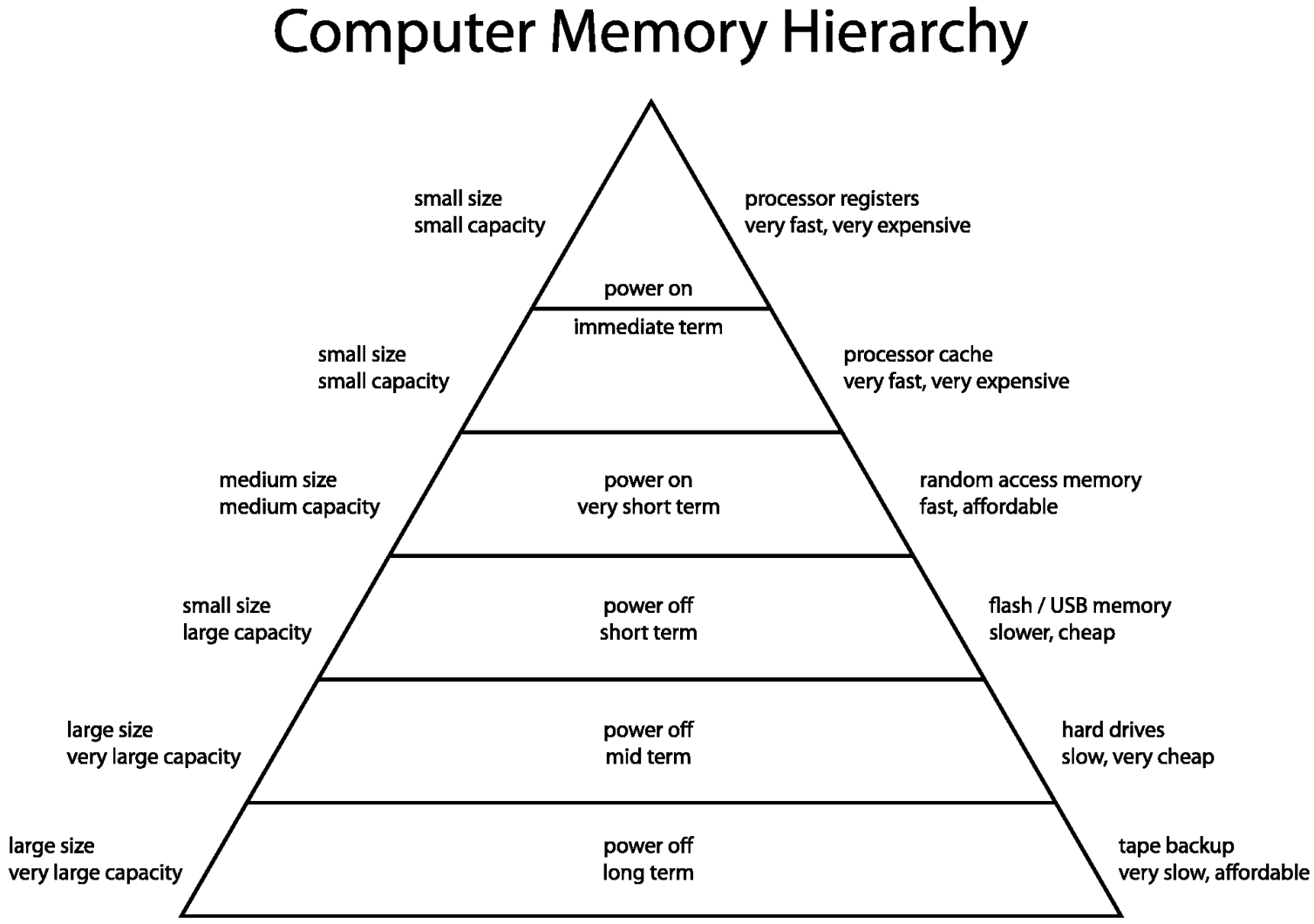
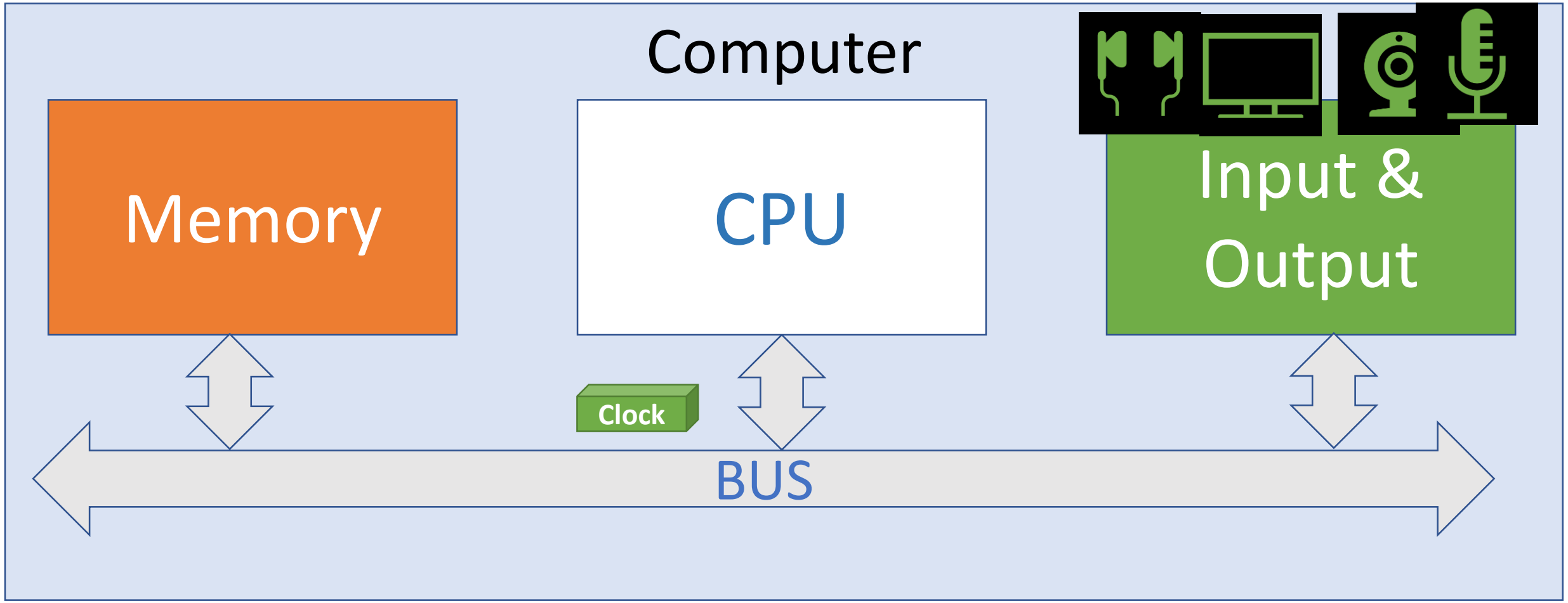
Today's Lecture

- Memory Design
  - Memory Design Aspects
  - Memory Hierarchy
  - Virtual Memory
- Key Research Topics
  - Optimizations (techniques)
  - Memory Access Paradigms
  - Memory technologies
- Reading Materials:
  - Chapter 5 – COAHP
  - Chapters 4,5 – COAWS



Source: Wikipedia

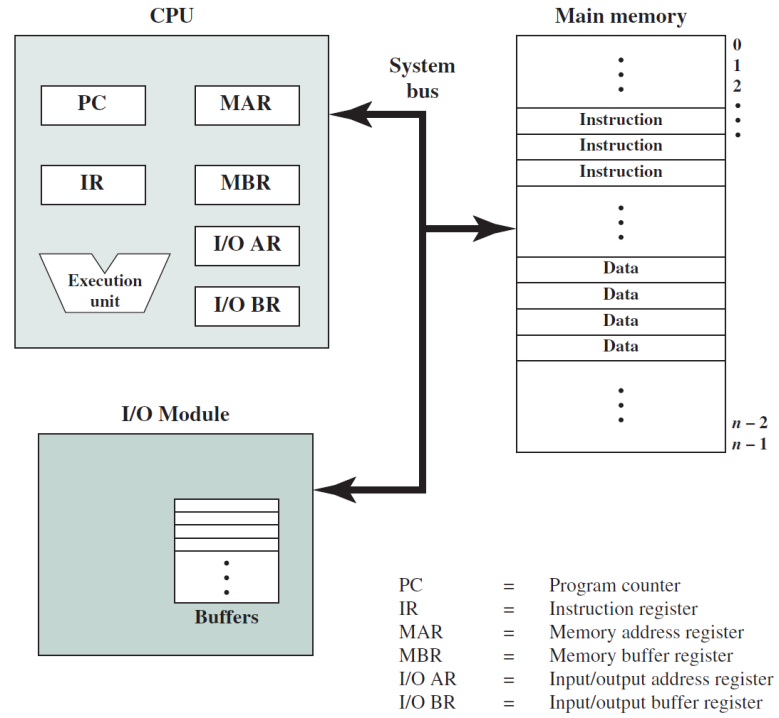
High-level Components of a Computer



INSTRUCTIONS & OPERANDS

- **A computer needs a physical location from which to retrieve the Instructions & operands**
- **A computer retrieves operands fro**
  - Registers
  - Constants (also called immediates)
  - Memory

~300 cycles to hit Main Memory;
Registers & Caches:
Register (CPU Scratchpad): 1 cycle.
L1: 3-4 cycles, 32kB Inst/Data
L2: 8-14 cycles, 256kB (combined)
L3: 20-50, few MBs (combined)

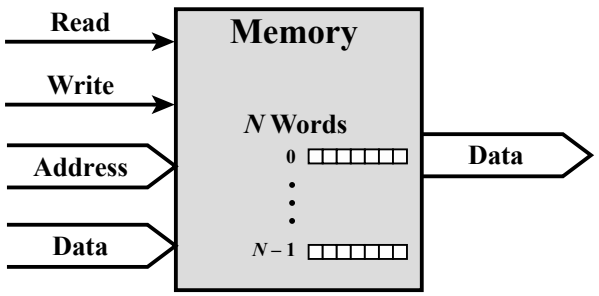


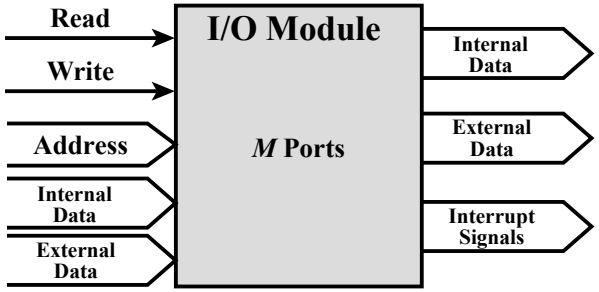
- **Main Memory is slow – Registers for Faster Access.**
- **Most architectures have a small set of (fast) registers**
  - MIPS has *thirty-two* 32-bit registers

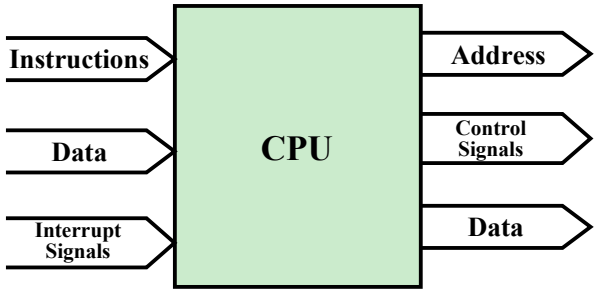
Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.

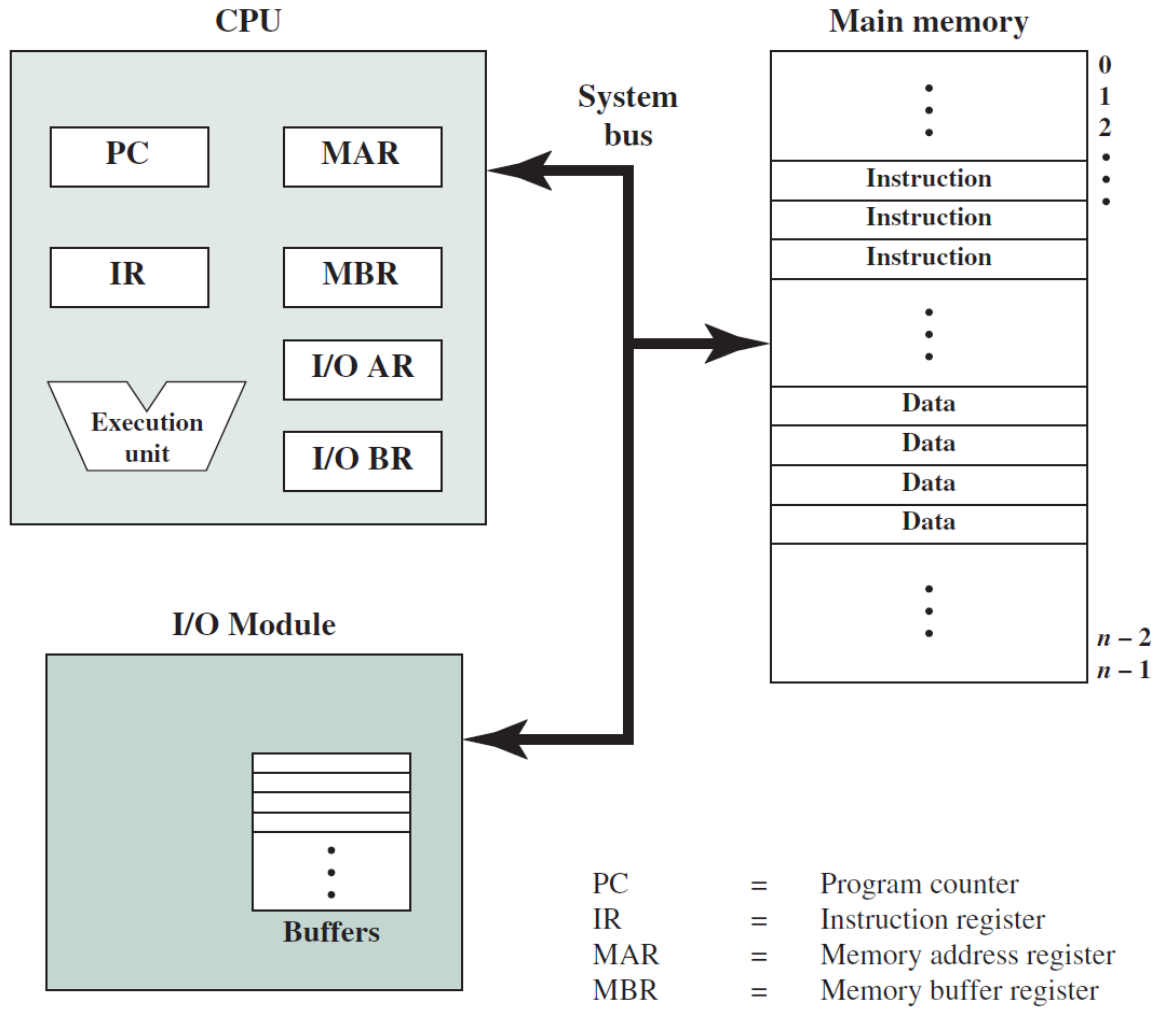
- **MIPS is called a 32-bit architecture because its CPU operates on 32-bit data**
  - A 64-bit version of MIPS also exists, but we will consider only the 32-bit version

Peeling another layer – Conceptual View of Computer Components



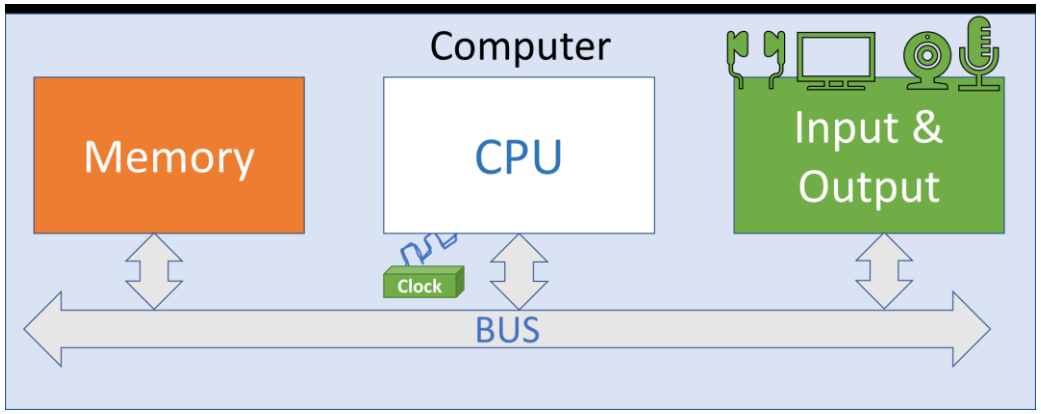




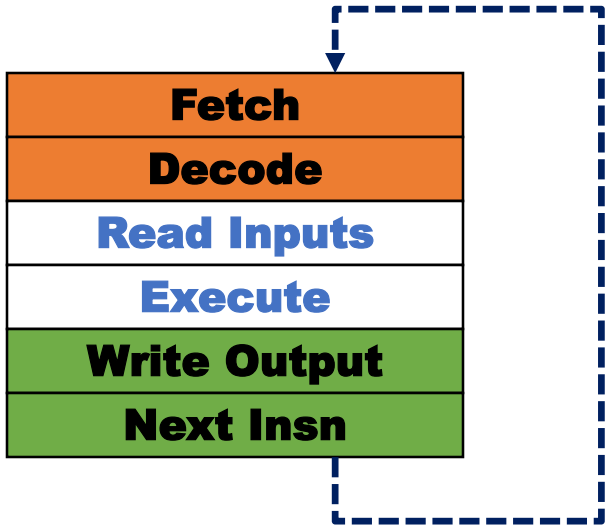


PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

Instruction Execution Model



Fetch → Decode → Execute "cycle"

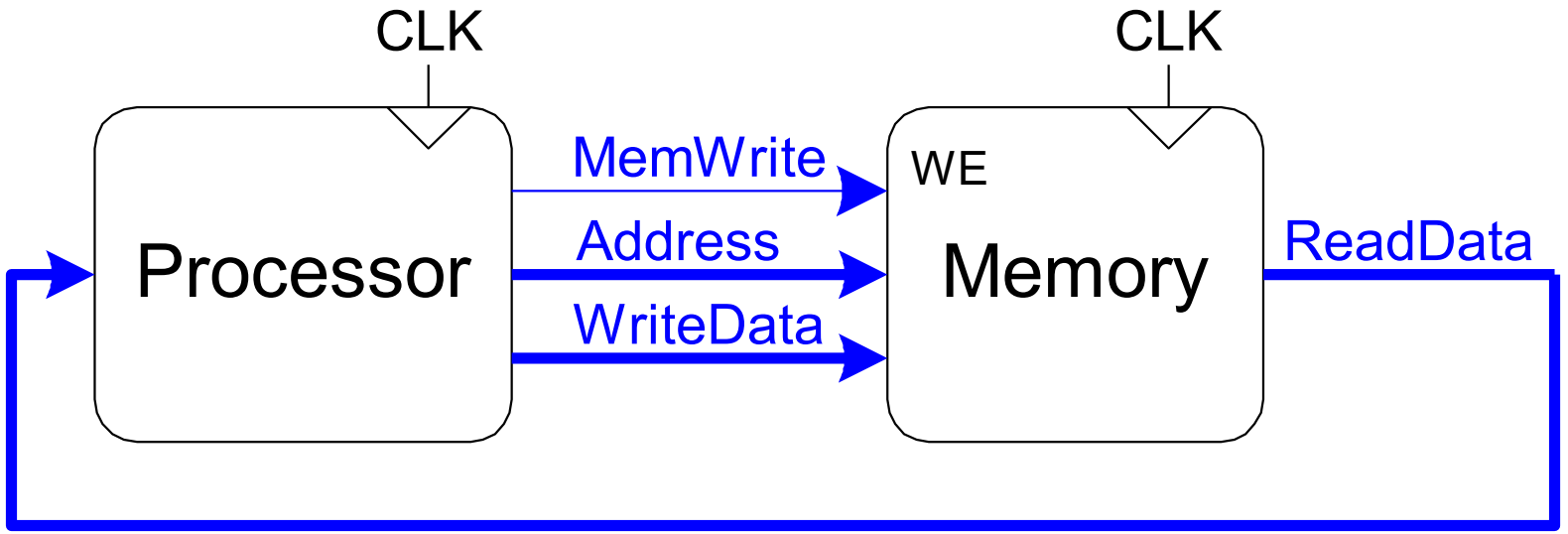


- The computer is just finite state machine
  - **Registers** (few of them, but fast)
  - **Memory** (lots of memory, but slower)
  - **Program counter** (next instruction to execute)

- A computer executes **instructions**
  - **Fetches** next instruction from memory
  - **Decodes** it (figure out what it does)
  - **Reads** its **inputs** (registers & memory)
  - **Executes** it (adds, multiply, etc.)
  - **Write** its **outputs** (registers & memory)
  - **Next insn** (adjust the program counter)

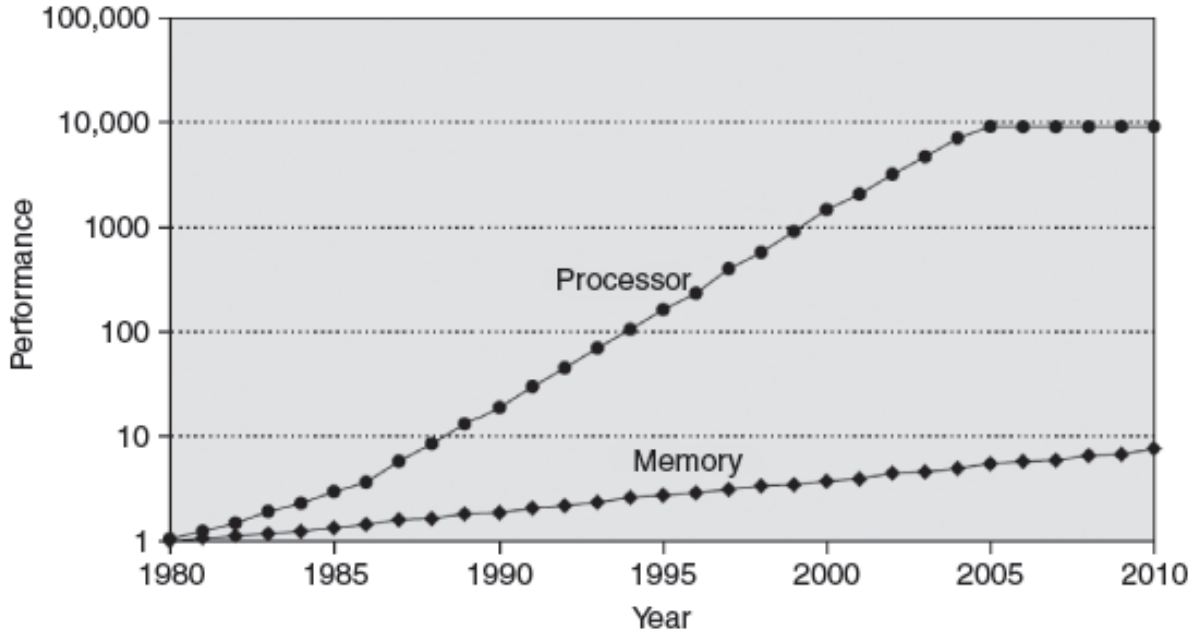
- **Program is just "data in memory"**
  - Makes computers programmable ("universal")

- Computer performance depends on:
  - Processor performance
  - Memory system performance
- Processor can compute only as fast as memory
  - A 3Ghz processor can execute an "add" operation in 0.33ns
  - Today's "Main memory" latency is more than 33ns (more than 100x slower)
  - Naïve implementation: loads/stores can be 100x slower than other operations



The Memory Wall

We assumed access memory in 1 clock cycle – but hasn't been true since the 1980's



Processors get faster more quickly than memory (note the log scale)

- Processor speed improvement: 35% to 55%
- Memory latency improvement: 7%

Key Question is How do we bridge this Gap?

- System Design and Optimizations
- Latency Hiding Techniques..

Principle of Locality

Exploit locality to make memory accesses fast

- **Temporal Locality:**
  - Locality in time
  - If data used recently, likely to use it again soon
  - **How to exploit:** keep recently accessed data in higher levels of memory hierarchy

- **Spatial Locality:**
  - Locality in space
  - If data used recently, likely to use nearby data soon
  - **How to exploit:** when access data, bring nearby data into higher levels of memory hierarchy too

Spatial and Temporal Locality Example

- Which memory accesses demonstrate spatial locality?
- Which memory accesses demonstrate temporal locality?

```
int sum = 0;
int X[1000];

for(int c = 0; c < 1000; c++){
  sum += c;

  X[c] = 0;
}
```

```
for (i=0;i<100;i++)
   for (j=0;j<3;j++)
```

```
for (i=0;i<5;i++)
   for (j=0;j<1;j++)
```

MEMORY SYSTEMS – KEY CHALLENGES

- Ideal memory:
  - Fast
  - Large (capacity)
  - Cheap (inexpensive)

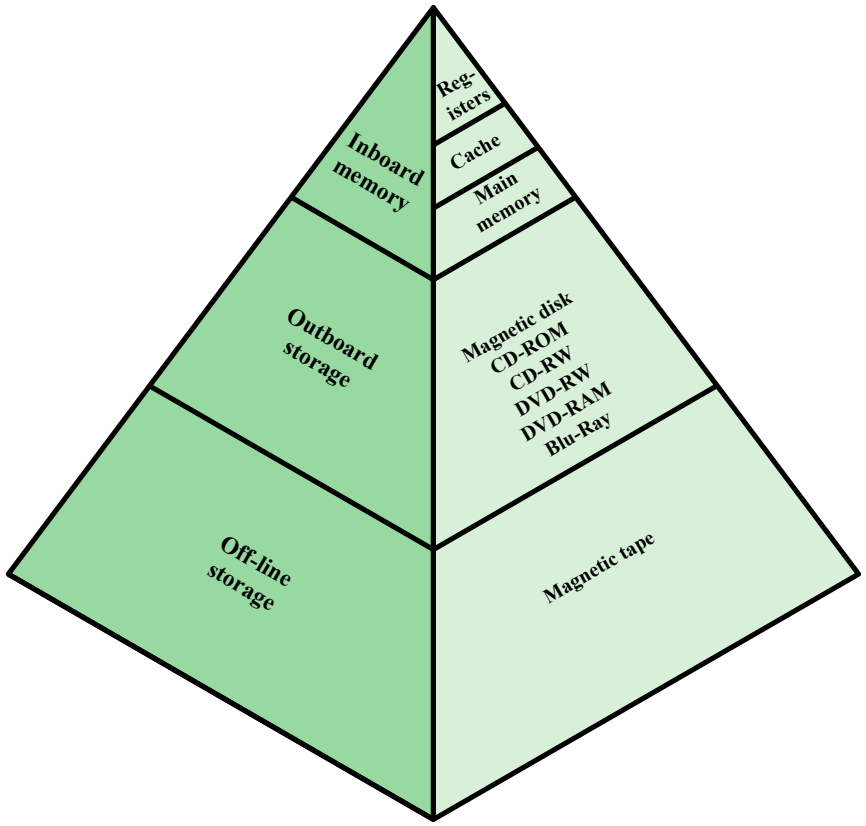
- Unobtainable goal:
  - Memory that operates at processor speeds
  - Memory as large as needed for all running programs
  - Memory that is cost effective

Memory Hierarchy

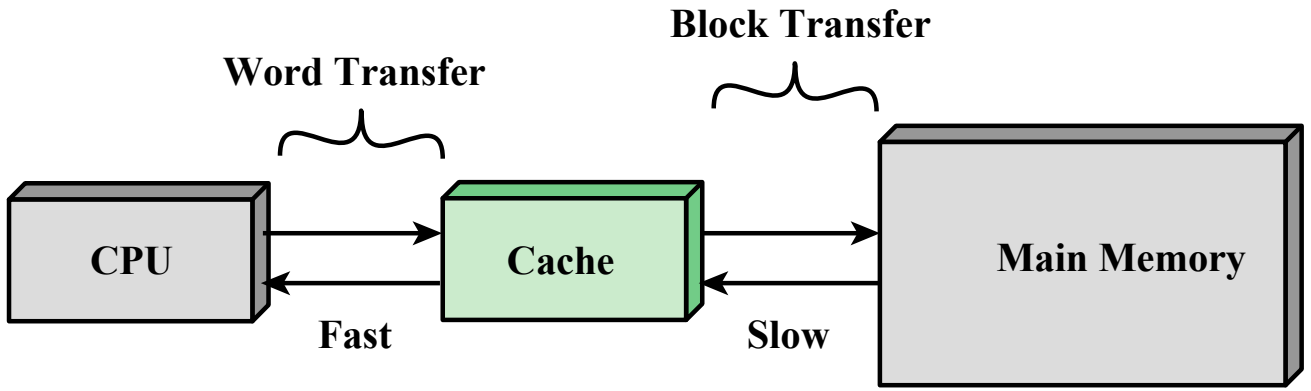
- Design constraints on a computer's memory can be summed up by three questions:
  - How much? how fast? how expensive?

- There is a trade-off among capacity, access time, and cost
  - Faster access time, greater cost per bit
  - Greater capacity, smaller cost per bit
  - Greater capacity, slower access time

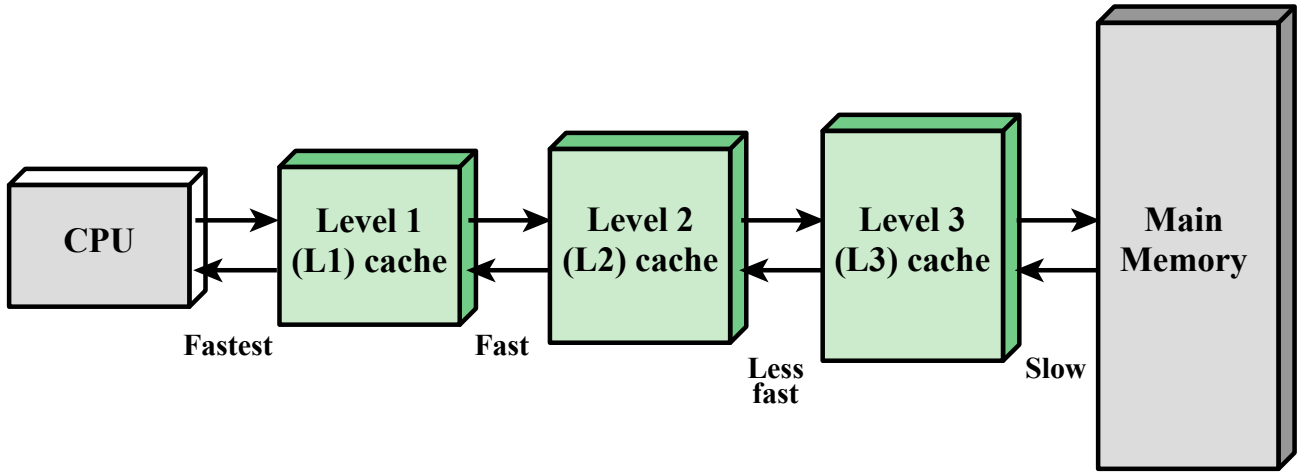
- The way out of the ***memory dilemma*** is not to rely on a single memory component or technology, but to employ a memory hierarchy



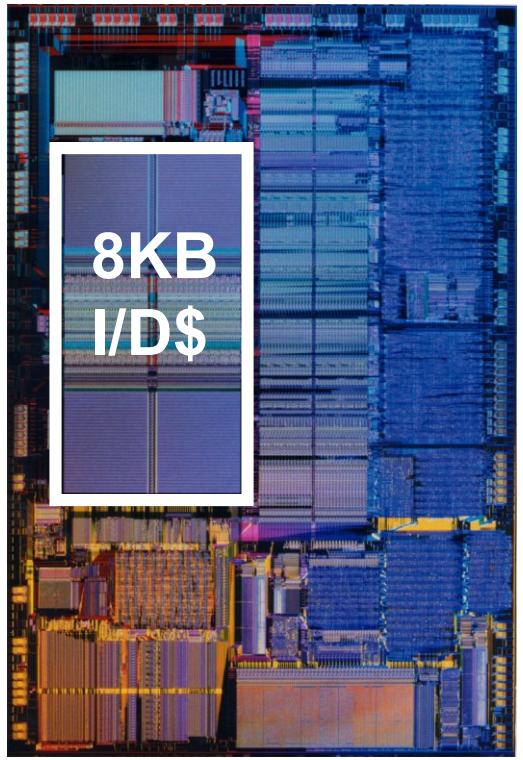
Evolution of Cache Hierarchies



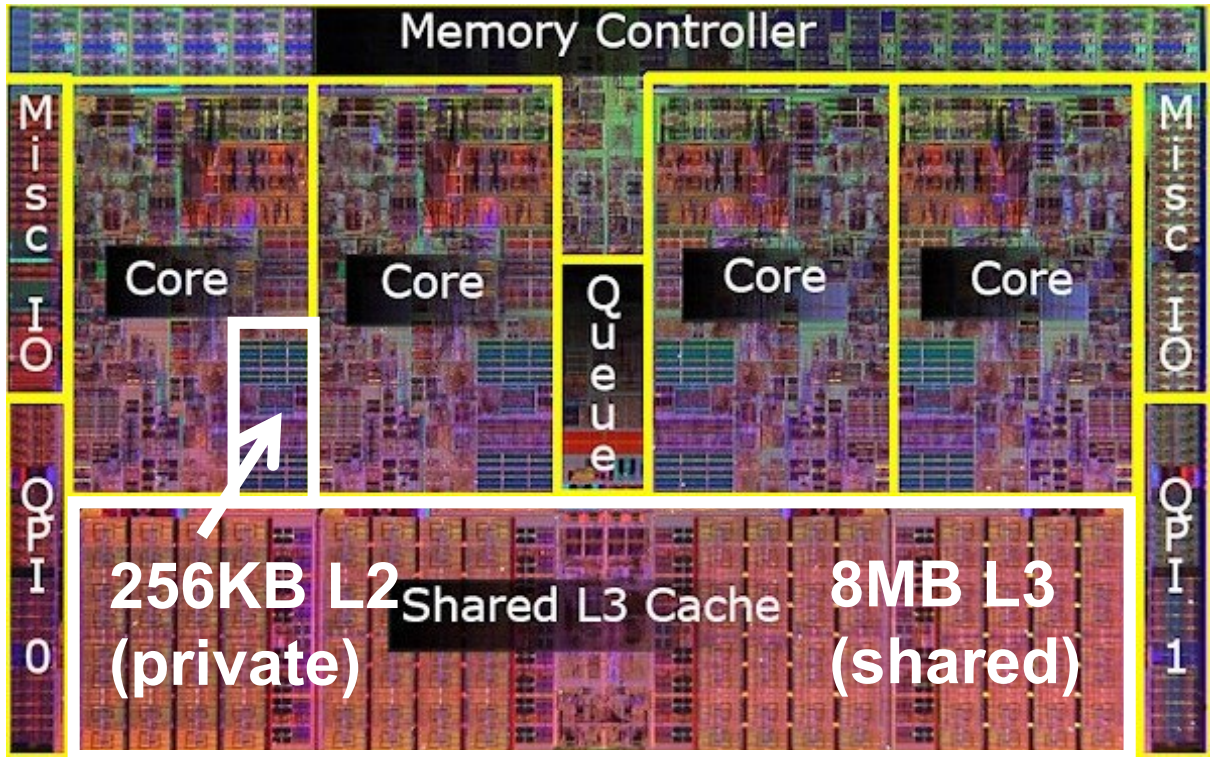
(a) Single cache



Evolution of Cache Hierarchies



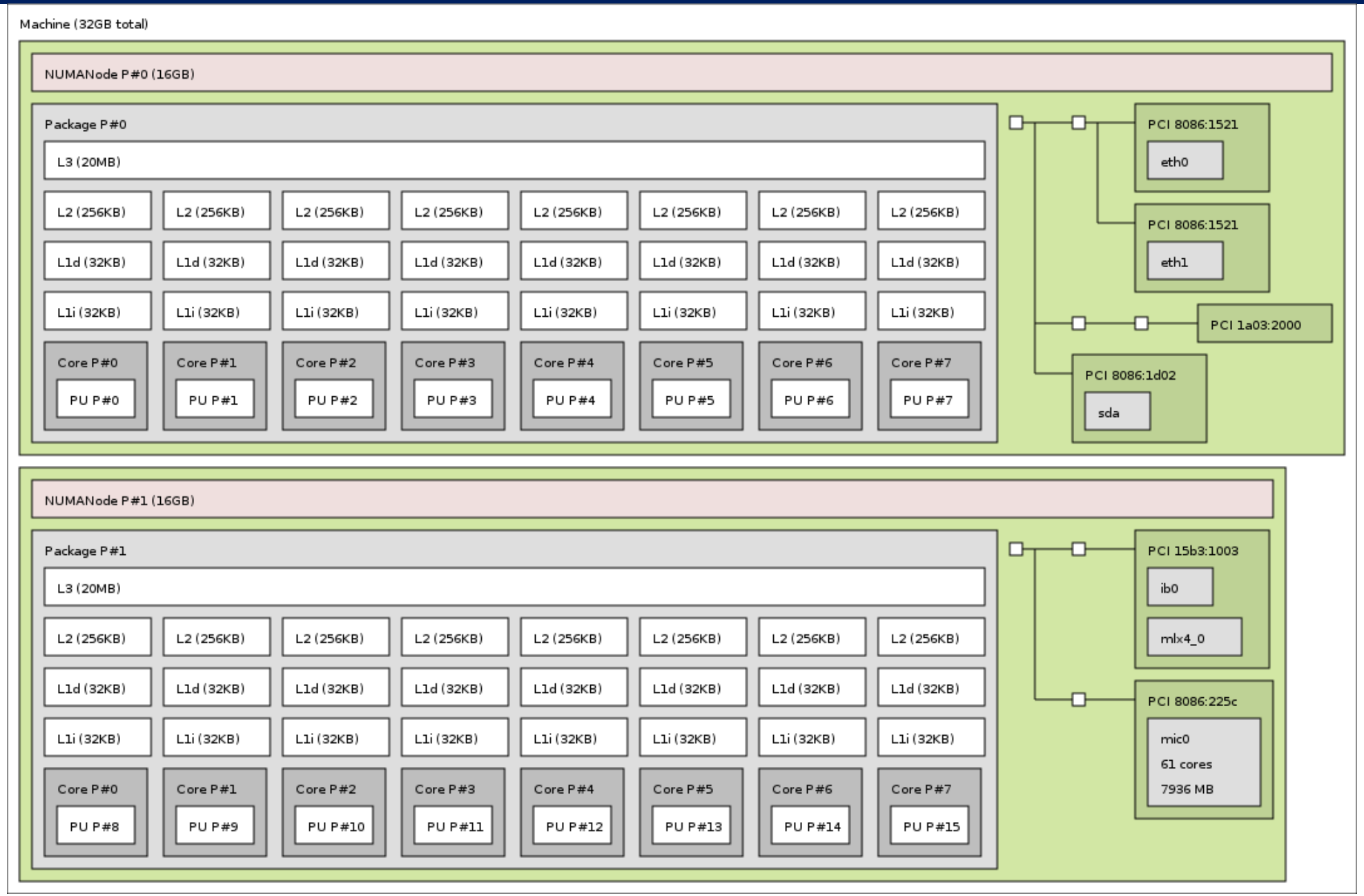
Intel 486



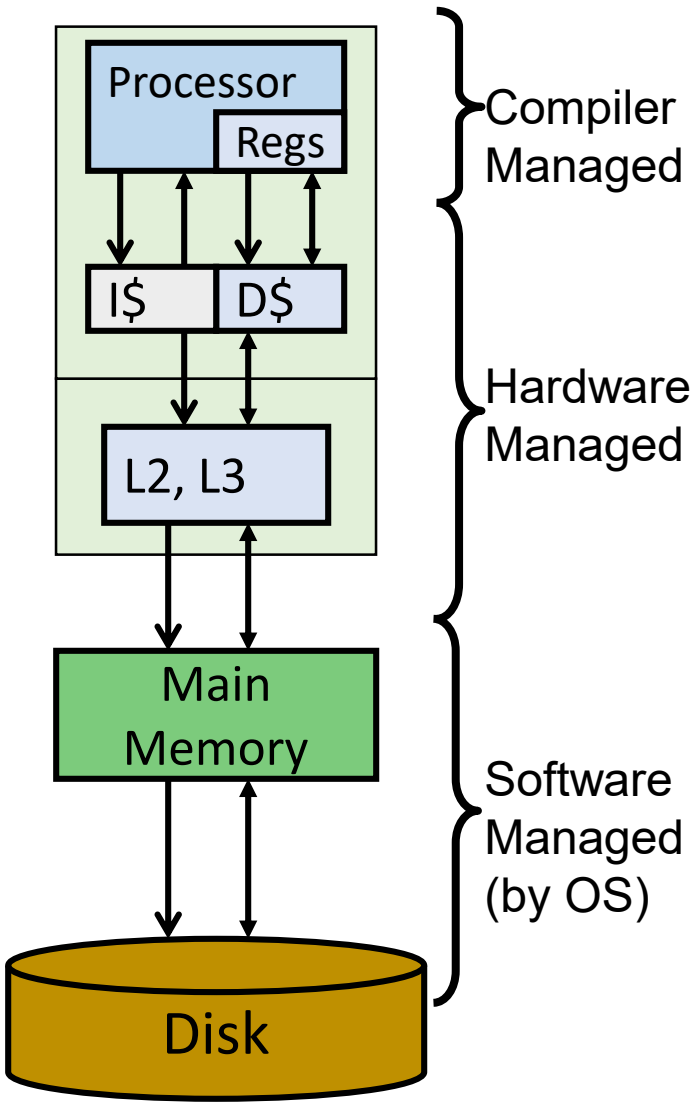
Intel Core i7 (quad core)

- Chips today are 30–70% cache by area

Memory Hierarchy in Modern Server Machines

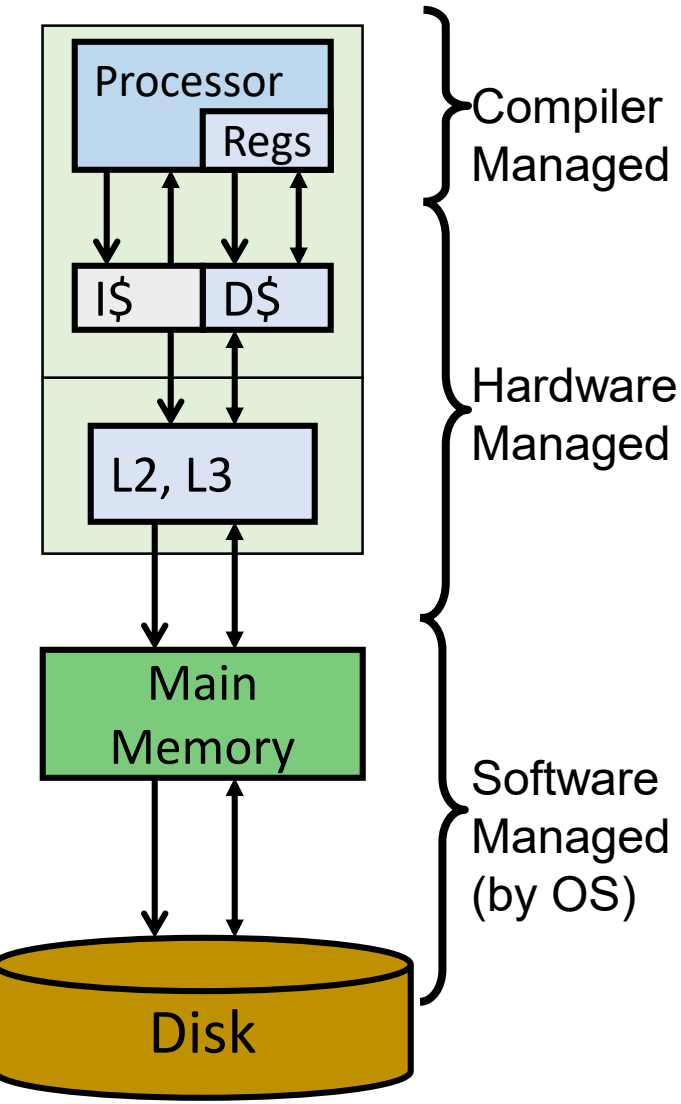


Concrete Memory Hierarchy

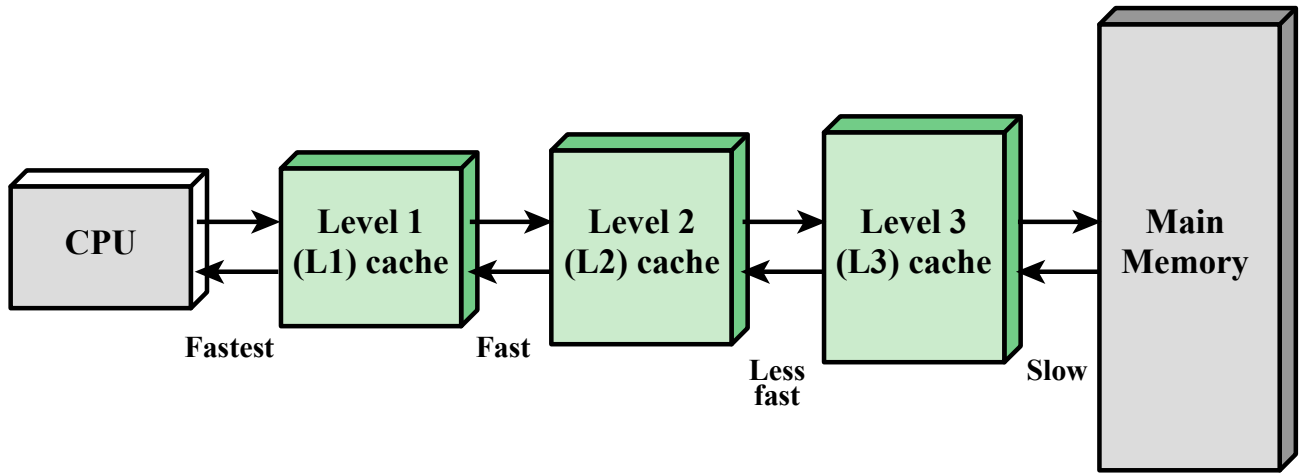


- 0th level: **Registers**
- 1st level: **Primary caches**
  - Split instruction (I$) and data (D$)
  - Typically 8KB to 64KB each
- 2nd level: **2nd and 3rd cache** (L2, L3)
  - On-chip, typically made of SRAM
  - 2nd level typically ~256KB to 512KB
  - "Last level cache" typically 4MB to 16MB
- 3rd level: **main memory**
  - Made of DRAM ("Dynamic" RAM)
  - Typically 1GB to 4GB for desktops/laptops
    - Servers can have 100s of GB
- 4th level: **disk (swap and files)**
  - Uses magnetic disks or flash drives

General Memory Hierarchy

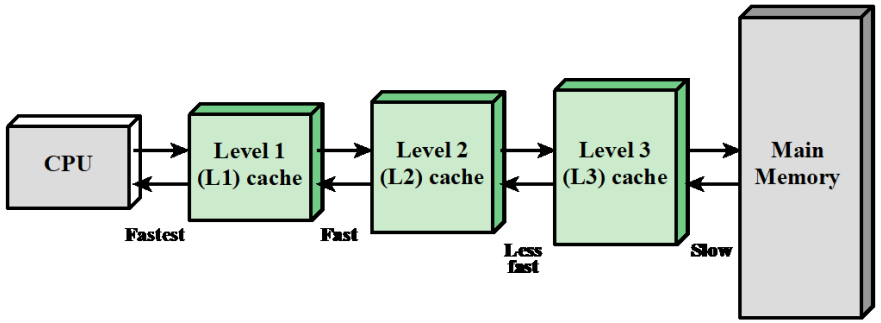


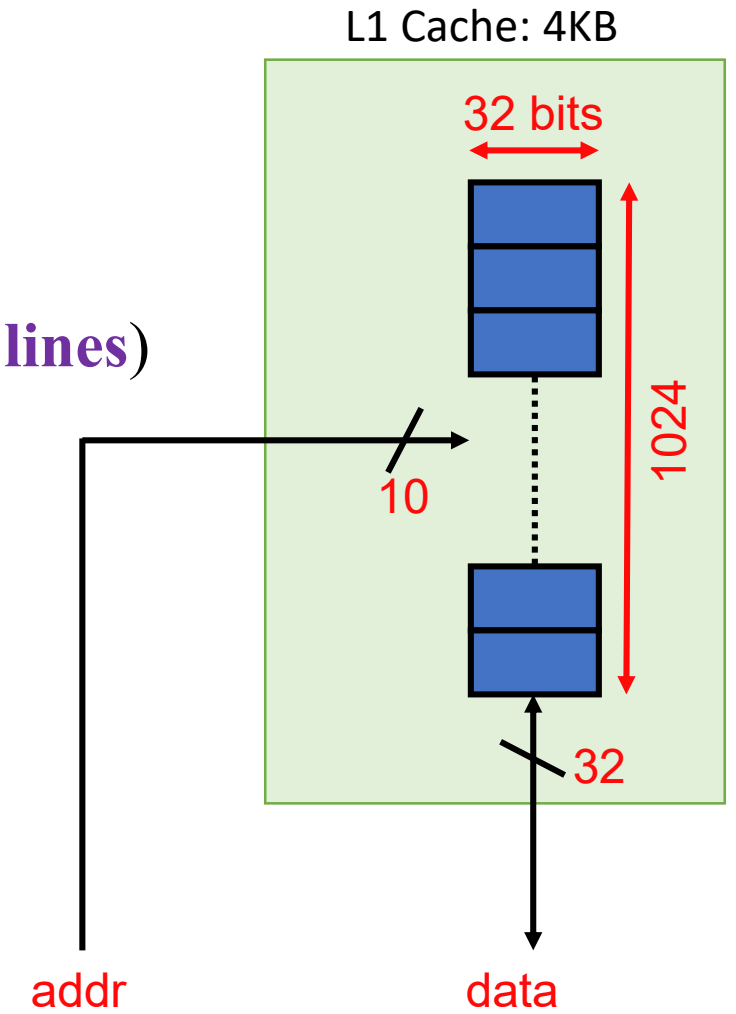
	Speed	Size/Capacity	Access Time	Technology	Price/GB
Register	Fastest	Smallest (~1KB)	<1ns	FFs	--
Cache (L1/L2/L3)	Fast	Small (32KB~MB)	~1-10ns	SRAM	$100- $1,000
Main Memory	Slow	Large (1-256 GB)	10-100 ns	DRAM	$1-$10
Virtual/Secondary Memory	Slowest	Largest (Few TBs)	0.1-10 ms	SSD/HDD	$1/$0.1



Cache Organization

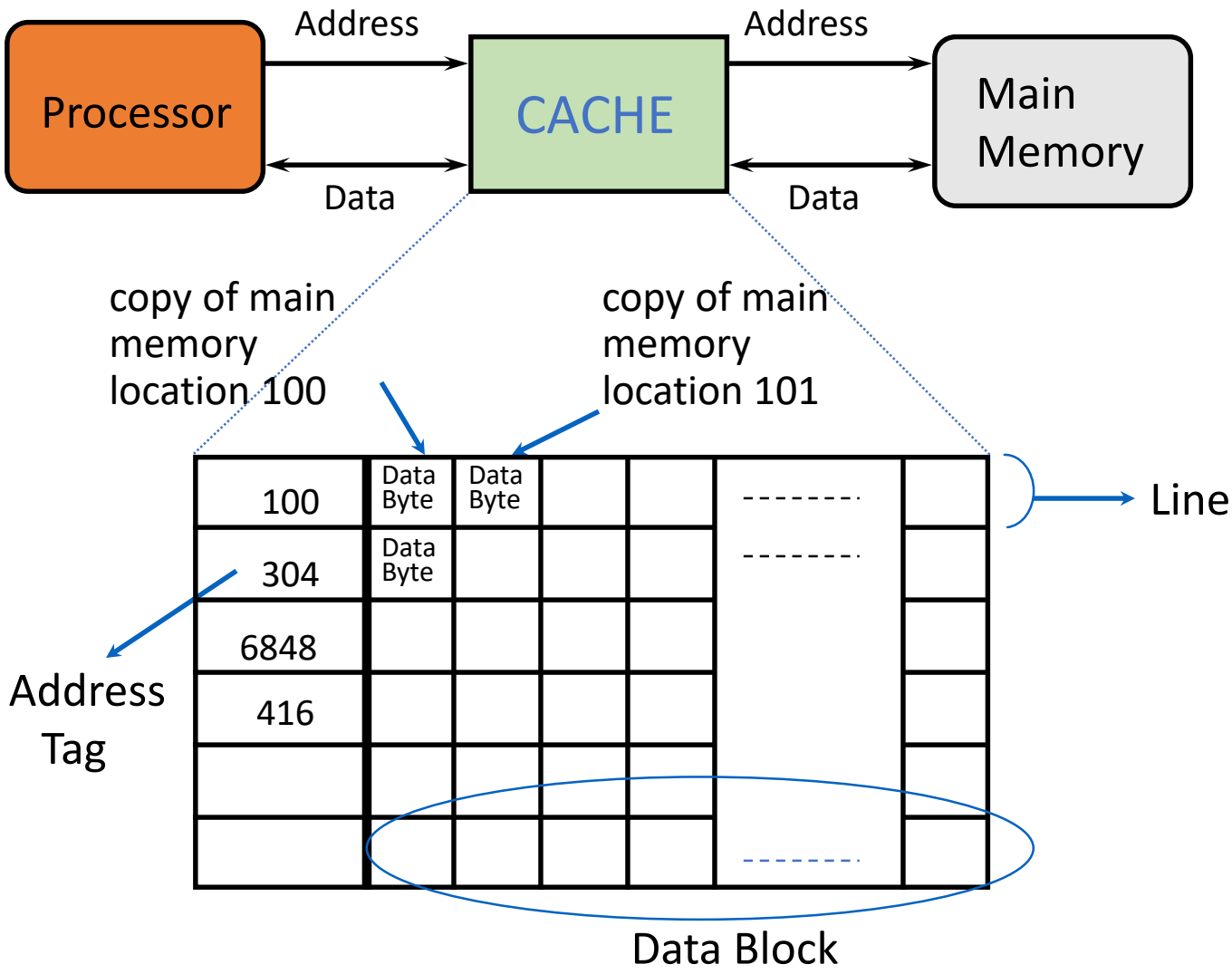
- Cache – a generic term for any structure that "***memoizes***".
- Memory Cache is a hardware hashtable – an array of **sets;**
  - Ideally supplies most data to processor
  - Usually holds most recently accessed data
- Logical cache organization: ***Sets*** and ***Blocks***
  - 32KB, organized as 1K (1024 **entries**) of 4B **blocks** (aka **cache lines**)
  - Each block can hold a 4-byte word.
- Physical cache implementation
  - Automatically (Hardware) managed memory, typically SRAM
  - 1K (1024 entries) by 32bits (4 Bytes)
  - 10-bit address input
  - 32-bit data input/output



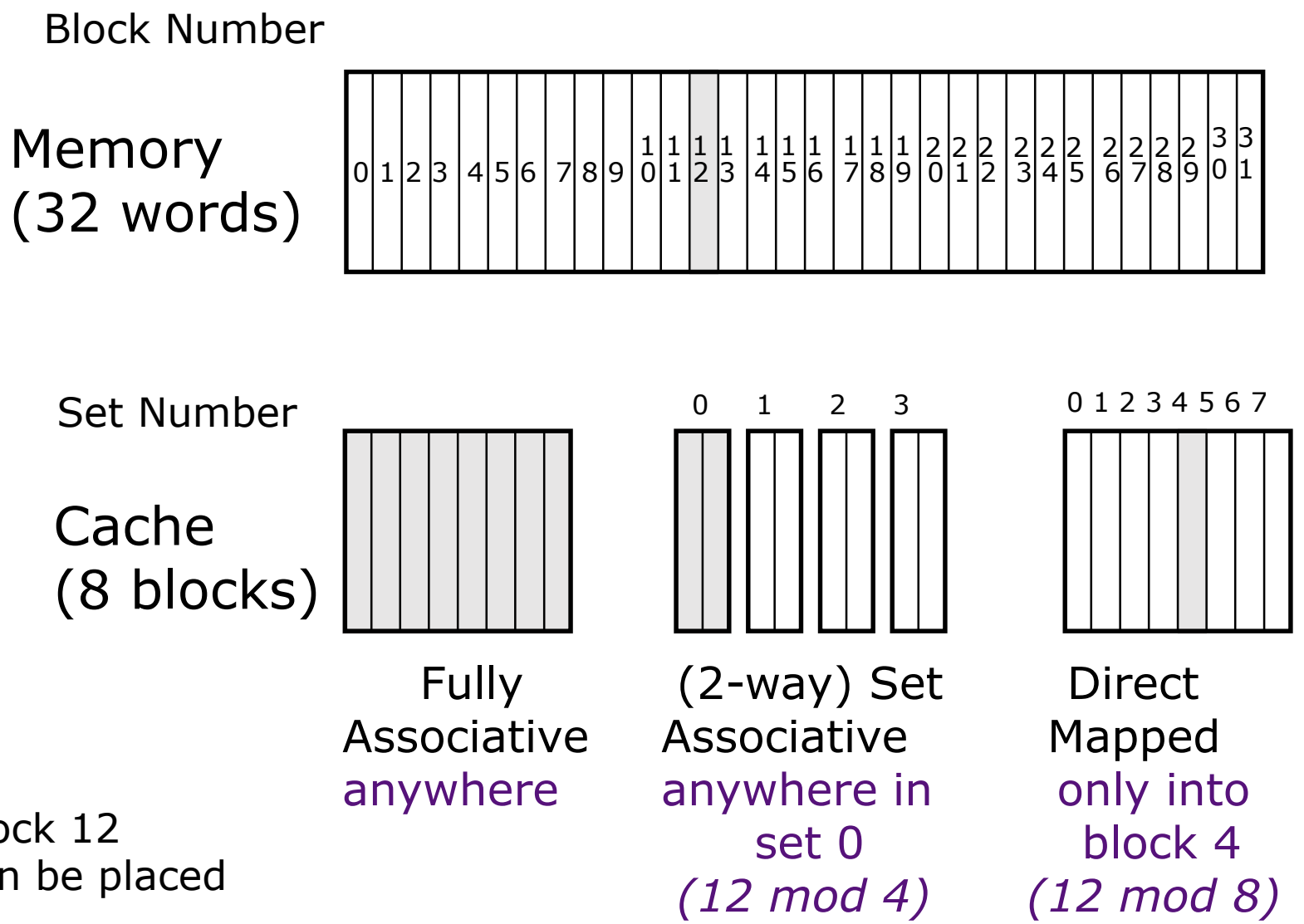


Cache Characteristics and Basic Cache Terminology

- **Capacity (*C*):**
  - Total number of data bytes in cache
- **Block size (*b*):**
  - bytes of data brought into cache at once
- **Number of blocks (*B = C/b*):**
  - number of blocks in cache: *B = C/b*
- **Degree of associativity (*N*):**
  - number of blocks in a set
- **Number of sets (*S = B/N*):**
  - each memory address maps to exactly one cache set



Different Cache Organizations

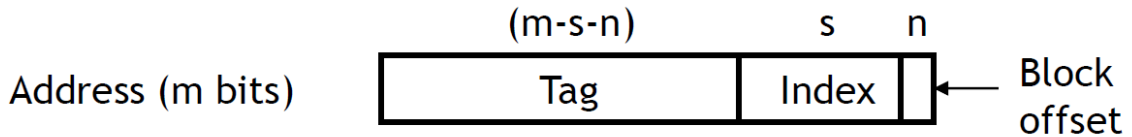


4-bit Address, 8B Cache, 2B Blocks

Main memory (16 Bytes, Byte addressable) → Total Address bits = $Log_2(16)$

0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	I
1001	J
1010	K
1011	L
1100	M
1101	N
1110	P
1111	Q

How do we access cache from the Memory Address bits ?



Address bits m = 4

Cache (8 Bytes, Direct Mapped, 2B blocks)

		0	1
00	0	A	B
01	0	C	D
10	0	E	F
11	0	G	H

2B blocks➔ *b = 2*

Block offset bits n = $Log_2(2)$=1; n=1 (LSB)

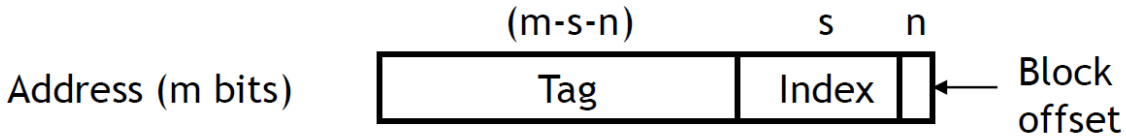
Direct Mapped ➔ (N=1 way associative)

#sets = #blocks = C/b = 4 ➔ Index bits = 2

#Tag bits = (m - s - n) = (4-2-1) = 1 (MSB)

Direct Mapping vs N-Way Set Associative Cache

- **Capacity (*C*):**
  - number of data bytes in cache
- **Block size (*b*):**
  - bytes of data brought into cache at once
- **Number of blocks (*B = C/b*):**
  - number of blocks in cache: *B = C/b*
- **Degree of associativity (*N*):**
  - number of blocks in a set
- **Number of sets (*S = B/N*):**
  - each memory address maps to exactly one cache set



$$C = B \; x \; b$$

$$C = \{S \; x \; N\} \; x \; b$$

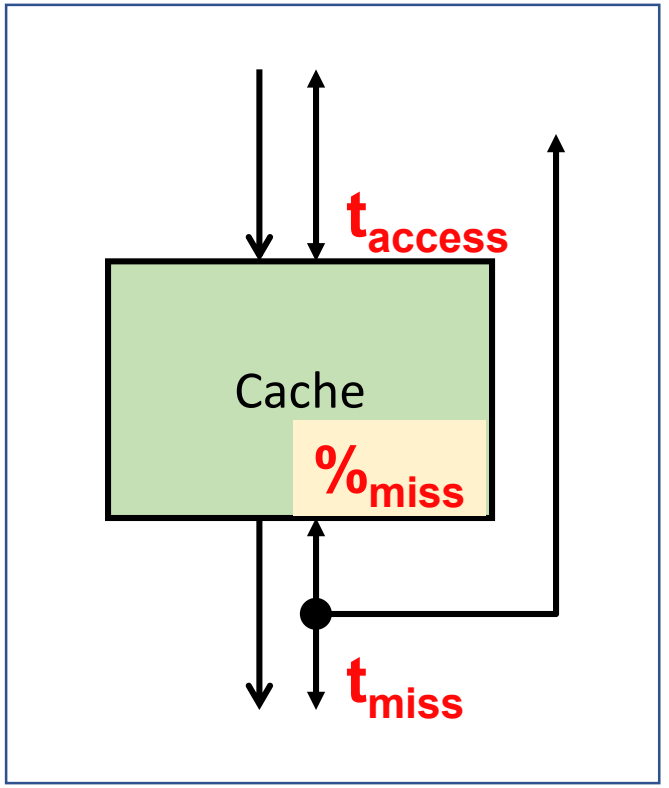
$$set\ index = (block\ address) mod (S)$$

Consider Memory: 32-bit Address and Cache 32KB, 4B Blocks

Cache organization	Degree of Associativity (N)	Num of sets (S=B/N)	#bits for block offset	#bits for set index	# bits for Tag	Tag Overhead (Tag bits/Block bits)
Direct-Mapped	1	8K	2	13	17	17/32
2-way Associative	2	4K	2	12	18	18/32
4-way Associative	4	2K	2	11	19	19/32
Fully Associative	8K	1	2	0	30	30/32

Cache Performance

- **Access**: read or write to cache
- **Hit**: desired data found in cache
- **Miss**: desired data not found in cache
  - Must get from another component (next component in the hierarchy)
  - No notion of "miss" in register file
- **Fill**: action of placing data into cache
- **$\%_{miss}$** (miss-rate): #misses / #accesses
- **t_{access}**: time to check cache. If hit, we're done.
- **t_{miss}**: time to read data into cache



- Performance metric: average access time

$$t_{avg} = t_{access} + (\%_{miss} * t_{miss})$$

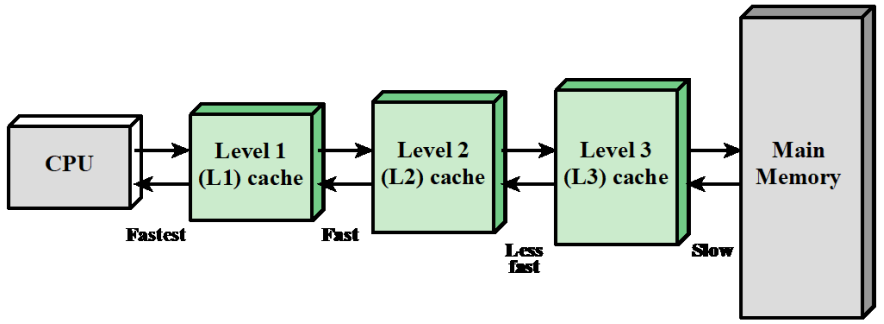
CACHE -- HIERARCHICAL LATENCY ANALYSIS

- Memory hierarchy (level i) has a technology-intrinsic access time of t_i.

- Except for the *outer-most hierarchy (MM)*, when looking for a given address there is
  - a chance (hit-rate h_i) you "hit" with access time is t_i
  - a chance (miss-rate m_i) you "miss" and fetch from next level with access time $t_i + T_{i+1}$
  - $h_i + m_i = 1$
- Thus

$$T_i = h_i \cdot t_i + m_i \cdot (t_i + T_{i+1})$$

$$T_i = t_i + m_i \cdot T_{i+1}$$



Note: h_i and m_i are the hit-rate and miss-rate of the references that missed at L_{i-1}

- Recursive cache access latency equation.

$$T_i = t_i + m_i \cdot T_{i+1}$$

CPI Calculation with Cache Misses

- Parameters
  - Simple pipeline with base CPI of 1
  - Instruction mix: 30% loads/stores
  - I$: $\%_{miss}$ = 2%, t_{miss} = 10 cycles
  - D$: $\%_{miss}$ = 10%, t_{miss} = 10 cycles

- What is new CPI?
  - $CPI_{I\$} = \%_{missI\$} * t_{miss}$ = 0.02*10 cycles = 0.2 cycle
  - $CPI_{D\$} = \%_{load/store} * \%_{missD\$} * t_{missD\$}$ = 0.3 * 0.1*10 cycles = 0.3 cycle
  - $CPI_{new} = CPI + CPI_{I\$} + CPI_{D\$}$ = 1+0.2+0.3 = 1.5
  - Note: Two factors impact CPI: ***Miss Rate*** and ***Miss Penalty***

- What is new CPU time?

$$CPUtime = IC * (CPI_{new}) * \text{Clock Cycle time}$$

Cache Miss and Ways to reduce Cache Miss Rate

- Understanding the types of cache miss: The Three C's + '*C*'
  - **Compulsory (Cold-start)**: the first access to a block in cache
  - **Capacity**: desired data cannot be contained in the cache (size limitation)
  - **Conflict (Collision)**: desired data conflicts with other content in the cache.
  - ***Coherence*** : miss due to external invalidations (multiprocessor architecture)

Question: How do we minimize these Cache Misses?

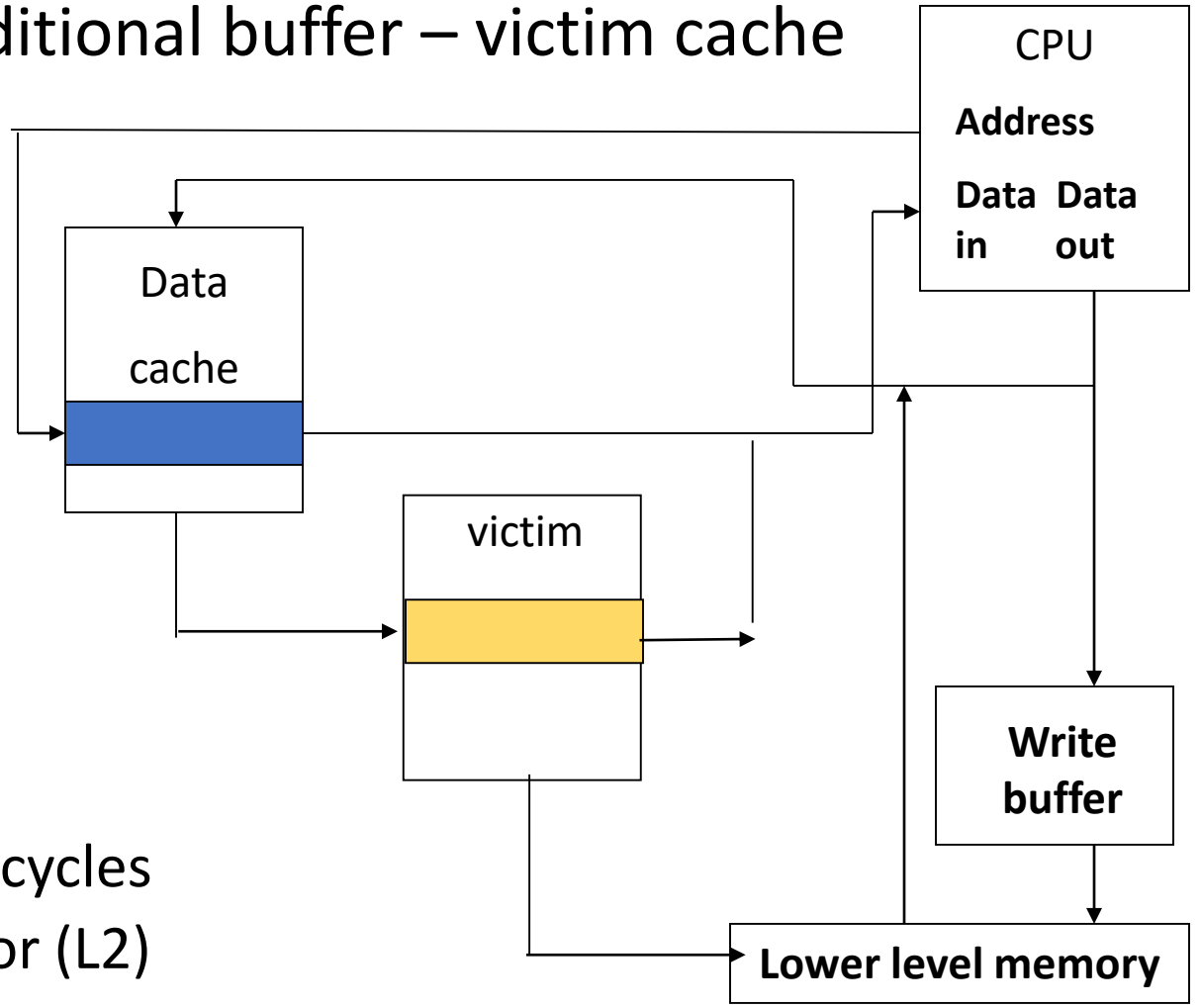
Reduce the Miss Rate and/or Reduce the ***Miss Penalty***
May Also ***Reduce the Hit Time***

Techniques to reduce the Cache Misses

1. Larger Caches
2. Larger Block Size
3. Higher associativity
4. Victim and Pseudo-associative caches
5. Prefetching of Instruction and Data
6. Code optimizations (Developer/Compiler)

VICTIM CACHES

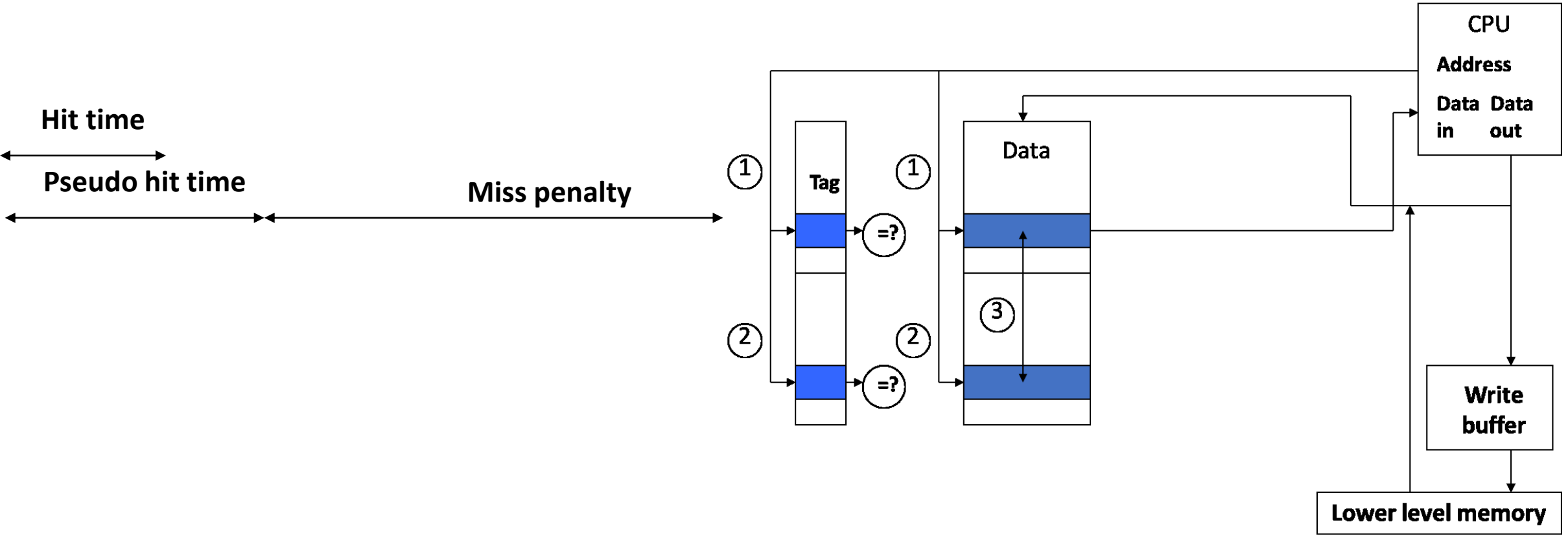
- Minimize penalty for frequent Conflict/Capacity misses.
- Save data discarded from cache into a additional buffer – victim cache



- Drawback:
  - CPU pipeline design is hard if hit takes 1 or 2 cycles
  - Better for caches not tied directly to processor (L2)
  - Used in MIPS R1000 L2 cache

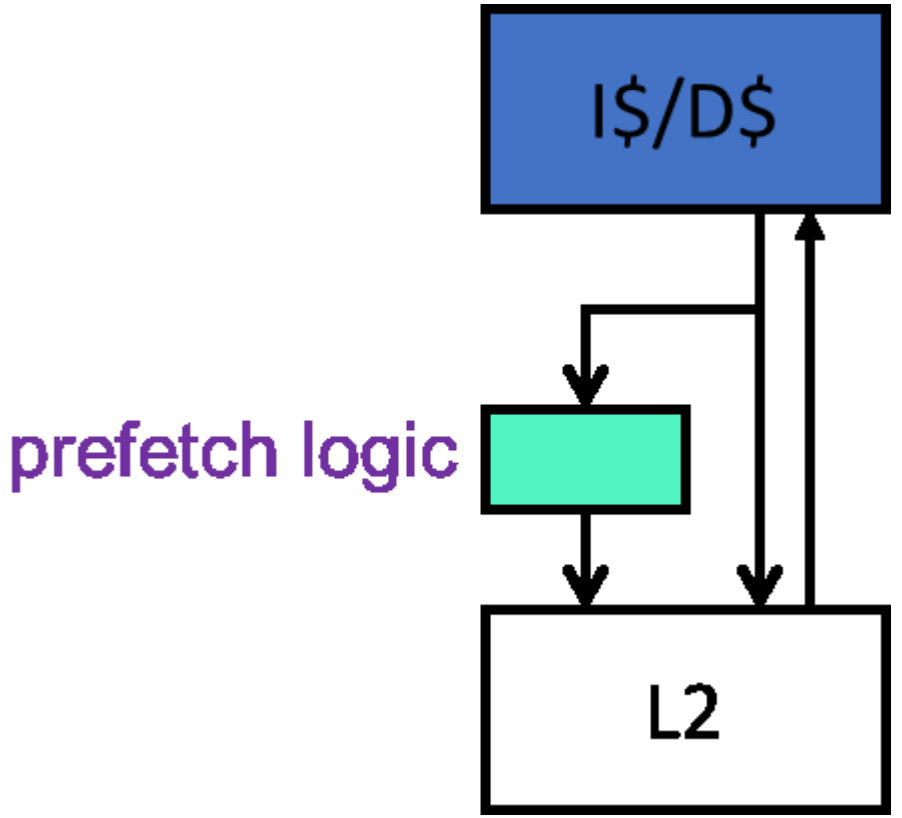
Pseudo-Associative Caches

- Fast hit time of *direct mapped* + lower conflict misses of 2-way SA cache.
- Divide cache: on a miss, check other half of cache; yes? then a pseudo-hit (slow)



PREFETCHING

- Bring data into cache **proactively/speculatively**
  - If successful, reduces the number of caches misses
- Key: ***anticipate*** *upcoming miss addresses* accurately
  - Can be done in either software or hardware.



- Simple hardware prefetching: **next block prefetching**
  - Miss on address **X** $\rightarrow$ anticipate miss on **X+block-size**

  \+ Works for insns: sequential execution

  \+ Works for data: arrays
- Table-driven hardware prefetching
  - Use **predictor** to detect strides, common access patter
- Effectiveness of Prefetching is determined by:
  - **Coverage**: prefetch for as many misses as possible
  - **Timeliness**: initiate prefetches sufficiently in advance
  - **Accuracy**: don't pollute with unnecessary data

Software Prefetching – ISA support?

- Use a special "**prefetch**" instruction
  - Tells the hardware to bring in data
  - Just a hint
- Inserted by programmer or compiler to assist the HW.

Example

```
int tree_add(tree_t* t) {
  if (t == NULL) return 0;
  __builtin_prefetch(t->left);
  __builtin_prefetch(t->right);
  return t->val + tree_add(t->right) + tree_add(t->left);
}
```

- Multiple prefetches bring multiple blocks in parallel → More "Memory-level" parallelism (MLP)

Question: How early should we prefetch?

For more details: Read https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html

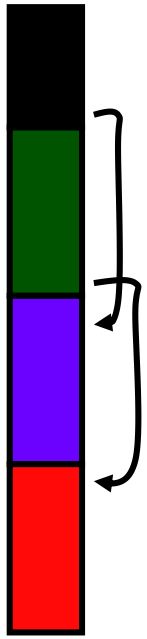
Code Optimizations

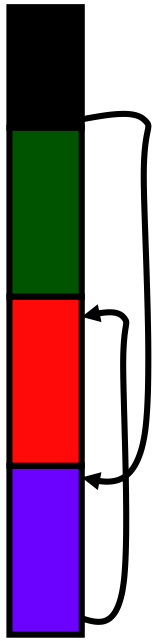
- Avoid hardware changes
- Instructions
  - ***Code Restructuring/Reordering*** to better utilize cache locality.
  - ***Explicit Profiling*** to look at conflicts between groups of instructions

- Data
  - ***Merging Arrays*****:** improve spatial locality by single array of compound elements vs. 2 arrays
  - ***Loop Interchange***: change nesting of loops to access data in order stored in memory
  - ***Loop Fusion***: Combine 2 independent loops that have same looping construct and some overlapping variables
  - ***Blocking***: Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows.

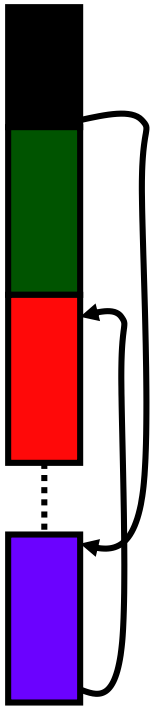
Software Restructuring: Code

- Compiler can lay out code for temporal and spatial locality
  - **If** (**a**) { **code1;** } else { **code2;** } **code3; Nextcode;**
  - But, **code2** case never happens (say, error condition)





+ Better locality
+ Fewer taken branches



+ Better locality for code after **code3**
+ Fewer taken branches

Merging Arrays, Loop Interchange and Loop Fusion

```
/* Before: 2 sequential arrays */
        int key[SIZE];
        int val[SIZE];
```

2 misses per access to key & value + potential conflict;

```
/* After: 1 array of stuctures */
        struct merge {
                int key;
                int val;
        };
        struct merge
merged_array[SIZE];
```

Reducing conflicts between val & key; improved spatial locality

```
/* Before : Column Major Access*/
        for (j = 0; j < COLS; j = j+1)
                for (i = 0; i < ROWS; i = i+1)
                        sum += x[i][j];
```

•Striding through memory every "COL" words;

```
/* After */
        for (i = 0; i < ROWS; i = i+1)
                for (j = 0; j < COLS; j = j+1)
                        sum += x[i][j];
```

•Sequential accesses instead of striding through memory every COL words; improved spatial locality

```
/* Before */
for (i = 0; i < N; i = i+1)
        for (j = 0; j < N; j = j+1)
                a[i][j] = b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
        for (j = 0; j < N; j = j+1)
                d[i][j] = a[i][j] + c[i][j];
```

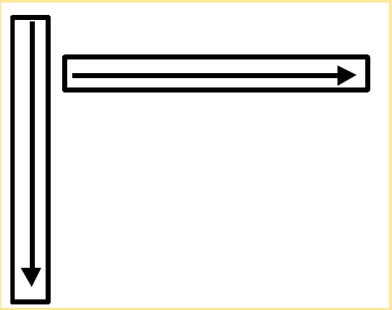
2 misses per access to a & c

```
/* After */
for (i = 0; i < N; i = i+1)
        for (j = 0; j < N; j = j+1)
        {       a[i][j] = b[i][j] * c[i][j];
                d[i][j] = a[i][j] + c[i][j];
        }
```

Now one miss per access; improve temporal locality

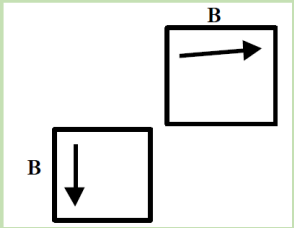
BLOCKING

```
/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
  {
    r = 0;
    for (k = 0; k < N; k = k+1)
      r + = y[i][k]*z[k][j];
    x[i][j] = r;
  }
```



- Two Inner Loops:
  - Read all NxN elements of z[]
  - Read N elements of 1 row of y[] (repeat
  - Write N elements of 1 row of x[]
- Capacity Misses: function of N & CacheSize

```
/* After BLOCKING :: B = BLOCK_SIZE*/
  for (jj = 0; jj < N; jj = jj+B)
    for (kk = 0; kk < N; kk = kk+B)
      for (i = 0; i < N; i = i+1)
        for (j = jj; j < min(jj+B-1,N); j=j+1)
        {
          r = 0;
          for(k=kk; k<min(kk+B-1,N);k =k+1)
            r += y[i][k]*z[k][j];
          x[i][j] = x[i][j] + r;
        }
```



- Idea: compute on BxB submatrix that fits
  - B called *Blocking Factor*
  - Reduce Capacity Misses from $2N^3 + N^2$ to $2N^3/B + N^2$

Techniques to reduce the Cache Miss Penalty

1. Multi-Level Caches
2. Non-blocking Caches
3. Early restart / Critical Word First (Wrapped Fetch)
4. Sectoring or Sub-block Placement
5. Prioritize Read misses over writes

Techniques to reduce the Cache Hit time

1. Small and Simple caches
2. TLB/ Avoid Address Translations – more on this after later.
3. Pipelined writes.

Cache Optimization Summary

Technique	Miss Rate	Miss Penalty	Hit Time	Complexity
Large Cache	+		-	1
Large Block size	+	-		0
Higher Associativity	+		-	1
Victim cache	+			2
Pseudo Associate Cache	+			2
Prefetching	+			3
Prioritize Read misses		+		1
Critical Word first		+		1
Subblock placement		+	+	2
Non blocking Caches		+		3
Multi-Level Caches		+		2
Small and Simple Cache	-		+	0
TLB / Address Translation			+	2
Pipelined Writes			+	1
Multi-ports and Banking				

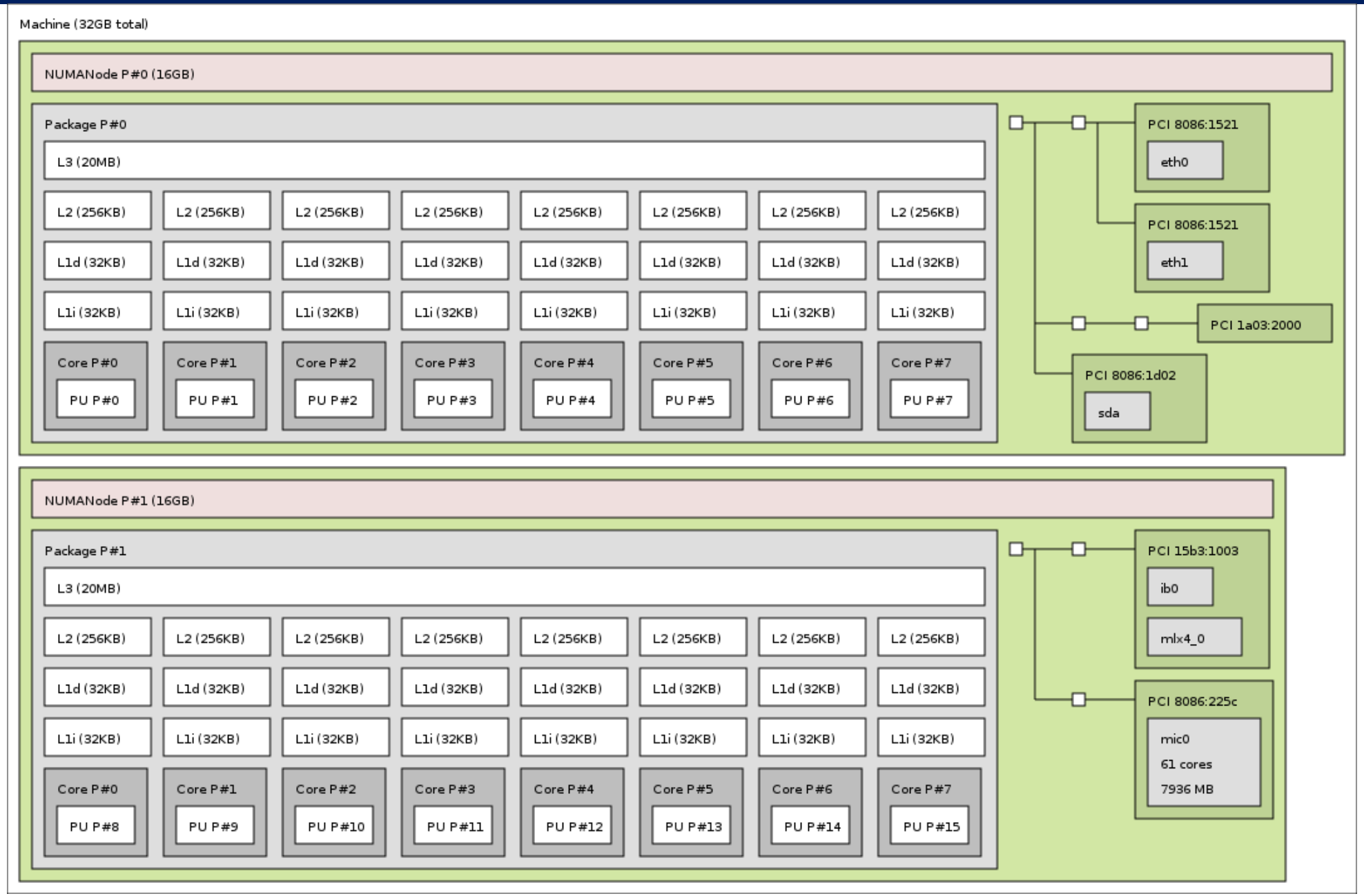
This slide is inspired by presentations of Randy Katz (UC Berkeley), CS 252.

Split vs. Unified Caches

- **Split I$/D$**: insns and data in different caches
  - To minimize structural hazards and t_{access}
  - Larger unified I$/D$ would be slow, 2nd port even slower
  - Optimize I$ and D$ separately
    - Not writes for I$, smaller reads for D$
  - Why is 486 I/D$ unified?

- **Unified L2, L3**: insns and data together
  - To minimize $\%_{miss}$
  - \+ Fewer capacity misses: ***unused insn capacity*** can be **used for data**
  - – More conflict misses: insn/data conflicts
    - A much smaller effect in large caches
  - Insn/data structural hazards are rare: simultaneous I$/D$ miss
  - Go even further: unify L2, L3 of multiple cores in a multi-core

Memory Hierarchy in Modern Server Machines



Hierarchy: Inclusion versus Exclusion

- **Inclusion**
  - Bring block from memory into L2 then L1
    - A block in the L1 is always in the L2
  - If block evicted from L2, must also evict it from L1
    - Why? more on this when we talk about multicore

- **Exclusion**
  - Bring block from memory into L1 but not L2
    - Move block to L2 on L1 eviction
      - L2 becomes a large victim cache
    - Block is either in L1 or L2 (never both)
  - Good if L2 is small relative to L1
    - Example: AMD's Duron 64KB L1s, 64KB L2

Cache Write Policy

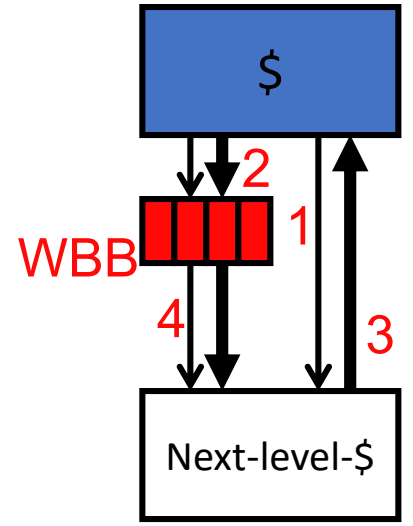
- Write is more complex than read
  - Write and tag comparison can not proceed simultaneously – Can a read operation do this?
  - Only a portion of the line has to be updated

- Write policies (on cache Hit)
  - Write through – write to the cache and to all next-level cache/memory
  - Write back – write only to the cache and track modifications for each block (dirty bit) and update to next-level cache/memory only on replacement.

- Write miss: (store without load, or store on evicted entry)
  - Write allocate – load block to the cache on a write miss
  - No-write allocate – update directly in memory without loading the block to the cache

Write Propagation

- When to propagate new value to (next level) memory?
- **Option #1: Write-through**: immediately
  - On hit, update cache
  - Immediately send the write to the next level
- **Option #2: Write-back**: when block is replaced
  - Requires additional "**dirty**" bit per block
    - Replace **clean** block: **no extra traffic**
    - Replace **dirty** block: **extra "writeback" of block**
  - + **Writeback-buffer (WBB)**:
    - Hide latency of writeback (keep off critical path)
    - Step#1: Send "fill" request to next-level
    - Step#2: While waiting, write dirty block to buffer
    - Step#3: When new blocks arrives, put it into cache
    - Step#4: Write buffer contents to next-level



Write Propagation Comparison

Write-through

- Advantages:
  - Easy to implement; simplifies miss handling
  - Cache is always consistent with main-memory
  - No need for dirty bits in cache.
- Disadvantages:
  - Write operations get slower – every write needs main memory access
  - Requires additional bus bandwidth
- Consider repeated write hits
- Next level must handle small writes (1, 2, 4, 8-bytes)
- Used in GPUs, as they have low write temporal locality

Write-back

- Advantages:
  - Write operations at the speed of cache
  - Coalesced write → uses less bandwidth
- Disadvantages:
  - Cache and main-memory can be inconsistent
  - Need additional bits to track modified cache lines (dirty/modified bits).
  - Harder to implement; complex miss handling
  - Even read operations may pay additional penalty when replacing the modified block
- Reverse pros/cons compared to write-through
- Used in most CPU designs → high temporal locality;

Write Miss Handling

- How is a write miss actually handled?

- **Write-allocate**: fill block from next level, then write it
  - + Decreases read misses (next read to block will hit)
  - – Requires additional bandwidth
  - Commonly used (especially with write-back caches)

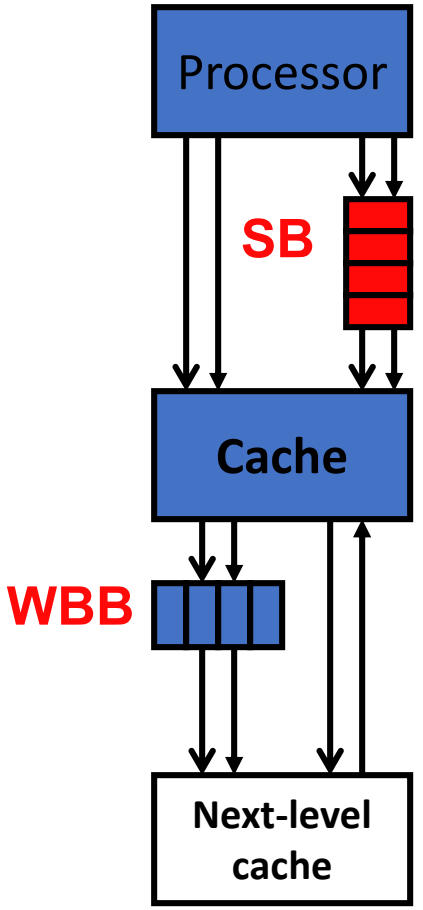
- **Write-non-allocate**: just write to next level, no allocate
  - – Potentially more read misses
  - + Uses less bandwidth
  - Use with write-through

Write Misses and Store Buffers

- Read miss?
  - Load can't go on without the data, it must stall
- Write miss?
  - Technically, no instruction is waiting for data, why stall?

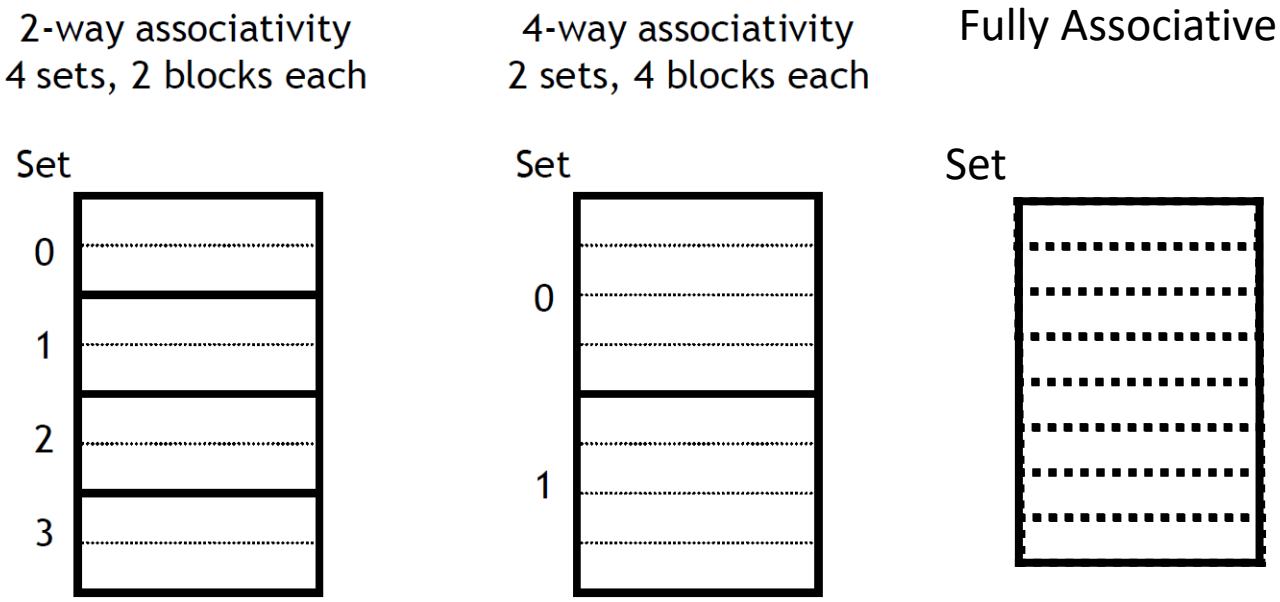
- **Store buffer**: a small buffer
  - Stores put address/value to store buffer, **keep going**
  - Store buffer writes stores to D$ in the background
  - Loads must search store buffer (in addition to D$)
  - \+ Eliminates stalls on write misses (mostly)
  - – Creates some problems for multicore (later)

- Store buffer vs. writeback-buffer
  - Store buffer: "in front" of D$, for hiding store misses
  - Writeback buffer: "behind" D$, for hiding writebacks



Replacement Policies (Cache and Pages)

- On cache/page miss, which block/page in set to replace (kick out)?
- Some replacement policies
  - **Random**
    - Randomly chose one of the blocks
  - **FIFO (first-in first-out)**
    - Iterate 1 block at a time.
  - **LRU (least recently used)**
    - Fits with temporal locality, LRU
    - = least likely to be used in future
  - **LFU (least frequently used)**
    - Instead of recency, rely on frequency of access over specific interval
  - **NMRU (not most recently used)**
    - An easier to implement approximation of LRU
    - Is LRU for 2-way set-associative caches
  - **Belady's (Optimum)**: replace block that will be used furthest in future
    - Unachievable optimum



IMPLEMENTING LRU

- Idea: Evict the least recently accessed block
- Problem: Need to keep track of access ordering of blocks

- Question: 2-way set associative cache:
  - What do you need to implement LRU perfectly?

- Question: 4-way or higher-way set associative cache:
  - What do you need to implement LRU perfectly?
  - What is the logic needed to determine the LRU victim?
  - How many bits needed to encode the LRU order of a block?

APPROXIMATIONS OF LRU

- Most modern processors do not implement “true LRU” (also called “perfect LRU”) in highly-associative caches.

- Why?
  - True LRU is complex
  - LRU is an approximation to predict locality anyway (i.e., not the best possible cache management policy)

- Examples:
  - Not MRU (not most recently used)
  - Hierarchical LRU: divide the 4-way set into 2-way “groups”, track the MRU group and the MRU way in each group
  - Victim-NextVictim Replacement: Only keep track of the victim and the next victim

REMEMBER: CACHE VERSUS PAGE REPLACEMENT

- Physical memory (DRAM) is a cache for disk
  - Usually managed by system software via the virtual memory subsystem

- Page replacement is similar to cache replacement
- Page table is the "tag store" for physical memory data store

- What is the difference?
  - Role of hardware versus software
  - Required speed of access to cache vs. physical memory
  - Number of blocks in a cache vs. physical memory
  - "Tolerable" amount of time to find a replacement candidate

  (disk versus memory access latency)

Scott Meyers:

CPU Caches and Why you Care?

Intel: Architecture All Access

Modern CPU Architecture