B38DF
Computer Architecture and Embedded Systems

Alexander Belyaev

Heriot-Watt University
School of Engineering & Physical Sciences
Electrical, Electronic and Computer Engineering

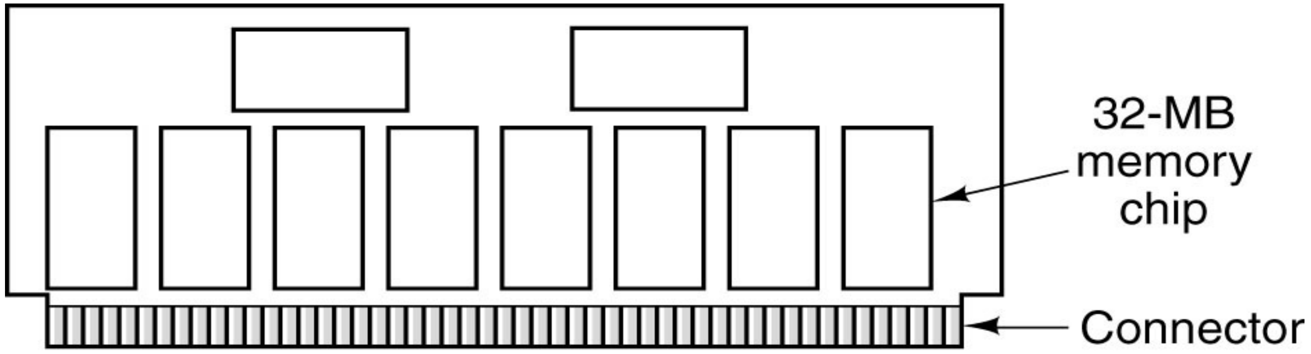
E-mail: a.belyaev@hw.ac.uk
Office: EM2.29

Based on the slides prepared by Dr. Mustafa Suphi Erden

Primary Memory

- **Memory (or Store or Storage)**

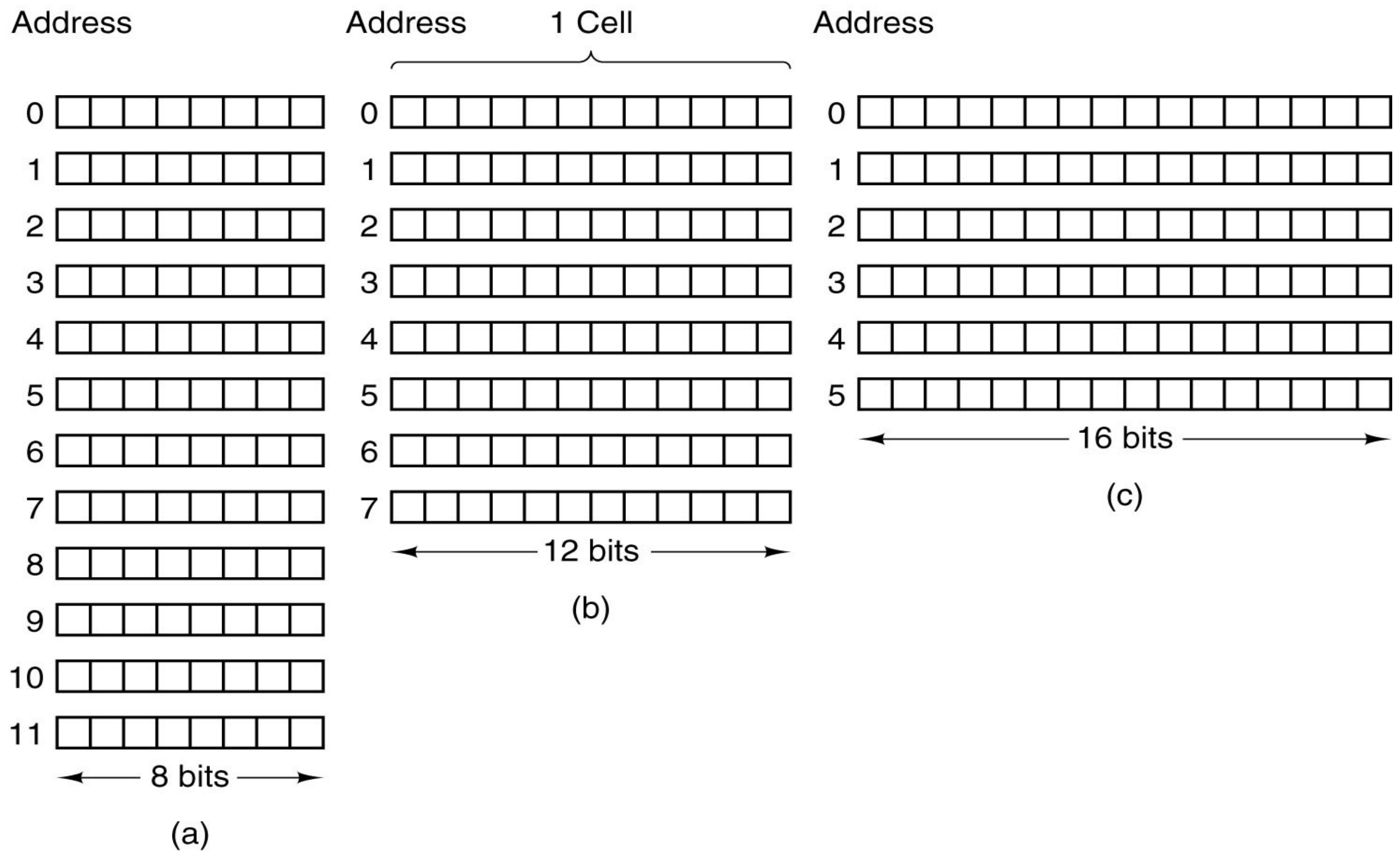
  Part of the computer where programs and data are stored (most common: Random-Access Memory – RAM)



A single inline memory module (SIMM) holding 256 MB. Two of the chips control the SIMM.

Tanenbaum, Structured Computer Organization, Fifth Edition, (c) 2006 Pearson Education, Inc.

Memory Addresses



Three ways of organizing a 96-bit memory.

Address

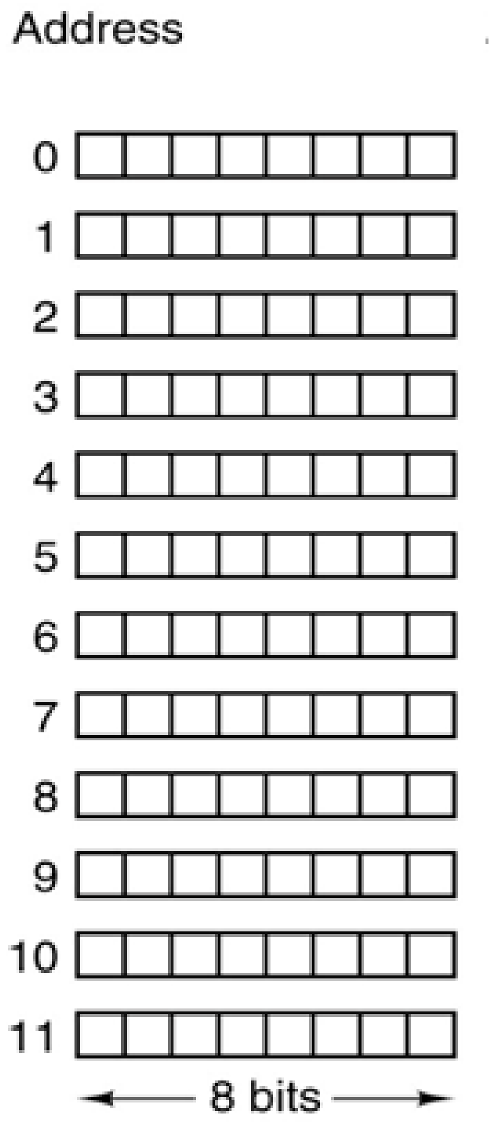
- Memories consist of **cells** (or **locations**) each of which can store a piece of information.
- Each cell has a number, called its **address**.
- If a memory has *n* cells, they will have **addresses 0 to *n*-1**.
- All cells contain the same number of bits.
- If a cell consists of *k* bits, it can hold any one of 2^k different bit combinations.

Address - Example

- Memory addresses expressed as binary numbers
- How many bits are there in the address used to reference the memory on the right?
  - ➢ Number of cells : 12
  - ➢ Addresses: 0 to 11
  - ➢ How many bits?

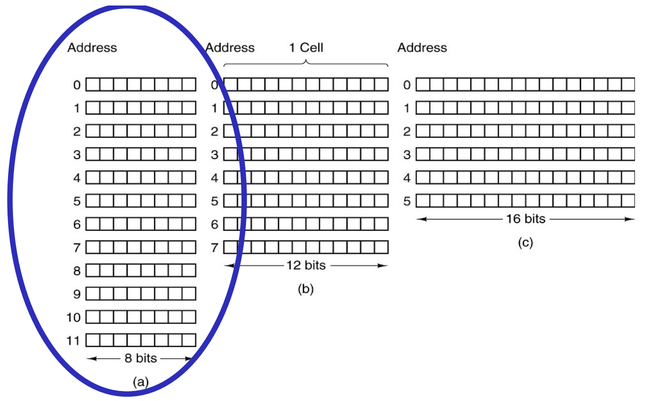
    → **4** ($2^3 < 11 < 2^4$)
  - ➢ Address of cell 5 ?

    → 0 1 0 1



Tanenbaum, Structured Computer Organization, Fifth Edition, (c) 2006 Pearson Education, Inc.

Byte and Word



- **Cell →** smallest addressable unit

- **Byte →** 8-bit cells (nearly all computer manufacturers have standardized on an 8-bit cell)

- **Word →** Bytes are grouped into **words**. Most instructions operate on entire words.

  - A computer with 32-bit word has 4 bytes/word

  - A computer with 64-bit word has 8 bytes/word

- A 32(64)-bit machine will have 32(64)-bit registers and instructions for manipulating 32(64)-bit words

Tanenbaum, Structured Computer Organization, Fifth Edition, (c) 2006 Pearson Education, Inc.

Byte Ordering: Big Endian versus Little Endian

The term *endian* refers to a computer architecture's "byte order," or the way the computer stores the bytes of a multiple-byte data element.

Virtually all computer architectures today are byte-addressable and must, therefore, have a standard for storing information requiring more than a single byte.

Some machines store a two-byte integer, for example, with the **least significant byte first** (at the lower address) followed by the most significant byte. Therefore, a byte at a lower address has lower significance. These machines are called ***little endian*** machines.

Other machines store this same two-byte integer with its **most significant byte first**, followed by its least significant byte. These are called ***big endian*** machines because they store the most significant bytes at the lower addresses.

These two terms, little and big endian, are from the book Gulliver's Travels: the Lilliputians were divided into two camps: those who ate their eggs by opening the "big" end (big endians) and those who ate their eggs by opening the "little" end (little endians).

However *endianness* turns out to be a major architectural consideration.

Byte Ordering: Big Endian versus Little Endian

Endianness turns out to be a major architectural consideration.

CPU manufacturers are also divided into two factions. For example, **Intel** has always done things the “**little endian**” way whereas **IBM** (mainframes) and **Motorola** use the “**big endian**” way. (It is also worth noting that some CPUs can handle both little and big endian.)

For example, consider an integer requiring 4 bytes:

Byte 3	Byte 2	Byte 1	Byte 0

little endian

```
Base Address + 0 = Byte0
Base Address + 1 = Byte1
Base Address + 2 = Byte2
Base Address + 3 = Byte3
```

big endian

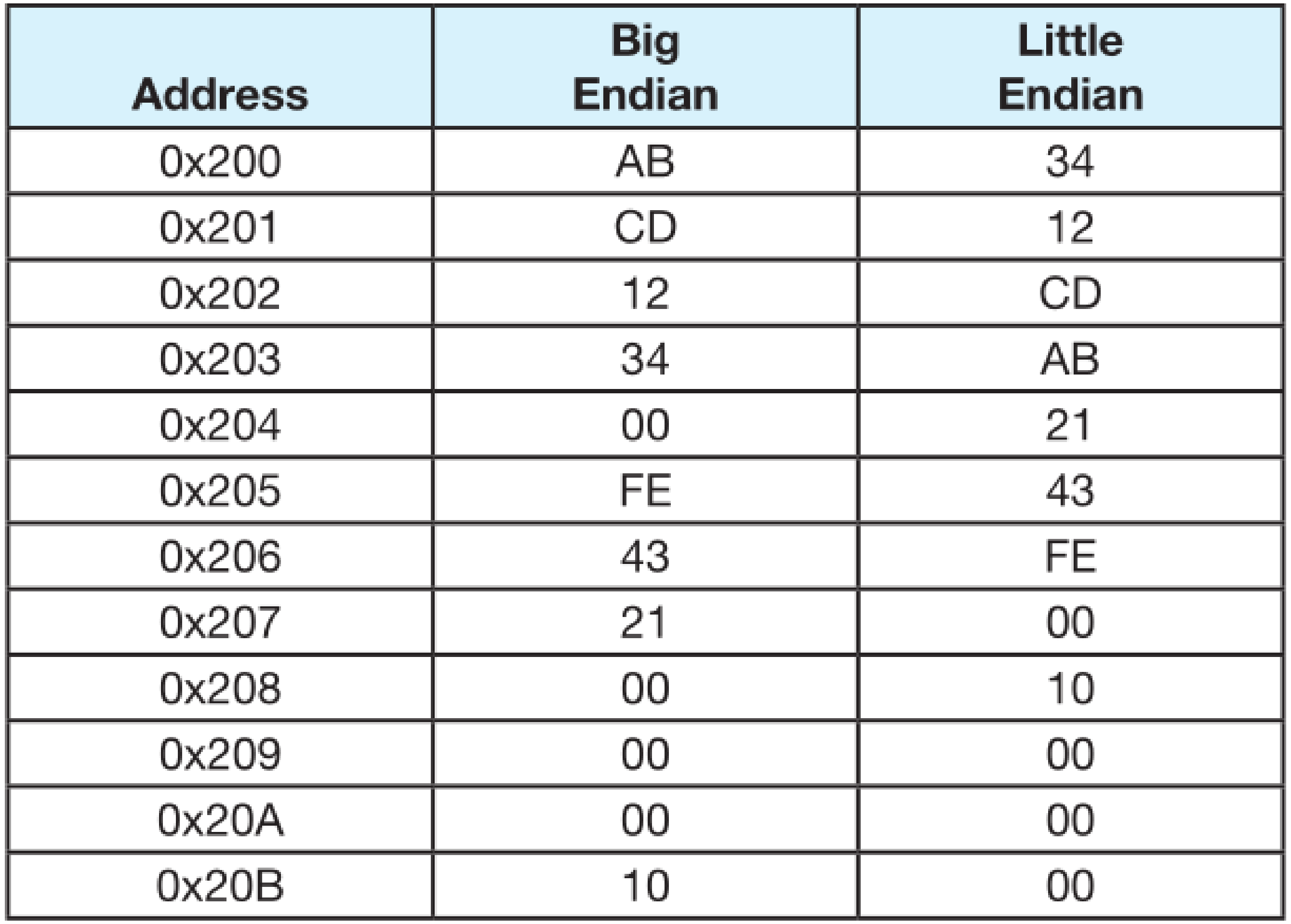
```
Base Address + 0 = Byte3
Base Address + 1 = Byte2
Base Address + 2 = Byte1
Base Address + 3 = Byte0
```

Storing hex number 0x12345678

Address →	**00**	**01**	**10**	**11**
Big Endian	12	34	56	78
Little Endian	78	56	34	12

Byte Ordering: Big Endian versus Little Endian

- A larger example: A computer uses 32-bit (4-byte) integers. The values 0xABCD1234, 0x00FE4321, and 0x10 would be stored sequentially in memory, starting at address 0x200 as below.



Address	Big Endian	Little Endian
0x200	AB	34
0x201	CD	12
0x202	12	CD
0x203	34	AB
0x204	00	21
0x205	FE	43
0x206	43	FE
0x207	21	00
0x208	00	10
0x209	00	00
0x20A	00	00
0x20B	10	00

Byte Ordering: Big Endian versus Little Endian

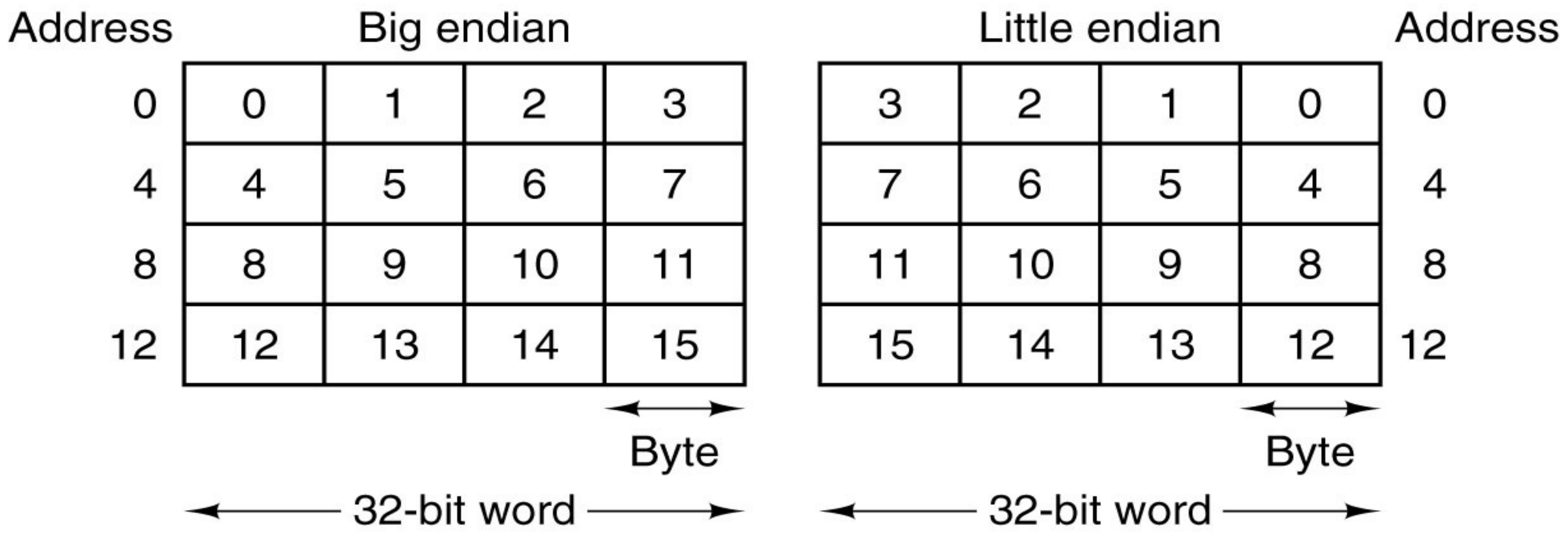
- Big endian:
  - Is more natural (?)
  - The sign of the number can be determined by looking at the byte at address offset 0.
  - Strings and integers are stored in the same order.

- Little endian:
  - Makes it easier to place values on non-word boundaries.
  - Conversion from a 16-bit integer address to a 32-bit integer address does not require any arithmetic.

Byte Ordering: Big Endian versus Little Endian

The most significant byte first (at the lower address) → **Big Endian** (e.g. big IBM mainframes)

The least significant byte first (at the lower address) → **Little Endian** (e.g. Intel family)



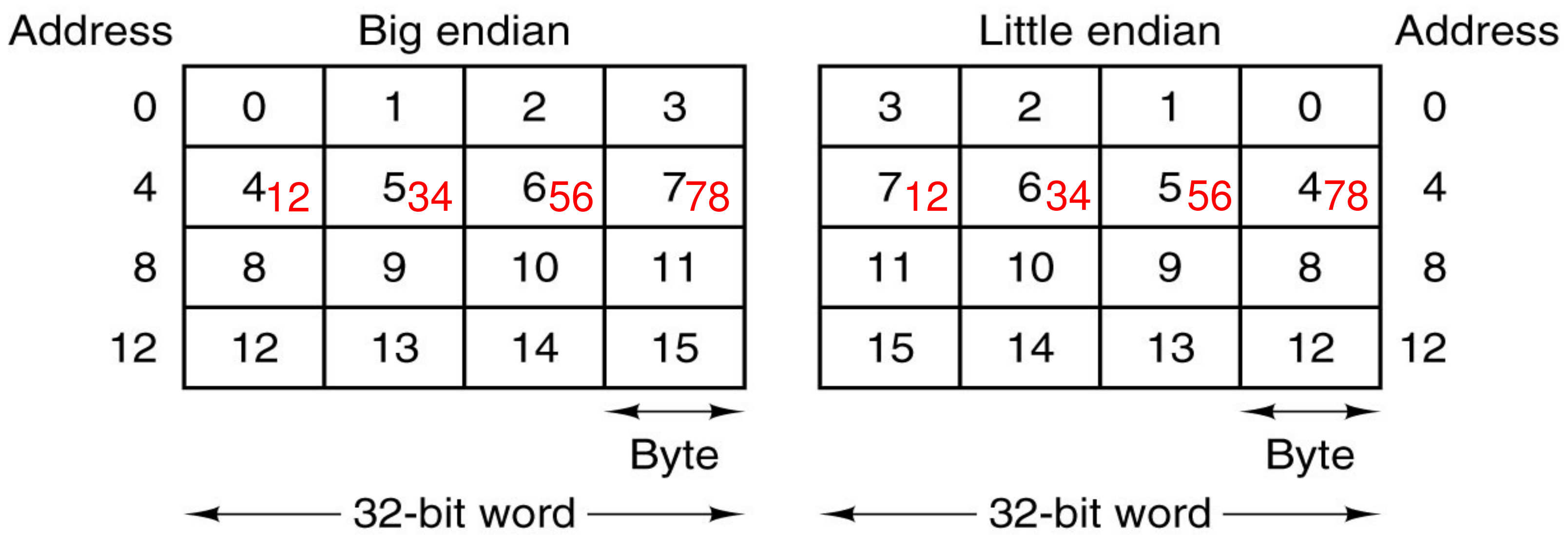
Address	Big endian				Little endian				Address
0	0	1	2	3	3	2	1	0	0
4	4	5	6	7	7	6	5	4	4
8	8	9	10	11	11	10	9	8	8
12	12	13	14	15	15	14	13	12	12

Storing hex number 0x12345678

Byte Ordering: Big Endian versus Little Endian

The most significant byte first (at the lower address) → **Big Endian** (e.g. big IBM mainframes)

The least significant byte first (at the lower address) → **Little Endian** (e.g. Intel family)



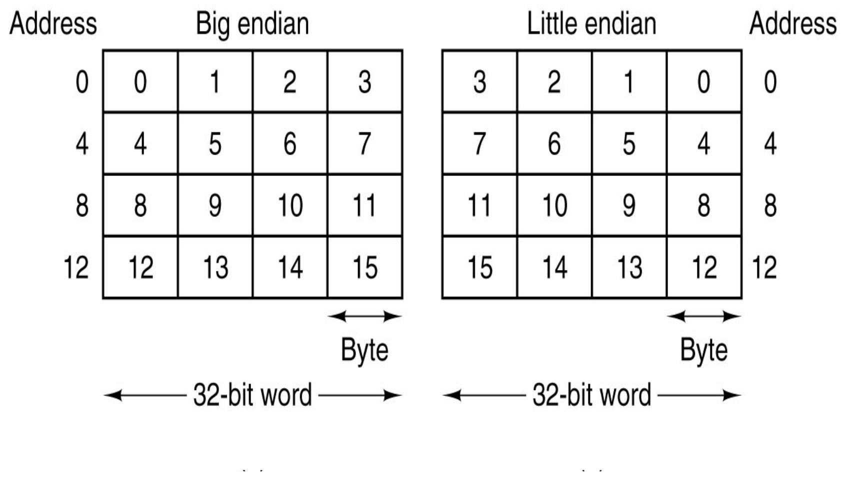
Storing hex number 0x12345678

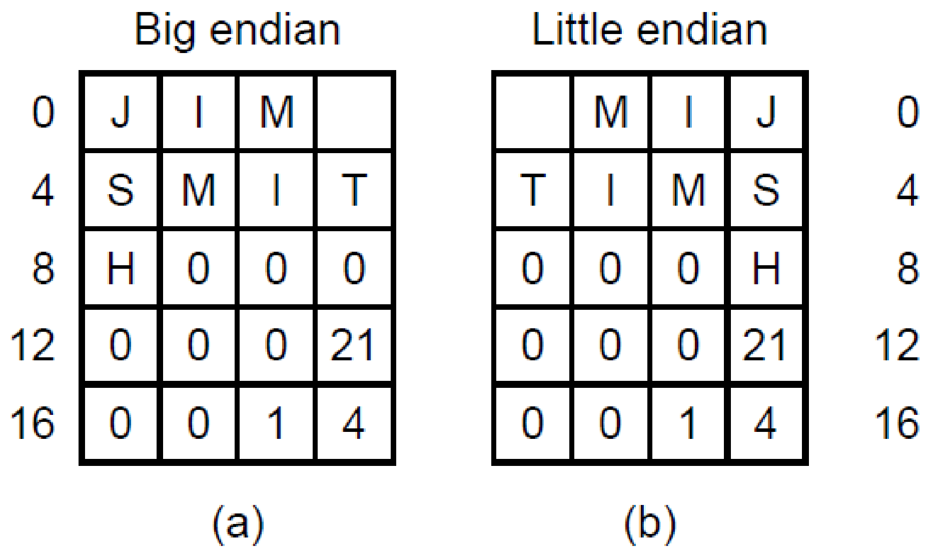
No problems when we deal with integers

An example: Storing integers and strings

Problems appear when we deal with a mixture of integers, character strings, and other data types.

Consider, for example, a simple personnel record consisting of a string (employee name) and two integers (age and department number). The string is terminated with one or more 0 bytes to fill out a word. The big and little endian representations are shown below for Jim Smith, age 21, department 260 ($1 \times 256 + 4 = 260$).

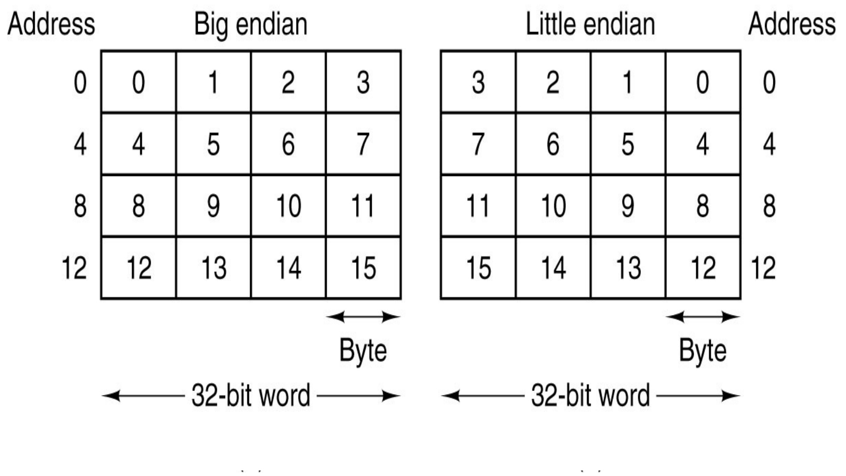


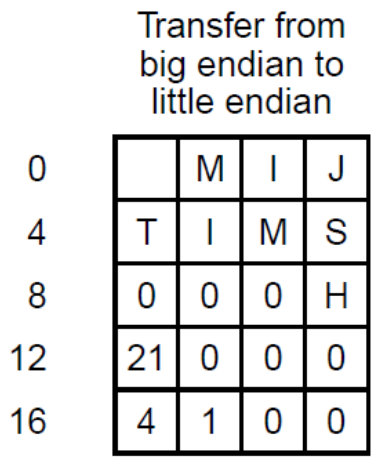


An example: Storing integers and strings

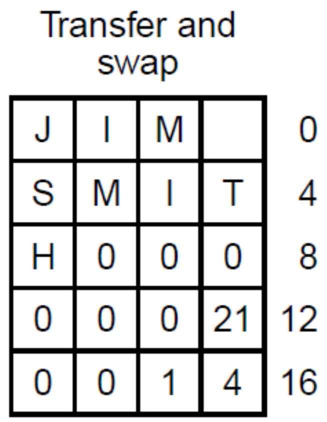
Problems appear when we deal with a mixture of integers, character strings, and other data types and when one machine tries to send the record to the other one over a network.

Let us assume that the big endian sends the record to the little endian one byte at a time, starting with byte 0 and ending with byte 19.





(c)



(d)

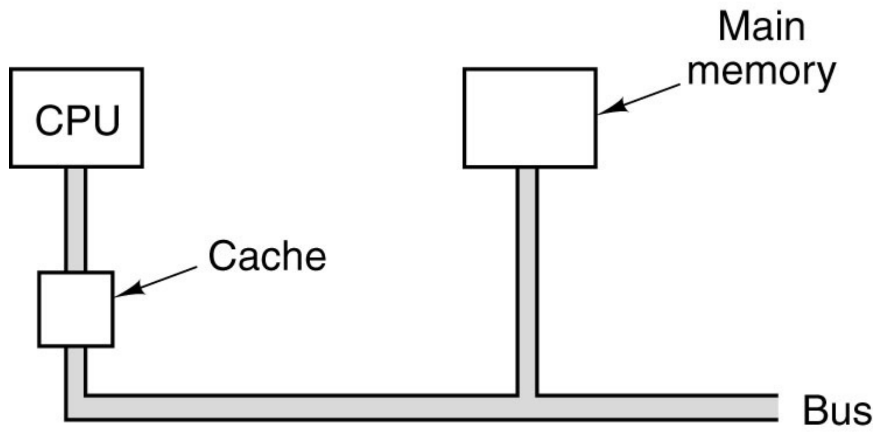
When the little endian tries to print the name, it works fine, but the age comes out as 21×2^{24} and the department is just as garbled. An obvious solution is to have the software reverse the bytes within a word after the copy has been made. This makes the two integers fine but turns the string into “MIJTIMS” with the “H” hanging in the middle of nowhere.

Byte Ordering: Big Endian versus Little Endian

- In any machine, aggregates such as files, data structures, and arrays are composed of multiple data units, each with endianness.
- Thus, **conversion** of a block of memory from one style endianness to the other **requires knowledge of the data structure.**
- One way that works, but is inefficient, is to include a header in front of each data item telling what kind of data follows (string, integer, or other) and how long it is. This allows the receiver to perform only the necessary conversions.

There is no simple solution.

Cache Memory

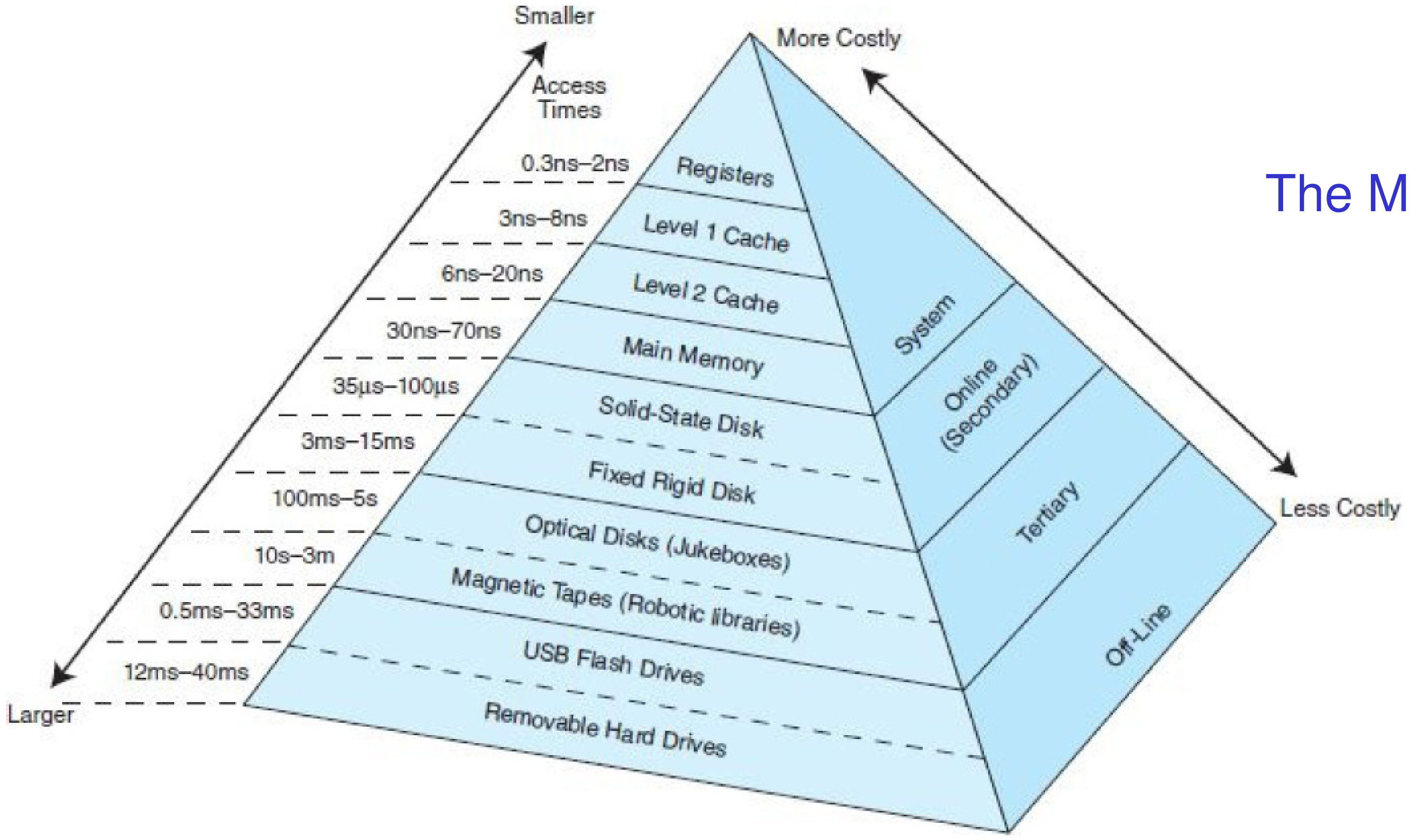


The cache is logically between the CPU and main memory. Physically, there are several possible places it could be located.

- **Cache →** from the French word *cacher* meaning to hide
- The most heavily used memory words are kept in the cache
- When the CPU needs a word, <u>it first looks in the cache</u>, then goes to the main memory

Cache Memory

A computer processor is very fast and is constantly reading information from memory, which means it often has to wait for the information to arrive, because the memory access times are slower than the processor speed. A cache memory is a small, temporary, but fast memory that the processor uses for information it is likely to need again in the very near future.



The Memory Hierarchy

Cache Memory

Suppose you are writing a paper on quantum computing. Would you go to the library, check out one book, return home, get the necessary information from that book, go back to the library, check out another book, return home, and so on?

No, you would go to the library and check out all the books you might need and bring them all home.

The library is analogous to main memory, and your home is similar to cache.

Cache memory works on the same basic principle, by copying frequently used data into the cache rather than requiring an access to main memory to retrieve the data.

Cache memory in a computer differs from the above real-life example in one important way: The computer really has no way to know, *a priori*, what data is most likely to be accessed, so it uses the **locality principle** and transfers an entire block from main memory into cache whenever it has to make a main memory access. If the probability of using something else in that block is high, then transferring the entire block saves on access time.

Cache Memory – Locality Principle

- **Locality Principle:** The memory references made in any short time interval tend to use <u>only a small fraction of the total memory</u>.
- If a given memory reference is to address *A*, it is **likely that the next memory reference will be in the general vicinity** of *A.*
  - Program itself, loops, matrix manipulations
- Except for branches and procedure calls, **instructions are fetched from consecutive locations in the memory.**
- **<u>General idea behind cache usage:</u>**
  - When a word is referenced, <u>it and some of its neighbors are brought from the large slow memory into the cache</u>, so that the next time it is used, it can be accessed quickly.
  - It is <u>faster to fetch *k* words from the main memory all at once than one word *k* times</u>.

Cache Memory – Unified versus Split

- **Unified cache:**

  Instructions and data use the same cache

- **Split cache:**

  Instruction in one cache and data in the other

  (resembles the Harvard architecture)

- A **split cache allows parallel access**, which is good for pipelined CPUs, where the instruction fetch unit needs to access instructions at the same time the operand fetch unit needs to access to data.
- A unified cache does not allow parallel access.