Cache Memory

- The memory used in a computer consists of a hierarchy
- Fastest/Nearest CPU Registers
- Cache (may have levels itself)
- Main Memory
- Slowest/Furthest Virtual Memory (on disc)
- Fast CPUs require very fast access to memory
  - we have seen this with the DLX machine
  - access to data cache
- One other way for fast data access is the use of large register sets
  - this is typical of RISC architectures (i.e. programmer architectures)
  - and the set of actual registers is larger than this
    - because of re-allocation/renaming
- and another aspect of this is the use of cache memory
  - or indeed, the use of multiple cache memories

Basic concepts

- recently used instructions and data are kept in a very fast memory so that the CPU does not have to access the main memory every time it requires access to data
- the amount of data that can be held is such a *cache* is limited
  - generally, it needs to be on-chip to be effective
  - it needs to be accessed in one cycle
  - so the size is limited by what can be placed on (the rest of) the chip
- the whole advantage of the cache is predicated on *locality of reference*
  - that is, that instructions and data recently used is likely to be used again soon
  - instructions clearly show locality
    - the great majority of executed instructions are inside loops
  - data also shows locality
    - though the advantage is a little less than the advantage of an instruction cache

Aspects of Caches

- Transparency
  - the cache should not be visible to the programmer at all
  - it should take advantage of general characteristics of programs
    - not require programs to be specially designed to take advantage of it
    - this does mean that it is possible to write cache-defeating programs!
- Hit Ratio
  - If a memory access finds the datum in the cache, this is a *hit*
  - if not, this is a *miss*
  - the hit ratio is defined to be

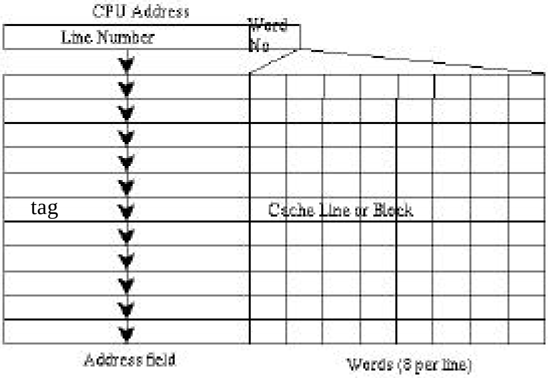
$$\frac{\text{Hits}}{\text{Hits + Misses}}$$

Clearly, high hit ratios (near one) are desirable.

Questions about caches

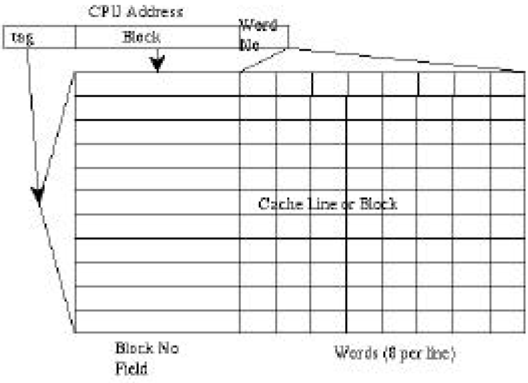
- there are 4 basic questions that a cache designer needs to answer in designing a cache
  - Where should a block be placed?
  - How do we find a block in the cache?
  - Which block should we replace on a miss?
  - What happens on a write?
- The answers to these questions define the type of cache in use
- If a block of memory from the main memory can be placed in exactly one place, we have a cache which is direct-mapped
- If the block can be placed anywhere, the cache is fully associative
- If there are a restricted set of places that the block can be placed, the cache is set associative
- We will look at each of these in turn.

Fully associative caches



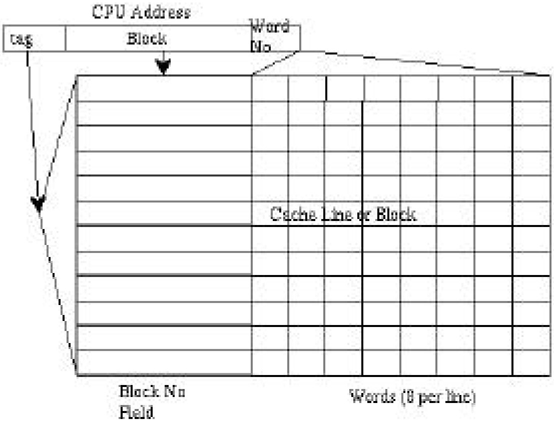
- Cache line contains more than one word (8 here)
- CPU address must identify
  - which (if any) line has the word
  - which word is required
- Most significant part of address identifies the cache line
- less significant part identifies the word in the line
  - here, 29, 3 bits
- An associative memory search identifies which if any line holds the address
  - all the cache block address *tags* are compared with the CPU address simultaneously
  - this is essential to fast operation

Direct mapping caches



- Each block can only be inserted in one place
- No associative mapping
- CPU address is in 3 parts:
  - tag identifies exactly 1 cache block
  - block no is compared with the block no field in the cache line
  - word (least significant) identifies the word in the cache line
- For a 128-line cache tag field would be 7 bits,
- block field 22 bits, and word field 3 bits.
- Simpler hardware then associative cache

Set-Associative Cache



- 2-way set-associative cache
- Tag address selects one out of 2 possible lines
- Block number is associatively compared with these 2 block ids
- For a 128 line cache, we have a 6-bit (not 7) tag field
- Block field is 23 bits
- This is a compromise between direct and fully associative caches.

Comparing cache techniques (I)

- On hardware complexity:
  - Fully associative cache requires special fast associative memory hardware
  - Direct mapping caches are much simpler in hardware terms
  - Set-associative caches offer a compromise
- On usefulness
  - direct mapping caches cannot normally cache blocks N, N+1 from main memory (since they would go into the same cache line)
    - This is a serious problem: many loops are bigger than one cache line, resulting ina cache miss (and cache reload) during the loop
    - This reduces the effectiveness of the cache
  - fully associative caches do not suffer from this problem at all (but are complex)
  - set-associative caches again proffer a compromise

On replacement algorithms (II)

- straightforward for direct mapping caches
  - which is a problem in its own right
- for set and fully associative caches, we need to decide which cache line to overwrite.
  - Random replacement
    - choose one at random and replace it.
    - Simple, easy to implement
  - Least Recently Used
    - choose the line read least recently, and replace it
    - requires a counter associated with each cache line, and this is relatively expensive to implement
- Random replacement is most frequently used.

Writing the cache back

- For caches which hold only instructions, this problem does not arise.
- For caches which hold only data, or for caches which hold instructions and data this is a problem.
  - Not as big a problem as one might imagine: reads dominate memory accesses, making about 7% of the overall traffic (or 25% of the data cache traffic) writes.
  - Still, it may not be neglected
- Two different techniques are in use
  - write-through
    - the information is written simultaneously to both the cache and the lower-level memory
  - write-back (also known as copy-back)
    - the information is written only to the cache
    - when the cache block is replaced, it is written back to the lower-level memory

Write-through versus write-back

- write-through has fewer problems - but leads to more traffic on the bus
  - slower but easier to implement
  - main memory writes are more predictable
    - they only occur on CPU stores
- write-back is more complex
  - cache lines require an associated bit to show whether they have been altered or not
    - called the *dirty* bit.
    - If the dirty bit is set, the the cache block must be written back when it is overwritten
  - write-back may occur on a read as well as a write.
  - Faster, since normal writes occur at cache speed
  - some writes never go to memory at all!
    - E.g. when a word is written, then written again before the cache line is overwritten

Example Caches (I: 68040 processor)

- 68040 processor
  - 2 independent caches
    - one for data, one for instructions
  - Both are 4Kbyte long
  - Both are 4-way set associative, with 64 sets, each of 16 bytes (64 * 4 * 16 = 4096)
  - Each cache line has
    - a valid bit (used t startup)
  - and each data cache line has a 4 dirty bits (one per 32 bit word)
  - The system can use either write-through or write-back techniques.

Example Caches (II: Digital Alpha processor)

- The Alpha has 3 on-chip caches
  - an instruction cache
  - a data cache
  - a second level cache
  - it also allow for a 3rd level (off-chip) cache.
- Instruction cache
  - 8Kbytes long
  - Each line is 32 bytes long
  - direct mapped
- Data cache is similar
  - it is dual-read ported, and single write-ported
  - uses write-through
- Write-through uses a write buffer
  - this has 6 32-byte entries
  - used to hold data to be written to main memory
  - also uses write merging: the write buffer is updated by new write requests

---

Example Caches (III: Pentium)

- Like the Alpha, the Pentium has more than one level of cache.
- Different versions have different amounts of cache:
  - 8, 16, 32Kb
- ...for both data and instruction
- Additionally, external (off-chip) cache can be added
  - 256Kb or 512Kb
- and both write-back and write-thru are supported.
- Pentium 4:
  - Data cache has 2 levels:
  - L1 8Kbyte, 4 way set associative, 64 bytes/cache line. Write-through (to L2 cache). 2 clock load latency
  - L2 256Kbyte, 8 way set associative, 128 bytes/line. Write-back. Latency is 7 clock cycles.
  - Instruction cache: Trace cache (uOP). 12K uOPs