Chapter 4 - Cache Memory

Luis Tarrataca
luis.tarrataca@gmail.com

CEFET-RJ

Table of Contents I

Table of Contents I

Table of Contents II

Table of Contents I

Introduction

Remember this guy? Why was he famous for?

- John von Neumann;
- Hungarian-born scientist;
- Manhattan project;
- von Neumann Architecture:
  - CPU ;
  - Memory;
  - I/O Module

Today's focus: memory module of von Neumann's architecture.

- Why may you ask?
  - Because that is the order that your book follows =P

Although simple in concept computer memory exhibits wide range of:

- type;
- technology;
- organization;
- performance;
- and cost.

No single technology is optimal in satisfying all of these...

Typically:

- Higher performance $\rightarrow$ higher cost;
- Lower performance $\rightarrow$ lower cost;

Typically:

- Higher performance $\Rightarrow$ higher cost;
- Lower performance $\Rightarrow$ lower cost;

What to do then? Any ideas?

Typically, a computer has a **hierarchy** of memory subsystems:

- some internal to the system
  - *i.e.* directly accessible by the processor;
- some external
  - accessible via an I/O module;

Typically, a computer has a hierarchy of memory subsystems:

- some internal to the system
  - *i.e.* directly accessible by the processor;
- some external
  - accessible via an I/O module;

Can you see any advantages / disadvantages with using each one?

Computer Memory System Overview

Classification of memory systems according to their key characteristics:

Location
- Internal (e.g., processor registers, cache, main memory)
- External (e.g., optical disks, magnetic disks, tapes)

Capacity
- Number of words
- Number of bytes

Unit of Transfer
- Word
- Block

Access Method
- Sequential
- Direct
- Random
- Associative

Performance
- Access time
- Cycle time
- Transfer rate

Physical Type
- Semiconductor
- Magnetic
- Optical
- Magneto-optical

Physical Characteristics
- Volatile/nonvolatile
- Erasable/nonerasable

Organization
- Memory modules

Figure: Key Characteristics Of Computer Memory Systems (Source: (Stallings, 2015))

Lets see if you can guess what each one of these signifies... Any ideas?

- **Location:** either internal or external to the processor.
  - Forms of internal memory:
    - registers;
    - cache;
    - and others;
  - Forms of external memory:
    - disk;
    - magnetic tape (too old... =P);
    - devices that are accessible to the processor via I/O controllers.

- **Capacity:** amount of information the memory is capable of holding.
  - Typically expressed in terms of bytes (1 byte = 8 bits) or **words**;
  - A word represents each addressable block of the memory
    - common word lengths are 8, 16, and 32 bits;
  - External memory capacity is typically expressed in terms of bytes;

- **Unity of transfer:** number of bytes read / written into memory at a time.
  - Need not equal a word or an addressable unit;
  - Also possible to transfer **blocks**:
    - Sets of words;
    - Used in external memory...
    - External memory is slow...
    - **Idea:** minimize number of acesses, optimize amount of data transfer;

- **Access Method:** How are the units of memory accessed?
  - **Sequential Method:** Memory is organized into units of data, called records.
    - Access must be made in a specific linear sequence;
    - Stored addressing information is used to assist in the retrieval process.
    - A shared read-write head is used;
    - The head must be moved from its one location to the another;
    - Passing and rejecting each intermediate record;
    - Highly variable times.

Figure: Sequential Method Example: Magnetic Tape

- **Access Method:** How are the units of memory accessed?
  - **Direct Access Memory:**
    - Involves a shared read-write mechanism;
    - Individual records have a unique address;
    - Requires accessing general record vicinity plus sequential searching, counting, or waiting to reach the final location;
    - Access time is also variable;

Figure: Direct Access Memory Example: Magnetic Disk

- **Access Method:** How are the units of memory accessed?
  - **Random Access:** Each addressable location in memory has a unique, physically wired-in addressing mechanism.
    - Constant time;
    - independent of the sequence of prior accesses;
    - Any location can be selected at random and directly accessed;
    - Main memory and some cache systems are random access.

- **Access Method:** How are the units of memory accessed?
  - **Associative:** RAM that enables one to make a comparison of desired bit locations within a word for a specified match
    - Word is retrieved based on a portion of its contents rather than its address;
    - Retrieval time is constant independent of location or prior access patterns
    - *E.g.:* neural networks.

- **Performance:**
  - **Access time (latency):**
    - For RAM: time to perform a read or write operation;
    - For Non-RAM: time to position the read-write head at desired location;
  - **Memory cycle time:** Primarily applied to RAM:
    - Access time + additional time required required before a second access;
    - Required for electrical signals to be terminated/regenerated;
    - Concerns the system bus.

- **Transfer time:** Rate at which data can be transferred in / out of memory;
  - For RAM: $\frac{1}{\text{cycle time}}$
  - For Non-RAM: $T_n = T_A + \frac{n}{R}$, where:
    - T_n: Average time to read or write n bits;
    - T_A: Average access time;
    - n: Number of bits
    - R: Transfer rate, in bits per second (bps)

- **Physical characteristics:**
  - **Volatile:** information decays naturally or is lost when powered off;
  - **Nonvolatile:** information remains without deterioration until changed:
    - no electrical power is needed to retain information.;
    - *E.g.:* Magnetic-surface memories are nonvolatile;
  - Semiconductor memory (memory on integrated circuits) may be either volatile or nonvolatile.

Now that we have a better understanding of key memory aspects:

- We can try to relate some of these dimensions...

Memory Hierarchy

Design constraints on memory can be summed up by three questions:

- **How much?**
  - If memory exists, applications will likely be developed to use it.
- **How fast?**
  - Best performance achieved when memory keeps up with the processor;
  - *I.e.* as the processor execute instructions, memory should minimize pausing / waiting for instructions or operands.
- **How expensive?**
  - Cost of memory must be reasonable in relationship to other components;

Memory tradeoffs are a sad part of reality =’(

- Faster access time, greater cost per bit;
- Greater capacity:
  - Smaller cost per bit;
  - Slower access time;

These tradeoff imply a **dilemma:**

- Large capacity memories are desired:
  - low cost and because the capacity is needed;
- However, to meet performance requirements, the designer needs:
  - to use expensive, relatively lower-capacity memories with short access times.

These tradeoff imply a **dilemma:**

- Large capacity memories are desired:
  - low cost and because the capacity is needed;
- However, to meet performance requirements, the designer needs:
  - to use expensive, relatively lower-capacity memories with short access times.

How can we solve this issue? Or at least mitigate the problem? Any ideas?

The way out of this dilemma:

- Don't rely on a single memory;
- Instead employ a memory hierarchy;
- **Supplement:**
  - smaller, more expensive, faster memories with...
  - ...larger, cheaper, slower memories;
- engineering FTW =)

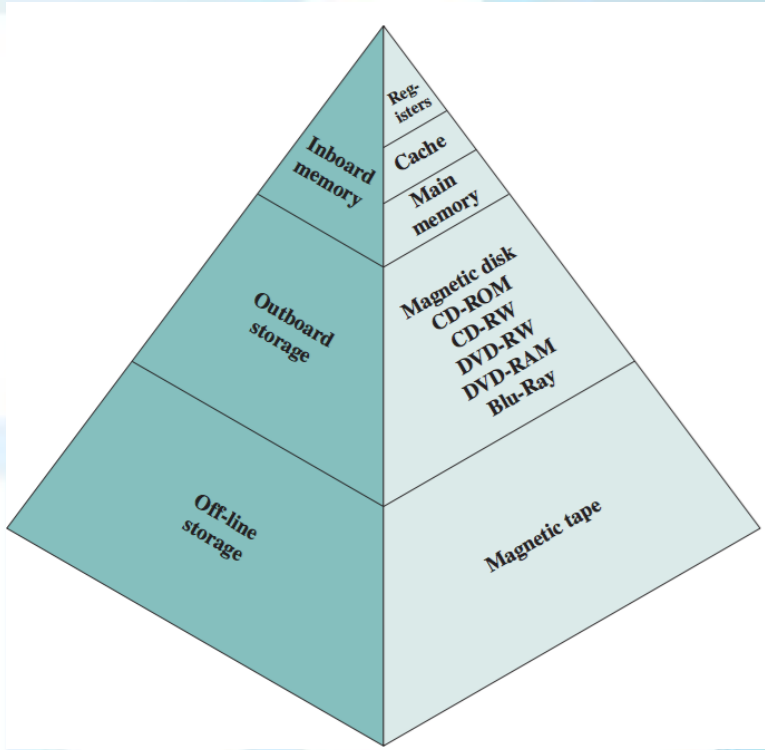

Figure: The memory hierarchy (Source: (Stallings, 2015))

The way out of this dilemma:

- As one goes down the hierarchy:
  - Decreasing cost per bit;
  - Increasing capacity;
  - Increasing access time;
  - Decreasing frequency of access of memory by processor

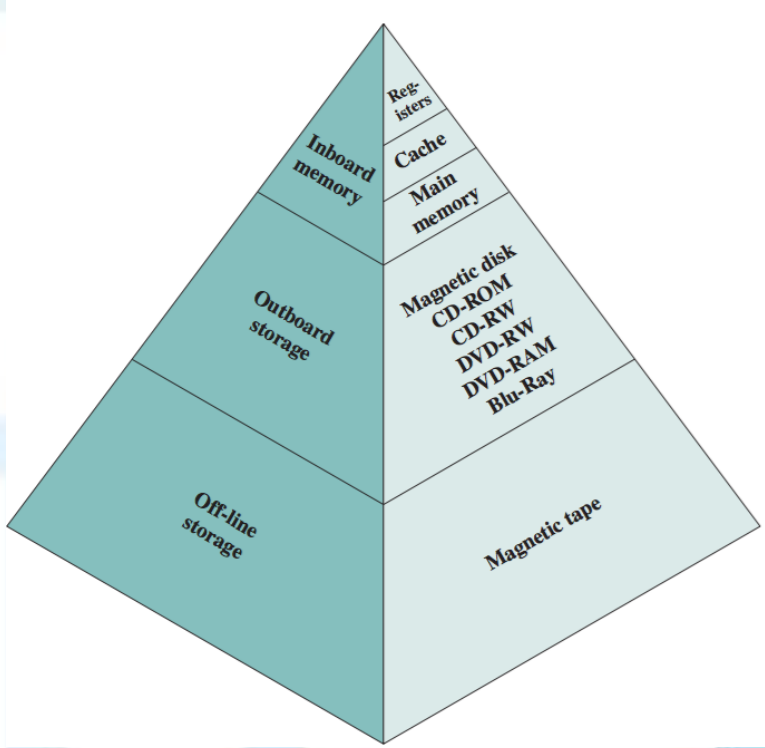

Figure: The memory hierarchy (Source: (Stallings, 2015))

Key to the success of this organization is the last item:

- Decreasing frequency of memory access by processor.

But why is this key to success? Any ideas?

Key to the success of this organization is the last item:

- Decreasing frequency of memory access by processor.

But why is this key to success? Any ideas?

- As we go down the hierarchy we gain in size but lose in speed;
- Therefore: not efficient for the processor to access these memories;
- Requires having specific strategies to minimize such accesses;

So now the question is...

How can we develop strategies to minimize these accesses? Any ideas?

How can we develop strategies to minimize these accesses? Any ideas?

Space and Time locality of reference principle:

- **Space:**
  - if we access a memory location, close by addresses will very likely be accessed;
- **Time:**
  - if we access a memory location, we will very likely access it again;

Space and Time locality of reference principle:

- **Space:**
  - if we access a memory location, close by addresses will very likely be accessed;
- **Time:**
  - if we access a memory location, we will very likely access it again;

But why does this happen? Any ideas?

Space and Time locality of reference principle:

- **Space:**
  - if we access a memory location, close by addresses will very likely be accessed;
- **Time:**
  - if we access a memory location, we will very likely access it again;

But why does this happen? Any ideas?

This a consequence of using iterative loops and subroutines:

- instructions and data will be accessed multiple times;

Example (1/5)

Suppose that the processor has access to two levels of memory:

- **Level 1 - L_1:**
  - contains 1000 words and has an access time of 0.01μs;
- **Level 2 - L_2:**
  - contains 100,000 words and has an access time of 0.1μs.
- Assume that:
  - if word $\in L_1$, then the processor accesses it directly;
  - If word $\in L_2$, then word is transferred to L_1 and then accessed by the processor.

Example (2/5)

For simplicity:

- ignore time required for processor to determine whether word is in L_1 or L_2.

Also, let:

- H define the fraction of all memory accesses that are found L1;
- T_1 is the access time to L1;
- T_2 is the access time to L2

Example (3/5)

General shape of the curve that covers this situation:

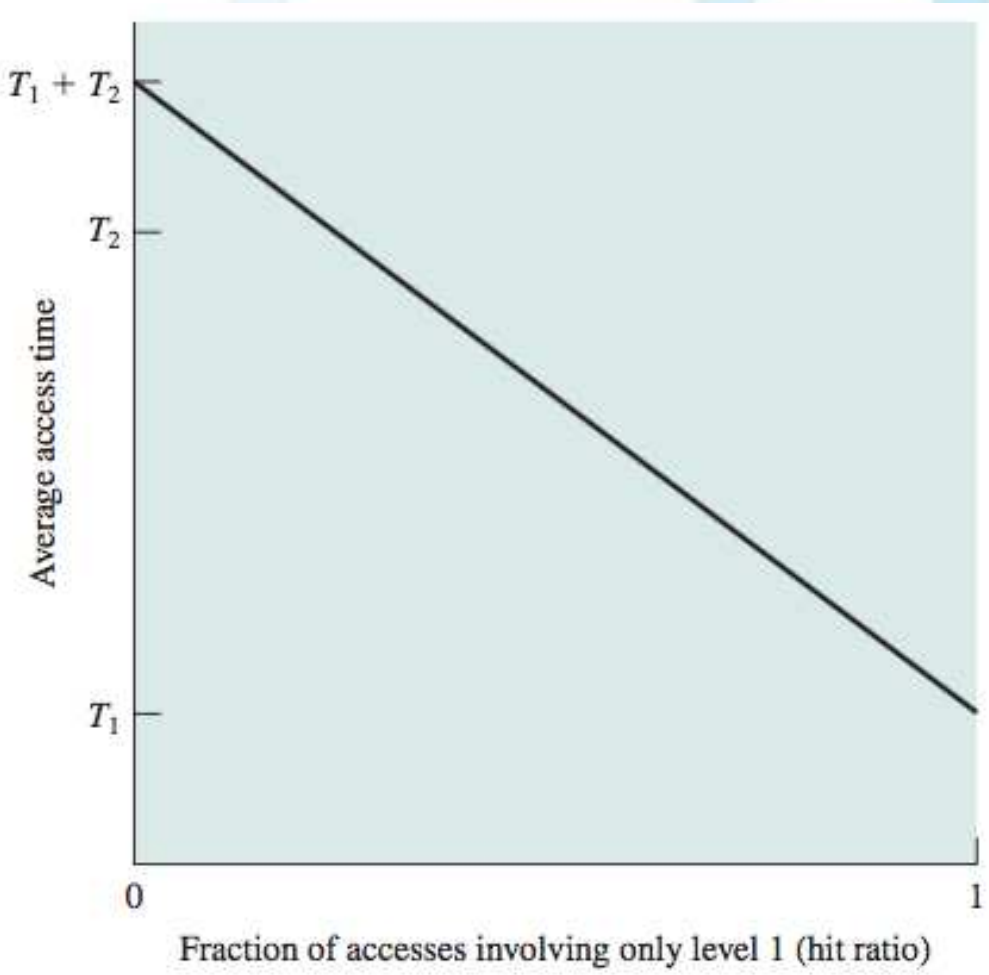

Figure: Performance of accesses involving only L1 (Source: (Stallings, 2015))

Example (4/5)

Textual description of the previous plot:

- For high percentages of L_1 access, the average total access time is much closer to that of L_1 than that of L_2;

Now lets consider the following scenario:

- Suppose 95% of the memory accesses are found in L_1.
- Average time to access a word is:

$$(0.95)(0.01\mu s) + (0.05)(0.01\mu s + 0.1\mu s) = 0.0095 + 0.0055 = 0.015\mu s$$

- Average access time is much closer to $0.01\mu s$ than to $0.1\mu s$, as desired.

Example (5/5)

Strategy to minimize accesses should be:

- Organize data across the hierarchy such that
  - % of accesses to lower levels is substantially less than that of upper levels
- *I.e.* L_2 memory contains all program instructions and data:
  - Data that is currently being used should be in L_1;
  - Eventually:
    - Data $\in L_1$ will be swapped to L_2 to make room for new data;
  - On average, most references will be to data contained in L_1.

This principle can be applied across more than two levels of memory:

- Processor registers:
  - Fastest, smallest, and most expensive type of memory
- Followed immediately by the cache:
  - Stages data movement between registers and main memory;
  - Improves perfomance;
  - Is not usually visible to the processor;
  - Is not usually visible to the programmer.
- Followed by main memory:
  - Principal internal memory system of the computer;
  - Each location has a unique address.

This means that we should maybe have a closer look at the cache =)

Guess what the next section is...

Cache Memory Principles

Cache memory is designed to combine (1/2):

- Memory access time of expensive, high-speed memory combined with...
- ...the large memory size of less expensive, lower-speed memory.

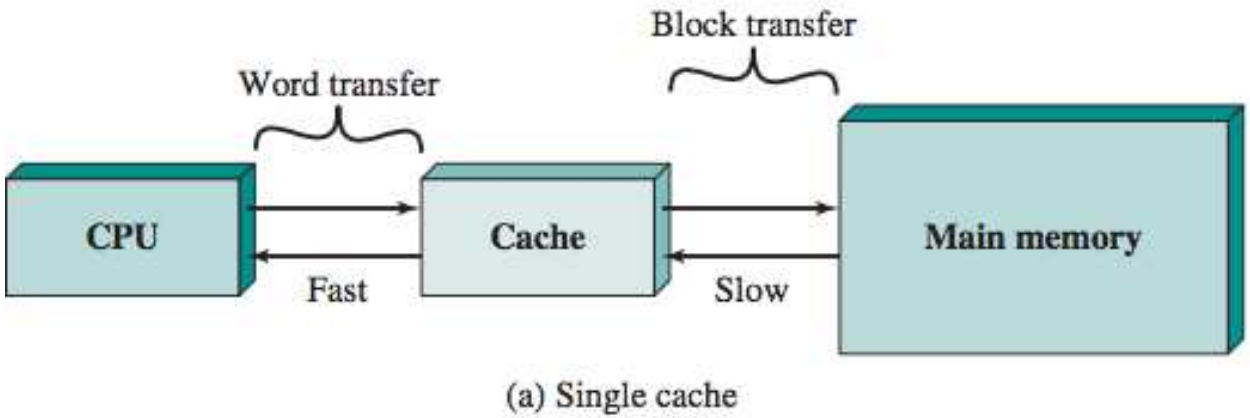

Figure: Cache and main memory - single cache approach (Source: (Stallings, 2015))

Cache memory is designed to combine (2/2):

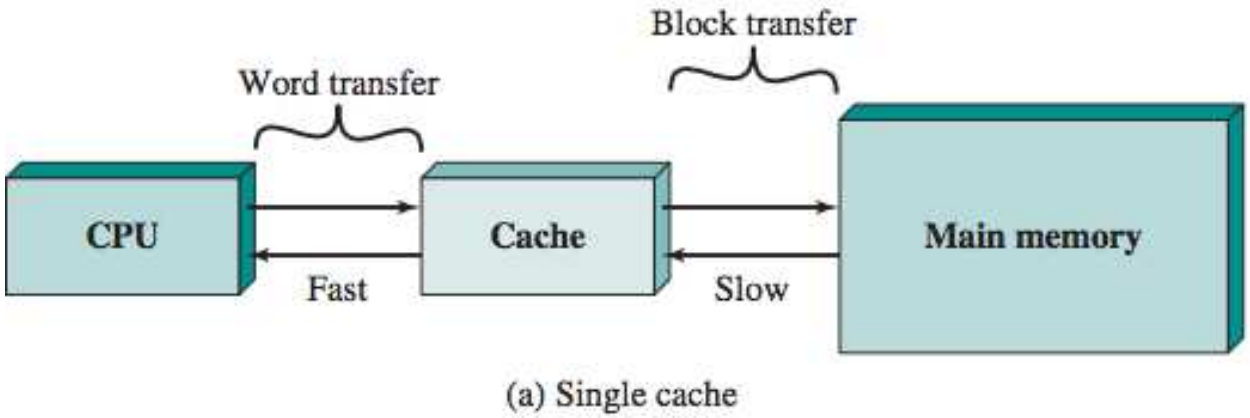

Figure: Cache and main memory - single cache approach (Source: (Stallings, 2015))

- Cache contains a copy of portions of main memory.

When the processor attempts to read a word of memory:

- Check is made to determine if the word is in the cache;
  - If so **(**Cache Hit): word is delivered to the processor.
  - If the word is not in cache **(Cache Miss)**:
    - Block of main memory is read into the cache;
    - Word is delivered to the processor.
- Because of the locality of reference principle:
  - When a block of data is fetched into the cache...
  - ...it is likely that there will be future references to that same memory location;

Can you see any way of improving the cache concept? Any ideas?

Can you see any way of improving the cache concept? Any ideas?

- What if we introduce multiple levels of cache?
  - L2 cache is slower and typically larger than the L1 cache
  - L3 cache is slower and typically larger than the L2 cache.

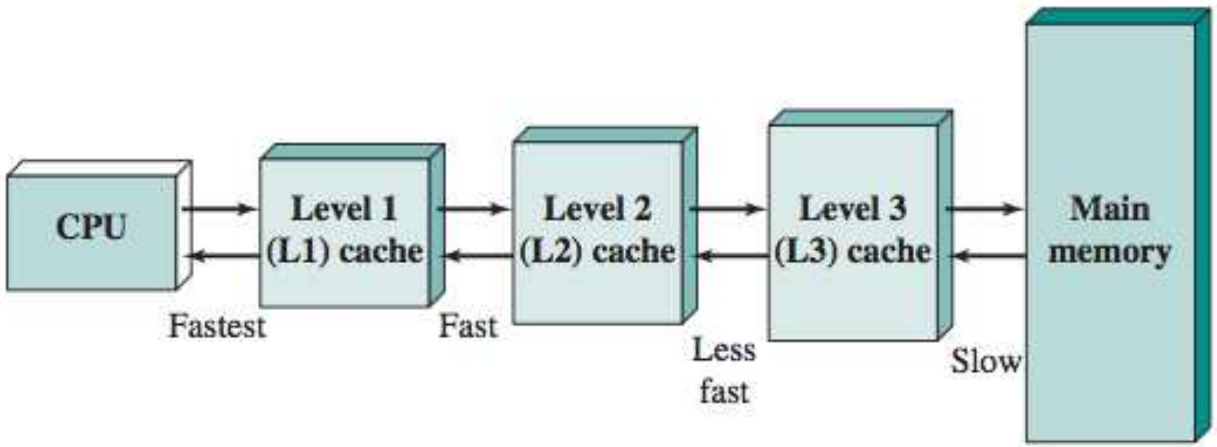

Figure: Cache and main memory - three-level cache organization (Source: (Stallings, 2015))

So, what is the structure of the main-memory system?

Main memory:

- Consists of 2^n addressable words;
- Each word has a unique n-bit address;
- Memory consists of a number of fixed-length blocks of K words each;
- There are $M = 2^n/K$ blocks;

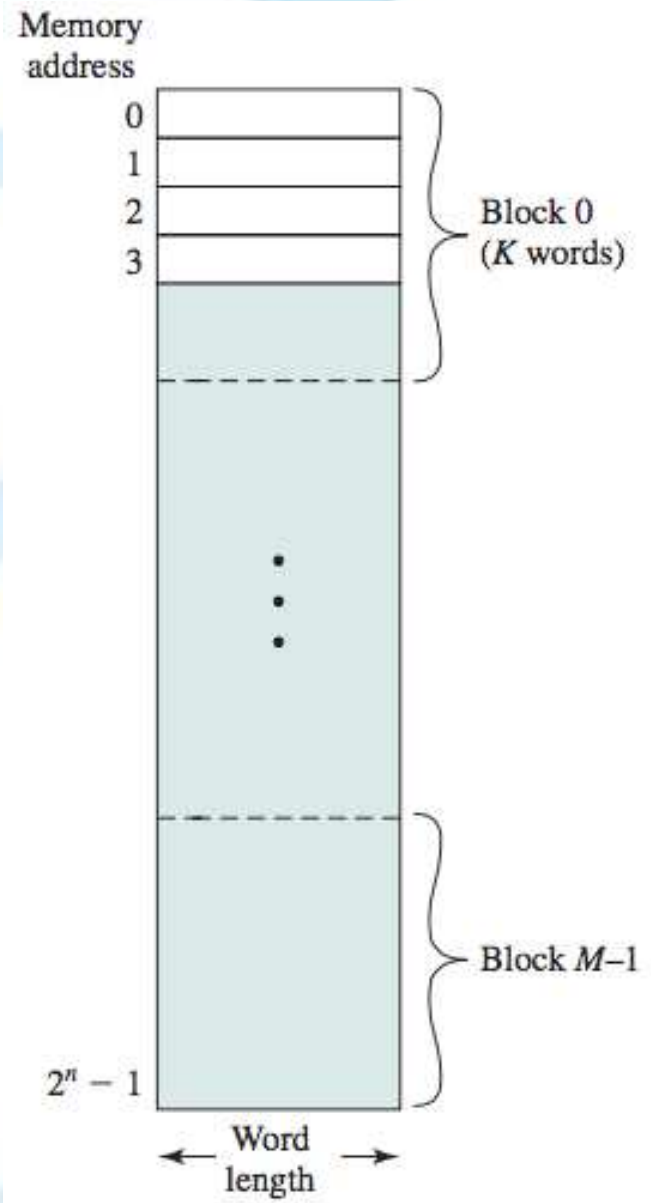

Figure: Main memory (Source:

So, what is the structure of the cache system?

Cache memory (1/2):

- Consisting of m blocks, called lines;
- Each line contains K words;
- $m \ll M$
- Each line also includes control bits:
  - Not shown in the figure;
  - MESI protocol (later chapter).

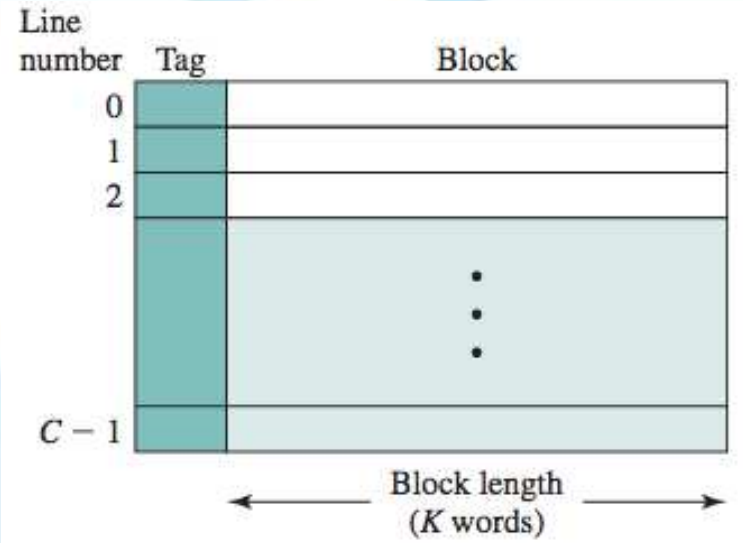

Figure: Cache memory (Source: (Stallings, 2015))

Cache memory (2/2):

- If a word in a block of memory is read:
  - Block is transferred to a cache line;
- Because $m \ll M$, lines:
  - Cannot permanently store a block.
  - Need to identify the block stored;
    - Info stored in the tag field;

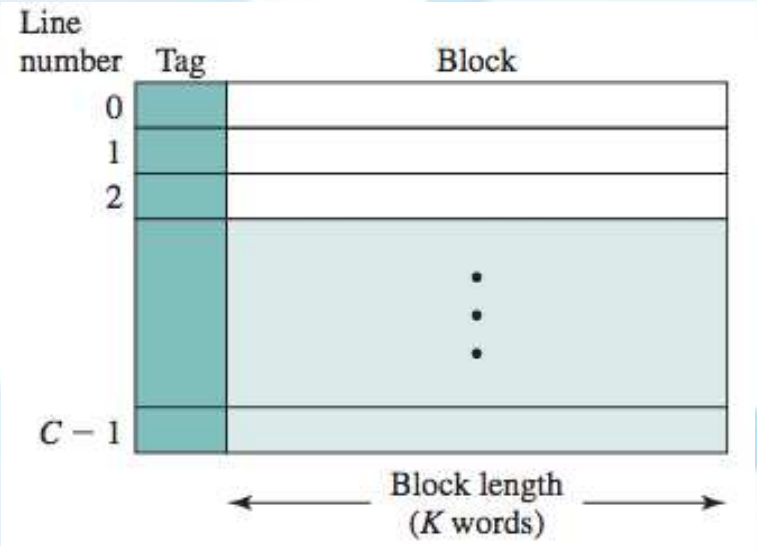

Figure: Cache memory (Source: (Stallings, 2015))

Now that we have a better understanding of the cache structure:

What is the specific set of operations that need to be performed for a read operation issued by the processor? Any ideas?

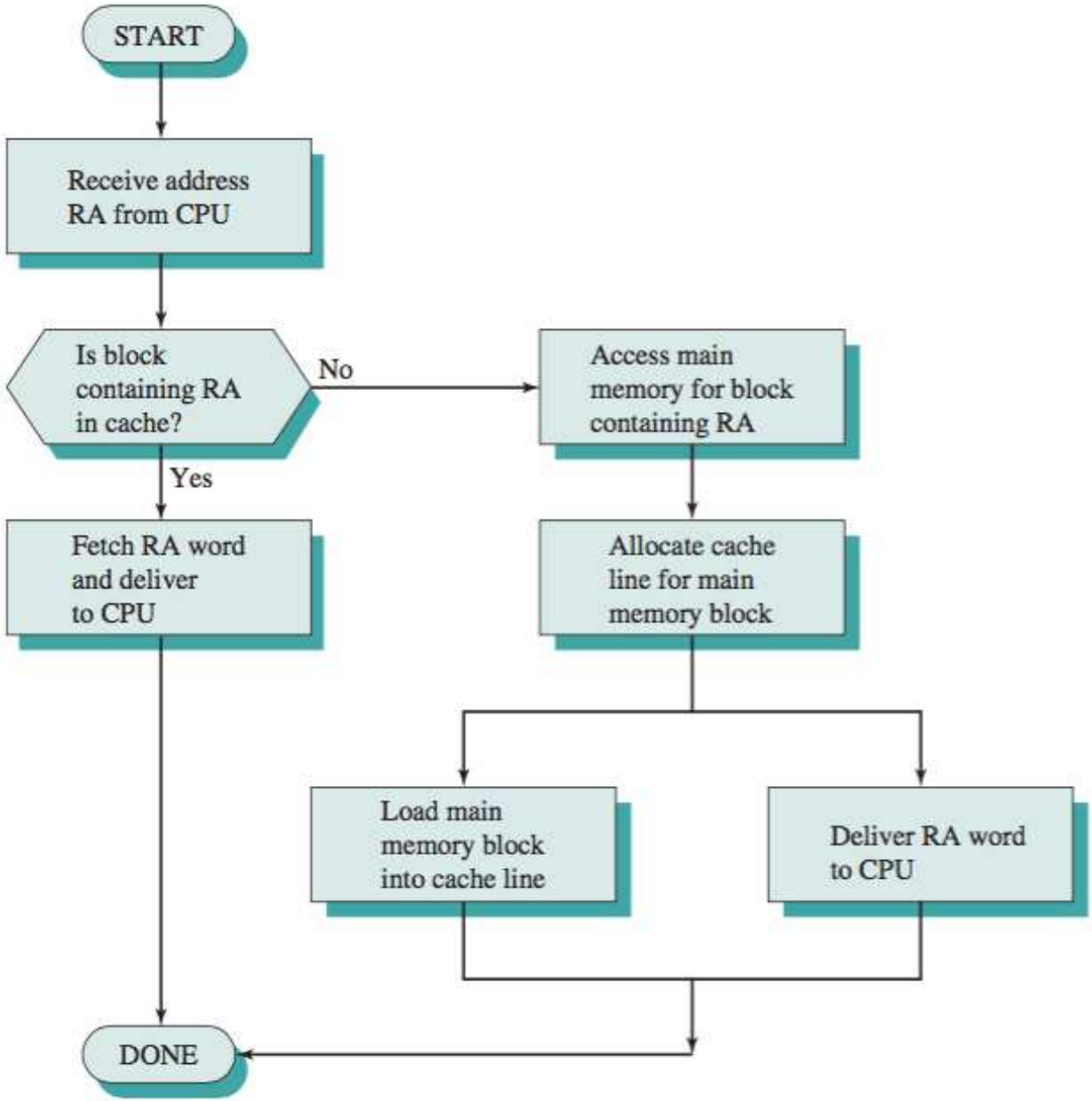

Figure: Cache read address **(RA)** (Source: (Stallings, 2015))

Read operation:

- Processor generates **read address (RA)** of word to be read;
- If the word $\in$ cache, it is delivered to the processor;
- Otherwise:
  - Block containing that word is loaded into the cache;
  - Word is delivered to the processor;
  - These last two operations occurring in parallel.

Typical contemporary cache organization:

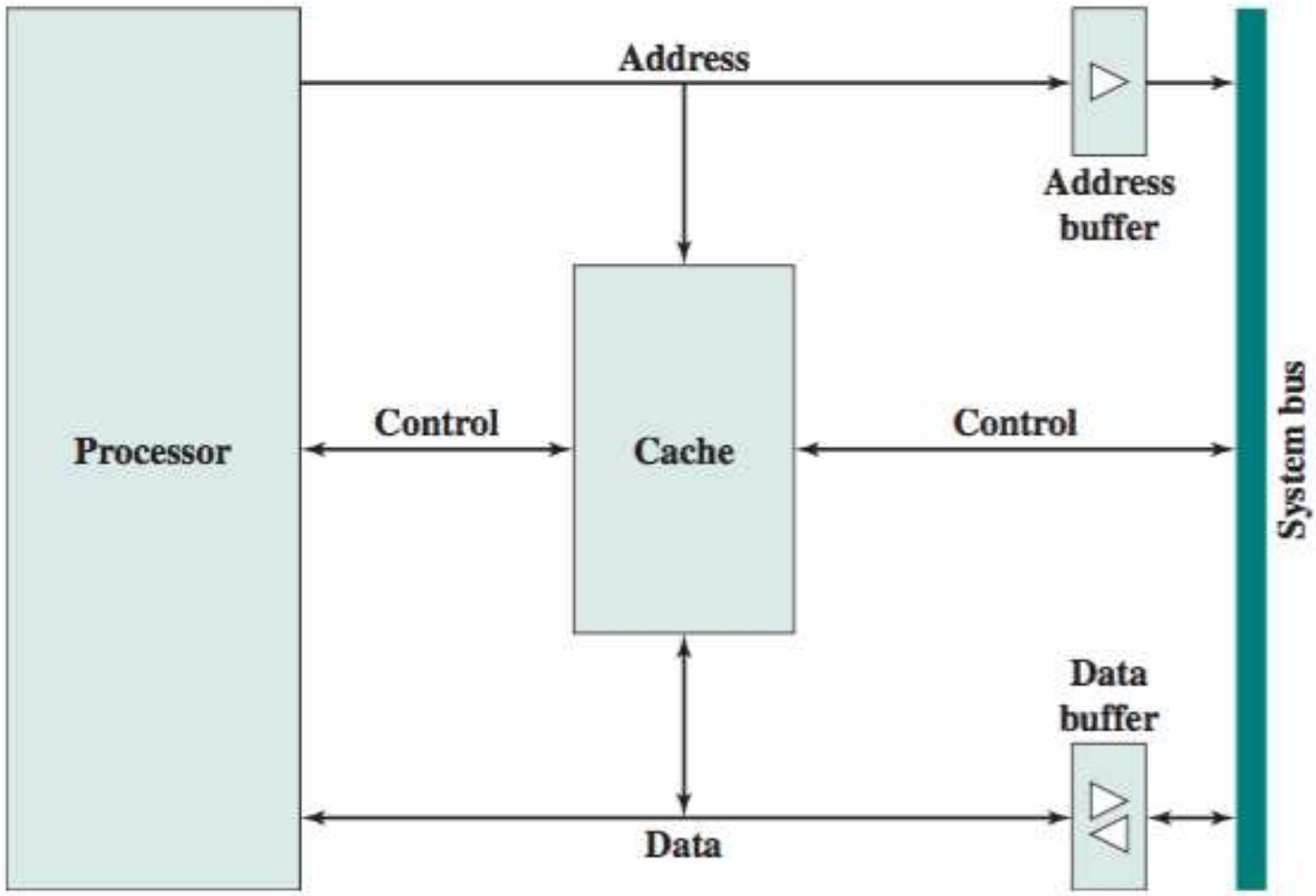

Figure: Typical cache organization (Source: (Stallings, 2015))

In this organization the **cache:**

- Connects to the processor via data, control, and address lines;
- Data and address lines also attach to data and address buffers:
  - Which attach to a system bus...
  - ...from which main memory is reached.

What do you think happens when a word is **in** cache? Any ideas?

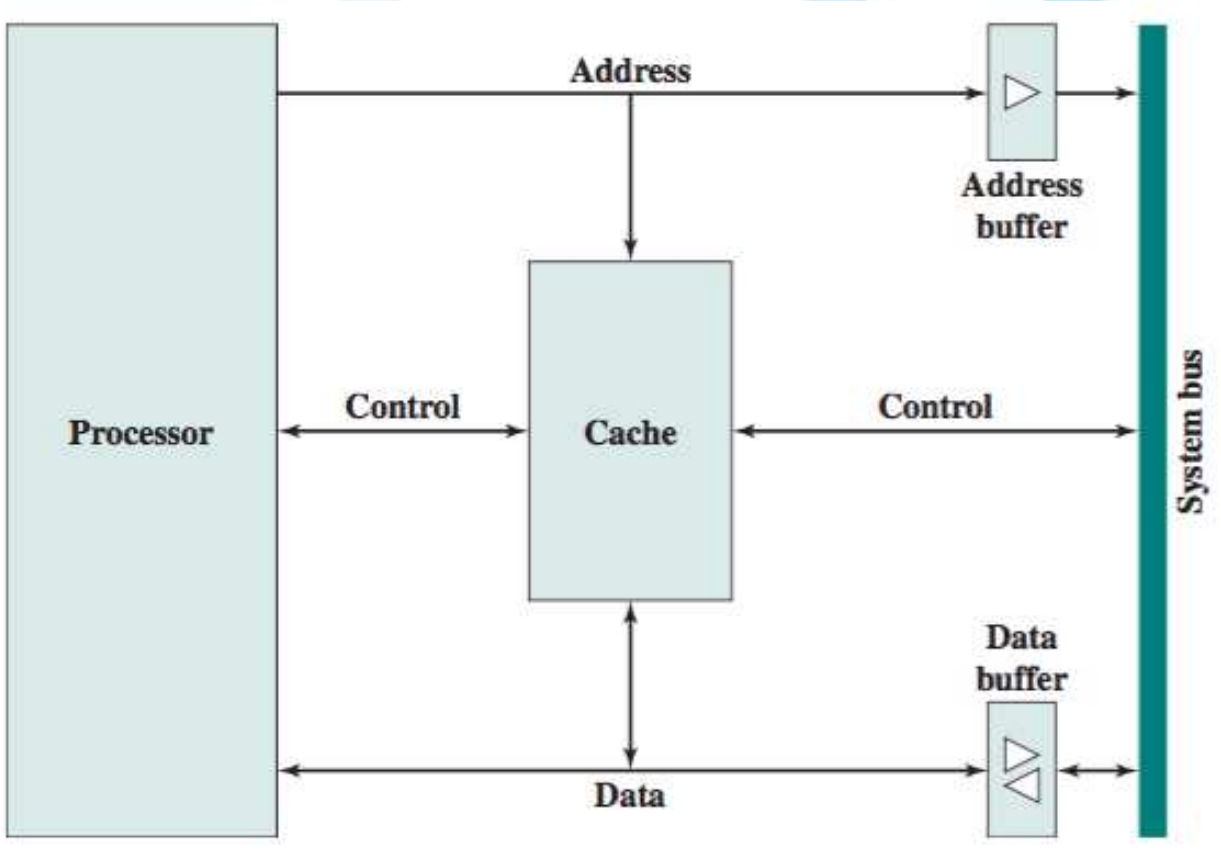

Figure: Typical cache organization (Source: (Stallings, 2015))

What do you think happens when a word is **in** cache? Any ideas?

When a **cache hit** occurs (word is in cache):

- the data and address buffers are disabled;
- communication is only between processor and cache;
- no system bus traffic.

What do you think happens when a word is **not in** cache? Any ideas?

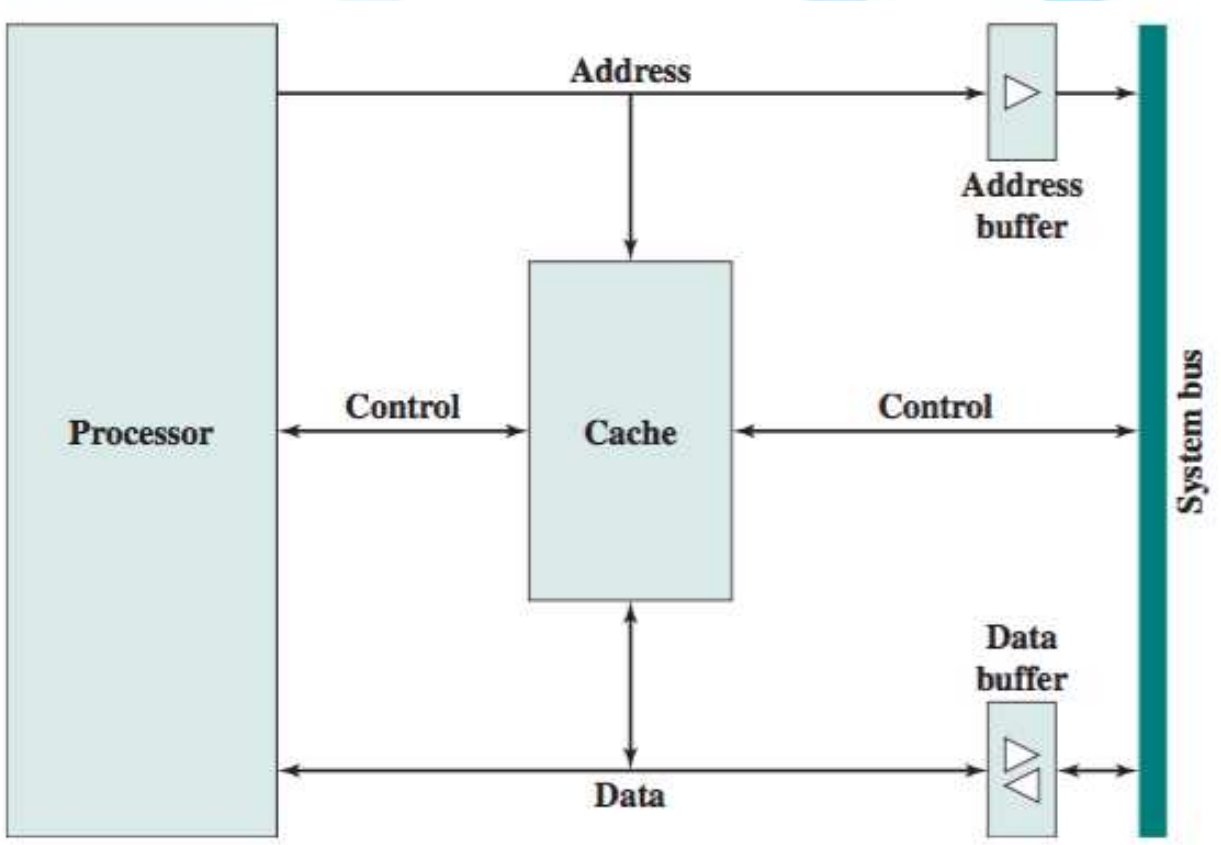

Figure: Typical cache organization (Source: (Stallings, 2015))

What do you think happens when a word is **not in** cache? Any ideas?

When a **cache miss** occurs (word is not in cache):

- the desired address is loaded onto the system bus;
- the data are returned through the data buffer...
- ...to both the cache and the processor

Elements of Cache Design

Cache architectures can be classified according to key elements:

Cache Addresses	**Write Policy**
Logical	Write through
Physical	Write back
Cache Size	**Line Size**
Mapping Function	**Number of Caches**
Direct	Single or two level
Associative	Unified or split
Set associative	
Replacement Algorithm	
Least recently used (LRU)	
First in first out (FIFO)	
Least frequently used (LFU)	
Random	

Figure: Elements of cache design (Source: (Stallings, 2015))

Cache Addresses

There are two types of cache addresses:

- **Physical addresses:**
  - Actual memory addresses;
- **Logical addresses:**
  - Virtual-memory addresses;

Cache Addresses

What is virtual memory?

Virtual memory performs mapping between:

- **Logical addresses** used by a program into **physical addresses**.
- Why is this important?
  - Virtual memory;
  - We will see in a later chapter...

Cache Addresses

Main idea behind virtual memory:

- Disregard amount of main memory available;
- Transparent transfers to/from:
  - main memory and...
  - ...secondary memory:
  - **Idea:** use RAM, when space runs out use HD ;)
- Requires a hardware memory management unit (MMU):
  - to translate virtual addresses into a physical addresses;

With virtual memory cache may be placed:

- between the processor and the MMU;

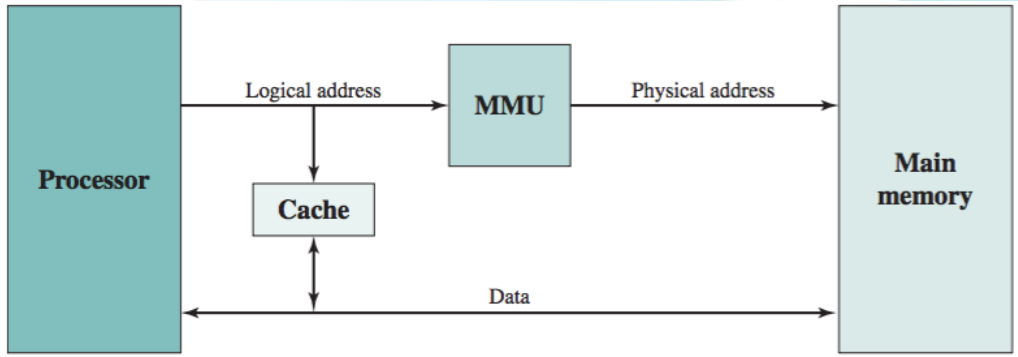

Figure: Virtual Cache (Source: (Stallings, 2015))

- between the MMU and main memory;

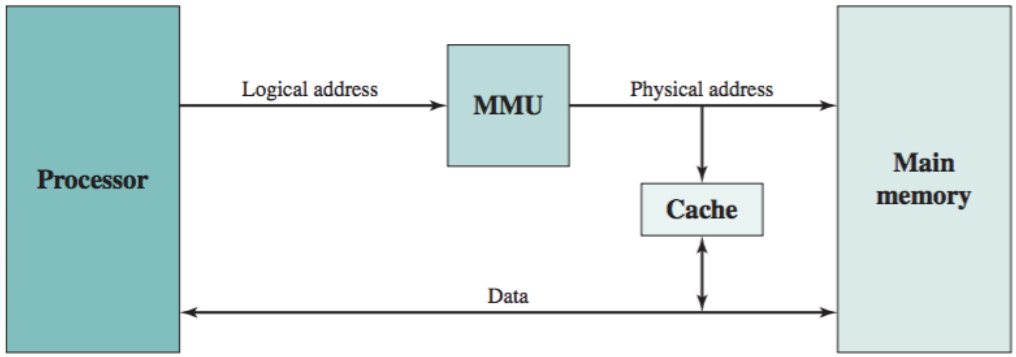

Figure: Physical Cache (Source: (Stallings, 2015))

What is the difference? Any ideas?

Virtual cache stores data using logical addresses.

- Processor accesses the cache directly, without going through the MMU.
- Advantage:
  - Faster access speed;
  - Cache can respond without the need for an MMU address translation;
- Disadvantage:
  - Same virtual address in two different applications refers to two different physical addresses;
  - Therefore cache must be flushed with each application context switch...
  - ...or extra bits must be added to each cache line
    - to identify which virtual address space this address refers to.

Cache Size

What about cache size? What can be said? Any ideas?

Cache Size

Cache size should be:

- Small enough so that overall:
  - **Average cost per bit** is close to that of main memory alone;
- Large enough so that the overall
  - **Average access time** is close to that of the cache alone;

The larger the cache, the more complex the addressing logic:

- Result: large caches tend to be slightly slower than small ones

Available chip and board area also limits cache size.

Conclusion: It is impossible to arrive at a single ``optimal'' cache size.

- as illustrated by the table in the next slide...

Processor	Type	Year of Introduction	L1 Cache[a]	L2 Cache	L3 Cache
IBM 360/85	Mainframe	1968	16–32 kB	—	—
PDP-11/70	Minicomputer	1975	1 kB	—	—
VAX 11/780	Minicomputer	1978	16 kB	—	—
IBM 3033	Mainframe	1978	64 kB	—	—
IBM 3090	Mainframe	1985	128–256 kB	—	—
Intel 80486	PC	1989	8 kB	—	—
Pentium	PC	1993	8 kB/8 kB	256–512 kB	—
PowerPC 601	PC	1993	32 kB	—	—
PowerPC 620	PC	1996	32 kB/32 kB	—	—
PowerPC G4	PC/server	1999	32 kB/32 kB	256 kB to 1 MB	2 MB
IBM S/390 G6	Mainframe	1999	256 kB	8 MB	—
Pentium 4	PC/server	2000	8 kB/8 kB	256 kB	—
IBM SP	High-end server/ supercomputer	2000	64 kB/32 kB	8 MB	—
CRAY MTA[b]	Supercomputer	2000	8 kB	2 MB	—
Itanium	PC/server	2001	16 kB/16 kB	96 kB	4 MB
Itanium 2	PC/server	2002	32 kB	256 kB	6 MB
IBM POWER5	High-end server	2003	64 kB	1.9 MB	36 MB
CRAY XD-1	Supercomputer	2004	64 kB/64 kB	1 MB	—
IBM POWER6	PC/server	2007	64 kB/64 kB	4 MB	32 MB
IBM z10	Mainframe	2008	64 kB/128 kB	3 MB	24–48 MB
Intel Core i7 EE 990	Workstation/ server	2011	6 × 32 kB/ 32 kB	1.5 MB	12 MB
IBM zEnterprise 196	Mainframe/ server	2011	24 × 64 kB/ 128 kB	24 × 1.5 MB	24 MB L3 192 MB L4

Mapping Function

Recall that there are fewer cache lines than main memory blocks

How should one map main memory blocks into cache lines? Any ideas?

Three techniques can be used for mapping blocks into cache lines:

- Direct;
- Associative;
- Set associative

Lets have a look into each one of these...

- I know that you like when we go into specific details ;)

Direct Mapping

Maps each block of main memory into only one possible cache line as:

$$i = j \mod m$$

where:

- i = cache line number;
- j = main memory block number;
- m = number of lines in the cache

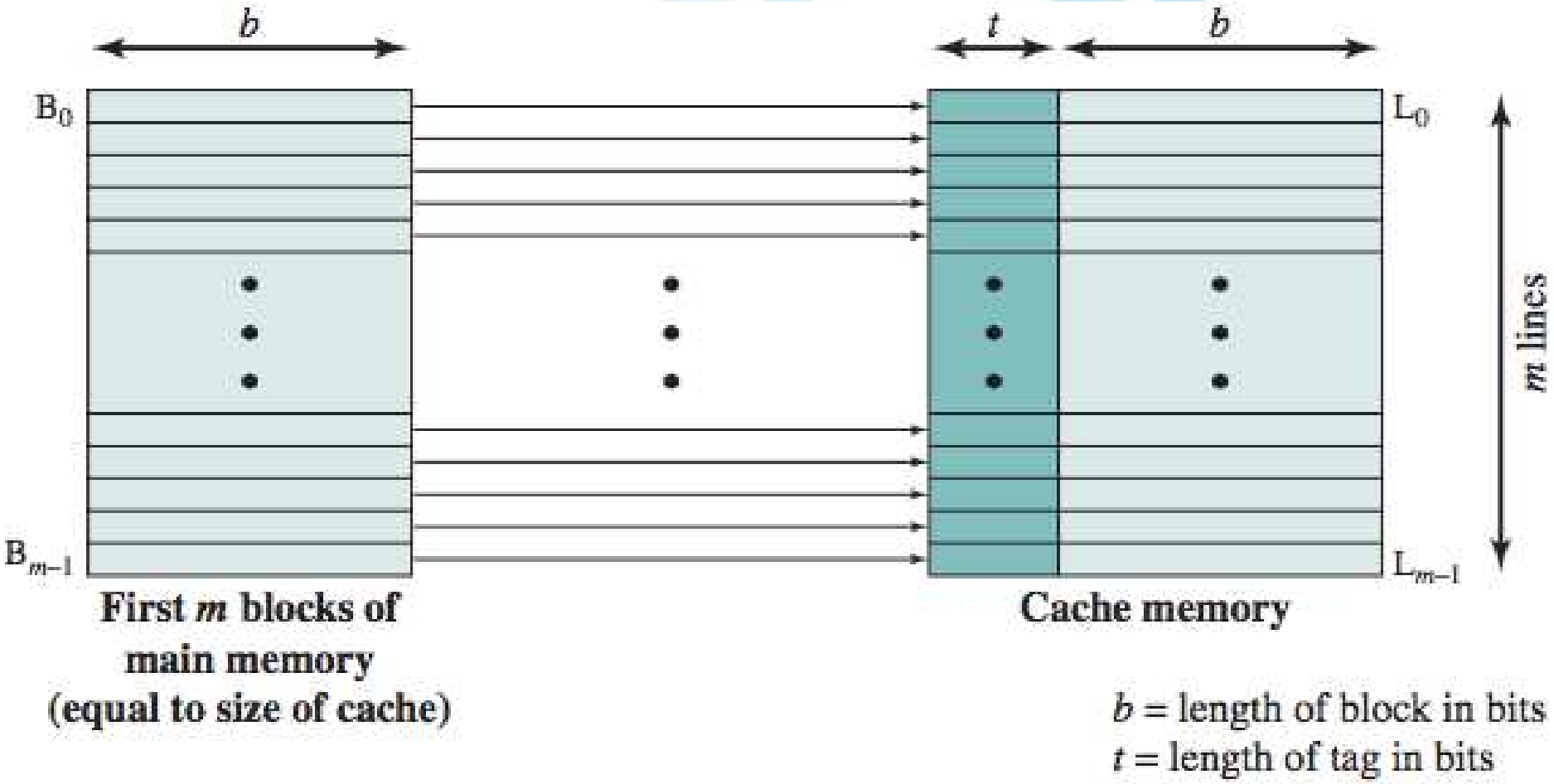

Figure: Direct mapping (Source: (Stallings, 2015))

Previous picture shows mapping of main memory blocks into cache:

- First m main memory blocks map into each line of the cache;
- Next m blocks of main memory map in the following manner:
  - B_m maps into line L_0 of cache;
  - B_{m+1} maps into line L_1;
  - and so on...
- Modulo operation implies repetitive structure;

With direct mapping blocks are assigned to lines as follows:

Cache line	Main memory blocks assigned
0	$0, m, 2m, \ldots, 2^s - m$
1	$1, m+1, 2m+1, \ldots, 2^s - m + 1$
$\vdots$	$\vdots$
$m-1$	$m-1, 2m-1, 3m-1, \ldots, 2^s - 1$

Figure: (Source: (Stallings, 2015))

Over time:

- Each line can have a different main memory block;
- We need the ability to distinguish between these;
- Most significant bits, the **tag**, serve this purpose.

Each main **memory address** ($s + w$ bits) can be viewed as:

- **Block** (s bits): identifies the memory block;
- **Offset** (w bits): identifies a word within a block of main memory;

If the **cache** has 2^r lines ($m \ll M$):

- **Line** (r bits): specify one of the 2^r cache lines;
- **Tag** ($s - r$ bits): to distinguish blocks that are mapped to the same line;

Why does the tag field only required $s - r$ bits?

Why does the tag field only required $s - r$ bits?

- Cache lines $2^r \ll 2^s$ blocks of memory;
- No need for tag field to use s bits;
- Instead we can use $\log_2 \frac{2^s}{2^r} = s - r$ bits:
- See Slide 81:
  - Does the line contain the 1^{st} block that can be assigned?
  - Does the line contain the 2^{nd} block that can be assigned?
  - ...
  - Does the line contain the 2^{s-r} block that can be assigned?

To determine whether a block is in the cache:

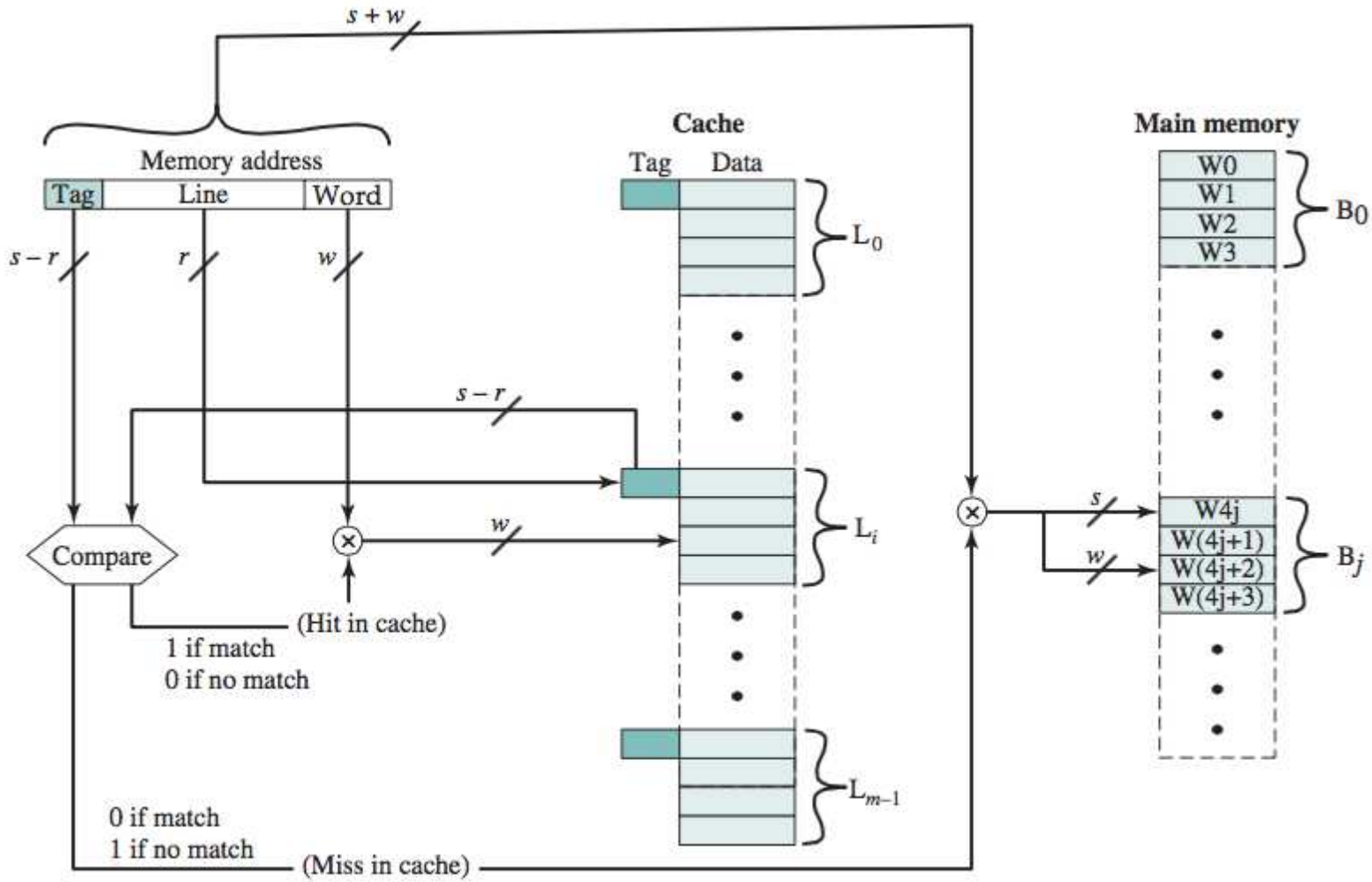

Figure: Direct mapping cache organization (Source: (Stallings, 2015))

To determine whether a block is in the cache:

1. Use the line field of the memory address to index the cache line;
2. Compare the tag from the memory address with the line tag;
    1. If **both match**, then **Cache Hit:**
        1. Use the line field of the memory address to index the cache line;
        2. Retrieve the corresponding word from the cache line;
    2. If **both do not match**, then **Cache Miss:**
        1. Use the line field of the memory address to index the cache line;
        2. Update the cache line (word + tag);

Direct mapping technique:

- **Advantage:** simple and inexpensive to implement;
- **Disadvantage:** there is a fixed cache location for any given block;
  - if a program happens to reference words repeatedly from two different blocks that map into the same line;
  - then the blocks will be continually swapped in the cache;
  - hit ratio will be low (a.k.a. **thrashing**).

Direct mapping is simple but problematic:

What would be a better mapping strategy? Any ideas?

Direct mapping is simple but problematic:

What would be a better mapping strategy? Any ideas?

- Associative mapping;
- Guess what we will be seeing next? ;)

Associative Mapping

Overcomes the disadvantage of direct mapping by:

- permitting each block to be loaded into any cache line:

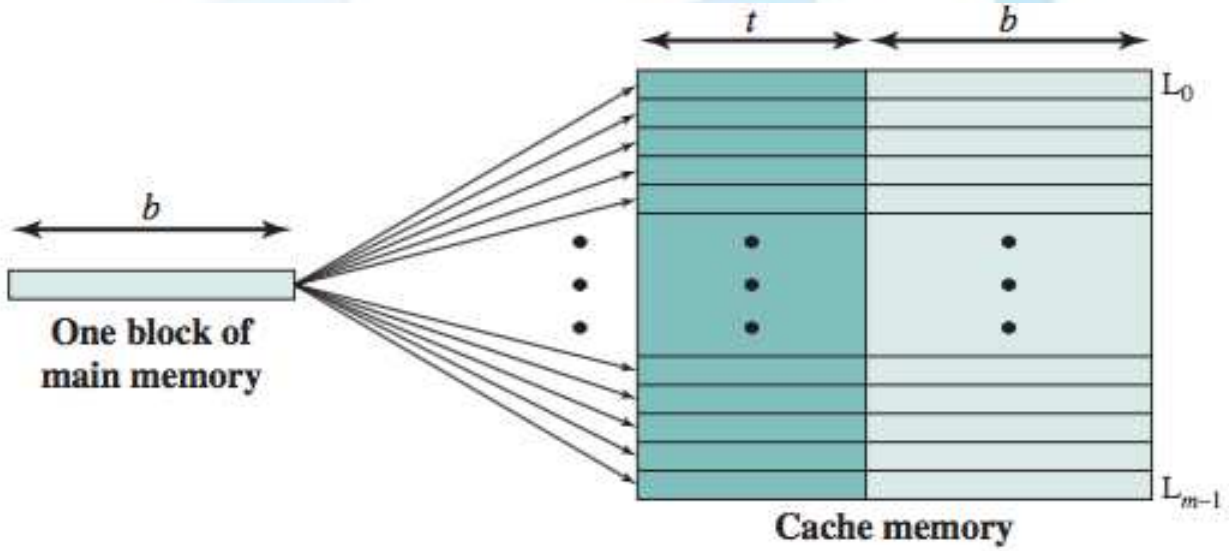

Figure: Associative Mapping (Source: (Stallings, 2015))

Cache interprets a memory address as a **Tag** and a **Word** field:

- **Tag:** (s bits) uniquely identifies a block of main memory;
- **Word:** (w bits) uniquely identifies a word within a block;

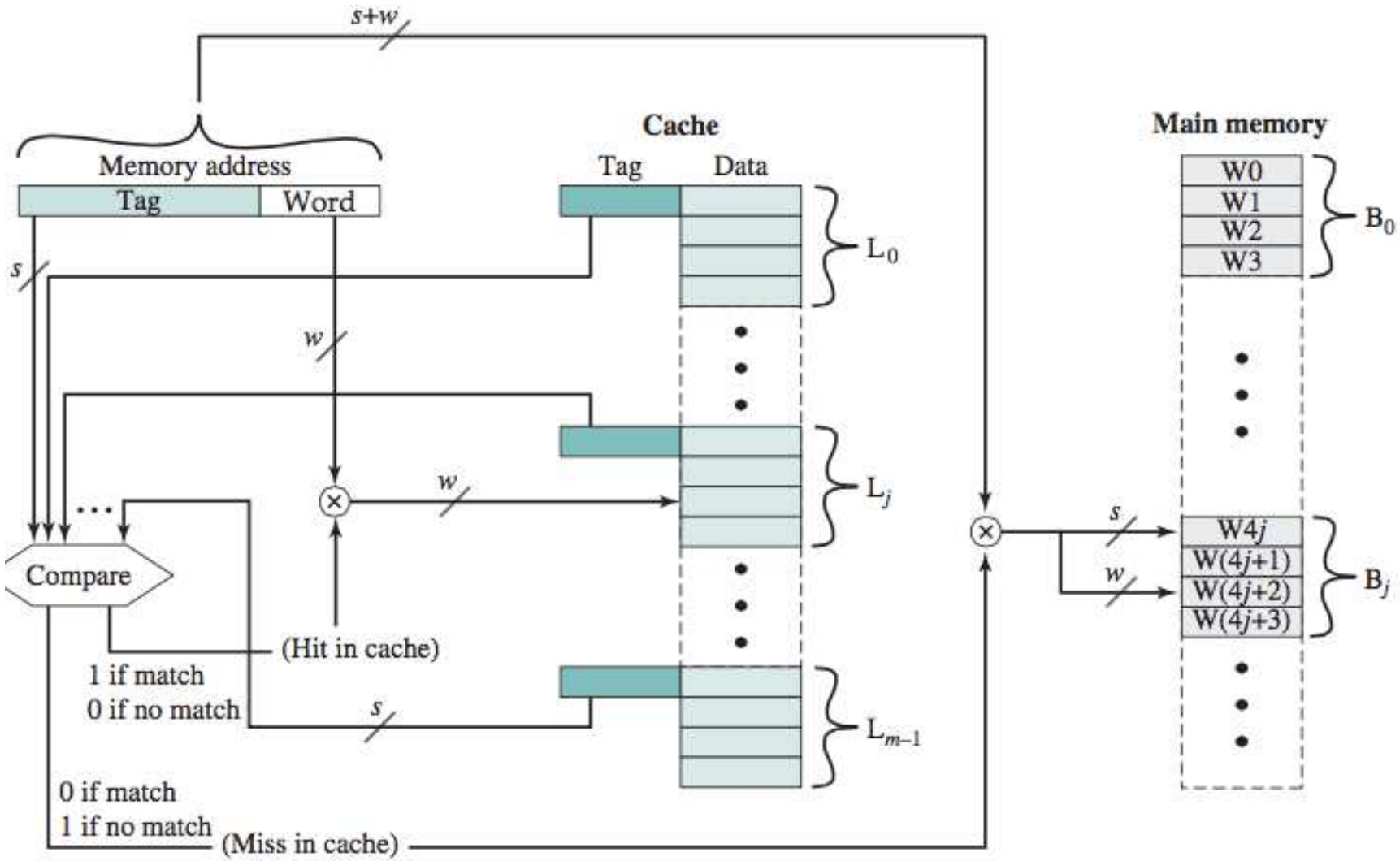

Figure: Fully associative cache organization (Source: (Stallings, 2015))

To determine whether a block is in the cache:

- simultaneously **compare** every line's tag for a match:
- If a **match exists**, then **Cache Hit:**
  1. Use the tag field of the memory address to index the cache line;
  2. Retrieve the corresponding word from the cache line;
- If a **match does not exist**, then **Cache Miss:**
  1. Choose a cache line. How?
  2. Update the cache line (word + tag);

What is the main **advantage** of associative mapping? Any ideas?

What is the main **advantage** of associative mapping? Any ideas?

- Flexibility as to which block to replace when a new block is read into the cache;

What is the main **disadvantage** of associative mapping? Any ideas?

What is the main **disadvantage** of associative mapping? Any ideas?

- Complex circuitry required to examine the tags of **all** cache lines in parallel.

Can you see any way of improving the associative scheme? Any ideas?

Can you see any way of improving the associative scheme? Any ideas?

Idea: **Perform less comparisons**

- Instead of comparing the tag against all lines
- Compare only against a subset of the cache lines.
- Welcome to set-associative mapping =)

Set-associative mapping

Combination of direct and associative approaches:

- Cache consists of a number of sets, each consisting of a number of lines.
- From **direct mapping:**
  - each block can only be mapped into a single set;
  - *I.e.* Block B_j always maps to set j;
  - Done in a modulo way =)
- From **associative mapping:**
  - each block can be mapped into any cache line of a certain set.

- The relationships are:

$$m = v \times k$$
$$i = j \mod v$$

where:

- i = cache set number;
- j = main memory block number;
- m = number of lines in the cache;
- v = number of sets;
- k = number of lines in each set

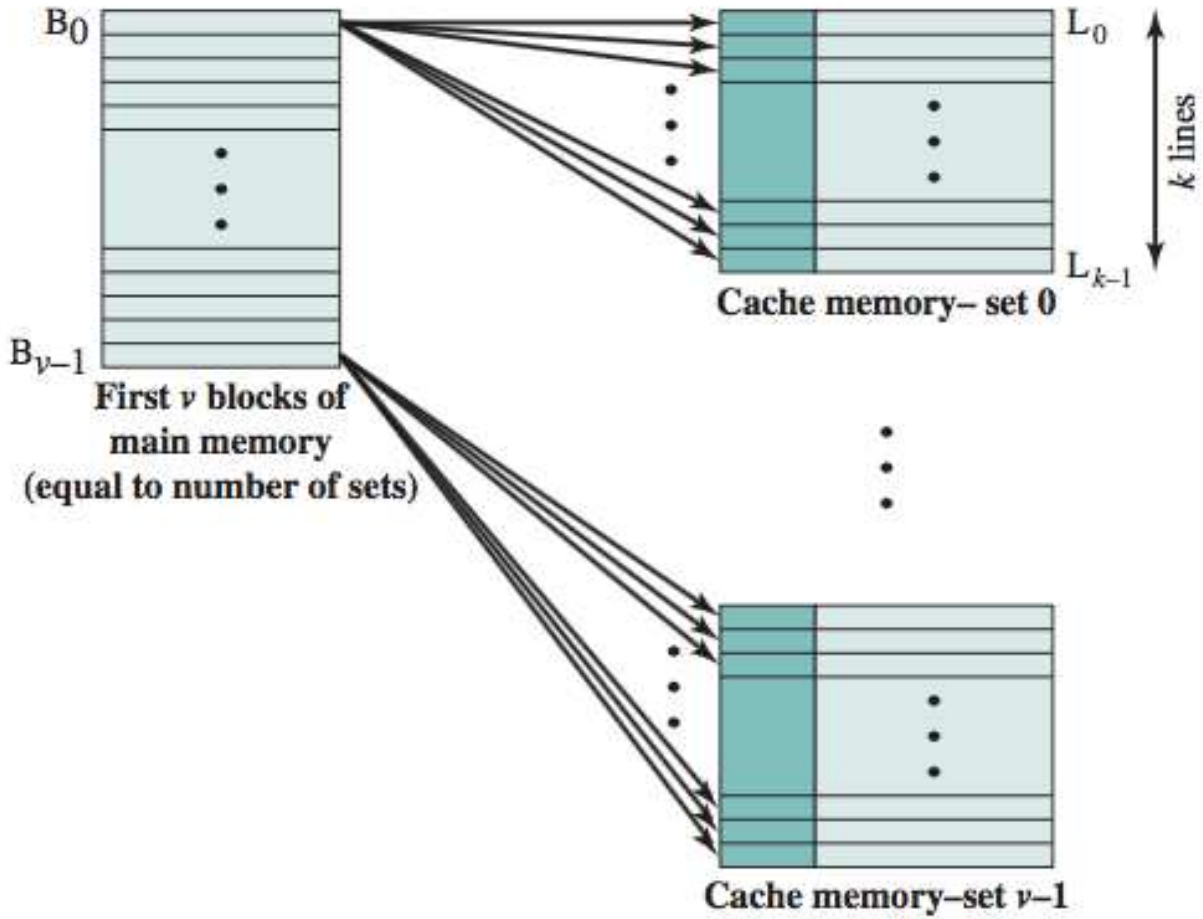

Figure: v associative mapped caches (Source: (Stallings, 2015))

Idea:

- 1 memory block $\longrightarrow$ 1 single set, but to any row of that set.
- can be physically implemented as v associative caches

Cache interprets a memory address as a **Tag**, a **Set** and a **Word** field:

- **Set:** identifies a set (d bits, $v = 2^d$ sets);
- **Tag:** used in conjunction with the set bits to identify a block ($s - d$ bits);
- **Word:** identifies a word within a block;

To determine whether a block is in the cache:

1. Determine the **set** through the set fields;
2. Compare address tag simultaneously with all cache line tags;
3. If a **match exists**, then **Cache Hit:**
    1. Retrieve the corresponding word from the cache line;
4. If a **match does not exist**, then **Cache Miss:**
    1. Choose a cache line within the set. How?
    2. Update the cache line (word + tag);

To determine whether a block is in the cache:

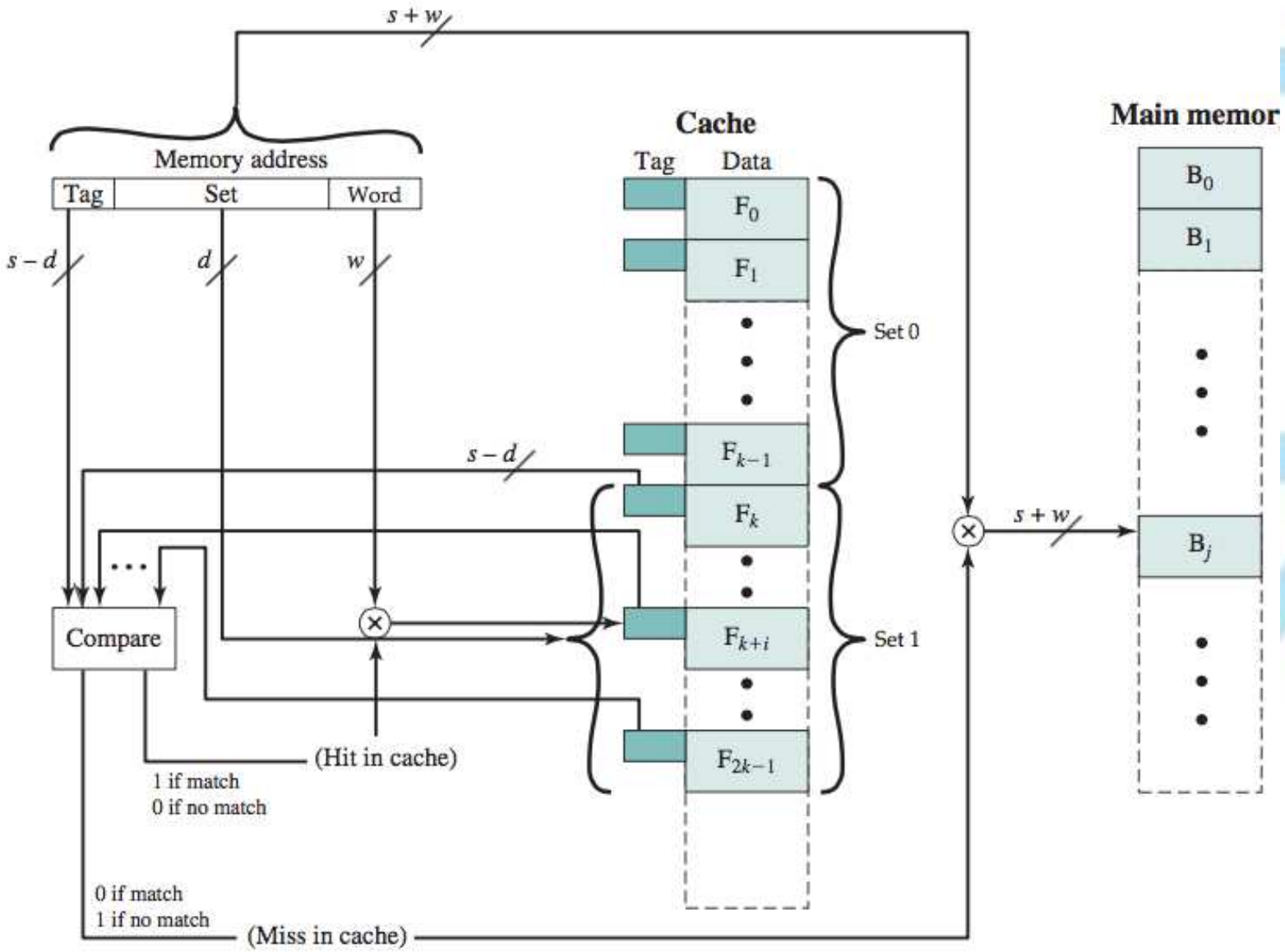

Figure: *K*-Way Set Associative Cache Organization (Source: (Stallings, 2015))

Exercise (1/4)

Consider a **set-associative** cache consisting of:

- 64 lines divided into four-line sets;
- Main memory contains 4K blocks of 128 words each;

Questions:

- How many bits are required for encoding words, sets and tag?
- What is the format of main memory addresses?

Exercise (2/4)

How many bits are required for the words? Any ideas?

Exercise (2/4)

How many bits are required for the words? Any ideas?

Each block contains 128 words:

- 7 bits are required to identify 128 words;

Exercise (3/4)

How many bits are required for the set? Any ideas?

Exercise (3/4)

How many bits are required for the sets? Any ideas?

Each set contains four lines:

- Cache has 64 lines in total;
- Therefore we need $\frac{64}{4} = 16$ sets;
- 4 bits are required to identify 16 sets;

Exercise (4/4)

How many bits are required for the tag? Any ideas?

Exercise (4/4)

How many bits are required for the tag? Any ideas?

Main memory contains 4K blocks:

- 12 bits are required to identify 4K blocks;
- Of these 12 bits, 4 bits are reserved for the set field;
- Therefore 8 bits are required for the tag field;

Hint: The specific details about these models would make great exam questions ;)

Ok, we saw a lot of details, but:

What happens with cache performance?

E.g.: How does the direct mapping compare against others?

E.g.: what happens when we vary the number of lines k in each set?

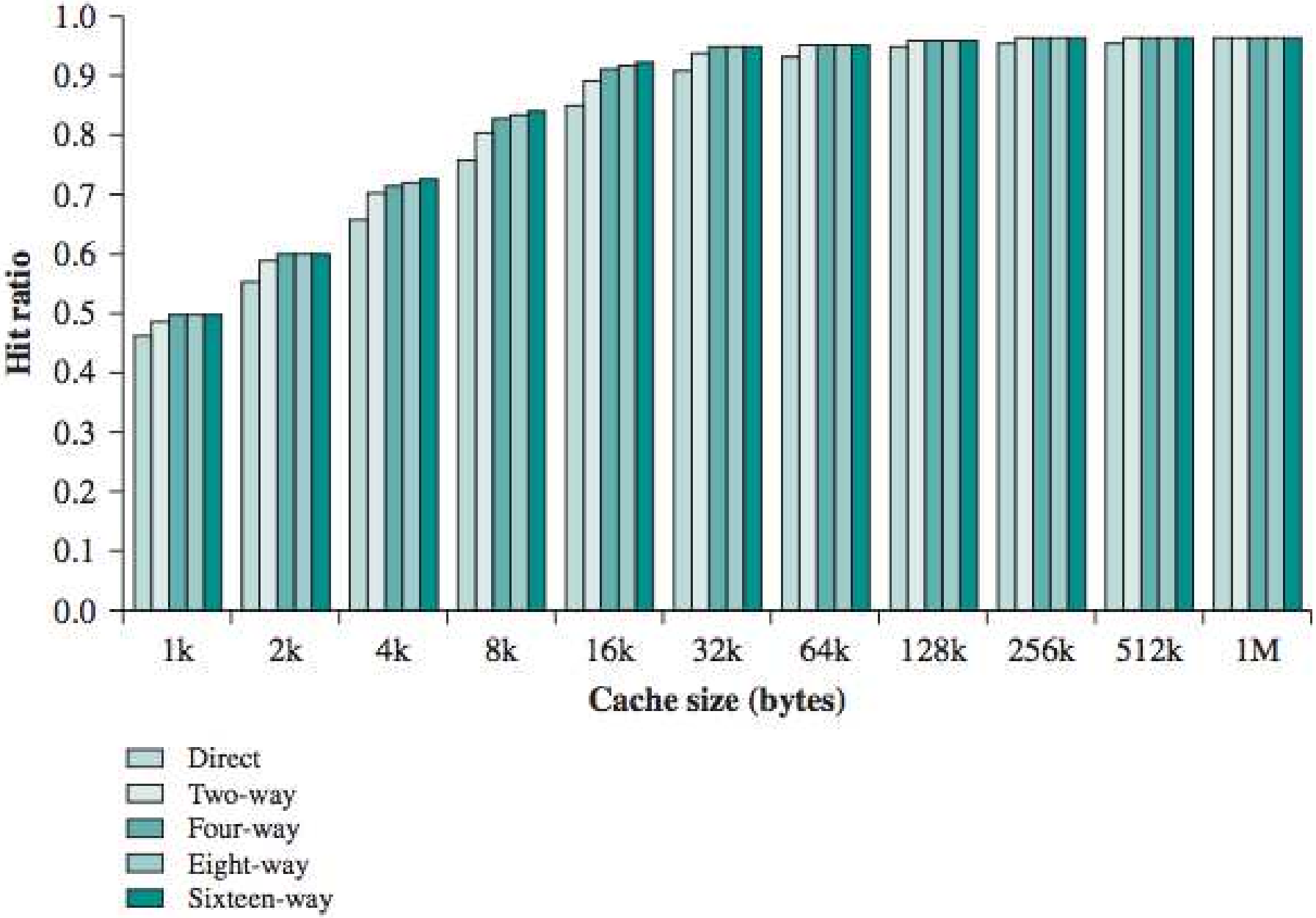

Figure: Varying associativity degree ***k*** (lines per set) over cache size

Key points from the plot:

- k-way: each set has k lines;
- Based on simulating the execution of GCC compiler:
  - Different applications may yield different results;
- Significant performance difference between:
  - Direct and 2-way set associative up to at least 64kB;
- Beyond 32kB:
  - increase in cache size brings no significant increase in performance.
- Difference between:
  - 2-way and 4-way at 4kB is much less than the...
  - ...difference in going from for 4kB to 8kB in cache size;

Replacement Algorithms

He have seen three mapping techniques:

- Direct Mapping;
- Associative Mapping;
- Set-Associative Mapping

Why do we need replacement algorithms? Any ideas?

Replacement Algorithms

Eventually: cache will fill and blocks will need to be replaced:

- For **direct mapping**, there is only one possible line for any particular block:
  - Thus no choice is possible;
- For the **associative** and **set-associative** techniques:
  - a replacement algorithm is needed

Most common replacement algorithms (1/2):

- **Least recently used (LRU):**
  - Probably the most effective;
  - Replace block in the set that has been in the cache longest:
    - With no references to it!
  - Maintains a list of indexes to all the lines in the cache:
    - Whenever a line is used move it to the front of the list;
    - Choose the line at the back of the list when replacing a block;

Most common replacement algorithms (2/2):

- **First-in-first-out (FIFO):**
  - Replace the block in the set that has been in the cache longest:
    - Regardless of whether or not there exist references to the block;
  - easily implemented as a round-robin or circular buffer technique
- **Least frequently used (LFU):**
  - Replace the block in the set that has experienced the fewest references;
  - implemented by associating a counter with each line.

Can you think of any other technique?

Strange possibility: **random line replacement:**

- studies have shown only slightly inferior performance to LRU, LFU and FIFO.
- =)

Write Policy

What happens when a block resident in cache needs to be replaced? Any ideas?

Can you see any implications that having a cache has on memory management? Any ideas?

Write Policy

Two cases to consider:

- If the old block in the cache has not been altered:
  - simply overwrite with a new block;
- If at least one write operation has been performed:
  - main memory must be updated before bringing in the new block.

Some problem examples of having multiple memories:

- more than one device may have access to main memory, *e.g.:*
  - I/O module may be able to read-write directly to memory;
  - if a word has been altered only in the cache:
    - the corresponding memory word is invalid.
  - If the I/O device has altered main memory:
    - then the cache word is invalid.
- Multiple processors, each with its own cache
  - if a word is altered in one cache, invalidates the same word in other caches.

How can we tackle these issues? Any ideas?

How can we tackle these issues? Any ideas?

We have two possible techniques:

- Write through;
- Write back;

Lets have a look at these two techniques =)

Write through technique:

- All write operations are made to main memory as well as to the cache;
- Ensuring that main memory is always valid;
- **Disadvantage:**
  - lots of memory accesses $\rightarrow$ worse performance;

Write back technique:

- Minimizes memory writes;
- Updates are made only in the cache:
  - When an update occurs, a **use bit**, associated with the line is set.
  - When a block is replaced, it is written to memory *iff* the use bit is on.
- **Disadvantage:**
  - I/O module can access main memory (later chapter)...
  - But now all updates must pass through the cache...
  - This makes for complex circuitry and a potential bottleneck

Example (1/2)

Consider a system with:

- 32 byte cache line size;
- 30 ns main memory transfer time for a 4-byte word;

> What is the number of times that the line must be written before being swapped out for a write-back cache to be more efficient that a write-through cache?

Example (2/2)

What is the number of times that the line must be written before being swapped out for a write-back cache to be more efficient that a write-through cache?

- **Write-back case:**
  - At swap-out time we need to transfer $32/4 = 8$ words;
  - Thus we need $8 \times 30 = 240ns$
- **Write-through case:**
  - Each line update requires that one word be written to memory, taking $30ns$
- **Conclusion:**
  - If line gets written more than 8 times, the write-back method is more efficient;

But what happens when we have multiple caches?

But what happens when we have multiple caches?

Can you see the implications of having multiple caches for memory management?

But what happens when we have multiple caches?

Can you see the implications of having multiple caches for memory management?

What happens if data is altered in one cache?

If data in one cache is altered:

- invalidates not only the corresponding word in main memory...
- ...but also that same word in other caches:
  - if any other cache happens to have that same word
- Even if a write-through policy is used:
  - other caches may contain invalid data;
- We want to guarantee **cache coherency** (Chapter 5).

What are the possible mechanisms for dealing with cache coherency? Any ideas?

Possible approaches to cache coherency (1/3):

- **Bus watching with write through:**
  - Each cache monitors the address lines to detect write operations to memory;
  - If a write is detected to memory that also resides in the cache:
    - cache line is invalidated;

Possible approaches to cache coherency (2/3):

- **Hardware transparency:**
  - Use additional hardware to ensure that all updates to main memory via cache are reflected in all caches

Possible approaches to cache coherency (3/3):

- **Noncacheable memory:**
  - Only a portion of main memory is shared by more than one processor, and this is designated as noncacheable;
  - All accesses to shared memory are cache misses, because the shared memory is never copied into the cache.
- **MESI Protocol:**
  - We will see this in better detail later on...

Line Size

Another design element is the line size:

- Lines store memory blocks:
  - Includes not only the desired word but also some adjacent words.
- As the block size increases from very small to larger sizes:
  - Hit ratio will at first increase because of the principle of locality;
- However, as the block becomes even bigger:
  - Hit ratio will begin to decrease;
    - A lot of the words in bigger blocks will be irrelevant...

Two specific effects come into play:

- Larger blocks reduce the number of blocks that fit into a cache.
  - Also, because each block fetch overwrites older cache contents...
  - ...a small number of blocks results in data being overwritten shortly after they are fetched.
- As a block becomes larger:
  - each additional word is farther from the requested word...
  - ... and therefore less likely to be needed in the near future.

The relationship between block size and hit ratio is complex:

- depends on the locality characteristics of a program;
- no definitive optimum value has been found

Number of caches

Recent computer systems:

- use multiple caches;

This design issue covers the following topics

- number of cache levels;
- also, the use of unified versus split caches;

Lets have a look at the details of each one of these...

Multilevel caches

As logic density increased:

- became possible to have a cache on the same chip as the processor:
  - reduces the processor's external bus activity;
  - therefore improving performance;
- when the requested instruction or data is found in the on-chip cache:
  - bus access is eliminated;
  - because of the short data paths internal to the processor:
    - cache accesses will be faster than even zero-wait state bus cycles.
    - Furthermore, during this period the bus is free to support other transfers.

With the continued shrinkage of processor components:

- processors now incorporate a second cache level (L2) or more:
- savings depend on the hit rates in both the L1 and L2 caches.
  - In general: use of a second-level cache does improve performance;
  - However, multilevel caches complicate design issues:
    - size;
    - replacement algorithms;
    - write policy;

Two-level cache performance as a function of cache size:

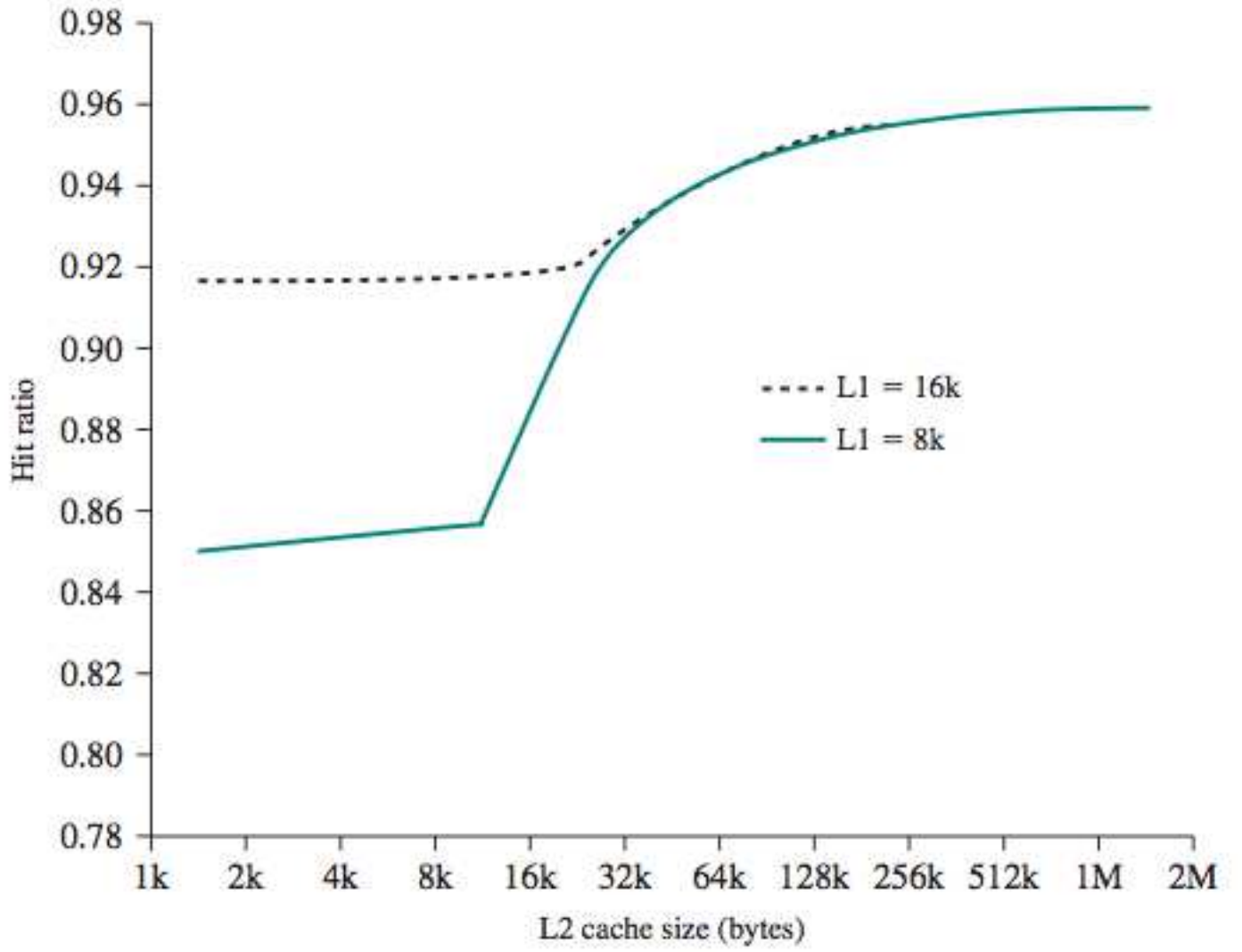

Figure: Total hit ratio (L1 and L2) for 8-Kbyte and 16-Kbyte L1 (Source: (Stallings, 2015))

Figure from previous slide (1/2):

- assumes that both caches have the same line size;
- shows the total hit ratio:

Figure from previous slide (2/2):

- shows the impact of L2 size on total hits with respect to L1 size.
- Steepest part of the slope for an L1 cache:
  - of 8 Kbytes is for an L2 cache of 16 Kbytes;
  - of 16 Kbytes is for an L2 cache size of 32 Kbytes;
- L2 has little effect on performance until it is at least double the L1 cache size.
- Otherwise, L2 cache has little impact on total cache performance.

It may be a strange question: but why do we need an L2 cache to be larger than L1? Any ideas?

It may be a strange question but: why do we need an L2 cache to be larger than L1? Any ideas?

- If the L2 cache has the same line size and capacity as the L1 cache...
- ...its contents will more or less mirror those of the L1 cache.

Also, there is a performance advantage to adding a L3 and a L4.

Unified versus split caches

In recent computer systems:

- it has become common to **split** the cache into two:
  - Instruction cache;
  - Data cache;
- both exist at the same level:
  - typically as two L1 caches:
  - When the processor attempts to fetch:
    - an instruction from main memory, it first consults the instruction L1 cache,
    - data from main memory, it first consults the data L1 cache.

Two potential advantages of a **unified cache:**

- Higher hit rate than split caches:
  - automatically load balancing between instruction and data fetches, *i.e.:*
    - if an execution pattern involves more instruction fetches than data fetches...
    - ...the cache will tend to fill up with instructions;
    - if an execution pattern involves relatively more data fetches...
    - ...the cache will tend to fill up with data;
- Only one cache needs to be designed and implemented.

Unified caches seems pretty good...

So why the need for split caches? Any ideas?

Key advantage of the **split cache** design:

- eliminates competition for the cache between
  - Instruction fetch/decode/execution stages...
  - and the load / store data stages;
- Important in any design that relies on the **pipelining of instructions:**
  - fetch instructions ahead of time
  - thus filling a pipeline with instructions to be executed.
  - Chapter 14

With a **unified** instruction / data cache:

- Data / instructions will be stored in a single location;
- Pipelining:
  - Multiples stages of the instruction cycle can be executed simultaneously
  - Chapter 14;
- Because there is a single cache:
  - Executions of multiple stages cannot be performed;
  - Performance bottleneck;

Split cache structure overcomes this difficulty.

Intel Cache Evolution

Problem	Solution	Processor on Which Feature First Appears
External memory slower than the system bus.	Add external cache using faster memory technology.	386
Increased processor speed results in external bus becoming a bottleneck for cache access.	Move external cache on-chip, operating at the same speed as the processor.	486
Internal cache is rather small, due to limited space on chip.	Add external L2 cache using faster technology than main memory.	486
Contention occurs when both the Instruction Prefetcher and the Execution Unit simultaneously require access to the cache. In that case, the Prefetcher is stalled while the Execution Unit's data access takes place.	Create separate data and instruction caches.	Pentium
Increased processor speed results in external bus becoming a bottleneck for L2 cache access.	Create separate back-side bus that runs at higher speed than the main (front-side) external bus. The BSB is dedicated to the L2 cache.	Pentium Pro
	Move L2 cache on to the processor chip.	Pentium II
Some applications deal with massive databases and must have rapid access to large amounts of data. The on-chip caches are too small.	Add external L3 cache.	Pentium III
	Move L3 cache on-chip.	Pentium 4

Figure: Intel Cache Evolution (Source: (Stallings, 2015))

Intel Pentium 4 Block diagram

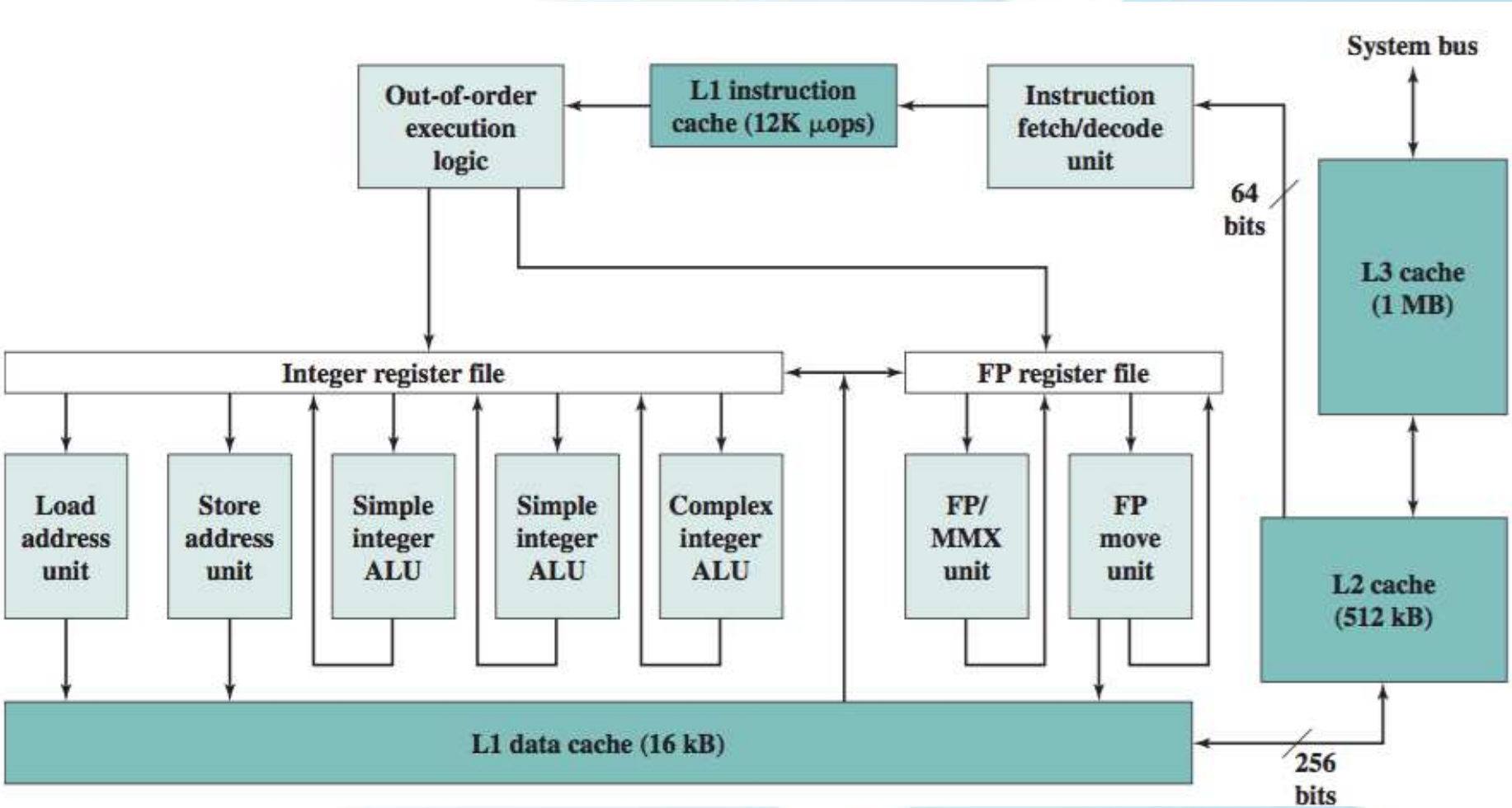

Figure: Pentium 4 block diagram (Source: (Stallings, 2015))

References I

Stallings, W. (2015).

Computer Organization and Architecture.

Pearson Education.