Integers (continued)
Floating point

Jinyang Li

Lesson plan

- 2’s complement
  - The negation trick
- A short history of processors:
  - from 8-bit to 64-bit machines
- Byte ordering
  - Big vs. small endian
- Intro to floating points

Two’s complement: 8-bit signed integer

01011000 $= 0*(-2^7) + 1*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 0*2^0 = 88$

11011000 $= 1*(-2^7) + 1*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 0*2^0 = -40$

00000000 $= 0$

11111111 $= -1$

10000000 $= -2^7 = -128$

01111111 $= 2^{7-1} = -127$

2’s complement: find a number’s negation

$(40)_{10}$ → $(-40)_{10}$

00101000 → ??

A useful trick to do negation:

Step-1: flip all bits

Step-2: add 1

00101000 $(40)_{10}$

$\downarrow$ Step-1: flip bits

11010111

$\downarrow$ Step-2: +00000001

11011000 $(-40)_{10}$

Why does the negation trick work

$$\vec{b} + (\sim \vec{b}) = 11\ldots11_2 = -1$$

b with bits flipped

$$-\vec{b} = (\sim \vec{b}) + 1$$

Negation trick lets us find bit pattern of a negative number more easily

The bit pattern of 8-bit signed integer -33?

Negation trick helps computers do subtraction

Instead of doing this:

$$
\begin{array}{rl}
 & 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1 \quad (5)_{10} \\
- & 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1 \quad (7)_{10} \\
\hline
 & 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0 \quad (-2)_{10}
\end{array}
$$

Do this instead:

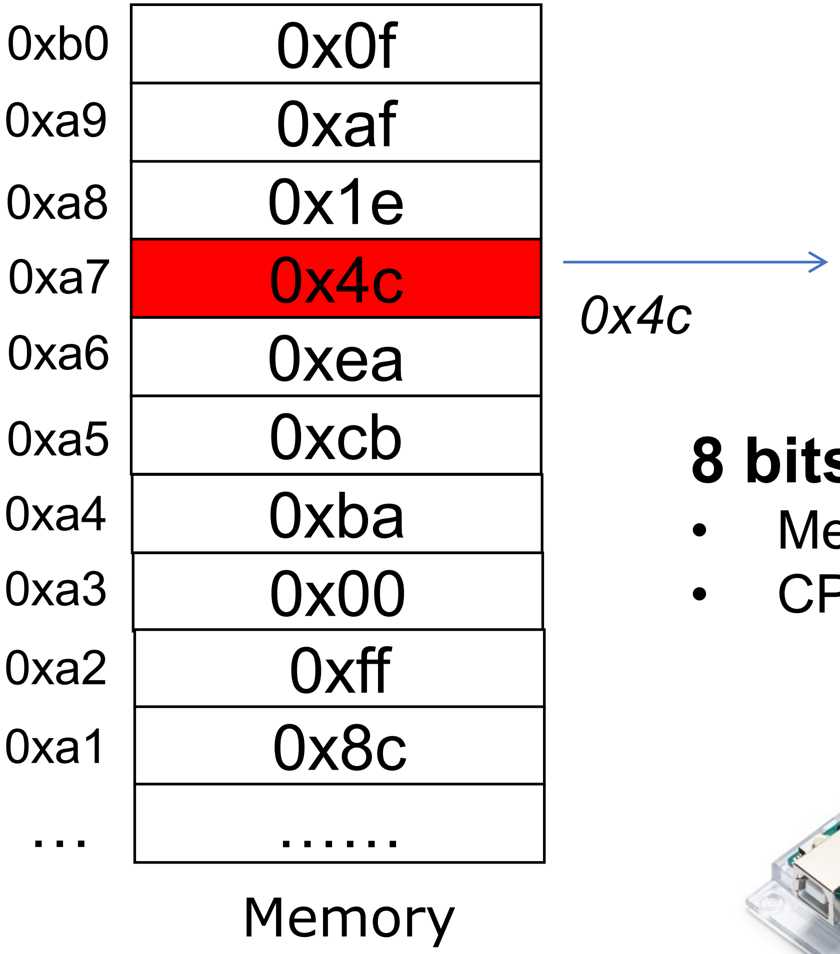
$$
0\ 0\ 0\ 0\ 0\ 1\ 1\ 1 \ (7)_{10} \Rightarrow \left.\begin{array}{l} 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0 \\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1 \end{array}\right\} (-7)_{10}
$$

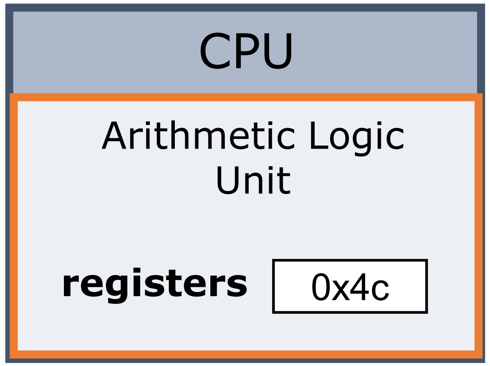
$$
\begin{array}{rl}
 & 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1 \quad (5)_{10} \\
 & 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0 \\
+ & 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1 \\
\hline
 & 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0 \quad (-2)_{10}
\end{array}
$$

Works for both signed and unsigned subtraction

The evolution of integer sizes in processors

8-bit processors: e.g. Intel 8080 (1974)



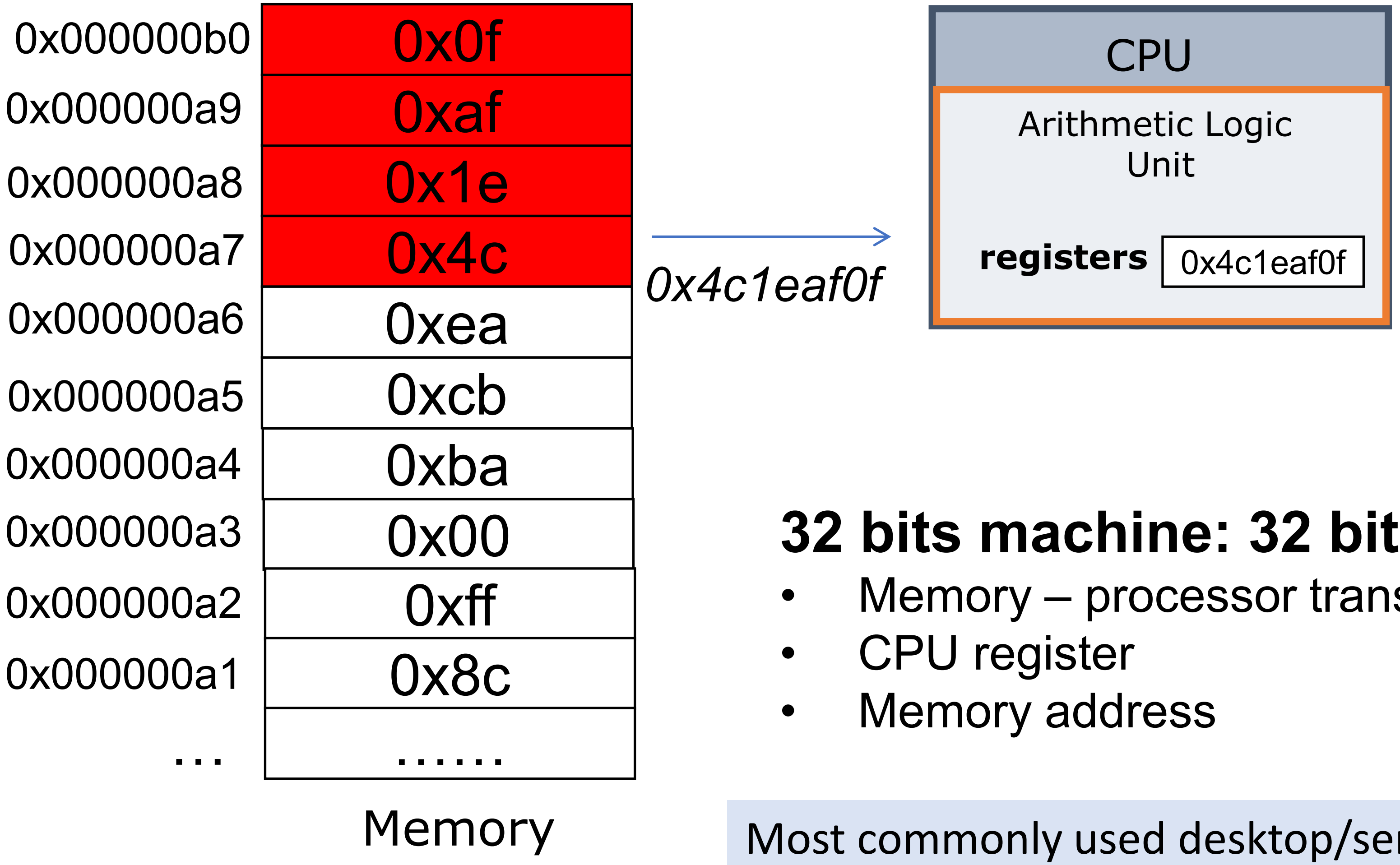


8 bits machine: 8 bits length of

- Memory – processor transfer
- CPU register

Nowadays: 8-bit processor (microcontroller)
Is used for embedded systems

32-bit processors: Intel 386 (1985)

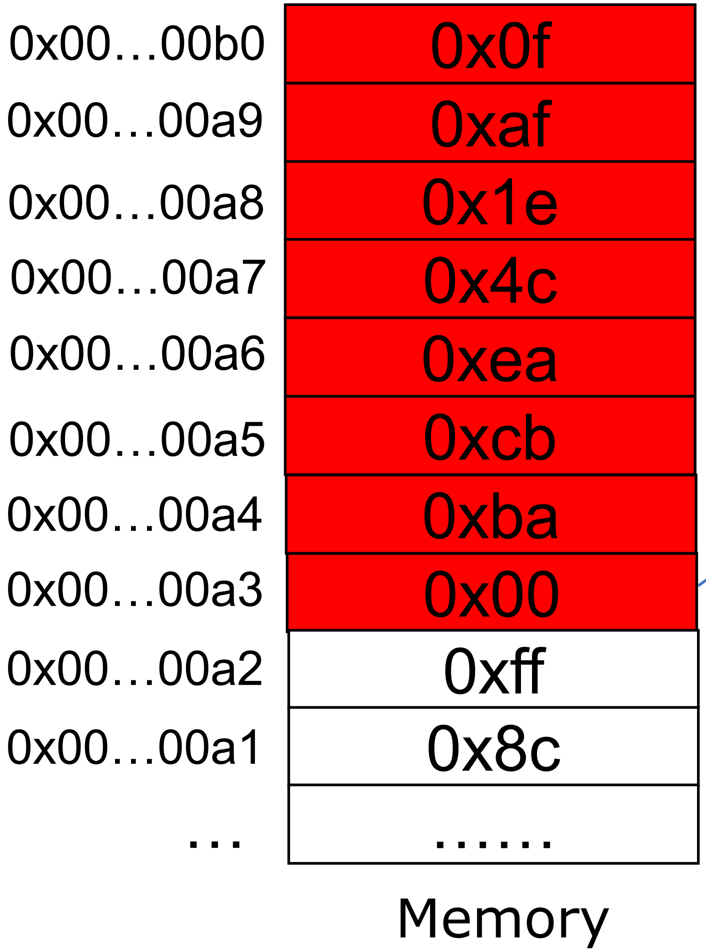


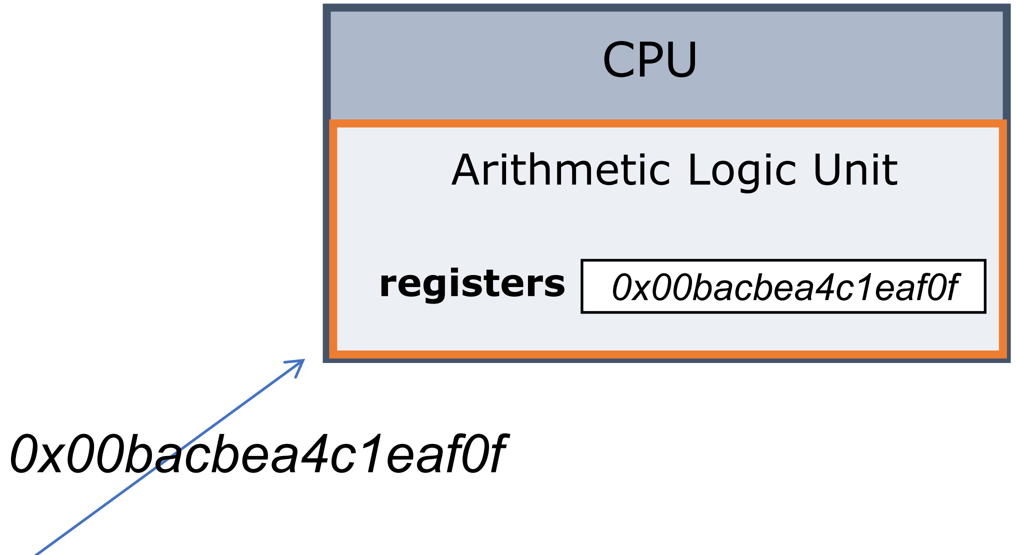
32 bits machine: 32 bits length of

- Memory – processor transfer
- CPU register
- Memory address

Most commonly used desktop/server processors in the latest 80s to early 00s

64-bit processors: Intel Pentium 4 (2000)





64 bits machine: 64 bits length of

- Memory – processor transfer
- CPU register
- Memory address

Nowadays: Intel/AMD 64-bit x86 processors used for servers/laptops
Mobile phones/tablets: 64-bit ARM processors (made by Apple/Qualcomm/Samsung etc)

C’s Integer data types on 64 bits machine

	Length	Min	Max
char	1 byte	-2^{7}	$2^{7} - 1$
unsigned char	1 byte	0	$2^{8} - 1$
short	2 bytes	-2^{15}	$2^{15} - 1$
unsigned short	2 bytes	0	$2^{16} - 1$
int	4 bytes	-2^{31}	$2^{31} - 1$
unsigned int	4 bytes	0	$2^{32} - 1$
long	8 bytes	-2^{63}	$2^{63} - 1$
unsigned long	8 bytes	0	$2^{64} - 1$

Your first C program

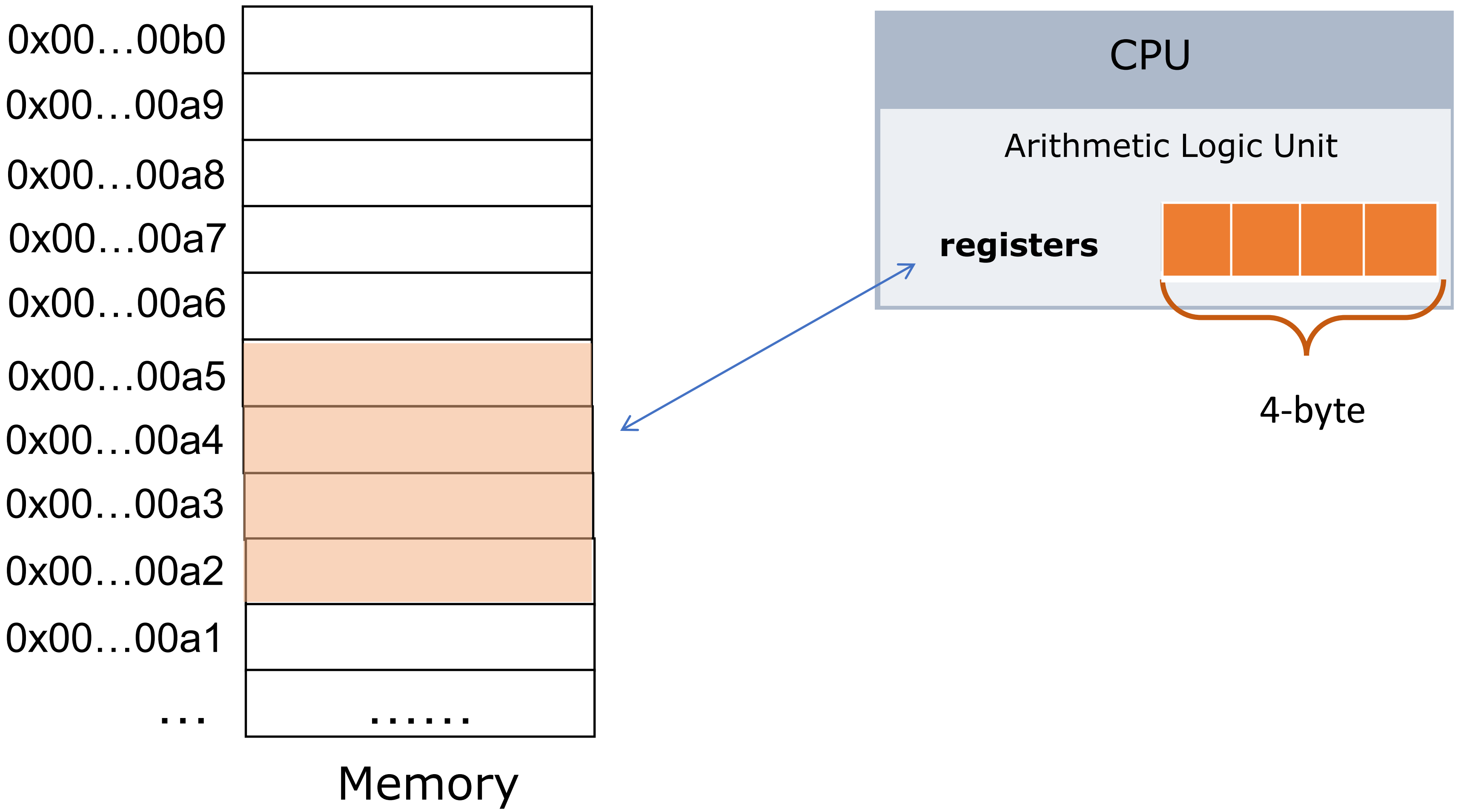
```
#include <stdio.h>

int
main()
{
      char x = -127;
      char y = 0x81;
      char z = x + y;
      printf("hello world sum is %d\n", z);
}
```

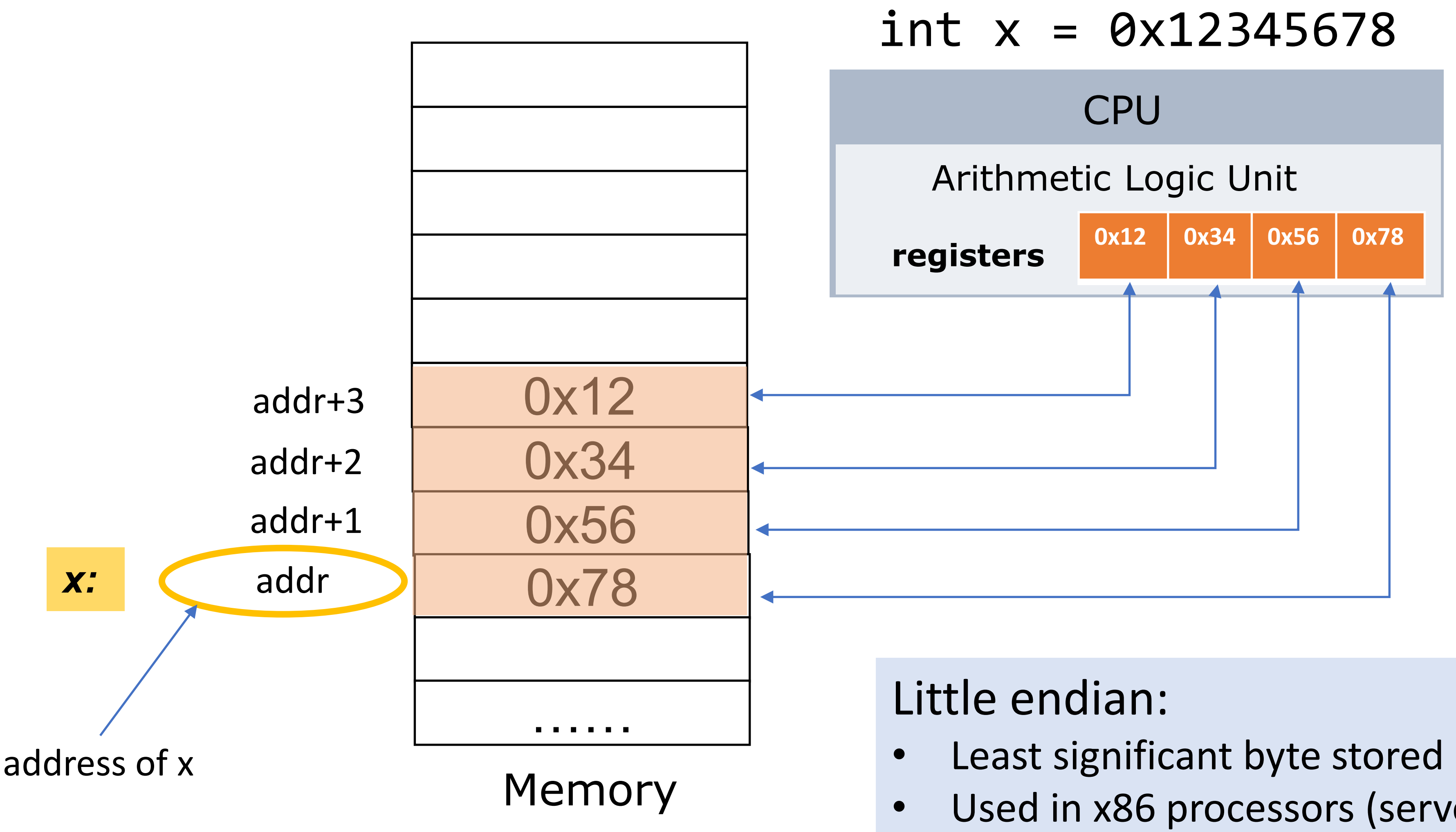
```
$ gcc helloworld.c
$ ./a.out
```

	1 0 0 0 0 0 0 1	-127
+	1 0 0 0 0 0 0 1	-127
1	0 0 0 0 0 0 1 0	2

Memory layout for multi-byte integers



Memory layout: Little Endian



Little endian:

- Least significant byte stored at smallest address
- Used in x86 processors (servers/laptops) and most ARM (phones/tablets) implementation

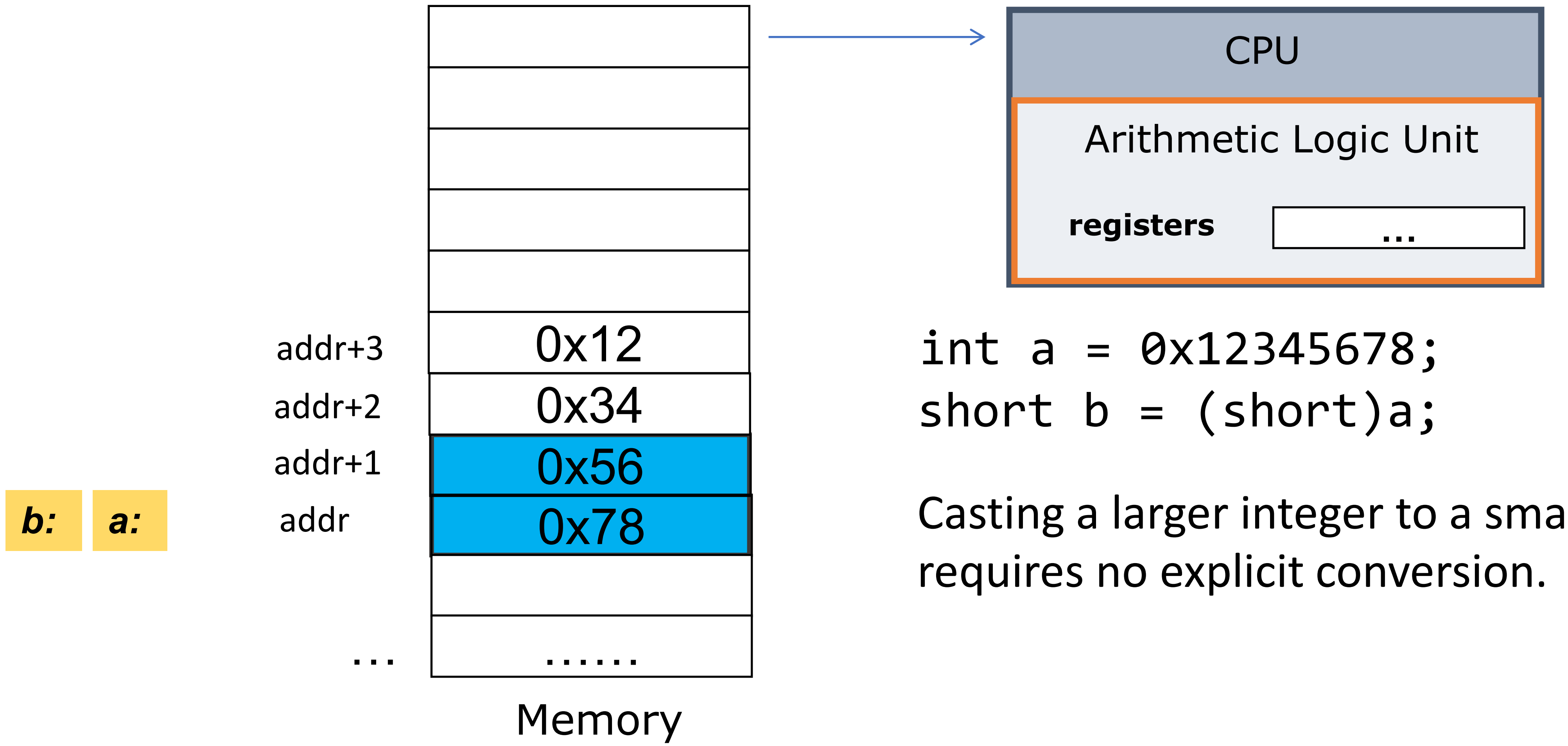
Advantage of Little Endian

$$
\begin{array}{rr}
 & \text{0x12345678} \\
+ & \text{0x12131415}
\end{array}
$$

Processor performs the calculation from the least significant bit

Processor can simultaneously perform memory transfer and calculation.

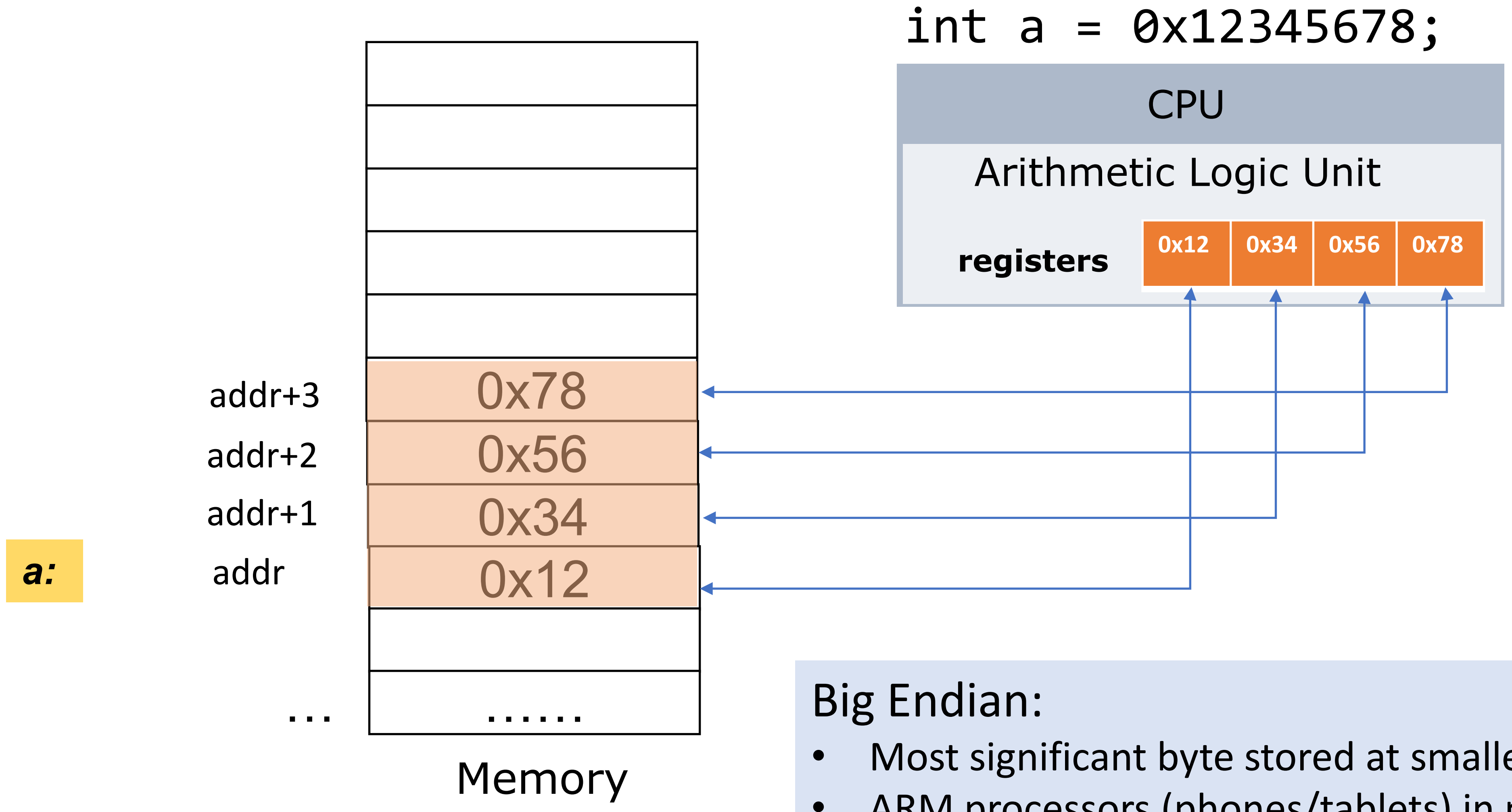
Another advantage of Little Endian




```
int a = 0x12345678;
short b = (short)a;
```

Casting a larger integer to a smaller one requires no explicit conversion.

Memory layout: Big Endian



Big Endian:

- Most significant byte stored at smallest address
- ARM processors (phones/tablets) in principle support both endian

Advantages of Big Endian

Quick to test whether the number is positive or negative

- Examine byte stored at the address offset zero.

Breakout time!

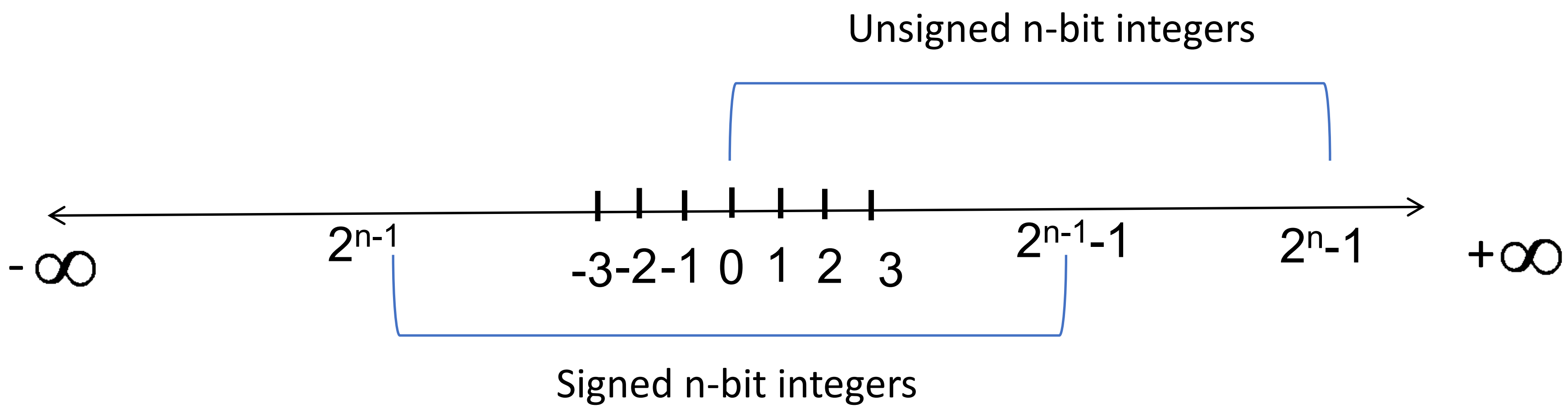
Breakout exercises

- 2's complement of -15 (8-bit, in hex)
- Computers use different hardware circuitry to perform addition vs subtraction.
- Computers use different hardware circuitry to perform unsigned addition vs. signed addition.
- long x = 0xdeadbeef01234567 Assume a Little Endian machine, and x is stored in memory starting at address a. What 1-byte value is stored at address a+3?

Lesson plan

- 2’s complement
  - The negation trick
- A short history of processors:
  - From 8-bit to 64-bit machines
- Byte ordering
  - Big vs. small endian
- Intro to floating points

Representing Real Numbers using bits



What about real numbers?

Decimal Representation

Real Numbers	Decimal Representation (Expansion)
$11 / 2$	$(5.5)_{10}$
$1 / 3$	$(0.3333333...)_{10}$
$\sqrt{2}$	$(1.4128...)_{10}$

Decimal Representation

Real Numbers	Decimal Representation (Expansion)
11 / 2	$(5.5)_{10}$
1 / 3	$(0.3333333...)_{10}$
$\sqrt{2}$	$(1.4128...)_{10}$

$(5.5)_{10} = 5 * 10^{0} + 5 * 10^{-1}$

$(0.333333...)_{10} = 3 * 10^{-1} + 3 * 10^{-2} + 3 * 10^{-3} + ...$

$(1.4128...)_{10} = 1 * 10^{0} + 4 * 10^{-1} + 1 * 10^{-2} + 2 * 10^{-3} + ...$

Decimal Representation

Real Numbers	Decimal Representation (Expansion)
$11 / 2$	$(5.5)_{10}$
$1 / 3$	$(0.3333333...)_{10}$
$\sqrt{2}$	$(1.4128...)_{10}$

$$(1.4128...)_{10} = 1 * 10^{0} + 4 * 10^{-1} + 1 * 10^{-2} + 2 * 10^{-3} + ...$$

Binary Representation

$(5.5)_{10} =$

$= (101.1)_2$

Binary Representation

$$(0.333333...)_{10} = 1 / 4 + 1 / 16 + 1 / 64 + ...$$

$$= (0.01010101...)_2$$

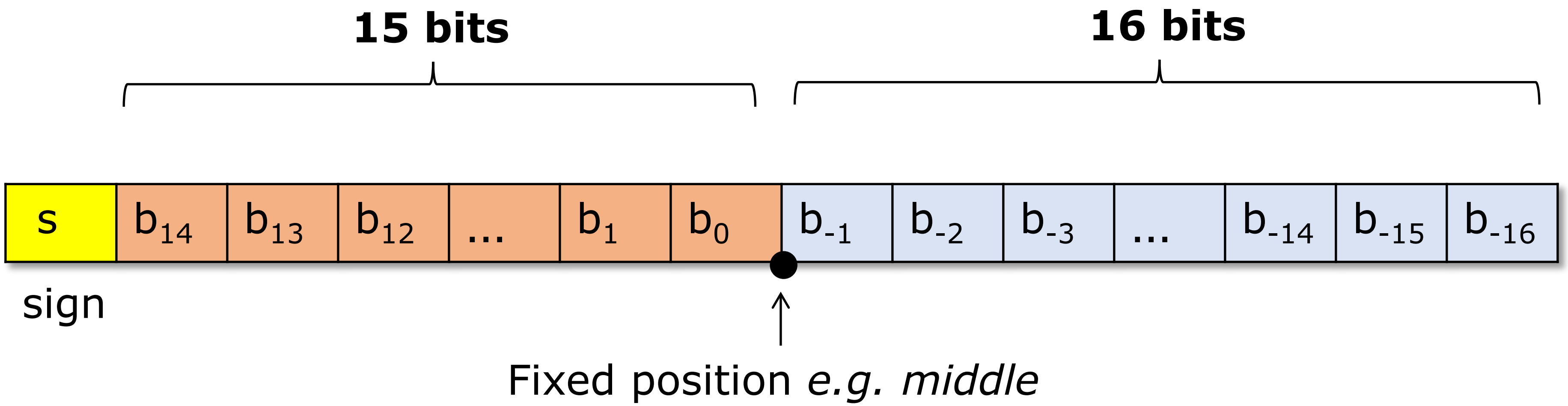
Binary Representation

$$\overset{2^p}{b_p}\,\overset{2^{p-1}}{b_{p-1}}\cdots\overset{2}{b_1}\,\overset{1}{b_0}\,.\,\underset{1/2}{b_{-1}}\,\underset{1/4}{b_{-2}}\cdots\underset{1/2^q}{b_{-q}} = \sum_{i=-q}^{p} 2^i \times b_i$$

Binary representation

What's the decimal value of $(10.01)_2$

Strawman representation: fixed point



Fixed point representation

	15 bits						16 bits						
s	b_{14}	b_{13}	b_{12}	…	b_{1}	b_{0}	b_{-1}	b_{-2}	b_{-3}	…	b_{-14}	b_{-15}	b_{-16}
sign													

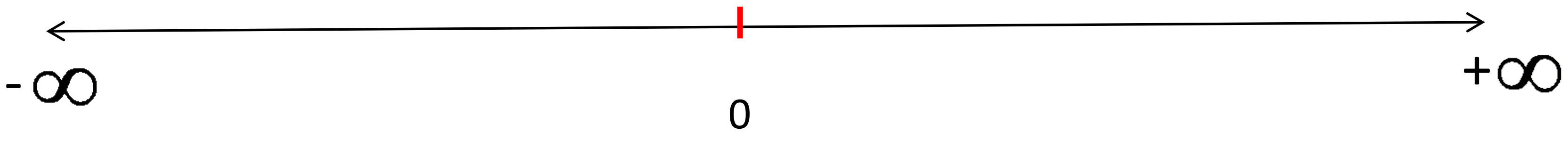
Example: $(\ 10.011\ )_2$

0	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0

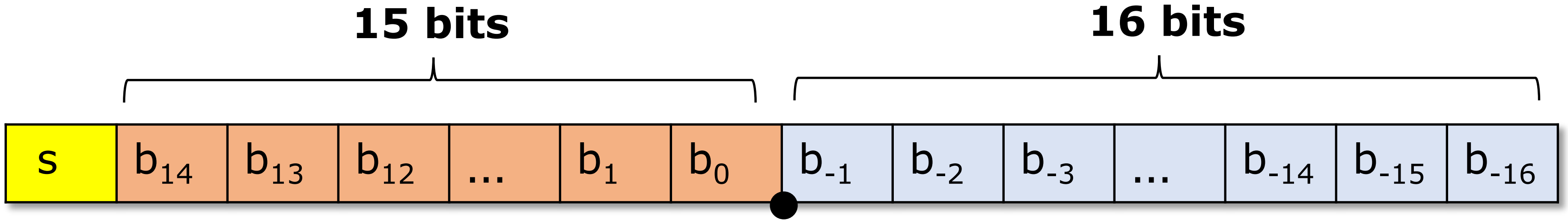
Problems of Fixed Point

	15 bits						16 bits						
s	b_{14}	b_{13}	b_{12}	…	b_1	b_0	b_{-1}	b_{-2}	b_{-3}	…	b_{-14}	b_{-15}	b_{-16}

Range?
Precision?



Problems of Fixed Point



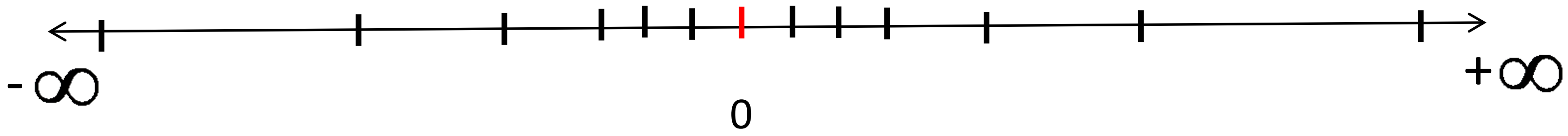
- Limited range and precision: e.g., 32 bits
  - Range: $[-2^{15}+2^{-16}, 2^{15}-2^{-16}]$
  - Highest precision: 2^{-16}

→Rarely used (No built-in hardware support)

Floating point: key idea

- Limitation of fixed point:
  - Even spacing results in hard tradeoff between high precision and high magnitude

- How about un-even spacing between numbers?



Floating Point: decimal

Based on exponential notation (aka normalized scientific notation)

$$r_{10} = \pm M * 10^{E}, \text{ where } 1 <= M < 10$$

M: significant (mantissa), E: exponent

Floating Point: decimal

Example:

$365.25 = 3.6525 * 10^{2}$

$0.0123 = 1.23 * 10^{-2}$

Decimal point **floats** to the position immediately after the first nonzero digit.

Floating Point: binary

Binary exponential representation

$r_{10} = \pm M * 2^E$, where $1 <= M < 2$

$M = (1.b_1b_2b_3\ldots b_n)_2$

M: significant, E: exponent

$(5.5)_{10} = (101.1)_2 = (1.011)_2 * 2^2$

Floating Point

Binary exponential representation

$$\left.\begin{array}{l} r_{10} = \underline{+}M * 2^E\text{, where } 1 <= M < 2 \\ M = (\ 1.b_1b_2b_3 \ldots b_n\ )_2 \end{array}\right\}$$

Normalized representation of r

M: significant, E: exponent

$(5.5)_{10} = (101.1)_2 = (1.011)_2 * 2^2$

Normalization: give a number r, obtain its normalized representation

Normalized representation of $(10.25)_{10}$?

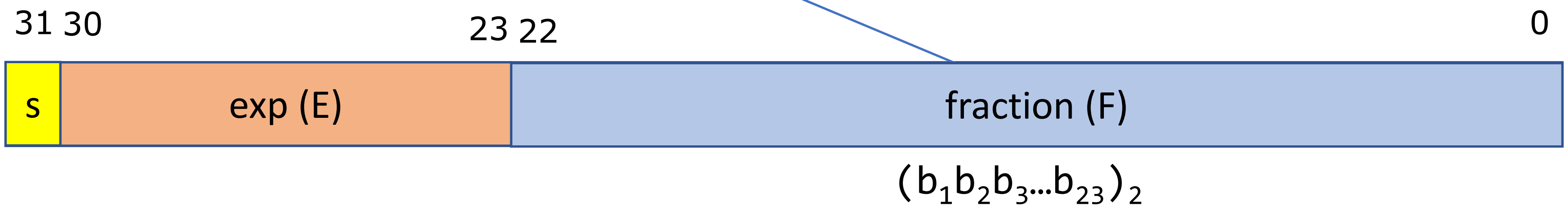
Normalized representation in computer

significant

exponent

$\pm M * 2^{E}$, where $1 <= M < 2$

$M = (\ 1.b_1b_2b_3\ldots b_{23}\ )_2$



Normalized representation

significant

exponent

$$\underline{+}M * 2^{E}, \text{ where } 1 <= M < 2$$

$$M = (1.b_1b_2b_3 \ldots b_{23})_2$$

Example: $(5.5)_{10} = (101.1)_2 = (1.011)_2 * 2^2$

31	30 23	22 0
0	0000 0010	0110 0000 0000 0000 0000 000

$(b_1b_2b_3 \ldots b_{23})_2$

Summary

- 2's complement
  - The negation trick, and its use for subtraction
- What are 32-bit or 64-bit processors?
- Byte ordering
  - Big vs. small endian
- Intro to floating points