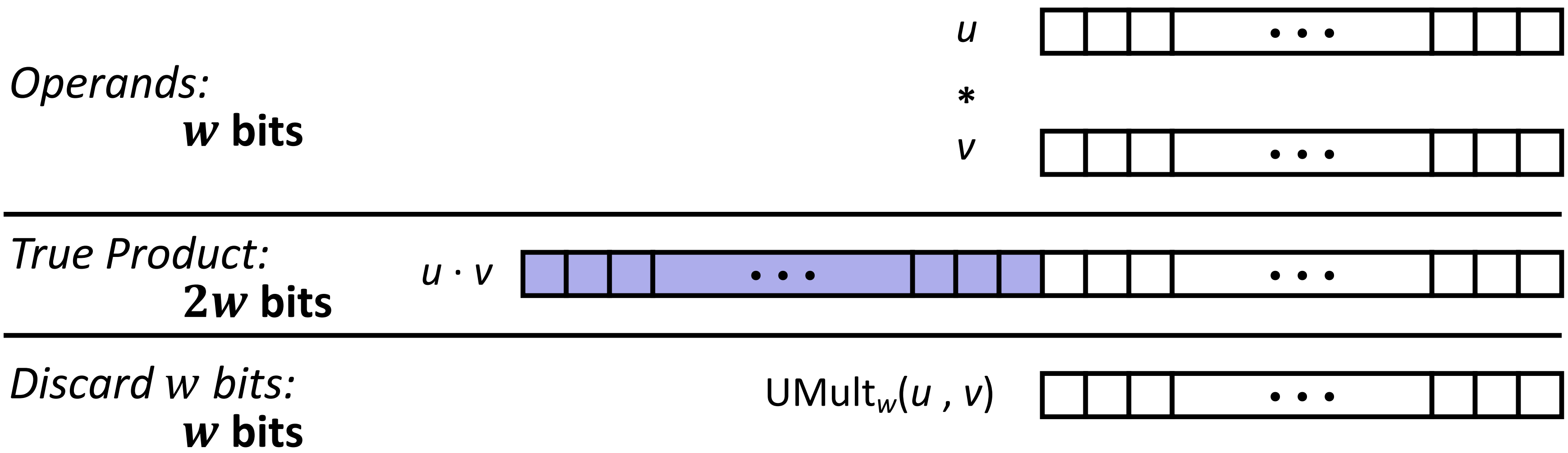
Lecture 03
Memory, Data and Addressing – part 1

Euhyun Moon, Ph.D.
Machine Learning Systems (MLSys) Lab
Computer Science and Engineering
Sogang University

Slides adapted from Randy Bryant and Dave O'Hallaron: Introduction to Computer Systems, CMU

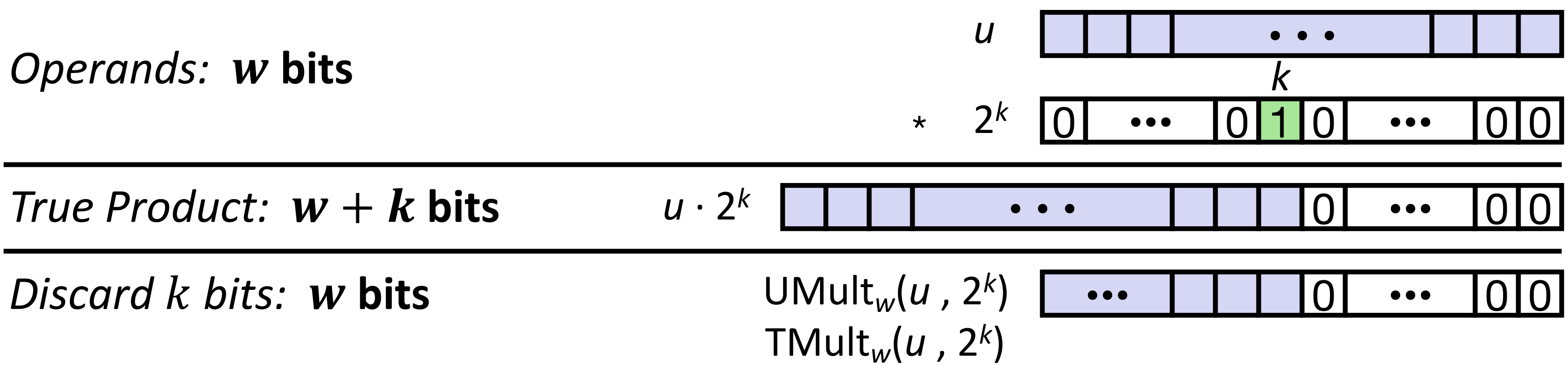
Aside: Unsigned Multiplication in C



- Standard Multiplication Function
  - Ignores high order w bits
- Implements Modular Arithmetic
  - $\text{UMult}_w(u\ ,\ v)\ = u \cdot v \mod 2^w$

Aside: Multiplication with Shift and Add

- Operation `u<<k` gives `u*`$\mathbf{2^k}$
  - Both signed and unsigned

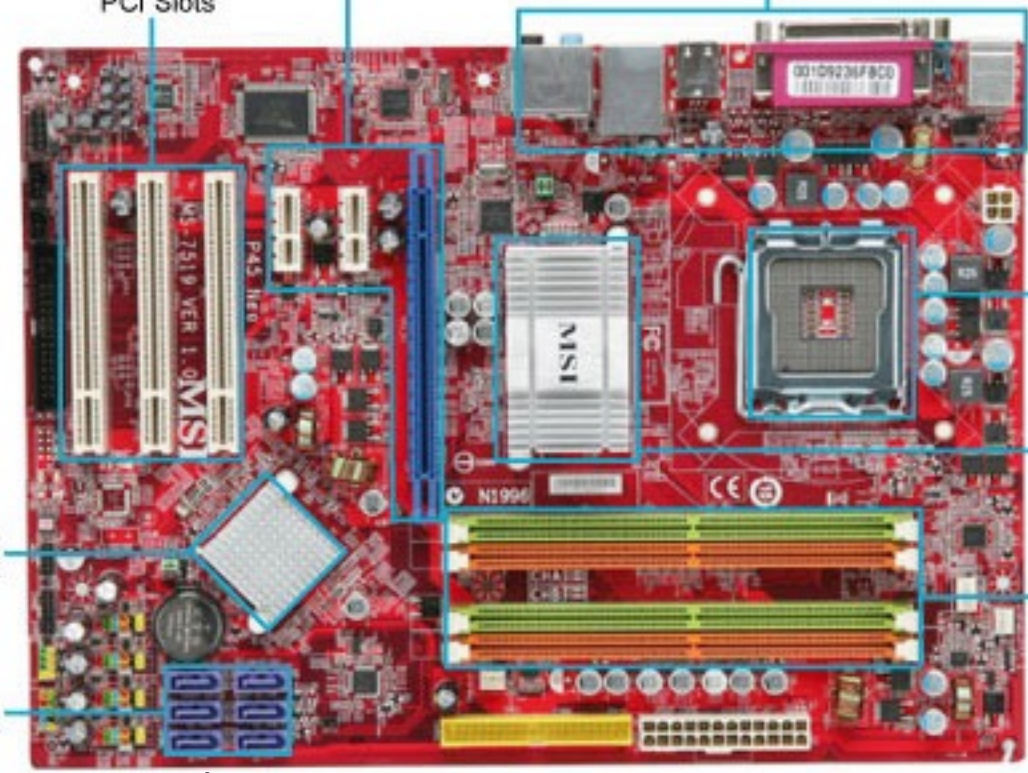


- <u>Examples</u>:
  - `u<<3          ==   u * 8`
  - `u<<5 - u<<3   ==   u * 24`
  - Most machines shift and add faster than multiply
    - ***Compiler generates this code automatically***

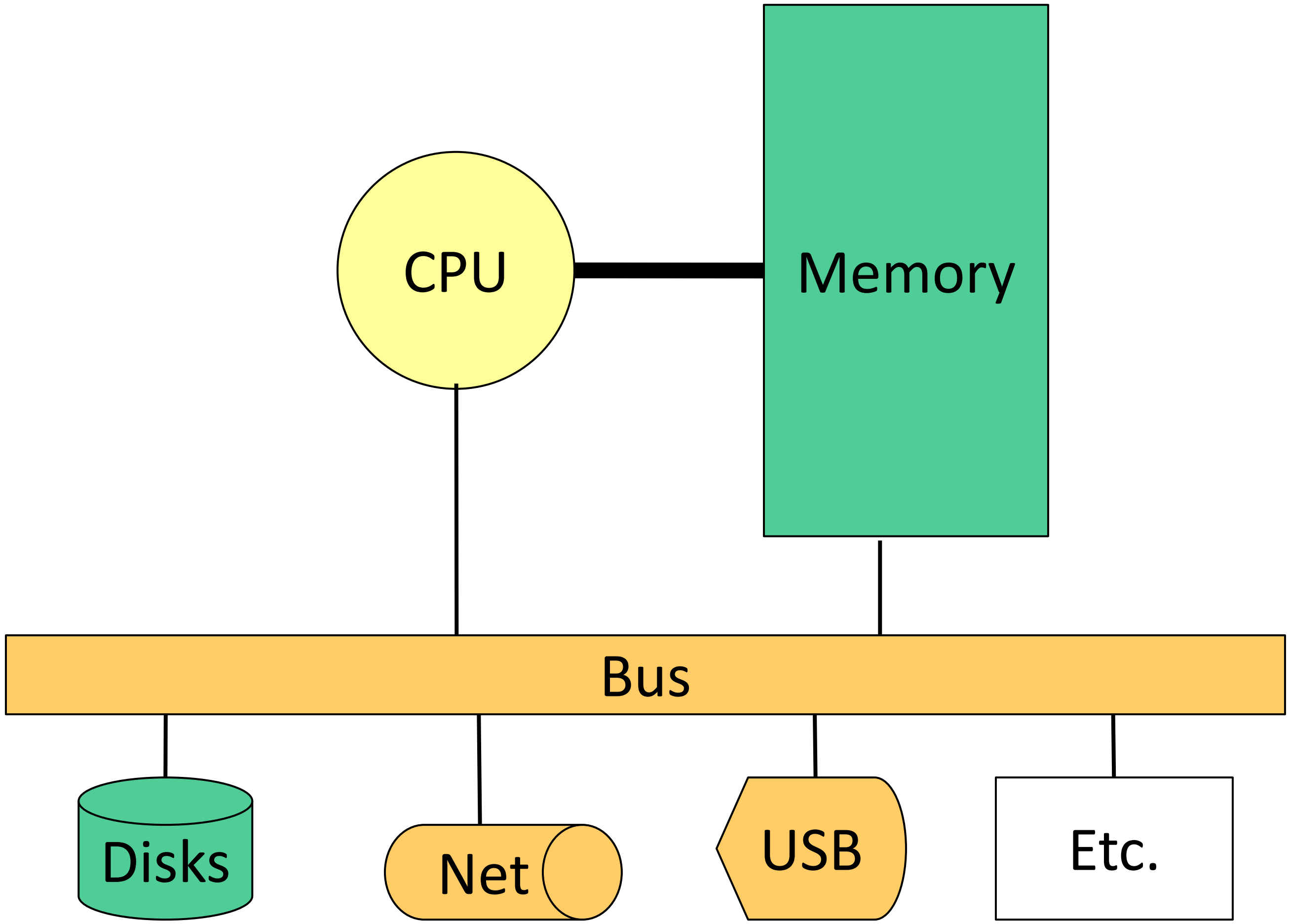
Memory, Data, and Addressing

- **Hardware - High Level Overview**
- **Representing information as bits and bytes**
  - **Memory is a byte-addressable array**
  - **Machine “word” size = address size = register size**
  - **Endianness – ordering bytes in memory**
- Manipulating data in memory using C
  - Assignment
  - Pointers, pointer arithmetic, and arrays
- Boolean algebra and bit-level manipulations

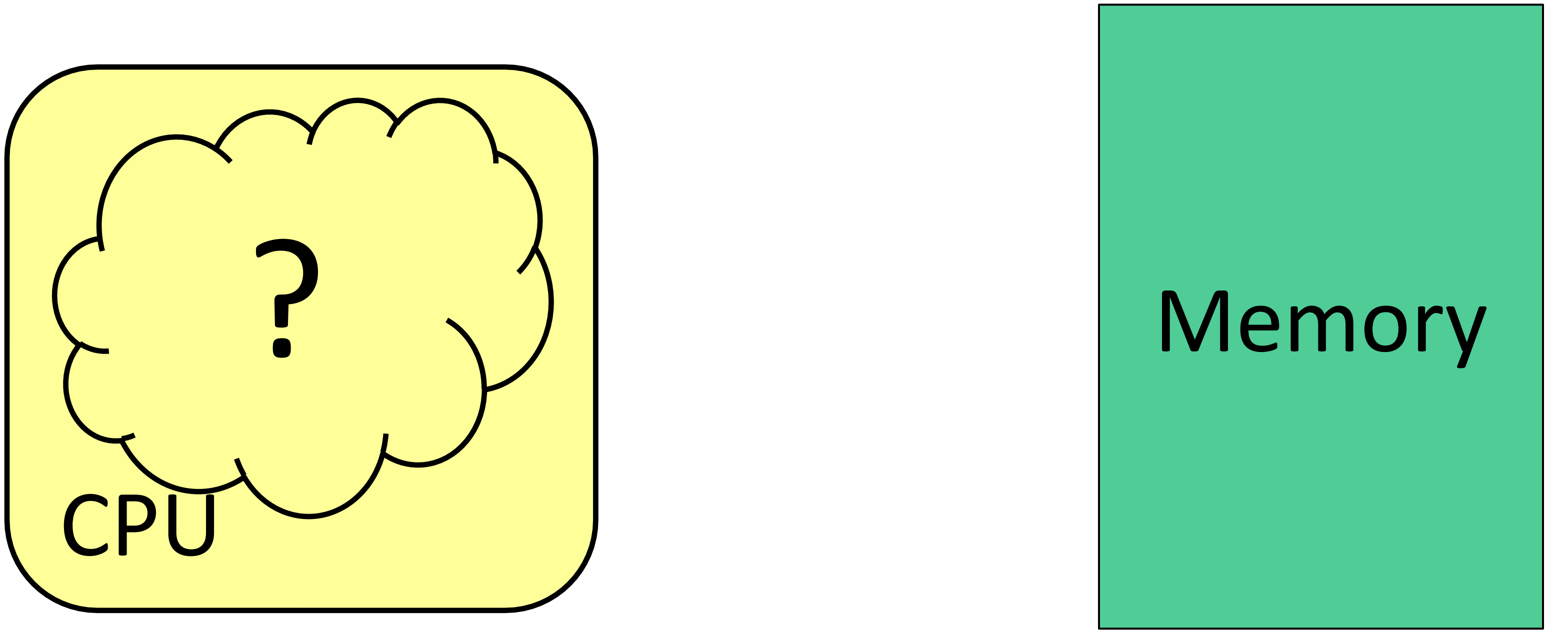
Hardware: Physical View



Hardware: Logical View



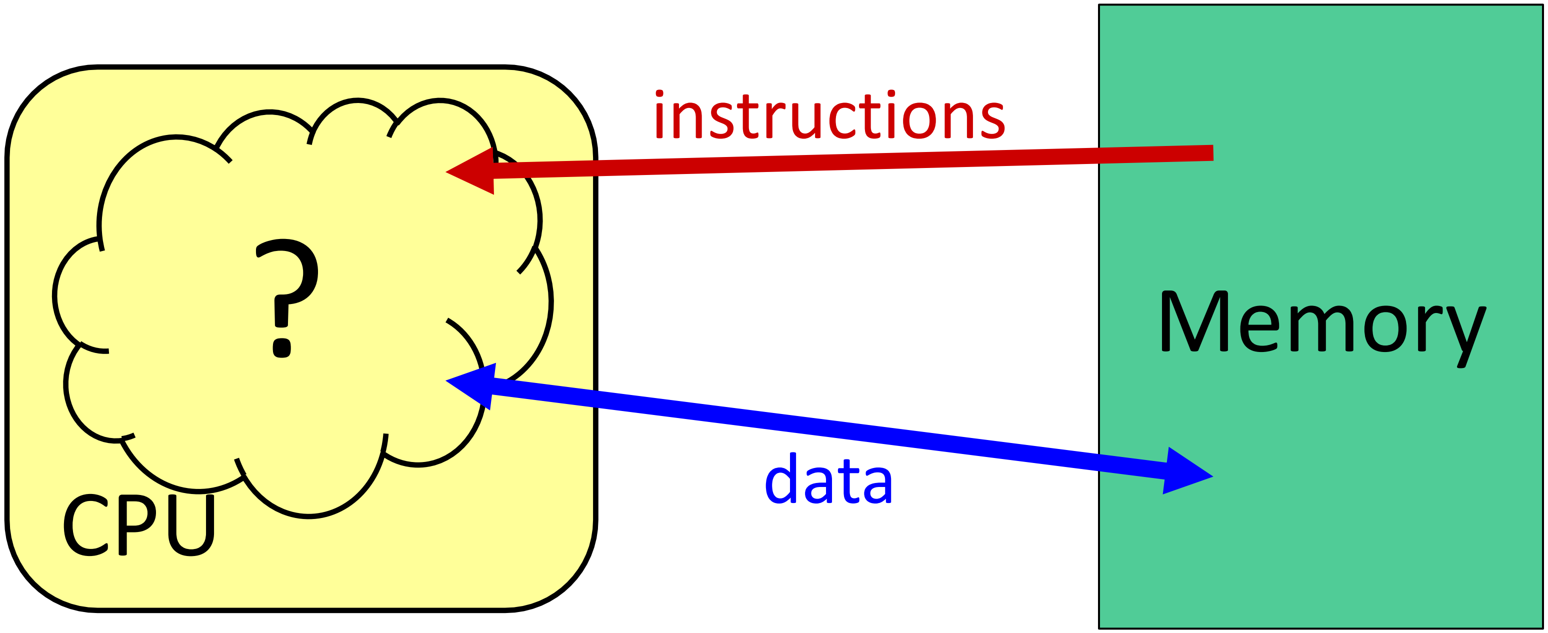
Hardware: 3030 View (version 0)



- The CPU executes instructions
- Memory stores data
- Binary encoding!
  - Instructions *are* just data

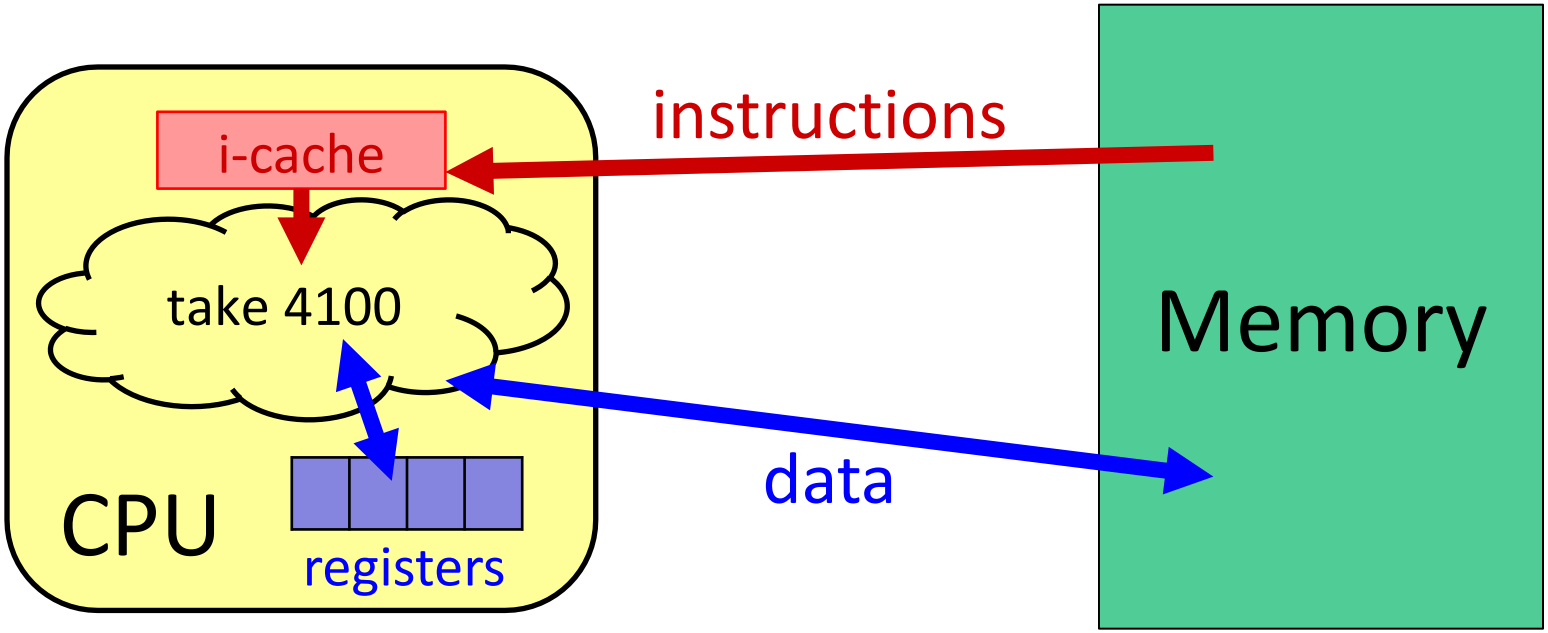
How are data and instructions represented?

Hardware: 3030 View (version 0)



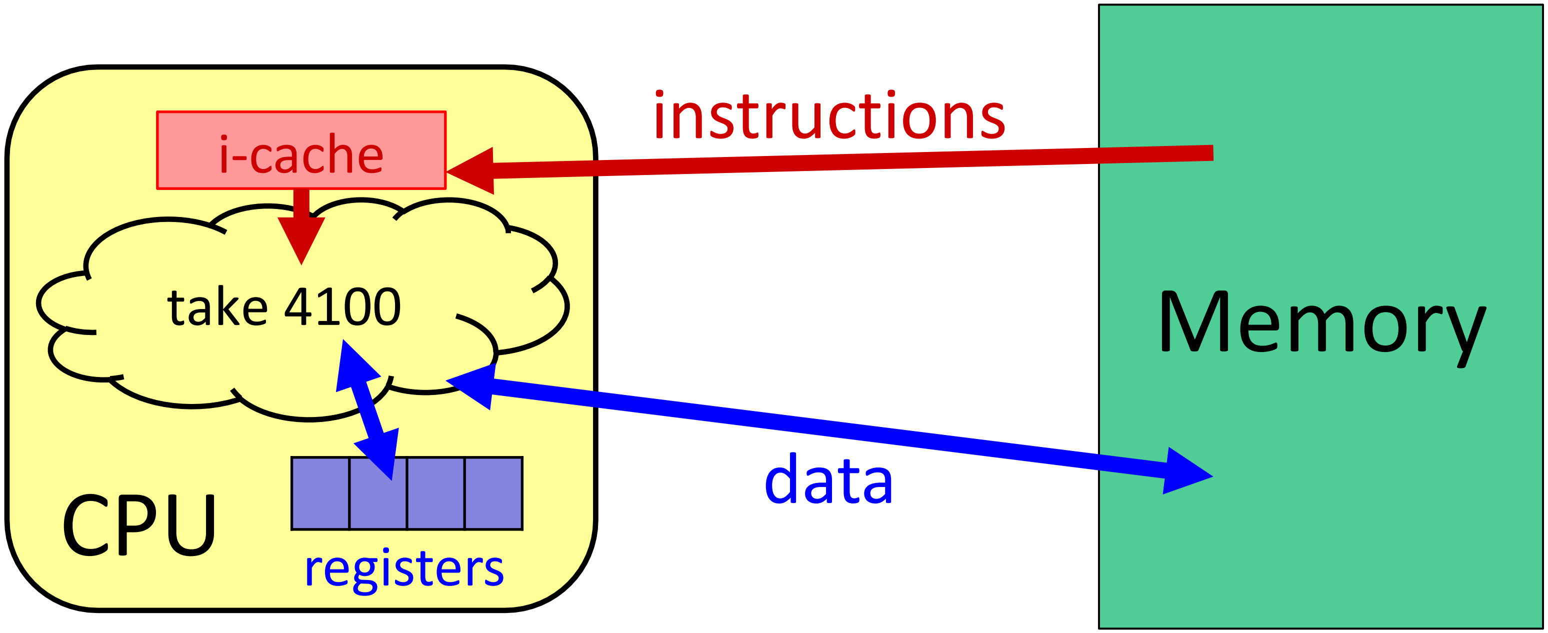
- To execute an instruction, the CPU must:
  1) Fetch the instruction
  2) (if applicable) Fetch data needed by the instruction
  3) Perform the specified computation
  4) (if applicable) Write the result back to memory

Hardware: 3030 View (version 1)



- More CPU details:
  - Instructions are held temporarily in the instruction cache
  - Other data are held temporarily in registers
- Instruction fetching is hardware-controlled
- Data movement is programmer-controlled (assembly)

Hardware: 3030 View (version 1)



- We will start by learning about Memory

How does a program find its data in memory?

Fixed-Length Binary

- Because storage is finite in reality, everything is stored as "fixed" length
  - Data is moved and manipulated in fixed-length chunks
  - Multiple fixed lengths (*e.g.,* 1 byte, 4 bytes, 8 bytes)
  - Leading zeros now *must* be included up to "fill out" the fixed length

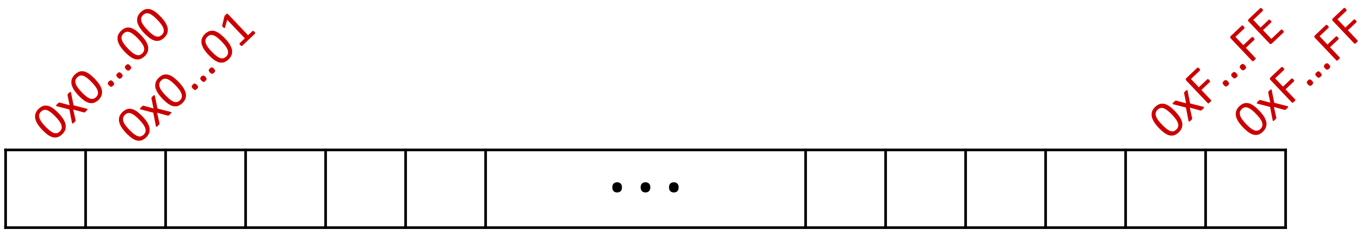
Least Significant Bit (LSB)

- Example:

  the "eight-bit" representation of the number 4 is 0b00000100

Most Significant Bit (MSB)

An Address Refers to a Byte of Memory



- Conceptually, memory is a single, large array of bytes, each with a unique *address* (index)
  - Each address is just a number represented in *fixed-length* binary
- Programs refer to bytes in memory by their *addresses*
  - Domain of possible addresses = *address space*
  - We can store addresses as data to “remember” where other data is in memory
- But not all values fit in a single byte…
  - Many operations actually use multi-byte values

Machine "Words"

- Instructions encoded into machine code (0's and 1's)
  - Historically (still true in some assembly languages), all instructions were exactly the size of a word

- We have *chosen* to tie word size to address size/width
  - word size = address size = register size
  - word size = w bits → 2^w addresses

- Current x86 systems use **64-bit (8-byte) words**
  - Potential address space: $\mathbf{2^{64}}$ addresses
    2^{64} bytes ≈ $\mathbf{1.8 \times 10^{19}}$ **bytes**
    = 18 billion billion bytes = 18 EB (exabytes)
  - Actual physical address space: **48 bits**

Data Representations

- Sizes of data types (in bytes)

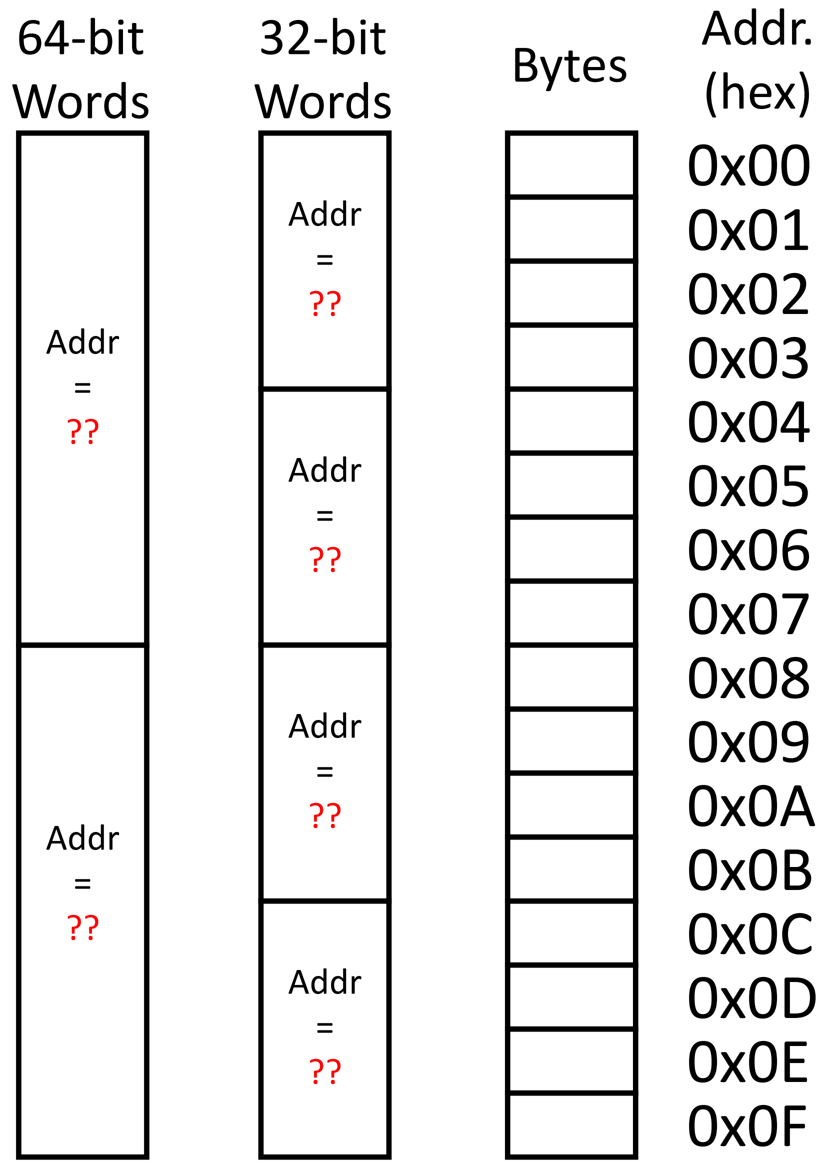
Java Data Type	C Data Type	32-bit (old)	x86-64
`boolean`	`bool`	1	1
`byte`	`char`	1	1
`char`		2	2
`short`	`short int`	2	2
`int`	`int`	4	4
`float`	`float`	4	4
	`long int`	4	8
`double`	`double`	8	8
`long`	`long long`	8	8
	`long double`	8	16
`(reference)`	**`pointer *`**	**4**	**8**

address size = word size

To use "`bool`" in C, you must `#include <stdbool.h>`

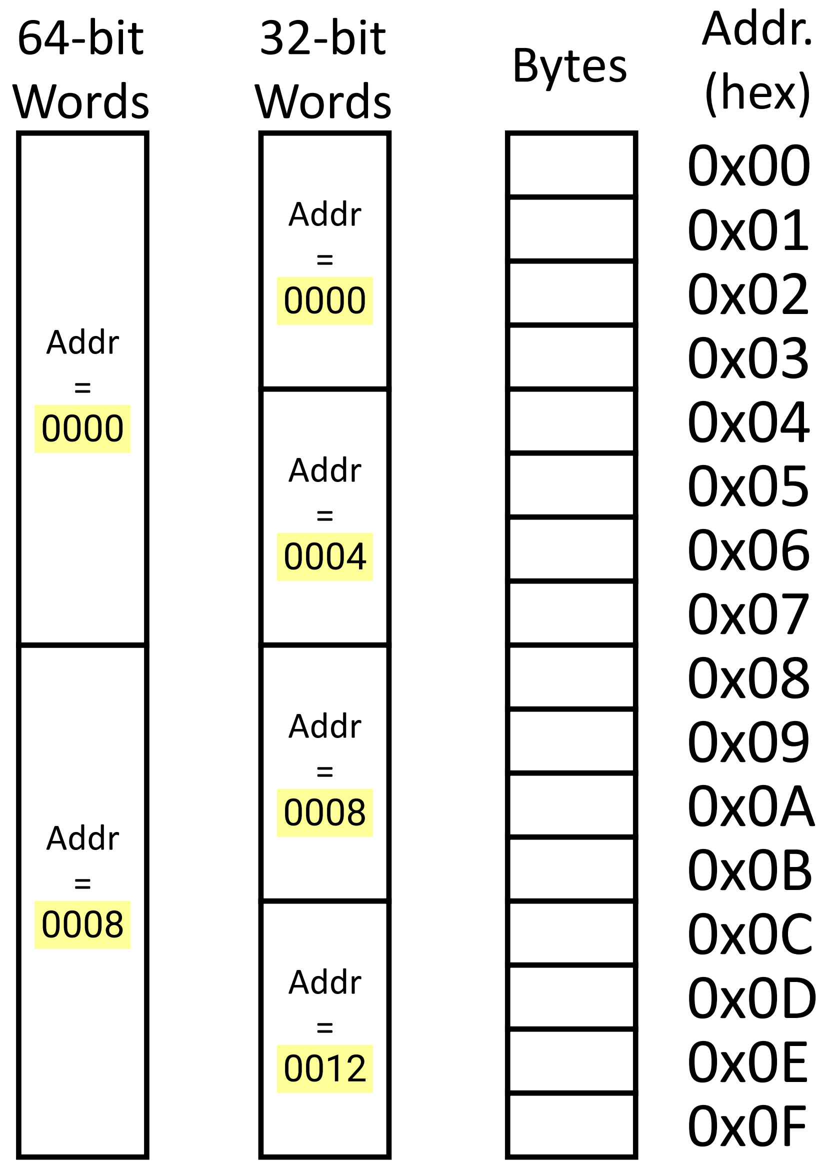
Address of Multibyte Data

- Addresses still specify locations of bytes in memory, but we can choose to *view* memory as a series of chunk of fixed-sized data instead
  - Addresses of successive chunks differ by data size
  - Which byte's address should we use for each word?



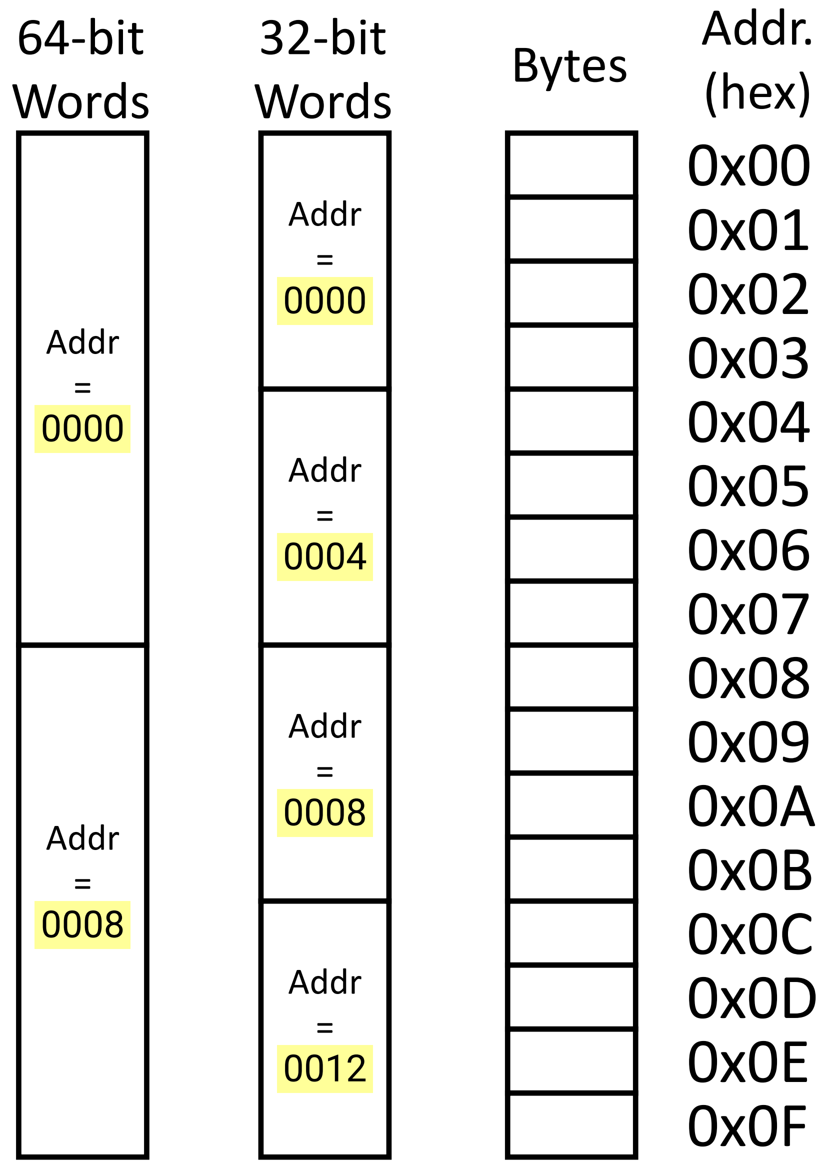
Address of Multibyte Data

- Addresses still specify locations of <u>bytes</u> in memory, but we can choose to *view* memory as a series of <u>chunk</u> of fixed-sized data instead
    - Addresses of successive chunks differ by data size
    - Which byte's address should we use for each word?
- The address of *any* chunk of memory is given by the address of the first byte
    - To specify a chunk of memory, need *both* its **address** and its **size**



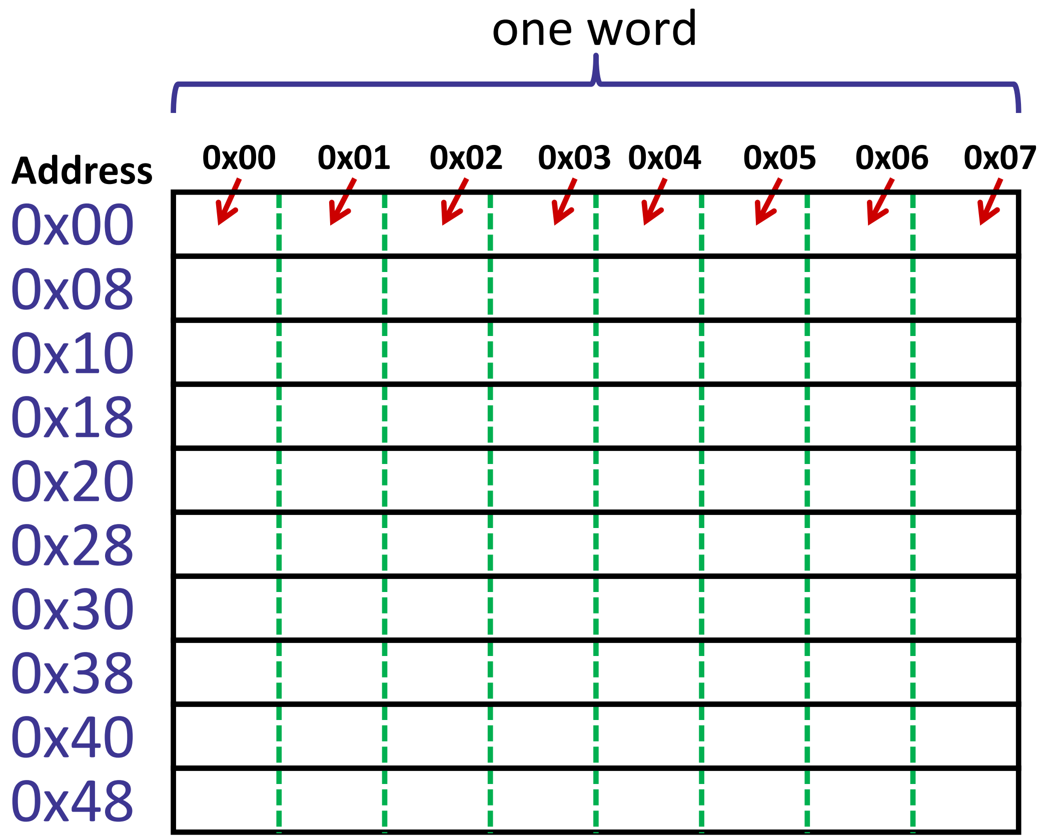
Alignment

- The address of a chunk of memory is considered aligned if its address is a multiple of its size
  - View memory as a series of consecutive chunks of this particular size and see if your chunk doesn't cross a boundary



A Picture of Memory (64-bit view)

- A “64-bit (8-byte) word-aligned” view of memory:
  - In this type of picture, each row is composed of 8 bytes
  - Each cell is a byte
  - An aligned, 64-bit chunk of data will fit on one row



A Picture of Memory (64-bit view)

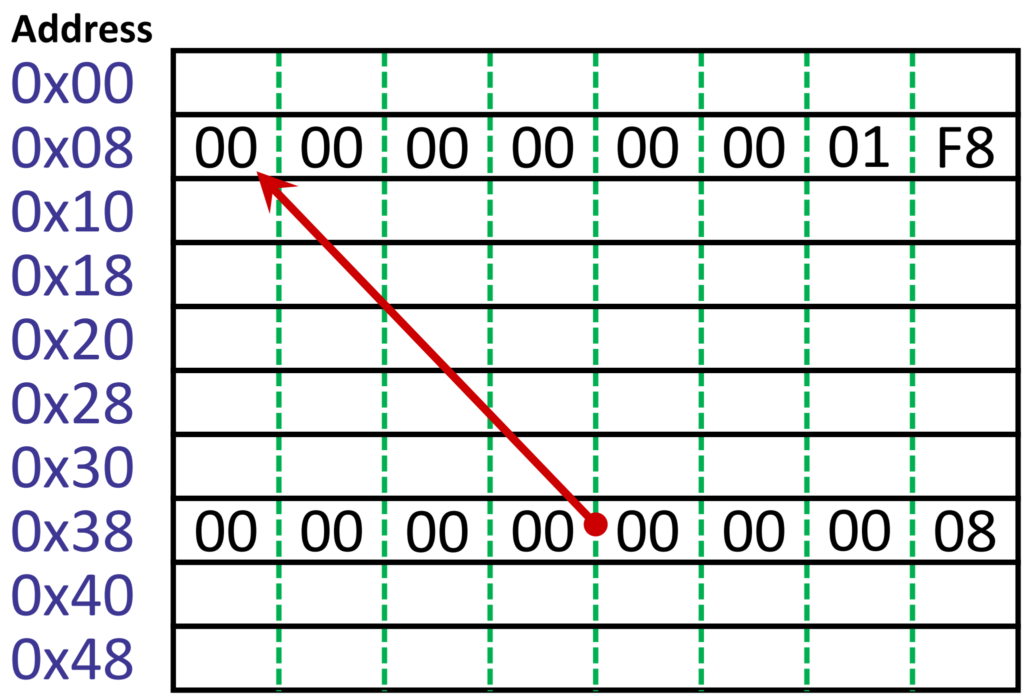
- A “64-bit (8-byte) word-aligned” <u>view</u> of memory:
  - In this type of picture, each row is composed of 8 bytes
  - Each cell is a byte
  - An aligned, 64-bit chunk of data will fit on one row

Addresses and Pointers

64-bit example
(pointers are 64-bits wide)

big-endian

- An *address* refers to a location in memory
- A *pointer* is a data object that holds an address
  - Address can point to *any* data
- Value 504 stored at address 0x08
  - $504_{10} = 1F8_{16}$
    $= 0x\ 00\ ...\ 00\ 01\ F8$
- Pointer stored at 0x38 points to address 0x08

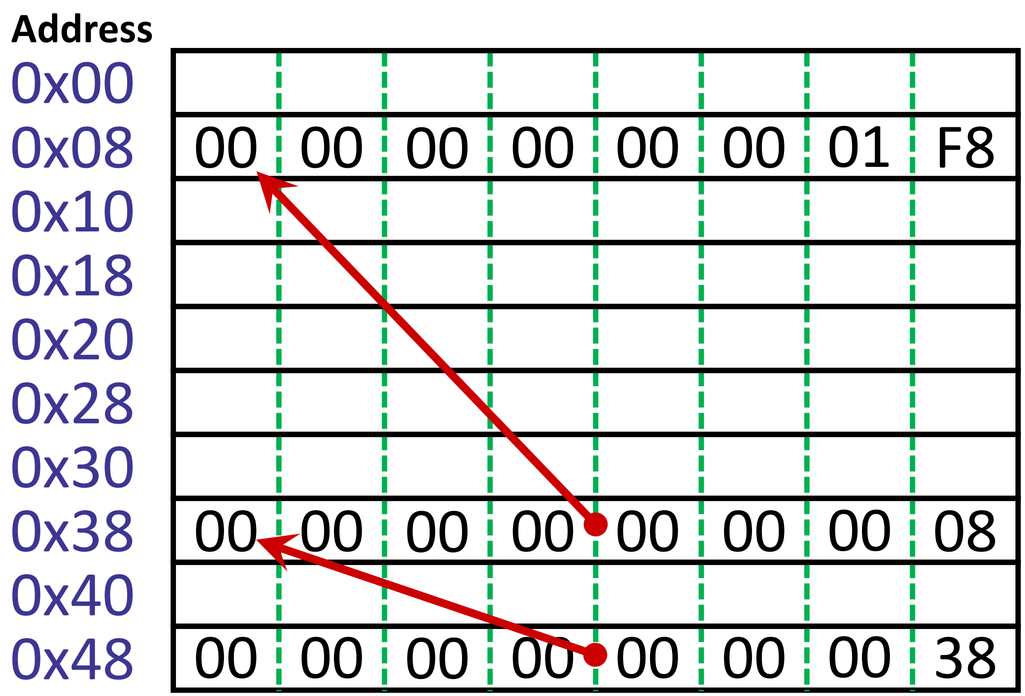


Addresses and Pointers

64-bit example
(pointers are 64-bits wide)

big-endian

- An *address* refers to a location in memory
- A *pointer* is a data object that holds an address
  - Address can point to *any* data
- Pointer stored at 0x48 points to address 0x38
  - Pointer to a pointer!
- Is the data stored at 0x08 a pointer?
  - Could be, depending on how you use it



Byte Ordering

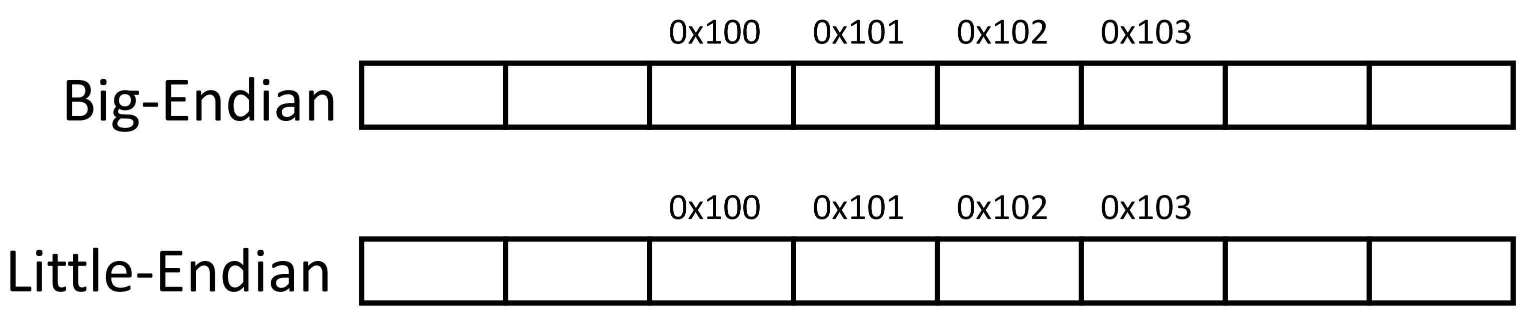
- How should bytes within a word be ordered *in memory?*
  - Want to keep consecutive bytes in consecutive addresses
  - **Example:** store the 4-byte (32-bit) `int`:
    `0x A1 B2 C3 D4`

- By convention, ordering of bytes called *endianness*
  - The two options are big-endian and little-endian
    - In which address does the least significant *byte* go?
    - Based on *Gulliver's Travels*: tribes cut eggs on different sides (big, little)

Byte Ordering

- Big-endian (SPARC, z/Architecture)
  - Least significant byte has highest address
- Little-endian (x86, x86-64)
  - Least significant byte has lowest address
- Bi-endian (ARM, PowerPC)
  - Endianness can be specified as big or little

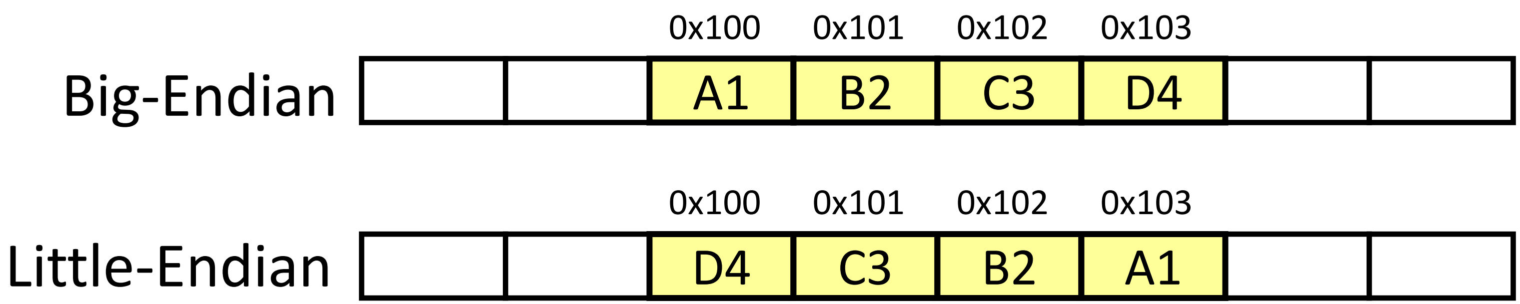
- **Example:** 4-byte data 0xA1B2C3D4 at address 0x100



Byte Ordering

- Big-endian (SPARC, z/Architecture)
  - Least significant byte has highest address
- Little-endian (x86, x86-64)
  - Least significant byte has lowest address
- Bi-endian (ARM, PowerPC)
  - Endianness can be specified as big or little

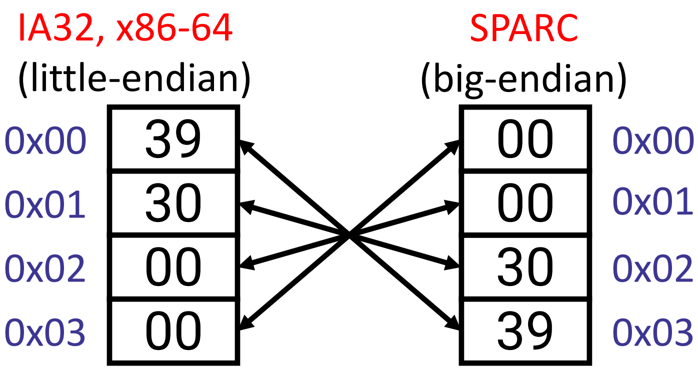
- **Example:** 4-byte data 0xA1B2C3D4 at address 0x100



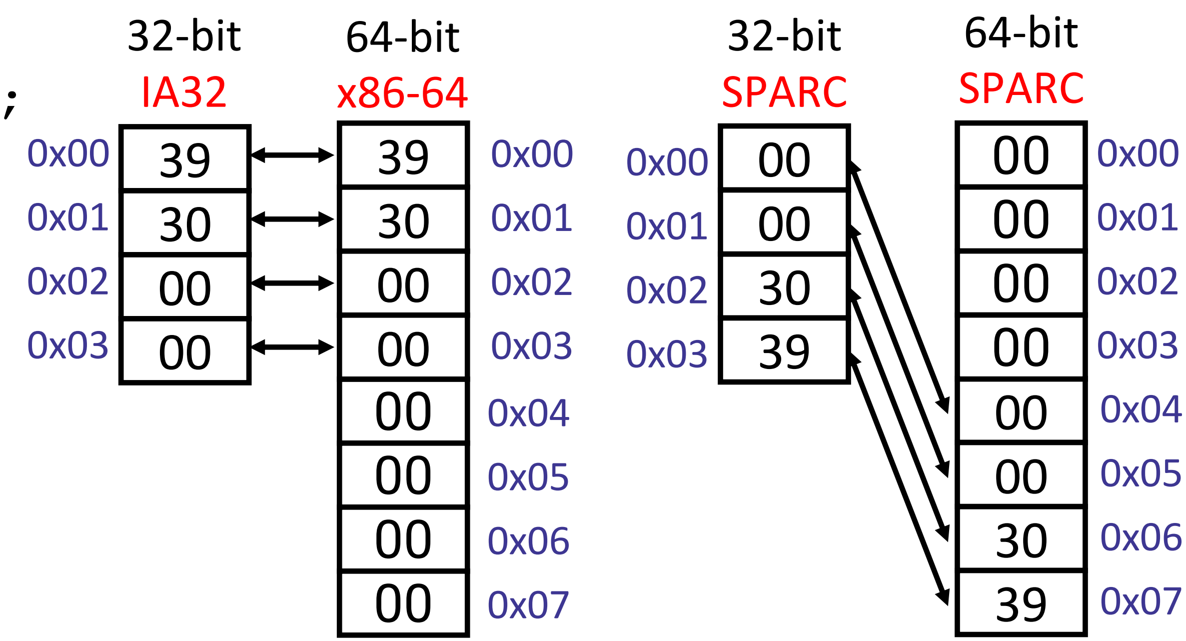
Byte Ordering Examples

Decimal:	12345			
Binary:	0011	0000	0011	1001
Hex:	3	0	3	9

```
int x = 12345;
// or x = 0x3039;
```




```
long int y = 12345;
// or y = 0x3039;
```



(A `long int` is the size of a word)

Endianness

- *Endianness only applies to memory storage*
- Often programmer can ignore endianness because it is handled for you
  - Bytes wired into correct place when reading or storing from memory (hardware)
  - Compiler and assembler generate correct behavior (software)
- Endianness still shows up:
  - Logical issues: accessing different amount of data than how you stored it (*e.g.* store `int`, access byte as a `char`)
  - Need to know exact values to debug memory errors
  - Manual translation to and from machine code

Summary

- Memory is a long, *byte-addressed* array
  - Word size bounds the size of the *address space* and memory
  - Different data types use different number of bytes
  - Address of chunk of memory given by address of lowest byte in chunk
  - Object of K bytes is *aligned* if it has an address that is a multiple of K
- Pointers are data objects that hold addresses
- Endianness determines memory storage order for multi-byte data