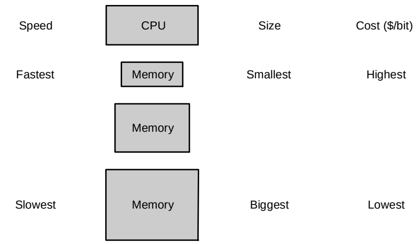
Chapter 7
Large and Fast: Exploiting Memory Hierarchy

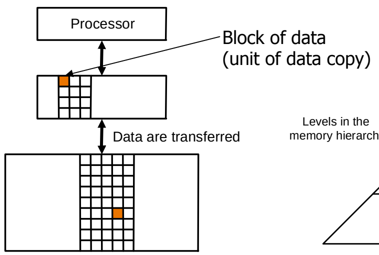
Memories: Review

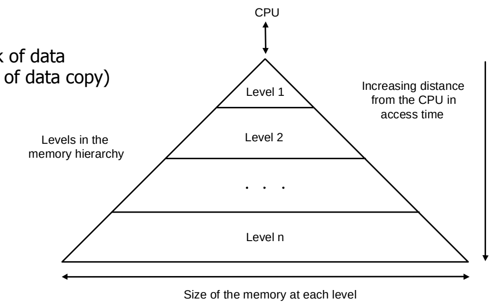
- *DRAM* (Dynamic Random Access Memory):
  - value is stored as a charge on capacitor that must be *periodically refreshed*, which is why it is called *dynamic*
  - very small – 1 transistor per bit – but factor of 5 to 10 slower than SRAM
  - used for *main memory*
- *SRAM* (Static Random Access Memory):
  - value is stored on a pair of inverting gates that will *exist indefinitely* as long as there is power, which is why it is called *static*
  - very fast but takes up more space than DRAM – 4 to 6 transistors per bit
  - used for *cache*



Memory Hierarchy

- Users want large and fast memories…
  - expensive and they don't like to pay…
- Make it seem like they have what they want…
  - *memory hierarchy*
  - hierarchy is *inclusive*, every level is *subset* of lower level
  - performance depends on *hit rates*





Locality

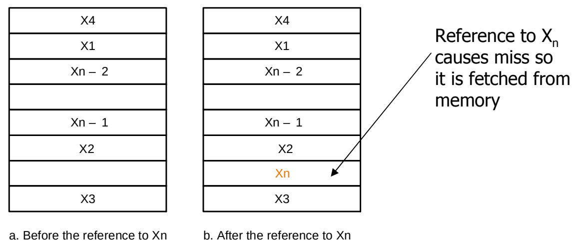
- *Locality* is a principle that makes having a memory hierarchy a good idea
- If an item is referenced then because of
  - *temporal locality*: it will tend to be *again* referenced soon
  - *spatial locality*: *nearby items* will tend to be referenced soon
  - *why does code have locality – consider instruction and data?*

Hit and Miss

- Focus on *any two adjacent* levels – called, *upper* (closer to CPU) and *lower* (farther from CPU) – in the memory hierarchy, because each block copy is always between two adjacent levels
- Terminology:
  - *block*: minimum unit of data to move between levels
  - *hit*: data requested is in upper level
  - *miss*: data requested is not in upper level
  - *hit rate*: fraction of memory accesses that are hits (i.e., found at upper level)
  - *miss rate*: fraction of memory accesses that are not hits
    - miss rate = 1 – hit rate
  - *hit time*: time to determine if the access is indeed a hit + time to access and deliver the data from the upper level to the CPU
  - *miss penalty*: time to determine if the access is a miss + time to replace block at upper level with corresponding block at lower level + time to deliver the block to the CPU

Caches

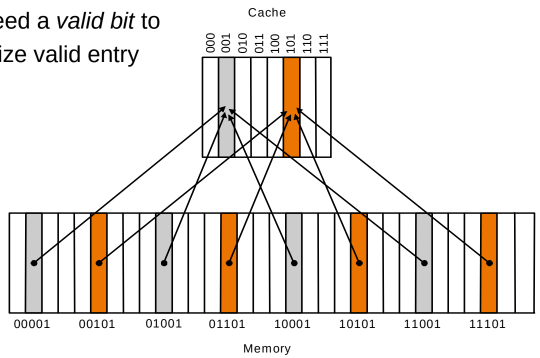
- By simple example
  - assume block size = one word of data



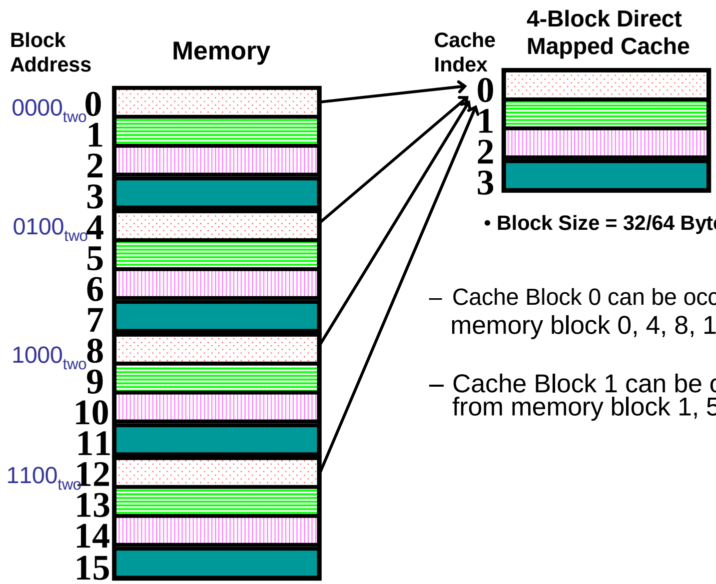
- Issues:
  - *how do we know if a data item is in the cache?*
  - *if it is, how do we find it?*
  - *if not, what do we do?*
- Solution depends on *cache addressing scheme*…

Direct Mapped Cache

- Addressing scheme in *direct mapped* cache:
  - cache block address = memory block address *mod* cache size (*unique*)
  - if cache size = 2^m, cache address = lower m bits of n-bit memory address
  - remaining upper n-m bits kept kept as *tag bits* at each cache block
  - also need a *valid bit* to recognize valid entry



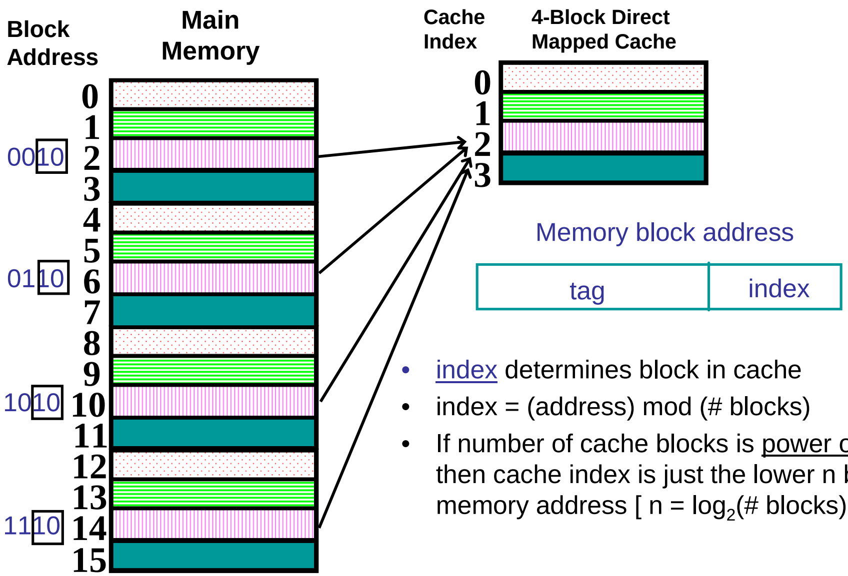
Simplest Cache: Direct Mapped



- Block Size = 32/64 Bytes

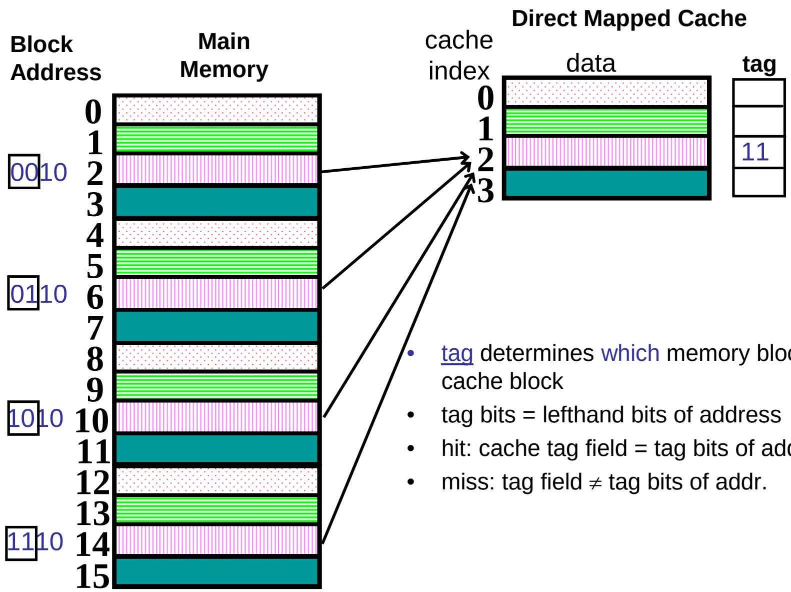
- Cache Block 0 can be occupied by data from memory block 0, 4, 8, 12
- Cache Block 1 can be occupied by data from memory block 1, 5, 9, 13

Simplest Cache: Direct Mapped



- index determines block in cache
- index = (address) mod (# blocks)
- If number of cache blocks is power of 2, then cache index is just the lower n bits of memory address [$n = \log_2(\# \text{ blocks})$]

Simplest Cache: Direct Mapped w/Tag



- tag determines which memory block occupies cache block
- tag bits = lefthand bits of address
- hit: cache tag field = tag bits of address
- miss: tag field ≠ tag bits of addr.

Accessing Cache

- Example:

(0) Initial state:

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

(1) Address referred 10110 (*miss*):

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Mem(10110)
111	N		

(2) Address referred 11010 (*miss*):

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem(11010)
011	N		
100	N		
101	N		
110	Y	10	Mem(10110)
111	N		

(3) Address referred 10110 (*hit*):

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem(11010)
011	N		
100	N		
101	N		
110	Y	10	Mem(10110)
111	N		

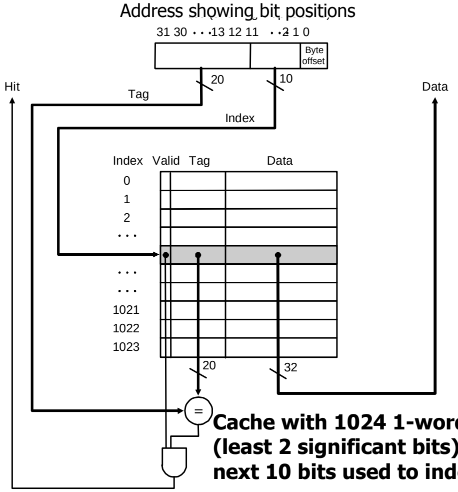
to CPU

(4) Address referred 10010 (*miss*):

Index	V	Tag	Data
000	N		
001	N		
010	Y	10	Mem(10010)
011	N		
100	N		
101	N		
110	Y	10	Mem(10110)
111	N		

Direct Mapped Cache

- MIPS style:



Cache with 1024 1-word blocks: *byte offset* (least 2 significant bits) is ignored and next 10 bits used to index into cache

Cache Read Hit/Miss

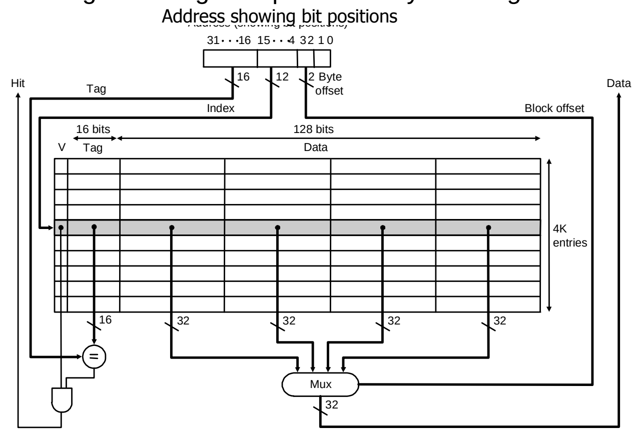
- *Cache read hit*: no action needed
- *Instruction cache read miss*:
  1. *Send original PC value* (*current PC – 4*, as PC has already been incremented in first step of instruction cycle) to memory
  2. Instruct main memory to perform read and wait for memory to complete access – *stall* on read
  3. After read completes *write cache* entry
  4. *Restart* instruction execution at first step to refetch instruction
- *Data cache read miss*:
  – Similar to instruction cache miss
  – To reduce data miss penalty allow processor to execute instructions while waiting for the read to complete *until* the word is required – *stall on use* (why won't this work for instruction misses?)

Cache Write Hit/Miss

- *Write-through* scheme
  – on *write hit*: replace data in cache *and* memory with *every* write hit to avoid *inconsistency*
  – on *write miss*: write the word into cache *and* memory – obviously no need to read missed word from memory!
  – Write-through is slow because of always required memory write
    - performance is improved with a *write buffer* where words are stored while waiting to be written to memory – processor can continue execution until write buffer is full
    - when a word in the write buffer completes writing into main that buffer slot is freed and becomes available for future writes
    - DEC 3100 write buffer has 4 words
- *Write-back* scheme
  – write the data block *only* into the cache and *write-back* the block to main *only when* it is replaced in cache
  – more efficient than write-through, more complex to implement

Direct Mapped Cache: Taking Advantage of Spatial Locality

- Taking advantage of spatial locality with *larger* blocks:



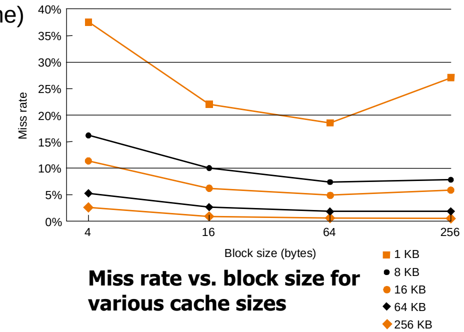
Cache with 4K 4-word blocks: *byte offset* (least 2 significant bits) is ignored, next 2 bits are *block offset*, and the next 12 bits are used to index into cache

Direct Mapped Cache: Taking Advantage of Spatial Locality

- *Cache replacement* in large (multiword) blocks:
  - word *read miss*: read entire block from main memory
  - word *write miss*: *cannot* simply write word and tag! *Why*?!
  - writing in a *write-through* cache:
    - if *write hit*, i.e., tag of requested address and and cache entry are equal, continue as for 1-word blocks by replacing word and writing block to both cache and memory
    - if *write miss*, i.e., tags are unequal, fetch block from memory, replace word that caused miss, and write block to both cache and memory
    - therefore, unlike case of 1-word blocks, a write miss with a multiword block causes a memory read

Direct Mapped Cache: Taking Advantage of Spatial Locality

- Miss rate falls at first with increasing block size as expected, but, as block size becomes a large fraction of total cache size, miss rate may go up because
  - there are few blocks
  - competition for blocks increases
  - blocks get ejected before most of their words are accessed (*thrashing* in cache)



Miss rate vs. block size for various cache sizes

Example Problem

- *How many total bits are required for a direct-mapped cache with 128 KB of data and 1-word block size, assuming a 32-bit address?*

- Cache data = 128 KB = 2^{17} bytes = 2^{15} words = 2^{15} blocks
- Cache entry size = block data bits + tag bits + valid bit
  = 32 + (32 – 15 – 2) + 1 = 48 bits
- Therefore, cache size = $2^{15} \times 48$ bits = $2^{15} \times (1.5 \times 32)$ bits = 1.5×2^{20} bits = 1.5 Mbits
  - data bits in cache = 128 KB × 8 = 1 Mbits
  - total cache size/actual cache data = 1.5

Example Problem

- *How many total bits are required for a direct-mapped cache with 128 KB of data and 4-word block size, assuming a 32-bit address?*

- Cache size = 128 KB = 2^{17} bytes = 2^{15} words = 2^{13} blocks
- Cache entry size = block data bits + tag bits + valid bit
  = 128 + (32 – 13 – 2 – 2) + 1 = 144 bits
- Therefore, cache size = $2^{13} \times 144$ bits = $2^{13} \times (1.25 \times 128)$ bits = 1.25×2^{20} bits = 1.25 Mbits
  - data bits in cache = 128 KB × 8 = 1 Mbits
  - total cache size/actual cache data = 1.25

Improving Cache Performance

- Use split caches for instruction and data because there is more spatial locality in instruction references:

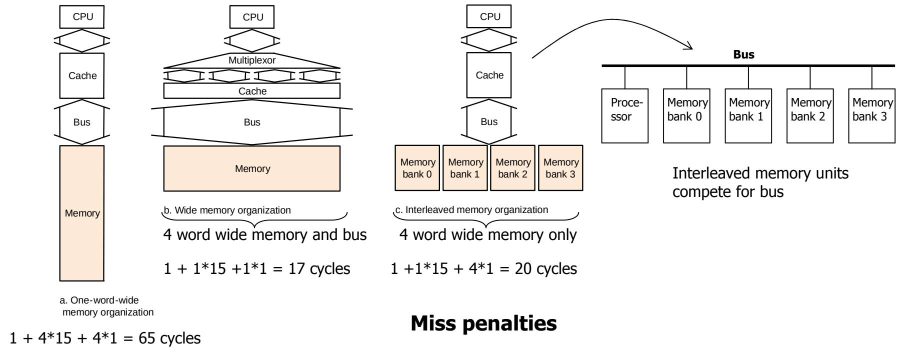
Program	Block size in words	Instruction miss rate	Data miss rate	Effective combined miss rate
gcc	1	6.1%	2.1%	5.4%
	4	2.0%	1.7%	1.9%
spice	1	1.2%	1.3%	1.2%
	4	0.3%	0.6%	0.4%

Miss rates for gcc and spice in a MIPS R2000 with one and four word block sizes

- Make reading multiple words (higher bandwidth) possible by increasing physical or logical width of the system…

Improving Cache Performance by Increasing Bandwidth

- Assume:
  - cache block of 4 words
  - 1 clock cycle to send address to memory address buffer (1 bus trip)
  - 15 clock cycles for each memory data access
  - 1 clock cycle to send data to memory data buffer (1 bus trip)



Miss penalties

Performance

- Simplified model assuming equal read and write miss penalties:
  - CPU time = (execution cycles + memory stall cycles) x cycle time
  - memory stall cycles = memory accesses x miss rate x miss penalty
- Therefore, two ways to improve performance in cache:
  - decrease miss rate
  - decrease miss penalty
  - *what happens if we increase block size?*

Example Problems

- Assume for a given machine and program:
  - instruction cache miss rate 2%
  - data cache miss rate 4%
  - miss penalty always 40 cycles
  - CPI of 2 without memory stalls
  - frequency of load/stores 36% of instructions

1. *How much faster is a machine with a perfect cache that never misses?*
2. *What happens if we speed up the machine by reducing its CPI to 1 without changing the clock rate?*
3. *What happens if we speed up the machine by doubling its clock rate, but if the absolute time for a miss penalty remains same?*

Solution

1.

- Assume instruction count = I
- Instruction miss cycles = I × 2% × 40 = 0.8 × I
- Data miss cycles = I × 36% × 4% × 40 = 0.576 × I
- So, total memory-stall cycles = 0.8 × I + 0.576 × I = 1.376 × I
  - in other words, 1.376 stall cycles per instruction
- Therefore, CPI with memory stalls = 2 + 1.376 = 3.376
- Assuming instruction count and clock rate remain same for a perfect cache and a cache that misses:

  CPU time with stalls / CPU time with perfect cache

  = 3.376 / 2 = 1.688
- Performance with a perfect cache is better by a factor of 1.688

Solution (cont.)

2.

- CPI without stall = 1
- CPI with stall = 1 + 1.376 = 2.376 (clock has not changed so stall cycles per instruction remains same)
- CPU time with stalls / CPU time with perfect cache

  = CPI with stall / CPI without stall

  = 2.376
- Performance with a perfect cache is better by a factor of 2.376

Solution (cont.)

3.

- With doubled clock rate, miss penalty = 2 × 40 = 80 clock cycles
- Stall cycles per instruction = (I × 2% × 80) + (I × 36% × 4% × 80)

  = 2.752 × I
- So, faster machine with cache miss has CPI = 2 + 2.752 = 4.752
- CPU time with stalls / CPU time with perfect cache

  = CPI with stall / CPI without stall

  = 4.752 / 2 = 2.376
- Performance with a perfect cache is better by a factor of 2.376