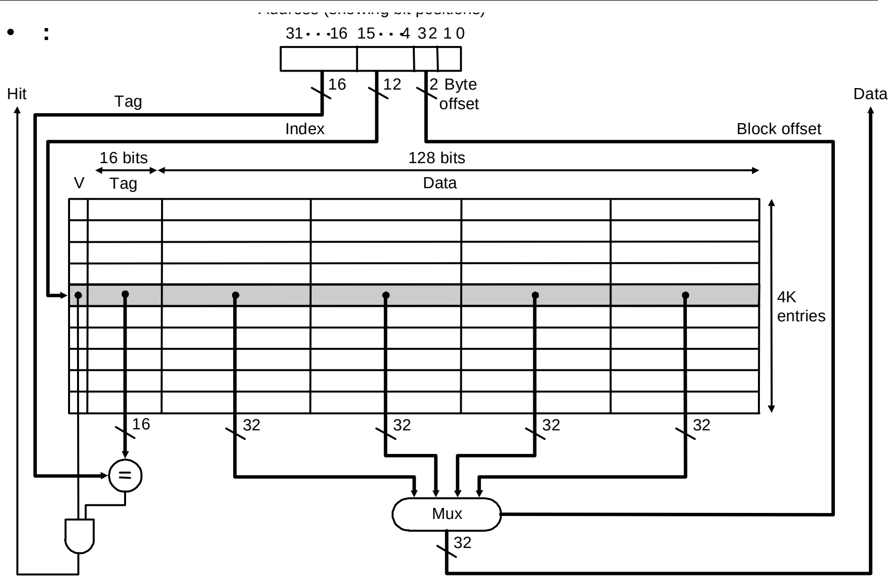
- The alternative to write-through is write-back scheme.
- In write-back scheme:
  - When a write occurs, the new value is written only to the block in the cache.
  - The modified block is written to the lower level of the hierarchy when it is replaced.
  - Write-back schemes can improve performance , especially when processor can generate writes as fast or faster than the writes can be handled by main memory.
  - That is write-back is faster than write-through.
  - Write-back is more complex to implement than write-through.

Taking Advantage of Spatial Locality

- Spatial locality exists naturally in programs.
- To take advantage of spatial locality, we want to have a cache block that is larger than one word in length.
- When a miss occurs, we will then fetch multiple words that are adjacent and carry a high probability of being needed shortly.
- Next slide shows a cache that holds 64KB of data, but with blocks of four words (16 bytes) each.

Direct Mapped Cache: Taking advantage of spatial locality

- :



Cache with multiword block

- Comparing the cache with multiword block (Slide 30) with cache single word block (Slide 19):
  - An extra block index field occurs in the address of the cache in Slide 30 (bits 2 and 3 in this case).
  - This block index field is used to control the multiplexer, which selects the requested word from the 4 words in the indexed block.
  - The total number of tags and valid bits in the cache with a multiword block is smaller than cache with a single word block because each tag and valid bit is used for 4 words.

Cache with multiword block (Continues)

- How de we find the cache block for a particular address?
  - We can use same mapping that we used for a cache with a one-word block:

    `(block address) modulo (Number of cache blocks)`
  - The block address is the word address divided by the number of words in the block
  - Or equivalently, the byte address divided by the number of bytes in the block.

Example: Mapping an Address to a Multiword Cache Block

- Consider a cache with 64 blocks and a block size of 16 bytes (4 words). What block number does byte address 1200 map to?
- Answer: The block is given by:

```
    (block address) modulo (Number of cache blocks)
```

  Where the address of the block is:

```
    Byte address / Bytes per block
```

  Notice that this block address is the block containing all addresses between:

  {Flour of (Byte address / Bytes per block)} x Bytes per block

  and

  {Flour of (Byte address / Bytes per block)} x Bytes per block + (Bytes per block – 1)

Answer: Continues

- That is between 1200 and 1215.
- Thus, with 16 bytes per block, byte address 1200 is block address:

  {Flour of 1200 / 16} = 75

  which maps to cache block number:

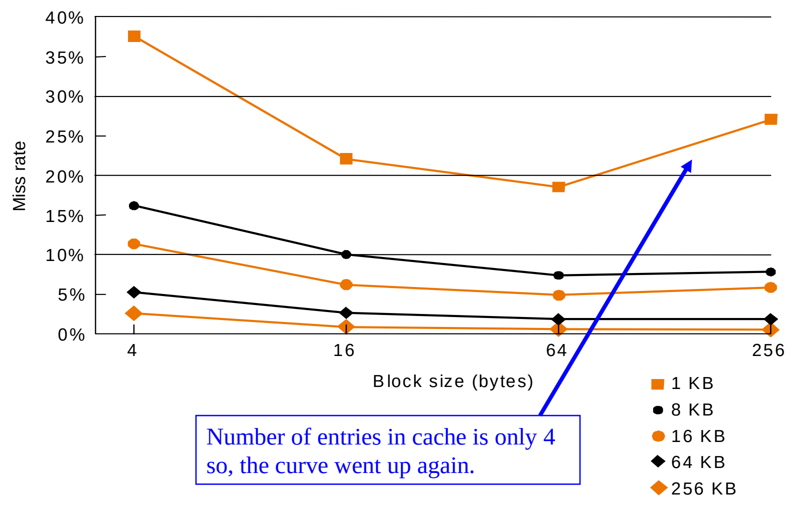
  (75 modulo 64) = 11

Multiword Cache Block Performance

- The reason for increasing the block size was to take advantage of spatial locality to improve performance.
- How does a larger block size affect performance?
- In general, the miss rate falls when we increase the block size.
- Example: Suppose the following addresses are requested by a program: 16, …, 24, …, 20 and none of these addresses are in the cache.
- If the cache has four word block, then the miss to address 16 will cause the block containing addresses 16, 20, 24, and 28 to be loaded into the cache.
- Only one miss is encountered for the three references
- With a one-word block, two additional misses are required because each miss brings in only a single word.

Multiword Cache Block Performance Continues

- Increasing the block size tends to decrease miss rate:



Performance Continues

- A problem associated with increasing block size is the cost of a miss increases.
- The miss penalty is determined by the time required to fetch the block from the next lower level of the hierarchy and load it into the cache.
- The time to fetch the block has two parts:
  - The latency to the first word and
  - The transfer time for the rest of the block.
- Clearly, the transfer time (miss penalty) will increase as the block size grows and thus cache performance decreases.

Designing Memory System to Support Caches

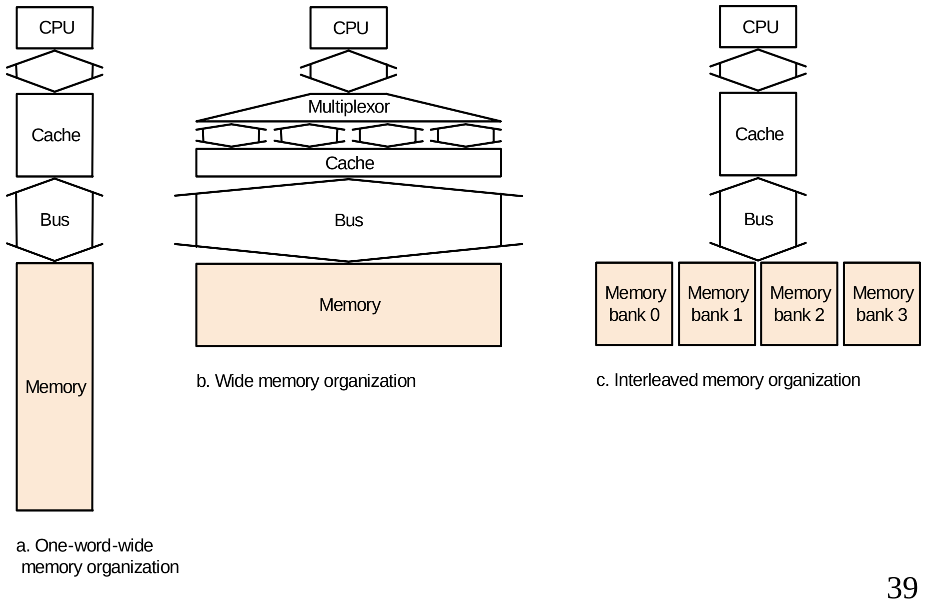
- We can reduce the miss penalty if we increase the bandwidth from the memory to the cache.
- To understand the impact of different organizations for memory, let's define a set of memory access times:
  - 1 clock cycle to send the address
  - 15 clock cycles for each DRAM access initiated.
  - 1 clock cycle to send a word of data.
- Example: If we have a cache of 4 words and one-word-wide bank of DRAMs, the miss penalty would be:

  1 + 4 x 15 + 4 x 1 = 65 clock cycles.

  Thus, the number of bytes transferred per clock cycle for a single miss would be:

  (4 words x 4 bytes) / 65 = 0.25

Three Options for Designing Memory System



Achieving Higher Memory Bandwidth

- **From previous slide, the first option (<u>one-word-wide memory organization</u>), memory is one word wide, and all accesses are made sequentially.**
- **Second option (<u>wide memory organization</u>), increases the bandwidth to memory by widening the memory and the buses between the processor and memory; this allows parallel access to all the words of the block.**
- **Third option (<u>interleaved memory organization</u>), increases the bandwidth by widening the memory but not the interconnection bus.**

Wide Memory Organization Performance

- Increasing the width of the memory and the bus will increase the memory bandwidth proportionally, decreasing both the access time and transfer time portions of the miss penalty.
- From our previous example, with a main memory of two words, the miss penalty drops from 65 clock cycles to 1 + 2 x 15 + 2 x 1 = 33 clock cycles. Thus, the number of bytes transferred per clock cycle for a single miss would be: (4 words x 4 bytes) / 33 = 0.48
- However, with four-word-wide memory, the miss penalty is 1 + 1 x 15 + 1 x 1 = 17 clock cycles. Thus, the number of bytes transferred per clock cycle for a single miss would be: (4 words x 4 bytes) / 17 = 0.94

Interleaving Memory Organization Performance

- Instead of making the entire path between the memory and cache wider, the memory chips can be organized in banks to read or write multiple words in one access time rather than reading or writing a single word each time.
- Each bank could be one word wide so that the width of the bus and the cache need not change, but sending an address to several banks permits them all to read simultaneously. This scheme is called interleaving.
- Example: with four banks, the time to get a four-word block would consist of 1 cycle to transmit the address and read request to the banks, 15 cycles for all four banks to access memory, and 4 cycles to send the four words back to the cache.

  This yields a miss penalty of 1 + 1 x 15 + 4 x 1 = 20 clock cycles

  This is an effective bandwidth per miss of (4 x 4) / 20 = 0.80 bytes per clock.

Reducing Cache Misses by More Flexible Placement of Blocks

- Direct Mapped: A block can go in exactly one place in the cache, that is there is a direct mapping from any block address in memory to a single location in the upper level of the hierarchy.

- Fully Associative: A block can be placed in any location in the cache, that is a block in memory may be associated with any entry in the cache.

  To find a given block in a fully associative cache, all the entries in the cache must be searched because a block can be placed in any one.

  To make the search practical, it is done in parallel with a comparator associated with each cache entry.

Reducing Cache Misses …. (Continues)

- Set Associative: This is the middle range of designs between direct mapped and fully associative, because there are a fixed number of locations (at least two) where each block can be placed.

  A set-associative cache with n locations for a block is called an ***n-way set-associative cache***.

  An n-way set-associative cache consists of a number of sets, each of which consists of n blocks. Each block in the memory maps to a unique set in the cache given by the index field, and a block can be placed in any element of that set.

Reducing Cache Misses …. (Continues)

- **A <u>set-associative placement</u> combines direct-mapped placement and fully associative placement:**

  A block is directly mapped into a set, and then all the blocks in the set are searched for a match.

- **Remember that <u>in a direct-mapped</u> cache, the position of a memory block is given by:**

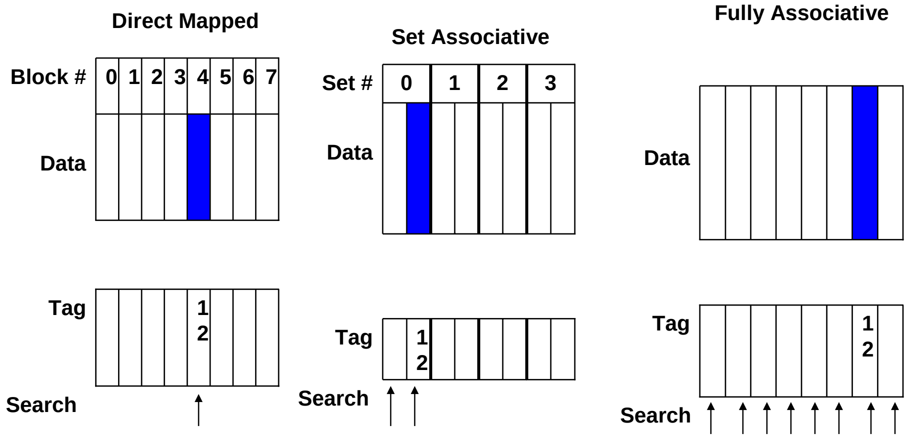
  (Block number) modulo (Number of cache <u>blocks</u>)

- **In a set-associative cache, the set containing a memory block is given by:**

  (Block number) modulo (Number of <u>sets</u> in the cache)

Example: On Placement

- **The location of a memory block whose address is 12 in a cache with 8 blocks varies for direct-mapped, set-associative, and fully associative placement.**



Example: Continues

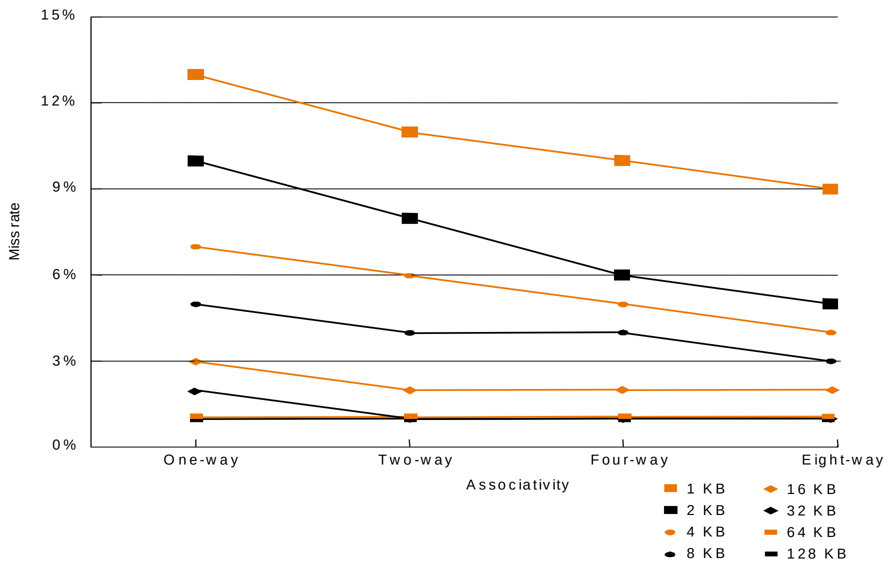
- In <u>direct-mapped</u> placement, there is only one cache block where memory block 12 can be found, and that block is given by (12 mod 8) = 4.
- In a <u>two-way set-associative</u> cache, there would be 4 sets, and memory block must be in set

  (12 mod 4) = 0; the memory block could be in either element of the set.
- In <u>fully associative</u> placement, the memory block for block address 12 can appear in any of the 8 cache blocks.

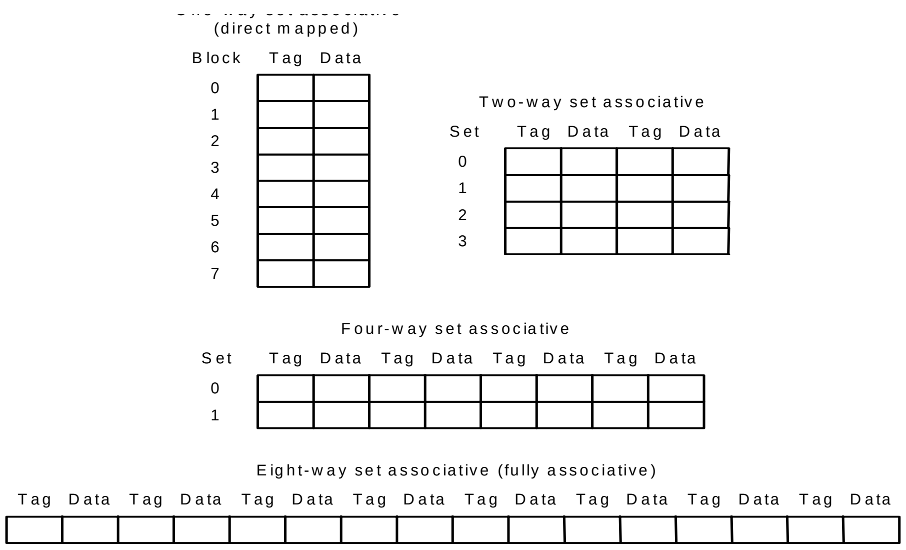
Mapping Techniques

- A <u>direct-mapped</u> cache is a one-way set-associative cache: each cache entry holds one block and forms a set with one element.
- A <u>fully associative</u> cache with m entries is an m-way set-associative cache: it has one set with m blocks, and an entry can reside in any block within that set.
- The advantage of increasing the degree of associativity is that it decreases the miss rate (See next slide).
- Slide 50 shows the possible associativity structures for an eight-block cache.

Performance: Associativity versus Miss Rate



An 8-block cache configured as direct-mapped, 2-way set associative, 4-way set associative, & fully associative



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

Set Associativity

- Total size of the cache in blocks is equal to the number of sets times the associativity.
- For a fixed cache size, increasing the associativity decreases the number of sets, while increasing the number of elements per set.
- From previous slide, with eight blocks, an eight-way set associative cache is the same as a fully associative cache.

Example: Associativity in Caches

- There are 3 small caches, each consisting of 4 one-word blocks.
  - One cache is fully associative
  - Second is two-way set associative
  - Third is direct mapped
- Find the number of misses for each organization given the following sequence of block addresses:

  0, 8, 0, 6, 8

Answer: The direct-mapped case is easiest.

- First let's determine to which cache block each block address maps:

Block address	Cache block
0	(0 mod 4) = 0
6	(6 mod 4) = 2
8	(8 mod 4) = 0

Answer: (Continues)

- Now we can fill in cache contents after each reference: using a blank entry to mean that the block is invalid and colored entry to show a new entry added to cache for the associative reference.
- The direct-mapped cache generates 5 misses.

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	Miss	Memory[0]			
8	Miss	Memory[8]			
0	Miss	Memory[0]			
6	Miss	Memory[0]		Memory[6]	
8	Miss	Memory[8]		Memory[6]	

Answer: The two-way set-associative cache.

- The set-associative has two sets with indices 0 and 1 with two elements per set.
- Let's first determine to which set each block address maps:

Block address	Cache set
0	(0 mod 2) = 0
6	(6 mod 2) = 0
8	(8 mod 2) = 0

Answer: (Continues)

- Because we have a choice of which entry in a set to replace on a miss, we need a replacement rule.
- Set-associative caches usually replace the least recently used block within a set; that is, the block that was used furthest in the past is replaced.
- Using this replacement rule, the contents of the set-associative cache after reference looks like this:

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	Miss	Memory[0]			
8	Miss	Memory[0]	Memory[8]		
0	Hit	Memory[0]	Memory[8]		
6	Miss	Memory[0]	Memory[6]		
8	Miss	Memory[8]	Memory[6]		

Answer: Continues

- Notice that when block 6 is referenced, it replaces block 8, since block 8 has been less recently referenced than block 0.
- The two-way set-associative cache has a total of 4 misses, one less than the direct-mapped cache.

Answer: Fully associative cache

- The fully associative cache has 4 cache blocks (in a single set); any memory block can be stored in any cache block.
- The fully associative cache has the best performance, with only 3 misses:

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	Miss	Memory[0]			
8	Miss	Memory[0]	Memory[8]		
0	Hit	Memory[0]	Memory[8]		
6	Miss	Memory[0]	Memory[8]	Memory[6]	
8	Hit	Memory[0]	Memory[8]	Memory[6]	

Answer: (Continues)

- For this series of references (0,8,0,6,8), three misses is the best we can do because three unique block addresses are accessed.
- Notice that if we had 8 blocks in the cache, there would be no replacement in the two-way set-associative cache (check this for yourself), and it would have the same number of misses as the fully associative cache.
- Similarly, if we had 16 blocks, all three caches would have the same number of misses (check it out yourself).
- This change in miss rate shows us that cache size and associativity are not independent in determining cache performance.

Locating a Block in the Cache

- Finding a block in a cache that is set associative.
  - Three portions of an address in a set-associative or direct-mapped cache:

Tag	Index	Block Offset

  - The <u>index</u> is used to select the set
  - Then the <u>tag</u> is used to choose the block by comparison with the blocks in the selected set.
  - Because speed is of the essence, all the tags in the selected set are searched in parallel.
  - The <u>block offset</u> is the address of the desired data within the block.

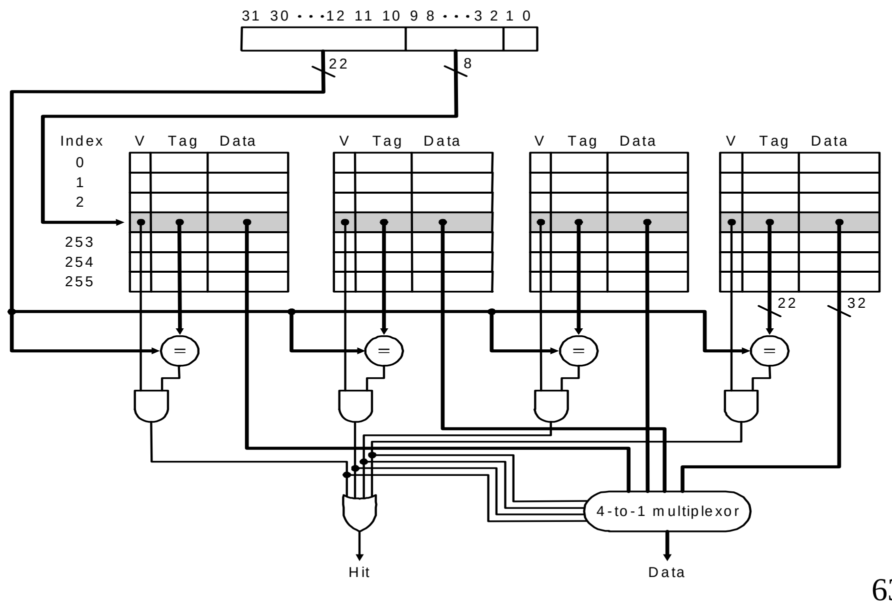
Locating a Block in Cache (Continues)

- If the total size is kept the same, increasing the associativity increases the number of blocks per set, which is the number of simultaneous compares needed to perform the search in parallel:
  - Each increase by a factor of two in associativity doubles the number of blocks per set and halves the number of sets.
  - Accordingly, each factor-of-two increase in associativity decreases the size of index by 1 bit and increases the size of the tag by 1 bit.

Locating a Block in Cache (Continues)

- In a fully associative cache, there is effectively only one set, and all the blocks must be checked in parallel.
- Thus, there is no index, and the entire address, excluding the block offset, is compared against the tag of every block. That is we search the entire cache without any indexing.
- In a direct-mapped cache, only a single comparator is needed, because the entry can be in only one block, and we access the cache simply by indexing.
- In a 4-way set-associative cache, 4 comparators are needed, together with 4-to-1 multiplexor to choose the 4 potential set members of the selected set (See next slide).

Implementation of 4-way set-associative cache requires 4 comparators and 4-to1 multiplexor.



Example: Size Tags versus Set Associativity

- Increasing associativity requires more comparators, as well as, more tag bits per cache block.
- Assuming a cache of 4K blocks, a 4-word block size, and a 32-bit address.
- Find the total number of sets and the total number of tag bits for caches that are direct mapped, 2-way and 4-way set associative, and fully associative.

Answer: For Direct-Mapped Cache

- Since there are 16 ($=2^4$) bytes per block, a 32-bit address yields 32 – 4 = 28 bits to be used for index and tag.
- The direct-mapped cache:
  - has the same number of sets as blocks;
  - hence 12 bits of index, since $\log_2$ (4K) = 12 (or 4K = $2^{10} \times 2^2 = 2^{12}$);
  - hence the total number of tag bits is (28 – 12) x 4K (sets or entries) = 64 Kbits.

Answer: For 2-way, 4-way, & fully Set-Associative Cache

- Each degree of associativity decreases the number of sets by a factor of 2 and thus decreases the number of bits used to index the cache by 1 and increases the number of bits in the tag by 1.
- Thus, for a 2-way set-associative cache, there are 2K sets, and the total number of tag bits is

  (28 – 11) x 2 (blocks) x 2K (sets) = 68Kbits.
- For a 4-way set associative cache, the total number of sets is 1K, and the total number of tag bits is

  (28 – 10) x 4 (blocks) x 1K (sets) = 72Kbits.
- For a fully associative cache, there is only one set with 4K blocks, and the tag is 28 bits, leading to a total of 28 x 4K (blocks) x 1 (set) = 112K tag bits.

Choosing Which Block to Replace

- In a <u>direct-mapped</u> cache, when a miss occurs, the requested block can go in exactly one position, and the block occupying that position must be replaced.
- In an <u>associative</u> cache, we have a choice of where to place the requested block, and hence a choice of which block to replace.
  - In a fully associative cache, all blocks are candidates for replacement.
  - In a set-associative cache, we must choose among the blocks in the selected set.
- The most common used scheme is least recently used (LRU).
- In an LRU scheme the block replaced is the one that has been unused for the longest time.