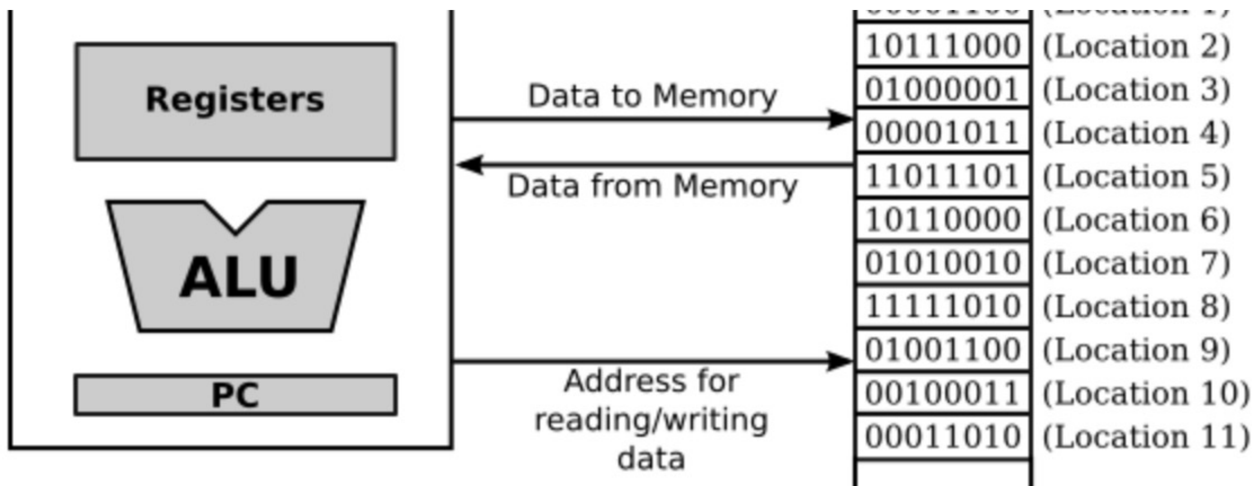
Chapter 5: Large and Fast: Exploiting Memory Hierarchy

ITSC 3181 Introduction to Computer Architecture
Spring 2022
https://passlab.github.io/ITSC3181/

Department of Computer Science
Yonghong Yan
yyan7@uncc.edu
https://passlab.github.io/yanyh/

loiting Memory



- 5.3 The Basics of Caches

- **Lecture**
  - 5.
  - 5.
  - 5.

FLOPs: floating point operations, e.g. add/sub/mul/div per second

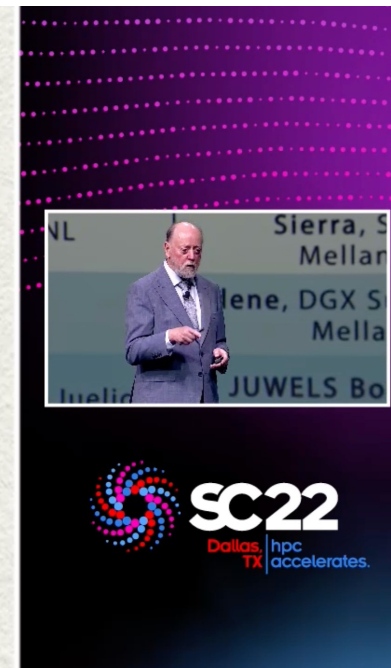
20 - 100 flops per word transferred

Recently, 200 flops per word transferred

- **Lectu**
  - 5.6 Virtual Memory

HPCG Top 10, November 2022

Rank	Site	Computer	Country	Cores	Rmax [Pflop/s]	Top 500 Rank	HPCG [Pflop/s]	% of Peak
1	RIKEN Center for Computational Science	**Fugaku**, Fujitsu A64FX 48C 2.2 GHz, Tofu D, Fujitsu	Japan	7,630,848	422	2	16.0	3.0%
2	DOE / SC / ORNL	**Frontier**, HPE Cray Ex235a, AMD 3rd EPYC 64C 2 GHz, AMD Instinct MI250X, Slingshot 10	USA	8,730,112	1,102	1	14.1	0.8%
3	EuroHPC / CSC	**LUMI**, HPE Cray EX235a, AMD Zen 3 (Milan) 64C 2GHz, AMD MI250X, Slingshot-11	Finland	2,174,976	304	3	3.41	0.8%
4	DOE / SC / ORNL	**Summit**, AC922, IBM POWER9 22C 3.7 GHz, Dual-rail Mellanox FDR, NVIDIA Volta V100, IBM	USA	2,414,592	149	5	2.93	[illegible]5%
5	EuroHPC / CINECA	Leonardo, BullSequana XH2000, Xeon Platinum 8358 32C 2.6 GHz, NVIDIA A100 SXM4 40 GB, Quad-rail NVIDIA HDR100 InfiniBand	Italy	1,463,616	175	4	2.57	1.0%
6	DOE / SC / LBNL	**Perlmutter**, HPE Cray EX235n, AMD EPYC 7763 64C 2.45 GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10	USA	761,856	70.9	8	1.91	2.0%
7	DOE / NNSA / LLNL	**Sierra**, S922LC, IBM POWER9 20C 3.1 GHz, Mellanox EDR, NVIDIA Volta V100, IBM	USA	1,572,480	94.6	6	1.80	1.4%
8	NVIDIA	**Selene**, DGX SuperPOD, AMD EPYC 7742 64C 2.25 GHz, Mellanox HDR, NVIDIA Ampere A100	USA	555,520	63.5	9	1.62	2.0%
9	Forschungszentrum Juelich (FZJ)	**JUWELS Booster Module**, Bull Sequana XH2000, AMD EPYC 7402 24C 2.8 GHz, Mellanox HDR InfiniBand, NVIDIA Ampere A100, Atos	Germany	449,280	44.1	12	1.28	1.8%
10	Saudi Aramco	**Dammam-7**, Cray CS-Storm, Xeon Gold 6252 20C 2.5 GHz, InfiniBand HDR 100, NVIDIA Volta V100, HPE	Saudi Arabia	672,520	22.4	20	0.88	1.6%

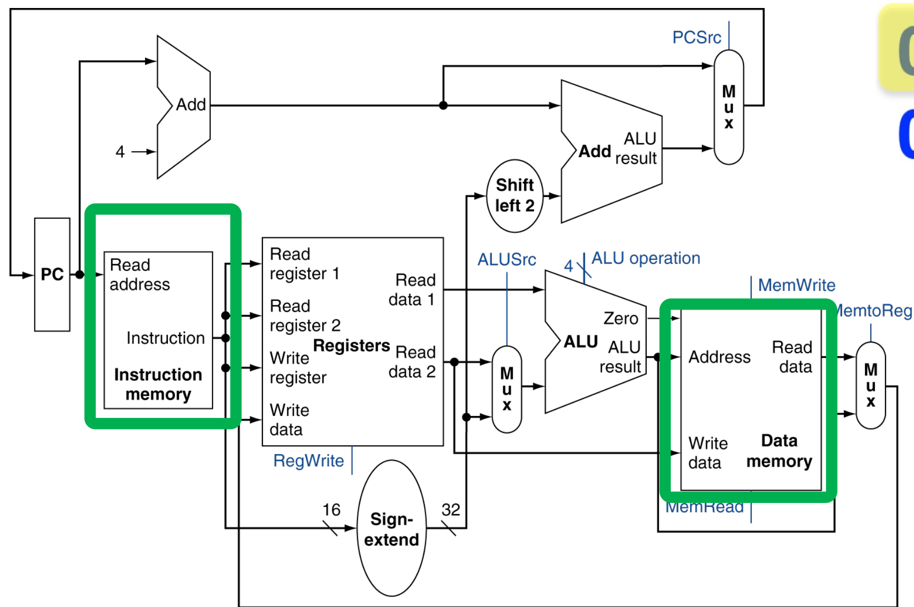


The Big Picture: Where are We Now?

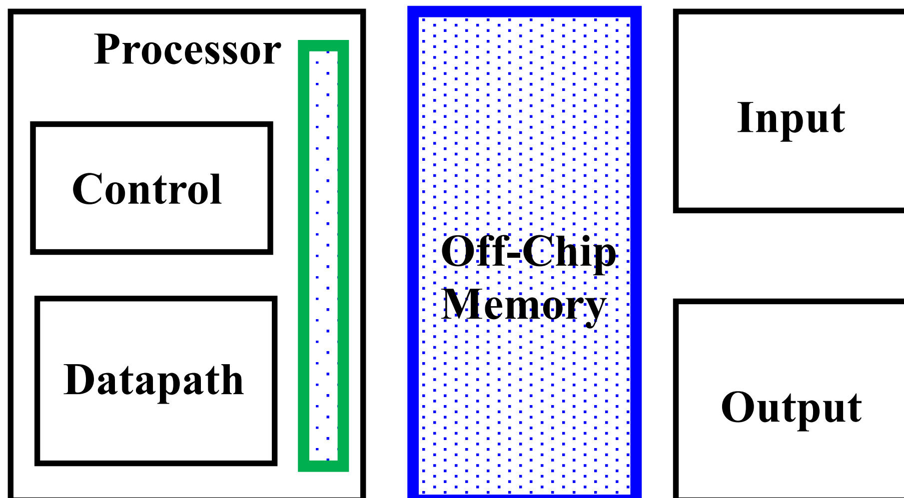
0x0FFE1230: add x1, x2, x3

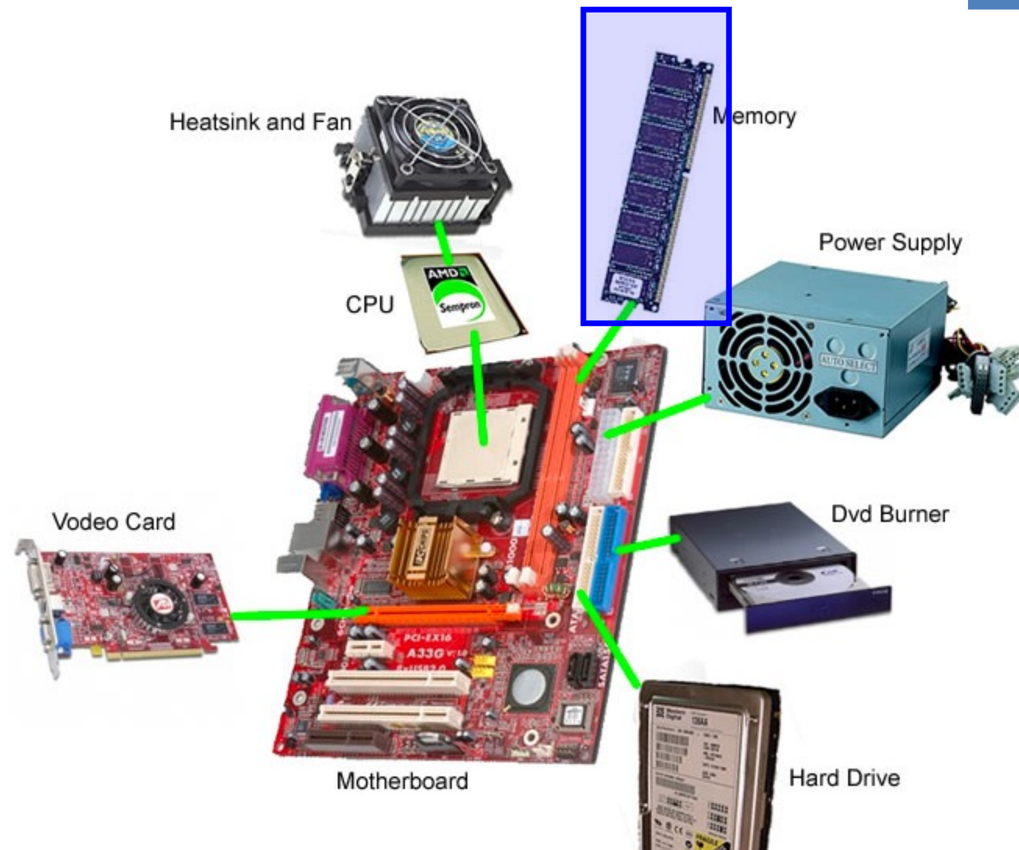
0x0FFE1234: lw|sw x1, 32(x2)

0x0FFE1238: beq x1, x2, offset

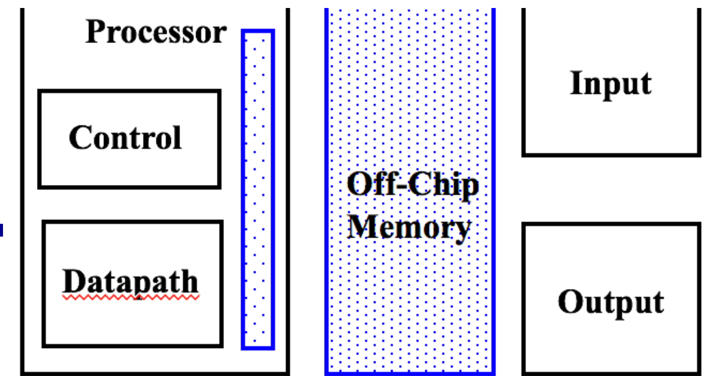


Instruction Fetch (Load)





Overview



- Programmers want memory system
  - **Speed**: To supply data on time for computation
  - **Capacity:** To be large enough to hold everything needed

- However, fast memory technology is more expensive per bit than slower memory

- **Solution: organize memory system into a hierarchy**
  - Entire addressable memory space available in largest, slowest memory ➔capacity, store "always" in large memory
  - Incrementally smaller and faster memories, each containing a subset of the memory below it, proceed in steps up toward the processor ➔speed, access "always" from fast memory

Memory Hierarchy

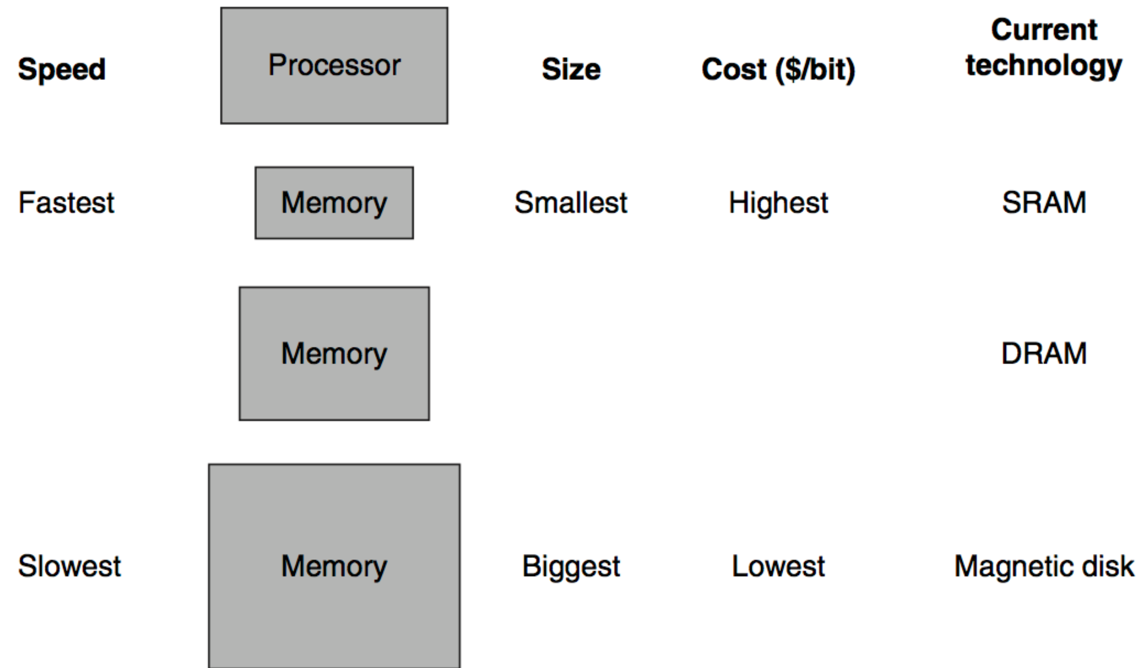


FIGURE 5.1 The basic structure of a memory hierarchy. By implementing the memory system as a hierarchy, the user has the illusion of a memory that is as large as the largest level of the hierarchy, but can be accessed as if it were all built from the fastest memory. Flash memory has replaced disks in many personal mobile devices, and may lead to a new level in the storage hierarchy for desktop and server computers; see Section 5.2.

- Memories of different technologies are organized in a hierarchy
  - The closer to the CPU, the faster and smaller of the memory
  - The farther from the CPU, the slower and larger of the memory
  - Data movement from far level to close level are via blocks
    - 64 bytes from DRAM to cache, and 4KB from disk to DRAM (paging)

Why Memory Hierarchy Works?
The Principle of Locality

- Programs access a small proportion of their address space at any time
- **Temporal locality - Time**
  - **Items accessed recently are likely to be accessed again soon**
  - **e.g., instructions in a loop, induction variables**
- **Spatial locality – Space**
  - **Items near those accessed recently are likely to be accessed soon**
  - **E.g., sequential instruction access, array data**

- **Data**
  - **Reference array elements in succession: Spatial Locality**
  - **Reference sum each iteration: Temporal Locality**
- **Instructions**
  - **Reference instructions in sequence: Spatial Locality**
  - **Cycle through loop repeatedly: Temporal Locality**

```
sum = 0;
for(i=0; i<n; i++)
  sum += a[i];
return sum;
```

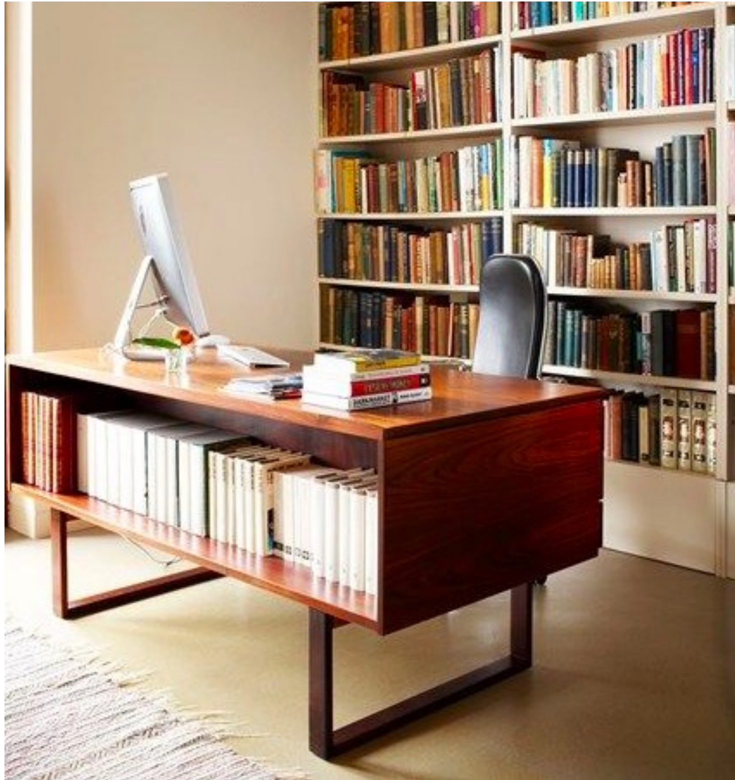
The Principle of Locality: Explained using Borrowing and Reading Books to Study a Topic

- Spatial locality: one reads pages/books and those nearby pages or related books at the same time
- Temporal locality: one reads the same pages or books multiple times

- If we need lots of books, we need to borrow from the library, put on our bookshelf and on our study desk.

Books are Data, Study is the Program, Desk/Bookshelf/Library are Memory Hierarchy

- **You are the CPU**



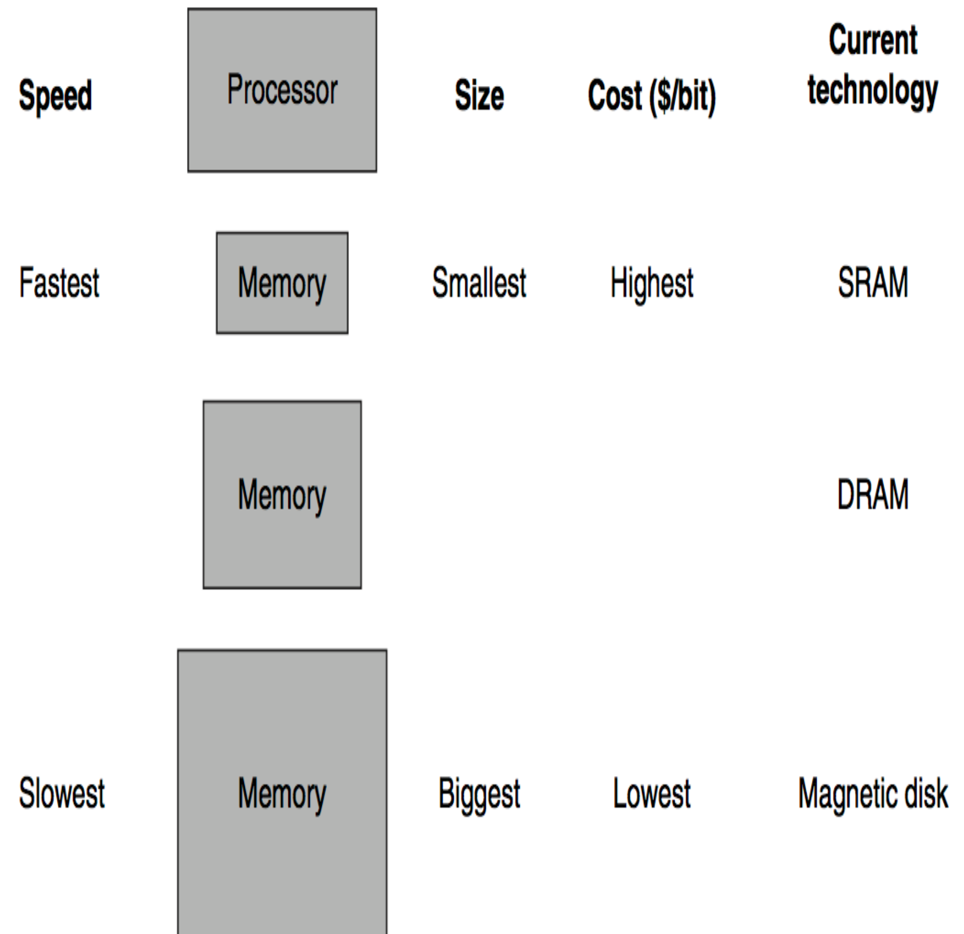
Memory Hierarchy: Explained with Desk, Bookshelf and Library

You, as a processor study and read books of a topic (a program)

Books that are used often are on your desk; fast access, small # of books

Your bookshelf can hold more books, less often used than those on the desk. you need to stand up to grab a book and you often grab more than one books a time

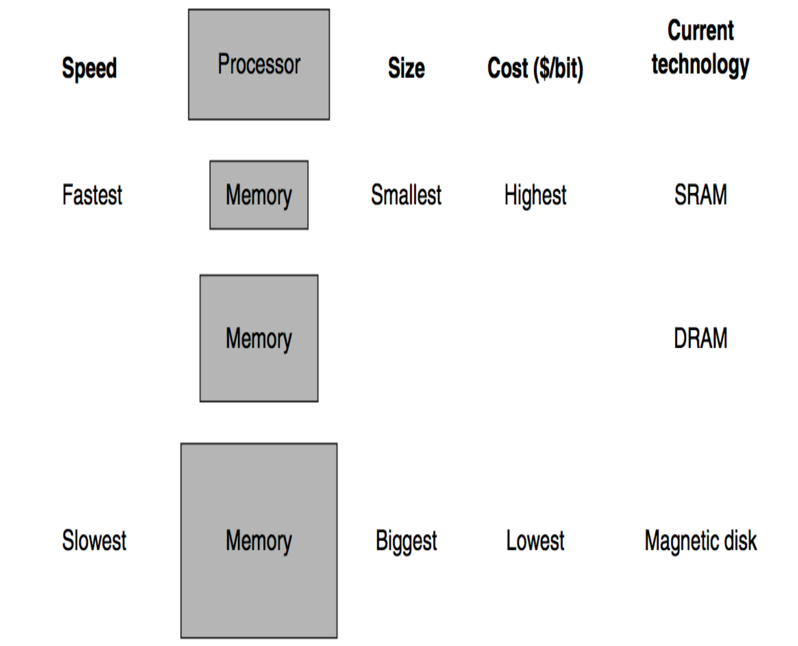
Library has more books, but you do not go often since it is far. Each time you go, you borrow a bag of books.



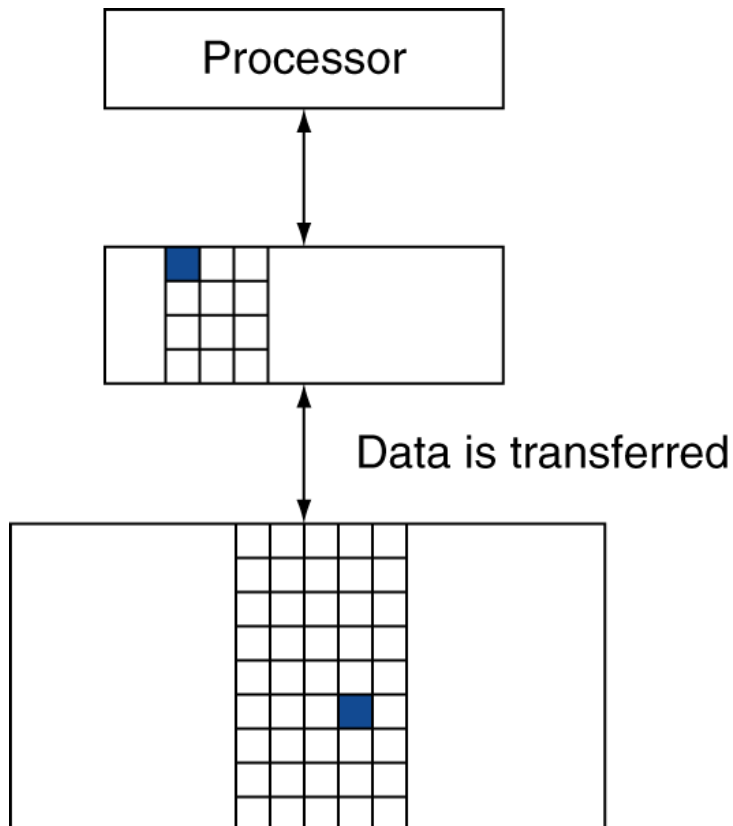
- Except, compared with memory hierarchy
  - Upper level do not have copies of the books

Taking Advantage of Locality

- Memory hierarchy
- Store everything on disk (library)
- Copy recently accessed (and nearby) items from disk to smaller DRAM memory (bookshelf)
  - Main memory
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory (desk)
  - Cache memory attached to CPU



Memory Hierarchy Levels



- **Block (aka line): unit of copying**
  - **May be multiple words, just like we move multiple books a time**
- If accessed data is present in upper level
  - Hit: access satisfied by upper level
    - Hit ratio: hits/accesses
- If accessed data is absent
  - Miss: block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses
      = 1 – hit ratio
  - Then accessed data supplied from upper level

Memory Hierarchy

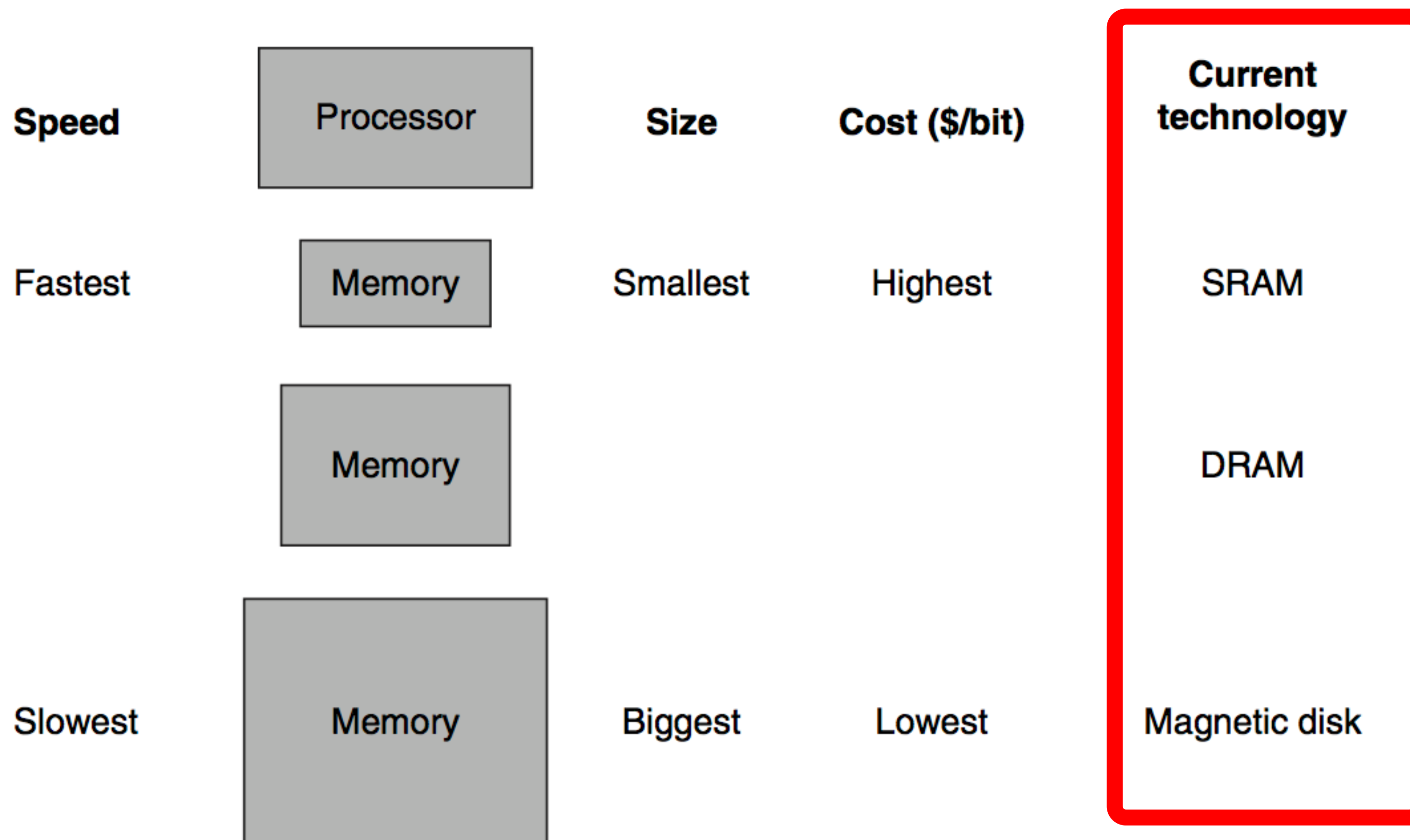


FIGURE 5.1 The basic structure of a memory hierarchy. By implementing the memory system as a hierarchy, the user has the illusion of a memory that is as large as the largest level of the hierarchy, but can be accessed as if it were all built from the fastest memory. Flash memory has replaced disks in many personal mobile devices, and may lead to a new level in the storage hierarchy for desktop and server computers; see Section 5.2.

Memory Technology

- CPU speedup, e.g. 2GHz → 0.5ns per cycle
  - Pipelined CPU CPI = 1

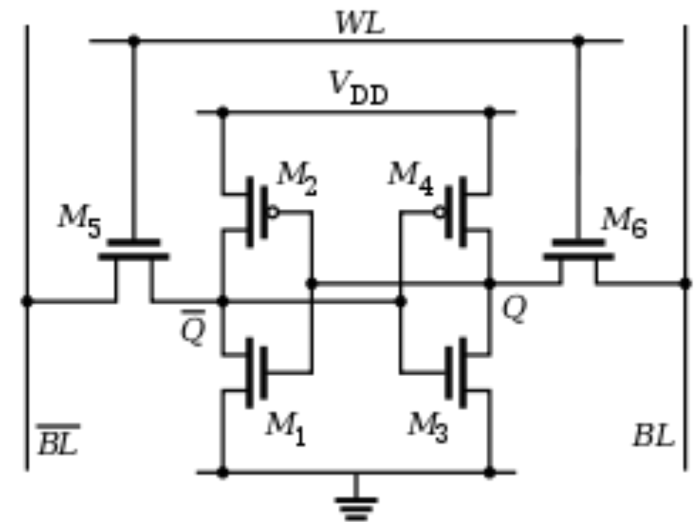
Speed		Size	Cost ($/bit)
	Processor		
Fastest	Memory	Smallest	Highest
	Memory		
Slowest	Memory	Biggest	Lowest

Current technology	
SRAM	**Static RAM (SRAM): Cache** **0.5ns – 2.5ns, $2000 – $5000 per GB**
DRAM	**Dynamic RAM (DRAM)** **50ns – 70ns, $20 – $75 per GB**
Magnetic disk	**Magnetic disk** **5ms – 20ms, $0.20 – $2 per GB**

- Ideal memory
  - Access time of SRAM
  - Capacity and cost/GB of disk

Random-Access Memory (RAM)

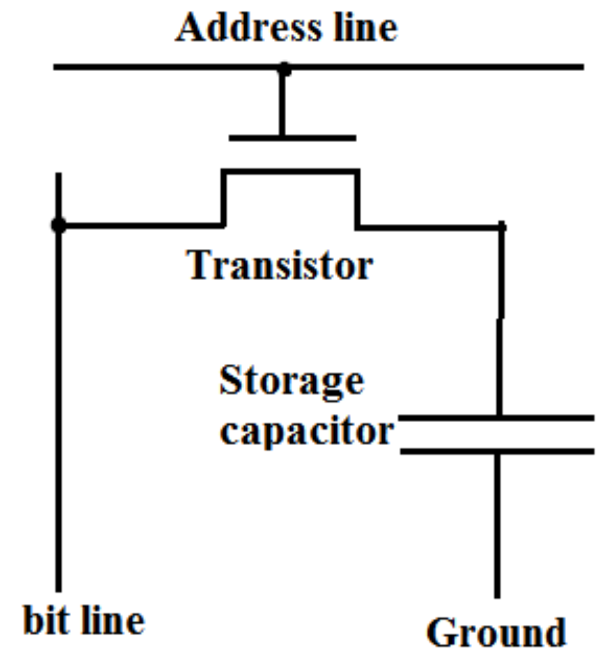
- Key features
  - RAM is packaged as a chip.
  - Basic storage unit is a cell (one bit per cell).
  - Multiple RAM chips form a memory.



- **Static RAM (SRAM)**
  - Each cell stores bit with a six-transistor circuit.
  - Retains value indefinitely, as long as it is kept powered.
  - Relatively insensitive to disturbances such as electrical noise.
  - Faster and more expensive than DRAM.

DRAM Technology

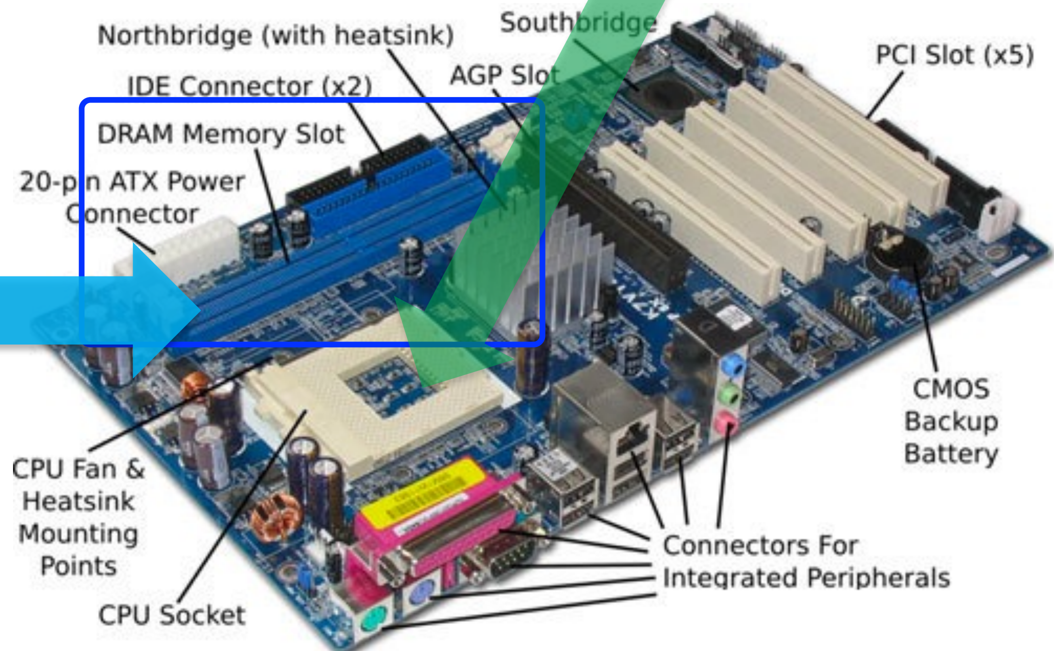
- Data stored as a charge in a capacitor
  - Single transistor used to access the charge
  - Dynamic: need to be "refreshed" regularly, every 10-100 ms.
  - Sensitive to disturbances.
  - Slower and cheaper than SRAM.

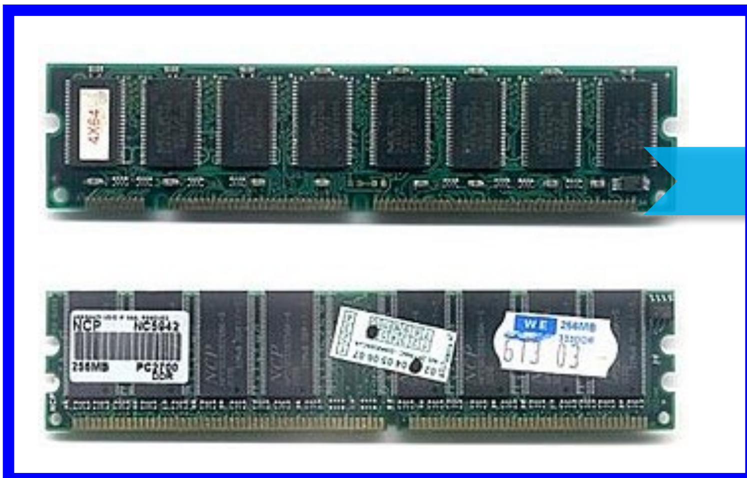


DRAM in Real is Main Memory, off-chip

- In reality,
  - Several DRAM chips are bundled into Memory Modules
- SIMMS - Single Inline Memory Module
- DIMMS - Dual Inline Memory Module
- DDR- Dual data Read
  - Reads twice every clock cycle
- Quad Pump: Simultaneous R/ W

CPU is The chip.

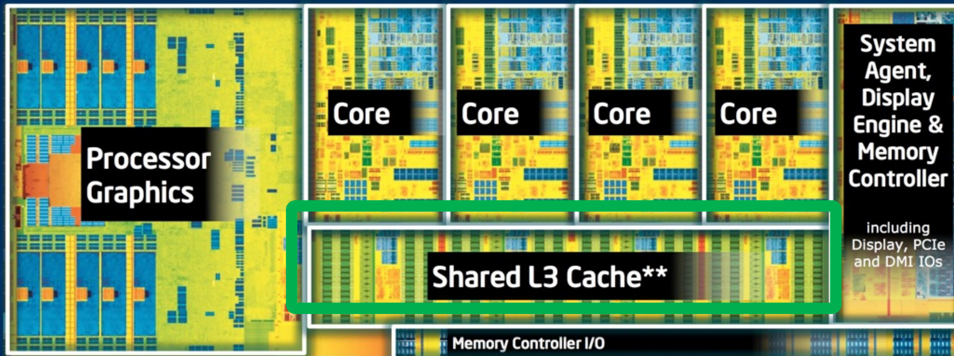


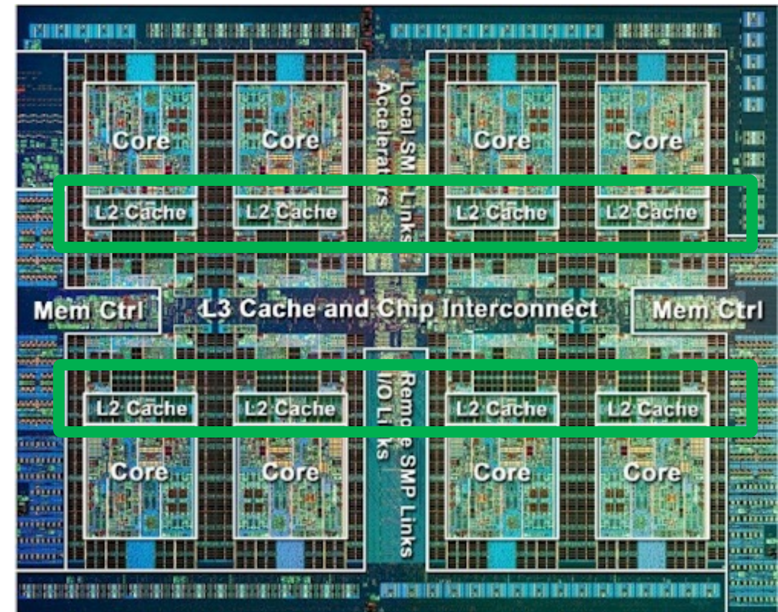


SRAM in Real is Cache in the CPU, on-chip

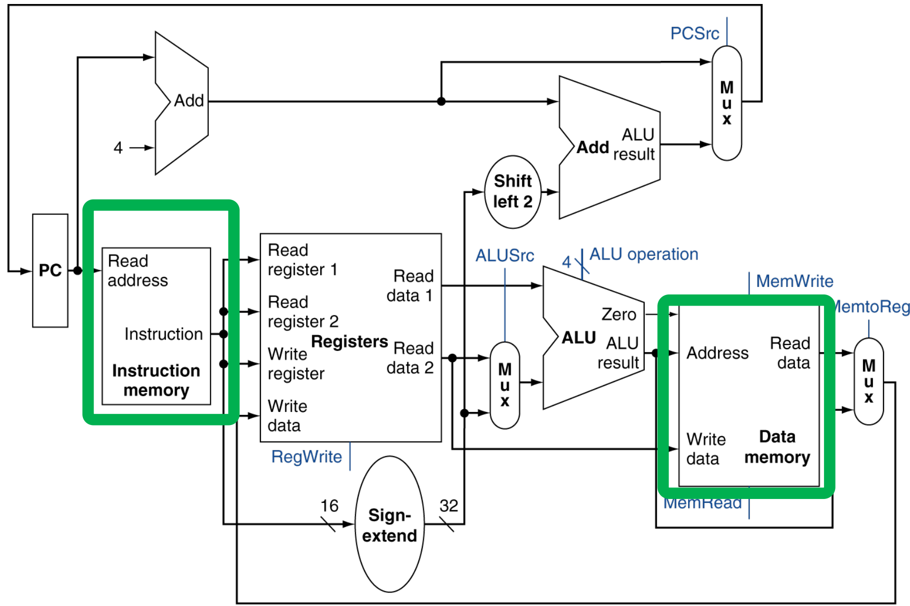
CPU is The chip.

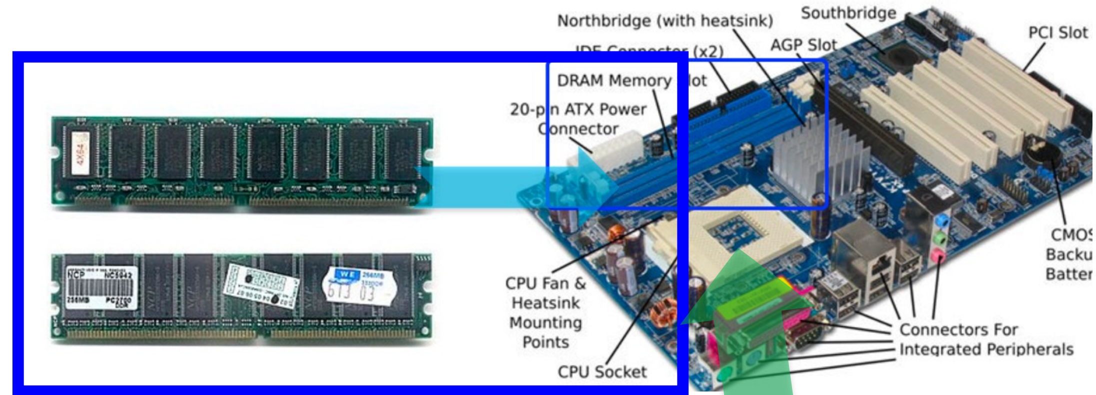
Cache is Cash



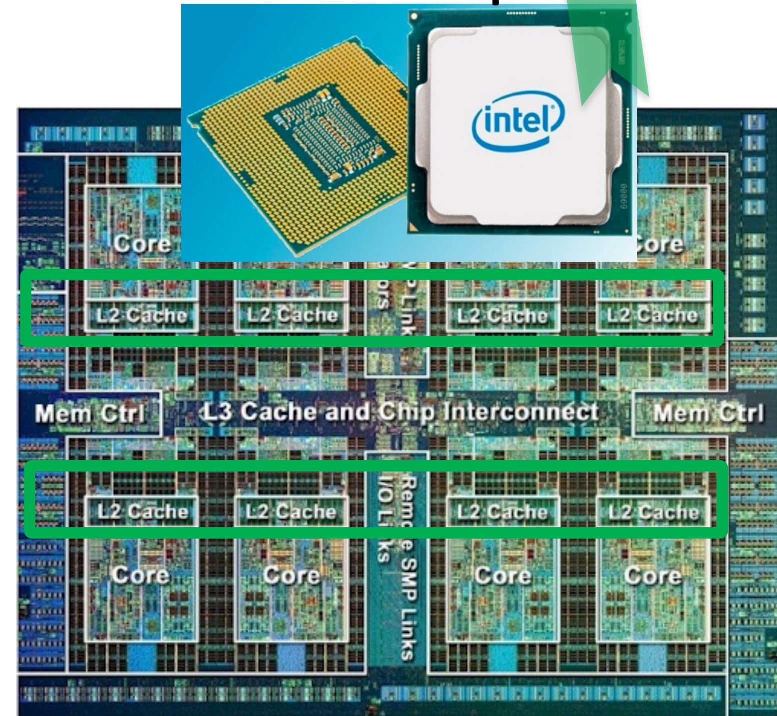


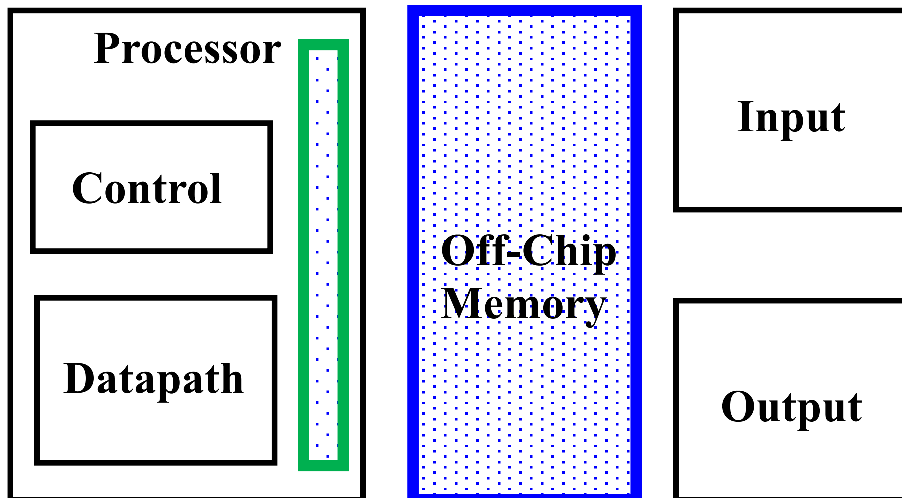
Green Boxes: Cache, on-chip, SRAM, fast, small, expensive
Blue Boxes: Main memory, off-chip, DRAM, slower, large, not expensive





CPU is The chip.





Memory Technology

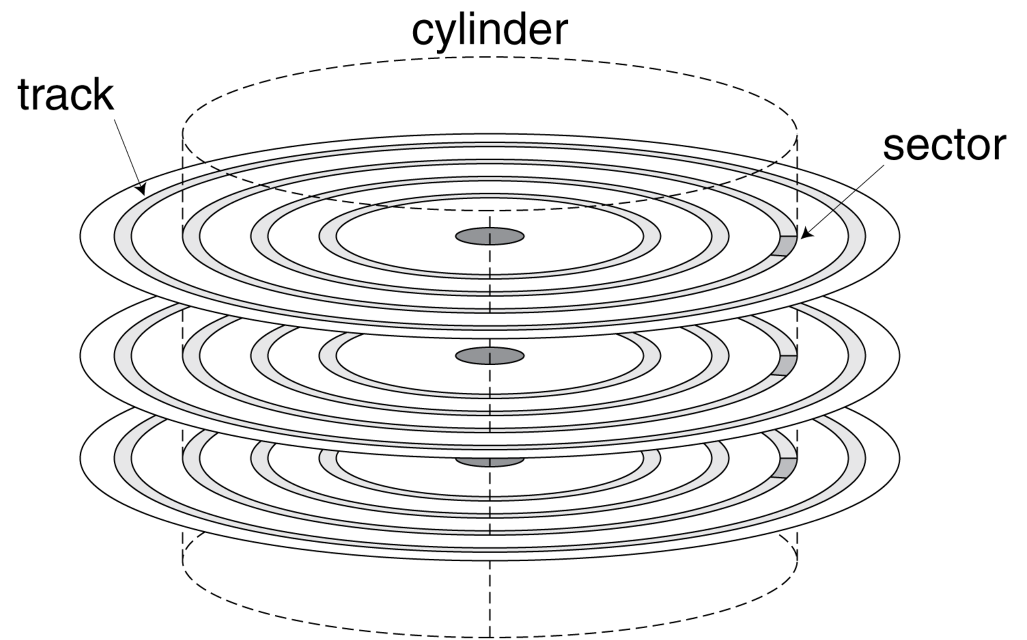
- CPU speedup, e.g. 2GHz →0.5ns per cycle
  - Pipelined CPU CPI = 1

Speed	Processor	Size	Cost ($/bit)	Current technology	
Fastest	Memory	Smallest	Highest	SRAM	**Static RAM (SRAM)** **0.5ns – 2.5ns, $2000 – $5000 per GB**
	Memory			DRAM	**Dynamic RAM (DRAM)** **50ns – 70ns, $20 – $75 per GB**
Slowest	Memory	Biggest	Lowest	Magnetic disk	**Magnetic disk** **5ms – 20ms, $0.20 – $2 per GB**

- Ideal memory
  - Access time of SRAM
  - Capacity and cost/GB of disk

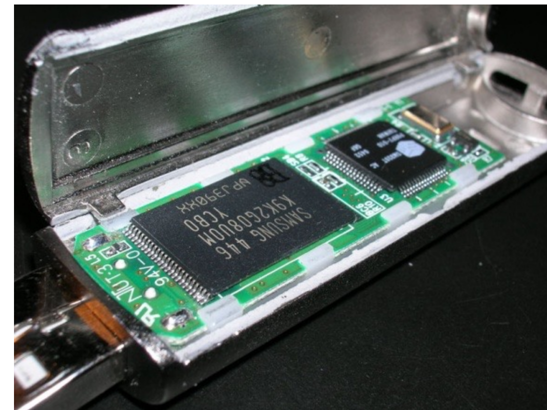
Disk Storage

- Nonvolatile, rotating magnetic storage



Flash Storage

- Nonvolatile semiconductor storage
  - 100× – 1000× faster than disk
  - Smaller, lower power, more robust
  - But more $/GB (between disk and DRAM)



Flash Types

- NOR flash: bit cell like a NOR gate
  - Random read/write access
  - Used for instruction memory in embedded systems
- NAND flash: bit cell like a NAND gate
  - Denser (bits/area), but block-at-a-time access
  - Cheaper per GB
  - Used for USB keys, media storage, …
- Flash bits wears out after 1000’s of accesses
  - Not suitable for direct RAM or disk replacement
  - Wear leveling: remap data to less used blocks

History: Further Back

Ideally one would desire an indefinitely large memory capacity such that any particular ... word would be immediately available. ... We are ... forced to recognize the possibility of constructing a ***hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.***

A. W. Burks, H. H. Goldstine, and J. von Neumann
Preliminary Discussion of the Logical Design of an Electronic Computing Instrument, **1946**

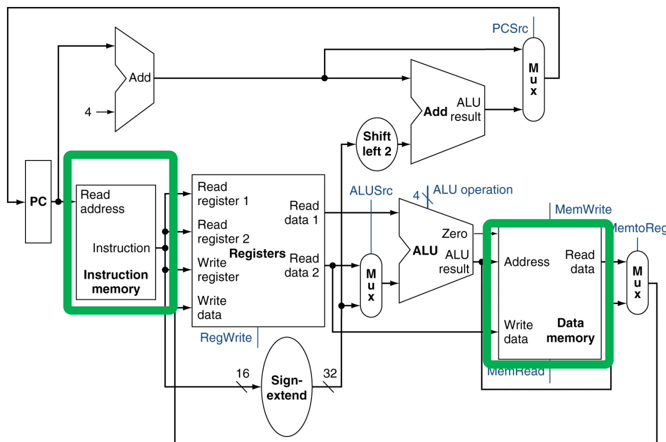
Chapter 5: Large and Fast: Exploiting Memory Hierarchy

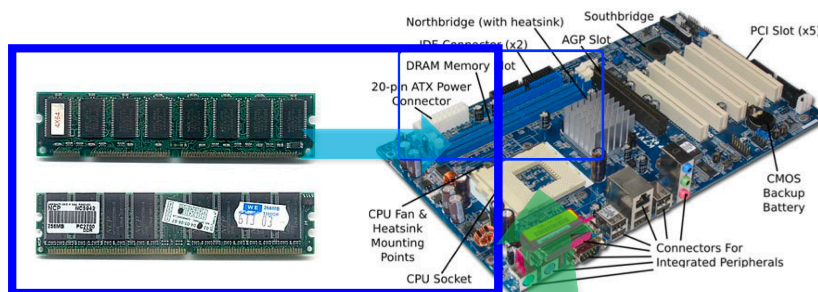
- Lecture
  - 5.1 Introduction
  - 5.2 Memory Technologies

- Lecture
  - 5.3 The Basics of Caches
- Lecture
  - 5.4 Measuring and Improving Cache Performance
  - ~~5.5 Dependable Memory Hierarchy~~
  - ~~5.6 Virtual Machines~~
- Lecture
  - 5.6 Virtual Memory
  - ~~5.8 A Common Framework for Memory Hierarchy~~
- Lecture 26
  - ~~5.9 Using a Finite-State Machine to Control a Simple Cache~~
  - ~~5.10 Parallelism and Memory Hierarchies: Cache Coherence~~
  - ~~5.11 Parallelism and Memory Hierarchy: Redundant Arrays of Inexpensive Disks~~
  - ~~5.12 Advanced Material: Implementing Cache Controllers~~
  - 5.13 Real Stuff: The ARM Cortex-A53 and Intel Core i7 Memory Hierarchies
  - ~~5.14 Going Faster: Cache Blocking and Matrix Multiply~~
  - ~~5.15 Fallacies and Pitfalls~~
  - 5.16 Concluding Remarks

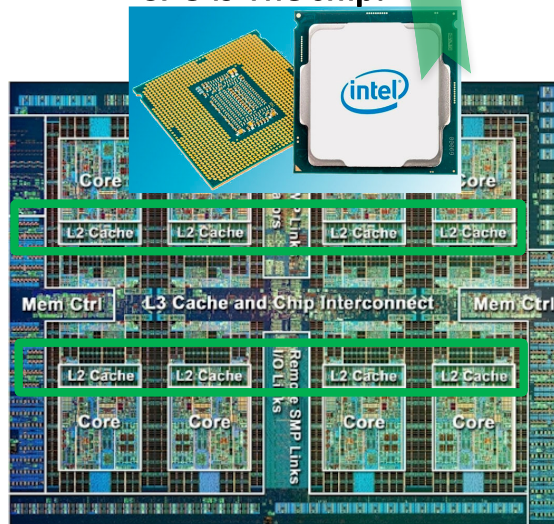
Cache Memory

- Cache memory
  - The level of the memory hierarchy closest to the CPU, our desk
  - **Green Boxes: Cache, on-chip, SRAM, fast, small, expensive**

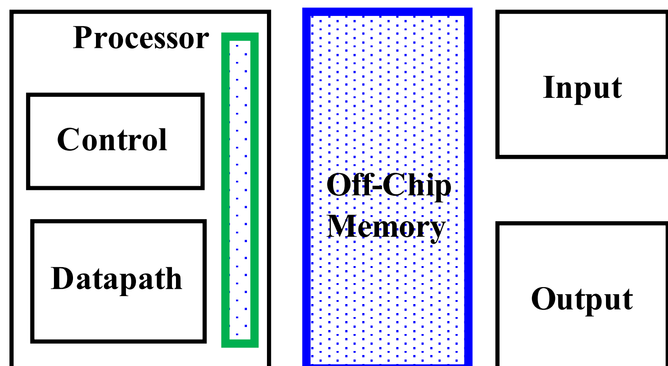




CPU is The chip.



Blue Boxes Covered in 5.7 Lecture: Main memory, off-chip, DRAM, slower, large not expensive



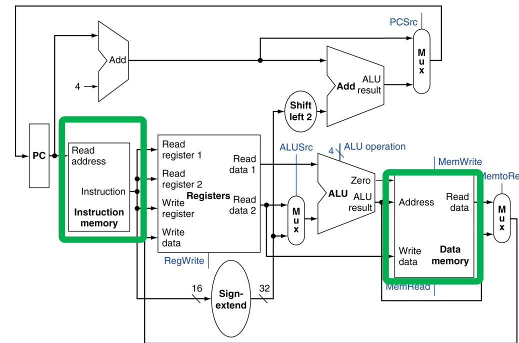
Cache Memory and Main Memory

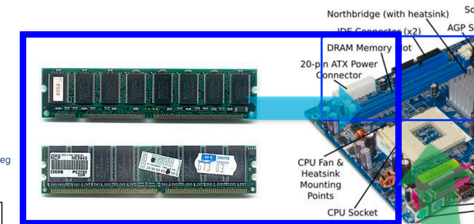
- Memory access:
  - Instruction Fetch
  - Data memory access: LW|SW

0x0FFE1230: add x1, x2, x3

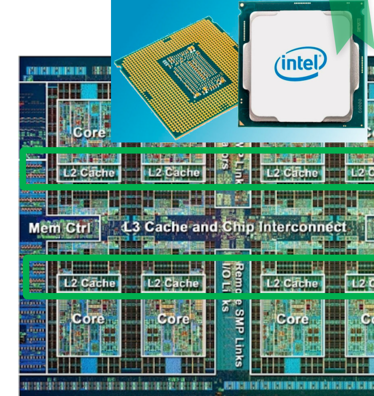
0x0FFE1234: lw|sw x1, 32(x2)

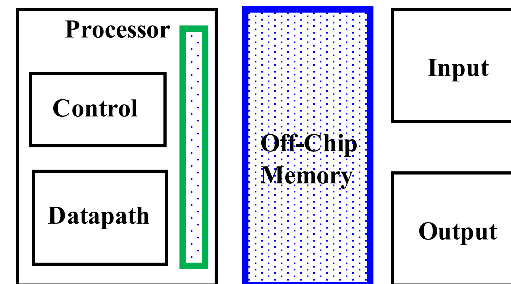
0x0FFE1238: beq x1, x2, offset





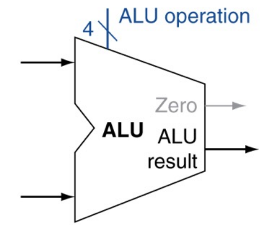
CPU is The chip.





- **Instruction Fetch and LW|SW use off-chip main memory address (DRAM) to read or write a word**
  - **Instruction/Data is fetched from off-chip memory to on-chip cache first, and then to register**
  - **CPU calculates (arithmetic and logic operation) using data in register only**

Load/Store Instructions

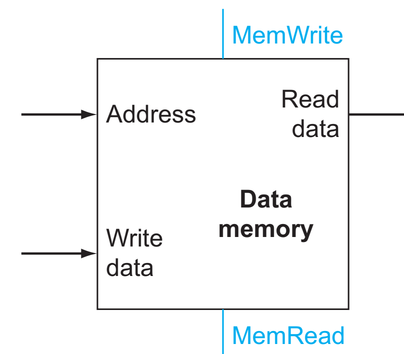


- Read register operands
  - x2, and x1(for sw only)
- **Calculate main memory address using 12-bit signed offset**
  - **32 + x2**
  - ALU operation is +
- **Load: Read memory and update register**
  - **x1 ← MEM(32+x2)**
  - MemRead signal is on
- **Store: Write register value to memory**
  - **x1 → MEM(32+x2)**
  - MemWrite is on

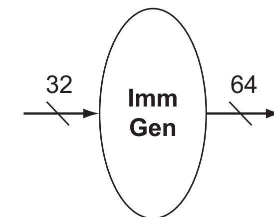
0x0FFE1230: add x1, x2, x3

0x0FFE1234: lw|sw x1, 32(x2)

0x0FFE1238: beq x1, x2, offset



a. Data memory unit

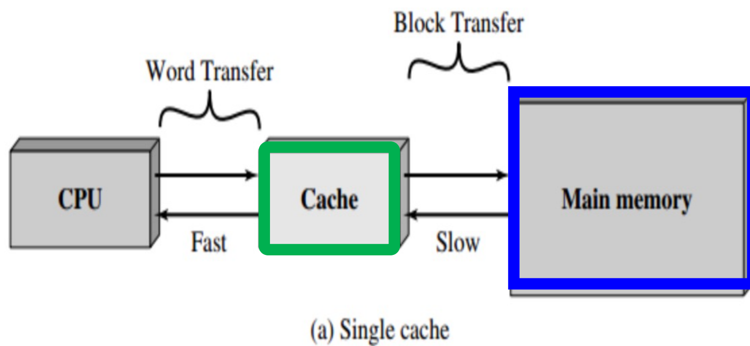


b. Immediate generation unit

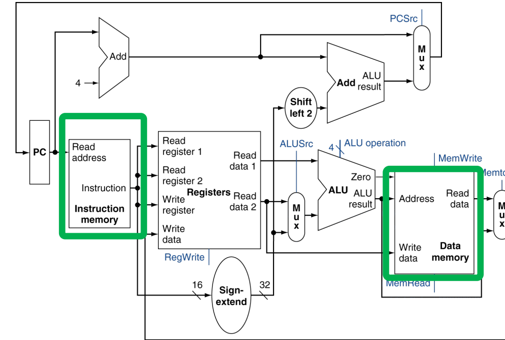
Memory addresses for IF and for the calculated address of Load/Store are off-chip memory addresses

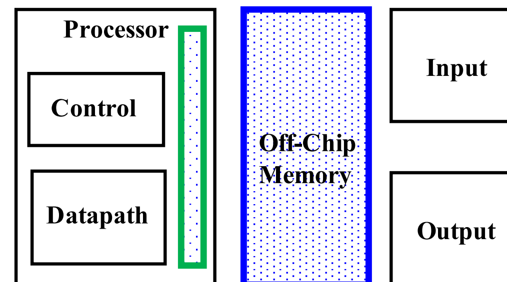
Cache is Transparent to Programmers

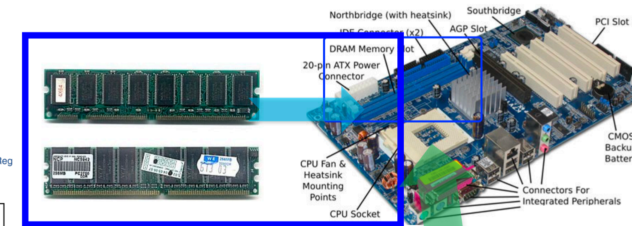
- **A program does not need to know the existence of cache**
- Instruction Fetch and LW/SW: read/write data from memory
  - **Main memory address is used for IF/LW/SW, not cache mem address**

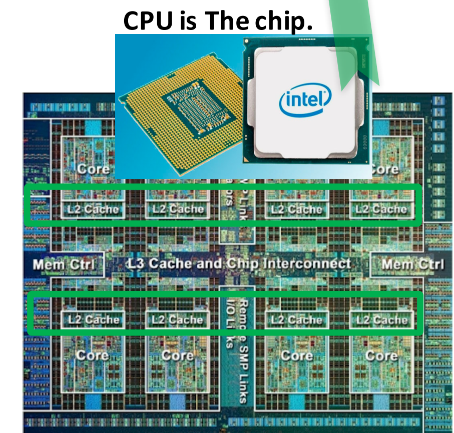


(a) Single cache



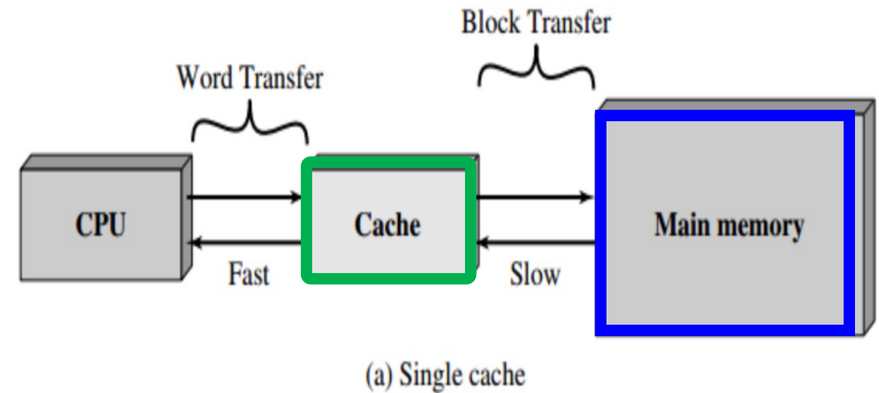






The BIG Question: How cache is used for IF/LW/SW, i.e. how hardware knows the cache memory address to access a word addressed by main memory address in IF/LW/SW instructions

Cache Memory



- Memory access: IF/LW/SW
  - If data is in cache, IF/LW/SW from cache
  - If data is not in the cache, IF/LW/SW from main memory
- Given accesses $X_1, \ldots, X_{n-1}, X_n$

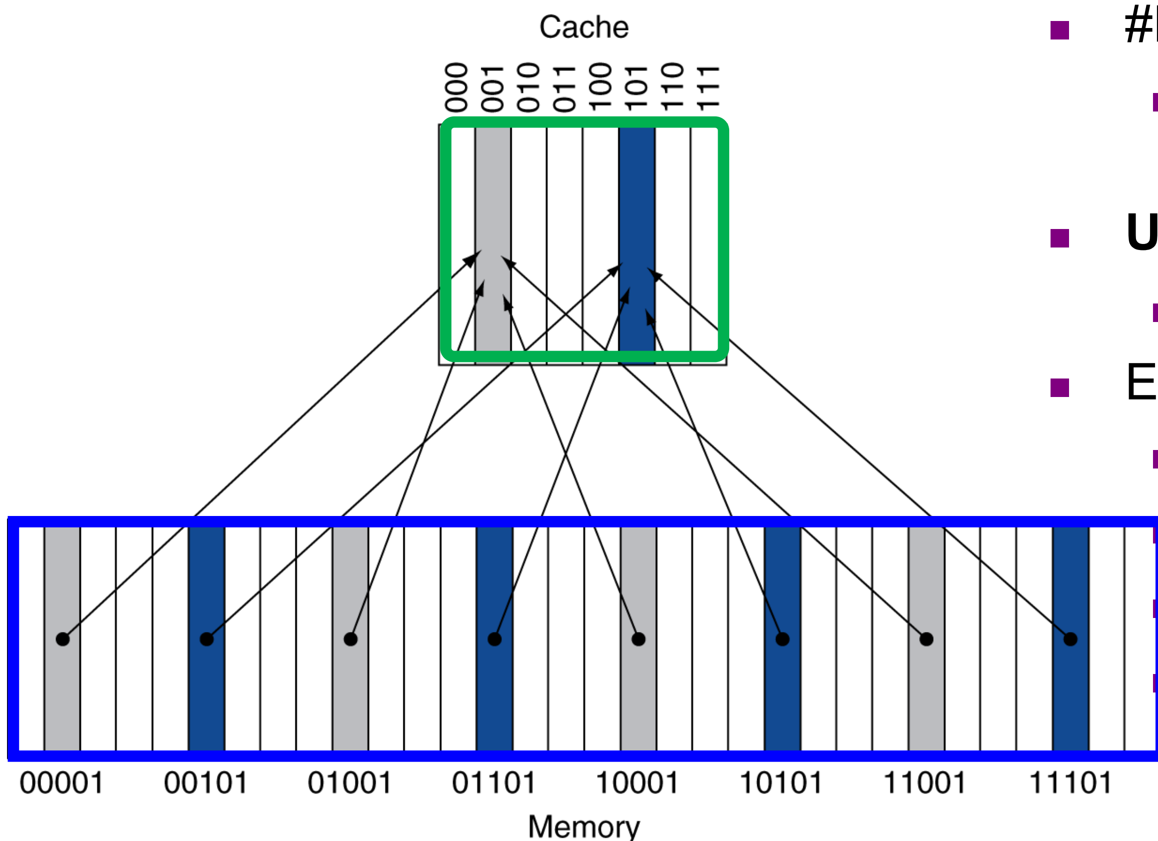
a. Before the reference to X_n	b. After the reference to X_n
X_4	X_4
X_1	X_1
X_{n-2}	X_{n-2}
X_{n-1}	X_{n-1}
X_2	X_2
	X_n
X_3	X_3

0x0FFE1230: add x1, x2, x3
0x0FFE1234: lw|sw x1, 32(x2)
0x0FFE1238: beq x1, x2, offset

- How do we know if the data is present?
- Where do we look?

Cache Organization: Direct Mapped Cache

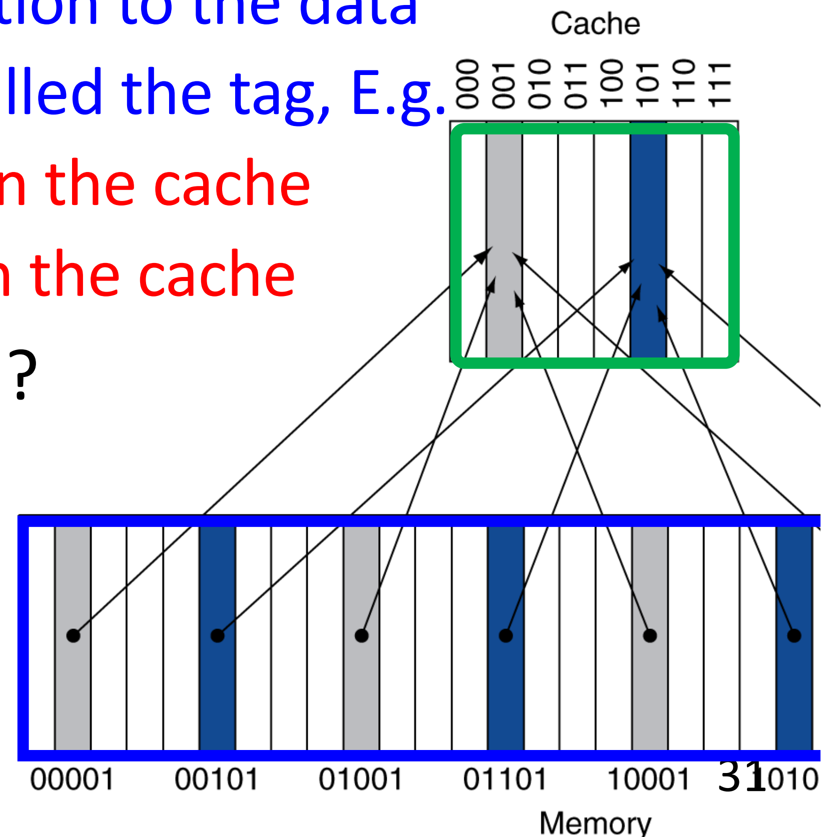
- Location determined by address
  - Main memory addresses 00001, 01001 in the example
- Direct mapped: only one choice
  - (Memory address) modulo (#Blocks in cache)



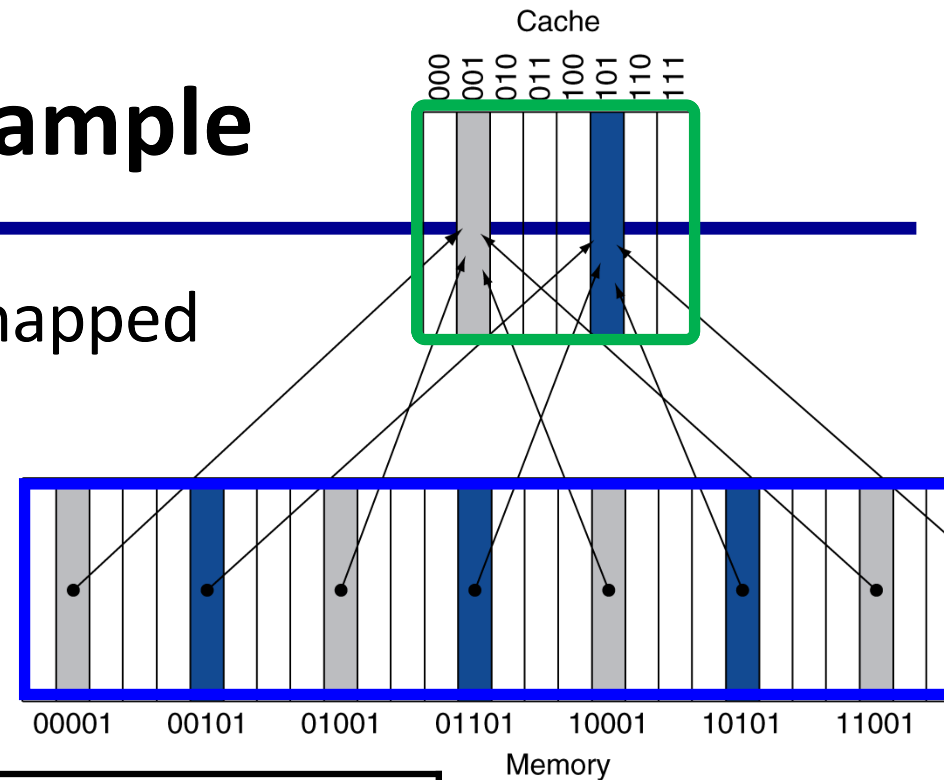
- #Blocks is a power of 2
  - 8 in this example, need 3 bit to address a block
- **Use low-order address bits**
  - **The last three bits**
- E.g. just check the last three bits
  - 00001 is in block 00001 % 8 = 001
  - 01001 is in block 01001 % 8 = 001
  - **10101 is in block 10101 % 8 = 101**
  - **11101 is in block 11101 % 8 = 101**

Tags and Valid Bits

- Much less cache blocks than main memory since cache is small
  - Multiple memory blocks end up in the same cache block
- How do we know which/whether a particular main memory block is stored in a cache location?
  - Store block/memory address, in addition to the data
  - **But only need the high-order bits,** called the tag, E.g.
    - For **00**001 block, **00 (tag)** is stored in the cache
    - For **01**001 block, **01(tag)** is stored in the cache
- What if there is no data in a location?
  - Valid bit: 1 = present, 0 = not present
  - Initially 0



Cache Example



- 8-blocks, 1 word/block, direct mapped
- Initial state
  - Only green box are the cache
  - Index is the block address, not part of the cache

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

Cache Example

- A sequence of main memory access
  - Load and store instruction
- Given a word address, we can easily calculate the block address and tag bits

Decimal address of reference	Binary address of reference	Assigned cache block (where found or placed)
22	10110_{two}	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	$(11010_{two} \bmod 8) = 010_{two}$
22	10110_{two}	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	$(11010_{two} \bmod 8) = 010_{two}$
16	10000_{two}	$(10000_{two} \bmod 8) = 000_{two}$
3	00011_{two}	$(00011_{two} \bmod 8) = 011_{two}$
16	10000_{two}	$(10000_{two} \bmod 8) = 000_{two}$
18	10010_{two}	$(10010_{two} \bmod 8) = 010_{two}$
16	10000_{two}	$(10000_{two} \bmod 8) = 000_{two}$

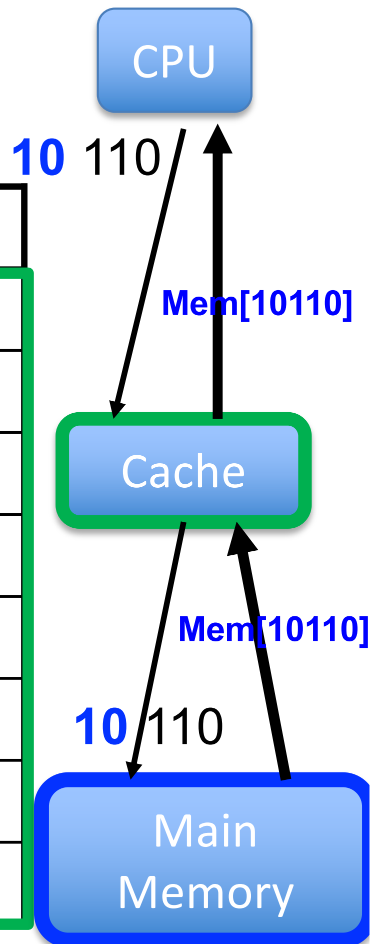
Cache Example

Decimal addr	Binary addr	Hit/miss	Cache block
22	**10** 110	Miss	110

E.g. Ld x1, 22(x0)

ecimal add of referen	Binary address of reference
22	10110_{two}
26	11010_{two}
22	10110_{two}
26	11010_{two}
16	10000_{two}
3	00011_{two}
16	10000_{two}
18	10010_{two}
16	10000_{two}

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	**Y**	**10**	**Mem[10110]**
111	N		



Cache Example

Decimal addr	Binary addr	Hit/miss	Cache block
26	**11** 010	Miss	010

ecimal add of referen	Binary address of reference
22	10110_{two}
26	11010_{two}
22	10110_{two}
26	11010_{two}
16	10000_{two}
3	00011_{two}
16	10000_{two}
18	10010_{two}
16	10000_{two}

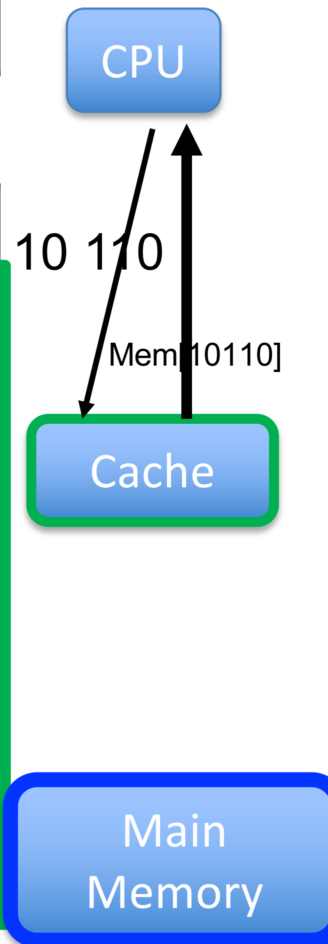
Index	V	Tag	Data
000	N		
001	N		
010	**Y**	**11**	**Mem[11010]**
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Decimal addr	Binary addr	Hit/miss	Cache block
22	10 110	**Hit**	110
26	11 010	**Hit**	010

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		

Decimal address of reference	Binary address of reference
22	10110_{two}
26	11010_{two}
22	10110_{two}
26	11010_{two}
16	10000_{two}
3	00011_{two}
16	10000_{two}
18	10010_{two}
16	10000_{two}



Cache Example

Decimal addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

ecimal add of referen	Binary address of reference
22	10110_{two}
26	11010_{two}
22	10110_{two}
26	11010_{two}
16	10000_{two}
3	00011_{two}
16	10000_{two}
18	10010_{two}
16	10000_{two}

Index	V	Tag	Data
000	**Y**	**10**	**Mem[10000]**
001	N		
010	Y	11	Mem[11010] (26)
011	**Y**	**00**	**Mem[00011]**
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

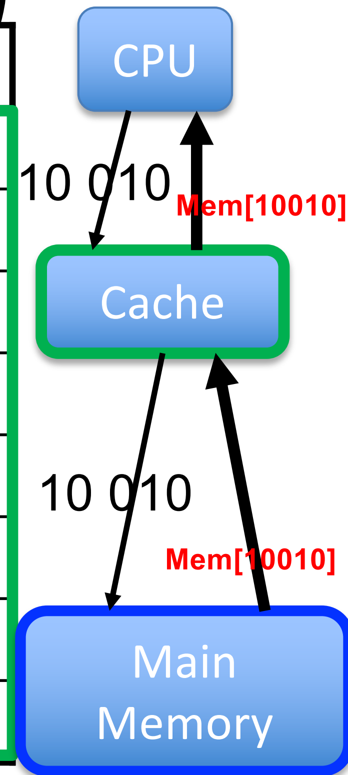
Cache Example

Decimal addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

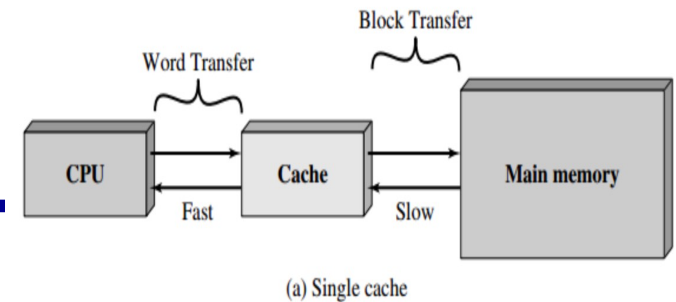
Replace Mem[11 010] (26)

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010] (18)**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Decimal address of reference	Binary address of reference
22	10110_{two}
26	11010_{two}
22	10110_{two}
26	11010_{two}
16	10000_{two}
3	00011_{two}
16	10000_{two}
18	10010_{two}
16	10000_{two}

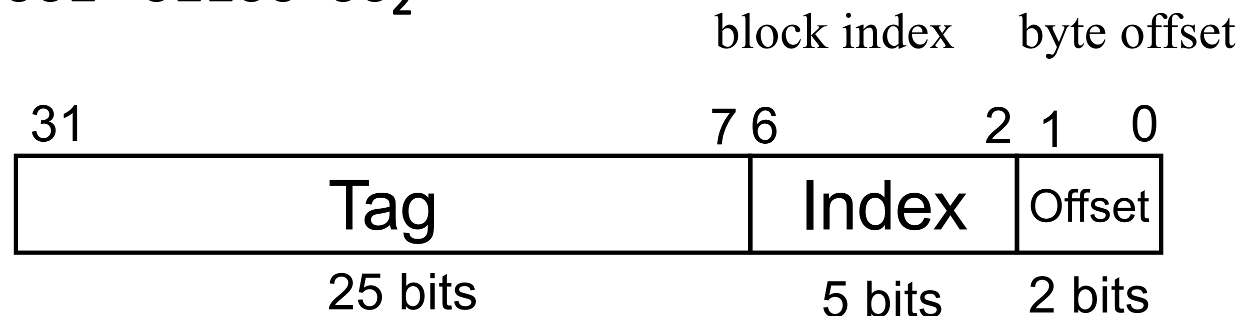


Byte-address Memory Access



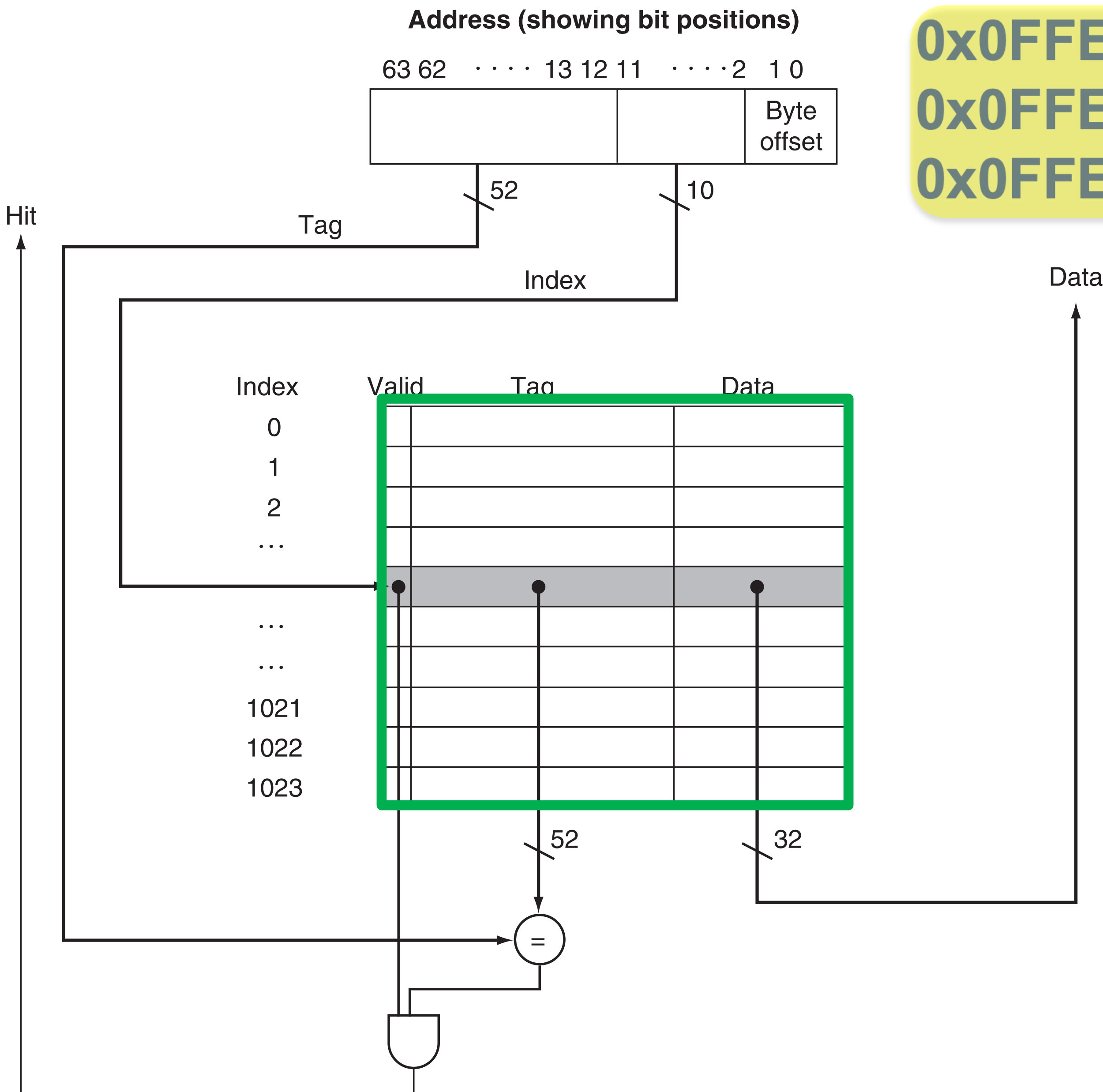
- 32 blocks, 4 bytes/block (1 word/blk)
  - **2 bits for addressing a byte within a block, byte offset**
  - 5 bits for addressing a block of the cache, block address
- To what block number does byte-address 1200 map?
- Use binary address to find out the solution:

$$\mathbf{1200_{10} = ...01001\ \ 01100\ \ 00_2}$$



- **Bytes at addresses 1200, 1201, 1203, 1203 are all in the same block**
  - **$01001\ \ 01100\ \ 00_2$**
  - **$01001\ \ 01100\ \ 01_2$**
  - **$01001\ \ 01100\ \ 10_2$**
  - **$01001\ \ 01100\ \ 11_2$**
- **01001 01100 00 can be considered as the block address of any of the four bytes**

Address Subdivision



0x0FFE1230: add x1, x2, x3
0x0FFE1234: lw|sw x1, 32(x2)
0x0FFE1238: beq x1, x2, offset

64-bit address
Each block has 4 bytes
→2 bits for byte offset
1024 (2^^10 blocks)
→ 10 bits for block address
64-2-10 = 52 bits tag

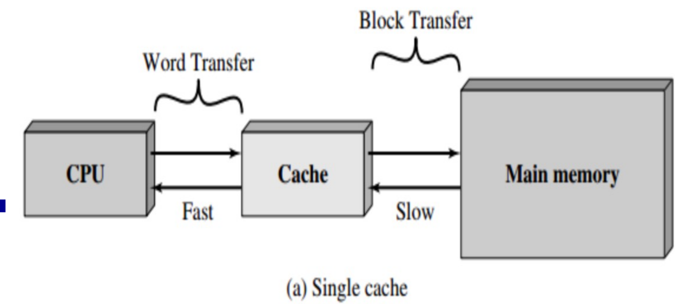
Pseudo Code for Simulating Directed-Mapped Cache Read Access (lb instruction)

```
#typedef struct cacheLine {
        int v;
        int tag;
        char Datablock[4]; //each datablock is 4 bytes, the address is byte address
} cacheLine_t;
cacheLine_t cache [1024]; //cache has 1024 blocks

int memory[1024x1024]; //memory has 1024*1024 blocks

char loadByte(long int address) { //directed mapped cache
        int cacheIndex = address[2:11]; //2-11 bit of the 32-bit address, which is the index
        cacheLine_t cline = cache[cacheIndex];
        If (cline.v && cline.tag == address[12:63]) { //cache hit
                return cline.Datablock[address[0:1]];
        } else { //miss
                //fetch data from memory and update cache line
                cline.dataBlock = fetchABlockMemory(memory, address); //e.g. memory[address]
                cline.v = 1;    cline.tag = address[12:63];
                return cline.Datablock[address[0:1]];
        }
}
```

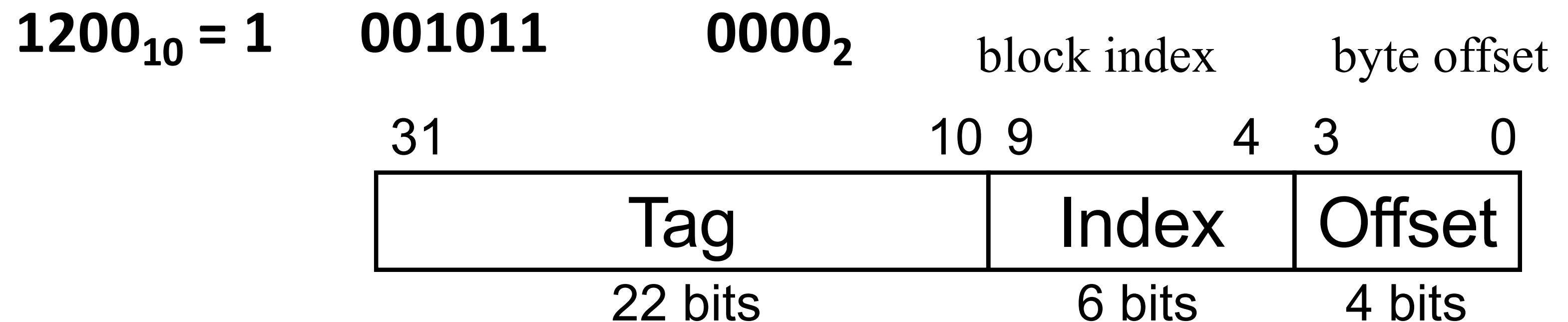
Larger Block Size



(a) Single cache

- 64 blocks, **16 bytes/block (4 words/blk)**
  - **4 bits for addressing a byte within a block, byte offset**
  - 6 bits for addressing a block of the cache, block address
- To what block number does byte-address 1200 map?
- Use binary address to find out the solution:

1200_{10} = 1 001011 0000_2

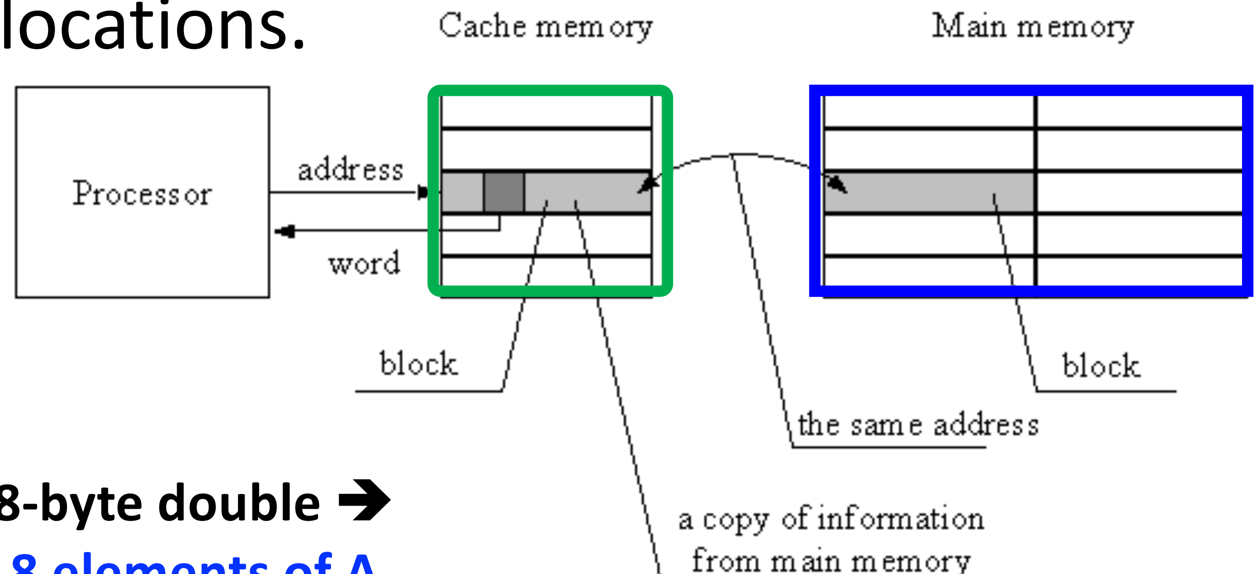


- **Bytes at addresses 1200 to 1215 are all in the same block**
  - **01 001011 0000_2**
  - **01 001011 0001_2**
  - **01 001011 0010_2**
  - …
  - **01 001011 1111_2**
- **01 001011 0000 can be considered as the block address of any of the 16 bytes at 1200 to 1215**

Caching Exploits Both Types of Locality by Preloading and Keeping Data in Faster Memory

- Exploit temporal locality by keeping the contents of recently accessed locations in the cache.
- Exploit spatial locality by fetching blocks of data around recently accessed locations.

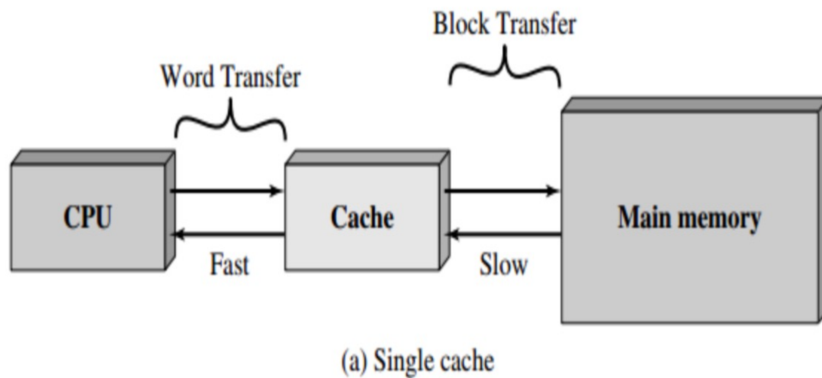
```
double A[N];
sum = 0;
for(i=0; i<N; i++)
  sum += A[i];
return sum;
```



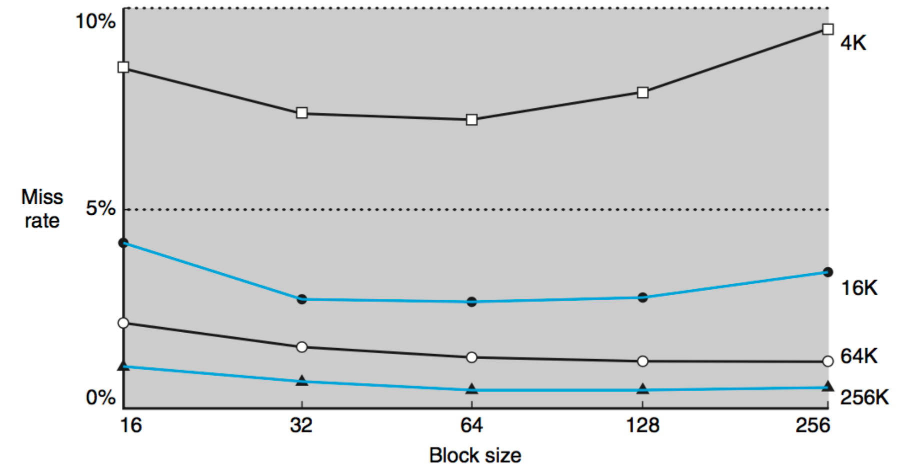
64-byte block size, A[i] is an 8-byte double ➔
A block (cache line) can hold 8 elements of A.

Referencing to A[0] (or A[1], …, A[7]) will cause the memory system to bring A[0:7] to the cache ➔
Future reference to A[1:7] are all hits in cache ➔ faster access than reading from memory

Block Size Considerations



(a) Single cache



- Larger blocks should reduce miss rate
  - Due to spatial locality
- But in a fixed-sized cache
  - Larger blocks ⇒ fewer of them
    - More competition ⇒ increased miss rate
  - Larger blocks ⇒ pollution
- Larger miss penalty
  - Can override benefit of reduced miss rate
  - Early restart and critical-word-first can help

Cache Memory Size Calculation

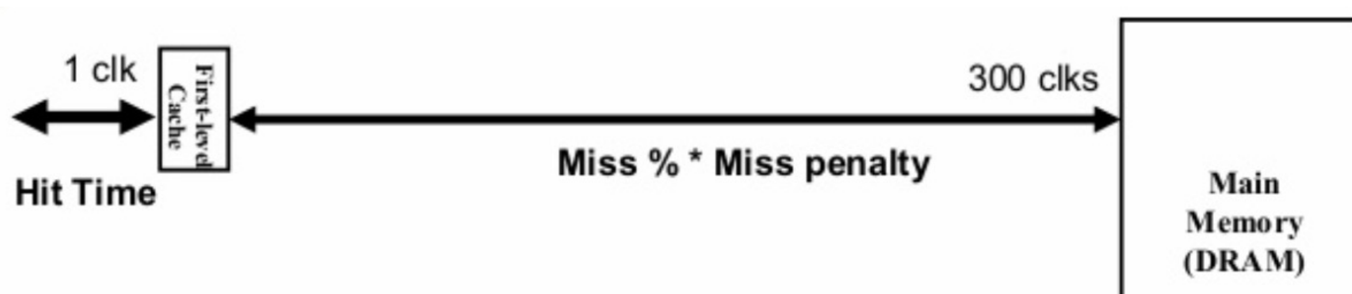
- A cache line: Valid bit + Tag + Data

Index	V	Tag	Data
000	N		

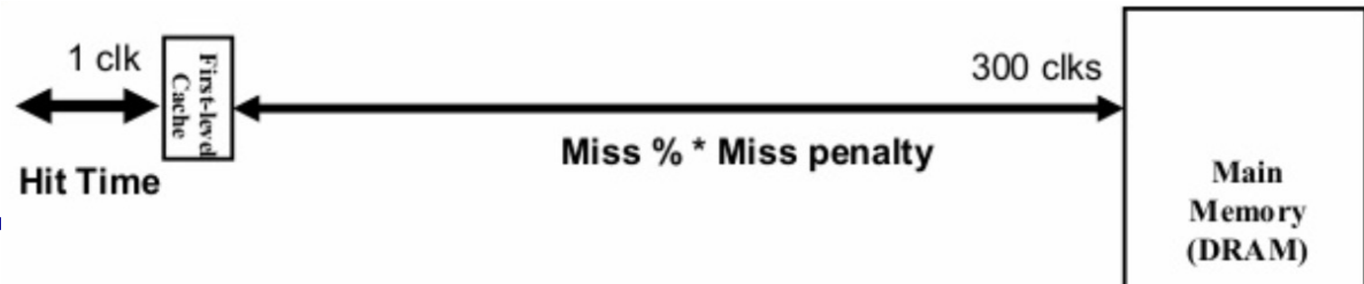
- Directed Mapped Cache, 16KiB for data, four words blocks, 64-bit address
  - 16KiB is 4096 (2^{12}) words (2^{14} bytes), Each block is 4 (2^2) words = 16 bytes (2^4)
    - Thus there are $2^{14}/2^4$ = 1024 (2^{10}) blocks
    - 4 bit for byte offset within a block
      - 2 bit for word offset within a block, 2 bit for byte offset within a word
    - Thus # tag bits: 64 – 10 – 4 = 50
    - 1 bit for valid
  - Total bits for the cache 16KiB + 2^{10} * 51 = 179 Kibits = 22.4 KiB for 16KiB cache
    - Total SRAM needed is 1.4 times of SRAM for data

Terminology for Memory Hierarchy

- **Hit**: Data appears in cache
  - **Hit Rate:** the fraction of memory access found in the cache.
  - **Hit Time:** Time to access the cache
- **Miss**: data needs to be retrieve from a block in memory
  - **Miss Rate** = 1 - (Hit Rate), i.e. # misses / # memory access
  - **Miss Penalty:** Time to replace a block in cache
- **Hit Time << Miss Penalty (100x cycles)**



Cache Misses



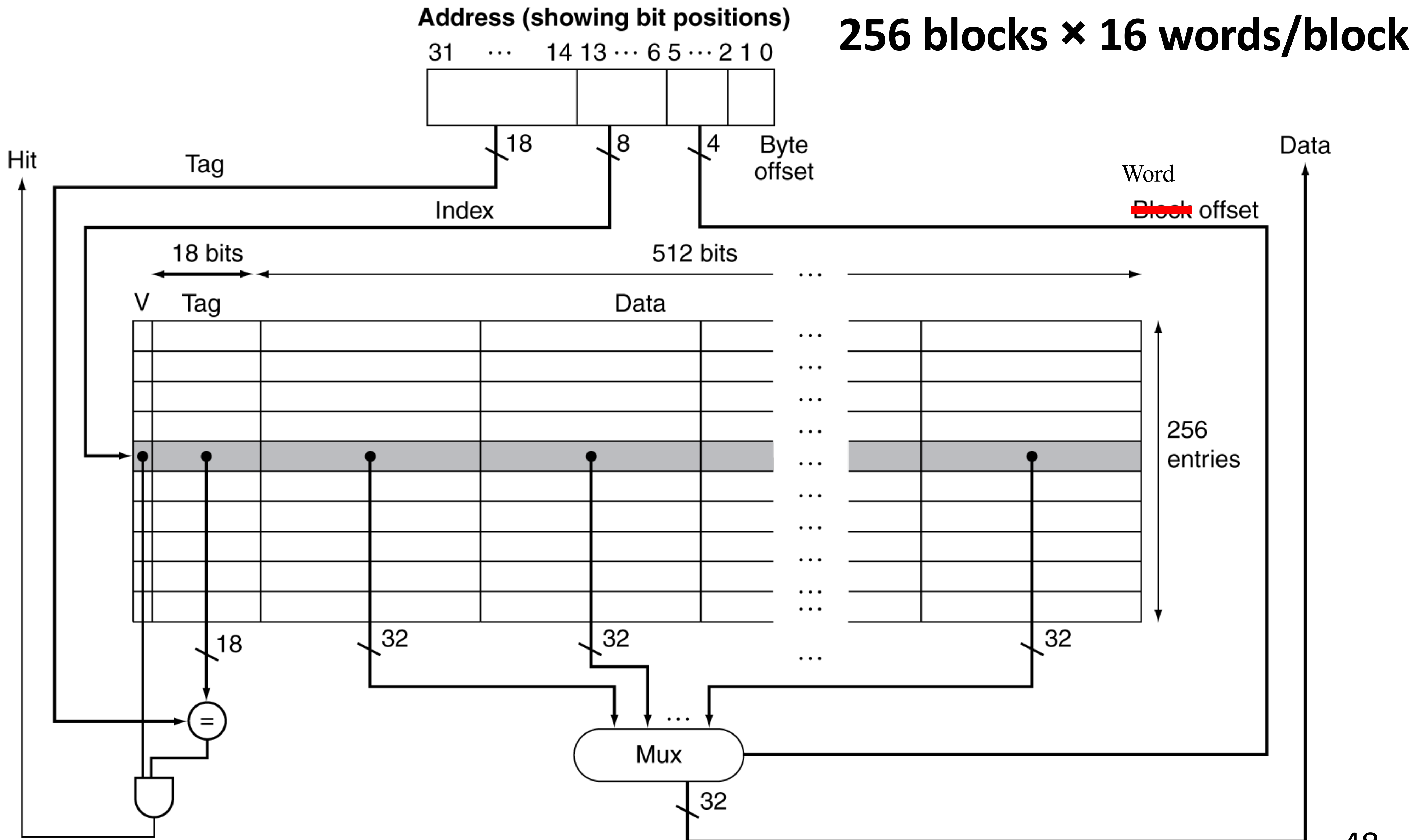
- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline in MEM stage

IF ID EX Mem **stall stall stall … stall** Mem Wr
IF ID EX **stall stall stall … stall stall** Ex Wr

  - Fetch block from next level of hierarchy
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access

Example: Intrinsity FastMATH

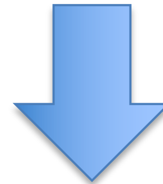
256 blocks × 16 words/block



Example: Intrinsity FastMATH

- Embedded MIPS processor
  - 12-stage pipeline
  - Instruction and data access on each cycle
- Split cache: separate I-cache and D-cache
  - Each 16KB: 256 blocks × 16 words/block
  - D-cache: write-through or write-back
- SPEC2000 miss rates
  - I-cache: 0.4%
  - D-cache: 11.4%
  - Weighted average: 3.2%

What are the low miss rate indicating?



Very good principle locality and memory system works well with the principle.

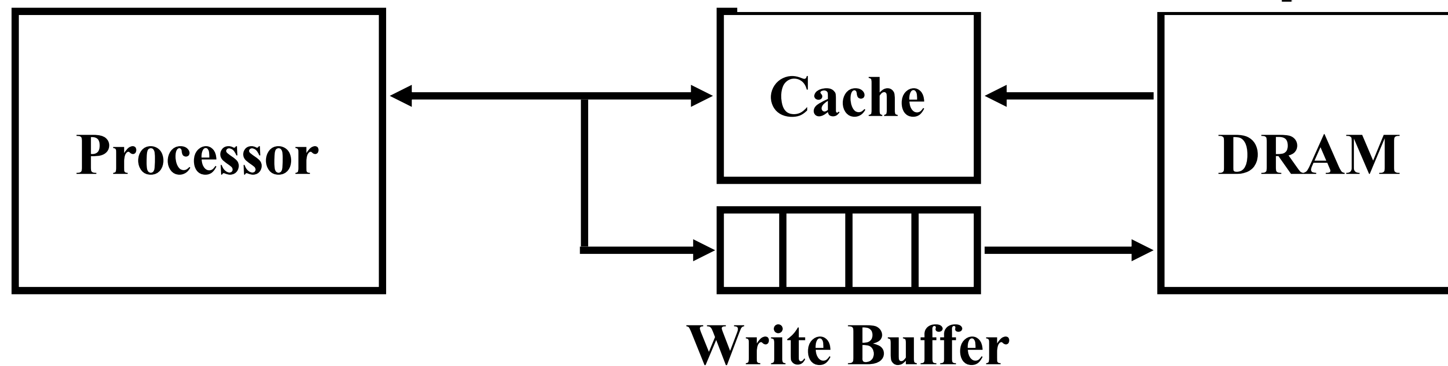
Main Memory Supporting Caches

- Use DRAMs for main memory
  - Fixed width (e.g., 1 word)
  - Connected by fixed-width clocked bus
    - Bus clock is typically slower than CPU clock
- **Example cache block read**
  - **1 bus cycle for address transfer**
  - **15 bus cycles per DRAM access**
  - **1 bus cycle per data transfer**

- **For 4-word block, 1-word-wide DRAM**
  - **Miss penalty = 1 + 4×15 + 4×1 = 65 bus cycles**
  - **Bandwidth = 16 bytes / 65 cycles = 0.25 B/cycle**

Write-Through

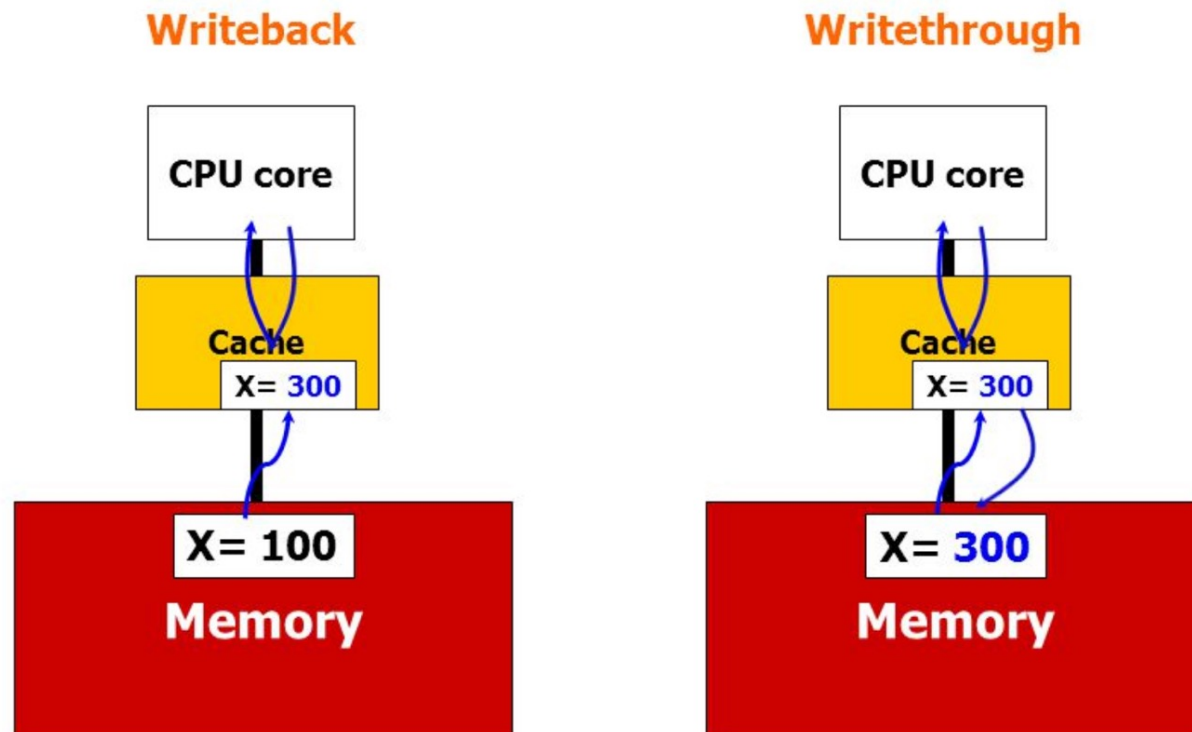
```
0x0FFE1230: add x1, x2, x3
0x0FFE1234: lw|sw x1, 32(x2)
0x0FFE1238: beq x1, x2, offset
```



- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: also update memory
- But makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = 1 + 0.1×100 = 11
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full

Write-Back

- Alternative: On data-write hit, just update the block in cache
  - Keep track of whether each block is dirty
- When a dirty block is replaced
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first



Write Allocation: What should happen on a write miss?

- Alternatives for write-through
  - Allocate on miss: fetch the block
  - Write around: don't fetch the block
    - Since programs often write a whole block before reading it (e.g., initialization)
- For write-back
  - Usually fetch the block

Write Policy

Hit:
- write-through
- write-back

Miss:
- write-allocate = read miss
- no-write allocate

CPU
Address
Data
Cache
Address
Data
Main Memory

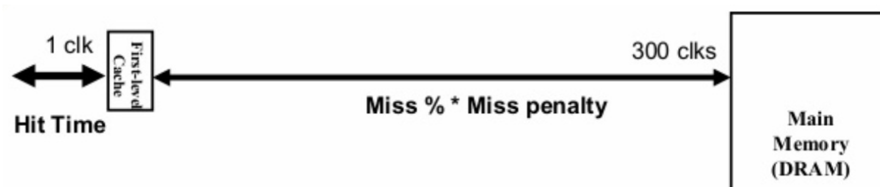
Chapter 5: Large and Fast: Exploiting Memory Hierarchy

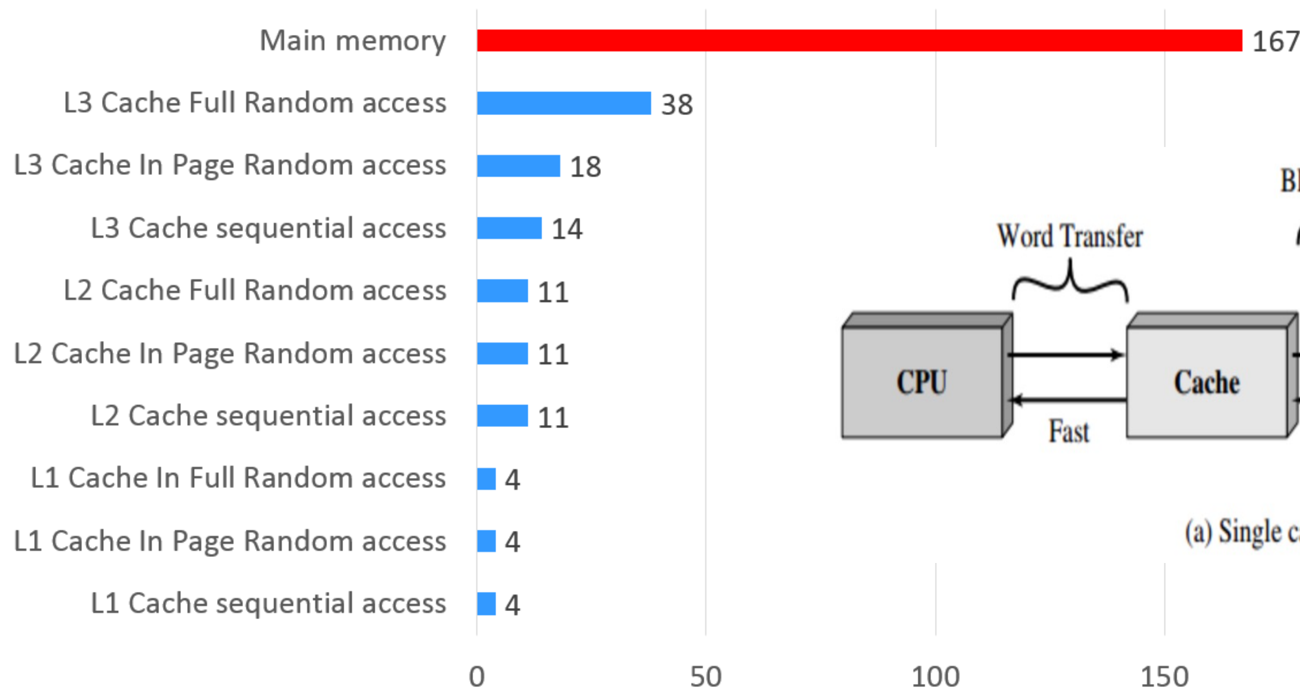
- Lecture
  - 5.1 Introduction
  - 5.2 Memory Technologies
- Lecture
  - 5.3 The Basics of Caches
- ☛ Lecture
  - 5.4 Measuring and Improving Cache Performance
  - ~~5.5 Dependable Memory Hierarchy~~
  - ~~5.6 Virtual Machines~~
- Lecture
  - 5.6 Virtual Memory
  - ~~5.8 A Common Framework for Memory Hierarchy~~
- Lecture 26
  - ~~5.9 Using a Finite-State Machine to Control a Simple Cache~~
  - ~~5.10 Parallelism and Memory Hierarchies: Cache Coherence~~
  - ~~5.11 Parallelism and Memory Hierarchy: Redundant Arrays of Inexpensive Disks~~
  - ~~5.12 Advanced Material: Implementing Cache Controllers~~
  - 5.13 Real Stuff: The ARM Cortex-A53 and Intel Core i7 Memory Hierarchies
  - ~~5.14 Going Faster: Cache Blocking and Matrix Multiply~~
  - ~~5.15 Fallacies and Pitfalls~~
  - 5.16 Concluding Remarks

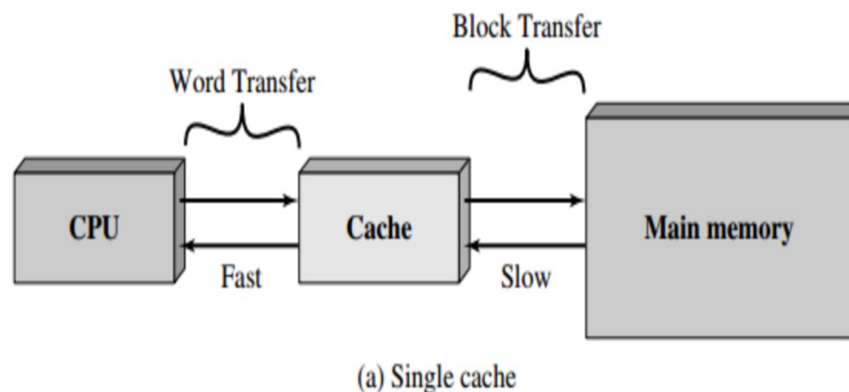
Measuring Cache Performance

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware

$$CPU\ \text{Time} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$



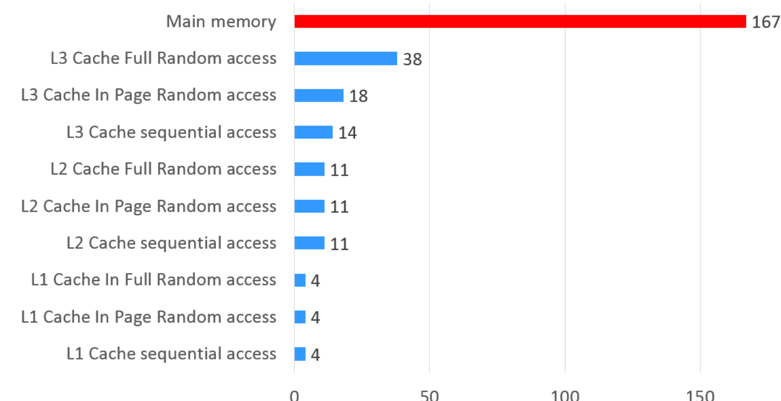




(a) Single cache

CPU Performance with Memory Factor

- Components of CPU time
  - Program execution cycles
    - Includes cache hit time
  - **Memory stall cycles**
    - **Mainly from cache misses**



- **Memory Stall Cycles:** the number of cycles during which the processor is stalled waiting for a memory access.

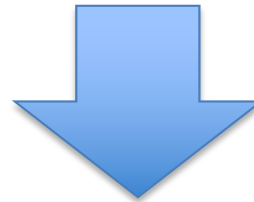
$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle time}$$

$$CPU\ \text{Time} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

Memory Stall Cycles per Instruction

$$CPU\ \text{Time} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle time}$$



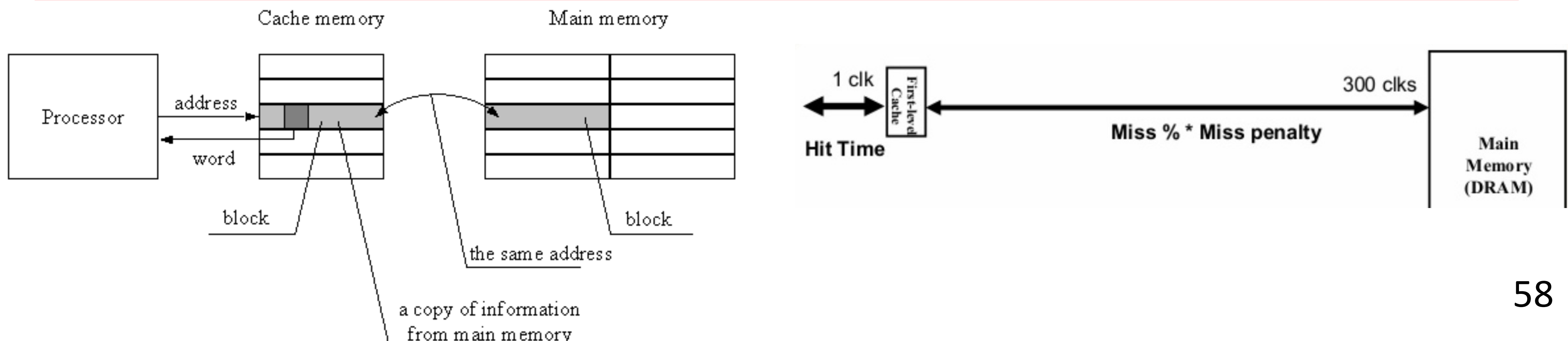
$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}}\right) \times \text{Clock cycle time}$$

- CPI and Memory stall cycles/instruction are averages

Memory Stall Cycles

- **Memory Stall Cycles:** the number of cycles during which the processor is stalled waiting for a memory access.
  - Depends on both the number of misses and the cost per miss, i.e. the miss penalty:

$$\text{Memory stall cycles} = \text{Number of misses} \times \text{Miss penalty}$$
$$= \text{IC} \times \frac{\text{Misses}}{\text{Instrution}} \times \text{Miss Penalty}$$
$$= \text{IC} \times \frac{\text{Memory accesses}}{\text{Instrution}} \times \text{Miss rate} \times \text{Miss Penalty}$$



Miss Rate: Miss per Memory Reference

Miss Rate = # Misses / # Memory reference

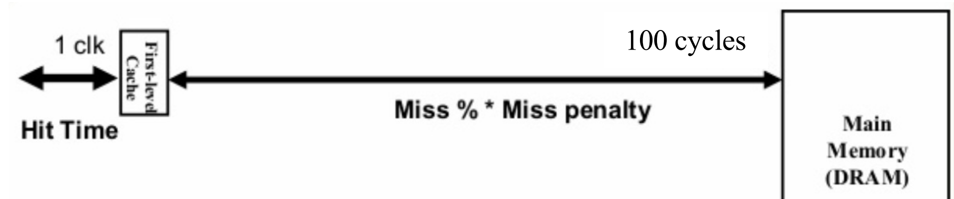
- Memory access include both instruction access and data access
  - Each instruction needs to be read from memory: one I-mem access
  - LW/SW are memory access instructions: one D-mem access (in addition to one I-mem access)
  - **Count # memory accesses: Each iteration, 14 instructions in total, 9 ld/lw/sw ➔ 14+9 or 9*2+5 = 23 memory accesses**
    - **9 are data-mem accesses**

```
int sum(int N, int a, int *X) {
    int i;
    int result = 0;
    for (i = 0; i < N; ++i)
        result += X[i];
    return result;
}
```

```
.L3:
        lw      a5,-20(s0)     /* a5 = i */
        sll     a5,a5,2        /* a5 = i<<2, which is i=i*4
        ld      a4,-48(s0)     /* a4 = X */
        add     a5,a4,a5       /* the &X[i] */
        lw      a5,0(a5)       /* the X[i] */
        lw      a4,-24(s0)     /* load result */
        addw    a5,a4,a5       /* result += X[i] */
        sw      a5,-24(s0)     /* store to result */
        lw      a5,-20(s0)     /* i */
        addw    a5,a5,1        /* i++ */
        sw      a5,-20(s0)     /* store i */
.L2:
        lw      a4,-20(s0)     /* i */
        lw      a5,-36(s0)     /* N */
        blt     a4,a5,.L3      /* if (i < N) goto .L3 */
```

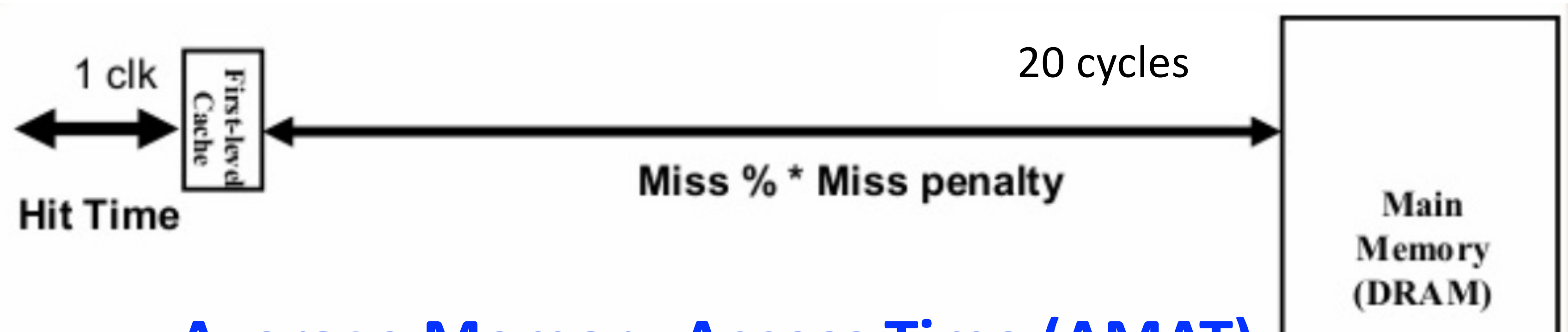
Cache Performance

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}} \right) \times \text{Clock cycle time}$$



- Given
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions
- Stall cycles per instruction (Misses/Instruction * Miss Penalty)
  - I-mem: 1* 0.02 × 100 = 2 (each instruction has one I-cache/mem access)
  - D-mem: 0.36 × 0.04 × 100 = 1.44 (only load/store has D-cache access)
- Actual CPI = 2 + 2 + 1.44 = 5.44
  - Ideal CPU is (5.44/2 =2.72) times faster
  - 2+1.44 cycles/instruction on memory stalls → **3.44/5.44 = 63%**
- **Miss penalty (100 cycles) is the killing factor**
  - **DRAM speed**

Average Memory Access Time (AMAT)



Average Memory Access Time (AMAT)
= Hit Time + Miss Rate * Miss Penalty

- **Miss penalty**: Time to fetch a block from lower memory level
  - Access time: function of latency
  - Transfer time: function of bandwith b/w levels
    - Transfer one "cache block/line" at a time
- Example:
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
  - AMAT = 1 + 0.05 × 20 = 2ns
    - 2 cycles per instruction

Three Important Equations for Cache Performance

Average Memory Access Time (AMAT)

= Hit Time + Miss Rate * Miss Penalty

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}} \right) \times \text{Clock cycle time}$$

$$\begin{aligned} \text{Memory stall cycles} &= \text{Number of misses} \times \text{Miss penalty} \\ &= \text{IC} \times \frac{\text{Misses}}{\text{Instrution}} \times \text{Miss Penalty} \\ &= \text{IC} \times \frac{\text{Memory accesses}}{\text{Instrution}} \times \text{Miss rate} \times \text{Miss Penalty} \end{aligned}$$

Performance Summary

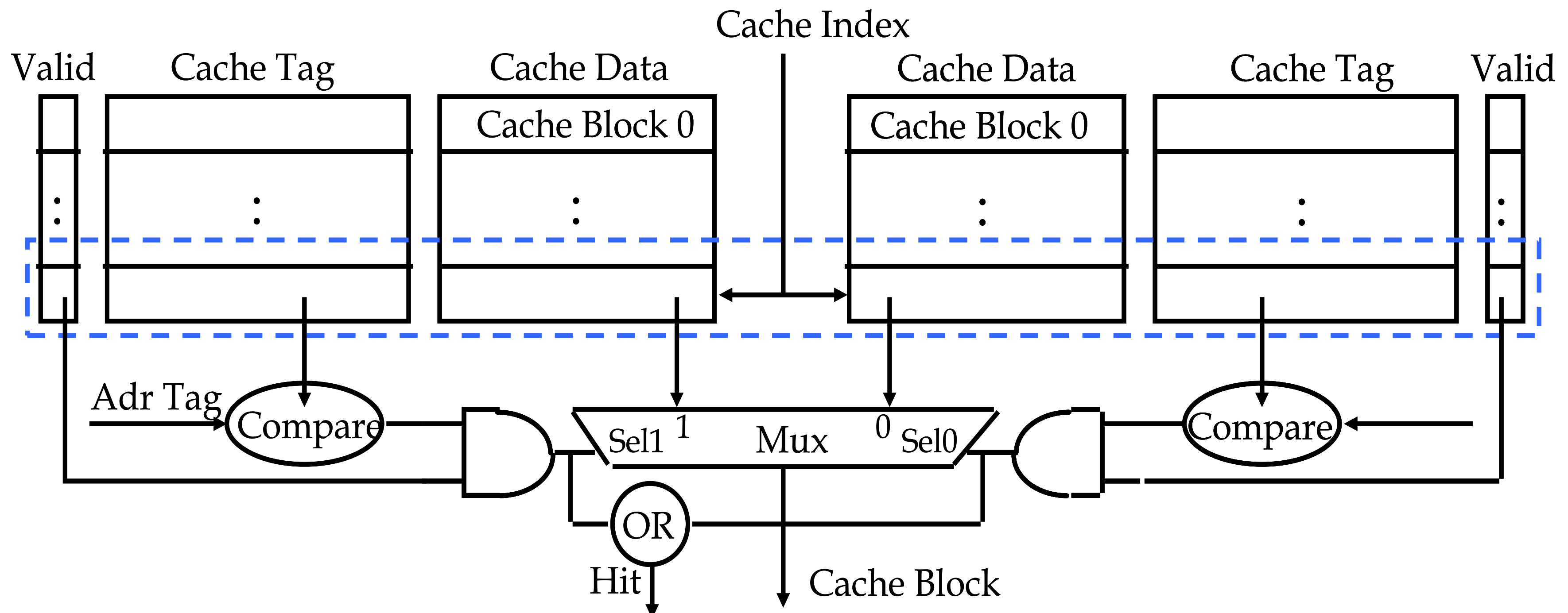
- When CPU performance increased
  - Miss penalty becomes more significant
- Decreasing base CPI
  - Greater proportion of time spent on memory stalls
- Increasing clock rate
  - Memory stalls account for more CPU cycles
- **Can’t neglect cache behavior when evaluating system performance**

Associative Caches

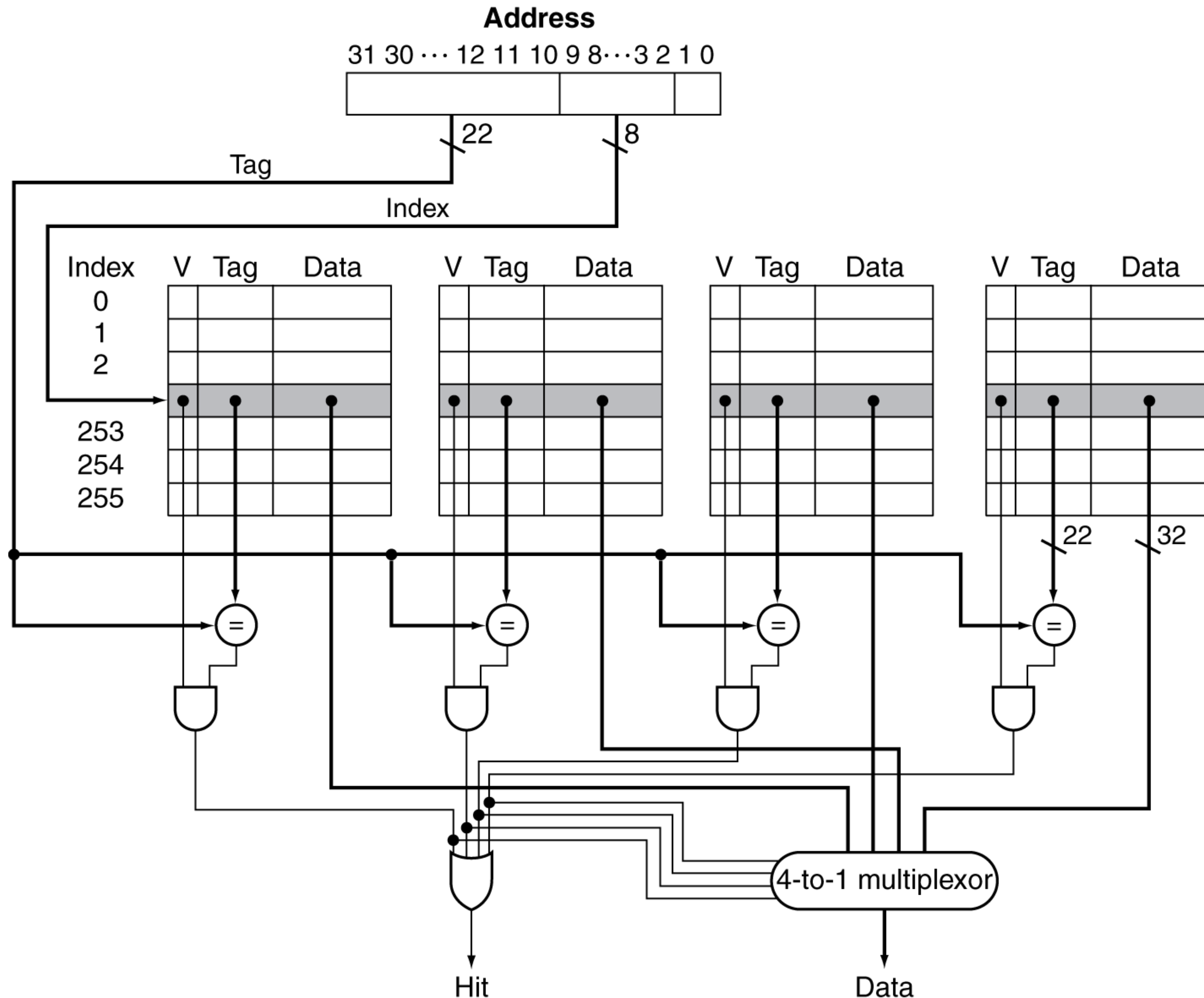
- Fully associative
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
  - Comparator per entry (expensive)
- *n*-way set associative
  - Each set contains *n* entries
  - Block number determines which set
    - (Block number) modulo (#Sets in cache)
  - Search all entries in a given set at once
  - *n* comparators (less expensive)

Set Associative Cache

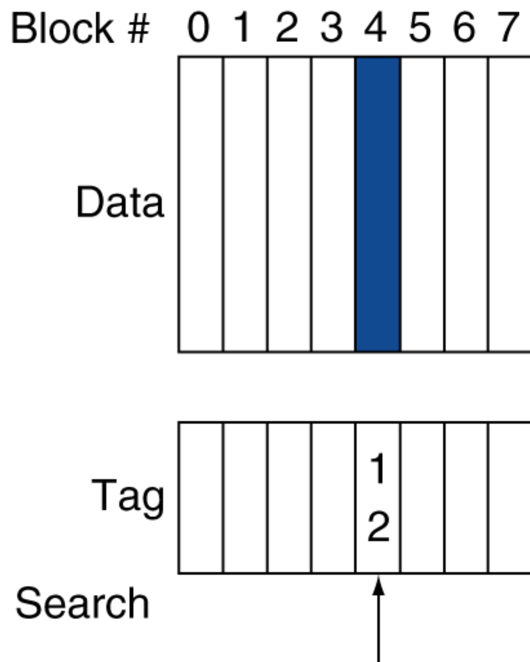
- *N*-way set associative: *N* entries for each **Cache Index**
  - *N* direct mapped caches operates in parallel
- Example: Two-way set associative cache
  - **Cache Index** selects a “set” from the cache;
  - The two tags in the set are compared to the input in parallel;
  - Data is selected based on the tag result.

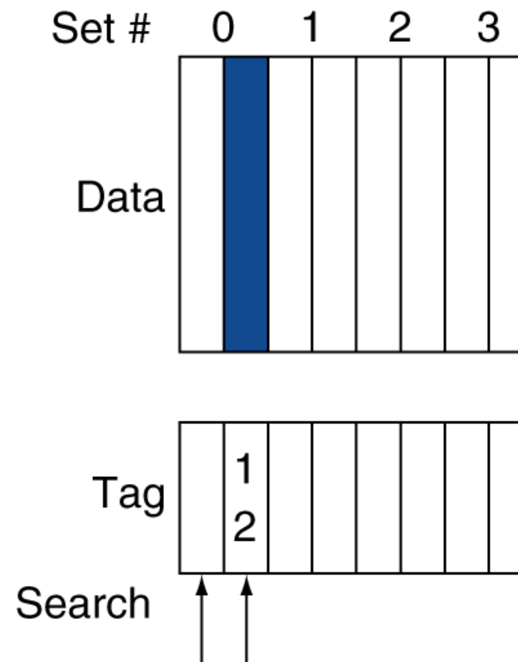


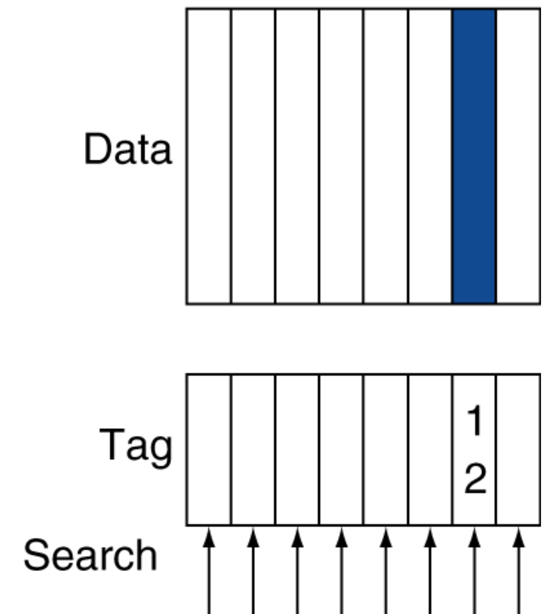
4-Way Set Associative Cache



Associative Cache Example







Spectrum of Associativity

- For a cache with 8 entries

One-way set 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

Associativity Example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8
- Direct mapped

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	

Associativity Example

- 2-way set associative

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	**Mem[0]**			
8	0	miss	Mem[0]	**Mem[8]**		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	Mem[0]	**Mem[6]**		
8	0	miss	**Mem[8]**	Mem[6]		

- Fully associative

Block address		Hit/miss	Cache content after access			
0		miss	**Mem[0]**			
8		miss	Mem[0]	**Mem[8]**		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	**Mem[6]**	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

How Much Associativity

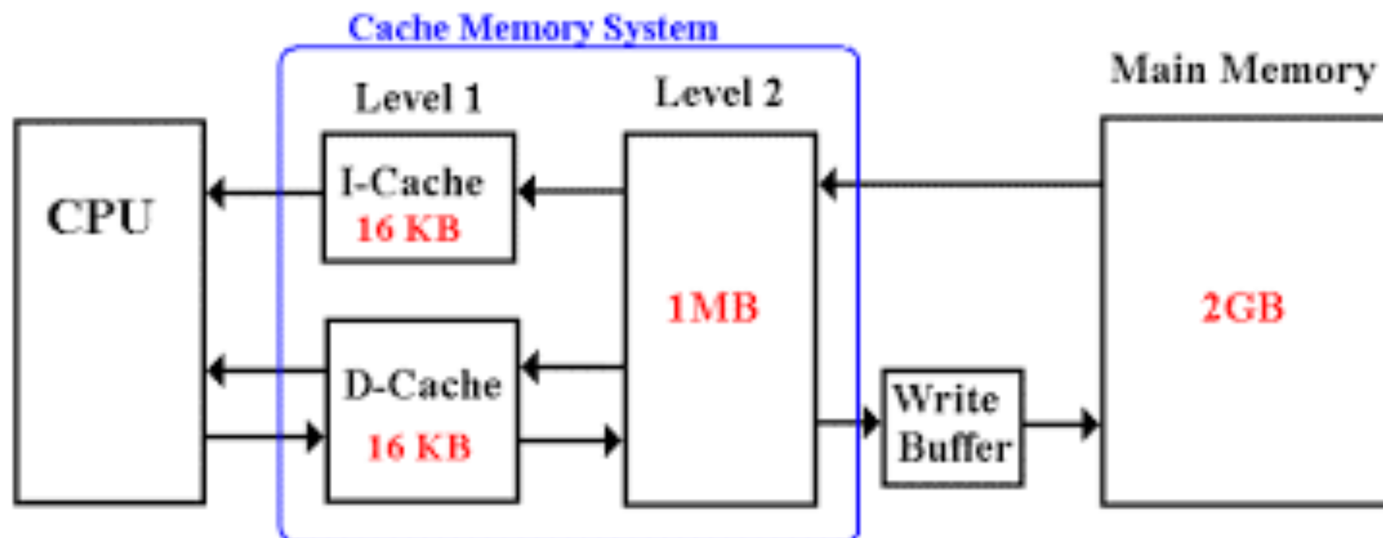
- Increased associativity decreases miss rate
  - But with diminishing returns
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000; Miss rate:
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%

Replacement Policy

- Direct mapped: no choice
- Set associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
- Least-recently used (LRU)
  - Choose the one unused for the longest time
    - Simple for 2-way, manageable for 4-way, too hard beyond that
- Random
  - Gives approximately the same performance as LRU for high associativity

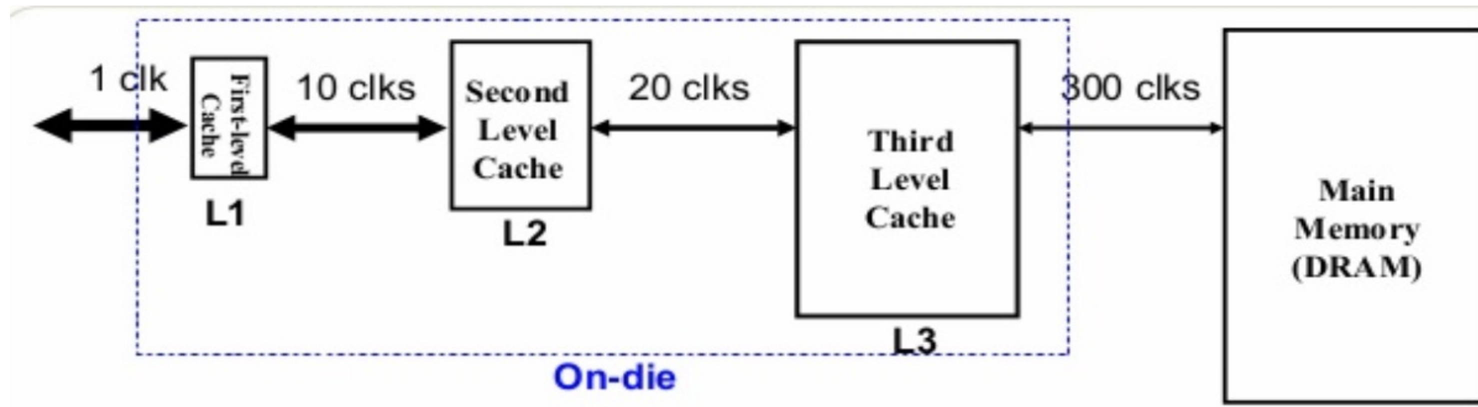
Multilevel Caches

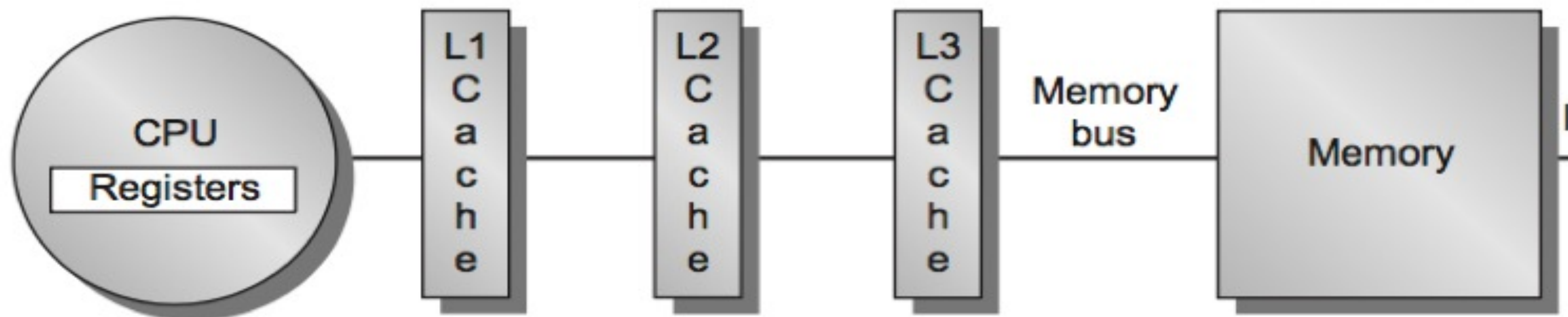
- Primary cache attached to CPU
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache



Multilevel Caches

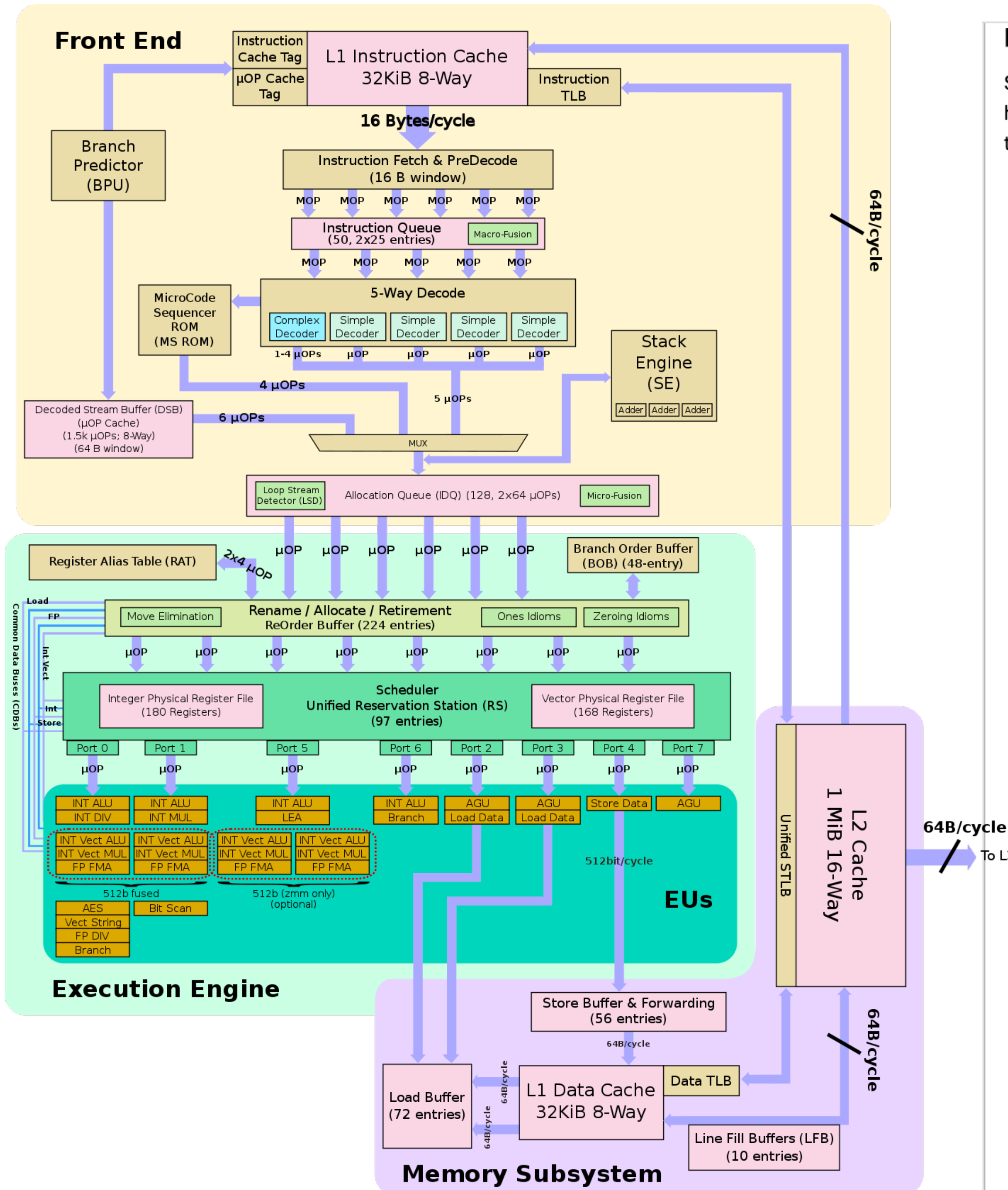
- More info for multilevel cache





Sky Lake - Microarchitectures – Intel

https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)



Memory Hierarchy [edit]

Some major organizational changes were done to the cache hierarchy in Skylake server configi
hierarchy for Skylake's server and HEDT processors has been rebalanced. Note that the L3 is n
the L3 cache was moved into the private L2 cache.

- Cache
  - L0 µOP cache:
    - 1,536 µOPs/core, 8-way set associative
      - 32 sets, 6-µOP line size
      - statically divided between threads, inclusive with L1I
  - L1I Cache:
    - 32 KiB/core, 8-way set associative
      - 64 sets, 64 B line size
      - competitively shared by the threads/core
  - L1D Cache:
    - 32 KiB/core, 8-way set associative
    - 64 sets, 64 B line size
    - competitively shared by threads/core
    - 4 cycles for fastest load-to-use (simple pointer accesses)
      - 5 cycles for complex addresses
    - 128 B/cycle load bandwidth
    - 64 B/cycle store bandwidth
    - Write-back policy
  - L2 Cache:
    - 1 MiB/core, 16-way set associative
    - 64 B line size
    - Inclusive
    - 64 B/cycle bandwidth to L1$
    - Write-back policy
    - 14 cycles latency
  - L3 Cache:
    - 1.375 MiB/core, 11-way set associative, shared across all cores
      - Note that a few models have non-default cache sizes due to disabled cores
    - 2,048 sets, 64 B line size
    - Non-inclusive victim cache
    - Write-back policy
    - 50-70 cycles latency

Multilevel Cache Example

- Given
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns
- With just primary cache
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = $1 + 0.02 \times 400 = 9$

Example (cont.)

- Now add L-2 cache
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
  - Penalty = 5ns/0.25ns = 20 cycles
- Primary miss with L-2 miss
  - Extra penalty = 500 cycles
- CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4
- Performance ratio = 9/3.4 = 2.6

Multilevel Cache Considerations

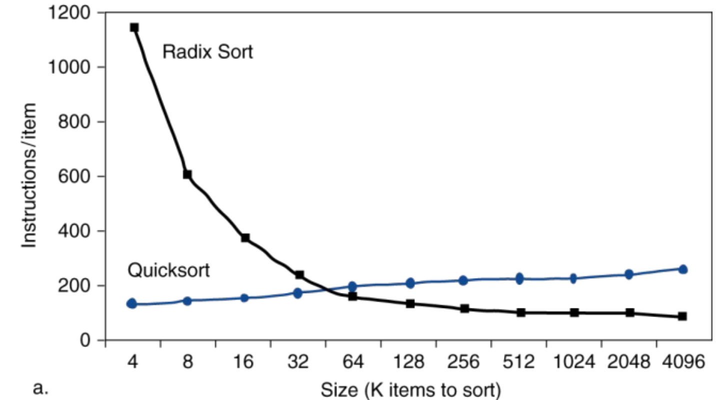
- Primary cache
  - Focus on minimal hit time
- L-2 cache
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact
- Results
  - L-1 cache usually smaller than a single cache
  - L-1 block size smaller than L-2 block size

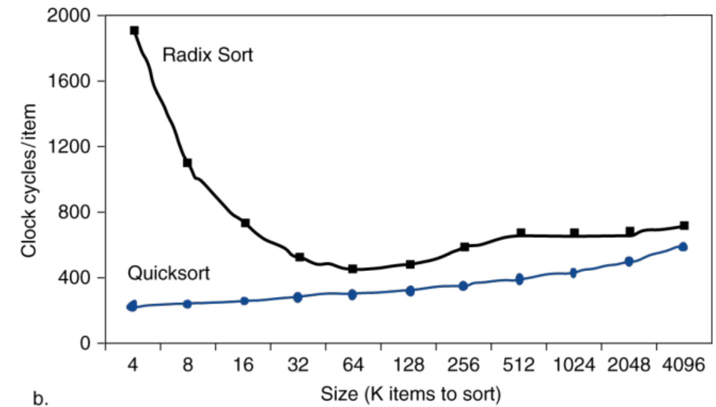
Interactions with Advanced CPUs

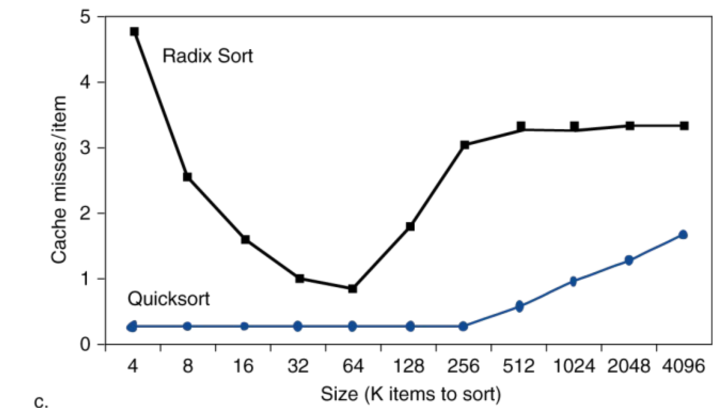
- Out-of-order CPUs can execute instructions during cache miss
  - Pending store stays in load/store unit
  - Dependent instructions wait in reservation stations
    - Independent instructions continue
- Effect of miss depends on program data flow
  - Much harder to analyse
  - Use system simulation

Interactions with Software

- Misses depend on memory access patterns
  - Algorithm behavior
  - Compiler optimization for memory access







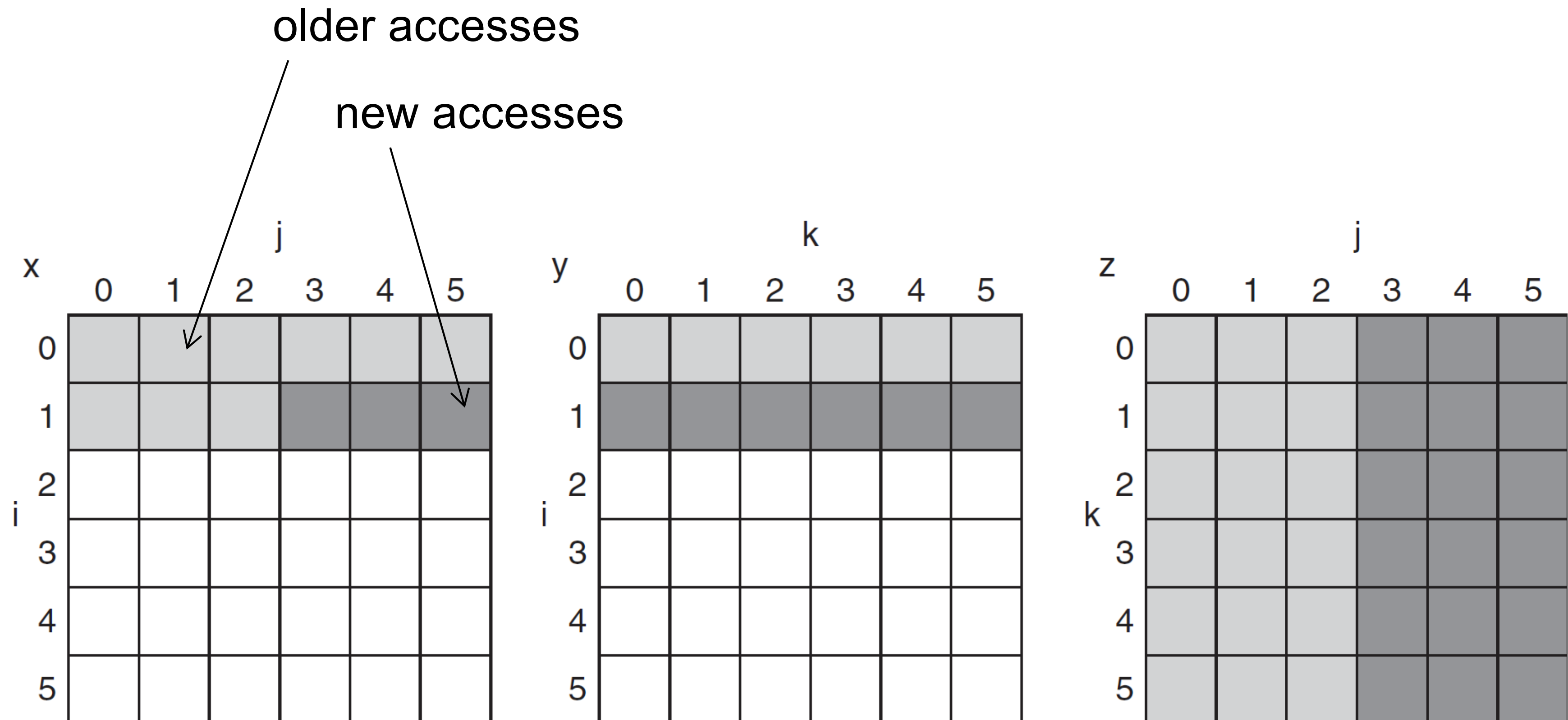
Software Optimization via Blocking

- Goal: maximize accesses to data before it is replaced
- Consider inner loops of DGEMM:

```
for (int j = 0; j < n; ++j)
{
  double cij = C[i+j*n];
  for( int k = 0; k < n; k++ )
    cij += A[i+k*n] * B[k+j*n];
  C[i+j*n] = cij;
}
```

DGEMM Access Pattern

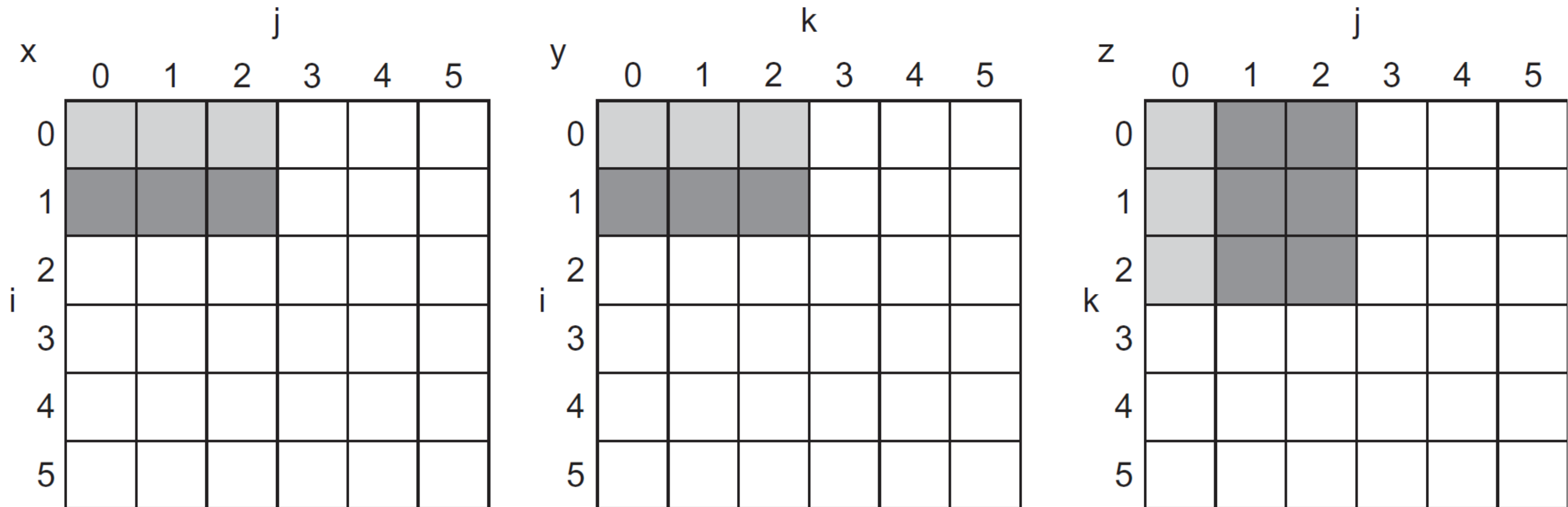
- C, A, and B arrays

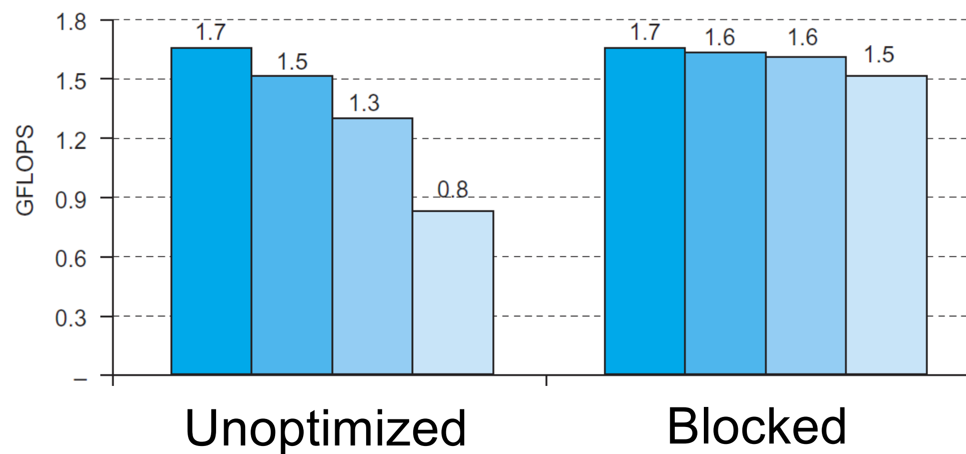


Cache Blocked DGEMM

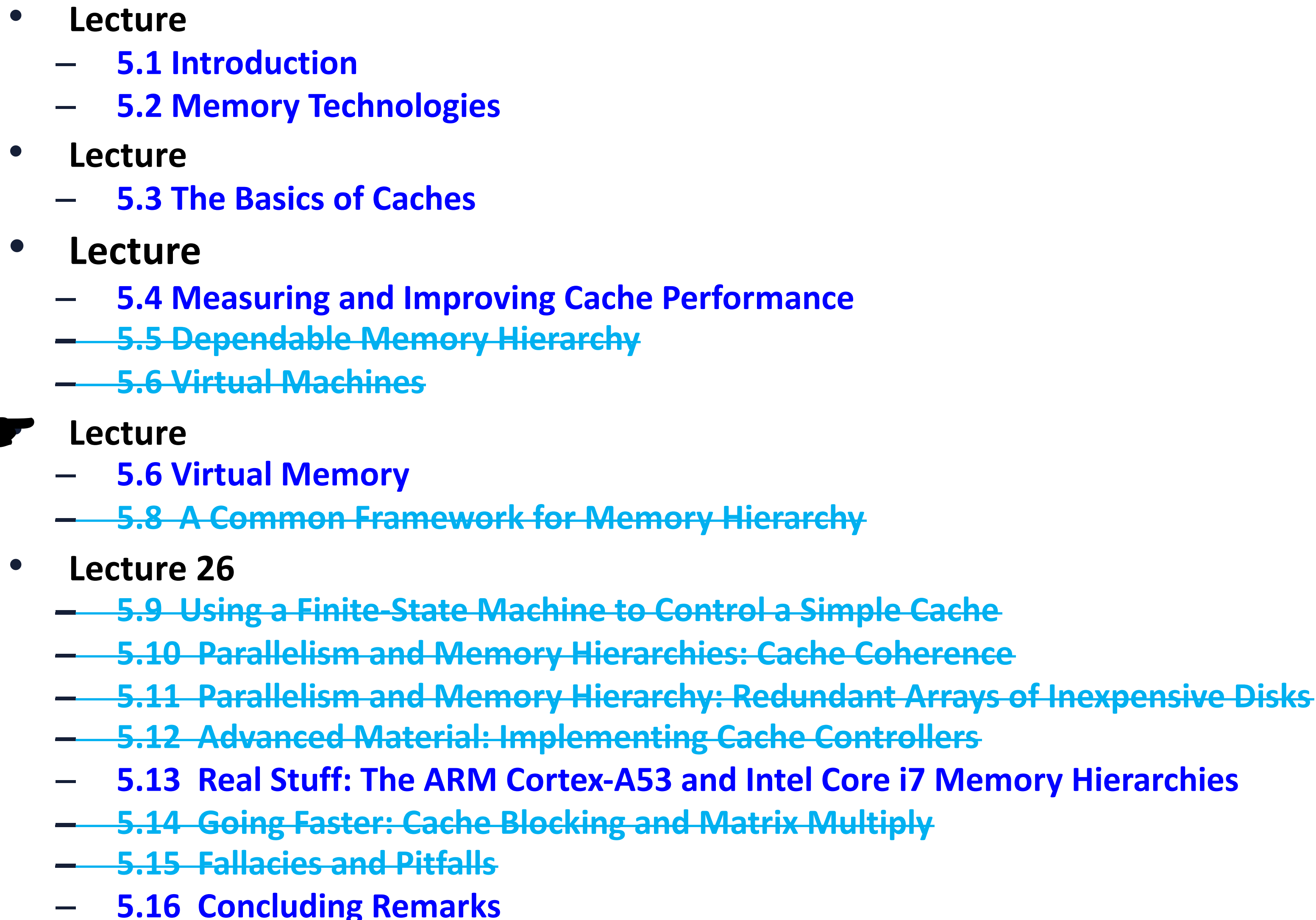
```
1 #define BLOCKSIZE 32
2 void do_block (int n, int si, int sj, int sk, double *A, double
3 *B, double *C)
4 {
5  for (int i = si; i < si+BLOCKSIZE; ++i)
6   for (int j = sj; j < sj+BLOCKSIZE; ++j)
7   {
8    double cij = C[i+j*n];/* cij = C[i][j] */
9    for( int k = sk; k < sk+BLOCKSIZE; k++ )
10    cij += A[i+k*n] * B[k+j*n];/* cij+=A[i][k]*B[k][j] */
11   C[i+j*n] = cij;/* C[i][j] = cij */
12  }
13 }
14 void dgemm (int n, double* A, double* B, double* C)
15 {
16  for ( int sj = 0; sj < n; sj += BLOCKSIZE )
17   for ( int si = 0; si < n; si += BLOCKSIZE )
18    for ( int sk = 0; sk < n; sk += BLOCKSIZE )
19     do_block(n, si, sj, sk, A, B, C);
20 }
```

Blocked DGEMM Access Pattern



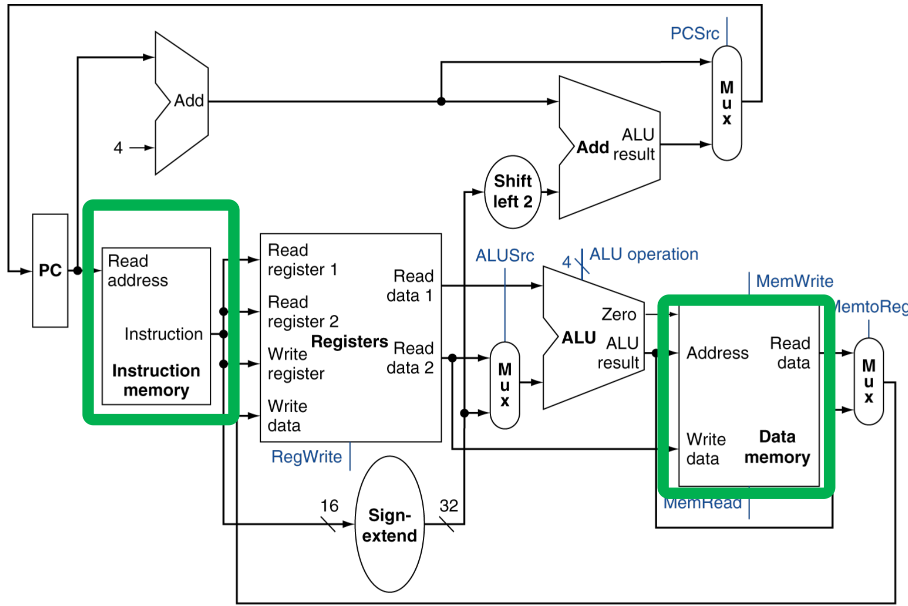


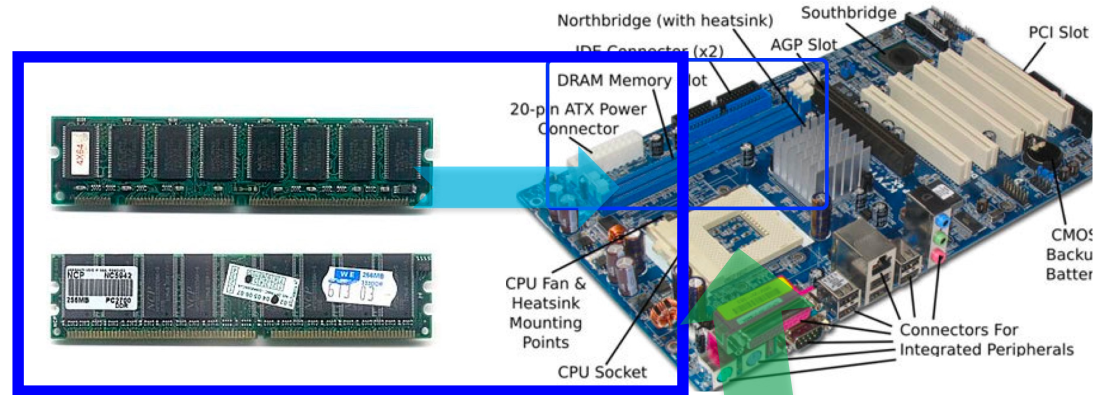
Chapter 5: Large and Fast: Exploiting Memory Hierarchy



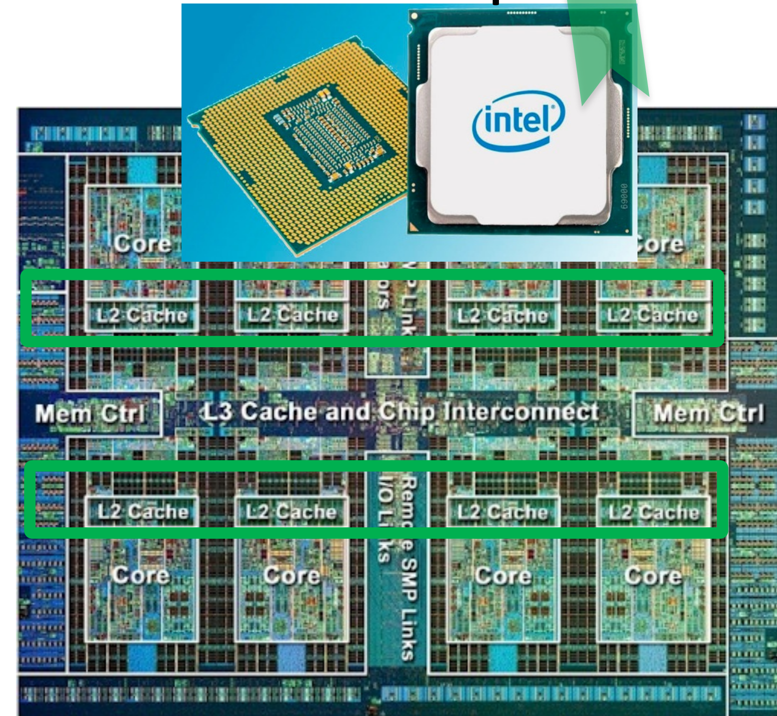
- Lecture
  - 5.1 Introduction
  - 5.2 Memory Technologies
- Lecture
  - 5.3 The Basics of Caches
- Lecture
  - 5.4 Measuring and Improving Cache Performance
  - ~~5.5 Dependable Memory Hierarchy~~
  - ~~5.6 Virtual Machines~~
- ☛ Lecture
  - 5.6 Virtual Memory
  - ~~5.8 A Common Framework for Memory Hierarchy~~
- Lecture 26
  - ~~5.9 Using a Finite-State Machine to Control a Simple Cache~~
  - ~~5.10 Parallelism and Memory Hierarchies: Cache Coherence~~
  - ~~5.11 Parallelism and Memory Hierarchy: Redundant Arrays of Inexpensive Disks~~
  - ~~5.12 Advanced Material: Implementing Cache Controllers~~
  - 5.13 Real Stuff: The ARM Cortex-A53 and Intel Core i7 Memory Hierarchies
  - ~~5.14 Going Faster: Cache Blocking and Matrix Multiply~~
  - ~~5.15 Fallacies and Pitfalls~~
  - 5.16 Concluding Remarks

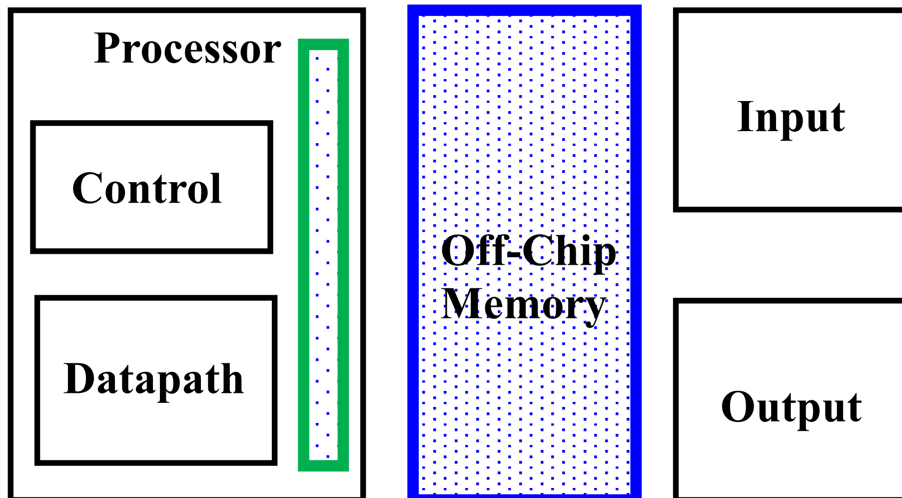
Green Boxes: Cache, on-chip, SRAM, fast, small, expensive
Blue Boxes: Main memory, off-chip, DRAM, slower, large not expensiv



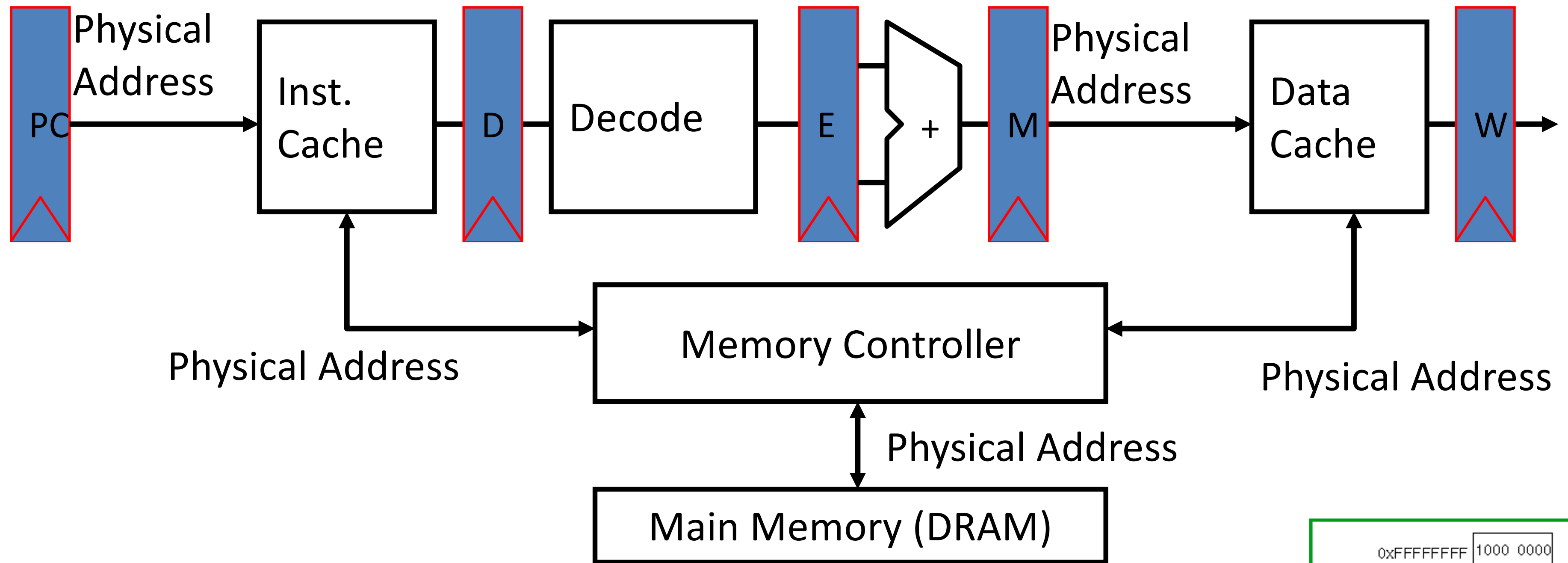


CPU is The chip.





Memory System of A Bare Machine




```
0x0FFE1230: add $t1, $t2, $t3
0x0FFE1234: lw|sw $t1, 32($t2)
0x0FFE1238: beq $t1, $t2, offset
```

- **In a bare machine,** the only kind of address is a physical address
  - The address of a memory byte

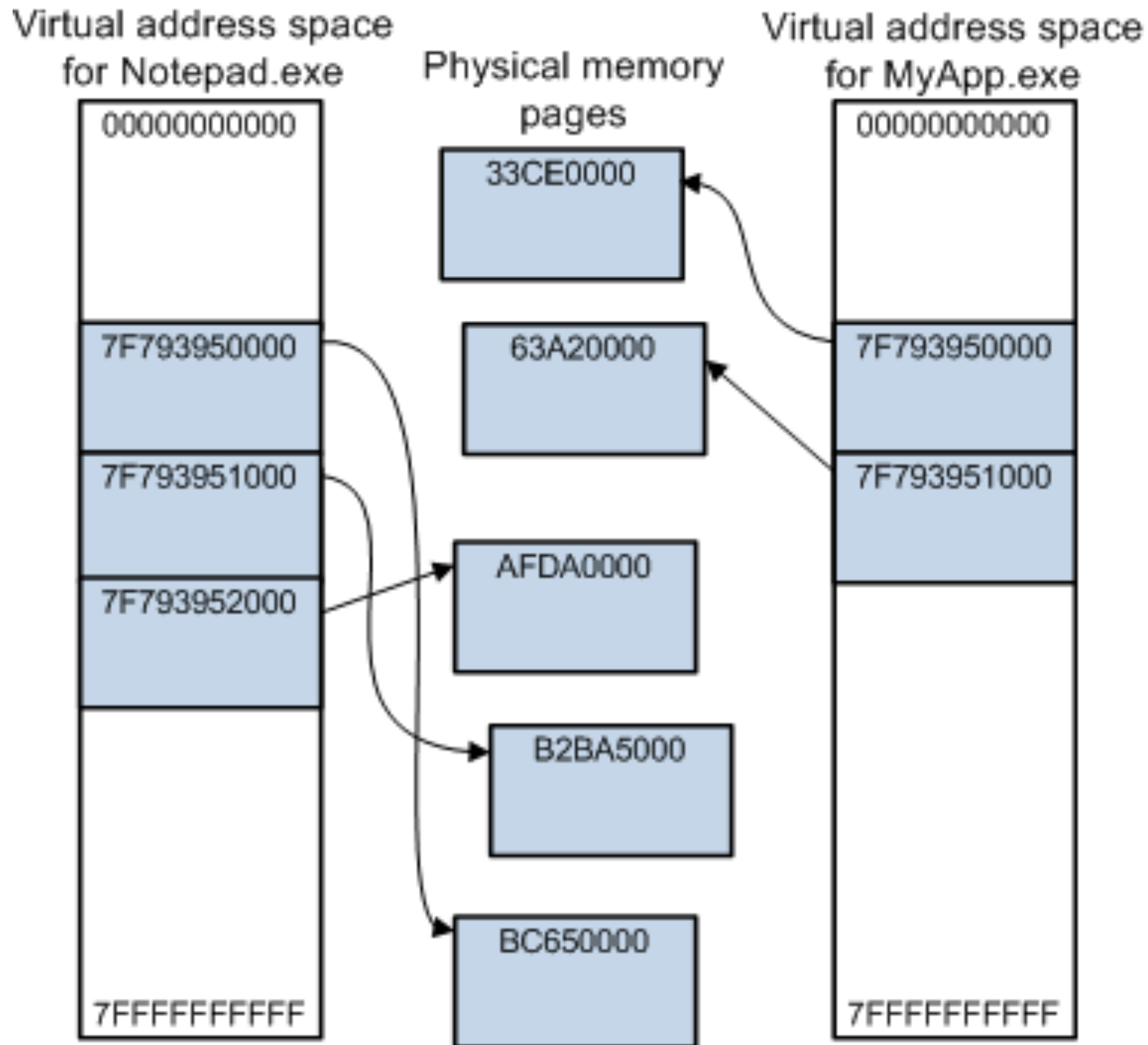
Addresses	Main Memory
0xFFFFFFFF	1000 0000
. . . . .	. . . . .
0x00000008	0100 1001
0x00000007	1100 1100
0x00000006	0110 1110
0x00000005	0110 1110
0x00000004	0000 0000
0x00000003	0110 1011
0x00000002	0101 0001
0x00000001	1100 1001
0x00000000	0100 1111

Main Memory

Virtual Memory

- Programs share main memory
  - Each gets a private virtual address space holding its frequently used code and data
  - Protected from other programs
- CPU and OS translate virtual addresses to physical addresses
  - VM "block" is called a page
  - VM translation "miss" is called a page fault
- Use main memory as a "cache" for secondary (disk) storage
  - Managed jointly by CPU hardware and the operating system (OS)

Virtual Memory to Physical Memory Mapping Example



Virtual Memory: Motivations and Benefits

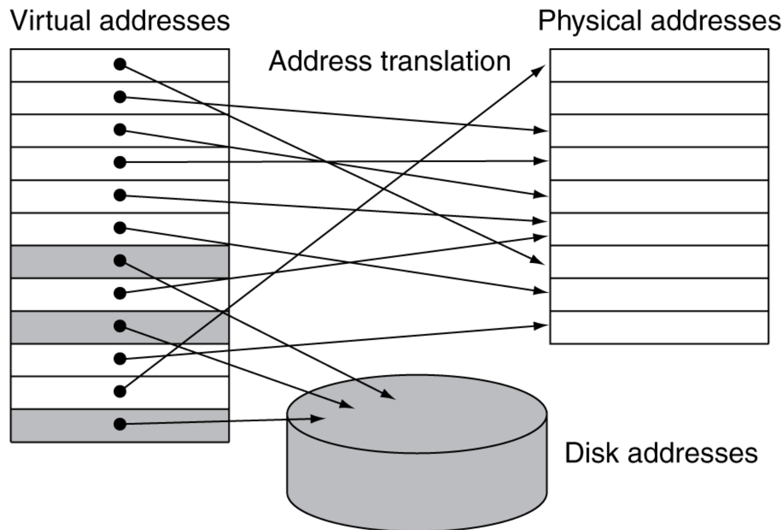
- **Protection**: to allow efficient and safe sharing of memory among multiple programs
  - Conventional multi-programming, time-sharing OS
  - Today for the memory needed by multiple virtual machines for cloud computing
- **Virtualization**: to remove the programming burdens of a small, limited amount of main memory.
  - 4G memory space of 32-bit OS/machine even with << 4GB physical memory, e.g. 256MB
    - Each sees 0x00000000 – 0xFFFFFFFF memory
- **Relocation**: simplifies loading the program for execution.
  - allows the same program to run in any location in physical memory.
- It is called **Virtual Memory**, thus **NOT REAL or PHYSICAL** memory

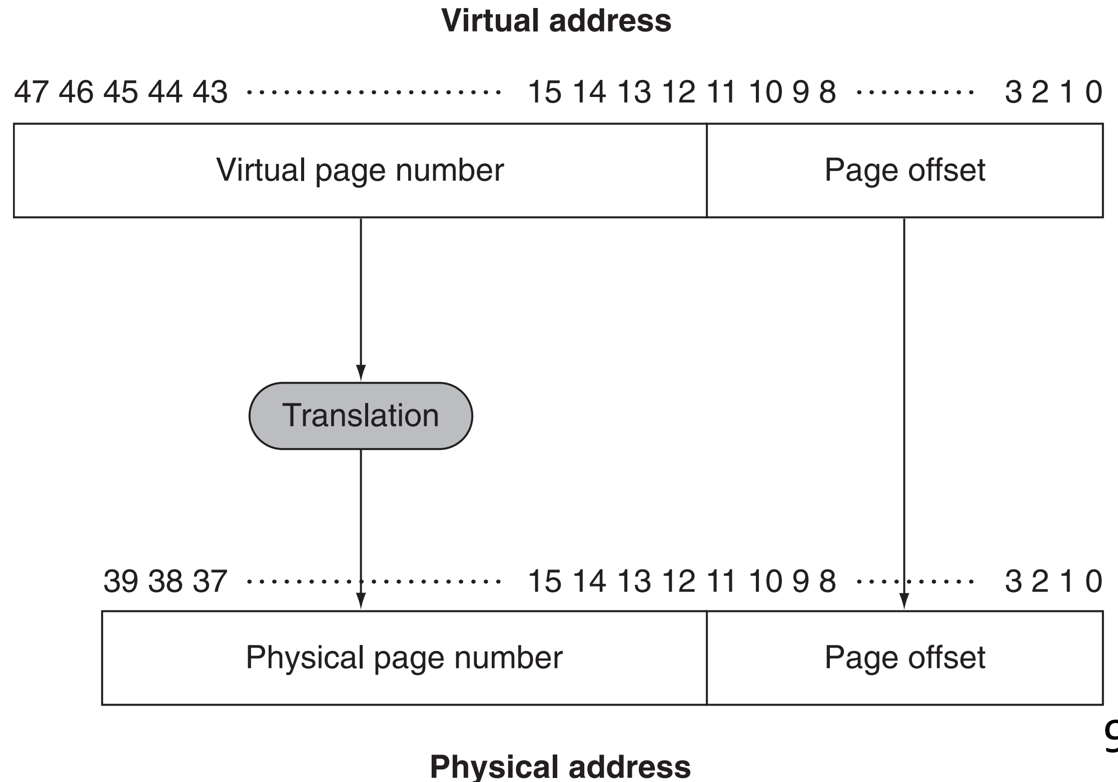
Address Translation

- Fixed-size pages (e.g., 4K)
  - 4K (2^{12}) byes per page
  - 12 bits to address a byte within a page

```
0x0FFE1230: add x1, x2, x3
0x0FFE1234: lw|sw x1, 32(x2)
0x0FFE1238: beq x1, x2, offset
```

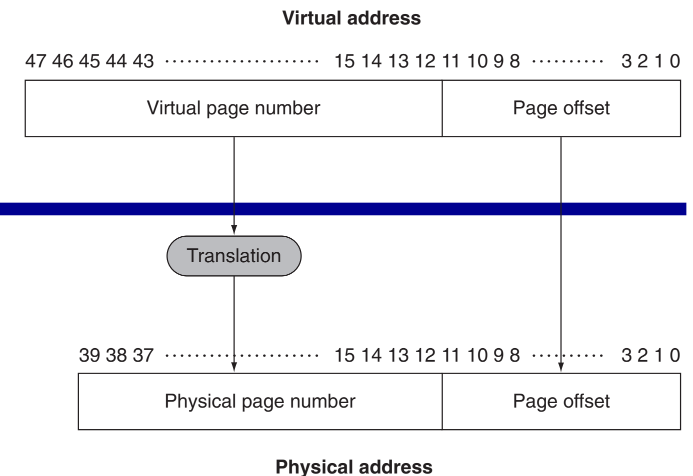
- Address translation: to map the upper [47:12] bits of the virtual address, i.e. virtual page number, to a physical page number [39:12]





Example: 0x0FFE1230: add x1, x2, x3 / 0x0FFE1234: lw|sw x1, 32(x2) / 0x0FFE1238: beq x1, x2, offset

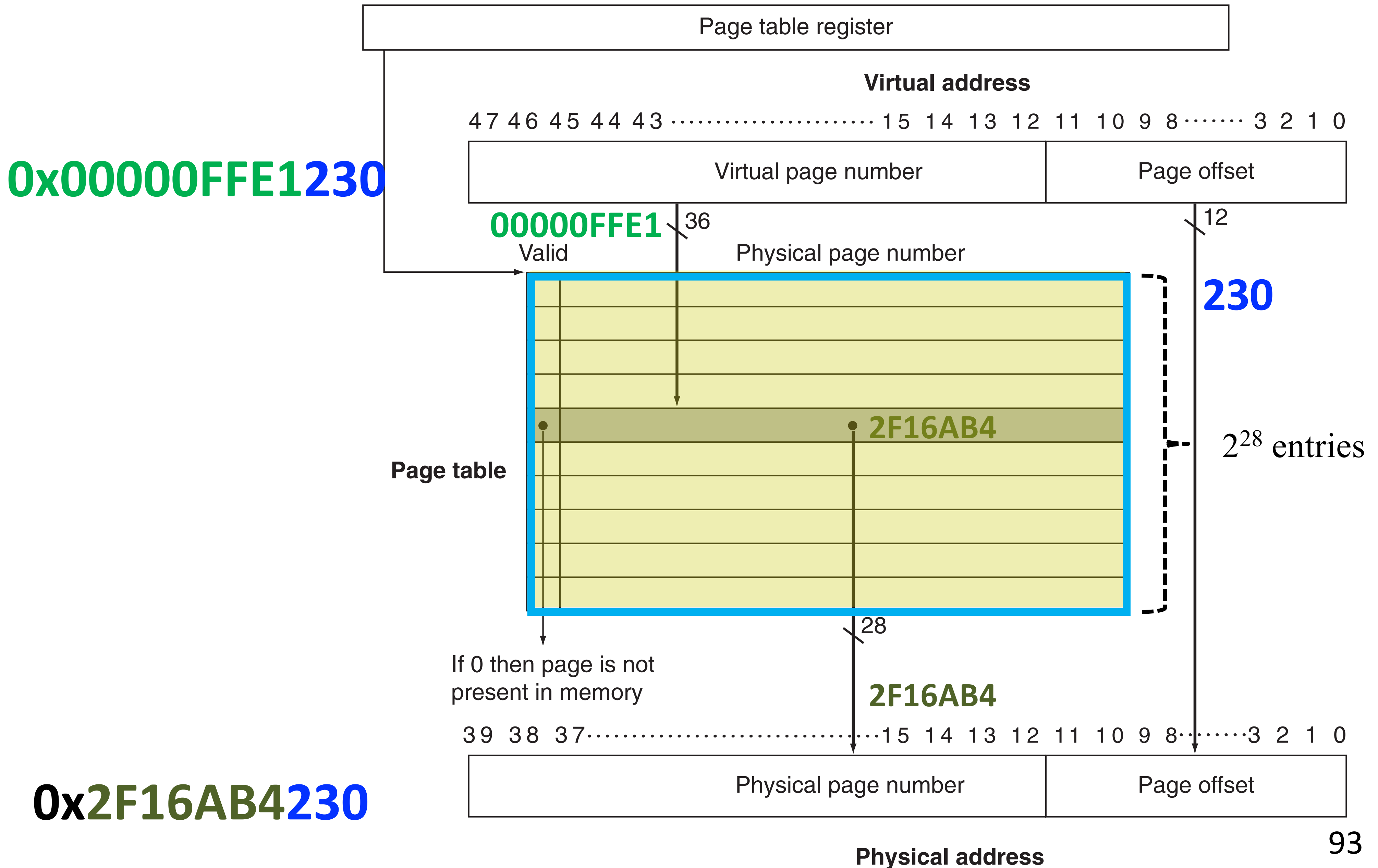
```
0x0FFE1230: add x1, x2, x3
0x0FFE1234: lw|sw x1, 32(x2)
0x0FFE1238: beq x1, x2, offset
```



The physical address of 0x0FFE1230

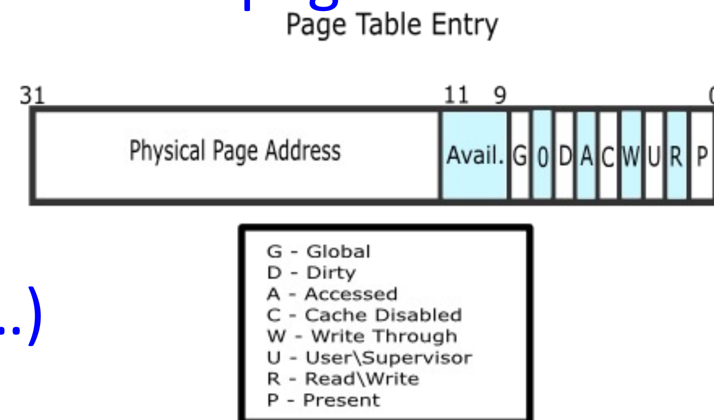
- A computer with 2^{40} (1024 Gbytes = 1 Tibytes) physical memory
  - Each process (the execution of a program) can access 2^{48} (4 TiB) bytes of address space, thus a virtual address has 48 bits
    - For 4K-byte (2^{12}) of pages, a process can have 2^{36} virtual pages
  - For 1 TiGbyte of physical memory, which is 2^{40} Bytes = 2^{28} pages
    - Thus a physical page number should have 28 bits
- Address translation
  - No need to translate the lower 12 bit (230) since it is for addressing a byte within a page, i.e. 0x00000FFE1230
  - Only need to translate 0x00000FFE1 (virtual page number) to physical page number in 28 bits, e.g. 0x2F16AB4
  - Thus the physical address is: 0x2F16AB4230

Translation Using a Page Table

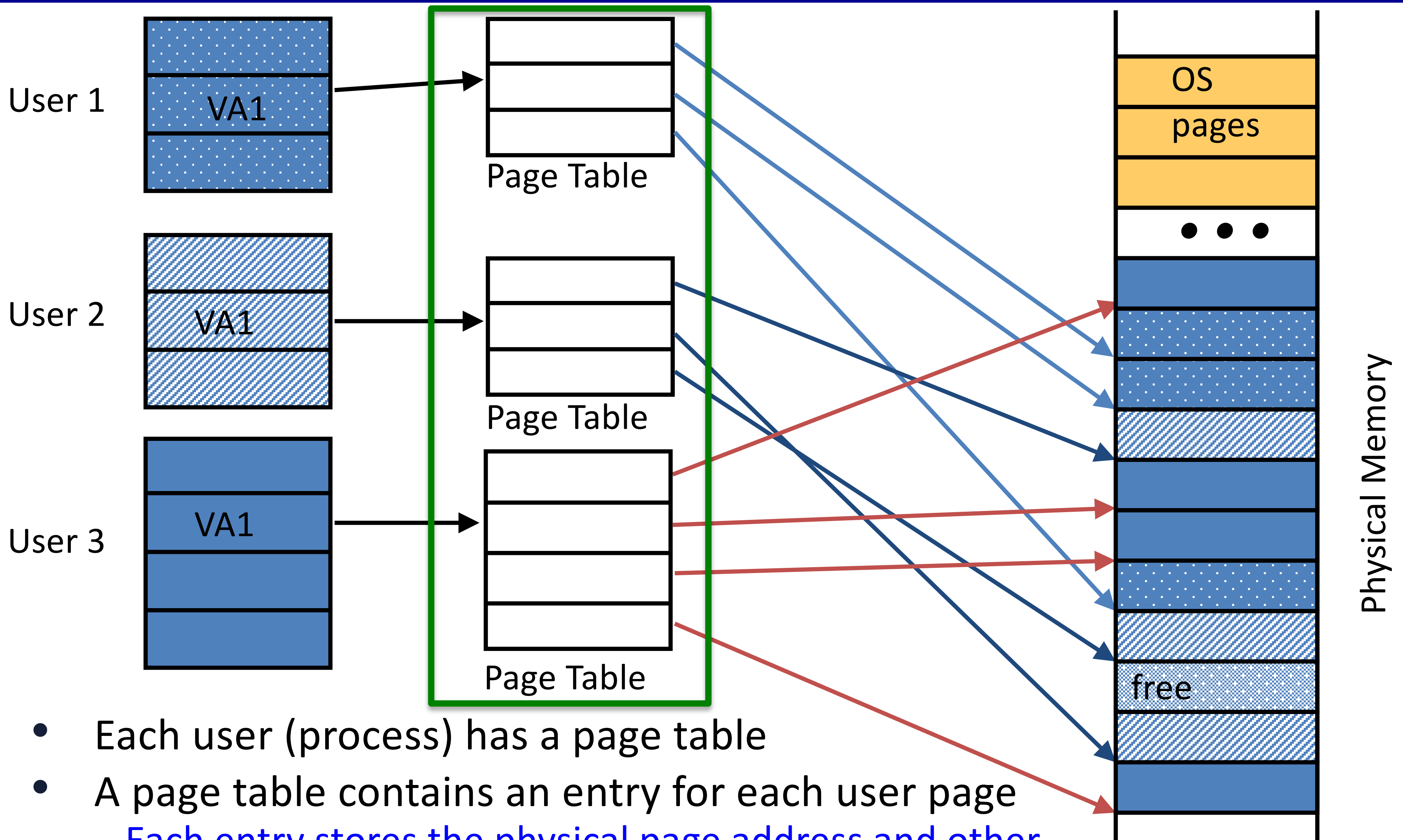


Page Tables for Address Translation

- **A page table is a (DRAM) memory area** that stores mapping information between virtual page number to physical page number
  - Array of page table entries, indexed by virtual page number
  - **Page table register** in CPU has the address of the page table in physical memory
- If page is present in memory
  - PTE stores the physical page number
  - Plus other status bits (referenced, dirty, …)
- If page is not present
  - PTE can refer to location in swap space on disk



Paging: Private Address Space per Process (User)



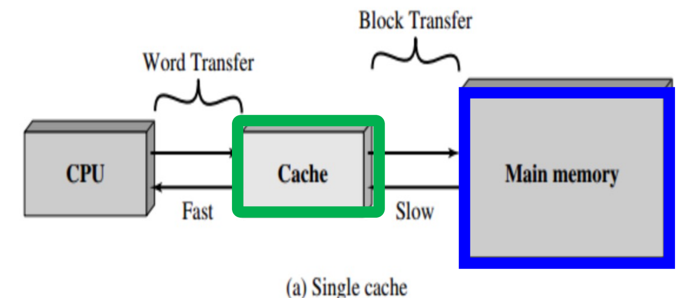
- Each user (process) has a page table
- A page table contains an entry for each user page
  - Each entry stores the physical page address and other info

Where Should Page Tables Reside?

- Space required by the page tables (PT) is proportional to the address space, number of users, ...

  $\Rightarrow$ *Too large to keep in registers or SRAM cache in full*

- Idea: Keep PTs in the main memory

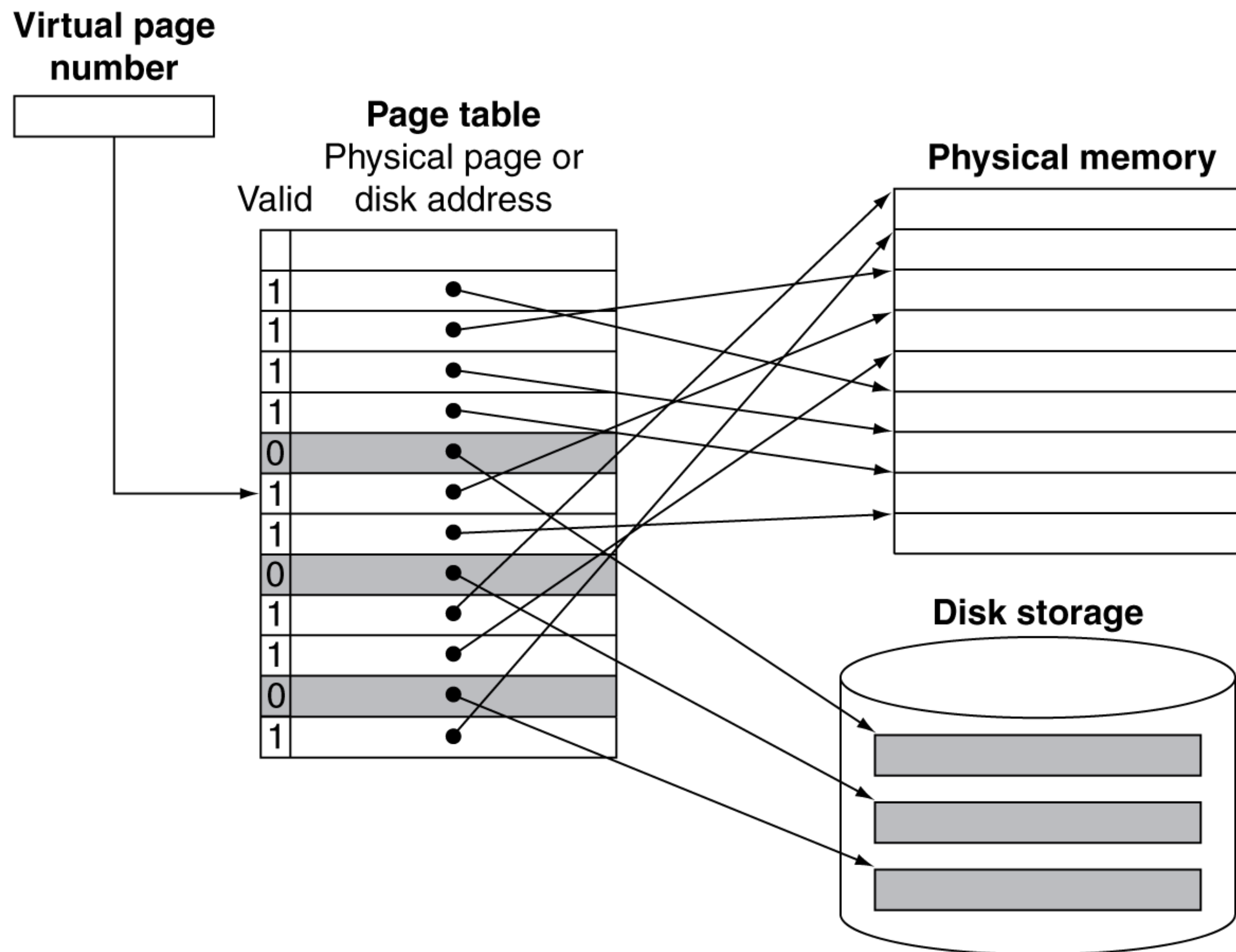


(a) Single cache

  - A page table register is used to store the address of the page table
  - Each user has her own page table

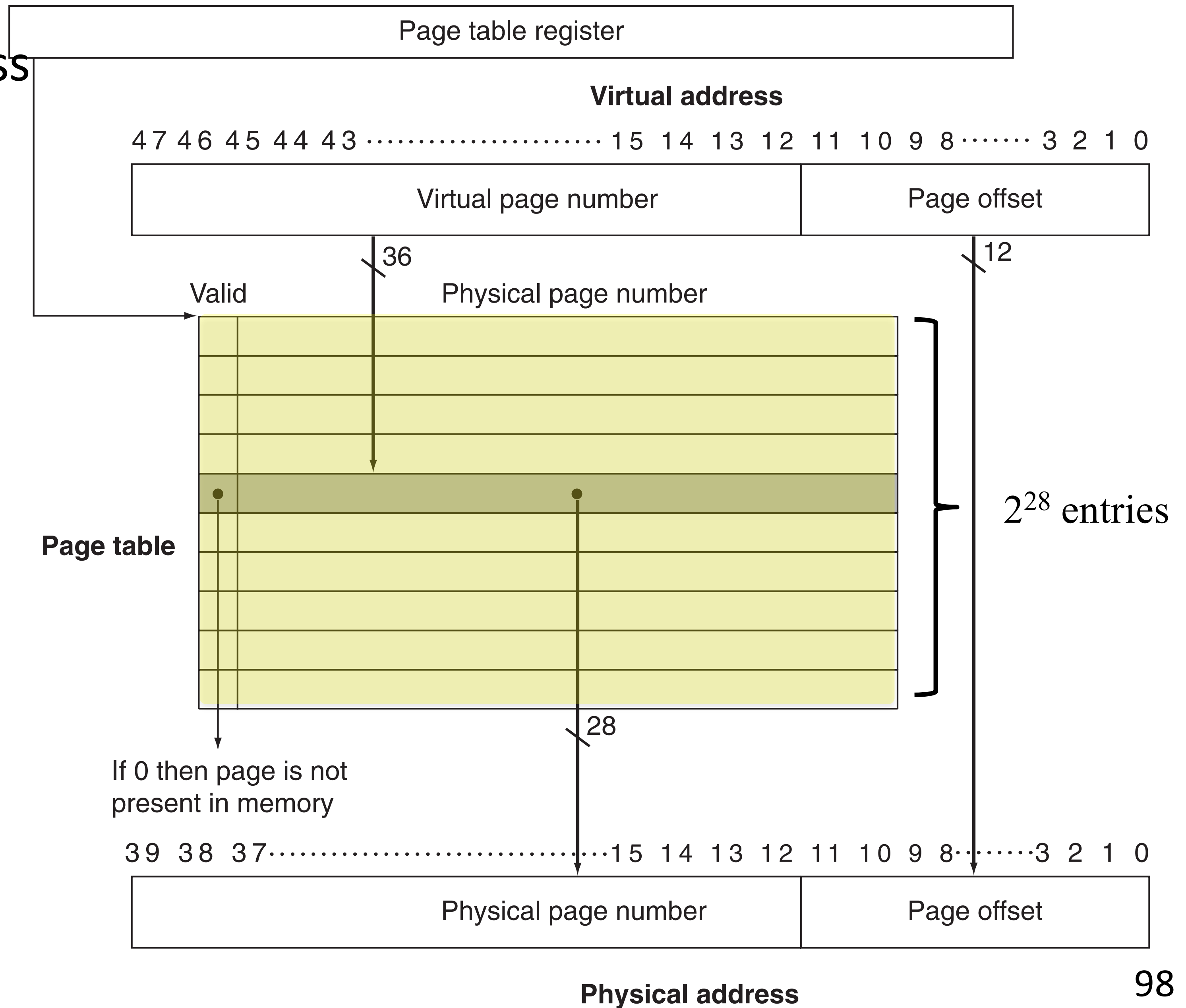
Mapping Pages to Storage

- Demand paging
  - Valid bit to indicate whether a page is in physical mem or not



Linear Page Table Example (4K pages)

- 48-bit address
- 4K-size page
- 4-byte PTE



Linear Page Table

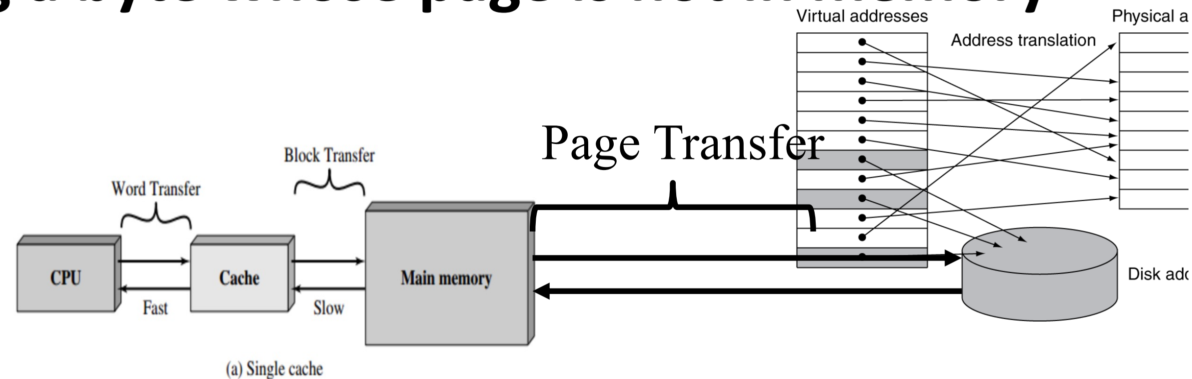
- With 48-bit virtual addresses, 4-KB pages & 4-byte PTEs, and 40-bit physical address
  - 2^{36} PTEs (48-12)
  - 4 TiB of swap needed to back up full virtual address space, in real, no need that much
- Larger pages?
  - Internal fragmentation (Not all memory in page is used)
  - Larger page fault penalty (more time to read from disk)
- What about the full 64-bit virtual address space???
  - How many page table entries (PTEs)?

Page Fault and Page Fault Penalty

- **Page Fault: when fetching a byte whose page is not in memory**
  - **Memory miss**

ld x10, 0x3540(x5)

Virtual address

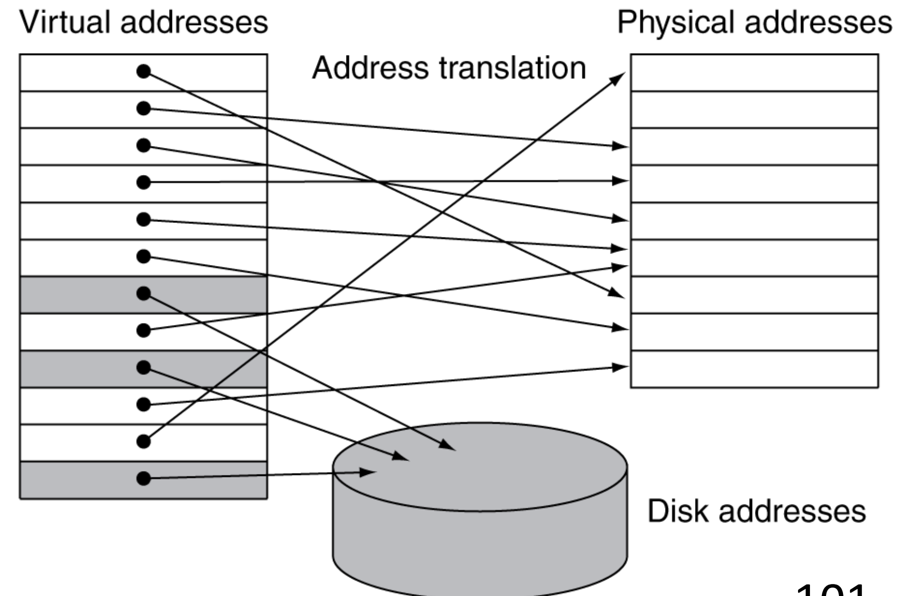


(a) Single cache

- On page fault, the page must be fetched from disk to memory
  - Takes millions of clock cycles
  - Handled by OS code, a process is swapped and context switched to another process
    - Different from cash miss, in which CPU stall to wait for memory access to be fulfilled.
- Try to minimize page fault rate
  - Fully associative placement
  - Smart replacement algorithms

Page Fault Handler

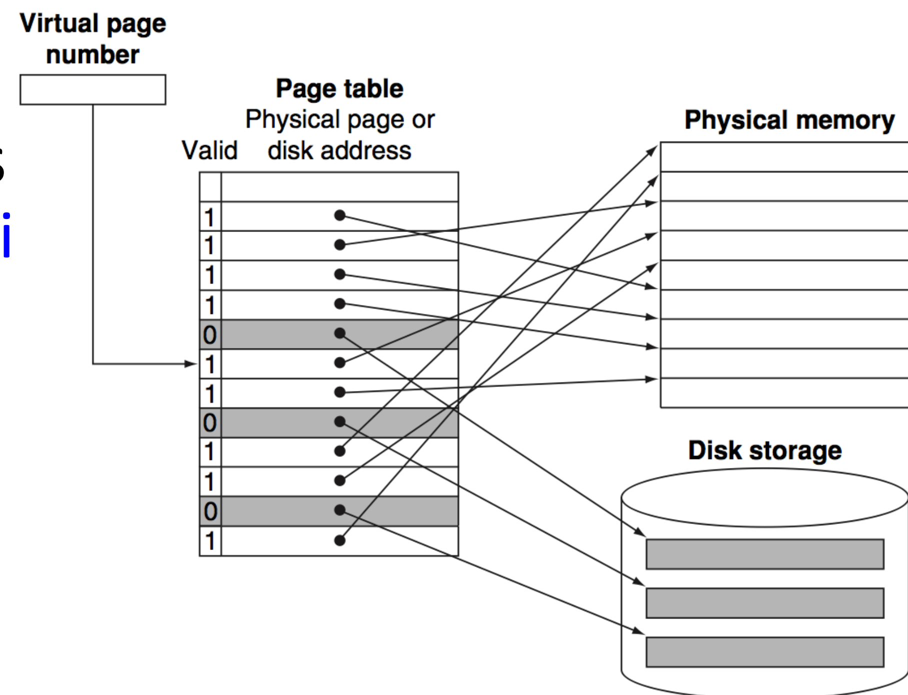
- Use faulting virtual address to find PTE
- Locate page on disk
- Choose page to replace
  - If dirty, write to disk first
- Read page into memory and update page table
- Make process runnable again
  - Restart from faulting instruction



Replacement and Writes

- To reduce page fault rate, prefer least-recently used (LRU) replacement
  - Reference bit (aka use bit) in PTE set to 1 on access to page
  - Periodically cleared to 0 by OS
  - A page with reference bit = 0 has not been used recently
  - **Work with principle of locality**

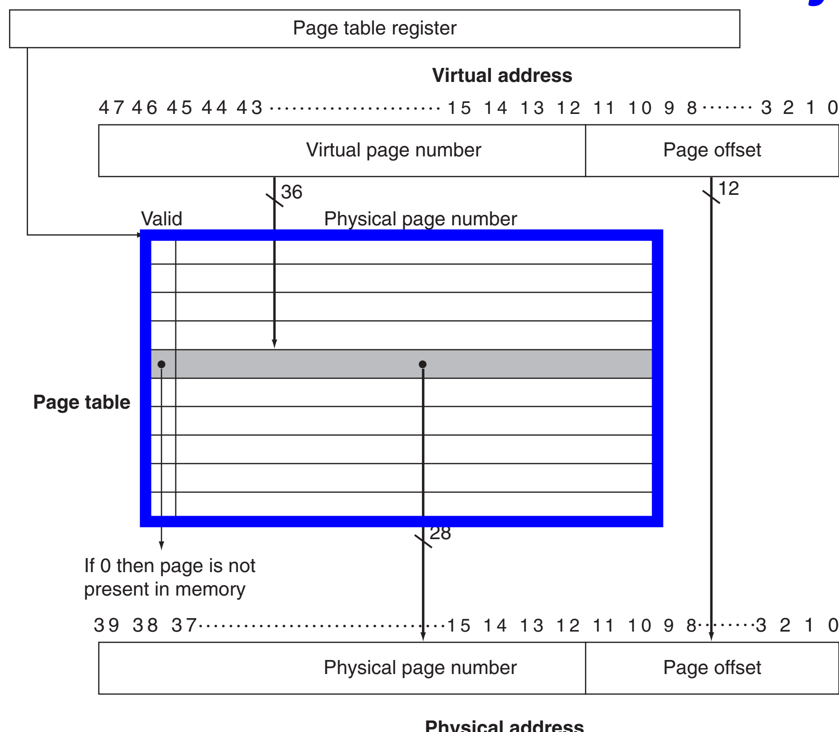
- Disk writes take millions of cycles
  - Block at once, not individual locati
  - Write through is impractical
  - Use write-back
  - Dirty bit in PTE set when page is written

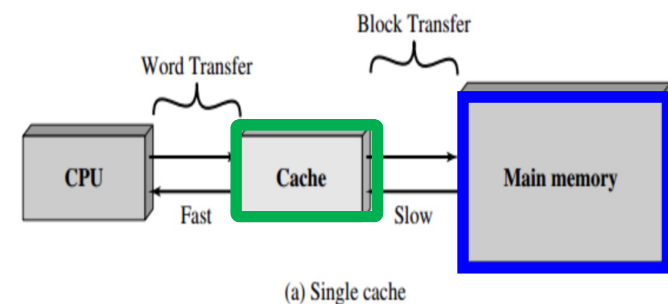


Page Table in Main Memory

```
0x0FFE1230: add x1, x2, x3
0x0FFE1234: lw|sw x1, 32(x2)
0x0FFE1238: beq x1, x2, offset
```

- One memory address needs:
  - One reference to access the page table for the physical page number and
  - Then another reference to access the data word
  - ➔ ***doubles the number of memory references/accesses!***





Fast Translation Using a TLB

TLB are cache (in SRAM) for page tables

- Address translation requires extra memory references
  - One to access the PTE
  - Then the actual memory access
- But access to page tables has good locality
  - So use a fast cache of PTEs within the CPU
  - Called a **Translation Look-aside Buffer (TLB)**
  - Typical: 16–512 PTEs, 0.5–1 cycle for hit, 10–100 cycles for miss, 0.01%–1% miss rate
  - Misses could be handled by hardware or software

Fast Translation Using a TLB

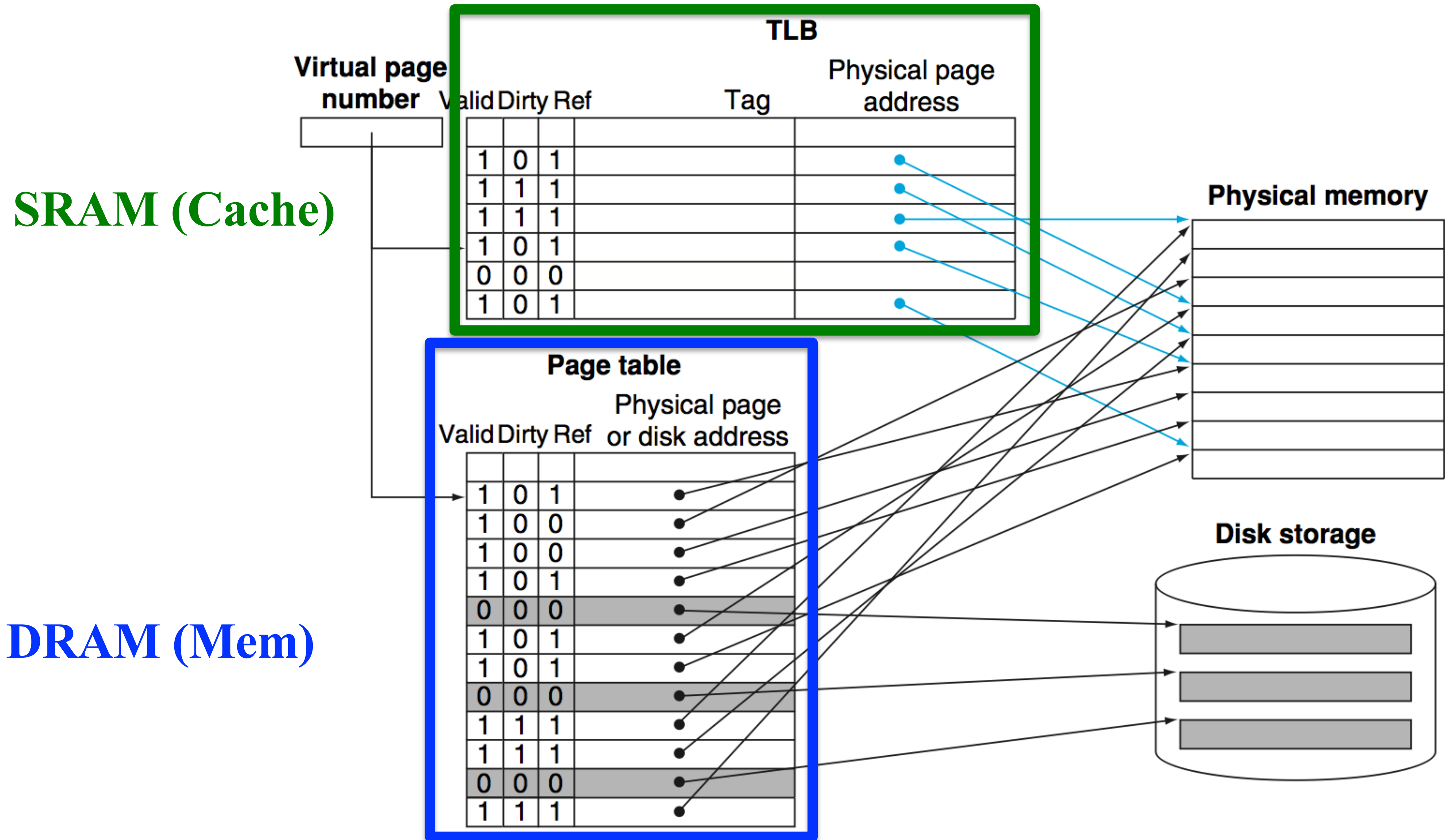


FIGURE 5.29 The TLB acts as a cache of the page table for the entries that map to physical pages only. The TLB contains a subset of the virtual-to-physical page mappings that are in the

TLB Misses

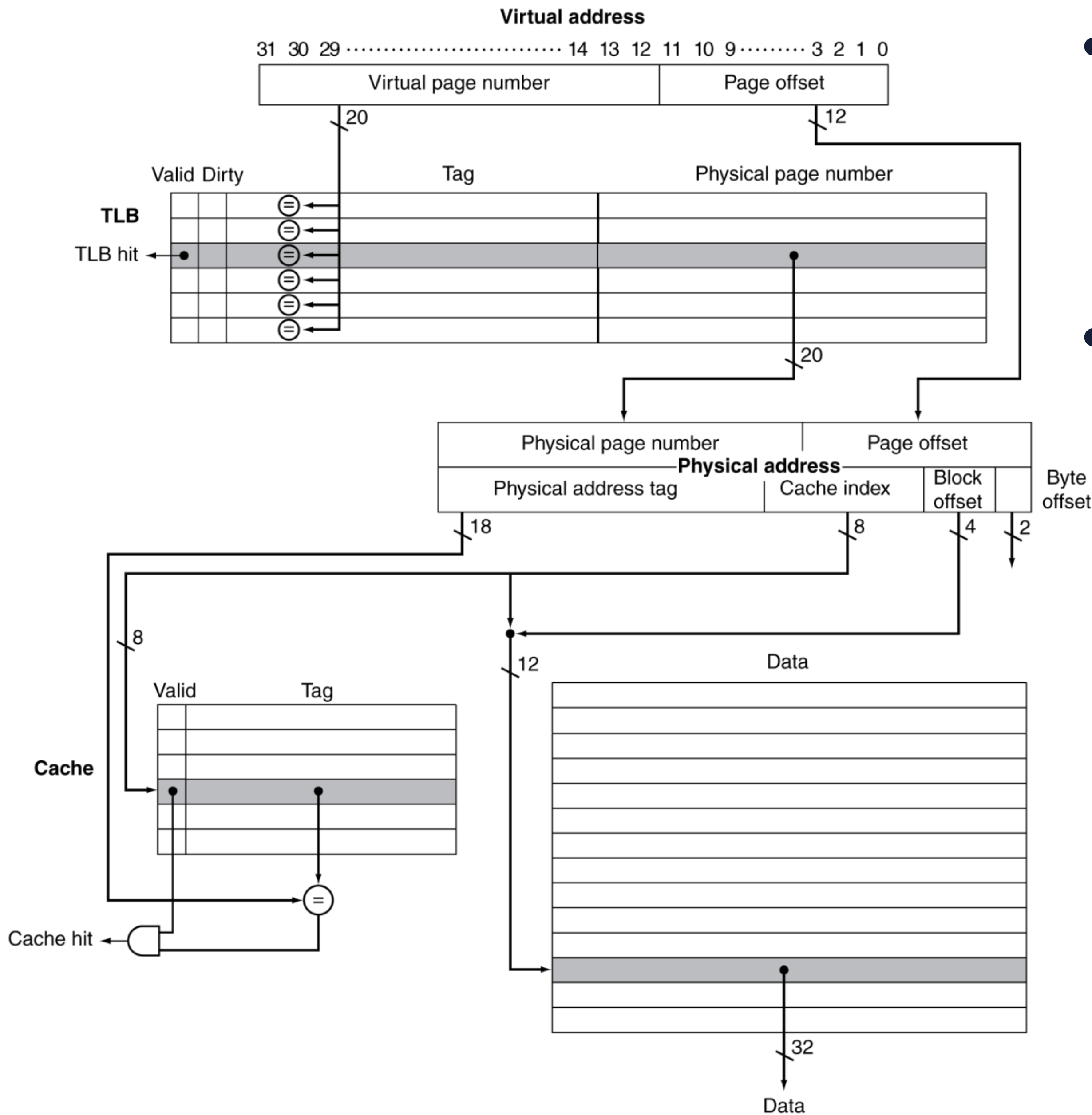
- If page is in memory
  - Load the PTE from memory and retry
  - Could be handled in hardware
    - Can get complex for more complicated page table structures
  - Or in software
    - Raise a special exception, with optimized handler
- If page is not in memory (page fault)
  - OS handles fetching the page and updating the page table
  - Then restart the faulting instruction

TLB Miss Handler

- TLB miss indicates
  - Page present, but PTE not in TLB
  - Page not present
- Must recognize TLB miss before destination register overwritten
  - Raise exception
- Handler copies PTE from memory to TLB
  - Then restarts instruction
  - If page not present, page fault will occur

TLB and Cache Interaction:

From VA to Byte via TLB and L-1 Cache



- If cache tag uses physical address
  - Need to translate before cache lookup
- Alternative: use virtual address tag
  - Complications due to aliasing
    - Different virtual addresses for shared physical address

From VA to Data via TLB and Cache

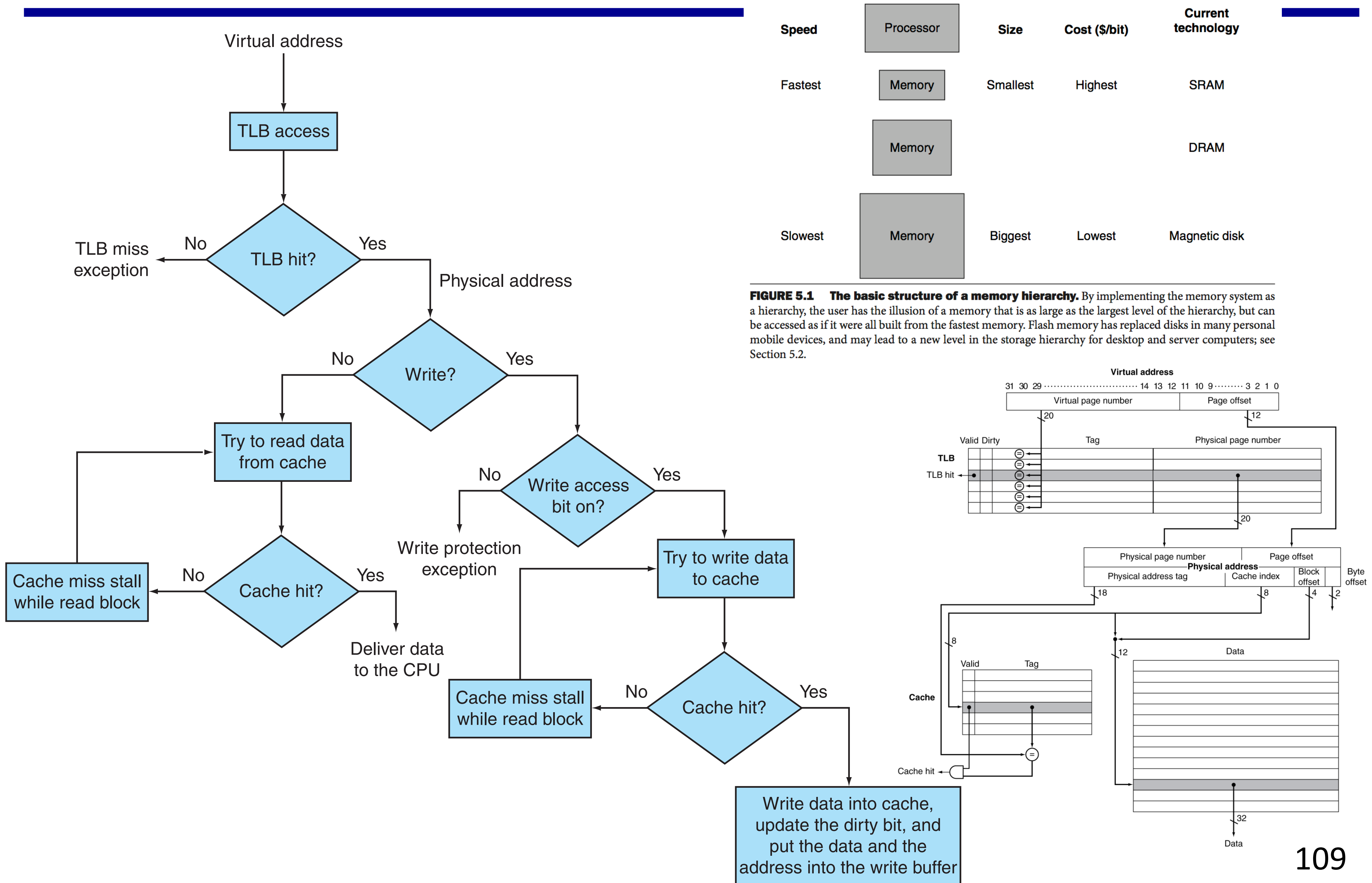
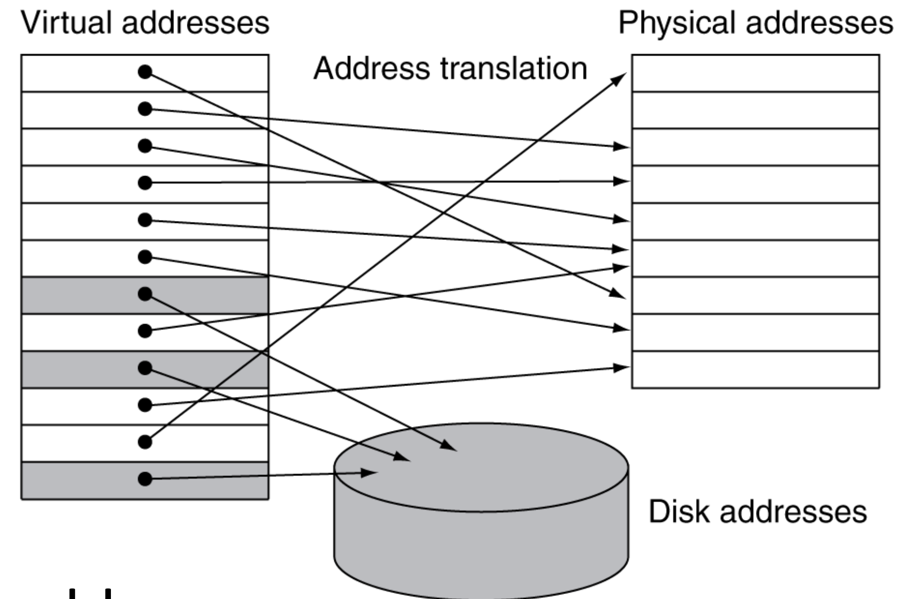


FIGURE 5.1 The basic structure of a memory hierarchy. By implementing the memory system as a hierarchy, the user has the illusion of a memory that is as large as the largest level of the hierarchy, but can be accessed as if it were all built from the fastest memory. Flash memory has replaced disks in many personal mobile devices, and may lead to a new level in the storage hierarchy for desktop and server computers; see Section 5.2.

VM: On-Demand Paging and Swap: Protection, Virtualization and Relocation

- Fixed-size pages (e.g., 4K)

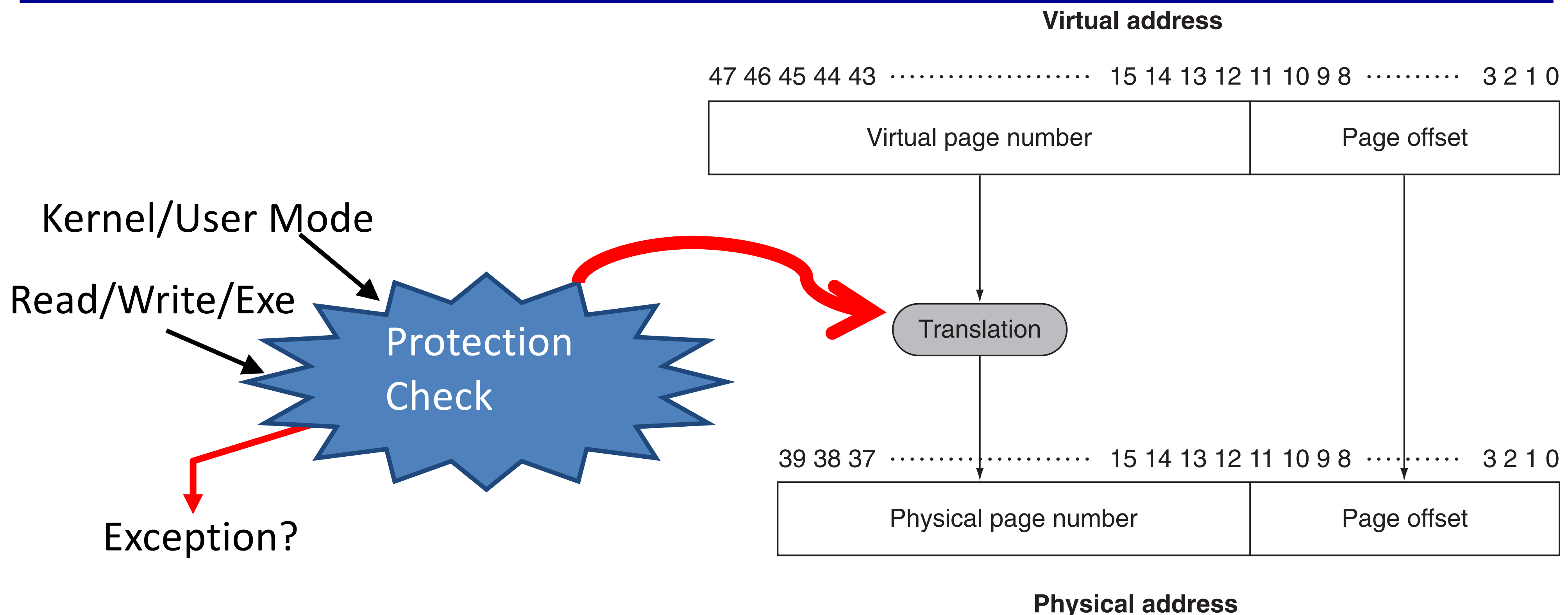


ld x10, 0x3540(x5)

Virtual address

- **Protection**: with multiple virtual address spaces, errors are confined to one address space
  - Between programs (processes)
- **Virtualization** via on-demand paging: move only frequently used pages to VM
  - Principle of locality
- **Relocation**: pages on disk can be loaded to any free physical pages

VM: Address Translation & Protection



- Every instruction and data access needs address translation and protection checks
  - Within a program: writes to EXE or Read-only segment are violations
- A good VM design needs to be fast (~ one cycle) and space efficient

Summary

- Virtual Memory:
  - Protection, Virtualization and Relocation

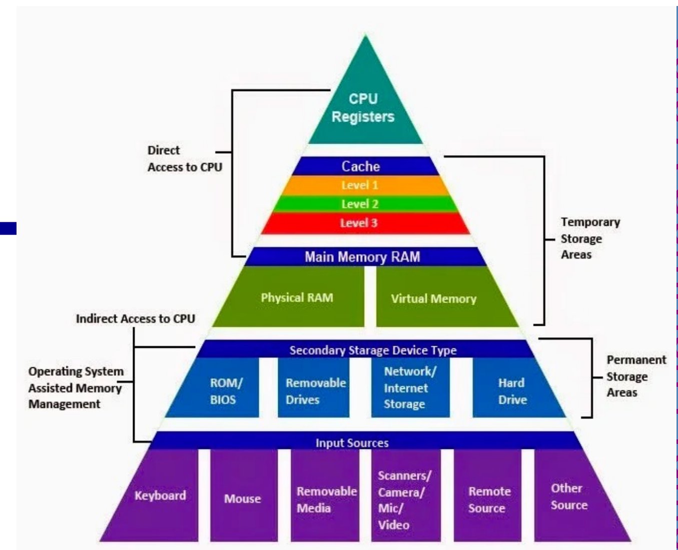
- Paging:
  - Page table, address translation
  - In main memory
- TLB:
  - Cache for page tables

Chapter 5: Large and Fast: Exploiting Memory Hierarchy

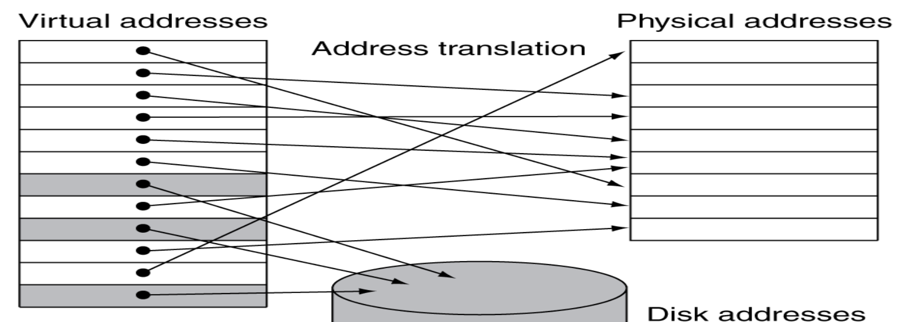
- Lecture
  - 5.1 Introduction
  - 5.2 Memory Technologies
- Lecture
  - 5.3 The Basics of Caches
- Lecture
  - 5.4 Measuring and Improving Cache Performance
  - ~~5.5 Dependable Memory Hierarchy~~
  - ~~5.6 Virtual Machines~~
- Lecture
  - 5.6 Virtual Memory
  - 5.8 A Common Framework for Memory Hierarchy
- Lecture 26
  - ~~5.9 Using a Finite-State Machine to Control a Simple Cache~~
  - ~~5.10 Parallelism and Memory Hierarchies: Cache Coherence~~
  - ~~5.11 Parallelism and Memory Hierarchy: Redundant Arrays of Inexpensive Disks~~
  - ~~5.12 Advanced Material: Implementing Cache Controllers~~
  - 5.13 Real Stuff: The ARM Cortex-A53 and Intel Core i7 Memory Hierarchies
  - ~~5.14 Going Faster: Cache Blocking and Matrix Multiply~~
  - ~~5.15 Fallacies and Pitfalls~~
  - 5.16 Concluding Remarks

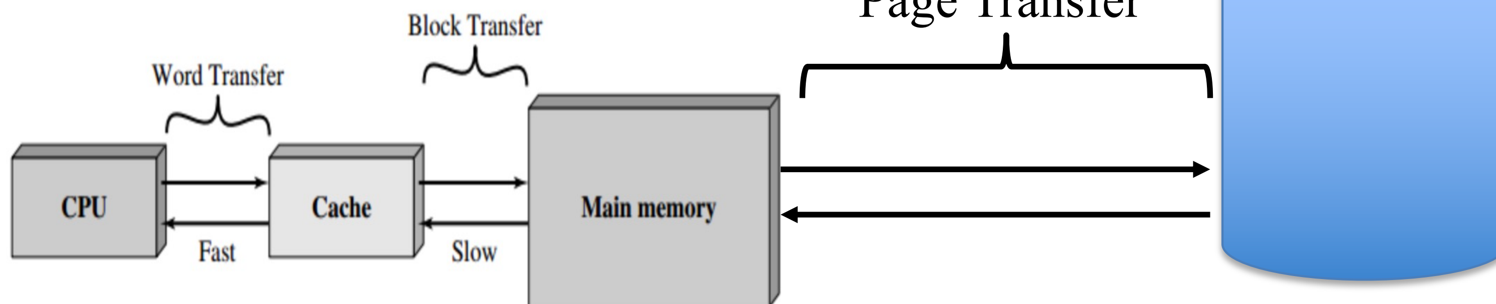
The Memory Hierarchy

1. Common principles apply at all levels of the memory hierarchy
   - Based on notions of caching and locality
     - Loading frequently used item and its surrounding in fast mem



- At each level in the hierarchy
  - Block placement
  - Finding a block
  - Replacement on a miss
  - Write policy



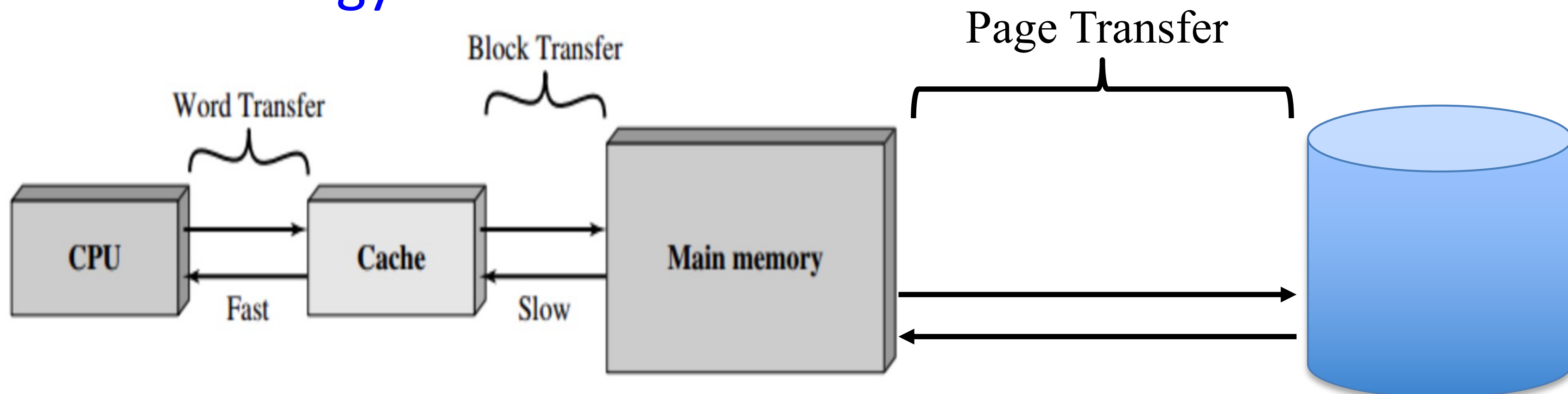


(a) Single cache

§5.8 A Common Framework for Memory Hierarchies

4 Questions for Each Level

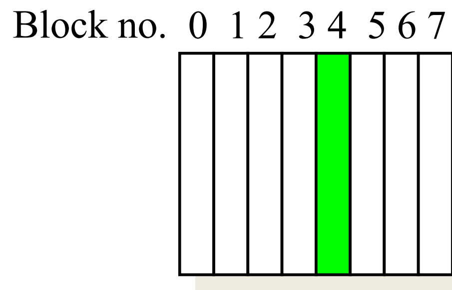
- Q1: Where can a block be placed in the upper level?
  - Block placement
- Q2: How is a block found if it is in the upper level?
  - Block identification
- Q3: Which block should be replaced on a miss?
  - Block replacement
- Q4: What happens on a write?
  - Write strategy



Q1: Where Can a Block be Placed in The Upper Level?

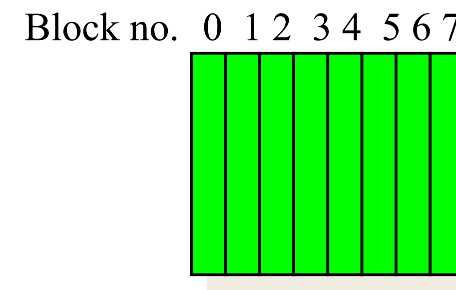
- Block Placement
  - Direct Mapped, Fully Associative, Set Associative
    - Direct mapped: (Block number) mod (Number of blocks in cache)
    - Set associative: (Block number) mod (Number of sets in cache)
      - # of set ≤ # of blocks
      - *n*-way: *n* blocks in a set
      - 1-way = direct mapped
    - Fully associative: # of set = 1

Direct mapped: data block 12 can go only into block 4 (12 mod 8)

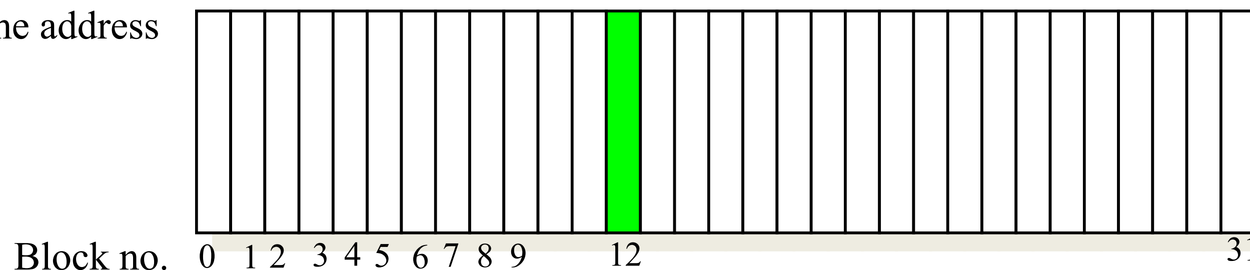


Set associative: data block 12 can go anywhere in set 0 (12 mod 4)

Fully associative: data block 12 can go anywhere

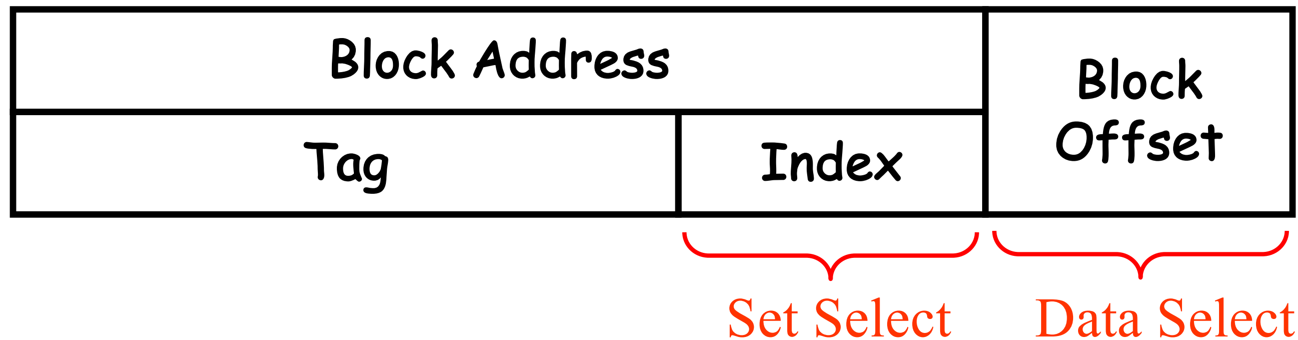


Block-frame address



Q2: Block Identification

- Tag on each block
  - No need to check index or block offset
- Increasing associativity shrinks index, expands tag



$$\text{Cache size} = \text{Associativity} \times 2^{\text{index_size}} \times 2^{\text{offest_size}}$$

Finding a Block

Associativity	Location method	Tag comparisons
Direct mapped	Index	1
n-way set associative	Set index, then search entries within the set	n
Fully associative	Search all entries	#entries
	Full lookup table	0

- Hardware caches
  - Reduce comparisons to reduce cost
- Virtual memory
  - Full table lookup makes full associativity feasible
  - Benefit in reduced miss rate

Q3: Which block should be replaced on a miss?

- Easy for Direct Mapped
- Set Associative or Fully Associative
  - Random
  - LRU (Least Recently Used)
  - First in, first out (FIFO)

	Associativity								
	Two-way			Four-way			Eight-way		
Size	**LRU**	**Random**	**FIFO**	**LRU**	**Random**	**FIFO**	**LRU**	**Random**	**FIFO**
16 KB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

Figure B.4 Data cache misses per 1000 instructions comparing least recently used, random, and first in, first out replacement for several sizes and associativities. There is little difference between LRU and random for the largest size cache, with LRU outperforming the others for smaller caches. FIFO generally outperforms random in the smaller cache sizes. These data were collected for a block size of 64 bytes for the Alpha architecture using 10 SPEC2000 benchmarks. Five are from SPECint2000 (gap, gcc, gzip, mcf, and perl) and five are from SPECfp2000 (applu, art, equake, lucas, and swim). We will use this computer and these benchmarks in most figures in this appendix.

Replacement

- Choice of entry to replace on a miss
  - Least recently used (LRU)
    - Complex and costly hardware for high associativity
  - Random
    - Close to LRU, easier to implement
- Virtual memory
  - LRU approximation with hardware support

Q4: What Happens on a Write?

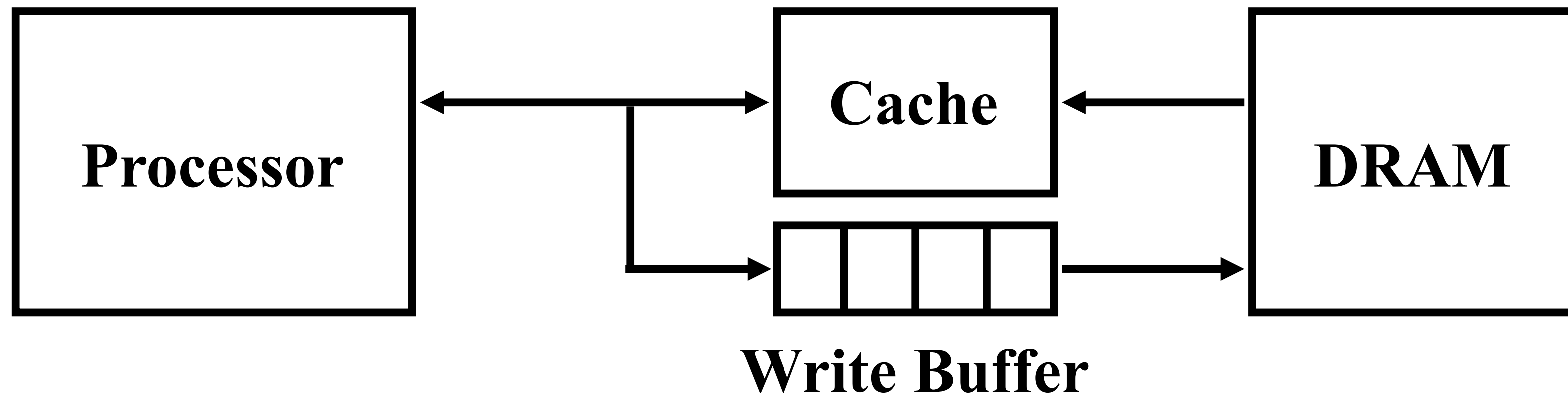
	Write-Through	Write-Back
Policy	Data written to cache block, also written to lower-level memory	1. Write data only to the cache 2. Update lower level when a block falls out of the cache
Debug	Easy	Hard
Do read misses produce writes?	No	Yes
Do repeated writes make it to lower level?	Yes	No

Additional option -- let writes to an un-cached address allocate a new cache line ("write-allocate").

Write Policy

- Write-through
  - Update both upper and lower levels
  - Simplifies replacement, but may require write buffer
- Write-back
  - Update upper level only
  - Update lower level when block is replaced
  - Need to keep more state
- Virtual memory
  - Only write-back is feasible, given disk write latency

Write Buffers for Write-Through Caches



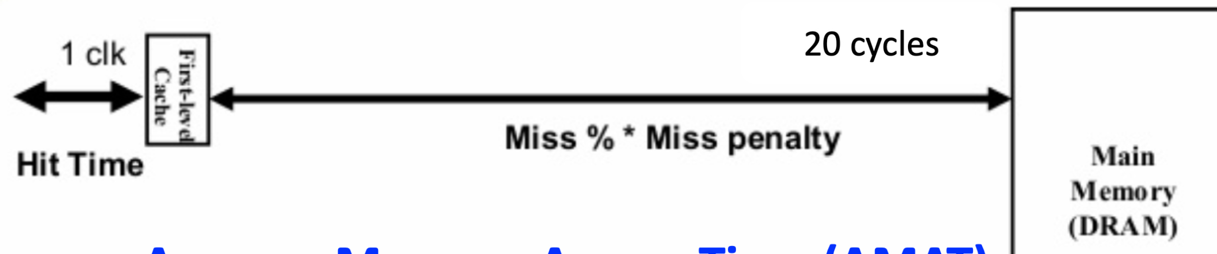
- Q. Why a write buffer ?
  - A. So CPU doesn't stall
- Q. Why a buffer, why not just one register ?
  - A. Bursts of writes are common.
- Q. Are Read After Write (RAW) hazards an issue for write buffer?
  - A. Yes! Drain buffer before next read, or send read 1st after check write buffers.

Sources of Misses

- Compulsory misses (aka cold start misses)
  - First access to a block
- Capacity misses
  - Due to finite cache size
  - A replaced block is later accessed again
- Conflict misses (aka collision misses)
  - In a non-fully associative cache
  - Due to competition for entries in a set
  - Would not occur in a fully associative cache of the same total size

Cache Design Trade-offs

Average Memory Access Time (AMAT)



Average Memory Access Time (AMAT)
= Hit Time + Miss Rate * Miss Penalty

Design change	Effect on miss rate	Negative performance effect
Increase cache size	Decrease capacity misses	May increase access time
Increase associativity	Decrease conflict misses	May increase access time
Increase block size	Decrease compulsory misses	Increases miss penalty. For very large block size, may increase miss rate due to pollution.

Multilevel On-Chip Caches

Characteristic	ARM Cortex-A53	Intel Core i7
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	Configurable 16 to 64 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	Two-way (I), four-way (D) set associative	Four-way (I), eight-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, variable allocation policies (default is Write-allocate)	Write-back, No-write-allocate
L1 hit time (load-use)	Two clock cycles	Four clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 2 MiB	256 KiB (0.25 MiB)
L2 cache associativity	16-way set associative	8-way set associative
L2 replacement	Approximated LRU	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	12 clock cycles	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, shared
L3 cache associativity	–	16-way set associative
L3 replacement	–	Approximated LRU
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	35 clock cycles

2-Level TLB Organization

Characteristic	ARM Cortex-A53	Intel Core i7
Virtual address	48 bits	48 bits
Physical address	40 bits	44 bits
Page size	Variable: 4, 16, 64 KiB, 1, 2 MiB, 1 GiB	Variable: 4 KiB, 2/4 MiB
TLB organization	1 TLB for instructions and 1 TLB for data per core Both micro TLBs are fully associative, with 10 entries, round robin replacement 64-entry, four-way set-associative TLBs TLB misses handled in hardware	1 TLB for instructions and 1 TLB for data per core Both L1 TLBs are four-way set associative, LRU replacement L1 I-TLB has 128 entries for small pages, seven per thread for large pages L1 D-TLB has 64 entries for small pages, 32 for large pages The L2 TLB is four-way set associative, LRU replacement The L2 TLB has 512 entries TLB misses handled in hardware

Supporting Multiple Issue

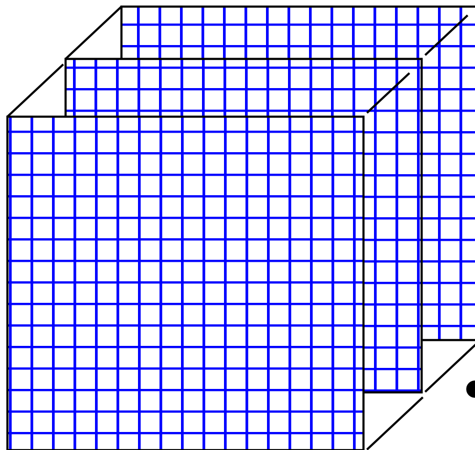
- Both have multi-banked caches that allow multiple accesses per cycle assuming no bank conflicts
- Core i7 cache optimizations
  - Return requested word first
  - Non-blocking cache
    - Hit under miss
    - Miss under miss
  - Data prefetching

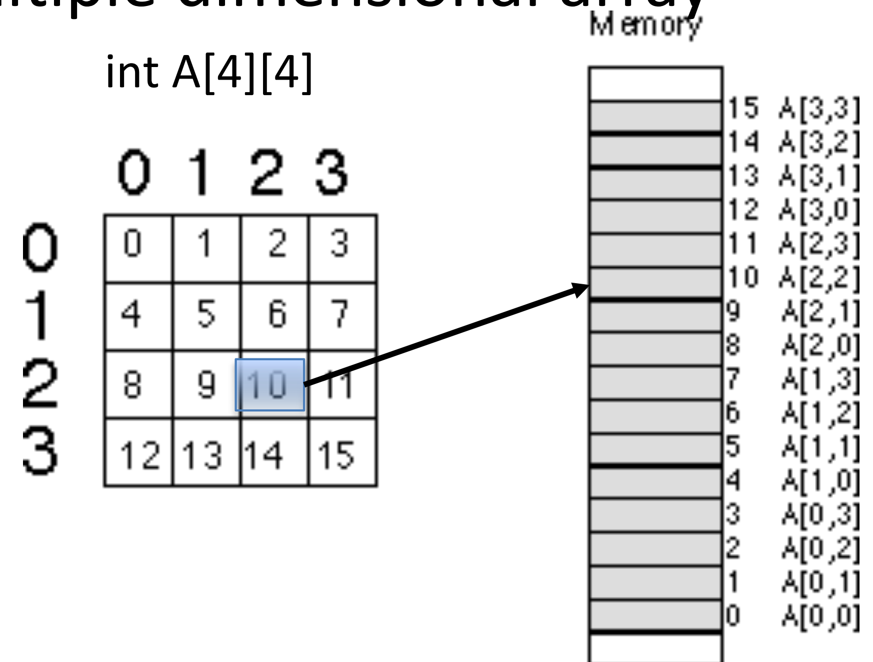
Concluding Remarks

- Fast memories are small, large memories are slow
  - We really want fast, large memories ☹
  - Caching gives this illusion ☺
- Principle of locality
  - Programs use a small part of their memory space frequently
- Memory hierarchy
  - L1 cache ↔ L2 cache ↔ … ↔ DRAM memory ↔ disk
- Memory system design is critical for multiprocessors

Vector/Matrix and Array in C

- C has row-major storage for multiple dimensional array
  - A[2,2] is followed by A[2,3]
- 3-dimensional array
  - B[3][100][100]





- **Stepping through columns in one row:**
  for (i=0; i<4; i++) sum += A[0][i];
  accesses successive elements
- **Stepping through rows in one column:**
  for (i=0; i<4; i++) sum += A[i][0];
  Stride-4 access

Locality Example

- **Claim:** **Being able to look at code and *get qualitative sense* of its locality is key skill for professional programmer**

- Question: Does this function have good locality?

```
int sumarrayrows(int a[M][N]){
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Locality Example

- Question: Does this function have good locality?

```
int sumarraycols(int a[M][N]){
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Locality Example

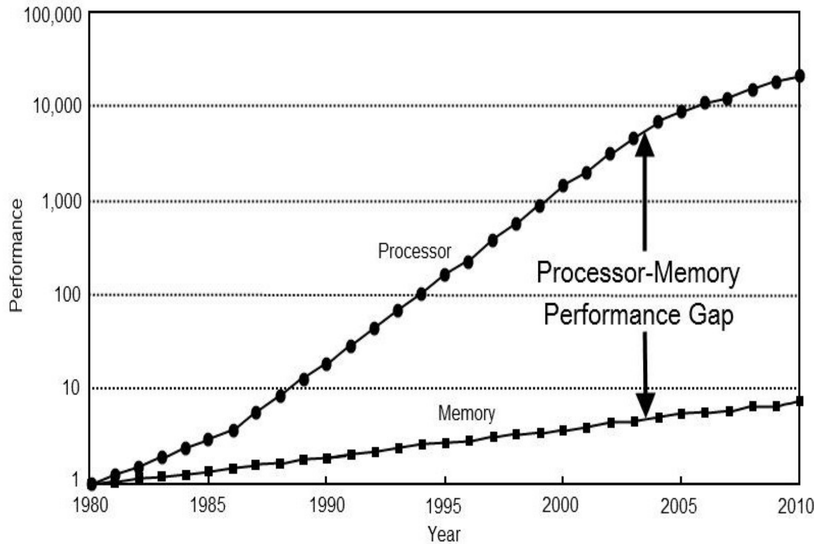
- Question: Can you permute the loops so that the function scans the 3-d array `a[]` with a stride-1 reference pattern (and thus has good spatial locality)?

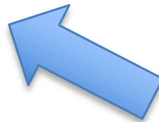
```
int sumarray3d(int a[M][N][N]) {
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < M; k++)
                sum += a[k][i][j];
    return sum;
}
```

Review: Memory Hierarchy

CPU-Mem Performance Gap: *Memory Wall*





Program Behavior: *Principle of Locality*

- Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves
- **Spatial locality:** Items with nearby addresses tend to be referenced close together in time
- **Temporal locality:** Recently referenced items are likely to be referenced in the near future

```
sum = 0;
for(i=0; i<n; i++)
  sum += a[i];
return sum;
```

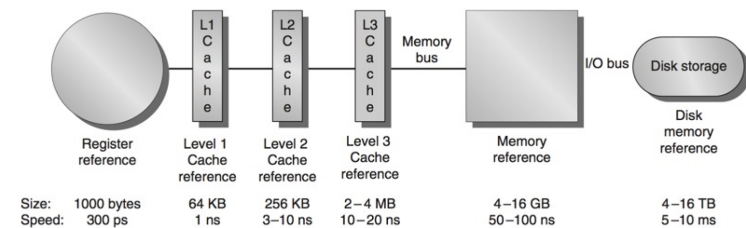
- **Data**
  - Reference array elements in succession (stride-1 reference pattern): Spatial Locality
  - Reference sum each iteration: Temporal Locality
- **Instructions**
  - Reference instructions in sequence: Spatial Locality
  - Cycle through loop repeatedly: Temporal Locality

Locality-Friendly Code: *Locality to Work With Memory Hierarchy*

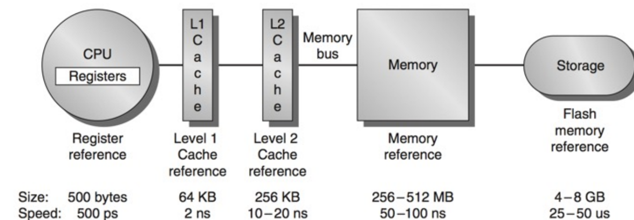
```
int sumarrayrows(int a[M][N]){
  int i, j, sum = 0;

  for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
      sum += a[i][j];
  return sum;
}
```

Architecture Approach: *Memory Hierarchy*



(a) Memory hierarchy for server



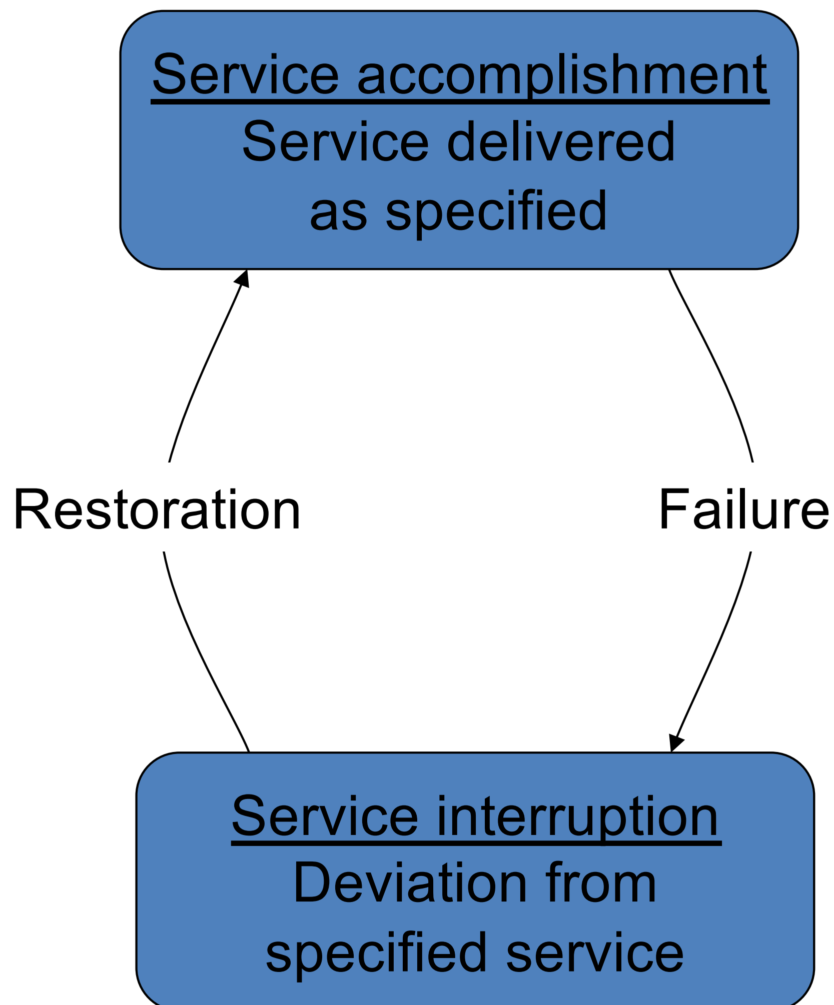
(b) Memory hierarchy for a personal mobile device

Slides for Other Sections of the Chapter

Disk Performance Issues

- Manufacturers quote average seek time
  - Based on all possible seeks
  - Locality and OS scheduling lead to smaller actual average seek times
- Smart disk controller allocate physical sectors on disk
  - Present logical sector interface to host
  - SCSI, ATA, SATA
- Disk drives include caches
  - Prefetch sectors in anticipation of access
  - Avoid seek and rotational delay

Dependability



- Fault: failure of a component
  - May or may not lead to system failure

Dependability Measures

- Reliability: mean time to failure (MTTF)
- Service interruption: mean time to repair (MTTR)
- Mean time between failures
  - MTBF = MTTF + MTTR
- Availability = MTTF / (MTTF + MTTR)
- Improving Availability
  - Increase MTTF: fault avoidance, fault tolerance, fault forecasting
  - Reduce MTTR: improved tools and processes for diagnosis and repair

The Hamming SEC Code

- Hamming distance
  - Number of bits that are different between two bit patterns
- Minimum distance = 2 provides single bit error detection
  - E.g. parity code
- Minimum distance = 3 provides single error correction, 2 bit error detection

Encoding SEC

- To calculate Hamming code:
  - Number bits from 1 on the left
  - All bit positions that are a power 2 are parity bits
  - Each parity bit checks certain data bits:

Bit position		1	2	3	4	5	6	7	8	9	10	11	12
Encoded date bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverate	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

Decoding SEC

- Value of parity bits indicates which bits are in error
  - Use numbering from encoding procedure
  - E.g.
    - Parity bits = 0000 indicates no error
    - Parity bits = 1010 indicates bit 10 was flipped

SEC/DEC Code

- Add an additional parity bit for the whole word (p_n)
- Make Hamming distance = 4
- Decoding:
  - Let H = SEC parity bits
    - H even, p_n even, no error
    - H odd, p_n odd, correctable single bit error
    - H even, p_n odd, error in p_n bit
    - H odd, p_n even, double error occurred
- Note: ECC DRAM uses SEC/DEC with 8 bits protecting each 64 bits

Virtual Machines

- Host computer emulates guest operating system and machine resources
  - Improved isolation of multiple guests
  - Avoids security and reliability problems
  - Aids sharing of resources
- Virtualization has some performance impact
  - Feasible with modern high-performance comptuers
- Examples
  - IBM VM/370 (1970s technology!)
  - VMWare
  - Microsoft Virtual PC

Virtual Machine Monitor

- Maps virtual resources to physical resources
  - Memory, I/O devices, CPUs
- Guest code runs on native machine in user mode
  - Traps to VMM on privileged instructions and access to protected resources
- Guest OS may be different from host OS
- VMM handles real I/O devices
  - Emulates generic virtual I/O devices for guest

Example: Timer Virtualization

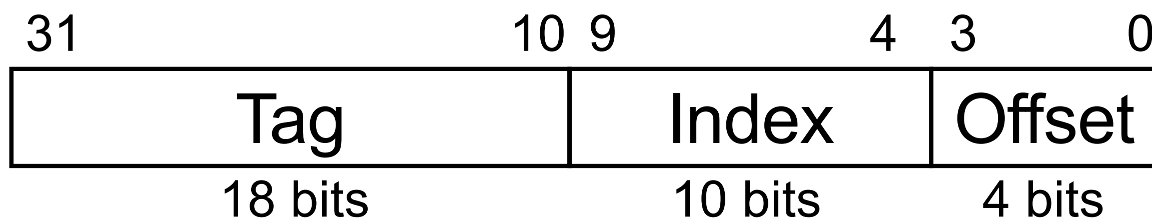
- In native machine, on timer interrupt
  - OS suspends current process, handles interrupt, selects and resumes next process
- With Virtual Machine Monitor
  - VMM suspends current VM, handles interrupt, selects and resumes next VM
- If a VM requires timer interrupts
  - VMM emulates a virtual timer
  - Emulates interrupt for VM when physical timer interrupt occurs

Instruction Set Support

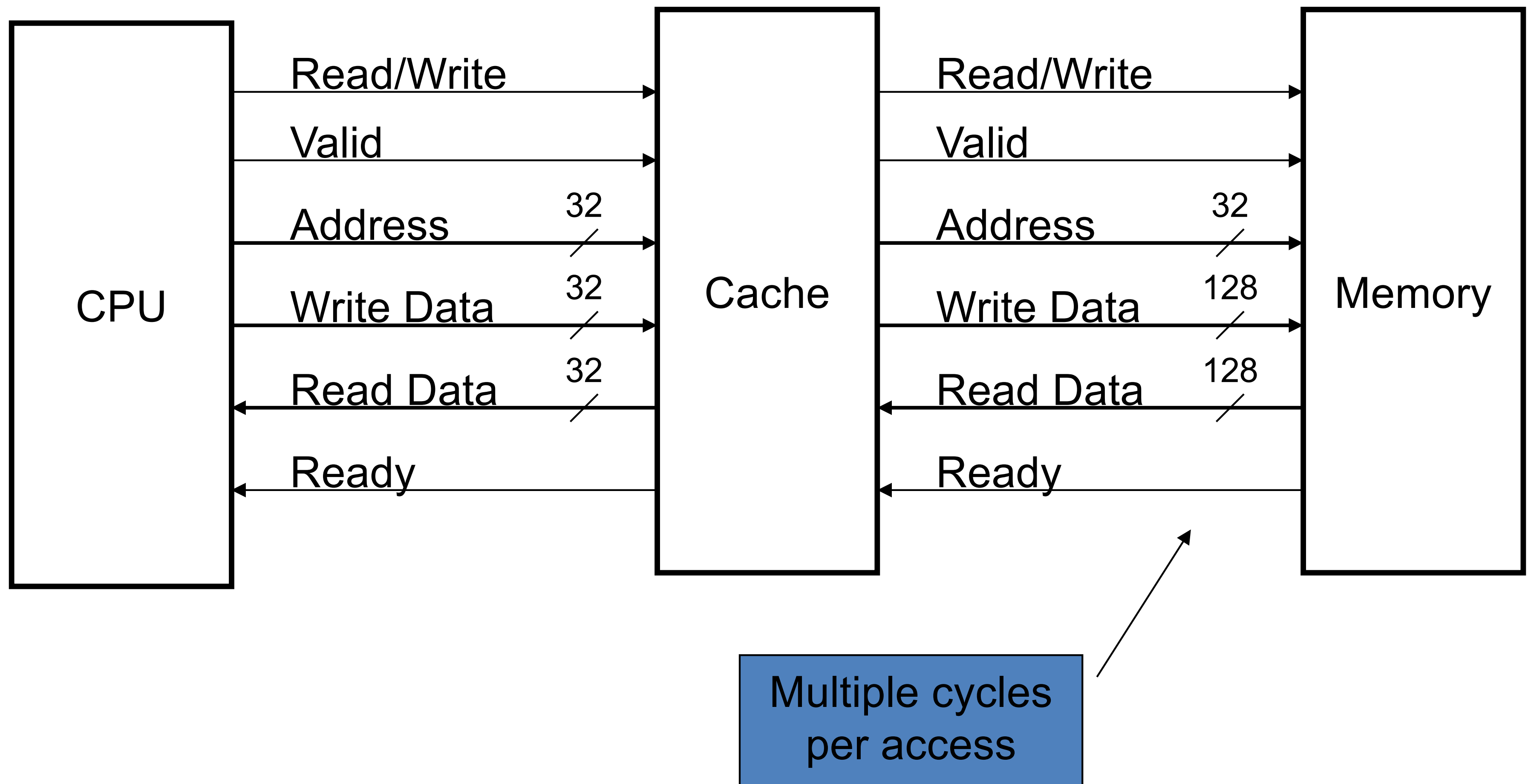
- User and System modes
- Privileged instructions only available in system mode
  - Trap to system if executed in user mode
- All physical resources only accessible using privileged instructions
  - Including page tables, interrupt controls, I/O registers
- Renaissance of virtualization support
  - Current ISAs (e.g., x86) adapting

Cache Control

- Example cache characteristics
  - Direct-mapped, write-back, write allocate
  - Block size: 4 words (16 bytes)
  - Cache size: 16 KB (1024 blocks)
  - 32-bit byte addresses
  - Valid bit and dirty bit per block
  - Blocking cache
    - CPU waits until access is complete

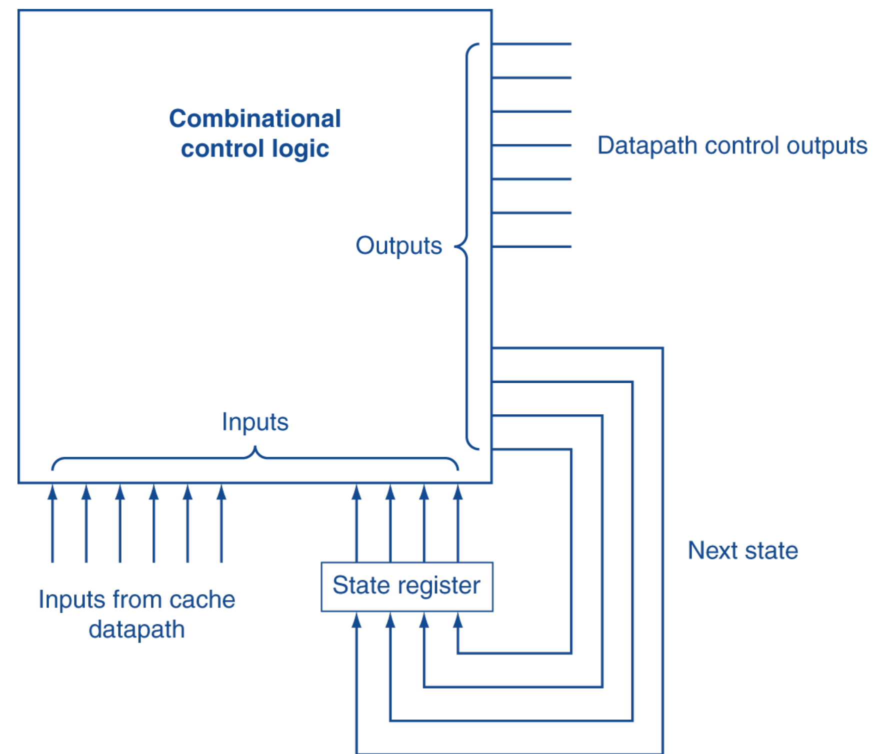


Interface Signals

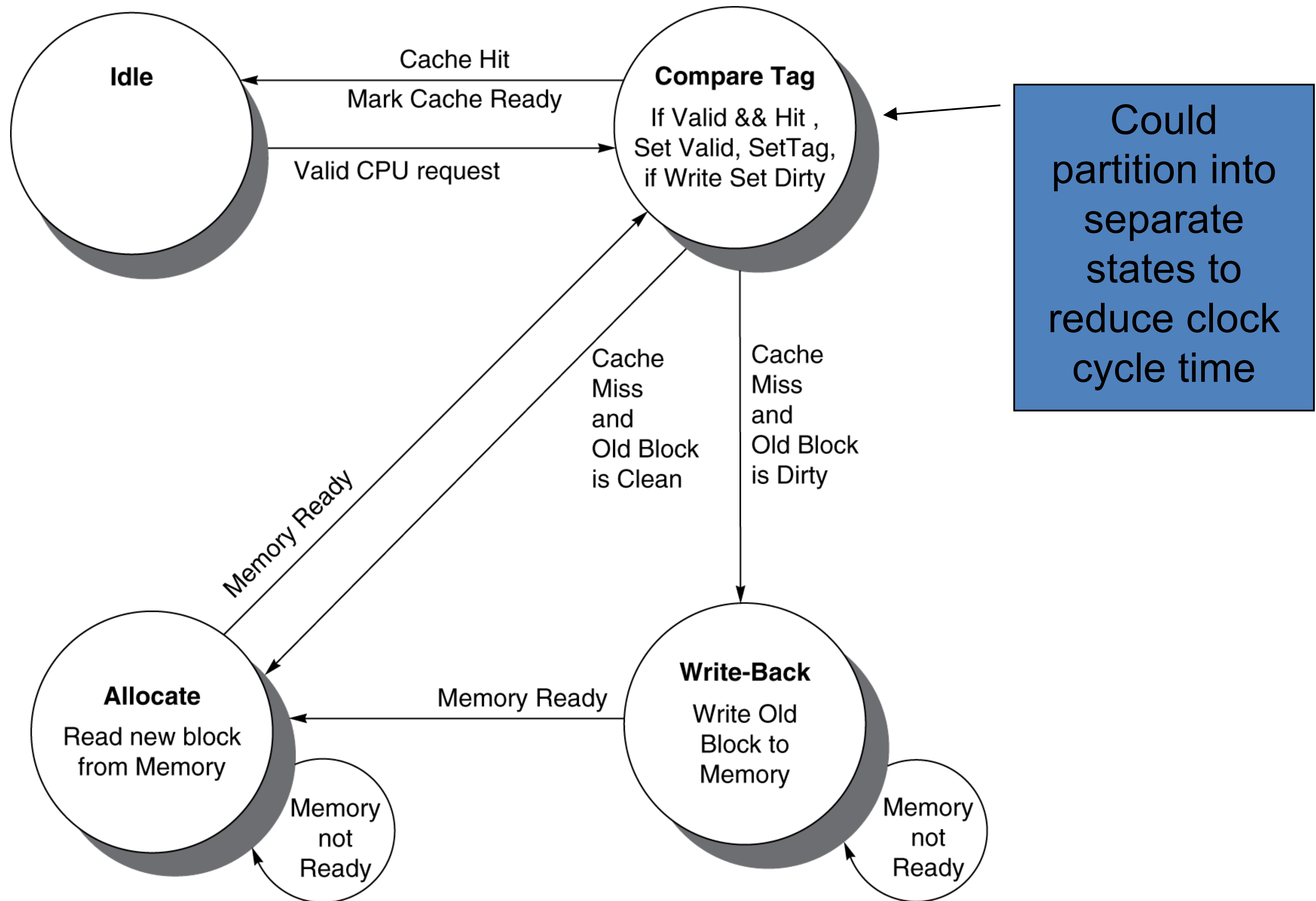


Finite State Machines

- Use an FSM to sequence control steps
- Set of states, transition on each clock edge
  - State values are binary encoded
  - Current state stored in a register
  - Next state $= f_n$ (current state, current inputs)
- Control output signals $= f_o$ (current state)



Cache Controller FSM



Cache Coherence Problem

- Suppose two CPU cores share a physical address space
  - Write-through caches

Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1

Coherence Defined

- Informally: Reads return most recently written value
- Formally:
  - P writes X; P reads X (no intervening writes)
    $\Rightarrow$ read returns written value
  - P_1 writes X; P_2 reads X (sufficiently later)
    $\Rightarrow$ read returns written value
    - c.f. CPU B reading X after step 3 in example
  - P_1 writes X, P_2 writes X
    $\Rightarrow$ all processors see writes in the same order
    - End up with the same final value for X

Cache Coherence Protocols

- Operations performed by caches in multiprocessors to ensure coherence
  - Migration of data to local caches
    - Reduces bandwidth for shared memory
  - Replication of read-shared data
    - Reduces contention for access
- Snooping protocols
  - Each cache monitors bus reads/writes
- Directory-based protocols
  - Caches and memory record sharing status of blocks in a directory

Invalidating Snooping Protocols

- Cache gets exclusive access to a block when it is to be written
  - Broadcasts an invalidate message on the bus
  - Subsequent read in another cache misses
    - Owning cache supplies updated value

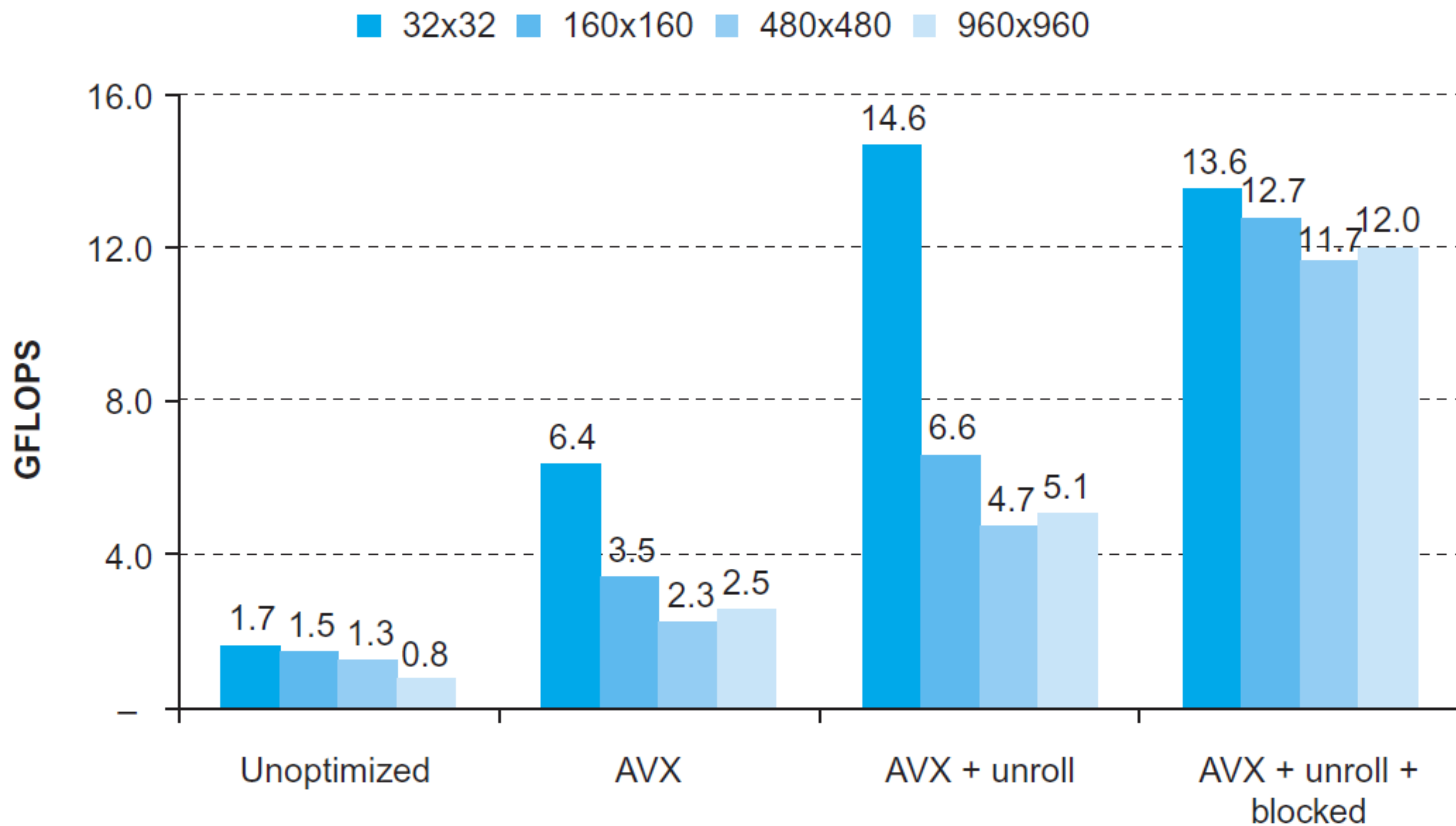
CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidate for X	1		0
CPU B read X	Cache miss for X	1	1	1

Memory Consistency

- When are writes seen by other processors
  - "Seen" means a read returns the written value
  - Can't be instantaneously
- Assumptions
  - A write completes only when all processors have seen it
  - A processor does not reorder writes with other accesses
- Consequence
  - P writes X then writes Y
    $\Rightarrow$ all processors that see new Y also see new X
  - Processors can reorder reads, but not writes

DGEMM

- Combine cache blocking and subword parallelism



Pitfalls

- Byte vs. word addressing
  - Example: 32-byte direct-mapped cache, 4-byte blocks
    - Byte 36 maps to block 1
    - Word 36 maps to block 4
- Ignoring memory system effects when writing or generating code
  - Example: iterating over rows vs. columns of arrays
  - Large strides result in poor locality

Pitfalls

- In multiprocessor with shared L2 or L3 cache
  - Less associativity than cores results in conflict misses
  - More cores $\Rightarrow$ need to increase associativity
- Using AMAT to evaluate performance of out-of-order processors
  - Ignores effect of non-blocked accesses
  - Instead, evaluate performance by simulation

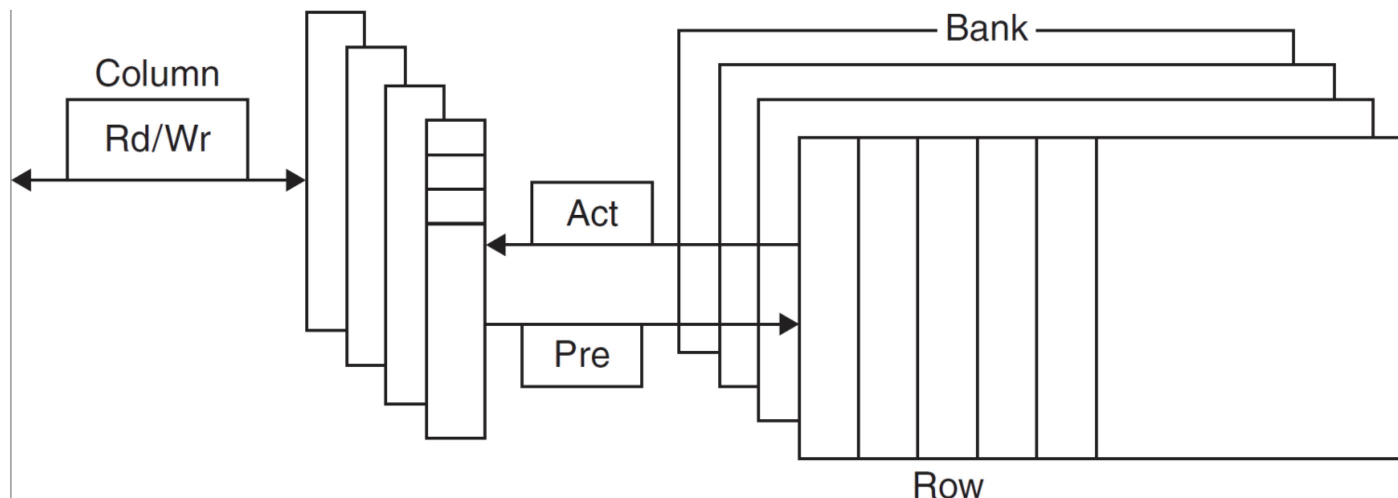
Pitfalls

- Extending address range using segments
  - E.g., Intel 80286
  - But a segment is not always big enough
  - Makes address arithmetic complicated
- Implementing a VMM on an ISA not designed for virtualization
  - E.g., non-privileged instructions accessing hardware resources
  - Either extend ISA, or require guest OS not to use problematic instructions

Slides that are not used

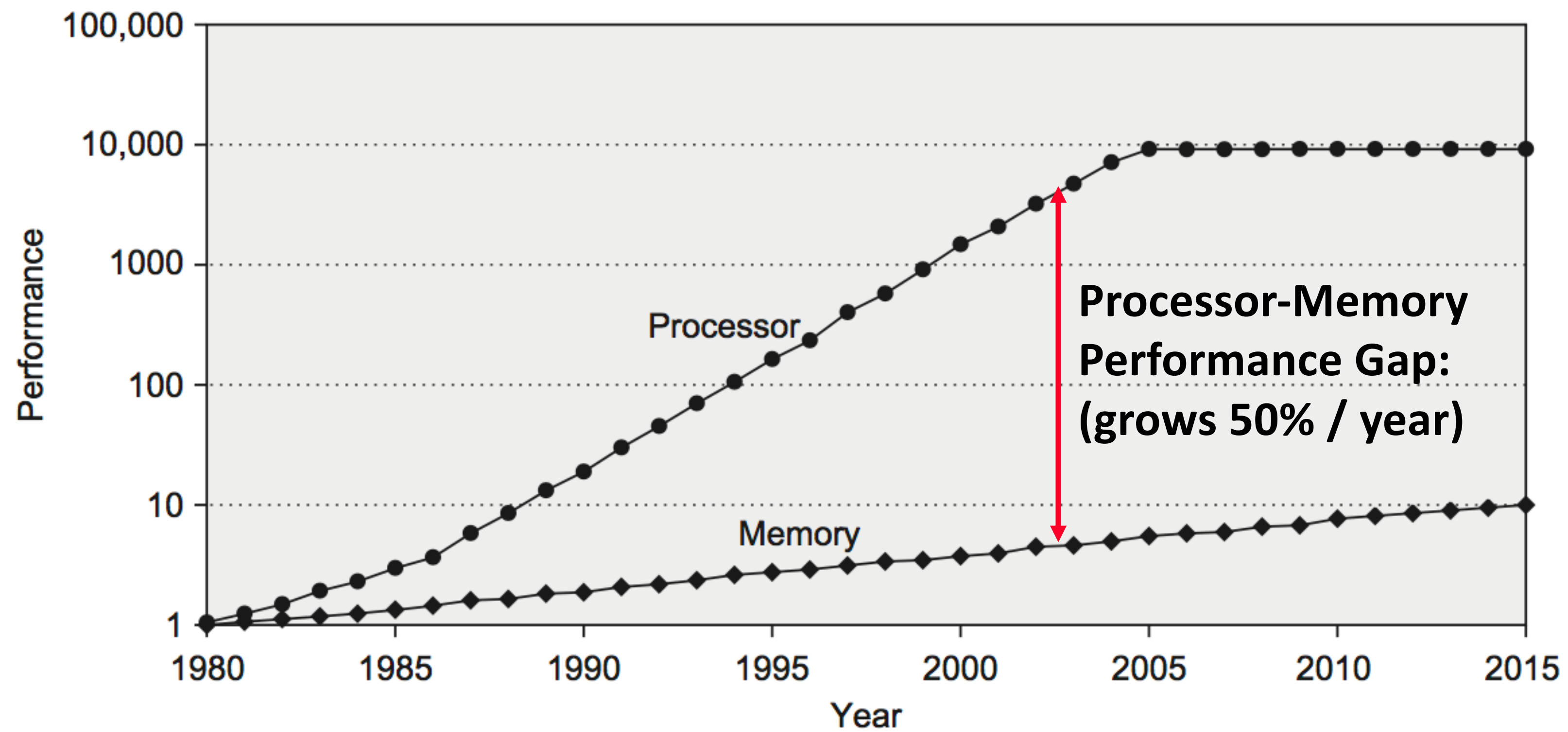
Advanced DRAM Organization

- Bits in a DRAM are organized as a rectangular array
  - DRAM accesses an entire row
  - Burst mode: supply successive words from a row with reduced latency
- Double data rate (DDR) DRAM
  - Transfer on rising and falling clock edges
- Quad data rate (QDR) DRAM
  - Separate DDR inputs and outputs



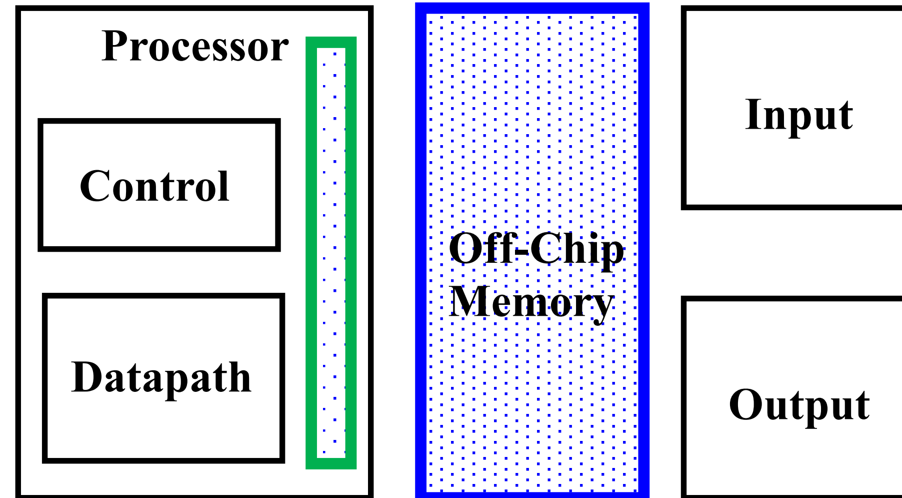
Memory is Much Slower Compared with CPU

CPU-DRAM Memory **Latency** Gap → Memory Wall



of the processor-DRAM performance gap. The memory baseline is 64 KiB DRAM in 1980, with a 1.07 per year performance improvement in latency (see Figure 2.4 on page 88). The processor line assumes a 1.25 improvement per year until 1986, a 1.52 improvement until 2000, a 1.20 improvement between 2000 and 2005, and only small improvements in processor performance (on a per-core basis) between 2005 and 2015. As you

Memory Hierarchy Works



- *capacity*: Register << SRAM << DRAM
- *latency*: Register << SRAM << DRAM
- *bandwidth:* on-chip >> off-chip

On a data access:

if data $\in$ fast memory $\Rightarrow$ low latency access *(SRAM)*

if data $\notin$ fast memory $\Rightarrow$ high latency access *(DRAM)*

Disk Sectors and Access

- Each sector records
  - Sector ID
  - Data (512 bytes, 4096 bytes proposed)
  - Error correcting code (ECC)
    - Used to hide defects and recording errors
  - Synchronization fields and gaps
- Access to a sector involves
  - Queuing delay if other accesses are pending
  - Seek: move the heads
  - Rotational latency
  - Data transfer
  - Controller overhead

Disk Access Example

- Given
  - 512B sector, 15,000rpm, 4ms average seek time, 100MB/s transfer rate, 0.2ms controller overhead, idle disk
- Average read time
  - **4ms seek time**
    + ½ / (15,000/60) = 2ms rotational latency
    + 512 / 100MB/s = 0.005ms transfer time
    + 0.2ms controller delay
    = 6.2ms
- If actual average seek time is 1ms
  - Average read time = 3.2ms

Slides for Lab 13/14

Sources of locality

- Temporal locality
  - Code within a loop
  - Same instructions fetched repeatedly
- Spatial locality
  - Data arrays
  - Local variables in stack
  - Data allocated in chunks (contiguous bytes)

```
for (i=0; i<N; i++) {
   A[i] = B[i] + C[i] * a;
}
```

Writing Cache Friendly Code

- Repeated references to variables are good (temporal locality)
- Stride-1 reference patterns are good (spatial locality)
- Examples:
  - cold cache, 4-byte words, 4-word cache blocks

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = 1/4 = 25%

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = 100%

Matrix Multiplication Example

- Major cache effects to consider
  - Total cache size
    - Exploit temporal locality and blocking)
  - Block size
    - Exploit spatial locality

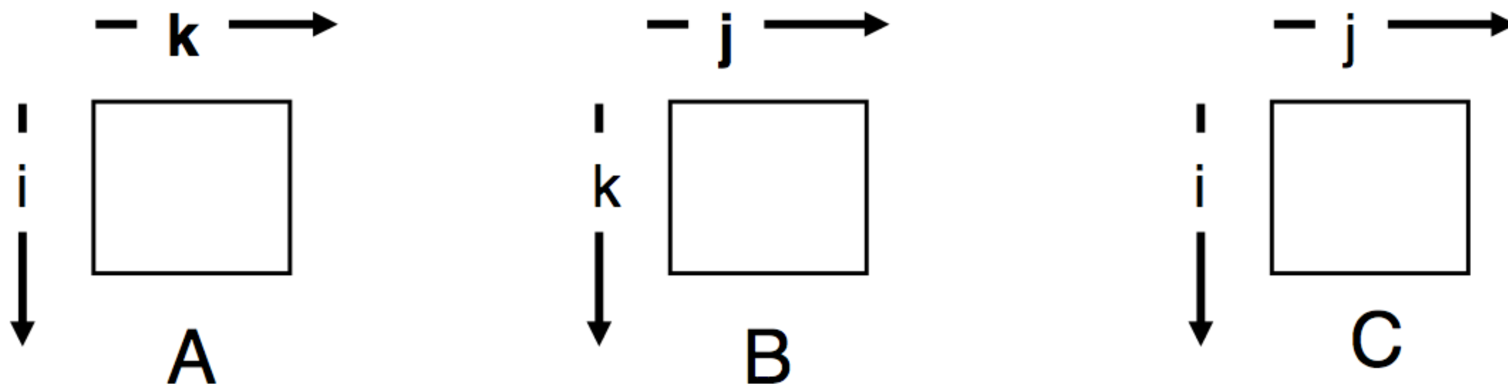
- Description:
  - Multiply N x N matrices
  - $O(N^3)$ total operations
  - Accesses
    - N reads per source element
    - N values summed per destination
      - but may be able to hold in register

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Variable `sum` held in register

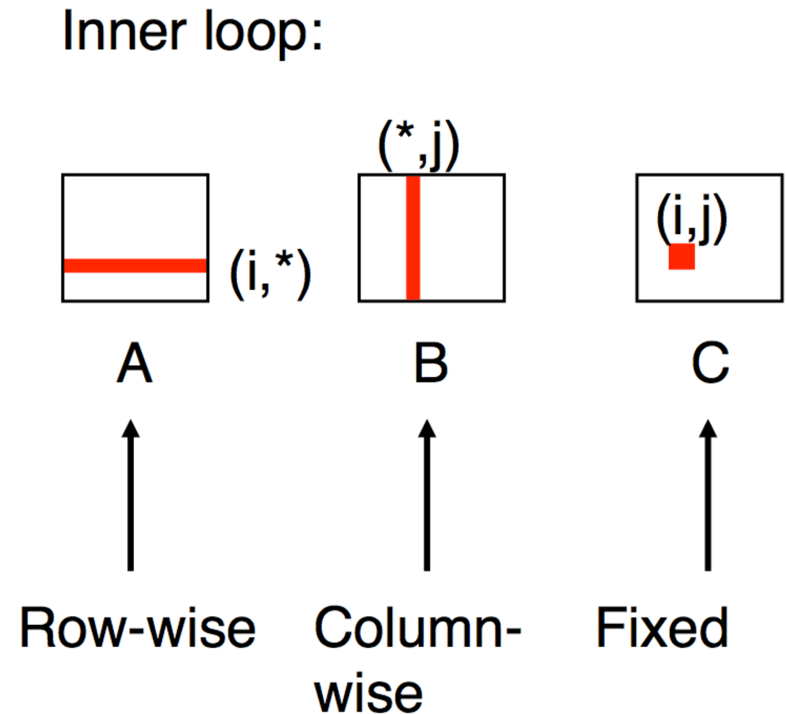
Miss Rate Analysis for Matrix Multiply

- Assume:
  - Cache line size = 32 Bytes (big enough for 4 64-bit words)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows
- Analysis method:
  - Look at access pattern of inner loop



Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

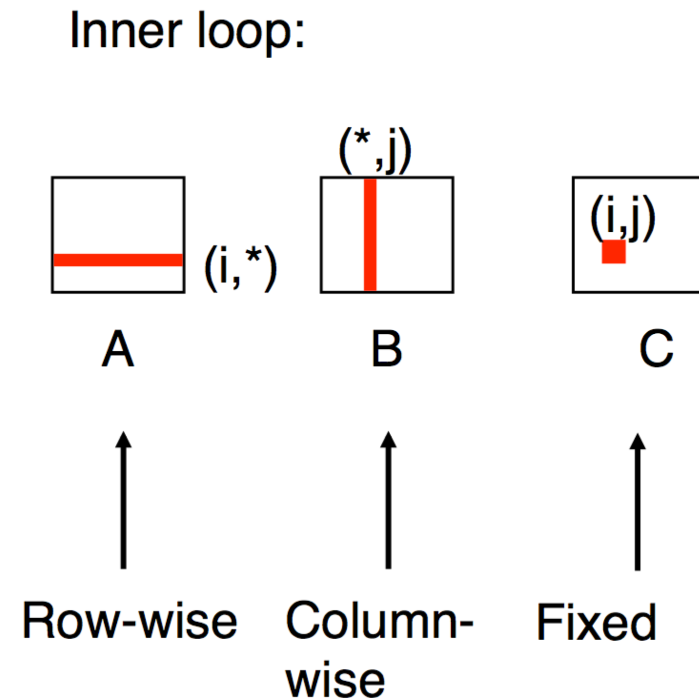


- Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```

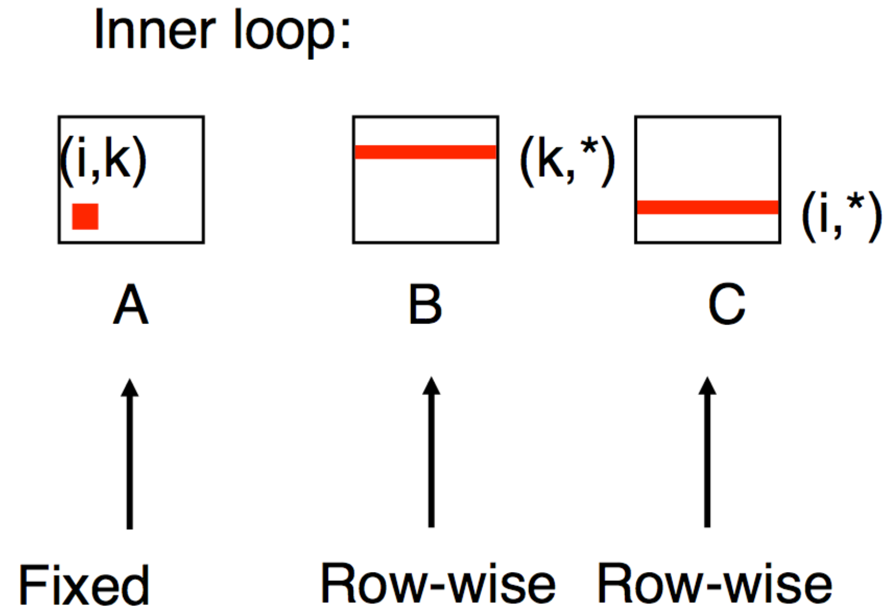


- Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

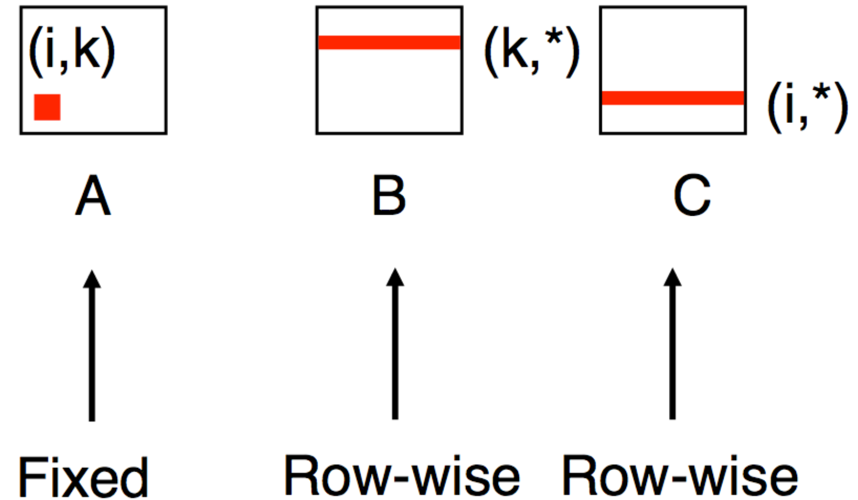


- Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

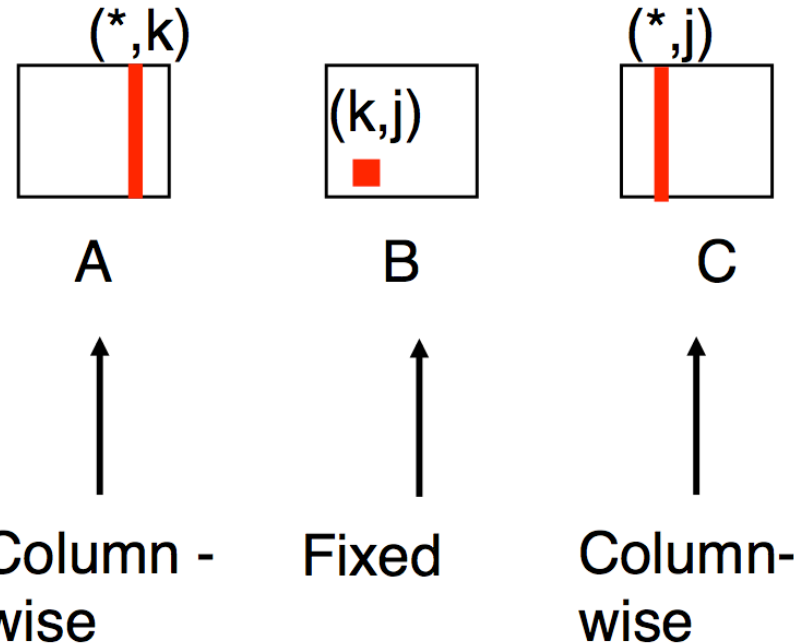


- Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

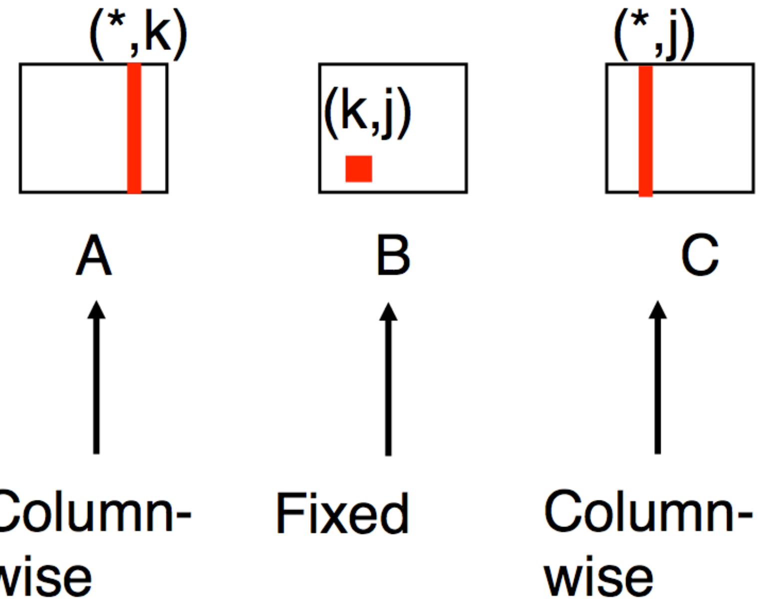


- Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```



- Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = **1.25**

```
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] *
    b[k][j]; c[i][j] = sum;
  }
}
```

kij (& ikj):
- 2 loads, 1 store
- misses/iter = **0.5**

```
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

jki (& kji):
- 2 loads, 1 store
- misses/iter = **2.0**

```
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```