Advanced Computer Architecture

LECTURE 2: MEMORY
MEMORY HIERARCHIES
MEMORY DATA FLOW
MEMORY ACCESS OPTIMIZATION

JAN LEMEIRE

What is memory?

Cf Von-Neumann model: Random Access Memory

- Memory
  - Just an “ocean of bits”
  - Many technologies are available
- Key issues
  - Technology (how bits are stored)
  - Placement (where bits are stored)
  - Identification (finding the right bits)
  - Replacement (finding space for new bits)
  - Write policy (propagating changes to bits)
- Must answer these regardless of memory type

Ideal Memory

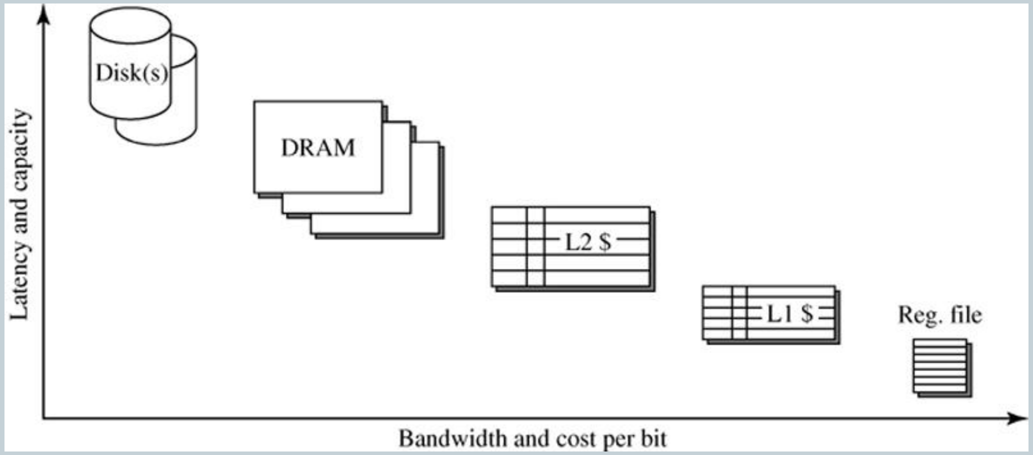
1. **Infinite capacity**
2. **Infinite bandwidth**
3. **Zero latency**
4. **Persistence (non-volatility)**
5. **Very low implementation cost**
6. **Very low power consumption**

- Non-existent …
- But maintain the illusion…

Latency versus Bandwidth

- Latency
  - a form of time
  - time between issuing a request and receiving a response
  - latency time = response time = service time + queueing time
  - service time: minimum, depends on HW
  - queueing time: time to get access to a resource, can be zero
  - driven by technology, very difficult to improve (done by avoiding serialization)
- Bandwidth
  - a form of quantity/time
  - number of requests that can be handled, rate of request handling
  - can be > 1/latency in case of concurrent handling of multiple requests
  - raw or peak bandwidth: neglect all potential bottlenecks in system
  - sustainable bandwidth: considers bottlenecks, but not necessarily real-life access patterns
  - driven by product cost, such as number of wires (trivially higher bandwidth)

Real Memories

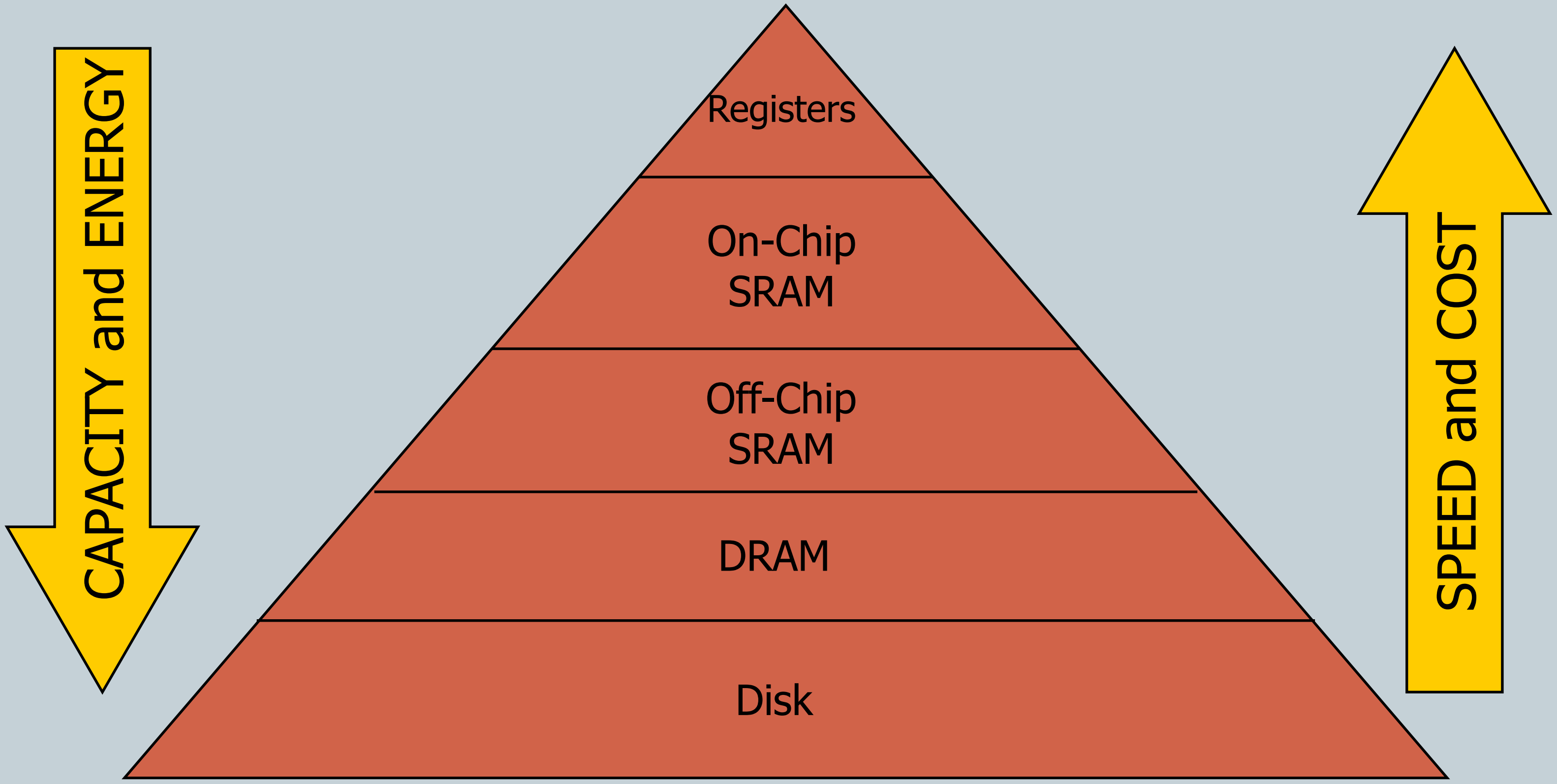


Type	Size	Speed	Cost/bit
Register	< 1KB	< 1ns	€€€€
On-chip SRAM	8KB-6MB	< 10ns	€€€
Off-chip SRAM	1Mb – 16Mb	< 20ns	€€
DRAM	64MB – 1TB	< 100ns	€
Disk	40GB – 1PB	< 20ms	~0

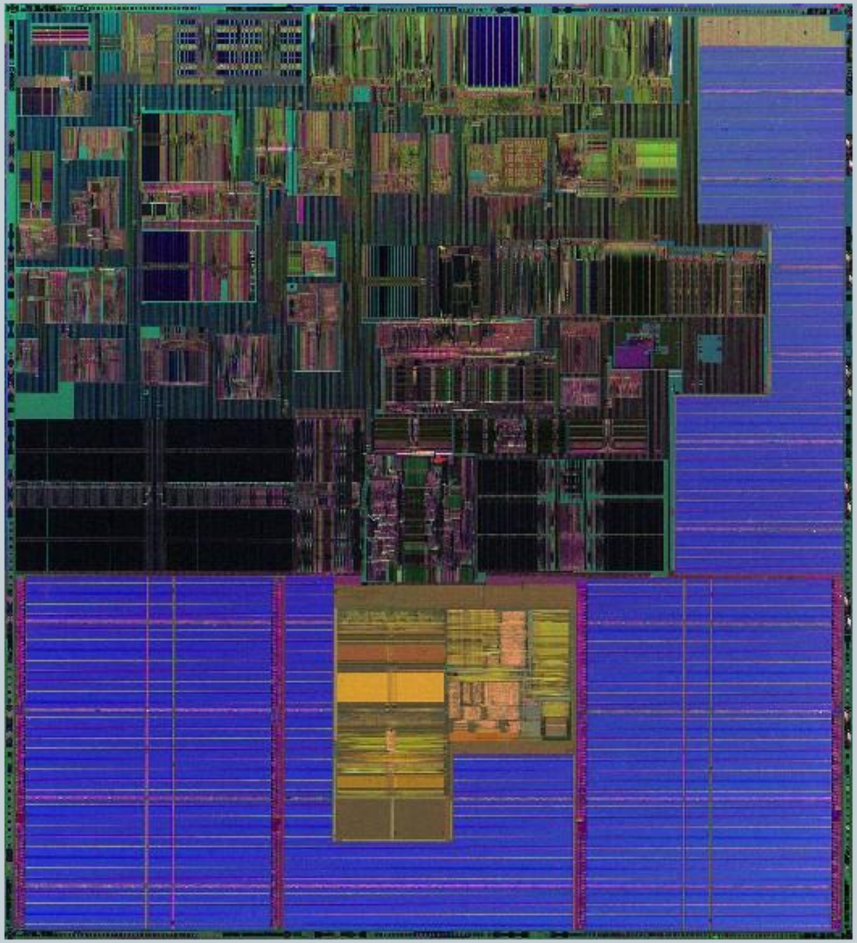
disclaimer:
numbers are several years old

Memory Hierarchy

- Try to place and access data as much as possible in top of hierarchy



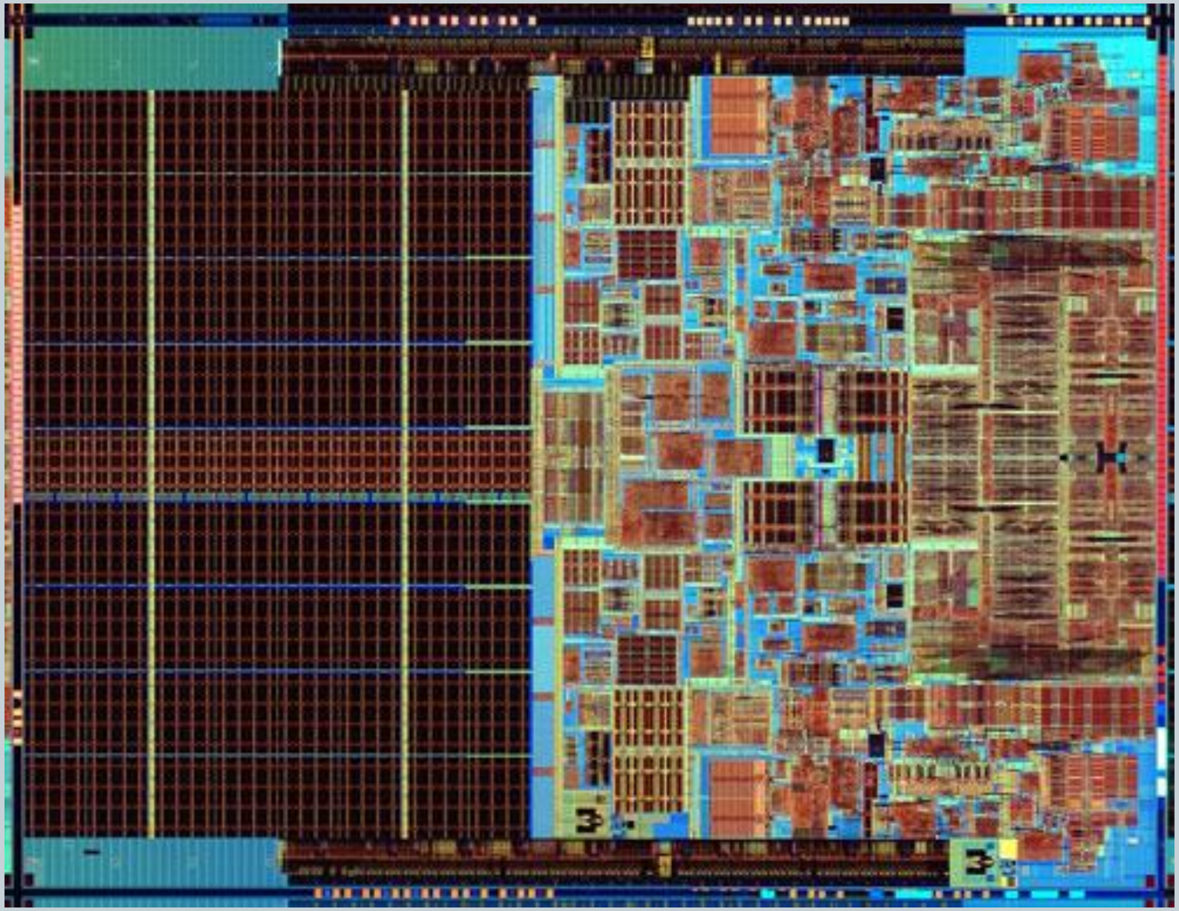
Example: Intel Itanium



- 800MHz
- L1 32KB (I- en D-)
- L2 96KB
- L3 4MB
- CPU core: 25M transistors
- Caches: 300M transistors (denser)

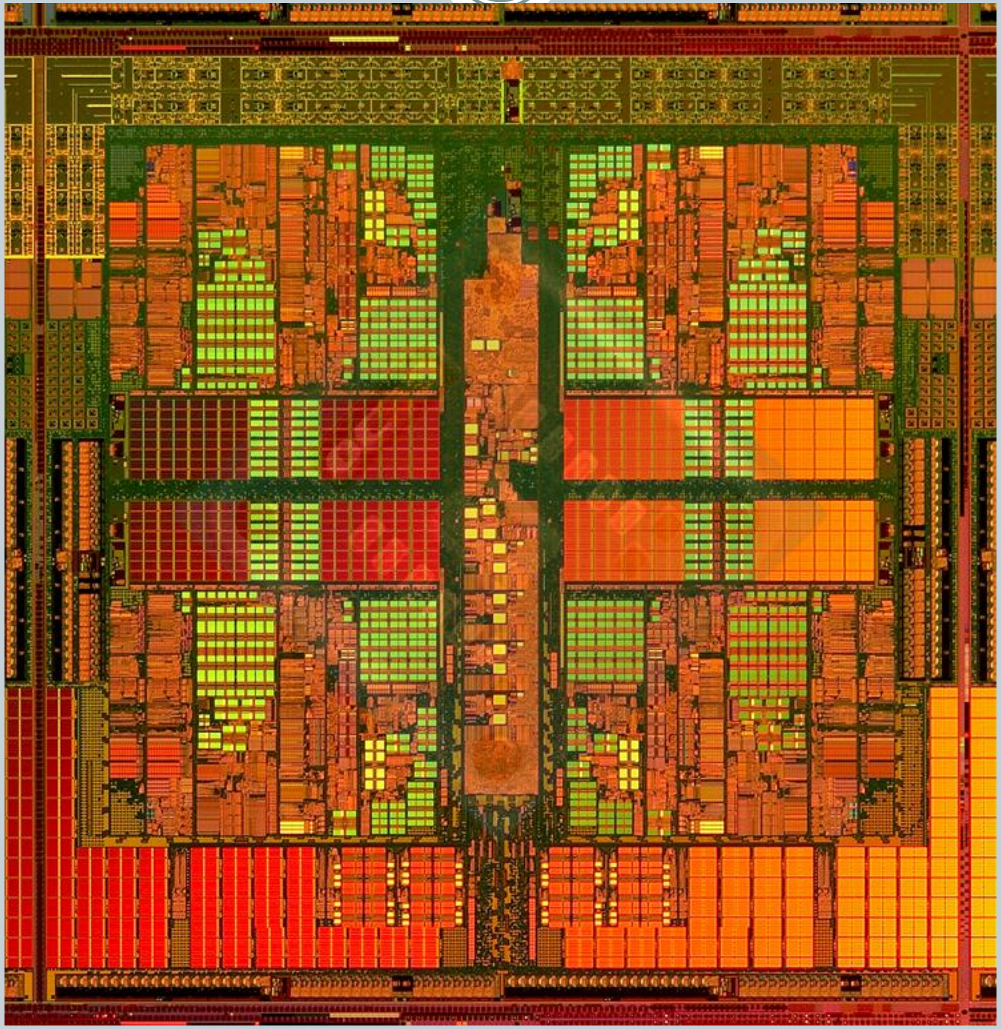
Today: up to ¾ of die area is cache…

Example: Intel Core 2 Duo



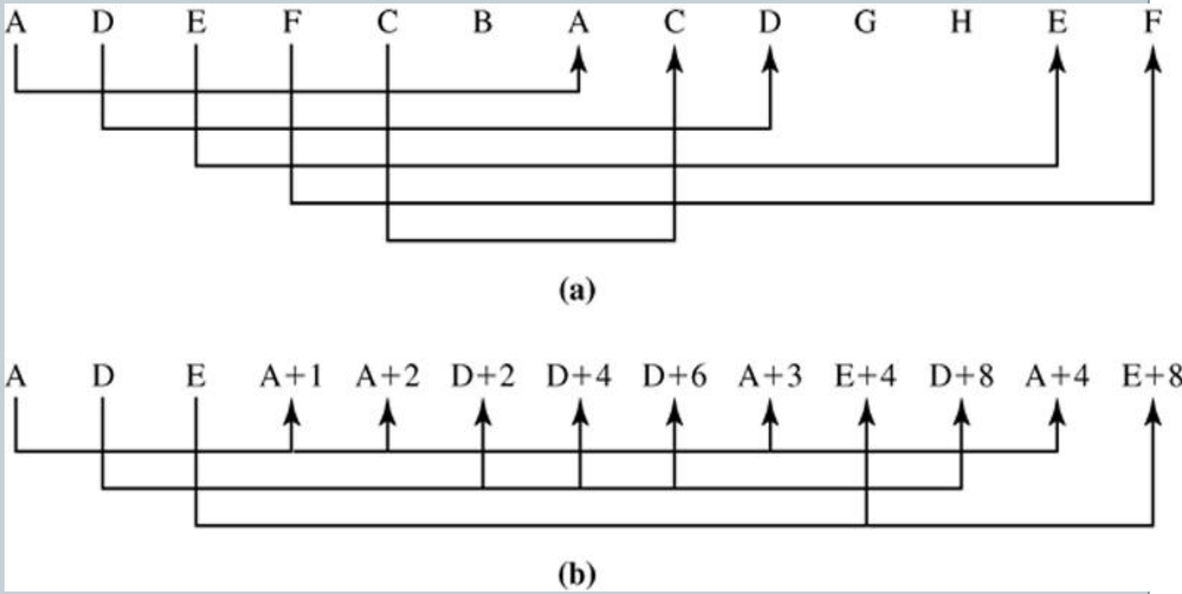
Notice the big caches but also all the small memories (caches, tables) inside the cores.

Example: AMD Barcelona



Locality and Caching (1)

- Two forms of locality in most if not all programs
  - ***temporal locality***: repeated accesses to same data occurring close together in time
  - ***spatial locality***: consecutive accesses to neighbouring data

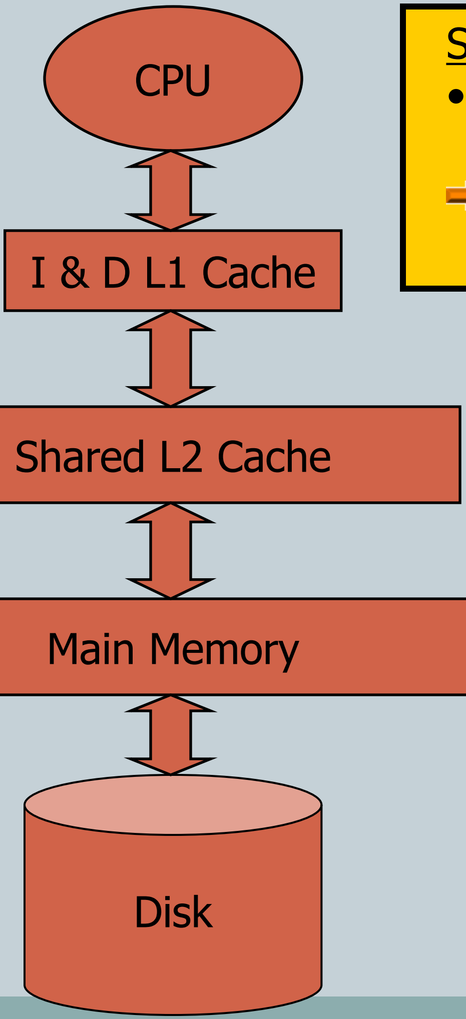

A D E F C B A C D G H E F
(a)
A D E A+1 A+2 D+2 D+4 D+6 A+3 E+4 D+8 A+4 E+8
(b)

- Enables us to move data up in the hierarchy at run time to access it there instead of in lower, slower, higher-energy memories: caches
- Can give us the impression of large, low latency, high bandwidth memories

Locality & Caching (2)

Temporal Locality
- Keep recently referenced items at higher levels

⇨ Future references satisfied quickly



Spatial Locality
- Bring neighbors of recently referenced to higher levels

⇨ Future references satisfied quickly

Terminology

- *Hit*: when some data is found in a memory
- *Miss*: when it is not, need to look in lower memory
- Results are miss and hit rates

- *Global*: for all levels down to a component
- *Local*: only for accesses that reach as deep as a component

- Example:
  - Global hit rate L2$
    = local hit rate L1$ + local miss rate L1$ * local hit rate L2
- Latency = sum hit rate of level i * latency of level i

Multilevel Cache Hierarchies [574]

- Average access time with 1 cache level:

  $$\text{hit\_time}_{L1} + \text{miss\_rate}_{L1} \times \text{latentie}_{MEM}$$

- Average access time with 2 cache levels:

  $$\text{hit\_time}_{L1} + \text{miss\_rate}_{L1} \times (\text{hit\_time}_{L2} + \text{miss\_rate}_{L2} \times \text{latentie}_{MEM})$$

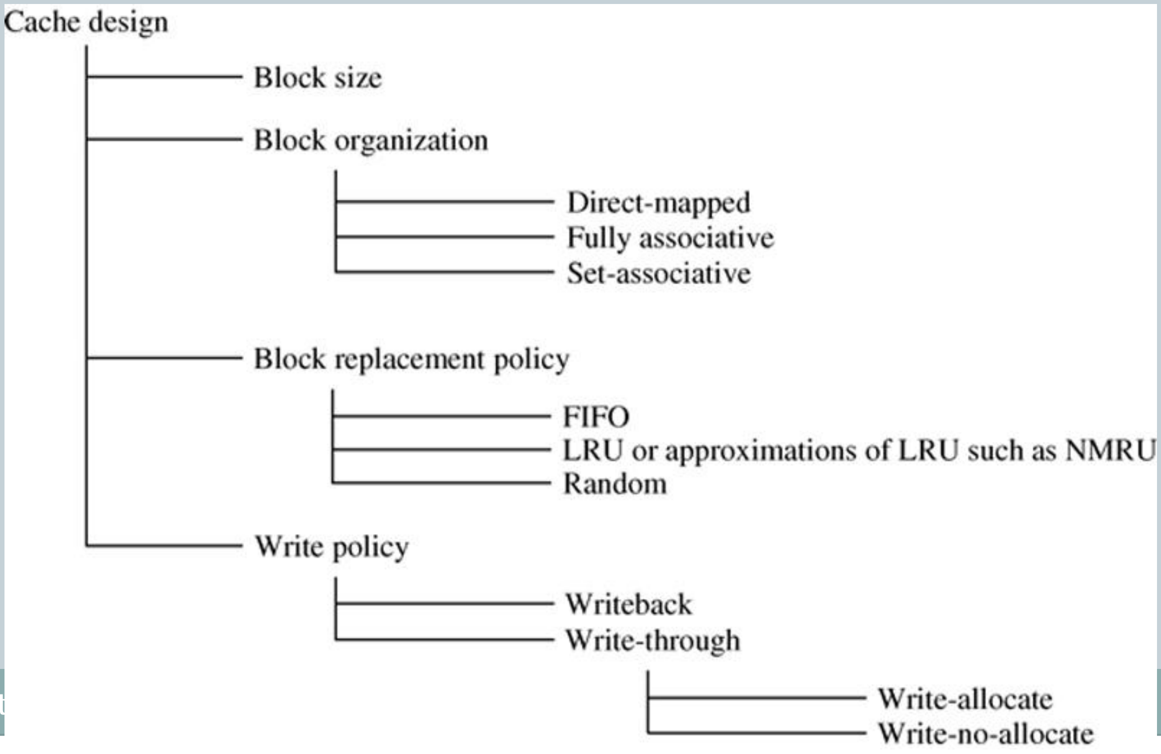
- In other words: fewer accesses have to reach to main memory
- Is expensive in terms of hardware

Cache Organization

- *Cache line* or block size: granularity, in number of bytes with which a cache copies data from lower levels
- Per line, tags store address of bytes stored in a line
- On access:
  - *hit* if tags indicate that data at address is present
  - *miss* if data is not present, is then fetched from lower level
  - current data in cache needs to be *evicted* to make room
  - maybe that data needs to be *written back* to memory

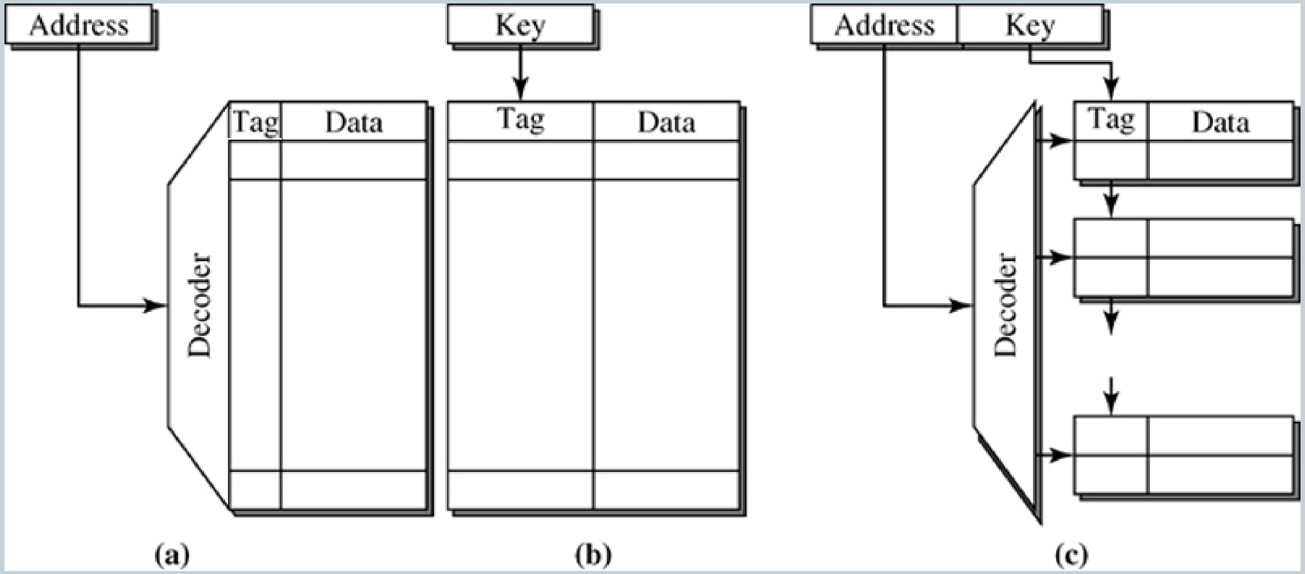
Cache Policies

- Policies for [p. 118]
  A. Locating data
  B. Evicting data
  C. Handle data updates



(A) Locating data

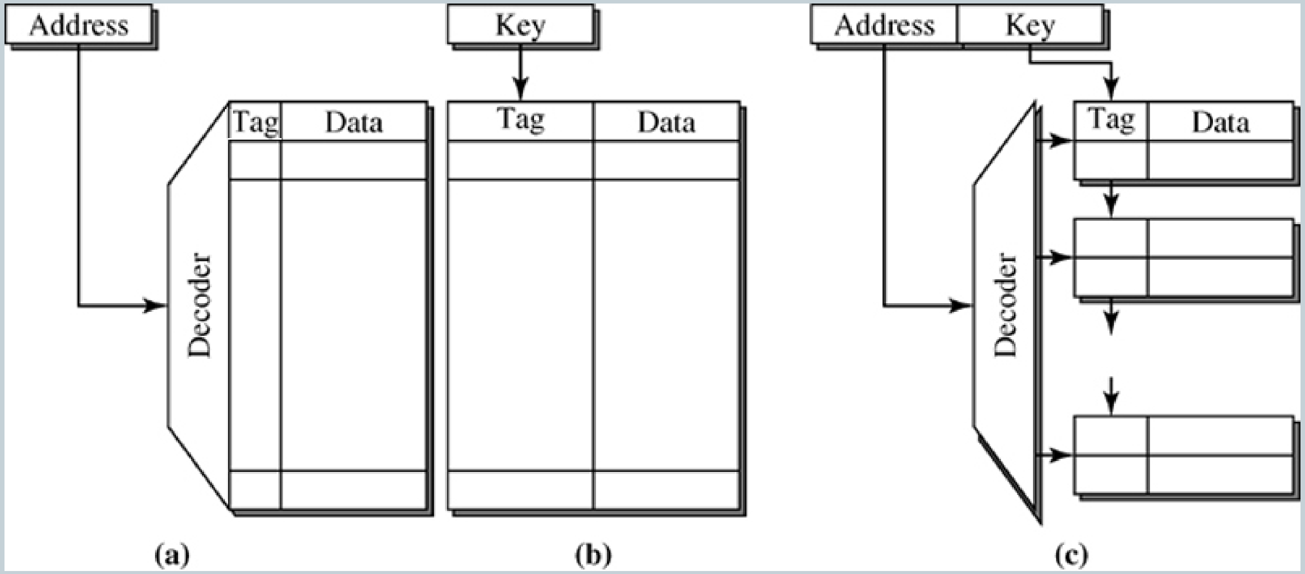
- (a): direct mapped cache
  - each address mapped to one cache line
  - decoder extracts line from address bits
  - tags indicate which address is on the cache line
  - many-to-one mapping
  - cheap, fast
  - little flexibility in mapping
  - lower IPC



(a) (b) (c)

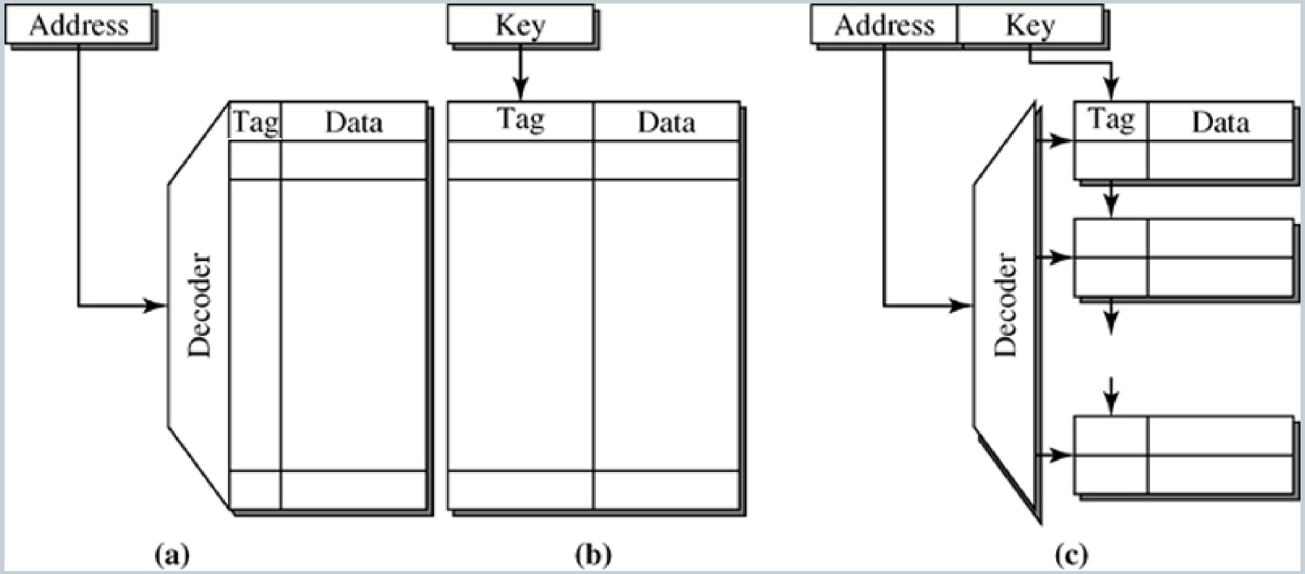
(A) Locating data

- (b): fully associative cache
  - each address can be mapped to all cache lines
  - all locations must be searched for each access (many, expensive concurrent accesses)
  - selection of data is time-consuming (large crossbar – multi input mux)
  - many-to-many mapping
  - Slow ?, complex yes, high energy consumption, but not slow
  - high flexibility
  - higher IPC



(A) Locating data

- (c): N-way set-associative cache
  - each address can be mapped to a number of cache lines
  - decoder extracts "set" of line from address bits, key is used to searched within the set
  - selection of data is time-consuming (large crossbar – multi input mux)
  - many-to-few mapping
  - compromise between direct-mapped and fully associative
  - the numbers of lines per set is also called the number of "ways": 2-way, 4-way, 8-way, ...



(B) Evicting data

- Replacement policy
  - in case of associative caches, have to choose which way to evict and replace
  - several options
    - least-recently used (LRU) way is evicted
      - excellent performance (IPC)
      - very good model (for worst-case-execution-time estimation)
      - difficult to implement
    - not-most-recently-used (NMRU)
      - performance close to LRU
      - much simpler to implement (fewer bits, only remember MRU instead of total order)
      - not as easy to model as LRU
    - random
      - good average performance
      - implemented with pseudo-random generator
      - difficult to model

(C) Handle data updates

- Several options to handling updates to a block
  - ***write-through***
    - pass each write to lower level immediately
    - two options for writing to non present block
      - write-allocate: fetch a block on a write
      - write-no-allocate: do not fetch a block on a write
      - write-no-allocate is better if streaming writes overwrite a full block without reading from it
  - ***write-back***
    - only write back data when you have to
      - on eviction
      - when another processor needs the data
      - a dirty bit indicates whether data was changed and needs to be written back
  - write-back most often used because much less bandwidth is needed
  - write-through sometimes used when both levels are on chip and bandwidth is available and cheap

Cache Organization: varia

- Caches may be shared or separate (non shared)
  - shared includes code and data
  - usually L1$ not shared, but L2$ shared
  - because otherwise too many ports and too many conflicts
- Caches may be inclusive or exclusive
  - inclusive
    - L2$ contains all data that is in L1$
    - loses some space
    - line size may be different
    - easier for coherency: external processors need to check only L2
  - exclusive: data in L1$ is not in L2$
    - save space
    - must have same line size

Measuring Cache Performance (1)

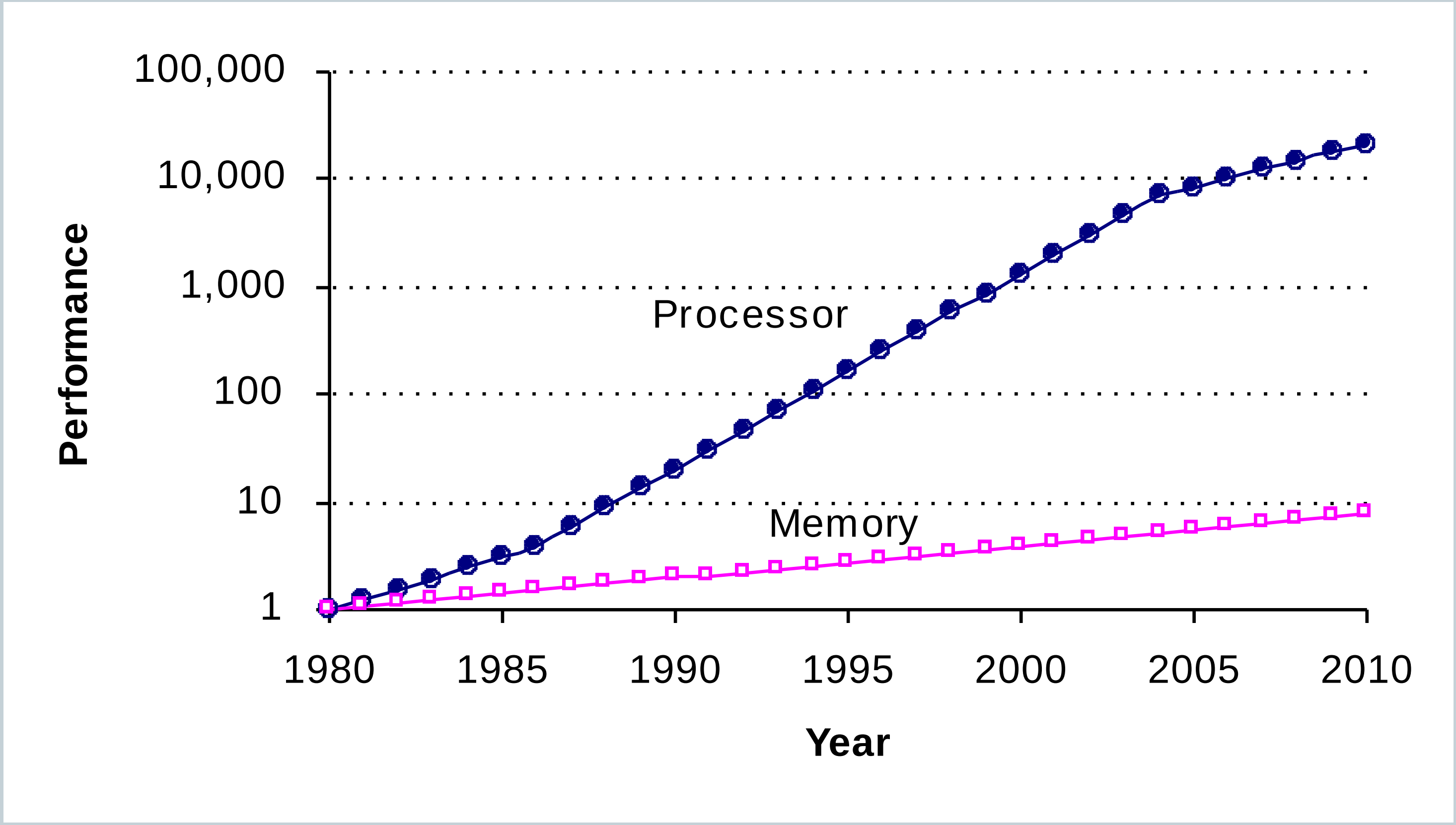
- Misses are categorized
  - Cold or compulsary misses
    - when a block or line is accessed for the first time
    - fundamental, cannot be avoided by caching
  - Capacity misses
    - result from cache being too small, not fundamental
  - Conflict misses
    - result from small associativity or eviction policy
    - not fundamental
    - fully associative can eliminate all conflict misses assuming a perfect eviction policy

Measuring Cache Performance (2)

- How to measure?
  - m_a: #misses in the cache being measured
  - m_f: #misses in same cache, but fully associative
  - m_c: #misses in cache with same block size, but infinite and fully associative
- Then
  - cold misses = m_c
  - capacity misses = $m_f - m_c$
  - conflict misses = $m_a - m_f$

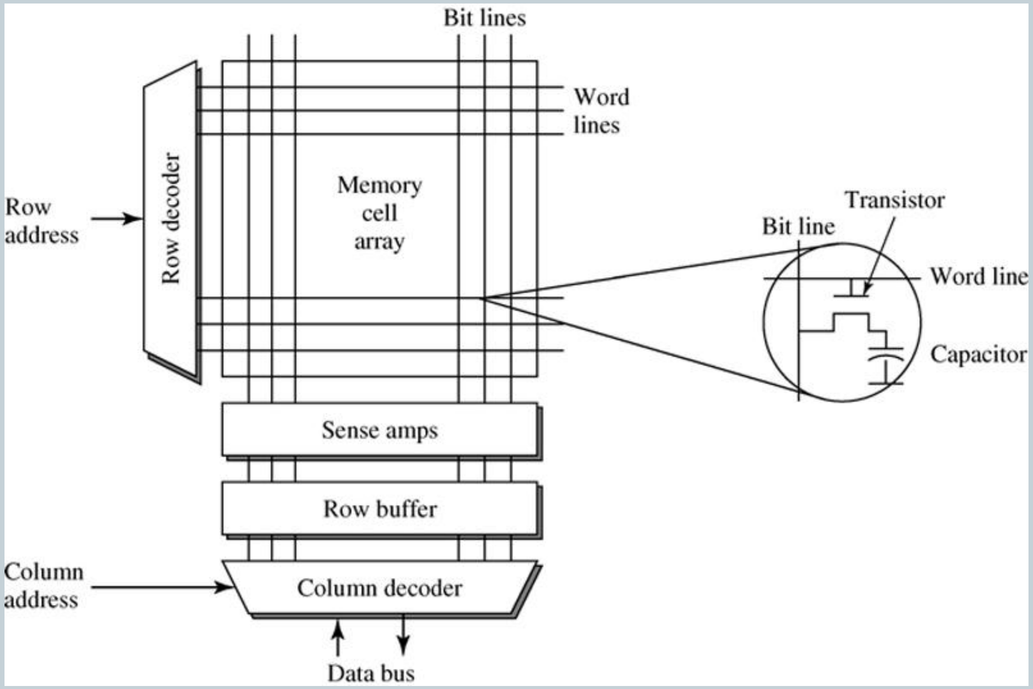
Main Memory: DRAM (1)

- Memory wall: compared to the CPU, main memory is becoming slower and slower



Main Memory: DRAM (2)

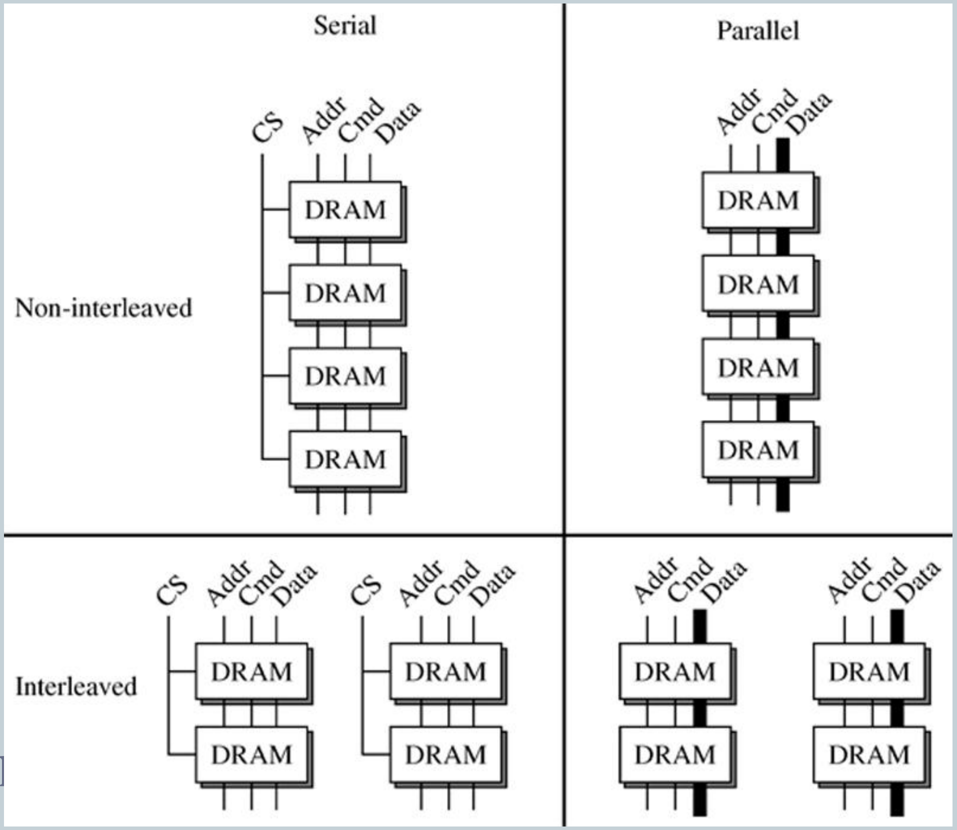
- Access:
  - First specify row address
  - Load a row into the row buffer (=precharging)
  - Then choose bytes with column address
- Data in row buffer can be reused (form of caching)
- Data needs to be refreshed
- Trends:
  - better signaling
  - synchronous interface (SDRAM)
  - use rising and falling clock edges to double data rate (DDR)
- Data bus shared for reads and writes => when they alternate, have bus turnaround time



Main Memory: DRAM (3)

- Interleaving (banking) and parallelism

1. serial access
   - control signal selects a single DRAM that outputs its data
   - output data width limited to that of a single DRAM module
   - needs control signal wires
   - can share other wires
2. parallel access
   - addresses send to multiple components
   - output data is concatenated to wider word
   - no control signal, sharing of other wires
3. interleaving
   - multiple banks with separate control
   - can handle independent accesses in paral
   - cost is extra controllers and wires

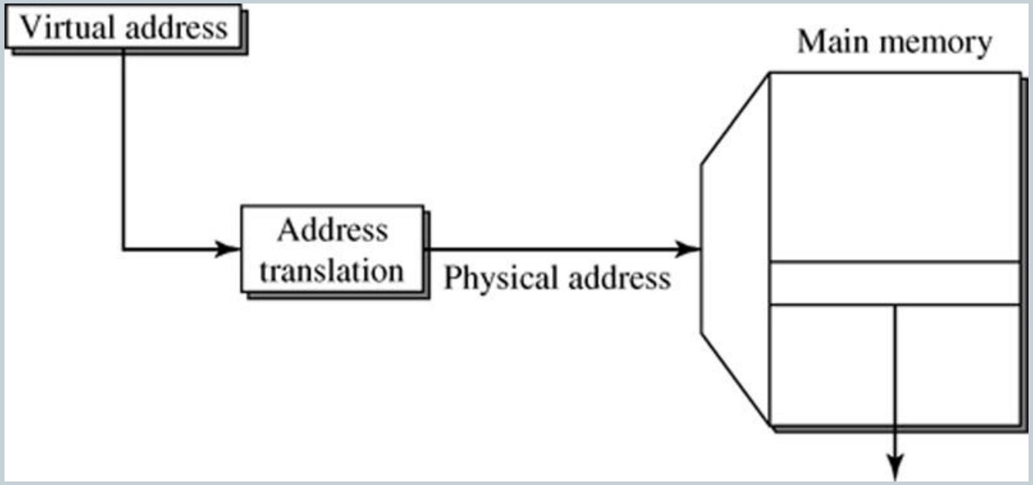


Virtual Memory (1)

- Programmer experiences 32-bit or 64-bit memory space, physical memory is smaller than 2^{64} bits
- Virtual memory system translates programmer's image (virtual addresses) to real hardware (physical addresses)
- Gives impression that full virtual memory is available
- And does this concurrently for multiple processes, each with their own full address space → time-sharing
- This requires the *translation* of virtual to physical addresses and *demand paging*
- Older than caching (first paper in 1962)
- **But is a form of caching, the same mechanism/principles**
  - **different implementation, extra functionalities necessary**

Virtual Memory (2)

- Virtual memory is partitioned into pages (4K, 8K, …)
- a page's data resides on disc or in main memory
- on access, processors verifies that page is in main memory
    - if not, a page fault exception is raised
    - **operating system (OS) catches it**, and fetches page into physical page in memory
    - if necessary, OS also evicts dirty page back to disc
- is called "demand paging"
- through lazy allocation
- CPU helps OS
    - translation of addresses
    - page faults
- OS helps CPU
    - maintains a page table



Virtual Memory (3)

- Replacement and eviction policy
  - LRU is best, but too complex to implement (too many pages)
  - approximated by clock algorithm
    - cooperation between OS and CPU
    - upon every access to a page, CPU sets "referenced bit"
    - every so many milliseconds, OS clears reference bits
    - when eviction is necessary, only pages with cleared bits are chosen
  - alternative: FIFO or others, always rely on reference bits

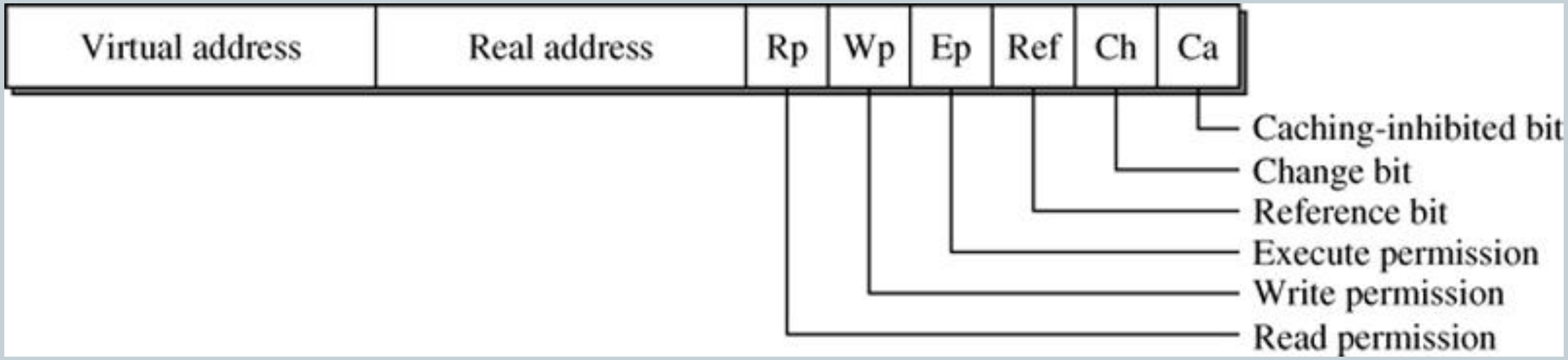
- Access to backing store (disk, network) is slow
  - process that has to wait will be put to sleep, other will get CPU

Virtual Memory (4)

- Memory protection
  - when multiple processes have pages in memory, they need to protected from each other
  - furthermore, it may be useful to share data up to some point, but not anymore as soon as one process tries to update it
  - read, write and execute access are set per page
  - with bits like dirty bit, referenced bit
  - cacheable bit: indicates whether or not a page can be cached
    - is important for memory-mapped IO
    - for simplicity of maintaining coherency

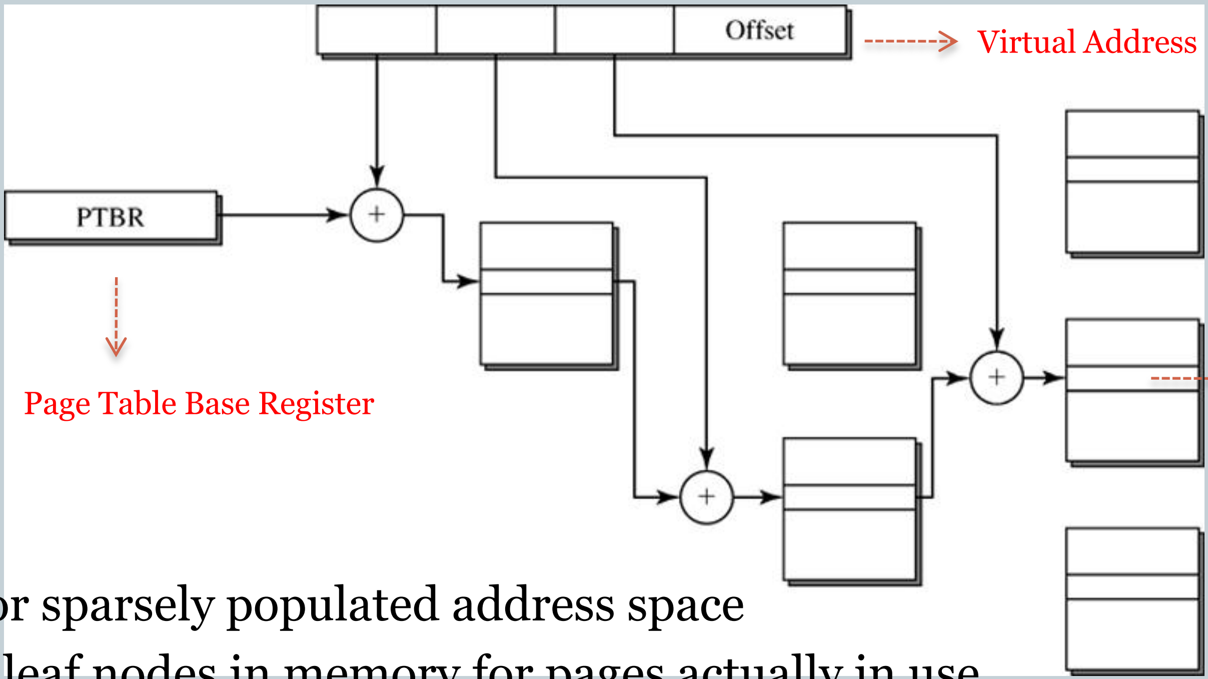
Virtual Memory (5)

- Page tables
  - OS maintains tables that store all relevant operation
  - CPU uses those pages
  - entries in page table look like this



Virtual Memory (6)

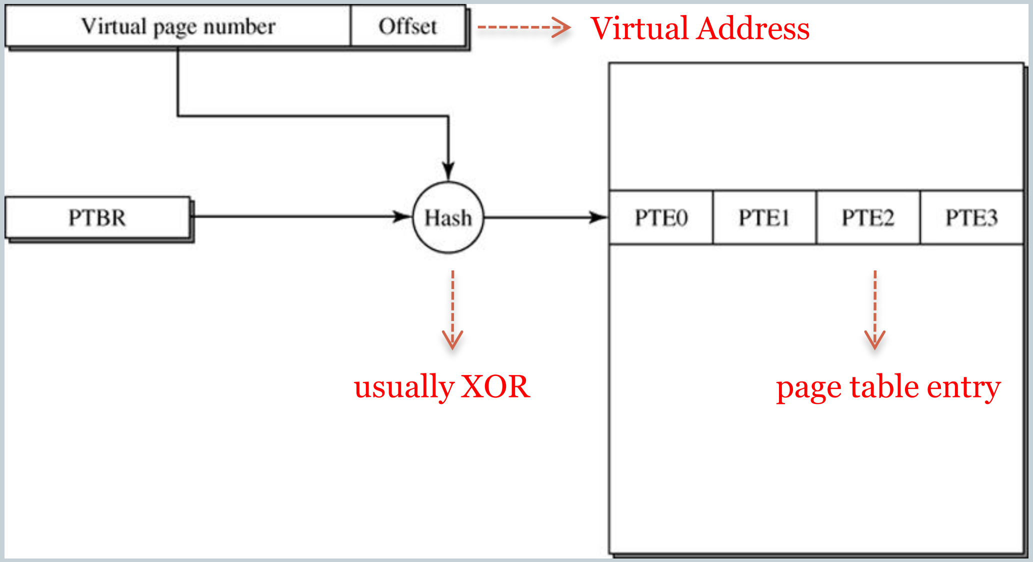
- Implementation 1: **Multilevel Forward Table**



- Efficient for sparsely populated address space
- Only need leaf nodes in memory for pages actually in use
  - but this can be more than the number of physical pages!
- Nodes can be stored on backing store themselves
  - pageable page table, need OS support
- Large pages to avoid page tables as big as memory!

Virtual Memory (7)

- Implementation 2: **Inverted or hashed page tables**



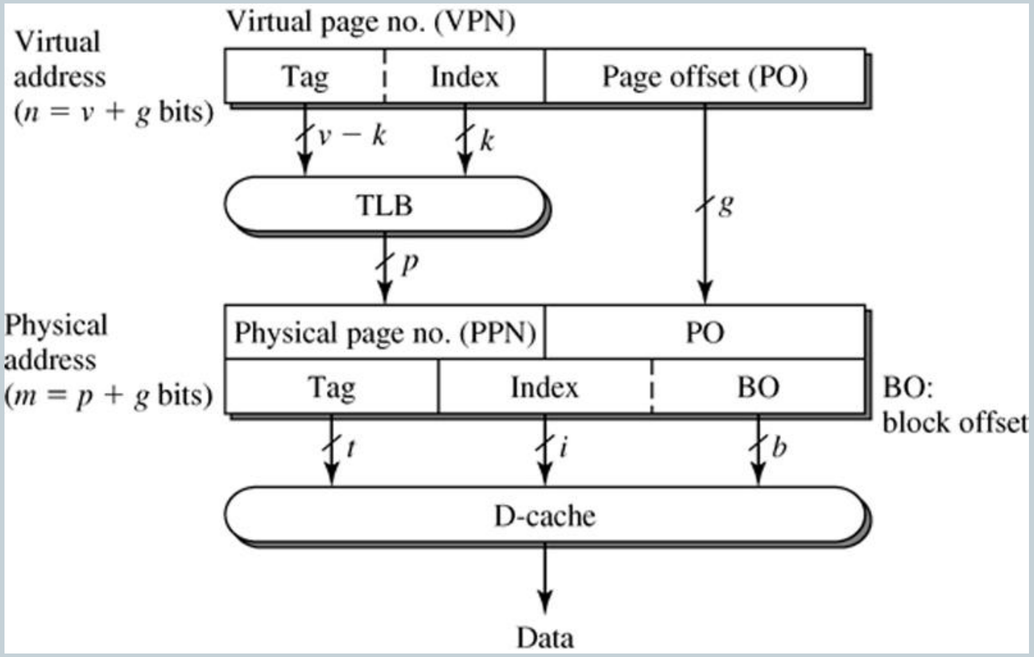
- Page table entries are now in a set (set-associative), needs to be searched
- Only entries for allocated pages are needed
  - #entries = physical memory size/page size
  - whole table always fits in memory, no paging of page table, no OS support for that
- Conflict misses: use second hash, and try there (rarely both fail)
- Problem: OS needs to store PTE's of non-allocated pages somewhere else

Virtual Memory (8)

- Translation Lookaside Buffer
  - fully associative cache holding page table entries
  - then hardware does not need to look in actual page tables
  - Hardware-controlled cache of ‘cache’ map
  - Two maintenance modes:
    - software TLB miss handler
      - on a miss in the TLB, an exception is raised
      - on exception handler in the OS uses privileged instructions to update the TLB
    - hardware TLB miss handler
      - hardware goes looking in the page tables itself

Memory Hierachy Implementation (1)

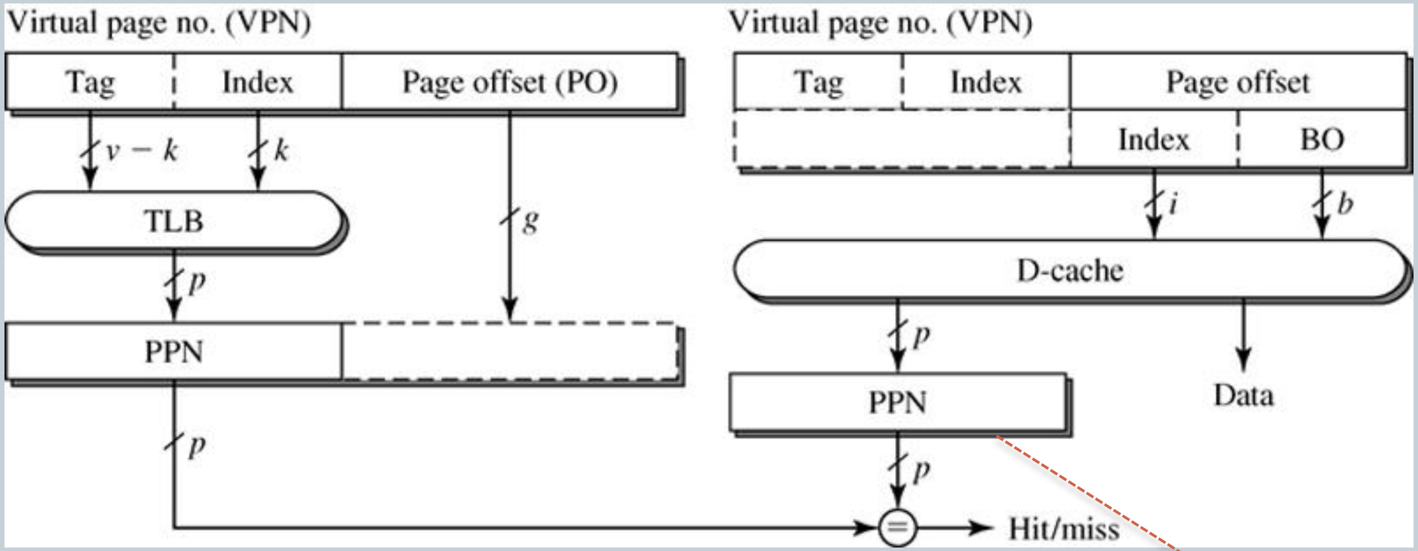
- D$ and TLB combined
- Option 1: physically addressed D$



- Drawback: TLB access and D$ are serialized -> extra pipeline steps

Memory Hierachy Implementation (2)

- Option 2: Virtually Indexed, Physically Tagged Caches



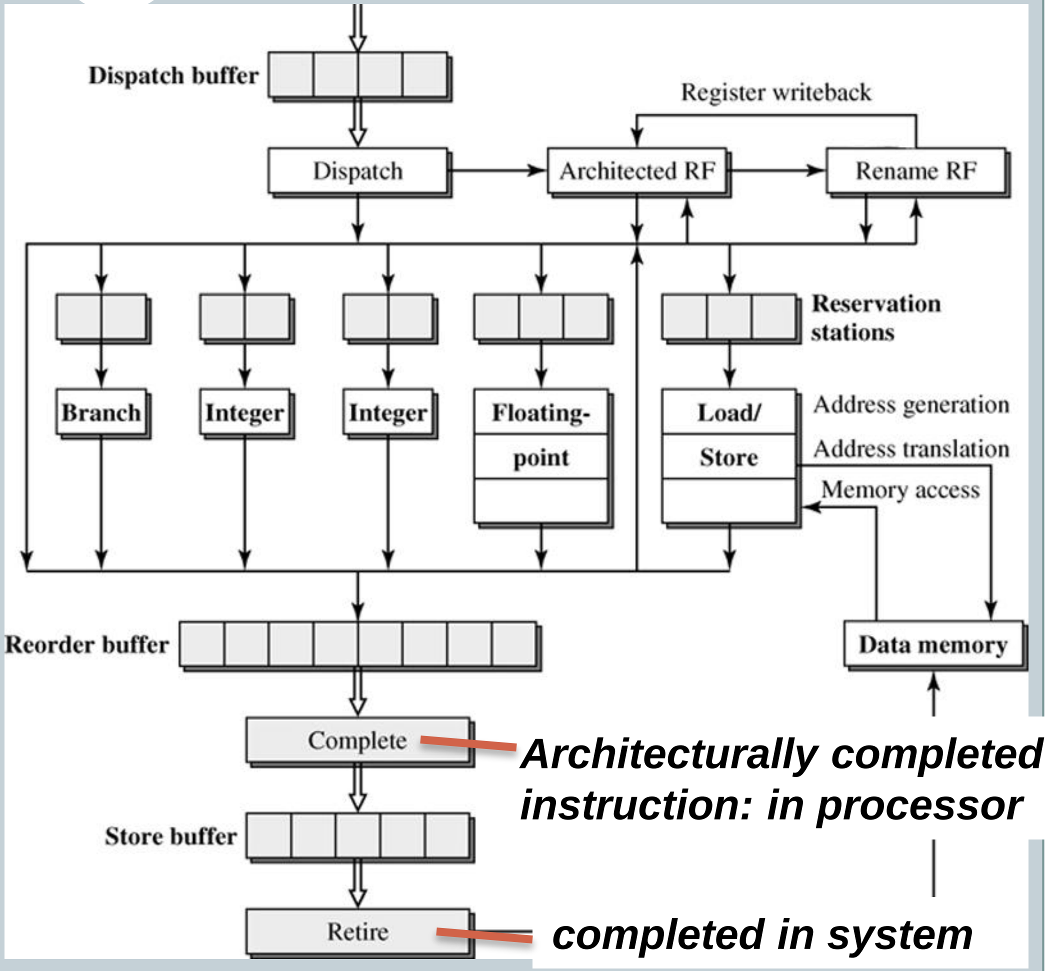
physical page number!

- Disadvantages
  - Page size equals the size of one way
  - Hence direct-mapped cache cannot be bigger than a page
  - Can only become bigger by adding associativity, which increases latency (problematic for L1$)

Memory Flow in OoO Processors [5.3]

- BASIC addressing mode
  - base register + immediate offset
- LOAD: Typically three steps
  1. add base address and offset
  2. TLB access (may cause miss)
  3. load access (may cause miss)
- STORE: Two pipeline steps
  1. add base address and offset
  2. TLB access (may cause miss)

Then the data is written in the store buffer and in the reorder buffer. When it leaves the reorder buffer (in order), the instruction is considered completed, and the store buffer is informed that the data can actually be written to memory (i.e, retiring, in order). Speculative operation is not a problem since speculative instructions do not retire.



Data dependencies through memory

- Same as ordinary data dependencies when accessing same address
  - RAW, WAW, WAR
- In-order execution of load/stores is simple
- Out-of-order execution is better, but more complex
- WAW and WAR dependencies are not a problem because of in-order retirement
- RAW may be a problem …

- *Goal*:
  Try to avoid that a load needs to wait for all stores to be completed

- *Idea*: dependency means that data is at CPU!

Load/Store Reordering

- To recover from exceptions, writes need to be retired in order
- Is also best for sequential consistency model in multithreaded programs
  - sequential consistency model requires that all threads see all changes to the shared memory in program order
- WAW and WAR are no problem when writes retire in program order
- Again, RAW data dependencies remain to be enforced
- *Problem*: unlike for register renaming, RAW dependencies are not known after decoding the instruction

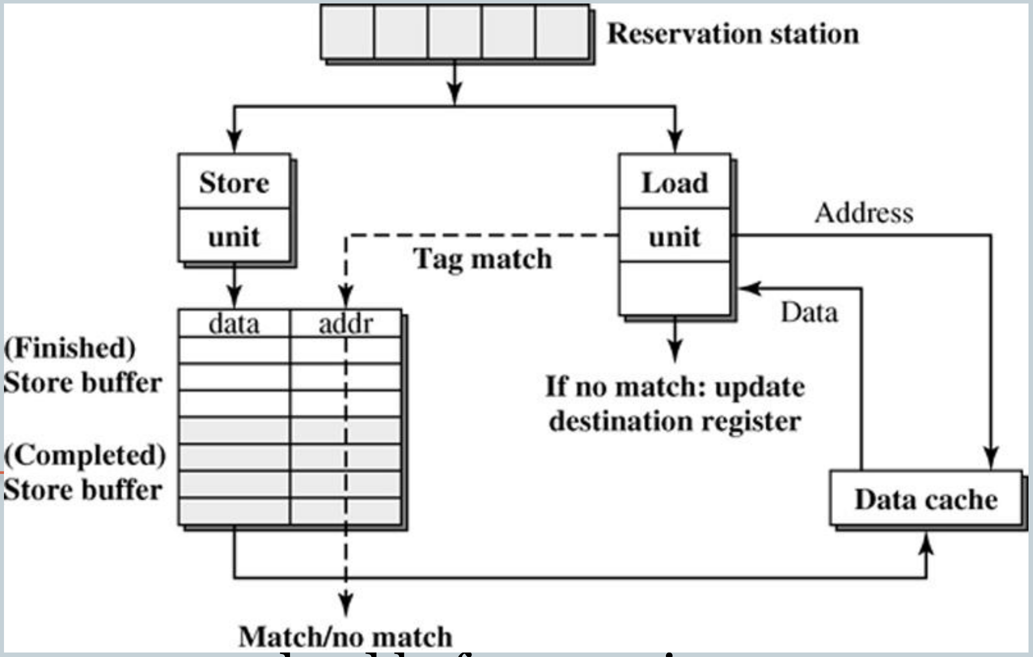
Load Bypassing

+11-19%

- Goal: execute loads out-of-order, as early as possible
  - they are usually followed immediately by uses of the loaded value

may be speculative instructions

instructions complete by reorder buffer

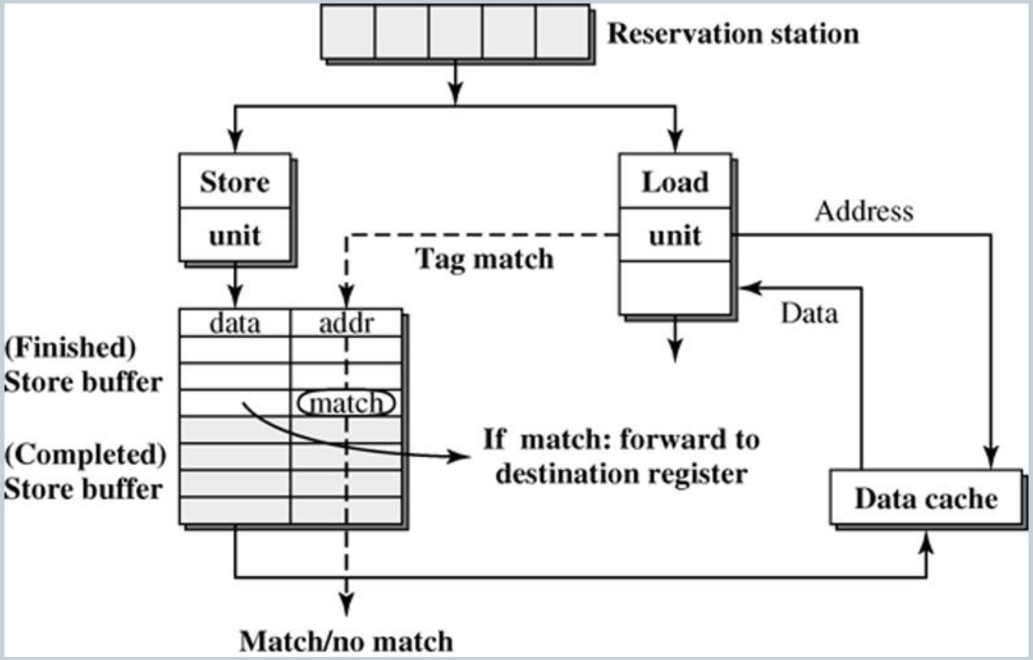


- Load bypassing: execute a load before previous store
  - all stores in flight are in store buffer
  - so check the store buffer for aliasing (write addresses are present in store buffer)
  - check (matching) can be on fewer bits, a bit pessimistic, but no problem in practice

Load Forwarding

+1-4%

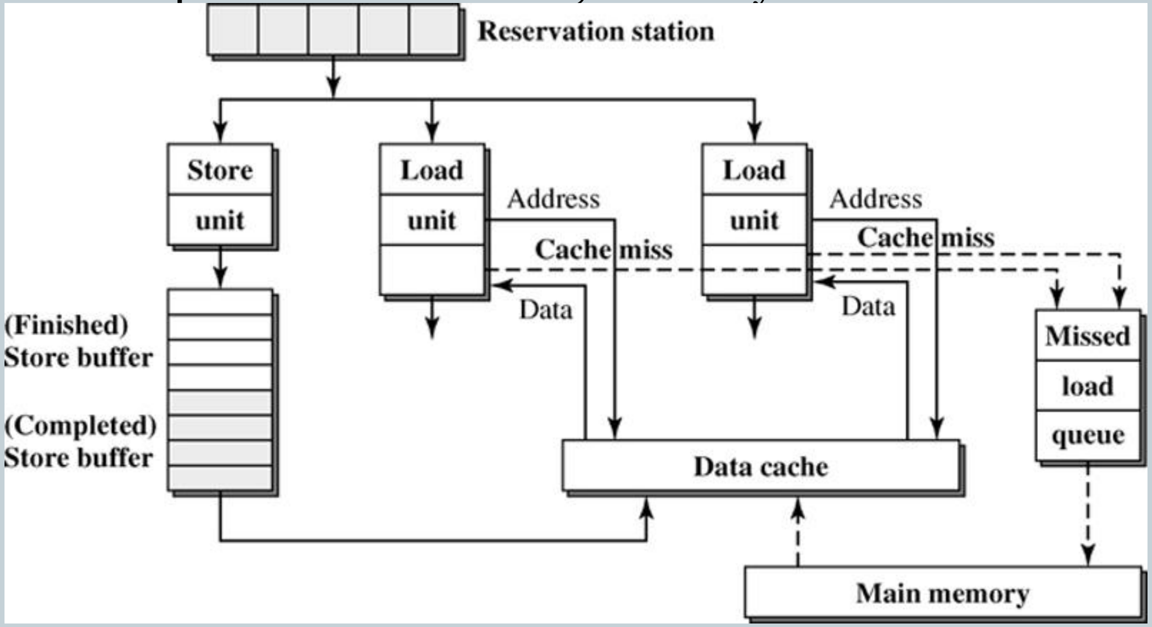
- If match is found in store buffer
  - forward value instead of just blocking the load
  - matching requires full addresses now
  - ordering mechanism for when multiple stores are in store buffer, may be of different sized data



gains:
11%-19% for load bypassing
1%-4% extra for load forwarding

Multiported Caches [273]

- When width of superscalar increases, memory accesses become a bottleneck.



- Three implementations for multiported caches
  - *true multiported* memory: very expensive and slow and power hungry, no conflicts whatsoever
  - *duplicate caches*: is expensive, but no conflicts in reading
  - *cache banks:* is cheaper (no duplication), but conflict resolution required

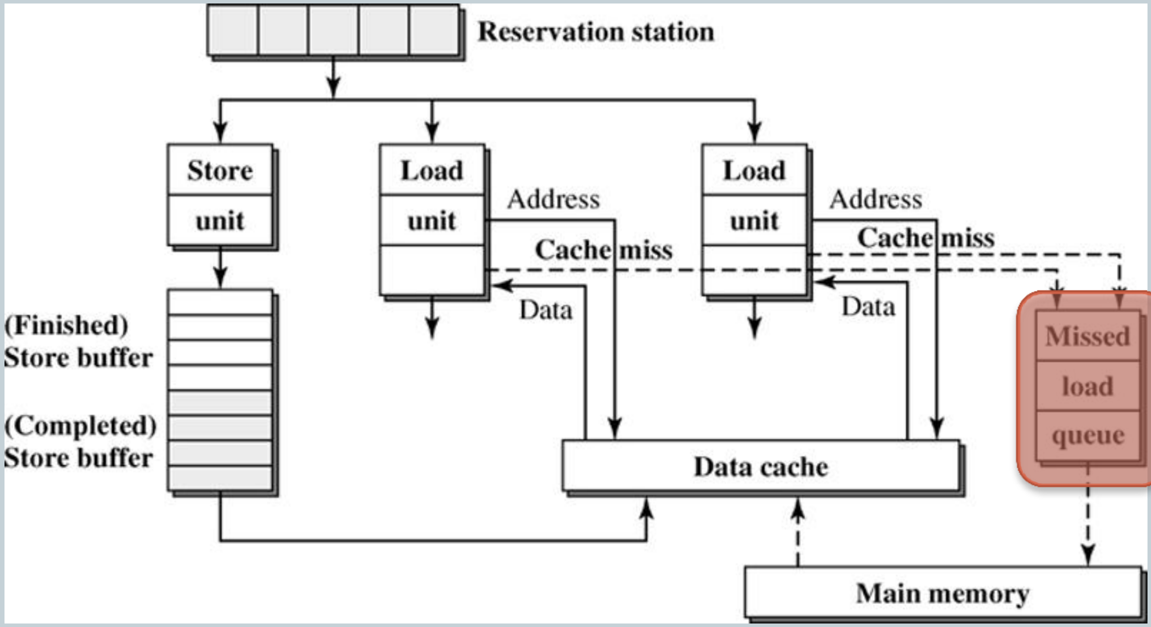
Pipelined Caches

- Alternative to multiported caches
- Pro
  - less hardware
- Con
  - dependences may cause blocking
- This is the current trend

Non-blocking Caches [274]

+15%

- Instead of blocking the loads unit pipeline on a cache miss, put load in a load missed load queue where it waits for the cache to fetch the data.
- In the mean time, other loads can be handled => *overlap load miss penalties*

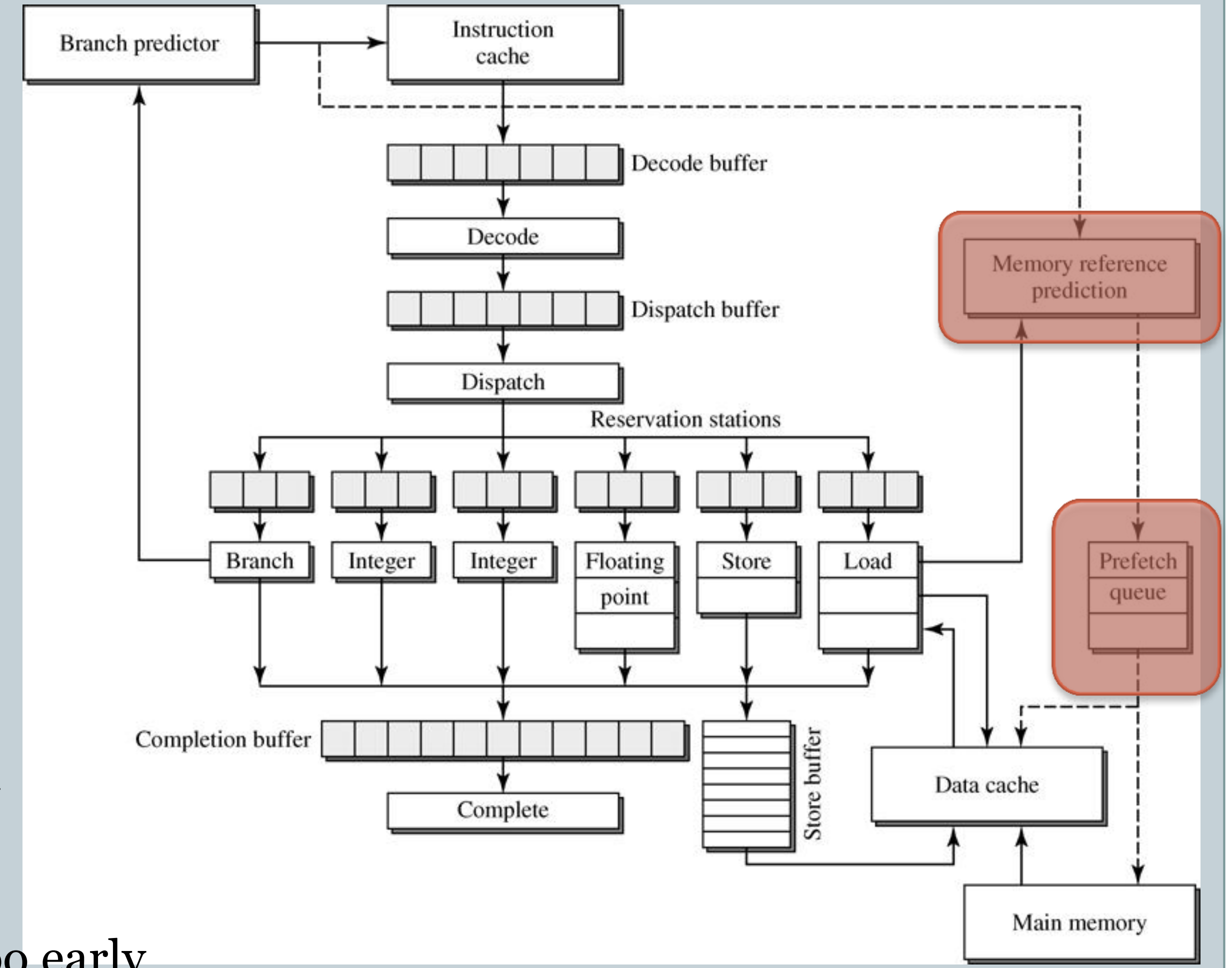


- Complication: caches misses come in bursts => handled in pipelined fashion
- Must cache misses of speculative loads be handled? Trade-off with memory bandwidth

THE OTHER PROBLEM: COLD MISSES

Prefetching Caches [275]

- Memory reference prediction table stores
  - load instruction address
  - previous data address
  - stride of previous accesses
- Stride + current data address is a good guess for the next execution of this instruction
- current data address - previous data address becomes new stride
- Prefetch queue steers prefetching of data (with low priority).
- Take care not to pollute the cache or evict data too early.



Prefetching Stream Buffers

- On cache miss, also fetch the next block, and store this in a stream buffer
- If a later load accesses the data in the stream buffer, it was a correct prediction, so again load the next block, storing the current one in the cache
- Multiple stream buffers are possible

Load address prediction [277]

- Issue speculative load instruction with predicted address
- As soon as new load instruction is encountered
- No need to wait for decoding, dispatching and register values to be available
- Instructions dependent on this data won't have to wait

⇔ Prefetching Cache technique: instruction still needs to be executed

Software Prefetching

- Add instructions to ISA that
  - do not load data into a register
  - but only load it into the cache
- Faulting or non-faulting
  - Typically non-faulting
- Extra instructions do cause overhead
- Is often useful when combined with software pipelining, loop unrolling, … where multiple iterations of a loop are intertwined and addresses are hence known upfront
- Usefulness depends (as with all data prefetching) on access patterns: direct array accesses in regular loop are easier to predict than iterating through a linked list

FURTHER CACHE OPTIMIZATIONS & ISSUES TO CONSIDER

Efficient Cache loading

- At level 1, cache lines are typically 32 to 128 bytes wide
- Data accesses are typically less than 8 bytes wide
- Accessed data may not be first in a cache line
- Two techniques
  - *Early restart*
    - As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
    - Spatial locality ⇒ tend to want next sequential word, so the benefit of just early restart is not clear
  - *Critical word first*
    - fetch the accessed data first, send it to the CPU immediately, and then fetch the rest of the block
    - widely used now that cache blocks are larger

Cache Access Prioritization

- Give priority to load cache misses over stores

- Write-through D-cache
  - To avoid blocking: *write buffer* (not the same as store buffer)
    - Store data to be written temporarily until bus is available
  - On load miss, first check the write buffer for data

- Write-back D-cache
  - Suppose load-miss results in eviction that requires write-back
  - First write-back and then load? Nope
  - Instead: copy write-back block to write buffer, fetch new block, and then write from write buffer to memory

Merged Writing of Cached Data

- Merging Write Buffer
  - Write buffer to allow processor to continue while waiting to write to memory
  - If buffer contains modified blocks, the addresses can be checked to see if address of new data matches the address of a valid write buffer entry
  - If so, new data are combined with that entry
  - This way, separate narrow writes are merged into single wider writes, which are more efficient (bus-contention, ...) and result in less blockings caused by a full write buffer
  - The Sun T1 (Niagara) processor, among many others, uses write merging

Victim Caches

- Small, fully associative cache
- Blocks that are evicted from a cache are stored in the victim cache
- On L1 cache miss
  - first look at victim cache
    - victim cache hit: swap blocks in L1 cache and victim cache
    - victim cache miss: access main memory
- Intended to reduce the number of conflict misses
- In AMD Athlon: victim cache with 8 elements

Cache Line Size Tuning

- Larger blocks exploit spatial locality more
  - eliminates cold misses
  - at the cost of higher miss latency
  - with fixed total size: larger blocks are at the cost of number of sets or associativity, hence more conflict and capacity misses

- For fixed cache sizes, optimal line sizes exist

- Solution: vary it over the different cache levels
  - For example
    - 32-128 bytes for L1
    - 64-256 bytes for L2

Cache Size Tuning

- Bigger caches are the trend these days because of
  - increasing gap between processor and memory clock frequencies (memory wall)
  - higher number of transistors that is available
- For example, 6MB L3 cache on chip (Intel Itanium 2)
- But
  - it has a high cost
  - larger sized caches are slower (can be compensated with banking)

Cache Associativity Tuning

- Higher associativity is usually better for hit rate
- 2:1 rule
  - A direct-mapped cache with size N has a comparable miss rate to a 2-way set-associative cache of size N/2
- Drawback
  - Higher associativity results in larger hit time, which might be a problem for the clock frequency

Compiler Optimization (1)

- For code caches
  - determine code such that frequently executed code does not cause conflicts in cache
  - align basic blocks at cache block boundaries
- Also for data compiler optimizations exist
  - goal is usually to increase temporal locality
    - avoid long live ranges (time over which a value needs to be kept somewhere)
    - make reuse distance smaller (this is the numbers of other locations accessed between two accesses to one location), larger reuse distance increases the risk for eviction
  - sometimes to increase spatial locality

Compiler Optimization (2)

- Loop interchange

```
/* before */
for (j = 0; j < 100; j++)
   for (i = 0; i < 5000; i++)
      x [i][j] = 2 * x [i][j];


/* after */
for (i = 0; i < 5000; i++)
   for (j = 0; j < 100; j++)
      x [i][j] = 2 * x [i][j];
```

- Inner loop has better spatial locality

Compiler Optimization (3)

- Loop tiling

```
/* before */
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        c[i] = c[i]+ a[i,j]*b[j];

/* after */
for (i=0; i<N; i+=T)
    for (j=0; j<N; j+=T)
        for (ii=i; ii<min(i+T,N); ii++)
            for (jj=j; jj<min(j+T,N); jj++)
                c[ii] =c[ii]+ a[ii,jj]*b[jj];
```

- "Before" iterates N times over N elements of array b
- "After" iterates N/T * N/T * T times over T elements of array B
- "After" has reuse distance T within b, while "Before" has reuse distance N
- In other words: T elements might fit in cache when N do not

Exercise

- We have seen a huge number of hardware and software techniques
- Categorize each of them under one or more of the following qualifiers:
  - reduces hit latency
  - reduces miss latency
  - reduces load latency in front-end of the pipeline
  - reduces load latency in the back-end of the pipeline
  - increases bandwidth
  - hides miss latency
  - lowers miss rates
  - …
- You may also add other useful qualifiers.
- This exercise is useful to see how all aspects of memory behavior are being optimized, and it presents insights in the cooperation of all techniques.

Acknowledgement

- Thanks for (parts of) slides
  - Bjorn De Sutter
  - Lieven Eeckhout
  - Mikko H. Lipasti
  - James C. Hoe
  - John P. Shen
  - Per Lindgren