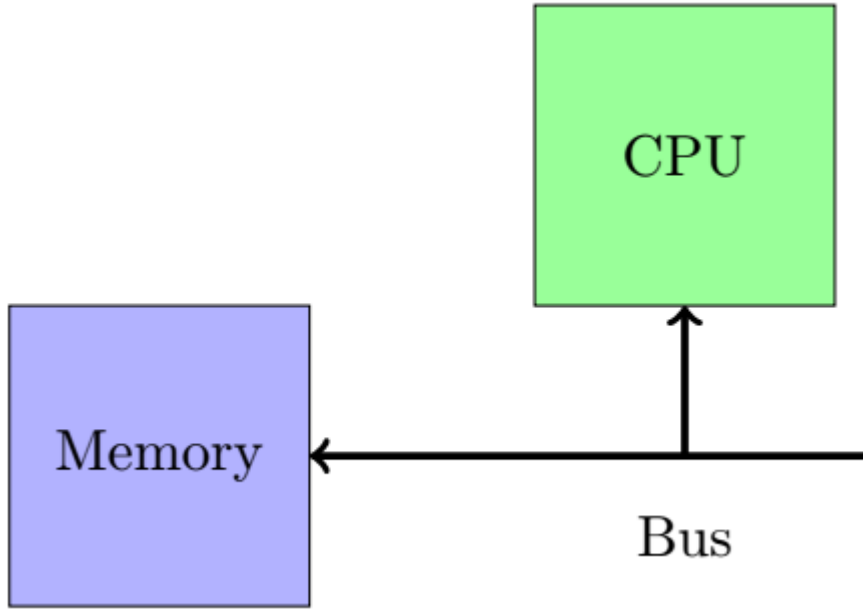
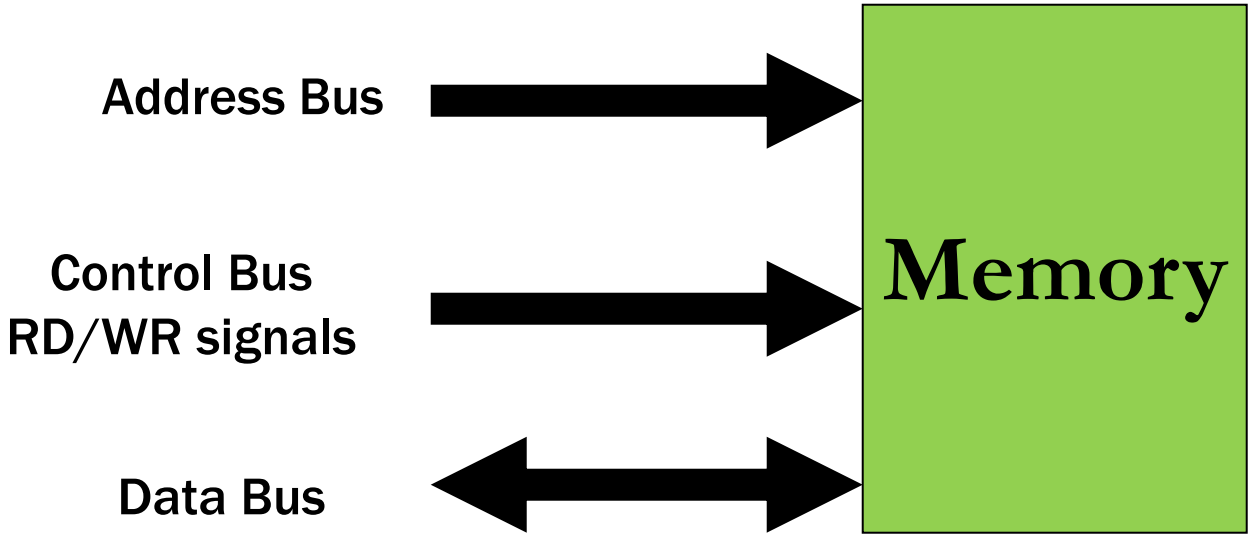
CS2053 Computer Architecture

Memory Hierarchy

Dr. Sulochana Sooriyaarachchi

Accessing Memory



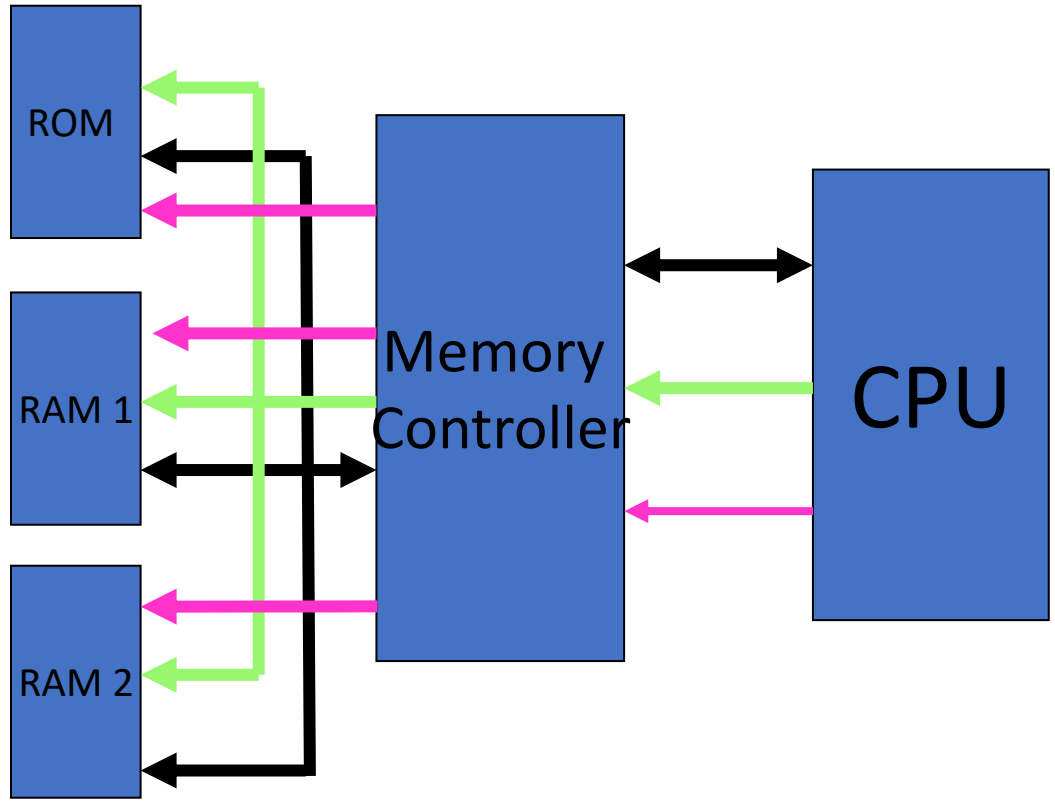


Reading From & Writing to Memory

- Reading from memory
  - **Address** of memory location to read is placed on Address Bus
  - **Read Signal** (RD) in control bus is activated
  - **Data** is fetched (read) from Data Bus
- Writing to memory
  - **Address** of memory location to write is placed on Address Bus
  - **Data** is placed on Data Bus
  - **Write Signal** (WR) in control bus is activated

Connecting Memory & CPU

Details of handling different memory modules & memory hierarchy is off load to memory controller



Addressing Modes

- How architectures specify the address of an object they access
- Specifies
  - Constants
  - Registers
  - Memory locations
- In addressing memory locations
  - Actual memory address is called *effective address*

Example Addressing Modes

Addressing mode	Example instruction	Meaning	When used
Register	`Add R4,R3`	`Regs[R4]←Regs[R4] +Regs[R3]`	When a value is in a register
Immediate	`Add R4,3`	`Regs[R4]←Regs[R4]+3`	For constants
Displacement	`Add R4,100(R1)`	`Regs[R4]←Regs[R4] +Mem[100+Regs[R1]]`	Accessing local variables (+ simulates register indirect, direct addressing modes)
Register indirect	`Add R4,(R1)`	`Regs[R4]←Regs[R4] +Mem[Regs[R1]]`	Accessing using a pointer or a computed address
Indexed	`Add R3,(R1+R2)`	`Regs[R3]←Regs[R3] +Mem[Regs[R1]+Regs [R2]]`	Sometimes useful in array addressing: `R1` = base of array; `R2` = index amount
Direct or absolute	`Add R1,(1001)`	`Regs[R1]←Regs[R1] +Mem[1001]`	Sometimes useful for accessing static data; address constant may need to be large
Memory indirect	`Add R1,@(R3)`	`Regs[R1]←Regs[R1] +Mem[Mem[Regs[R3]]]`	If `R3` is the address of a pointer p, then mode yields $*p$
Autoincrement	`Add R1,(R2)+`	`Regs[R1]←Regs[R1] +Mem[Regs[R2]]` `Regs[R2]←Regs[R2]+d`	Useful for stepping through arrays within a loop. `R2` points to start of array; each reference increments `R2` by size of an element, d
Autodecrement	`Add R1, -(R2)`	`Regs[R2]←Regs[R2] - d` `Regs[R1]←Regs[R1] +Mem[Regs[R2]]`	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	`Add R1,100(R2)[R3]`	`Regs[R1]←Regs[R1] +Mem[100+Regs[R2] +Regs[R3] * d]`	Used to index arrays. May be applied to any indexed addressing mode in some computers

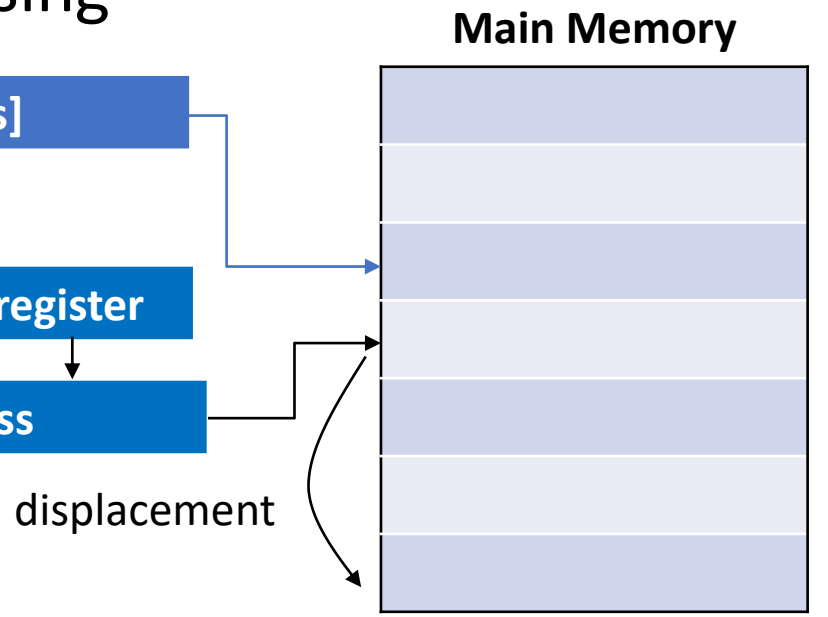
Addressing modes illustrations

- Register

- Immediate value

- Direct = Absolute addressing

- Displacement

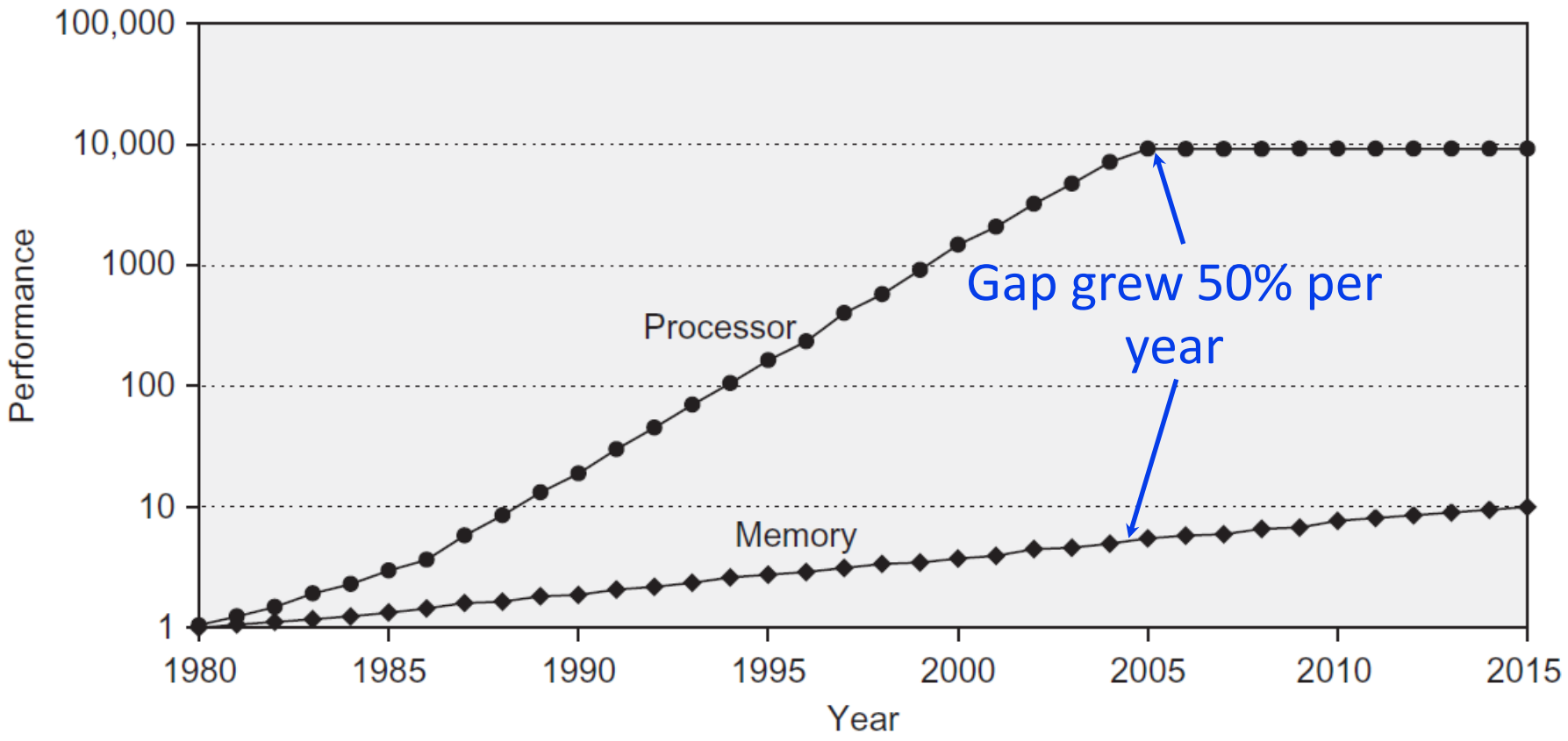


Take home assignment 1

- Study the addressing modes in the processor architecture in your Lab Series.
  - PC-relative: via *auipc, jal* and *br** instructions
  - Register-offset: via the *jalr, addi* and all memory instructions.
  - Absolute: via the *lui* instruction
- Read relevant sections in: RISCV ISA SpecificationsURL
  - https://online.uom.lk/mod/url/view.php?id=343622
    - Volume 1, Unprivileged Specification version 20191213 [PDF]

Submit a short note for each of above instructions with an illustration of addressing involved

Processor Memory Gap



Source: Computer Architecture, A Quantitative Approach by John L. Hennessy and David A. Patterson

Memory Hierarchy

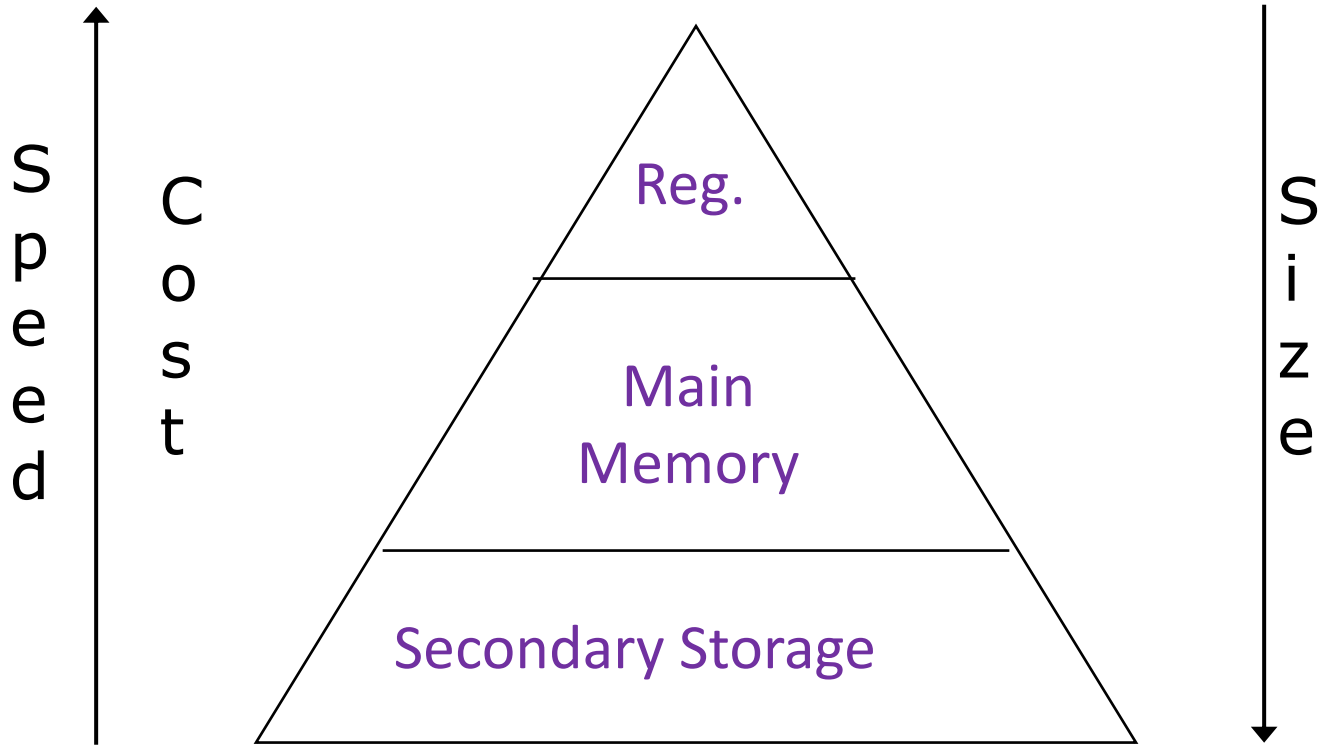
- Memory has to be organized in such a way that its slowness doesn’t reduce performance of overall system
- Some memory types are fast but expensive
  - Registers, Static RAM
- Some other types are cheap but slow
  - Dynamic RAM
- Solution
  - Hierarchical memory system
  - Limited capacity of fast but expensive memory types
  - Larger capacity of slow but cheap memory types

Memory Hierarchy (Cont.)

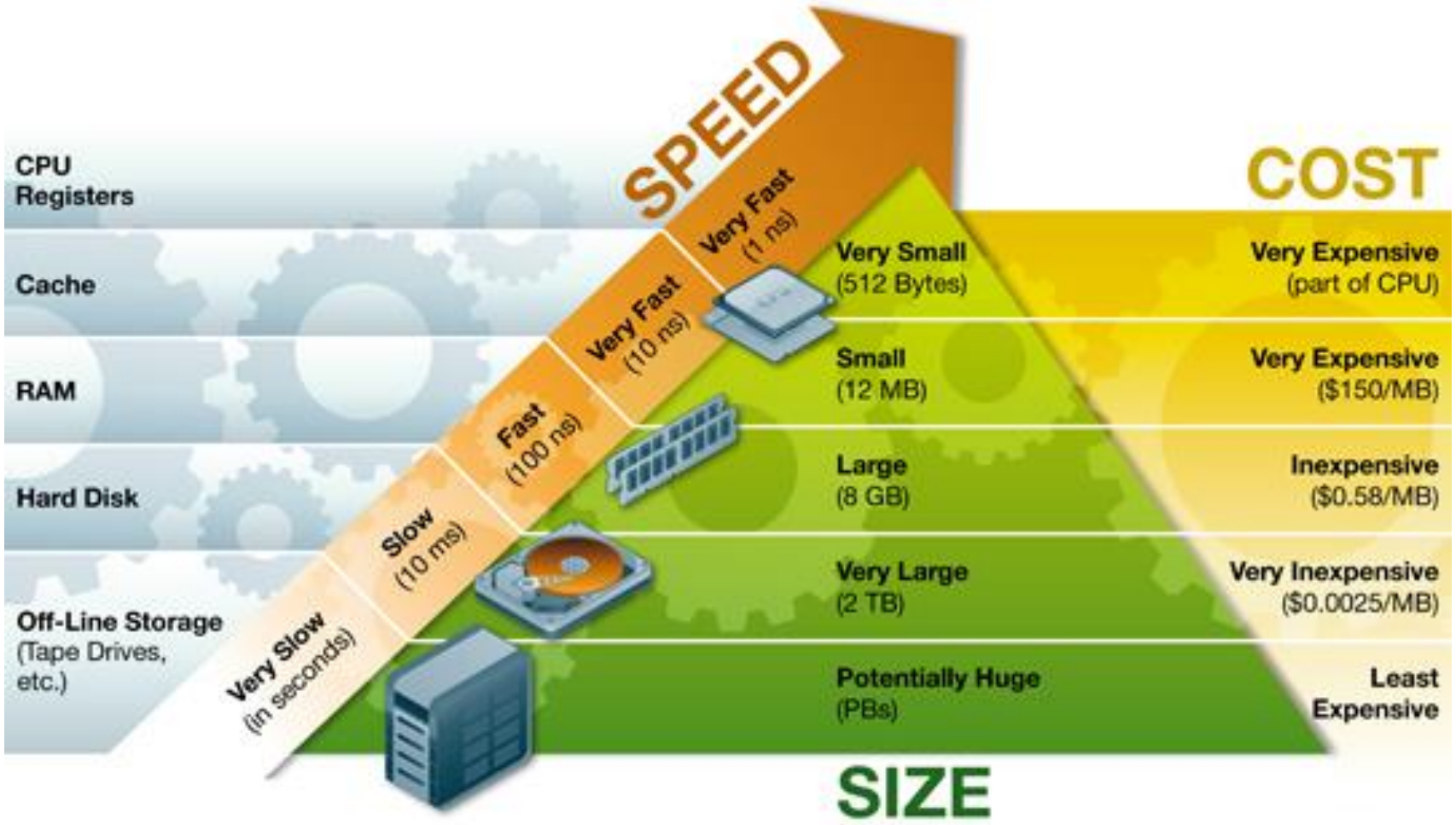
- Objective is to have a memory system
  - with sufficient speed
  - with sufficient capacity
  - as cheap as possible

> 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).

Traditional Memory Hierarchy

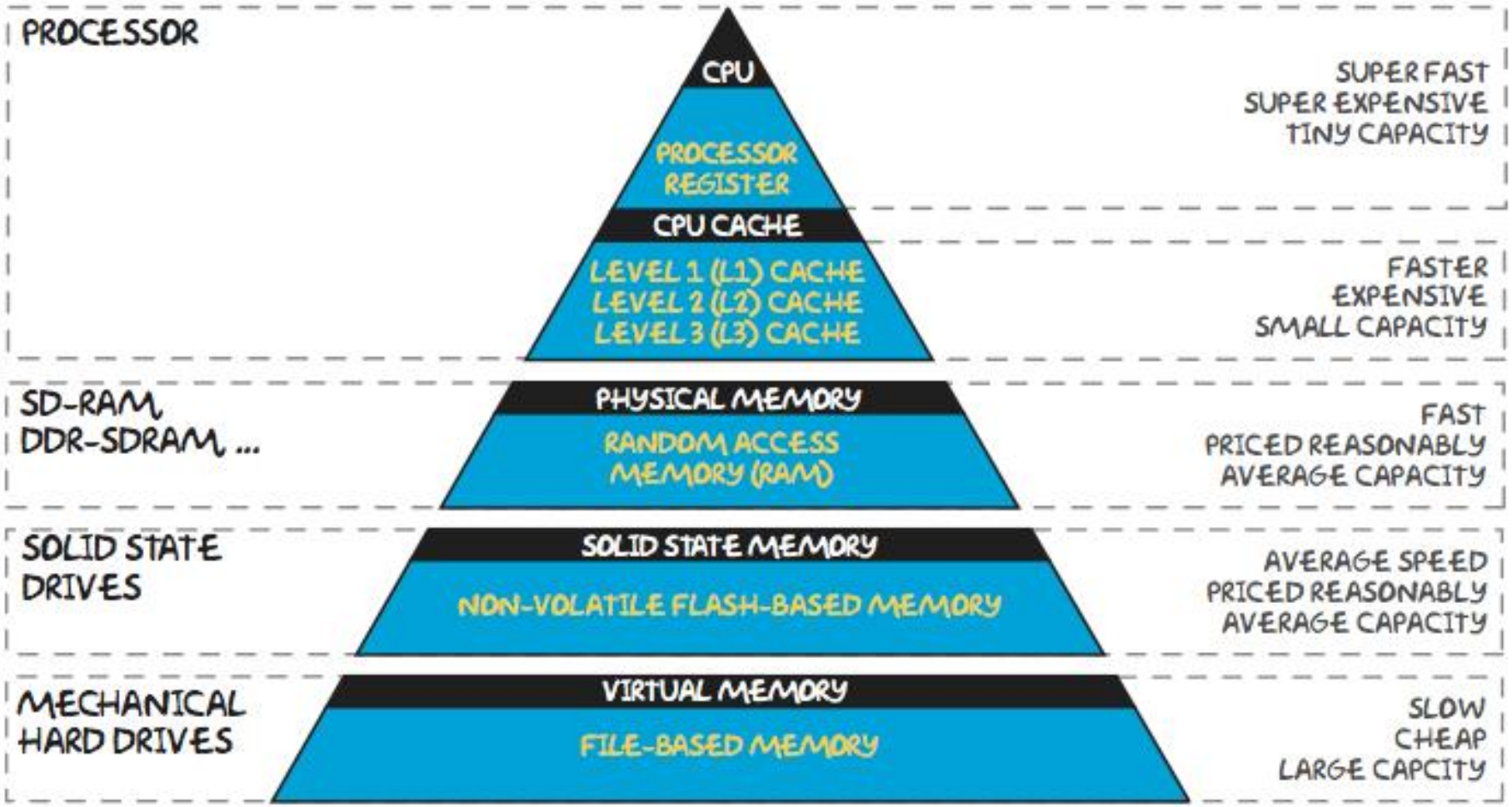


Extended Memory Hierarchy



Source: http://www.ts.avnet.com/uk/products_and_solutions/storage/hierarchy.html

Modern Memory Hierarchy



Source: http://blog.teachbook.com.au/index.php/2012/02/memory-hierarchy/

Principle of Locality

- Programs tend to reuse data & instructions that are close to each other or they have used recently
- Temporal locality
  - Recently referenced items are likely to be referenced in the near future
  - A block tend to be accessed again & again
- Spatial locality
  - Items with nearby addresses tend to be referenced close together in time
  - Near by blocks tend to be accessed

Locality – Example

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

- Data
  - Access 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

Locality examples

- Loops
- Sequential instructions
- Array elements

```
int sum_array_cols(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;
}
```

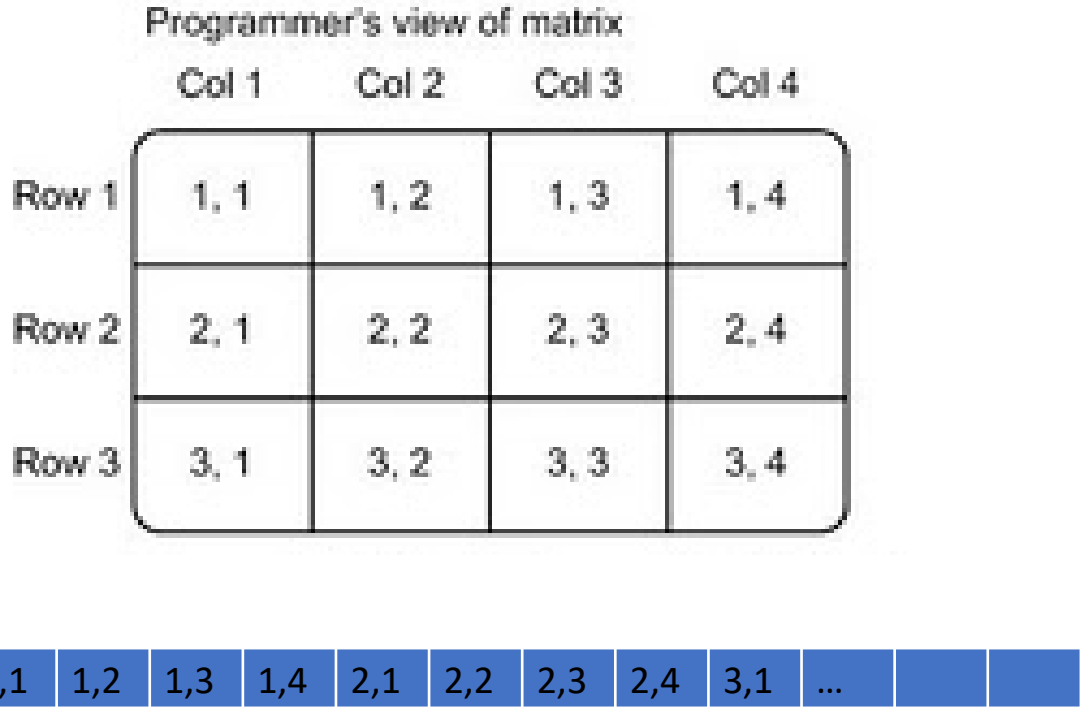
	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Prefetching – Example

- Which of the following code is faster?

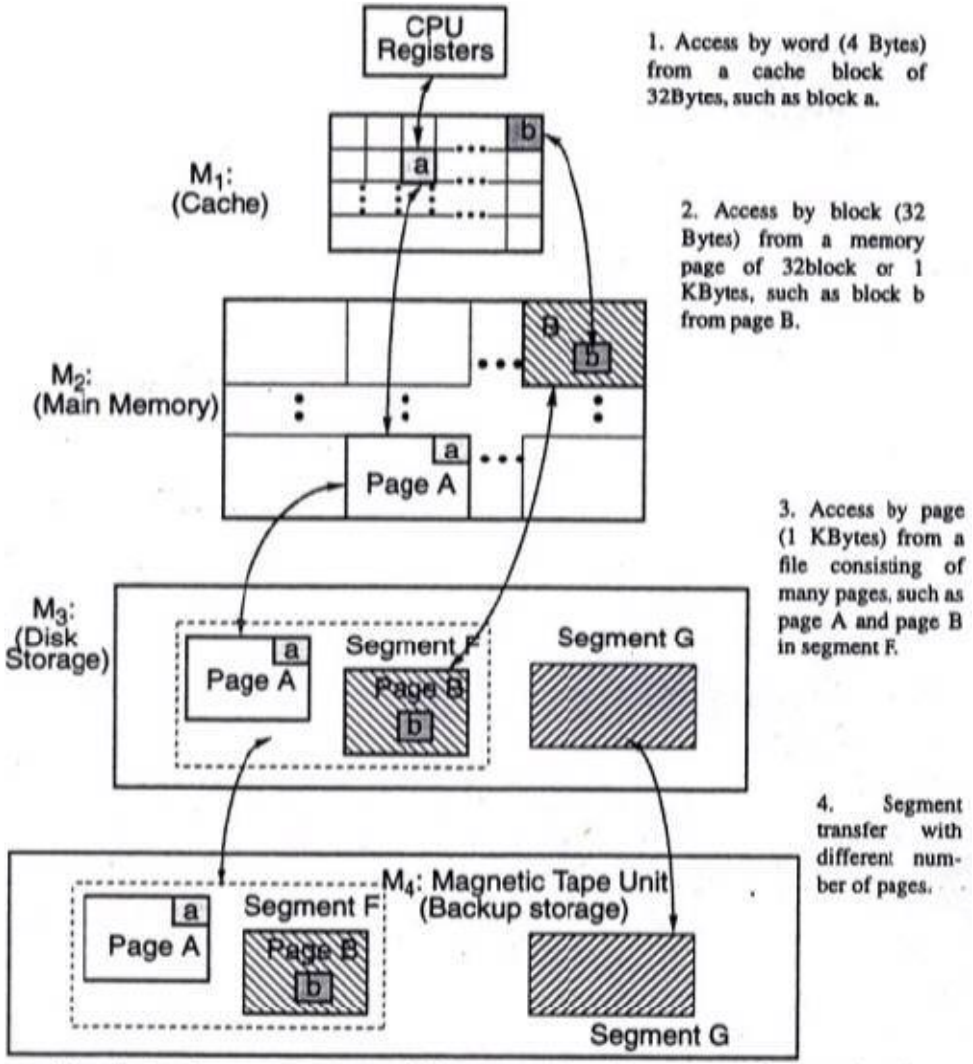
```
sum = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < m; j++)
    sum += a[i][j];
return sum;

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



Inclusion Property

- Often data in lower levels of hierarchy are a superset of higher level memories
- Terminology
  - Block (=line)
  - Hit, Miss
  - Hit rate, Miss rate
  - Hit time, Miss penalty

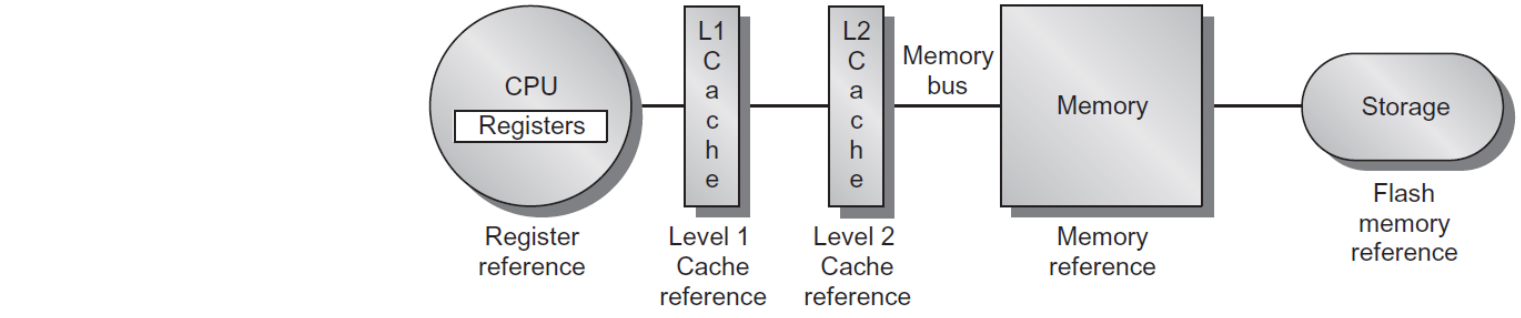


The inclusion property and data transfers between adjacent levels

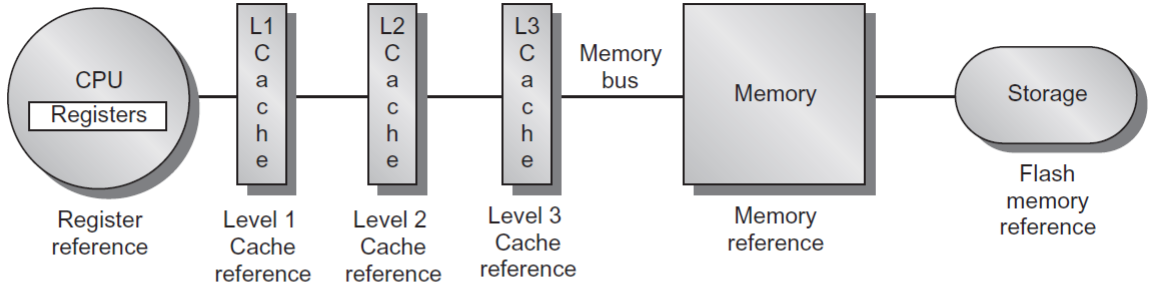
Impact of Hierarchical Memory

- Memory is no more a flat structure
- Affect many aspects of a computer
  - How OS manages memory & IO
  - How compilers generate codes
  - How applications use the computers
- Programs spend more time on memory access
  - Processors *stall* waiting for data from memory
- Need to know hierarchy for optimizing

Memory Hierarchy in Operation



(A) Memory hierarchy for a personal mobile device



(B) Memory hierarchy for a laptop or a desktop

Summary

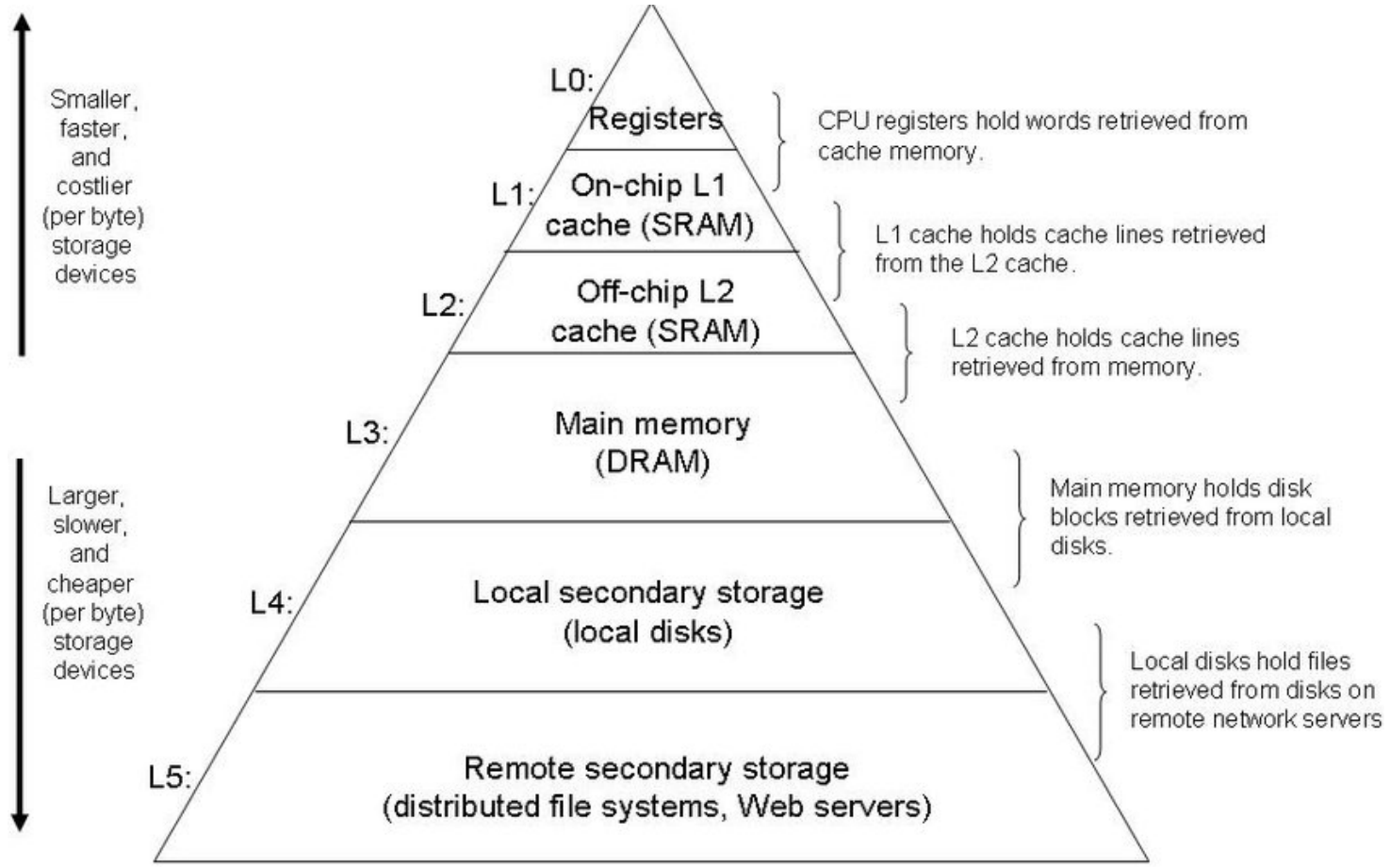
- Memory hierarchy - Illusion of a large amount of fast memory
- All data and instructions in memory are not accessed at once with equal probability
- Principle of Locality – accessing small portion of address space at any instance of time
  - Temporal locality – refer the same item again and again
  - Spatial locality – refer the items stored close to each other

Hierarchical Organization of Memory

- Hardware:
  - Small
  - Fast
  - Expensive
- Data:

  Data in upper level

  $\subseteq$ data in lower level

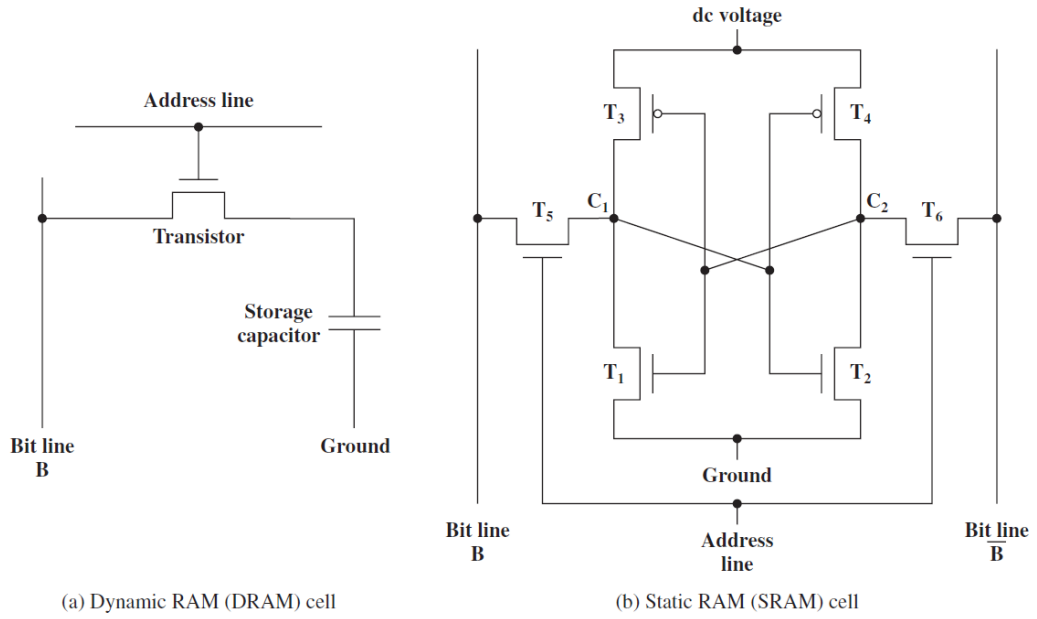


Memory Technologies

Typical access time	$ Per GiB in 2020
0.5 – 2.5ns	$500 – $1000
50 – 70ns	$3 – $6
5µs – 50µs	$0.06 – $0.12
5ms – 20ms	$0.01 – $0.02

- Dynamic Random Access Memory (DRAM)
  - Less cost per bit, slow, less area per bit → larger capacity
  - Main memory made of DRAM
- Static Random Access Memory (SRAM)
  - High cost per bit, fast, more area per bit → small capacity
  - Cache memory made of SRAM

Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10



(a) Dynamic RAM (DRAM) cell

(b) Static RAM (SRAM) cell

Memory types

Memory Type	Category	Erasure	Write Mechanism	Volatility
Random-access memory (RAM)	Read-write memory	Electrically, byte-level	Electrically	Volatile
Read-only memory (ROM)	Read-only memory	Not possible	Masks	Nonvolatile
Programmable ROM (PROM)			Electrically	
Erasable PROM (EPROM)	Read-mostly memory	UV light, chip-level		
Electrically Erasable PROM (EEPROM)		Electrically, byte-level		
Flash memory		Electrically, block-level		

Take home assignment 2

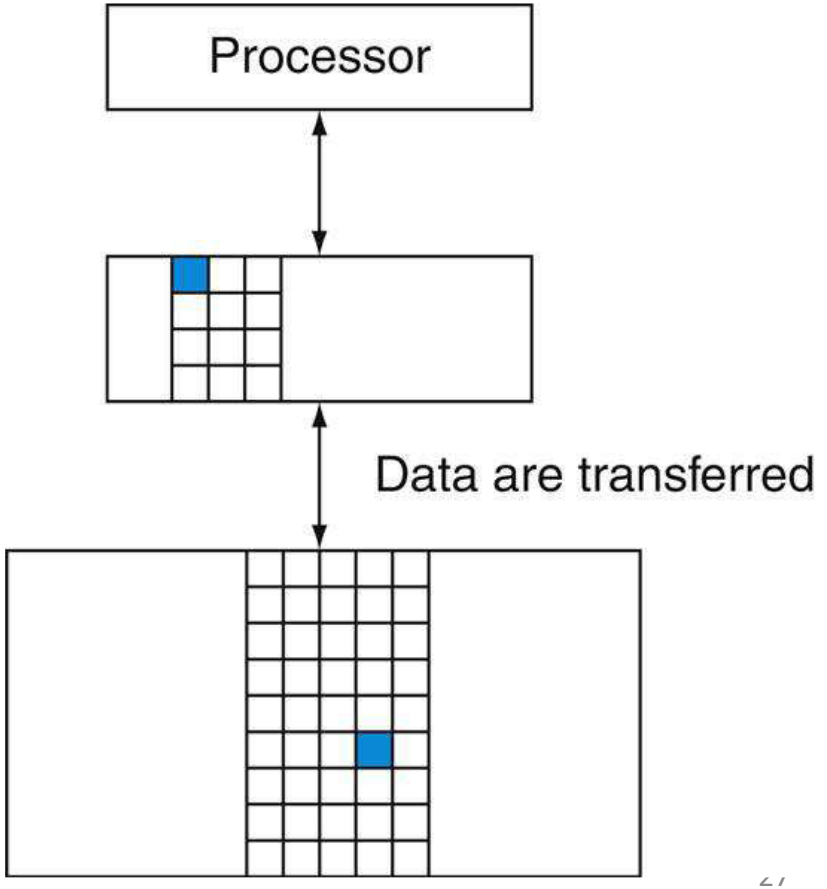
Read Section 5.2 of

Patterson, David. "Computer organization and design RISC-V edition: the hardware." (2017) and

Answer the Quiz during my next lecture.

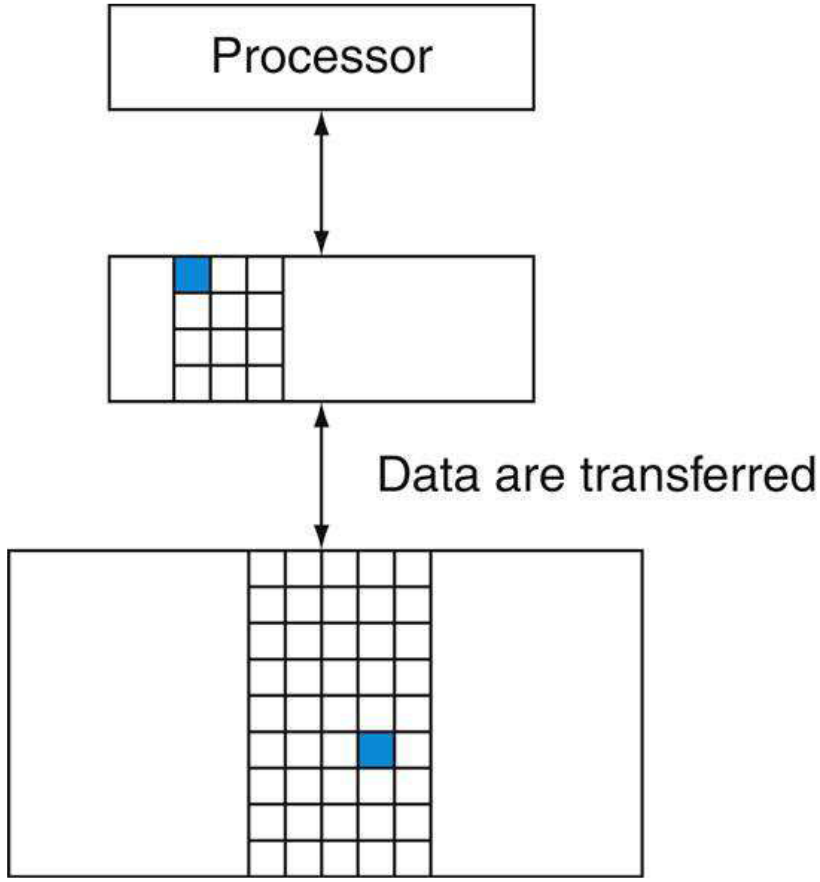
Cache

- Small amount of memory that is faster than DRAM
  - Slower than registers
  - Built using SRAM
  - Range from few KB to few MB
- Used by CPU to store frequently used instructions & data
  - Spatial & temporal locality
- Use multiple levels of cache
  - L1 Cache – Very fast, usually within CPU itself
  - L2 Cache – Slower than L1, but faster than DRAM
  - Today there's even L3 Cache

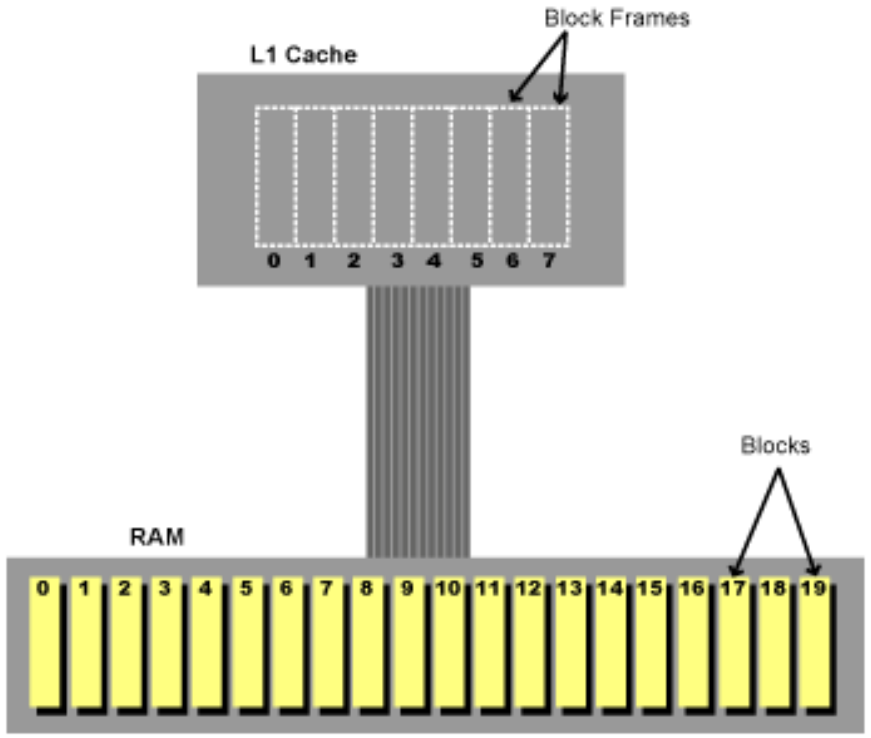


Cache - terminology

- Minimum unit of data: block = line
- If data requested by the processor
  - is found in Cache: hit
  - Is NOT found in Cache: miss
- Hit ratio: fraction of memory accesses found in cache
- Miss ratio: fraction of memory accesses not found in cache
- Hit time: time to access a block in cache
- Miss penalty: time to bring a block from memory plus deliver it to processor



Cache Blocks



Source:
http://archive.arstechnica.com/paedia/c/caching/m-caching-5.html

- A collection of *words* are called a *block*
- Multiple blocks are moved between levels in cache hierarchy
- Blocks are tagged with memory address
  - Tags are searched parallel

Cache Misses

- When required item is not found in cache
- Miss rate – fraction of cache accesses that result in a failure
- Types of misses
  - Compulsory – 1st access to a block
  - Capacity – limited cache capacity force blocks to be removed from a cache & later retrieved
  - Conflict (collision)– multiple blocks compete for the same set/block
- Average memory access time

  = *Hit time* + *Miss rate* x *Miss penalty*

Cache Access

Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
10110_{two}	miss (5.9b)	$(10110_{two} \bmod 8) = 110_{two}$
11010_{two}	miss (5.9c)	$(11010_{two} \bmod 8) = 010_{two}$
10110_{two}	hit	$(10110_{two} \bmod 8) = 110_{two}$
11010_{two}	hit	$(11010_{two} \bmod 8) = 010_{two}$
10000_{two}	miss (5.9d)	$(10000_{two} \bmod 8) = 000_{two}$
00011_{two}	miss (5.9e)	$(00011_{two} \bmod 8) = 011_{two}$
10000_{two}	hit	$(10000_{two} \bmod 8) = 000_{two}$
10010_{two}	miss (5.9f)	$(10010_{two} \bmod 8) = 010_{two}$
10000_{two}	hit	$(10000_{two} \bmod 8) = 000_{two}$

Cache Misses – Example

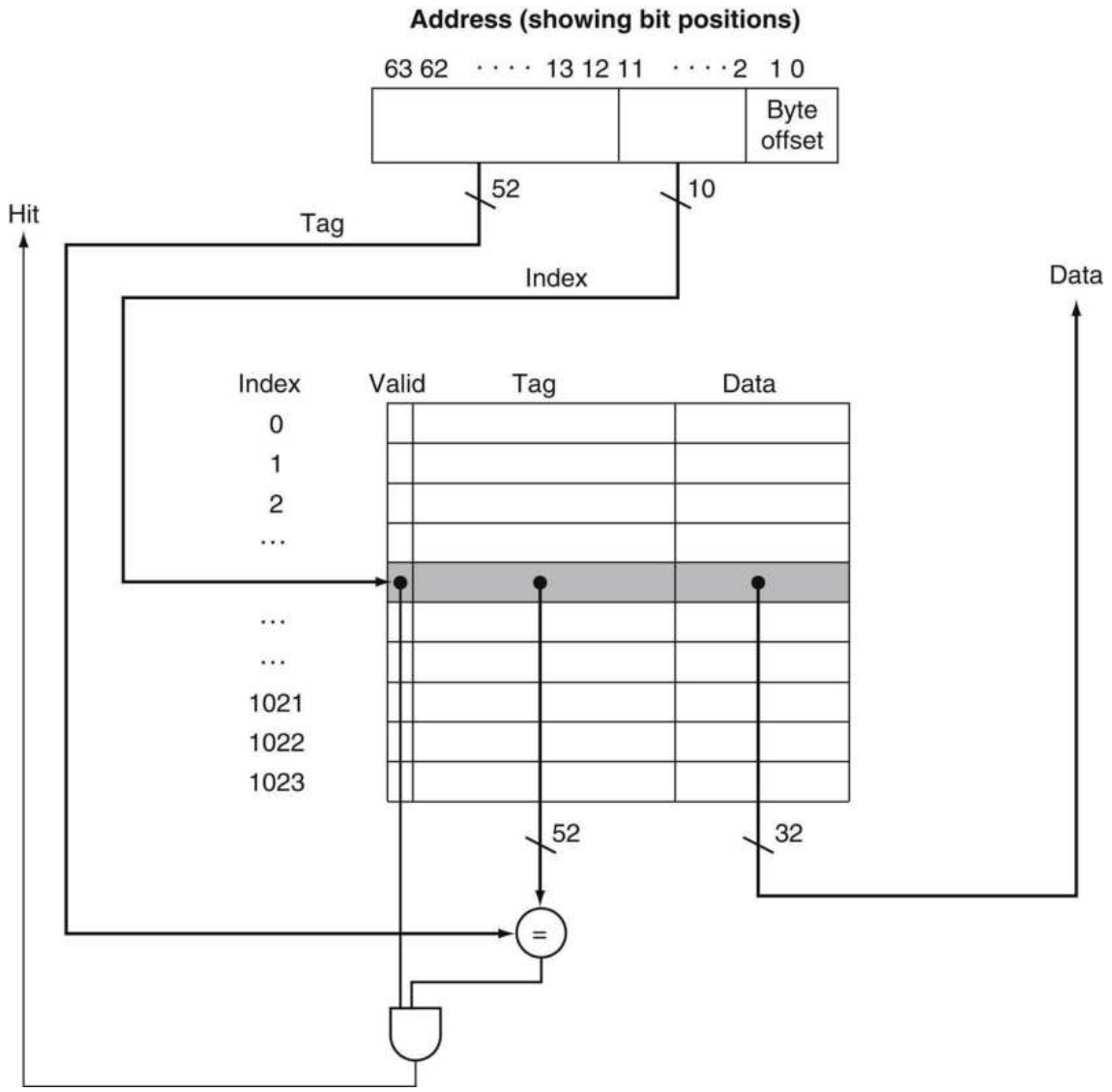
- Assume 40% of the instructions are data accessing instructions
- A hit takes 1 clock cycle & miss penalty is 100 clock cycles
- Assume instruction miss rate is 4% & data access miss rate is 12%, what is the average memory access time?

Cache Misses – Example

- Consider a cache with a block size of 4 words
- It takes 1 clock cycle to access a word in cache
- It takes 15 clock cycles to access a word from main memory
  - How much time will it take to access a word that’s not in cache?
  - What is the average memory access time if miss rate is 0.4?
  - What is the average memory access time if miss rate is 0.1?

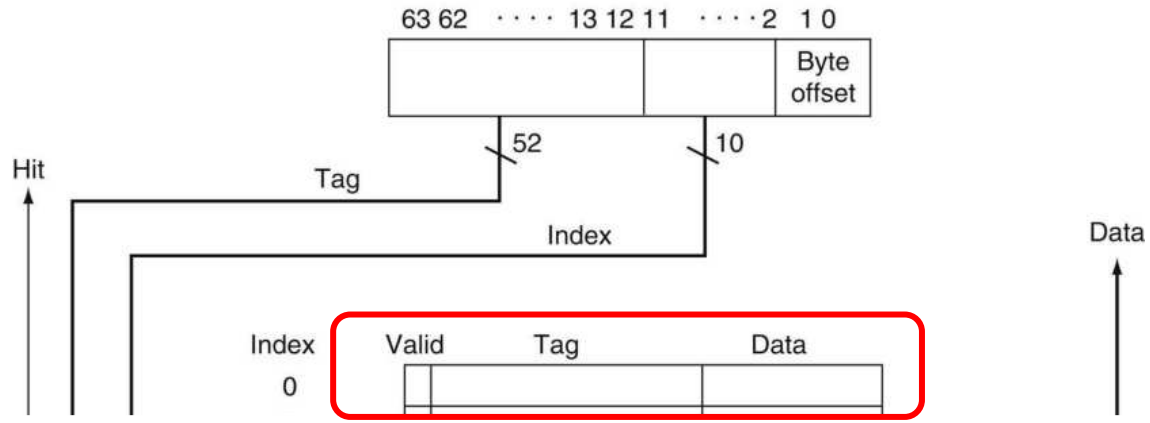
Accessing Cache

- Tag = upper portion of address
- Index = (Memory block address) modulo #Cache blocks
- Valid bit



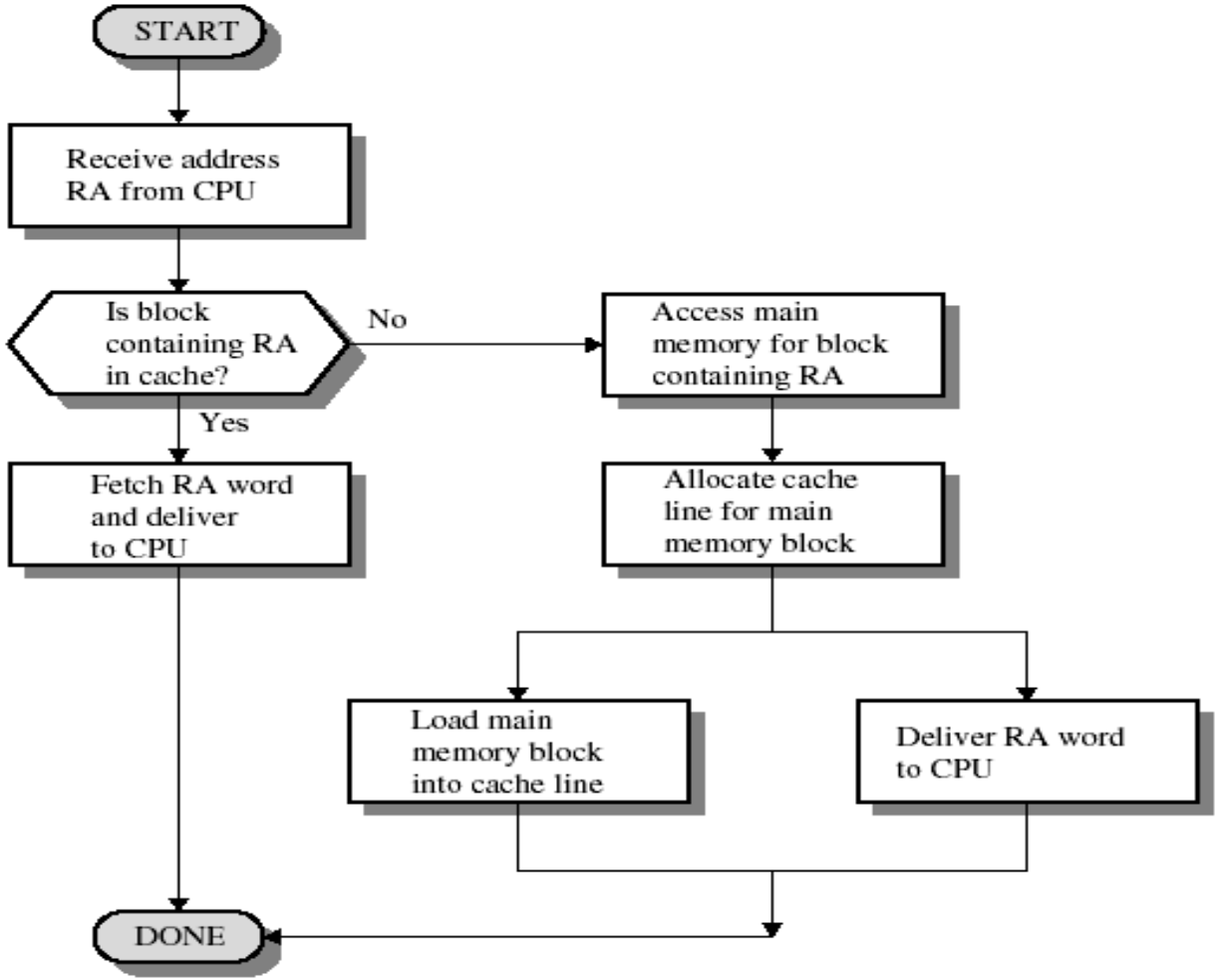
Cache organization

- In a Cache Entry
  - Valid bit
  - Tag
  - Data
- What is the size of a Cache Entry?



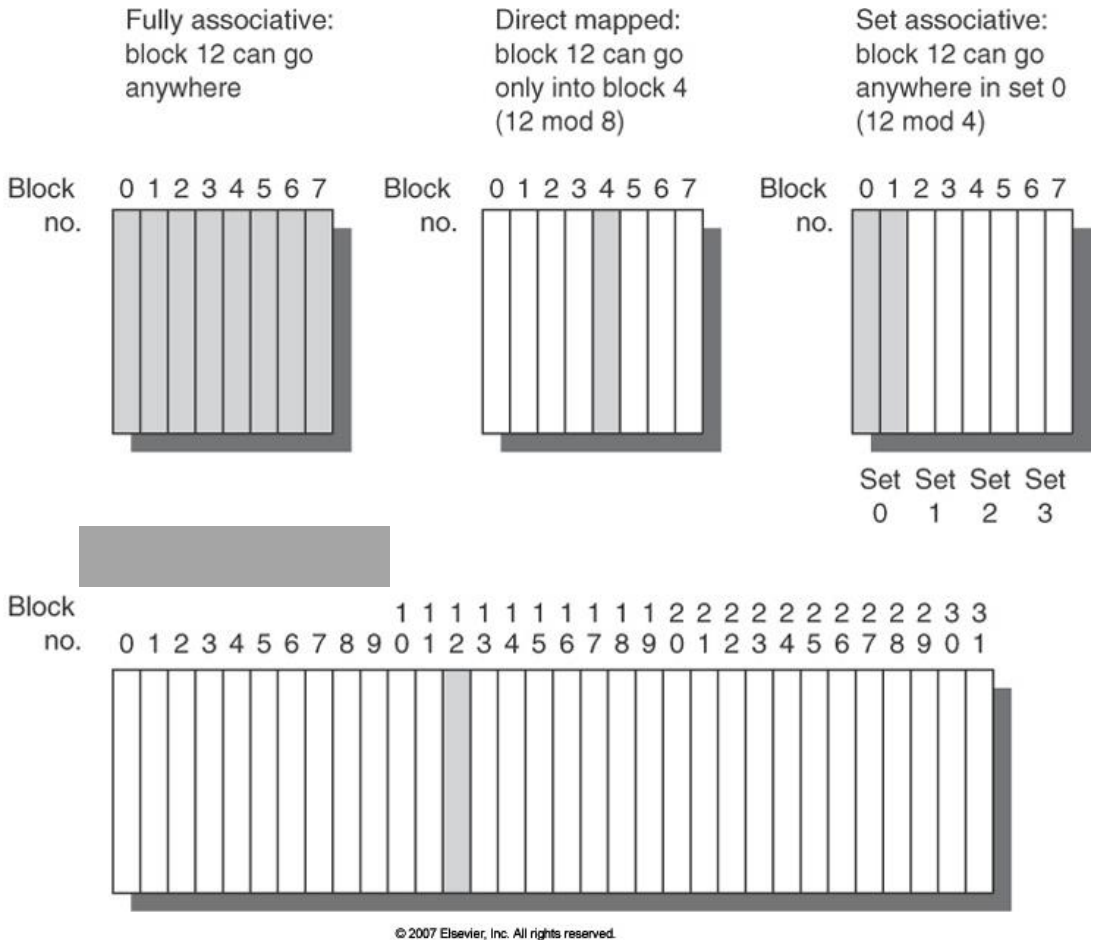
- Requested Address (RA) by processor is decoded as
  - Tag
  - Block index
  - Byte offset within the block

Cache Read Operations



Cache Associativity

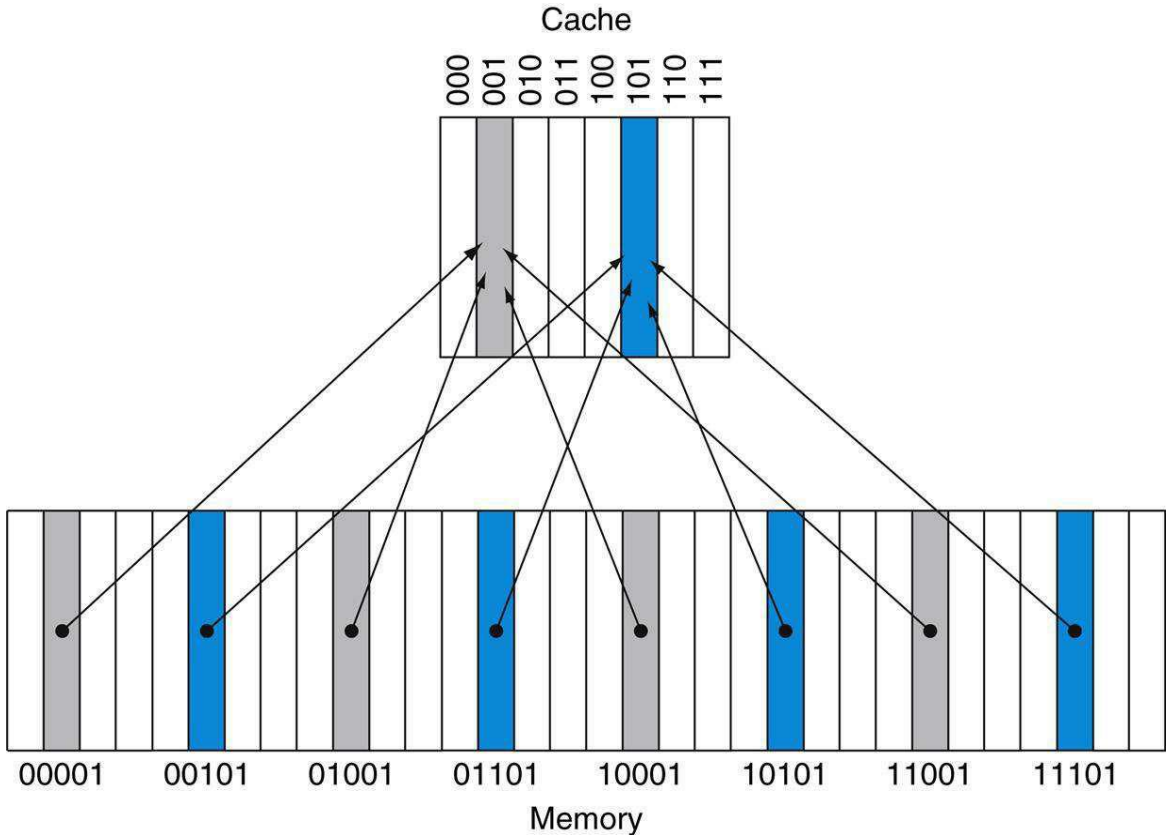
- Defines where blocks can be placed in a cache



Source: Computer Architecture, A Quantitative Approach by John L. Hennessy and David A. Patterson

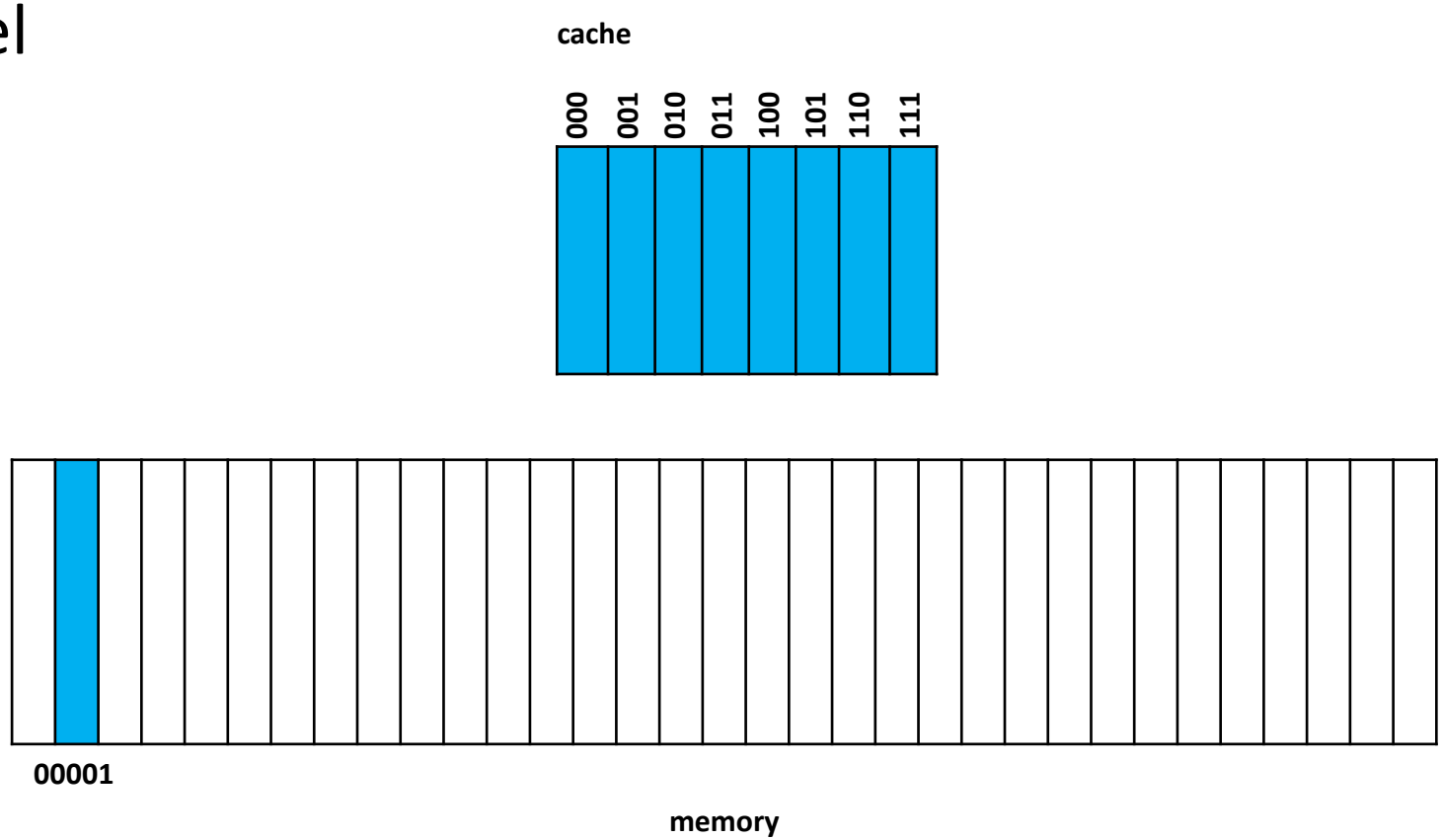
Direct mapping

- A memory block can go exactly to one place in cache



Fully associative

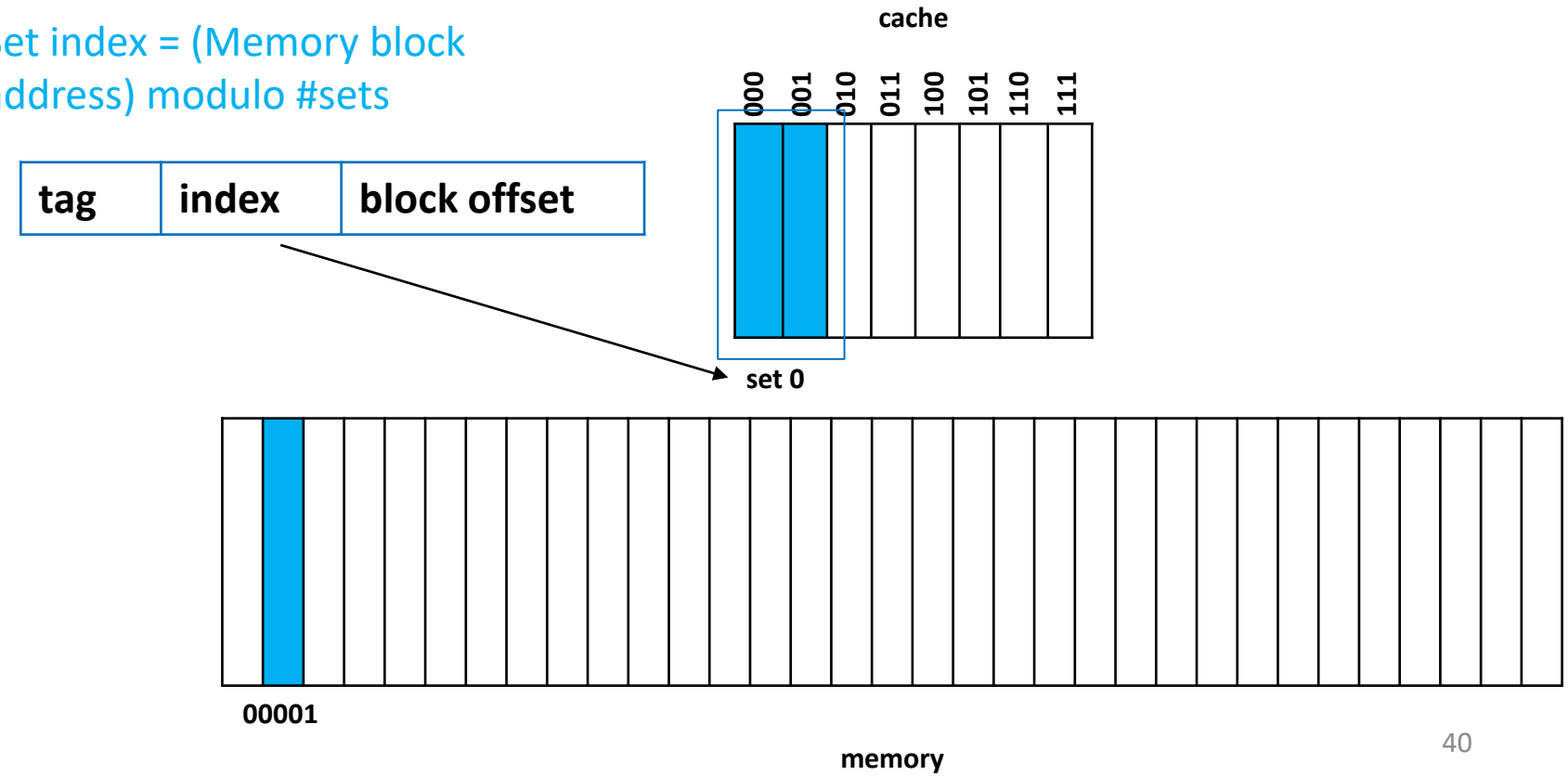
- A block in main memory can go to any block in cache
- Searched parallel



Set associative

- A block in main memory can go to any block in a set of cache blocks
- *n-way set associative*: *n* blocks for a set

Set index = (Memory block address) modulo #sets



Cache configuration examples

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

Cache Replacement Policies

- When cache is full some of the cached blocks need to be removed before bringing new ones in
  - If cached blocks are dirty (written/updated), then they need to be written to RAM
- Cache replacement policies
  - Random
  - Least Recently Used (LRU)
    - Need to track last access time
  - Least Frequently Used (LFU)
    - Need to track no of accesses
  - First In First Out (FIFO)

Increasing Cache Performance

- Large cache capacity
- Multiple-levels of cache
- Prefetching
  - a block of data is brought into the cache before it is actually referenced
- Fully associative cache

Thank you